Assignment 2.1 - Clustering

Please submit your solution of this notebook in the Whiteboard at the corresponding Assignment entry as .ipynb-file and as .pdf.

Please do **NOT** rename the file!

State both names of your group members here:

[S M Shameem Ahmed Khan and Rashid Harvey]

In [ ]:

Grading Info/Details - Assignment 2.1:

The assignment will be graded semi-automatically, which means that your code will be tested against a set of predefined test cases and qualitatively assessed by a human. This will speed up the grading process for us.

- For passing the test scripts:
  - Please make sure to **NOT** alter predefined class or function names, as this would lead to failing of the test scripts.
  - Please do **NOT** rename the files before uploading to the Whiteboard!
- **(RESULT)** tags indicate checkpoints that will be specifically assessed by a human.
- You will pass the assignment if you pass the majority of test cases and we can at least confirm effort regarding the **(RESULT)**-tagged checkpoints per task.

In [ ]:

Task 2.1.1 - kMeans

kMeans is an unsupervised learning algorithm that partitions n observations into k clusters. Each observation belongs to the cluster with the nearest mean (cluster center or centroid).

1. kMeans Implementation

- Implement the kMeans clustering algorithm using `numpy` only. Use the `KMeans` class structure below. **(RESULT)**
- Test the convergence of your implementation by creating a 2D synthetic dataset yourself. Report on the convergence. **(RESULT)**

In [12]:
```
import numpy as np
import matplotlib.pyplot as plt
```

In [13]:
```
# inpired by https://medium.com/@juanc.olamendy/back-to-basics-mastering-k-means-cl
class KMeans:
    def __init__(self, k=3, max_iters=100, tol=1e-4, random_state=None):
        """
        Initialize KMeans clusterer.

        Parameters:
        -----------
        k : int
            Number of clusters
        max_iters : int
            Maximum number of iterations
        tol : float
            Tolerance for convergence (change in centroids)
        random_state : int or None
            Random seed for reproducibility
        """
        self.k = k
        self.max_iters = max_iters
        self.tol = tol
        self.random_state = random_state
        self.labels_ = None
        self.report_convergence = False

    def initialize_centroids(self, X):
        """
        Initialize cluster centers using random selection from data points.
        """
        idx = np.random.choice(len(X), self.k, replace=False)
        self.centroids = X[idx]

    def initialize_centroids_plusplus(self, X): # for the following Subtask
        best_centroids = None
        best_loss = float("inf")

        tries = 10
```

```
        for t in range(tries):
            self.initialize_centroids(X)
            centroids = self.centroids

            # # compute loss = sum squared distance from each point to nearest cent
            # diff_all = X[:, None, :] - centroids[None, :, :]
            # dists2 = np.sum(diff_all, axis=2)
            # min_dists2 = np.min(dists2, axis=1)
            # loss = np.sum(min_dists2)

            loss = self._calculate_loss(X)

            if loss < best_loss:
                best_loss = loss
                best_centroids = centroids.copy()

        self.centroids = best_centroids

    def fit(self, X):
        """
        Fit the KMeans model to data X.
        """
        if self.random_state is not None:
            np.random.seed(self.random_state)

        for i in range(self.max_iters):
            distances = self._calculate_distances(X)
            labels = np.argmin(distances, axis=0)

            new_centroids = np.array([
                X[labels == i].mean(axis=0)
                for i in range(self.k)
            ])

            if all(
                np.linalg.norm(self.centroids[i] - new_centroids[i]) < self.tol
                for i in range(self.k)
            ):
                if self.report_convergence:
                    print("Convergence:", i)
                break

            self.centroids = new_centroids

        self.labels_ = labels

    def predict(self, X):
        """
        Predict cluster labels for new data.
        """
        distances = self._calculate_distances(X)
        return np.argmin(distances, axis=0)

    def fit_predict(self, X):
        """
```

```
        Perform KMeans clustering and return cluster labels.
        """
        self.fit(X)
        return self.labels_

    def _calculate_distances(self, X):
        n = len(X)
        distances = np.zeros((self.k, n))

        for i in range(n):
            diff = self.centroids - X[i]
            distances[:, i] = np.linalg.norm(diff, axis=1)

        return distances

    def _calculate_loss(self, X):
        distances = self._calculate_distances(X)

        return np.sum(
            np.min(distances, axis=0)
        )
```

