CS 4375 Intro to ML
Project Report
Kevin Puga (kxp220070)

KMeans and KMeans++

Features

For KMeans, the features I implemented for my custom model are the following:

```
    def __init__(self, n_clusters=3, init='random', max_iter=300, tol=1e-4,
random_state=None, n_init=10):
        self.n_clusters = n_clusters
        self.init = init
        self.max_iter = max_iter
        self.tol = tol
        self.random_state = random_state
        self.n_init = n_init
        self.centroids = None
        self.labels_ = None
        self.cluster_centers_ = None
```

- n_clusters refer to the number of clusters (centroids) that will be made during testing
- 'init' determines the initialization strategy that will be used during testing (kmeans++ = distance aware method or random = selects randomly)
- max_iter refers to the maximum number of iterations per test and controls how long the algorithm will keep updating clusters before giving up on convergence
- tol refers to the tolerance threshold which will stop the algorithm early if the change in clusters is smaller than the threshold
- random_state is the random seed for reproducibility and ensures consisted cluster initialization across runs
- n_init refers to the number of times the KMeans algorithm is run with different cluster seeds where the best result across all runs is selected as the final model

These features where implemented because they support multiple initialization strategies, controlled randomness, multiple restarts to improve robustness, and tol for early stopping based on convergence

What was attempted

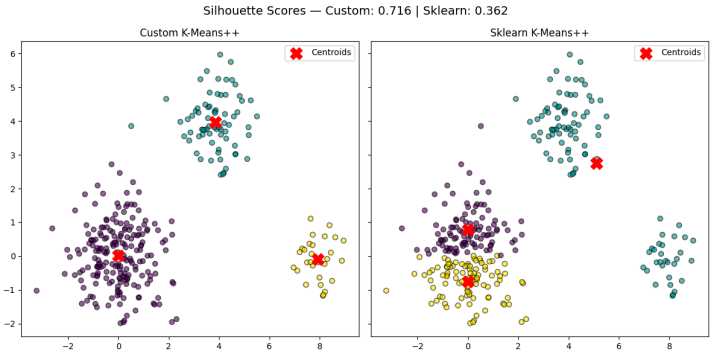
For KMeans and KMeans++, I attempted to recreate a clustering method that attempts to cluster similar data into clusters. This included implementing custom centroid initialization for 'random' and 'kmeans++', iterative assignment and update steps, and convergence checking with tolerance threshold.

What worked

When testing on different noise on *make_moons,* my model and sklearn performed identically for KMeans random and kmeans++ initialization while my custom minibatch performed slightly better. Likewise, with *make_circles*, both my model and sklearn performed similarly when testing on different value for noise with either implementation scoring slightly higher that other, while the same can be said about the when changing the factor value as well. When testing on unbalanced data sets using kmeans++ initialization, there was an instance where my model performed better than sklearn. This was the code I was testing for both:

```
kmeans_plus = CustomKMeans(n_clusters=3, init='k-means++', random_state=42, n_init=1)
```

This is basically running multiple runs until it finds the best plot. In one iteration my model found the best fit while sklearn didn't. I ran this multiple times while changing the n_init and they performed the same on the other tests.

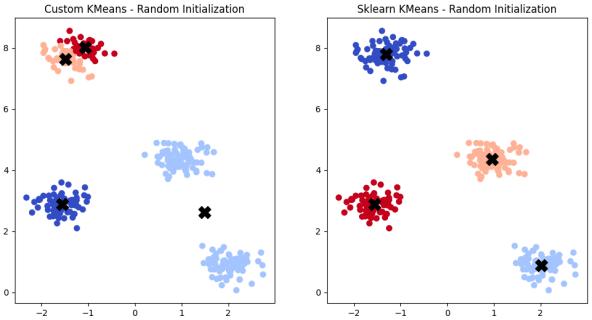


I also tested my custom model on an anisotropic dataset, the iris dataset, and the wine dataset. My model compared to those datasets performed the same as they had similar or equal silhouette scores.

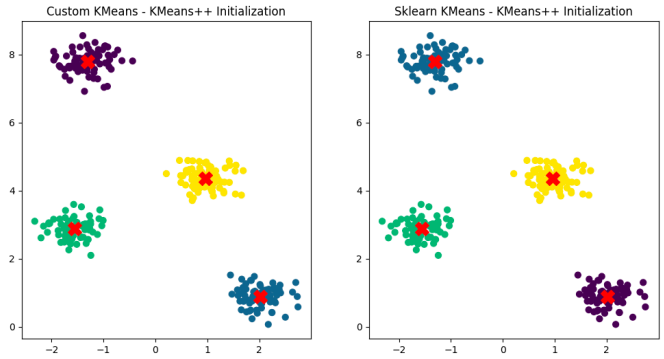
What didn't work

For the most part, my custom KMeans implementation was able to perform similarly to Sklearn's implementation. While testing it on multiple datasets like *make_moons, make_circles,* and *make_blobs*, my model returned a similar plot and silhouette score compared to Sklearn. However, when manipulating the features, I noticed some differences between my models and sklearn. For one, for random initialization, when the standard deviation in *make_blobs* was below .5, it seemed that my model's silhouette score was half of sklearn's score. The silhouette score is used to evaluate the quality of clustering by measuring how similar each data point is to its own cluster compared to other clusters, essentially combining cohesion and separation.

```
X, _ = make_blobs(n_samples=300, centers=4, cluster_std=0.30, random_state=0)
```



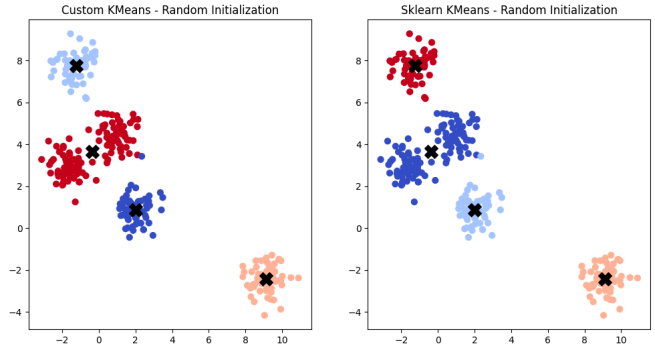

```
Custom KMeans (Random Init) – Silhouette Score: 0.497
Sklearn KMeans (Random Init) – Silhouette Score: 0.841
```



