PROJECT 4

## K-Means Results:

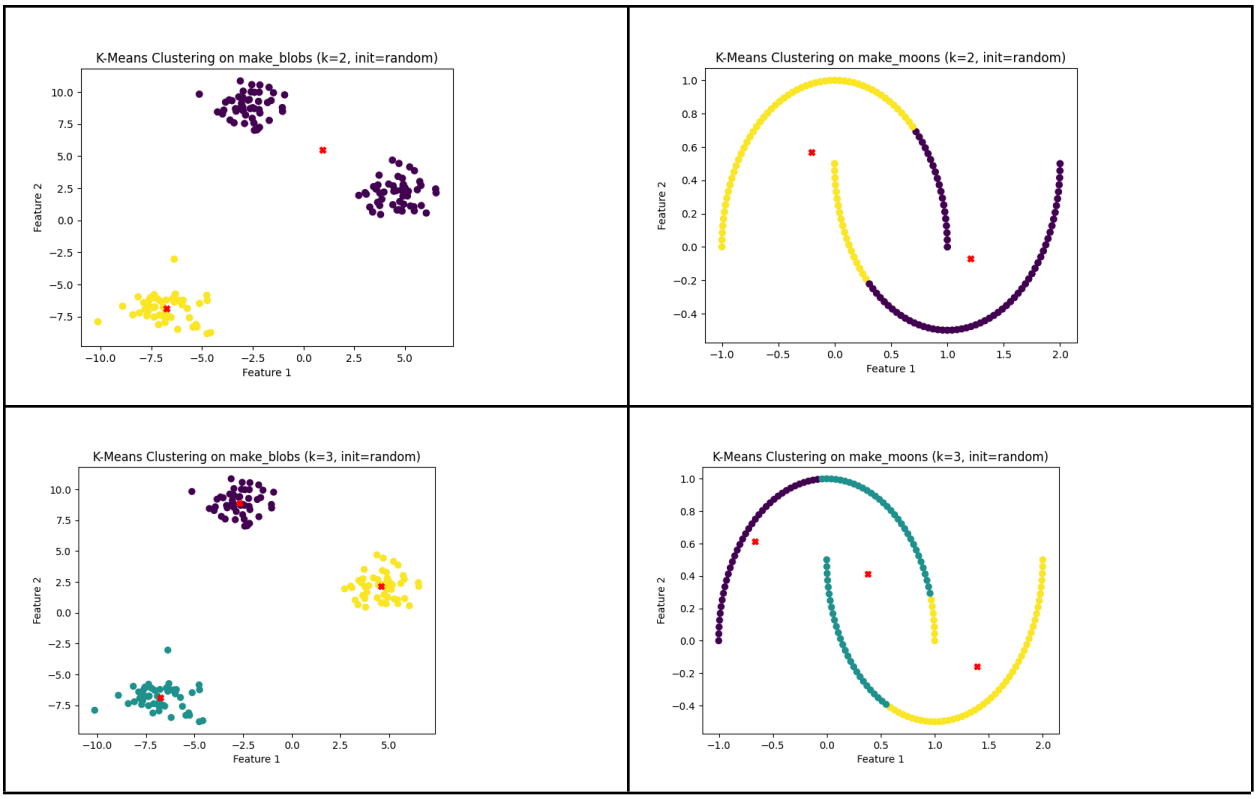
📊 Table 1 — K-Means Silhouette Scores

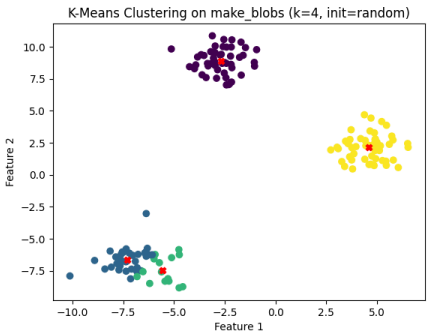
| k | make_blobs | make_moons |
|---|---|---|
| 2 | 0.7041664296644728 | 0.4876566217540581 |
| 3 | 0.844795031920084 | 0.42243336354986594 |
| 4 | 0.6678174026762854 | 0.45336863065619637 |
| 5 | 0.49683720657816316 | 0.4827790537349756 |

