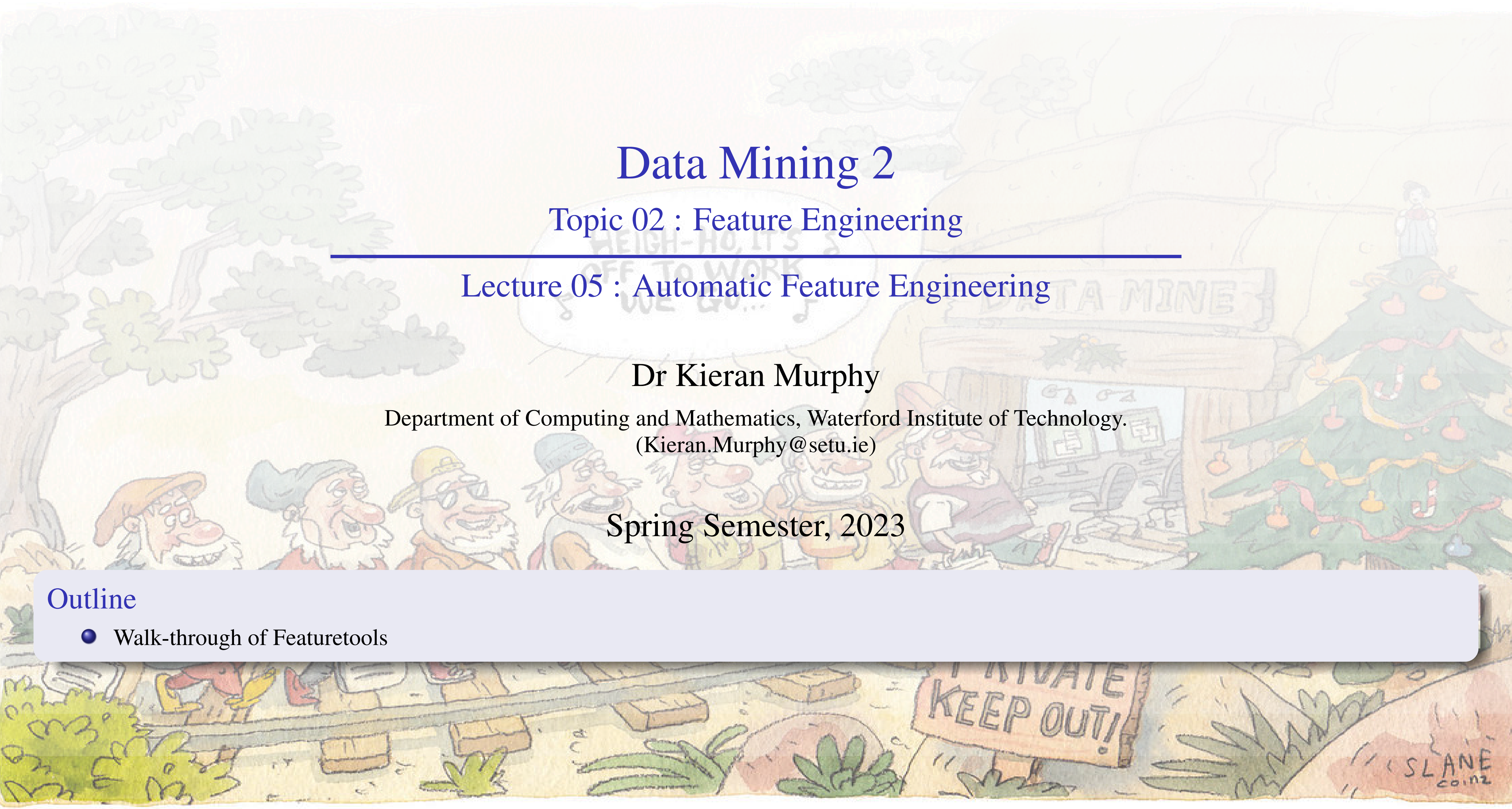



# Data Mining 2

## Topic 02 : Feature Engineering

---

## Lecture 05 : Automatic Feature Engineering

Dr Kieran Murphy

Department of Computing and Mathematics, Waterford Institute of Technology.
(Kieran.Murphy@setu.ie)

Spring Semester, 2023

### Outline

- Walk-through of Featuretools

# Motivation

- Typically, feature engineering is a drawn-out manual process, relying on domain knowledge, intuition, and data manipulation.
- This process can be extremely tedious and the final features will be limited both by human subjectivity and time.
- While each individual new feature may be easy to create/develop the process is not scalable and often not applicable across datasets.

