k-Means - Determining the Number of Clusters

Module 1 | Chapter 3 | Notebook 5

Now that you have cleaned and prepared the order data, we will use it to form customer groups in this lesson. We will use the k-Means algorithm, which is an unsupervised learning approach. Afterwards we will take a closer look at the groups formed. By the end of this lesson you will be able to:

- Understand the k-Means score
- Estimate the number of clusters - k
- Reproduce results exactly

Importing and standardising 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. It is not yet clear exactly what characteristics the customers will be grouped by.

However, the customers in the groups should have something in common by the end of the project. The marketing department can then use this common ground to tailor its measures to the most important customer groups.

You have been provided with data containing all orders and cancellations in a one year period, in order to gain some initial insights. The data is stored in the files *online_retail_data.csv* and *online_retail_cancellations_data.csv*.

In the last two lessons you created a `DataFrame` from both files based on customers and named it `df_customers`. You saved this in the last code cell as *customer_data_prepared.p*. You can now import it again directly with `pandas`. Use the `pd.read_pickle()` function. Store the `DataFrame` as `df_customers` and print the first 5 rows.

Tip: If the *customer_data_prepared.p* file is not there, go to the last exercise *Feature Engineering for Customer Segmentation* and run the last code cell.

In [1]:
```
import pandas as pd
df_customers = pd.read_pickle('customer_data_prepared.p')
```

```
df_customers.info()
df_customers.head()
```

```
<class 'pandas.core.frame.DataFrame'>
Float64Index: 2869 entries, 12347.0 to 18281.0
Data columns (total 8 columns):
 #   Column               Non-Null Count  Dtype  
---  ------               --------------  -----  
 0   InvoiceNo            2869 non-null   int64  
 1   Revenue              2869 non-null   float64
 2   Quantity             2869 non-null   int64  
 3   RevenueMean          2869 non-null   float64
 4   QuantityMean         2869 non-null   float64
 5   PriceMean            2869 non-null   float64
 6   DaysBetweenInvoices  2869 non-null   int64  
 7   DaysSinceInvoice     2869 non-null   int64  
dtypes: float64(4), int64(4)
memory usage: 201.7 KB
```

Out[1]:

CustomerID	InvoiceNo	Revenue	Quantity	RevenueMean	QuantityMean	PriceMean	DaysBetweenIn
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	

It should only contain the `float64` and `int64`. The first `'CustomerID'` is `12347`, it has the `'Revenue'` `4310.00`.

Each row represents one customer. The following data dictionary explains what the columns represent:

Column number	Column name	Type	Description
0	`'Revenue'`	continuous (`float`)	total revenue in GBP
1	`'Quantity'`	continuous (`int`)	total number of items purchased
2	`'InvoiceNo'`	continuous (`float`)	the order number
3	`'RevenueMean'`	continuous (`float`)	average revenue per order
4	`'QuantityMean'`	continuous (`float`)	average number of items per order
5	`'PriceMean'`	continuous (`float`)	average item price

Column number	Column name	Type	Description
6	`'DaysBetweenInvoices'`	continuous (`int`)	average number of days between orders
7	`'DaysSinceInvoice'`	continuous (`int`)	number of days since the customer's last order

There is one final data preparation step that we haven't performed yet, but which is important for clustering: standardization. In the *Clustering with k-Means* lesson, you saw that `KMeans` from `sklearn` uses the Euclidean distance between the data. Each point is assigned to the centroid (cluster center) that is closest to it. However, this means that properties with values of different magnitudes are also weighted differently. In principle, the problem is similar to with the k-nearest neighbors classification model, see *k-Nearest-Neighbors (Module 1 Chapter 2)*.

For example, the values 4310 and 1079 are available in `'Revenue'`. If we had two data points with these values, that are otherwise the same, their absolute distance would be 4310-1079 = 3231. If, on the other hand, we have two data points that differ only in `'InvoiceNo'`, for example, their distance is much smaller. Then we would have values such as 4 or 1. Here the distance would only be 3. The revenue distance 3231 is roughly a thousand times more important for `KMeans`, and `'InvoiceNo'` would then be ignored when forming clusters.

For this reason, we have to adjust the scales of the features. One possibility is to standardize them. We can do this with `StandardScaler` from the submodule `sklearn.preprocessing`, for example, see *Regularization (Module 1 Chapter 1)*.

