

# Clustering with DBSCAN

Module 1 | Chapter 3 | Notebook 7


---

In the previous lessons in this chapter you used k-Means to cluster data, but k-Means is not suitable for every data set. For this reason we will now look at another algorithm: DBSCAN. At the end of this Lesson you will have gained an understanding of:

- The weaknesses of k-Means
  - The parameters of DBSCAN
  - Outliers in the context of DBSCAN
- 

## The weaknesses of k-Means

As we learned in the last exercises, k-Means is based on the Euclidean distance between cluster centers and data points. The *within-cluster sum of squares* of `KMeans` and the *silhouette* coefficient we used to validate the clusters are also based on the mean values of Euclidean distances. However, this means that these methods work best for compact, spherical clusters. In two dimensions these are circular and in three dimensions they are spherical. At the beginning of the chapter you saw a sample data set which is very well suited for k-Means:

 Circular clusters

If we calculate the *silhouette* coefficients for this, we only get positive values that are relatively high. Run the following code cell to make sure.

```
In [1]: # import everything we need
import pandas as pd
import numpy as np
from sklearn.cluster import KMeans
from sklearn.metrics import silhouette_samples
import matplotlib.pyplot as plt
import seaborn as sns

# read data
spherical_cluster = pd.read_csv('clustering-data.csv')

# define figure and colors
fig, ax = plt.subplots(figsize=[15,10])
colors = sns.dark_palette("#3399db",4) # seaborn can be used to derive a color palette

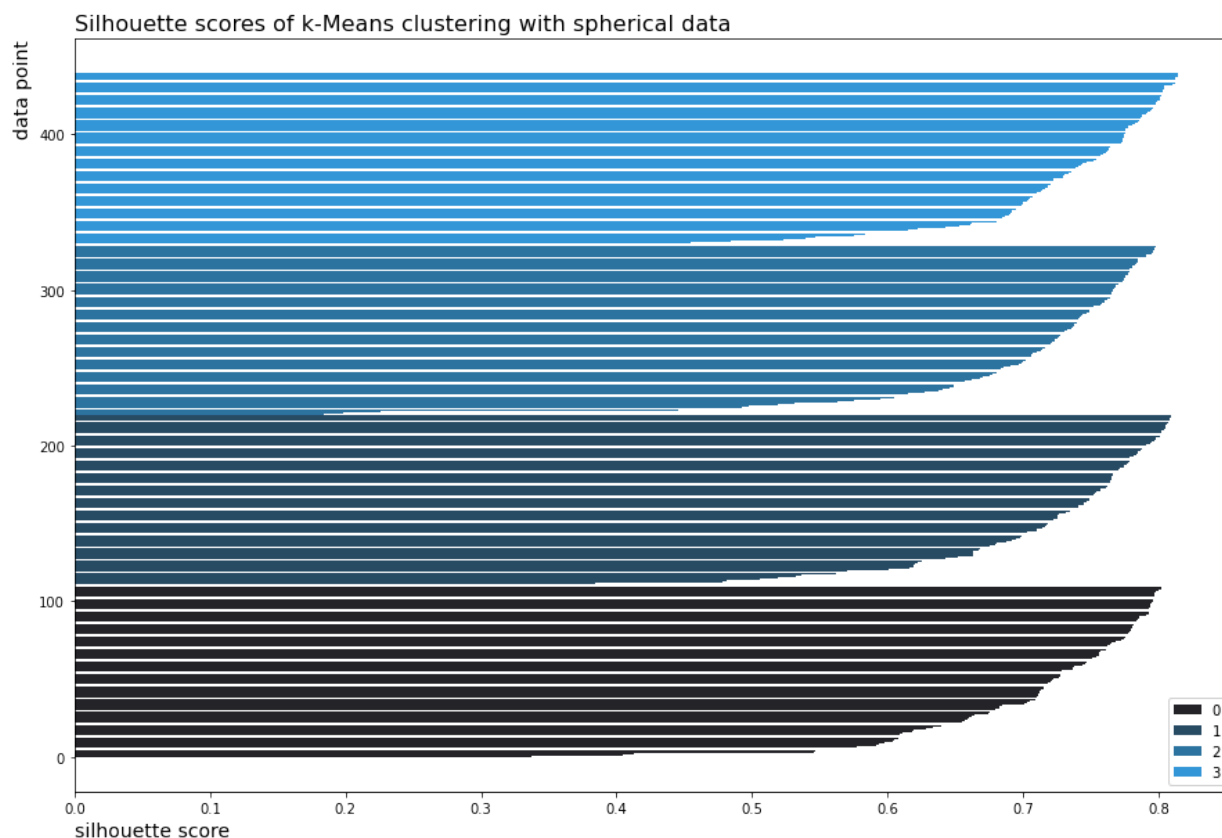
# fit new KMeans with n_clusters
model = KMeans(n_clusters=4, random_state=0)
model.fit(spherical_cluster)
#calculate silhouette score and coefficients
arr_sil = silhouette_samples(spherical_cluster, model.labels_)
```

```

#plot the coefficients
start = 0
end = 0
for c in range(4):
    mask = model.labels_ == c
    sv_len = len(arr_sil[mask])
    sv_sorted = np.sort(arr_sil[mask])
    end = end + sv_len
    ax.barh([i for i in range(start, end)], width=sv_sorted, label=c, color=colors[c])
    start = end
ax.set_title('Silhouette scores of k-Means clustering with spherical data', size=16, l
ax.set_xlabel(xlabel='silhouette score',
               position=[0, 0],
               horizontalalignment='left',
               size=14)
ax.set_ylabel(ylabel='data point',
               position=[0, 1],
               horizontalalignment='right',
               size=14)
ax.legend()

