Feature\_engineering.py

Pavel Sůva, Risk Research & Development, HCI

Documentation for version 1.0

This file is part of Home Credit scoring library. The classes present in feature\_engineering.py are made for automatic preprocessing of data which are stored in the transaction granularity. From the transaction *(by transaction we mean for example: bank account transaction (in transactional data), instalment (in CB or internal behavioral data), call or text message (in telco data), CB query (in CB data) etc.)* it creates aggregations in granularity of observation which enters scorecard development. These aggregations (i.e. features) can be used as predictors in the scorecard then.

# How to use feature\_engineering.py

## Installation

The feature\_engineering.py file is part of scoring library.

Place the folder “scoring” anywhere on your drive, then import it in you python script using this code (where instead of c:/pythonscripts you write the actual path to the directory where you place scoring folder):

import sys

sys.path.insert(0, ' C:/pythonscripts')

import scoring

## Typical application

Typically, the features you want to use in your scorecard development tasks are things like

* time from last application of the same client
* time from last instalment which was paid with more than 15 DPD
* maximal DPD from last 3 months
* average monthly sum of credit transactions from last 12 months
* ratio of count of incoming to count of outgoing calls from last 7 days

You never know which aggregation will be most predictive (last 6 months? 9 months? 12 months?) and you want to create the aggregations as quickly as possible. The feature engineering classes are trying to solve this task efficiently.

The input data which you base your feature engineering on are data in granularity of **transaction**. By transaction we mean a general concept of one event in a chain of similar events *(e.g. instalment in instalment table, one query in CB queries list etc.).*

To this table of transaction, the table with the desired final granularity must be joined. The desired final granularity is usually the granularity of **application** (i.e. granularity of scorecard training set). The join is usually based on ID of the client (but it can be also by POS, SA, same IP address etc.). So the input data consist of:

* application ID
* application time
* transaction time
* transaction properties (dimensions and metrics specifying what kind of transaction it was)
* To this data several methods from feature\_engineering.py can be applied:
* **TimeSinceCalc** which calculates time since last/first transaction which holds given properties. It basically calculates difference of application time and transaction in desired unit (monts, days etc.) as float, and then takes maximum or minimum of the time differences of specified subset of transactions.
* **Slicer** which creates some basic aggregations in granularity of time unit (you can specify whether it should be day, month etc.). The output from slicer will be table in granularity of application ID x time unit index (where x means Cartesian product). For example with 6 months of transactional history, you will get up to 6 rows per each application ID, and the columns in this table can be maximums, minimums, means, sums and other aggregations of the transactions per each month. The output is meant to be processed further.
* **OrderAssigner** which just adds one new column – difference of application time and transaction in desired unit (monts, days etc.) as integer – so the transactions can be grouped by this time unit later. The output is meant to be processed further.

We mentioned that the outputs of Slicer and OrderAssigner are meant to be processed further. The class for this processing is called **FeatureEngineeringFromSlice**. Using the time unit index, this class can calculate:

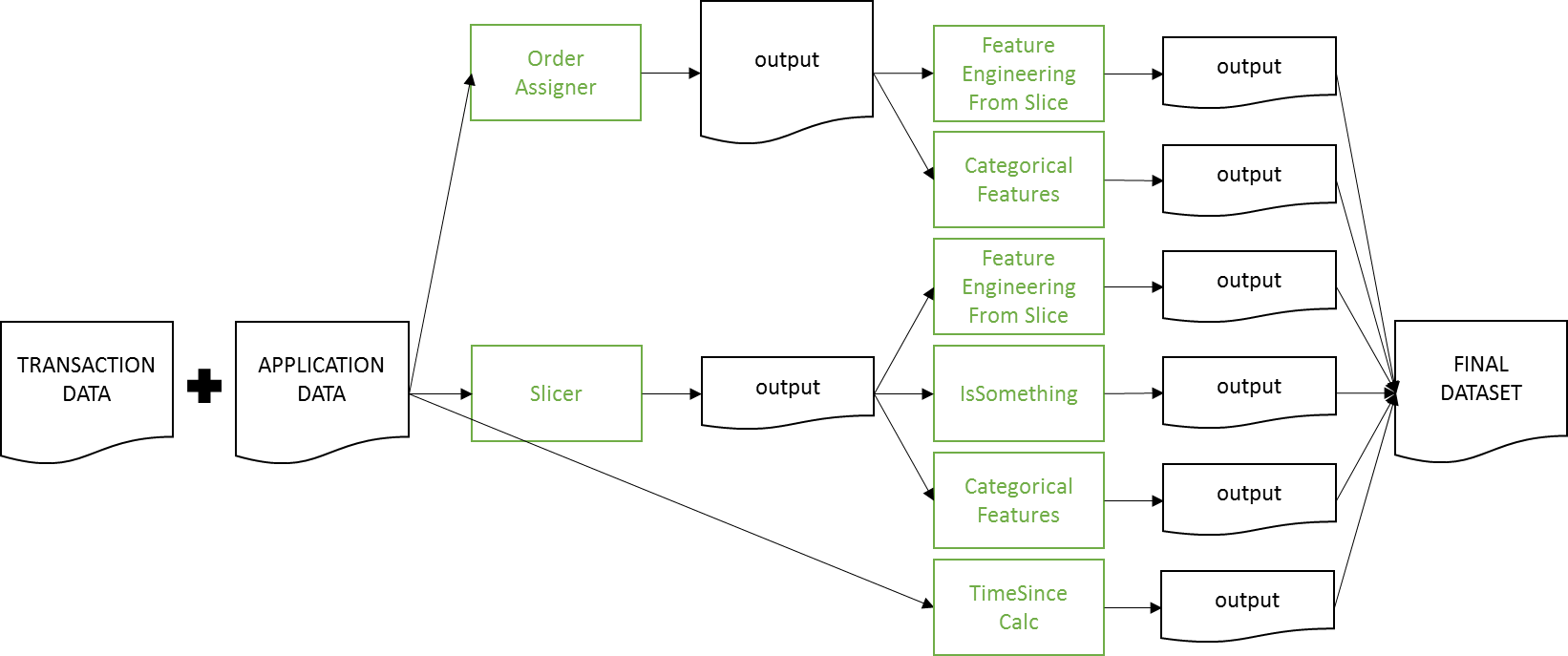
* First level aggregations and their ratios from the TimeSinceCalc output*, e.g. average of transaction amounts of last 3 months, average of transaction amounts of last 3 months / maximum of transaction amounts of last 1 month*
* Second level aggregations and their ratios from Slicer output, *e.g. maximum of monthly averages of transaction amounts of last 3 months, maximum of monthly averages of transaction amounts of last 3 months / maximum of monthly averages of transaction amounts of months with index from 4 to 6*

There is also ability to create categorical features. Imagine we have a categorical variable (*e.g. city where a transacation was performed*) and we want to create some aggregations of it *(e.g. most common city where clients makes transactions, how many time first city within a month where client made transaction changed within last 6 monts, city where client made maximal amount of transactions during last 3 months*). For this purpose, categorical features for slices can be calculated inside Slicer and there is also a separate module **CategoricalFeatures** which, similarly to FeatureEngineeringFromSlice, can take categorical variable either from Slicer or from OrderAssigner and create final categorical features.

Outputs from Slicer can be also processed by class **IsSomething**. This class calculates aggregations telling how many time intervals in row a specific event occurred and when was last time it occurred. Examples of such events are: *monthly max transcation – how many times in a row was it greater than 100 EUR? When was last time it happened at least 3 times in a row? How long is the current series of monthly sum of transaction ascending each month?*

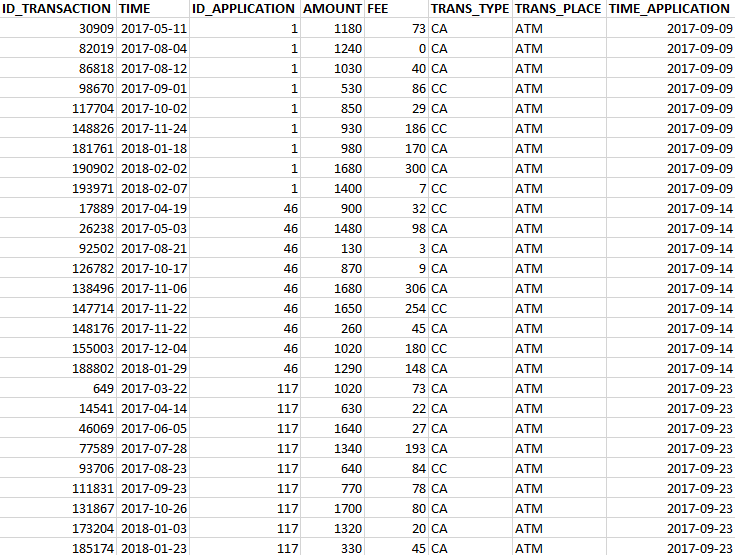
There is one more class which doesn’t generate new time-dependent features, but just combines the columns of the data set. It is called **Interactions** and it can be called in any phase of feature engineering process, hence it is not mentioned in the flow below.

In conclusion, the flow goes like this:

**

And of course it can be repeated for various classes of transactions (e.g. if you are developing a CB scorecard you can do it once for instalment data and once again for query data).