Import `StandardScaler`, instantiate it with the name `standardizer`. Fit it to `df_customers` and transform the data. Store the transformed data as `arr_customers_std`. This is a two-dimensional array. Then print the first row. You can ignore or suppress the `DataConversionWarning` as usual.

In [2]:
```
#suppress conversion warning
from sklearn.exceptions import DataConversionWarning
import warnings
warnings.filterwarnings(action='ignore', category=DataConversionWarning)

print(df_customers.columns)

#df_customers =pd.merge(df_customers.loc[:,['Revenue', 'Quantity']],
#                       df_customers.loc[:, ['InvoiceNo','RevenueMean', 'QuantityMean'
#                           'PriceMean', 'DaysBetweenInvoices', 'DaysSinceInvoice']],
#                       left_index=True,
#                       right_index=True,
#                       how='left')

from sklearn.preprocessing import StandardScaler
standardizer = StandardScaler()
standardizer.fit(df_customers)
arr_customers_std = standardizer.transform(df_customers)
arr_customers_std[0, :]
```

Out[2]:
```
Index(['InvoiceNo', 'Revenue', 'Quantity', 'RevenueMean', 'QuantityMean',
       'PriceMean', 'DaysBetweenInvoices', 'DaysSinceInvoice'],
      dtype='object')
array([ 0.34113668,  0.26768646,  0.25037287,  0.51706333,  0.38968173,
       -0.05571575,  0.61314937, -0.90204604])
```

The first value of the first row should be around 0.268.

Congratulations: You imported the data and got it ready for the algorithm. Now let's take a look at clustering itself and see what the `KMeans` *score* is all about.

Finding the right *k*

A weakness of k-Means is that you have to tell it the number of clusters in advance. In reality, however, you rarely know this number. Now we'll look at how we can estimate it. But to do this, we need the model first. Import `KMeans` directly from the `sklearn.cluster` submodule. Then instantiate it with the parameter `n_clusters=3` and save it as a variable named `model`.

In [3]:
```
from sklearn.cluster import KMeans
model = KMeans(n_clusters=3)
```

Now we have specified three clusters, although we wanted to estimate the correct number. But for this we have to try different numbers of clusters. Only then can we use the internal score to determine which number is a good fit. First let's look at the score to understand why we estimate the number of clusters the way we do.

Fit model to the data. Then apply the `my_model.score()` method to the same data. What score do you get?

In [4]:
```
model.fit(arr_customers_std)
model.score(arr_customers_std)
```

Out[4]:
```
-15073.11511768112
```

Our score is about -15500. With `sklearn`, negative values are often used for metrics. This means that the actual score is 15550. The higher this value, the further away the data points are from the cluster center. So a smaller value is better.

The exact value may differ, since the exact state of the model depends on the initial position of the cluster centers. These are chosen randomly. We'll look at how we can reproduce the results very shortly. What does the score mean?

As we have already experienced, k-Means tries to place the cluster centers in such a way that the distance between them and the associated data points is as small as possible. As a quality metric for these clusters, k-Means offers us what's called the *within-cluster sum of squares* (WCSS), i.e. the sum of the squared distances from the center (${\mu }_{i}$) to the cluster points ($x \in S_{i}$). This value is summed across the all clusters (S_{i}). The calculation formally follows the following formula:

\begin{equation} {\displaystyle \operatorname {WCSS} = \sum _{i=1}^{k}\sum _{\mathbf {x} \in S_{i}}\left\|\mathbf {x} -{\boldsymbol {\mu }}_{i}\right\|^{2}} \end{equation}

If we write out the calculation of the score, we get the following loop. Run it. Do you get the same value as with the `my_model.score()` method?

In [5]:
```
from numpy.linalg import norm  # import norm to calculate the distance

wcss = 0  # define the distance
for cluster in range(model.n_clusters):  # itereate through every cluster
    cluster_center = model.cluster_centers_[cluster]  # select the cluster center
    cluster_mask = model.labels_ == cluster  # create a mask to get only customers bel
    for row in arr_customers_std[cluster_mask, :]:  # iterate through every data point
        distance = norm(cluster_center - row)**2  # calculate the distance from row to
        wcss = wcss + distance  # add the squared distance to the within-cluster sum o
print(wcss)
```

```
15073.115117681098
```

The value is the same, only positive. So now we know what the score means. It's the sum of the squared distances between the centers and the corresponding data points. We can use the *within-cluster sum of squares* as a quality metric for the clusters.

