Programming assignment #3

Course: CHE1147H - Data Mining in Engineering

1 Feature engineering

Here, you are going to create features from a very simple dataset: retail transaction data from Kaggle. The dataset provides the customer ID, date of the transaction and transaction amount as shown in the table below. Although this may look like a very simple dataset, you will build a wide range of features. The features will then be used as inputs in several models in upcoming assignments, in which you will try to predict the client's response to a promotion campaign.

	customer_id	trans_date	tran_amount
0	CS5295	11-Feb-13	35
1	CS4768	15-Mar-15	39
2	CS2122	26-Feb-13	52
3	CS1217	16-Nov-11	99
4	CS1850	20-Nov-13	78

1.1 Import the data and create the anchor date columns

In order to create features, you need to create some anchor dates. The most typical for transaction data is the end of the month and the year.

1. Import the dataset as **txn**[1] and identify the number of rows[2].
2. The date-format in column 'trans_date' is not standard. Create a new column 'txn_date' from 'trans_date' with pd.to_datetime and drop the column 'trans_date'.
3. Identify the min() and max() of column 'txn_date'.

[1] Bold indicates dataset or Python object. Single quotation mark indicates column.
[2] When necessary use the Markdown language in your notebook to answer the questions.

4. Create the column 'ME_DT': the last day of the month in the 'trans_date' column. DateOffset objects is a simple way to do this in pandas.

5. Create the column 'YEAR': the year in the 'trans_date' column. DatetimeIndex with attribute .year will help you do so.

The **table output** should look like the snapshot below. Make sure that the column 'ME_DT' works as expected. E.g. for the first line 'trans_date': 2018-08-31 is converted to 2018-08-31. A common mistake in implementing the DateOffset transformation is to convert 2018-08-31 to 2018-09-30 (a date that falls on the last day of a month is converted to the last day of the next month!!!).

	customer_id	tran_amount	txn_date	ME_DT	YEAR
55	CS2662	88	2014-08-31	2014-08-31	2014
56	CS2209	35	2012-03-12	2012-03-31	2012
57	CS4530	40	2011-06-05	2011-06-30	2011
58	CS2848	53	2013-02-04	2013-02-28	2013
59	CS2596	55	2011-09-19	2011-09-30	2011

1.2 Create features that capture annual spending

Here the approach is to capture the client's annual spending. The rationale behind this approach is that the clients spend is not very frequent to capture in a monthly aggregation.

1. Using groupby and NamedAgg create **clnt_annual_aggregations**, the annual aggregations dataframe: with sum, mean, std, var, sem, max, min, count as the aggregation functions. A snapshot of the output table is shown below. Notice that the output is a typical MultiIndex pandas dataframe.

		ann_txn_amt_sum	ann_txn_amt_ave	ann_txn_amt_std	ann_txn_amt_var	ann_txn_amt_sem	ann_txn_amt_max	ann_txn_amt_min	ann_txn_cnt
customer_id	YEAR								
CS1112	2011	212	70.666667	22.030282	485.333333	12.719189	96	56	3
	2012	337	67.400000	12.720063	161.800000	5.688585	81	52	5
	2013	212	70.666667	34.501208	1190.333333	19.919282	105	36	3
	2014	212	70.666667	16.862186	284.333333	9.735388	90	59	3
	2015	39	39.000000	NaN	NaN	NaN	39	39	1

2. Plot the histogram of the sum and count.

3. Reset the index and reshape the table with the pivot_table function to create the **clnt_annual_aggregations_pivot** table shown below with 40 columns (why 40?).

   You should expect columns with NaN values. Impute the NaN entries when you perform the pivot_table function and explain your choice of values.

	ann_txn_amt_ave					ann_txn_amt_max					...
YEAR	2011	2012	2013	2014	2015	2011	2012	2013	2014	2015	...
customer_id											
CS1112	70.666667	67.400000	70.666667	70.666667	39.000000	96	81	105	90	39	...
CS1113	81.333333	74.800000	85.200000	56.500000	73.333333	94	95	97	97	98	...
CS1114	85.200000	75.000000	70.400000	70.833333	79.000000	97	97	105	95	79	...
CS1115	87.000000	67.571429	79.571429	78.250000	55.000000	102	104	94	98	55	...
CS1116	58.750000	76.666667	59.000000	66.600000	0.000000	87	105	59	96	0	...