In [14]:
```
size = 10
data_range = 100

dataset = []

for _ in range(size):
    dataset.append(
        [np.random.randint(0, data_range), np.random.randint(0, data_range)]
    )

dataset = np.array(dataset)

kmeans = KMeans()

kmeans.report_convergence = True

kmeans.initialize_centroids(dataset)
kmeans.fit(dataset)
```

```
Convergence: 1
```

2. kMeans++ initialization

- Implement the kMeans++ initialization method in the KMeans class. **(RESULT)**
- Compare the convergence speed of kMeans with random initialization and kMeans++ initialization on your synthetic dataset from Part 1. **(RESULT)**

In [15]:
```
print("Normal init")

kmeans = KMeans()

kmeans.report_convergence = True
```

```
kmeans.initialize_centroids(dataset)
kmeans.fit(dataset)

print("init ++")
kmeans_plus = KMeans()

kmeans.report_convergence = True

kmeans.initialize_centroids_plusplus(dataset)
kmeans.fit(dataset)
```

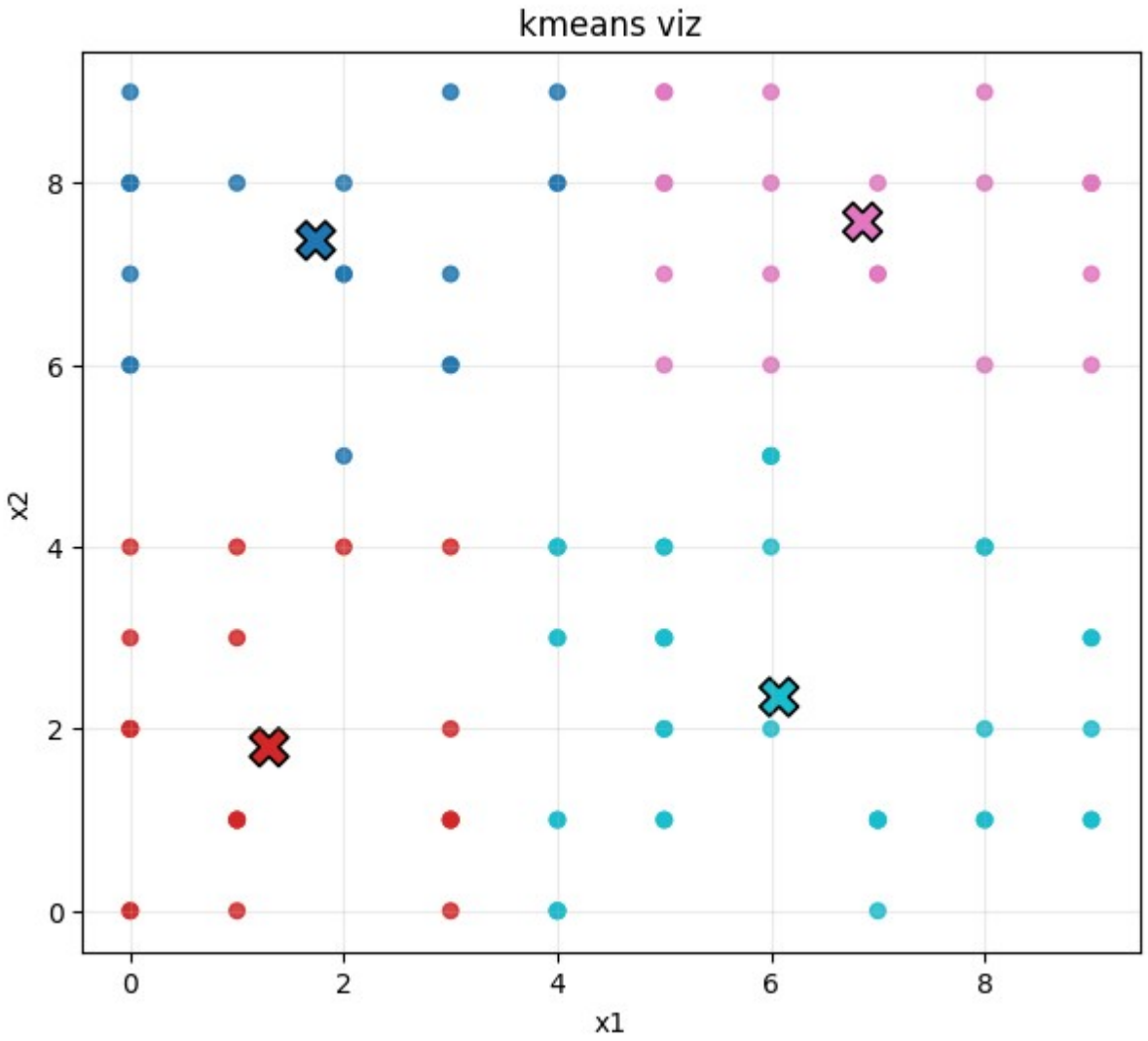
```
Normal init
Convergence: 3
init ++
Convergence: 1
```