```

Out[1]: <matplotlib.legend.Legend at 0x7f7ea6e68f70>



This is due to the fact that the clusters are nicely separated and circular. What about other data? For example, look at the following data points:



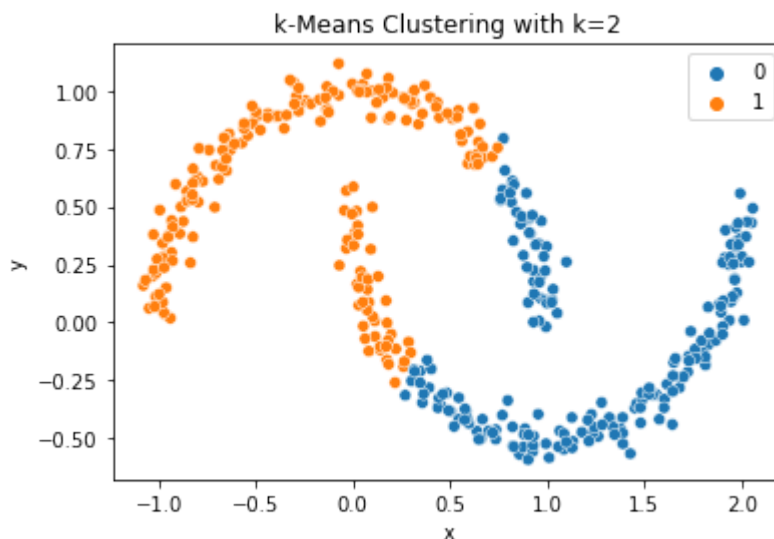
You can see immediately that these are two semicircular clusters. How good do you think k-Means is at identifying them? Try it out. This data is all located in the file *clustering-data-2.csv*. The columns are called 'x' and 'y'. Import it as `other_cluster`, instantiate `KMeans`

with the correct number of clusters and use `seaborn` for a scatter plot where the color depends on the cluster membership.

```
In [2]: other_cluster = pd.read_csv('clustering-data-2.csv')

model = KMeans(n_clusters=2, random_state = 0)
model.fit(other_cluster)

# plot the data:
import seaborn as sns
sns.scatterplot(data = other_cluster, x='x', y='y', hue=model.labels_).set_title("k-Means Clustering with k=2")
```



We end up with the following results:

 k-Means performs badly

Dividing the data points into clusters visually is very easy. However, k-Means has significant problems doing this. This is because the data no longer fits well into K-Means' circular worldview. We'll look at what the *silhouette* method produces when values are correctly assigned, later in the lesson.

Now you know several of k-Means' weaknesses. These are all related to the fact that k-Means only assesses the distances between data points and cluster centers:

1. k-Means has problems with values on different scales
2. k-Means can't determine the number of clusters itself
3. k-Means can't deal with clusters whose shapes differ widely from that of a circle or sphere

In the last few lessons, you learned how to deal with the first two weaknesses by standardizing the data and using various scores for the number of clusters. But if you come across the third weak point with your data, you might find that another algorithm will be more helpful. We'll look at this next.

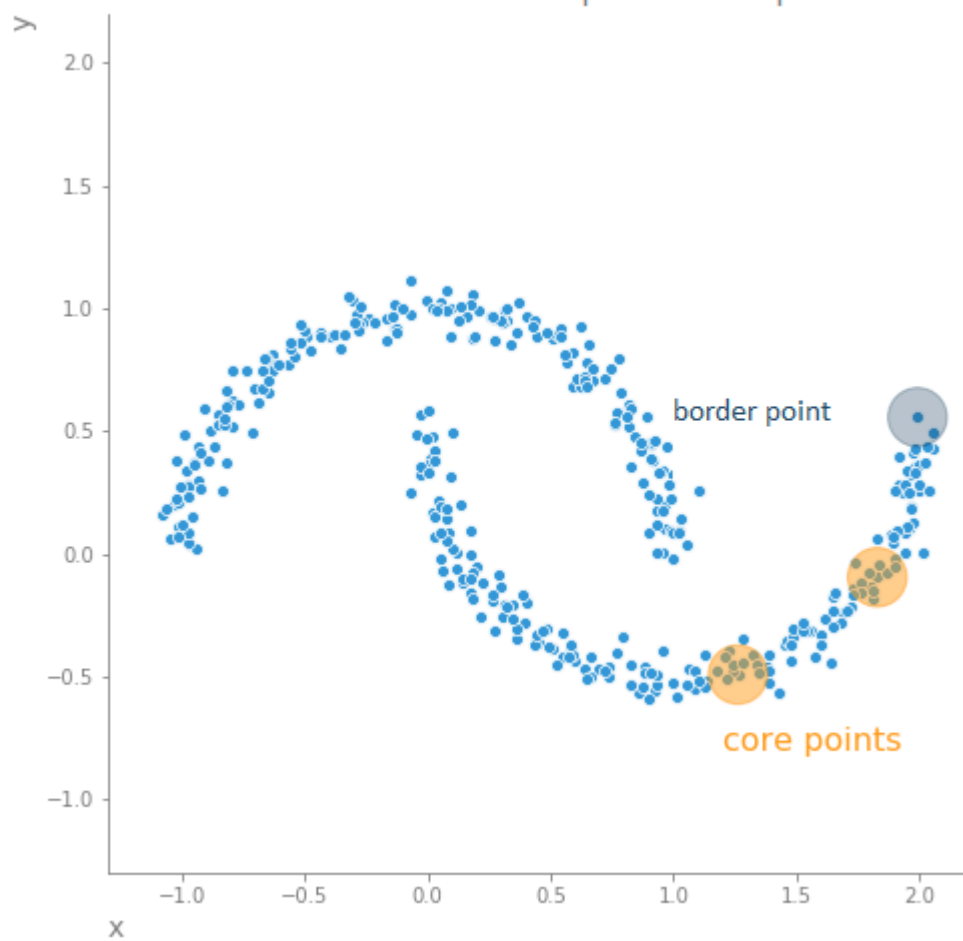
**Congratulations:** You have learned more about k-Means' weaknesses. It is important to keep this in mind when clustering.

