

Clustering Recap

Module 1 | Chapter 4 | Notebook 1

In this notebook, we'll recap the most important things from the previous chapter. This is all repetition. If you discover a lot of new things here, we recommend that you take a look back at Chapter 3. The relevant lessons for each section are clearly marked.

Clustering with k-Means

K-Means is a clustering algorithm, i.e. unsupervised learning. The steps we already know from regression and classification problems can also be followed for clustering. However, the difference here is that we do not have target data that we want to predict. If you take this into account, you end up with the following steps, see *Clustering with k-Means (Module 1 Chapter 3)*:

1. Select model type
2. Instantiate a model with certain settings known as hyperparameters
3. Organize data into a feature matrix
4. Model fitting

The model type we use is `KMeans` from the `sklearn.cluster` module.

```
In [1]: from sklearn.cluster import KMeans
```

When clustering the data, k-Means proceeds as follows:

1. Create a number of k points that serve as cluster centers
2. Assign the closest cluster center to each point
3. Calculate the mean value of the data points for each cluster and move the cluster center there
4. Repeat steps 2 and 3 until the cluster centers no longer change significantly

`KMeans` uses the following hyperparameters:

- `n_cluster` : Number of clusters that the data points should be grouped into. This is the most important parameter.
- `n_init` : Number of iterations (excluding `NaN`). Then the best cluster assignment is used.
- `random_state` : Number representing the state of the random number generator. Makes it possible to reproduce the results exactly.

The number you use for `n_cluster` is the deciding factor for how well the clusters are assigned. You can use the elbow method, see *k-Means - Determining the Number of Clusters (Module 1 Chapter 3)*, to help you find a suitable value for this parameter. First you have to import and standardize the data.

Important: Standardizing data is an important step. `KMeans` is based on the Euclidean distances, which depend on the values of the features. If the features are on different scales, they are not considered as having the same importance by `KMeans`. We can avoid this problem by standardizing the data.

```
In [2]: # import pandas and read in the data
import pandas as pd
df = pd.read_csv('clustering-data.csv')

# import, instantiate and fit StandardScaler
from sklearn.preprocessing import StandardScaler
scaler = StandardScaler()
scaler.fit(df)
# standardize the data
features = scaler.transform(df)
```

With the *elbow method*, we use the *within-cluster sum of squares*, i.e. the average sum of the squared distances of the data points to the corresponding cluster centers for each cluster. You can calculate them with the `my_model.score(features)` method. We vary `n_clusters` and look for a bend in the resulting line. If the *within-cluster sum of squares* no longer improves significantly as the number of clusters increases, then this is a good value for `n_clusters`.

```
In [17]: wcss = [] # create a list to hold the within-cluster sum of squares values
for number_of_clusters in range(1, 10): # we have to start with at least one cluster.
    model = KMeans(n_clusters=number_of_clusters, random_state=0) # instantiate KMeans
    model.fit(features) # search for good cluster centers
    wcss.append(model.score(features)) # calculate and append the within-cluster sum
```

Now we'll visualize the values to find the bend.