⇒ Automated feature engineering[*] aims to help the data scientist by automatically creating many candidate features out of a dataset from which the best can be selected and used for training.,

Featuretools

featuretools is an attempt to automate the feature generation using their Deep Feature Synthesis (DFS) process which performs feature engineering on relational and temporal data.

[*]Note that this is the trend in all stages of the data mining pipeline — e.g. model selection, hyper-parameter training — more and more of the process is being automated.

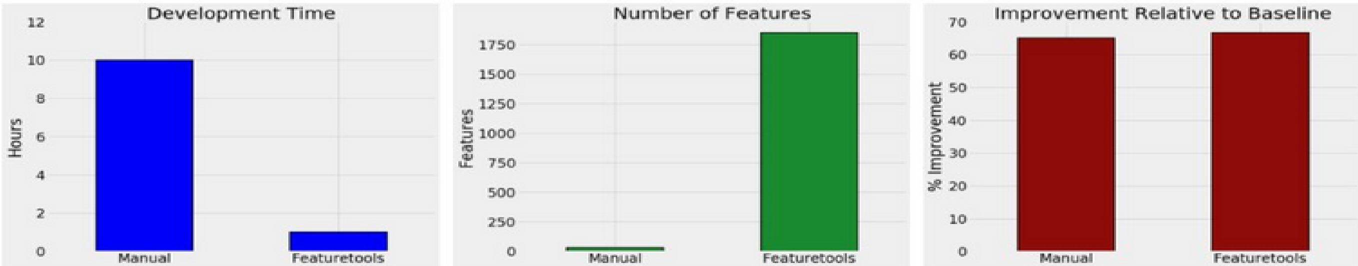
# Why (Learn Yet Another Library)?

## **KDnuggets:** Why Automated Feature Engineering Will Change the Way You Do Machine Learning

After a little feature selection and model optimization, these features did slightly better in a predictive model compared to the manual features with an overall development time of **1 hour**, a 10x reduction compared to the manual process. Featuretools is much faster both because it requires less domain knowledge and because there are considerably fewer lines of code to write.

I'll admit that there is a slight time cost to learning Featuretools but it's an investment that will pay off. After taking an hour or so to learn Featuretools, you can apply it *to any machine learning* problem.

The following graphs sum up my experience for the loan repayment problem:




Comparison between automated and manual feature engineering on time, number of features, and performance.

- Development time: accounts for everything required to make the final feature engineering code: **10 hours manual vs 1 hour automated**
- Number of features produced by the method: **30 features manual vs 1820 automated**
- Improvement relative to baseline is the % gain over the baseline compared to the top public leaderboard score using a model trained on the features: **65% manual vs 66% automated**

> My takeaway is that automated feature engineering will not replace the data scientist, but rather by significantly increasing efficiency, it will free her to spend more time on other aspects of the machine learning pipeline.

Furthermore, the Featuretools code I wrote for this first project could be applied to any dataset while the manual engineering code would have to be thrown away and entirely rewritten for the next dataset!

# Example Applications of Featuretools

There exists online a number of guides covering the application of featuretools. In particular:

- A Hands-On Guide to Automated Feature Engineering using Featuretools in Python
  Which uses data from Analytics Vidhya BigMart Sales challenge is to build a predictive model to estimate the sales of each product at a particular store.
  - This is a single table dataset, but featuretools created 29 new features resulting in a model RMSE score of 1155.12, compared to initial model of 1183 (lower is better).
- Predicting a customer's next purchase using automated feature engineering
  Uses Instacart's multi-table dataset of 3 million online grocery orders to predict what product a customer buys next.
  - Generate 150+ features using Deep Feature Synthesis and select the 20 most important features for predictive modeling.
  - Develop a model on a subset of the data and validate on the entire dataset in a scalable manner using Dask.
- Automated Feature Engineering in Python
  Small multi-table, loan repayment dataset — not real data but demonstrates the potential explosion of features when merging tables.

source for this lecture

# Loan Repayment Dataset

| client_id | joined | income | credit_score |
|---|---|---|---|
| 46109 | 2002-04-16 | 172677 | 527 |
| 49545 | 2007-11-14 | 104564 | 770 |
| 41480 | 2013-03-11 | 122607 | 585 |
| 46180 | 2001-11-06 | 43851 | 562 |
| 25707 | 2006-10-06 | 211422 | 621 |