4. The pivoted object you created is a MultiIndex object with hierarchical indexes. You can see the first level (i.e. 0) in the snapshot above with names 'ann_txn_amt_ave', 'ann_txn_amt_max' (and more as indicated by the ...) and the second level (i.e. 1) with names '2011', '2012', etc. You can confirm the multiple levels of the columns with the following two expressions.

   What are your observations regarding the number of levels and the column names?

```
clnt_annual_aggregations_pivot.columns.nlevels
clnt_annual_aggregations_pivot.columns
```

5. Finally, you want to save the dataframe **clnt_annual_aggregations_pivot** as an .xlsx file for future use in the machine learning assignment. To do so, you want to remove the two levels in columns and create a single level with column names: 'ann_txn_amt_ave_2011', 'ann_txn_amt_ave_2012', etc. To do so, use the code snippet below prior to saving the dataframe as an Excel file.

```
level_0 = clnt_annual_aggregations_pivot.columns.get_level_values(0).
                                                astype(str)
level_1 = clnt_annual_aggregations_pivot.columns.get_level_values(1).
                                                astype(str)
clnt_annual_aggregations_pivot.columns = level_0 + '_' + level_1
```

   Describe what each line of code in the box does and save the output dataframe as an Excel file **annual_features.xlsx**. A snapshot of the desired final output is shown below.

	ann_txn_amt_ave_2011	ann_txn_amt_ave_2012	ann_txn_amt_ave_2013	ann_txn_amt_ave_2014	ann_txn_amt_ave_2015	ann_txn_amt_max_2011
customer_id						
CS1112	70.666667	67.400000	70.666667	70.666667	39.000000	96
CS1113	81.333333	74.800000	85.200000	56.500000	73.333333	94
CS1114	85.200000	75.000000	70.400000	70.833333	79.000000	97
CS1115	87.000000	67.571429	79.571429	78.250000	55.000000	102
CS1116	58.750000	76.666667	59.000000	66.600000	0.000000	87
...	...	...	...	...	...	...

6. What are the possible disadvantages in capturing client transaction behavior with the annual features described in this section (if any)?

1.3 Create monthly aggregations

Here, you want to explore the **monthly** sum of amounts and count of clients transactions.

1. Create the dataframe that captures the monthly sum and count of transactions per client (name it **clnt_monthly_aggregations**). Use the groupby function with the Named Aggregation feature which was introduced in pandas version 0.25.0. Make sure that you name the columns as shown in the figure sample on the right.

2. Create a histogram of both columns you created. What are your observations? What are the most common and maximum values for each column? How do they compare with the ones in section 1.2?

ME_DT	mth_txn_amt_sum	mth_txn_cnt
2011-06-30	56	1
2011-08-31	96	1
2011-10-31	60	1
2012-04-30	56	1
2012-06-30	52	1
2012-07-31	81	1
2012-09-30	72	1
2012-12-31	76	1
2013-03-31	105	1
2013-07-31	36	1
2013-11-30	71	1
2014-04-30	63	1
2014-07-31	90	1
2014-12-31	59	1
2015-01-31	39	1

The **output dataframe** should look like the snapshot shown on the right for client with ID CS1112 (confirm this with slicing your output dataframe).

Most clients in this dataset shop a few times a year. For example, the client with 'customer_id' CS1112 shown here made purchases in 15 out of 47 months of data in the txn table. The information in this dataset is "irregular"; some clients may have an entry for a month, while others do not have an entry (e.g. when they don't shop for this particular month).

1.4 Create the base table for the rolling window features

In order to create the rolling window features (more on this in the next section), you need to create a **base table** with **all** possible combinations of 'customer_id' and 'ME_DT'. For example, customer CS1112 should have 47 entries, one for each month, in which 15 will have the value of transaction amount and the rest 32 will have zero value for transaction amount. This will essentially help you convert the "irregular" **clnt_monthly_aggregations** table into a "regular" one.

1. Create the numpy array of the unique elements in columns 'customer_id' and 'ME_DT' of the **txn** table you created in section 1.1. Confirm that you have 6,889 unique clients and 47 unique month-end-dates.

2. Use itertools.product to generate all the possible combinations of 'customer_id' and 'ME_DT'. Itertools is a Python module that iterates over data in a computationally efficient way. You can perform the same task with a for-loop, but the execution

may be inefficient. For a brief overview of the Itertools module see here. If you named the numpy arrays with the unique elements: **clnt_no** and **me_dt**, then the code below will create an itertools.product object (you can confirm this by running: `type(base_table)`).

```
from itertools import product
base_table = product(clnt_no, me_dt)
```

3. Next, you want to convert the itertools.product object **base_table** into a pandas object called **base_table_pd**. To do so, use pd.DataFrame.from_records and name the columns 'CLNT_NO' and 'ME_DT'.