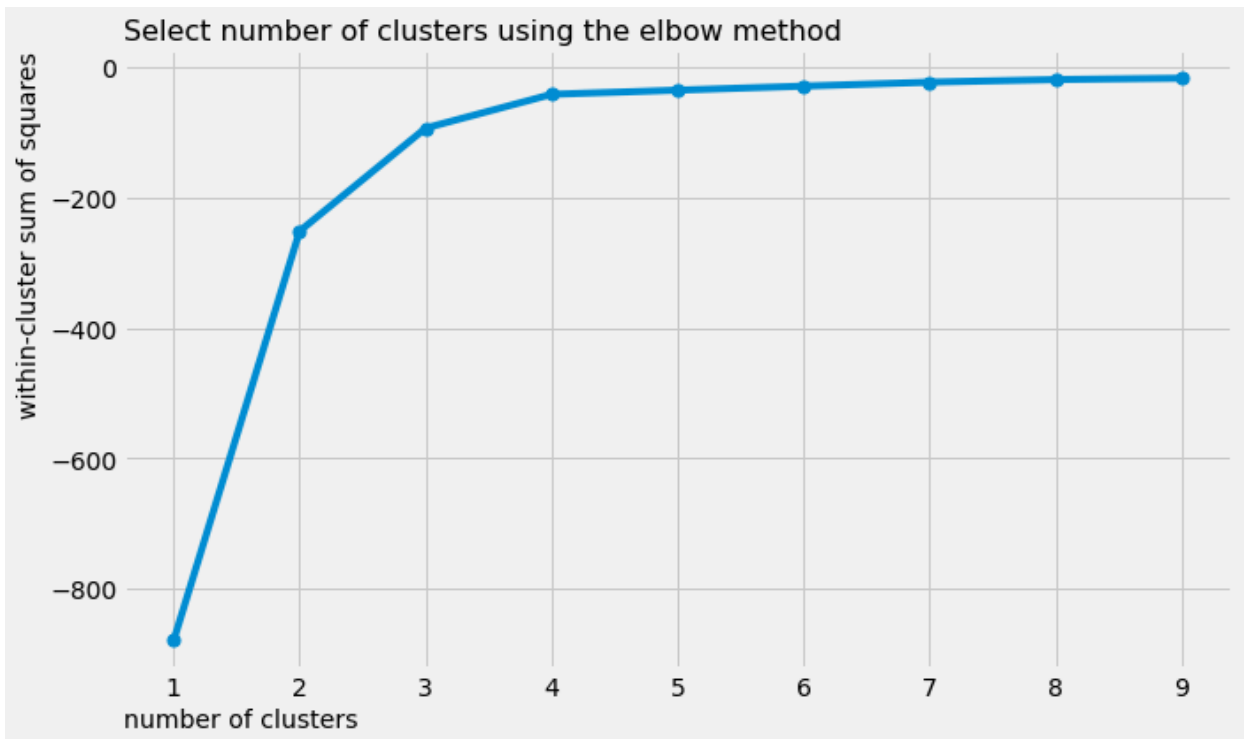
```
In [18]: # visualize the within-cluster sum of squares values

# import and set everything we need for a nice plot
import matplotlib.pyplot as plt
plt.style.use('fivethirtyeight') # use styling for a nice plot

# create figure and axis and plot the scores
fig, ax = plt.subplots(figsize=[10, 6])
ax.plot(range(1, 10), wcss, marker='o', markersize=7) # use big markers to see which

# set title and labels of the plot
ax.set_title('Select number of clusters using the elbow method', size=16, loc='left')
ax.set_xlabel(xlabel='number of clusters',
              position=[0, 0],
              horizontalalignment='left',
              size=14)
ax.set_ylabel(ylabel='within-cluster sum of squares',
              position=[0, 1],
```

```
horizontalalignment='right',  
size=14);
```



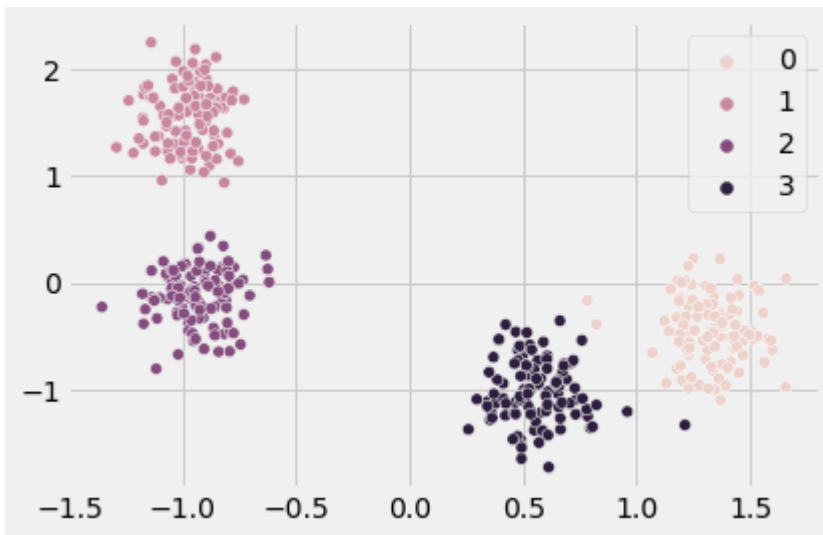
We can see that the *within-cluster sum of squares* changes very little after 4 clusters. Now we can instantiate `KMeans` with the correct number of clusters and fit it to the data.

```
In [19]: model = KMeans(n_clusters=4, random_state=0) # instantiate KMeans with 4 clusters. Use  
model.fit(features) # fit k-means to the data
```

```
Out[19]: KMeans(n_clusters=4, random_state=0)
```

`model` now contains the names of the clusters in the attribute `model.labels_`. We can now use these to represent the clusters visually. Since the data only has 2 dimensions, we can present it directly as a scatter plot.

```
In [6]: import seaborn as sns # import seaborn to use scatterplot  
  
ax = sns.scatterplot(x=features[:, 0], y=features[:, 1], hue=model.labels_) # plot the
```