**Clients**
Basic information about clients at a credit union. Each client appears in exactly one row.

1-*

| client_id | loan_type | loan_amount | repaid | loan_id | loan_start | loan_end | rate |
|---|---|---|---|---|---|---|---|
| 25707 | other | 9942 | 1 | 10438 | 2009-03-26 | 2010-10-22 | 2.39 |
| 39384 | other | 13131 | 1 | 10579 | 2012-08-12 | 2014-06-17 | 2.95 |
| 49624 | other | 2572 | 1 | 10578 | 2004-05-04 | 2005-12-16 | 2.28 |
| 29841 | credit | 10537 | 1 | 10157 | 2010-08-04 | 2013-03-11 | 3.43 |
| 39505 | other | 6484 | 1 | 10407 | 2011-02-14 | 2012-12-07 | 1.14 |
| 44601 | home | 4475 | 1 | 10362 | 2005-07-29 | 2007-07-06 | 6.58 |
| 39384 | credit | 1770 | 1 | 10868 | 2013-08-03 | 2016-04-28 | 2.64 |
| 48177 | other | 1383 | 0 | 11264 | 2009-08-08 | 2012-01-03 | 5.69 |
| 32885 | home | 11783 | 0 | 10301 | 2000-08-10 | 2003-03-12 | 2.64 |
| 49068 | cash | 6473 | 1 | 11546 | 2002-09-01 | 2004-10-23 | 5.18 |

**Loans**
loans made to the clients. Each loan has exactly one row, but clients may have multiple loans.

*-1

| loan_id | payment_amount | payment_date | missed |
|---|---|---|---|
| 10302 | 489 | 2006-06-17 | 1 |
| 11652 | 1896 | 2014-08-17 | 0 |
| 11827 | 2755 | 2005-02-26 | 1 |
| 10078 | 624 | 2005-05-28 | 1 |
| 10177 | 1474 | 2002-05-03 | 0 |
| 10660 | 701 | 2005-09-22 | 1 |
| 11251 | 568 | 2000-08-27 | 0 |
| 10826 | 2538 | 2005-03-20 | 0 |
| 11896 | 1055 | 2004-06-02 | 0 |
| 10742 | 437 | 2005-06-04 | 1 |

**Payments**
Payments made on the loans. Each payment has exactly one row, but each loan will have multiple payments.

# Transformations versus Aggregations

New features can be organised into two categories — transformations and aggregations.

## Transformation

Transformations act on a single table by creating new features out of one or more of the existing columns.

In the clients table, features showing the month of the joined column or the natural log of the income column are transformations.

| client_id | joined | income | credit_score | join_month | log_income |
|---|---|---|---|---|---|
| 46109 | 2002-04-16 | 172677 | 527 | 4 | 12.059178 |
| 49545 | 2007-11-14 | 104564 | 770 | 11 | 11.557555 |
| 41480 | 2013-03-11 | 122607 | 585 | 3 | 11.716739 |
| 46180 | 2001-11-06 | 43851 | 562 | 11 | 10.688553 |
| 25707 | 2006-10-06 | 211422 | 621 | 10 | 12.261611 |

## Aggregations

Aggregations are performed across tables, and use a one-to-many relationship to group observations and then calculate statistics.

Combining the clients and loans tables, where each client may have multiple loans, we can calculate statistics such as the average, maximum, and minimum of loans for each client.

| client_id | joined | income | credit_score | join_month | log_income | mean_loan_amount | max_loan_amount | min_loan_amount |
|---|---|---|---|---|---|---|---|---|
| 46109 | 2002-04-16 | 172677 | 527 | 4 | 12.059178 | 8951.600000 | 14049 | 559 |
| 49545 | 2007-11-14 | 104564 | 770 | 11 | 11.557555 | 10289.300000 | 14971 | 3851 |
| 41480 | 2013-03-11 | 122607 | 585 | 3 | 11.716739 | 7894.850000 | 14399 | 811 |
| 46180 | 2001-11-06 | 43851 | 562 | 11 | 10.688553 | 7700.850000 | 14081 | 1607 |
| 25707 | 2006-10-06 | 211422 | 621 | 10 | 12.261611 | 7963.950000 | 13913 | 1212 |
| 39505 | 2011-10-14 | 153873 | 610 | 10 | 11.943883 | 7424.050000 | 14575 | 904 |
| 32726 | 2006-05-01 | 235705 | 730 | 5 | 12.370336 | 6633.263158 | 14802 | 851 |
| 35089 | 2010-03-01 | 131176 | 771 | 3 | 11.784295 | 6939.200000 | 13194 | 773 |
| 35214 | 2003-08-08 | 95849 | 696 | 8 | 11.470529 | 7173.555556 | 14767 | 667 |
| 48177 | 2008-06-09 | 190632 | 769 | 6 | 12.158100 | 7424.368421 | 14740 | 659 |

# Featuretools Concepts/Termonology

Featuretools Feature engineering means building additional features out of existing data which is often spread across multiple related tables. Feature engineering requires extracting the relevant information from the data and getting it into a single table which can then be used to train a machine learning model.