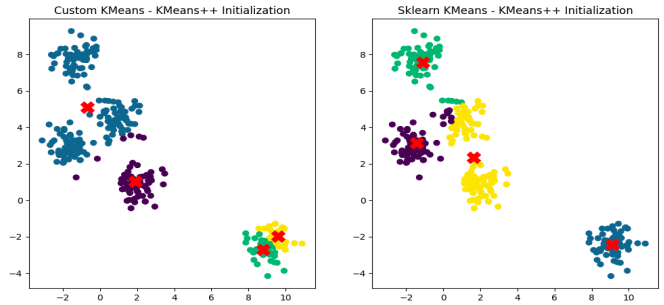

```
Custom KMeans (KMeans++ Init) – Silhouette Score: 0.841
Sklearn KMeans (KMeans++ Init) – Silhouette Score: 0.841
```

In the following case, the center was upped to 5 and the standard deviation of the clusters was reset to what it was originally set to.

```
X, _ = make_blobs(n_samples=300, centers=5, cluster_std=0.60, random_state=0)
```



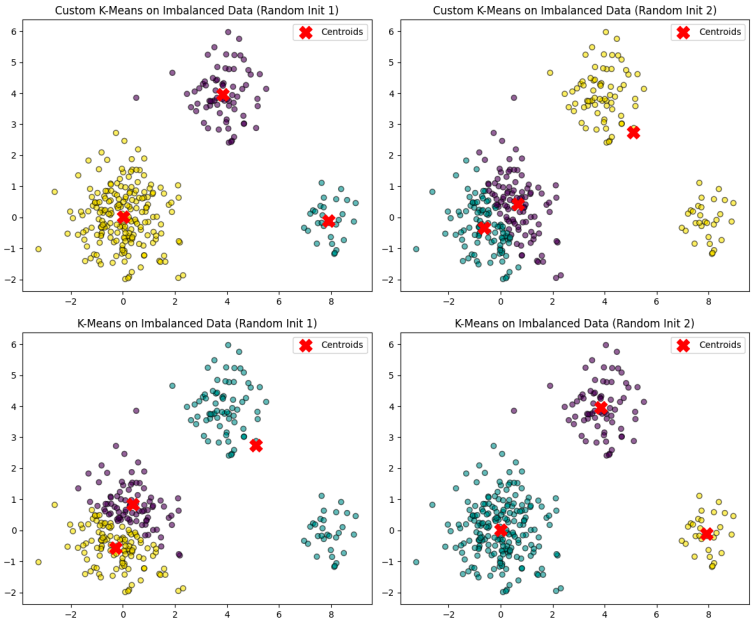

```
Custom KMeans (Random Init) – Silhouette Score: 0.643
Sklearn KMeans (Random Init) – Silhouette Score: 0.643
```




```
Custom KMeans (KMeans++ Init) – Silhouette Score: 0.439
Sklearn KMeans (KMeans++ Init) – Silhouette Score: 0.586
```

As can be seen from the results, my model didn't perform as well using kmeans++ initialization. While the score isn't bad, it is more that .1 off which isn't ideal.

With unbalanced data, there was also some things that caught my attention. Firstly, I used the same imbalanced set that was present in the demo shown in class to test using my custom KMeans. Essentially, there are 3 datasets with different sample sizes and feature values. One test in using *random_state* = 42 and the other is *random_state* = 1. Both were set to random initialization as well. This was the results:



I noticed that the plots looked similar but swapped in a sense. Looking into it, I believe it is a result of the difference in randomness techniques between my custom KMeans and Sklearn's. I tried to modify mine to match the random method Sklearn is using but to no avail. With these observations, I can determine that my model isn't as robust as sklearn's implementation. While it does perform very similar or even slightly better, there are some instances where my implementation falls in comparison to sklearn. This could a result of the way I have my cluster splitting, handling of empty clusters, or even the randomization I use compared to sklearn that could cause these slight differences with my model.

What was learned

Implementing KMeans I learned more about kmeans++ and how that affects the cluster initialization as well as learning more about silhouette score and how helpful that metric is for KMeans

Computational Cost

For computational cost, the time complexity would be O(n * k * i * d) for both KMeans and KMeans++

- n: number of samples
- k: number of clusters
- i: number of iterations
- d: number of dimensions

However, KMeans++ would be slightly higher that random init of KMeans because it improves convergence

Strengths / Weaknesses

Strengths

KMeans

- Very useful for large datasets as it is a simple and fast algorithm
- The KMeans algorithm or class is easy to implement
- KMeans also works very well on spherical, equally sized clusters
-

KMeans++

- KMeans++ is better at initializing centroids which leads to faster convergence
- KMeans++ is also less likely to fall into poor local minima

Weaknesses

KMeans and KMeans++ have the same weaknesses

- One weakness relates to having to specify the number of clusters (k) which was something I noticed and tested
- KMeans is also sensitive to initialization as poor initial centroids can lead to suboptimal clustering
    - *Kmeans++* and multiple runs with *n_init* helps reduce that risk
- Kmeans also assumes spherical, equally sized clusters which means that it performs poorly with clusters that aren't of that format like overlapping clusters or elongated clusters
- KMeans is also sensitive to outliers as they can shift centroids and complicate results significantly

MiniBatchKMeans

Features

The features of minibatch come from its parent class, KMeans

```
    def __init__(self, n_clusters=3, batch_size=100, **kwargs):
        super().__init__(n_clusters=n_clusters, **kwargs)
        self.batch_size = batch_size
```

- n-clusters refers to the number of clusters
- batch size refers to the number of samples per iteration or the size of each mini batch
- **kwargs essentially passes the additional keyword arguments that KMeans has like tol, init, etc. but it will include batch size

The batch size feature was added for minibatch because it's the feature that differentiates it from KMeans class while also using Kmeans and its features as its parent class.