We can see that the cluster analysis worked well and that we do actually have four clusters. Now we can add the cluster centers to the figure. Their positions are stored in the `model.cluster_centers_` attribute. However, these refer to the standardized data. We therefore have to convert them back with our `scaler`. You can use `my_scaler.inverse_transform()` to do this.

```
In [7]: # plot the data points and save returned axis as ax
ax = sns.scatterplot(x=df.loc[:, 'x'],
                    y=df.loc[:, 'y'],
                    hue=model.labels_)

# inverse transform cluster centers
cluster_centers = scaler.inverse_transform(model.cluster_centers_)

# plot the cluster centers as X with size 200
sns.scatterplot(x=cluster_centers[:, 0],
                y=cluster_centers[:, 1],
                marker='X',
                s=200,
                ax=ax)
```

Out[7]: <AxesSubplot:xlabel='x', ylabel='y'>



As you would expect, the cluster centres, or centroids, are located in the middles of the data point clouds.

If the data has more than two dimensions, it becomes difficult to estimate its quality just by using a scatterplot. The silhouette method comes in handy here. You calculate the silhouette coefficient for each data point as follows:

$$\mathrm{silhouette_coef}(x_i) = \frac{\mathrm{mean_dist_cc}(x_i) - \mathrm{mean_dist_mc}(x_i)}{\max(\mathrm{mean_dist_cc}(x_i), \mathrm{mean_dist_mc}(x_i))}$$

Where $\mathrm{mean_dist_cc}$ is the average distance to the points of the closest cluster. $\mathrm{mean_dist_mc}$ is the average distance to the points in the same cluster (referred to as *my cluster*). The maximum function in the divider only selects the maximum of the two average distances.

The coefficient can take values in the range from -1 to 1 and can be interpreted as follows:

- If $\mathrm{silhouette_coef}$ is positive, the point is closer to its own cluster. The higher the value, the more typical the point is for the cluster.
- If $\mathrm{silhouette_coef}$ is negative, the point is further away from its own cluster. It could have been assigned to the wrong cluster.
- A score of close to zero implies that the data point is located equidistant between the clusters. This may be because the clusters overlap.

Note that the position of the centroids is not directly reflected in the *silhouette* coefficient.

To calculate the silhouette coefficients of all the data points, `sklearn.metrics` includes the `silhouette_samples` function. We'll calculate them and store them as an array named `arr_sil`.

```
In [8]: from sklearn.metrics import silhouette_samples # import silhouette_samples to calculate
arr_sil = silhouette_samples(X=features, labels=model.labels_) # calculate the coefficient
```

Now let's display the silhouette coefficients as a bar chart. We'll rearrange the bars so that the points of a cluster are on top of each other and have the same color.