3. Visualization of Cluster Quality

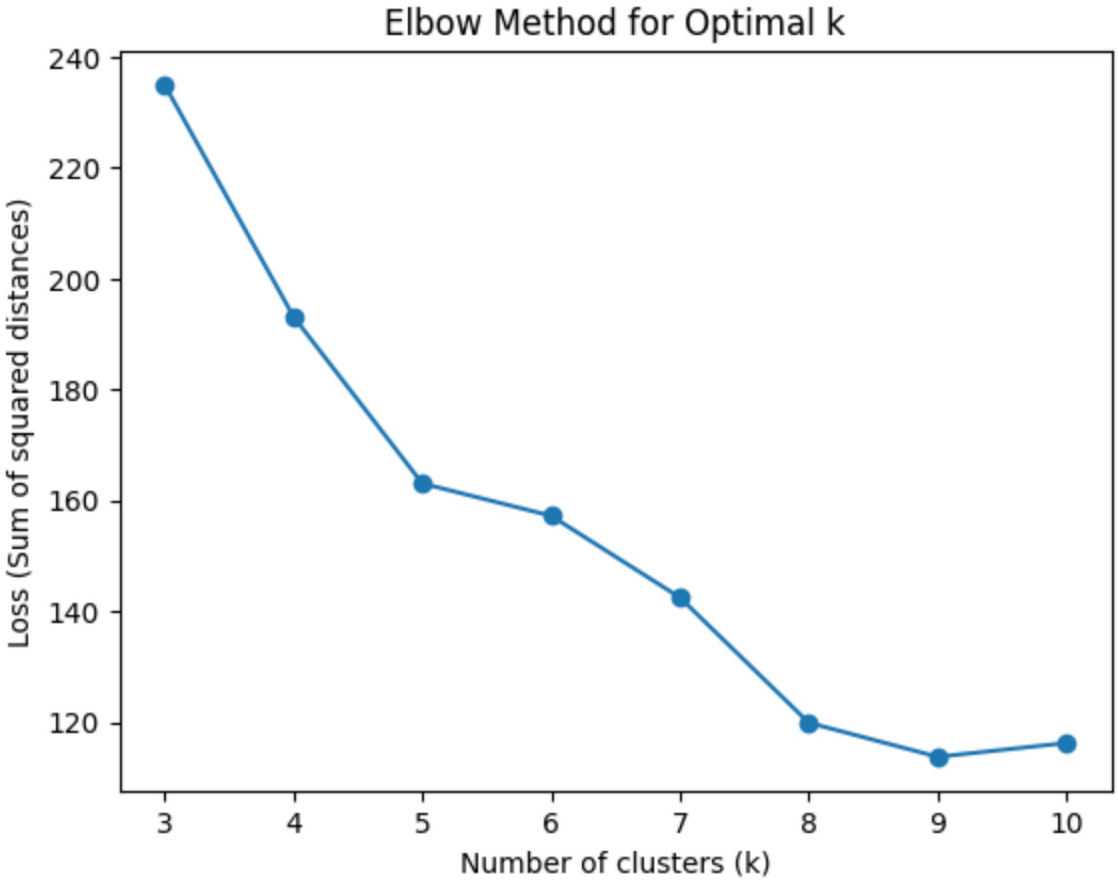
- Visualize the clustering results of your kMeans implementation on a synthetic 2D dataset with at least 4 clusters using matplotlib. **(RESULT)**
- Determine the optimal number of clusters using the elbow method. Report on your findings using a simple plot. **(RESULT)**
- Report on the silhouette score of your clustering results for the optimal k and k-1. **(RESULT)**