What was attempted

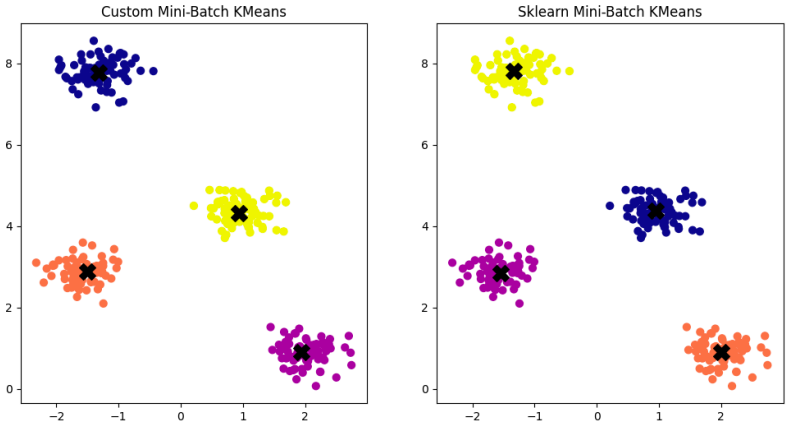
Minibatch is a variant of KMeans that builds upon it by having a clustering algorithm that's improves efficiency of the standard algorithm. Unlike KMeans, minibatch updates centroids by using small, randomly selected subsets (minibatches) of the dataset as opposed to the full set of data.

What worked

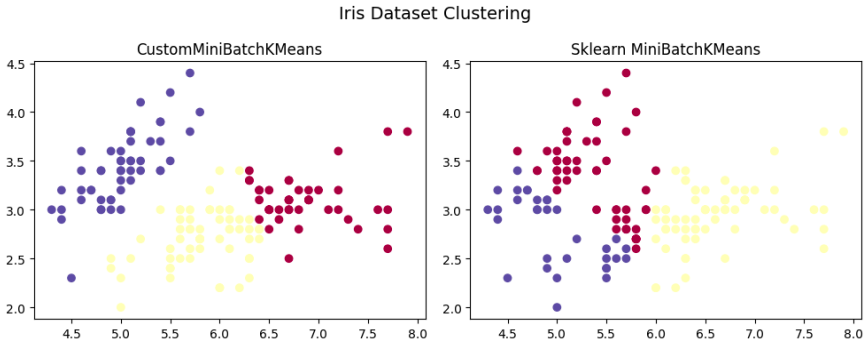
In terms of what worked, I would say my model or custom implementation was quite successful. For the most part, my model performed like sklearn for most tests. For standard tests on *make_moons, make_blobs, and make_circles*, my minibatch implementation performed the same

as sklearn. Similarly, on an anisotropic dataset, the iris dataset, and the wine dataset, my custom minibatch implementation performed equal or slightly better than sklearn.

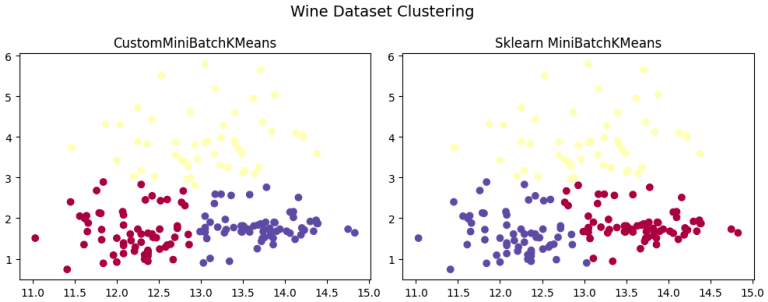
```
Custom Mini-Batch KMeans - Silhouette Score: 0.841
Sklearn Mini-Batch KMeans - Silhouette Score: 0.841
```




```
Iris Dataset
CustomMiniBatchKMeans Silhouette Score: 0.4414
Sklearn MiniBatchKMeans Silhouette Score: 0.3545
```



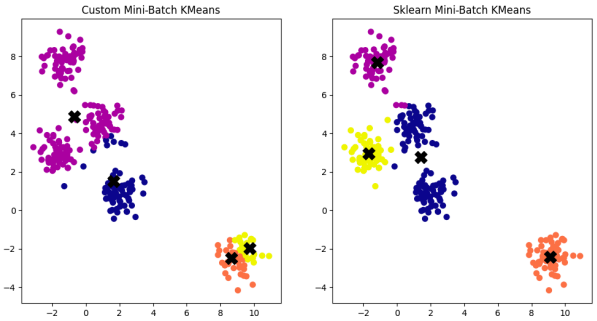

```
Wine Dataset
CustomMiniBatchKMeans Silhouette Score: 0.4807
Sklearn MiniBatchKMeans Silhouette Score: 0.4746
```



What didn't work

For what didn't work, there was case where my model didn't perform as well as sklearn. In the following case using the *make_blobs* dataset, the centers were upped to 5 and the standard deviation of the clusters was reset to what it was originally set to.

```
X, _ = make_blobs(n_samples=300, centers=5, cluster_std=0.60, random_state=0)
```




```
Custom Mini-Batch KMeans - Silhouette Score: 0.430
Sklearn Mini-Batch KMeans - Silhouette Score: 0.599
```

As can be seen from the results, my model didn't perform as well compared to sklearn. While the score isn't bad, it is more that .1 off which isn't ideal. I believe this issue lies with something within my custom KMeans as minibatch references the algorithm.

What I learned

What I learned for minibatch was a variation of KMeans that is more efficient that its parent class. It was interesting to see how it compared with KMeans random and Kmeans++ initializations as well.

