Assignment Clustering (ML 12)

Theoretical Questions

Q1. What is unsupervised learning in the context of Machine Learning?

Ans-> Unsupervised learning in machine learning is a type of algorithm that learns patterns and relationships from unlabeled data without explicit guidance or labeled examples. Unlike supervised learning, where the model is trained on data with known outcomes, unsupervised learning models discover hidden structures, patterns, and groupings within the data on their own.

Q2. How does K-Means clustering algorithm work?

Ans-> K-means clustering is an unsupervised machine learning algorithm that partitions data into K clusters based on feature similarity. The algorithm iteratively assigns data points to the nearest cluster centroid and then recalculates the centroids based on the assigned points, repeating this process until convergence.

- Here's a breakdown of the process:

1. Initialization: Choose K: Determine the desired number of clusters (K) beforehand. Random Centroids: Randomly initialize K centroids (points representing the center of each cluster) within the data space.
2. Assignment Step: Distance Calculation: Calculate the distance (often Euclidean distance) between each data point and all K centroids. Cluster Assignment: Assign each data point to the cluster whose centroid is closest.
3. Update Step: Recalculate Centroids: For each cluster, calculate the new centroid by taking the mean of all data points assigned to that cluster.
4. Iteration and Convergence: Repeat: Steps 2 and 3 are repeated iteratively until the cluster assignments no longer change significantly, or a maximum number of iterations is reached. Convergence: When the cluster assignments stabilize, the algorithm has converged, and the final clusters are formed. In essence, K-means aims to find the optimal placement of K centroids to minimize the within-cluster variance (or distance) and maximize the between-cluster variance.

Q3. Explain the concept of dendogram in hiearchical clustering?

Ans-> A dendrogram is a tree-like diagram that visualizes the hierarchical relationships between data points in hierarchical clustering. It shows how clusters are merged or split

at different levels of similarity, essentially representing the history of the clustering process. The height of the branches in the dendrogram indicates the distance or dissimilarity between the merged clusters.

- Here's a breakdown of the key aspects:
- Tree Structure: A dendrogram is a tree structure where the leaves (lowest nodes) represent individual data points, and internal nodes represent clusters formed by merging smaller clusters.
- Levels of Similarity: The height of the branches in the dendrogram indicates the distance (or dissimilarity) between the clusters being merged. Shorter branches indicate higher similarity between the merged clusters.
- Visualizing Cluster Merges: The dendrogram visually represents the step-by-step merging of clusters during the hierarchical clustering process.
- Determining Clusters: By cutting the dendrogram at a specific height, you can determine the number of clusters you want to create. Data points connected below the cut-off line belong to the same cluster.
- Agglomerative vs. Divisive: Dendrograms are used for both agglomerative (bottom-up) and divisive (top-down) hierarchical clustering methods.
- Examples: Dendrograms are used in various fields, including biology (taxonomy), data analysis, and more.

Q4. What is the main difference between K-means and hierarchical clustering?

Ans-> The primary difference between K-means and hierarchical clustering lies in how they form clusters. K-means requires a pre-defined number of clusters (k) and iteratively assigns data points to the nearest centroid, while hierarchical clustering builds a tree-like structure (dendrogram) showing how data points are merged or split, without needing a predetermined number of clusters.

- Here's a more detailed breakdown:
- K-means Clustering:
- Requires a pre-defined number of clusters (k): You need to specify the number of clusters you want before running the algorithm.
- Iterative process: It starts with random cluster centers (centroids) and iteratively reassigns data points to the nearest centroid, recalculating the centroids until convergence.
- Suitable for large datasets: K-means is generally faster and more scalable for large datasets compared to hierarchical clustering.
- May not capture complex cluster shapes: It can struggle with non-spherical or irregularly shaped clusters.
- Hierarchical Clustering:
- Does not require a pre-defined number of clusters: It creates a hierarchy of clusters, allowing you to choose the desired number of clusters from the dendrogram later.
- Builds a dendrogram: It represents the data points and their relationships in a tree-like structure, showing how clusters are merged or split at different levels.

- Can be computationally expensive: It can be slower than K-means, especially for large datasets.
- More flexible in capturing cluster shapes: It can identify clusters with more complex shapes. Can be sensitive to outliers: Outliers can significantly affect the formation of clusters.

Q5. What are the advantages of DBSCAN over K-means?

Ans-> DBSCAN offers several advantages over K-Means, particularly when dealing with datasets containing irregularly shaped clusters, outliers, or varying densities. DBSCAN can identify clusters of arbitrary shape, is more robust to noise and outliers, and doesn't require specifying the number of clusters beforehand.

- Here's a more detailed breakdown:

1. Handling Arbitrary Cluster Shapes:

- K-Means: Assumes clusters are spherical and equally sized. This can lead to poor clustering when dealing with non-spherical or differently sized clusters.
- DBSCAN: Can identify clusters of any shape by focusing on density. It groups together points that are closely packed together, regardless of the overall shape.

2. Noise and Outlier Handling:

- K-Means: Sensitive to outliers, as they can pull the centroids (means) of clusters away from the true center.
- DBSCAN: Explicitly identifies noise points (outliers) as those that don't belong to any dense cluster. This makes it more robust to noisy data.

3. Number of Clusters:

- K-Means: Requires the user to pre-define the number of clusters (k). This can be challenging, especially when the data structure is unknown.
- DBSCAN: Automatically determines the number of clusters based on the data density. The user only needs to specify parameters like epsilon (neighborhood radius) and minPts (minimum number of points within the radius).

4. Density-Based Approach:

- K-Means: Based on the concept of centroids and distances, which can be problematic when dealing with varying densities.
- DBSCAN: Uses a density-based approach, making it suitable for datasets with clusters of varying densities. It identifies clusters as dense regions separated by sparser regions.

5. When DBSCAN is a better choice:

- Complex data: When dealing with complex datasets where clusters have irregular shapes, varying densities, or contain noise, DBSCAN is a more suitable choice.
- Unknown number of clusters: If the number of clusters is not known beforehand, DBSCAN can automatically determine it.
- Outlier detection: DBSCAN's ability to identify outliers as noise points makes it useful for outlier detection tasks.

Q6. When would you use Silhouette Score in Clustering?

Ans-> The silhouette score is used to evaluate the quality of clustering results, specifically to assess how well data points are assigned to their respective clusters. It helps determine the optimal number of clusters and validate the chosen clustering method. It's particularly useful when there are no predefined labels for the data.

When to use it:

- K-means Clustering: The silhouette score is commonly used to evaluate the quality of K-means clustering results and to find the optimal number of clusters.
- Other Clustering Algorithms: It can be applied to assess the results of other clustering methods as well.
- No Predefined Labels: When you don't have ground truth labels for your data, the silhouette score offers an internal evaluation method.
- Interpretation:
- Ranges from -1 to 1: A score close to 1 indicates well-defined and separated clusters, while a score close to -1 suggests that data points might be assigned to the wrong cluster.
- 0 indicates overlapping clusters: A score of 0 or close to 0 indicates that the data points are on or very close to the decision boundary between two clusters.

Q7. What are the limitations of Hierarchical Clustering?

Ans-> Hierarchical clustering, despite its strengths, faces limitations like high computational cost for large datasets, sensitivity to noise and outliers, and the inability to revisit merging decisions. These drawbacks can lead to suboptimal cluster formations and make it unsuitable for certain large-scale applications.

- Here's a more detailed look at the limitations:

1. Computational Complexity and Scalability: Hierarchical clustering algorithms, especially agglomerative methods, have a time complexity typically of $O(n^3)$ or $O(n^2 \log n)$, where n is the number of data points. This means that as the dataset size increases, the computation time grows rapidly, making it impractical for very large datasets. Storing the distance matrix between all data points also requires significant memory, further limiting its scalability.

2. Sensitivity to Noise and Outliers: Hierarchical clustering can be heavily influenced by noise and outliers in the data. Outliers can cause the algorithm to merge dissimilar data points early on, leading to inaccurate cluster assignments. This sensitivity is particularly pronounced in methods like single linkage, which are prone to forming long, chained clusters due to outliers.
3. Inability to Revise Merging Decisions: Once clusters are merged or split in hierarchical clustering, the algorithm doesn't revisit these decisions. This inflexibility can lead to suboptimal clustering if early merging decisions are based on noisy data or if the optimal cluster structure requires adjustments later in the process.
4. Difficulty in Determining the Number of Clusters: While hierarchical clustering produces a dendrogram, a tree-like structure representing the hierarchical relationships between clusters, determining the optimal number of clusters to extract from the dendrogram can be subjective and challenging. Different cutting points on the dendrogram can lead to different numbers of clusters, and there isn't always a clear criterion for choosing the best one.
5. Dependence on Distance Metrics and Linkage Criteria: The choice of distance metric (e.g., Euclidean distance, Manhattan distance) and linkage criteria (e.g., single linkage, complete linkage, average linkage) significantly impacts the clustering results. Different choices can lead to vastly different cluster formations, and there isn't a universally optimal choice.
6. Not Ideal for High-Dimensional Data: Hierarchical clustering can struggle with high-dimensional data, where the curse of dimensionality can make distance calculations less meaningful and increase the likelihood of encountering irrelevant features.

Q8. Why is feature scaling important in clustering algorithms like K-means?

Ans-> Feature scaling is crucial for k-means clustering because it ensures that all features contribute equally to the distance calculations, preventing features with larger scales from dominating the process. Without scaling, the algorithm might cluster data based on the features with the largest values, leading to inaccurate results.

- Here's why feature scaling is important for k-means:
- Distance-based algorithm: K-means relies on the Euclidean distance to group data points. Features with larger scales can disproportionately influence the distance calculations, making the algorithm biased towards those features.
- Equal contribution of features: Scaling ensures that all features have a similar range of values, so each feature contributes equally to the distance calculation, preventing features with larger values from having a greater impact.
- Improved clustering accuracy: By scaling the features, k-means can identify meaningful clusters based on all the features, leading to more accurate and reliable results.
- Faster convergence: Scaling can also improve the convergence speed of the k-means algorithm, as it helps the algorithm to find the optimal cluster centers more efficiently.

Q9. How does DBSCAN indentify noise points?

Ans-> DBSCAN identifies noise points (or outliers) by first defining core, border, and noise points based on density. A core point has at least min_samples points within its radius (epsilon). A border point is within the epsilon radius of a core point but doesn't have enough neighbors itself. Noise points are neither core nor border points, meaning they don't meet the density criteria to belong to any cluster.

- Here's a more detailed breakdown:

1. Define Parameters:

- DBSCAN uses two key parameters: Epsilon (ε): The radius around a data point to search for neighbors.
- MinPts: The minimum number of data points required within the epsilon radius to define a dense region.

2. Identify Core Points: For each data point, DBSCAN checks if it has at least min_samples points within its epsilon radius (including itself). If it does, it's a core point.
3. Identify Border Points: If a point is not a core point but is within the epsilon radius of a core point, it's classified as a border point.
4. Identify Noise Points: Any point that is neither a core point nor a border point is considered a noise point (or outlier).
5. Cluster Formation: Clusters are formed by connecting core points that are within each other's epsilon neighborhood. Border points are then assigned to the clusters of their neighboring core points.
6. Noise Points are Excluded: Noise points are not included in any cluster because they don't meet the density criteria to be part of a dense region. DBSCAN Clustering in ML - Density based clustering In essence, DBSCAN uses density to distinguish between meaningful clusters and noise. Points that are isolated and sparsely populated are labeled as noise.

Q10. Define inertia in the context of K-means?

Ans-> K-Means: Inertia It is calculated by measuring the distance between each data point and its centroid, squaring this distance, and summing these squares across one cluster. A good model is one with low inertia AND a low number of clusters (K). However, this is a tradeoff because as K increases, inertia decreases.

Q11. What is the elbow method in K-means clustering?

Ans-> The elbow method is a heuristic used in K-means clustering to determine the optimal number of clusters (k) for a dataset. It involves plotting the within-cluster sum of squares (WCSS) against the number of clusters and identifying the "elbow point" on the

curve. This point, where the rate of decrease in WCSS slows down, suggests the ideal number of clusters, as adding more clusters beyond this point doesn't significantly improve the clustering.

Q12. Describe the concept of "density" in DBSCAN?

Ans-> In DBSCAN (Density-Based Spatial Clustering of Applications with Noise), density refers to the concentration of data points within a specific region of the feature space. It's determined by the number of data points found within a defined neighborhood around a given point. A dense region has many points close together, while a sparse region has few points in close proximity. DBSCAN uses this density concept to identify clusters as tightly packed groups of data points separated by areas of lower density.

- Here's a more detailed breakdown:
- Core Points: A point is considered a core point if it has at least a specified number of other data points (minPts) within a certain radius (epsilon or eps) of itself.
- Epsilon (eps): This parameter defines the radius of the neighborhood around a point.
- MinPts: This parameter sets the minimum number of data points required within the eps radius for a point to be considered a core point.
- Clusters: DBSCAN forms clusters by linking core points and their density-reachable points. A cluster grows as it includes neighboring core points and their reachable points.
- Noise Points: Points that are not core points and do not fall within the neighborhood of any core point are classified as noise.

Q13. Can hierarchical clustering be used on categorical data?

Ans-> Yes, hierarchical clustering can be used with categorical data, but it requires a different approach than when working with numerical data. You'll need to define a distance or dissimilarity measure suitable for categorical variables.

- Here's a breakdown:
- Traditional Hierarchical Clustering: Typically uses numerical data and calculates distances like Euclidean distance. This doesn't directly apply to categories.
- Categorical Data: Requires a different distance metric or dissimilarity measure. Some researchers propose using measures like the Hamming distance (for binary or indicator variables) or other specialized metrics for categorical data.
- Distance/Dissimilarity Measures: Hamming Distance: Counts the number of differing attributes between two data points.
- Gower's Distance: A more general approach that can handle mixed data types (numerical and categorical).
- Other methods: There are other specialized measures designed for categorical data, sometimes based on information theory or probabilistic models.

- Encoding: Categorical data often needs to be encoded before clustering. One-hot encoding is a common method, but other techniques might be more appropriate depending on the specific data and research question.
- Algorithm: Once a suitable distance or dissimilarity measure is chosen, you can apply standard hierarchical clustering algorithms like agglomerative hierarchical * clustering.
- Considerations:
- Outliers: Hierarchical clustering can be sensitive to outliers in categorical data.
- Interpretability: It's important to consider the interpretability of the resulting clusters, as combinations of categorical values can be meaningful or not.
- Computational Cost: Hierarchical clustering can be computationally expensive, especially for large datasets.
- Stability: The stability of the clustering solution (how consistent it is across different runs) can be a concern with categorical data.

Q14. What does a negative Silhouette Score indicate?

Ans-> A negative Silhouette Score usually signals that something isn't quite right with your clustering solution.

Here's what it suggests:

- **Poor cluster assignment**: A negative value means the sample is, on average, **closer to points in a different cluster** than to those in its assigned cluster.
- **Overlapping clusters**: The boundaries between clusters may not be well defined — they might be muddled, overlapping, or too close together.
- **Potential noise or outliers**: Especially relevant in methods like DBSCAN, points with negative scores could be misclassified noise or located in sparse regions.
- **Wrong number of clusters**: It might be a hint to reconsider your choice of *k* in K-Means or revisit how clusters are being formed.

- To improve your clustering:

- Try **visualizing with t-SNE or PCA** to inspect cluster separation.
- Re-evaluate your **distance metric** — especially for non-Euclidean spaces.
- Explore **alternative algorithms** like Gaussian Mixture Models if K-Means isn't performing well.

Q15. Explain the term "linkage criteria" in hierarchical clustering?

Ans-> In hierarchical clustering, linkage criteria define how the distance between clusters is calculated when merging them. These criteria determine which clusters are combined at each step of the algorithm based on the distances between their data points. Different linkage methods exist, each with its own way of measuring cluster proximity.

- Here's a breakdown of common linkage criteria:

1. Single Linkage: The distance between two clusters is defined as the minimum distance between any two points in those clusters. This method can be sensitive to noise and outliers, potentially leading to long, chain-like clusters.
2. Complete Linkage: The distance between two clusters is defined as the maximum distance between any two points in those clusters. This method tends to create tighter, more compact clusters compared to single linkage.
3. Average Linkage: The distance between two clusters is defined as the average of all pairwise distances between points in the two clusters. It provides a balance between single and complete linkage.
4. Centroid Linkage: The distance between two clusters is defined as the distance between their centroids (means of the data points within each cluster).
5. Ward's Method: This method focuses on minimizing the increase in variance within clusters when merging. It aims to create clusters with low internal variance, indicating that data points within each cluster are highly similar.

Q16. Why might K-means clustering perform poorly on data with varying cluster sizes or densities?

Ans->

Limitations of K-Means on Varying Cluster Sizes or Densities

1. **Assumption of Equal Variance and Spherical Clusters**

- K-means assumes that all clusters are spherical and roughly equal in size.
- It minimizes intra-cluster variance using Euclidean distance, which works best when clusters are compact and similar in scale.
- If one cluster is large and sparse while another is small and dense, K-means may split the large one or merge the small one incorrectly.

2. **Centroid Bias**

- K-means places cluster centroids based on the mean of assigned points.
- In uneven clusters, the centroid may be pulled toward denser regions, misrepresenting the actual center of sparse clusters.

3. **Sensitivity to Initial Centroid Placement**

- Poor initialization can lead to suboptimal clustering, especially when clusters differ in size or density.
- Even with techniques like K-means++, the algorithm may still converge to a local minimum that doesn’t reflect the true structure.