4. Finally, you want to validate that you created the table you originally wanted. There are two checks you want to perform:

   - Filter client CS1112 and confirm that the dates fall between the min and max month-dates you identified in section 1.1. Also, confirm that the snapshot of client CS1112 has 47 rows, one for each month in the dataset.
   - Confirm that the **base_table_pd** has 323,783 rows, which is the expected value of combinations for 6,889 unique clients and 47 unique month-end dates.

1.5 Create the monthly rolling window features

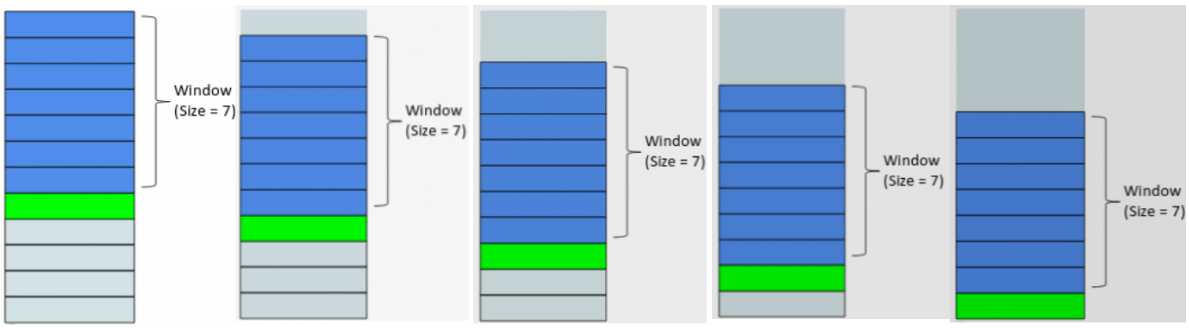
With the **base_table_pd** as a starting point you can convert the irregular transaction data into the typical **time series** data; data captured at equal intervals. Feature engineering of time series data gives you the potential to build very powerful predictive models.

1. Left-join the **base_table_pd** with the **clnt_monthly_aggregations** table from section 1.3 on [CLNT_NO, ME_DT] to create the table **base_clnt_mth**. Comment on the following questions in Markdown:

   - Why do some rows have NaN values?
   - What values will you choose to impute NaN values in the sum and count columns? Perform the imputation you suggest.
   - Confirm that the number of rows is what you expect. What is the value?
   - How are tables **base_clnt_mth** and **clnt_monthly_aggregations** different? Comment on the number of rows and the content of each table.

2. For the next step, the calculation of the rolling window features, you need to sort the data first by 'CLNT_NO' and then by 'ME_DT' in ascending order. This is necessary to create the order for rolling windows, e.g. 2011-05-31, 2011-06-30, etc.

3. The idea behind rolling window features is captured in the image below. You calculate some statistical properties (e.g. average) based on a window that is sliding. In the image below, the window is 7 which means that the last 7 points are used at every row to calculate the statistical property.



Here, you have to calculate separately the 3, 6 and 12-month rolling window features (tables: **rolling_features_3M, rolling_features_6M, rolling_features_12M**) for every client that calculates the aggregations 'sum', mean' and 'max' for both columns 'mth_txn_amt_sum' and 'mth_txn_cnt'. The steps to achieve this with **base_clnt_mth** as the starting dataframe are:

- groupby the client number
- select the two columns you want to aggregate
- use the rolling function with the appropriate windows
- aggregate with 'sum', mean' and 'max'

The output of the 3-month rolling window dataframe is shown below. Also, answer the following questions in the .ipynb notebook as Markdown comments.

		mth_txn_amt_sum			mth_txn_cnt		
		sum	mean	max	sum	mean	max
CLNT_NO							
CS1112	157064	NaN	NaN	NaN	NaN	NaN	NaN
	157058	NaN	NaN	NaN	NaN	NaN	NaN
	157044	56.0	18.666667	56.0	1.0	0.333333	1.0
	157069	152.0	50.666667	96.0	2.0	0.666667	1.0
	157060	96.0	32.000000	96.0	1.0	0.333333	1.0

- How many rows appear with NaN values at the beginning of each client for 3, 6 and 12-month windows, respectively? Why do they appear?
- How many levels do the index and columns have? Are these MultiIndex dataframes?