## Example input data



# Classes in feature\_engineering.py

## Slicer

### Description

Slicer creates some basic aggregations in granularity of time unit (you can specify whether it should be day, month etc.). The output from slicer will be table in granularity of application ID x time unit index (where x means Cartesian product). For example with 6 months of transactional history, you will get up to 6 rows per each application ID, and the columns in this table can be maximums, minimums, means, sums and other aggregations of the transactions per each month. The output is meant to be processed further.

New instance of this class is initiated with parameters which point to metadata of the dataset that will be transformed.

Names of the following variables are given as parameters during initiation: application ID, application time, transaction time.

The most important metadata is the **“slice metadata matrix”.** This matrix has 4 columns. The first two columns are mandatory – name of the variable and aggregation type. *Example: for variable AMOUNT and type SUM, the slicer creates variable with sum of amount in each time slice.*

The other two columns of the matrix are optional. Third column is the name of segmentation variable (which must be categorical). If the segmentation variable is filled, the aggregations are calculated “group by” this variable. The last column is 0/1 flag, whether these segmented aggregations should be calculated relatively (as ratio to non-segmented version). *Examples: sum of amount of transactions on credit cards, sum of amount of transactions on debit cards, sum of amount of transactions on credit cards relatively to sum of amount of all transactions.*

After initiation, **fit** method is called. Argument of the fit method is the dataset to be transformed.

After fit, **transform** method is called. Argument of the transform method is the dataset to be transform. Output of transform method is the transformed data. During the run of transform method, sql code which is doing the same transformations in Oracle database is generated as property **strsql\_**.

### Parameters

* id\_name : string - ID column name of the datasets that will be used in fit and transform procedures
* time\_name : string - TIME ID column name of the datasets that will be used in fit and transform procedures
* time\_max\_name : string - TIME MAX column is the column of "current time" for each ID, i.e. time from which the history is calculated back from, Example: time\_max\_name = application date, time\_name = transaction date, only rows with time\_name <= time\_max\_name are taken into account
* slicemeta : matrix - a matrix of slice metadata - defining how the raw data set should be sliced, columns:

1. variable name (variable to be aggregated),
2. aggregation func (e.g. sum),
3. segmentation variable (meaning the aggregation should be segmented by this variable) which can be also empty,
4. relative segmentation flag (int 0/1) (if the segmented aggr. should be calculated as ratio to the non-segm. aggr.)
5. categorical variable flag (int 0/1) flag whether the variable is categorical (for such variables columns 3 and 4 are ignored)
6. 1 if np.nan is considered a category (applies only if column 5 indicates the variable is categorical) or 0 if removed
7. metric for some categorical variable aggregation: for argmax/argmin/argmaxsum/argminsum/argmaxmean/argminmean type of aggregations, this is the metric the max/sum/mean is calculated from

* time\_granularity : string, optional - granularity of the time slices, can be 'months', 'weeks', 'days', 'hours'
* history\_length : int, optional - how long the history should be (examplle values: 12 or 24 for months, 30 for days etc.)
* time\_format : string - the format of time columns
* uppercase\_suffix : boolean, optional - boolean if the suffix of the aggregation type should be uppercase

### Attributes

* X\_in : DataFrame - input
* X\_out : DataFrame - output
* strsql\_ : string - SQL query which makes the same transformation on Oracle database

### Methods

* fit(X, y = None) : go through the aggregation metadata and put all valid aggregations into a special structure
  + X : dataframe - the dataframe you want the aggregations to be executed on
* transform(X) : execute the slice aggregations
  + X : dataframe - the dataframe you want the aggregations to be executed on

### Example of matrix with metadata



### Metadata template



### Example of calling

import numpy as np

import pandas as pd

import scoring

from scoring.feature\_engineering import Slicer

rawdata = pd.read\_csv('C:/Analyses/HQ\_20171213\_PythonWorkflow/fe/rawsim.csv', sep=',',decimal='.',encoding='ANSI')

slicemeta = pd.read\_csv('C:/Analyses/HQ\_20171213\_PythonWorkflow/fe/slicemeta.csv',sep=',',decimal='.',encoding='ANSI')

sl = Slicer(id\_name = 'ID\_APPLICATION',

time\_name = 'TIME',

time\_max\_name = 'TIME\_APPLICATION',

slicemeta = slicemeta,

time\_granularity = 'months',

history\_length = 6,

time\_format = '%Y-%m-%d %H:%M:%S')

sl.fit(rawdata)

newdata = sl.transform(rawdata)

print(sl.strsql\_)

## OrderAssigner

### Description

OrderAssigner which adds new column – difference of application time and transaction in desired unit (monts, days etc.) as integer – so the transactions can be grouped by this time unit later. The output is meant to be processed further.

New instance of this class is initiated with parameters which point to metadata of the dataset that will be transformed.

Names of the following variables are given as parameters during initiation: application time, transaction time.

New instance of this class is initiated with parameters which point to metadata of the dataset that will be transformed.

