
Featuretools Documentation

Release 0.13.2

Feature Labs, Inc.

Feb 01, 2020

GETTING STARTED

1	5 Minute Quick Start	3
2	What's next?	11
3	Table of contents	13
4	Other links	251
	Index	253



Featuretools is a framework to perform automated feature engineering. It excels at transforming temporal and relational datasets into feature matrices for machine learning.

5 MINUTE QUICK START

Below is an example of using Deep Feature Synthesis (DFS) to perform automated feature engineering. In this example, we apply DFS to a multi-table dataset consisting of timestamped customer transactions.

```
In [1]: import featuretools as ft
```

1.1 Load Mock Data

```
In [2]: data = ft.demo.load_mock_customer()
```

1.2 Prepare data

In this toy dataset, there are 3 tables. Each table is called an `entity` in Featuretools.

- **customers:** unique customers who had sessions
- **sessions:** unique sessions and associated attributes
- **transactions:** list of events in this session

```
In [3]: customers_df = data["customers"]
```

```
In [4]: customers_df
```

```
Out [4]:
```

	customer_id	zip_code	join_date	date_of_birth
0	1	60091	2011-04-17 10:48:33	1994-07-18
1	2	13244	2012-04-15 23:31:04	1986-08-18
2	3	13244	2011-08-13 15:42:34	2003-11-21
3	4	60091	2011-04-08 20:08:14	2006-08-15
4	5	60091	2010-07-17 05:27:50	1984-07-28

```
In [5]: sessions_df = data["sessions"]
```

```
In [6]: sessions_df.sample(5)
```

```
Out [6]:
```

	session_id	customer_id	device	session_start
13	14	1	tablet	2014-01-01 03:28:00
6	7	3	tablet	2014-01-01 01:39:40
1	2	5	mobile	2014-01-01 00:17:20
28	29	1	mobile	2014-01-01 07:10:05

(continues on next page)

(continued from previous page)

```
24          25          3 desktop 2014-01-01 05:59:40

In [7]: transactions_df = data["transactions"]

In [8]: transactions_df.sample(5)
Out [8]:
```

	transaction_id	session_id	transaction_time	product_id	amount
74	232	5	2014-01-01 01:20:10	1	139.20
231	27	17	2014-01-01 04:10:15	2	90.79
434	36	31	2014-01-01 07:50:10	3	62.35
420	56	30	2014-01-01 07:35:00	3	72.70
54	444	4	2014-01-01 00:58:30	4	43.59

First, we specify a dictionary with all the entities in our dataset.

```
In [9]: entities = {
...:     "customers" : (customers_df, "customer_id"),
...:     "sessions" : (sessions_df, "session_id", "session_start"),
...:     "transactions" : (transactions_df, "transaction_id", "transaction_time")
...: }
```

Second, we specify how the entities are related. When two entities have a one-to-many relationship, we call the “one” entity, the “parent entity”. A relationship between a parent and child is defined like this:

```
(parent_entity, parent_variable, child_entity, child_variable)
```

In this dataset we have two relationships

```
In [10]: relationships = [("sessions", "session_id", "transactions", "session_id"),
...:                      ("customers", "customer_id", "sessions", "customer_id")]
...: 
```

Note: To manage setting up entities and relationships, we recommend using the *EntitySet* class which offers convenient APIs for managing data like this. See *Representing Data with EntitySets* for more information.

1.3 Run Deep Feature Synthesis

A minimal input to DFS is a set of entities, a list of relationships, and the “target_entity” to calculate features for. The output of DFS is a feature matrix and the corresponding list of feature definitions.

Let’s first create a feature matrix for each customer in the data

```
In [11]: feature_matrix_customers, features_defs = ft.dfs(entities=entities,
...:                                                     relationships=relationships,
...:                                                     target_entity="customers")
...: 
```

```
In [12]: feature_matrix_customers
Out [12]:
```

	zip_code	COUNT(sessions)	NUM_UNIQUE(sessions.device)	MODE(sessions. ↪device)	SUM(transactions.amount)	STD(transactions.amount)	MAX(transactions. ↪amount)	SKEW(transactions.amount)	MIN(transactions.amount)	MEAN(transactions. ↪amount)	COUNT(transactions)	NUM_UNIQUE(transactions.product_id)	MODE(sessions. ↪MODE(transactions.product_id)	DAY(date_of_birth)	YEAR(join_date)	YEAR(date_of ↪birth)	YEAR(join_date)	MONTH(date_of_birth)	MONTH(join_date)	WEEKDAY(date_of ↪birth)	WEEKDAY(join_date)	SUM(sessions.MAX(transactions.amount))	SUM(sessions. ↪SKEW(transactions.amount))	SUM(sessions.MIN(transactions.amount))	SUM(sessions. ↪NUM_UNIQUE(transactions.product_id))	SUM(sessions.MEAN(transactions.amount))	SUM(sessions.STD(transactions.amount))	STD(sessions.MAX(transactions.amount))
4																												

(continued from previous page)

2	13244		7		3		
→desktop		7200.28			37.705178		146.
→81		0.098259			8.73		77.422366
→	93				5		
→4		18		15	1986		2012
→	8		4		0		6
→		931.63					-0.277640
→		154.60					35
→		548.905851					258.700528
→		17.221593					251.609234
→		0.509798					15.874374
→		0.000000					11.477071
→		3.450328					1320.64
→		0.755711					56.46
→			5				96.581000
→		18					47.935920
→		-1.539467					-0.440929
→		2.154929					0.000000
→		0.235296					-0.303276
→		0.013087					100.04
→		634.84					-0.763603
→			5				61.910000
→		8					27.839228
→	133.090000						1028.611429
→	-0.039663						22.085714
→		5.000000					78.415122
→		13.285714					36.957218
→		1					1
→		1					1
→			4				2014
→		2					1
→	1						3
→		1					3
→		2					desktop
3	13244		6		3		
→desktop		6236.62			43.683296		149.
→15		0.418230			5.89		67.060430
→	93				5		
→1		21		13	2003		2011
→	11		8		4		5
→		847.63					2.286086
→		66.21					29
→		405.237462					257.299895
→		10.724241					219.021420
→		0.429374					5.424407
→		0.408248					11.174282
→		2.428992					1477.97
→		0.854976					20.06
→			5				82.109444
→		18					50.110120
→		-0.941078					2.246479
→		1.000771					-2.449490
→		0.678544					-1.507217
→		-0.245703					126.74
→		889.21					-0.289466
→			4				55.579412
→		11					35.704680
→	141.271667						1039.436667
→	0.381014						11.035000
→		4.833333					67.539577
6	15.500000						42.883916
→		1					1
→		1					1
→			4				2014

(continues on next page)

(continued from previous page)

4	60091		8		3		
→mobile		8727.68			45.068765		149.
→95		-0.036348			5.73		80.070459
→	109				5		
→ 2		15		8	2006		2011
→	8		4		1		4
→		1157.99				0.002764	
→		131.51					37
→		649.657515				356.125829	
→		3.514421				235.992478	
→		0.387884				16.960575	
→		0.517549					13.027258
→		3.335416				1351.46	
→		0.382868				54.83	
→			5			110.450000	
→		18				54.293903	
→		0.027256				-0.391805	
→		2.103510				-0.644061	
→		1.980948				0.282488	
→		-1.065663				139.20	
→		771.68				-0.711744	
→			4			70.638182	
→		10				29.026424	
→	144.748750					1090.960000	
→	0.000346					16.438750	
→		4.625000				81.207189	
→		13.625000				44.515729	
→		1				1	
→		1				1	
→			5			2014	
→		2				1	
→	1					1	
→		1				3	
→		4			mobile		
5	60091		6		3		
→mobile		6349.66			44.095630		149.
→02		-0.025941			7.55		80.375443
→	79				5		
→ 5		28		17	1984		2010
→	7		7		5		5
→		839.76				0.014384	
→		86.49					30
→		472.231119				259.873954	
→		7.928001				402.775486	
→		0.415426				4.961414	
→		0.000000					11.007471
→		3.600926				1700.67	
→		0.602209				20.65	
→			5			94.481667	
→		18				51.149250	
→		-0.333796				0.472342	
→		-0.470410				0.000000	
→		0.335175				-0.317685	
→		0.204548				128.51	
→		543.18				-0.539060	
→			5			66.666667	
→		8				36.734681	
→	139.960000				1058.276667		(continues on next page)
→	0.002397				14.415000		
→		5.000000			78.705187		
→		13.166667			43.312326		
→		1			1		
→		1			1		
→			5			2014	

1.3. Run Deep Feature Synthesis

(continued from previous page)

2		5	mobile		746.96		45.893591	u
→		135.25			-0.160550		9.32	u
→		74.696000		10			5	u
→			5		1	2014		u
→	1		2		60091			u
→		1				1		u
→			1				1	u
→			2014					2u
→				1				u
→	1		6			3		u
→		mobile			6349.66		44.	u
→	095630				149.02		-0.025941	u
→			7.55			80.375443		u
→		79				5		u
→		5			28		17	u
→		1984			2010		7	u
→			7			5		u
→	5							u
3		4	mobile		1329.00		46.240016	u
→		147.73			-0.324012		8.70	u
→		88.600000		15			5	u
→			1		1	2014		u
→	1		2		60091			u
→		1				1		u
→			1				1	u
→			2014					2u
→				1				u
→	1		8			3		u
→		mobile			8727.68		45.	u
→	068765				149.95		-0.036348	u
→			5.73			80.070459		u
→		109				5		u
→		2			15		8	u
→		2006			2011			8u
→			4			1		u
→	4							u
4		1	mobile		1613.93		40.187205	u
→		129.00			0.234349		6.29	u
→		64.557200		25			5	u
→			5		1	2014		u
→	1		2		60091			u
→		1				1		u
→			1				1	u
→			2014					2u
→				1				u
→	1		8			3		u
→		mobile			9025.62		40.	u
→	442059				139.43		0.019698	u
→			5.81			71.631905		u
→		126				5		u
→		4			18		17	u
→		1994			2011		7	u
→			4			0		u
→	6							u
5		4	mobile		777.02		48.918663	u
→		139.20			0.336381		7.43	u
→		70.638182		11			5	u
→			5		1	2014		(continues on next page)
→	1		2		60091			u
→		1				1		u
→			1				1	9u
→			2014					2u
→				1				u
→	1		8			3		u

1.4. Change target entity

→			1				1	9
→				2014				2u
→					1			u
→	1		8			3		u

(continued from previous page)



WHAT'S NEXT?

- Learn about *Representing Data with EntitySets*
- Apply automated feature engineering with *Deep Feature Synthesis*
- Explore [runnable demos](#) based on real world use cases
- Can't find what you're looking for? Ask for *Help*

TABLE OF CONTENTS

3.1 Install

Featuretools is available for Python 3.5, 3.6 and 3.7 The recommended way to install Featuretools is using pip or conda:

```
python -m pip install featuretools
```

or from the Conda-forge channel on anaconda.org:

```
conda install -c conda-forge featuretools
```

3.1.1 Add-ons

You can install add-ons individually or all at once by running:

```
python -m pip install featuretools[complete]
```

Update checker: Receive automatic notifications of new Featuretools releases:

```
python -m pip install featuretools[update_checker]
```

TSFresh Primitives: Use 60+ primitives from `tsfresh` in Featuretools:

```
python -m pip install featuretools[tsfresh]
```

Categorical Encoding: Encode categorical data for integration into Featuretools/machine learning workflows:

```
python -m pip install featuretools[categorical_encoding]
```

NLP Primitives: Use Natural Language Processing Primitives for data with text in Featuretools:

```
python -m pip install featuretools[nlp_primitives]
```

AutoNormalize: Automated creation of normalized `EntitySet` from denormalized data:

```
python -m pip install featuretools[autonormalize]
```

Featuretools Sklearn Transformer: Deep Feature Synthesis as a scikit-learn pipelines transformer

```
python -m pip install featuretools[sklearn_transformer]
```

3.1.2 Installing Graphviz

In order to use `EntitySet.plot` you will need to install the graphviz library.

Conda users:

```
conda install python-graphviz
```

Ubuntu:

```
sudo apt-get install graphviz  
pip install graphviz
```

Mac OS:

```
brew install graphviz  
pip install graphviz
```

Windows:

```
conda install python-graphviz
```

3.1.3 Install from Source

To install featuretools from source, clone the repository from [github](#):

```
git clone https://github.com/featuretools/featuretools.git  
cd featuretools  
python setup.py install
```

or use `pip` locally if you want to install all dependencies as well:

```
pip install .
```

You can view the list of all dependencies within the `extras_require` field of `setup.py`.

3.1.4 Development

Before making contributing to the codebase, please follow the [guidelines here](#)

Virtualenv

We recommend developing in a [virtualenv](#):

```
mkvirtualenv featuretools
```

Install development requirements

Run:

```
make installdeps
```

Test

Note: In order to run the featuretools tests you will need to have graphviz installed as described above.

Run featuretools tests:

```
make test
```

Before committing make sure to run linting in order to pass CI:

```
make lint
```

Some linting errors can be automatically fixed by running the command below:

```
make lint-fix
```

Build Documentation

Build the docs with the commands below:

```
cd docs/  
  
# small changes  
make html  
  
# rebuild from scratch  
make clean html
```

Note: The Featuretools library must be import-able to build the docs.

3.2 Representing Data with EntitySets

An `EntitySet` is a collection of entities and the relationships between them. They are useful for preparing raw, structured datasets for feature engineering. While many functions in Featuretools take entities and relationships as separate arguments, it is recommended to create an `EntitySet`, so you can more easily manipulate your data as needed.

3.2.1 The Raw Data

Below we have a two tables of data (represented as Pandas DataFrames) related to customer transactions. The first is a merge of transactions, sessions, and customers so that the result looks like something you might see in a log file:

```
In [1]: import featuretools as ft  
  
In [2]: data = ft.demo.load_mock_customer()  
  
In [3]: transactions_df = data["transactions"].merge(data["sessions"]).merge(data[  
↪ "customers"])
```

(continues on next page)

(continued from previous page)

```
In [4]: transactions_df.sample(10)
Out [4]:
```

	transaction_id	session_id	transaction_time	product_id	amount	customer_id
↳device		session_start	zip_code	join_date	date_of_birth	
264		380	21 2014-01-01 05:14:10	5	57.09	4
↳desktop	2014-01-01 05:02:15	60091	2011-04-08 20:08:14	2006-08-15		
19	244	10	2014-01-01 02:34:55	2	116.95	2
↳tablet	2014-01-01 02:31:40	13244	2012-04-15 23:31:04	1986-08-18		
314	299	6	2014-01-01 01:32:05	4	64.99	1
↳tablet	2014-01-01 01:23:25	60091	2011-04-17 10:48:33	1994-07-18		
290	78	4	2014-01-01 00:54:10	1	37.50	1
↳mobile	2014-01-01 00:44:25	60091	2011-04-17 10:48:33	1994-07-18		
379	457	27	2014-01-01 06:37:35	1	19.16	1
↳mobile	2014-01-01 06:34:20	60091	2011-04-17 10:48:33	1994-07-18		
335	477	9	2014-01-01 02:30:35	3	41.70	1
↳desktop	2014-01-01 02:15:25	60091	2011-04-17 10:48:33	1994-07-18		
293	103	4	2014-01-01 00:57:25	5	20.79	1
↳mobile	2014-01-01 00:44:25	60091	2011-04-17 10:48:33	1994-07-18		
271	390	22	2014-01-01 05:21:45	2	54.83	4
↳desktop	2014-01-01 05:21:45	60091	2011-04-08 20:08:14	2006-08-15		
404	476	29	2014-01-01 07:24:10	4	121.59	1
↳mobile	2014-01-01 07:10:05	60091	2011-04-17 10:48:33	1994-07-18		
179	90	3	2014-01-01 00:35:45	1	75.73	4
↳mobile	2014-01-01 00:28:10	60091	2011-04-08 20:08:14	2006-08-15		

And the second dataframe is a list of products involved in those transactions.

```
In [5]: products_df = data["products"]

In [6]: products_df
Out [6]:
```

	product_id	brand
0	1	B
1	2	B
2	3	B
3	4	B
4	5	A

3.2.2 Creating an EntitySet

First, we initialize an EntitySet. If you'd like to give it name, you can optionally provide an id to the constructor.

```
In [7]: es = ft.EntitySet(id="customer_data")
```

3.2.3 Adding entities

To get started, we load the transactions dataframe as an entity.

```
In [8]: es = es.entity_from_dataframe(entity_id="transactions",
...:                                 dataframe=transactions_df,
...:                                 index="transaction_id",
...:                                 time_index="transaction_time",
```

(continues on next page)

(continued from previous page)

```

.....:         variable_types={"product_id": ft.variable_types.
↳Categorical,
.....:         "zip_code": ft.variable_types.
↳ZIPCode})
.....:

In [9]: es
Out [9]:
Entityset: customer_data
Entities:
  transactions [Rows: 500, Columns: 11]
Relationships:
  No relationships

```

Note: You can also display your entity set structure graphically by calling `EntitySet.plot()`.

This method loads each column in the dataframe in as a variable. We can see the variables in an entity using the code below.

```

In [10]: es["transactions"].variables
Out [10]:
[<Variable: transaction_id (dtype = index)>,
 <Variable: session_id (dtype = numeric)>,
 <Variable: transaction_time (dtype: datetime_time_index, format: None)>,
 <Variable: amount (dtype = numeric)>,
 <Variable: customer_id (dtype = numeric)>,
 <Variable: device (dtype = categorical)>,
 <Variable: session_start (dtype: datetime, format: None)>,
 <Variable: join_date (dtype: datetime, format: None)>,
 <Variable: date_of_birth (dtype: datetime, format: None)>,
 <Variable: product_id (dtype = categorical)>,
 <Variable: zip_code (dtype = zipcode)>]

```

In the call to `entity_from_dataframe`, we specified three important parameters

- The `index` parameter specifies the column that uniquely identifies rows in the dataframe
- The `time_index` parameter tells Featuretools when the data was created.
- The `variable_types` parameter indicates that “product_id” should be interpreted as a Categorical variable, even though it just an integer in the underlying data.

Now, we can do that same thing with our products dataframe

```

In [11]: es = es.entity_from_dataframe(entity_id="products",
.....:         dataframe=products_df,
.....:         index="product_id")
.....:

In [12]: es
Out [12]:
Entityset: customer_data
Entities:
  transactions [Rows: 500, Columns: 11]
  products [Rows: 5, Columns: 2]

```

(continues on next page)

```
Relationships:
  No relationships
```

With two entities in our entity set, we can add a relationship between them.

3.2.4 Adding a Relationship

We want to relate these two entities by the columns called “product_id” in each entity. Each product has multiple transactions associated with it, so it is called it the **parent entity**, while the transactions entity is known as the **child entity**. When specifying relationships we list the variable in the parent entity first. Note that each *ft.Relationship* must denote a one-to-many relationship rather than a relationship which is one-to-one or many-to-many.

```
In [13]: new_relationship = ft.Relationship(es["products"]["product_id"],
      ..: ..:                 es["transactions"]["product_id"])
      ..: ..:
      ..: ..:

In [14]: es = es.add_relationship(new_relationship)

In [15]: es
Out [15]:
Entityset: customer_data
Entities:
  transactions [Rows: 500, Columns: 11]
  products [Rows: 5, Columns: 2]
Relationships:
  transactions.product_id -> products.product_id
```

Now, we see the relationship has been added to our entity set.

3.2.5 Creating entity from existing table

When working with raw data, it is common to have sufficient information to justify the creation of new entities. In order to create a new entity and relationship for sessions, we “normalize” the transaction entity.

```
In [16]: es = es.normalize_entity(base_entity_id="transactions",
      ..: ..:                   new_entity_id="sessions",
      ..: ..:                   index="session_id",
      ..: ..:                   make_time_index="session_start",
      ..: ..:                   additional_variables=["device", "customer_id", "zip_
      ↪code", "session_start", "join_date"])
      ..: ..:

In [17]: es
Out [17]:
Entityset: customer_data
Entities:
  transactions [Rows: 500, Columns: 6]
  products [Rows: 5, Columns: 2]
  sessions [Rows: 35, Columns: 6]
Relationships:
  transactions.product_id -> products.product_id
  transactions.session_id -> sessions.session_id
```

Looking at the output above, we see this method did two operations

1. It created a new entity called “sessions” based on the “session_id” and “session_start” variables in “transactions”
2. It added a relationship connecting “transactions” and “sessions”.

If we look at the variables in transactions and the new sessions entity, we see two more operations that were performed automatically.

```
In [18]: es["transactions"].variables
Out [18]:
[<Variable: transaction_id (dtype = index)>,
 <Variable: session_id (dtype = id)>,
 <Variable: transaction_time (dtype: datetime_time_index, format: None)>,
 <Variable: amount (dtype = numeric)>,
 <Variable: date_of_birth (dtype: datetime, format: None)>,
 <Variable: product_id (dtype = id)>]

In [19]: es["sessions"].variables
////////////////////////////////////
↪
[<Variable: session_id (dtype = index)>,
 <Variable: device (dtype = categorical)>,
 <Variable: customer_id (dtype = numeric)>,
 <Variable: zip_code (dtype = zipcode)>,
 <Variable: session_start (dtype: datetime_time_index, format: None)>,
 <Variable: join_date (dtype: datetime, format: None)>]
```

1. It removed “device”, “customer_id”, “zip_code” and “join_date” from “transactions” and created a new variables in the sessions entity. This reduces redundant information as the those properties of a session don’t change between transactions.
2. It copied and marked “session_start” as a time index variable into the new sessions entity to indicate the beginning of a session. If the base entity has a time index and make_time_index is not set, normalize_entity will create a time index for the new entity. In this case it would create a new time index called “first_transactions_time” using the time of the first transaction of each session. If we don’t want this time index to be created, we can set make_time_index=False.

If we look at the dataframes, can see what the normalize_entity did to the actual data.

```
In [20]: es["sessions"].df.head(5)
Out [20]:
   session_id  device  customer_id  zip_code  session_start  join_date
1           1  desktop             2   13244 2014-01-01 00:00:00 2012-04-15 23:31:04
2           2  mobile             5   60091 2014-01-01 00:17:20 2010-07-17 05:27:50
3           3  mobile             4   60091 2014-01-01 00:28:10 2011-04-08 20:08:14
4           4  mobile             1   60091 2014-01-01 00:44:25 2011-04-17 10:48:33
5           5  mobile             4   60091 2014-01-01 01:11:30 2011-04-08 20:08:14

In [21]: es["transactions"].df.head(5)
////////////////////////////////////
↪
   transaction_id  session_id  transaction_time  amount  date_of_birth  product_id
298              298           1 2014-01-01 00:00:00  127.64  1986-08-18           5
2                  2           1 2014-01-01 00:01:05  109.48  1986-08-18           2
308              308           1 2014-01-01 00:02:10   95.06  1986-08-18           3
116             116           1 2014-01-01 00:03:15   78.92  1986-08-18           4
371             371           1 2014-01-01 00:04:20   31.54  1986-08-18           3
```

To finish preparing this dataset, create a “customers” entity using the same method call.

```
In [22]: es = es.normalize_entity(base_entity_id="sessions",
.....:                          new_entity_id="customers",
.....:                          index="customer_id",
.....:                          make_time_index="join_date",
.....:                          additional_variables=["zip_code", "join_date"])
.....:
```

In [23]: es

Out [23]:

```
Entityset: customer_data
Entities:
  transactions [Rows: 500, Columns: 6]
  products [Rows: 5, Columns: 2]
  sessions [Rows: 35, Columns: 4]
  customers [Rows: 5, Columns: 3]
Relationships:
  transactions.product_id -> products.product_id
  transactions.session_id -> sessions.session_id
  sessions.customer_id -> customers.customer_id
```

3.2.6 Using the EntitySet

Finally, we are ready to use this EntitySet with any functionality within Featuretools. For example, let's build a feature matrix for each product in our dataset.

```
In [24]: feature_matrix, feature_defs = ft.dfs(entityset=es,
.....:                                       target_entity="products")
.....:
```

In [25]: feature_matrix

Out [25]:

```
brand SUM(transactions.amount) STD(transactions.amount)
↳MAX(transactions.amount) SKEW(transactions.amount) MIN(transactions.amount)
↳MEAN(transactions.amount) COUNT(transactions) NUM_UNIQUE(transactions.session_id)
↳ MODE(transactions.session_id) NUM_UNIQUE(transactions.YEAR(date_of_birth)) NUM_
↳UNIQUE(transactions.sessions.customer_id) NUM_UNIQUE(transactions.sessions.device)
↳ NUM_UNIQUE(transactions.WEEKDAY(date_of_birth)) NUM_UNIQUE(transactions.
↳YEAR(transaction_time)) NUM_UNIQUE(transactions.MONTH(date_of_birth)) NUM_
↳UNIQUE(transactions.WEEKDAY(transaction_time)) NUM_UNIQUE(transactions.DAY(date_of_
↳birth)) NUM_UNIQUE(transactions.MONTH(transaction_time)) NUM_UNIQUE(transactions.
↳DAY(transaction_time)) MODE(transactions.YEAR(date_of_birth)) MODE(transactions.
↳sessions.customer_id) MODE(transactions.sessions.device) MODE(transactions.
↳WEEKDAY(date_of_birth)) MODE(transactions.YEAR(transaction_time))
↳MODE(transactions.MONTH(date_of_birth)) MODE(transactions.WEEKDAY(transaction_
↳time)) MODE(transactions.DAY(date_of_birth)) MODE(transactions.MONTH(transaction_
↳time)) MODE(transactions.DAY(transaction_time))
product_id
```

(continues on next page)

(continued from previous page)

1		B	7489.79	42.479989		
→	149.56		0.125525	6.84	73.	↳
→	429314		102	34		↳
→		3		5		↳
→			5		3	↳
→			4			↳
→					1	↳
→						↳
→	1			4		↳
→		1			1	↳
→		1994			1	↳
→	desktop			0		↳
→	2014			7		↳
→		2		18		↳
→		1			1	↳
2		B	7021.43	46.336308		↳
→	149.95		0.151934	5.73	76.	↳
→	319891		92	34		↳
→		28		5		↳
→			5		3	↳
→			4			↳
→					1	↳
→	1			4		↳
→		1			1	↳
→		2006			4	↳
→	desktop			0		↳
→	2014			8		↳
→		2		18		↳
→		1			1	↳
3		B	7008.12	38.871405		↳
→	148.31		0.223938	5.89	73.	↳
→	001250		96	35		↳
→		1		5		↳
→			5		3	↳
→			4			↳
→					1	↳
→	1			4		↳
→		1			1	↳
→		2006			4	↳
→	desktop			0		↳
→	2014			8		↳
→		2		18		↳
→		1			1	↳
4		B	8088.97	42.492501		↳
→	146.46		-0.132077	5.81	76.	↳
→	311038		106	34		↳
→		29		5		↳
→			5		3	↳
→			4			↳
→					1	↳
→	1			4		↳
→		1			1	↳
→		1994			1	↳
→	desktop			0		↳
→	2014			7		↳
→		2		18		↳
→		1			1	↳
5		A	7931.55	42.131902		↳
→	149.02		0.098248	5.91		↳
→	264904		104	34		↳

(continues on next page)

3.2. Representing Data with EntitySets

As we can see, the features from DFS use the relational structure of our entity set. Therefore it is important to think carefully about the entities that we create.

3.3 Deep Feature Synthesis

Deep Feature Synthesis (DFS) is an automated method for performing feature engineering on relational and temporal data.

3.3.1 Input Data

Deep Feature Synthesis requires structured datasets in order to perform feature engineering. To demonstrate the capabilities of DFS, we will use a mock customer transactions dataset.

Note: Before using DFS, it is recommended that you prepare your data as an *EntitySet*. See *Representing Data with EntitySets* to learn how.

```
In [1]: import featuretools as ft

In [2]: es = ft.demo.load_mock_customer(return_entityset=True)

In [3]: es
Out[3]:
Entityset: transactions
  Entities:
    transactions [Rows: 500, Columns: 5]
    products [Rows: 5, Columns: 2]
    sessions [Rows: 35, Columns: 4]
    customers [Rows: 5, Columns: 4]
  Relationships:
    transactions.product_id -> products.product_id
    transactions.session_id -> sessions.session_id
    sessions.customer_id -> customers.customer_id
```

Once data is prepared as an *EntitySet*, we are ready to automatically generate features for a target entity - e.g. customers.

3.3.2 Running DFS

Typically, without automated feature engineering, a data scientist would write code to aggregate data for a customer, and apply different statistical functions resulting in features quantifying the customer's behavior. In this example, an expert might be interested in features such as: *total number of sessions* or *month the customer signed up*.

These features can be generated by DFS when we specify the `target_entity` as `customers` and `"count"` and `"month"` as primitives.

```
In [4]: feature_matrix, feature_defs = ft.dfs(entityset=es,
...:                                       target_entity="customers",
...:                                       agg_primitives=["count"],
```

(continues on next page)

(continued from previous page)

```

...:         trans_primitives=["month"],
...:         max_depth=1)
...:
In [5]: feature_matrix
Out [5]:
   zip_code  COUNT(sessions)  MONTH(date_of_birth)  MONTH(join_date)
customer_id
5         60091             6                    7                7
4         60091             8                    8                4
1         60091             8                    7                4
3         13244             6                   11                8
2         13244             7                    8                4

```

In the example above, "count" is an **aggregation primitive** because it computes a single value based on many sessions related to one customer. "month" is called a **transform primitive** because it takes one value for a customer transforms it to another.

Note: Feature primitives are a fundamental component to Featuretools. To learn more read *Feature primitives*.

3.3.3 Creating “Deep Features”

The name Deep Feature Synthesis comes from the algorithm’s ability to stack primitives to generate more complex features. Each time we stack a primitive we increase the “depth” of a feature. The `max_depth` parameter controls the maximum depth of the features returned by DFS. Let us try running DFS with `max_depth=2`

```

In [6]: feature_matrix, feature_defs = ft.dfs(entityset=es,
...:         target_entity="customers",
...:         agg_primitives=["mean", "sum", "mode"],
...:         trans_primitives=["month", "hour"],
...:         max_depth=2)
...:
In [7]: feature_matrix
Out [7]:
   zip_code  ...  MODE(transactions.sessions.device)
customer_id
5         60091  ...                mobile
4         60091  ...                mobile
1         60091  ...                mobile
3         13244  ...                desktop
2         13244  ...                desktop

[5 rows x 17 columns]

```

With a depth of 2, a number of features are generated using the supplied primitives. The algorithm to synthesize these definitions is described in this [paper](#). In the returned feature matrix, let us understand one of the depth 2 features

```

In [8]: feature_matrix[['MEAN(sessions.SUM(transactions.amount))']]
Out [8]:
   MEAN(sessions.SUM(transactions.amount))
customer_id
5                                1058.276667

```

(continues on next page)

(continued from previous page)

4	1090.960000
1	1128.202500
3	1039.436667
2	1028.611429

For each customer this feature

1. calculates the `sum` of all transaction amounts per session to get total amount per session,
2. then applies the `mean` to the total amounts across multiple sessions to identify the *average amount spent per session*

We call this feature a “deep feature” with a depth of 2.

Let’s look at another depth 2 feature that calculates for every customer *the most common hour of the day when they start a session*

```
In [9]: feature_matrix[['MODE(sessions.HOUR(session_start))']]
Out [9]:
           MODE(sessions.HOUR(session_start))
customer_id
5                0
4                1
1                6
3                5
2                3
```

For each customer this feature calculates

1. The `hour` of the day each of his or her sessions started, then
2. uses the statistical function `mode` to identify the most common hour he or she started a session

Stacking results in features that are more expressive than individual primitives themselves. This enables the automatic creation of complex patterns for machine learning.

3.3.4 Changing Target Entity

DFS is powerful because we can create a feature matrix for any entity in our dataset. If we switch our target entity to “sessions”, we can synthesize features for each session instead of each customer. Now, we can use these features to predict the outcome of a session.

```
In [10]: feature_matrix, feature_defs = ft.dfs(entityset=es,
.....:                                         target_entity="sessions",
.....:                                         agg_primitives=["mean", "sum", "mode"],
.....:                                         trans_primitives=["month", "hour"],
.....:                                         max_depth=2)

In [11]: feature_matrix.head(5)
Out [11]:
           customer_id  ...  customers.HOUR(join_date)
session_id
1                2  ...                23
2                5  ...                5
3                4  ...                20
4                1  ...                10
```

(continues on next page)

(continued from previous page)

```
5          4  ...          20
[5 rows x 19 columns]
```

As we can see, DFS will also build deep features based on a parent entity, in this case the customer of a particular session. For example, the feature below calculates the mean transaction amount of the customer of the session.

```
In [12]: feature_matrix[['customers.MEAN(transactions.amount)']].head(5)
Out [12]:
           customers.MEAN(transactions.amount)
session_id
1                77.422366
2                80.375443
3                80.070459
4                71.631905
5                80.070459
```

Improve feature output

To learn about the parameters to change in DFS read [Tuning Deep Feature Synthesis](#).

3.4 Feature primitives

Feature primitives are the building blocks of Featuretools. They define individual computations that can be applied to raw datasets to create new features. Because a primitive only constrains the input and output data types, they can be applied across datasets and can stack to create new calculations.

3.4.1 Why primitives?

The space of potential functions that humans use to create a feature is expansive. By breaking common feature engineering calculations down into primitive components, we are able to capture the underlying structure of the features humans create today.

A primitive only constrains the input and output data types. This means they can be used to transfer calculations known in one domain to another. Consider a feature which is often calculated by data scientists for transactional or event logs data: *average time between events*. This feature is incredibly valuable in predicting fraudulent behavior or future customer engagement.

DFS achieves the same feature by stacking two primitives "time_since_previous" and "mean"

```
In [1]: feature_defs = ft.dfs(entityset=es,
...:                         target_entity="customers",
...:                         agg_primitives=["mean"],
...:                         trans_primitives=["time_since_previous"],
...:                         features_only=True)
...:

In [2]: feature_defs
Out [2]:
[<Feature: zip_code>,
 <Feature: MEAN(transactions.amount)>,
 <Feature: TIME_SINCE_PREVIOUS(join_date)>]
```

(continues on next page)

(continued from previous page)

```
<Feature: MEAN(sessions.MEAN(transactions.amount))>,
<Feature: MEAN(sessions.TIME_SINCE_PREVIOUS(session_start))>]
```

Note: When `dfs` is called with `features_only=True`, only feature definitions are returned as output. By default this parameter is set to `False`. This parameter is used quickly inspect the feature definitions before the spending time calculating the feature matrix.

A second advantage of primitives is that they can be used to quickly enumerate many interesting features in a parameterized way. This is used by Deep Feature Synthesis to get several different ways of summarizing the time since the previous event.

```
In [3]: feature_matrix, feature_defs = ft.dfs(entityset=es,
...:                                       target_entity="customers",
...:                                       agg_primitives=["mean", "max", "min",
↪ "std", "skew"],
...:                                       trans_primitives=["time_since_previous
↪"])
...:
```

```
In [4]: feature_matrix[["MEAN(sessions.TIME_SINCE_PREVIOUS(session_start))",
...:                   "MAX(sessions.TIME_SINCE_PREVIOUS(session_start))",
...:                   "MIN(sessions.TIME_SINCE_PREVIOUS(session_start))",
...:                   "STD(sessions.TIME_SINCE_PREVIOUS(session_start))",
...:                   "SKEW(sessions.TIME_SINCE_PREVIOUS(session_start))"]]
...:
```

```
Out [4]:
           MEAN(sessions.TIME_SINCE_PREVIOUS(session_start))  MAX(sessions.TIME_
↪ SINCE_PREVIOUS(session_start))  MIN(sessions.TIME_SINCE_PREVIOUS(session_start))
↪ STD(sessions.TIME_SINCE_PREVIOUS(session_start))  SKEW(sessions.TIME_SINCE_
↪ PREVIOUS(session_start))
customer_id
↪
↪
↪
5           1170.0           1007.500000           715.0
↪
↪           157.884451
↪ -1.507217
4           1625.0           999.375000           650.0
↪
↪           308.688904
↪ 1.065177
1           1170.0           966.875000           715.0
↪
↪           171.754341
↪ -0.254557
3           1170.0           888.333333           650.0
↪
↪           177.613813
↪ 0.434581
2           975.0           725.833333           520.0
↪
↪           194.638554
↪ 0.162631
```

3.4.2 Aggregation vs Transform Primitive

In the example above, we use two types of primitives.