4. **Hard Assignments**

- Each point is assigned to the nearest centroid, regardless of how close or far it is.
- This binary decision-making doesn't account for uncertainty or overlapping clusters, which is problematic in varied-density scenarios.

Q17. What are the core parameters in DBSCAN, and how do they influence clustering?

Ans-> DBSCAN, a density-based clustering algorithm, relies on two core parameters: Epsilon (ε) and MinPts. These parameters significantly influence the clustering results, determining the density threshold for identifying clusters and handling noise. Epsilon defines the radius of a point's neighborhood, while MinPts dictates the minimum number of points required within that radius to form a dense region (and thus, a cluster).

- Here's a breakdown of how these parameters affect clustering:
- Epsilon (ε):
- Smaller ε: Results in tighter, more numerous clusters, as points need to be very close to be considered neighbors. This can lead to more noise points, as fewer points will meet the density threshold.
- Larger ε: Can merge clusters, leading to fewer, larger clusters. If ε is too large, it might include points from different clusters into a single one or classify most points as part of a single cluster. Choosing an appropriate ε: Analyzing the k-distance graph or using methods like OPTICS can help determine a suitable value. MinPts:
- Smaller MinPts: Leads to more clusters, potentially including noise points within the clusters. It can also be less robust to noise and can generate smaller, more numerous clusters.
- Larger MinPts: Results in fewer, more robust clusters. It increases the density requirement for a point to be considered a core point, leading to more points being labeled as noise or border points.
- Choosing an appropriate MinPts: A general guideline is to set MinPts to at least D+1, where D is the number of dimensions in the data.
- Influence on Cluster Formation:
- Density Threshold: Both parameters together define the density threshold for cluster formation. Points within ε distance of at least MinPts points are considered core points and form the core of a cluster.
- Noise Handling: DBSCAN effectively identifies noise by classifying points that don't meet the density criteria as outliers.
- Shape of Clusters: DBSCAN can discover clusters of arbitrary shapes, unlike algorithms like K-Means, which assume spherical clusters.
- Sensitivity: DBSCAN's performance is sensitive to the choice of ε and MinPts. Careful selection is crucial for effective clustering.

Q18. How does K-means++ improve upon standard K-means initialization?

Ans->

What Is K-means++?

K-means++ is a smarter way to **initialize the centroids** before running the standard K-means algorithm. Instead of choosing all centroids randomly, it spreads them out more strategically to reduce the chances of bad clustering.

🔍 How K-means++ Initialization Works

1. **Choose the first centroid randomly** from the data points.
2. **For each remaining point**, compute its distance (D(x)) to the nearest already chosen centroid.
3. **Select the next centroid** from the remaining points **with probability proportional to (D(x)^2)**.
   - This means points farther from existing centroids are more likely to be chosen.
4. **Repeat** until (k) centroids are chosen.
5. Proceed with standard K-means using these smarter initial centroids.

Benefits Over Standard K-means

Feature	Standard K-means	K-means++
Centroid Initialization	Random	Distance-aware, probabilistic
Risk of Poor Convergence	High	Lower
Speed of Convergence	Slower	Faster
Final Clustering Quality	Often suboptimal	More consistent and accurate

---

📈 Why It Matters

- **Reduces the chance of empty clusters** or centroids being placed too close together.
- **Improves clustering stability**, especially on complex datasets.
- **Often leads to lower total within-cluster variance**, which is the objective of K-means.

Q19. What is agglomerative clustering?

Ans-> Agglomerative clustering is a hierarchical clustering algorithm that builds a hierarchy of clusters in a "bottom-up" manner. It starts with each data point as a separate cluster and iteratively merges the closest clusters until all data points are in a single cluster. The process is visualized as a dendrogram, a tree-like diagram showing the hierarchical relationships between clusters.

- Here's a more detailed breakdown:
- How it works:

1. Initialization: Each data point is considered an individual cluster.
2. Iteration: The algorithm iteratively merges the two closest clusters based on a chosen distance metric (e.g., Euclidean distance) and linkage criteria (e.g., single, complete, average, Ward).
3. Dendrogram: The merging process is recorded, resulting in a dendrogram that visually represents the cluster hierarchy.
4. Stopping Condition: The merging process stops when all data points are in a single cluster, or when a predefined number of clusters or a distance threshold is reached.

Q20. What makes Silhouette Score a better metric than just inertia for model evaluation?

Ans-> The Silhouette Score is generally a better metric than inertia for evaluating clustering models because it considers both cohesion (how close data points are within a cluster) and separation (how far clusters are from each other), while inertia only focuses on compactness within clusters.

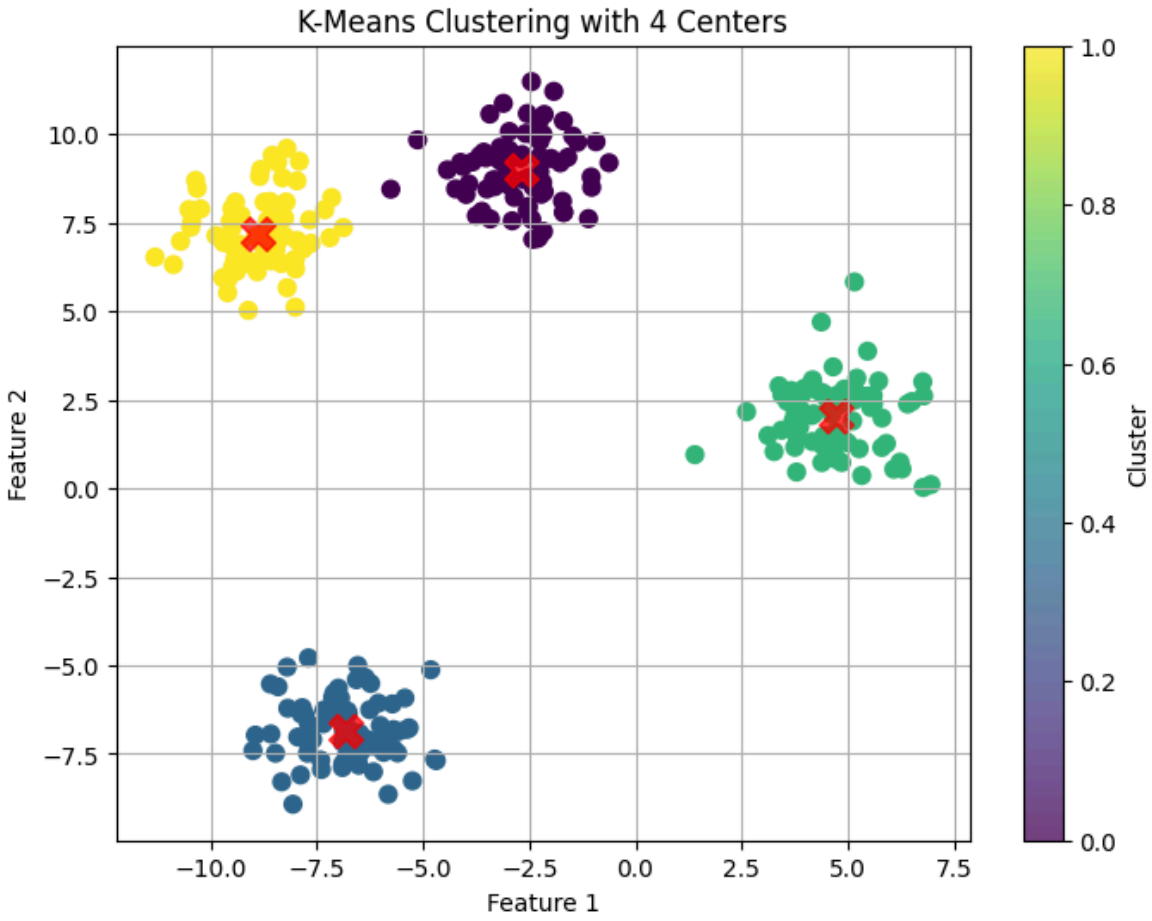
- Here's a more detailed breakdown:
- Inertia (or Within-Cluster Sum of Squares - WCSS):
- Focus: Measures the compactness of clusters by calculating the sum of squared distances of each data point to its cluster's centroid.
- Limitation: While a lower inertia indicates tighter clusters, it doesn't tell us how well-separated those clusters are. A model with very small inertia might simply have created many small clusters, which might not be a good representation of the underlying data structure.
- Silhouette Score:
- Focus: Calculates a score for each data point, representing how well it fits within its cluster compared to other clusters. It considers both the average distance to other points within the same cluster (cohesion) and the average distance to points in the nearest neighboring cluster (separation).
- Range: The Silhouette Score ranges from -1 to +1. A score close to +1 indicates well-defined, separated clusters, while a score close to 0 suggests overlapping or poorly separated clusters. A negative score indicates that points might be assigned to the wrong cluster.
- Advantage: By considering both cohesion and separation, the Silhouette Score provides a more holistic view of clustering quality, helping to identify if clusters are distinct and well-formed.
- Why Silhouette Score is preferred:
- Better Interpretation: The Silhouette Score provides a more intuitive understanding of how well clusters are separated, which is crucial for many real-world clustering * applications.
- Avoids Overfitting: While inertia can be minimized by creating many small clusters, the Silhouette Score can help prevent overfitting by highlighting when the separation between clusters is not optimal.

- More Robust: It is less sensitive to the number of clusters compared to inertia, making it a more reliable metric for comparing different clustering solutions.

Practical Questions:

Q21. Generate synthetic data with 4 centers using make_blobs and apply K-means clustering. Visualize using a Scatter plot.

In [ ]:
```
# prompt: Generate synthetic data with 4 centers using make_blobs and apply K-me

import matplotlib.pyplot as plt
from sklearn.datasets import make_blobs
from sklearn.cluster import KMeans

# Generate synthetic data with 4 centers
n_samples = 300
n_features = 2
n_centers = 4
X, y = make_blobs(n_samples=n_samples, n_features=n_features, centers=n_centers,

# Apply K-means clustering
kmeans = KMeans(n_clusters=n_centers, random_state=42)
kmeans.fit(X)
y_kmeans = kmeans.predict(X)

# Visualize the data and the clusters
plt.figure(figsize=(8, 6))
plt.scatter(X[:, 0], X[:, 1], c=y_kmeans, s=50, cmap='viridis')

# Plot the cluster centers
centers = kmeans.cluster_centers_
plt.scatter(centers[:, 0], centers[:, 1], c='red', s=200, alpha=0.75, marker='X'

plt.title('K-Means Clustering with 4 Centers')
plt.xlabel('Feature 1')
plt.ylabel('Feature 2')
plt.colorbar(label='Cluster')
plt.grid(True)
plt.show()
```



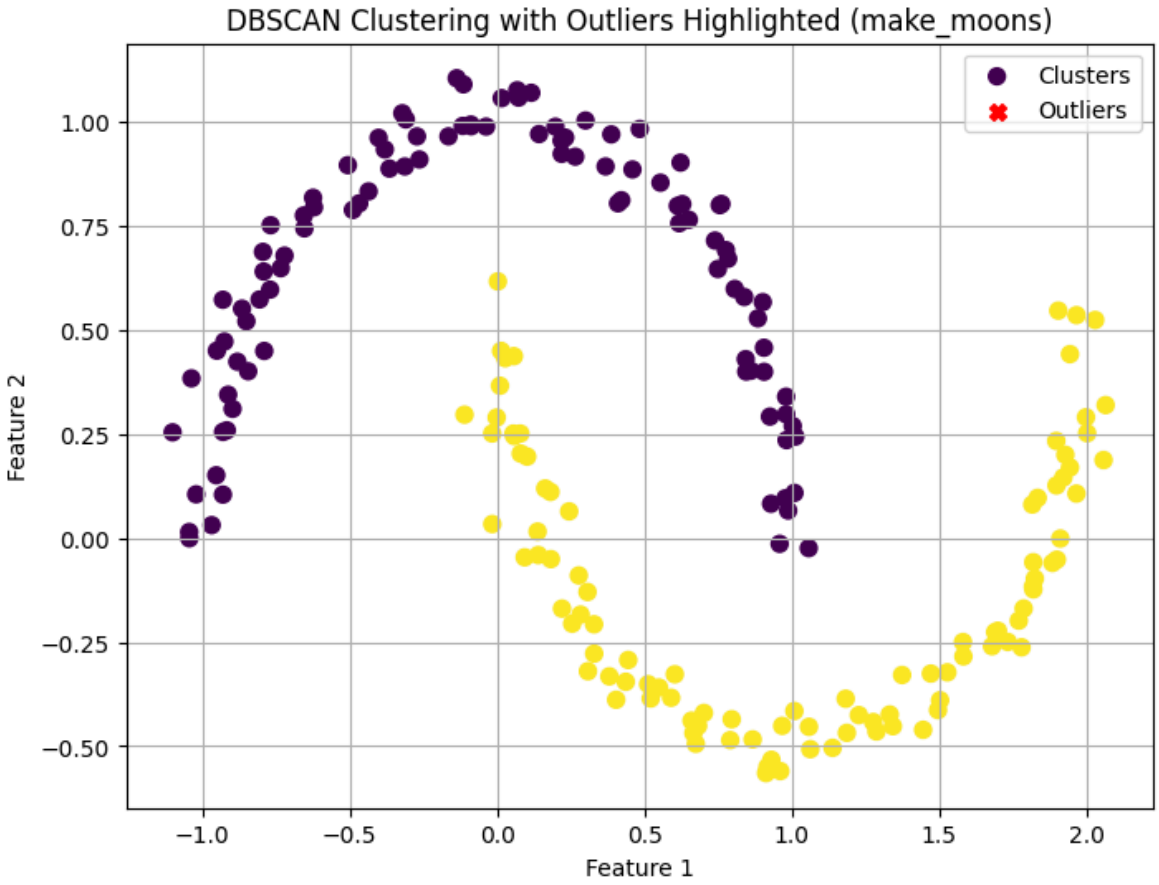
Q22. Load the Iris dataset and use Agglomerative Clustering to group the data into 3 clusters. Display the first 10 predicted labels.

In [1]:
```
# prompt: Load the Iris dataset and use Agglomerative Clustering to group the da

from sklearn.datasets import load_iris
from sklearn.cluster import AgglomerativeClustering
import numpy as np

# Load the Iris dataset
iris = load_iris()
X = iris.data

# Use Agglomerative Clustering
n_clusters = 3
agg_clustering = AgglomerativeClustering(n_clusters=n_clusters)
agg_clustering.fit(X)

# Get the predicted labels
predicted_labels = agg_clustering.labels_

# Display the first 10 predicted labels
print("First 10 predicted labels:", predicted_labels[:10])
```

```
First 10 predicted labels: [1 1 1 1 1 1 1 1 1 1]
```

Q23. Generate synthethic data using make_moons and apply DBSCAN. Highlight outliers in the plot.

In [5]:
```
# prompt: Generate synthethic data using make_moons and apply DBSCAN. Highlight

from sklearn.datasets import make_moons
from sklearn.cluster import DBSCAN
import matplotlib.pyplot as plt

# Generate synthetic data using make_moons
n_samples = 200
X, _ = make_moons(n_samples=n_samples, noise=0.05, random_state=42)

# Apply DBSCAN
# Adjust eps and min_samples as needed based on the data density
dbscan = DBSCAN(eps=0.3, min_samples=5)
clusters = dbscan.fit_predict(X)

# Highlight outliers in the plot
plt.figure(figsize=(8, 6))
# Points labeled as -1 are considered outliers by DBSCAN
outliers_mask = (clusters == -1)
clustered_mask = (clusters != -1)

plt.scatter(X[clustered_mask, 0], X[clustered_mask, 1], c=clusters[clustered_mas
plt.scatter(X[outliers_mask, 0], X[outliers_mask, 1], c='red', s=50, marker='X',

plt.title('DBSCAN Clustering with Outliers Highlighted (make_moons)')
plt.xlabel('Feature 1')
plt.ylabel('Feature 2')
plt.legend()
plt.grid(True)
plt.show()
```



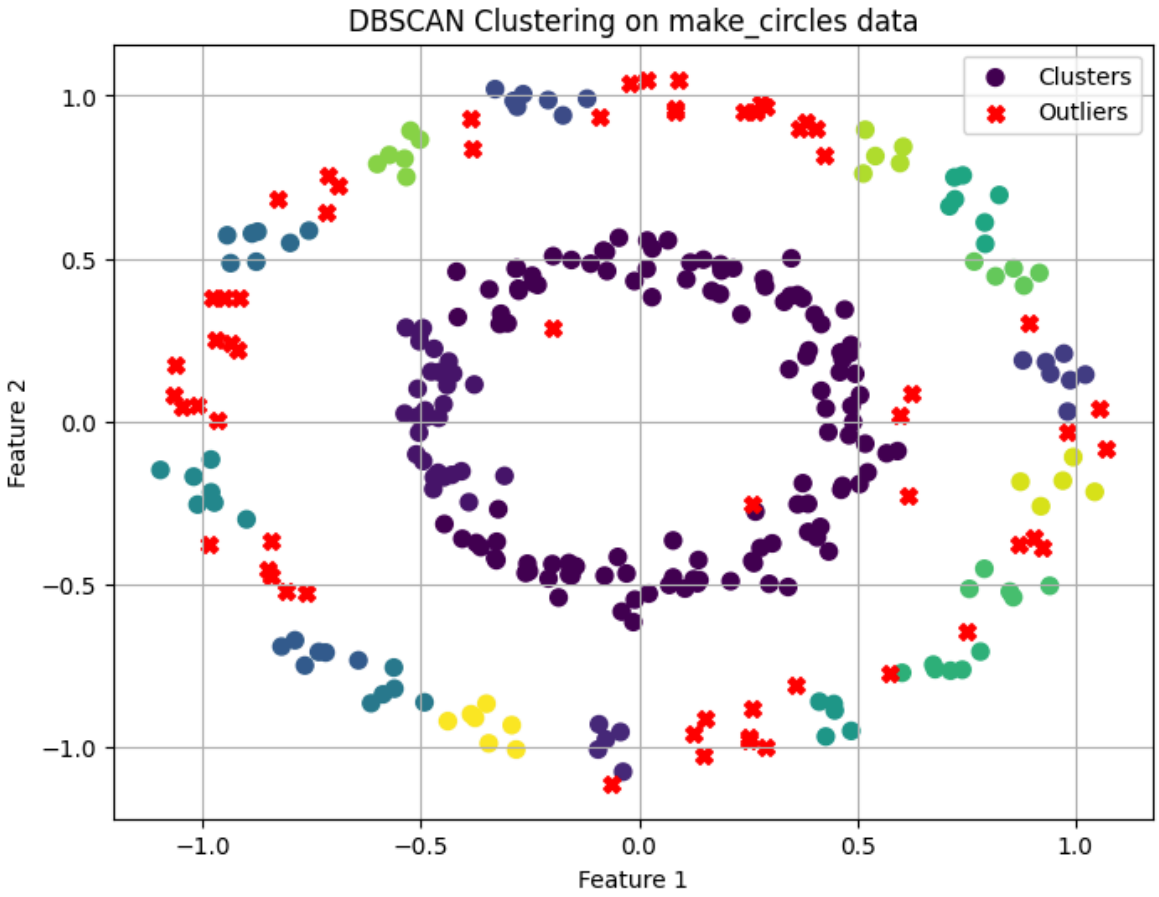
Q24. Load the Wine dataset and apply K-means clustering after standardizing in the features. Print the size of each cluster.

