# Clustering with k-Means

Module 1 | Chapter 3 | Notebook 2

---

In the last two chapters you have been using supervised learning algorithms. Now we will focus on unsupervised learning, clustering in particular. Clustering is about dividing data points into groups, or clusters. Now, that might sound like classification, However, the difference is that with clustering we have no idea what groups there are. We often don't even know how many groups there are. So with clustering, we take data and create new categories. Clustering algorithms take a look at the data and use its structure to achieve this. Clustering's great strength is bringing order to chaos. In this lesson, you will get to know the k-means algorithm. This will be the first time you use a clustering algorithm. You will also learn what you have to consider when using this algorithm. By the end of this lesson you will be able to:

- Use k-means to split your data into clusters
- Estimate the k parameter of k-means
- Understand what is required for successful clustering

---

## How k-means works

We'll start by importing synthetic data. It's structured so we can already see the groups with the naked eye. This will help give you a better understanding of how k-means works. In exercises later on in the course, you will then use real data, which you will divide into groups using k-means.

With unsupervised clustering, you also follow the same familiar steps for data modeling in `sklearn`. The only difference is that you don't have target values that you are predicting. This means that there are only 4 steps:

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

First, import the data in the *clustering-data.csv* file. which is located in the current working directory. Save it as a `DataFrame` named `df` and print the first 5 rows. Remember to import `pandas` with its conventional alias.

In [3]:
```
import pandas as pd
df = pd.read_csv('clustering-data.csv')
df.head()
```

Out[3]:

| | x | y |
|---|---|---|
| **0** | 2.020802 | -2.715714 |
| **1** | 9.765899 | 0.128339 |
| **2** | 6.948591 | 1.605763 |
| **3** | -9.575013 | 1.730110 |
| **4** | -8.699537 | 1.921142 |

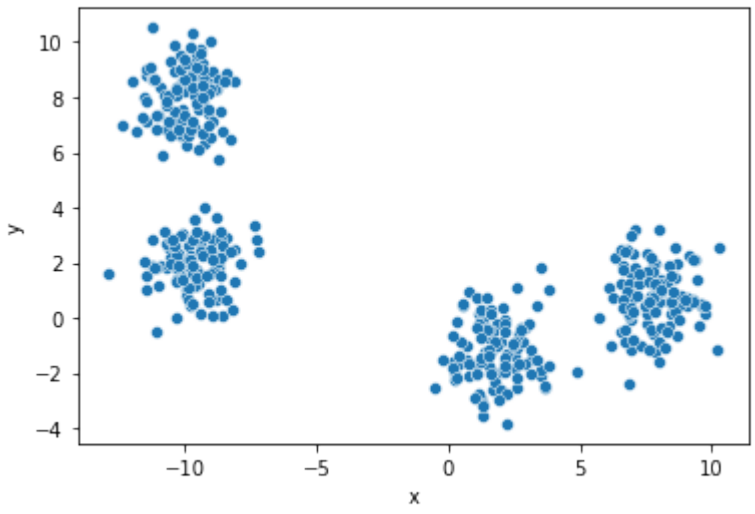
Our data has two columns `'x'` and `'y'`. These could be the GPS coordinates of customers using an app on their smartphone, for example. What do the data points look like? We'll use the `sns.scatterplot()` function from the `seaborn` module to visualize this quickly.

`sns.scatterplot()` displays the data as a scatter plot. The most important parameters of this function are `x` and `y`. `x` is assigned the values that are displayed on the x-axis and `y` is assigned the y-values accordingly. You can assign them lists, arrays, series or the name of a `DataFrame` column, and if you do this you also have to assign this `DataFrame` to the `data` parameter.

Use `sns.scatterplot()` to visualize the data points. Pass `df` to the `data` parameter and the column names `'x'` and `'y'` to the corresponding parameters. Don't forget to import `seaborn` with its conventional alias `sns`.

In [5]:
```
import seaborn as sns
sns.scatterplot(x=df.loc[:,'x'],  y=df.loc[:,'y'])
```

Out[5]:
```
<AxesSubplot:xlabel='x', ylabel='y'>
```




You can see straight away that the data is divided into four groups. These are very cleanly separated from each other. For example, the groups could represent different cities where an app's users are located.

Our goal is to have the data points neatly divided into groups. So we want to add a new column to the data, which indicates which group the respective data point belongs to. If the data is

neatly separated like in this case, you could do this task yourself. However that would take a long time and would not be an enjoyable task. It's nice that there are clustering algorithms to do this for you!

The classic clustering algorithm is the k-Means algorithm. It's called this because it forms a given number k of clusters and achieves this by calculating averages.