## DBSCAN parameters

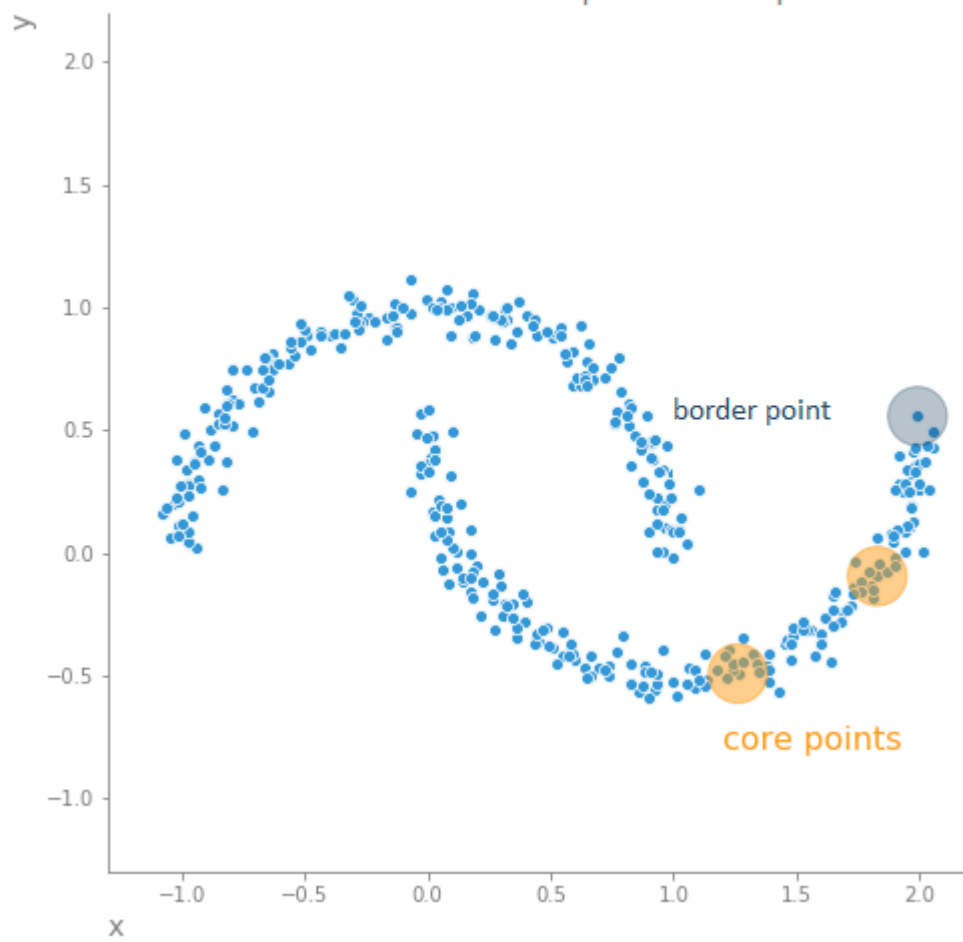
It is easy for us to recognize the semicircular groups as clusters, because we immediately see that the data points are close together. DBSCAN is based on the same principle. DBSCAN stands for *Density-Based Spatial Clustering of Applications with Noise*. DBSCAN works as follows:

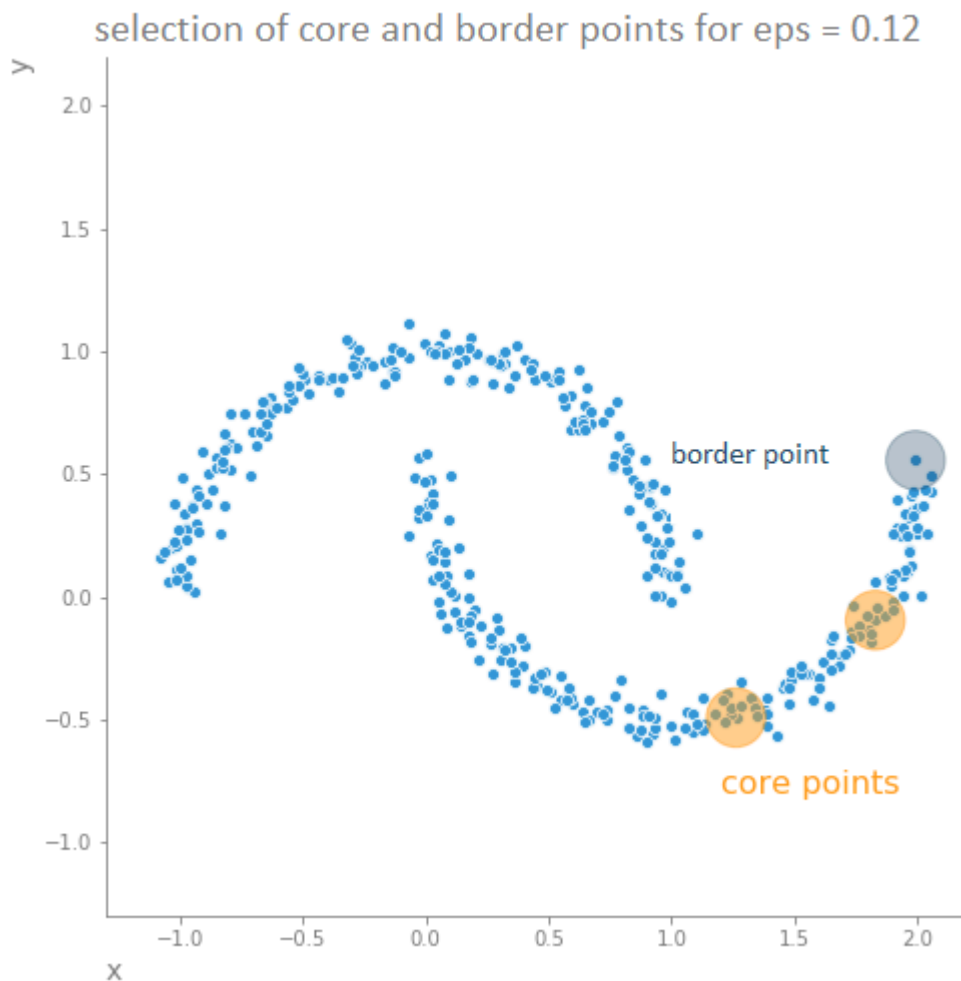
- A maximum distance `eps` is defined.
- All points that have at least `min_samples` of neighbors within this radius, are what are known as core points of a cluster
- Data points that are close enough to core points but don't have enough neighbors themselves are called border points
- Neighboring core and directly reachable points are assigned to the same cluster
- Data points that lie outside a cluster's neighborhood that don't have enough neighbors themselves to form their own cluster, are known as outliers or noise points

selection of core and border points for  $\text{eps} = 0.12$



selection of core and border points for  $\text{eps} = 0.12$





The following figure illustrates this for our data set. The maximum distance `eps` here is `0.12`.

 Core points and directly reachable points

The circles have a radius of `eps=0.12`. The boundary point has only one neighbor, but the core points have several. The circles in the figure only show a very small selection of boundary and core points.

In this example, `min_samples=4`. This means that 4 points are sufficient to form a cluster. There are only 2 points in the blue circle. The corresponding point is therefore not a core point (4 points would have to be in the circle for this to be the case).

If we only need 4 points in the radius to form a new cluster, it sounds like we might end up with a lot of different clusters. In this case, however, each core point of a semicircle is adjacent to another core point. They then merge and form a shared cluster. The neighbor of the boundary point is part of the cluster, so the boundary point is also included. However, the selected radius is small enough that the points in the two different semicircles are not neighbors. This means that a total of two different clusters are identified.

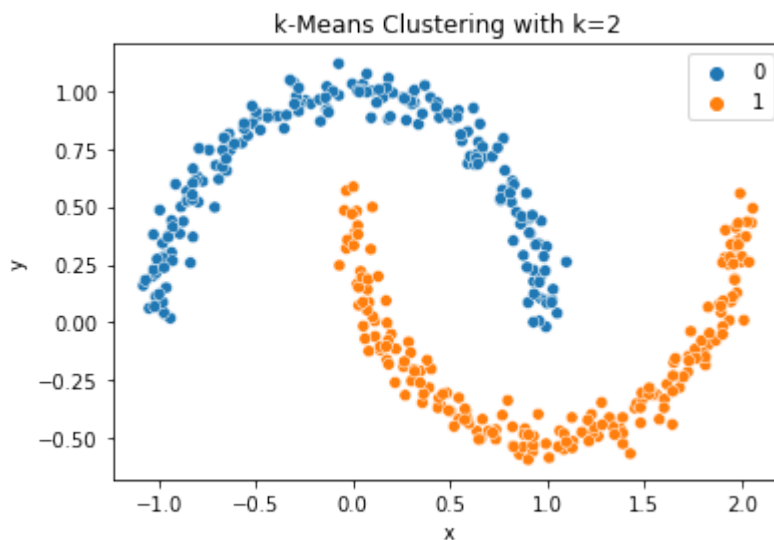
The parameter settings `eps=0.12` and `min_samples=4` result in all the points to be assigned to one of the two clusters. Now let's try that out.