- Rename the columns as following: 'amt_sum_3M', 'amt_mean_3M', 'amt_max_3M', 'txn_cnt_sum_3M', 'txn_cnt_mean_3M', 'txn_cnt_max_3M' and follow the same naming convention for 6M and 12M.

4. Merge the 4 tables: **base_clnt_mth, rolling_features_3M, rolling_features_6M, rolling_features_12M** in the output **all_rolling_features**. It is recommended to drop the level:0 of the rolling features MultiIndex table and join with **base_clnt_mth** on the indexes.

   Make sure you understand why joining on the indexes preserves the CLNT_NO and ME_DT for each index.

5. Confirm that your final output **all_rolling_features** has 323,783 rows and 22 columns and save it as **mth_rolling_features.xlsx**.

1.6 Date-related features: date of the week

In this section, you will create the date-related features that capture information about the day of the week the transactions were performed.

1. The DatetimeIndex object you used earlier allows you to extract many components of a DateTime object. Here, you want to use the attributes **dt.dayofweek** and/or **dt.day_name()** to extract the day of the week from column 'txn_date' of the **txn** table (with Monday=0, Sunday=6). The expected output below shows both columns.

	customer_id	tran_amount	txn_date	ME_DT	YEAR	day_of_the_week	day_name
0	CS5295	35	2013-02-11	2013-02-28	2013	0	Monday
1	CS4768	39	2015-03-15	2015-03-31	2015	6	Sunday
2	CS2122	52	2013-02-26	2013-02-28	2013	1	Tuesday
3	CS1217	99	2011-11-16	2011-11-30	2011	2	Wednesday

2. Plot the histogram that shows the count of transactions per day of the week.

3. Following the same logic as in section 1.2, generate the features that capture the count of transactions per client, year and day of the week. The intermediate MultiIndex dataframe (with nlevels=3) and the final pivoted output with a single index are shown in the snapshots below.

YEAR							2011			2012	...			2014							cnt 2015
day_name	Friday	Monday	Saturday	Sunday	Thursday	Tuesday	Wednesday	Friday	Monday	Saturday	...	Thursday	Tuesday	Wednesday	Friday	Monday	Saturday	Sunday	Thursday	Tuesday	Wednesday
customer_id																					
CS1112	1	0	0	1	0	0	1	0	0	1	...	1	1	1	0	0	0	0	0	0	1
CS1113	1	1	0	1	0	0	0	2	1	0	...	0	3	0	0	2	0	0	0	0	1
CS1114	0	1	0	1	1	0	2	0	0	1	...	0	1	1	0	0	0	0	1	0	0
CS1115	0	0	1	1	0	0	1	1	1	0	...	2	0	2	0	0	0	0	1	0	0
CS1116	1	1	0	0	1	1	0	1	0	0	...	0	0	0	0	0	0	0	0	0	0
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
CS8996	0	1	0	0	0	1	0	0	0	3	...	1	1	2	0	0	0	0	0	0	0
CS8997	0	0	0	0	1	0	1	1	2	0	...	0	0	0	0	0	0	0	0	0	0
CS8998	0	0	0	0	0	1	1	2	0	0	...	0	1	1	0	0	0	0	0	0	0
CS8999	0	1	1	0	0	1	0	1	0	2	...	0	0	2	0	0	0	0	0	0	0
CS9000	0	0	0	2	0	0	0	1	0	1	...	1	1	0	0	0	1	0	0	0	0

	cnt_2011_Friday	cnt_2011_Monday	cnt_2011_Saturday	cnt_2011_Sunday	cnt_2011_Thursday	cnt_2011_Tuesday	cnt_2011_Wednesday	cnt_2012_Friday	cnt_2012_Monday	cnt_2012_Saturday	...	cnt_2014_Thursday
customer_id												
CS1112	1	0	0	1	0	0	1	0	0	1	...	1
CS1113	1	1	0	1	0	0	0	2	1	0	...	0
CS1114	0	1	0	1	1	0	2	0	0	1	...	0
CS1115	0	0	1	1	0	0	1	1	1	0	...	2
CS1116	1	1	0	0	1	1	0	1	0	0	...	0
...	...	...	...	...	...	...	...	...	...	...	...	...
CS8996	0	1	0	0	0	1	0	0	0	3	...	1
CS8997	0	0	0	0	1	0	1	1	2	0	...	0
CS8998	0	0	0	0	0	1	1	2	0	0	...	0
CS8999	0	1	1	0	0	1	0	1	0	2	...	0
CS9000	0	0	0	2	0	0	0	1	0	1	...	1

4. Confirm that your output has the same number of rows as the final output in section 1.2 and save it as **annual_day_of_week_counts_pivot.xlsx**. How many features/columns did you create in this section?