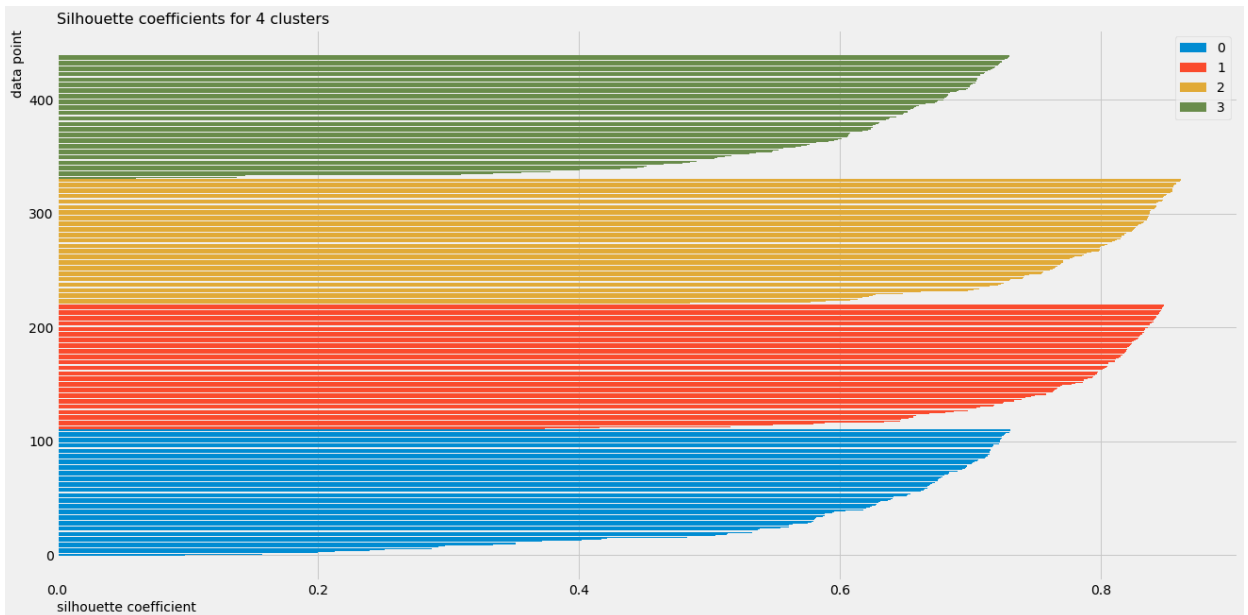
```
In [9]: import numpy as np # import numpy for sorting with np.sort()

fig, ax = plt.subplots(figsize=[20, 10]) # use big plot to be able to spot the tiny bars

start = 0 # we need the variable to plot the clusters above each other
for cluster in range(4): # iterate through every one of the 4 clusters
    mask = model.labels_ == cluster # create mask to select only data points from the cluster
    sv_sorted = np.sort(arr_sil[mask]) # sort the values to get a better feeling for the distribution
    ax.barh(range(start, start + sum(mask)), width=sv_sorted, label=cluster) # plot the bars
    start = start + sum(mask) # increase start by the number of data points in the cluster

# set title and labels
ax.set_title('Silhouette coefficients for 4 clusters', size=16, loc='left')
```

```
ax.set_xlabel(xlabel='silhouette coefficient',
              position=[0, 0],
              horizontalalignment='left',
              size=14)
ax.set_ylabel(ylabel='data point',
              position=[0, 1],
              horizontalalignment='right',
              size=14)
ax.legend();
```



Most of the silhouette coefficients are relatively high. We don't have any negative values. This shows us that the clusters are relatively well isolated from each other. Some values are close to 0. These points are then located between clusters. This is especially the case with cluster 0 and 3.

We can calculate the average coefficient to combine the cluster quality in a single number.

`sklearn.metrics` offers us the function `silhouette_score`. What value do we get?

```
In [10]: from sklearn.metrics import silhouette_score
          silhouette_score(X=features, labels=model.labels_)
```

```
Out[10]: 0.6913743679472646
```

An average silhouette coefficient of 0.69 is very good. You have now recapped two quality metrics for evaluating clusters:

The *within-cluster sum of squares* only takes the distance of each data point to its own cluster center into account. The distance to other clusters is ignored. It is not standardized. So if the clusters contain more data points, this value also increases. Therefore the value of the *within-cluster sum of squares* is not suitable for comparing different data sets. We should only use it to determine the number of clusters with the *elbow method*.

With the *silhouette* method, the distance to the cluster center is not directly included in the calculation. Instead, this metric focuses on the distance between the data points of one cluster and the nearest cluster. If the distances between the clusters increase, the average *silhouette*

coefficient increases. It is also standardized to a value between -1 and 1. This value can be used to compare the structure of different data sets. The higher the average *silhouette* coefficient, the more isolated the clusters making up the data set are. Furthermore, you can use the *silhouette* coefficients for the individual data points to check whether a specific point has been well assigned.

Congratulations: You have recapped how to cluster data with k-Means. To do this, you used the elbow method and the within-cluster sum of squares to evaluate the most important hyperparameter `n_clusters`. With silhouette coefficients, we can examine how clusters are made up even more precisely. Next, we'll take another quick look at the clustering algorithm `DBSCAN`.

