Lesson 3: Neighborhood Approaches and DBSCAN

Lesson Objectives

By the end of this lesson, you will be able to:

* Explain the intuition behind neighborhood approaches – mainly DBSCAN
* Implement the DBSCAN algorithm from scratch and using packages
* Understand the differences between K-Means, Hierarchical Clustering, and DBSCAN

Introduction

Clustering Refresh

So far, we have covered two popular ways of approaching the clustering problem: K-Means and Hierarchical Clustering. Both clustering techniques have pros and cons associated with how they are carried out. Once again let’s revisit where we have been in the first two lessons so we can gain further context to where we will be going in this lesson.

In the challenge space of unsupervised learning you will be presented with a collection of feature data, but no complementary labels telling you what these feature variables necessarily mean. While you may not get a discrete view into what the target labels are, you can get some semblance of structure out of the data by clustering similar groups together and see what is similar within groups. The first approach we covered to achieve this goal of clustering similar data points is K-Means.

K-Means clustering works by finding “K” number clusters in your data through pairwise Euclidean distance calculations. “K” points (also called centroids) are randomly initialized in your data and the distance is calculated from each data point to each of the centroids. The minimum of these distances designates which cluster a data point belongs to. Once every point has been assigned to a cluster, the mean intra-cluster data point is calculated as the new centroid. This process is repeated until the newly calculated cluster centroid no longer changes position.

K-Means works best for simpler data challenges where speed is paramount. By simply looking at closest data points there is not a lot of computational overhead, however there is also a greater degree of challenge when it comes to higher dimension data sets. K-Means is also not ideal if you are unaware of the potential number of clusters you would be looking for. An example we have worked with in the past is looking at e-commerce logs for your online store. If your manager asks you to find 3-4 groups in the data that can serve as marketing segments, K-Means could be a great approach as long as your features are differentiable enough that linear hyperplanes could successfully group class members apart.

The second clustering approach we explored was Hierarchical Clustering. This method can work in multiple ways – either agglomerative of divisive. Agglomerative clustering works from the bottoms-up approach, treating each data point as its own “cluster” and recursively grouping them together with a linkage criteria. Divisive clustering works in the opposite direction by treating all data points as one large class and recursively breaking them down into smaller clusters. This approach has the benefit of fully understanding the entire data distribution as it calculates splitting potential, however it is typically not done in practice due to its greater complexity. Hierarchical clustering is a strong contender for your clustering needs when it comes to not know anything about the data. Using a dendrogram you can visualize all the splits in your data and consider what number of clusters makes sense after the fact. This can be really helpful in your specific use case, however it also comes at a higher computational cost that is seen in K-Means.

Finally, in this lesson, we will cover a clustering approach that will serve best in highly complex data: DBSCAN. Canonically, this method has always been seen as a high performer in datasets that have a lot of densely interspersed data. Let’s walk through why it does so well in these use cases.

Who are my neighbors?

In the first two chapters we explored the concept of likeliness being described as a function of Euclidean distance – data points that are closer to any one point can be seen as similar, whereas those that are further away in Euclidean space can be seen as dissimilar. This notion is seen once again in the DBSCAN (Density-Based Spatial Clustering of Applications with Noise) algorithm. As alluded to by the lengthy name, the DBSCAN approach expands upon basic distance metric evaluation by also incorporating the notion of density. If there are clumps of data points that all exist in the same area of each other, they can be seen as members of the same cluster.

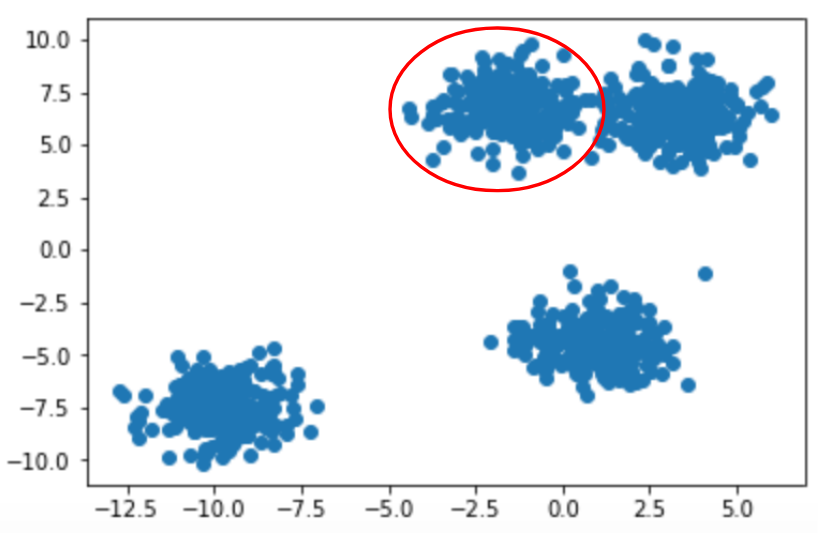


Figure 3.1: Neighbors has a direct connection to clusters. In this simple example we can see 4 neighborhoods.