**Important:** Don't confuse k-means with k-nearest neighbors! k-means is an unsupervised learning clustering algorithm, k-nearest neighbors is a supervised learning classification algorithm. k always describes a hyperparameter.

The procedure for k-means consists of the following steps:

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

The following gif explains this:


Now let's look at how well that works. To do this, import `KMeans` directly from the `sklearn.cluster` submodule. The most important hyperparameter for k-means is `n_clusters`. This is actually k that gives the algorithm its name, and it specifies the number of clusters. However, to make it clear what the parameter expresses, the people who wrote `sklearn` chose an unambiguous name. Instantiate the model with the value `n_clusters=4` and store it with the name `model`.

In [7]:
```
from sklearn.cluster import KMeans
model = KMeans(n_clusters=4)
```

You've already completed the first two steps with `sklearn`: "Select model type" and "Instantiate a model with certain settings known as hyperparameters". You can skip the 3rd step - "Organize data into a feature matrix". We will use the `DataFrame` `df` directly, as we will use every column in this case. The data does not contain a target variable that we would have to split.

Now it's time for the 4th step - "Model fitting". Like all `sklearn` models, `KMeans` offers the `my_model.fit()` method that you can use to achieve this. The final cluster centers are stored in the attribute `my_model.cluster_centers_` after the fitting. The assignment of the individual points is also stored directly in an attribute - in `my_model.labels_`.

Fit `model` to the data. Remember, this is unsupervised learning. so you don't need to pass a target vector. Then visualize the data again with `sns.scatterplot()`. Now use the `hue` parameter. It allows us to differentiate between datapoints using color. To do this, pass the

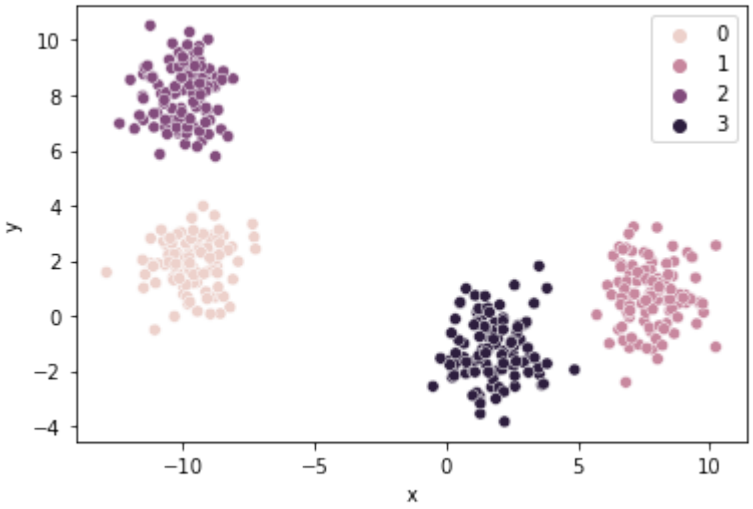
`labels` attribute from `model` to `hue`. `sns.scatterplot()` then assigns each cluster a different color. Note that attributes in `sklearn` are written with an underscore after them, unlike `my_df.shape`, for example. For example, you can access the labels of a model using `my_model.labels_`.

In [11]:
```
model.fit(df)
#print('cluster_centers_', model.cluster_centers_)
#print('labels_', model.labels_)

sns.scatterplot(x=df.loc[:,'x'],  y=df.loc[:,'y'], hue=model.labels_)

print('shape', df.shape)
```

```
shape (440, 2)
```




The division of data points into clusters looks very good. Now we have already completed the `sklearn` modeling steps.

In this case, you can see immediately in the visualization that the cluster allocation makes sense. For other datasets, you would have to look at each cluster in more detail. Unlike supervised learning, you do not predict new data points to validate the model. In this chapter, you will learn methods that can help you to verify cluster analyses.

Since the cluster centers are always moved to the centers of their associated points (step 4 of k-means), we would expect them now to be in the middle of the visible clusters. Their `'x'` and `'y'` values should be roughly the same as the averages of each group. Is that the case? Calculate the averages for each group and compare them to the values in the `model.cluster_centers_` attribute.

Tip: To do this, add a new column named `'labels'` to the `DataFrame`, containing the labels from `model.labels_`. Now you can group them with `my_df.groupby()`.