Aggregation primitives: These primitives take related instances as an input and output a single value. They are applied across a parent-child relationship in an entity set. E.g: "count", "sum", "avg_time_between".

Transform primitives: These primitives take one or more variables from an entity as an input and output a new variable for that entity. They are applied to a single entity. E.g: "hour", "time_since_previous", "absolute".

For a DataFrame that lists and describes each built-in primitive in Featuretools, call `ft.list_primitives()`.

```
In [5]: ft.list_primitives().head(5)
Out [5]:
```

	name	type	description
0	trend	aggregation	Calculates the trend of a variable over time.
1	first	aggregation	Determines the first value in a list.
2	skew	aggregation	Computes the extent to which a distribution di...
3	avg_time_between	aggregation	Computes the average number of seconds between...
4	percent_true	aggregation	Determines the percent of `True` values.

3.4.3 Defining Custom Primitives

The library of primitives in Featuretools is constantly expanding. Users can define their own primitive using the APIs below. To define a primitive, a user will

- Specify the type of primitive `Aggregation` or `Transform`
- Define the input and output data types
- Write a function in python to do the calculation
- Annotate with attributes to constrain how it is applied

Once a primitive is defined, it can stack with existing primitives to generate complex patterns. This enables primitives known to be important for one domain to automatically be transferred to another.

Simple Custom Primitives

```
In [6]: from featuretools.primitives import make_agg_primitive, make_trans_primitive
In [7]: from featuretools.variable_types import Text, Numeric
In [8]: def absolute(column):
...:     return abs(column)
...:
In [9]: Absolute = make_trans_primitive(function=absolute,
...:                                   input_types=[Numeric],
...:                                   return_type=Numeric)
...:
```

Above we created a new transform primitive that can be used with Deep Feature Synthesis using `make_trans_primitive` and a python function we defined. Additionally, we annotated the input data types that the primitive can be applied to and the data type it returns.

Similarly, we can make a new aggregation primitive using `make_agg_primitive`.

```
In [10]: def maximum(column):
.....:     return max(column)
.....:

In [11]: Maximum = make_agg_primitive(function=maximum,
.....:                                 input_types=[Numeric],
.....:                                 return_type=Numeric)
.....:
```

Because we defined an aggregation primitive, the function takes in a list of values but only returns one.

Now that we've defined two primitives, we can use them with the `dfs` function as if they were built-in primitives.

```
In [12]: feature_matrix, feature_defs = ft.dfs(entityset=es,
.....:                                       target_entity="sessions",
.....:                                       agg_primitives=[Maximum],
.....:                                       trans_primitives=[Absolute],
.....:                                       max_depth=2)
.....:

In [13]: feature_matrix[["customers.MAXIMUM(transactions.amount)",
↳ "MAXIMUM(transactions.ABSOLUTE(amount)) "]].head(5)
Out [13]:
           customers.MAXIMUM(transactions.amount)  MAXIMUM(transactions.
↳ ABSOLUTE(amount))
session_id
↳
1           146.81
↳ 141.66
2           149.02
↳ 135.25
3           149.95
↳ 147.73
4           139.43
↳ 129.00
5           149.95
↳ 139.20
```

Word Count Example

Here we define a function, `word_count`, which counts the number of words in each row of an input and returns a list of the counts.

```
In [14]: def word_count(column):
.....:     '''
.....:     Counts the number of words in each row of the column. Returns a list
.....:     of the counts for each row.
.....:     '''
.....:     word_counts = []
.....:     for value in column:
.....:         words = value.split(None)
.....:         word_counts.append(len(words))
.....:     return word_counts
.....:
```

Next, we need to create a custom primitive from the `word_count` function.


```
In [15]: WordCount = make_trans_primitive(function=word_count,
.....:                                 input_types=[Text],
.....:                                 return_type=Numeric)
.....:
```

```
In [16]: from featuretools.tests.testing_utils import make_ecommerce_entityset
```

```
In [17]: es = make_ecommerce_entityset()
```

Since WordCount is a transform primitive, we need to add it to the list of transform primitives DFS can use when generating features.

```
In [18]: feature_matrix, features = ft.dfs(entityset=es,
.....:                                   target_entity="sessions",
.....:                                   agg_primitives=["sum", "mean", "std"],
.....:                                   trans_primitives=[WordCount])
.....:
```

```
In [19]: feature_matrix[["customers.WORD_COUNT(favorite_quote)", "STD(log.WORD_
↳COUNT(comments))", "SUM(log.WORD_COUNT(comments))", "MEAN(log.WORD_COUNT(comments)
↳"]]
```

Out[19]:

	customers.WORD_COUNT(favorite_quote)	STD(log.WORD_COUNT(comments))	SUM(log.WORD_
	↳COUNT(comments))	MEAN(log.WORD_COUNT(comments))	
id			
0		9	540.436860
↳	2500		500
1		9	583.702550
↳	1732		433
2		9	NaN
↳	246		246
3		6	883.883476
↳	1256		628
4		6	0.000000
↳	9		3
5		12	19.798990
↳	68		34

By adding some aggregation primitives as well, Deep Feature Synthesis was able to make four new features from one new primitive.

Multiple Input Types

If a primitive requires multiple features as input, `input_types` has multiple elements, eg `[Numeric, Numeric]` would mean the primitive requires two Numeric features as input. Below is an example of a primitive that has multiple input features.

```
In [20]: from featuretools.variable_types import Datetime, Timedelta, Variable
```

```
In [21]: import pandas as pd
```

```
In [22]: def mean_sunday(numeric, datetime):
.....:     '''
.....:     Finds the mean of non-null values of a feature that occurred on Sundays
```

(continues on next page)

(continued from previous page)

```

.....:     '''
.....:     days = pd.DatetimeIndex(datetime).weekday.values
.....:     df = pd.DataFrame({'numeric': numeric, 'time': days})
.....:     return df[df['time'] == 6]['numeric'].mean()
.....:
In [23]: MeanSunday = make_agg_primitive(function=mean_sunday,
.....:                                     input_types=[Numeric, Datetime],
.....:                                     return_type=Numeric)
.....:
In [24]: feature_matrix, features = ft.dfs(entityset=es,
.....:                                     target_entity="sessions",
.....:                                     agg_primitives=[MeanSunday],
.....:                                     trans_primitives=[],
.....:                                     max_depth=1)
.....:
In [25]: feature_matrix[["MEAN_SUNDAY(log.value, datetime)", "MEAN_SUNDAY(log.value_2,
↪ datetime)"]]
Out [25]:
MEAN_SUNDAY(log.value, datetime)  MEAN_SUNDAY(log.value_2, datetime)
id
0                                NaN                                NaN
1                                NaN                                NaN
2                                NaN                                NaN
3                                2.5                                1.0
4                                7.0                                3.0
5                                NaN                                NaN

```

3.5 Handling Time

When performing feature engineering with temporal data, carefully selecting the data that is used for any calculation is paramount. By annotating *entities* with a **time index** column and providing a **cutoff time** during feature calculation, Featuretools will automatically filter out any data after the cutoff time before running any calculations.

3.5.1 What is the Time Index?

The time index is the column in the data that specifies when the data in each row became known. For example, let's examine a table of customer transactions:

```

In [1]: import featuretools as ft

In [2]: es = ft.demo.load_mock_customer(return_entityset=True, random_seed=0)

In [3]: es['transactions'].df.head()
Out [3]:
  transaction_id  session_id  transaction_time  amount  product_id
298             298         1 2014-01-01 00:00:00  127.64         5
2              2         1 2014-01-01 00:01:05   109.48         2
308            308         1 2014-01-01 00:02:10    95.06         3
116            116         1 2014-01-01 00:03:15    78.92         4
371            371         1 2014-01-01 00:04:20    31.54         3

```

In this table, there is one row for every transaction and a `transaction_time` column that specifies when the transaction took place. This means that `transaction_time` is the time index because it indicates when the information in each row became known and available for feature calculations.

However, not every datetime column is a time index. Consider the `customers` entity:

```
In [4]: es['customers'].df
Out [4]:
   customer_id  join_date  date_of_birth  zip_code
5            5 2010-07-17 05:27:50  1984-07-28  60091
4            4 2011-04-08 20:08:14  2006-08-15  60091
1            1 2011-04-17 10:48:33  1994-07-18  60091
3            3 2011-08-13 15:42:34  2003-11-21  13244
2            2 2012-04-15 23:31:04  1986-08-18  13244
```

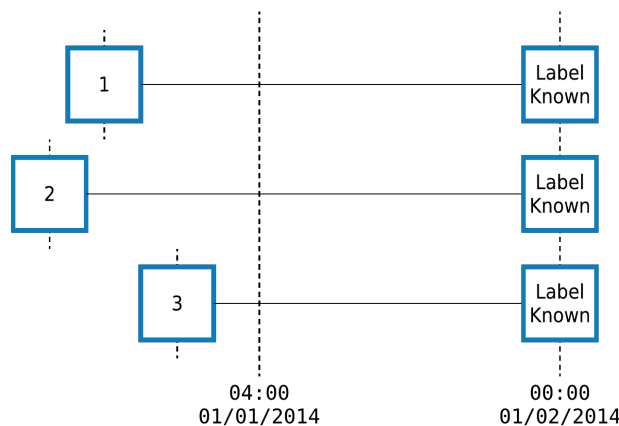
Here, we have two time columns, `join_date` and `date_of_birth`. While either column might be useful for making features, the `join_date` should be used as the time index because it indicates when that customer first became available in the dataset.

Important: The **time index** is defined as the first time that any information from a row can be used. If a cutoff time is specified when calculating features, rows that have a later value for the time index are automatically ignored.

3.5.2 What is the Cutoff Time?

The **cutoff_time** specifies the last point in time that a row’s data can be used for a feature calculation. Any data after this point in time will be filtered out before calculating features.

For example, let’s consider a dataset of timestamped customer transactions, where we want to predict whether customers 1, 2 and 3 will spend \$500 between 04:00 on January 1 and the end of the day. When building features for this prediction problem, we need to ensure that no data after 04:00 is used in our calculations.



We pass the cutoff time to `featuretools.dfs()` or `featuretools.calculate_feature_matrix()` using the `cutoff_time` argument like this:

```
In [5]: fm, features = ft.dfs(entityset=es,
...:                        target_entity='customers',
...:                        cutoff_time=pd.Timestamp("2014-1-1 04:00"),
...:                        instance_ids=[1,2,3],
...:                        )
```

(continues on next page)

(continued from previous page)

1	2014-01-01 04:00:00	60091	4				
↪3	tablet	4958.19			42.309717		
↪	139.23	-0.006928			5.81		
↪	74.002836	67			5		
↪	4	18	17		1994		
↪	2011	7	4		0		
↪	6	540.04			-0.		
↪505043		27.62					
↪	20	304.601700					
↪169.572874		5.027226			271.		
↪917637		0.500353			1.		
↪285833			0.0				
↪	10.426572	5.678908					
↪1613.93		0.234349					
↪8.74			5				
↪	85.469167	25			46.		
↪905665		-0.451371			1.		
↪197406		1.452325					
↪	0.0		-0.233453				
↪	1.614843	1.235445					
↪129.00		1025.63			-0.		
↪830975			5				
↪	64.557200		12		39.		
↪825249		135.0100			1239.		
↪5475		-0.126261			6.		
↪905			5				
↪	76.150425	16.75			42.		
↪393218		1					
↪	1		1				
↪	1			3			
↪	2014		2				
↪	1	1					
↪	4		1				
↪	3				tablet		
2	2014-01-01 04:00:00	13244	4				
↪2	desktop	4150.30			39.289512		
↪	146.81	-0.134786			12.07		
↪	84.700000	49			5		
↪	4	18	15		1986		
↪	2012	8	4		0		
↪	6	569.29			0.		
↪045171		105.24					
↪	20	340.791792					
↪157.262738		3.470527			307.		
↪743859		0.324809			20.		
↪424007			0.0				
↪	8.983533	3.862210					
↪1320.64		0.295458					
↪56.46			5				
↪	96.581000	16			47.		
↪935920		0.459305			-0.		
↪823347		1.815491					
↪	0.0		0.651941				
↪-0.169238		-0.966834					
↪138.38		634.84			-0.		
↪455197			5				
↪	76.813125		8		27.		
↪839228		142.3225					
↪5750		0.011293			26.		
↪310			5				

(continues on next page)

3.5. Handling Time

↪	85.19948	12.25			39.		
↪	315685	1					
↪	1		1				
↪	1			3			

(continued from previous page)

3	2014-01-01 04:00:00	13244		1					
→1	tablet		941.87			47.264797			↵
→	146.31		0.618455			8.19			↵
→	62.791333		15			5			↵
→	1		21		13	2003			↵
→	2011	11		8		4			↵
→	5			146.31				0.	↵
→618455				8.19					↵
→	5			62.791333					↵
→	47.264797			NaN					↵
→	NaN			NaN					↵
→NaN				NaN					↵
→	NaN			NaN				941.	↵
→87			0.618455					8.19	↵
→				5					↵
→62.791333			15			47.			↵
→264797			NaN						↵
→	NaN		NaN						↵
→	NaN			NaN					↵
→	NaN			NaN					↵
→146.31			941.87					0.	↵
→618455					5				↵
→	62.791333			15				47.	↵
→264797			146.3100					941.	↵
→8700			0.618455					8.	↵
→190					5				↵
→	62.791333			15.00				47.	↵
→264797				1					↵
→	1			1					↵
→	1				1				↵
→	2014				2				↵
→	1		1						↵
→	1			1					↵
→	1			3				tablet	↵

Even though the entityset contains the complete transaction history for each customer, only data with a time index up to and including the cutoff time was used to calculate the features above.

Using a Cutoff Time DataFrame

Oftentimes, the training examples for machine learning will come from different points in time. To specify a unique cutoff time for each row of the resulting feature matrix, we can pass a dataframe where the first column is the instance id and the second column is the corresponding cutoff time.

Note: Only the first two columns are used to calculate features. Any additional columns passed through are appended to the resulting feature matrix. This is typically used to pass through machine learning labels to ensure that they stay aligned with the feature matrix.

```
In [7]: cutoff_times = pd.DataFrame()
In [8]: cutoff_times['customer_id'] = [1, 2, 3, 1]
In [9]: cutoff_times['time'] = pd.to_datetime(['2014-1-1 04:00',
```

(continues on next page)

(continued from previous page)

```

...:         '2014-1-1 05:00',
...:         '2014-1-1 06:00',
...:         '2014-1-1 08:00'])
...:

```

In [10]: cutoff_times['label'] = [True, True, False, True]

In [11]: cutoff_times

Out [11]:

	customer_id	time	label
0	1	2014-01-01 04:00:00	True
1	2	2014-01-01 05:00:00	True
2	3	2014-01-01 06:00:00	False
3	1	2014-01-01 08:00:00	True

In [12]: fm, features = ft.dfs(entityset=es,

```

.....:         target_entity='customers',
.....:         cutoff_time=cutoff_times,
.....:         cutoff_time_in_index=True)
.....:

```

In [13]: fm

Out [13]:

```

zip_code COUNT(sessions) NUM_UNIQUE(sessions.
↳device) MODE(sessions.device) SUM(transactions.amount) STD(transactions.amount)
↳MAX(transactions.amount) SKEW(transactions.amount) MIN(transactions.amount)
↳MEAN(transactions.amount) COUNT(transactions) NUM_UNIQUE(transactions.product_id)
↳ MODE(transactions.product_id) DAY(date_of_birth) DAY(join_date) YEAR(date_of_
↳birth) YEAR(join_date) MONTH(date_of_birth) MONTH(join_date) WEEKDAY(date_of_
↳birth) WEEKDAY(join_date) SUM(sessions.MAX(transactions.amount)) SUM(sessions.
↳SKEW(transactions.amount)) SUM(sessions.MIN(transactions.amount)) SUM(sessions.
↳NUM_UNIQUE(transactions.product_id)) SUM(sessions.MEAN(transactions.amount))
↳SUM(sessions.STD(transactions.amount)) STD(sessions.MAX(transactions.amount))
↳STD(sessions.SUM(transactions.amount)) STD(sessions.SKEW(transactions.amount))
↳STD(sessions.MIN(transactions.amount)) STD(sessions.NUM_UNIQUE(transactions.
↳product_id)) STD(sessions.MEAN(transactions.amount)) STD(sessions.
↳COUNT(transactions)) MAX(sessions.SUM(transactions.amount)) MAX(sessions.
↳SKEW(transactions.amount)) MAX(sessions.MIN(transactions.amount)) MAX(sessions.
↳NUM_UNIQUE(transactions.product_id)) MAX(sessions.MEAN(transactions.amount))
↳MAX(sessions.COUNT(transactions)) MAX(sessions.STD(transactions.amount))
↳SKEW(sessions.MAX(transactions.amount)) SKEW(sessions.SUM(transactions.amount))
↳SKEW(sessions.MIN(transactions.amount)) SKEW(sessions.NUM_UNIQUE(transactions.
↳product_id)) SKEW(sessions.MEAN(transactions.amount)) SKEW(sessions.
↳COUNT(transactions)) SKEW(sessions.STD(transactions.amount)) MIN(sessions.
↳MAX(transactions.amount)) MIN(sessions.SUM(transactions.amount)) MIN(sessions.
↳SKEW(transactions.amount)) MIN(sessions.NUM_UNIQUE(transactions.product_id))
↳MIN(sessions.MEAN(transactions.amount)) MIN(sessions.COUNT(transactions))
↳MIN(sessions.STD(transactions.amount)) MEAN(sessions.MAX(transactions.amount))
↳MEAN(sessions.SUM(transactions.amount)) MEAN(sessions.SKEW(transactions.amount))
↳MEAN(sessions.MIN(transactions.amount)) MEAN(sessions.NUM_UNIQUE(transactions.
↳product_id)) MEAN(sessions.MEAN(transactions.amount)) MEAN(sessions.
↳COUNT(transactions)) MEAN(sessions.STD(transactions.amount)) NUM_UNIQUE(sessions.
↳YEAR(session_start)) NUM_UNIQUE(sessions.WEEKDAY(session_start)) NUM_
↳UNIQUE(sessions.DAY(session_start)) NUM_UNIQUE(sessions.MONTH(session_start)) NUM_
↳UNIQUE(sessions.MODE(transactions.product_id)) MODE(sessions.YEAR(session_start))
↳MODE(sessions.WEEKDAY(session_start)) MODE(sessions.DAY(session_start))
↳MODE(sessions.MONTH(session_start)) MODE(sessions.MODE(transactions.product_id))
↳NUM_UNIQUE(transactions.sessions.customer_id) NUM_UNIQUE(transactions.sessions.
↳device) MODE(transactions.sessions.customer_id) MODE(transactions.sessions.device)
↳label

```

(continues on next page)

(continued from previous page)

2	2014-01-01 05:00:00	13244	5			
↪2	desktop	5155.26		38.047944		
↪	146.81	-0.121811		12.07		
↪	83.149355	62		5		
↪	4	18	15	1986		
↪	2012	8	4	0		
↪	6	688.14				-0.
↪269747		127.06				
↪	25	418.096407				
↪190.987775		10.919023				266.
↪912832		0.316873				17.
↪801322			0.0			
↪	8.543351	3.361547				
↪1320.64		0.295458				
↪56.46			5			
↪	96.581000		16			47.
↪935920		-1.814717				-0.
↪667256		1.959531				
↪	0.0		1.082192			
↪-0.379092		-0.213518				
↪118.85		634.84				-0.
↪455197			5			
↪	76.813125		8			27.
↪839228		137.62800				1031.
↪0520		-0.053949				25.
↪41200			5			
↪	83.619281		12.40			
↪38.197555			1			
↪	1		1			
↪	1			4		
↪	2014		2			
↪	1	1				
↪	2		1			
↪	2		2			
↪desktop	True					
3	2014-01-01 06:00:00	13244	4			
↪2	desktop	2867.69		40.349758		
↪	146.31	0.318315		6.65		
↪	65.174773	44		5		
↪	1	21	13	2003		
↪	2011	11	8	4		
↪	5	493.07				0.
↪860577		126.66				
↪	16	290.968018				
↪119.136697		22.808351				417.
↪557763		0.500999				40.
↪508892			2.0			
↪	16.540737	7.118052				
↪944.85		0.618455				
↪91.76			5			
↪	91.760000		17			47.
↪264797		-1.060639				-1.
↪977878		1.874170				
↪	-2.0		0.201588			
↪-1.330938		1.722323				
↪	91.76	91.76				-0.
↪289466			1			
↪	55.579412		1			
↪704680		123.26750				716.
↪9225		0.286859				31.
↪66500			4			
↪	72.742004		11.00			
↪39.712232			1			
↪	1		1			

(continues on next page)

3.5. Handling Time

(continued from previous page)

1	2014-01-01 08:00:00	60091	8					
↪3	mobile		9025.62		40.442059			↪
↪	139.43		0.019698		5.81			↪
↪	71.631905	126			5			↪
↪	4	18		17	1994			↪
↪	2011	7		4	0			↪
↪	6		1057.97					↪ -0.
↪476122			78.59					↪
↪	40		582.193117					↪
↪312.745952			7.322191					↪ 279.
↪510713			0.589386					↪ 6.
↪954507				0.0				↪
↪	13.759314		4.062019					↪
↪1613.93			0.640252					↪
↪26.36				5				↪
↪	88.755625		25					↪ 46.
↪905665			-0.780493					↪ 0.
↪778170			2.440005					↪
↪	0.0			-0.424949				↪
↪	1.946018			-0.312355				↪
↪118.90			809.97					↪ -1.
↪038434				5				↪
↪	50.623125		12					↪ 30.
↪450261			132.24625					↪ 1128.
↪2025			-0.059515					↪ 9.
↪82375				5				↪
↪	72.774140		15.75					↪
↪39.093244			1					↪
↪	1			1				↪
↪	1				4			↪
↪	2014			2				↪
↪	1		1					↪
↪	4			1				↪
↪	3			1				↪
↪mobile	True							↪

We can now see that every row of the feature matrix is calculated at the corresponding time in the cutoff time dataframe. Because we calculate each row at a different time, it is possible to have a repeat customer. In this case, we calculated the feature vector for customer 1 at both 04:00 and 08:00.

3.5.3 Training Window

By default, all data up to and including the cutoff time is used. We can restrict the amount of historical data that is selected for calculations using a “training window.”

Here’s an example of using a two hour training window:

```
In [14]: window_fm, window_features = ft.dfs(entityset=es,
.....:                                     target_entity="customers",
.....:                                     cutoff_time=cutoff_times,
.....:                                     cutoff_time_in_index=True,
.....:                                     training_window="2 hour")
.....:

In [15]: window_fm
```

(continues on next page)

(continued from previous page)

1	2014-01-01 04:00:00	60091	2		
→2	desktop		2077.66	43.772157	
→	139.09		-0.187686	5.81	
→	76.950370	27		5	
→	4	18	17	1994	
→	2011	7	4	0	
→	6		271.81	-0.	
→604638			12.59		
→	10		155.604500		
→ 86.730914			4.504270	18.	
→667619			0.747633	0.	
→685894			0.000000		
→ 10.842658			2.121320		
→1052.03			0.226337		
→6.78				5	
→ 85.469167			15	46.	
→905665			NaN		
→ NaN			NaN		
→	NaN		NaN		
→	NaN		NaN		
→132.72			1025.63	-0.	
→830975				5	
→	70.135333		12	39.	
→825249			135.905000	1038.	
→830000			-0.302319	6.	
→295000			5.000000		
→	77.802250		13.500000		
→43.365457			1		
→	1		1		
→	1			2	
→	2014		2		
→	1		1		
→	1		1		
→	2		1		
→desktop	True				
2	2014-01-01 05:00:00	13244	3		
→2	desktop		2605.61	36.077146	
→	146.81		-0.198611	12.07	
→	84.051935	31		5	
→	4	18	15	1986	
→	2012	8	4	0	
→	6		404.04	-0.	
→110009			90.35		
→	15		253.240615		
→109.500185			14.342521	203.	
→331699			0.242542	23.	
→329038			0.000000		
→	10.587085		2.516611		
→1004.96			0.130019		
→56.46				5	
→	96.581000		13	47.	
→935920			-1.083626	-1.	
→660092			1.397956		
→	0.000000		1.659252		
→ 0.585583			1.121470		
→118.85			634.84	-0.	
→314918				5	
→	77.304615		8		
→839228			134.680000	868.	
→536667			-0.036670	30.	
40	116667		5.000000		
→	84.413538		10.333333		
→36.500062			1		
→	1		1		

(continues on next page)

(continued from previous page)

3	2014-01-01 06:00:00	13244	3		
→1	desktop	1925.82		37.130891	┌
→	128.26	0.110145		6.65	┌
→	66.407586	29		5	┌
→	1	21	13	2003	┌
→	2011	11	8	4	┌
→	5	346.76		0.	┌
→242122		118.47			┌
→	11	228.176684			┌
→ 71.871900		20.648490		477.	┌
→281339		0.580573		45.	┌
→761028		2.309401			┌
→	18.557570	8.082904			┌
→944.85		0.531588			┌
→91.76			5		┌
→	91.760000	17		36.	┌
→167220		-1.721498		-1.	┌
→705607		1.566223			┌
→	-1.732051	-1.081879			┌
→-0.722109		NaN			┌
→ 91.76		91.76		-0.	┌
→289466			1		┌
→	55.579412		1	35.	┌
→704680		115.586667		641.	┌
→940000		0.121061		39.	┌
→490000			3.666667		┌
→	76.058895	9.666667			┌
→35.935950			1		┌
→	1		1		┌
→	1			2	┌
→	2014		2		┌
→	1	1			┌
→	1		1		┌
→	1		3		┌
→desktop	False				┌
1	2014-01-01 08:00:00	60091	3		
→2	mobile	3124.15		38.952172	┌
→	139.43	0.047120		5.91	┌
→	66.471277	47		5	┌
→	4	18	17	1994	┌
→	2011	7	4	0	┌
→	6	384.44		-0.	┌
→003438		24.61			┌
→	15	198.984750			┌
→107.128899		10.415432		330.	┌
→655558		0.906666		3.	┌
→016195		0.000000			┌
→	19.935229	0.577350			┌
→1420.09		0.640252			┌
→11.62			5		┌
→	88.755625	16		44.	┌
→354104		0.846298		1.	┌
→606791		1.443486			┌
→	0.000000	1.344879			┌
→-1.732051		1.612576			┌
→118.90		809.97		-1.	┌
→038434			5		┌
→	50.623125		15		┌

(continues on next page)

3.5. Handling Time

→450261		128.146667		1041.	┌
→383333		-0.001146		8.	┌
→203553		5.000000			┌
→	66.328250	15.666667			┌
→35.709633		1			┌
→	1		1		┌

We can see that that the counts for the same feature are lower after we shorten the training window:

```
In [16]: fm[["COUNT(transactions)"]]
Out[16]:
```

customer_id	time	COUNT(transactions)
1	2014-01-01 04:00:00	67
2	2014-01-01 05:00:00	62
3	2014-01-01 06:00:00	44
1	2014-01-01 08:00:00	126

```
In [17]: window_fm[["COUNT(transactions)"]]
////////////////////////////////////////////////////////////////////////////////
↔
```

customer_id	time	COUNT(transactions)
1	2014-01-01 04:00:00	27
2	2014-01-01 05:00:00	31
3	2014-01-01 06:00:00	29
1	2014-01-01 08:00:00	47

3.5.4 Setting a Last Time Index

The training window in Featuretools limits the amount of past data that can be used while calculating a particular feature vector. A row in the entity is filtered out if the value of its time index is either before or after the training window. This works for entities where a row occurs at a single point in time. However, a row can sometimes exist for a duration.

For example, a customer’s session has multiple transactions which can happen at different points in time. If we are trying to count the number of sessions a user has in a given time period, we often want to count all the sessions that had *any* transaction during the training window. To accomplish this, we need to not only know when a session starts, but also when it ends. The last time that an instance appears in the data is stored as the `last_time_index` of an *Entity*. We can compare the time index and the last time index of the `sessions` entity above:

```
In [18]: es['sessions'].df['session_start'].head()
Out[18]:
```

1	2014-01-01 00:00:00
2	2014-01-01 00:17:20
3	2014-01-01 00:28:10
4	2014-01-01 00:44:25
5	2014-01-01 01:11:30

Name: session_start, dtype: datetime64[ns]

```
In [19]: es['sessions'].last_time_index.head()
////////////////////////////////////////////////////////////////////////////////
↔
```

1	2014-01-01 00:16:15
2	2014-01-01 00:27:05
3	2014-01-01 00:43:20
4	2014-01-01 01:10:25
5	2014-01-01 01:22:20

Name: last_time, dtype: datetime64[ns]

Featuretools can automatically add last time indexes to every *Entity* in an *Entityset* by running `EntitySet.add_last_time_indexes()`. If a `last_time_index` has been set, Featuretools will check to see if the `last_time_index` is after the start of the training window. That, combined with the cutoff time, allows DFS to discover which data is relevant for a given training window.

3.5.5 Approximating Features by Rounding Cutoff Times

For each unique cutoff time, Featuretools must perform operations to select the data that's valid for computations. If there are a large number of unique cutoff times relative to the number of instances for which we are calculating features, the time spent filtering data can add up. By reducing the number of unique cutoff times, we minimize the overhead from searching for and extracting data for feature calculations.

One way to decrease the number of unique cutoff times is to round cutoff times to an earlier point in time. An earlier cutoff time is always valid for predictive modeling — it just means we're not using some of the data we could potentially use while calculating that feature. So, we gain computational speed by losing a small amount of information.

To understand when an approximation is useful, consider calculating features for a model to predict fraudulent credit card transactions. In this case, an important feature might be, “the average transaction amount for this card in the past”. While this value can change every time there is a new transaction, updating it less frequently might not impact accuracy.

Note: The bank BBVA used approximation when building a predictive model for credit card fraud using Featuretools. For more details, see the “Real-time deployment considerations” section of the [white paper](#) describing the work involved.

The frequency of approximation is controlled using the `approximate` parameter to `featuretools.dfs()` or `featuretools.calculate_feature_matrix()`. For example, the following code would approximate aggregation features at 1 day intervals:

```
fm = ft.calculate_feature_matrix(features=features,
                               entityset=es_transactions,
                               cutoff_time=ct_transactions,
                               approximate="1 day")
```

In this computation, features that can be approximated will be calculated at 1 day intervals, while features that cannot be approximated (e.g “what is the destination of this flight?”) will be calculated at the exact cutoff time.

3.5.6 Secondary Time Index

It is sometimes the case that information in a dataset is updated or added after a row has been created. This means that certain columns may actually become known after the time index for a row. Rather than drop those columns to avoid leaking information, we can create a secondary time index to indicate when those columns become known.

The *Flights* entityset is a good example of a dataset where column values in a row become known at different times. Each trip is recorded in the `trip_logs` entity, and has many times associated with it.

```
In [20]: es_flight = ft.demo.load_flight(nrows=100)
Downloading data ...
```

```
In [21]: es_flight
Out[21]:
Entityset: Flight Data
```

(continues on next page)

(continued from previous page)

```

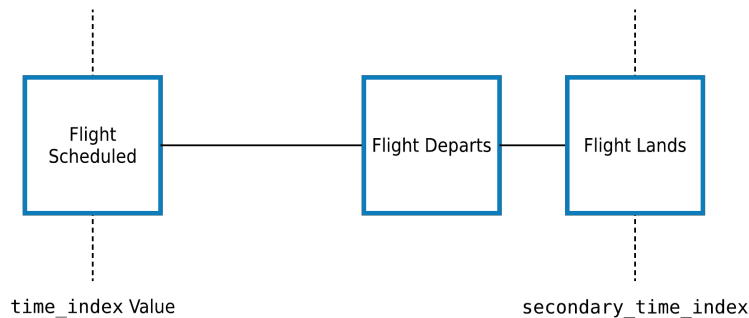
Entities:
  trip_logs [Rows: 100, Columns: 21]
  flights [Rows: 13, Columns: 9]
  airlines [Rows: 1, Columns: 1]
  airports [Rows: 6, Columns: 3]
Relationships:
  trip_logs.flight_id -> flights.flight_id
  flights.carrier -> airlines.carrier
  flights.dest -> airports.dest

In [22]: es_flight['trip_logs'].df.head(3)
//////////////////////////////////////////////////
↳
  trip_log_id      flight_id date_scheduled  scheduled_dep_time  scheduled_arr_
↳time            dep_time            arr_time dep_delay taxi_out taxi_in arr_
↳delay scheduled_elapsed_time  air_time  distance  carrier_delay  weather_delay
↳national_airspace_delay security_delay late_aircraft_delay canceled diverted
30              30 AA-494:RSW->CLT    2016-09-03 2017-01-01 13:14:00 2017-01-01
↳15:05:00 2017-01-01 13:03:00 2017-01-01 14:53:00      -11.0     12.0     10.0
↳-12.0           6660000000000      88.0      600.0          0.0          0.0
↳           0.0          0.0          0.0          0.0          0.0
38              38 AA-495:ATL->PHX    2016-09-03 2017-01-01 11:30:00 2017-01-01
↳15:40:00 2017-01-01 11:24:00 2017-01-01 15:41:00       -6.0     28.0      5.0
↳   1.0           15000000000000      224.0     1587.0          0.0          0.0
↳           0.0          0.0          0.0          0.0          0.0
46              46 AA-495:CLT->ATL    2016-09-03 2017-01-01 09:25:00 2017-01-01
↳10:42:00 2017-01-01 09:23:00 2017-01-01 10:39:00       -2.0     18.0      8.0
↳  -3.0           4620000000000      50.0      226.0          0.0          0.0
↳           0.0          0.0          0.0          0.0          0.0

```

For every trip log, the time index is `date_scheduled`, which is when the airline decided on the scheduled departure and arrival times, as well as what route will be flown. We don't know the rest of the information about the actual departure/arrival times and the details of any delay at this time. However, it is possible to know everything about how a trip went after it has arrived, so we can use that information at any time after the flight lands.

Using a secondary time index, we can indicate to Featuretools which columns in our flight logs are known at the time the flight is scheduled, plus which are known at the time the flight lands.



In Featuretools, when creating the entity, we set the secondary time index to be the arrival time like this:

```

es = ft.EntitySet('Flight Data')
arr_time_columns = ['arr_delay', 'dep_delay', 'carrier_delay', 'weather_delay',
                   'national_airspace_delay', 'security_delay',
                   'late_aircraft_delay', 'canceled', 'diverted',

```

(continues on next page)

(continued from previous page)

```

        'taxi_in', 'taxi_out', 'air_time', 'dep_time']
es.entity_from_dataframe('trip_logs',
                        data,
                        index='trip_log_id',
                        make_index=True,
                        time_index='date_scheduled',
                        secondary_time_index={'arr_time': arr_time_columns})
    
```

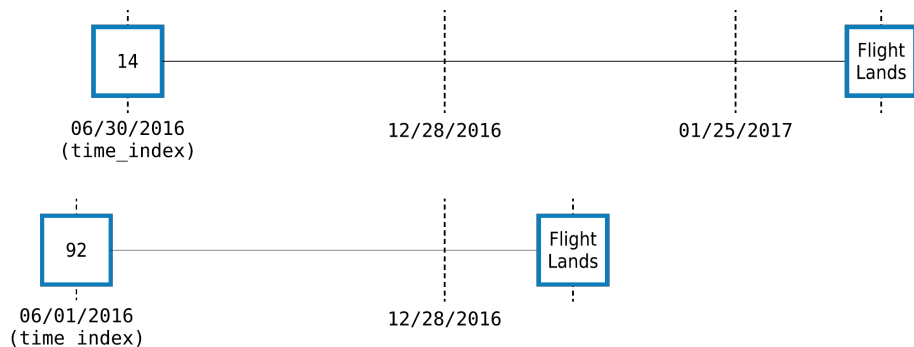
By setting a secondary time index, we can still use the delay information from a row, but only when it becomes known.