The density-based approach has a number of benefits when compared to the past approaches we’ve covered that focus exclusively on distance. If you were just focusing on distance as a clustering threshold then you may find your clustering to make little sense if faced with a sparse feature space with outliers. Both K-Means and Hierarchical Clustering will automatically group together all data points in the space until no points are left. While Hierarchical Clustering does somewhat provide a path around this issue since you can dictate where clusters are formed using a dendrogram post clustering run, K-Means is the most susceptible to failing as it is the most “simple” approach to clustering.

By incorporating the notion of neighbor density in DBSCAN, we can leave outliers out of clusters if we choose to based on the hyperparameters we choose at run time. Only the data points that have close neighbors will be seen as members within the same cluster and those that are farther away can be left as un-clustered outliers.

Introduction: DBSCAN

As mentioned in the previous section, the strength of DBSCAN becomes apparent when we analyze the benefits of taking a density-based approach to clustering. DBSCAN evaluates density as a combination of neighborhood radius and minimum points found in a neighborhood to be deemed a cluster.

This concept can be driven home if we re-consider the scenario where you are tasked with analyzing the invoice logs of an e-commerce store. In the past examples it was made clear that we can find similar customers based off their demographics and what they have ordered. Knowing this information, we can take a more targeted approach to marketing products that would be of interest to them specifically instead of any generic visitor to your store website. Hopefully by now that is clear - but what may not have been clear is the fact that a store never has consistent multiple-order customers. In most cases the bulk of a standard online store’s orders will come from long-tail customers that simply order once and will never have an interest in coming back.

DBSCAN differs from K-Means and Hierarchical Clustering because you can build this intuition into how we evaluate the clusters of customers we are interested in forming. It can cut through the noise in an easier fashion and only point out customers that we have the highest potential to be remarketed towards in a campaign.

By clustering through the concept of a neighborhood we can separate out the one-off customers that can be seen as random noise relative to the more valuable customers that come back to our store time and time again. This approach of course calls into question how we establish the best numbers when it comes to neighborhood radius and minimum points per neighborhood.

As a high-level heuristic, we want to have our neighborhood radius to be small but not too small. At one end of the extreme you can have the neighborhood radius be quite high – this can max out at treating all points in the feature space as one massive cluster. On the opposite end of the extreme you can have a very small neighborhood radius. Too small neighborhood radii can result in no points being clustered together and having a large collection of single member clusters.

Similar logic applies when it comes to the minimum number of points. Minimum points can be seen as a secondary threshold that tunes the neighborhood radius a bit depending on what data you have available in your space. If all of the data in your feature space is extremely sparse, minimum points becomes extremely valuable in tandem with neighborhood radius to make sure you don’t just have a large number of uncorrelated data points. When you have very dense data, minimum points threshold becomes less of a driving factor as opposed to neighborhood radius.

As you can see from the above two hyperparameter rules, the best options are as usual dependent on what your data set looks like. Oftentimes you will want to finding the perfect “goldilocks” zone of being not too small in your hyperparameters but also not too large.

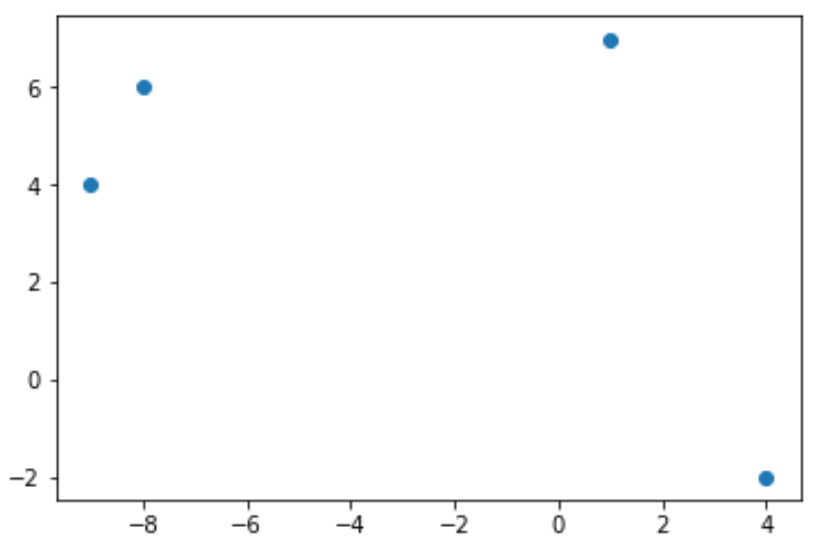
In-Depth: DBSCAN

To see how DBSCAN works, we can trace the path of a simple toy program as it merges together to form a variety of clusters and noise-labeled data points:

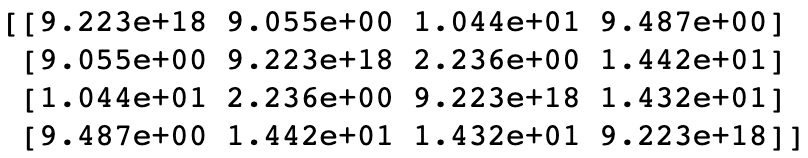
1. Given **n** unvisited sample data points, move through each point in a loop and mark as visited.
2. From each point look at the distance to every other point in the data set.
3. For all points that fall within the neighborhood radius hyperparameter, connect them as neighbors.
4. Check to see if the number of neighbors is at least as many as minimum points required.
5. If minimum point threshold reached, group together as a cluster. If not, mark point as noise.
6. Repeat until all data points are categorized in clusters or as noise.

DBSCAN is fairly straightforward in some senses – while there are the new concepts of density through neighborhood radius and minimum points, at its core it is still just evaluating using a distance metric. Here is a simple example walking through the above steps in slightly more detail:

1. Given **4** sample data points, view each point as its own cluster.

* [ (1,7) ], [ (-8,6) ], [ (-9,4) ] , [ (4, -2) ]
* 

1. Calculate pairwise Euclidean distance between each of the points.

* 

1. From each point expand out a neighborhood size and form clusters.

* For the purpose of this example, lets imagine we passed through a neighborhood radius of 3. This means that any two points will be neighbors if the distance between them is less than 3.
* Points (-8,6) and (-9,4) are now candidates for clustering.

1. Points that **have no neighbors** are marked as noise and remain un-clustered.

* Points (1,7) and (4,-2) fall out of our frame of interest as useless in terms of clustering.

1. Points that **have neighbors** are then evaluated to see whether they pass the minimum points threshold

* In this example, if we had passed through a minimum points threshold of 2 then points (-8,6) and (-9,4) can formally be grouped together as a cluster.
* If we had a minimum points threshold of 3, then all 4 data points in this set would be considered superfluous noise.

1. Repeat this process on remaining un-visited data points.

At the end of this process you will have your entire dataset established as either within clusters or as unrelated noise. Hopefully as you can tell by walking through the toy example, DBSCAN performance is highly dependent on the threshold hyperparameters you choose a priori. This means that you may have to run DBSCAN a couple of times with different hyperparameter options to get an understanding of how they influence overall performance.

One great thing to notice about DBSCAN is that it does away with concepts of centroids that we saw in both K-Means and a centroid-focused implementation of Hierarchical Clustering. This feature allows DBSCAN to work better for complex datasets since most data in the wild is not shaped like clean blobs.

Exercise 1: Evaluating the Impact of Neighborhood Radius Size

For this exercise we will work in reverse of what we have typically seen in previous examples, by first seeing the packaged implementation of DBSCAN in sci-kit learn and then implementing it on our own. This is done on purpose to fully explore how different neighborhood radius sizes drastically impact DBSCAN performance.

First let’s generate some dummy data:

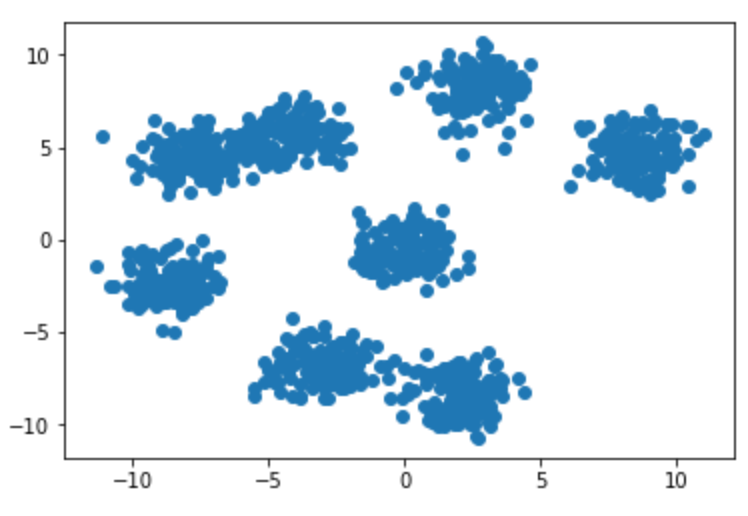
# Generate a random cluster dataset to experiment on. X = coordinate points, y = cluster labels (not needed)

X, y = make\_blobs(n\_samples=1000, centers=8, n\_features=2, random\_state=800)

# Visualize the data

plt.scatter(X[:,0], X[:,1])

plt.show()



After plotting the dummy data for this toy problem, you will see that the data set has 2 features and approximately 7-8 clusters. To implement DBSCAN using sci-kit learn you will need to instantiate a new sci-kit learn class:

db = DBSCAN(eps=0.5, min\_samples=10, metric='euclidean')

Our example DBSCAN instance is stored in the ‘db’ variable, and our hyperparameters are passed through on creation. For the sake of this example, you can see that the neighborhood radius (eps) is set to 0.5 while the minimum number of points is set to 10. To keep in line with our past lessons we will once again be using Euclidean distance as our distance metric.