Import `DBSCAN` directly from `sklearn.cluster`. Instantiate it as `model_db` with the parameters mentioned above, and fit it to the data. Just like `KMeans`, `DBSCAN` has the attribute `my_model.labels_`. Use this to create a scatterplot where the clusters have different colors. Attention, `DBSCAN` does not have the a `random_state` parameter.


Tip: You don't need to standardize the data, because the x and y values are on very similar scales.

```
In [9]: from sklearn.cluster import DBSCAN
model_db = DBSCAN(eps=0.12, min_samples=4)
model_db.fit(other_cluster)
model_db.labels_

sns.scatterplot(data = other_cluster, x='x', y='y', hue=model_db.labels_).set_title("k
```



With these parameters, `DBSCAN` recognizes the clusters very well:

 Successful cluster analysis with DBSCAN

But how do you now decide which values are suitable for `eps` and `min_samples`? As with the number of clusters in `KMeans` you have to estimate it. When choosing a minimum distance, it can be helpful to visualize the distances in the data set. Calculate them. Import `euclidean_distances` from `sklearn.metrics`. Pass the `DataFrame` with the coordinates to this function. It then returns a two-dimensional array with the distances between all data points. The row and column index numbers represent the row indices of the corresponding data points in the `DataFrame`. Store these distances in the variable `arr_dist`.

```
In [11]: from sklearn.metrics import euclidean_distances
arr_dist = euclidean_distances(other_cluster)
```

Now we have the distances between all the data points. In order to better estimate `eps`, we are mainly interested in the nearest neighbors. We get the corresponding distances if we sort the values in the rows of `arr_dist` by size. You can achieve this by using the function `np.sort()` with the parameter `axis=1`. `axis=1` tells `numpy` that we want to sort the rows.



`axis=0` would indicate we want to sort the columns. Sort `arr_dist` and save the result in `arr_dist_sorted`.

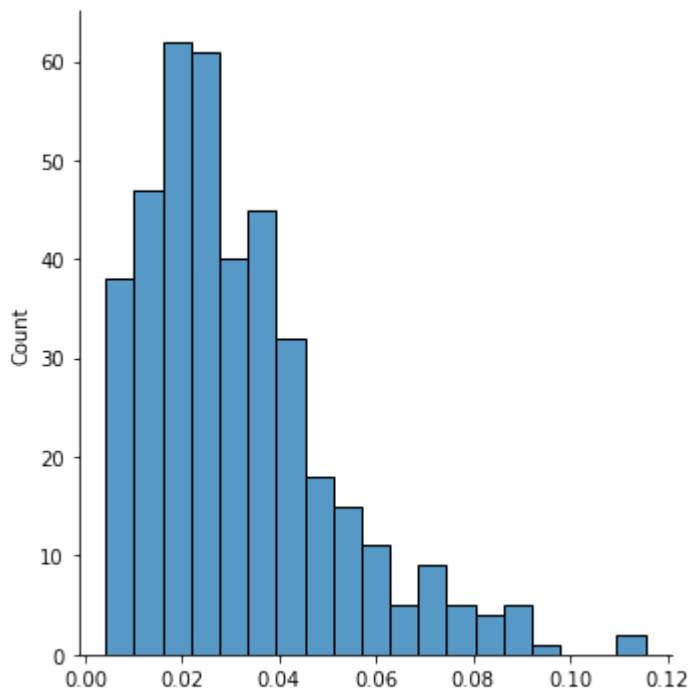
```
In [13]: arr_dist_sorted = np.sort(arr_dist, axis=1)
```

The first value of each row in the sorted array is `0`. This is the distance from the point to itself. The next largest value is the distance to the nearest neighbor. Generate a histogram from the distances to the nearest neighbors. Use for example `sns.displot()` (alternatively you can use `sns.distplot()` in which case you will see a `FutureWarning`). Pass this function a one-dimensional array with the distances to the nearest neighbors and the parameter `kde = False` to get a simple histogram.

```
In [16]: # arr_dist_sorted

sns.displot(arr_dist_sorted[:,1], kde = False )
```

```
Out[16]: <seaborn.axisgrid.FacetGrid at 0x7f7ea561b5e0>
```



The distances follow a right-skewed distribution. The largest distance between the nearest neighbors is just under 0.12. So with this value we can ensure that each point has at least one neighbor. This is a good choice for this data set, since we know that there are only two clusters and no outliers.

You can follow this rule of thumb when choosing `min_samples`:

$$\text{min\_samples} = 2 \cdot \text{number\_of\_dimensions}$$

So `min_samples` is about twice the number of columns in the data. So in our example it's 4, because we only have the columns `'x'` and `'y'` in the data. But the exact choice depends on

the result. A small value can result in the data being split into a large number of clusters. If the value is large, you can end up with a lot of outliers.