## Entity / Entityset

- An Entity can be considered as a representation of a Pandas DataFrame (or a single table in a relational database). A collection of multiple entities is called an Entityset.

## Deep Feature Synthesis (DFS)

- DFS is the automatic feature engineering method at the core of Featuretools. It enables the creation of new features from single, as well as multiple dataframes.
- DFS create features by applying feature primitives to the entity relationships in an entityset. These primitives are the often-used methods to generate features manually. For example, the primitive 'mean' would find the mean of a variable at an aggregated level.

# Walk-Through — Step 1: Create an empty EntitySet

## Step 0: Load data into Pandas DataFrames

1
```
import pandas as pd
import numpy as np
import featuretools as ft

clients = pd.read_csv("data/clients.csv", parse_dates = ["joined"])
loans = pd.read_csv("data/loans.csv", parse_dates = ["loan_start", "loan_end"])
payments = pd.read_csv("data/payments.csv", parse_dates = ["payment_date"])
```

## Step 1 — Create an empty EntitySet

An EntitySet is a collection of tables and the relationships between them.

2
```
es = ft.EntitySet(id="clients")
```

# Walk-Through: Step 2 — Create each Entity I

## Step 2 — Create each Entity

- Each entity must have an index, which is a column with all unique elements. (Primary key in traditional database terms.)
- The index in the clients dataframe is the `client_id` because each client has only one row in this dataframe.

We add an entity with an existing index to an entityset using the following syntax:

3

```
# Create an entity from the client dataframe
# This dataframe already has an index and a time index
es = es.add_dataframe(dataframe_name="clients",
    dataframe=clients,
    index="client_id",
    time_index="joined")
print(es)
```

```
Entityset: clients
  DataFrames:
    clients [Rows: 25, Columns: 4]
  Relationships:
    No relationships
```

# Walk-Through: Step 2 — Create each Entity II

- The `loans` dataframe also has a unique index,`loan_id`.
- Although featuretools will automatically infer the data type of each column in an entity, we can override this by passing in a dictionary of column types to the parameter `variable_types`.

4

```
# Create an entity from the loans dataframe
# This dataframe already has an index and a time index
es = es.add_dataframe(dataframe_name="loans",
    dataframe = loans,
    logical_types = {"repaid": "Categorical"},
    index = "loan_id",
    time_index = "loan_start")
print(es)
```

```
Entityset: clients
  DataFrames:
    clients [Rows: 25, Columns: 4]
    loans [Rows: 443, Columns: 8]
  Relationships:
    No relationships
```

# Walk-Through: Step 2 — Create each Entity III

- For the payments dataframe, there is no unique index. Hence we need to pass in the parameter `make_index=True` and specify the name of the index.
- For this dataframe, even though `missed` is an integer, this is not a numeric variable since it can only take on 2 discrete values, so we tell featuretools to treat is as a categorical variable.

5

```
# Create an entity from the payments dataframe
# This does not yet have a unique index
es = es.add_dataframe(dataframe_name="payments",
    dataframe = payments,
    logical_types = {"missed": "Categorical"},
    time_index = "payment_date",
    make_index = True,
    index = "payment_id")
print(es)
```

```
Entityset: clients
  DataFrames:
    clients [Rows: 25, Columns: 4]
    loans [Rows: 443, Columns: 8]
    payments [Rows: 3456, Columns: 5]
  Relationships:
    No relationships
```

# Walk-Through: Step 2 — Create each Entity IV

We can access any of the entities in the entityset using Python dictionary syntax.