Now let’s set up a loop that allows us to explore potential neighborhood radius size options interactively:

eps = [0.2,0.7]

for ep in eps:

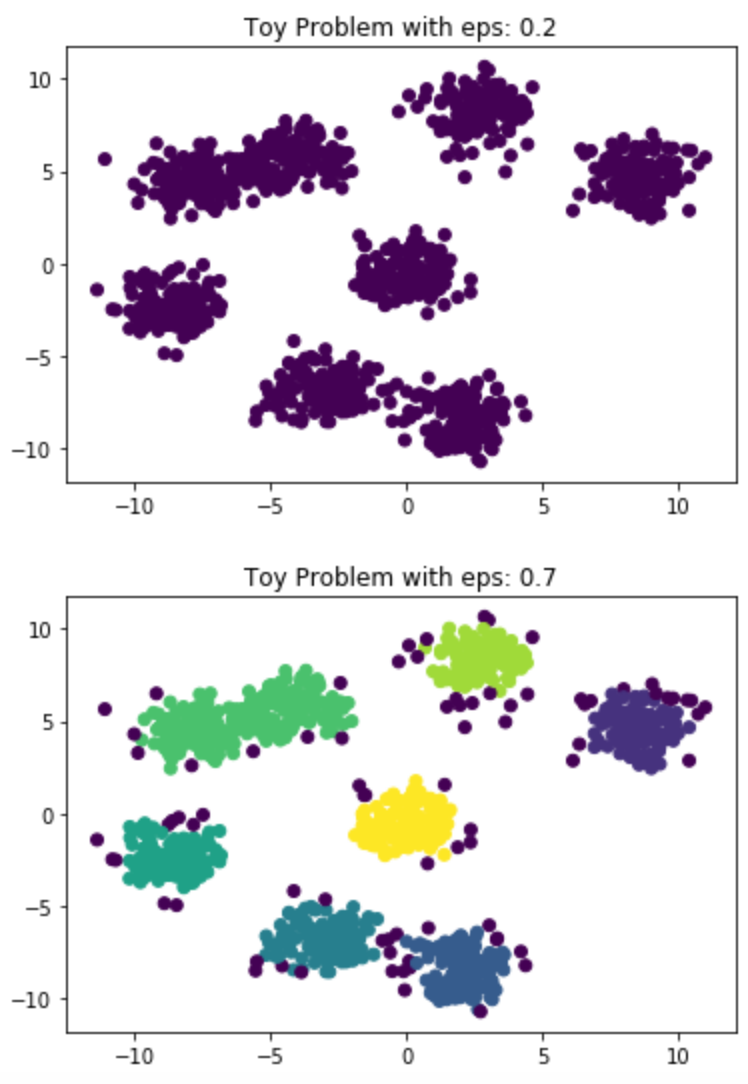
db = DBSCAN(eps=ep, min\_samples=10, metric='euclidean')

plt.scatter(X[:,0], X[:,1], c=db.fit\_predict(X))

plt.title('Toy Problem with eps: ' + str(ep))

plt.show()

The above code results in the following two plots:



As you can see from the plots, setting our neighborhood size too small will cause everything to be seen as random noise (purple points). Bumping our neighborhood size up a little bit allows us to form clusters that make more sense. Try recreating the above plots and experiment with varying eps sizes.

DBSCAN Attributes: Neighborhood Radius

In Exercise 1 you saw how impactful setting the proper neighborhood radius is on the performance of your DBSCAN implementation. If your neighborhood is too small, then you will run into issues where all the data is left un-clustered. If you set your neighborhood too large, then all of the data will similarly be grouped together into one cluster and not provide any value. If you explored the above exercise further with your own eps sizes, you may have noticed that it is very difficult to land on great clustering using only the neighborhood size. This is where a minimum points threshold comes in handy. We will visit that topic later.

To go deeper into the neighborhood concept of DBSCAN let’s take a deeper look at the “eps” hyperparameter you pass at instantiation time. Eps stands for epsilon and is the distance that your algorithm will look within when searching for neighbors. This epsilon value is converted to a radius that sweeps around any given data point in a circular manner to serve as a neighborhood.