We have divided the semicircular clusters very well with `DBSCAN`. What *silhouette* coefficients would you expect now? Use the `silhouette_samples()` function to calculate them. Store the result as `arr_sil`.

```
In [17]: arr_sil = silhouette_samples(other_cluster, model_db.labels_)
```

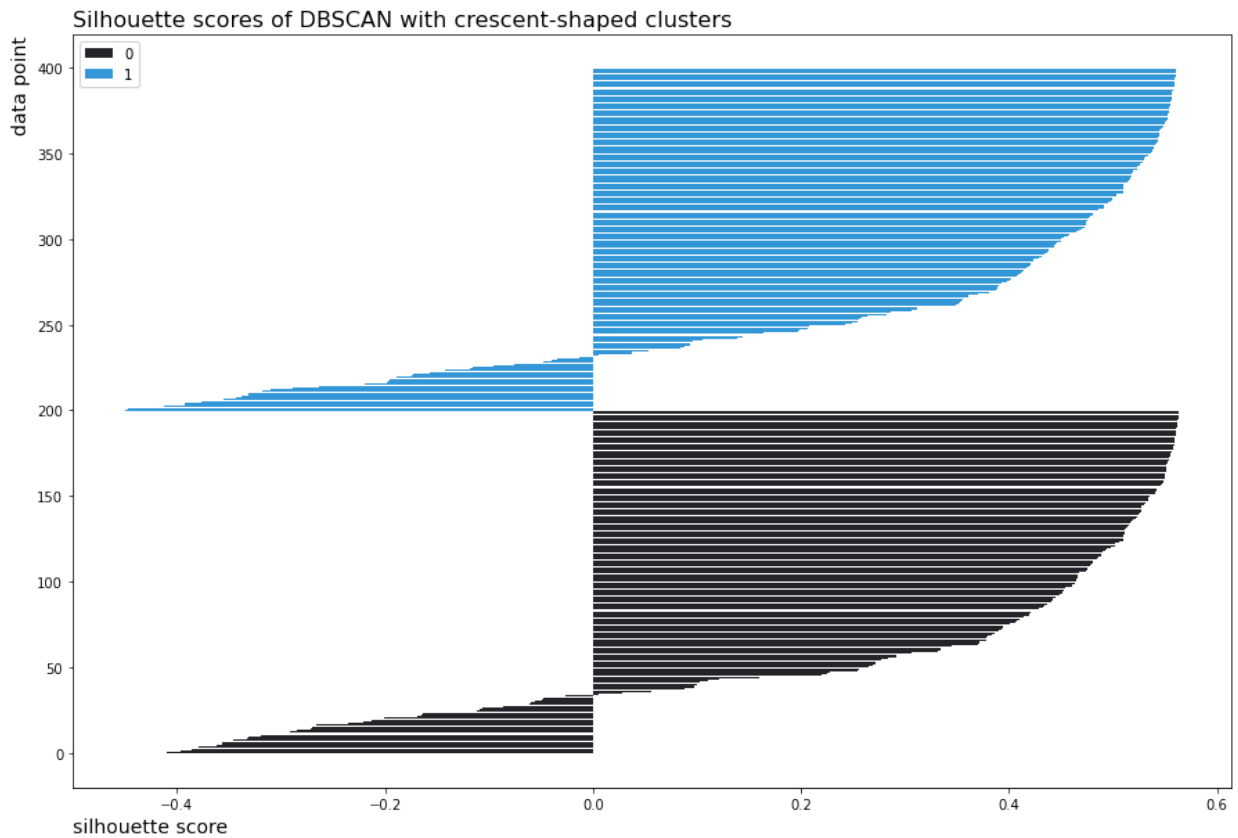
Executing the following code cell will plot the coefficients. Is this the result you were hoping for?

```
In [18]: # define figure and colors
fig, ax = plt.subplots(figsize=[15, 10])
colors = sns.dark_palette("#3399db", 2) # seaborn can be used to derive a color palette

# plot the coefficients
start = 0 # start and end are needed to plot one group above the other
end = 0
for cluster in range(2):
    mask = model_db.labels_ == cluster # create a mask to select data points from the
    sv_len = len(arr_sil[mask]) # the length is needed to increase end for plotting the
    sv_sorted = np.sort(arr_sil[mask]) # sort the silhouette scores within each cluster
    end = end + sv_len # increase end: be able to get a range with the length of this
    ax.barh(range(start, end), width=sv_sorted, label=cluster, color=colors[cluster])
    start = end # increase start: the next cluster will be plotted above this one

# set title and labels of the plot
ax.set_title('Silhouette scores of DBSCAN with crescent-shaped clusters', size=16, loc='top')
ax.set_xlabel(xlabel='silhouette score',
              position=[0, 0],
              horizontalalignment='left',
              size=14)
ax.set_ylabel(ylabel='data point',
              position=[0, 1],
              horizontalalignment='right',
              size=14)
ax.legend()
```

```
Out[18]: <matplotlib.legend.Legend at 0x7f7ea42ed820>
```



A lot of the data points have a negative score. But we know that they were correctly classified. The *silhouette* method is not suitable for this data set, because the clusters are not circular and the distance between them is very small in some places.

**Congratulations:** You've performed your first cluster analysis with DBSCAN. Now you have added a second clustering algorithm to your repertoire. DBSCAN is a very good addition to k-Means, because they both proceed very differently. Can DBSCAN also be applied to customer data?

## Clustering customer data

**Scenario:** You work for an online retailer that sells various gift items. The company is mainly aimed at business customers. Your customer base is also very diverse. In order to better address the people using your online platform, the marketing department would like to get more insight into customer behavior. The customer base should be divided into groups based on orders they have placed to date.