In [16]:
```
seed = 42

# Visualization helper for a given dataset and a fitted KMeans object
def visualize_kmeans(X, kmeans, title="kmeans viz", show_centroids=True, cmap='tab1

    labels = kmeans.labels_

    plt.figure(figsize=(7, 6))
    plt.scatter(X[:, 0], X[:, 1], c=labels, cmap=cmap, s=30, alpha=0.85)

    if show_centroids and hasattr(kmeans, 'centroids') and kmeans.centroids is not
        cent = np.asarray(kmeans.centroids)
        if cent.size and cent.shape[1] == 2:
            # Color centroids to match cluster colors by giving them the same label
            centroid_labels = np.arange(len(cent))
            plt.scatter(
                cent[:, 0], cent[:, 1],
                c=centroid_labels,
                cmap=cmap,
                s=200,
                marker='X',
                edgecolors='k',
                linewidths=1.2,
                label='centroids'
            )

    plt.title(title)
    plt.xlabel('x1')
```

```
    plt.ylabel('x2')
    plt.grid(alpha=0.25)
    plt.show()

size = 100
data_range = 10

dataset = []

for _ in range(size):
    dataset.append(
        [np.random.randint(0, data_range), np.random.randint(0, data_range)]
    )

dataset = np.array(dataset)

kmeans = KMeans(k=4, random_state=seed)

kmeans.initialize_centroids_plusplus(dataset)
kmeans.fit(dataset)

visualize_kmeans(dataset, kmeans)
```



In [ ]:
```python
# Elbow method

INITIAL_K = 3

k = INITIAL_K
losses = []
labels = []

while True:
    kmeans = KMeans(k=k, random_state=seed)

    kmeans.initialize_centroids_plusplus(dataset)
    kmeans.fit(dataset)

    loss = kmeans._calculate_loss(dataset)

    if losses and (losses[-1]-loss < kmeans.tol):
        losses.append(loss)
        break

    losses.append(loss)
    labels.append(kmeans.labels_)
    k += 1

# simple graph of loss over k
import matplotlib.pyplot as plt
plt.plot(range(INITIAL_K, INITIAL_K + len(losses)), losses, marker='o')
plt.xlabel('Number of clusters (k)')
plt.ylabel('Loss (Sum of squared distances)')
plt.title('Elbow Method for Optimal k')
plt.show()

# silhoute score for k and k-1
from sklearn.metrics import silhouette_score
if hasattr(kmeans, 'labels_'):
    print('Silhouette score k:', silhouette_score(dataset, kmeans.labels_))
    print('Silhouette score k-1:', silhouette_score(dataset, labels[-1]))
```




```
Silhouette score k: 0.38120175831062925
Silhouette score k-1: 0.42639370664408327
```

Task 2.1.2 - DBSCAN (BONUS)

DBSCAN is a density-based clustering algorithm that groups together points that are closely packed together, marking outliers points that lie alone in low-density regions.

- Implement the DBSCAN algorithm using `numpy` only. Use the `DBSCAN` class structure below. **(RESULT)**
- Test your DBSCAN implementation on a synthetic 2D dataset with noise. Visualize the clustering results using matplotlib. **(RESULT)**
- Compare the performance of your DBSCAN implementation with your kMeans implementation on the same synthetic 2D dataset using silhouette score as a metric. Please use the same random seed to make it comparable. **(RESULT)**

In [26]:
```python
from collections import deque   # Useful for efficient BFS implementation for findi

class DBSCAN:
    def __init__(self, eps=0.5, min_samples=5, metric='euclidean'):
        """
        Density-Based Spatial Clustering of Applications with Noise (DBSCAN)

        Parameters:
        -----------
```