After initiation, **fit** method is called. Argument of the fit method is the dataset to be transformed.

After fit, **transform** method is called. Argument of the transform method is the dataset to be transform. Output of transform method is the transformed data. During the run of transform method, sql code which is doing the same transformations in Oracle database is generated as property **strsql\_**.

### Parameters

* time\_name : string - TIME ID column name of the datasets that will be used in fit and transform procedures
* time\_max\_name : string - TIME MAX column is the column of "current time" for each ID, i.e. time from which the history is calculated back from, Example: time\_max\_name = application date, time\_name = transaction date, only rows with time\_name <= time\_max\_name are taken into account
* time\_granularity : granularity of the time intervals, can be 'years', 'months', 'weeks', 'days', 'hours', 'minutes', 'seconds' or 'order' (in such case, just the order of the row in time (sorted descending from time\_max\_name) is returned)
* history\_length : int, optional - how long the history should be (examplle values: 12 or 24 for months, 30 for days etc.)
* time\_format : string - the format of time columns
* partition\_name : string - if time\_granularity is 'order', put the name of the column with the ID of the client (or other partitioning entity) here. the order is the calculated in each partition (e.g. for each client) separately
* uppercase\_suffix : boolean, optional - boolean if the suffix of the aggregation type should be uppercase

### Attributes

* X\_out : DataFrame - output
* strsql\_ : string - SQL query which makes the same transformation on Oracle database

### Methods

* fit(X, y = None) : checks if the columns specified in parameters are present in the dataset
  + X : dataframe - the dataframe you want the aggregations to be executed on
* transform(X) : execute the time since aggregations
  + X : dataframe - the dataframe you want the aggregations to be executed on

### Example of calling

import numpy as np

import pandas as pd

import scoring

from scoring.feature\_engineering import OrderAssigner

rawdata = pd.read\_csv('C:/Analyses/HQ\_20171213\_PythonWorkflow/fe/rawsim.csv', sep=',',decimal='.',encoding='ANSI')

oa = OrderAssigner(time\_name = 'TIME',

time\_max\_name = 'TIME\_APPLICATION',

time\_granularity = 'months',

history\_length = 6,

time\_format = '%Y-%m-%d %H:%M:%S',

partition\_name = None)

oa.fit(rawdata)

oo = oa.transform(rawdata)

print(oa.strsql\_)

## TimeSinceCalc

### Description

TimeSinceCalccalculates time since last/first transaction which holds given properties. It basically calculates difference of application time and transaction in desired unit (monts, days etc.) as float, and then takes maximum or minimum of the time differences of specified subset of transactions.

New instance of this class is initiated with parameters which point to metadata of the dataset that will be transformed.

Names of the following variables are given as parameters during initiation: application ID, application time, transaction time.

The most important metadata is **“time since metadata matrix”.** This matrix has 14 columns. The first 2 columns are mandatory: in first one you specify time granularity (in which time units should be the specific variable calculated in) and in the second one you specify whether you want to know the time since last or first transaction. If the other columns are not filled, the aggregation will be basically time since last or first transaction in the table.

In the other columns user can specify conditions which will be applied – see below how to fill them. If the conditions are specified, the aggregation will be: time since last or first transaction in the table which is fulfilling the conditions.

After initiation, **fit** method is called. Argument of the fit method is the dataset to be transformed.

After fit, **transform** method is called. Argument of the transform method is the dataset to be transform. Output of transform method is the transformed data. During the run of transform method, sql code which is doing the same transformations in Oracle database is generated as property **strsql\_**.

### Parameters

* id\_name : string - ID column name of the datasets that will be used in fit and transform procedures
* time\_name : string - TIME ID column name of the datasets that will be used in fit and transform procedures
* time\_max\_name : string - TIME MAX column is the column of "current time" for each ID, i.e. time from which the history is calculated back from, Example: time\_max\_name = application date, time\_name = transaction date, only rows with time\_name <= time\_max\_name are taken into account
* timesincemeta : matrix - a matrix of slice metadata - defining how the raw data set should be sliced, columns:

1. granularity: time units in which the time since should be calculated in, can be 'years', 'months', 'weeks', 'days', 'hours', 'minutes', 'seconds'
2. type: "last" or "first" - whether to calculated time since first or last transaction which happened in the history and is fulfilling the given condition

The other columns can specify up to two conditions which transaction must fulfill to enter the aggregation:

1. condition1: name of the column which the condition is based on. if empty, the condition is considered to be always true.
2. from1: if condition1 column is numeric, specify the left boundary of interval of values where the condition is considered fulfilled. If empty, the left boundary is considered to be -infinity.
3. from1eq: 0 or 1, specifying whether the inequality is sharp (1 for sharp)
4. to1: if condition1 column is numeric, specify the right boundary of interval of values where the condition is considered fulfilled. If empty, the left boundary is considered to be +infinity.
5. to1eq: 0 or 1, specifying whether the inequality is sharp (1 for sharp)
6. category1: if condition1 column is categorical, specify the category which the column should be equal to for the condition to be true. If empty, the algorithm scans the dataset for all the possible categories and uses each one of them as a separate condition. NaN is not considered a category.
7. - 14) the same for second condition. If empty, the condition is considered to be always true.