Computational Cost

For computational cost, the time complexity would be O(b * k * i * d) for MiniBatchKMeans

- b: batch size << n
- k: number of clusters
- i: number of iterations
- d: number of dimensions

Strengths / Weaknesses

Strengths

- MiniBatch is very scalable for large datasets
- MiniBatch is also faster and more memory efficient than KMeans
- MiniBatch also converges reasonably well compared to KMeans

Weaknesses

- One of the weaknesses of MiniBatch is that it has a lower accuracy than KMeans
- Another weakness is that results depend on batch size and convergence criteria

Agglomerative Clustering (Bottom-Up Clustering)

Features

For agglomerative clusters, these are the features I used:

```
    def __init__(self, n_clusters=2, linkage='single'):
        self.n_clusters = n_clusters
        self.linkage = linkage
```

- n-clusters refers to the number of clusters

- linkage refers to the linkage method that the algorithm will use to calculate the distance between cluster
- custom model supports 4 linkage types: single, average, complete, and ward

While not much, these features are helpful as they support all 4 linkage types that agglomerative clustering has for testing against Sklearn as well as making the data easy to interpret and visualize with dendrograms

What was attempted

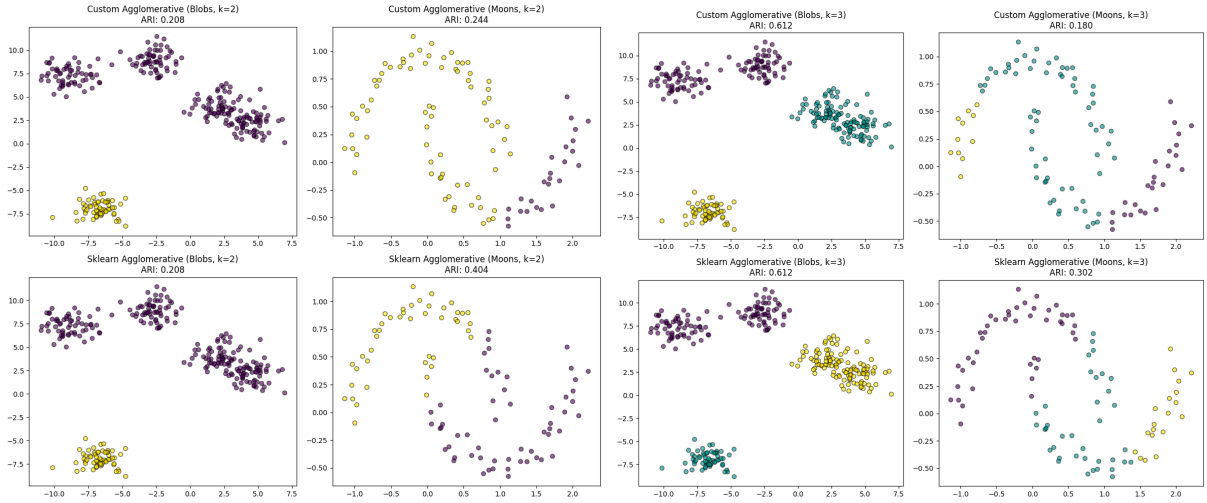
What was attempted was a custom agglomerative clustering algorithm that builds nested clusters from the bottom up. In my implementation, each data point would start as its own cluster, and they would iteratively merge to the closest pair of clusters. My implementation will also support 4 linkage types: single, complete, average, and ward.

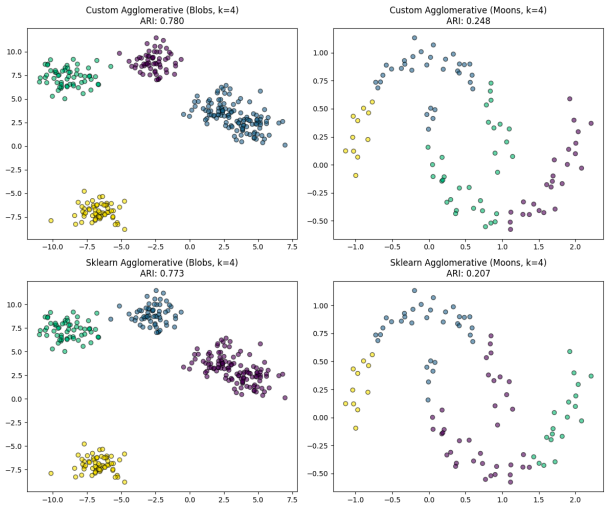
What worked

In terms of what worked, I would say that my custom implementation performed similarly to sklearn during most tests. I tested it against *make_blobs, make_circles,* and *make_moons* and it performed similarly or equal to sklearn both visually when plotted and in ari score. The ari score (Adjusted Rand Index) is a metric used to evaluate the similarity between 2 clustering, one predicted and one ground truth. This was tested on single, complete, and average linking types.

What didn't work

When testing my custom agglomerative clustering on different k values, this is the moment I saw that my model wasn't as precise as I had thought. The k value references to the number of clusters that would be present. For *make_blobs*, my model performed equal or very similar to sklearn for all tests of k = 2 - 4. That wasn't the case for *make_moons* though.





As we can see from the plots, my custom implementation was either close to half of sklearns's ari score or slightly above it means that for k = 4, it performed better and for k = 2 and 3, it performed a lot worse. Using this we can say that my model is inconsistent in a way as it is less accurate for k = 2 and 3 but performs better for k = 4 compared to sklearn. Since I was using the linkage type 'ward' for this testing, I can say that my implementation of this linkage type is slightly different that how it is implemented by sklearn.

More testing was done on the iris and wine datasets. For iris, my implementation had a .5 difference in ARI score and had a somewhat similar plot. For wine, my implementation performed very poorly with an ARI score of -0.007 compared to the 0.790 of Sklearn. The plot was also very telling as my implementation had no only one cluster appear while Sklearn had 3.

What I learned

What I learned implementing Agglomerative clustering was how different k values affect the model performance as mine had very differentiating results for some instances.

