Feature Engineering for Customer Segmentation

Module 1 | Chapter 3 | Notebook 4

In the last exercise, we cleaned up and prepared our order data so that it's now available based on individual customers. In this exercise, we will continue preparing the data and we'll generate even more features that we can then use for the cluster analysis. By the end of this lesson you will be able to:

- Import DataFrames from the *pickle* data type
- Use tuples
- Deal with hierarchical column names
- Merge DataFrames

Creating features with time 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 bene provided with data containing all orders and cancellations in a one year period, in order to gain some initial insights. You have already merged and aggregated the data.

In the last exercise you saved the data as a *pickle*. This is a file type for objects in Python. Compared to a *CSV*, it has the advantage that you can save the objects just as they are in the program. With DataFrames, for example, this means that the data types of the columns are already set. To import a *pickle* you can use the `pd.read_pickle()` function from `pandas`. The function only needs the name of the file.

You saved the `DataFrame` `df_customers` in the file `customer_data.p`. Import it and store it as `df_customers` again. Then print the first five rows.

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

```
df_customers.head()
```

Out[1]:

	InvoiceNo	Revenue	Quantity	RevenueMean	QuantityMean	PriceMean
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

The rows indicate the customer numbers. 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

We also need the original data. Run the following code cell to import it and remove the missing values:

In [2]:
```
df = pd.read_csv('online_retail_data.csv', parse_dates = ['InvoiceDate'])
df = df.dropna()
df.loc[:, 'CustomerID'] = df.loc[:, 'CustomerID'].astype(int)
```

The columns are explained in this data dictionary:

Column number	Column name	Type	Description
0	`'InvoiceNo'`	categorical/nominal	the order number
1	`'StockCode'`	categorical/nominal	the item number
2	`'Description'`	text (`str`)	Description of the item
3	`'Quantity'`	continuous (`int`)	Number of ordered items (negative for cancellations)
4	`'InvoiceDate'`	continuous (`datetime`)	time and date of the order
5	`'UnitPrice'`	continuous (`float`)	item price in GBP
6	`'CustomerID'`	categorical/nominal (`int`)	customer number

Column number	Column name	Type	Description
7	`'Country'`	categorical/nominal	The country the order came from

Now we will focus on the task of creating new features from the time entries in the `'InvoiceDate'` column. Specifically, we will calculate the average time interval between orders and the time elapsed since the last order. For this we need each person's first and last activity. We can find these with `my_groups.agg()` again. To do this, we present a `dict` with the *values* `'first'` and `'last'`. This time we only need one column `'InvoiceDate'`, so we only have one *key* in the `dict`. This *key* has 2 *values*, so you have to put `'first'` and `'last'` in a list.

Group `df` by `'CustomerID'` and aggregate the data so that you get the first and last day. Store the result in a `DataFrame` named `df_customers_date` and print the first 5 rows.

In [3]:
```
groups = df.groupby('CustomerID')
dictionary = {'InvoiceDate': ['first', 'last']}
df_customers_date = df.groupby('CustomerID').agg(dictionary)

df_customers_date.head()
```

Out[3]:

	InvoiceDate	
	first	**last**
CustomerID		
12346	2011-01-18 10:01:00	2011-01-18 10:01:00
12347	2010-12-07 14:57:00	2011-12-07 15:52:00
12348	2010-12-16 19:09:00	2011-09-25 13:13:00
12349	2011-11-21 09:51:00	2011-11-21 09:51:00
12353	2011-05-19 17:47:00	2011-05-19 17:47:00

The resulting `DataFrame` now has two levels of column names. The first level corresponds to the column we aggregated. The second level corresponds to the functions we used for this purpose. If a `DataFrame` has a hierarchical naming structure, we call this a `MultiIndex`. How do you access this? Try it like you did before with `df_customers_date.loc[:, 'InvoiceDate']`.

In [4]:
```
df_customers_date.loc[:, 'InvoiceDate']
```

Out[4]:

	first	last
CustomerID		
12346	2011-01-18 10:01:00	2011-01-18 10:01:00
12347	2010-12-07 14:57:00	2011-12-07 15:52:00
12348	2010-12-16 19:09:00	2011-09-25 13:13:00
12349	2011-11-21 09:51:00	2011-11-21 09:51:00
12353	2011-05-19 17:47:00	2011-05-19 17:47:00
...	...	...
18276	2011-10-27 10:54:00	2011-10-27 10:54:00
18277	2011-10-12 15:22:00	2011-10-12 15:22:00
18278	2011-09-27 11:58:00	2011-09-27 11:58:00
18280	2011-03-07 09:52:00	2011-03-07 09:52:00
18281	2011-06-12 10:53:00	2011-06-12 10:53:00

2877 rows × 2 columns

Now the column naming structure only has one level. So by chaining `my_df.loc[]` we can select a single column. The column `'last'` is obtained, for example, by using `df_customers_date.loc[:, 'InvoiceDate'].loc[:, 'last']`.