Hint: It's often a good idea to use a secondary time index if your entityset has inline labels. If you know when the label would be valid for use, it's possible to automatically create very predictive features using historical labels.

Flight Predictions

Let's make some features at varying times using the flight example described above. Trip 14 is a flight from CLT to PHX on January 31, 2017 and trip 92 is a flight from PIT to DFW on January 1. We can set any cutoff time before the flight is scheduled to depart, emulating how we would make the prediction at that point in time.

We set two cutoff times for trip 14 at two different times: one which is more than a month before the flight and another which is only 5 days before. For trip 92, we'll only set one cutoff time, three days before it is scheduled to leave.



Our cutoff time dataframe looks like this:

```

In [23]: ct_flight = pd.DataFrame()
In [24]: ct_flight['trip_log_id'] = [14, 14, 92]
In [25]: ct_flight['time'] = pd.to_datetime(['2016-12-28',
.....:                                     '2017-1-25',
.....:                                     '2016-12-28'])
.....:
In [26]: ct_flight['label'] = [True, True, False]
In [27]: ct_flight
Out[27]:
trip_log_id    time    label
    
```

(continues on next page)

- `start` (`datetime.datetime` or `pd.Timestamp`): The first cutoff time in the created time series.
- `num_windows` (`int`): The number of cutoff times to create in the created time series.

Only two of the three options `window_size`, `start`, and `num_windows` need to be specified to uniquely determine an equally-spaced set of cutoff times at which to compute each instance.

If your cutoff times are the ones used above:

```
In [30]: cutoff_times
Out [30]:
```

	customer_id	time	label
0	1	2014-01-01 04:00:00	True
1	2	2014-01-01 05:00:00	True
2	3	2014-01-01 06:00:00	False
3	1	2014-01-01 08:00:00	True

Then passing in `window_size='1h'` and `num_windows=2` makes one row an hour over the last two hours to produce the following new dataframe. The result can be directly passed into DFS to make features at the different time points.

```
In [31]: temporal_cutoffs = ft.make_temporal_cutoffs(cutoff_times['customer_id'],
.....:                                             cutoff_times['time'],
.....:                                             window_size='1h',
.....:                                             num_windows=2)
.....:

In [32]: temporal_cutoffs
Out [32]:
```

	time	instance_id
0	2014-01-01 03:00:00	1
1	2014-01-01 04:00:00	1
2	2014-01-01 04:00:00	2
3	2014-01-01 05:00:00	2
4	2014-01-01 05:00:00	3
5	2014-01-01 06:00:00	3
6	2014-01-01 07:00:00	1
7	2014-01-01 08:00:00	1

```
In [33]: fm, features = ft.dfs(entityset=es,
.....:                         target_entity='customers',
.....:                         cutoff_time=temporal_cutoffs,
.....:                         cutoff_time_in_index=True)
.....:

In [34]: fm
Out [34]:
```

	zip_code	COUNT(sessions)	NUM_UNIQUE(sessions.
↪device)	MODE(sessions.device)	SUM(transactions.amount)	STD(transactions.amount)
↪MAX(transactions.amount)	SKEW(transactions.amount)	MIN(transactions.amount)	
↪MEAN(transactions.amount)	COUNT(transactions)	NUM_UNIQUE(transactions.product_id)	
↪MODE(transactions.product_id)	DAY(date_of_birth)	DAY(join_date)	YEAR(date_of_
↪birth)	YEAR(join_date)	MONTH(date_of_birth)	MONTH(join_date)
↪WEEKDAY(date_of_	WEEKDAY(join_date)	SUM(sessions.MAX(transactions.amount))	SUM(sessions.
↪SKEW(transactions.amount))	SUM(sessions.MIN(transactions.amount))	SUM(sessions.	NUM_UNIQUE(transactions.product_id))
↪SUM(sessions.STD(transactions.amount))	STD(sessions.MAX(transactions.amount))		
↪STD(sessions.SUM(transactions.amount))	STD(sessions.SKEW(transactions.amount))		
↪STD(sessions.MIN(transactions.amount))	STD(sessions.NUM_UNIQUE(transactions.		
↪product_id))	STD(sessions.MEAN(transactions.amount))	STD(sessions.	(continues on next page)
↪COUNT(transactions)	MAX(sessions.SUM(transactions.amount))	MAX(sessions.	
↪SKEW(transactions.amount))	MAX(sessions.MIN(transactions.amount))	MAX(sessions.	
↪SUM(transactions.product_id))	MAX(sessions.MEAN(transactions.amount))		
↪MAX(sessions.COUNT(transactions))	MAX(sessions.STD(transactions.amount))		
↪SKEW(sessions.MAX(transactions.amount))	SKEW(sessions.SUM(transactions.amount))		
↪SKEW(sessions.MIN(transactions.amount))	SKEW(sessions.NUM_UNIQUE(transactions.		
↪product_id))	SKEW(sessions.MEAN(transactions.amount))	SKEW(sessions.	

(continued from previous page)

	2014-01-01 04:00:00	60091		4				
→3	tablet		4958.19			42.309717		↵
→	139.23		-0.006928			5.81		↵
→	74.002836	67				5		↵
→		4	18	17		1994		↵
→	2011	7		4		0		↵
→	6		540.04				-0.	↵
→505043			27.62					↵
→	20		304.601700					↵
→169.572874			5.027226				271.	↵
→917637			0.500353				1.	↵
→285833				0.0				↵
→	10.426572		5.678908					↵
→1613.93			0.234349					↵
→8.74				5				↵
→	85.469167		25				46.	↵
→905665			-0.451371				1.	↵
→197406			1.452325					↵
→	0.0			-0.233453				↵
→	1.614843		1.235445					↵
→129.00			1025.63				-0.	↵
→830975				5				↵
→	64.557200			12			39.	↵
→825249			135.010000				1239.	↵
→547500			-0.126261				6.	↵
→905000				5				↵
→	76.150425		16.750000					↵
→42.393218				1				↵
→	1			1				↵
→	1				3			↵
→	2014			2				↵
→	1		1					↵
→	4			1				↵
→	3			1				↵
→tablet								↵
2	2014-01-01 04:00:00	13244		4				
→2	desktop		4150.30			39.289512		↵
→	146.81		-0.134786			12.07		↵
→	84.700000	49				5		↵
→		4	18	15		1986		↵
→	2012	8		4		0		↵
→	6		569.29				0.	↵
→045171			105.24					↵
→	20		340.791792					↵
→157.262738			3.470527				307.	↵
→743859			0.324809				20.	↵
→424007				0.0				↵
→	8.983533		3.862210					↵
→1320.64			0.295458					↵
→56.46				5				↵
→	96.581000			16			47.	↵
→935920			0.459305				-0.	↵
→823347			1.815491					↵
→	0.0			0.651941				↵
→-0.169238			-0.966834					↵
→138.38			634.84				-0.	↵
→455197				5				↵
→	76.813125			8				↵

(continues on next page)

3.5. Handling Time

→575000			0.011293				26.	
→510000				5				↵
→	85.197948		12.250000					↵
→39.315685				1				↵
→	1			1				↵

(continued from previous page)

	2014-01-01 05:00:00	13244		5			
↪2	desktop		5155.26			38.047944	↪
↪	146.81		-0.121811			12.07	↪
↪	83.149355	62				5	↪
↪		4	18	15		1986	↪
↪	2012			4		0	↪
↪	6	8					↪
↪269747			688.14				-0.↪
↪			127.06				↪
↪	25		418.096407				↪
↪190.987775			10.919023				266.↪
↪912832			0.316873				17.↪
↪801322				0.0			↪
↪	8.543351		3.361547				↪
↪1320.64			0.295458				↪
↪56.46				5			↪
↪	96.581000			16			47.↪
↪935920			-1.814717				-0.↪
↪667256			1.959531				↪
↪	0.0			1.082192			↪
↪-0.379092			-0.213518				↪
↪118.85			634.84				-0.↪
↪455197				5			↪
↪	76.813125			8			27.↪
↪839228			137.628000				1031.↪
↪052000			-0.053949				25.↪
↪412000				5			↪
↪	83.619281		12.400000				↪
↪38.197555				1			↪
↪	1			1			↪
↪	1				4		↪
↪	2014			2			↪
↪	1		1				↪
↪	2			1			↪
↪	2			2			↪
↪desktop							↪
3	2014-01-01 05:00:00	13244		2			↪
↪2	desktop		1886.72			41.199361	↪
↪	146.31		0.637074			6.65	↪
↪	58.960000	32				5	↪
↪		1	21	13		2003	↪
↪	2011			8		4	↪
↪	5	11					1.↪
↪150043			273.05				↪
↪			14.84				↪
↪	10		118.370745				↪
↪	83.432017		13.838080				2.↪
↪107178			0.061424				1.↪
↪088944				0.0			↪
↪	5.099599		1.414214				↪
↪944.85			0.618455				↪
↪8.19				5			↪
↪	62.791333			17			47.↪
↪264797			NaN				↪
↪	NaN		NaN				↪
↪	NaN			NaN			↪
↪	NaN		NaN				↪
↪126.74			941.87				0.↪
↪531588				5			↪

(continues on next page)

↪	55.579412			15			↪
↪167220			136.525000				943.↪
↪360000			0.575022				7.↪
50	420000			5			↪
↪	59.185373		16.000000				↪
↪41.716008				1			↪
↪	1			1			↪

(continued from previous page)

	2014-01-01 06:00:00	13244		4			
↪2	desktop		2867.69			40.349758	↪
↪	146.31		0.318315			6.65	↪
↪	65.174773	44				5	↪
↪		1	21	13		2003	↪
↪	2011	11		8		4	↪
↪	5		493.07				↪
↪860577			126.66				↪
↪	16		290.968018				↪
↪119.136697			22.808351				↪417.
↪557763			0.500999				↪40.
↪508892				2.0			↪
↪	16.540737		7.118052				↪
↪944.85			0.618455				↪
↪91.76				5			↪
↪	91.760000			17			↪47.
↪264797			-1.060639				↪-1.
↪977878			1.874170				↪
↪	-2.0			0.201588			↪
↪-1.330938			1.722323				↪
↪	91.76		91.76				↪-0.
↪289466				1			↪
↪	55.579412			1			↪35.
↪704680			123.267500				↪716.
↪922500			0.286859				↪31.
↪665000				4			↪
↪	72.742004		11.000000				↪
↪39.712232				1			↪
↪	1			1			↪
↪	1				2		↪
↪	2014			2			↪
↪	1		1				↪
↪	1			1			↪
↪	2			3			↪
↪desktop							↪
1	2014-01-01 07:00:00	60091		7			↪
↪3	tablet		7605.53			41.018896	↪
↪	139.43		0.149908			5.81	↪
↪	69.141182	110				5	↪
↪		4	18	17		1994	↪
↪	2011	7		4		0	↪
↪	6		931.86				↪0.
↪562312			66.97				↪
↪	35		493.437492				↪
↪280.421418			7.441648				↪273.
↪713405			0.471955				↪7.
↪470707				0.0			↪
↪	13.123365		4.386125				↪
↪1613.93			0.640252				↪
↪26.36				5			↪
↪	85.469167			25			↪46.
↪905665			-1.277394				↪1.
↪377768			2.552328				↪
↪	0.0			-0.282093			↪
↪	1.927658		-0.755846				↪
↪118.90			809.97				↪-0.
↪830975				5			↪
↪	50.623125			12			↪

(continues on next page)

3.5. Handling Time

↪450261			133.122857			1086.	↪
↪504286			0.080330			9.	↪
↪561143				5			↪
↪	70.491070		15.714286				↪
↪40.060203				1			↪
↪	1			1			↪

(continued from previous page)

	2014-01-01 08:00:00	60091		8			
↪3	mobile		9025.62			40.442059	↪
↪	139.43		0.019698			5.81	↪
↪	71.631905		126			5	↪
↪		4	18		17	1994	↪
↪	2011		7		4	0	↪
↪	6					1057.97	↪
↪476122						78.59	↪
↪	40					582.193117	↪
↪312.745952						7.322191	↪
↪510713						0.589386	↪
↪954507					0.0		↪
↪	13.759314					4.062019	↪
↪1613.93						0.640252	↪
↪26.36						5	↪
↪	88.755625				25		↪
↪905665						-0.780493	↪
↪778170						2.440005	↪
↪	0.0					-0.424949	↪
↪	1.946018					-0.312355	↪
↪118.90						809.97	↪
↪038434							↪
↪	50.623125						↪
↪					12		↪
↪450261						132.246250	↪
↪202500						-0.059515	↪
↪823750							↪
↪	72.774140					15.750000	↪
↪39.093244						1	↪
↪	1					1	↪
↪	1						↪
↪	2014						↪
↪					2		↪
↪	1					1	↪
↪	4					1	↪
↪						1	↪
↪mobile							↪

3.6 Tuning Deep Feature Synthesis

There are several parameters that can be tuned to change the output of DFS.

```
In [1]: import featuretools as ft

In [2]: es = ft.demo.load_mock_customer(return_entityset=True)

In [3]: es
Out[3]:
Entityset: transactions
Entities:
  transactions [Rows: 500, Columns: 5]
  products [Rows: 5, Columns: 2]
  sessions [Rows: 35, Columns: 4]
  customers [Rows: 5, Columns: 4]
Relationships:
  transactions.product_id -> products.product_id
```

(continues on next page)

(continued from previous page)

```
transactions.session_id -> sessions.session_id
sessions.customer_id -> customers.customer_id
```

3.6.1 Using “Seed Features”

Seed features are manually defined, problem specific, features a user provides to DFS. Deep Feature Synthesis will then automatically stack new features on top of these features when it can.

By using seed features, we can include domain specific knowledge in feature engineering automation.

```
In [4]: expensive_purchase = ft.Feature(es["transactions"]["amount"]) > 125

In [5]: feature_matrix, feature_defs = ft.dfs(entityset=es,
...:                                       target_entity="customers",
...:                                       agg_primitives=["percent_true"],
...:                                       seed_features=[expensive_purchase])
...:

In [6]: feature_matrix[['PERCENT_TRUE(transactions.amount > 125)']]
Out [6]:
          PERCENT_TRUE(transactions.amount > 125)
customer_id
5                0.227848
4                0.220183
1                0.119048
3                0.182796
2                0.129032
```

We can now see that PERCENT_TRUE was automatically applied to this boolean variable.

3.6.2 Add “interesting” values to variables

Sometimes we want to create features that are conditioned on a second value before we calculate. We call this extra filter a “where clause”.

By default, where clauses are built using the interesting_values of a variable.

```
In [7]: es["sessions"]["device"].interesting_values = ["desktop", "mobile", "tablet"]
```

We then specify the aggregation primitive to make where clauses for using where_primitives

```
In [8]: feature_matrix, feature_defs = ft.dfs(entityset=es,
...:                                       target_entity="customers",
...:                                       agg_primitives=["count", "avg_time_
↪between"],
...:                                       where_primitives=["count", "avg_time_
↪between"],
...:                                       trans_primitives=[])
...:
...:
```

```
In [9]: feature_matrix
```

```
Out [9]:
          zip_code  COUNT(sessions)  AVG_TIME_BETWEEN(sessions.session_start)
↪COUNT(transactions)  AVG_TIME_BETWEEN(transactions.transaction_time)
↪COUNT(sessions WHERE device = mobile)  COUNT(sessions WHERE device = tablet)
↪COUNT(sessions WHERE device = desktop)  AVG_TIME_BETWEEN(sessions.session_start
↪WHERE device = mobile)  AVG_TIME_BETWEEN(sessions.session_start WHERE device =
tablet)  AVG_TIME_BETWEEN(sessions.session_start WHERE device = desktop)  AVG_TIME
↪BETWEEN(transactions.sessions.session_start)  COUNT(transactions WHERE sessions.
↪device = desktop)  COUNT(transactions WHERE sessions.device = mobile)
↪COUNT(transactions WHERE sessions.device = tablet)  AVG_TIME_BETWEEN(transactions.
↪transaction_time WHERE sessions.device = desktop)  AVG_TIME_BETWEEN(transactions.
```


(continued from previous page)

2	13244	7	4907.500000
↪	93		328.532609
↪	2		2
↪	3		1690.000000
↪		5330.0	
↪	6890.0		320.
↪054348			34
↪		31	28
↪		435.303030	
↪		82.333333	
↪		226.296296	
↪		417.575758	
↪		56.333333	
↪		197.407407	

Now, we have several new potentially useful features. For example, the two features below tell us *how many sessions a customer completed on a tablet*, and *the time between those sessions*.

```
In [10]: feature_matrix[["COUNT(sessions WHERE device = tablet)", "AVG_TIME_
↪BETWEEN(sessions.session_start WHERE device = tablet)"]]
Out [10]:
      COUNT(sessions WHERE device = tablet)  AVG_TIME_BETWEEN(sessions.session_
↪start WHERE device = tablet)
customer_id
5
↪      NaN
4
↪      NaN
1
↪      8807.5
3
↪      NaN
2
↪      5330.0
```

We can see that customer who only had 0 or 1 sessions on a tablet, had NaN values for average time between such sessions.

3.6.3 Encoding categorical features

Machine learning algorithms typically expect all numeric data. When Deep Feature Synthesis generates categorical features, we need to encode them.

```
In [11]: feature_matrix, feature_defs = ft.dfs(entityset=es,
.....:                                     target_entity="customers",
.....:                                     agg_primitives=["mode"],
.....:                                     max_depth=1)
.....:

In [12]: feature_matrix
Out [12]:
      zip_code MODE(sessions.device)  DAY(date_of_birth)  DAY(join_date)
↪YEAR(date_of_birth)  YEAR(join_date)  MONTH(date_of_birth)  MONTH(join_date)
↪WEEKDAY(date_of_birth)  WEEKDAY(join_date)
```

(continues on next page)

The returned feature matrix is now all numeric. Additionally, we get a new set of feature definitions that contain the encoded values.

```
In [15]: print(features_enc)
[<Feature: zip_code = 60091>, <Feature: zip_code = 13244>, <Feature: zip_code is_
↳unknown>, <Feature: MODE(sessions.device) = mobile>, <Feature: MODE(sessions.
↳device) = desktop>, <Feature: MODE(sessions.device) is unknown>, <Feature: DAY(date_
↳of_birth) = 18>, <Feature: DAY(date_of_birth) = 28>, <Feature: DAY(date_of_birth) =_
↳21>, <Feature: DAY(date_of_birth) = 15>, <Feature: DAY(date_of_birth) is unknown>,
↳<Feature: DAY(join_date) = 17>, <Feature: DAY(join_date) = 15>, <Feature: DAY(join_
↳date) = 13>, <Feature: DAY(join_date) = 8>, <Feature: DAY(join_date) is unknown>,
↳<Feature: YEAR(date_of_birth) = 2006>, <Feature: YEAR(date_of_birth) = 2003>,
↳<Feature: YEAR(date_of_birth) = 1994>, <Feature: YEAR(date_of_birth) = 1986>,
↳<Feature: YEAR(date_of_birth) = 1984>, <Feature: YEAR(date_of_birth) is unknown>,
↳<Feature: YEAR(join_date) = 2011>, <Feature: YEAR(join_date) = 2012>, <Feature:_
↳YEAR(join_date) = 2010>, <Feature: YEAR(join_date) is unknown>, <Feature:_
↳MONTH(date_of_birth) = 8>, <Feature: MONTH(date_of_birth) = 7>, <Feature:_
↳MONTH(date_of_birth) is unknown>, <Feature: MONTH(date_of_birth) is unknown>, <Feature:_
↳MONTH(join_date) = 4>, <Feature: MONTH(join_date) = 8>, <Feature: MONTH(join_date)_
↳= 7>, <Feature: MONTH(join_date) is unknown>, <Feature: WEEKDAY(date_of_birth) = 0>,
↳ <Feature: WEEKDAY(date_of_birth) = 5>, <Feature: WEEKDAY(date_of_birth) = 4>,
↳<Feature: WEEKDAY(date_of_birth) = 1>, <Feature: WEEKDAY(date_of_birth) is unknown>,
↳ <Feature: WEEKDAY(join_date) = 6>, <Feature: WEEKDAY(join_date) = 5>, <Feature:_
↳WEEKDAY(join_date) = 4>, <Feature: WEEKDAY(join_date) is unknown>]
```

These features can be used to calculate the same encoded values on new data. For more information on feature engineering in production, read [Deployment](#).

3.7 Specifying Primitive Options

By default, DFS will apply primitives across all entities and columns. This behavior can be altered through a few different parameters. Entities and variables can be optionally ignored or included for an entire DFS run or on a per-primitive basis, enabling greater control over features and less run time overhead.

```
In [1]: from featuretools.tests.testing_utils import make_ecommerce_entityset

In [2]: es = make_ecommerce_entityset()

In [3]: feature_matrix, features_list = ft.dfs(entityset=es,
...:                                         target_entity='customers',
...:                                         agg_primitives=['mode'],
...:                                         trans_primitives=['weekday'])
...:

In [4]: features_list
Out[4]:
[<Feature: cohort>,
 <Feature: age>,
 <Feature: région_id>,
 <Feature: loves_ice_cream>,
 <Feature: cancel_reason>,
 <Feature: engagement_level>]
```

(continues on next page)

(continued from previous page)

```

<Feature: MODE (sessions.device_type)>,
<Feature: MODE (sessions.device_name)>,
<Feature: MODE (log.product_id)>,
<Feature: MODE (log.subregioncode)>,
<Feature: MODE (log.countrycode)>,
<Feature: MODE (log.priority_level)>,
<Feature: MODE (log.zipcode)>,
<Feature: WEEKDAY (signup_date)>,
<Feature: WEEKDAY (date_of_birth)>,
<Feature: WEEKDAY (upgrade_date)>,
<Feature: WEEKDAY (cancel_date)>,
<Feature: cohorts.cohort_name>,
<Feature: régions.language>,
<Feature: MODE (sessions.MODE (log.product_id))>,
<Feature: MODE (sessions.MODE (log.zipcode))>,
<Feature: MODE (sessions.MODE (log.priority_level))>,
<Feature: MODE (sessions.MODE (log.countrycode))>,
<Feature: MODE (sessions.MODE (log.subregioncode))>,
<Feature: MODE (log.sessions.device_name)>,
<Feature: MODE (log.sessions.device_type)>,
<Feature: MODE (log.sessions.customer_id)>,
<Feature: cohorts.MODE (customers.cancel_reason)>,
<Feature: cohorts.MODE (customers.région_id)>,
<Feature: cohorts.MODE (customers.engagement_level)>,
<Feature: cohorts.MODE (sessions.device_type)>,
<Feature: cohorts.MODE (sessions.device_name)>,
<Feature: cohorts.MODE (log.product_id)>,
<Feature: cohorts.MODE (log.subregioncode)>,
<Feature: cohorts.MODE (log.countrycode)>,
<Feature: cohorts.MODE (log.priority_level)>,
<Feature: cohorts.MODE (log.zipcode)>,
<Feature: cohorts.WEEKDAY (cohort_end)>,
<Feature: régions.MODE (customers.cancel_reason)>,
<Feature: régions.MODE (customers.cohort)>,
<Feature: régions.MODE (customers.engagement_level)>,
<Feature: régions.MODE (sessions.device_type)>,
<Feature: régions.MODE (sessions.device_name)>,
<Feature: régions.MODE (log.product_id)>,
<Feature: régions.MODE (log.subregioncode)>,
<Feature: régions.MODE (log.countrycode)>,
<Feature: régions.MODE (log.priority_level)>,
<Feature: régions.MODE (log.zipcode)>]

```

3.7.1 Specifying Options for an Entire Run

The `ignore_entities` and `ignore_variables` parameters of DFS control entities and variables (columns) that should be ignored for all primitives. This is useful for ignoring columns or entities that don't relate to the problem or otherwise shouldn't be included in the DFS run.

```

# ignore the 'log' and 'cohorts' entities entirely
# ignore the 'date_of_birth' variable in 'customers' and the 'device_name' variable_
↳ in 'sessions'
In [5]: feature_matrix, features_list = ft.dfs(entityset=es,
...:                                         target_entity='customers',
...:                                         agg_primitives=['mode'],

```

(continues on next page)

(continued from previous page)

```

...: trans_primitives=['weekday'],
...: ignore_entities=['log', 'cohorts'],
...: ignore_variables={
...:     'sessions': ['device_name'],
...:     'customers': ['date_of_birth']})
...:

In [6]: features_list
Out [6]:
[<Feature: cohort>,
 <Feature: age>,
 <Feature: région_id>,
 <Feature: loves_ice_cream>,
 <Feature: cancel_reason>,
 <Feature: engagement_level>,
 <Feature: MODE(sessions.device_type)>,
 <Feature: WEEKDAY(signup_date)>,
 <Feature: WEEKDAY(upgrade_date)>,
 <Feature: WEEKDAY(cancel_date)>,
 <Feature: régions.language>,
 <Feature: régions.MODE(customers.cancel_reason)>,
 <Feature: régions.MODE(customers.cohort)>,
 <Feature: régions.MODE(customers.engagement_level)>,
 <Feature: régions.MODE(sessions.device_type)>]

```

DFS completely ignores the 'log' and 'cohorts' entities when creating features. It also ignores the variables 'device_name' and 'date_of_birth' in 'sessions' and 'customers' respectively. However, both of these options can be overridden by individual primitive options in the `primitive_options` parameter.

3.7.2 Specifying for Individual Primitives

Options for individual primitives or groups of primitives are set by the `primitive_options` parameter of DFS. This parameter maps any desired options to specific primitives. In the case of conflicting options, options set at this level will override options set at the entire DFS run level, and the include options will always take priority over their ignore counterparts.

Specifying Entities for Individual Primitives

Which entities to include/ignore can also be specified for a single primitive or a group of primitives. Entities can be ignored using the `ignore_entities` option in `primitive_options`, while entities to explicitly include are set by the `include_entities` option. When `include_entities` is given, all entities not listed are ignored by the primitive. No variables from any excluded entity will be used to generate features with the given primitive.

```

# ignore the 'cohorts' and 'log' entities, but only for the primitive 'mode'
# include only the 'customers' entity for the primitives 'weekday' and 'day'
In [7]: feature_matrix, features_list = ft.dfs(entityset=es,
...:     target_entity='customers',
...:     agg_primitives=['mode'],
...:     trans_primitives=['weekday', 'day'],
...:     primitive_options={
...:         'mode': {'ignore_entities': [
↪ 'cohorts', 'log']},
...:         ('weekday', 'day'): {'include_
↪ entities': ['customers']}

```

(continues on next page)

(continued from previous page)

```

...:         })
...:
In [8]: features_list
Out [8]:
[<Feature: cohort>,
 <Feature: age>,
 <Feature: région_id>,
 <Feature: loves_ice_cream>,
 <Feature: cancel_reason>,
 <Feature: engagement_level>,
 <Feature: MODE(sessions.device_type)>,
 <Feature: MODE(sessions.device_name)>,
 <Feature: WEEKDAY(signup_date)>,
 <Feature: WEEKDAY(date_of_birth)>,
 <Feature: WEEKDAY(upgrade_date)>,
 <Feature: WEEKDAY(cancel_date)>,
 <Feature: DAY(signup_date)>,
 <Feature: DAY(date_of_birth)>,
 <Feature: DAY(upgrade_date)>,
 <Feature: DAY(cancel_date)>,
 <Feature: cohorts.cohort_name>,
 <Feature: régions.language>,
 <Feature: cohorts.MODE(customers.cancel_reason)>,
 <Feature: cohorts.MODE(customers.région_id)>,
 <Feature: cohorts.MODE(customers.engagement_level)>,
 <Feature: cohorts.MODE(sessions.device_type)>,
 <Feature: cohorts.MODE(sessions.device_name)>,
 <Feature: régions.MODE(customers.cancel_reason)>,
 <Feature: régions.MODE(customers.cohort)>,
 <Feature: régions.MODE(customers.engagement_level)>,
 <Feature: régions.MODE(sessions.device_type)>,
 <Feature: régions.MODE(sessions.device_name)>]

```

In this example, DFS would only use the 'customers' entity for both weekday and day, and would use all entities except 'cohorts' and 'log' for mode.

Specifying Columns for Individual Primitives

Specific variables (columns) can also be explicitly included/ignored for a primitive or group of primitives. Variables to ignore is set by the `ignore_variables` option, while variables to include is set by `include_variables`. When the `include_variables` option is set, no other variables from that entity will be used to make features with the given primitive.

```

# Include the variables 'product_id' and 'zipcode', 'device_type', and 'cancel_reason
↪' for 'mean'
# Ignore the variables 'signup_date' and 'cancel_date' for 'weekday'
In [9]: feature_matrix, features_list = ft.dfs(entityset=es,
...:         target_entity='customers',
...:         agg_primitives=['mode'],
...:         trans_primitives=['weekday'],
...:         primitive_options={
...:             'mode': {'include_variables': {'log
↪': ['product_id', 'zipcode'],
...:
↪'sessions': ['device_type'],

```

(continues on next page)

(continued from previous page)

```

.....:
↪ 'customers': ['cancel_reason']}},
.....:                               'weekday': {'ignore_variables': {
↪ 'customers':
.....:
↪   ['signup_date',
.....:
↪   'cancel_date']}}}})
.....:

```

In [10]: features_list

Out [10]:

```

[<Feature: cohort>,
 <Feature: age>,
 <Feature: région_id>,
 <Feature: loves_ice_cream>,
 <Feature: cancel_reason>,
 <Feature: engagement_level>,
 <Feature: MODE(sessions.device_type)>,
 <Feature: MODE(log.product_id)>,
 <Feature: MODE(log.zipcode)>,
 <Feature: WEEKDAY(date_of_birth)>,
 <Feature: WEEKDAY(upgrade_date)>,
 <Feature: cohorts.cohort_name>,
 <Feature: régions.language>,
 <Feature: MODE(sessions.MODE(log.product_id))>,
 <Feature: MODE(sessions.MODE(log.zipcode))>,
 <Feature: MODE(log.sessions.device_type)>,
 <Feature: cohorts.MODE(customers.cancel_reason)>,
 <Feature: cohorts.MODE(sessions.device_type)>,
 <Feature: cohorts.MODE(log.product_id)>,
 <Feature: cohorts.MODE(log.zipcode)>,
 <Feature: cohorts.WEEKDAY(cohort_end)>,
 <Feature: régions.MODE(customers.cancel_reason)>,
 <Feature: régions.MODE(sessions.device_type)>,
 <Feature: régions.MODE(log.product_id)>,
 <Feature: régions.MODE(log.zipcode)>]

```

Here, mode will only use the variables 'product_id' and 'zipcode' from the entity 'log', 'device_type' from the entity 'sessions', and 'cancel_reason' from 'customers'. For any other entity, mode will use all variables. The weekday primitive will use all variables in all entities except for 'signup_date' and 'cancel_date' from the 'customers' entity.

Specifying GroupBy Options

GroupBy Transform Primitives also have the additional options `include_groupby_entities`, `ignore_groupby_entities`, `include_groupby_variables`, and `ignore_groupby_variables`. These options are used to specify entities and columns to include/ignore as groupings for inputs. By default, DFS only groups by ID columns. Specifying `include_groupby_variables` overrides this default, and will only group by variables given. On the other hand, `ignore_groupby_variables` will continue to use only the ID columns, ignoring any variables specified that are also ID columns. Note that if including non-ID columns to group by, the included columns must also be a discrete type.

```

In [11]: feature_matrix, features_list = ft.dfs(entityset=es,
.....:                                     target_entity='log',

```

(continues on next page)

customer_id) by product_id> is still made. The groupby is from the target entity log, so the feature is valid given the associated options. To ignore the sessions entity for cum_count, the ignore_entities option for cum_count would need to include sessions.

3.7.3 Specifying for each Input for Multiple Input Primitives

For primitives that take multiple columns as input, such as Trend, the above options can be specified for each input by passing them in as a list. If only one option dictionary is given, it is used for all inputs. The length of the list provided must match the number of inputs the primitive takes.

```
In [13]: feature_matrix, features_list = ft.dfs(entityset=es,
.....:                                     target_entity='customers',
.....:                                     agg_primitives=['trend'],
.....:                                     trans_primitives=[],
.....:                                     primitive_options={
.....:                                         'trend': [{'ignore_variables': {
↳ 'log': ['value_many_nans']}},
.....:                                         {'include_variables':
↳ {'customers': ['signup_date']},
.....:                                         'log': ['datetime']}}}))
.....:

In [14]: features_list
Out[14]:
[<Feature: cohort>,
 <Feature: age>,
 <Feature: région_id>,
 <Feature: loves_ice_cream>,
 <Feature: cancel_reason>,
 <Feature: engagement_level>,
 <Feature: TREND(log.value, datetime)>,
 <Feature: TREND(log.value_2, datetime)>,
 <Feature: cohorts.cohort_name>,
 <Feature: régions.language>,
 <Feature: cohorts.TREND(customers.age, signup_date)>,
 <Feature: cohorts.TREND(log.value, datetime)>,
 <Feature: cohorts.TREND(log.value_2, datetime)>,
 <Feature: régions.TREND(customers.age, signup_date)>,
 <Feature: régions.TREND(log.value, datetime)>,
 <Feature: régions.TREND(log.value_2, datetime)>]
```

Here, we pass in a list of primitive options for trend. We ignore the variable 'value_many_nans' for the first input to trend, and include the variables 'signup_date' from 'customers' for the second input.

3.8 Improving Computational Performance

Feature engineering is a computationally expensive task. While Featuretools comes with reasonable default settings for feature calculation, there are a number of built-in approaches to improve computational performance based on dataset and problem specific considerations.

3.8.1 Reduce number of unique cutoff times

Each row in a feature matrix created by Featuretools is calculated at a specific cutoff time that represents the last point in time that data from any entity in an entity set can be used to calculate the feature. As a result, calculations incur an overhead in finding the subset of allowed data for each distinct time in the calculation.

Note: Featuretools is very precise in how it deals with time. For more information, see [Handling Time](#).

If there are many unique cutoff times, it is often worthwhile to figure out how to have fewer. This can be done manually by figuring out which unique times are necessary for the prediction problem or automatically using [approximate](#).

3.8.2 Adjust chunk size

By default, Featuretools calculates rows with the same cutoff time simultaneously. The `chunk_size` parameter limits the maximum number of rows that will be grouped and then calculated together. If calculation is done using parallel processing, the default chunk size is set to be $1 / n_jobs$ to ensure the computation can be spread across available workers. Normally, this behavior works well, but if there are only a few unique cutoff times it can lead to higher peak memory usage (due to more intermediate calculations stored in memory) or limited parallelism (if the number of chunks is less than n_jobs).

By setting `chunk_size`, we can limit the maximum number of rows in each group to specific number or a percentage of the overall data when calling `ft.dfs` or `ft.calculate_feature_matrix`:

```
# use maximum 100 rows per chunk
feature_matrix, features_list = ft.dfs(entityset=es,
                                     target_entity="customers",
                                     chunk_size=100)
```

We can also set chunk size to be a percentage of total rows:

```
# use maximum 5% of all rows per chunk
feature_matrix, features_list = ft.dfs(entityset=es,
                                     target_entity="customers",
                                     chunk_size=.05)
```

3.8.3 Partition and Distribute Data

When an entire dataset is not required to calculate the features for a given set of instances, we can split the data into independent partitions and calculate on each partition. For example, imagine we are calculating features for customers and the features are “number of other customers in this zip code” or “average age of other customers in this zip code”. In this case, we can load in data partitioned by zip code. As long as we have all of the data for a zip code when calculating, we can calculate all features for a subset of customers.

An example of this approach can be seen in the [Predict Next Purchase demo notebook](#). In this example, we partition data by customer and only load a fixed number of customers into memory at any given time. We implement this easily using [Dask](#), which could also be used to scale the computation to a cluster of computers. A framework like [Spark](#) could be used similarly.