You have been provided with data containing all orders and cancellations in a one year period, in order to gain some initial insights. You have already prepared and saved the data in *customer\_data\_prepared.p*.

First of all, let's start by importing and standardizing the data. Run the following cell to do this.

```
In [21]: from sklearn.metrics import euclidean_distances
import numpy as np
```

```
import matplotlib.pyplot as plt
from sklearn.cluster import DBSCAN
```

```
In [22]: # read the data
import pandas as pd
df_customers = pd.read_pickle('customer_data_prepared.p')

# standardize the values
from sklearn.preprocessing import StandardScaler
standardizer = StandardScaler()
standardizer.fit(df_customers)
arr_customers_std = standardizer.transform(df_customers)

df_customers.head()
```

```
Out[22]:
```

	InvoiceNo	Revenue	Quantity	RevenueMean	QuantityMean	PriceMean	DaysBetweenIn
CustomerID							
12347.0	7	4310.00	2458	615.714286	351.142857	1.753458	
12348.0	4	1437.24	2332	359.310000	583.000000	0.616312	
12349.0	1	1457.55	630	1457.550000	630.000000	2.313571	
12353.0	1	89.00	20	89.000000	20.000000	4.450000	
12354.0	1	1079.40	530	1079.400000	530.000000	2.036604	

Now let's get an overview of the distances between the data points. Calculate the Euclidean distances of `arr_customers_std`. Store these as `arr_dist`. Sort the rows and store them as `arr_dist_sorted`.

```
In [23]: arr_dist = euclidean_distances( arr_customers_std )
arr_dist_sorted = np.sort(arr_dist, axis=1)
```

You can often use what's called a *k-distance plot* to help estimate the distance. Here the distance to the *k*th nearest neighbor is displayed for each data point. The idea behind this is that data points in a cluster have many neighbors. The distance therefore only changes slightly from the nearest neighbor to the nearest but one. But if we look at a point outside the cluster, this is no longer the case. The neighbors are far away and the distance increases from one to the next.

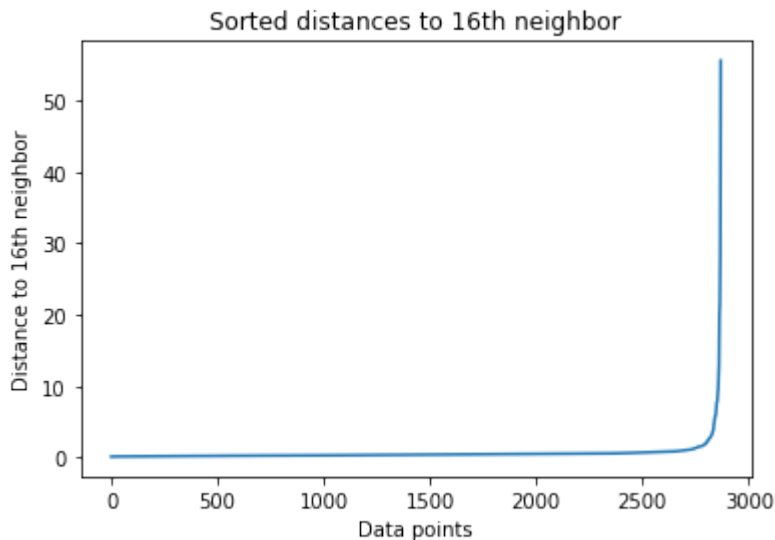
In the diagram, we would therefore expect that all points in clusters to have a small distance to their *k*th neighbor. However, this distance rises sharply for data points outside the cluster. So for `eps` you choose a value which lies just before the sudden increase. *k* corresponds to `min_samples`. Usually the images are relatively stable for small deviations in *k*.

Now draw a *k-distance plot*. It's important to know that the zeroth neighbor is always the data point itself. We can recognize this by the fact that this only has distances with the value zero. Therefore for the question about the 16th neighbor, choose the 17th value (column index 16) of each row of `arr_dist_sorted`. Our data has 8 columns, so we want to have this value in

particular (twice the number of dimensions of the data points). Sort these distances and put them in a diagram.

```
In [24]: fig, ax = plt.subplots()
ax.plot(range(len(arr_dist_sorted)), np.sort(arr_dist_sorted[:, 16]))
ax.set(title='Sorted distances to 16th neighbor', xlabel='Data points', ylabel='Distar
```

```
Out[24]: [Text(0.5, 1.0, 'Sorted distances to 16th neighbor'),
Text(0.5, 0, 'Data points'),
Text(0, 0.5, 'Distance to 16th neighbor')]
```



The distances are very flat at first and then rise sharply. We've zoomed in on the bend here, so that you can read the value more easily.

 Distances to the 16th Neighbor

So there are about 2800 data points that have 16 very close neighbors. Even the 16th. Neighbor is never more than 3 units away from the data point. In addition, there are about 100 data points, where the 16th neighbor is further away. So they don't appear to have a dense neighborhood and are possibly outliers, i.e. they don't really belong to a cluster.