In [6]:
```
# prompt: Load the Wine dataset and apply K-means clustering after standardizing

from sklearn.datasets import load_wine
from sklearn.preprocessing import StandardScaler
from sklearn.cluster import KMeans

# Load the Wine dataset
wine = load_wine()
X = wine.data

# Standardize the features
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)

# Apply K-means clustering (assuming 3 clusters based on the original dataset cl
kmeans = KMeans(n_clusters=3, random_state=42, n_init=10)
kmeans.fit(X_scaled)
y_kmeans = kmeans.predict(X_scaled)

# Print the size of each cluster
unique_labels, counts = np.unique(y_kmeans, return_counts=True)
for label, count in zip(unique_labels, counts):
  print(f"Cluster {label}: {count} samples")
```

```
Cluster 0: 65 samples
Cluster 1: 51 samples
Cluster 2: 62 samples
```

Q25. Use make_circles to generate synthetic data and cluster it using DBSCAN. plot the result.

In [7]:
```python
# prompt: Use make_circles to generate synthetic data and cluster it using DBSCA

from sklearn.datasets import make_circles

# Generate synthetic data using make_circles
n_samples = 300
X, _ = make_circles(n_samples=n_samples, noise=0.05, factor=0.5, random_state=42

# Apply DBSCAN
# Adjust eps and min_samples as needed based on the data density of circles
dbscan = DBSCAN(eps=0.1, min_samples=5)
clusters = dbscan.fit_predict(X)

# Plot the result
plt.figure(figsize=(8, 6))
# Points labeled as -1 are considered outliers by DBSCAN
outliers_mask = (clusters == -1)
clustered_mask = (clusters != -1)

plt.scatter(X[clustered_mask, 0], X[clustered_mask, 1], c=clusters[clustered_mas
plt.scatter(X[outliers_mask, 0], X[outliers_mask, 1], c='red', s=50, marker='X',

plt.title('DBSCAN Clustering on make_circles data')
plt.xlabel('Feature 1')
plt.ylabel('Feature 2')
plt.legend()
plt.grid(True)
plt.show()
```



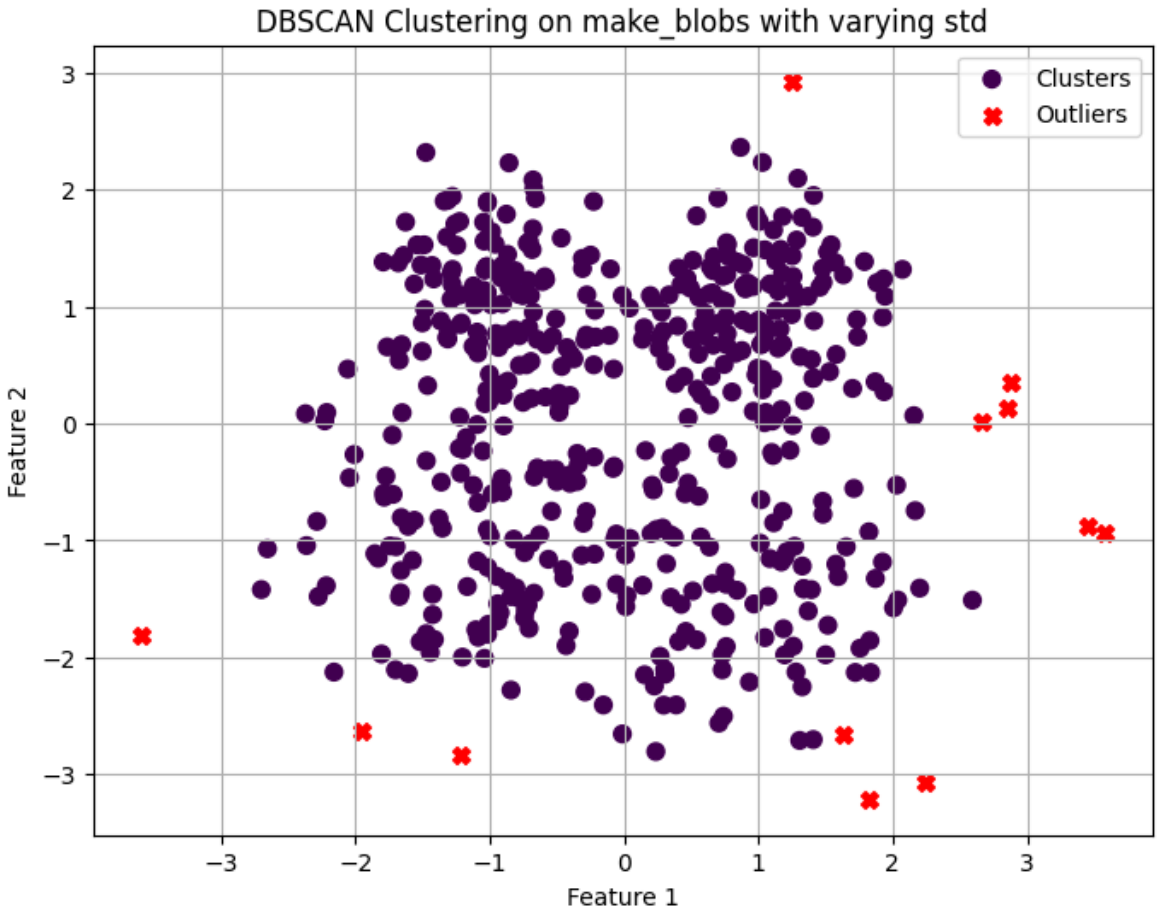
Q26. Load the Breast Cancer dataset, apply MinMaxScaler, and use K-means with 2 clusters. Output the cluster centroids.

In [8]:
```
# prompt: Load the Breast Cancer dataset, apply MinMaxScaler, and use K-means wi

from sklearn.datasets import load_breast_cancer
from sklearn.preprocessing import MinMaxScaler

# Load the Breast Cancer dataset
breast_cancer = load_breast_cancer()
X = breast_cancer.data

# Apply MinMaxScaler
scaler = MinMaxScaler()
X_scaled = scaler.fit_transform(X)

# Use K-means with 2 clusters
kmeans = KMeans(n_clusters=2, random_state=42, n_init=10)
kmeans.fit(X_scaled)

# Output the cluster centroids
centroids = kmeans.cluster_centers_
print("Cluster Centroids:\n", centroids)
```

```
Cluster Centroids:
 [[0.50483563 0.39560329 0.50578661 0.36376576 0.46988732 0.42226302
  0.41838662 0.46928035 0.45899738 0.29945886 0.19093085 0.19112073
  0.17903433 0.13086432 0.18017962 0.25890126 0.12542475 0.30942779
  0.190072   0.13266975 0.48047448 0.45107371 0.4655302  0.31460597
  0.49868817 0.36391461 0.39027292 0.65827197 0.33752296 0.26041387]
 [0.25535358 0.28833455 0.24696416 0.14388369 0.35743076 0.18019471
  0.10344776 0.1306603  0.34011829 0.25591606 0.06427485 0.18843043
  0.05975663 0.02870108 0.18158628 0.13242941 0.05821528 0.18069336
  0.17221057 0.08403996 0.2052406  0.32069002 0.19242138 0.09943446
  0.3571115  0.14873935 0.13142287 0.26231363 0.22639412 0.15437354]]
```

Q27. Generate synthetic data using make_blobs with varying cluster standard deviations and cluster with DBSCAN

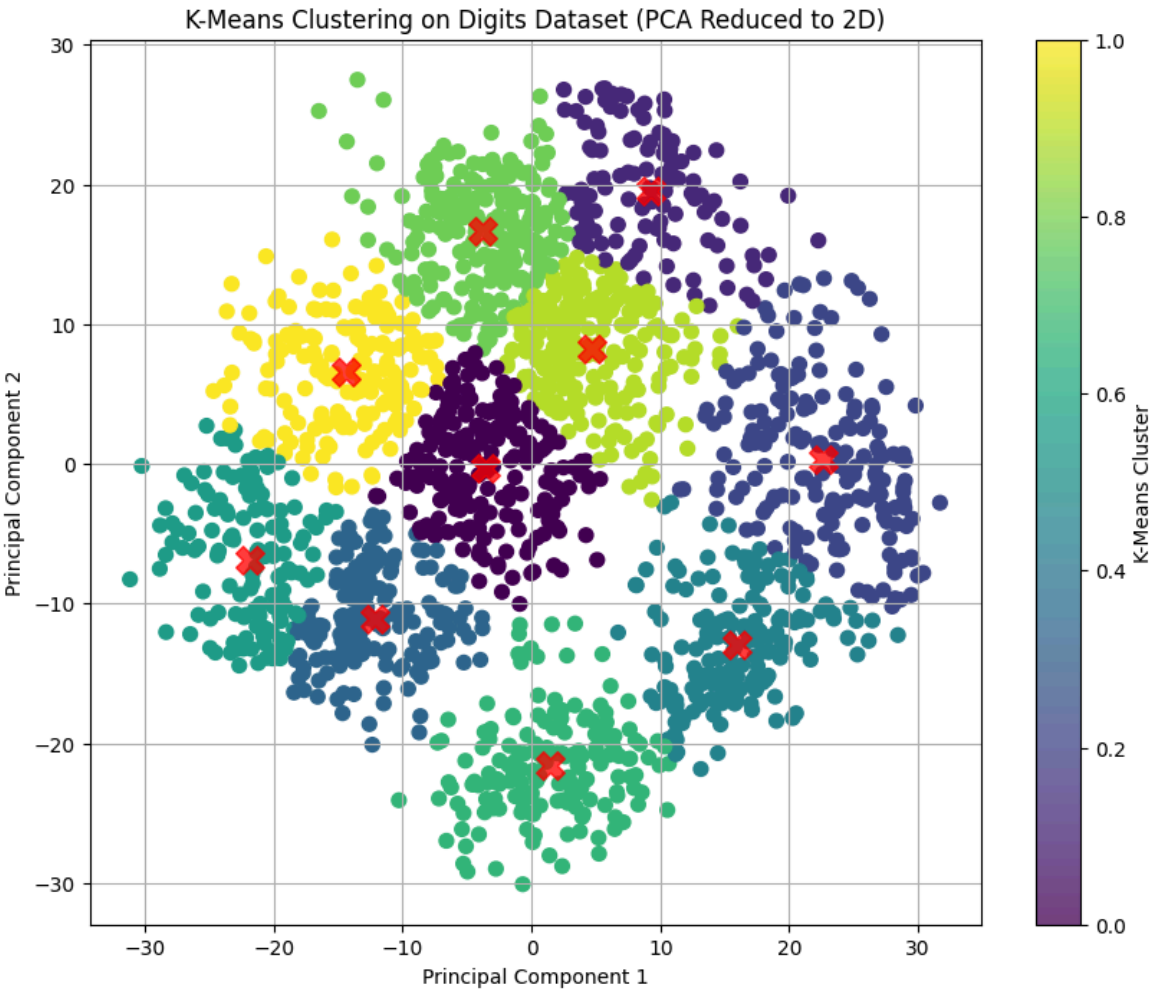
In [11]:
```python
# prompt: Generate synthetic data using make_blobs with varying cluster standard

from sklearn.datasets import make_blobs
from sklearn.cluster import DBSCAN
import matplotlib.pyplot as plt

# Generate synthetic data with varying cluster standard deviations
n_samples = 500
n_features = 2
centers = [[1, 1], [-1, -1], [1, -1], [-1, 1]]
cluster_std = [0.5, 0.8, 1.0, 0.5] # Varying standard deviations

X, y = make_blobs(n_samples=n_samples, n_features=n_features, centers=centers,
                  cluster_std=cluster_std, random_state=42)

# Apply DBSCAN
# Adjust eps and min_samples based on the data density
dbscan = DBSCAN(eps=0.5, min_samples=5)
clusters = dbscan.fit_predict(X)

# Plot the result
plt.figure(figsize=(8, 6))
# Points labeled as -1 are considered outliers by DBSCAN
outliers_mask = (clusters == -1)
clustered_mask = (clusters != -1)

plt.scatter(X[clustered_mask, 0], X[clustered_mask, 1], c=clusters[clustered_mas
plt.scatter(X[outliers_mask, 0], X[outliers_mask, 1], c='red', s=50, marker='X',

plt.title('DBSCAN Clustering on make_blobs with varying std')
plt.xlabel('Feature 1')
plt.ylabel('Feature 2')
plt.legend()
plt.grid(True)
plt.show()
```



Q28. Load the Digits dataset, reduce it to 2D using PCA, and visualize clusters from K-means.

In [12]:
```
# prompt: Load the Digits dataset, reduce it to 2D using PCA, and visualize clus

from sklearn.datasets import load_digits
from sklearn.decomposition import PCA
from sklearn.cluster import KMeans

# Load the Digits dataset
digits = load_digits()
X = digits.data
y = digits.target # True labels

# Reduce the data to 2D using PCA
n_components = 2
pca = PCA(n_components=n_components)
X_pca = pca.fit_transform(X)

# Apply K-means clustering (assuming 10 clusters for digits 0-9)
n_clusters = 10
kmeans = KMeans(n_clusters=n_clusters, random_state=42, n_init=10)
kmeans.fit(X_pca)
y_kmeans = kmeans.predict(X_pca)

# Visualize the clusters from K-means
plt.figure(figsize=(10, 8))
plt.scatter(X_pca[:, 0], X_pca[:, 1], c=y_kmeans, s=50, cmap='viridis')
```

```
# Optional: Plot the cluster centers
centers = kmeans.cluster_centers_
plt.scatter(centers[:, 0], centers[:, 1], c='red', s=200, alpha=0.75, marker='X'

plt.title('K-Means Clustering on Digits Dataset (PCA Reduced to 2D)')
plt.xlabel('Principal Component 1')
plt.ylabel('Principal Component 2')
plt.colorbar(label='K-Means Cluster')
plt.grid(True)
plt.show()
```

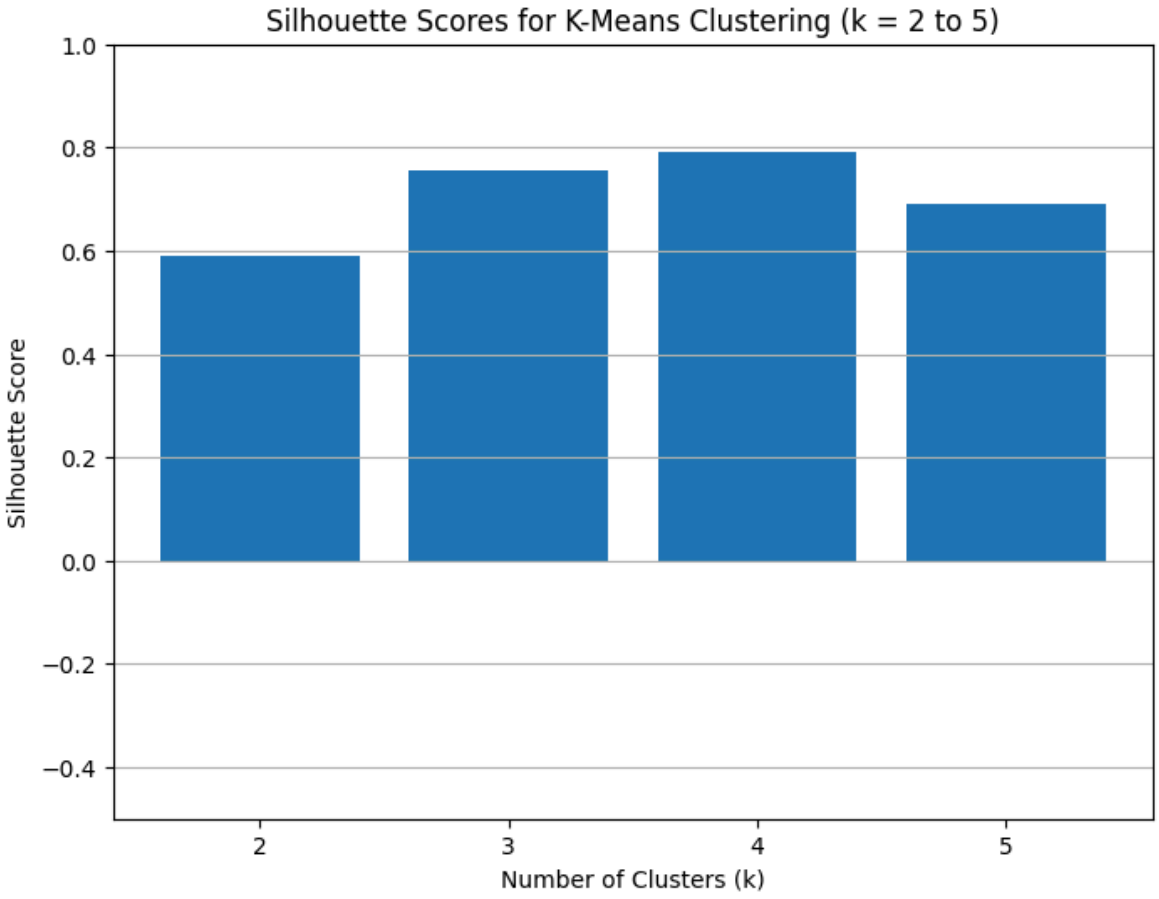


Q29. Create synthetic data using make_blobs and evaluate silhouette scores for k = 2 to 5. Display as a bar chart.