An additional example of partitioning data to distribute on multiple cores or a cluster using [Dask](#) can be seen in the [Featuretools on Dask notebook](#). This approach is detailed in the [Parallelizing Feature Engineering with Dask](#) article on the Feature Labs engineering blog. [Dask](#) allows for simple scaling to multiple cores on a single computer or multiple machines on a cluster.

For a similar partition and distribute implementation using Apache Spark with PySpark, refer to the [Feature Engineering on Spark notebook](#). This implementation shows how to carry out feature engineering on a cluster of EC2 instances using Spark as the distributed framework. A write-up of this approach is described in the [Featuretools on Spark article](#) on the Feature Labs engineering blog.

3.8.4 Featuretools Enterprise

If you don't want to build it yourself, Featuretools Enterprise has native integrations with Apache Spark and Dask. More information is available [here](#).

If you would like to test [Featuretools Enterprise APIs](#) for running Featuretools natively on Apache Spark or Dask, please let us know [here](#).

3.9 Parallel Feature Computation

Featuretools can optionally compute features on multiple cores. The simplest way to control the amount of parallelism is to specify the `n_jobs` parameter:

```
fm = ft.calculate_feature_matrix(features=features,
                                entityset=entityset,
                                cutoff_time=cutoff_time,
                                n_jobs=2,
                                verbose=True)
```

The above command will start 2 processes to compute chunks of the feature matrix in parallel. Each process receives its own copy of the entity set, so memory use will be proportional to the number of parallel processes. Because the entity set has to be copied to each process, there is overhead to perform this operation before calculation can begin. To avoid this overhead on successive calls to `calculate_feature_matrix`, read the section below on using a persistent cluster.

3.9.1 Using persistent cluster

Behind the scenes, Featuretools uses `dask's` distributed scheduler to implement multiprocessing. When you only specify the `n_jobs` parameter, a cluster will be created for that specific feature matrix calculation and destroyed once calculations have finished. A drawback of this is that each time a feature matrix is calculated, the entity set has to be transmitted to the workers again. To avoid this, we would like to reuse the same cluster between calls. The way to do this is by creating a cluster first and telling featuretools to use it with the `dask_kwargs` parameter:

```
import featuretools as ft
from dask.distributed import LocalCluster

cluster = LocalCluster()
fm_1 = ft.calculate_feature_matrix(features=features_1,
                                  entityset=entityset,
                                  cutoff_time=cutoff_time,
                                  dask_kwargs={'cluster': cluster},
                                  verbose=True)
```

The 'cluster' value can either be the actual cluster object or a string of the address the cluster's scheduler can be reached at. The call below would also work. This second feature matrix calculation will not need to resend the entityset data to the workers because it has already been saved on the cluster.:

```
fm_2 = ft.calculate_feature_matrix(features=features_2,
                                  entityset=entityset,
                                  cutoff_time=cutoff_time,
                                  dask_kwargs={'cluster': cluster.scheduler.address},
                                  verbose=True)
```

Note: When using a persistent cluster, Featuretools publishes a copy of the `EntitySet` to the cluster the first time it calculates a feature matrix. Based on the `EntitySet`'s metadata the cluster will reuse it for successive computations. This means if two `EntitySets` have the same metadata but different row values (e.g. new data is added to the `EntitySet`), Featuretools won't recopy the second `EntitySet` in later calls. A simple way to avoid this scenario is to use a unique `EntitySet` id.

3.9.2 Using the distributed dashboard

Dask.distributed has a web-based diagnostics dashboard that can be used to analyze the state of the workers and tasks. It can also be useful for tracking memory use or visualizing task run-times. An in-depth description of the web interface can be found [here](#).



The dashboard requires an additional python package, bokeh, to work. Once bokeh is installed, the web interface will be launched by default when a LocalCluster is created. The cluster created by featuretools when using `n_jobs` does not enable the web interface automatically. To do so, the port to launch the main web interface on must be specified in `dask_kwargs`:

```
fm = ft.calculate_feature_matrix(features=features,
                                entityset=entityset,
                                cutoff_time=cutoff_time,
                                n_jobs=2,
                                dask_kwargs={'diagnostics_port': 8787}
                                verbose=True)
```

3.9.3 Parallel Computation by Partitioning Data

As an alternative to Featuretool’s parallelization, the data can be partitioned and the feature calculations run on multiple cores or a cluster using Dask or Apache Spark with PySpark. This approach may be necessary with a large `EntitySet` because the current parallel implementation sends the entire `EntitySet` to each worker which may exhaust the worker memory. For more information on partitioning the data and using Dask or Spark, see [Improving](#)

Computational Performance. Dask and Spark allow Featuretools to scale to multiple cores on a single machine or multiple machines on a cluster.

3.10 Deployment

Deployment of machine learning models requires repeating feature engineering steps on new data. In some cases, these steps need to be performed in near real-time. Featuretools has capabilities to ease the deployment of feature engineering.

3.10.1 Saving Features

First, let's build some generate some training and test data in the same format. We use a random seed to generate different data for the test.

Note: Features saved in one version of Featuretools are not guaranteed to load in another. This means the features might need to be re-created after upgrading Featuretools.

```
In [1]: import featuretools as ft

In [2]: es_train = ft.demo.load_mock_customer(return_entityset=True)

In [3]: es_test = ft.demo.load_mock_customer(return_entityset=True, random_seed=33)
```

Now let's build some features definitions using DFS. Because we have categorical features, we also encode them with one hot encoding based on the values in the training data.

```
In [4]: feature_matrix, feature_defs = ft.dfs(entityset=es_train,
      ...:                                     target_entity="customers")
      ...:

In [5]: feature_matrix_enc, features_enc = ft.encode_features(feature_matrix, feature_
      ↪_defs)

In [6]: feature_matrix_enc
Out [6]:
      zip_code = 60091  zip_code = 13244  zip_code is unknown  COUNT(sessions)
↪ NUM_UNIQUE(sessions.device)  MODE(sessions.device) = mobile  MODE(sessions.device)
↪ = desktop  MODE(sessions.device) is unknown  SUM(transactions.amount)
↪ STD(transactions.amount)  MAX(transactions.amount)  SKEW(transactions.amount)
↪ MIN(transactions.amount)  MEAN(transactions.amount)  COUNT(transactions)  NUM_
↪ UNIQUE(transactions.product_id)  MODE(transactions.product_id) = 4
↪ MODE(transactions.product_id) = 5  MODE(transactions.product_id) = 2
↪ MODE(transactions.product_id) = 1  MODE(transactions.product_id) is unknown
↪ DAY(date_of_birth) = 18  DAY(date_of_birth) = 28  DAY(date_of_birth) = 21  DAY(date_
↪ of_birth) = 15  DAY(date_of_birth) is unknown  DAY(join_date) = 17  DAY(join_date)
↪ = 15  DAY(join_date) = 13  DAY(join_date) = 8  DAY(join_date) is unknown  YEAR(date_
↪ of_birth) = 2006  YEAR(date_of_birth) = 2003  YEAR(date_of_birth) = 1994  YEAR(date_
↪ of_birth) = 1986  YEAR(date_of_birth) = 1984  YEAR(date_of_birth) is unknown
↪ YEAR(join_date) = 2011  YEAR(join_date) = 2012  YEAR(join_date) = 2010  YEAR(join_
↪ date) is unknown  MONTH(date_of_birth) = 8  MONTH(date_of_birth) = 7  MONTH(date_of_
↪ birth) = 11  MONTH(date_of_birth) is unknown  MONTH(join_date) = 4  MONTH(join_
↪ date) = 8  MONTH(join_date) = 7  MONTH(join_date) is unknown  WEEKDAY(date_of_
↪ birth) = 0  WEEKDAY(date_of_birth) = 5  WEEKDAY(date_of_birth) = 4  WEEKDAY(date_of
↪ birth) = 1  WEEKDAY(date_of_birth) is unknown  WEEKDAY(join_date) = 6  WEEKDAY(join_
↪ date) = 5  WEEKDAY(join_date) = 4  WEEKDAY(join_date) is unknown  SUM(sessions.
      ↪ MAX(transactions.amount))  SUM(sessions.SKEW(transactions.amount))  SUM(sessions.
      ↪ MIN(transactions.amount))  SUM(sessions.NUM_UNIQUE(transactions.product_id))
      ↪ SUM(sessions.MEAN(transactions.amount))  SUM(sessions.STD(transactions.amount))
      ↪ STD(sessions.MAX(transactions.amount))  STD(sessions.SUM(transactions.amount))
      ↪ STD(sessions.SKEW(transactions.amount))  STD(sessions.MIN(transactions.amount))
```

(continues on next page)

(continued from previous page)

Now, we can use `featuretools.save_features()` to save a list features to a json file

```
In [7]: ft.save_features(features_enc, "feature_definitions.json")
```

3.10.2 Calculating Feature Matrix for New Data

We can use `featuretools.load_features()` to read in a list of saved features to calculate for our new entity set.

```
In [8]: saved_features = ft.load_features('feature_definitions.json')
```

After we load the features back in, we can calculate the feature matrix.

```
In [9]: feature_matrix = ft.calculate_feature_matrix(saved_features, es_test)
```

```
In [10]: feature_matrix
```

```
Out [10]:
```

```

zip_code = 60091 zip_code = 13244 zip_code is unknown COUNT(sessions)
↳ NUM_UNIQUE(sessions.device) MODE(sessions.device) = mobile MODE(sessions.device)
↳ = desktop MODE(sessions.device) is unknown SUM(transactions.amount)
↳ STD(transactions.amount) MAX(transactions.amount) SKEW(transactions.amount)
↳ MIN(transactions.amount) MEAN(transactions.amount) COUNT(transactions) NUM_
↳ UNIQUE(transactions.product_id) MODE(transactions.product_id) = 4
↳ MODE(transactions.product_id) = 5 MODE(transactions.product_id) = 2
↳ MODE(transactions.product_id) = 1 MODE(transactions.product_id) is unknown
↳ DAY(date_of_birth) = 18 DAY(date_of_birth) = 28 DAY(date_of_birth) = 21 DAY(date_
↳ of_birth) = 15 DAY(date_of_birth) is unknown DAY(join_date) = 17 DAY(join_date)
↳ = 15 DAY(join_date) = 13 DAY(join_date) = 8 DAY(join_date) is unknown YEAR(date_
↳ of_birth) = 2006 YEAR(date_of_birth) = 2003 YEAR(date_of_birth) = 1994 YEAR(date_
↳ of_birth) = 1986 YEAR(date_of_birth) = 1984 YEAR(date_of_birth) is unknown
↳ YEAR(join_date) = 2011 YEAR(join_date) = 2012 YEAR(join_date) = 2010 YEAR(join_
↳ date) is unknown MONTH(date_of_birth) = 8 MONTH(date_of_birth) = 7 MONTH(date_of_
↳ birth) = 11 MONTH(date_of_birth) is unknown MONTH(join_date) = 4 MONTH(join_
↳ date) = 8 MONTH(join_date) = 7 MONTH(join_date) is unknown WEEKDAY(date_of_
↳ birth) = 0 WEEKDAY(date_of_birth) = 5 WEEKDAY(date_of_birth) = 4 WEEKDAY(date_of_
↳ birth) = 1 WEEKDAY(date_of_birth) is unknown WEEKDAY(join_date) = 6 WEEKDAY(join_
↳ date) = 5 WEEKDAY(join_date) = 4 WEEKDAY(join_date) is unknown SUM(sessions.
↳ MAX(transactions.amount)) SUM(sessions.SKEW(transactions.amount)) SUM(sessions.
↳ MIN(transactions.amount)) SUM(sessions.NUM_UNIQUE(transactions.product_id))
↳ SUM(sessions.MEAN(transactions.amount)) SUM(sessions.STD(transactions.amount))
↳ STD(sessions.MAX(transactions.amount)) STD(sessions.SUM(transactions.amount))
↳ STD(sessions.SKEW(transactions.amount)) STD(sessions.MIN(transactions.amount))
↳ STD(sessions.NUM_UNIQUE(transactions.product_id)) STD(sessions.MEAN(transactions.
↳ amount)) STD(sessions.COUNT(transactions)) MAX(sessions.SUM(transactions.amount))
↳ MAX(sessions.SKEW(transactions.amount)) MAX(sessions.MIN(transactions.amount))
↳ MAX(sessions.NUM_UNIQUE(transactions.product_id)) MAX(sessions.MEAN(transactions.
↳ amount)) MAX(sessions.COUNT(transactions)) MAX(sessions.STD(transactions.amount))
↳ SKEW(sessions.MAX(transactions.amount)) SKEW(sessions.SUM(transactions.amount))
↳ SKEW(sessions.MIN(transactions.amount)) SKEW(sessions.NUM_UNIQUE(transactions.
↳ product_id)) SKEW(sessions.MEAN(transactions.amount)) SKEW(sessions.
↳ COUNT(transactions)) SKEW(sessions.STD(transactions.amount)) MIN(sessions.
↳ MAX(transactions.amount)) MIN(sessions.SUM(transactions.amount)) MIN(sessions.
↳ SKEW(transactions.amount)) MIN(sessions.NUM_UNIQUE(transactions.product_id))
↳ MIN(sessions.MEAN(transactions.amount)) MIN(sessions.COUNT(transactions))
↳ MIN(sessions.STD(transactions.amount)) MEAN(sessions.MAX(transactions.amount))
↳ MEAN(sessions.SUM(transactions.amount)) MEAN(sessions.SKEW(transactions.amount))
↳ MEAN(sessions.MIN(transactions.amount)) MEAN(sessions.NUM_UNIQUE(transactions.
↳ product_id)) MEAN(sessions.MEAN(transactions.amount)) MEAN(sessions.
↳ COUNT(transactions)) MEAN(sessions.STD(transactions.amount)) NUM_UNIQUE(sessions.
↳ YEAR(session_start)) NUM_UNIQUE(sessions.WEEKDAY(session_start)) NUM_
↳ UNIQUE(sessions.DAY(session_start)) NUM_UNIQUE(sessions.MONTH(session_start)) NUM_

```

(continues on next page)

(continued from previous page)

As you can see above, we have the exact same features as before, but calculated using the test data.

3.10.3 Exporting Feature Matrix

Save as csv

The feature matrix is a pandas dataframe that we can save to disk

```
In [11]: feature_matrix.to_csv("feature_matrix.csv")
```

We can also read it back in as follows:

```
In [12]: saved_fm = pd.read_csv("feature_matrix.csv", index_col="customer_id")
```

```
In [13]: saved_fm
```

```
Out [13]:
```

```

zip_code = 60091 zip_code = 13244 zip_code is unknown COUNT(sessions)
↳ NUM_UNIQUE(sessions.device) MODE(sessions.device) = mobile MODE(sessions.device)
↳ = desktop MODE(sessions.device) is unknown SUM(transactions.amount)
↳ STD(transactions.amount) MAX(transactions.amount) SKEW(transactions.amount)
↳ MIN(transactions.amount) MEAN(transactions.amount) COUNT(transactions) NUM_
↳ UNIQUE(transactions.product_id) MODE(transactions.product_id) = 4
↳ MODE(transactions.product_id) = 5 MODE(transactions.product_id) = 2
↳ MODE(transactions.product_id) = 1 MODE(transactions.product_id) is unknown
↳ DAY(date_of_birth) = 18 DAY(date_of_birth) = 28 DAY(date_of_birth) = 21 DAY(date_
↳ of_birth) = 15 DAY(date_of_birth) is unknown DAY(join_date) = 17 DAY(join_date)
↳ = 15 DAY(join_date) = 13 DAY(join_date) = 8 DAY(join_date) is unknown YEAR(date_
↳ of_birth) = 2006 YEAR(date_of_birth) = 2003 YEAR(date_of_birth) = 1994 YEAR(date_
↳ of_birth) = 1986 YEAR(date_of_birth) = 1984 YEAR(date_of_birth) is unknown
↳ YEAR(join_date) = 2011 YEAR(join_date) = 2012 YEAR(join_date) = 2010 YEAR(join_
↳ date) is unknown MONTH(date_of_birth) = 8 MONTH(date_of_birth) = 7 MONTH(date_of_
↳ birth) = 11 MONTH(date_of_birth) is unknown MONTH(join_date) = 4 MONTH(join_
↳ date) = 8 MONTH(join_date) = 7 MONTH(join_date) is unknown WEEKDAY(date_of_
↳ birth) = 0 WEEKDAY(date_of_birth) = 5 WEEKDAY(date_of_birth) = 4 WEEKDAY(date_of_
↳ birth) = 1 WEEKDAY(date_of_birth) is unknown WEEKDAY(join_date) = 6 WEEKDAY(join_
↳ date) = 5 WEEKDAY(join_date) = 4 WEEKDAY(join_date) is unknown SUM(sessions.
↳ MAX(transactions.amount)) SUM(sessions.SKEW(transactions.amount)) SUM(sessions.
↳ MIN(transactions.amount)) SUM(sessions.NUM_UNIQUE(transactions.product_id))
↳ SUM(sessions.MEAN(transactions.amount)) SUM(sessions.STD(transactions.amount))
↳ STD(sessions.MAX(transactions.amount)) STD(sessions.SUM(transactions.amount))
↳ STD(sessions.SKEW(transactions.amount)) STD(sessions.MIN(transactions.amount))
↳ STD(sessions.NUM_UNIQUE(transactions.product_id)) STD(sessions.MEAN(transactions.
↳ amount)) STD(sessions.COUNT(transactions)) MAX(sessions.SUM(transactions.amount))
↳ MAX(sessions.SKEW(transactions.amount)) MAX(sessions.MIN(transactions.amount))
↳ MAX(sessions.NUM_UNIQUE(transactions.product_id)) MAX(sessions.MEAN(transactions.
↳ amount)) MAX(sessions.COUNT(transactions)) MAX(sessions.STD(transactions.amount))
↳ SKEW(sessions.MAX(transactions.amount)) SKEW(sessions.SUM(transactions.amount))
↳ SKEW(sessions.MIN(transactions.amount)) SKEW(sessions.NUM_UNIQUE(transactions.
↳ product_id)) SKEW(sessions.MEAN(transactions.amount)) SKEW(sessions.
↳ COUNT(transactions)) SKEW(sessions.STD(transactions.amount)) MIN(sessions.
↳ MAX(transactions.amount)) MIN(sessions.SUM(transactions.amount)) MIN(sessions.
↳ SKEW(transactions.amount)) MIN(sessions.NUM_UNIQUE(transactions.product_id))
↳ MIN(sessions.MEAN(transactions.amount)) MIN(sessions.COUNT(transactions))
↳ MIN(sessions.STD(transactions.amount)) MEAN(sessions.MAX(transactions.amount))
↳ MEAN(sessions.SUM(transactions.amount)) MEAN(sessions.SKEW(transactions.amount))
↳ MEAN(sessions.MIN(transactions.amount)) MEAN(sessions.NUM_UNIQUE(transactions.
↳ product_id)) MEAN(sessions.MEAN(transactions.amount)) MEAN(sessions.
↳ COUNT(transactions)) MEAN(sessions.STD(transactions.amount)) NUM_UNIQUE(sessions.
↳ YEAR(session_start)) NUM_UNIQUE(sessions.WEEKDAY(session_start)) NUM_
↳ UNIQUE(sessions.DAY(session_start)) NUM_UNIQUE(sessions.MONTH(session_start)) NUM_
↳ UNIQUE(sessions.MODE(transactions.product_id)) MODE(sessions.YEAR(session_start))

```

(continues on next page)

(continued from previous page)

2		False		True		False		8
↪			3		False			↪
↪	True			False		9877.67		↪
↪	39.913352		148.34		0.040395			↪
↪	8.00		76.571085		129			↪
↪	5			False			False	↪
↪		False			True			↪
↪		False		False		False		↪
↪	False		False		True			↪
↪	False	False		False		False		↪
↪	True		False		False			↪
↪	False		False		False			↪
↪	True		False		False			↪
↪	True		False		False			↪
↪	False		False		True			↪
↪	False	False		False		True		↪
↪		False		False			True	↪
↪		False		False	True		True	↪
↪		False		False			False	↪
↪		False		False			True	↪
↪		1100.82				-0.082021		↪
↪		136.01					39	↪
↪		615.714934				315.817331		↪
↪		9.491736				346.152626		↪
↪		0.305345				10.517928		↪
↪		0.353553				10.757169		↪
↪		4.086126				1690.97		↪
↪		0.454842				40.88		↪
↪			5			87.669412		↪
↪		21				43.950396		↪
↪	-0.964539					-0.576079		↪
↪	1.981423					-2.828427		↪
↪		-2.048650				-1.158217		↪
↪	-0.403715					120.06		↪
↪	619.93					-0.522578		↪
↪		4				52.288421		↪
↪		8				33.618728		↪
↪	137.602500					1234.708750		↪
↪	-0.010253					17.001250		↪
↪		4.875000				76.964367		↪
↪	16.125000					39.477166		↪
↪		1				1		↪
↪		1				1		↪
↪			4				True	↪
↪			False				True	↪
↪				False				↪
↪	True			False				↪
↪	True			False				↪
↪		False			False		True	↪
↪			False					↪
↪	False					1		↪
↪		3				False		↪
↪		False				False		↪
↪		True				False		↪
↪			False					↪
↪	False			True				↪
↪	False				True			↪
↪	False		True		False			↪
↪		3			False			↪
↪			False					↪
↪			True					↪
↪				False				↪
↪				True				↪
↪	False				True			↪
↪	False		True		False			↪
↪		3			False			↪
↪			False					↪
↪			True					↪
↪				False				↪
↪				True				↪
↪	False				True			↪
↪	False		True		False			↪
↪		3			False			↪
↪			False					↪
↪			True					↪
↪				False				↪
↪				True				↪
↪	False				True			↪
↪	False		True		False			↪
↪		3			False			↪
↪			False					↪
↪			True					↪
↪				False				↪
↪				True				↪
↪	False				True			↪
↪	False		True		False			↪
↪		3			False			↪
↪			False					↪
↪			True					↪
↪				False				↪
↪				True				↪
↪	False				True			↪
↪	False		True		False			↪
↪		3			False			↪
↪			False					↪
↪			True					↪
↪				False				↪
↪				True				↪
↪	False				True			↪
↪	False		True		False			↪
↪		3			False			↪
↪			False					↪
↪			True					↪
↪				False				↪
↪				True				↪
↪	False				True			↪
↪	False		True		False			↪
↪		3			False			↪
↪			False					↪
↪			True					↪
↪				False				↪
↪				True				↪
↪	False				True			↪
↪	False		True		False			↪
↪		3			False			↪
↪			False					↪
↪			True					↪
↪				False				↪
↪				True				↪
↪	False				True			↪
↪	False		True		False			↪
↪		3			False			↪
↪			False					↪
↪			True					↪
↪				False				↪
↪				True				↪
↪	False				True			↪
↪	False		True		False			↪
↪		3			False			↪
↪			False					↪
↪			True					↪
↪				False				↪
↪				True				↪
↪	False				True			↪
↪	False		True		False			↪
↪		3			False			↪
↪			False					↪
↪			True					↪
↪				False				↪
↪				True				↪
↪	False				True			↪
↪	False		True		False			↪
↪		3			False			↪
↪			False					↪
↪			True					↪
↪				False				↪
↪				True				↪
↪	False				True			↪
↪	False		True		False			↪
↪		3			False			↪
↪			False					↪
↪			True					↪
↪				False				↪
↪				True				↪
↪	False				True			↪
↪	False		True		False			↪
↪		3			False			↪
↪			False					↪
↪			True					↪
↪				False				↪
↪				True				↪
↪	False				True			↪
↪	False		True		False			↪
↪		3			False			↪
↪			False					↪
↪			True					↪
↪				False				↪
↪				True				↪
↪	False				True			↪
↪	False		True		False			↪
↪		3			False			↪
↪			False					↪
↪			True					↪
↪				False				↪
↪				True				↪
↪	False				True			↪
↪	False		True		False			↪
↪		3			False			↪
↪			False					↪
↪			True					↪
↪				False				↪
↪				True				↪
↪	False				True			↪
↪	False		True		False			↪
↪		3			False			↪
↪			False					↪
↪			True					↪
↪				False				↪
↪				True				↪
↪	False				True			↪
↪	False		True		False			↪
↪		3			False			↪
↪			False					↪
↪			True					↪
↪				False				↪
↪				True				↪
↪	False				True			↪
↪	False		True		False			↪
↪		3			False			↪
↪			False					↪
↪			True					↪
↪				False				↪
↪				True				↪
↪	False				True			↪
↪	False		True		False			↪
↪		3			False			↪
↪			False					↪
↪			True					↪
↪				False				↪
↪				True				↪
↪	False				True			↪
↪	False		True		False			↪
↪		3			False			↪
↪			False					↪
↪			True					↪
↪				False				↪
↪				True				↪
↪	False				True			↪
↪	False		True		False			↪
↪		3			False			↪
↪			False					↪
↪			True					↪
↪				False				↪
↪				True				↪
↪	False				True			↪
↪	False		True		False			↪
↪		3			False			↪
↪			False					↪
↪			True					↪
↪				False				↪
↪				True				↪
↪	False				True			↪
↪	False		True		False			↪
↪		3			False			↪
↪			False					↪
↪			True					↪
↪				False</				

(continued from previous page)

```
In [8]: feature_matrix[['STD(log.STRING_COUNT(comments, "the"))', 'SUM(log.STRING_
↳COUNT(comments, "the"))', 'MEAN(log.STRING_COUNT(comments, "the"))']]
////////////////////////////////////
↳
    STD(log.STRING_COUNT(comments, "the"))  SUM(log.STRING_COUNT(comments, "the"))  ↳
↳MEAN(log.STRING_COUNT(comments, "the"))
id
↳
0          47.124304          209  ↳
↳          41.80
1          36.509131          109  ↳
↳          27.25
2          NaN          29  ↳
↳          29.00
3          49.497475          70  ↳
↳          35.00
4          0.000000          0  ↳
↳          0.00
5          1.414214          4  ↳
↳          2.00
```

3.11.2 Features with Multiple Outputs

With the `make_primitive` functions, it is possible to have multiple columns output from a single feature. In order to do that, the output must be formatted as a list of arrays/series where each item in the list corresponds to an output from the primitive. In each of these list items (either arrays or series), there must be one element for each input element.

Take, for example, a primitive called `case_count`. For each given string, this primitive outputs the number of uppercase and the number of lowercase letters. So, this primitive must return a list with 2 elements, one corresponding to the number of lowercase letters and one corresponding to the number of uppercase letters. Each element in the list is a series/array having the same number of elements as the number of input strings. Below you can see this example in action, as well as the proper way to specify multiple outputs in the `make_trans_primitive` function.

```
In [9]: def case_count(array):
...:     '''Return the count of upper case and lower case letters in text'''
...:     upper = np.array([len(re.findall('[A-Z]', i)) for i in array])
...:     lower = np.array([len(re.findall('[a-z]', i)) for i in array])
...:     ret = [upper, lower]
...:     return ret
...:
```

We must use the `num_output_features` attribute to specify the number of outputs when creating the primitive using the `make_trans_primitive` function.

```
In [10]: CaseCount = make_trans_primitive(function=case_count,
...:                                     input_types=[Text],
...:                                     return_type=Numeric,
...:                                     number_output_features=2)
...:

In [11]: es = make_ecommerce_entityset()
```

When we call `dfs` on this entityset, there are 6 instances (one for each of the strings in the dataset) of our two created

features in this feature matrix.

```
In [12]: feature_matrix, features = ft.dfs(entityset=es,
     . . . :                               target_entity="sessions",
     . . . :                               agg_primitives=[],
     . . . :                               trans_primitives=[CaseCount])
     . . . :
```

```
In [13]: feature_matrix.columns
Out [13]: Index(['device_name', 'customer_id', 'device_type', 'customers.cohort',
     ↪ 'customers.age', 'customers.région_id', 'customers.loves_ice_cream', 'customers.
     ↪ cancel_reason', 'customers.engagement_level', 'customers.cohorts.cohort_name',
     ↪ 'customers.régions.language', 'customers.CASE_COUNT(favorite_quote) [0]', 'customers.
     ↪ CASE_COUNT(favorite_quote) [1]'], dtype='object')
```

```
In [14]: feature_matrix[['customers.CASE_COUNT(favorite_quote) [0]', 'customers.CASE_
     ↪ COUNT(favorite_quote) [1]']]
     ↪
     ↪
     ↪ customers.CASE_COUNT(favorite_quote) [0]  customers.CASE_COUNT(favorite_quote) [1]
id
0                1                44
1                1                44
2                1                44
3                1                41
4                1                41
5                1                57
```

3.12 Frequently Asked Questions

Here we are attempting to answer some commonly asked questions that appear on Github, and Stack Overflow.

```
[1]: import featuretools as ft
import pandas as pd
import numpy as np
```

3.12.1 EntitySet

How do I get a list of variable (column) names, and types in an EntitySet?

After you create your EntitySet, you may wish to view the column names. An EntitySet contains multiple Dataframes, one for each entity.

```
[2]: es = ft.demo.load_mock_customer(return_entityset=True)
es
```

```
[2]: Entityset: transactions
Entities:
  transactions [Rows: 500, Columns: 5]
  products [Rows: 5, Columns: 2]
  sessions [Rows: 35, Columns: 4]
  customers [Rows: 5, Columns: 4]
Relationships:
  transactions.product_id -> products.product_id
```

(continues on next page)

(continued from previous page)

```
transactions.session_id -> sessions.session_id
sessions.customer_id -> customers.customer_id
```

If you want view the variables (columns), and types for the “transactions” entity, you can do the following:

```
[3]: es['transactions'].variables
[3]: [<Variable: transaction_id (dtype = index)>,
      <Variable: session_id (dtype = id)>,
      <Variable: transaction_time (dtype: datetime_time_index, format: None)>,
      <Variable: amount (dtype = numeric)>,
      <Variable: product_id (dtype = id)>]
```

If you want to view the underlying Dataframe, you can do the following:

```
[4]: es['transactions'].df.head()
[4]:
```

	transaction_id	session_id	transaction_time	amount	product_id
298	298	1	2014-01-01 00:00:00	127.64	5
2	2	1	2014-01-01 00:01:05	109.48	2
308	308	1	2014-01-01 00:02:10	95.06	3
116	116	1	2014-01-01 00:03:15	78.92	4
371	371	1	2014-01-01 00:04:20	31.54	3

What is the difference between copy_variables and additional_variables?

The function `normalize_entity` creates a new entity and a relationship from unique values of an existing entity. It takes 2 similar arguments:

- `additional_variables` removes variables from the base entity and moves them to the new entity.
- `copy_variables` keeps the given variables in the base entity, but also copies them to the new entity.

```
[5]: data = ft.demo.load_mock_customer()
transactions_df = data["transactions"].merge(data["sessions"]).merge(data["customers"]
↪)
products_df = data["products"]

es = ft.EntitySet(id="customer_data")
es = es.entity_from_dataframe(entity_id="transactions",
                             dataframe=transactions_df,
                             index="transaction_id",
                             time_index="transaction_time")

es = es.entity_from_dataframe(entity_id="products",
                             dataframe=products_df,
                             index="product_id")

new_relationship = ft.Relationship(es["products"]["product_id"], es["transactions"]
↪["product_id"])
es = es.add_relationship(new_relationship)
```

Before we normalize to create a new entity, let’s look at base entity

```
[6]: es['transactions'].df.head()
```



```
[6]:
```

transaction_id	session_id	transaction_time	product_id	amount	\
298	298	1 2014-01-01 00:00:00	5	127.64	
2	2	1 2014-01-01 00:01:05	2	109.48	
308	308	1 2014-01-01 00:02:10	3	95.06	
116	116	1 2014-01-01 00:03:15	4	78.92	
371	371	1 2014-01-01 00:04:20	3	31.54	

customer_id	device	session_start	zip_code	join_date	\
298	2 desktop	2014-01-01	13244	2012-04-15 23:31:04	
2	2 desktop	2014-01-01	13244	2012-04-15 23:31:04	
308	2 desktop	2014-01-01	13244	2012-04-15 23:31:04	
116	2 desktop	2014-01-01	13244	2012-04-15 23:31:04	
371	2 desktop	2014-01-01	13244	2012-04-15 23:31:04	

date_of_birth
298 1986-08-18
2 1986-08-18
308 1986-08-18
116 1986-08-18
371 1986-08-18

Notice the columns `session_id`, `session_start`, `join_date`, `device`, `customer_id`, and `zip_code`.

```
[7]: es = es.normalize_entity(base_entity_id="transactions",
                             new_entity_id="sessions",
                             index="session_id",
                             make_time_index="session_start",
                             additional_variables=["join_date"],
                             copy_variables=["device", "customer_id", "zip_code", "session_
↵start"])
```

Above, we normalized the columns to create a new entity. - For `additional_variables`, the following column `['join_date']` will be removed from the `products` entity, and moved to the new `device` entity.

- For `copy_variables`, the following columns `['device', 'customer_id', 'zip_code', 'session_start']` will be copied from the `products` entity to the new `device` entity.

Let's see this in the actual `EntitySet`.

```
[8]: es['transactions'].df.head()
```

```
[8]:
```

transaction_id	session_id	transaction_time	product_id	amount	\
298	298	1 2014-01-01 00:00:00	5	127.64	
2	2	1 2014-01-01 00:01:05	2	109.48	
308	308	1 2014-01-01 00:02:10	3	95.06	
116	116	1 2014-01-01 00:03:15	4	78.92	
371	371	1 2014-01-01 00:04:20	3	31.54	

customer_id	device	session_start	zip_code	date_of_birth
298	2 desktop	2014-01-01	13244	1986-08-18
2	2 desktop	2014-01-01	13244	1986-08-18
308	2 desktop	2014-01-01	13244	1986-08-18
116	2 desktop	2014-01-01	13244	1986-08-18
371	2 desktop	2014-01-01	13244	1986-08-18

Notice above how `['device', 'customer_id', 'zip_code', 'session_start']` are still in the `transactions_entity`, while `['join_date']` is not. But, they have all been moved to the `sessions` entity, as seen below.

```
[9]: es['sessions'].df.head()
[9]:   session_id      join_date  device  customer_id  zip_code  \
1           1  2012-04-15  23:31:04  desktop         2    13244
2           2  2010-07-17  05:27:50  mobile         5    60091
3           3  2011-04-08  20:08:14  mobile         4    60091
4           4  2011-04-17  10:48:33  mobile         1    60091
5           5  2011-04-08  20:08:14  mobile         4    60091

      session_start
1  2014-01-01 00:00:00
2  2014-01-01 00:17:20
3  2014-01-01 00:28:10
4  2014-01-01 00:44:25
5  2014-01-01 01:11:30
```

Why did variable type change to Id, Index, or datetime_time_index?

During the creation of your EntitySet, you might be wondering why your variable type changed.

```
[10]: data = ft.demo.load_mock_customer()
transactions_df = data["transactions"].merge(data["sessions"]).merge(data["customers
↪"])
products_df = data["products"]

es = ft.EntitySet(id="customer_data")
es = es.entity_from_dataframe(entity_id="transactions",
                             dataframe=transactions_df,
                             index="transaction_id",
                             time_index="transaction_time")
es.plot()
```

[10]: Notice how the variable type of `session_id` is Numeric, and the variable type of `session_start` is Datetime.

Now, let's normalize the transactions entity to create a new entity.

```
[11]: es = es.normalize_entity(base_entity_id="transactions",
                              new_entity_id="sessions",
                              index="session_id",
                              make_time_index="session_start",
                              additional_variables=["session_start"])
es.plot()
```

[11]: The type for `session_id` is now Id in the transactions entity, and Index in the new entity, sessions. This is the case because when we normalize the entity, we create a new relationship between the transactions and sessions. There is a one to many relationship between the parent entity, sessions, and child entity, transactions.

Therefore, `session_id` has type Id in transactions because it represents an Index in another entity. There would be a similar effect if we added another entity using `entity_from_dataframe` and `add_relationship`.