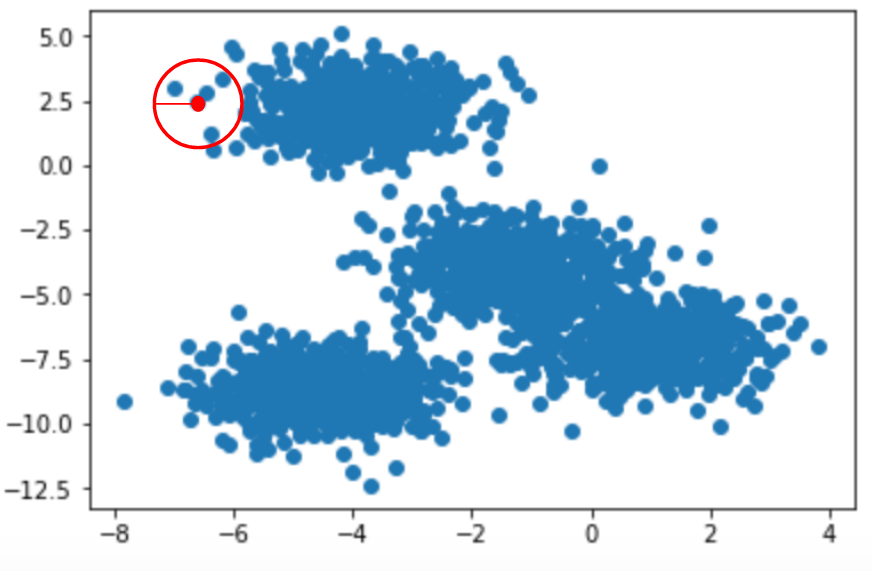


Figure 3.2: Visualization of neighborhood radius where red circle is the neighborhood. In this instance there will be 4 neighbors of the center point.

One key aspect to notice here is that the shape formed by your neighborhood search is a circle in two dimensions, and a sphere in three dimensions. This may impact the performance of your model simply based on how the data is structured. Once again blobs may seem like an intuitive structure to find – this may not always be the case. Fortunately DBSCAN is well equipped to handle this dilemma of clusters that you may be interested in that do not fit the explicit blob structure.

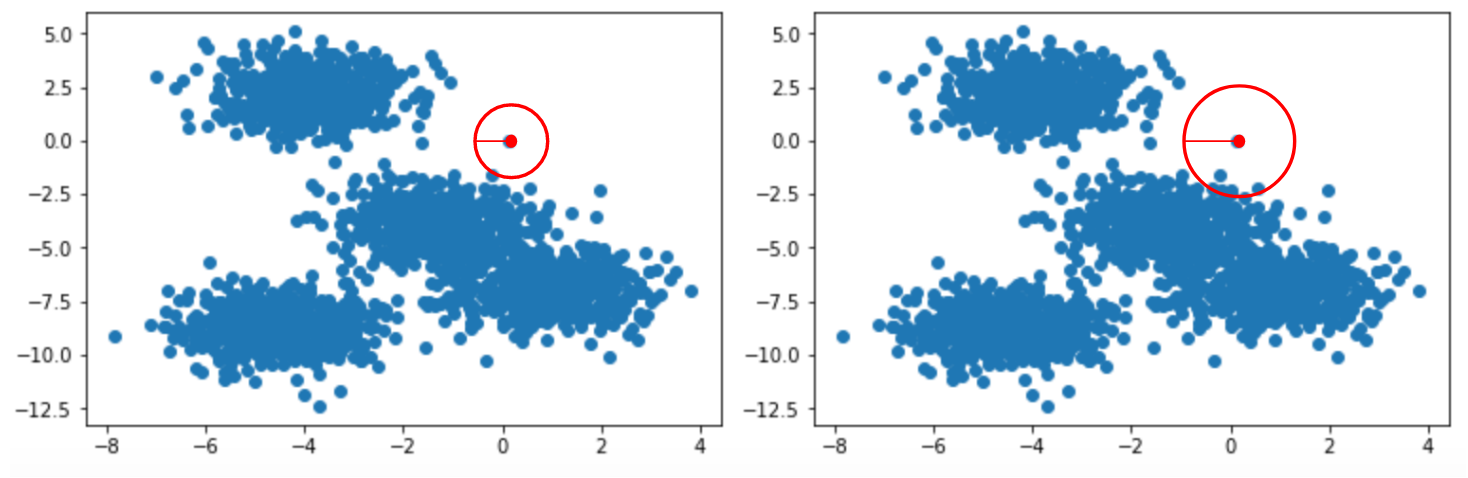


Figure 3.3: Impact of varying neighborhood radius size. On the left the data point will be classified as random noise. On the right the data point has multiple neighbors and could be its own cluster.

Activity 1: Implement DBSCAN from Scratch

**Scenario:** You are given a data set without prior background information and are requested to find the clusters that fit the data set best.

**Aim:** Given what you’ve learned about DBSCAN and distance metrics from prior lessons, build an implementation of DBSCAN from scratch in Python. You are free to use NumPy and SciPy to evaluate distances here.