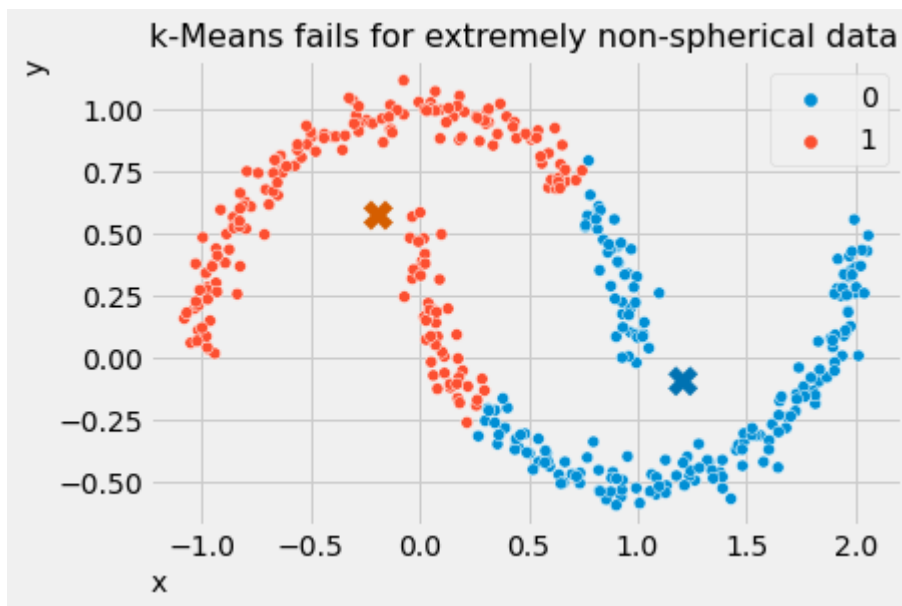
Clustering with DBSCAN

k-Means works best with spherical clusters. This means clusters where the Euclidean distances are as small as possible in all directions. However, the data isn't always shaped like this. The following example shows how k-Means copes with data that has a significantly different structure:

```
In [11]: df_non_spherical = pd.read_csv('clustering-data-2.csv') # read crescent shaped data

model = KMeans(n_clusters=2, random_state=0) # instantiate k-means with 2 clusters
model.fit(df_non_spherical) # fit the model


# plot the data:
ax = sns.scatterplot(data=df_non_spherical, x='x', y='y', hue=model.labels_)
plt.scatter(x=model.cluster_centers[:, 0],
            y=model.cluster_centers[:, 1],
            color=['#0072B2', '#D55E00'],
            marker='X',
            s=200)
# set title and labels
ax.set_title('k-Means fails for extremely non-spherical data', size=16, loc='left')
ax.set_xlabel(xlabel='x',
              position=[0, 0],
              horizontalalignment='left',
              size=14)
ax.set_ylabel(ylabel='y',
              position=[0, 1],
              horizontalalignment='right',
              size=14)
ax.legend();
```



You can see that k-Means has great difficulty in organizing the data correctly here. This is because it is not relevant here how close the data points are to the cluster center or all the other data points in the cluster. Instead, what counts here is that the nearest neighbors are not too far away and belong to the same cluster.

For this reason, it's good to know an alternative to k-Means. You added `DBSCAN` to your repertoire in the last chapter, see *Clustering with DBSCAN* (Chapter 3).

`DBSCAN` is also included in `sklearn.cluster`. The most important hyperparameters are `eps` and `min_samples`. `eps` describes the distance used for the cluster analysis. If a point has a core point as a neighbor within the radius specified by `eps`, then it belongs to the same cluster as the core point. If this point has at least `min_samples` neighbors within this radius, then it is a core point. Each core point can form its own cluster if it is not neighboring another core point, so `min_samples` also describes the smallest possible cluster size. The following diagram visualizes the concept of border and core points. The maximum distance `eps` here is 0.12 and `min_samples` is 4.

 Core points and directly reachable points

You can use the following rule of thumb for a good starting value for `min_samples`:

$$\text{min_samples} = 2 \cdot \text{number_of_dimensions}$$

`min_samples` should be about double the number of columns in the data. In our example it's 4. 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.

What's called a *k-distance plot* helps to determine `eps`. 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.

To create the *k-distance plot* we first need to decide which number neighbor to consider. We decide on the fourth closest neighbors, since this corresponds to `min_samples`. So we calculate all the Euclidean distances with `euclidean_distances` from `sklearn.metrics` and store them as an array named `arr_dist`. Then we sort the distances row by row and take the fifth value, because the first value is the distance from the data point to itself, which is 0.

```
In [12]: from sklearn.metrics import euclidean_distances # import distance function
arr_dist = euclidean_distances(df_non_spherical) # calculate all distances
arr_dist = np.sort(arr_dist, axis=1) # sort each row
```

In the diagram, we would therefore expect that all points in clusters would have a small distance to their 4th neighbor. However, this distance rises sharply for data points outside the clusters. So for `eps` you choose a value which comes just before the sudden increase. `k` corresponds to `min_samples`.