In addition, when we created the new entity, we specified a `time_index` which was the variable (column) `session_start`. This changed the type of `session_start` to `datetime_time_index` in the new sessions entity because it now represents a `time_index`.

How do I combine two or more interesting values?

You might want to create features that are conditioned on multiple values before they are calculated. This would require the use of `interesting_values`. However, since we are trying to create the feature with multiple conditions, we will need to modify the Dataframe before we create the `EntitySet`.

Let's look at how you might accomplish this.

First, let's create our Dataframes.

```
[12]: data = ft.demo.load_mock_customer()
transactions_df = data["transactions"].merge(data["sessions"]).merge(data["customers"]
↳)
products_df = data["products"]
```

```
[13]: transactions_df.head()
```

```
[13]:   transaction_id  session_id  transaction_time  product_id  amount  \
0                298           1 2014-01-01 00:00:00          5  127.64
1                 2           1 2014-01-01 00:01:05          2  109.48
2                308           1 2014-01-01 00:02:10          3   95.06
3                116           1 2014-01-01 00:03:15          4   78.92
4                371           1 2014-01-01 00:04:20          3   31.54

   customer_id  device  session_start  zip_code  join_date  \
0              2  desktop  2014-01-01   13244 2012-04-15 23:31:04
1              2  desktop  2014-01-01   13244 2012-04-15 23:31:04
2              2  desktop  2014-01-01   13244 2012-04-15 23:31:04
3              2  desktop  2014-01-01   13244 2012-04-15 23:31:04
4              2  desktop  2014-01-01   13244 2012-04-15 23:31:04

   date_of_birth
0  1986-08-18
1  1986-08-18
2  1986-08-18
3  1986-08-18
4  1986-08-18
```

```
[14]: products_df.head()
```

```
[14]:   product_id  brand
0           1      B
1           2      B
2           3      B
3           4      B
4           5      A
```

Now, let's modify our `transactions` Dataframe to create the additional column that represents multiple conditions for our feature.

```
[15]: transactions_df['product_id_device'] = transactions_df['product_id'].astype(str) + '_'
↳and ' + transactions_df['device']
```

Here, we created a new column called `product_id_device`, which just combines the `product_id` column, and the `device` column.

Now let's create our `EntitySet`.

```
[16]: es = ft.EntitySet(id="customer_data")
es = es.entity_from_dataframe(entity_id="transactions",
                             dataframe=transactions_df,
                             index="transaction_id",
                             time_index="transaction_time",
                             variable_types={"product_id": ft.variable_types.
↳Categorical,
                                             "product_id_device": ft.variable_types.
↳Categorical,
                                             "zip_code": ft.variable_types.ZIPCode})

es = es.entity_from_dataframe(entity_id="products",
                             dataframe=products_df,
                             index="product_id")
es = es.normalize_entity(base_entity_id="transactions",
                        new_entity_id="sessions",
                        index="session_id",
                        additional_variables=["device", "product_id_device",
↳"customer_id"])
es = es.normalize_entity(base_entity_id="sessions",
                        new_entity_id="customers",
                        index="customer_id")

es
```

```
[16]: Entityset: customer_data
      Entities:
        transactions [Rows: 500, Columns: 9]
        products [Rows: 5, Columns: 2]
        sessions [Rows: 35, Columns: 5]
        customers [Rows: 5, Columns: 2]
      Relationships:
        transactions.session_id -> sessions.session_id
        sessions.customer_id -> customers.customer_id
```

Now, we are ready to add our interesting values.

First, let's view our options for what the interesting values could be.

```
[17]: interesting_values = transactions_df['product_id_device'].unique().tolist()
interesting_values
```

```
[17]: ['5 and desktop',
       '2 and desktop',
       '3 and desktop',
       '4 and desktop',
       '1 and desktop',
       '1 and tablet',
       '3 and tablet',
       '5 and tablet',
       '2 and tablet',
       '4 and tablet',
       '4 and mobile',
       '2 and mobile',
       '3 and mobile',
       '5 and mobile',
       '1 and mobile']
```

If you wanted to, you could pick a subset of these, and the `where` features created would only use those conditions. In our example, we will use all the possible interesting values.

Here, we set all of these values as our interesting values for this specific entity and variable. If we wanted to, we could make interesting values in the same way for more than one variable, but we will just stick with this one for this example.

```
[18]: es['sessions']['product_id_device'].interesting_values = interesting_values
```

Now we can run DFS.

```
[19]: feature_matrix, feature_defs = ft.dfs(entityset=es,
                                          target_entity="customers",
                                          agg_primitives=["count"],
                                          where_primitives=["count"],
                                          trans_primitives=[])
feature_matrix.head()
```

```
[19]:
```

customer_id	COUNT(sessions)	COUNT(transactions)	\
2	7	93	
5	6	79	
4	8	109	
1	8	126	
3	6	93	


```

COUNT(sessions WHERE product_id_device = 5 and tablet) \
customer_id
2 1.0
5 0.0
4 1.0
1 0.0
3 0.0

```

```

COUNT(sessions WHERE product_id_device = 3 and mobile) \
customer_id
2 0.0
5 1.0
4 1.0
1 0.0
3 1.0

```

```

COUNT(sessions WHERE product_id_device = 2 and tablet) \
customer_id
2 0.0
5 0.0
4 0.0
1 0.0
3 1.0

```

```

COUNT(sessions WHERE product_id_device = 1 and tablet) \
customer_id
2 1.0
5 0.0
4 0.0
1 0.0
3 0.0

```

```

COUNT(sessions WHERE product_id_device = 3 and tablet) \
customer_id
2 0.0

```

(continues on next page)

(continued from previous page)

```

5                                0.0
4                                0.0
1                                1.0
3                                0.0

COUNT(sessions WHERE product_id_device = 4 and tablet) \
customer_id
2                                0.0
5                                1.0
4                                0.0
1                                2.0
3                                0.0

COUNT(sessions WHERE product_id_device = 5 and desktop) \
customer_id
2                                1
5                                1
4                                1
1                                1
3                                2

COUNT(sessions WHERE product_id_device = 1 and mobile) ... \
customer_id
2                                0.0    ...
5                                1.0    ...
4                                1.0    ...
1                                0.0    ...
3                                0.0    ...

COUNT(transactions WHERE sessions.product_id_device = 5 and desktop) \
customer_id
2                                16
5                                15
4                                10
1                                12
3                                29

COUNT(transactions WHERE sessions.product_id_device = 1 and mobile) \
customer_id
2                                0.0
5                                18.0
4                                15.0
1                                0.0
3                                0.0

COUNT(transactions WHERE sessions.product_id_device = 4 and mobile) \
customer_id
2                                18.0
5                                10.0
4                                0.0
1                                56.0
3                                0.0

COUNT(transactions WHERE sessions.product_id_device = 2 and desktop) \
customer_id
2                                0.0
5                                0.0

```

(continues on next page)

(continued from previous page)

```

4                                10.0
1                                15.0
3                                0.0

COUNT(transactions WHERE sessions.product_id_device = 5 and tablet) \
customer_id
2                                13.0
5                                0.0
4                                18.0
1                                0.0
3                                0.0

COUNT(transactions WHERE sessions.product_id_device = 3 and desktop) \
customer_id
2                                0.0
5                                0.0
4                                0.0
1                                0.0
3                                0.0

COUNT(transactions WHERE sessions.product_id_device = 4 and desktop) \
customer_id
2                                10.0
5                                14.0
4                                18.0
1                                0.0
3                                0.0

COUNT(transactions WHERE sessions.product_id_device = 4 and tablet) \
customer_id
2                                0.0
5                                14.0
4                                0.0
1                                27.0
3                                0.0

COUNT(transactions WHERE sessions.product_id_device = 1 and tablet) \
customer_id
2                                15.0
5                                0.0
4                                0.0
1                                0.0
3                                0.0

COUNT(transactions WHERE sessions.product_id_device = 2 and tablet)
customer_id
2                                0.0
5                                0.0
4                                0.0
1                                0.0
3                                15.0

[5 rows x 32 columns]
```

To better understand the where clause features, let's examine one of those features. The feature `COUNT(sessions WHERE product_id_device = 5 and tablet)`, tells us how many sessions the customer purchased product_id 5 while on a tablet. Notice how the feature depends on multiple conditions (**product_id = 5 & device = tablet**).

```
[20]: feature_matrix[["COUNT(sessions WHERE product_id_device = 5 and tablet)"]]

```

```
[20]:          COUNT(sessions WHERE product_id_device = 5 and tablet)
customer_id
2                                1.0
5                                0.0
4                                1.0
1                                0.0
3                                0.0

```

3.12.2 DFS

Why is DFS not creating aggregation features?

You may have created your `EntitySet`, and then applied DFS to create features. However, you may be puzzled as to why no aggregation features were created.

- **This is most likely because you have a single table in your entity, and DFS is not capable of creating aggregation features with fewer than 2 entities. Featuretools looks for a relationship, and aggregates based on that relationship.**

Let's look at a simple example.

```
[21]: data = ft.demo.load_mock_customer()
transactions_df = data["transactions"].merge(data["sessions"]).merge(data["customers
→"])

es = ft.EntitySet(id="customer_data")
es = es.entity_from_dataframe(entity_id="transactions",
                             dataframe=transactions_df,
                             index="transaction_id")
es

```

```
[21]: Entityset: customer_data
      Entities:
         transactions [Rows: 500, Columns: 11]
      Relationships:
         No relationships

```

Notice how we only have 1 entity in our `EntitySet`. If we try to create aggregation features on this `EntitySet`, it will not be possible because DFS needs 2 entities to generate aggregation features.

```
[22]: feature_matrix, feature_defs = ft.dfs(entityset=es, target_entity="transactions")
feature_defs

```

```
[22]: [<Feature: session_id>,
<Feature: product_id>,
<Feature: amount>,
<Feature: customer_id>,
<Feature: device>,
<Feature: zip_code>,
<Feature: DAY(date_of_birth)>,
<Feature: DAY(transaction_time)>,
<Feature: DAY(join_date)>,
<Feature: DAY(session_start)>,
<Feature: YEAR(date_of_birth)>,
<Feature: YEAR(transaction_time)>,

```

(continues on next page)

(continued from previous page)

```

<Feature: YEAR(join_date)>,
<Feature: YEAR(session_start)>,
<Feature: MONTH(date_of_birth)>,
<Feature: MONTH(transaction_time)>,
<Feature: MONTH(join_date)>,
<Feature: MONTH(session_start)>,
<Feature: WEEKDAY(date_of_birth)>,
<Feature: WEEKDAY(transaction_time)>,
<Feature: WEEKDAY(join_date)>,
<Feature: WEEKDAY(session_start)>]

```

None of the above features are aggregation features. To fix this issue, you can add another entity to your `EntitySet`.

Solution #1 - You can add new entity if you have additional data.

```

[23]: products_df = data["products"]
es = es.entity_from_dataframe(entity_id="products",
                             dataframe=products_df,
                             index="product_id")
es

```

```

[23]: Entityset: customer_data
      Entities:
        transactions [Rows: 500, Columns: 11]
        products [Rows: 5, Columns: 2]
      Relationships:
        No relationships

```

Notice how we now have an additional entity in our `EntitySet`, called `products`.

Solution #2 - You can normalize an existing entity.

```

[24]: es = es.normalize_entity(base_entity_id="transactions",
                              new_entity_id="sessions",
                              index="session_id",
                              make_time_index="session_start",
                              additional_variables=["device", "customer_id", "zip_code",
↔ "join_date"],
                              copy_variables=["session_start"])
es

```

```

[24]: Entityset: customer_data
      Entities:
        transactions [Rows: 500, Columns: 7]
        products [Rows: 5, Columns: 2]
        sessions [Rows: 35, Columns: 6]
      Relationships:
        transactions.session_id -> sessions.session_id

```

Notice how we now have an additional entity in our `EntitySet`, called `sessions`. Here, the normalization created a relationship between `transactions` and `sessions`. However, we could have specified a relationship between `transactions` and `products` if we had only used Solution #1.

Now, we can generate aggregation features.

```

[25]: feature_matrix, feature_defs = ft.dfs(entityset=es, target_entity="transactions")
      feature_defs[:-10]

```

```
[25]: [<Feature: session_id>,
<Feature: product_id>,
<Feature: amount>,
<Feature: DAY(date_of_birth)>,
<Feature: DAY(transaction_time)>,
<Feature: DAY(session_start)>,
<Feature: YEAR(date_of_birth)>,
<Feature: YEAR(transaction_time)>,
<Feature: YEAR(session_start)>,
<Feature: MONTH(date_of_birth)>,
<Feature: MONTH(transaction_time)>,
<Feature: MONTH(session_start)>,
<Feature: WEEKDAY(date_of_birth)>,
<Feature: WEEKDAY(transaction_time)>,
<Feature: WEEKDAY(session_start)>,
<Feature: sessions.device>,
<Feature: sessions.customer_id>,
<Feature: sessions.zip_code>,
<Feature: sessions.SUM(transactions.amount)>,
<Feature: sessions.STD(transactions.amount)>,
<Feature: sessions.MAX(transactions.amount)>,
<Feature: sessions.SKEW(transactions.amount)>,
<Feature: sessions.MIN(transactions.amount)>,
<Feature: sessions.MEAN(transactions.amount)>,
<Feature: sessions.COUNT(transactions)>]
```

A few of the aggregation features are:

- <Feature: sessions.SUM(transactions.amount)>
- <Feature: sessions.STD(transactions.amount)>
- <Feature: sessions.MAX(transactions.amount)>
- <Feature: sessions.SKEW(transactions.amount)>
- <Feature: sessions.MIN(transactions.amount)>
- <Feature: sessions.MEAN(transactions.amount)>
- <Feature: sessions.COUNT(transactions)>

How do I speed up the runtime of DFS?

One issue you may encounter while running `ft.dfs` is slow performance. While Featuretools has generally optimal default settings for calculating features, you may want to speed up performance when you are calculating on a large number of features.

One quick way to speed up performance is by adjusting the `n_jobs` settings of `ft.dfs` or `ft.calculate_feature_matrix`.

```
# setting n_jobs to -1 will use all cores

feature_matrix, feature_defs = ft.dfs(entityset=es,
                                     target_entity="customers",
                                     n_jobs=-1)

feature_matrix, feature_defs = ft.calculate_feature_matrix(entityset=es,
```

(continues on next page)

(continued from previous page)

```
features=feature_defs,
n_jobs=-1)
```

For more ways to speed up performance, please visit:

- [Improving Computational Performance](#)

How do I include only certain features when running DFS?

When using DFS to generate features, you may wish to include only certain features. There are multiple ways that you do this:

- Use the `ignore_variables` to specify variables in an entity that should not be used to create features. It is a dictionary mapping an entity id to a list of variable names to ignore.
- Use `drop_contains` to drop features that contain any of the strings listed in this parameter.
- Use `drop_exact` to drop features that exactly match any of the strings listed in this parameter.

Here is an example of using all three parameters:

```
[26]: es = ft.demo.load_mock_customer(return_entityset=True)

feature_matrix, feature_defs = ft.dfs(entityset=es,
                                     target_entity="customers",
                                     ignore_variables={
                                         "transactions": ["amount"],
                                         "customers": ["age", "gender", "date_of_
↳birth"]
                                     }, # ignore these variables
                                     drop_contains=["customers.SUM("], # drop_
↳features that contain these strings
                                     drop_exact=["STD(transactions.quantity)"]) #_
↳drop features that exactly match
```

How do I specify primitives on a per column or per entity basis?

When using DFS to generate features, you may wish to use only certain features or entities for specific primitives. This can be done through the `primitive_options` parameter. The `primitive_options` parameter is a dictionary or list of dictionaries that maps a primitive or a tuple of primitives to a dictionary containing options for the primitive(s). This parameter can also be a list of option dictionaries if the primitive takes multiple inputs. Each dictionary supplies options for their respective input column. There are multiple ways to control how primitives get applied through these options:

- Use `ignore_entities` to specify entities that should not be used to create features for that primitive. It is a list of entity ids to ignore.
- Use `include_entities` to specify the only entities to be included to create features for that primitive. It is a list of entity ids to include.
- Use `ignore_variables` to specify variables in an entity that should not be used to create features for that primitive. It is a dictionary mapping an entity id to a list of variable names to ignore.
- Use `include_variables` to specify the only variables in an entity that should be used to create features for that primitive. It is a dictionary mapping an entity id to a list of variable names to include.

You can also use `primitive_options` to specify which entities or variables you wish to use as groupbys for groupby transformation primitives:

- Use `ignore_groupby_entities` to specify entities that should not be used to get groupbys for that primitive. It is a list of entity ids to ignore.
- Use `include_groupby_entities` to specify the only entities that should be used to get groupbys for that primitive. It is a list of entity ids to include.
- Use `ignore_groupby_variables` to specify variables in an entity that should not be used as groupbys for that primitive. It is a dictionary mapping an entity id to a list of variable names to ignore.
- Use `include_groupby_variables` to specify the only variables in an entity that should be used as groupbys for that primitive. It is a dictionary mapping an entity id to a list of variable names to include.

Here is an example of using some of these options:

```
[27]: es = ft.demo.load_mock_customer(return_entityset=True)

feature_matrix, feature_defs = ft.dfs(entityset=es,
                                     target_entity="customers",
                                     primitive_options={"mode": {"ignore_entities": [
↳ "sessions"],
                                                         "include_variables
↳ ": {"products": ["brand"],
                                     "transactions": ["product_id"]}},
                                     # For mode, ignore the
                                     # "products" entity and
↳ "sessions" entity and only include "brands" in the
                                     ("count", "mean"): {"include_
↳ "product_id" in the "transactions" entity
                                     ("count", "mean"): {"include_
↳ entities": ["sessions", "transactions"]}
                                     # For count and mean, only_
↳ include the entities "sessions" and "transactions"
                                     })
```

For a more examples of specifying options for DFS, please visit:

- [Specifying Primitive Options](#)

If I didn't specify the `cutoff_time`, what date will be used for the feature calculations?

The cutoff time will be set to the current time using `cutoff_time = datetime.now()`.

How do I select a certain amount of past data when calculating features?

You may encounter a situation when you wish to make prediction using only a certain amount of historical data. You can accomplish this using the `training_window` parameter in `ft.dfs`. When you use the `training_window`, Featuretools will use the historical data between the `cutoff_time` and `cutoff_time - training_window`.

In order to make the calculation, Featuretools will check the time in the `time_index` column of the `target_entity`.

```
[28]: es = ft.demo.load_mock_customer(return_entityset=True)
es['customers'].time_index
```

```
[28]: 'join_date'
```

Our `target_entity` has a `time_index`, which is needed for the `training_window` calculation. Here, we are creating a cutoff time dataframe so that we can have a unique training window for each customer.

```
[29]: cutoff_times = pd.DataFrame()
cutoff_times['customer_id'] = [1, 2, 3, 1]
cutoff_times['time'] = pd.to_datetime(['2014-1-1 04:00', '2014-1-1 05:00', '2014-1-1-1_
↪06:00', '2014-1-1 08:00'])
cutoff_times['label'] = [True, True, False, True]
```

```
feature_matrix, feature_defs = ft.dfs(entityset=es,
                                     target_entity="customers",
                                     cutoff_time=cutoff_times,
                                     cutoff_time_in_index=True,
                                     training_window="1 hour")
feature_matrix.head()
```

```
[29]:
```

customer_id	time	zip_code	COUNT(sessions)	\
1	2014-01-01 04:00:00	60091	1	
2	2014-01-01 05:00:00	13244	1	
3	2014-01-01 06:00:00	13244	2	
1	2014-01-01 08:00:00	60091	1	

customer_id	time	NUM_UNIQUE(sessions.device)	\
1	2014-01-01 04:00:00	1	
2	2014-01-01 05:00:00	1	
3	2014-01-01 06:00:00	1	
1	2014-01-01 08:00:00	1	

customer_id	time	MODE(sessions.device)	\
1	2014-01-01 04:00:00	tablet	
2	2014-01-01 05:00:00	tablet	
3	2014-01-01 06:00:00	desktop	
1	2014-01-01 08:00:00	mobile	

customer_id	time	SUM(transactions.amount)	\
1	2014-01-01 04:00:00	1025.63	
2	2014-01-01 05:00:00	1004.96	
3	2014-01-01 06:00:00	980.97	
1	2014-01-01 08:00:00	1420.09	

customer_id	time	STD(transactions.amount)	\
1	2014-01-01 04:00:00	39.825249	
2	2014-01-01 05:00:00	33.725036	
3	2014-01-01 06:00:00	34.188788	
1	2014-01-01 08:00:00	32.324534	

customer_id	time	MAX(transactions.amount)	\
1	2014-01-01 04:00:00	139.09	
2	2014-01-01 05:00:00	118.85	
3	2014-01-01 06:00:00	128.26	

(continues on next page)

(continued from previous page)

1	2014-01-01 08:00:00	126.11	
			SKEW(transactions.amount) \
customer_id	time		
1	2014-01-01 04:00:00	-0.830975	
2	2014-01-01 05:00:00	-0.314918	
3	2014-01-01 06:00:00	-0.386592	
1	2014-01-01 08:00:00	-1.038434	
			MIN(transactions.amount) \
customer_id	time		
1	2014-01-01 04:00:00	6.78	
2	2014-01-01 05:00:00	21.82	
3	2014-01-01 06:00:00	20.06	
1	2014-01-01 08:00:00	11.62	
			MEAN(transactions.amount) ... \
customer_id	time		...
1	2014-01-01 04:00:00	85.469167	...
2	2014-01-01 05:00:00	77.304615	...
3	2014-01-01 06:00:00	81.747500	...
1	2014-01-01 08:00:00	88.755625	...
			MODE(sessions.MODE(transactions.product_id)) \
customer_id	time		
1	2014-01-01 04:00:00		4
2	2014-01-01 05:00:00		1
3	2014-01-01 06:00:00		1
1	2014-01-01 08:00:00		4
			MODE(sessions.MONTH(session_start)) \
customer_id	time		
1	2014-01-01 04:00:00		1
2	2014-01-01 05:00:00		1
3	2014-01-01 06:00:00		1
1	2014-01-01 08:00:00		1
			MODE(sessions.WEEKDAY(session_start)) \
customer_id	time		
1	2014-01-01 04:00:00		2
2	2014-01-01 05:00:00		2
3	2014-01-01 06:00:00		2
1	2014-01-01 08:00:00		2
			MODE(sessions.YEAR(session_start)) \
customer_id	time		
1	2014-01-01 04:00:00		2014
2	2014-01-01 05:00:00		2014
3	2014-01-01 06:00:00		2014
1	2014-01-01 08:00:00		2014
			MODE(sessions.DAY(session_start)) \
customer_id	time		
1	2014-01-01 04:00:00		1
2	2014-01-01 05:00:00		1
3	2014-01-01 06:00:00		1
1	2014-01-01 08:00:00		1

(continues on next page)

(continued from previous page)

```

NUM_UNIQUE(transactions.sessions.customer_id) \
customer_id time
1      2014-01-01 04:00:00      1
2      2014-01-01 05:00:00      1
3      2014-01-01 06:00:00      1
1      2014-01-01 08:00:00      1

NUM_UNIQUE(transactions.sessions.device) \
customer_id time
1      2014-01-01 04:00:00      1
2      2014-01-01 05:00:00      1
3      2014-01-01 06:00:00      1
1      2014-01-01 08:00:00      1

MODE(transactions.sessions.customer_id) \
customer_id time
1      2014-01-01 04:00:00      1
2      2014-01-01 05:00:00      2
3      2014-01-01 06:00:00      3
1      2014-01-01 08:00:00      1

MODE(transactions.sessions.device) label
customer_id time
1      2014-01-01 04:00:00      tablet  True
2      2014-01-01 05:00:00      tablet  True
3      2014-01-01 06:00:00      desktop False
1      2014-01-01 08:00:00      mobile  True

[4 rows x 78 columns]
```

Above, we ran DFS with `training_window` argument of 1 hour to create features that only used customer data collected in the last hour (from the cutoff time we provided).

How do I apply DFS to a single table?

You can run DFS on a single table. Featuretools will be able to generate features for your data, but only transform features.

For example:

```
[30]: transactions_df = ft.demo.load_mock_customer(return_single_table=True)

es = ft.EntitySet(id="customer_data")
es = es.entity_from_dataframe(entity_id="transactions",
                             dataframe=transactions_df,
                             index="transaction_id",
                             time_index="transaction_time")

feature_matrix, feature_defs = ft.dfs(entityset=es,
                                       target_entity="transactions",
                                       trans_primitives=['time_since', 'day', 'is_
↳ weekend',
                                                         'cum_min', 'minute',
                                                         'num_words', 'weekday', 'cum_
↳ count',
```

(continues on next page)

(continued from previous page)

```
'percentile', 'year', 'week',
'cum_mean']])
```

Before we examine the output, let's look at our original single table.

```
[31]: transactions_df.head()
```

```
[31]:   transaction_id  session_id  transaction_time  product_id  amount  \
0              298           1 2014-01-01 00:00:00           5  127.64
1              10           1 2014-01-01 00:09:45           5   57.39
2             495           1 2014-01-01 00:14:05           5   69.45
3             460          10 2014-01-01 02:33:50           5  123.19
4             302          10 2014-01-01 02:37:05           5   64.47

   customer_id  device  session_start  zip_code  join_date  \
0             2  desktop 2014-01-01 00:00:00  13244 2012-04-15 23:31:04
1             2  desktop 2014-01-01 00:00:00  13244 2012-04-15 23:31:04
2             2  desktop 2014-01-01 00:00:00  13244 2012-04-15 23:31:04
3             2  tablet 2014-01-01 02:31:40  13244 2012-04-15 23:31:04
4             2  tablet 2014-01-01 02:31:40  13244 2012-04-15 23:31:04

   date_of_birth  brand
0  1986-08-18     A
1  1986-08-18     A
2  1986-08-18     A
3  1986-08-18     A
4  1986-08-18     A
```

Now we can look at the transformations that Featuretools was able to apply to this single entity (table) to create feature matrix.

```
[32]: feature_matrix.head()
```

```
[32]:   session_id  product_id  amount  customer_id  device  zip_code  \
transaction_id
298           1           5  127.64           2  desktop  13244
2            1           2  109.48           2  desktop  13244
308           1           3   95.06           2  desktop  13244
116           1           4   78.92           2  desktop  13244
371           1           3   31.54           2  desktop  13244

   brand  TIME_SINCE(transaction_time)  DAY(date_of_birth)  \
transaction_id
298     A           1.919868e+08           18
2       B           1.919867e+08           18
308     B           1.919867e+08           18
116     B           1.919866e+08           18
371     B           1.919865e+08           18

   DAY(transaction_time)  ...  \
transaction_id           ...
298                     1  ...
2                       1  ...
308                     1  ...
116                     1  ...
371                     1  ...
```

(continues on next page)

(continued from previous page)

	CUM_MEAN (TIME_SINCE (transaction_time)) \	
transaction_id		
298	1.919868e+08	
2	1.919868e+08	
308	1.919867e+08	
116	1.919867e+08	
371	1.919867e+08	
	CUM_MEAN (CUM_MIN (amount)) CUM_MEAN (MINUTE (date_of_birth)) \	
transaction_id		
298	127.640000	0.0
2	118.560000	0.0
308	110.726667	0.0
116	102.775000	0.0
371	88.528000	0.0
	CUM_MEAN (PERCENTILE (session_id)) \	
transaction_id		
298	0.017	
2	0.017	
308	0.017	
116	0.017	
371	0.017	
	CUM_MEAN (PERCENTILE (customer_id)) \	
transaction_id		
298	0.346	
2	0.346	
308	0.346	
116	0.346	
371	0.346	
	CUM_MEAN (MINUTE (join_date)) CUM_MEAN (PERCENTILE (amount)) \	
transaction_id		
298	31.0	0.846000
2	31.0	0.793000
308	31.0	0.743333
116	31.0	0.693500
371	31.0	0.597200
	CUM_MEAN (CUM_MIN (session_id)) CUM_MEAN (CUM_MIN (customer_id)) \	
transaction_id		
298	1.0	2.0
2	1.0	2.0
308	1.0	2.0
116	1.0	2.0
371	1.0	2.0
	CUM_MEAN (MINUTE (transaction_time))	
transaction_id		
298	0.0	
2	0.5	
308	1.0	
116	1.5	
371	2.0	
[5 rows x 61 columns]		

Can I automatically normalize a single table?

Yes, another open source library [AutoNormalize](#), also produced by Feature Labs, automates table normalization and integrates with Featuretools. To install run:

```
python -m pip install featuretools[autonormalize]
```

A normalized EntitySet will help Featuretools to generate more features. For example:

```
[33]: from featuretools.autonormalize import autonormalize as an
      es = an.normalize_entity(es)
      es.plot()
```

```
100%|| 10/10 [00:03<00:00, 3.20it/s]
```

[33]: As you can see, AutoNormalize creates a relational EntitySet. Below, we run dfs on the EntitySet, and you can see all the features created; take note of the aggregated features.

```
[34]: feature_matrix, feature_defs = ft.dfs(entityset=es,
      target_entity="transaction_id",
      trans_primitives=[])
      feature_matrix.head()
```

```
[34]:      session_id product_id amount session_id.customer_id \
transaction_id
298          1          5 127.64          2
2            1          2 109.48          2
308          1          3  95.06          2
116          1          4  78.92          2
371          1          3  31.54          2
```

```
      session_id.device product_id.brand \
transaction_id
298          desktop          A
2            desktop          B
308          desktop          B
116          desktop          B
371          desktop          B
```

```
      session_id.SUM(transaction_id.amount) \
transaction_id
298          1229.01
2            1229.01
308          1229.01
116          1229.01
371          1229.01
```

```
      session_id.STD(transaction_id.amount) \
transaction_id
298          41.600976
2            41.600976
308          41.600976
116          41.600976
371          41.600976
```

```
      session_id.MAX(transaction_id.amount) \
transaction_id
298          141.66
```

(continues on next page)

(continued from previous page)

```

2                141.66
308              141.66
116              141.66
371              141.66

        session_id.SKEW(transaction_id.amount) ... \
transaction_id  ...
298            0.295458 ...
2              0.295458 ...
308            0.295458 ...
116            0.295458 ...
371            0.295458 ...

        session_id.customer_id.zip_code \
transaction_id
298            13244
2              13244
308            13244
116            13244
371            13244

        product_id.SUM(transaction_id.amount) \
transaction_id
298            7931.55
2              7021.43
308            7008.12
116            8088.97
371            7008.12

        product_id.STD(transaction_id.amount) \
transaction_id
298            42.131902
2              46.336308
308            38.871405
116            42.492501
371            38.871405

        product_id.MAX(transaction_id.amount) \
transaction_id
298            149.02
2              149.95
308            148.31
116            146.46
371            148.31

        product_id.SKEW(transaction_id.amount) \
transaction_id
298            0.098248
2              0.151934
308            0.223938
116            -0.132077
371            0.223938

        product_id.MIN(transaction_id.amount) \
transaction_id
298            5.91
2              5.73

```

(continues on next page)

(continued from previous page)

```

308                    5.89
116                    5.81
371                    5.89

                    product_id.MEAN(transaction_id.amount) \
transaction_id
298                    76.264904
2                    76.319891
308                    73.001250
116                    76.311038
371                    73.001250

                    product_id.COUNT(transaction_id) \
transaction_id
298                    104
2                    92
308                    96
116                    106
371                    96

                    product_id.NUM_UNIQUE(transaction_id.session_id) \
transaction_id
298                    34
2                    34
308                    35
116                    34
371                    35

                    product_id.MODE(transaction_id.session_id)
transaction_id
298                    4
2                    28
308                    1
116                    29
371                    1

[5 rows x 25 columns]
```

How do I prevent label leakage with DFS?

One concern you might have with using DFS is about label leakage. You want to make sure that labels in your data aren't used incorrectly to create features and the feature matrix.

Featuretools is particularly focused on helping users avoid label leakage.

There are two ways to prevent label leakage depending on if your data has timestamps or not.

1. Data without timestamps

In the case where you do not have timestamps, you can create one `EntitySet` using only the training data and then run `ft.dfs`. This will create a feature matrix using only the training data, but also return a list of feature definitions. Next, you can create an `EntitySet` using the test data and recalculate the same features by calling `ft.calculate_feature_matrix` with the list of feature definitions from before.

Here is what that flow would look like:

First, let's create our training data.

```
[35]: train_data = pd.DataFrame({"customer_id": [1, 2, 3, 4, 5],
                                "age": [40, 50, 10, 20, 30],
                                "gender": ["m", "f", "m", "f", "f"],
                                "signup_date": pd.date_range('2014-01-01 01:41:50',
                                ↪periods=5, freq='25min'),
                                "labels": [True, False, True, False, True]})
train_data.head()
```

```
[35]:   customer_id  age gender      signup_date  labels
0           1   40     m 2014-01-01 01:41:50    True
1           2   50     f 2014-01-01 02:06:50   False
2           3   10     m 2014-01-01 02:31:50    True
3           4   20     f 2014-01-01 02:56:50   False
4           5   30     f 2014-01-01 03:21:50    True
```

Now, we can create an entityset for our training data.

```
[36]: es_train_data = ft.EntitySet(id="customer_train_data")
es_train_data = es_train_data.entity_from_dataframe(entity_id="customers",
                                                    dataframe=train_data,
                                                    index="customer_id")

es_train_data
```

```
[36]: Entityset: customer_train_data
      Entities:
        customers [Rows: 5, Columns: 5]
      Relationships:
        No relationships
```

Next, we are ready to create our features, and feature matrix for the training data.

```
[37]: feature_matrix_train, feature_defs = ft.dfs(entityset=es_train_data,
                                                    target_entity="customers")

feature_matrix_train
```

```
[37]:   customer_id  age gender  labels  DAY(signup_date)  YEAR(signup_date)  \
1           1   40     m    True                1             2014
2           2   50     f   False                1             2014
3           3   10     m    True                1             2014
4           4   20     f   False                1             2014
5           5   30     f    True                1             2014

      MONTH(signup_date)  WEEKDAY(signup_date)
customer_id
1                   1                2
2                   1                2
3                   1                2
4                   1                2
5                   1                2
```

We will also encode our feature matrix to make machine learning compatible features.

```
[38]: feature_matrix_train_enc, features_enc = ft.encode_features(feature_matrix_train,
                                ↪feature_defs)
feature_matrix_train_enc.head()
```

```
[38]:
```

customer_id	age	gender = f	gender = m	gender is unknown	labels
1	40	0	1	0	True
2	50	1	0	0	False
3	10	0	1	0	True
4	20	1	0	0	False
5	30	1	0	0	True

customer_id	DAY(signup_date) = 1	DAY(signup_date) is unknown
1	1	0
2	1	0
3	1	0
4	1	0
5	1	0

customer_id	YEAR(signup_date) = 2014	YEAR(signup_date) is unknown
1	1	0
2	1	0
3	1	0
4	1	0
5	1	0

customer_id	MONTH(signup_date) = 1	MONTH(signup_date) is unknown
1	1	0
2	1	0
3	1	0
4	1	0
5	1	0

customer_id	WEEKDAY(signup_date) = 2	WEEKDAY(signup_date) is unknown
1	1	0
2	1	0
3	1	0
4	1	0
5	1	0

Notice how the the whole feature matrix only includes numeric values now.