In [13]:
```
# prompt: Create synthetic data using make_blobs and evaluate silhouette scores

from sklearn.metrics import silhouette_score

# Generate synthetic data with make_blobs
n_samples = 300
n_features = 2
X, y_true = make_blobs(n_samples=n_samples, n_features=n_features, centers=4, ra

# Evaluate silhouette scores for k from 2 to 5
silhouette_scores = []
k_values = range(2, 6)
```

```
for k in k_values:
  kmeans = KMeans(n_clusters=k, random_state=42, n_init=10)
  kmeans.fit(X)
  labels = kmeans.predict(X)
  score = silhouette_score(X, labels)
  silhouette_scores.append(score)
  print(f"Silhouette Score for k = {k}: {score:.4f}")

# Display as a bar chart
plt.figure(figsize=(8, 6))
plt.bar(k_values, silhouette_scores)
plt.title('Silhouette Scores for K-Means Clustering (k = 2 to 5)')
plt.xlabel('Number of Clusters (k)')
plt.ylabel('Silhouette Score')
plt.xticks(k_values)
plt.ylim(-0.5, 1)
plt.grid(axis='y')
plt.show()
```

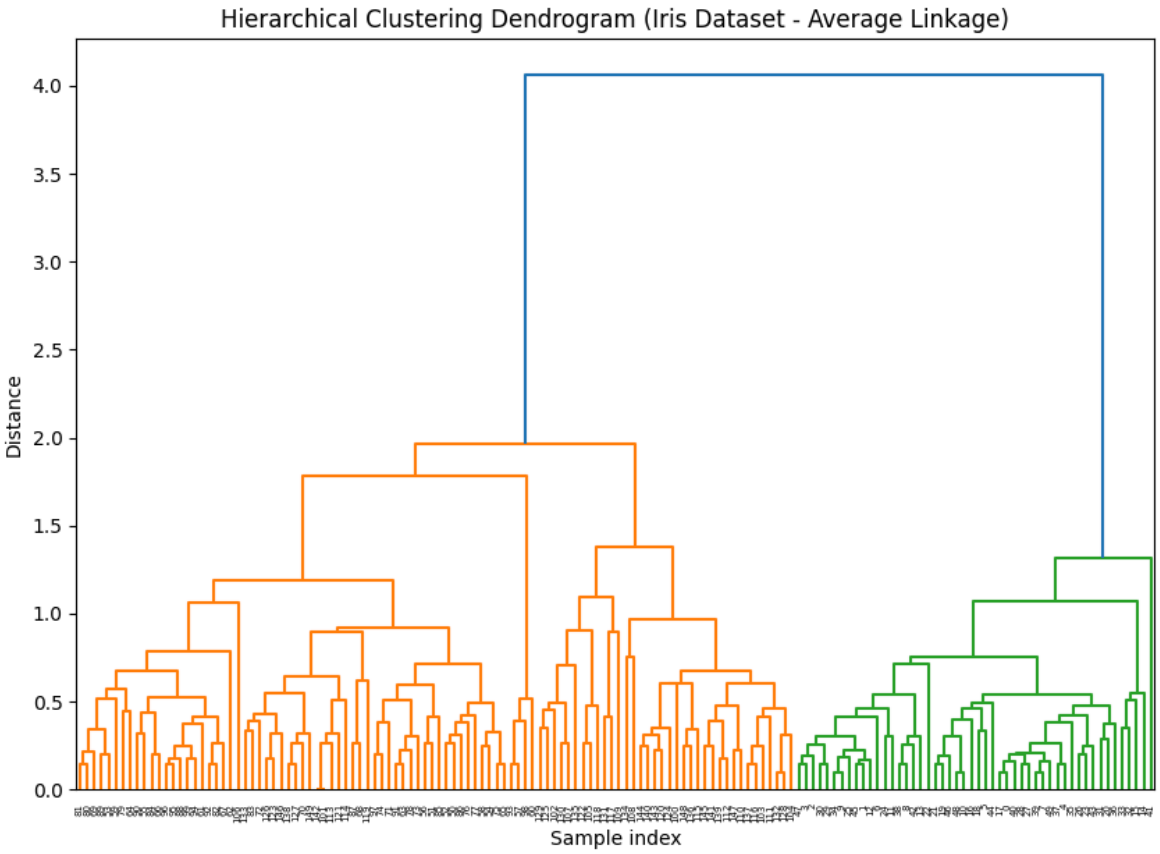
```
Silhouette Score for k = 2: 0.5902
Silhouette Score for k = 3: 0.7569
Silhouette Score for k = 4: 0.7916
Silhouette Score for k = 5: 0.6890
```



Q30. Load the Iris dataset and use hierarchical clustering to group data. Plot a dendogram with average linkage

In [14]:
```
# prompt: Load the Iris dataset and use hierarchical clustering to group data. P
```

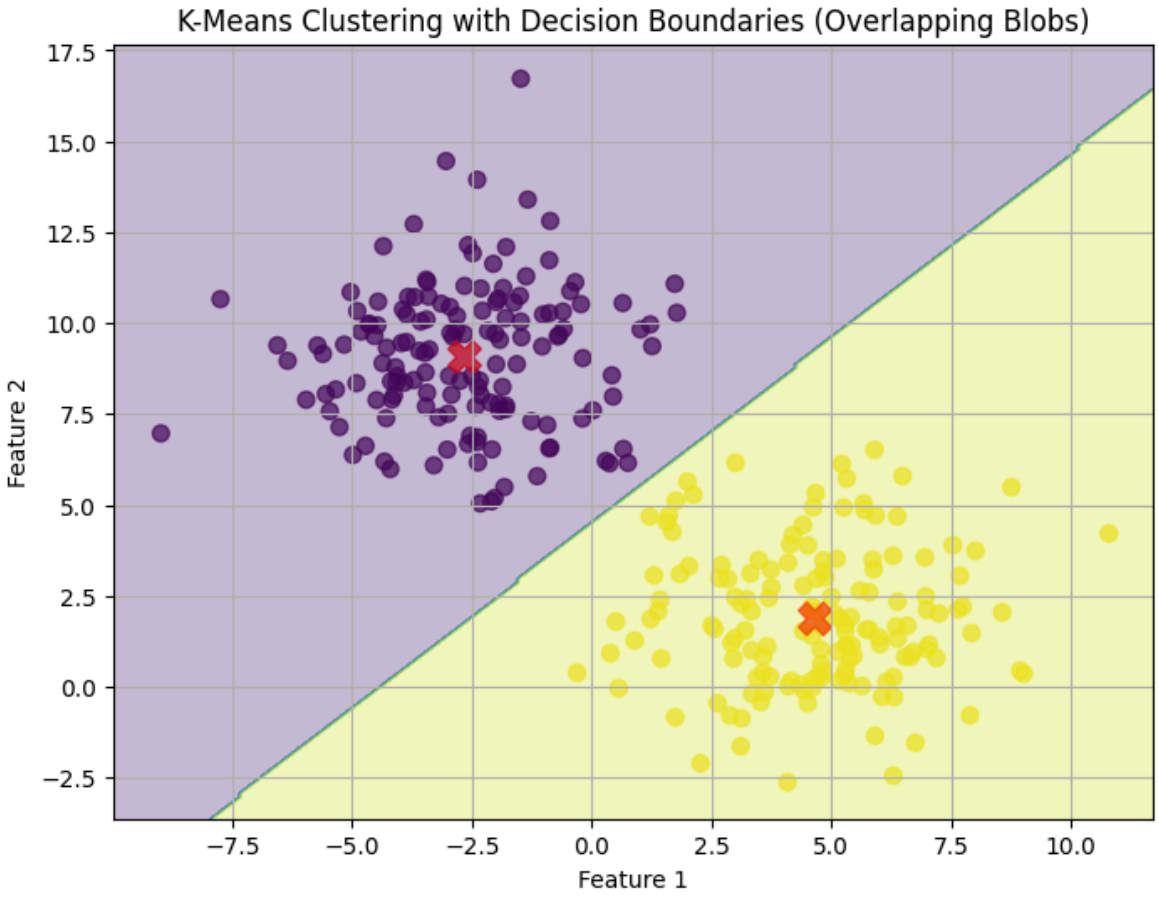
```
from scipy.cluster.hierarchy import dendrogram, linkage

# Load the Iris dataset
iris = load_iris()
X = iris.data

# Use hierarchical clustering with average linkage
linked = linkage(X, 'average')

# Plot the dendrogram
plt.figure(figsize=(10, 7))
dendrogram(linked,
           orientation='top',
           distance_sort='descending',
           show_leaf_counts=True)
plt.title('Hierarchical Clustering Dendrogram (Iris Dataset - Average Linkage)')
plt.xlabel('Sample index')
plt.ylabel('Distance')
plt.show()
```



Q31. Generate synthetic data with overlapping clusters using make_blobs, then apply K-means and visualize with decision boundaries

In [15]:
```
# prompt: Generate synthetic data with overlapping clusters using make_blobs, th

# Generate synthetic data with overlapping clusters
n_samples = 300
n_features = 2
n_centers = 2 # Using 2 centers for potentially overlapping clusters
cluster_std = 2.0 # Increase standard deviation for overlap
```

```
X, y = make_blobs(n_samples=n_samples, n_features=n_features, centers=n_centers,
                  cluster_std=cluster_std, random_state=42)

# Apply K-means clustering
kmeans = KMeans(n_clusters=n_centers, random_state=42, n_init=10)
kmeans.fit(X)
y_kmeans = kmeans.predict(X)

# Visualize with decision boundaries
plt.figure(figsize=(8, 6))

# Plot the data points
plt.scatter(X[:, 0], X[:, 1], c=y_kmeans, s=50, cmap='viridis', alpha=0.7)

# Plot the cluster centers
centers = kmeans.cluster_centers_
plt.scatter(centers[:, 0], centers[:, 1], c='red', s=200, alpha=0.75, marker='X'

# Plot the decision boundaries
x_min, x_max = X[:, 0].min() - 1, X[:, 0].max() + 1
y_min, y_max = X[:, 1].min() - 1, X[:, 1].max() + 1
xx, yy = np.meshgrid(np.arange(x_min, x_max, 0.1),
                     np.arange(y_min, y_max, 0.1))

Z = kmeans.predict(np.c_[xx.ravel(), yy.ravel()])
Z = Z.reshape(xx.shape)

plt.contourf(xx, yy, Z, cmap='viridis', alpha=0.3)

plt.title('K-Means Clustering with Decision Boundaries (Overlapping Blobs)')
plt.xlabel('Feature 1')
plt.ylabel('Feature 2')
plt.grid(True)
plt.show()
```

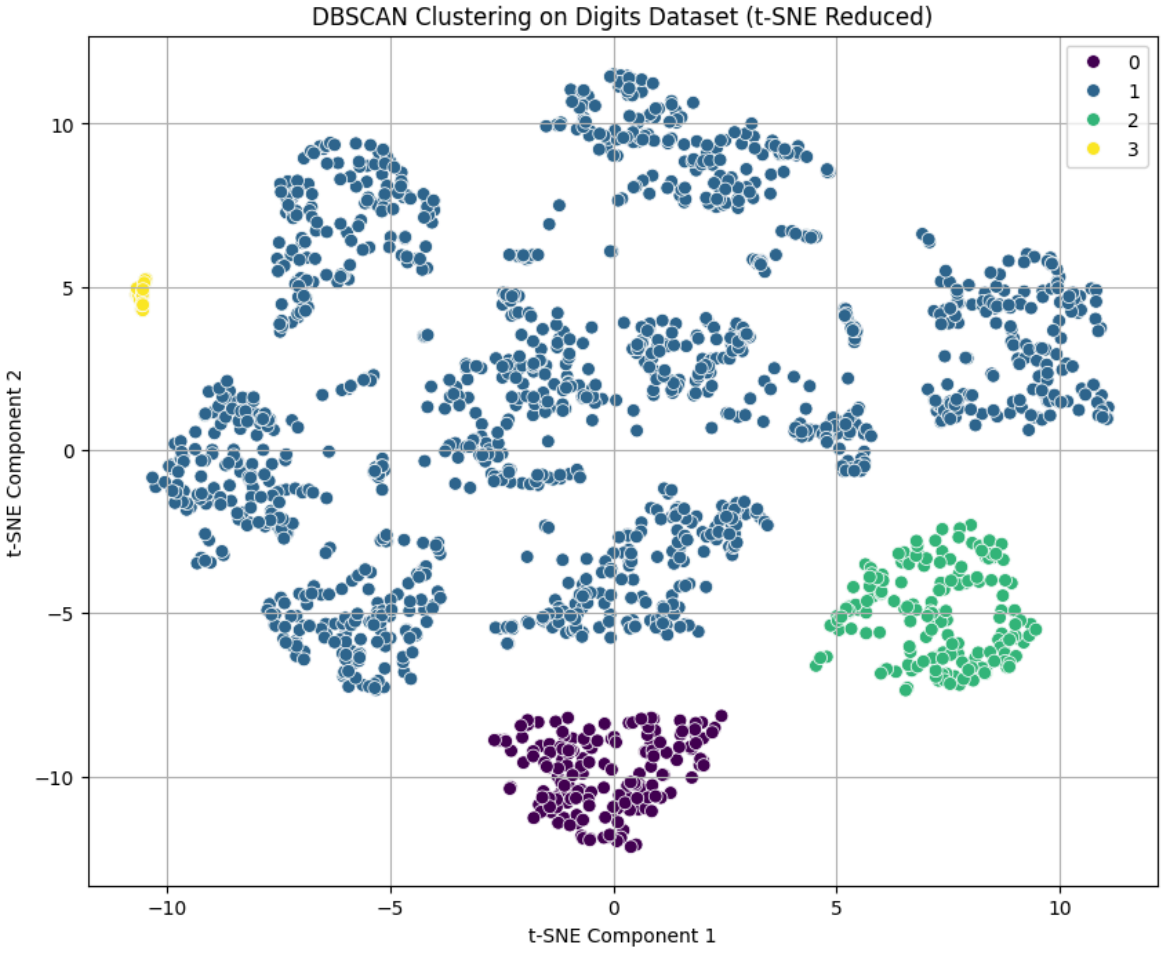


Q32. Load the Digits dataset and apply DBSCAN after reducing dimensions with t-SNE. Visualize the results

In [17]:
```
# prompt: Load the Digits dataset and apply DBSCAN after reducing dimensions wit

from sklearn.manifold import TSNE
import seaborn as sns

# Load the Digits dataset
digits = load_digits()
X = digits.data
y = digits.target # True labels

# Reduce the data to 2D using t-SNE
# Adjust perplexity and n_iter for better visualization if needed
tsne = TSNE(n_components=2, random_state=42, perplexity=30, n_iter=300)
X_tsne = tsne.fit_transform(X)

# Apply DBSCAN on the t-SNE reduced data
# Adjust eps and min_samples based on the visual density of the t-SNE plot
dbscan = DBSCAN(eps=2, min_samples=10) # These parameters might need tuning
clusters = dbscan.fit_predict(X_tsne)

# Visualize the results
plt.figure(figsize=(10, 8))
# Use seaborn for better color mapping, especially for -1 (outliers)
sns.scatterplot(x=X_tsne[:, 0], y=X_tsne[:, 1], hue=clusters, palette='viridis',

plt.title('DBSCAN Clustering on Digits Dataset (t-SNE Reduced)')
```

```
plt.xlabel('t-SNE Component 1')
plt.ylabel('t-SNE Component 2')
plt.grid(True)
plt.show()
```