DBSCAN Attributes: Minimum Points

The other core component to a successful implementation of DBSCAN beyond the neighborhood radius is the minimum number of points required to justify membership within a cluster. As mentioned earlier, this lower bound more obviously benefits your algorithm when it comes to sparser data sets. That’s not to say it is a useless parameter when you have very dense data however – while having single data points randomly interspersed through your feature space can be easily bucketed as “noise,” it becomes more of a gray area when we have random patches of 2-3 points for example. Should these data points be their own cluster, or also categorized as noise? Minimum points thresholding helps solve this problem.

In the Sci-Kit Learn implementation of DBSCAN this hyperparameter is seen in the “min\_samples” field passed on DBSCAN instance creation. This field is very valuable to play with in tandem with the neighborhood radius size hyperparameter to fully round out your density-based clustering approach.

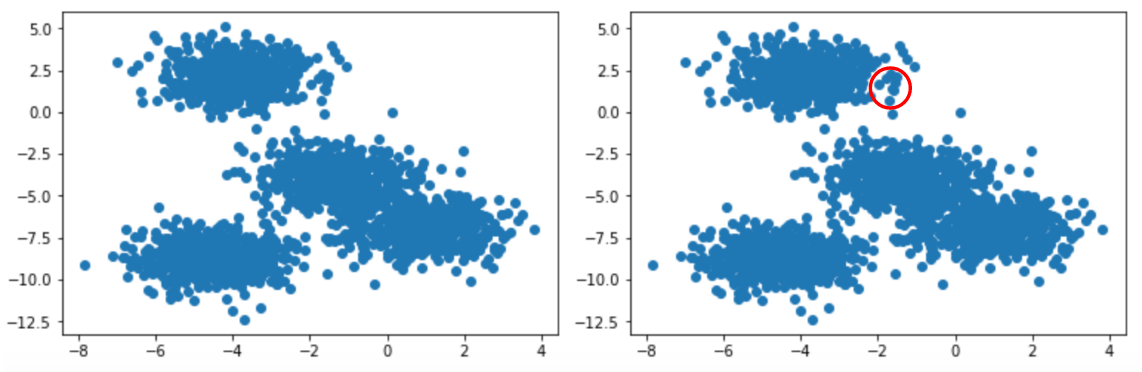


Figure 3.4: Minimum points threshold deciding whether a group of data points is noise or a cluster. On the right, if minimum points threshold is 10 points it will classify data in this neighborhood as noise.

In real world scenarios you can view minimum points being highly impactful when you have truly large amounts of data. Going back to the e-commerce store invoice example, if you had millions of orders per month then there may be enough instances of one-off customers that they can easily be viewed as their own cluster. This may be helpful depending on your use case, however it is important to keep in mind the subjective magnitudes that come with your data. If you have millions of data points then random noise can easily be seen as hundreds or even thousands of random one-off sales. However, if your data is on the scale of hundreds or thousands, single data points can be seen as random noise.

Exercise 2: Evaluating the Impact of Minimum Points Threshold

Similar to our first exercise where we explored the value of setting a proper neighborhood radius size, we will repeat the exercise but instead change the minimum points threshold on a variety of data sets. Once again let’s start with randomly generated data:

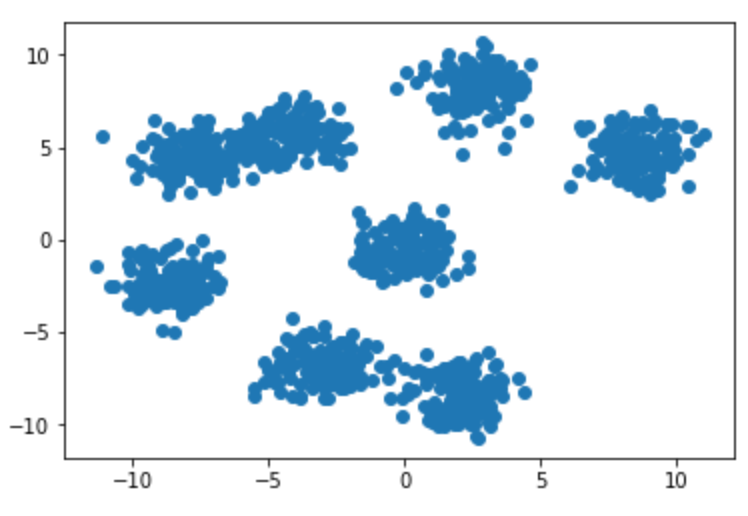
# Generate a random cluster dataset to experiment on. X = coordinate points, y = cluster labels (not needed)

X, y = make\_blobs(n\_samples=1000, centers=8, n\_features=2, random\_state=800)

# Visualize the data

plt.scatter(X[:,0], X[:,1])

plt.show()