To find the optimal number of clusters, there's unfortunately no alternative to just trying out a lot of them. Generate a list `cluster_scores` containing the scores for all numbers of clusters `n_clusters` from 1 to 29. The best way to do this is to write a loop which instantiates and fits `KMeans` each time.

In [6]:
```
cluster_scores = []
for x in range(1, 30):
    modelTry = KMeans(n_clusters=x)
    modelTry.fit(arr_customers_std)
    cluster_scores.append(modelTry.score(arr_customers_std))
    print(x, '-', modelTry.score(arr_customers_std))
```

```
1 - -22951.99999999996
2 - -18226.99638755524
3 - -15456.59577602862
4 - -12374.055808517776
5 - -10148.92735337501
6 - -8766.997142231208
7 - -7268.334360358565
8 - -6532.987680668903
9 - -5777.356291632254
10 - -5192.38604224378
11 - -4707.5215744666875
12 - -4235.182215060884
13 - -4022.8070190186154
14 - -3770.057586393671
15 - -3534.734253503927
16 - -3314.1880945494136
17 - -3171.662511219539
18 - -3046.8972129867284
19 - -2805.1617213157338
20 - -2677.5885592193104
21 - -2516.888897558856
22 - -2397.597992095998
23 - -2326.00088707282
24 - -2187.245265201924
25 - -2118.626923727374
26 - -1974.9077821281755
27 - -1897.0297295900377
28 - -1822.0278522644298
29 - -1767.7756437695637
```

We'll now use what's called the *elbow method* to choose the number of clusters. It involves plotting the *scores* on the y-axis. The corresponding number of clusters is then shown on the x-axis. The result is a line that has approximately the shape of an elbow. The appropriate number of clusters is located where the line bends. This is not always clear, but that doesn't matter. In cases like that, there isn't just one suitable cluster number.

What does it look like for our data? Plot the line. Don't forget to import `matplotlib.pyplot` with its conventional alias.

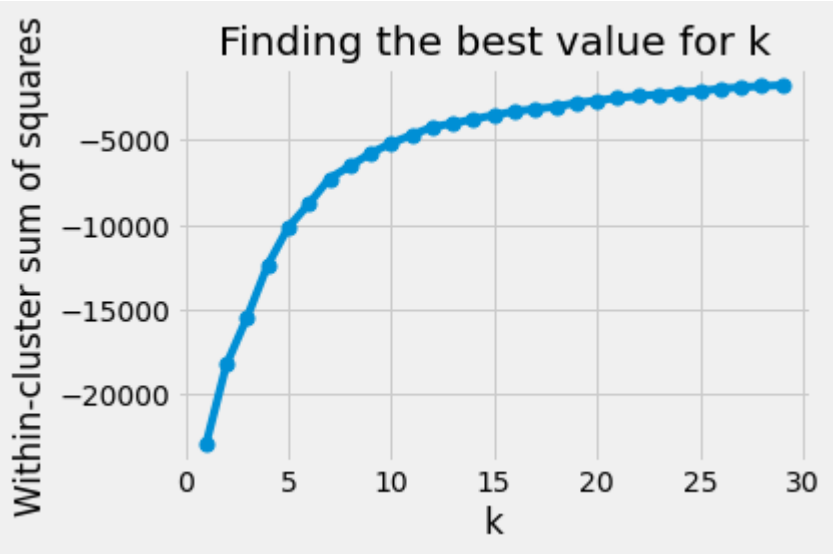
Tips: For the x-values on the graph, you can use `range(1, 30)` directly. By using a style, you can make your images look nicer with little effort, for example with `plt.style.use('fivethirtyeight')`.