However, it's more elegant to use what is called a tuple. A tuple is an arrangement of values and is enclosed by round parentheses. `('InvoiceDate', 'last')` is a tuple with the values `'InvoiceDate'` and `'last'`. Unlike lists, tuples do not offer methods. The values in them cannot be changed either, they can only be accessed. You do this with square brackets just like you do with lists. We can use a tuple to directly select the relevant column. There are other ways to access a `MultiIndex`. If you are interested, then have a look at the official documentation.

Important: We need the tuple, because lists are already used for the column names in DataFrames. You can use lists to select columns that are equal and next to each other: `my_df.loc[:, [col_1, col_2]]`. However, you can use tuples to select columns that follow a hierarchy: `my_df.loc[:, (col_upper, col_lower)]`.

Now let's return to our features. Calculate the average time between orders.

To do this, subtract the `'first'` column from the `'last'` column. This way you get the whole time period of one person's orders. Divide this by the number of orders. The number is located in the `'InvoiceNo'` column of `df_customers`. Since we have already removed rows in `df_customers`, you can use `my_df.div()` with `fill_value=1` just like with `my_df.add()`. We need the 1 here as a fill value, because the number of orders has to be in the divider (within the method call), because you can't divide something by 0.

Store the result as a column `'DaysBetweenInvoices'` in `df_customers_date` and print it.

Remember to use tuples for the `'first'` and `'last'` columns!

In [5]:
```
df_customers_date.loc[:, 'DaysBetweenInvoices'] = df_customers_date.loc[:, 'InvoiceDat
df_customers_date.loc[:, 'DaysBetweenInvoices'] = df_customers_date.loc[:, 'DaysBetwee
df_customers_date.head()
```

Out[5]:

	InvoiceDate		DaysBetweenInvoices
	first	last	
CustomerID			
12346	2011-01-18 10:01:00	2011-01-18 10:01:00	0 days 00:00:00
12347	2010-12-07 14:57:00	2011-12-07 15:52:00	52 days 03:33:34.285714286
12348	2010-12-16 19:09:00	2011-09-25 13:13:00	70 days 16:31:00
12349	2011-11-21 09:51:00	2011-11-21 09:51:00	0 days 00:00:00
12353	2011-05-19 17:47:00	2011-05-19 17:47:00	0 days 00:00:00

The first value should be something like `0 days 00:00:00`, the second is `52 days 03:33:34.285714286`.

We are satisfied with an accuracy in whole days. You can make use of the *datetime* functionality by using the column's `dt` attribute, see *Automatically Detecting Room Occupancy* (Module 1 Chapter 2). This allows you to use the `days` attribute. It returns the time intervals as days. Store them again in the `'DaysBetweenInvoices'` column.

In [6]:
```
df_customers_date.loc[:, 'DaysBetweenInvoices'] = df_customers_date.loc[:, 'DaysBetwee
```

Now all that's missing is the time since the last order. The period of our data starts with 01.12.2010 and ends with 09.12.2011. It therefore relates to just over a year and was recorded at the end of the year. Now let's pretend that it's 01.01.2012. If our analysis will always take place at the end of the year so that it can be used at the beginning of the next year, then this assumption makes sense. Run the following code cell to create the `date_now` variable. It contains the date.

In [7]:
```
date_now = pd.to_datetime('2012-01-01 00:00:00')
```

Now subtract `df_customers_date.loc[:, ('InvoiceDate', 'last')]` from `date_now`. Save the result as a new column named `'DaysSinceInvoice'` in `df_customers_date` and print the first value.

Tip: Since `date_now` is only a fixed value, you can use the minus sign `-` directly. It doesn't generate any missing values.

In [8]:
```
df_customers_date.loc[:, 'DaysSinceInvoice'] = date_now - df_customers_date.loc[:, ('I
df_customers_date.head()
```

Out[8]:

	InvoiceDate		DaysBetweenInvoices	DaysSinceInvoice
	first	last		
CustomerID				
12346	2011-01-18 10:01:00	2011-01-18 10:01:00	0	347 days 13:59:00
12347	2010-12-07 14:57:00	2011-12-07 15:52:00	52	24 days 08:08:00
12348	2010-12-16 19:09:00	2011-09-25 13:13:00	70	97 days 10:47:00
12349	2011-11-21 09:51:00	2011-11-21 09:51:00	0	40 days 14:09:00
12353	2011-05-19 17:47:00	2011-05-19 17:47:00	0	226 days 06:13:00

For the first value you should get `347 days 13:59:00`. Only access the full days of `'DaysSinceInvoice'` and overwrite this column with them.