6
```
#print(es["clients"])
#print(es["loans"])
print(es["payments"])
```

```
      payment_id loan_id
payment_amount payment_date missed
2113        2113    11988              2053
2000-03-05     0
726          726    11140               402
2000-03-19     0
2114        2114    11988              2627
2000-03-30     0
3223        3223    11430              1284
2000-04-05     0
2115        2115    11988              1911
2000-04-11     1
...          ...      ...
...          ...    ...
1415        1415    11072               957
2015-07-01     0
1308        1308    10684               115
2015-07-06     0
1416        1416    11072               988
2015-07-14     1
1417        1417    11072               940
2015-07-29     0
1418        1418    11072               932
2015-08-21     1

[3456 rows x 5 columns]
```

# Walk-Through: Step 3 — Create Table Relationships I

## Step 3 — Create Table Relationships

New we need to define the relationship between tables and its multiplicity, i.e., one-to-many.† In our dataset:

- `clients` dataframe is a parent of the `loans` dataframe.
  Each client has only one row in clients but may have multiple rows in loans.
- `loans` is the parent of `payments` since each loan will have multiple payments.
  The parents are linked to their children by a shared variable. When we perform aggregations, we group the child table by the parent variable and calculate statistics across the children of each parent.

To formalise a relationship in featuretools, we only need to specify the variable that links two tables together. The clients and the loans table are linked via the `client_id` variable and loans and payments are linked with the `loan_id`.

†The best way to think of a relationship between two tables is the analogy of parent to child. This is a one-to-many relationship: each parent can have multiple children. In the realm of tables, a parent table has one row for every parent, but the child table may have multiple rows corresponding to multiple children of the same parent.

# Walk-Through: Step 3 — Create Table Relationships II

7
```
# Relationship between clients and previous loans

es.add_relationship(
    parent_dataframe_name="clients",
    parent_column_name="client_id",
    child_dataframe_name="loans",
    child_column_name="client_id",
)
print(es)
```

```
Entityset: clients
  DataFrames:
    clients [Rows: 25, Columns: 4]
    loans [Rows: 443, Columns: 8]
    payments [Rows: 3456, Columns: 5]
  Relationships:
    loans.client_id -> clients.client_id
```

# Walk-Through: Step 3 — Create Table Relationships III

8
```
# Relationship between previous loans and previous payments

es.add_relationship(
    parent_dataframe_name="loans",
    parent_column_name="loan_id",
    child_dataframe_name="payments",
    child_column_name="loan_id",
)
print(es)
```

```
Entityset: clients
  DataFrames:
    clients [Rows: 25, Columns: 4]
    loans [Rows: 443, Columns: 8]
    payments [Rows: 3456, Columns: 5]
  Relationships:
    loans.client_id -> clients.client_id
    payments.loan_id -> loans.loan_id
```

# Walk-Through: Step 4 — Create Feature Primitives I

New features are created in featuretools using primitives either by themselves or stacking multiple primitives. Below is a list of some of the feature primitives in featuretools[‡]:

| | name | type | dask_compatible | spark_compatible | description | valid_inputs | return_type |
|---|---|---|---|---|---|---|---|
| **0** | trend | aggregation | False | False | Calculates the trend of a column over time. | <ColumnSchema (Logical Type = Datetime) (Seman... | <ColumnSchema (Semantic Tags = ['numeric'])> |
| **1** | max | aggregation | True | True | Calculates the highest value, ignoring `NaN` v... | <ColumnSchema (Semantic Tags = ['numeric'])> | <ColumnSchema (Semantic Tags = ['numeric'])> |
| **2** | time_since_last | aggregation | False | False | Calculates the time elapsed since the last dat... | <ColumnSchema (Logical Type = Datetime) (Seman... | <ColumnSchema (Logical Type = Double) (Semanti... |
| **3** | any | aggregation | True | False | Determines if any value is 'True' in a list. | <ColumnSchema (Logical Type = Boolean)>, <Colu... | <ColumnSchema (Logical Type = Boolean)> |
| **4** | mode | aggregation | False | False | Determines the most commonly repeated value. | <ColumnSchema (Semantic Tags = ['category'])> | None |
| **...** | ... | ... | ... | ... | ... | ... | ... |
| **108** | isin | transform | True | True | Determines whether a value is present in a pro... | <ColumnSchema> | <ColumnSchema (Logical Type = Boolean)> |
| **109** | rolling_max | transform | False | False | Determines the maximum of entries over a given... | <ColumnSchema (Logical Type = Datetime) (Seman... | <ColumnSchema (Logical Type = Double) (Semanti... |
| **110** | is_quarter_start | transform | True | True | Determines the is_quarter_start attribute of a... | <ColumnSchema (Logical Type = Datetime)> | <ColumnSchema (Logical Type = BooleanNullable)> |
| **111** | subtract_numeric | transform | True | False | Element-wise subtraction of two lists. | <ColumnSchema (Semantic Tags = ['numeric'])> | <ColumnSchema (Semantic Tags = ['numeric'])> |
| **112** | and | transform | True | True | Element-wise logical AND of two lists. | <ColumnSchema (Logical Type = Boolean)>, <Colu... | <ColumnSchema (Logical Type = BooleanNullable)> |

113 rows × 7 columns

[‡]We can also define custom primitives.