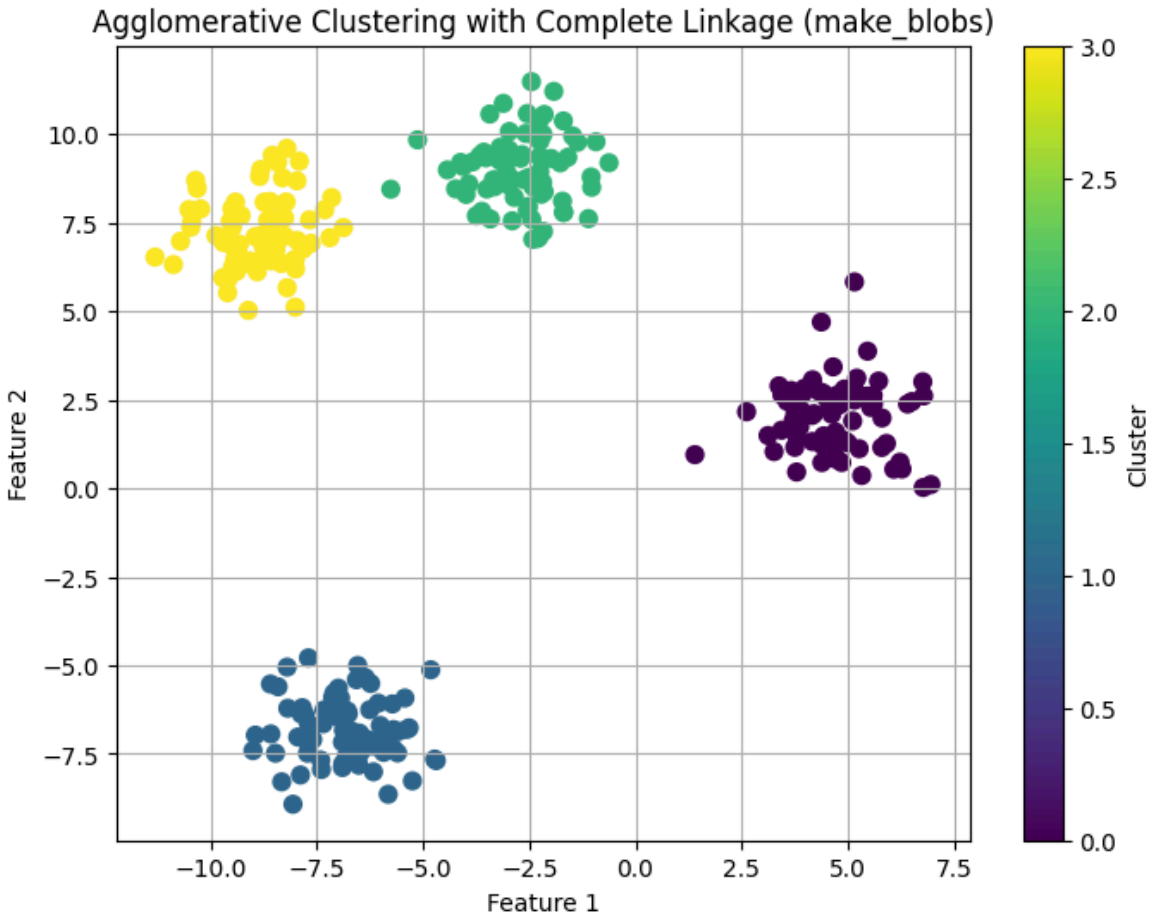
```
/usr/local/lib/python3.11/dist-packages/sklearn/manifold/_t_sne.py:1164: FutureWa
rning: 'n_iter' was renamed to 'max_iter' in version 1.5 and will be removed in
1.7.
  warnings.warn(
```



Q33. Generate synthetic data using make_blobs and apply Agglomerative Clustering with complete linkage. Plot the result

In [18]:
```
# prompt: Generate synthetic data using make_blobs and apply Agglomerative Clust

# Generate synthetic data using make_blobs
n_samples = 300
n_features = 2
n_centers = 4
X, y_true = make_blobs(n_samples=n_samples, n_features=n_features, centers=n_cen

# Apply Agglomerative Clustering with complete linkage
n_clusters = 4 # Assuming the same number of clusters as in make_blobs
agg_clustering = AgglomerativeClustering(n_clusters=n_clusters, linkage='complet
agg_labels = agg_clustering.fit_predict(X)

# Plot the result
plt.figure(figsize=(8, 6))
```

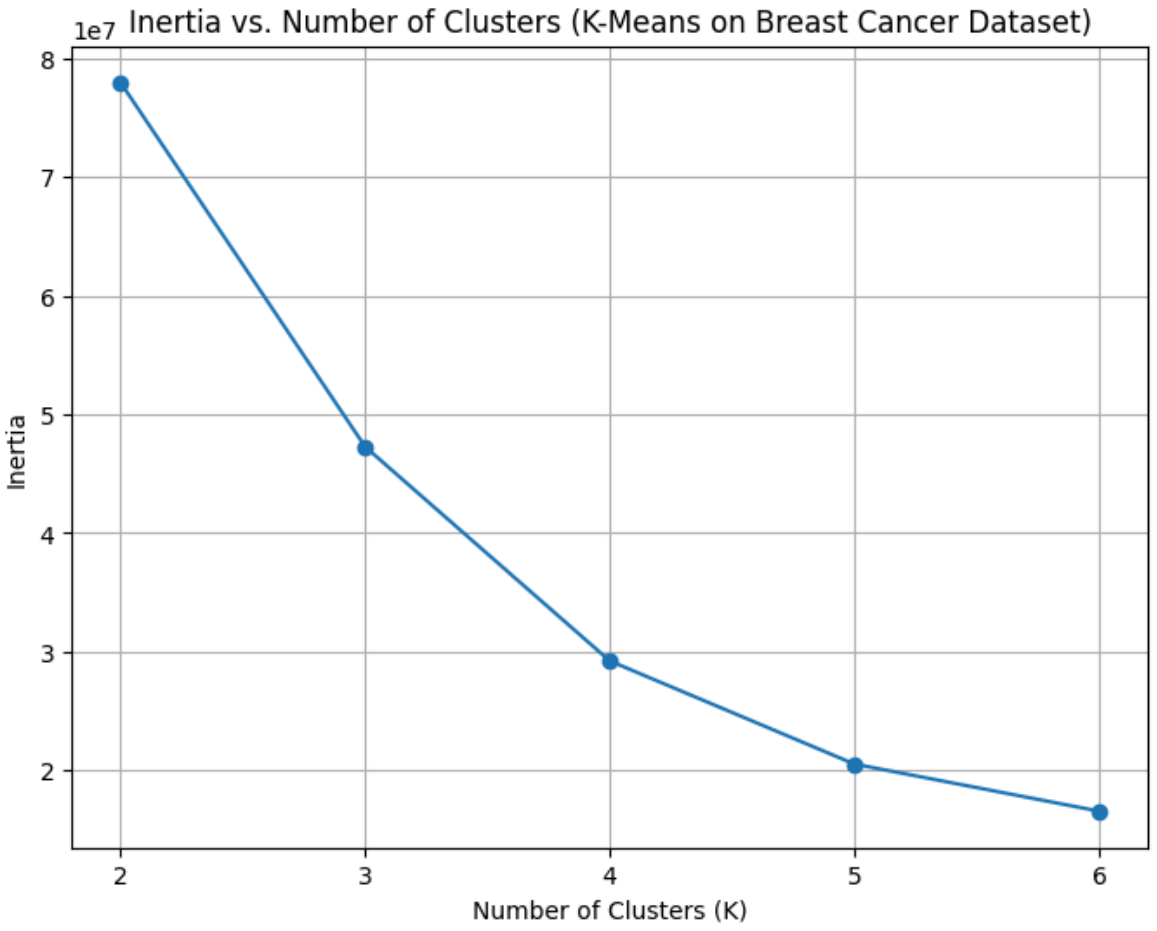
```
plt.scatter(X[:, 0], X[:, 1], c=agg_labels, s=50, cmap='viridis')

plt.title('Agglomerative Clustering with Complete Linkage (make_blobs)')
plt.xlabel('Feature 1')
plt.ylabel('Feature 2')
plt.colorbar(label='Cluster')
plt.grid(True)
plt.show()
```



Q34. Load the Breast Cancer dataset and compare inertia values for K = 2 to 6 using K-means. show results in a line plot

In [19]:
```
# prompt: Load the Breast Cancer dataset and compare inertia values for K = 2 to

# Load the Breast Cancer dataset
breast_cancer = load_breast_cancer()
X = breast_cancer.data

# Compare inertia for K = 2 to 6
inertia_values = []
k_values = range(2, 7)

for k in k_values:
  kmeans = KMeans(n_clusters=k, random_state=42, n_init=10)
  kmeans.fit(X)
  inertia_values.append(kmeans.inertia_)

# Show results in a line plot
```

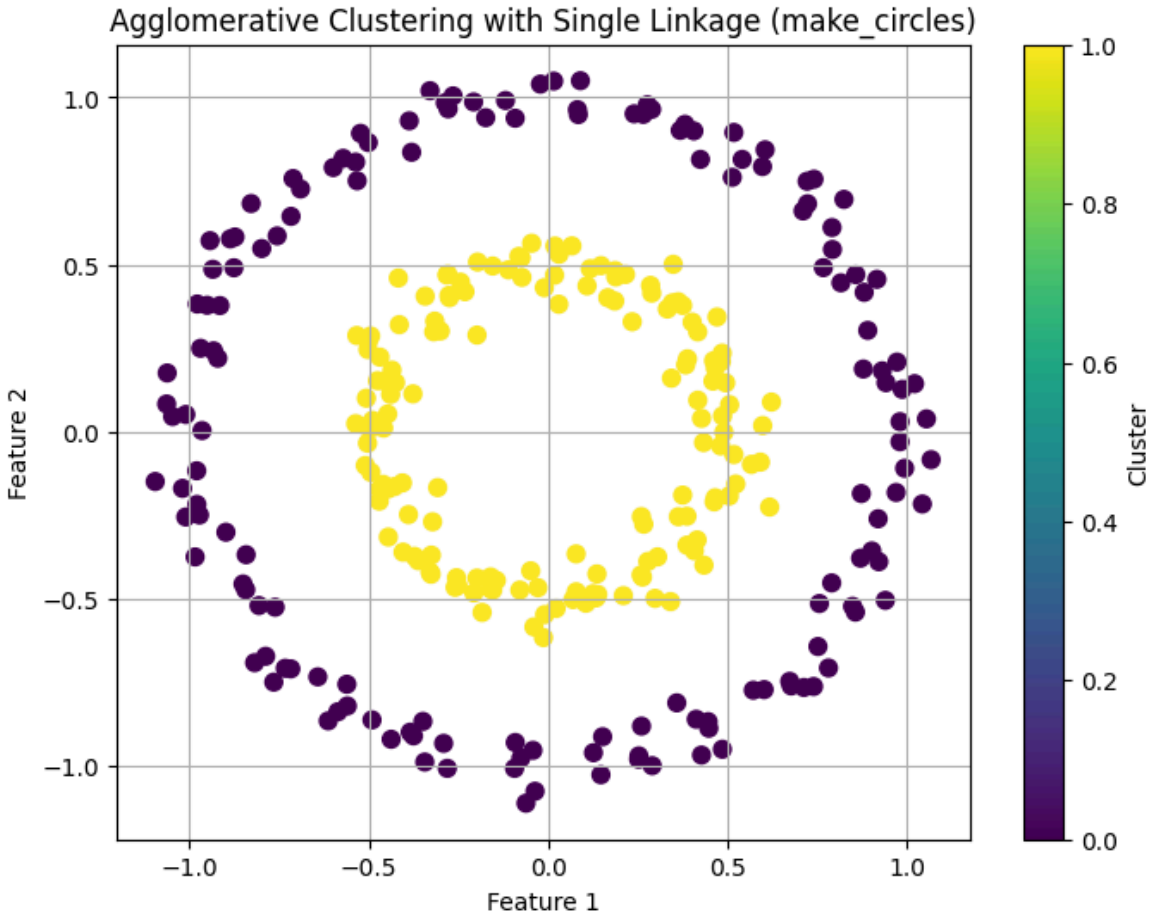
```
plt.figure(figsize=(8, 6))
plt.plot(k_values, inertia_values, marker='o')
plt.title('Inertia vs. Number of Clusters (K-Means on Breast Cancer Dataset)')
plt.xlabel('Number of Clusters (K)')
plt.ylabel('Inertia')
plt.xticks(k_values)
plt.grid(True)
plt.show()
```



Q35. Generate synthetic concentric circles using make_circles and cluster using Agglomerative Clustering with single linkage

In [20]:
```
# prompt: Generate synthetic concentric circles using make_circles and cluster u

# Generate synthetic concentric circles
n_samples = 300
X, y_true = make_circles(n_samples=n_samples, noise=0.05, factor=0.5, random_sta

# Cluster using Agglomerative Clustering with single linkage
# For concentric circles, single linkage often works well as it connects nearby
# and can trace the shape of the circles. We expect 2 clusters.
n_clusters = 2
agg_clustering = AgglomerativeClustering(n_clusters=n_clusters, linkage='single'
agg_labels = agg_clustering.fit_predict(X)

# Plot the result
plt.figure(figsize=(8, 6))
plt.scatter(X[:, 0], X[:, 1], c=agg_labels, s=50, cmap='viridis')
```

```
plt.title('Agglomerative Clustering with Single Linkage (make_circles)')
plt.xlabel('Feature 1')
plt.ylabel('Feature 2')
plt.colorbar(label='Cluster')
plt.grid(True)
plt.show()
```



Q36. Use the wine dataset, apply DBSCAN after scaling the data, and count the number of clusters (excluding noise)

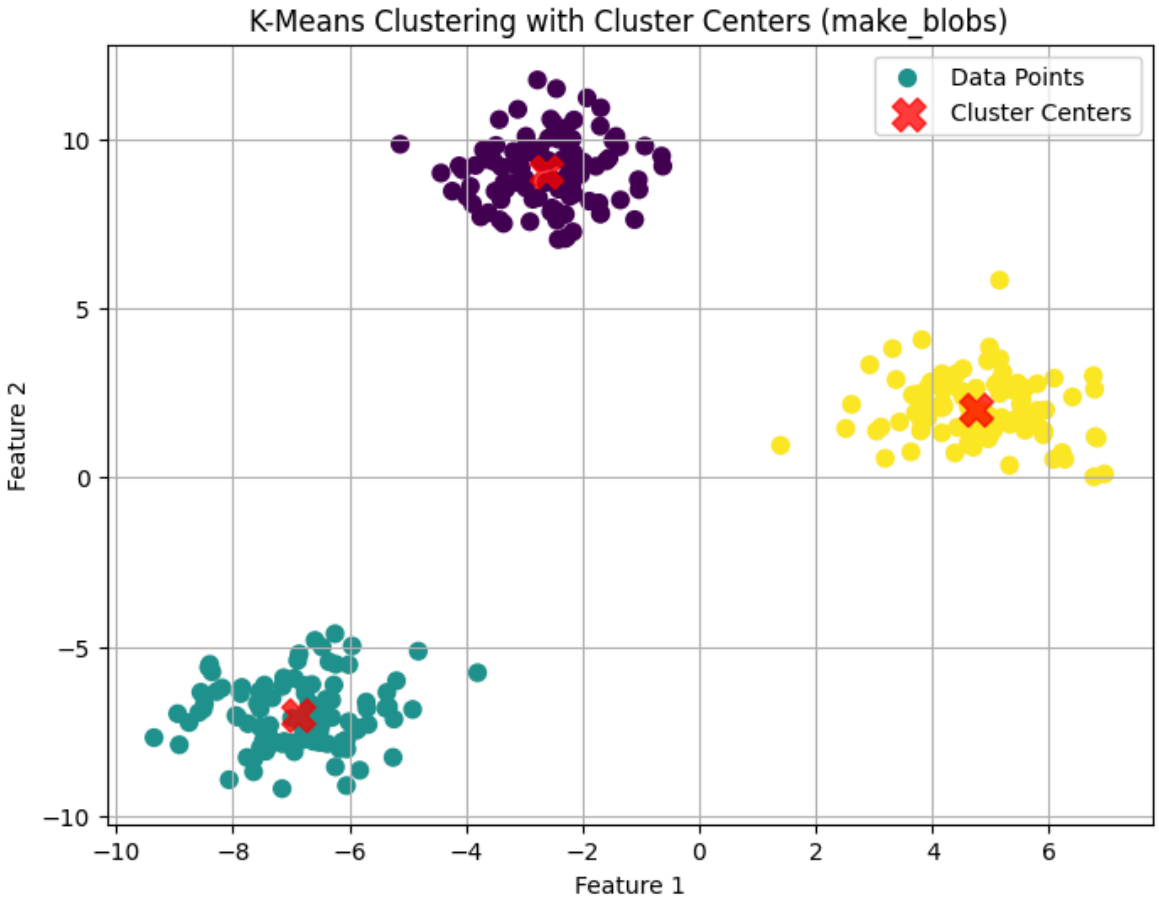
In [21]:
```
# prompt: Use the wine dataset, apply DBSCAN after scaling the data, and count t

# Load the Wine dataset
wine = load_wine()
X = wine.data

# Scale the data
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)

# Apply DBSCAN
# These parameters might need tuning based on the dataset's characteristics
# A common approach is to use a k-distance plot to find a suitable eps
dbscan = DBSCAN(eps=0.8, min_samples=5)
clusters = dbscan.fit_predict(X_scaled)

# Count the number of clusters (excluding noise)
```

```
# Cluster label -1 corresponds to noise points in DBSCAN
num_clusters = len(set(clusters)) - (1 if -1 in clusters else 0)

print(f"Number of clusters (excluding noise): {num_clusters}")

# Optional: Print the number of noise points
noise_points = list(clusters).count(-1)
print(f"Number of noise points: {noise_points}")

# Optional: Print the size of each cluster
unique_labels, counts = np.unique(clusters, return_counts=True)
for label, count in zip(unique_labels, counts):
    if label != -1:
        print(f"Cluster {label}: {count} samples")
```

```
Number of clusters (excluding noise): 0
Number of noise points: 178
```