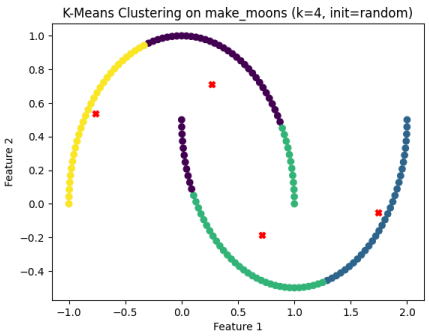
📊 Table 2 — K-Means++ Silhouette Scores

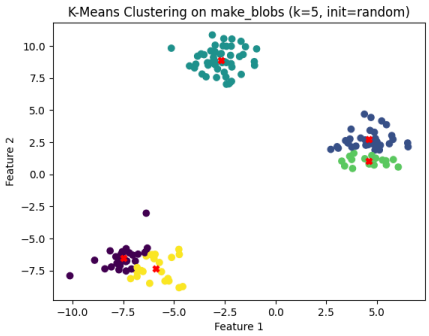
| k | make_blobs | make_moons |
|---|---|---|
| 2 | 0.7041664296644728 | 0.4876566217540581 |
| 3 | 0.844795031920084 | 0.42243336354986594 |
| 4 | 0.6738274708886695 | 0.4542390539036048 |
| 5 | 0.6826643042015322 | 0.47288480821688844 |

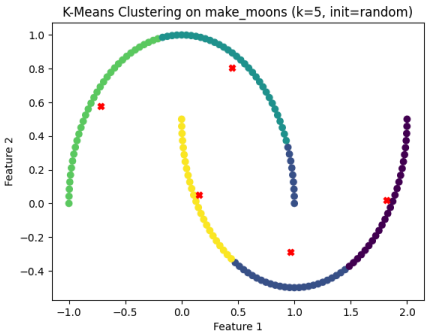
## K-Means(Random) Data Visualization:



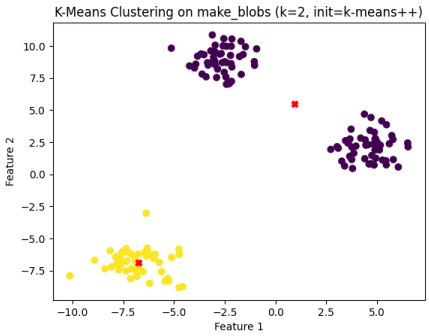


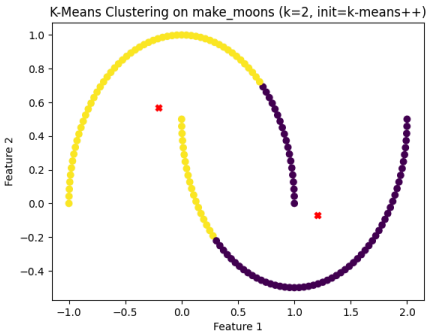



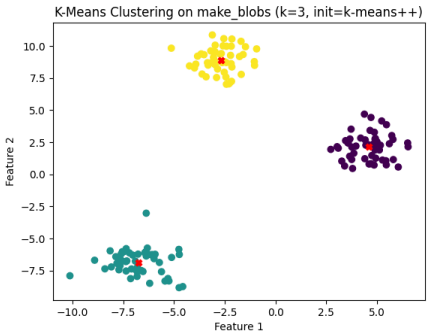



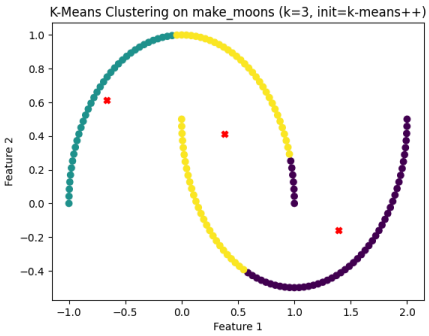



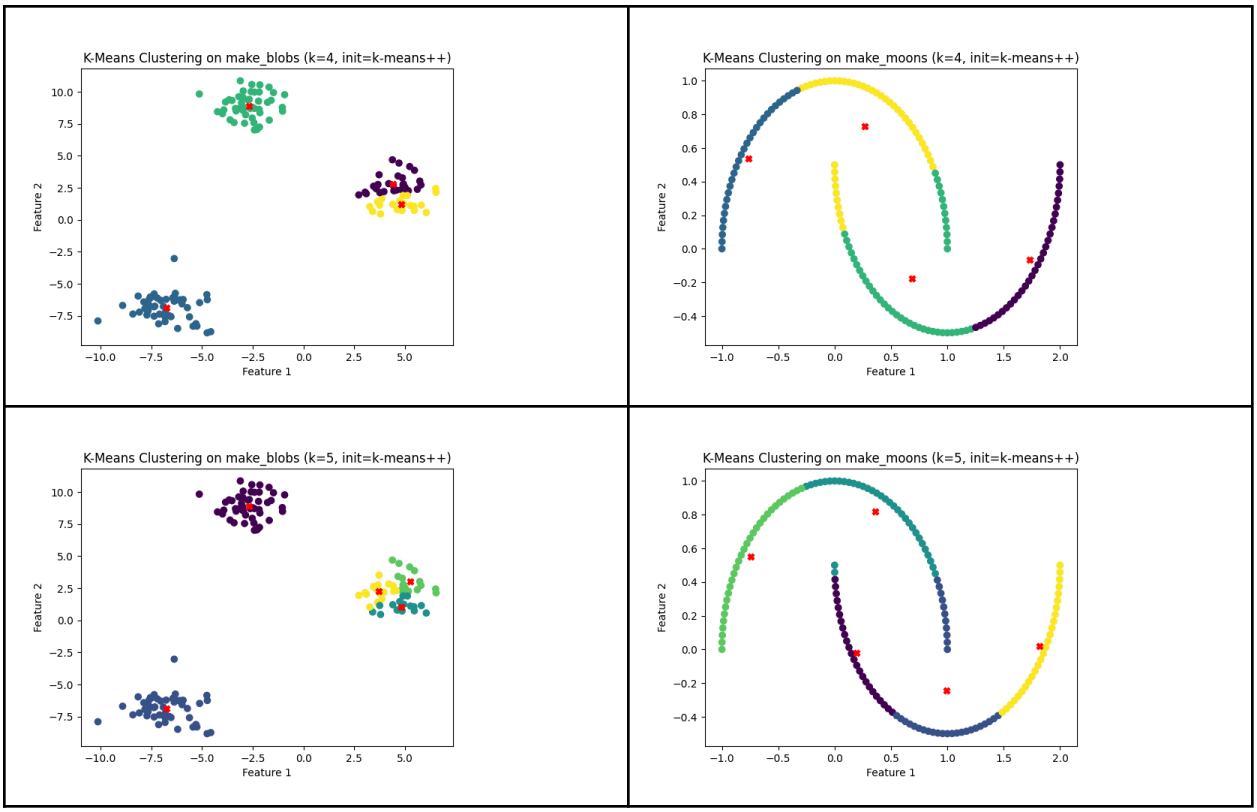

## K-Means(k-means++) Data Visualization:






## Compare the effect of different initializations

As we can see from the results, both initializations get almost identical Silhouette scores. The reason for each dataset:

- make_blobs dataset: the data is well separated, making it easier to cluster. Since the dataset is simple, even random initialization usually places the centroids close to their true locations. So, K-means++ initial smart centroid assigning will not benefit unless we get super unlucky with the K-means random initial centroid assigning. So in the end, both initialization methods converge to a very similar solution. However, as we can see from our results, K-means++ outperformed K-means by a lot when k = 5 (0.682 to 0.496), the reason for this is that when the number of k increases the chance of choosing initial centriods close together with k-means increases too which results in bad performance, this is not the case with k-means++ since its smart initialization able to spread out the initial centroids as much as it can resulting in better clustering hence better performance.