In [15]:
```
df.loc[:,'labels'] = model.labels_
print(df.groupby('labels').mean())
print(model.cluster_centers_)
```

```
                x         y
labels
0       -9.607692  1.938419
1        7.853347  0.765660
2       -9.931848  8.068650
3        1.780975 -1.216927
[[-9.60769184  1.93841895]
 [ 7.85334674  0.76565961]
 [-9.93184787  8.06865024]
 [ 1.78097483 -1.21692708]]
```

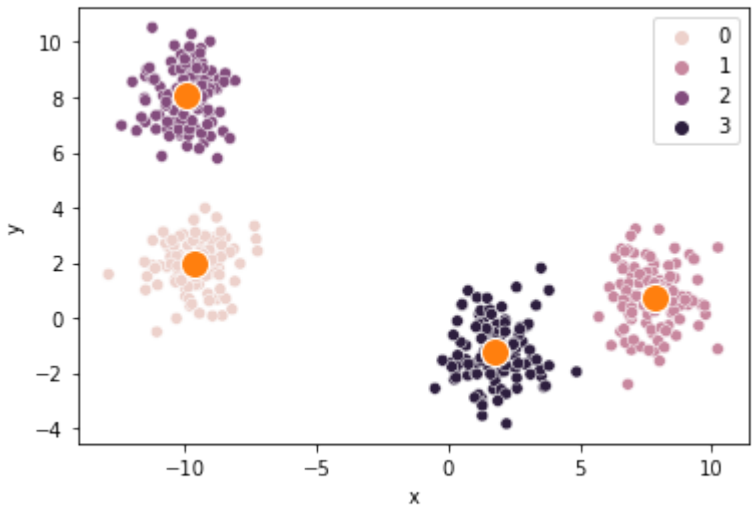
We found that the cluster centers are identical to the calculated average values of the clusters, as we expected. Now it would be nice if the visualization also showed the cluster centers. `sns.scatterplot()` returns an *axes* object, which you already know from `matplotlib`. Produce the same visualization as before and store the return value as `ax_cluster`.

Then use `sns.scatterplot()` again, but this time visualize the cluster centers. To show them in the same visualization, pass the argument `ax = ax_cluster` to the function. So you are assigning the variable `ax_cluster`, which is an *axes* object, to the parameter `ax`. Also assign the values `'X'` and `200` to the `marker` and `s` parameters. The `marker` parameter specifies how data points should appear. You want to distinguish the centers from the normal datapoints. Using the value `'X'` makes them appear as crosses. The `s` parameter stands for size and determines the size of the markers. `s = 200` will make sure that you can quickly spot the cluster centers.

Tip: `KMeans.cluster_centers_` contains an `ndarray` from the `numpy` module. You access the values in it like the values in a `DataFrame`, only you omit `.iloc`. For example, with `model.cluster_centers_[:,0]` you get the values of all rows in the first column. Try passing this to the `x` parameter of `sns.scatterplot()`.

In [45]:
```
ax = sns.scatterplot(x=df.loc[:,'x'],  y=df.loc[:,'y'], hue=model.labels_, markers=200
sns.scatterplot(markers=200, ax=ax, s=200,
                x=pd.DataFrame(model.cluster_centers_).iloc[:,0],
                y=pd.DataFrame(model.cluster_centers_).iloc[:,1])
```

Out[45]:
```
<AxesSubplot:xlabel='x', ylabel='y'>
```



Our visualization looks like this: although the colors and label allocation might be different for you.


**Congratulations:** You have instantiated and used your first clustering model. Then you displayed the results of the cluster analysis visually. Now you know that the k Means algorithm creates a cluster centre (also known as a centroid) for every group and how it assigns these to the data points.

# k-means methods and distance measure

You have already encountered estimators with a `my_estimator.fit()` method to estimate something, e.g. `LinearRegression` estimates the slope values of the individual features. Some *estimators* are *predictors* at the same time. They predict something with a `my_predictor.predict()` method. For example, `LinearRegression` calculates continuous values from the features. Other estimators offer the `my_transformer.transform()` method. They change the data and are therefore called *transformers*. An example of this is `StandardScaler`.

`KMeans` has both the `my_model.transform()` method and the `my_model.predict()` method. So it is both a *predictor* and a *transformer*, and `my_model.predict()` is used to assign a cluster center to new data points. So if we pass the same data we already used for `my_model.fit()`, the result should consist of the same labels as in the attribute `my_model.labels_`.

Now let's test this out briefly. Then use `my_model.predict()` on the data. Store the result in of the variable `labels_predicted`. A quick way to compare this with `model.labels_` is to compare the two arrays with one another and calculate the sum from the result. Both `labels_predicted` and `model.labels_` are equal-sized arrays from `numpy`. A comparison therefore creates a new array consisting of Boolean values. If you interpret these as numbers (e.g. when summing up), then `True` is counted as 1 and `False` as 0. So if the sum of the array with the Boolean values is the same as the length of the array, then all the values are `True`. If it results in zero, then all the values are `False`. If we now want to check if all the values are correct, it's easier to count the incorrect values. If there are zero incorrect values, then we **always** know that everything is correct.