Q37. Generate synthetic data with make_blobs and apply KMeans. Then plot the cluster centers on the top of the data points

In [22]:
```
# prompt: Generate synthetic data with make_blobs and apply KMeans. Then plot th

# Generate synthetic data with make_blobs
n_samples = 300
n_features = 2
n_centers = 3 # Let's use 3 centers for this example
X, y = make_blobs(n_samples=n_samples, n_features=n_features, centers=n_centers,

# Apply K-means clustering
kmeans = KMeans(n_clusters=n_centers, random_state=42, n_init=10)
kmeans.fit(X)
y_kmeans = kmeans.predict(X)

# Plot the data points and the cluster centers
plt.figure(figsize=(8, 6))
plt.scatter(X[:, 0], X[:, 1], c=y_kmeans, s=50, cmap='viridis', label='Data Poin

# Plot the cluster centers
centers = kmeans.cluster_centers_
plt.scatter(centers[:, 0], centers[:, 1], c='red', s=200, alpha=0.75, marker='X'

plt.title('K-Means Clustering with Cluster Centers (make_blobs)')
plt.xlabel('Feature 1')
plt.ylabel('Feature 2')
plt.legend()
plt.grid(True)
plt.show()
```



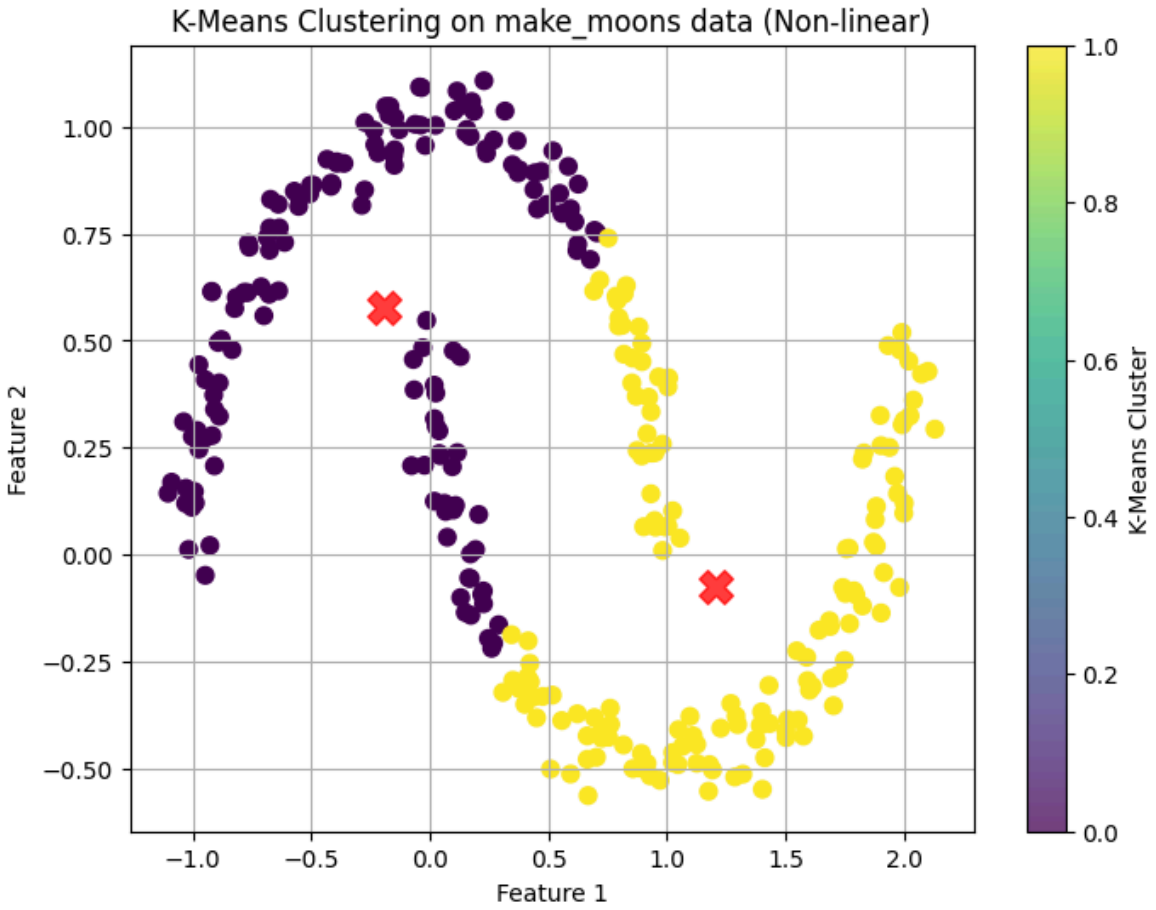
Q38. Load the Iris dataset, cluster with DBSCAN, and point how many samples were identified as noise

In [23]:
```
# prompt: Load the Iris dataset, cluster with DBSCAN, and point how many samples

# Load the Iris dataset
iris = load_iris()
X = iris.data

# Apply DBSCAN to the Iris dataset
# Adjust eps and min_samples as needed for the Iris data
# These parameters can significantly influence the results
dbscan = DBSCAN(eps=0.5, min_samples=5)
clusters = dbscan.fit_predict(X)

# Count the number of samples identified as noise
# In DBSCAN, noise points are labeled as -1
noise_samples_count = list(clusters).count(-1)

print(f"Number of samples identified as noise: {noise_samples_count}")

# Optional: Print the size of each identified cluster
unique_labels, counts = np.unique(clusters, return_counts=True)
print("\nCluster sizes (excluding noise):")
for label, count in zip(unique_labels, counts):
    if label != -1:
        print(f"Cluster {label}: {count} samples")
```

```
Number of samples identified as noise: 17

Cluster sizes (excluding noise):
Cluster 0: 49 samples
Cluster 1: 84 samples
```

Q39. Generate synthetic non-linearly separable data using make_moons, apply K-means, and visualize the clustering result

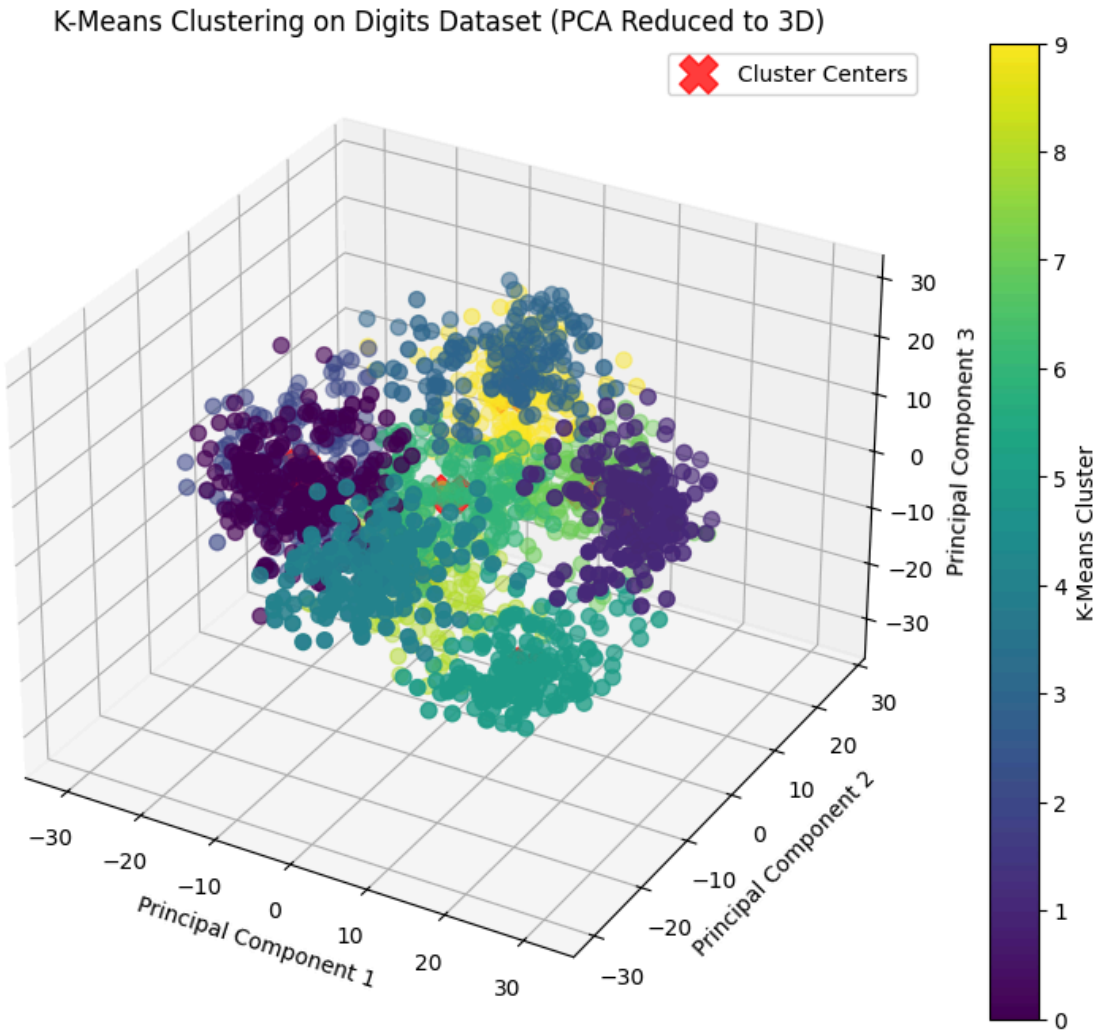
In [24]:
```
# prompt: Generate synthetic non-linearly separable data using make_moons, apply

# Generate synthetic non-linearly separable data using make_moons
n_samples = 300
X, y_true = make_moons(n_samples=n_samples, noise=0.05, random_state=42)

# Apply K-means clustering
# K-means will likely struggle with this non-linearly separable data
# Let's try with 2 clusters (assuming we know there are two underlying 'moons')
n_clusters = 2
kmeans = KMeans(n_clusters=n_clusters, random_state=42, n_init=10)
kmeans.fit(X)
y_kmeans = kmeans.predict(X)

# Visualize the clustering result
plt.figure(figsize=(8, 6))
plt.scatter(X[:, 0], X[:, 1], c=y_kmeans, s=50, cmap='viridis')

# Plot the cluster centers
centers = kmeans.cluster_centers_
plt.scatter(centers[:, 0], centers[:, 1], c='red', s=200, alpha=0.75, marker='X'

plt.title('K-Means Clustering on make_moons data (Non-linear)')
plt.xlabel('Feature 1')
plt.ylabel('Feature 2')
plt.colorbar(label='K-Means Cluster')
plt.grid(True)
plt.show()
```



Q40. Load the Digits dataset, apply PCA to reduce to 3 components, then use Kmeans and visualize with a 3D scatter plot

In [25]:
```
# prompt: Load the Digits dataset, apply PCA to reduce to 3 components, then use

from mpl_toolkits.mplot3d import Axes3D

# Load the Digits dataset
digits = load_digits()
X = digits.data
y_true = digits.target # True labels (digits 0-9)

# Apply PCA to reduce to 3 components
n_components = 3
pca = PCA(n_components=n_components)
X_pca = pca.fit_transform(X)

# Apply K-means clustering (assuming 10 clusters for digits 0-9)
n_clusters = 10
kmeans = KMeans(n_clusters=n_clusters, random_state=42, n_init=10)
kmeans.fit(X_pca)
y_kmeans = kmeans.predict(X_pca)

# Visualize with a 3D scatter plot
fig = plt.figure(figsize=(10, 8))
ax = fig.add_subplot(111, projection='3d')

# Scatter plot of data points
```

```
scatter = ax.scatter(X_pca[:, 0], X_pca[:, 1], X_pca[:, 2], c=y_kmeans, cmap='vi

# Plot the cluster centers (optional)
centers = kmeans.cluster_centers_
ax.scatter(centers[:, 0], centers[:, 1], centers[:, 2], c='red', s=300, alpha=0.

ax.set_title('K-Means Clustering on Digits Dataset (PCA Reduced to 3D)')
ax.set_xlabel('Principal Component 1')
ax.set_ylabel('Principal Component 2')
ax.set_zlabel('Principal Component 3')
fig.colorbar(scatter, label='K-Means Cluster')
ax.legend()

plt.show()
```

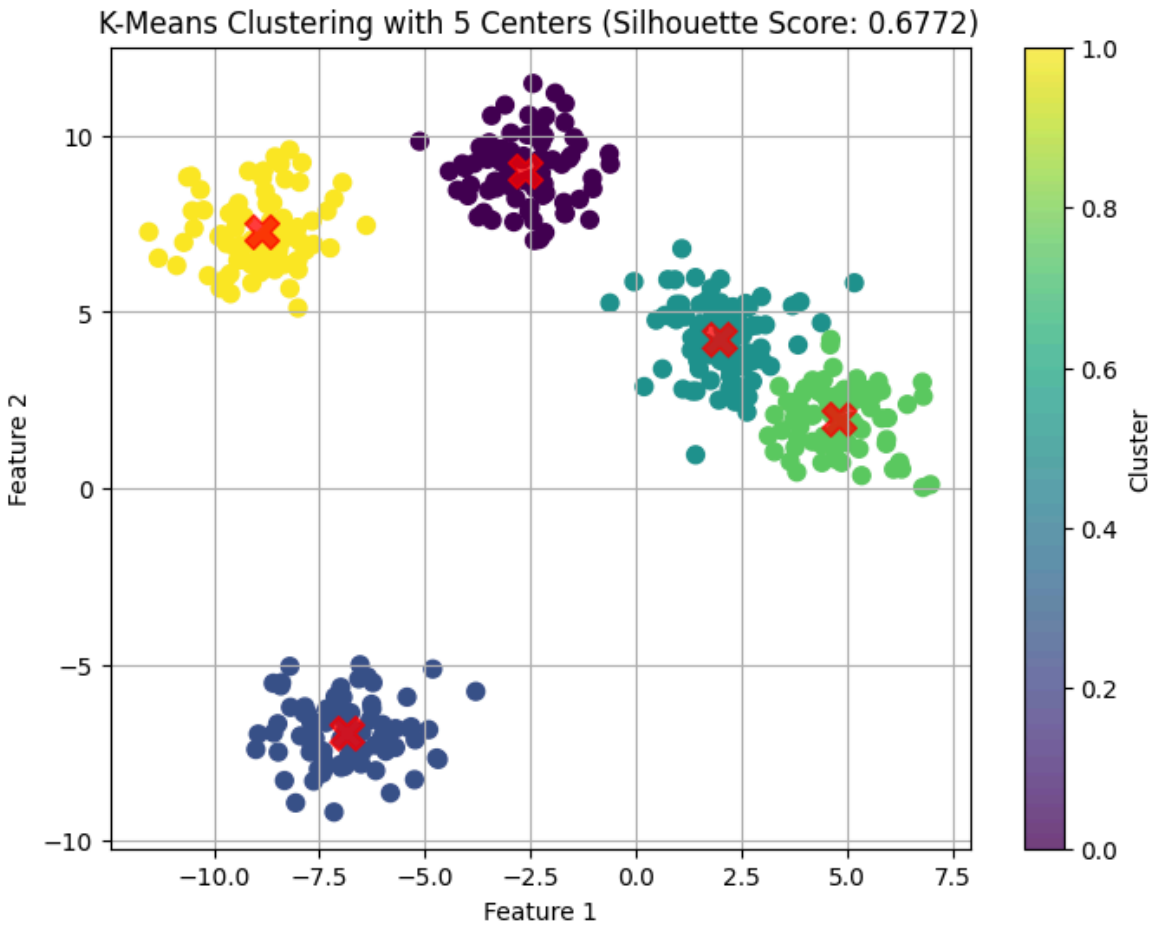


Q41. Generate synthetic blobs with 5 centers and apply Kmeans. Then use Silhoutte_score to evaluate the clustering

In [26]:
```
# prompt: Generate synthetic blobs with 5 centers and apply Kmeans. Then use Sil

# Generate synthetic blobs with 5 centers
n_samples = 400
n_features = 2
n_centers = 5
X, y_true = make_blobs(n_samples=n_samples, n_features=n_features, centers=n_cen
```

```
# Apply K-means clustering with 5 centers
kmeans = KMeans(n_clusters=n_centers, random_state=42, n_init=10)
kmeans.fit(X)
y_kmeans = kmeans.predict(X)

# Evaluate the clustering using Silhouette Score
silhouette_avg = silhouette_score(X, y_kmeans)

print(f"Silhouette Score for K-Means with {n_centers} clusters: {silhouette_avg:

# Optional: Visualize the clustering result
plt.figure(figsize=(8, 6))
plt.scatter(X[:, 0], X[:, 1], c=y_kmeans, s=50, cmap='viridis')

# Plot the cluster centers
centers = kmeans.cluster_centers_
plt.scatter(centers[:, 0], centers[:, 1], c='red', s=200, alpha=0.75, marker='X'

plt.title(f'K-Means Clustering with {n_centers} Centers (Silhouette Score: {silh
plt.xlabel('Feature 1')
plt.ylabel('Feature 2')
plt.colorbar(label='Cluster')
plt.grid(True)
plt.show()
```