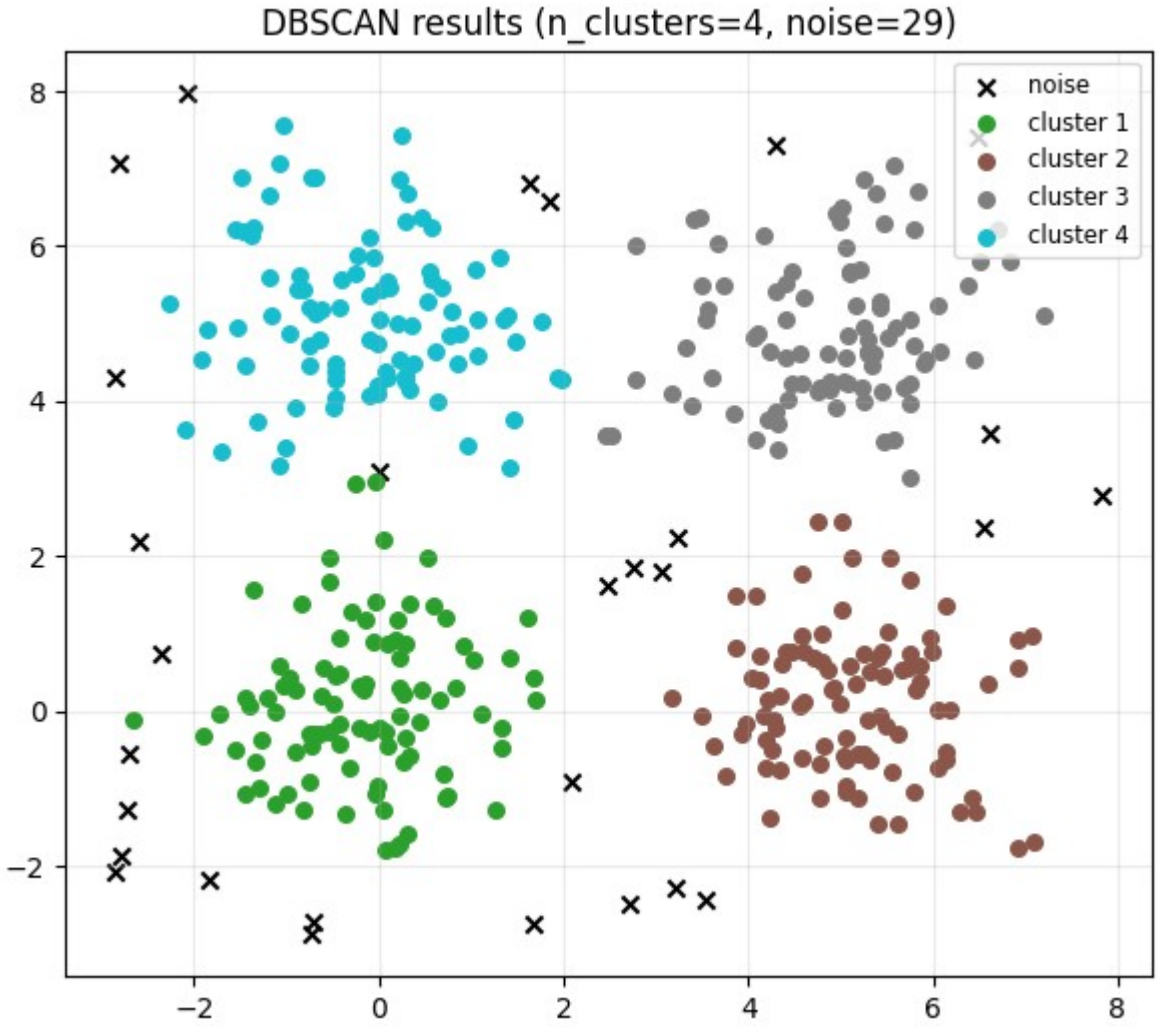
```
        eps : float
            Maximum distance between two samples for them to be considered neighbor
        min_samples : int
            Number of samples in a neighborhood for a point to be considered a core
            (including the point itself)
        metric : str
            Distance metric to use ('euclidean' or 'manhattan')
        """
        self.eps = eps
        self.min_samples = min_samples
        self.metric = metric
        self.labels_ = None
        self.core_sample_indices_ = None
        self.components_ = None
        self.n_clusters_ = None
        self.n_noise_ = None
        self.X_ = None

    def fit(self, X):
        """
        Perform DBSCAN clustering on X and set labels_.
        Labels: 0 = noise, 1..n = cluster ids
        """
        X = np.asarray(X)
        self.X_ = X
        n = len(X)
        labels = np.zeros(n, dtype=int)  # 0 means unassigned/noise
        current_label = 0

        # Iterate over points and expand clusters
        for point in range(n):
            neighbours = self.neighbours(X, point)

            if labels[point] != 0 or len(neighbours) < self.min_samples:
                continue

            # Start a new cluster
            current_label += 1
            labels[point] = current_label

            # BFS to expand cluster
            queue = deque(neighbours)
            while queue:
                idx = queue.popleft()
                if labels[idx] == 0:
                    labels[idx] = current_label
                    neigh2 = self.neighbours(X, idx)
                    # If idx is a core point, add its neighbors to the queue
                    if len(neigh2) >= self.min_samples:
                        for nb in neigh2:
                            if labels[nb] == 0:
                                queue.append(nb)

        self.labels_ = labels
        self.core_sample_indices_ = np.array([i for i in range(n) if len(self.neigh
```