Computational Cost

- Time complexity: O(n^3) naïve or O(n^2 log n) with optimizations
- Memory usage: O(n^2)

Strengths / Weaknesses

Strengths

- Agglomerative clustering removes the need to specify the number of clusters early
- Agglomerative works well with non-spherical clusters and can handle small datasets and nested cluster structures

Weaknesses

- Agglomerative is computationally expensive for larger datasets
- Agglomerative also requires a distance metric and linkage method
- Agglomerative is also not robust to noisy data or outliers

Divisive Clustering (Top-Down Hierarchical)

Features

For Divisive clustering, these are the features implemented:

```
    def __init__(self, n_clusters=2):
        self.n_clusters = n_clusters
```

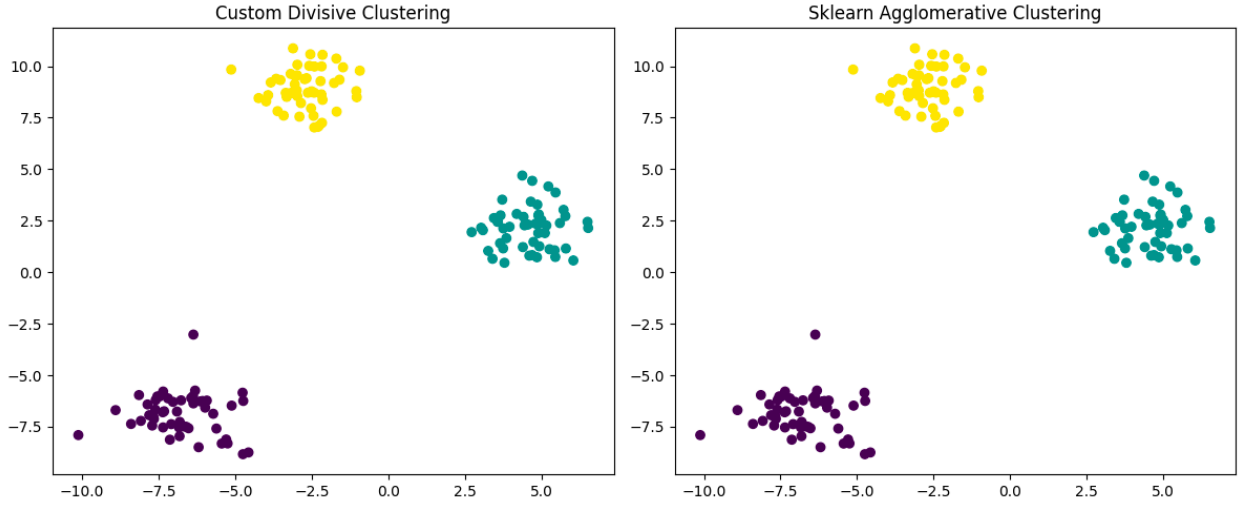
- n_clusters refer to the number of clusters

What was attempted

What was attempted for Divisive Clustering was implementing a top-down approach where we start with one big cluster and then split it into multiple smaller clusters.

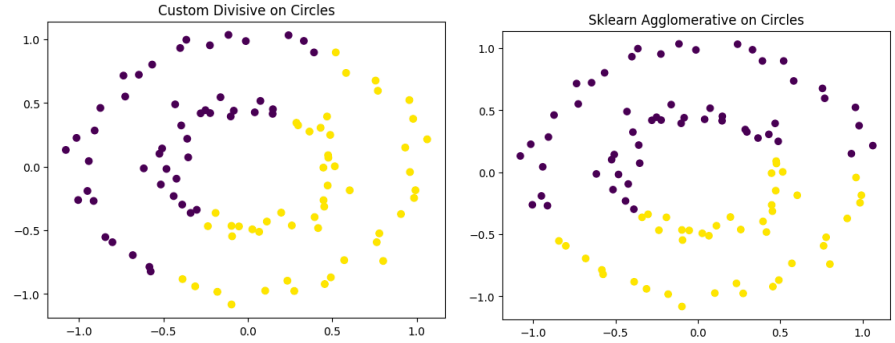
What worked

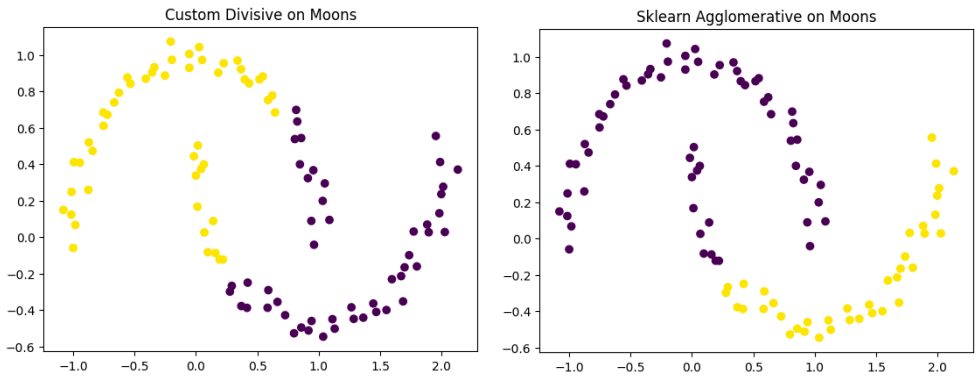
For Divisive Clustering, Sklearn doesn't have its own implementation of it, so I tested it against agglomerative clustering. It was hard to decide whether my implementation of this method had worked but it did get similar results to sklearn regardless. For make_blobs, my divisive clustering matched up very well with sklearn's agglomerative clustering.



What didn't work

Plotting using other datasets like *make_moons* or *make_circles* did show how different the algorithms are. In terms of ari score, my divisive clustering and sklearn's agglomerative clustering matched perfectly but that doesn't really say much other that the predicted plot matched the ground truth. What matter is how it was plotted.





These algorithms don't act the same since one is top down and the other is bottom up but it was interesting to see how they diferentiate from each other. I wouldn't say my implementaion was bad as there isnt an sklearn model to compare it to.