Now we can use the feature definitions to calculate our feature matrix for the test data, and avoid label leakage.

```
[39]: test_train = pd.DataFrame({"customer_id": [6, 7, 8, 9, 10],
                                "age": [20, 25, 55, 22, 35],
                                "gender": ["f", "m", "m", "m", "m"],
                                "signup_date": pd.date_range('2014-01-01 01:41:50',
                                ↪periods=5, freq='25min')})

# lets add NaN label column to the test Dataframe
test_train['labels'] = np.nan

es_test_data = ft.EntitySet(id="customer_test_data")
es_test_data = es_test_data.entity_from_dataframe(entity_id="customers",
                                                  dataframe=test_train,
                                                  index="customer_id",
                                                  time_index="signup_date")
```

(continues on next page)

(continued from previous page)

```

# Use the feature definitions from earlier
feature_matrix_enc_test = ft.calculate_feature_matrix(features=features_enc,
                                                    entityset=es_test_data)

feature_matrix_enc_test.head()

```

[39]:

customer_id	age	gender = f	gender = m	gender is unknown	labels	\
6	20	True	False	False	NaN	
7	25	False	True	False	NaN	
8	55	False	True	False	NaN	
9	22	False	True	False	NaN	
10	35	False	True	False	NaN	

customer_id	DAY(signup_date) = 1	DAY(signup_date) is unknown	\
6	True	False	
7	True	False	
8	True	False	
9	True	False	
10	True	False	

customer_id	YEAR(signup_date) = 2014	YEAR(signup_date) is unknown	\
6	True	False	
7	True	False	
8	True	False	
9	True	False	
10	True	False	

customer_id	MONTH(signup_date) = 1	MONTH(signup_date) is unknown	\
6	True	False	
7	True	False	
8	True	False	
9	True	False	
10	True	False	

customer_id	WEEKDAY(signup_date) = 2	WEEKDAY(signup_date) is unknown	\
6	True	False	
7	True	False	
8	True	False	
9	True	False	
10	True	False	

Note: Disregard the difference between the False/True above, and 0/1 in the earlier feature matrix. A simple casting would address this difference.

2. Data with timestamps

If your data has timestamps, the best way to prevent label leakage is to use a list of **cutoff times**, which specify the last point in time data is allowed to be used for each row in the resulting feature matrix. To use **cutoff times**, you need to set a time index for each time sensitive entity in your entity set.

Tip: Even if your data doesn't have time stamps, you could add a column with dummy timestamps that can be used by Featuretools as time index.

When you call `ft.dfs`, you can provide a Dataframe of cutoff times like this:

```
[40]: cutoff_times = pd.DataFrame({"customer_id": [1, 2, 3, 4, 5],
                                "time": pd.date_range('2014-01-01 01:41:50', periods=5,
                                ↪freq='25min')})
cutoff_times.head()
```

```
[40]:
```

	customer_id	time
0	1	2014-01-01 01:41:50
1	2	2014-01-01 02:06:50
2	3	2014-01-01 02:31:50
3	4	2014-01-01 02:56:50
4	5	2014-01-01 03:21:50

```
[41]: train_test_data = pd.DataFrame({"customer_id": [1, 2, 3, 4, 5],
                                     "age": [20, 25, 55, 22, 35],
                                     "gender": ["f", "m", "m", "m", "m"],
                                     "signup_date": pd.date_range('2010-01-01 01:41:50',
                                     ↪periods=5, freq='25min')})

es_train_test_data = ft.EntitySet(id="customer_train_test_data")
es_train_test_data = es_train_test_data.entity_from_dataframe(entity_id="customers",
                                                             ↪dataframe=train_test_
                                                             ↪data,
                                                             index="customer_id",
                                                             ↪time_index="signup_date
                                                             ↪")

feature_matrix_train_test, features = ft.dfs(entityset=es_train_test_data,
                                             target_entity="customers",
                                             cutoff_time=cutoff_times,
                                             cutoff_time_in_index=True)

feature_matrix_train_test.head()
```

```
[41]:
```

customer_id	time	age	gender	DAY(signup_date)	\
1	2014-01-01 01:41:50	20	f	1	
2	2014-01-01 02:06:50	25	m	1	
3	2014-01-01 02:31:50	55	m	1	
4	2014-01-01 02:56:50	22	m	1	
5	2014-01-01 03:21:50	35	m	1	

customer_id	time	YEAR(signup_date)	MONTH(signup_date)	\
1	2014-01-01 01:41:50	2010	1	
2	2014-01-01 02:06:50	2010	1	
3	2014-01-01 02:31:50	2010	1	
4	2014-01-01 02:56:50	2010	1	
5	2014-01-01 03:21:50	2010	1	

customer_id	time	WEEKDAY(signup_date)
1	2014-01-01 01:41:50	4
2	2014-01-01 02:06:50	4
3	2014-01-01 02:31:50	4
4	2014-01-01 02:56:50	4

(continues on next page)

(continued from previous page)

```
5          2014-01-01 03:21:50          4
```

Above, we have created a feature matrix that uses cutoff times to avoid label leakage. We could also encode this feature matrix using `ft.encode_features`.

What is the difference between passing a primitive object versus a string to DFS?

There are 2 ways to pass primitives to DFS: the primitive object, or a string of the primitive name.

We will use the Transform primitive called `TimeSincePrevious` to illustrate the differences.

First, let's use the string of primitive name.

```
[42]: es = ft.demo.load_mock_customer(return_entityset=True)
```

```
[43]: feature_matrix, feature_defs = ft.dfs(entityset=es,
                                          target_entity="customers",
                                          agg_primitives=[],
                                          trans_primitives=["time_since_previous"])

feature_matrix
```

```
[43]:      zip_code  TIME_SINCE_PREVIOUS(join_date)
customer_id
5          60091                NaN
4          60091          22948824.0
1          60091          744019.0
3          13244          10212841.0
2          13244          21282510.0
```

Now, let's use the primitive object.

```
[44]: from featuretools.primitives import TimeSincePrevious

feature_matrix, feature_defs = ft.dfs(entityset=es,
                                          target_entity="customers",
                                          agg_primitives=[],
                                          trans_primitives=[TimeSincePrevious])

feature_matrix
```

```
[44]:      zip_code  TIME_SINCE_PREVIOUS(join_date)
customer_id
5          60091                NaN
4          60091          22948824.0
1          60091          744019.0
3          13244          10212841.0
2          13244          21282510.0
```

As we can see above, the feature matrix is the same.

However, if we need to modify controllable parameters in the primitive, we should use the primitive object. For instance, let's make `TimeSincePrevious` return units of hours (the default is in seconds).

```
[45]: from featuretools.primitives import TimeSincePrevious

time_since_previous_in_hours = TimeSincePrevious(unit='hours')

feature_matrix, feature_defs = ft.dfs(entityset=es,
```

(continues on next page)

(continued from previous page)

```

target_entity="customers",
agg_primitives=[],
trans_primitives=[time_since_previous_in_hours])
feature_matrix
[45]:
      zip_code  TIME_SINCE_PREVIOUS(join_date, unit=hours)
customer_id
5           60091                               NaN
4           60091          6374.673333
1           60091          206.671944
3           13244          2836.900278
2           13244          5911.808333

```

3.12.3 Features

How can I select features based on some attributes (a specific string, an explicit primitive type, a return type, a given depth)?

You may wish to select a subset of your features based on some attributes.

Let's say you wanted to select features that had the string `amount` in its name. You can check for this by using the `get_name` function on the feature definitions.

```

[46]: es = ft.demo.load_mock_customer(return_entityset=True)

feature_defs = ft.dfs(entityset=es,
                      target_entity="customers",
                      features_only=True)

features_with_amount = []
for x in feature_defs:
    if 'amount' in x.get_name():
        features_with_amount.append(x)
features_with_amount[0:5]
[46]: [<Feature: SUM(transactions.amount)>,
<Feature: STD(transactions.amount)>,
<Feature: MAX(transactions.amount)>,
<Feature: SKEW(transactions.amount)>,
<Feature: MIN(transactions.amount)>]

```

You might also want to only select features that are aggregation features.

```

[47]: from featuretools import AggregationFeature

features_only_aggregations = []
for x in feature_defs:
    if type(x) == AggregationFeature:
        features_only_aggregations.append(x)
features_only_aggregations[0:5]
[47]: [<Feature: COUNT(sessions)>,
<Feature: NUM_UNIQUE(sessions.device)>,
<Feature: MODE(sessions.device)>,
<Feature: SUM(transactions.amount)>,
<Feature: STD(transactions.amount)>]

```

Also, you might only want to select features that are calculated at a certain depth. You can do this by using the `get_depth` function.

```
[48]: features_only_depth_2 = []
      for x in feature_defs:
          if x.get_depth() == 2:
              features_only_depth_2.append(x)
      features_only_depth_2[0:5]

[48]: [<Feature: SUM(sessions.NUM_UNIQUE(transactions.product_id))>,
      <Feature: SUM(sessions.SKEW(transactions.amount))>,
      <Feature: SUM(sessions.STD(transactions.amount))>,
      <Feature: SUM(sessions.MAX(transactions.amount))>,
      <Feature: SUM(sessions.MEAN(transactions.amount))>]
```

Finally, you might only want features that return a certain type. You can do this by using the `variable_type` function.

```
[49]: from featuretools.variable_types import Numeric

      features_only_numeric = []
      for x in feature_defs:
          if x.variable_type == Numeric:
              features_only_numeric.append(x)
      features_only_numeric[0:5]

[49]: [<Feature: COUNT(sessions)>,
      <Feature: NUM_UNIQUE(sessions.device)>,
      <Feature: SUM(transactions.amount)>,
      <Feature: STD(transactions.amount)>,
      <Feature: MAX(transactions.amount)>]
```

Once you have your specific feature list, you can use `ft.calculate_feature_matrix` to generate a feature matrix for only those features.

For our example, let's use the features with only the string `amount` in its name.

```
[50]: feature_matrix = ft.calculate_feature_matrix(entityset=es,
                                                features=features_with_amount) # change_
      ↪to your specific feature list
      feature_matrix.head()

[50]:
```

	SUM(transactions.amount)	STD(transactions.amount)	\
customer_id			
5	6349.66	44.095630	
4	8727.68	45.068765	
1	9025.62	40.442059	
3	6236.62	43.683296	
2	7200.28	37.705178	
	MAX(transactions.amount)	SKEW(transactions.amount)	\
customer_id			
5	149.02	-0.025941	
4	149.95	-0.036348	
1	139.43	0.019698	
3	149.15	0.418230	
2	146.81	0.098259	
	MIN(transactions.amount)	MEAN(transactions.amount)	\

(continues on next page)

(continued from previous page)

```
customer_id
5          7.55          80.375443
4          5.73          80.070459
1          5.81          71.631905
3          5.89          67.060430
2          8.73          77.422366
```

SUM(sessions.SKEW(transactions.amount)) \

```
customer_id
5          0.014384
4          0.002764
1         -0.476122
3          2.286086
2         -0.277640
```

SUM(sessions.STD(transactions.amount)) \

```
customer_id
5          259.873954
4          356.125829
1          312.745952
3          257.299895
2          258.700528
```

SUM(sessions.MAX(transactions.amount)) \

```
customer_id
5          839.76
4          1157.99
1          1057.97
3          847.63
2          931.63
```

SUM(sessions.MEAN(transactions.amount)) ... \

```
customer_id
5          472.231119 ...
4          649.657515 ...
1          582.193117 ...
3          405.237462 ...
2          548.905851 ...
```

MIN(sessions.SKEW(transactions.amount)) \

```
customer_id
5         -0.539060
4         -0.711744
1         -1.038434
3         -0.289466
2         -0.763603
```

MIN(sessions.STD(transactions.amount)) \

```
customer_id
5          36.734681
4          29.026424
1          30.450261
3          35.704680
2          27.839228
```

MIN(sessions.MAX(transactions.amount)) \

```
customer_id
```

(continues on next page)

(continued from previous page)

```

5          128.51
4          139.20
1          118.90
3          126.74
2          100.04

MIN(sessions.MEAN(transactions.amount)) \
customer_id
5          66.666667
4          70.638182
1          50.623125
3          55.579412
2          61.910000

MEAN(sessions.SUM(transactions.amount)) \
customer_id
5          1058.276667
4          1090.960000
1          1128.202500
3          1039.436667
2          1028.611429

MEAN(sessions.SKEW(transactions.amount)) \
customer_id
5          0.002397
4          0.000346
1          -0.059515
3          0.381014
2          -0.039663

MEAN(sessions.STD(transactions.amount)) \
customer_id
5          43.312326
4          44.515729
1          39.093244
3          42.883316
2          36.957218

MEAN(sessions.MAX(transactions.amount)) \
customer_id
5          139.960000
4          144.748750
1          132.246250
3          141.271667
2          133.090000

MEAN(sessions.MEAN(transactions.amount)) \
customer_id
5          78.705187
4          81.207189
1          72.774140
3          67.539577
2          78.415122

MEAN(sessions.MIN(transactions.amount))
customer_id
5          14.415000

```

(continues on next page)

(continued from previous page)

```

4                16.438750
1                9.823750
3                11.035000
2                22.085714

[5 rows x 37 columns]

```

Above, notice how all the column names for our feature matrix contain the string amount.

How do I create where features?

Sometimes, you might want to create features that are conditioned on a second value before it is calculated. This extra filter is called a “where clause”. You can create these features using the using the `interesting_values` of a variable.

If you have categorical columns in your `EntitySet`, you can use then `add_interesting_values`. This function will find interesting values for your categorical variables, which can then be used to generate “where” clauses.

First, let’s create our `EntitySet`.

```

[51]: es = ft.demo.load_mock_customer(return_entityset=True)
      es

[51]: Entityset: transactions
      Entities:
      transactions [Rows: 500, Columns: 5]
      products [Rows: 5, Columns: 2]
      sessions [Rows: 35, Columns: 4]
      customers [Rows: 5, Columns: 4]
      Relationships:
      transactions.product_id -> products.product_id
      transactions.session_id -> sessions.session_id
      sessions.customer_id -> customers.customer_id

```

Now we can add the interesting variables for the categorical variables.

```

[52]: es.add_interesting_values()

```

Now we can run DFS with the `where_primitives` argument to define which primitives to apply with where clauses. In this case, let’s use the primitive count.

```

[53]: feature_matrix, feature_defs = ft.dfs(entityset=es,
      target_entity="customers",
      agg_primitives=["count"],
      where_primitives=["count"],
      trans_primitives=[])

feature_matrix.head()

[53]:   customer_id  zip_code  COUNT(sessions)  COUNT(transactions)  \
5          60091          6                79
4          60091          8                109
1          60091          8                126
3          13244          6                93
2          13244          7                93

      COUNT(sessions WHERE device = desktop)  \

```

(continues on next page)

(continued from previous page)

```

customer_id
5                2
4                3
1                2
3                4
2                3

COUNT(sessions WHERE device = tablet) \
customer_id
5                1
4                1
1                3
3                1
2                2

COUNT(sessions WHERE device = mobile) \
customer_id
5                3
4                4
1                3
3                1
2                2

COUNT(sessions WHERE customers.zip_code = 60091) \
customer_id
5                6.0
4                8.0
1                8.0
3                0.0
2                0.0

COUNT(sessions WHERE customers.zip_code = 13244) \
customer_id
5                0.0
4                0.0
1                0.0
3                6.0
2                7.0

COUNT(transactions WHERE sessions.device = desktop) \
customer_id
5                29
4                38
1                27
3                62
2                34

COUNT(transactions WHERE sessions.device = mobile) \
customer_id
5                36
4                53
1                56
3                16
2                31

COUNT(transactions WHERE sessions.device = tablet)
customer_id

```

(continues on next page)

(continued from previous page)

5	14
4	18
1	43
3	15
2	28

We have now created some useful features. One example of a useful feature is the `COUNT(sessions WHERE device = tablet)`. This feature tells us how many sessions a customer completed on a tablet.

```
[54]: feature_matrix[["COUNT(sessions WHERE device = tablet)"]]
```

```
[54]:          COUNT(sessions WHERE device = tablet)
customer_id
5                1
4                1
1                3
3                1
2                2
```

3.12.4 Primitives

What is the difference between the primitive types (Transform, GroupBy Transform, & Aggregation)?

You might curious to know the difference between the primitive groups. Let's review the differences between transform, groupby transform, and aggregation primitives.

First, let's create a simple `EntitySet`.

```
[55]: import pandas as pd
import featuretools as ft

df = pd.DataFrame({
    "id": [1, 2, 3, 4, 5, 6],
    "time_index": pd.date_range("1/1/2019", periods=6, freq="D"),
    "group": ["a", "b", "a", "c", "a", "b"],
    "val": [5, 1, 10, 20, 6, 23],
})

es = ft.EntitySet()
es = es.entity_from_dataframe(entity_id="observations",
                             dataframe=df,
                             index="id",
                             time_index="time_index")

es = es.normalize_entity(base_entity_id="observations",
                        new_entity_id="groups",
                        index="group")

es.plot()
```

[55]: After calling `normalize_entity`, the variable "group" has the type "id" because it identifies another entity. Alternatively, it could be set using the `variable_types` parameter when we first call `es.entity_from_dataframe()`.

Transform Primitive

The `cum_sum` primitive calculates the running sum in list of numbers.

```
[56]: from featuretools.primitives import CumSum

cum_sum = CumSum()
cum_sum([1, 2, 3, 4, 5]).tolist()

[56]: [1, 3, 6, 10, 15]
```

If we apply it using the `trans_primitives` argument it will calculate it over the entire observations entity like this:

```
[57]: feature_matrix, feature_defs = ft.dfs(target_entity="observations",
                                          entityset=es,
                                          agg_primitives=[],
                                          trans_primitives=["cum_sum"],
                                          groupby_trans_primitives=[])

feature_matrix

[57]:   group  val  CUM_SUM(val)
id
1     a     5             5
2     b     1             6
3     a    10            16
4     c    20            36
5     a     6            42
6     b    23            65
```

Groupby Transform Primitive

If we apply it using `groupby_trans_primitives`, then DFS will first group by any id variables before applying the transform primitive. As a result, we get the cumulative sum by group.

```
[58]: feature_matrix, feature_defs = ft.dfs(target_entity="observations",
                                          entityset=es,
                                          agg_primitives=[],
                                          trans_primitives=[],
                                          groupby_trans_primitives=["cum_sum"])

feature_matrix

[58]:   group  val  CUM_SUM(val) by group
id
1     a     5             5.0
2     b     1             1.0
3     a    10            15.0
4     c    20            20.0
5     a     6            21.0
6     b    23            24.0
```

Aggregation Primitive

Finally, there is also the aggregation primitive “sum”. If we use `sum`, it will calculate the sum for the group at the cutoff time for each row. Because we didn’t specify a cutoff time it will use all the data for each group for each row.

```
[59]: feature_matrix, feature_defs = ft.dfs(target_entity="observations",
                                         entityset=es,
                                         agg_primitives=["sum"],
                                         trans_primitives=[],
                                         cutoff_time_in_index=True,
                                         groupby_trans_primitives=[])
```

feature_matrix

```
[59]:
```

	id	time	group	val	groups.SUM(observations.val)
	1	2020-02-01 01:40:08.439139	a	5	21
	2	2020-02-01 01:40:08.439139	b	1	24
	3	2020-02-01 01:40:08.439139	a	10	21
	4	2020-02-01 01:40:08.439139	c	20	20
	5	2020-02-01 01:40:08.439139	a	6	21
	6	2020-02-01 01:40:08.439139	b	23	24

If we set the cutoff time of each row to be the time index, then use sum as an aggregation primitive, the result is the same as cum_sum. (Though the order is different in the displayed dataframe).

```
[60]: cutoff_time = df[["id", "time_index"]]
      cutoff_time
```

```
[60]:
```

	id	time_index
0	1	2019-01-01
1	2	2019-01-02
2	3	2019-01-03
3	4	2019-01-04
4	5	2019-01-05
5	6	2019-01-06

```
[61]: feature_matrix, feature_defs = ft.dfs(target_entity="observations",
                                         entityset=es,
                                         agg_primitives=["sum"],
                                         trans_primitives=[],
                                         groupby_trans_primitives=[],
                                         cutoff_time_in_index=True,
                                         cutoff_time=cutoff_time)
```

feature_matrix

```
[61]:
```

	id	time	group	val	groups.SUM(observations.val)
	1	2019-01-01	a	5	5
	2	2019-01-02	b	1	1
	3	2019-01-03	a	10	15
	4	2019-01-04	c	20	20
	5	2019-01-05	a	6	21
	6	2019-01-06	b	23	24

How do I get a list of all Aggregation and Transform primitives?

You can do `featuretools.list_primitives()` to get all the primitive in Featuretools. It will return a Dataframe with the names, type, and description of the primitives.

```
[62]: df_primitives = ft.list_primitives()
df_primitives.head()

[62]:
```

	name	type	\
0	avg_time_between	aggregation	
1	percent_true	aggregation	
2	num_unique	aggregation	
3	entropy	aggregation	
4	all	aggregation	

	description
0	Computes the average number of seconds between...
1	Determines the percent of `True` values.
2	Determines the number of distinct values, igno...
3	Calculates the entropy for a categorical variable
4	Calculates if all values are 'True' in a list.

```
[63]: df_primitives.tail()

[63]:
```

	name	type	\
73	equal_scalar	transform	
74	haversine	transform	
75	cum_sum	transform	
76	divide_numeric	transform	
77	less_than_equal_to_scalar	transform	

	description
73	Determines if values in a list are equal to a ...
74	Calculates the approximate haversine distance ...
75	Calculates the cumulative sum.
76	Element-wise division of two lists.
77	Determines if values are less than or equal to...

How do I change the units for a TimeSince primitive?

There are a few primitives in Featuretools that make some time-based calculation. These include `TimeSince`, `TimeSincePrevious`, `TimeSinceLast`, `TimeSinceFirst`.

You can change the units from the default seconds to any valid time unit, by doing the following:

```
[64]: from featuretools.primitives import TimeSince, TimeSincePrevious, TimeSinceLast, \
↳ TimeSinceFirst

time_since = TimeSince(unit="minutes")
time_since_previous = TimeSincePrevious(unit="hours")
time_since_last = TimeSinceLast(unit="days")
time_since_first = TimeSinceFirst(unit="years")

es = ft.demo.load_mock_customer(return_entityset=True)

feature_matrix, feature_defs = ft.dfs(entityset=es,
                                     target_entity="customers",
                                     agg_primitives=[time_since_last, time_since_
↳ first],
                                     trans_primitives=[time_since, time_since_
↳ previous])
```

Above, we changed the units to the following: - minutes for TimeSince - hours for TimeSincePrevious - days for TimeSinceLast - years for TimeSinceFirst.

Now we can see that our feature matrix contains multiple features where the units for the TimeSince primitives are changed.

```
[65]: feature_matrix.head()
[65]: zip_code  TIME_SINCE_LAST(sessions.session_start, unit=days) \
customer_id
5          60091          2221.734769
4          60091          2221.846111
1          60091          2221.770880
3          13244          2221.705429
2          13244          2221.728750

          TIME_SINCE_FIRST(sessions.session_start, unit=years) \
customer_id
5          6.087057
4          6.087036
1          6.087005
3          6.086900
2          6.087090

          TIME_SINCE_LAST(transactions.transaction_time, unit=days) \
customer_id
5          2221.729503
4          2221.839341
1          2221.759595
3          2221.694144
2          2221.719723

          TIME_SINCE_FIRST(transactions.transaction_time, unit=years) \
customer_id
5          6.087057
4          6.087036
1          6.087005
3          6.086900
2          6.087090

          TIME_SINCE(join_date, unit=minutes) \
customer_id
5          5.019612e+06
4          4.637132e+06
1          4.624732e+06
3          4.454518e+06
2          4.099809e+06

          TIME_SINCE_PREVIOUS(join_date, unit=hours) \
customer_id
5          NaN
4          6374.673333
1          206.671944
3          2836.900278
2          5911.808333

          TIME_SINCE_LAST(transactions.sessions.session_start, unit=days) \
customer_id
5          2221.734769
```

(continues on next page)

(continued from previous page)

```

4                2221.846111
1                2221.770880
3                2221.705429
2                2221.728750

TIME_SINCE_FIRST(transactions.sessions.session_start, unit=years)
customer_id
5                6.087057
4                6.087036
1                6.087005
3                6.086900
2                6.087090

```

There are now features where time unit is different from the default of seconds, such as `TIME_SINCE_LAST(sessions.session_start, unit=days)`, and `TIME_SINCE_FIRST(sessions.session_start, unit=years)`.

3.12.5 Modeling

How does my train & test data work with Featuretools and sklearn's `train_test_split`?

You might be wondering how to properly use your train & test data with Featuretools, and sklearn's `train_test_split`. There are a few things you must do to ensure accuracy with this workflow.

Let's imagine we have a Dataframes for our train data, with the labels.

```
[66]: train_data = pd.DataFrame({"customer_id": [1, 2, 3, 4, 5],
                                "age": [20, 25, 55, 22, 35],
                                "gender": ["f", "m", "m", "m", "m"],
                                "signup_date": pd.date_range('2010-01-01 01:41:50',
                                ↪periods=5, freq='25min'),
                                "labels": [False, True, True, False, False]})
train_data.head()
[66]:   customer_id  age  gender      signup_date  labels
0             1   20     f 2010-01-01 01:41:50  False
1             2   25     m 2010-01-01 02:06:50   True
2             3   55     m 2010-01-01 02:31:50   True
3             4   22     m 2010-01-01 02:56:50  False
4             5   35     m 2010-01-01 03:21:50  False

```

Now we can create our `EntitySet` for the train data, and create our features. To prevent label leakage, we will use cutoff times (see *earlier question*).

```
[67]: es_train_data = ft.EntitySet(id="customer_data")
es_train_data = es_train_data.entity_from_dataframe(entity_id="customers",
                                                    dataframe=train_data,
                                                    index="customer_id")

cutoff_times = pd.DataFrame({"customer_id": [1, 2, 3, 4, 5],
                              "time": pd.date_range('2014-01-01 01:41:50', periods=5,
                              ↪freq='25min')})

feature_matrix_train, features = ft.dfs(entityset=es_train_data,
                                       target_entity="customers",

```

(continues on next page)

(continued from previous page)

```

cutoff_time=cutoff_times,
cutoff_time_in_index=True)
feature_matrix_train.head()
[67]:
customer_id time age gender labels DAY(signup_date) \
1 2014-01-01 01:41:50 20 f False 1
2 2014-01-01 02:06:50 25 m True 1
3 2014-01-01 02:31:50 55 m True 1
4 2014-01-01 02:56:50 22 m False 1
5 2014-01-01 03:21:50 35 m False 1

customer_id time YEAR(signup_date) MONTH(signup_date) \
1 2014-01-01 01:41:50 2010 1
2 2014-01-01 02:06:50 2010 1
3 2014-01-01 02:31:50 2010 1
4 2014-01-01 02:56:50 2010 1
5 2014-01-01 03:21:50 2010 1

customer_id time WEEKDAY(signup_date)
1 2014-01-01 01:41:50 4
2 2014-01-01 02:06:50 4
3 2014-01-01 02:31:50 4
4 2014-01-01 02:56:50 4
5 2014-01-01 03:21:50 4

```

We will also encode our feature matrix to compatible for machine learning algorithms.

```

[68]: feature_matrix_train_enc, feature_enc = ft.encode_features(feature_matrix_train,
↳ features)
feature_matrix_train_enc.head()
[68]:
customer_id time age gender = m gender = f \
1 2014-01-01 01:41:50 20 0 1
2 2014-01-01 02:06:50 25 1 0
3 2014-01-01 02:31:50 55 1 0
4 2014-01-01 02:56:50 22 1 0
5 2014-01-01 03:21:50 35 1 0

customer_id time gender is unknown labels \
1 2014-01-01 01:41:50 0 False
2 2014-01-01 02:06:50 0 True
3 2014-01-01 02:31:50 0 True
4 2014-01-01 02:56:50 0 False
5 2014-01-01 03:21:50 0 False

customer_id time DAY(signup_date) = 1 \
1 2014-01-01 01:41:50 1
2 2014-01-01 02:06:50 1
3 2014-01-01 02:31:50 1
4 2014-01-01 02:56:50 1
5 2014-01-01 03:21:50 1

```

(continues on next page)

(continued from previous page)

DAY(signup_date) is unknown \		
customer_id	time	
1	2014-01-01 01:41:50	0
2	2014-01-01 02:06:50	0
3	2014-01-01 02:31:50	0
4	2014-01-01 02:56:50	0
5	2014-01-01 03:21:50	0
YEAR(signup_date) = 2010 \		
customer_id	time	
1	2014-01-01 01:41:50	1
2	2014-01-01 02:06:50	1
3	2014-01-01 02:31:50	1
4	2014-01-01 02:56:50	1
5	2014-01-01 03:21:50	1
YEAR(signup_date) is unknown \		
customer_id	time	
1	2014-01-01 01:41:50	0
2	2014-01-01 02:06:50	0
3	2014-01-01 02:31:50	0
4	2014-01-01 02:56:50	0
5	2014-01-01 03:21:50	0
MONTH(signup_date) = 1 \		
customer_id	time	
1	2014-01-01 01:41:50	1
2	2014-01-01 02:06:50	1
3	2014-01-01 02:31:50	1
4	2014-01-01 02:56:50	1
5	2014-01-01 03:21:50	1
MONTH(signup_date) is unknown \		
customer_id	time	
1	2014-01-01 01:41:50	0
2	2014-01-01 02:06:50	0
3	2014-01-01 02:31:50	0
4	2014-01-01 02:56:50	0
5	2014-01-01 03:21:50	0
WEEKDAY(signup_date) = 4 \		
customer_id	time	
1	2014-01-01 01:41:50	1
2	2014-01-01 02:06:50	1
3	2014-01-01 02:31:50	1
4	2014-01-01 02:56:50	1
5	2014-01-01 03:21:50	1
WEEKDAY(signup_date) is unknown		
customer_id	time	
1	2014-01-01 01:41:50	0
2	2014-01-01 02:06:50	0
3	2014-01-01 02:31:50	0
4	2014-01-01 02:56:50	0
5	2014-01-01 03:21:50	0

```
[69]: from sklearn.model_selection import train_test_split

X = feature_matrix_train_enc.drop(['labels'], axis=1)
y = feature_matrix_train_enc['labels']

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.25)
```

Now you can use the encoded feature matrix with sklearn’s **train_test_split**. This will allow you to train your model, and tune your parameters.

How are categorical variables encoded when splitting training and testing data?

You might be wondering what happens when categorical variables are encoded with your training and testing data. You might be curious to know what happens if the train data has a categorical variable that is not present in the testing data.

Let’s explore a simple example to see what happens during the encoding process.

```
[70]: train_data = pd.DataFrame({"customer_id": [1, 2, 3, 4, 5],
                               "product_purchased": ["coke zero", "car", "toothpaste",
↪ "coke zero", "car"]})

es_train = ft.EntitySet(id="customer_data")
es_train = es_train.entity_from_dataframe(entity_id="customers",
                                         dataframe=train_data,
                                         index="customer_id")

feature_matrix_train, features = ft.dfs(entityset=es_train,
                                         target_entity='customers')
feature_matrix_train
```

```
[70]:      product_purchased
customer_id
1          coke zero
2              car
3        toothpaste
4          coke zero
5              car
```

We will use `ft.encode_features` to properly encode the `product_purchased` column.

```
[71]: feature_matrix_train_encoded, features_encoded = ft.encode_features(feature_matrix_
↪ train,
                                                                    features)
feature_matrix_train_encoded.head()
```

```
[71]:      product_purchased = coke zero  product_purchased = car \
customer_id
1              1              0
2              0              1
3              0              0
4              1              0
5              0              1

      product_purchased = toothpaste  product_purchased is unknown
customer_id
1              0              0
```

(continues on next page)

(continued from previous page)

2	0	0
3	1	0
4	0	0
5	0	0

Now lets imagine we have some test data that has doesn't have one of the categorical values (**toothpaste**). Also, the test data has a value that wasn't present in the train data (**water**).

```
[72]: test_data = pd.DataFrame({"customer_id": [6, 7, 8, 9, 10],
                               "product_purchased": ["coke zero", "car", "coke zero",
                               ↪ "coke zero", "water"]})

es_test = ft.EntitySet(id="customer_data")
es_test = es_test.entity_from_dataframe(entity_id="customers",
                                       dataframe=test_data,
                                       index="customer_id")

feature_matrix_test = ft.calculate_feature_matrix(entityset=es_test,
                                                  features=features_encoded)

feature_matrix_test.head()
```

```
[72]:      product_purchased = coke zero  product_purchased = car  \
customer_id
6                True                False
7                False               True
8                 True                False
9                 True                False
10               False                False

      product_purchased = toothpaste  product_purchased is unknown
customer_id
6                False                False
7                False                False
8                False                False
9                False                False
10               False                 True
```

As seen above, we were able to successfully handle the encoding, and deal with the following complications: - **toothpaste** was present in the training data but not present in the testing data - **water** was present in the test data but not present in the training data.

3.12.6 Errors & Warnings

Why am I getting this error 'Index is not unique on dataframe'?

You may be trying to create your EntitySet, and run into this error.

```
AssertionError: Index is not unique on dataframe
```

This is because each entity in your EntitySet needs a unique index.

Let's look at a simple example.

```
[73]: product_df = pd.DataFrame({'id': [1, 2, 3, 4, 4],
                               'rating': [3.5, 4.0, 4.5, 1.5, 5.0]})

product_df
```

```
[73]:
   id  rating
0    1    3.5
1    2    4.0
2    3    4.5
3    4    1.5
4    4    5.0
```

Notice how the `id` column has a duplicate index of 4. If you try to create an entity with this Dataframe, you will run into the following error.

```
es = ft.EntitySet(id="product_data")
es = es.entity_from_dataframe(entity_id="products",
                             dataframe=product_df,
                             index="id")
```

```
-----
AssertionError                                Traceback (most recent call last)
<ipython-input-63-a6e02ba6fa47> in <module>
      2 es = es.entity_from_dataframe(entity_id="products",
      3                               dataframe=product_df,
----> 4                               index="id")

~/featuretools/featuretools/entityset/entityset.py in entity_from_dataframe(self,
↳ entity_id, dataframe, index, variable_types, make_index, time_index, secondary_time_
↳ index, already_sorted)
    486         secondary_time_index=secondary_time_index,
    487         already_sorted=already_sorted,
--> 488         make_index=make_index)
    489         self.entity_dict[entity.id] = entity
    490         self.reset_data_description()

~/featuretools/featuretools/entityset/entity.py in __init__(self, id, df, entityset,
↳ variable_types, index, time_index, secondary_time_index, last_time_index, already_
↳ sorted, make_index, verbose)
    79
    80         self.df = df[[v.id for v in self.variables]]
--> 81         self.set_index(index)
    82
    83         self.time_index = None

~/featuretools/featuretools/entityset/entity.py in set_index(self, variable_id,
↳ unique)
    450         self.df.index.name = None
    451         if unique:
--> 452             assert self.df.index.is_unique, "Index is not unique on dataframe.
↳ (Entity {})".format(self.id)
    453
    454         self.convert_variable_type(variable_id, vtypes.Index, convert_
↳ data=False)

AssertionError: Index is not unique on dataframe (Entity products)
```

To fix the above error, you can do one of the following solutions:

Solution #1 - You can create a unique index on your Dataframe.

```
[74]: product_df = pd.DataFrame({'id': [1, 2, 3, 4, 5],
```

(continues on next page)

(continued from previous page)

```

                                'rating': [3.5, 4.0, 4.5, 1.5, 5.0])
product_df
[74]:   id  rating
      0    1    3.5
      1    2    4.0
      2    3    4.5
      3    4    1.5
      4    5    5.0

```

Notice how we now have a unique index column called `id`.

```

[75]: es = es.entity_from_dataframe(entity_id="products",
                                dataframe=product_df,
                                index="id")
es
[75]: Entityset: transactions
      Entities:
        transactions [Rows: 500, Columns: 5]
        products [Rows: 5, Columns: 2]
        sessions [Rows: 35, Columns: 4]
        customers [Rows: 5, Columns: 4]
      Relationships:
        transactions.product_id -> products.product_id
        transactions.session_id -> sessions.session_id
        sessions.customer_id -> customers.customer_id

```

As seen above, we can now create our entity for our `EntitySet` without an error by creating a unique index in our `Dataframe`.