- make_moons dataset: the reason both initializations perform equally badly is because of the nature of that dataset; the make_moons dataset was made by non-linearly separable/non-spherical clusters, so the K-means++ initial smart centroid assigning will also not benefit from this because the dataset itself is violates k-means assumptions, which assume circular decision boundaries, so since the k-means assumption is violated, the smart initial assignment of centroids with the k-means++ performed very

close to k-means random because smart initial centroid assignment can not overcome the k-means algorithm's limitations.

### Which dataset is better suited for K-Means, and why?

The make_blobs dataset is better suited for the k-means because it has a clearly separated and roughly spherical dataset where the k-means algorithm can easily cluster those data points. The make_moons dataset, on the other hand, is not a great dataset for k-means because the data follows a non-spherical moon-like shape, which is not convex. This causes both k-means and k-means++ to achieve relatively low scores because it violates the k-means assumption, which assumes circular decision boundaries, as we discussed in the previous question.

### How does the choice of k affect clustering quality?

As we can see from the Silhouette scores on the make_blobs dataset, we get the highest score when k = 3. This is because there are three different clear clusters, and when we assign k = 3, we assign each centroid its own cluster, which produces clean, well-separated partitions, hence, we get a better score compared to other choices of k. For the k=2,4,5 choices, some of the centroids have to share or split clusters incorrectly, which reduces the cluster quality, so it gets relatively lower Silhouette scores compared to k = 3. On the make_moon dataset, the choice of k really does not help with the  Silhouette scores. This is because cluster k-means have shared spherical structure, and as we discussed with the previous questions make_moons dataset violates the k-means assumption, so changing the number of k really does not improve the Silhouette scores that much.

## GMM Results:

📊 Table 1 — GMM (covariance_type = 'full') Results

make_blobs

| k | Silhouette Score | Avg Log-Likelihood | BIC |
|---|---|---|---|
| 2 | 0.7041664296644728 | -4.552524014119189 | 1420.8741924708156 |
| 3 | 0.844795031920084 | -3.8594484904532322 | 1243.015347135606 |
| 4 | 0.6741970383894089 | -3.8485648820815896 | 1269.8140763886909 |
| 5 | 0.4717854536372703 | -3.8444929809957027 | 1298.6563178275023 |

make_moons

| k | Silhouette Score | Avg Log-Likelihood | BIC |
|---|---|---|---|
| 2 | 0.462266864460179 | -1.7004804423206357 | 565.2611209312495 |
| 3 | 0.36664159170394756 | -1.3027650038905187 | 476.010301166792 |
| 4 | 0.43558872509870333 | -1.0023513832545885 | 415.95002674059043 |
| 5 | 0.4735440459907238 | -0.7449571892937301 | 368.79558031691045 |