# Walk-Through: Step 4 — Generate Feature Primitives I

## Step 4 — Generate Feature Primitives

To make features with specified primitives we use the `ft.dfs` function (standing for deep feature synthesis). We pass in the `entityset`, the `target_dataframe_name`, which is the table where we want to add the features, the selected `trans_primitives` (transformations), and `agg_primitives` (aggregations):

9
```python
# Create new features using specified primitives
features, feature_names = ft.dfs(
    entityset = es,
    target_dataframe_name = "clients",
    agg_primitives = ["mean", "max", "last"],
    trans_primitives = ["year", "month", "subtract_numeric", "divide_numeric"])

print("Number of features", len(features.columns))
```

```
Number of features 288
```

The result is a dataframe of new features for each client (because we made clients the `target_entity`).

# Walk-Through: Step 4 — Generate Feature Primitives II

To see the generated features (288 features) use

10
```
pd.DataFrame(features["MONTH(joined)"].head())
```

11
```
pd.DataFrame(features['MEAN(payments.payment_amount)'].head())
```

| | MONTH(joined) |
|---|---|
| **client_id** | |
| **42320** | 4 |
| **39384** | 6 |
| **26945** | 11 |
| **41472** | 11 |
| **46180** | 11 |

| | MEAN(payments.payment_amount) |
|---|---|
| **client_id** | |
| **42320** | 1021.483333 |
| **39384** | 1193.630137 |
| **26945** | 1109.473214 |
| **41472** | 1129.076190 |
| **46180** | 1186.550336 |

# Walk-Through: Step 4 — Generate Feature Primitives II

Or look at the entire feature dataframe using

12
```
features.head()
```

Even though we specified only a few feature primitives, featuretools created many new features by combining and stacking these primitives.

| | income | credit_score | LAST(loans.loan_amount) | LAST(loans.loan_id) | LAST(loans.loan_type) | LAST(loans.rate) | LAST(loans.repaid) | MAX(loans.loan_ |
|---|---|---|---|---|---|---|---|---|
| **client_id** | | | | | | | | |
| **42320** | 229481 | 563 | 8090 | 10156 | home | 3.18 | 0 | 13887.0 |
| **39384** | 191204 | 617 | 14654 | 11735 | other | 2.26 | 0 | 14654.0 |
| **26945** | 214516 | 806 | 9249 | 11482 | cash | 2.86 | 1 | 14593.0 |
| **41472** | 152214 | 638 | 10122 | 11936 | cash | 1.03 | 0 | 13657.0 |
| **46180** | 43851 | 562 | 3834 | 10887 | other | 1.38 | 0 | 14081.0 |

5 rows × 288 columns

# Deep Feature Synthesis

- Now that we have covered process, let's look at the last step — deep feature synthesis (DFS) — a little closer.
- A deep feature is simply a feature made of stacking multiple primitives and DFS is the name of process that makes these features. The depth of a deep feature is the number of primitives required to make the feature.
  - For example, the

    `MEAN(payments.payment_amount)`

    column is a deep feature with a depth of 1 because it was created using a single aggregation.
  - A feature with a depth of two is

    `LAST(loans(MEAN(payments.payment_amount))`

    This is made by stacking two aggregations: LAST (most recent) on top of MEAN. This represents the average payment size of the most recent loan for each client.
- We can stack features to any depth we want, but in practice, a max depth of 2 is recommended. After this point, the features are difficult to interpret.
- We do not have to manually specify the feature primitives, but can let featuretools automatically choose features for us. To do this, we call `ft.dfs` but do not pass in any feature primitives and set `max_depth`.

# Final Comments

- This dataset is incomplete (no target) and was synthetic, but we (you) will look at a real dataset in week 6 practical.
- However, generation of features is trivial and scalable, (think about the InstaCart dataset).
- Also — just in case you were thinking about it — it is probably of little use in the Churn dataset, The number of observations is so small that manual engineering would be more effective.
- Note: Featuretools underwent significant API when moved to version 1.0. Make sure you are using up to date docmentation.