Solution #2 - Set `make_index` to `True` in your call to `entity_from_dataframe` to create a new index on that data - `make_index` creates a unique index for each row by just looking at what number the row is, in relation to all the other rows.

```

[76]: product_df = pd.DataFrame({'id': [1, 2, 3, 4, 4],
                                'rating': [3.5, 4.0, 4.5, 1.5, 5.0]})

es = ft.EntitySet(id="product_data")
es = es.entity_from_dataframe(entity_id="products",
                              dataframe=product_df,
                              index="product_id",
                              make_index=True)

es['products'].df
[76]:   product_id  id  rating
      0         0    1    3.5
      1         1    2    4.0
      2         2    3    4.5
      3         3    4    1.5
      4         4    4    5.0

```

As seen above, we created our entity for our `EntitySet` without an error using the `make_index` argument.

Why am I getting the following warning 'Using training_window but last_time_index is not set'?

If you are using a training window, and you haven't set a `last_time_index` for your entity, you will get this warning. The training window attribute in Featuretools limits the amount of past data that can be used while calculating

a particular feature vector.

You can add the `last_time_index` to all entities automatically by calling `your_entityset.add_last_time_indexes()` after you create your `EntitySet`. This will remove the warning.

```
[77]: es = ft.demo.load_mock_customer(return_entityset=True)
      es.add_last_time_indexes()
```

Now we can run DFS without getting the warning.

```
[78]: cutoff_times = pd.DataFrame()
      cutoff_times['customer_id'] = [1, 2, 3, 1]
      cutoff_times['time'] = pd.to_datetime(['2014-1-1 04:00', '2014-1-1 05:00', '2014-1-1_
      ↪06:00', '2014-1-1 08:00'])
      cutoff_times['label'] = [True, True, False, True]

      feature_matrix, feature_defs = ft.dfs(entityset=es,
                                          target_entity="customers",
                                          cutoff_time=cutoff_times,
                                          cutoff_time_in_index=True,
                                          training_window="1 hour")
```

last_time_index vs. time_index

- The `time_index` is when the instance was first known.
- The `last_time_index` is when the instance appears for the last time.
- For example, a customer's session has multiple transactions which can happen at different points in time. If we are trying to count the number of sessions a user has in a given time period, we often want to count all the sessions that had any transaction during the training window. To accomplish this, we need to not only know when a session starts (**time_index**), but also when it ends (**last_time_index**). The last time that an instance appears in the data is stored as the `last_time_index` of an Entity.
- Once the `last_time_index` has been set, Featuretools will check to see if the `last_time_index` is after the start of the training window. That, combined with the cutoff time, allows DFS to discover which data is relevant for a given training window.

Why am I getting errors with Featuretools on Google Colab?

Google Colab, by default, has Featuretools 0.4.1 installed. You may run into issues following our newest guides, or latest documentation while using an older version of Featuretools. Therefore, we suggest you upgrade to the latest featuretools version by doing the following in your notebook in Google Colab:

```
!pip install -U featuretools
```

You may need to Restart the runtime by doing **Runtime -> Restart Runtime**. You can check latest Featuretools version by doing following:

```
import featuretools as ft
print(ft.__version__)
```

You should see a version greater than 0.4.1

3.13 Help

Couldn't find what you were looking for? The Featuretools community is happy to provide support to users of Featuretools.

3.13.1 Discussion

Conversation happens in the following places:

1. **General usage questions** are directed to [StackOverflow](#) with the #featuretools tag.
2. **Feature requests** can be made on the [Feature Request Board](#).
3. **Bug reports** are managed on the [GitHub issue tracker](#).
4. **Chat** and collaboration within the community occurs on [Slack](#). For general usage questions, please post on Stack Overflow where answers are more searchable by other users.
5. **Everything else, core developers** can be reached at help@featuretools.com.

3.13.2 Asking for help

All users levels, including beginners, should feel free to ask questions and report bugs when using featuretools. You can get better answers if follow a few simple guidelines:

1. **Use the right resource:** We suggest using Github or StackOverflow. Questions asked at these locations will be more searchable for other users.
 - Slack should be used for community discussion and collaboration.
 - For general questions on how something should work or tips, use StackOverflow.
 - Bugs should be reported on Github.
2. **Ask in one place only:** Please post your question in one place (StackOverflow or Github).
3. **Use examples:** Make [minimal, complete, verifiable examples](#). You will get much better answers if your provide code that people can use to reproduce your problem.
4. **Sharing data privately:** If you have data that you can't share publicly, feel free to send an email to help@featuretools.com.

3.13.3 Commercial support

Featuretools Enterprise offers expert support from the creators and core developers of Featuretools. Whether you need help getting a project off the ground or scaling Featuretools usage across your organization, we'll provide our expertise to help you build the best machine learning models possible. More information can be found [here](#).

3.14 Featuretools Enterprise

Featuretools Enterprise offers additional primitives, functionality, performance and expert support.

3.14.1 Premium Primitives

Feature primitives are the building blocks of Featuretools. They define individual computations that can be applied to raw datasets to create new features. Featuretools Enterprise contains over 100 domain-specific premium primitives to help you build better features for more accurate models.

3.14.2 Spark and Dask

Looking to easily scale Featuretools to bigger datasets or integrate it into your existing big data infrastructure? Whether it's on-premise or in the cloud, you can run Featuretools Enterprise with Apache Spark and Dask. We have yet to encounter a dataset that is too large to handle.

3.14.3 Expert Support

All subscriptions come with guidance from the creators and core developers of Featuretools. Whether you need help getting a project off the ground or scaling Featuretools usage across your organization, we'll provide our expertise to help you build the best machine learning models possible.

3.14.4 Early access

The team at Feature Labs is always working on the next big thing. With a subscription to Featuretools Enterprise, you can test and provide feedback on new tools and automation technologies before they are released.

Visit featurelabs.com to learn more about Featuretools Enterprise.

3.15 Limitations

3.15.1 In-memory

Featuretools is intended to be run on datasets that can fit in memory on one machine. For advice on handling large dataset refer to *Improving Computational Performance*.

If you would like to test [Feature Labs APIs](#) for running Featuretools natively on Apache Spark or Dask, please let us know [here](#).

3.15.2 Bring your own labels

If you are doing supervised machine learning, you must supply your own labels and cutoff times. To structure this process, you can use [Compose](#), which is an open source project for automatically generating labels with cutoff times.

3.16 Glossary

child entity An entity that references another entity via relationship. The “many” in a one-to-many relationship.

cutoff time The last point in time data is allowed to be used when calculating a feature

entity Equivalent to a table in relational database. Represented by the `Entity` class.

EntitySet A collection of entities and the relationships between them. Represented by the `EntitySet` class.

feature A transformation of data used for machine learning. featuretools has a custom language for defining features as described [here](#). All features are represented by subclasses of `FeatureBase`.

feature engineering The process of transforming data into representations that are better for machine learning.

instance Equivalent to a row in a relational database. Each entity has many instances, and each instance has a value for each variable and feature defined on the entity.

parent entity An entity that is referenced by another entity via relationship. The “one” in a one-to-many relationship.

relationship A mapping between a parent entity and a child entity. The child entity must contain a variable referencing the ID variable on the parent entity. Represented by the `Relationship` class.

target entity The entity on which we will be making a features for.

variable Equivalent to a column in a relational database. Represented by the `Variable` class.

3.17 Featuretools External Ecosystem

New projects are regularly being built on top of Featuretools, highlighting the importance of automated feature engineering. On this page, we have a list of libraries, use cases / demos, and tutorials that leverage Featuretools. If you would like to add a project, please contact us or submit a pull request on [GitHub](#).

Note: We are proud and excited to share the work of people using Featuretools, but we cannot endorse or provide support for the tools on this page.

3.17.1 Libraries

MLBlocks

- MLBlocks is a simple framework for composing end-to-end tunable Machine Learning Pipelines by seamlessly combining tools from any python library with a simple, common and uniform interface. MLBlocks contains a primitive that uses Featuretools.

Cardea

- Cardea is a machine learning library built on top of the FHIR data schema. It uses a number of **automl** tools, including Featuretools.

3.17.2 Demos & Use Cases

Predict customer lifetime value

- A common use case for machine learning is to predict customer lifetime value. This article walks through the importance of this prediction problem using Featuretools in the process.

Predict NHL playoff matches

- Many users of [Kaggle](#) are eager to use Featuretools to improve their model performance. In this blog post, a Kaggle user takes a dataset of plays from National Hockey League games and creates a model to predict if a game is a playoff match.

Predict poverty of households in Costa Rica

- Social programs have a difficult time determining the right people to give aid. Using a dataset of Costa Rican household characteristics, this Kaggle kernel predicts the poverty of households.

Predicting Functional Threshold Power (FTP)

- This notebook and accompanying report evaluates the use of machine learning for predicting a cyclist's FTP using data collected from previous training sessions. Featuretools is used to generate a set of independent variables that capture changes in performance over time.

Note: For more demos written by [Feature Labs](#), see featuretools.com/demos

3.17.3 Tutorials

Automated Feature Engineering in Python

- This article provides a walk-through of how to use a retail dataset with DFS.

A Hands-On Guide to Automated Feature Engineering

- A **in-depth** tutorial that works through using Featuretools to predict future product sales at “BigMart”.

Simple Automatic Feature Engineering

- A walk-through that applies Featuretools to a sample dataset and creates a classifier to predict clients who make large orders.

Introduction to Automated Feature Engineering Using DFS

- This article demonstrates using Featuretools helps automate the manual process of feature engineering on a dataset of home loans.

Automated Feature Engineering Workshop

- An automated feature engineering workshop using Featuretools hosted at the 2017 Data Summer Conference.

Tutorial in Japanese

- A tutorial of Featuretools that demonstrates integrating with the feature selection library [Boruta](#) and the hyper parameter tuning library [Optuna](#).

Building a Churn Prediction Model using Featuretools

- A video tutorial that shows how to build a churn prediction model using Featuretools along with Spark, XG-Boost, and Google Cloud Platform.

Automated Feature Engineering Workshop in Russian

- A video tutorial that shows how to predict if an applicant is capable of repaying a loan using Featuretools.

3.18 API Reference

3.18.1 Demo Datasets

<code>load_retail([id, nrows, return_single_table])</code>	Returns the retail entityset example.
<code>load_mock_customer([n_customers, ...])</code>	Return dataframes of mock customer data
<code>load_flight([month_filter, ...])</code>	Download, clean, and filter flight data from 2017.

featuretools.demo.load_retail

`featuretools.demo.load_retail(id='demo_retail_data', nrows=None, return_single_table=False)`

Returns the retail entityset example. The original dataset can be found [here](#).

We have also made some modifications to the data. We changed the column names, converted the `customer_id` to a unique fake `customer_name`, dropped duplicates, added columns for `total` and `cancelled` and converted amounts from GBP to USD. You can download the modified CSV in [gz compressed \(7 MB\)](#) or [uncompressed \(43 MB\)](#) formats.

Parameters

- **id** (*str*) – Id to assign to EntitySet.
- **nrows** (*int*) – Number of rows to load of the underlying CSV. If None, load all.
- **return_single_table** (*bool*) – If True, return a CSV rather than an EntitySet. Default is False.

Examples

```
In [1]: import featuretools as ft

In [2]: es = ft.demo.load_retail()

In [3]: es
Out[3]:
Entityset: demo_retail_data
Entities:
  orders (shape = [22190, 3])
  products (shape = [3684, 3])
  customers (shape = [4372, 2])
  order_products (shape = [401704, 7])
```

Load in subset of data

```
In [4]: es = ft.demo.load_retail(nrows=1000)
```

```
In [5]: es
```

```
Out [5]:
```

```
Entityset: demo_retail_data
Entities:
  orders (shape = [67, 5])
  products (shape = [606, 3])
  customers (shape = [50, 2])
  order_products (shape = [1000, 7])
```

featuretools.demo.load_mock_customer

```
featuretools.demo.load_mock_customer(n_customers=5, n_products=5, n_sessions=35,
                                     n_transactions=500, random_seed=0, return_single_table=False, return_entityset=False)
```

Return dataframes of mock customer data

featuretools.demo.load_flight

```
featuretools.demo.load_flight(month_filter=None, categorical_filter=None, nrows=None,
                              demo=True, return_single_table=False, verbose=False)
```

Download, clean, and filter flight data from 2017. The original dataset can be found [here](#).

Parameters

- **month_filter** (*list[int]*) – Only use data from these months (example is [1, 2]). To skip, set to None.
- **categorical_filter** (*dict[str->str]*) – Use only specified categorical values. Example is {'dest_city': ['Boston, MA'], 'origin_city': ['Boston, MA']} which returns all flights in OR out of Boston. To skip, set to None.
- **nrows** (*int*) – Passed to `nrows` in `pd.read_csv`. Used before filtering.
- **demo** (*bool*) – Use only two months of data. If False, use the whole year.
- **return_single_table** (*bool*) – Exit the function early and return a dataframe.
- **verbose** (*bool*) – Show a progress bar while loading the data.

Examples

```
In [1]: import featuretools as ft
```

```
In [2]: es = ft.demo.load_flight(verbose=True,
...:                           month_filter=[1],
...:                           categorical_filter={'origin_city':['Boston, MA']}
↪)
...:
100%|xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx| 100/100 [01:16<00:00, 1.31it/s]
```

```
In [3]: es
```

```
Out [3]:
```

```
Entityset: Flight Data
```

(continues on next page)

(continued from previous page)

```

Entities:
  airports [Rows: 55, Columns: 3]
  flights [Rows: 613, Columns: 9]
  trip_logs [Rows: 9456, Columns: 22]
  airlines [Rows: 10, Columns: 1]
Relationships:
  trip_logs.flight_id -> flights.flight_id
  flights.carrier -> airlines.carrier
  flights.dest -> airports.dest

```

3.18.2 Deep Feature Synthesis

<code>dfs(entities, relationships, entityset, ...)</code>	Calculates a feature matrix and features given a dictionary of entities and a list of relationships.
-----------------------------------------------------------	------------------------------------------------------------------------------------------------------

featuretools.dfs

`featuretools.dfs` (*entities=None, relationships=None, entityset=None, target_entity=None, cutoff_time=None, instance_ids=None, agg_primitives=None, trans_primitives=None, groupby_trans_primitives=None, allowed_paths=None, max_depth=2, ignore_entities=None, ignore_variables=None, primitive_options=None, seed_features=None, drop_contains=None, drop_exact=None, where_primitives=None, max_features=-1, cutoff_time_in_index=False, save_progress=None, features_only=False, training_window=None, approximate=None, chunk_size=None, n_jobs=1, dask_kwargs=None, verbose=False, return_variable_types=None, progress_callback=None*)

Calculates a feature matrix and features given a dictionary of entities and a list of relationships.

Parameters

- **entities** (*dict[str -> tuple(pd.DataFrame, str, str)]*) – Dictionary of entities. Entries take the format {entity id -> (dataframe, id column, (time_column))}.
- **relationships** (*list[(str, str, str, str)]*) – List of relationships between entities. List items are a tuple with the format (parent entity id, parent variable, child entity id, child variable).
- **entityset** (*EntitySet*) – An already initialized entityset. Required if entities and relationships are not defined.
- **target_entity** (*str*) – Entity id of entity on which to make predictions.
- **cutoff_time** (*pd.DataFrame or Datetime*) – Specifies times at which to calculate each instance. The resulting feature matrix will use data up to and including the cutoff_time. Can either be a DataFrame with ‘instance_id’ and ‘time’ columns, a DataFrame with the name of the index variable in the target entity and a time column, a list of values, or a single value to calculate for all instances. If the dataframe has more than two columns, any additional columns will be added to the resulting feature matrix.
- **instance_ids** (*list*) – List of instances on which to calculate features. Only used if cutoff_time is a single datetime.
- **agg_primitives** (*list[str or AggregationPrimitive], optional*) – List of Aggregation Feature types to apply.

Default: ["sum", "std", "max", "skew", "min", "mean", "count", "percent_true", "num_unique", "mode"]

- **trans_primitives** (*list[str or TransformPrimitive], optional*) – List of Transform Feature functions to apply.

Default: ["day", "year", "month", "weekday", "haversine", "num_words", "num_characters"]

- **groupby_trans_primitives** (*list[str or primitives.TransformPrimitive], optional*) – list of Transform primitives to make GroupByTransformFeatures with
- **allowed_paths** (*list[list[str]]*) – Allowed entity paths on which to make features.
- **max_depth** (*int*) – Maximum allowed depth of features.
- **ignore_entities** (*list[str], optional*) – List of entities to blacklist when creating features.
- **ignore_variables** (*dict[str -> list[str]], optional*) – List of specific variables within each entity to blacklist when creating features.

- **primitive_options** (*list[dict[str or tuple[str] -> dict] or dict[str or tuple[str] -> dict], optional*) – Specify options for a single primitive or a group of primitives. Lists of option dicts are used to specify options per input for primitives with multiple inputs. Each option dict can have the following keys:

"**include_entities**" List of entities to be included when creating features for the primitive(s). All other entities will be ignored (list[str]).

"**ignore_entities**" List of entities to be blacklisted when creating features for the primitive(s) (list[str]).

"**include_variables**" List of specific variables within each entity to include when creating feautres for the primitive(s). All other variables in a given entity will be ignored (dict[str -> list[str]]).

"**ignore_variables**" List of specific variables within each entity to blacklist when creating features for the primitive(s) (dict[str -> list[str]]).

"**include_groupby_entities**" List of Entities to be included when finding groupbys. All other entities will be ignored (list[str]).

"**ignore_groupby_entities**" List of entities to blacklist when finding groupbys (list[str]).

"**include_groupby_variables**" List of specific variables within each entity to include as groupbys, if applicable. All other variables in each entity will be ignored (dict[str -> list[str]]).

"**ignore_groupby_variables**" List of specific variables within each entity to blacklist as groupbys (dict[str -> list[str]]).

- **seed_features** (*list[FeatureBase]*) – List of manually defined features to use.
- **drop_contains** (*list[str], optional*) – Drop features that contains these strings in name.
- **drop_exact** (*list[str], optional*) – Drop features that exactly match these strings in name.

- **where_primitives** (*list[str or PrimitiveBase], optional*) – List of Primitives names (or types) to apply with where clauses.

Default:

[“count”]

- **max_features** (*int, optional*) – Cap the number of generated features to this number. If -1, no limit.
- **features_only** (*bool, optional*) – If True, returns the list of features without calculating the feature matrix.
- **cutoff_time_in_index** (*bool*) – If True, return a DataFrame with a MultiIndex where the second index is the cutoff time (first is instance id). DataFrame will be sorted by (time, instance_id).
- **training_window** (*Timedelta or str, optional*) – Window defining how much time before the cutoff time data can be used when calculating features. If None, all data before cutoff time is used. Defaults to None. Month and year units are not relative when Pandas Timedeltas are used. Relative units should be passed as a Featuretools Timedelta or a string.
- **approximate** (*Timedelta*) – Bucket size to group instances with similar cutoff times by for features with costly calculations. For example, if bucket is 24 hours, all instances with cutoff times on the same day will use the same calculation for expensive features.
- **save_progress** (*str, optional*) – Path to save intermediate computational results.
- **n_jobs** (*int, optional*) – number of parallel processes to use when calculating feature matrix
- **chunk_size** (*int or float or None or "cutoff time", optional*) – Number of rows of output feature matrix to calculate at time. If passed an integer greater than 0, will try to use that many rows per chunk. If passed a float value between 0 and 1 sets the chunk size to that percentage of all instances. If passed the string “cutoff time”, rows are split per cutoff time.
- **dask_kwargs** (*dict, optional*) – Dictionary of keyword arguments to be passed when creating the dask client and scheduler. Even if n_jobs is not set, using *dask_kwargs* will enable multiprocessing. Main parameters:

cluster (*str or dask.distributed.LocalCluster*): cluster or address of cluster to send tasks to. If unspecified, a cluster will be created.

diagnostics port (*int*): port number to use for web dashboard. If left unspecified, web interface will not be enabled.

Valid keyword arguments for LocalCluster will also be accepted.

- **return_variable_types** (*list[Variable] or str, optional*) – Types of variables to return. If None, default to Numeric, Discrete, and Boolean. If given as the string ‘all’, use all available variable types.
- **progress_callback** (*callable*) – function to be called with incremental progress updates. Has the following parameters:
 - update: percentage change (float between 0 and 100) in progress since last call
 - progress_percent: percentage (float between 0 and 100) of total computation completed
 - time_elapsed: total time in seconds that has elapsed since start of call

Examples

```

from featuretools.primitives import Mean
# cutoff times per instance
entities = {
    "sessions" : (session_df, "id"),
    "transactions" : (transactions_df, "id", "transaction_time")
}
relationships = [("sessions", "id", "transactions", "session_id")]
feature_matrix, features = dfs(entities=entities,
                               relationships=relationships,
                               target_entity="transactions",
                               cutoff_time=cutoff_times)

feature_matrix

features = dfs(entities=entities,
               relationships=relationships,
               target_entity="transactions",
               features_only=True)

```

3.18.3 Wrappers

Scikit-learn (BETA)

<i>wrappers.DFSTransformer</i> ([entities, ...])	Transformer using Scikit-Learn interface for Pipeline uses.
--------------------------------------------------	-------------------------------------------------------------

featuretools.wrappers.DFSTransformer

```

class featuretools.wrappers.DFSTransformer (entities=None,
                                             relationships=None,
                                             entityset=None,
                                             target_entity=None,
                                             agg_primitives=None,
                                             trans_primitives=None,
                                             allowed_paths=None,
                                             max_depth=2,
                                             ignore_entities=None,
                                             ignore_variables=None,
                                             seed_features=None,
                                             drop_contains=None,
                                             drop_exact=None,
                                             where_primitives=None,
                                             max_features=-1,
                                             verbose=False)

```

Transformer using Scikit-Learn interface for Pipeline uses.

```

__init__(entities=None,
         relationships=None,
         entityset=None,
         target_entity=None,
         agg_primitives=None,
         trans_primitives=None,
         allowed_paths=None,
         max_depth=2,
         ignore_entities=None,
         ignore_variables=None,
         seed_features=None,
         drop_contains=None,
         drop_exact=None,
         where_primitives=None,
         max_features=-1,
         verbose=False)

```

Creates Transformer

Parameters

- **entities** (*dict*[*str* -> *tuple*(*pd.DataFrame*, *str*, *str*)])
- **relationships** (*list*[(*str*, *str*, *str*, *str*)])

child entity id, child variable).

- **entityset** (`EntitySet`) – An already initialized entityset. Required if entities and relationships are not defined.
- **target_entity** (`str`) – Entity id of entity on which to make predictions.
- **agg_primitives** (`list[str or AggregationPrimitive], optional`) – List of Aggregation Feature types to apply.
 Default: [“sum”, “std”, “max”, “skew”, “min”, “mean”, “count”, “percent_true”, “num_unique”, “mode”]
- **trans_primitives** (`list[str or TransformPrimitive], optional`) – List of Transform Feature functions to apply.
 Default: [“day”, “year”, “month”, “weekday”, “haversine”, “num_words”, “num_characters”]
- **allowed_paths** (`list[list[str]]`) – Allowed entity paths on which to make features.
- **max_depth** (`int`) – Maximum allowed depth of features.
- **ignore_entities** (`list[str], optional`) – List of entities to blacklist when creating features.
- **ignore_variables** (`dict[str -> list[str]], optional`) – List of specific variables within each entity to blacklist when creating features.
- **seed_features** (`list[FeatureBase]`) – List of manually defined features to use.
- **drop_contains** (`list[str], optional`) – Drop features that contains these strings in name.
- **drop_exact** (`list[str], optional`) – Drop features that exactly match these strings in name.
- **where_primitives** (`list[str or PrimitiveBase], optional`) – List of Primitives names (or types) to apply with where clauses.
 Default:
 [“count”]
- **max_features** (`int, optional`) – Cap the number of generated features to this number. If -1, no limit.

Example

```
In [1]: import featuretools as ft
In [2]: import pandas as pd
In [3]: from featuretools.wrappers import DFSTransformer
In [4]: from sklearn.pipeline import Pipeline
In [5]: from sklearn.ensemble import ExtraTreesClassifier

# Get examle data
In [6]: n_customers = 3
```

(continues on next page)

(continued from previous page)

```
In [16]: pipeline.fit(ct.head(3), y=y)
Out [16]:
Pipeline(memory=None,
         steps=[('ft',
                <featuretools_sklearn_transformer.transformer.DFSTransformer_
↳object at 0x7f5b87a8f780>),
               ('et',
                ExtraTreesClassifier(bootstrap=False, ccp_alpha=0.0,
                                     class_weight=None, criterion='gini',
                                     max_depth=None, max_features='auto',
                                     max_leaf_nodes=None, max_samples=None,
                                     min_impurity_decrease=0.0,
                                     min_impurity_split=None,
                                     min_samples_leaf=1, min_samples_split=2,
                                     min_weight_fraction_leaf=0.0,
                                     n_estimators=100, n_jobs=None,
                                     oob_score=False, random_state=None,
                                     verbose=0, warm_start=False))],
         verbose=False)

In [17]: pipeline.predict_proba(ct.tail(2))
↳
array([[0.46, 0.54],
       [0.   , 1.   ]])

In [18]: pipeline.predict(ct.tail(2))
↳array([ True,  True])
```

Methods

<code>__init__</code> ([entities, relationships, ...])	Creates Transformer
<code>fit</code> (cutoff_time_ids[, y])	Wrapper for DFS
<code>fit_transform</code> (X[, y])	Fit to data, then transform it.
<code>get_params</code> ([deep])	
<code>transform</code> (cutoff_time_ids)	Wrapper for calculate_feature_matix

3.18.4 Timedelta

<code>Timedelta</code> (value[, unit, delta_obj])	Represents differences in time.
---------------------------------------------------	---------------------------------

featuretools.Timedelta

class featuretools.**Timedelta** (value, unit=None, delta_obj=None)
 Represents differences in time.

Timedeltas can be defined in multiple units. Supported units:

- “ms” : milliseconds
- “s” : seconds

- “h” : hours
- “m” : minutes
- “d” : days
- “o”/”observations” : number of individual events
- “mo” : months
- “Y” : years

Timedeltas can also be defined in terms of observations. In this case, the Timedelta represents the period spanned by *value*.

```
For observation timedeltas: >>> three_observations_log = Timedelta(3, "observations") >>>
three_observations_log.get_name() '3 Observations'
```

```
__init__(value, unit=None, delta_obj=None)
```

Parameters

- **value** (*float, str, dict*) – Value of timedelta, string providing both unit and value, or a dictionary of units and times.
- **unit** (*str*) – Unit of time delta.
- **delta_obj** (*pd.Timedelta or pd.DateOffset*) – A time object used internally to do time operations. If None is provided, one will be created using the provided value and unit.

Methods

```
__init__(value[, unit, delta_obj])
```

param value Value of timedelta, string providing

```
check_value(value, unit)
```

```
fix_units()
```

```
from_dictionary(dictionary)
```

```
get_arguments()
```

```
get_name()
```

```
get_unit_type()
```

```
get_units()
```

```
get_value([unit])
```

```
has_multiple_units()
```

```
has_no_observations()
```

```
is_absolute()
```

```
lower_readable_times()
```

```
make_singular(s)
```

3.18.5 Time utils

```
make_temporal_cutoffs(instance_ids, cutoffs)
```

Makes a set of equally spaced cutoff times prior to a set of input cutoffs and instance ids.

featuretools.make_temporal_cutoffs

`featuretools.make_temporal_cutoffs` (*instance_ids*, *cutoffs*, *window_size=None*, *num_windows=None*, *start=None*)

Makes a set of equally spaced cutoff times prior to a set of input cutoffs and instance ids.

If *window_size* and *num_windows* are provided, then *num_windows* of size *window_size* will be created prior to each cutoff time

If *window_size* and a *start* list is provided, then a variable number of windows will be created prior to each cutoff time, with the corresponding start time as the first cutoff.

If *num_windows* and a *start* list is provided, then *num_windows* of variable size will be created prior to each cutoff time, with the corresponding start time as the first cutoff

Parameters

- **instance_ids** (*list*, *np.ndarray*, or *pd.Series*) – list of instance ids. This function will make a new datetime series of multiple cutoff times for each value in this array.
- **cutoffs** (*list*, *np.ndarray*, or *pd.Series*) – list of datetime objects associated with each instance id. Each one of these will be the last time in the new datetime series for each instance id
- **window_size** (*pd.Timedelta*, *optional*) – amount of time between each date-time in each new cutoff series
- **num_windows** (*int*, *optional*) – number of windows in each new cutoff series
- **start** (*list*, *optional*) – list of start times for each instance id

3.18.6 Feature Primitives

Primitive Types

<code>TransformPrimitive()</code>	Feature for entity that is based off one or more other features in that entity.
<code>AggregationPrimitive()</code>	

featuretools.primitives.TransformPrimitive

class `featuretools.primitives.TransformPrimitive`

Feature for entity that is based off one or more other features in that entity.

`__init__()`

Initialize self. See `help(type(self))` for accurate signature.

Methods

<code>__init__()</code>	Initialize self.
<code>generate_name(base_feature_names)</code>	
<code>generate_names(base_feature_names)</code>	
<code>get_args_string()</code>	

Continued on next page

Table 9 – continued from previous page

get_arguments()	
get_filepath(filename)	
get_function()	
Attributes	
base_of	
base_of_exclude	
commutative	
default_value	
input_types	
max_stack_depth	
name	
number_output_features	
return_type	
uses_calc_time	
uses_full_entity	

featuretools.primitives.AggregationPrimitive

class featuretools.primitives.AggregationPrimitive

__init__()
 Initialize self. See help(type(self)) for accurate signature.

Methods

__init__()	Initialize self.
generate_name(base_feature_names, ...)	
generate_names(base_feature_names, ...)	
get_args_string()	
get_arguments()	
get_filepath(filename)	
get_function()	

Attributes

base_of
base_of_exclude
commutative
default_value
input_types
max_stack_depth
name
number_output_features
return_type

Continued on next page

Table 12 – continued from previous page

stack_on
stack_on_exclude
stack_on_self
uses_calc_time

Primitive Creation Functions

<code>make_agg_primitive(function, input_types, ...)</code>	Returns a new aggregation primitive class.
<code>make_trans_primitive(function, input_types, ...)</code>	Returns a new transform primitive class

featuretools.primitives.make_agg_primitive

`featuretools.primitives.make_agg_primitive` (*function*, *input_types*, *return_type*, *name=None*, *stack_on_self=True*, *stack_on=None*, *stack_on_exclude=None*, *base_of=None*, *base_of_exclude=None*, *description=None*, *cls_attributes=None*, *uses_calc_time=False*, *default_value=None*, *commutative=False*, *number_output_features=1*)

Returns a new aggregation primitive class. The primitive infers default values by passing in empty data.

Parameters

- **function** (*function*) – Function that takes in a series and applies some transformation to it.
- **input_types** (*list[Variable]*) – Variable types of the inputs.
- **return_type** (*Variable*) – Variable type of return.
- **name** (*str*) – Name of the function. If no name is provided, the name of *function* will be used.
- **stack_on_self** (*bool*) – Whether this primitive can be in *input_types* of self.
- **stack_on** (*list[PrimitiveBase]*) – Whitelist of primitives that can be in *input_types*.
- **stack_on_exclude** (*list[PrimitiveBase]*) – Blacklist of primitives that cannot be in *input_types*.
- **base_of** (*list[PrimitiveBase]*) – Whitelist of primitives that can have this primitive in *input_types*.
- **base_of_exclude** (*list[PrimitiveBase]*) – Blacklist of primitives that cannot have this primitive in *input_types*.
- **description** (*str*) – Description of primitive.
- **cls_attributes** (*dict[str -> anytype]*) – Custom attributes to be added to class. Key is attribute name, value is the attribute value.
- **uses_calc_time** (*bool*) – If True, the cutoff time the feature is being calculated at will be passed to the function as the keyword argument ‘time’.

- **default_value** (*Variable*) – Default value when creating the primitive to avoid the inference step. If no default value is provided, the inference happens.
- **commutative** (*bool*) – If True, will only make one feature per unique set of base features.
- **number_output_features** (*int*) – The number of output features (columns in the matrix) associated with this feature.

Example

```
In [1]: from featuretools.primitives import make_agg_primitive

In [2]: from featuretools.variable_types import DatetimeTimeIndex, Numeric

In [3]: def time_since_last(values, time=None):
...:     time_since = time - values.iloc[-1]
...:     return time_since.total_seconds()
...:

In [4]: TimeSinceLast = make_agg_primitive(
...:     function=time_since_last,
...:     input_types=[DatetimeTimeIndex],
...:     return_type=Numeric,
...:     description="Time since last related instance",
...:     uses_calc_time=True)
...:
```

featuretools.primitives.make_trans_primitive

featuretools.primitives.make_trans_primitive(*function*, *input_types*, *return_type*, *name=None*, *description=None*, *cls_attributes=None*, *uses_calc_time=False*, *commutative=False*, *number_output_features=1*)

Returns a new transform primitive class

Parameters

- **function** (*function*) – Function that takes in a series and applies some transformation to it.
- **input_types** (*list[Variable]*) – Variable types of the inputs.
- **return_type** (*Variable*) – Variable type of return.
- **name** (*str*) – Name of the primitive. If no name is provided, the name of *function* will be used.
- **description** (*str*) – Description of primitive.
- **cls_attributes** (*dict[str -> anytype]*) – Custom attributes to be added to class. Key is attribute name, value is the attribute value.
- **uses_calc_time** (*bool*) – If True, the cutoff time the feature is being calculated at will be passed to the function as the keyword argument ‘time’.
- **commutative** (*bool*) – If True, will only make one feature per unique set of base features.

- **number_output_features** (*int*) – The number of output features (columns in the matrix) associated with this feature.

Example

```
In [1]: from featuretools.primitives import make_trans_primitive

In [2]: from featuretools.variable_types import Variable, Boolean

In [3]: def pd_is_in(array, list_of_outputs=None):
...:     if list_of_outputs is None:
...:         list_of_outputs = []
...:     return pd.Series(array).isin(list_of_outputs)
...:

In [4]: def isin_generate_name(self):
...:     return u"%s.isin(%s)" % (self.base_features[0].get_name(),
...:                             str(self.kwarg['list_of_outputs']))
...:

In [5]: IsIn = make_trans_primitive(
...:     function=pd_is_in,
...:     input_types=[Variable],
...:     return_type=Boolean,
...:     name="is_in",
...:     description="For each value of the base feature, checks "
...:     "whether it is in a list that provided.",
...:     cls_attributes={"generate_name": isin_generate_name})
...:
```

Aggregation Primitives

<i>Count()</i>	Determines the total number of values, excluding <i>NaN</i> .
<i>Mean([skipna])</i>	Computes the average for a list of values.
<i>Sum()</i>	Calculates the total addition, ignoring <i>NaN</i> .
<i>Min()</i>	Calculates the smallest value, ignoring <i>NaN</i> values.
<i>Max()</i>	Calculates the highest value, ignoring <i>NaN</i> values.
<i>Std()</i>	Computes the dispersion relative to the mean value, ignoring <i>NaN</i> .
<i>Median()</i>	Determines the middlemost number in a list of values.
<i>Mode()</i>	Determines the most commonly repeated value.
<i>AvgTimeBetween([unit])</i>	Computes the average number of seconds between consecutive events.
<i>TimeSinceLast([unit])</i>	Calculates the time elapsed since the last datetime (default in seconds).
<i>TimeSinceFirst([unit])</i>	Calculates the time elapsed since the first datetime (in seconds).
<i>NumUnique()</i>	Determines the number of distinct values, ignoring <i>NaN</i> values.
<i>PercentTrue()</i>	Determines the percent of <i>True</i> values.
<i>All()</i>	Calculates if all values are 'True' in a list.
<i>Any()</i>	Determines if any value is 'True' in a list.

Continued on next page

Table 14 – continued from previous page

<i>First()</i>	Determines the first value in a list.
<i>Last()</i>	Determines the last value in a list.
<i>Skew()</i>	Computes the extent to which a distribution differs from a normal distribution.
<i>Trend()</i>	Calculates the trend of a variable over time.
<i>Entropy</i> ([dropna, base])	Calculates the entropy for a categorical variable

featuretools.primitives.Count

class featuretools.primitives.Count

Determines the total number of values, excluding *NaN*.