In [7]:
```
import matplotlib.pyplot as plt

plt.style.use('fivethirtyeight')

fig, ax = plt.subplots()
ax.plot(range(1, 30), cluster_scores, marker='o', markersize=7)
ax.set_title("Finding the best value for k")
ax.set_xlabel("k")
ax.set_ylabel("Within-cluster sum of squares")

plt.tight_layout()
```



After a little tweaking, our line looks like this:

You can see that the *within-cluster sum of squares* gets better and better, the more clusters there are. This is not surprising. In extreme cases, each data point could form a separate cluster. Then we would have a perfect *within-cluster sum of squares* of 0. However, the aim here is not to maximize the score. Instead, we want to gain new insights. To do that, we need clusters that we can interpret meaningfully. This is why you pick a number of clusters that is at the bend in the line. This is because a further increase would only result in an insignificant improvement of the quality standard.

The bend in the line here is more of a round curve. This makes it more difficult to decide on a value. But as already mentioned, there is not just one correct value and all the other values are wrong. We would recommend choosing the value `n_clusters=7`.

Congratulations: Now you know a model quality metric for clustering! You used this to determine a good value to use as the number of clusters. Next, we'll take a look at the clusters to analyze the different customer groups.

Interpreting the clusters

When you have finished your initial exploration and want to share your own cluster analysis, it is especially important that you are able to reproduce the results. So let's take a brief look at how we can achieve this with `sklearn`.

We have already learned that the score depends on the initial positions of the centers, which are chosen at random. To avoid ending up with very bad clusters due to bad luck, `KMeans` does not just perform clustering only once. The standard number of runs is 10. You can set this with the `n_init` parameter when instantiating the model. In the end, the cluster centers that achieved the best score are selected. If you increase this value, you will get better *scores* and they will remain more stable.

Try it out. Calculate the score ten times for each of the following `n_init` values: of 1, 10 and 50. Store these results in separate lists and return the the mean values and standard deviations. Use `n_clusters=7`.

Tips: You can use 2 nested loops to carry out this task. The first iterates through the values `[1, 10, 50]`. The second loop performs 10 iterations. Instantiating, training and measuring should take place in the second loop. With the `np.mean()` and `np.std()` functions, you can generate the mean and standard deviation of the values of a list or an array.

In [8]:
```
import numpy as np

for x in [1,10,50]:
    anotherClusterTest= []

    for y in range(10):
        model = KMeans(n_clusters=7, n_init=x)
        model.fit(arr_customers_std)
        anotherClusterTest.append(model.score(arr_customers_std))
    print(x)
    print( np.mean(anotherClusterTest) , np.std(anotherClusterTest), '\n')
```

```
1
-7657.2231552431895 332.7915554862762

10
-7304.888070690822 81.95079531242413

50
-7263.387711634981 8.241426019859803
```

As expected, we obtain smaller absolute numbers - better values - for mean values and standard deviations when `n_init` gets larger.

But with a higher `n_init`, our results still aren't quite exactly reproducible. Furthermore, a higher `n_init` has the disadvantage that it requires more computing time. When comparing and developing models, it is very important that we can compare our results thoroughly. For this reason, every model in `sklearn` that works with random values has the parameter `random_state`. We can assign this an `int` value, which is then used to set the internal random number generator. Thus the model comes to the same results for the same `random_state`. It's irrelevant which number you choose here. A higher number for `random_state` does not necessarily lead to a better result or require more computing time. However, it can change the order in which the clusters are numbered in this case. what *within-cluster sum of squares* do you get when you instantiate `KMeans` with the parameters `n_clusters=7` and `random_state=0`?

Important: The clusters are labelled at random. So if you cluster the same data twice and you get basically the same clusters, they might have different names. Cluster no. 3 could be cluster no. 5 the second time around. The same is true if you use a different value for `random_state`:

For example, if we used `random_state=42` instead of `random_state=0`, the indexing of the clusters might change. This can cause confusion. So you should always specify the `random_state` when you need to be able to reproduce the results.

Even then the results depend on the version of scikit-learn. In the following, we will show you the *exact* results. But keep in mind that it is possible for you to get slightly different results. In this case, check if your results match our results at least *qualitatively*.

In [9]:
```
model = KMeans(n_clusters=7, random_state=0)
model.fit(arr_customers_std)
model.score(arr_customers_std)
```

Out[9]:
```
-7270.115779921629
```

We get the value `-7270.115779921631`. You should get the same value. If not, check if the difference is at least less than 100. Now let's take a closer look at the clusters. Because now the results and cluster assignments are stable.