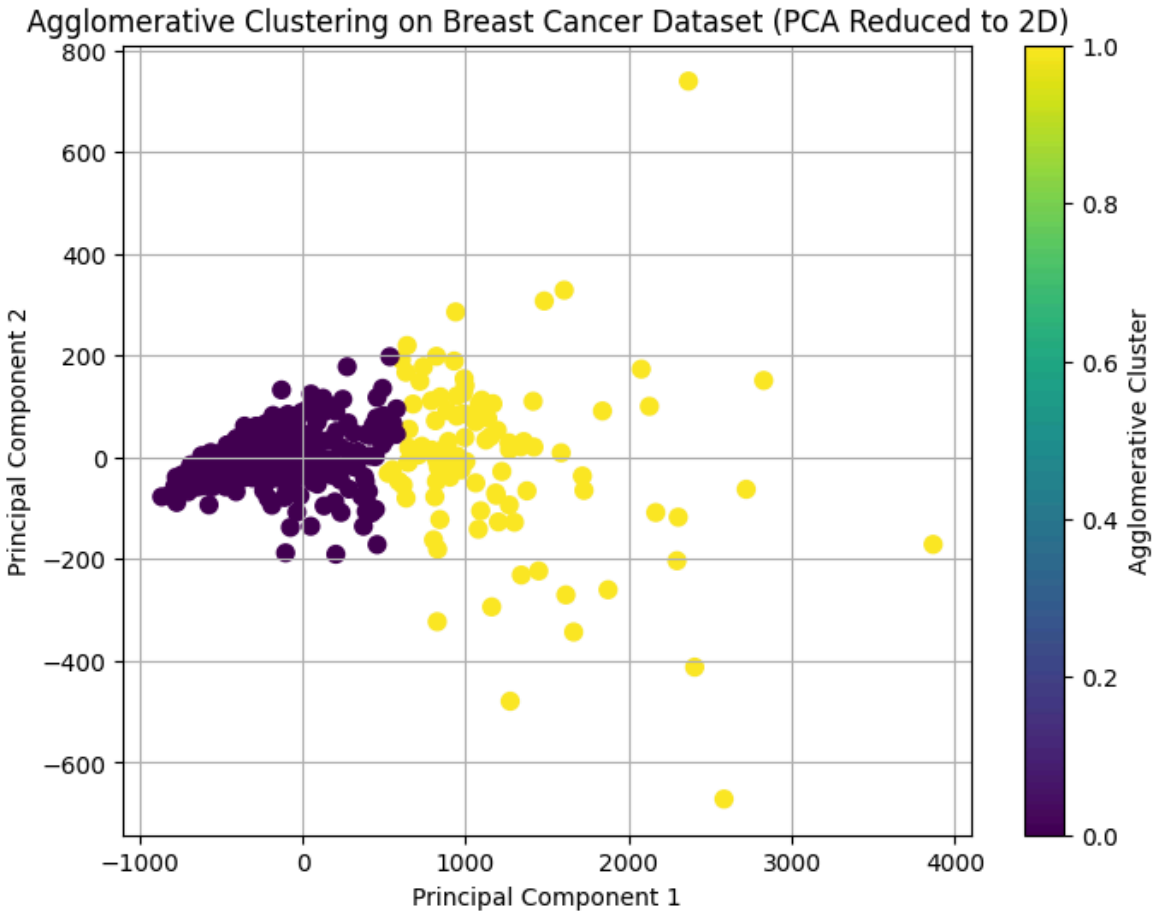
```
Silhouette Score for K-Means with 5 clusters: 0.6772
```



Q42. Load the Breast Cancer dataset, reduce dimensionality using PCA, and apply

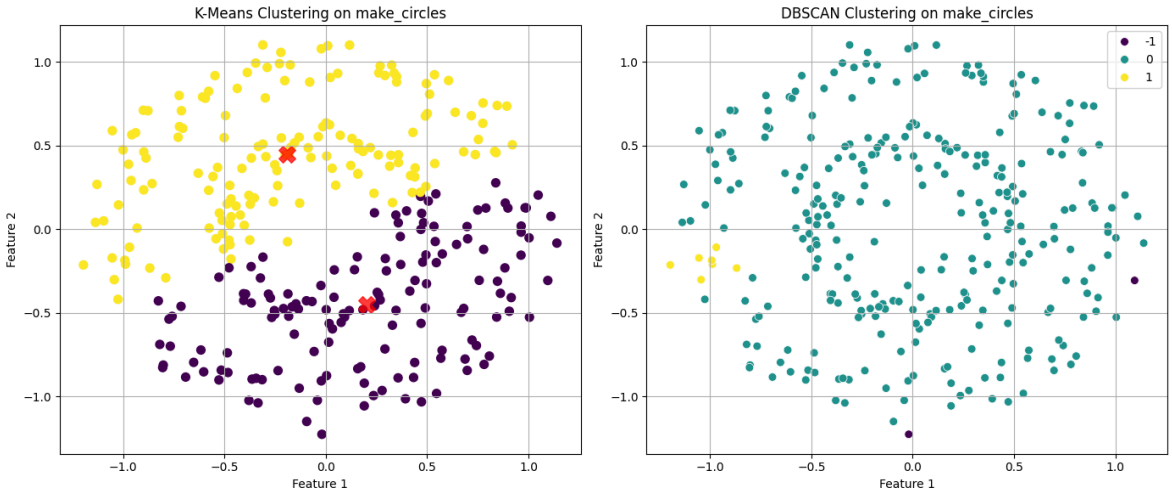
Agglomerative Clustering. visualize in 2D

In [28]:
```
# prompt: Load the Breast Cancer dataset, reduce dimensionality using PCA, and a

# Load the Breast Cancer dataset
breast_cancer = load_breast_cancer()
X = breast_cancer.data

# Reduce dimensionality using PCA to 2 components for visualization
n_components = 2
pca = PCA(n_components=n_components)
X_pca = pca.fit_transform(X)

# Apply Agglomerative Clustering
# We can choose a number of clusters, e.g., 2 (benign/malignant)
n_clusters = 2
agg_clustering = AgglomerativeClustering(n_clusters=n_clusters)
agg_labels = agg_clustering.fit_predict(X_pca)

# Visualize the results in 2D
plt.figure(figsize=(8, 6))
plt.scatter(X_pca[:, 0], X_pca[:, 1], c=agg_labels, s=50, cmap='viridis')

plt.title('Agglomerative Clustering on Breast Cancer Dataset (PCA Reduced to 2D)')
plt.xlabel('Principal Component 1')
plt.ylabel('Principal Component 2')
plt.colorbar(label='Agglomerative Cluster')
plt.grid(True)
plt.show()
```



Q43. Generate noisy circular data using make_circles and visualize clustering results from KMeans and DBSCAN side-by-side

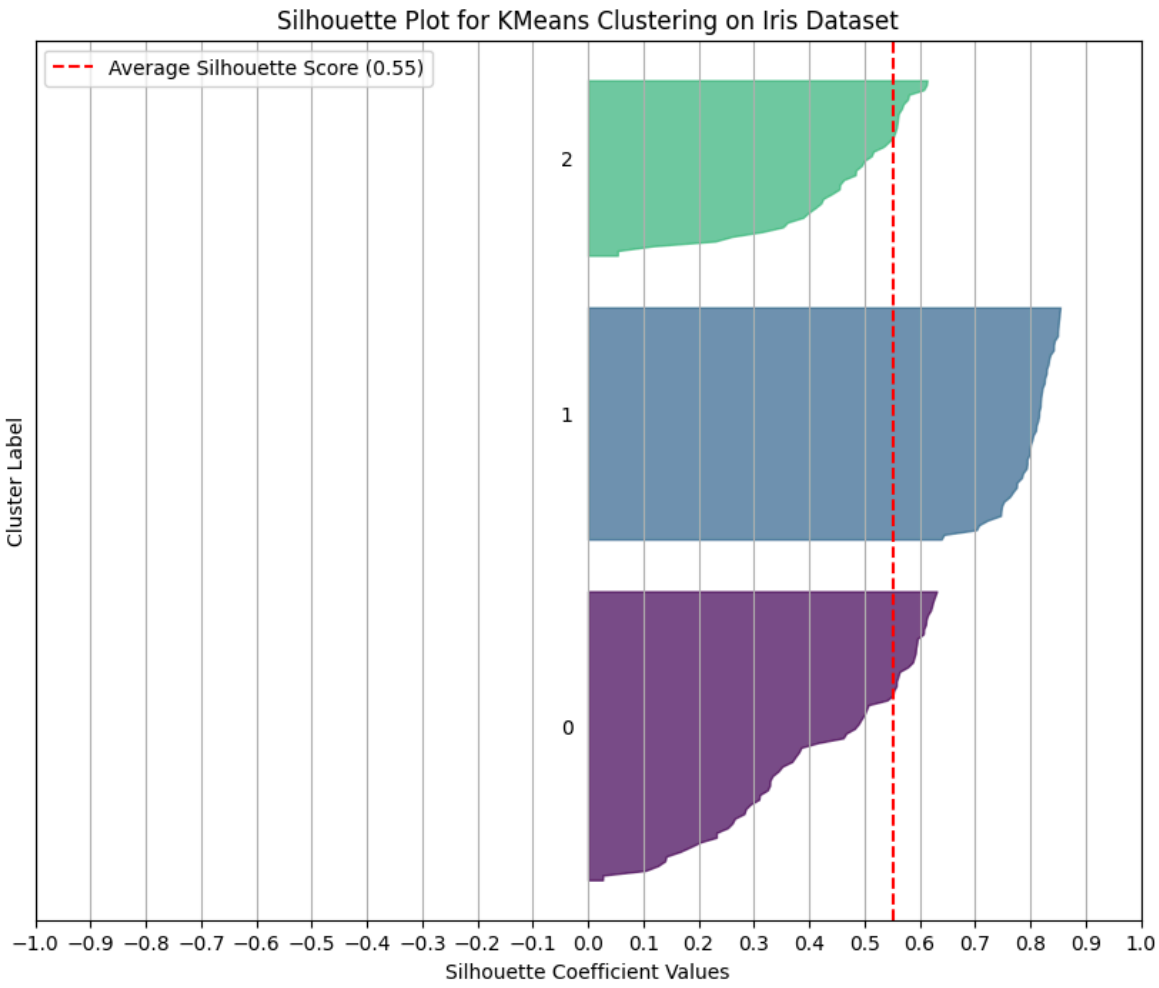
In [29]:
```
# prompt: Generate noisy circular data using make_circles and visualize clusteri

# Generate noisy circular data
n_samples = 300
X, y_true = make_circles(n_samples=n_samples, noise=0.1, factor=0.5, random_stat

# Apply K-means clustering
# K-means often struggles with circular data as it assumes spherical clusters
kmeans = KMeans(n_clusters=2, random_state=42, n_init=10) # Assuming 2 circles
kmeans.fit(X)
y_kmeans = kmeans.predict(X)

# Apply DBSCAN
# DBSCAN is typically better at finding density-based shapes like circles
# These parameters might need tuning
dbscan = DBSCAN(eps=0.2, min_samples=5)
y_dbscan = dbscan.fit_predict(X)

# Visualize the results side-by-side
fig, axes = plt.subplots(1, 2, figsize=(14, 6))

# K-Means Plot
axes[0].scatter(X[:, 0], X[:, 1], c=y_kmeans, s=50, cmap='viridis')
# Optional: Plot K-means centers
centers = kmeans.cluster_centers_
axes[0].scatter(centers[:, 0], centers[:, 1], c='red', s=200, alpha=0.75, marker
axes[0].set_title('K-Means Clustering on make_circles')
axes[0].set_xlabel('Feature 1')
axes[0].set_ylabel('Feature 2')
axes[0].grid(True)

# DBSCAN Plot
# Use seaborn for better color mapping, especially for -1 (outliers)
sns.scatterplot(x=X[:, 0], y=X[:, 1], hue=y_dbscan, palette='viridis', s=50, ax=
axes[1].set_title('DBSCAN Clustering on make_circles')
axes[1].set_xlabel('Feature 1')
axes[1].set_ylabel('Feature 2')
axes[1].grid(True)

plt.tight_layout()
plt.show()
```



Q44. Load the Iris dataset and plot the silhouette Coefficient for each sample after KMeans clustering.

In [32]:
```
# prompt: Load the Iris dataset and plot the silhouette Coefficient for each sam

from sklearn.datasets import load_iris
from sklearn.cluster import KMeans
from sklearn.metrics import silhouette_score, silhouette_samples
import matplotlib.pyplot as plt
import numpy as np

# Load the Iris dataset
iris = load_iris()
X = iris.data
y_true = iris.target # True labels

# Apply K-means clustering
# We'll use 3 clusters as per the known structure of the Iris dataset
n_clusters = 3
kmeans = KMeans(n_clusters=n_clusters, random_state=42, n_init=10)
kmeans.fit(X)
labels = kmeans.predict(X)

# Calculate the silhouette coefficient for each sample
# The silhouette_samples function gives the score for each data point
silhouette_individual = silhouette_samples(X, labels)

# Plot the silhouette coefficient for each sample
plt.figure(figsize=(10, 8))

y_lower = 10 # Starting point for the first cluster's plot
for i in range(n_clusters):
    # Aggregate the silhouette scores for samples belonging to cluster i, and so
    ith_cluster_silhouette_values = silhouette_individual[labels == i]
    ith_cluster_silhouette_values.sort()

    size_cluster_i = ith_cluster_silhouette_values.shape[0]
    y_upper = y_lower + size_cluster_i

    color = plt.cm.viridis(float(i) / n_clusters)
    plt.fill_betweenx(np.arange(y_lower, y_upper),
```

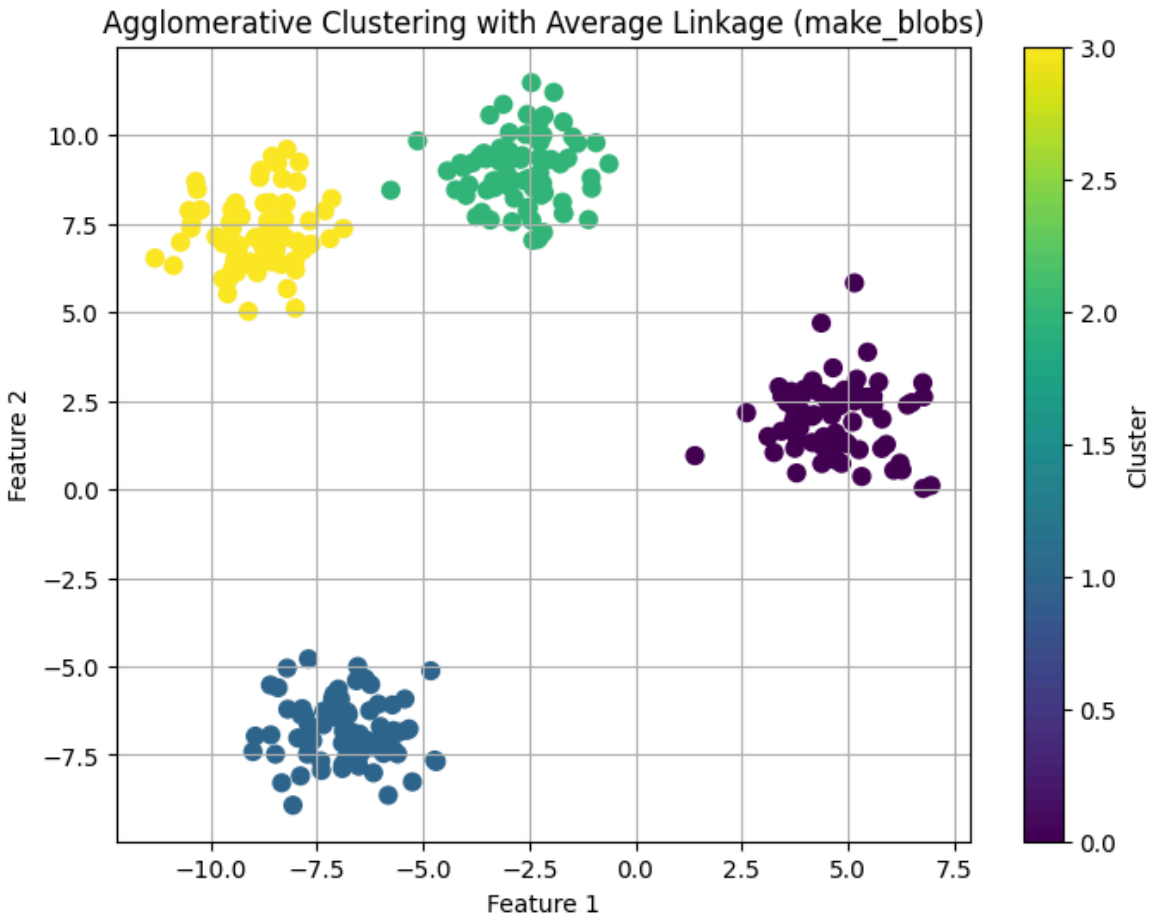
```
                    0, ith_cluster_silhouette_values,
                    facecolor=color, edgecolor=color, alpha=0.7)

    # Label the silhouette plots with their cluster numbers at the middle
    plt.text(-0.05, y_lower + 0.5 * size_cluster_i, str(i))

    # Compute the new y_lower for the next plot
    y_lower = y_upper + 10 # Add some spacing between clusters

# Plot the average silhouette score
silhouette_avg = silhouette_score(X, labels)
plt.axvline(x=silhouette_avg, color="red", linestyle="--", label=f'Average Silho

plt.yticks([]) # Clear the y-axis labels / ticks
plt.xticks(np.arange(-1, 1.1, 0.1))
plt.xlabel("Silhouette Coefficient Values")
plt.ylabel("Cluster Label")
plt.title("Silhouette Plot for KMeans Clustering on Iris Dataset")
plt.legend()
plt.grid(True)
plt.show()

print(f"Average Silhouette Score for KMeans on Iris dataset: {silhouette_avg:.4f
```




```
Average Silhouette Score for KMeans on Iris dataset: 0.5528
```