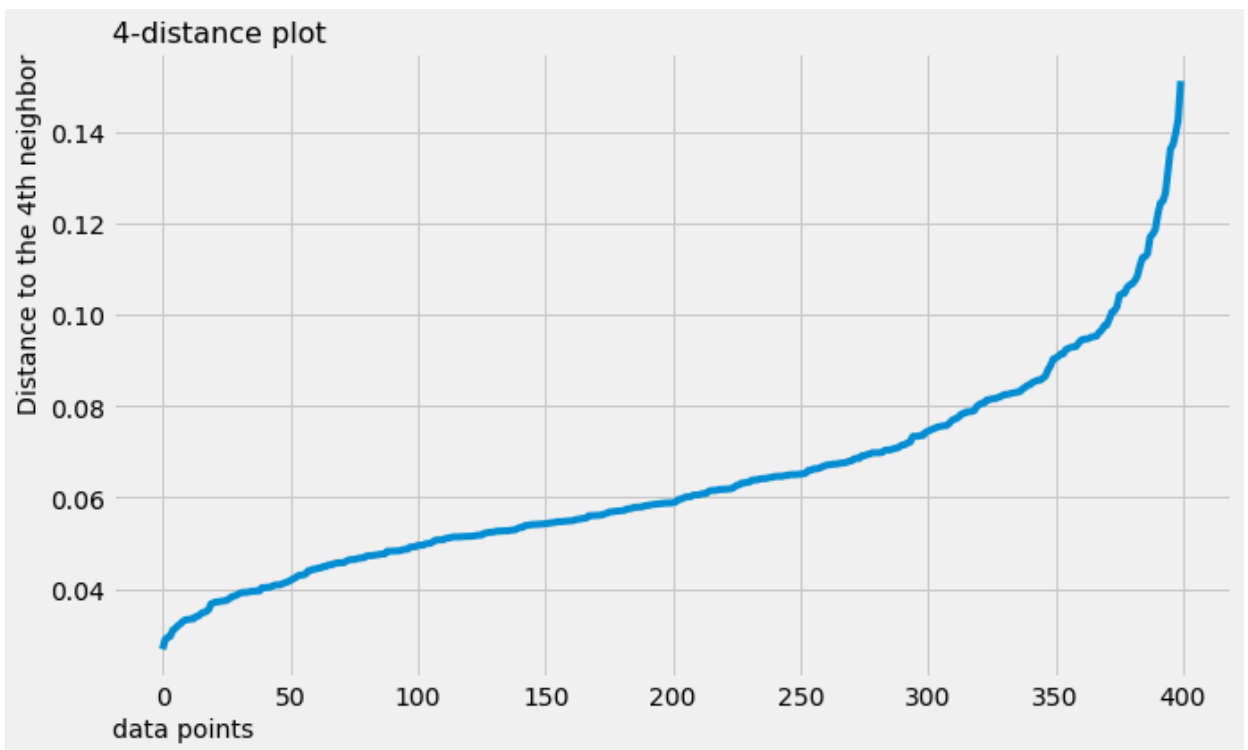
```
In [13]: fig, ax = plt.subplots(figsize=[10,6])

ax.plot(range(len(arr_dist)), np.sort(arr_dist[:,4])) # sort the distances to the fou

# set labels and title
ax.set_xlabel(xlabel='data points',
              position=[0, 0],
              horizontalalignment='left',
              size=14)
ax.set_ylabel(ylabel='Distance to the 4th neighbor',
              position=[0, 1],
              horizontalalignment='right',
              size=14)

ax.set_title(label='4-distance plot',
             loc='left',
             horizontalalignment='left',
             size=16)
```

```
Out[13]: Text(0.0, 1.0, '4-distance plot')
```