What I Learned

For Divisive clustering, what I learned was a new method in hierarchical clustering. With Divisive being top down compared to agglomerative being bottom up, it was interesting to see the varying results between both classes.

Computational Cost

For computational cost, the time complexity would be O(n^2) or worse as it is more expensive than agglomerative clustering in practice

Strengths / Weaknesses

Strengths

- Divisive clustering is good when the whole dataset should be considered before splitting
- Divisive clustering can also potentially avoid poor local merges as it does a global view first

Weaknesses

- Divisive clustering is harder to implement
- Divisible clustering is also more computationally expensive and less commonly used compared to agglomerative clustering

DBSCAN

Features

These are the features for my custom DBSCAN algorithm:

```
    def __init__(self, eps=0.5, min_samples=5):
        self.eps = eps
        self.min_samples = min_samples
```

- eps refer to the max radius of the neighborhood to consider for a point
- min_samples is the min number of points in an eps-neighborhood for a point to be considered a core point

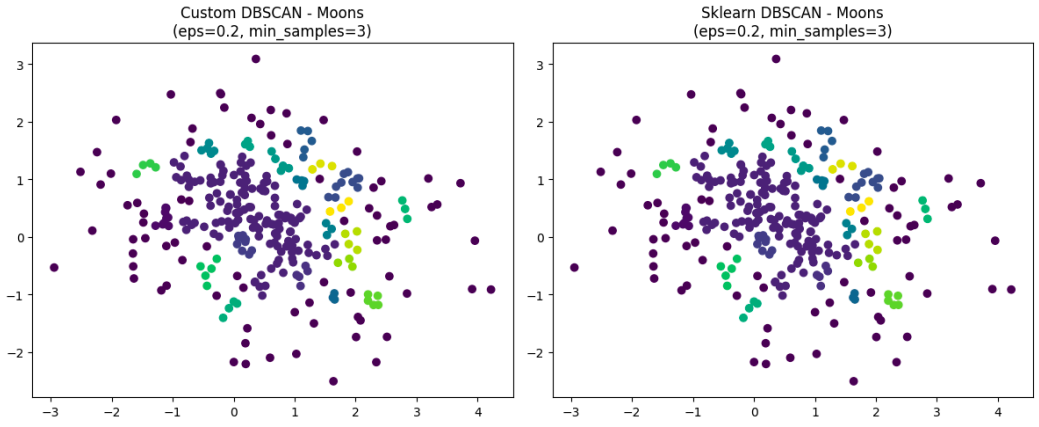
What was attempted

What was attempted was a custom implementation of DBSCAN, a clustering method that groups points that are closely packed together while marking the lone points as noise or outliers. This implementation also focused on eps and min_samples with eps being the max distance between two points for them to be considered in the same neighborhood while min_samples relate to the number of samples in a neighborhood for a point to be considered a core point.

What worked

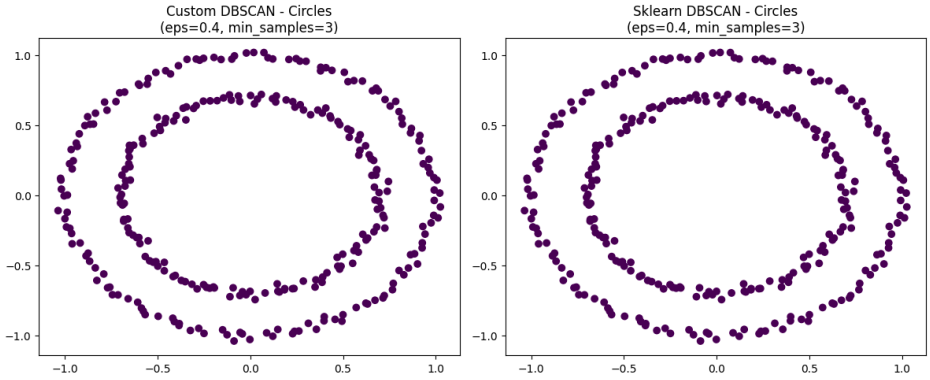
I did a couple of tests on *make_blobs, make_circles,* and *make_moons* with different eps and *min_samples* values. Throughout this, I was also changing the noise value in *make_moons*, the *cluster_std* in make_blobs, and the noise and factor values in make_circles and can say that my custom implementation of DBSCAN performed very well compared to sklearn. Using metrics like ARI, Silhouette Score, Homogeneity which checks if each cluster contains members of one class, Completeness which checks if all members of a class are in the same cluster, and V-measure which is the harmonic mean between homogeneity and completeness, my custom model was produced equal results to sklearn.

```
Dataset: Moons | eps: 0.2, min_samples: 3
Custom DBSCAN:
  ARI: -0.000
  Homogeneity: 0.104
  Completeness: 0.036
  V-Measure: 0.053
  Silhouette: 0.012
  Noise Points: 78
Sklearn DBSCAN:
  ARI: -0.000
  Homogeneity: 0.104
  Completeness: 0.036
  V-Measure: 0.053
  Silhouette: 0.012
  Noise Points: 78
```

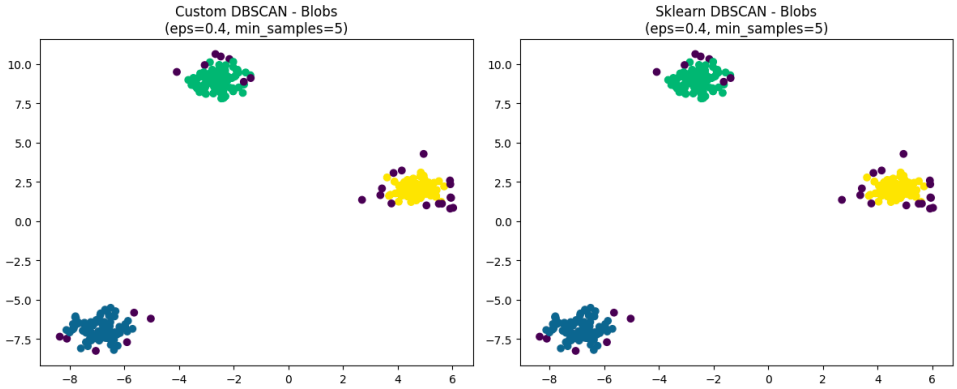



```
Dataset: Circles | eps: 0.4, min_samples: 3
Custom DBSCAN:
  ARI: 0.000
  Homogeneity: 0.000
  Completeness: 1.000
  V-Measure: 0.000
  Silhouette: nan
```

```
  Noise Points: 0
Sklearn DBSCAN:
  ARI: 0.000
  Homogeneity: 0.000
  Completeness: 1.000
  V-Measure: 0.000
  Silhouette: nan
  Noise Points: 0
```