To understand what the customer groups look like, we'll use `df_customers`. This is because the standardized data is more difficult to interpret. Add the cluster names to `df_customers` as a new column called `'Labels'`. You can find the clusters as an array in the attribute `my_model.labels_`. The order of the labels is the same as that of the customer numbers. So you don't need to pay any special attention to this. Then use the `pd.crosstab()` function. It creates a cross tabulation and counts how often each value occurs. Pass it the labels and the parameter `columns='count'`. This creates a `DataFrame` containing the `'count'` column with the number of each value.

In [30]:
```
df_customers.loc[:, 'Labels'] = model.labels_
pd.crosstab(df_customers.loc[:, 'Labels'], columns='count')
```

Out[30]:

col_0	count
Labels	
0	1520
1	625
2	3
3	1
4	693
5	12
6	15

We get the following results:

Do you get identical results? If not, make sure your results are at least similar. This means that there should be a large cluster with about 1500 data points, two medium-sized clusters and four small clusters.

You can see that in our case the cluster with the number 3 only has one customer number! What does that mean for this person? Print the relevant row of `df_customers`.

In [31]:
```
mask = df_customers.loc[:, 'Labels'] == 3
df_customers.loc[mask, :]
```

Out[31]:

CustomerID	InvoiceNo	Revenue	Quantity	RevenueMean	QuantityMean	PriceMean	DaysBetweenIn
15098.0	3	39619.5	61	13206.5	20.333333	649.5	

The customer number of the cluster is 15098. You can see that this person placed three orders (`'InvoiceNo'`) on the same day (`'DaysBetweenInvoices'`). The value of these orders was very high. The customer only bought a few items (`'QuantityMean'`), but these had a high price (`'PriceMean'`). This seems to be very unusual purchasing behavior.

Cluster number 2 also contains very few customers. Print the relevant rows.

In [33]:
```
mask = df_customers.loc[:, 'Labels'] == 2
df_customers.loc[mask, :]
```

Out[33]:

CustomerID	InvoiceNo	Revenue	Quantity	RevenueMean	QuantityMean	PriceMean	DaysBetweenI
12415.0	20	123638.18	77242	6181.909000	3862.100000	1.600660	
14646.0	73	278778.02	197132	3818.876986	2700.438356	1.414169	
18102.0	60	259657.30	64124	4327.621667	1068.733333	4.049300	

Cluster 2 contains the `'CustomerID'` values: 12415, 14646, and 18102. These are extremely profitable customers. They have made frequent orders and have spent an extremely large amount of money. The orders included many items that are relatively inexpensive individually. Cluster 2 is therefore made up of customers we definitely want to retain. But their most recent orders were not long ago. So advertising measures might not be necessary right now.

The other clusters have more customers, so we won't look at each customer number individually. Group the data according to the labels. Store the result as `groups_customers`.

In [39]:
```
groups_customers = df_customers.groupby('Labels')
```

Now print the typical value of each property for each group. Use the `my_dfGroupBy.median()` method. The median is the value that is exactly in the middle when

all values of a feature are ordered by size. This means that one half of the values is less than or equal to the median and the other half is greater than or equal to the median. The median is therefore often a good choice for quickly assessing the results.

In [40]:
```
groups_customers.median()
```

Out[40]:

Labels	InvoiceNo	Revenue	Quantity	RevenueMean	QuantityMean	PriceMean	DaysBetweenInvoic
0	3.0	1017.82	609.5	316.085600	185.958333	1.714620	19
1	3.0	771.17	443.0	266.606667	150.500000	1.805578	75
2	60.0	259657.30	77242.0	4327.621667	2700.438356	1.600660	6
3	3.0	39619.50	61.0	13206.500000	20.333333	649.500000	0
4	1.0	270.03	135.0	205.860000	106.000000	2.006522	0
5	59.5	53733.04	30420.5	662.862243	466.954360	1.623470	5
6	1.0	4314.72	4300.0	3096.000000	2672.000000	1.135169	0