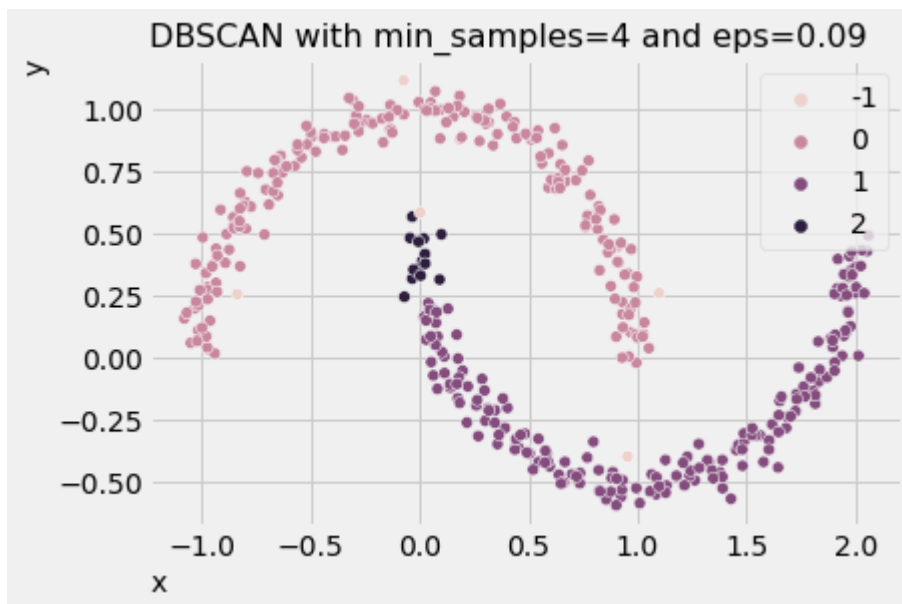
You can see that the distance to the fourth neighbor increases rapidly. The bend is approximately at a y-value of 0.09. We can use this value as `eps`. In the following code cell we'll instantiate `DBSCAN` with these hyperparameters and fit it to the data. Then we'll generate a diagram to visually validate the clusters.

```
In [14]: from sklearn.cluster import DBSCAN # import DBSCAN

model_db = DBSCAN(eps=0.09, min_samples=4) # use estimated hyperparameters
model_db.fit(df_non_spherical) # fit the model

ax = sns.scatterplot(data=df_non_spherical, x='x', y='y', hue=model_db.labels_) # plot

# set title and labels
ax.set_title('DBSCAN with min_samples=4 and eps=0.09', size=16, loc='left')
ax.set_xlabel(xlabel='x',
              position=[0, 0],
              horizontalalignment='left',
              size=14)
ax.set_ylabel(ylabel='y',
              position=[0, 1],
              horizontalalignment='right',
              size=14);
```



You can see that `DBSCAN` has found 3 clusters with these parameters. The left data points of the lower cluster have a slightly larger distance to this cluster. There are also a few outliers (noise points) which have no core points as neighbors. They receive the label -1.

If we didn't know the number of clusters, having three clusters would be fine. But we know that there should really be just 2 clusters. So our interpretation of the clusters does not quite match up what `DBSCAN` gave us as a result. So we'll proceed iteratively and change our hyperparameters until we get a better result.

In this case you can see that the clusters are relatively spread out. Perhaps selecting the distance 4th neighbor was not enough. What happens if we pass `min_samples=7` ?

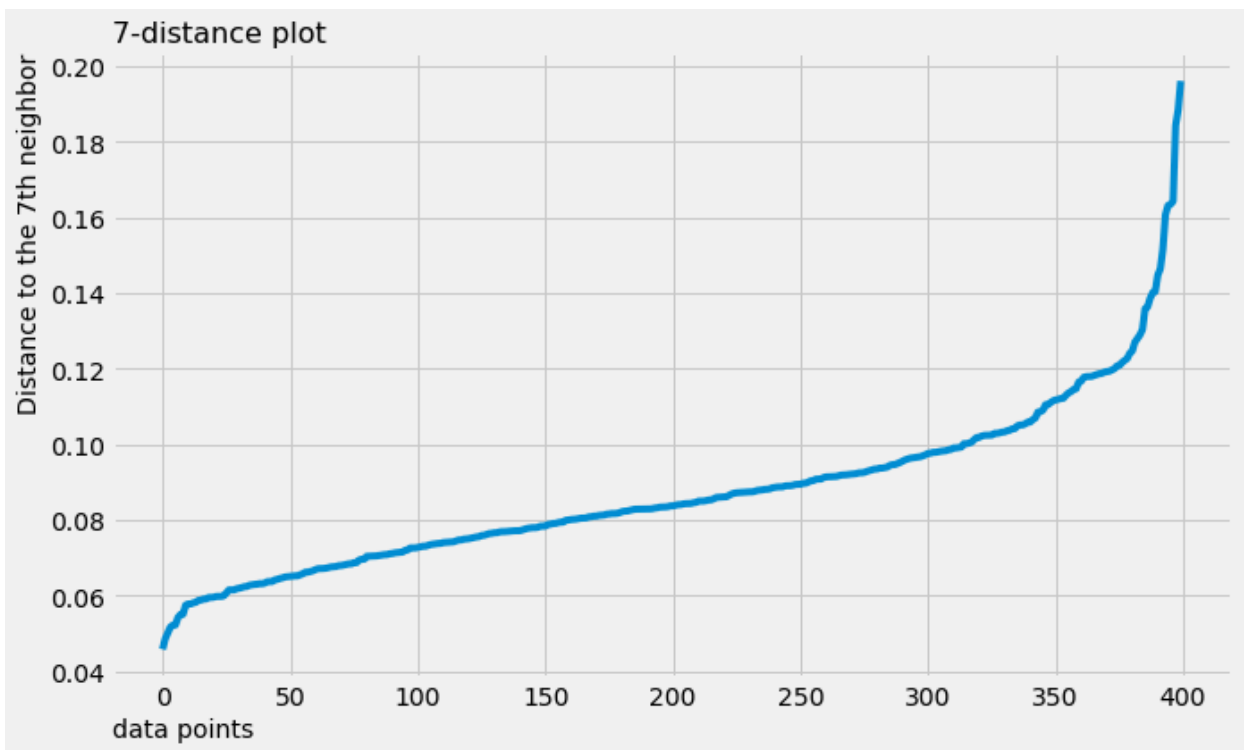
```
In [15]: fig, ax = plt.subplots(figsize=[10,6])

ax.plot(range(len(arr_dist)), np.sort(arr_dist[:,7])) # sort the distances to the sev

# set labels and title
ax.set_xlabel(xlabel='data points',
              position=[0, 0],
              horizontalalignment='left',
              size=14)
ax.set_ylabel(ylabel='Distance to the 7th neighbor',
              position=[0, 1],
              horizontalalignment='right',
              size=14)

ax.set_title(label='7-distance plot',
             loc='left',
             horizontalalignment='left',
             size=16)
```

```
Out[15]: Text(0.0, 1.0, '7-distance plot')
```