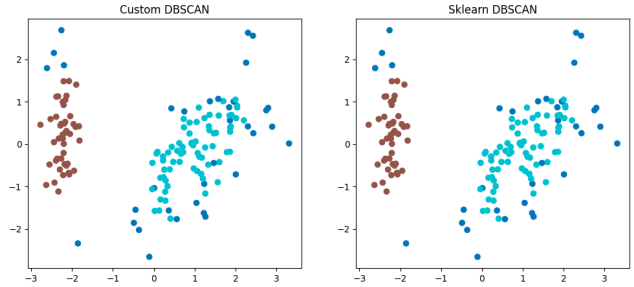



```
Dataset: Blobs | eps: 0.4, min_samples: 5
Custom DBSCAN:
  ARI: 0.853
  Homogeneity: 0.912
  Completeness: 0.766
  V-Measure: 0.833
  Silhouette: 0.919
  Noise Points: 29
Sklearn DBSCAN:
  ARI: 0.853
  Homogeneity: 0.912
  Completeness: 0.766
  V-Measure: 0.833
  Silhouette: 0.919
  Noise Points: 29
```

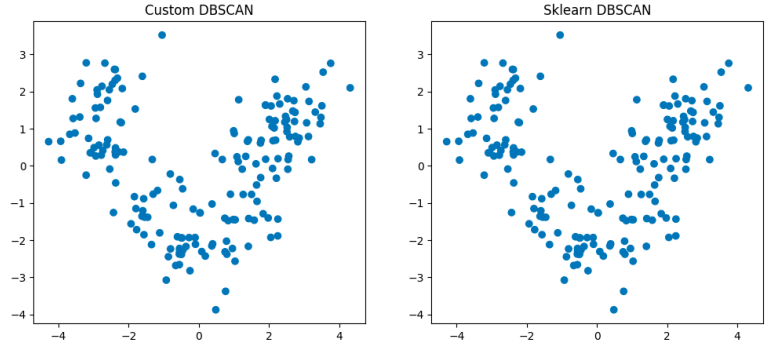


I also tested my model against Sklearn on the iris and wine datasets and it performed very well on multiple different eps and min_samples values.

Iris:



Wine:



What didn't work

For what I tested, I would say everything worked. My model was very precise when compared to sklearn and produced equal results to it.

What I learned

What I learned implementing DBSCAN is how sensitive the class can be with different eps and *min_samples* values. I also learned about new metric that I didn't know about before which helped compare my model to a better degree than for other cases.

Computational Cost

- Using KD-Tree: O(n log n) for lower dimensions
- O(n^2) without special index

Strengths / Weaknesses

Strengths

- DBSCAN does not require the number of clusters and can find arbitrarily shaped clusters
- DBSCAN is also robust to noise and outlier and handles clusters of different sizes and densities
-

Weaknesses

- DBSCAN struggles in high dimension spaces because of the curse or dimensionality
- The parameters for DBSCAN are also harder to tune
- DBSCAN also has poor performance when clusters have varying density

HDBSCAN (Hierarchical DBSCAN)

Features

Features for HDBSCAN:

```
    def __init__(self, min_samples=5, min_cluster_size=5):
        self.min_samples = min_samples
        self.min_cluster_size = min_cluster_size
        self.labels_ = None
```

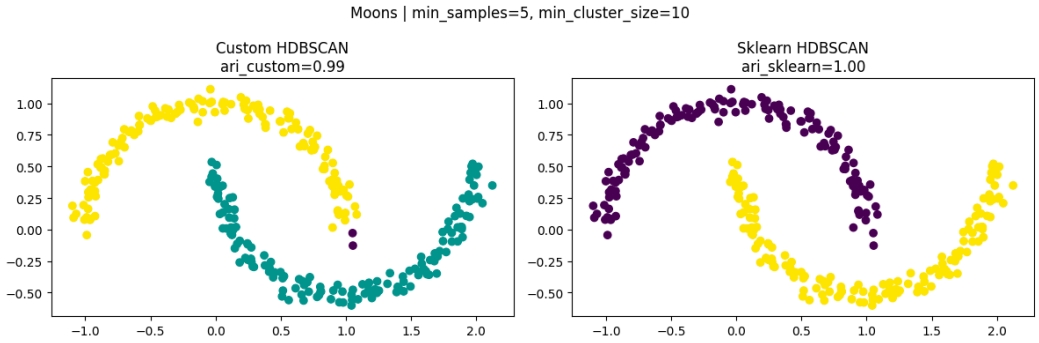
- min_samples is the number of points in a neighborhood to consider a point as a core point and this affects how strictly dense a region must be to be considered a core of a cluster
- min_cluster_size is the minimum number of point a group must have to be considered a cluster and determines the smallest allowable cluster size on the output

What was attempted

What was attempted was to implement a custom HDBSCAN class that difference from DBSCAN in that HDBSCAN varies by hierarchy and density levels.