* time\_format : string - the format of time columns
* uppercase\_suffix : boolean, optional - boolean if the suffix of the aggregation type should be uppercase

### Attributes

* X\_in : DataFrame - input
* X\_out : DataFrame - output
* strsql\_ : string - SQL query which makes the same transformation on Oracle database

### Methods

* fit(X, y = None) : go through the aggregation metadata and put all valid aggregations into a special structure.
  + X : dataframe, the dataframe you want the aggregations to be executed on
* transform(X) : execute the time since aggregations
  + X : dataframe, the dataframe you want the aggregations to be executed on

### Example of matrix with metadata



### Metadata template



### Example of calling

import numpy as np

import pandas as pd

import scoring

from scoring.feature\_engineering import TimeSinceCalc

dsload = pd.read\_csv('data4timeSince.csv',index\_col=0)

timesincemeta = pd.read\_csv('C:/Analyses/HQ\_20171213\_PythonWorkflow/fe/timesincemeta.csv',sep=',',decimal='.',encoding='ANSI')

tsc = TimeSinceCalc(id\_name = 'ID\_APPLICATION',

time\_name = 'TIME',

time\_max\_name = 'TIME\_APPLICATION',

timesincemeta = timesincemeta,

time\_format='%Y-%m-%d %H:%M:%S',

keyword='entry')

tsc.fit(dsload)

ds = tsc.transform(dsload)

print(tsc.strsql\_)

## FeatureEngineeringFromSlice

### Description

New instance of this class is initiated with parameters which point to metadata of the dataset that will be transformed.

Names of the following variables are given as parameters during initiation: application ID, time key (index of time interval, which can be generated using Slicer or OrderAssigner).

There are three metadata matrices which specify which aggregations should be created:

**“Meta matrix”**: Includes names of the variables from input data set which should be aggregated. In second column of this matrix, you should also define a “native” aggregating function for each of the variable (this function will be used in the Aggregation list matrix which is described below). All the variables specified here in Meta matrix are then aggregated into aggregations which are specified in Aggregation list matrix (see below).

**“Aggregation list matrix”**: Includes instructions what variables should be created from each of the variable mentioned in meta matrix. It has 12 columns. In first column you specify how long history is needed for the aggregation to be calculated. In 2nd and 3rd column, you specify type of aggregation. The aggregation can be “basic”/“parametric”/”varcomb”/”segmented” (specified in 2nd column) and “simple”/”ratio” (specified in 3rd column).

Basic aggregations are exactly specified in columns 4-12. You need to specify what time intervals are aggregated using which function. You can also specify whether there will be some kind of special condition applied before the aggregation to filter the rows. For this kind of “special condition” aggregations, you should specify also suffix which will differentiate the resulting variable from the common one.

Parametric aggregations are similar to basic aggregations but you can use the word “parametric” to specify aggregation function. In these aggregation, the function specified in meta matrix as “native” will be used for the corresponding variable

Varcomb aggregations are ratios of two different variables. Variables for these aggregations are specified in variable combination matrix which is described below

Segmented aggregations work the same way as basic aggregations, but they are performed on clusters of rows defined by segmentation variable (any categorical variable).

Simple aggregations are aggregations like “(agg) of (variable) within (time units) with indexes (from)-(to)”. Ratio aggregations on the other hand are adding a denominator to the formula, so they are computed like “(agg1) of (variable) within (time units) with indexes (from1)-(to1) divided by (agg2) of (variable) within (time units) with indexes (from2)-(to2)”, *e.g. maximal DPD in last 3 months* (simple) or *maximal DPD in last 3 months / maximal DPD in months 4-6*.

**“Variable combination matrix”**: Includes pairs of variables which are the used for ratios to be calculated. *E.g. row including a pair: SUM\_AMOUNT, COUNT\_TRANSACTION can be used for creating varibles such as sum of monthly sum of amounts in last 3 months / sum of monthly count of transactions in last 3 months*. The specific aggregations are specified in abovementioned Aggregation list matrix, as aggregations with type “varcomb” (the aggregations are the same for each pair of variables specified in variable combination matrix).

**“Segmentation variable matrix”**: Includes variables which should be used for segmentations in “segmented” type of aggregations. These aggregations can be for example *SUM of transaction amount of all transactions of type POS/ATM during last 6 months*, where *POS/ATM* are the distinct values of the segmentation variable. *For this example, two features would be created (one for POS, second for ATM).*

After initiation, **fit** method is called. Argument of the fit method is the dataset to be transformed.