## 📊 Table 2 — GMM (covariance_type = 'diag') Results

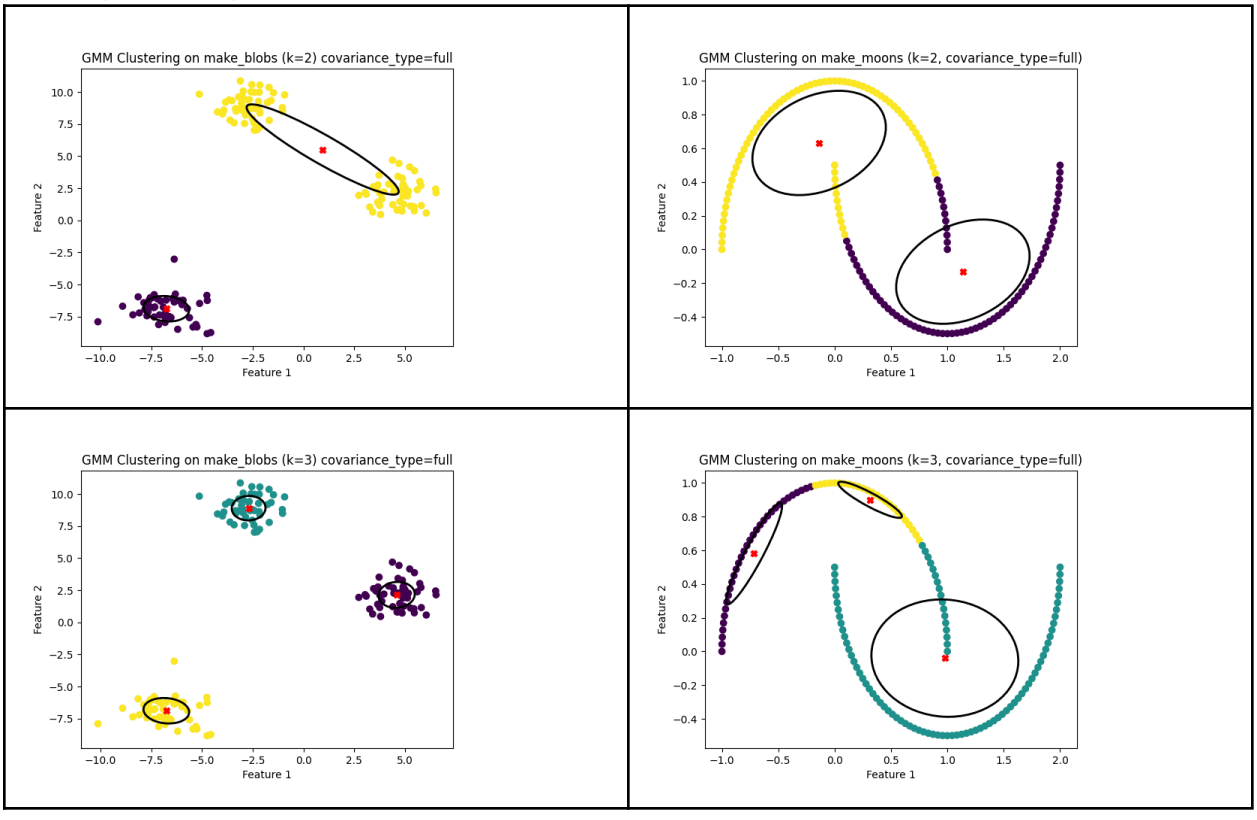
### make_blobs

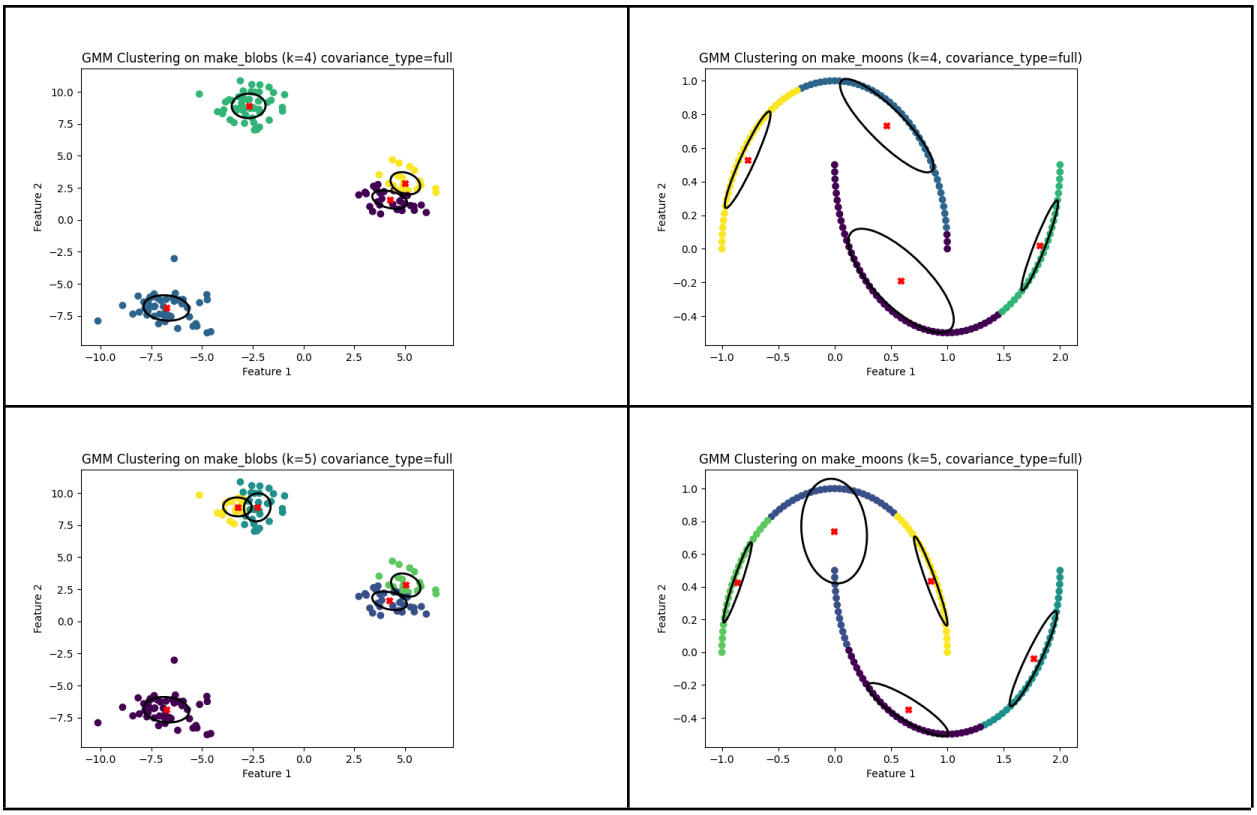
| k | Silhouette Score | Avg Log-Likelihood | BIC |
|---|---|---|---|
| 2 | 0.6942934115346856 | -5.212342694333686 | 1608.798525946972 |
| 3 | 0.844795031920084 | -3.862139750645074 | 1228.7908193108697 |
| 4 | 0.6630405305521229 | -3.848850708269765 | 1249.8572830687583 |
| 5 | 0.5230793833488356 | -3.799231336588244 | 1260.0246480347832 |