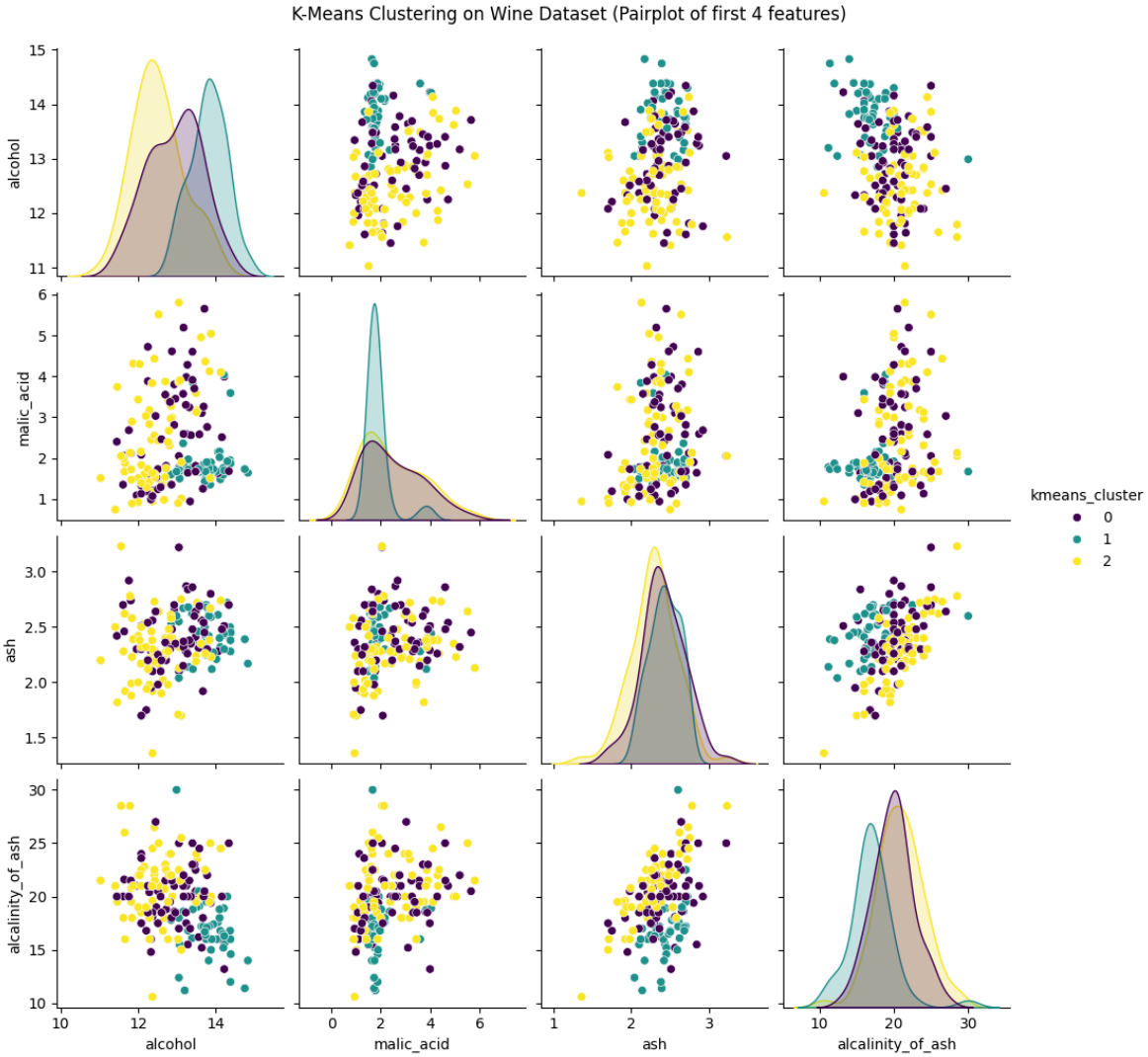
Q45. Generate synthetic data using make_blobs and apply Agglomerative Clustering with 'average' linkage

In [33]:
```
# prompt: Generate synthetic data using make_blobs and apply Agglomerative Clust

# Generate synthetic data using make_blobs
n_samples = 300
n_features = 2
n_centers = 4
X, y_true = make_blobs(n_samples=n_samples, n_features=n_features, centers=n_cen

# Apply Agglomerative Clustering with average linkage
n_clusters = 4 # Assuming the same number of clusters as in make_blobs
agg_clustering = AgglomerativeClustering(n_clusters=n_clusters, linkage='average
agg_labels = agg_clustering.fit_predict(X)

# Plot the result
plt.figure(figsize=(8, 6))
plt.scatter(X[:, 0], X[:, 1], c=agg_labels, s=50, cmap='viridis')

plt.title('Agglomerative Clustering with Average Linkage (make_blobs)')
plt.xlabel('Feature 1')
plt.ylabel('Feature 2')
plt.colorbar(label='Cluster')
plt.grid(True)
plt.show()
```



Q46. Load the Wine dataset, apply KMeans, and visualize the cluster assignments in a Seaborn pairplot (first 4 features)

In [37]:
```
# prompt: Load the Wine dataset, apply KMeans, and visualize the cluster assignm

# Load the Wine dataset
wine = load_wine()
X = wine.data
y_true = wine.target # True labels (for comparison, though not used in clusterin
feature_names = wine.feature_names

# Apply K-means clustering
# We know there are 3 classes in the wine dataset, so let's use 3 clusters
n_clusters = 3
kmeans = KMeans(n_clusters=n_clusters, random_state=42, n_init=10)
kmeans.fit(X)
y_kmeans = kmeans.predict(X)

# Add the cluster assignments to a pandas DataFrame for Seaborn pairplot
# Select the first 4 features as requested
import pandas as pd
df = pd.DataFrame(X[:, :4], columns=feature_names[:4])
df['kmeans_cluster'] = y_kmeans

# Visualize the cluster assignments using Seaborn pairplot
# Use the 'hue' parameter to color points by their KMeans cluster assignment
import seaborn as sns
import matplotlib.pyplot as plt
sns.pairplot(df, hue='kmeans_cluster', palette='viridis', diag_kind='kde')
plt.suptitle('K-Means Clustering on Wine Dataset (Pairplot of first 4 features)'
plt.show()
```

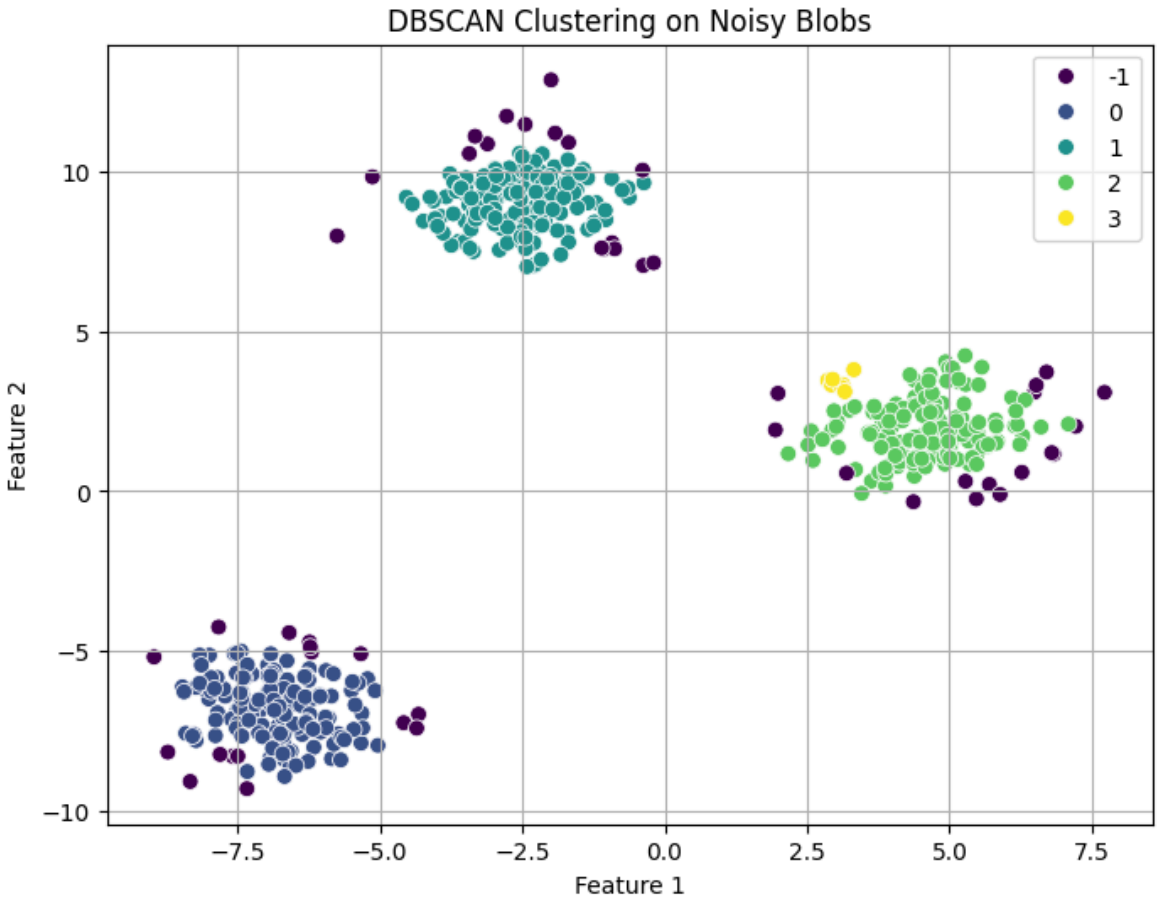


Q47. Generate noisy blobs using make_blobs and use DBSCAN to indentify both clusters and noise points, Print the count

In [38]:
```
# prompt: Generate noisy blobs using make_blobs and use DBSCAN to indentify both

# Generate noisy blobs
n_samples = 500
n_features = 2
n_centers = 3
# Increase cluster_std or add noise parameter if make_blobs supports it directly
# Let's use make_blobs with some spread, and DBSCAN's noise identification handl
X, y_true = make_blobs(n_samples=n_samples, n_features=n_features, centers=n_cen
                       cluster_std=1.0, random_state=42) # Increased std for pot

# Apply DBSCAN to identify clusters and noise points
# Adjust eps and min_samples based on the data density and noise level
# These parameters are crucial for DBSCAN performance
dbscan = DBSCAN(eps=0.5, min_samples=5)
clusters = dbscan.fit_predict(X)

# Count the number of clusters (excluding noise) and noise points
# Cluster label -1 corresponds to noise points
unique_labels, counts = np.unique(clusters, return_counts=True)
```

```
num_clusters = len(set(clusters)) - (1 if -1 in clusters else 0)
print(f"Number of identified clusters: {num_clusters}")

# Print the count of noise points
noise_points_count = dict(zip(unique_labels, counts)).get(-1, 0)
print(f"Number of noise points identified by DBSCAN: {noise_points_count}")

# Print the size of each identified cluster
print("\nCluster sizes (excluding noise):")
for label, count in zip(unique_labels, counts):
    if label != -1:
        print(f"Cluster {label}: {count} samples")

# Optional: Visualize the results to see the clusters and noise
plt.figure(figsize=(8, 6))
# Use seaborn for better color mapping, especially for -1 (outliers)
sns.scatterplot(x=X[:, 0], y=X[:, 1], hue=clusters, palette='viridis', s=50, leg

plt.title('DBSCAN Clustering on Noisy Blobs')
plt.xlabel('Feature 1')
plt.ylabel('Feature 2')
plt.grid(True)
plt.show()
```

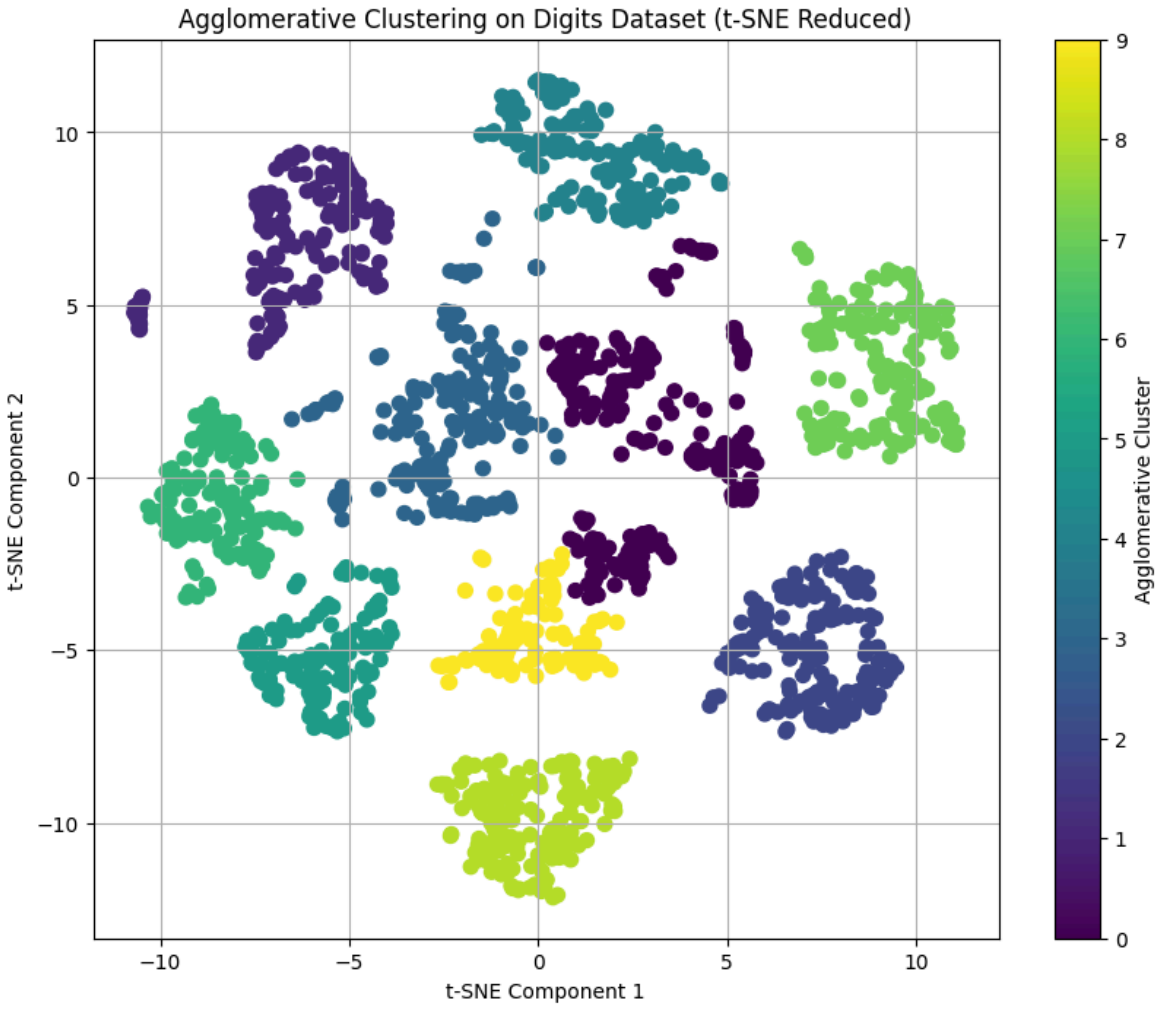
```
Number of identified clusters: 4
Number of noise points identified by DBSCAN: 49

Cluster sizes (excluding noise):
Cluster 0: 150 samples
Cluster 1: 150 samples
Cluster 2: 144 samples
Cluster 3: 7 samples
```



Q48. Load the Digits dataset, reduce dimensions using t-SNE, then apply Agglomerative Clustering and plot the clusters.

In [39]:
```
# prompt: Load the Digits dataset, reduce dimensions using t-SNE, then apply Agg

# Load the Digits dataset
digits = load_digits()
X = digits.data
y_true = digits.target # True labels (digits 0-9)

# Reduce dimensions to 2D using t-SNE
# t-SNE is computationally expensive, using a subset can be faster for initial t
# Consider adjusting perplexity and n_iter based on dataset size and complexity
tsne = TSNE(n_components=2, random_state=42, perplexity=30, n_iter=300)
X_tsne = tsne.fit_transform(X)

# Apply Agglomerative Clustering on the t-SNE reduced data
# We expect 10 clusters (for digits 0-9)
n_clusters = 10
agg_clustering = AgglomerativeClustering(n_clusters=n_clusters)
agg_labels = agg_clustering.fit_predict(X_tsne)

# Plot the clusters
plt.figure(figsize=(10, 8))
plt.scatter(X_tsne[:, 0], X_tsne[:, 1], c=agg_labels, s=50, cmap='viridis')

plt.title('Agglomerative Clustering on Digits Dataset (t-SNE Reduced)')
plt.xlabel('t-SNE Component 1')
plt.ylabel('t-SNE Component 2')
plt.colorbar(label='Agglomerative Cluster')
plt.grid(True)
plt.show()
```

```
/usr/local/lib/python3.11/dist-packages/sklearn/manifold/_t_sne.py:1164: FutureWa
rning: 'n_iter' was renamed to 'max_iter' in version 1.5 and will be removed in
1.7.
  warnings.warn(
```


Agglomerative Clustering on Digits Dataset (t-SNE Reduced)
t-SNE Component 1
t-SNE Component 2
Agglomerative Cluster
10
5
0
−5
−10
−10
−5
0
5
10
9
8
7
6
5
4
3
2
1
0