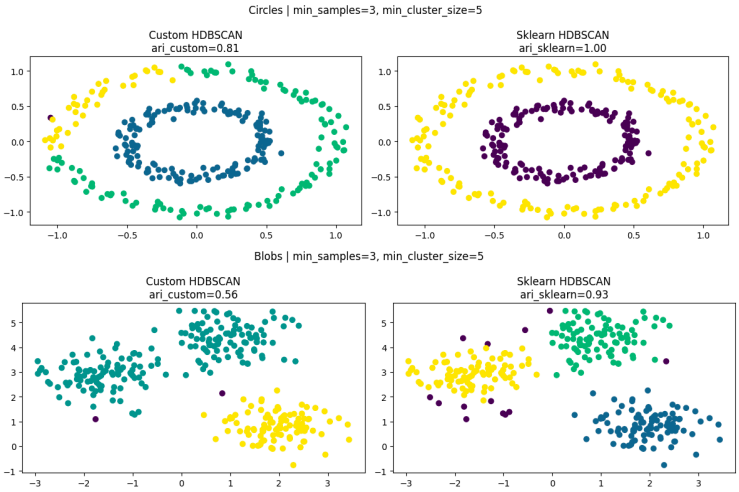
What worked

In terms of what worked, tests on *make_moons* were the most successful with the ari score having a difference of .01.



What didn't work

When testing for different *min_samples* and *min_cluster_size* on *make_moons, make_circles,* and *make_blobs*, I could see where my model was lacking and where it performed better than sklearn. There where instances when the ari score for my model and sklearn was so different with mine either being far lower or it being slightly higher.



What was learned
I learned how much *min_samples* and *min_cluster_size* affects the performance of the models. HDBSCAN has a complicated implementation that I did struggle with at some points and can be seen with some of the results of my model in comparison with sklearn.

Computational Cost
For computational cost, the time complexity would be O(n log n)

Strengths / Weaknesses
Strengths
- HDBSCAN removes the need to specify number of clusters, being able to handle clusters of different shapes, size, and density
- HDBSCAN also identifies noise
- HDBSCAN is also more robust than DBSCAN with better hierarchy

Weaknesses
- HDBSCAN is more complex to implement which I struggled with
- HDBSCAN can also still struggle with higher dimension data
- The performance of HDBSCAN is impacted by the choice of *min_cluster_size* and other parameters

OPTICS (Ordering Points to Identify Clustering Structure)

Features

```
    def __init__(self, min_samples=5, max_eps=np.inf):
        self.min_samples = min_samples
        self.max_eps = max_eps
```

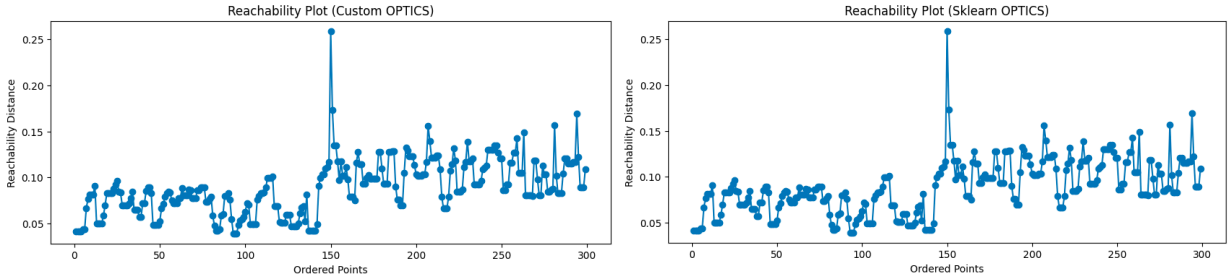
- min_samples is the minimum number of points to form a dense region
- max_eps are the maximum search radius for neighborhood expansion

What was attempted
OPTICS is a class that is very similar to DBSCAN, but the difference is that OPTICS has more flexibility in terms of varying density clusters. For my custom OPTICS class, I attempted to

implement a core distance calculation, cluster expansion, 2 types of cluster extraction, and a visualization of a reachability plot and compare that to sklearn.

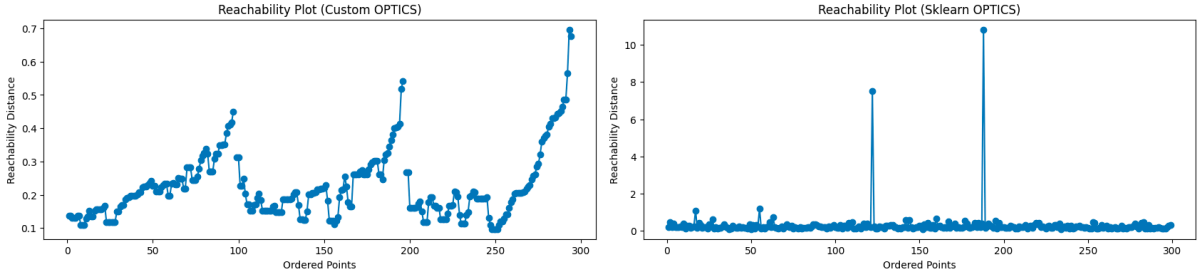
What worked

OPTICS was one of the classes that I felt my implementation did worse compared to sklearn in respect to all the other implementations. Doing multiple tests, there was instances where my model matches sklearn and other times when it didn't. My model was the most accurate in my eyes for the *make_circles* dataset. My model and sklearn produced similar reachability plots for this dataset than compared to the others.



What didn't work

For the other cases, my model produced different reachability plots compared to sklearn meaning my model wasn't as precise as sklearn's implementation. There was this test run on *make_blobs* that produced very different results.



What was learned

While implementing OPTICS , I learned that it was a very hard class to implement. With such contradicting results in ari scores and reachability plots, it was hard to tell if my model was that bad or somehow producing better results, ultimately concluding that my implementation wasn't as good as sklearn. I also learned about reachability plots which I didn't know about beforehand.

Computational Cost

- O(n log n)

Strengths / Weaknesses

<u>Strengths</u>

- OPTICS handles clusters of varying density and doesn't require the number of clusters ahead of time

- OPTICS also produced a reachability plot

Weaknesses

- The output of OPTICS is harder to interpret than DBSCAN
- OPTICS also requires post-processing