In [9]:
```
df_customers_date.loc[:, 'DaysSinceInvoice'] = df_customers_date.loc[:, 'DaysSinceInvc
```

We can use `'DaysBetweenInvoices'` to estimate whether our customers place orders very regularly, and `'DaysSinceInvoice'` can give us an indication as to whether they should be contacted again because they haven't ordered anything for a long time. We only need to add these two properties to `df_customers`, then we have finished preparing the data.

`pandas` offers various functions for this purpose. The `pd.merge()` function is very flexible and can therefore be used in many situations. You can find the documentation here. We will use it here too. You pass it two DataFrames. The parameters for this are called `left` and `right`. Then we determine which values should be compared with each other. In our case these are the row names. The rows of both DataFrames should refer to the same customer number. You can ensure this by setting the parameters `left_index` and `right_index` to `True`. We have fewer rows in `df_customers` than in `df_customers_date`. But we don't need the other rows, we removed them for a reason. So we only want the values for the row names of the first (left) `DataFrame`. That's why we put `how='left'`. So we need the following parameter values:

- `left=df_customers`
- `right=df_customers_date.loc[:, ['DaysBetweenInvoices', 'DaysSinceInvoice']]`
- `left_index=True`
- `right_index=True`
- `how='left'`

Make sure you only add the columns you need. Call the new `DataFrame` `df_customers`.

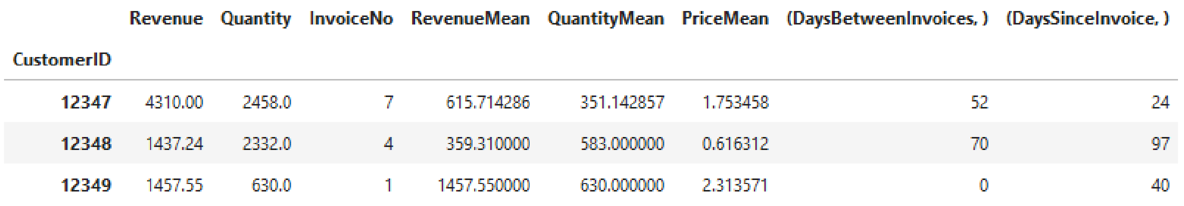
Tip: You will receive a *UserWarning* that *merging* DataFrames with different index levels can lead to unintended results. Since in our case we have no values in `df_customers_date` in the second level of the multi-index, `pd.merge()` will show exactly the desired behavior.

In [10]:
```python
df_customers =pd.merge(df_customers,
                       df_customers_date.loc[:, ['DaysBetweenInvoices', 'DaysSinceInvc
                       left_index=True,
                       right_index=True,
                       how='left')
df_customers.head()
```

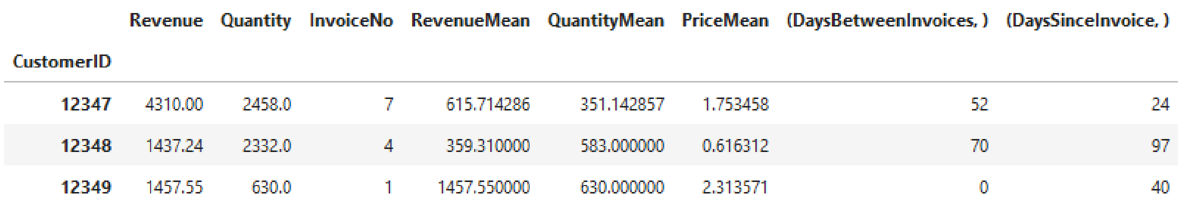
```
/home/jovyan/.virtualenvs/training_env/lib/python3.8/site-packages/pandas/core/reshap
e/merge.py:648: UserWarning: merging between different levels can give an unintended
result (1 levels on the left,2 on the right)
  warnings.warn(msg, UserWarning)
```

Out[10]:

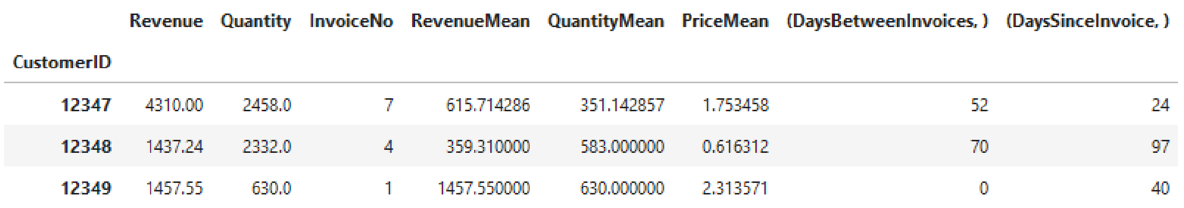
	InvoiceNo	Revenue	Quantity	RevenueMean	QuantityMean	PriceMean	(DaysBetweenIr
CustomerID							
12347.0	7	4310.00	2458	615.714286	351.142857	1.753458	
12348.0	4	1437.24	2332	359.310000	583.000000	0.616312	
12349.0	1	1457.55	630	1457.550000	630.000000	2.313571	
12353.0	1	89.00	20	89.000000	20.000000	4.450000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



	Revenue	Quantity	InvoiceNo	RevenueMean	QuantityMean	PriceMean	(DaysBetweenInvoices,)	(DaysSinceInvoice,)
CustomerID								
12347	4310.00	2458.0	7	615.714286	351.142857	1.753458	52	24
12348	1437.24	2332.0	4	359.310000	583.000000	0.616312	70	97
12349	1457.55	630.0	1	1457.550000	630.000000	2.313571	0	40



	Revenue	Quantity	InvoiceNo	RevenueMean	QuantityMean	PriceMean	(DaysBetweenInvoices,)	(DaysSinceInvoice,)
CustomerID								
12347	4310.00	2458.0	7	615.714286	351.142857	1.753458	52	24
12348	1437.24	2332.0	4	359.310000	583.000000	0.616312	70	97
12349	1457.55	630.0	1	1457.550000	630.000000	2.313571	0	40



	Revenue	Quantity	InvoiceNo	RevenueMean	QuantityMean	PriceMean	(DaysBetweenInvoices,)	(DaysSinceInvoice,)
CustomerID								
12347	4310.00	2458.0	7	615.714286	351.142857	1.753458	52	24
12348	1437.24	2332.0	4	359.310000	583.000000	0.616312	70	97
12349	1457.55	630.0	1	1457.550000	630.000000	2.313571	0	40

The first three rows of the merged `DataFrame` should look something like this:

Because `df_customers_date` has several levels in the row names, you will get a warning `UserWarning: merging between different levels can give an unintended result (1 levels on the left, 2 on the right)` when you merge for the first time. You can ignore this, the result is what we wanted. Our column names also only have one level. But we now have two column names that are enclosed in round brackets with a comma. This looks very strange.

Overwrite the column names `('DaysBetweenInvoices', )` and `('DaysSinceInvoice', )` with the names `'DaysBetweenInvoices'` and `'DaysSinceInvoice'`. The column names are located in the `my_df.columns` attribute. You can directly assign this attribute a list of the desired names. Then look at the first five lines of the DataFrame.

In [13]:
```
df_customers.columns =[
    'InvoiceNo',
    'Revenue',
    'Quantity',
    'RevenueMean',
    'QuantityMean',
    'PriceMean',
    'DaysBetweenInvoices',
    'DaysSinceInvoice'
]
df_customers.head()
```

Out[13]:
```
            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
```

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 (`float`)	total number of items purchased
2	`'Quantity'`	continuous (`float`)	total number of items purchased
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
6	`'DaysBetweenInvoices'`	continuous (`int`)	average number of days between orders

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

This completes the feature engineering. Save the DataFrame for later use as a pickle file called `customer_data_prepared.p`.

In [15]:
```
df_customers.info()
df_customers.to_pickle('customer_data_prepared.p')
```

```
<class 'pandas.core.frame.DataFrame'>
Index: 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: 266.3+ KB
```

Congratulations: You have merged and aggregated the data from the last exercise. You have added two more features, based on the date values. Now you have data that can be used for customer segmentation. It provides information about how much revenue customers generate, whether they order regularly, whether they prefer cheap goods and how long it has been since they last ordered something. In the next exercise you will use k-Means to divide the customers into groups.

Remember:

- Import a `DataFrame` from a pickle with `pd.read_pickle()`
- Use a tuple to select hierarchical columns in a `DataFrame`
- Divide DataFrames with different lengths with `my_df.div()`
- Merge DataFrames with `df = pd.merge(df_left, df_right, left_index=True, right_index=True, how='left')`

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