After fit, **transform** method is called. Argument of the transform method is the dataset to be transform. Output of transform method is the transformed data. During the run of transform method, sql code which is doing the same transformations in Oracle database is generated as property **strsql\_**.

### Parameters

* id\_name : string - ID column name of the datasets that will be used in fit and transform procedures
* time\_name : string - TIME ID column name of the datasets that will be used in fit and transform procedures
* metadata : matrix - a matrix of metadata - telling which aggregation families should be performed for which column, columns:

1. variable name (variable to be used for FE),
2. aggregation func (e.g. sum)

* agglist : matrix - a matrix of aggregation types - defining the aggregation families, columns:

1. minimal max month,
2. type of aggregation (basic/parametric/varcomb/segmented),
3. 2nd type of aggregation (simple/ratio),
4. from (...of basic aggregation or numerator of ratio),
5. to (...of basic aggregation or numerator of ratio),
6. func (...of basic aggregation or numerator of ratio),
7. from (...of denominator of ratio),
8. to (...of denominator of ratio),
9. func (...of denominator of ratio),
10. query (...additional condition for basic aggregation or numerator of ratio)
11. query (...additional condition for denominator of ratio)
12. suffix (...suffix for the columns using the queries)

* varcomb : matrix, optional - a matrix of varible combinations, columns:

1. variable for numerator.
2. variable for denominator

* segm: matrix, optional – a matrix of segmentation variables, for which aggregations of type “segmented” should be created, columns:

1. segmentation variable

* max\_time : int, optional - number of time units that the aggregations are based on
* uppercase\_suffix : boolean, optional - if the suffix of the aggregation type should be uppercase
* min\_fill\_share : decimal, optional - share of filled (non-NaN) rows of feature for this feature to be added to the new dataset (if set to 0, all columns will be added. If set to 1, only fully filled columns will be added)

### Attributes

* X\_in : DataFrame - input
* X\_out : DataFrame - output
* strsql\_ : string - SQL query which makes the same transformation on Oracle database

### Methods

* fit(X, y = None) : go through the aggregation metadata and put all valid aggregations into a special structure
  + X : dataframe - the dataframe you want the aggregations to be executed on
* transform(X) : execute the aggregations
  + X : dataframe - the dataframe you want the aggregations to be executed on

### Example of matrix with metadata



### Example of matrix with aggregation list



### Example of variable combination matrix



### Metadata template



### Example of calling

import numpy as np

import pandas as pd

import scoring

from scoring.feature\_engineering import FeatureEngineeringFromSlice

metadata = pd.read\_csv('C:/Analyses/HQ\_20171213\_PythonWorkflow/fe/aggregations\_metadata3.csv',sep=',',decimal='.',encoding='ANSI')

agglist = pd.read\_csv('C:/Analyses/HQ\_20171213\_PythonWorkflow/fe/aggregations\_agglist3.csv', sep=',',decimal='.',encoding='ANSI')

varcomb = pd.read\_csv('C:/Analyses/HQ\_20171213\_PythonWorkflow/fe/varcomb.csv',sep=',',decimal='.',encoding='ANSI')

fe2 = FeatureEngineeringFromSlice(id\_name = 'ID\_APPLICATION',

time\_name = 'TIME\_ORDER',

metadata = metadata,

agglist = agglist,

varcomb = varcomb,

segm = None,

max\_time = 6,

min\_fill\_share = 0)

fe2.fit(newdata)

transdata = fe2.transform(newdata)

print(fe2.strsql\_)

## IsSomething

### Description

New instance of this class is initiated with parameters which point to metadata of the dataset that will be transformed.

The transformation itself work on this principle:

* First, a binary mask is applied to a given column based on criteria from metadata. We take a specified period (from-to) and for each time unit in this period we check whether given column fulfills given criteria (is ascending/descending in comparison with previous time unit; or is lower/greater/equal to a given value)
* Based on this binary mask we can calculate aggregations like: how many time such event happened, how many time it happened in a row, how long ago such row of given length occurred etc.
* These aggregations are in granularity of application and so they are the final outputs (final features)

This way we end up with the following conceptual predictors (Is Something means that the value in vector = TRUE) that can based on the configuration and input vector/filter generate the same predictors:

* Sum Is Something {x, y} (Number of months with balance > $0 in last 12 months)
* Max Consecutive Is Something {x, y} (Maximum number of consecutive months with balance > $0 in last 12 months)
* Max Consecutive Is Not Something {x, y} (Maximum number of consecutive months with balance = $0 in last 12 months) – Note that the binary condition might be complicated, so it is easier to generate new feature by negating the condition rather than adding new configuration for the opposite
* Consecutive Is Something from Current {x, y}   (Number of consecutive months with balance $>0 in last 12 months, starting with current month)
* Consecutive Is Not Something from Current {x, y} (Number of consecutive months with balance $=0 in last 12 months, starting with current month)
* Avg Length of Consecutive Is Something {x, y} (Average length of consecutive months with balance >$0 in last 12 months)
* Avg Length of Consecutive Is Not Something {x, y} (Average length of consecutive months with balance = $0 in last 12 months)
* Number of Consecutive Is or Is Not Something instances {x, y} (Number of consecutive instances with Balance >$0 or Balance = $0 in last 12 months)
* Number of Instances of Consecutive Something {x, y} is greater than z (Number of cases where Balance >$0 for more than or equal 3 consecutive months in last 12 months)
* Number of Instances of Consecutive Not Something {x, y} is greater than z (Number of cases where Balance =$0 for more than or equal 3 consecutive months in last 12 months)
* Distance from last Consecutive Is Something {x, y} greater than z (Distance in months from the last occurrence of Balance >$0 for more than 3 consecutive months in last 12 months)
* Distance from last Consecutive Is Not Something {x, y} greater than z (...dtto)
* Ratio Sum Is Something {x, s} and Sum Is Something {s, y} (Percentage of months where Balance >$0 in last 12 months)
* Ratio of Is Something {x,y} and Number of Records

There is a metadata matrix **“isSomeMatrix”** which specify which aggregations should be created: you specify column name, condition for the binary transformation and time period for these aggregations to be calculated from. See below.

After initiation, **fit** method is called. Argument of the fit method is the dataset to be transformed.

After fit, **transform** method is called. Argument of the transform method is the dataset to be transform. Output of transform method is the transformed data.

### Parameters

* id\_name : string - ID column name of the datasets that will be used in fit and transform procedures
* time\_name : string – order (slice) name of the datasets that will be used in fit and transform procedures
* issomemeta : matrix - matrix of aggregation types - defining the aggregation families, columns:
* column - name of column the condition is based on
* min order - minimal time order index which the condition is calculated for
* max order - maximal time order index which the condition is calculated for
* mid order - if filled, ratios of the occurencies condition fulfilled in intervals [min order,mid order] and (mid order,max order] are calculated
* threshold - if filled, counts of how many time the condition was fulfilled at least threshold-times in a row are calculated
* condition type - can be:
* asc (column values were ascending in time)
* desc (column values were descending in time)
* notasc (column values were not ascending in time)
* notdesc (column values were not descending in time)(column values were greater than condition value)
* >= (column values were greater than or equal to condition value)
* < (column values were lower than condition value)
* <= (column values were lower than or equal to condition value)
* = (column values were equal to condition value)
* condition value - value for conditin types >,>=,<,<=,=
* max\_time : int, optional - number of time units that the aggregations are based on. if max\_order in issomemeta is greater than this, this overrides it. infinity by default
* uppercase\_suffix : boolean, optional - boolean if the suffix of the aggregation type should be uppercase

### Attributes

* X\_in : DataFrame - input
* X\_out : DataFrame - output
* strsql\_ : string - SQL query which makes the same transformation on Oracle database

### Methods

* fit(X, y = None) : go through the aggregation metadata and put all valid aggregations into a special structure
  + X : dataframe - the dataframe you want the aggregations to be executed on
* transform(X) : execute the aggregations
  + X : dataframe - the dataframe you want the aggregations to be executed on

### Example of matrix with issomemeta



### Metadata template



### Example of calling

import numpy as np

import pandas as pd

import scoring

from scoring.feature\_engineering import IsSomething

issomemeta = pd.read\_csv('C:/Analyses/HQ\_20171213\_PythonWorkflow/fe/issomemeta.csv', sep=',',decimal='.',encoding='ANSI')

iss = IsSomething(id\_name = 'ID\_APPLICATION',

time\_name = 'TIME\_ORDER',

issomemeta = issomemeta,

max\_time = np.inf)

iss.fit(newdata)

issomedata = iss.transform(newdata)

## CategoricalFeatures

### Description

Creates aggregation which are specified in a matrix with metadata. The input data must be slice-indexed. Either in transactional granularity (from OrderAssigner) or in slice granularity (from Slicer).