Now compare the predicted labels with the ones that already exist.

Tip: The comparison `==` returns `True` if the values are equal. The comparison `!=` returns `True` if the values are **not** equal. Only use the `'x'` and `'y'` columns of `df` to make the predictions, otherwise you will get an error.

In [50]:
```
labels_predicted = model.predict(df.loc[:, ['x', 'y']])
sum(labels_predicted != model.labels_)
```

Out[50]:
```
0
```

In our case, the predicted labels are the same as those already created. We use the term label synonymously with the terms category and class. So we are convinced that `my_model.predict()` can be used to assign data points to clusters. But what does `my_model.transform()` do? It is best to apply it to `df` and see what the result is. Print `model.transform(df.loc[:, ['x','y']])`.

In [53]:
```
model.transform(df.loc[:, ['x','y']])
```

Out[53]:
```
array([[12.52528699,  6.79253585, 16.0987062 ,  1.51785336],
       [19.45796479,  2.01594453, 21.23793214,  8.09745305],
       [16.55962438,  1.23464836, 18.07534585,  5.88827945],
       ...,
       [ 5.59059335, 20.03219808,  1.35583073, 15.43702638],
       [18.04355194,  0.93276513, 19.5023956 ,  7.18256735],
       [11.59733307,  6.68425604, 14.8232302 ,  0.2996433 ]])
```

The result is a two-dimensional array with 4 columns. It is obvious that each column has something to do with one of the four cluster centers. And this also happens to be the case. For each data point, we get a row with four values that represent the distance to the respective cluster centers. If the distance in the first column is the smallest, the data point receives the label 0. If the distance in the second column is the smallest, the data point receives the label 1, and so on.

But now there are different ways to measure the distance from one point to another. A distance not necessarily the same, so to speak, and depends on the specific problem. What definition does `KMeans` use? The distance here is based on the L2 norm, which is also called the Euclidean norm. You already learned about this in *Measuring Distances with Norms and Metrics (Module 0 Chapter 2)*. The L2 norm corresponds to the concept of distance used in everyday life. To do this, draw a line from one point to another and measure the length of this line. Mathematically speaking, two vectors (one-dimensional arrays) are subtracted from each other and then the values of the new vector are squared and summed up. In the end the root is taken from the sum.

So the following formula shows the distance between two data points $v$ and $u$ consisting of $n$ features each:

$$\begin{equation} |\vec{v} - \vec{u}| = \sqrt{(v_1-u_1)^2 + (v_2-u_2)^2 + ... + (v_n-u_n)^2} \end{equation}$$

The arrow indicates that it is a vector. The vertical lines indicate that the length of the vector is meant. The letters with the index describe the relevant elements or components of the vector.

Now we'll calculate the distances between the data points and the cluster centers ourselves to make sure that the Euclidean distance is actually what is used. For this, we'll use the `euclidean_distances` function from the submodule `sklearn.metrics.pairwise`. If we pass two-dimensional arrays (or DataFrames) we get the distance between the rows. Import

`euclidean_distances` directly and then calculate the distances. Compare them with the values we got from `model.transform()`.

In [55]:
```
from sklearn.metrics.pairwise import euclidean_distances
modelTransformed =  model.transform(df.loc[:, ['x','y']])
euclidian = euclidean_distances(df.loc[:,['x','y']], model.cluster_centers_)
sum(modelTransformed != euclidian)
```

Out[55]:
```
array([0, 0, 0, 0])
```

The values should be identical. We have made sure that `KMeans` really does use the L2 norm to calculate the distance. The values with the smallest Euclidean distance to a cluster center are assigned to this center.

**Congratulations:** You have learned that `KMeans` is a predictor as well as a transformer. You have taken a closer look at `my_model.predict()` and `my_model.transform()` do. In doing so you found that `KMeans` calculates the Euclidean distance for assigning clusters. This is good to know, because it results in a weakness that we will look at throughout this chapter. But in the next notebook, you will familiarize yourself with a real data set that we will perform a cluster analysis on.

**Remember:**

- k-means assigns every datapoint to the closest cluster center, using the L2 norm to calculate the distance
- `n_clusters` returns the number of cluster centers
- Cluster centers are moved to the mean values of the corresponding data points
- Create a scatter plot with different colored groups with `sns.scatterplot()` and its `hue` parameter

**Literature:**


- Alpaydin, Ethem. 2014. *Introduction to Machine Learning*. Cambridge: MIT Press, 2014. p. 165 pp.


---

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

---



---