Labels	Revenue	Quantity	InvoiceNo	RevenueMean	QuantityMean	PriceMean	DaysBetweenInvoices	DaysSinceInvoice
0	1017.82	609.5	3.0	316.085600	185.958333	1.714620	19.0	50.0
1	771.17	443.0	3.0	266.606667	150.500000	1.805578	75.0	61.0
2	259657.30	77242.0	60.0	4327.621667	2700.438356	1.600660	6.0	23.0
3	39619.50	61.0	3.0	13206.500000	20.333333	649.500000	0.0	204.0
4	270.03	135.0	1.0	205.860000	106.000000	2.006522	0.0	268.0
5	53733.04	30420.5	59.5	662.862243	466.954360	1.623470	5.5	23.0
6	4314.72	4300.0	1.0	3096.000000	2672.000000	1.135169	0.0	135.0

Labels	Revenue	Quantity	InvoiceNo	RevenueMean	QuantityMean	PriceMean	DaysBetweenInvoices	DaysSinceInvoice
0	1017.82	609.5	3.0	316.085600	185.958333	1.714620	19.0	50.0
1	771.17	443.0	3.0	266.606667	150.500000	1.805578	75.0	61.0
2	259657.30	77242.0	60.0	4327.621667	2700.438356	1.600660	6.0	23.0
3	39619.50	61.0	3.0	13206.500000	20.333333	649.500000	0.0	204.0
4	270.03	135.0	1.0	205.860000	106.000000	2.006522	0.0	268.0
5	53733.04	30420.5	59.5	662.862243	466.954360	1.623470	5.5	23.0
6	4314.72	4300.0	1.0	3096.000000	2672.000000	1.135169	0.0	135.0

Below you can see our results. Your results should match. If they do not match exactly, make sure they match at least qualitatively.

We see that cluster 0 contains the most customers (1520 in total) with `random_state=0`. This therefore appears to be typical customer behavior. They order from us from time to time

(`'InvoiceNo'` = 3) and buy on average a relatively large number of inexpensive individual items (`'QuantityMean'` and `'PriceMean'`). The average revenue they generate is not particularly high (`'RevenueMean'`).

The revenue of cluster 1 is even lower. The median values are similar to those of cluster 3. They just buy a slightly smaller quantity and generate slightly less revenue. The time interval between the individual orders is also larger (`'DaysBetweenInvoices'` = 76).

Another large customer group is Cluster 4. Many of these customers have only placed an order with us once, and that was a long time ago (`'DaysSinceInvoice'` = 268). In each case, the orders only generated a small amount of revenue (`'RevenueMean'`).

Clusters 5 and 6 had only 12 and 15 customer numbers. Group 5 is made up of customers who order from us very frequently. In doing so, they buy a lot of cheap items and generate a lot of revenue. Customers from group 6 very rarely buy from us, but have placed relatively large orders. It might be worthwhile to activate them with advertising.

We can summarize the entire analysis as follows:

The majority of our customers buy from us relatively rarely. The orders placed in this process generate relatively little turnover when viewed individually. These customers could perhaps be persuaded by advertising to buy from us again.

Groups 2, 3, 4, 5 and 6 contain profitable customers.

- Group 2 includes our most lucrative customers. These place orders extremely frequently and generate an extremely high revenue (`'InvoiceNo'` >= 60 and `'Revenue'` ~ 200000). They have integrated our shop as permanent fixture. We should definitely keep these customers interested and active, e.g. by providing very good support for inquiries.
- The individual customer in group 3 placed few orders and ordered very expensive items, which is very unusual compared to the other customers. It might be worthwhile to get this customer's attention again.
- Group 6 is very similar to group 5, but does not generate quite so much revenue. However, these customers should also be prioritized to make sure that we don't lose them.

Group 4 placed relatively large orders, but only once. These people mainly bought very cheap items. Maybe a discount campaign can persuade them to shop with us again.

Congratulations: Your employer is impressed with your quick analysis! By preparing the order data, you created features that help you perform the cluster analysis. You used the *elbow method* to find a reasonable number of clusters. The analysis of the customer groups brings out some characteristics of these groups which indicate which customers should be prioritized. And which customers should be reactivated by advertising measures. The marketing department will now take care of the exact measures. In the next exercise we will look at another method of evaluating clusters and their number.

Remember:

- The `KMeans` score describes the quality of the clustering with the average sum of the squared Euclidean distances from the data points to the cluster centers
- Make sure features are on the same scale before clustering
- Determine the number of clusters with the *elbow method*
- Make results reproducible by using `random_state`

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.

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