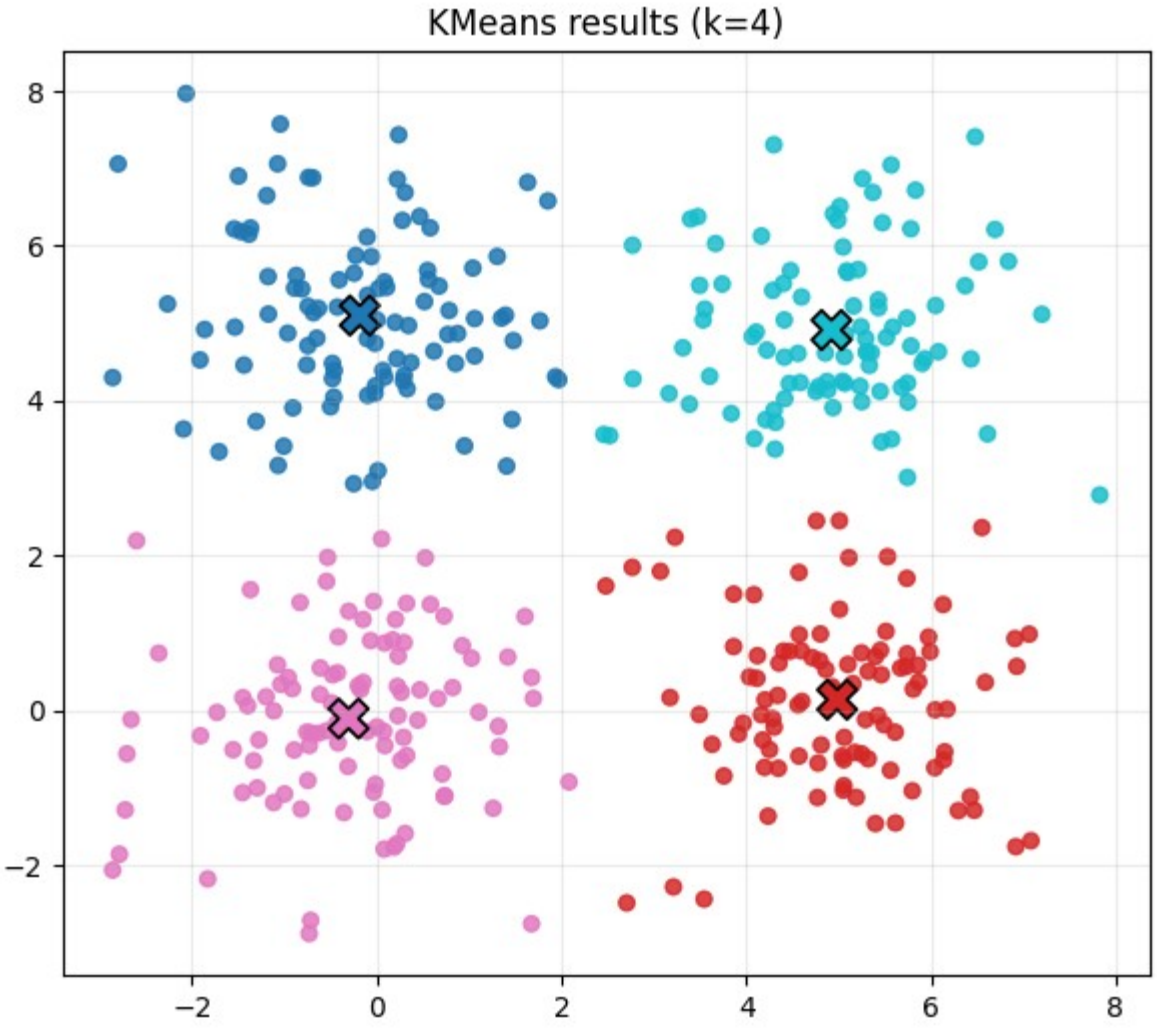
```
        self.n_clusters_ = int(labels.max())
        self.n_noise_ = int(np.sum(labels == 0))

    def predict(self, X_new):
        """
        Predict the closest cluster for new points using majority vote among labell
        New points can be labelled 0 (noise) if no labelled neighbours are found.
        """
        if self.X_ is None or self.labels_ is None:
            raise ValueError("fit must be called before predict")

        X_new = np.asarray(X_new)
        out_labels = np.zeros(len(X_new), dtype=int)
        for i, pt in enumerate(X_new):
            dists = self.distance(pt, self.X_)
            neighbours = np.where(dists <= self.eps)[0]
            neighbour_labels = [self.labels_[j] for j in neighbours if self.labels_
            if neighbour_labels:
                out_labels[i] = max(set(neighbour_labels), key=neighbour_labels.cou
            else:
                out_labels[i] = 0
        return out_labels

    def fit_predict(self, X):
        self.fit(X)
        return self.labels_

    def neighbours(self, X, p):
        """Return list of neighbour indices of point p (index) in X within eps."""
        point = X[p]
        dists = self.distance(point, X)
        return list(np.where(dists <= self.eps)[0])

    def distance(self, point1, point2):
        """Compute distance between point1 and each row in point2 (array-like) base

        if self.metric == 'euclidean':
            return np.linalg.norm(point1 - point2, axis=1)
        elif self.metric == 'manhattan':
            return np.sum(np.abs(point1 - point2), axis=1)
```