### make_moons

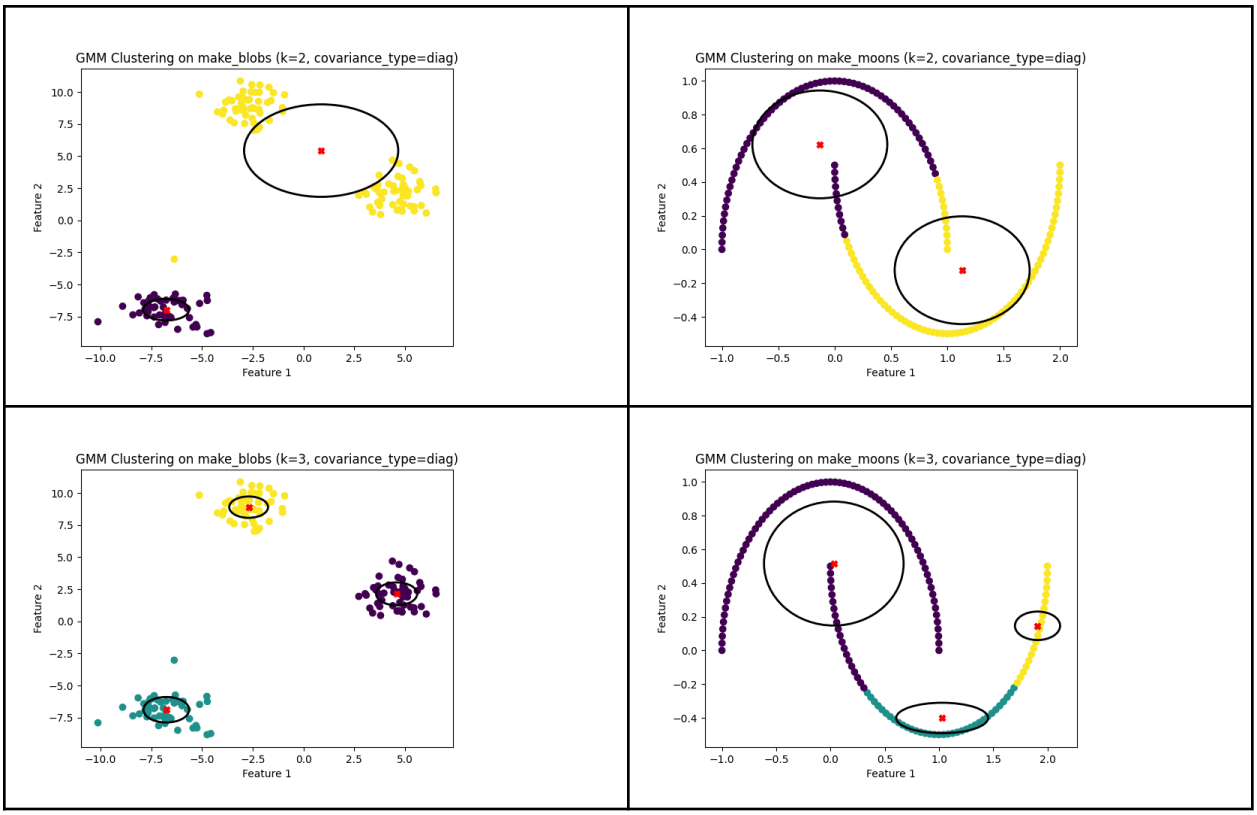
| k | Silhouette Score | Avg Log-Likelihood | BIC |
|---|---|---|---|
| 2 | 0.462266864460179 | -1.7404682613470428 | 567.2361960509792 |
| 3 | 0.3890902031782593 | -1.473285166878263 | 512.1344441808264 |
| 4 | 0.3123285474132567 | -1.3227961051610342 | 492.04090213613915 |
| 5 | 0.4202641486585265 | -1.110245124503161 | 453.32878440925845 |