With our same plotted data as before, let’s grab one of the better performing neighborhood radius sizes from Exercise 1 – eps = 0.7:

db = DBSCAN(eps=0.7, min\_samples=10, metric='euclidean')

After instantiating the DBSCAN clustering algorithm let’s treat the ‘min\_samples’ hyperparameter as the variable we wish to tune. We can cycle through a loop to find which minimum number of points works best for our use case:

num\_samples = [10,19,20]

for min\_num in num\_samples:

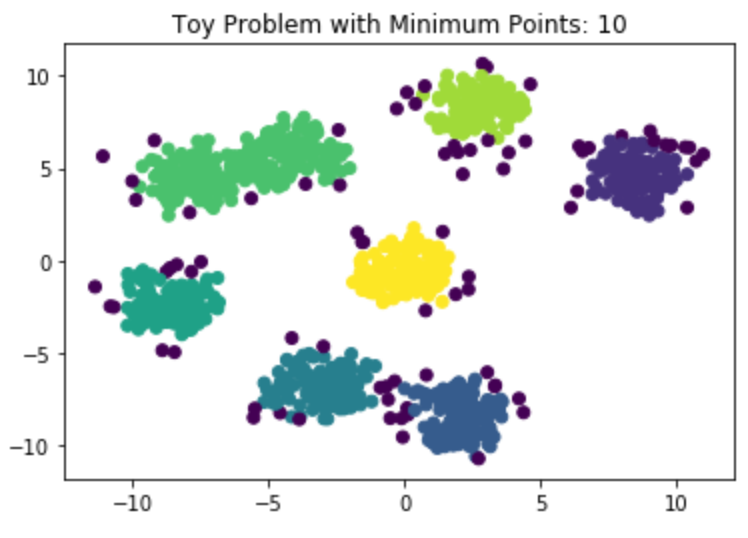
db = DBSCAN(eps=0.7, min\_samples=min\_num, metric='euclidean')

plt.scatter(X[:,0], X[:,1], c=db.fit\_predict(X))

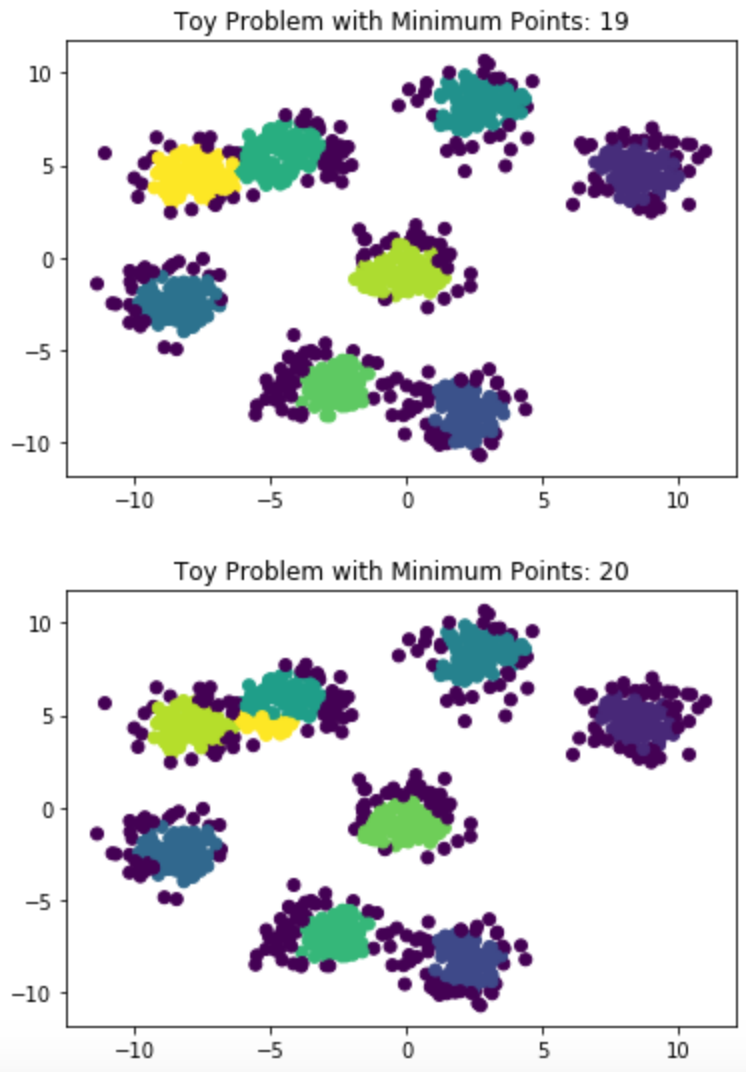
plt.title('Toy Problem with Minimum Points: ' + str(min\_num))

plt.show()

Looking at the first plot generated we can see where we ended if you followed Exercise 1 directly, using 10 minimum points to mark the threshold for cluster membership:



The remaining two hyperparameter options can be seen to greatly impact the performance of your DBSCAN clustering algorithm, and show how a shift in one number can greatly influence performance:



As you can see, simply changing the number of minimum points from 19 to 20 adds an additional (incorrect!) cluster to our feature space. Given what you’ve learned about minimum points through this exercise, now tweak both epsilon and minimum points thresholding in your Sci-Kit Learn implementation to achieve the optimal number of clusters.