In [33]:
```
# More advanced synthetic 2D dataset with 4 Gaussian clusters and uniform noise
n_clusters = 4
points_per_cluster = 80
cluster_spread = 0.9
noise_points = 80

centers = np.array([[0,0],[5,0],[0,5],[5,5]])
X_clusters = []
for c in centers:
    X_clusters.append(c + cluster_spread * np.random.randn(points_per_cluster, 2))
X_clusters = np.vstack(X_clusters)

xmin, ymin = centers.min(axis=0) - 3
xmax, ymax = centers.max(axis=0) + 3
X_noise = np.random.uniform(low=(xmin, ymin), high=(xmax, ymax), size=(noise_points
```

```
# join clusters and noise
X = np.vstack([X_clusters, X_noise])

# DBSCAN
np.random.seed(seed)
db = DBSCAN(eps=0.8, min_samples=5, metric='euclidean')
db.fit(X)
db_labels = db.labels_

# KMeans
kmeans = KMeans(k=4, random_state=seed)
kmeans.initialize_centroids_plusplus(X)
kmeans.fit(X)
k_labels = kmeans.labels_

# Compute silhouette scores
sil_db = silhouette_score(X, db_labels)
sil_k = silhouette_score(X, k_labels)

# summary
print(f'DBSCAN: n_clusters = {db.n_clusters_}, n_noise = {db.n_noise_}')
print('Silhouette DBSCAN:', sil_db)
print()
print('KMeans: k =', kmeans.k)
print('Silhouette KMeans:', sil_k)

# DBSCAN
plt.figure(figsize=(7,6))
unique_labels = np.unique(db_labels)
colors = plt.cm.tab10(np.linspace(0,1,len(unique_labels)))
for lbl, col in zip(unique_labels, colors):
    mask = db_labels == lbl
    if lbl:
        plt.scatter(X[mask,0], X[mask,1], color=col.reshape(1,-1), s=30, label=f'cl
    else:
        plt.scatter(X[mask,0], X[mask,1], c='k', marker='x', label='noise')

plt.title(f'DBSCAN results (n_clusters={db.n_clusters_}, noise={db.n_noise_})')
plt.legend(loc='best', fontsize='small')
plt.grid(alpha=0.25)
plt.show()

# KMeans
plt.figure(figsize=(7,6))
plt.scatter(X[:,0], X[:,1], c=k_labels, cmap='tab10', s=30, alpha=0.85)
if hasattr(kmeans, 'centroids') and kmeans.centroids is not None:
    cent = np.asarray(kmeans.centroids)
    plt.scatter(cent[:,0], cent[:,1], c=np.arange(len(cent)), cmap='tab10', s=200,
plt.title(f'KMeans results (k={kmeans.k})')
plt.grid(alpha=0.25)
plt.show()
```

```
DBSCAN: n_clusters = 4, n_noise = 29
Silhouette DBSCAN: 0.5428973367450394

KMeans: k = 4
Silhouette KMeans: 0.5951830206633478
```





Congratz, you made it! :)