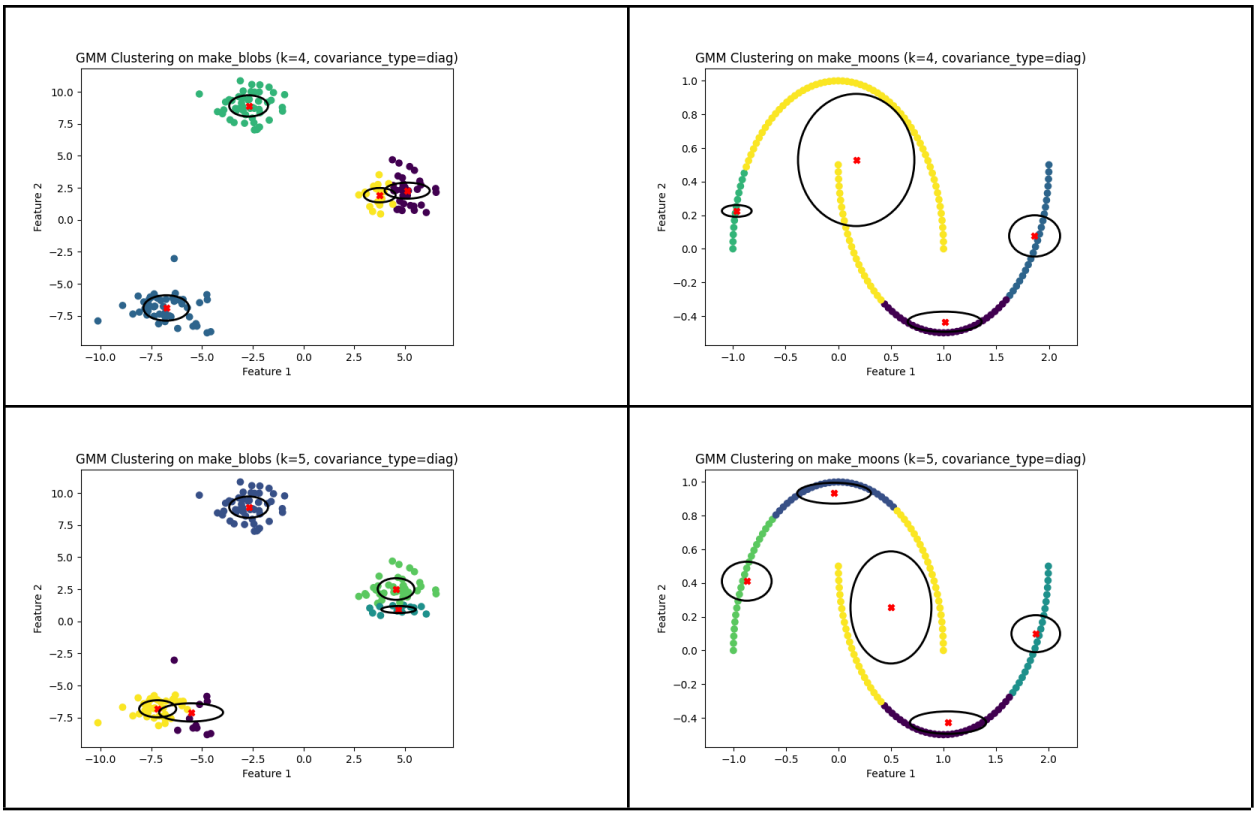
## GMM (ct = full) Data Visualization:






## GMM (ct = diag) Data Visualization:






## Analyze how GMMs handle elliptical clusters better than K-Means, especially on non-convex datasets.

The GMMs handle elliptical clusters better than K-Means. This is because GMMS captures the Gaussian distribution, where each distribution has its own variance and mean. This allows GMMs to be shaped more flexibly, in this case, an elliptical shape. So, using the Gaussian distribution, it can easily cluster the elliptical clusters. The K-means, on the other hand, does not have a Gaussian distribution, but rather assumes each cluster will be spherical and uses Euclidean distances to assign points to the clusters. So k-means has a fixed shape because of the assumption it has that every cluster has to be in spherical form. This is the reason GMMs handle elliptical clusters better than K-Means.

However, when the data is truly non-convex, such as in the make_moons dataset, this extra flexibility is still not enough, as we can see from our results tables, GMMs also struggled as much as the k-means algorithm struggled on the make_moons dataset. The reason both algorithms performed poorly was that the datapoints in the make_moons dataset are curving, which makes the true cluster shapes also curved, since the GMMs can not cluster datapoints that curve using ellipses, it does its best to get most out of them, but still was not enough, so in the end the both approaches resulted in a poor performance. Resulting in GMM performing slightly worse than k-means.

## In what scenarios does EM outperform K-Means?

There are several instances where the EM outperforms K-means. Let's give some of those instances as examples:

- The first instance is when the dataset has true clusters that are elliptical in shape. EM performs better because GMMs can model each cluster with a full covariance matrix. This makes EM generate clusters that are flexible, elliptical in shape, which will fit very close to the true cluster. K-means, on the other hand, is forced to spherical clusters, so it gets outperformed by EM.
- The second instance is when the datapoints are overlapping. EM does a soft clustering, which is very useful when the datapoints are overlapping, because with soft clustering, you do not need to assign a datapoint to a certain cluster directly, but you can assign a percentage to the datapoint, which represents how much that datapoint is part of a certain cluster. With k-means clustering, this is not the case; k-means uses hard clustering, so the datapoint has to belong to only one cluster, which results in lost information and leads to worse performance overall.
- The third instance is when the clusters are in different sizes, since EM does not assume clusters to be of equal size, unlike K-means. EM does a better job of clustering different-sized clusters using the variances captured in the Gaussians' covariance. This results in EM getting a better performance overall.

## How does the covariance type influence cluster shape and separation?

The covariance type is the key factor when it comes to the shape and separation of the clusters. With the full covariance, each of the clusters has its own covariance matrix, allowing it to capture correlations between features. With this, full covariances produce clusters that can stretch and rotate in any direction, which gives the model maximum flexibility to fit the true cluster shapes. With the diagonal covariance, on the other hand, each of the clusters has its own diagonal covariance matrix, allowing clusters to stretch along each axis but can not rotate like full covariance. This makes diagonal covariance less flexible compared to full covariance since it can not follow the true clusters that are angled. We can see the effects of different covariance types clearly if we look at the graphs included in this report. As we can see with full covariance, the clusters are following the direction of the datapoints(angled), while with the diagonal covariance, clusters remain fixed with angle = 0, limiting how well they can match the underlying structure.