The rise begins around the 2800th data point of the sorted array. You can also select a value before or after. As a general rule, we recommend an iterative procedure where you check the results and adjust the parameters afterwards. For example, what is the value of the sorted distances of the 16th neighbor at position 2798?

```
In [31]: np.sort(arr_dist_sorted[:, 16])[2798]
```

```
Out[31]: 2.056243105035968
```

We get a value around 2.056. We can now use this with the `eps` parameter. For `min_samples` we choose 16. Instantiate DBSCAN with these values and fit it to `arr_customers_std`. How many different clusters do you get? The cluster names are also located in the `my_model.labels_` attribute for `DBSCAN`.

```
In [32]: model = DBSCAN(eps=2.056, min_samples=16)
model.fit(arr_customers_std)
np.unique(model.labels_)
```

```
Out[32]: array([-1,  0])
```

We only get two different clusters. They have the designations -1 and 0. However, the value -1 in DBSCAN does not stand for a cluster, but for outliers that are not assigned to a cluster. How many outliers do we have?

```
In [38]: sum(model.labels_ == -1)
```

```
Out[38]: 45
```

You should get 45 outliers. This is a similar size as the number of points in the small clusters that we got with k-Means. To get only one cluster in when dividing customers into segments may feel a little unsatisfactory. Now we could change the parameters a little bit and check if this has an influence on the clustering. But probably not, because k-Means already gave us a similar picture. With that algorithm, we ended up with three large clusters, but they were only slightly different in a few points.

We could now use Principal Component Analysis (PCA) to look at the results of the cluster analysis and reduce the data set to two dimensions. You'll learn how this works in detail in the next chapter. For now you can simply run the following cell. It creates a PCA with 2 components - `n_components=2` and creates an array called `cluster_plot_arr` where the transformed values are stored. It then plots the values of this array in a scatter plot and uses `model.labels_` to color the dots according to their cluster assignment.

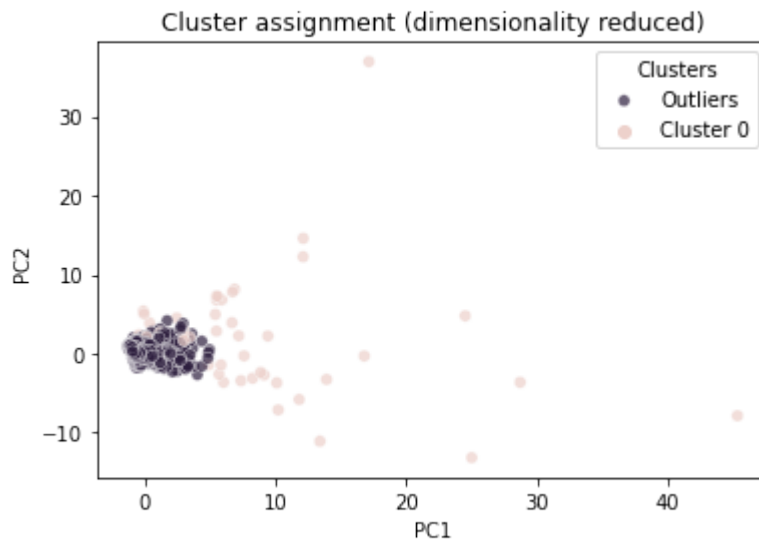
```
In [41]: from sklearn.decomposition import PCA
import seaborn as sns

#pca
pca = PCA(n_components=2)
cluster_plot_arr = pca.fit_transform(arr_customers_std)

#plot
ax = sns.scatterplot(x=cluster_plot_arr[:,0],
                    y=cluster_plot_arr[:,1],
                    alpha=0.7,
                    hue=model.labels_)

#style plot
ax.set(xlabel='PC1',
      ylabel='PC2',
      title='Cluster assignment (dimensionality reduced)')

#configure legend
import matplotlib.pyplot as plt
plt.legend(title='Clusters',
          labels=['Outliers', 'Cluster 0']);
```



If we only look only at the features we generated from the orders, the vast majority of customers are very similar. Only a few of them differ significantly. These form a premium segment, as we have already seen in the analysis with k-Means. You can find these outliers quite easily with DBSCAN. We can use k-Means to create segments based on small differences. It depends on each separate case, whether this is enough. In our example it might be worth creating other features from the data. For example, we didn't consider which product categories our customers prefer in each case. It could be worthwhile considering these characteristics as well.

**Congratulations:** Now you know the `KMeans` and `DBSCAN` algorithms. You know how to set values for their parameters. You have also seen how the data must be made up for these algorithms to work well.

For the customer analysis you applied 2 clustering algorithms which follow different methods, but produce similar results with this data set. Your employer thanks you for your assessment of the cluster analysis. He decides that the next analysis will be a much larger project, which should include even more of the customers' features.

**Remember:**

- `DBSCAN` forms clusters based on the density of data points
- $2 \cdot \text{number\_of\_dimensions}$  is a good guiding value for `min_samples`
- Estimate `eps` with a *k-distance plot*, which shows the distances to the nearest neighbors.
- `KMeans` and the silhouette method work best with spherical clusters

---

Do you have any questions about this exercise? Look in the [forum](#) to see if they have already been discussed.

---

Found a mistake? Contact Support at [support@stackfuel.com](mailto:support@stackfuel.com).

---

The data was published here first: Daqing Chen, Sai Liang Sain, and Kun Guo, Data mining for the online retail industry: A case study of RFM model-based customer segmentation using data mining, *Journal of Database Marketing and Customer Strategy Management*, Vol. 19, No. 3, pp. 197-208, 2012