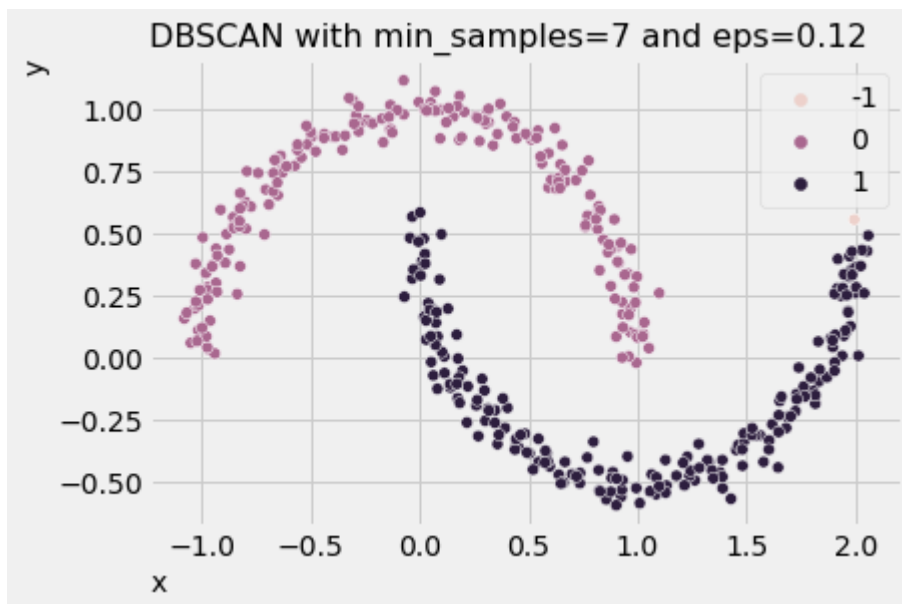
In this case, the sharp increase begins at 0.12. So we'll use that for `eps` now.

```
In [16]: from sklearn.cluster import DBSCAN # import DBSCAN

model_db = DBSCAN(eps=0.12, min_samples=7) # use estimated hyperparameters
model_db.fit(df_non_spherical) # fit the model

ax = sns.scatterplot(data=df_non_spherical, x='x', y='y', hue=model_db.labels_) # plot

# set title and labels
ax.set_title('DBSCAN with min_samples=7 and eps=0.12', size=16, loc='left')
ax.set_xlabel(xlabel='x',
              position=[0, 0],
              horizontalalignment='left',
              size=14)
ax.set_ylabel(ylabel='y',
              position=[0, 1],
              horizontalalignment='right',
              size=14);
```



With these parameters we get 2 clusters and one outlier. This is a satisfactory result.

Congratulations: You have recapped the most important things from the clustering chapter. Now you can use two different clustering algorithms, `KMeans` and `DBSCAN` so you have a have a solid foundation.

Remember:

- k-Means works well with spherical clusters
- `DBSCAN` can be a good choice for clusters with different shapes
- Estimate the hyperparameters using the distances

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.