5. Similarly, generate the features that capture the count of transactions per client, month-end-date and day of the week. In contrast with the annual pivot table in the previous step, here you want to create the pivot with ['customer_id', 'ME_DT'] as index to obtain the following output dataframe.

		cnt_Friday	cnt_Monday	cnt_Saturday	cnt_Sunday	cnt_Thursday	cnt_Tuesday	cnt_Wednesday
customer_id	ME_DT							
CS1112	2011-06-30	0	0	0	0	0	0	1
	2011-08-31	1	0	0	0	0	0	0
	2011-10-31	0	0	0	1	0	0	0
	2012-04-30	0	0	0	1	0	0	0
	2012-06-30	0	0	0	1	0	0	0

6. Join with **base_table_pd** as you did in section 1.5 and impute with your choice of value for NaN. Save the final output as **mth_day_counts.xlxs**.

1.7 Date-related features: days since last transaction

In this date-related features set, you want to capture the frequency of the transactions in terms of the days since the last transaction. This set of features applies only to the monthly features.

1. The starting point is again the **txn** table. Recall that most clients have a single purchase per month, but some clients have multiple purchases in a month. Since you want to calculate the "days since last transaction", you want to capture the last transaction in a month for every client.

   Use the appropriate groupby to create the table **last_monthly_purchase** that captures the last 'txn_date' (aggfunc=max) for every client and month.

2. Join **base_table_pd** with **last_monthly_purchase** as you did in section 1.5. The snapshot below shows the output of the created object **last_monthly_purchase_base** for client CS1112 who made her/his first purchase on June 2011, then no purchase on July and made a purchase again on August 2011. What values will you use to impute the NaT values here? NaT stands for "Not a Timestamp".

	CLNT_NO	ME_DT	last_monthly_purchase
157064	CS1112	2011-05-31	NaT
157058	CS1112	2011-06-30	2011-06-15
157044	CS1112	2011-07-31	NaT
157069	CS1112	2011-08-31	2011-08-19
157060	CS1112	2011-09-30	NaT
157048	CS1112	2011-10-31	2011-10-02
157029	CS1112	2011-11-30	NaT
157049	CS1112	2011-12-31	NaT

3. To answer the imputation problem, we have to think what value should we use for say July 2011 for 'last_monthly_purchase'? The answer is that in July the value for the last monthly purchase is the previous line value: 2011-06-15. In other words, for every client we want to forward-fill the NaT values.

   While pandas fillna() method has a method to forward-fill, here we want to use the apply and a lambda function with the forward-fill function ffill(), with the following expression: `.apply(lambda x: x.ffill())` applied on object **last_monthly_purchase_base** grouped by CLNT_NO. Below, I am showing a snapshot for lines [92:98] that confirm the transition between clients CS1113 and CS1114.

   You can also recreate the forward-fill with the fillna() method, however there is a disadvantage and a reason the `.apply()` method is preferred here.

	CLNT_NO	ME_DT	last_monthly_purchase
160011	CS1113	2015-02-28	2015-02-09
159989	CS1113	2015-03-31	2015-02-09
101134	CS1114	2011-05-31	NaT
101128	CS1114	2011-06-30	NaT
101114	CS1114	2011-07-31	2011-07-14

The forward-fill on the grouped by CLNT_NO object is expected to leave NaT values for the first months of every client until they purchase something. The above snapshot confirms that for client CS1114.

4. Subtract the two date columns and convert the output to .dt.days to calculate the column 'days_since_last_txn' as shown in the following snapshot.

	CLNT_NO	ME_DT	last_monthly_purchase	days_since_last_txn
157064	CS1112	2011-05-31	NaT	NaN
157058	CS1112	2011-06-30	2011-06-15	15.0
157044	CS1112	2011-07-31	2011-06-15	46.0
157069	CS1112	2011-08-31	2011-08-19	12.0
157060	CS1112	2011-09-30	2011-08-19	42.0

5. Plot a histogram of the 'days_since_last_txn'. Based on the values you observe in the histogram, impute the remaining NaN values (i.e. for the initial months before a client makes a purchase). Save the columns ['CLNT_NO', 'ME_DT', 'days_since_last_txn'] as **days_since_last_txn.xlsx**.