### Parameters

* id\_name : string - ID column name of the datasets that will be used in fit and transform procedures
* from type : string - can be either 'slice' or 'raw' determining whether the features are created from pre-aggregated slices (i.e. after Slicer) or from raw-granularity data (i.e. after OrderAssigner)
* catmeta : matrix - matrix of slice metadata - defining how the raw data set should be sliced, columns:

1. variable: name of categorical variable the features will be based on
2. aggregation type: mode, nunique, last, first, argmax, argmaxsum, argmaxmean, argmin, argminsum, argminmean, nchanges, tschange
3. from: minimal time index that will be taken into account for the aggragation calculation
4. to: maxinal time index that will be taken into account for the aggragation calculation
5. nancategory: whether NaN is a separate category or whether such rows should not be used
6. metric: for argmax/argmin/argmaxsum/argminsum/argmaxmean/argminmean type of aggregations, this is the metric the max/sum/mean is calculated from
7. granularity: for tschange aggregation (when calculated from raw data), this is the time unit the time since is calculated in

* slice\_name: string - time index (from Slicer or Order Assigner)
* time\_name: string, optional - from\_type = 'raw', this should be time of the transaction
* time\_max\_name: string, optional - for from\_type = 'raw', time dimension of the ID (e.g. in the final aggregations granularity)
* time\_format : string - the format of time columns
* uppercase\_suffix : boolean, optional - boolean if the suffix of the aggregation type should be uppercase

### Attributes

* X\_in : DataFrame - input
* X\_out : DataFrame - output
* strsql\_ : string - SQL query which makes the same transformation on Oracle database

### Methods

* fit(X, y = None) : go through the categorical aggregation metadata and put all valid aggregations into a special structure
  + X : dataframe - the dataframe you want the aggregations to be executed on
* transform(X) : execute the aggregations
  + X : dataframe - the dataframe you want the aggregations to be executed on

### Example of matrix with catmeta



### Metadata template



### Example of calling

import numpy as np

import pandas as pd

import scoring

from scoring.feature\_engineering import CategoricalFeatures

catmeta = pd.read\_csv('catmeta.csv',sep=',',decimal='.',encoding='ANSI')

ct = CategoricalFeatures(id\_name = 'ID\_APPLICATION',

from\_type = 'raw',

catmeta = catmeta,

slice\_name = 'TIME\_ORDER',

time\_name = 'TIME',

time\_max\_name = 'TIME\_APPLICATION',

time\_format = '%Y-%m-%d %H:%M:%S')

ct.fit(orderdata)

catdata = ct.transform(orderdata)

## Interactions

### Description

Creates combination from the columns of a dataset. The available types of combinations are:

* *sum*: sum of two variables, i.e. var1 + var2, makes sense for numbers only
* *product*: product of two variables, i.e. var1 \* var2, makes sense for numbers only
* *difference*: difference of two variables, i.e. var1 - var2, makes sense for numbers only
* *ratio*: ratio of two variables, i.e. var1/var2, makes sense for numbers only
* *cartesian*: Cartesian product of two variables. If any of these variables is numerical, it is automatically binned to quantiles, the number of quantiles can be specified in parametrization table
* *equality*: comparison of two variables, the result is one of following: “<”, “=”, ly “>”, can be applied either to strings or to numbers
* *quantiles*: bins a variable into quantiles, the number of quantiles can be specified in parametrization tale – in this case, variable2 is not used
* *length*: length of string variable – in this case, variable2 is not used
* *clean*: applies “cleaning”, i.e. filling of NaNs and infinites, to one variable – in this case, variable2 is not used

### Parameters

* id\_name : string - ID column name of the datasets that will be used in fit and transform procedures
* intermeta : matrix - matrix of slice metadata - defining how the raw data set should be sliced, columns:

1. variable1: name of variable that should be combined with variable2
2. variable2: name of variable that should be combined with variable1
3. type: type of combination (the available types are mentioned in the description above)
4. in\_nan: how NaNs should be treated BEFORE the transformation (can be left empty)
5. in\_inf: how Infinities should be treated BEFORE the transformation (can be left empty)
6. out\_nan: how NaNs should be treated AFTER the transformation (can be left empty)
7. out\_inf: how Infinities should be treated AFTER the transformation (can be left empty)
8. bins: for “quantiles” or “cartesian” type of transformation, how many quantiles should be created

* uppercase\_suffix : boolean, optional - boolean if the suffix of the aggregation type should be uppercase

### Attributes

* X\_in : DataFrame - input
* X\_out : DataFrame - output
* strsql\_ : string - SQL query which makes the same transformation on Oracle database

### Methods

* fit(X, y = None) : go through the interaction metadata and put all valid aggregations into a special structure
  + X : dataframe - the dataframe you want the aggregations to be executed on
* transform(X) : execute the interactions
  + X : dataframe - the dataframe you want the aggregations to be executed on

### Example of matrix with intermeta



### Metadata template



### Example of calling

import numpy as np

import pandas as pd

import scoring

from scoring.feature\_engineering import Interactions

intermeta = pd.read\_csv('intermeta.csv',sep=',',decimal='.',encoding='ANSI')

itr = Interactions(id\_name = 'ID\_APPLICATION',

intermeta = intermeta)

itr.fit(rawdata)

itrdata = itr.transform(rawdata)