Activity 2: Compare DBSCAN with K-Means and Hierarchical Clustering

**Scenario:** Your manager is asking you to evaluate different clustering approaches to see which works best for your use case. You need to be able to report back on which clustering method is most relevant based off the data you have available.

**Aim:** Using Sci-Kit Learn implementations, evaluate how each of the clustering methods we’ve learned thus far perform on sample data you generate. You should explore blob-like data features as well as more complex data landscapes such as swiss-roll.

DBSCAN vs K-Means and Hierarchical Clustering

Now that you’ve reached an understanding of how DBSCAN is implemented and how many different hyperparameters you can tweak to drive performance, let’s survey how it compares to the clustering methods we’ve covered in Lesson 1 and 2.

In general, DBSCAN can be seen as a strong contender when it comes to complex, intertwined data. DBSCAN easily performs best compared to K-Means and Hierarchical Clustering when it came to swiss roll data as seen in Activity 2. Going back to the retail store invoice example, one scenario where DBSCAN may outperform the other clustering methods we’ve seen is across purchases in a product category varied by customer type. A somewhat silly example to help convey this can be seen in sales of paper towels in a store. Most household consumers may buy bulk packs of 8-12 paper towel rolls. Commercial resellers may be sold 1 roll samples of paper towels to evaluate whether they want to sell the product in large quantities. This creates a situation where across the dimension of number of paper towel rolls sold, very small sample orders are made by commercial businesses, then medium orders are by residential clients, and then very large orders are made once again by commercial businesses. Once again this is a silly example but hopefully it helps illustrate the scenario where DBSCAN really shines.

Compared to K-Means and Hierarchical Clustering, DBSCAN can be seen as a potentially more efficient since it only has to look at each data point once. Instead of multiple iterations of finding new centroids and evaluating where its nearest neighbors are, once a point has been assigned to a cluster in DBSCAN it does not change cluster membership. The other key difference that DBSCAN and Hierarchical Clustering both share compared to K-Means is not needing to explicitly pass a number of clusters expected at time of creation. This can be extremely helpful when you have no external guidance on how to break your dataset down.

Summary

DBSCAN takes an interesting approach to clustering compared to K-Means and Hierarchical Clustering. While Hierarchical Clustering can in some aspects be seen as an extension of the nearest neighbors approach seen in K-Means, DBSCAN approaches the problem of finding neighbors by applying a notion of density. This can prove extremely beneficial when it comes to highly complex data that is intertwined in complex fashions. While DBSCAN is very powerful it is not infallible and can be seen as potentially overkill depending on what your original data looks like.

Combined with K-Means and Hierarchical Clustering however, DBSCAN completes a strong tool box when it comes to the unsupervised learning task of clustering your data. When faced with any problem in this space it is worthwhile to compare the performance of each method and see which performs best.

Practice Questions

1. The two hyperparameters that primarily drive performance in DBSCAN are neighborhood radius size and minimum number of points threshold.
   1. True
   2. False
2. Which of these clustering methods require specifying the number of clusters prior to runtime?
   1. K-Means
   2. Hierarchical Clustering
   3. DBSCAN
   4. A & B
3. DBSCAN uses solely the neighborhood measure and doesn’t use any distance functions.
   1. True
   2. False
4. Which of these clustering methods only need to cycle through all of your data points just one time?
   1. K-Means
   2. Hierarchical Clustering
   3. DBSCAN
   4. A & B
5. DBSCAN focuses on capturing what notion when it comes to approximating neighbors?
   1. Dilution
   2. Density
   3. Differential
   4. None of the Above
6. The neighborhood in a normal implementation of DBSCAN can be visualized as a circle
   1. True
   2. False
7. The end result of DBSCAN can have many data points assigned to no cluster
   1. True
   2. False
8. Typically, in what scenario will DBSCAN excel as a clustering method?
   1. Discrete blobs of data points
   2. Sandwiched data points where linear relationships are difficult to draw
   3. When there is a lot of noise in your data
   4. B & C
9. You should always default to DBSCAN as the best clustering method and then evaluate K-Means and Hierarchical Clustering only if DBSCAN fails
   1. True
   2. False
10. Neighborhood radius size and minimum number of points will not change as a result of how your data looks – good combinations of the hyperparameters will always be good combinations.
    1. True
    2. False