Examples

```
>>> count = Count()
>>> count([1, 2, 3, 4, 5, None])
5
```

__init__()

Initialize self. See help(type(self)) for accurate signature.

Methods

<i>__init__</i> ()	Initialize self.
<i>generate_name</i> (base_feature_names, ...)	
<i>generate_names</i> (base_feature_names, ...)	
<i>get_args_string</i> ()	
<i>get_arguments</i> ()	
<i>get_filepath</i> (filename)	
<i>get_function</i> ()	

Attributes

<i>base_of</i>
<i>base_of_exclude</i>
<i>commutative</i>
<i>default_value</i>
<i>input_types</i>
<i>max_stack_depth</i>
<i>name</i>
<i>number_output_features</i>
<i>stack_on</i>
<i>stack_on_exclude</i>
<i>stack_on_self</i>
<i>uses_calc_time</i>

featuretools.primitives.Mean

class featuretools.primitives.**Mean** (*skipna=True*)

Computes the average for a list of values.

Parameters **skipna** (*bool*) – Determines if to use NA/null values. Defaults to True to skip NA/null.

Examples

```
>>> mean = Mean()
>>> mean([1, 2, 3, 4, 5, None])
3.0
```

We can also control the way *NaN* values are handled.

```
>>> mean = Mean(skipna=False)
>>> mean([1, 2, 3, 4, 5, None])
nan
```

__init__ (*skipna=True*)

Initialize self. See help(type(self)) for accurate signature.

Methods

<code>__init__([skipna])</code>	Initialize self.
<code>generate_name(base_feature_names, ...)</code>	
<code>generate_names(base_feature_names, ...)</code>	
<code>get_args_string()</code>	
<code>get_arguments()</code>	
<code>get_filepath(filename)</code>	
<code>get_function()</code>	

Attributes

<code>base_of</code>
<code>base_of_exclude</code>
<code>commutative</code>
<code>default_value</code>
<code>input_types</code>
<code>max_stack_depth</code>
<code>name</code>
<code>number_output_features</code>
<code>stack_on</code>
<code>stack_on_exclude</code>
<code>stack_on_self</code>
<code>uses_calc_time</code>

featuretools.primitives.Sum

class featuretools.primitives.**Sum**
 Calculates the total addition, ignoring *NaN*.

Examples

```
>>> sum = Sum()
>>> sum([1, 2, 3, 4, 5, None])
15.0
```

__init__()
 Initialize self. See help(type(self)) for accurate signature.

Methods

<code>__init__()</code>	Initialize self.
<code>generate_name(base_feature_names, ...)</code>	
<code>generate_names(base_feature_names, ...)</code>	
<code>get_args_string()</code>	
<code>get_arguments()</code>	
<code>get_filepath(filename)</code>	
<code>get_function()</code>	

Attributes

<code>base_of</code>
<code>base_of_exclude</code>
<code>commutative</code>
<code>default_value</code>
<code>input_types</code>
<code>max_stack_depth</code>
<code>name</code>
<code>number_output_features</code>
<code>stack_on</code>
<code>stack_on_exclude</code>
<code>stack_on_self</code>
<code>uses_calc_time</code>

featuretools.primitives.Min

class featuretools.primitives.**Min**
 Calculates the smallest value, ignoring *NaN* values.

Examples

```
>>> min = Min()
>>> min([1, 2, 3, 4, 5, None])
1.0
```

`__init__()`

Initialize self. See help(type(self)) for accurate signature.

Methods

<code>__init__()</code>	Initialize self.
<code>generate_name(base_feature_names, ...)</code>	
<code>generate_names(base_feature_names, ...)</code>	
<code>get_args_string()</code>	
<code>get_arguments()</code>	
<code>get_filepath(filename)</code>	
<code>get_function()</code>	

Attributes

<code>base_of</code>
<code>base_of_exclude</code>
<code>commutative</code>
<code>default_value</code>
<code>input_types</code>
<code>max_stack_depth</code>
<code>name</code>
<code>number_output_features</code>
<code>stack_on</code>
<code>stack_on_exclude</code>
<code>stack_on_self</code>
<code>uses_calc_time</code>

featuretools.primitives.Max

class featuretools.primitives.Max

Calculates the highest value, ignoring *NaN* values.

Examples

```
>>> max = Max()
>>> max([1, 2, 3, 4, 5, None])
5.0
```

`__init__()`

Initialize self. See help(type(self)) for accurate signature.

Methods

<code>__init__()</code>	Initialize self.
<code>generate_name(base_feature_names, ...)</code>	
<code>generate_names(base_feature_names, ...)</code>	
<code>get_args_string()</code>	
<code>get_arguments()</code>	
<code>get_filepath(filename)</code>	
<code>get_function()</code>	

Attributes

<code>base_of</code>
<code>base_of_exclude</code>
<code>commutative</code>
<code>default_value</code>
<code>input_types</code>
<code>max_stack_depth</code>
<code>name</code>
<code>number_output_features</code>
<code>stack_on</code>
<code>stack_on_exclude</code>
<code>stack_on_self</code>
<code>uses_calc_time</code>

featuretools.primitives.Std

class featuretools.primitives.Std

Computes the dispersion relative to the mean value, ignoring *NaN*.

Examples

```
>>> std = Std()
>>> round(std([1, 2, 3, 4, 5, None]), 3)
1.414
```

`__init__()`
Initialize self. See `help(type(self))` for accurate signature.

Methods

<code>__init__()</code>	Initialize self.
<code>generate_name(base_feature_names, ...)</code>	
<code>generate_names(base_feature_names, ...)</code>	
<code>get_args_string()</code>	
<code>get_arguments()</code>	
<code>get_filepath(filename)</code>	
<code>get_function()</code>	

Attributes

base_of
base_of_exclude
commutative
default_value
input_types
max_stack_depth
name
number_output_features
stack_on
stack_on_exclude
stack_on_self
uses_calc_time

featuretools.primitives.Median

class featuretools.primitives.Median

Determines the middlemost number in a list of values.

Examples

```
>>> median = Median()
>>> median([5, 3, 2, 1, 4])
3.0
```

NaN values are ignored.

```
>>> median([5, 3, 2, 1, 4, None])
3.0
```

__init__()

Initialize self. See help(type(self)) for accurate signature.

Methods

<code>__init__()</code>	Initialize self.
<code>generate_name(base_feature_names, ...)</code>	
<code>generate_names(base_feature_names, ...)</code>	
<code>get_args_string()</code>	
<code>get_arguments()</code>	
<code>get_filepath(filename)</code>	
<code>get_function()</code>	

Attributes

base_of
base_of_exclude

Continued on next page

Table 28 – continued from previous page

commutative
default_value
input_types
max_stack_depth
name
number_output_features
stack_on
stack_on_exclude
stack_on_self
uses_calc_time

featuretools.primitives.Mode

class featuretools.primitives.Mode

Determines the most commonly repeated value.

Description: Given a list of values, return the value with the highest number of occurrences. If list is empty, return *NaN*.

Examples

```
>>> mode = Mode()
>>> mode(['red', 'blue', 'green', 'blue'])
'blue'
```

`__init__()`

Initialize self. See help(type(self)) for accurate signature.

Methods

<code>__init__()</code>	Initialize self.
<code>generate_name(base_feature_names, ...)</code>	
<code>generate_names(base_feature_names, ...)</code>	
<code>get_args_string()</code>	
<code>get_arguments()</code>	
<code>get_filepath(filename)</code>	
<code>get_function()</code>	

Attributes

base_of
base_of_exclude
commutative
default_value
input_types
max_stack_depth
name
number_output_features

Continued on next page

Table 30 – continued from previous page

return_type
stack_on
stack_on_exclude
stack_on_self
uses_calc_time

featuretools.primitives.AvgTimeBetween

class featuretools.primitives.AvgTimeBetween (*unit='seconds'*)

Computes the average number of seconds between consecutive events.

Description: Given a list of datetimes, return the average time (default in seconds) elapsed between consecutive events. If there are fewer than 2 non-null values, return *NaN*.

Parameters *unit* (*str*) – Defines the unit of time. Defaults to seconds. Acceptable values: years, months, days, hours, minutes, seconds, milliseconds, nanoseconds

Examples

```
>>> from datetime import datetime
>>> avg_time_between = AvgTimeBetween()
>>> times = [datetime(2010, 1, 1, 11, 45, 0),
...         datetime(2010, 1, 1, 11, 55, 15),
...         datetime(2010, 1, 1, 11, 57, 30)]
>>> avg_time_between(times)
375.0
>>> avg_time_between = AvgTimeBetween(unit="minutes")
>>> avg_time_between(times)
6.25
```

__init__ (*unit='seconds'*)

Initialize self. See help(type(self)) for accurate signature.

Methods

__init__ (<i>[unit]</i>)	Initialize self.
generate_name(<i>base_feature_names, ...</i>)	
generate_names(<i>base_feature_names, ...</i>)	
get_args_string()	
get_arguments()	
get_filepath(<i>filename</i>)	
get_function()	

Attributes

base_of
base_of_exclude
commutative
default_value

Continued on next page

Table 32 – continued from previous page

input_types
max_stack_depth
name
number_output_features
stack_on
stack_on_exclude
stack_on_self
uses_calc_time

featuretools.primitives.TimeSinceLast

class featuretools.primitives.**TimeSinceLast** (*unit='seconds'*)

Calculates the time elapsed since the last datetime (default in seconds).

Description: Given a list of datetimes, calculate the time elapsed since the last datetime (default in seconds). Uses the instance's cutoff time.

Parameters *unit* (*str*) – Defines the unit of time to count from. Defaults to seconds. Acceptable values: years, months, days, hours, minutes, seconds, milliseconds, nanoseconds

Examples

```
>>> from datetime import datetime
>>> time_since_last = TimeSinceLast()
>>> cutoff_time = datetime(2010, 1, 1, 12, 0, 0)
>>> times = [datetime(2010, 1, 1, 11, 45, 0),
...         datetime(2010, 1, 1, 11, 55, 15),
...         datetime(2010, 1, 1, 11, 57, 30)]
>>> time_since_last(times, time=cutoff_time)
150.0
```

```
>>> from datetime import datetime
>>> time_since_last = TimeSinceLast(unit = "minutes")
>>> cutoff_time = datetime(2010, 1, 1, 12, 0, 0)
>>> times = [datetime(2010, 1, 1, 11, 45, 0),
...         datetime(2010, 1, 1, 11, 55, 15),
...         datetime(2010, 1, 1, 11, 57, 30)]
>>> time_since_last(times, time=cutoff_time)
2.5
```

__init__ (*unit='seconds'*)

Initialize self. See help(type(self)) for accurate signature.

Methods

<code>__init__([unit])</code>	Initialize self.
<code>generate_name(base_feature_names, ...)</code>	
<code>generate_names(base_feature_names, ...)</code>	
<code>get_args_string()</code>	
<code>get_arguments()</code>	

Continued on next page

Table 33 – continued from previous page

get_filepath(filename)	
get_function()	
Attributes	
base_of	
base_of_exclude	
commutative	
default_value	
input_types	
max_stack_depth	
name	
number_output_features	
stack_on	
stack_on_exclude	
stack_on_self	
uses_calc_time	

featuretools.primitives.TimeSinceFirst

class featuretools.primitives.**TimeSinceFirst** (*unit='seconds'*)

Calculates the time elapsed since the first datetime (in seconds).

Description: Given a list of datetimes, calculate the time elapsed since the first datetime (in seconds). Uses the instance's cutoff time.

Parameters **unit** (*str*) – Defines the unit of time to count from. Defaults to seconds. Acceptable values: years, months, days, hours, minutes, seconds, milliseconds, nanoseconds

Examples

```
>>> from datetime import datetime
>>> time_since_first = TimeSinceFirst()
>>> cutoff_time = datetime(2010, 1, 1, 12, 0, 0)
>>> times = [datetime(2010, 1, 1, 11, 45, 0),
...         datetime(2010, 1, 1, 11, 55, 15),
...         datetime(2010, 1, 1, 11, 57, 30)]
>>> time_since_first(times, time=cutoff_time)
900.0
```

```
>>> from datetime import datetime
>>> time_since_first = TimeSinceFirst(unit = "minutes")
>>> cutoff_time = datetime(2010, 1, 1, 12, 0, 0)
>>> times = [datetime(2010, 1, 1, 11, 45, 0),
...         datetime(2010, 1, 1, 11, 55, 15),
...         datetime(2010, 1, 1, 11, 57, 30)]
>>> time_since_first(times, time=cutoff_time)
15.0
```

__init__ (*unit='seconds'*)

Initialize self. See help(type(self)) for accurate signature.

Methods

<code>__init__([unit])</code>	Initialize self.
<code>generate_name(base_feature_names, ...)</code>	
<code>generate_names(base_feature_names, ...)</code>	
<code>get_args_string()</code>	
<code>get_arguments()</code>	
<code>get_filepath(filename)</code>	
<code>get_function()</code>	

Attributes

<code>base_of</code>
<code>base_of_exclude</code>
<code>commutative</code>
<code>default_value</code>
<code>input_types</code>
<code>max_stack_depth</code>
<code>name</code>
<code>number_output_features</code>
<code>stack_on</code>
<code>stack_on_exclude</code>
<code>stack_on_self</code>
<code>uses_calc_time</code>

featuretools.primitives.NumUnique

class featuretools.primitives.NumUnique

Determines the number of distinct values, ignoring *NaN* values.

Examples

```
>>> num_unique = NumUnique()
>>> num_unique(['red', 'blue', 'green', 'yellow'])
4
```

NaN values will be ignored.

```
>>> num_unique(['red', 'blue', 'green', 'yellow', None])
4
```

`__init__()`
Initialize self. See help(type(self)) for accurate signature.

Methods

<code>__init__()</code>	Initialize self.
<code>generate_name(base_feature_names, ...)</code>	

Continued on next page

Table 37 – continued from previous page

<code>generate_names(base_feature_names, ...)</code>
<code>get_args_string()</code>
<code>get_arguments()</code>
<code>get_filepath(filename)</code>
<code>get_function()</code>

Attributes

<code>base_of</code>
<code>base_of_exclude</code>
<code>commutative</code>
<code>default_value</code>
<code>input_types</code>
<code>max_stack_depth</code>
<code>name</code>
<code>number_output_features</code>
<code>stack_on</code>
<code>stack_on_exclude</code>
<code>stack_on_self</code>
<code>uses_calc_time</code>

featuretools.primitives.PercentTrue

class featuretools.primitives.PercentTrue

Determines the percent of *True* values.

Description: Given a list of booleans, return the percent of values which are *True* as a decimal. *NaN* values are treated as *False*, adding to the denominator.

Examples

```
>>> percent_true = PercentTrue()
>>> percent_true([True, False, True, True, None])
0.6
```

`__init__()`

Initialize self. See `help(type(self))` for accurate signature.

Methods

<code>__init__()</code>	Initialize self.
<code>generate_name(base_feature_names, ...)</code>	
<code>generate_names(base_feature_names, ...)</code>	
<code>get_args_string()</code>	
<code>get_arguments()</code>	
<code>get_filepath(filename)</code>	
<code>get_function()</code>	

Attributes

base_of
base_of_exclude
commutative
default_value
input_types
max_stack_depth
name
number_output_features
stack_on
stack_on_exclude
stack_on_self
uses_calc_time

featuretools.primitives.All

class featuretools.primitives.All

Calculates if all values are 'True' in a list.

Description: Given a list of booleans, return *True* if all of the values are *True*.

Examples

```
>>> all = All()
>>> all([False, False, False, True])
False
```

__init__()

Initialize self. See help(type(self)) for accurate signature.

Methods

<code>__init__()</code>	Initialize self.
<code>generate_name(base_feature_names, ...)</code>	
<code>generate_names(base_feature_names, ...)</code>	
<code>get_args_string()</code>	
<code>get_arguments()</code>	
<code>get_filepath(filename)</code>	
<code>get_function()</code>	

Attributes

base_of
base_of_exclude
commutative
default_value
input_types

Continued on next page

Table 42 – continued from previous page

max_stack_depth
name
number_output_features
stack_on
stack_on_exclude
stack_on_self
uses_calc_time

featuretools.primitives.Any

class featuretools.primitives.Any

Determines if any value is 'True' in a list.

Description: Given a list of booleans, return *True* if one or more of the values are *True*.

Examples

```
>>> any = Any()
>>> any([False, False, False, True])
True
```

__init__()

Initialize self. See help(type(self)) for accurate signature.

Methods

<code>__init__()</code>	Initialize self.
<code>generate_name(base_feature_names, ...)</code>	
<code>generate_names(base_feature_names, ...)</code>	
<code>get_args_string()</code>	
<code>get_arguments()</code>	
<code>get_filepath(filename)</code>	
<code>get_function()</code>	

Attributes

base_of
base_of_exclude
commutative
default_value
input_types
max_stack_depth
name
number_output_features
stack_on
stack_on_exclude
stack_on_self
uses_calc_time

featuretools.primitives.First

class featuretools.primitives.**First**
Determines the first value in a list.

Examples

```
>>> first = First()
>>> first([1, 2, 3, 4, 5, None])
1.0
```

`__init__()`
Initialize self. See help(type(self)) for accurate signature.

Methods

<code>__init__()</code>	Initialize self.
<code>generate_name(base_feature_names, ...)</code>	
<code>generate_names(base_feature_names, ...)</code>	
<code>get_args_string()</code>	
<code>get_arguments()</code>	
<code>get_filepath(filename)</code>	
<code>get_function()</code>	

Attributes

<code>base_of</code>
<code>base_of_exclude</code>
<code>commutative</code>
<code>default_value</code>
<code>input_types</code>
<code>max_stack_depth</code>
<code>name</code>
<code>number_output_features</code>
<code>return_type</code>
<code>stack_on</code>
<code>stack_on_exclude</code>
<code>stack_on_self</code>
<code>uses_calc_time</code>

featuretools.primitives.Last

class featuretools.primitives.**Last**
Determines the last value in a list.

Examples

```
>>> last = Last()
>>> last([1, 2, 3, 4, 5, None])
nan
```

`__init__()`

Initialize self. See help(type(self)) for accurate signature.

Methods

<code>__init__()</code>	Initialize self.
<code>generate_name(base_feature_names, ...)</code>	
<code>generate_names(base_feature_names, ...)</code>	
<code>get_args_string()</code>	
<code>get_arguments()</code>	
<code>get_filepath(filename)</code>	
<code>get_function()</code>	

Attributes

<code>base_of</code>
<code>base_of_exclude</code>
<code>commutative</code>
<code>default_value</code>
<code>input_types</code>
<code>max_stack_depth</code>
<code>name</code>
<code>number_output_features</code>
<code>return_type</code>
<code>stack_on</code>
<code>stack_on_exclude</code>
<code>stack_on_self</code>
<code>uses_calc_time</code>

featuretools.primitives.Skew

class featuretools.primitives.Skew

Computes the extent to which a distribution differs from a normal distribution.

Description: For normally distributed data, the skewness should be about 0. A skewness value > 0 means that there is more weight in the left tail of the distribution.

Examples

```
>>> skew = Skew()
>>> skew([1, 10, 30, None])
1.0437603722639681
```

`__init__()`

Initialize self. See help(type(self)) for accurate signature.

Methods

<code>__init__()</code>	Initialize self.
<code>generate_name(base_feature_names, ...)</code>	
<code>generate_names(base_feature_names, ...)</code>	
<code>get_args_string()</code>	
<code>get_arguments()</code>	
<code>get_filepath(filename)</code>	
<code>get_function()</code>	

Attributes

<code>base_of</code>
<code>base_of_exclude</code>
<code>commutative</code>
<code>default_value</code>
<code>input_types</code>
<code>max_stack_depth</code>
<code>name</code>
<code>number_output_features</code>
<code>stack_on</code>
<code>stack_on_exclude</code>
<code>stack_on_self</code>
<code>uses_calc_time</code>

featuretools.primitives.Trend

class featuretools.primitives.Trend

Calculates the trend of a variable over time.

Description: Given a list of values and a corresponding list of datetimes, calculate the slope of the linear trend of values.

Examples

```
>>> from datetime import datetime
>>> trend = Trend()
>>> times = [datetime(2010, 1, 1, 11, 45, 0),
...         datetime(2010, 1, 1, 11, 55, 15),
...         datetime(2010, 1, 1, 11, 57, 30),
...         datetime(2010, 1, 1, 11, 12),
...         datetime(2010, 1, 1, 11, 12, 15)]
>>> round(trend([1, 2, 3, 4, 5], times), 3)
-0.053
```

`__init__()`

Initialize self. See help(type(self)) for accurate signature.

Methods

<code>__init__()</code>	Initialize self.
<code>generate_name(base_feature_names, ...)</code>	
<code>generate_names(base_feature_names, ...)</code>	
<code>get_args_string()</code>	
<code>get_arguments()</code>	
<code>get_filepath(filename)</code>	
<code>get_function()</code>	

Attributes

<code>base_of</code>
<code>base_of_exclude</code>
<code>commutative</code>
<code>default_value</code>
<code>input_types</code>
<code>max_stack_depth</code>
<code>name</code>
<code>number_output_features</code>
<code>stack_on</code>
<code>stack_on_exclude</code>
<code>stack_on_self</code>
<code>uses_calc_time</code>

featuretools.primitives.Entropy

class featuretools.primitives.**Entropy** (*dropna=False, base=None*)

Calculates the entropy for a categorical variable

Description: Given a list of observations from a categorical variable return the entropy of the distribution. NaN values can be treated as a category or dropped.

Parameters

- **dropna** (*bool*) – Whether to consider NaN values as a separate category Defaults to False.
- **base** (*float*) – The logarithmic base to use Defaults to e (natural logarithm)

Examples

```
>>> pd_entropy = Entropy()
>>> pd_entropy([1,2,3,4])
1.3862943611198906
```

`__init__` (*dropna=False, base=None*)
Initialize self. See help(type(self)) for accurate signature.

Methods

<code>__init__</code> ([dropna, base])	Initialize self.
<code>generate_name</code> (base_feature_names, ...)	
<code>generate_names</code> (base_feature_names, ...)	
<code>get_args_string</code> ()	
<code>get_arguments</code> ()	
<code>get_filepath</code> (filename)	
<code>get_function</code> ()	

Attributes

<code>base_of</code>
<code>base_of_exclude</code>
<code>commutative</code>
<code>default_value</code>
<code>input_types</code>
<code>max_stack_depth</code>
<code>name</code>
<code>number_output_features</code>
<code>stack_on</code>
<code>stack_on_exclude</code>
<code>stack_on_self</code>
<code>uses_calc_time</code>

Transform Primitives

Combine features

<code>IsIn</code> ([list_of_outputs])	Determines whether a value is present in a provided list.
<code>And</code> ()	Element-wise logical AND of two lists.
<code>Or</code> ()	Element-wise logical OR of two lists.
<code>Not</code> ()	Negates a boolean value.

featuretools.primitives.IsIn

class featuretools.primitives.IsIn (*list_of_outputs=None*)
 Determines whether a value is present in a provided list.

Examples

```
>>> items = ['string', 10.3, False]
>>> is_in = IsIn(list_of_outputs=items)
>>> is_in(['string', 10.5, False]).tolist()
[True, False, True]
```

`__init__` (*list_of_outputs=None*)
 Initialize self. See help(type(self)) for accurate signature.

Methods

<code>__init__([list_of_outputs])</code>	Initialize self.
<code>generate_name(base_feature_names)</code>	
<code>generate_names(base_feature_names)</code>	
<code>get_args_string()</code>	
<code>get_arguments()</code>	
<code>get_filepath(filename)</code>	
<code>get_function()</code>	

Attributes

<code>base_of</code>
<code>base_of_exclude</code>
<code>commutative</code>
<code>default_value</code>
<code>input_types</code>
<code>max_stack_depth</code>
<code>name</code>
<code>number_output_features</code>
<code>uses_calc_time</code>
<code>uses_full_entity</code>

featuretools.primitives.And

class featuretools.primitives.And

Element-wise logical AND of two lists.

Description: Given a list of booleans X and a list of booleans Y, determine whether each value in X is *True*, and whether its corresponding value in Y is also *True*.

Examples

```
>>> _and = And()
>>> _and([False, True, False], [True, True, False]).tolist()
[False, True, False]
```

`__init__()`
Initialize self. See `help(type(self))` for accurate signature.

Methods

<code>__init__()</code>	Initialize self.
<code>generate_name(base_feature_names)</code>	
<code>generate_names(base_feature_names)</code>	
<code>get_args_string()</code>	
<code>get_arguments()</code>	
<code>get_filepath(filename)</code>	

Continued on next page

Table 58 – continued from previous page

get_function()

Attributes

base_of

base_of_exclude

commutative

default_value

input_types

max_stack_depth

name

number_output_features

uses_calc_time

uses_full_entity

featuretools.primitives.Or

class featuretools.primitives.Or

Element-wise logical OR of two lists.

Description: Given a list of booleans X and a list of booleans Y, determine whether each value in X is *True*, or whether its corresponding value in Y is *True*.

Examples

```
>>> _or = Or()
>>> _or([False, True, False], [True, True, False]).tolist()
[True, True, False]
```

__init__()

Initialize self. See help(type(self)) for accurate signature.

Methods

<code>__init__()</code>	Initialize self.
-------------------------	------------------

generate_name(base_feature_names)

generate_names(base_feature_names)

get_args_string()

get_arguments()

get_filepath(filename)

get_function()

Attributes

base_of

base_of_exclude

commutative

Continued on next page

Table 61 – continued from previous page

default_value
input_types
max_stack_depth
name
number_output_features
uses_calc_time
uses_full_entity

featuretools.primitives.Not

class featuretools.primitives.**Not**

Negates a boolean value.

Examples

```
>>> not_func = Not()
>>> not_func([True, True, False]).tolist()
[False, False, True]
```

__init__()

Initialize self. See help(type(self)) for accurate signature.

Methods

__init__()	Initialize self.
generate_name(base_feature_names)	
generate_names(base_feature_names)	
get_args_string()	
get_arguments()	
get_filepath(filename)	
get_function()	

Attributes

base_of
base_of_exclude
commutative
default_value
input_types
max_stack_depth
name
number_output_features
uses_calc_time
uses_full_entity

General Transform Primitives

<i>Absolute()</i>	Computes the absolute value of a number.
<i>Percentile()</i>	Determines the percentile rank for each value in a list.
<i>TimeSince([unit])</i>	Calculates time from a value to a specified cutoff date-time.

featuretools.primitives.Absolute

class featuretools.primitives.**Absolute**

Computes the absolute value of a number.

Examples

```
>>> absolute = Absolute()
>>> absolute([3.0, -5.0, -2.4]).tolist()
[3.0, 5.0, 2.4]
```

__init__()

Initialize self. See help(type(self)) for accurate signature.

Methods

<i>__init__()</i>	Initialize self.
<i>generate_name(base_feature_names)</i>	
<i>generate_names(base_feature_names)</i>	
<i>get_args_string()</i>	
<i>get_arguments()</i>	
<i>get_filepath(filename)</i>	
<i>get_function()</i>	

Attributes

<i>base_of</i>
<i>base_of_exclude</i>
<i>commutative</i>
<i>default_value</i>
<i>input_types</i>
<i>max_stack_depth</i>
<i>name</i>
<i>number_output_features</i>
<i>uses_calc_time</i>
<i>uses_full_entity</i>

featuretools.primitives.Percentile

class featuretools.primitives.**Percentile**

Determines the percentile rank for each value in a list.

Examples

```
>>> percentile = Percentile()
>>> percentile([10, 15, 1, 20]).tolist()
[0.5, 0.75, 0.25, 1.0]
```

Nan values are ignored when determining rank

```
>>> percentile([10, 15, 1, None, 20]).tolist()
[0.5, 0.75, 0.25, nan, 1.0]
```

`__init__()`

Initialize self. See help(type(self)) for accurate signature.

Methods

<code>__init__()</code>	Initialize self.
<code>generate_name(base_feature_names)</code>	
<code>generate_names(base_feature_names)</code>	
<code>get_args_string()</code>	
<code>get_arguments()</code>	
<code>get_filepath(filename)</code>	
<code>get_function()</code>	

Attributes

<code>base_of</code>
<code>base_of_exclude</code>
<code>commutative</code>
<code>default_value</code>
<code>input_types</code>
<code>max_stack_depth</code>
<code>name</code>
<code>number_output_features</code>
<code>uses_calc_time</code>
<code>uses_full_entity</code>

featuretools.primitives.TimeSince

class featuretools.primitives.**TimeSince** (*unit='seconds'*)

Calculates time from a value to a specified cutoff datetime.

Parameters `unit` (*str*) – Defines the unit of time to count from. Defaults to Seconds. Acceptable values: years, months, days, hours, minutes, seconds, milliseconds, nanoseconds

Examples

```
>>> from datetime import datetime
>>> time_since = TimeSince()
```

(continues on next page)

(continued from previous page)

```
>>> times = [datetime(2019, 3, 1, 0, 0, 0, 1),
...          datetime(2019, 3, 1, 0, 0, 1, 0),
...          datetime(2019, 3, 1, 0, 2, 0, 0)]
>>> cutoff_time = datetime(2019, 3, 1, 0, 0, 0, 0)
>>> values = time_since(array=times, time=cutoff_time)
>>> list(map(int, values))
[0, -1, -120]
```

Change output to nanoseconds

```
>>> from datetime import datetime
>>> time_since_nano = TimeSince(unit='nanoseconds')
>>> times = [datetime(2019, 3, 1, 0, 0, 0, 1),
...          datetime(2019, 3, 1, 0, 0, 1, 0),
...          datetime(2019, 3, 1, 0, 2, 0, 0)]
>>> cutoff_time = datetime(2019, 3, 1, 0, 0, 0, 0)
>>> values = time_since_nano(array=times, time=cutoff_time)
>>> list(map(lambda x: int(round(x)), values))
[-1000, -10000000000, -1200000000000]
```

`__init__` (*unit='seconds'*)
 Initialize self. See help(type(self)) for accurate signature.

Methods

<code>__init__</code> ([unit])	Initialize self.
<code>generate_name</code> (base_feature_names)	
<code>generate_names</code> (base_feature_names)	
<code>get_args_string</code> ()	
<code>get_arguments</code> ()	
<code>get_filepath</code> (filename)	
<code>get_function</code> ()	

Attributes

<code>base_of</code>
<code>base_of_exclude</code>
<code>commutative</code>
<code>default_value</code>
<code>input_types</code>
<code>max_stack_depth</code>
<code>name</code>
<code>number_output_features</code>
<code>uses_calc_time</code>
<code>uses_full_entity</code>

Datetime Transform Primitives

<i>Second()</i>	Determines the seconds value of a datetime.
<i>Minute()</i>	Determines the minutes value of a datetime.
<i>Weekday()</i>	Determines the day of the week from a datetime.
<i>IsWeekend()</i>	Determines if a date falls on a weekend.
<i>Hour()</i>	Determines the hour value of a datetime.
<i>Day()</i>	Determines the day of the month from a datetime.
<i>Week()</i>	Determines the week of the year from a datetime.
<i>Month()</i>	Determines the month value of a datetime.
<i>Year()</i>	Determines the year value of a datetime.

featuretools.primitives.Second

class featuretools.primitives.Second

Determines the seconds value of a datetime.

Examples

```
>>> from datetime import datetime
>>> dates = [datetime(2019, 3, 1),
...          datetime(2019, 3, 3, 11, 10, 50),
...          datetime(2019, 3, 31, 19, 45, 15)]
>>> second = Second()
>>> second(dates).tolist()
[0, 50, 15]
```

__init__()

Initialize self. See help(type(self)) for accurate signature.

Methods

<i>__init__()</i>	Initialize self.
<i>generate_name(base_feature_names)</i>	
<i>generate_names(base_feature_names)</i>	
<i>get_args_string()</i>	
<i>get_arguments()</i>	
<i>get_filepath(filename)</i>	
<i>get_function()</i>	

Attributes

<i>base_of</i>
<i>base_of_exclude</i>
<i>commutative</i>
<i>default_value</i>
<i>input_types</i>
<i>max_stack_depth</i>
<i>name</i>
<i>number_output_features</i>

Continued on next page

Table 73 – continued from previous page

uses_calc_time
uses_full_entity

featuretools.primitives.Minute

class featuretools.primitives.Minute
 Determines the minutes value of a datetime.

Examples

```
>>> from datetime import datetime
>>> dates = [datetime(2019, 3, 1),
...          datetime(2019, 3, 3, 11, 10, 50),
...          datetime(2019, 3, 31, 19, 45, 15)]
>>> minute = Minute()
>>> minute(dates).tolist()
[0, 10, 45]
```

__init__()
 Initialize self. See help(type(self)) for accurate signature.

Methods

<code>__init__()</code>	Initialize self.
<code>generate_name(base_feature_names)</code>	
<code>generate_names(base_feature_names)</code>	
<code>get_args_string()</code>	
<code>get_arguments()</code>	
<code>get_filepath(filename)</code>	
<code>get_function()</code>	

Attributes

<code>base_of</code>
<code>base_of_exclude</code>
<code>commutative</code>
<code>default_value</code>
<code>input_types</code>
<code>max_stack_depth</code>
<code>name</code>
<code>number_output_features</code>
<code>uses_calc_time</code>
<code>uses_full_entity</code>

featuretools.primitives.Weekday

class featuretools.primitives.Weekday
 Determines the day of the week from a datetime.

Description: Returns the day of the week from a datetime value. Weeks start on Monday (day 0) and run through Sunday (day 6).

Examples

```
>>> from datetime import datetime
>>> dates = [datetime(2019, 3, 1),
...          datetime(2019, 6, 17, 11, 10, 50),
...          datetime(2019, 11, 30, 19, 45, 15)]
>>> weekday = Weekday()
>>> weekday(dates).tolist()
[4, 0, 5]
```

`__init__()`

Initialize self. See help(type(self)) for accurate signature.

Methods

<code>__init__()</code>	Initialize self.
<code>generate_name(base_feature_names)</code>	
<code>generate_names(base_feature_names)</code>	
<code>get_args_string()</code>	
<code>get_arguments()</code>	
<code>get_filepath(filename)</code>	
<code>get_function()</code>	

Attributes

<code>base_of</code>
<code>base_of_exclude</code>
<code>commutative</code>
<code>default_value</code>
<code>input_types</code>
<code>max_stack_depth</code>
<code>name</code>
<code>number_output_features</code>
<code>uses_calc_time</code>
<code>uses_full_entity</code>

featuretools.primitives.IsWeekend

class featuretools.primitives.IsWeekend

Determines if a date falls on a weekend.

Examples

```
>>> from datetime import datetime
>>> dates = [datetime(2019, 3, 1),
```

(continues on next page)

(continued from previous page)

```
...         datetime(2019, 6, 17, 11, 10, 50),
...         datetime(2019, 11, 30, 19, 45, 15)]
>>> is_weekend = IsWeekend()
>>> is_weekend(dates).tolist()
[False, False, True]
```

__init__()
Initialize self. See help(type(self)) for accurate signature.

Methods

<code>__init__()</code>	Initialize self.
<code>generate_name(base_feature_names)</code>	
<code>generate_names(base_feature_names)</code>	
<code>get_args_string()</code>	
<code>get_arguments()</code>	
<code>get_filepath(filename)</code>	
<code>get_function()</code>	

Attributes

<code>base_of</code>
<code>base_of_exclude</code>
<code>commutative</code>
<code>default_value</code>
<code>input_types</code>
<code>max_stack_depth</code>
<code>name</code>
<code>number_output_features</code>
<code>uses_calc_time</code>
<code>uses_full_entity</code>

featuretools.primitives.Hour

class featuretools.primitives.Hour
Determines the hour value of a datetime.

Examples

```
>>> from datetime import datetime
>>> dates = [datetime(2019, 3, 1),
...          datetime(2019, 3, 3, 11, 10, 50),
...          datetime(2019, 3, 31, 19, 45, 15)]
>>> hour = Hour()
>>> hour(dates).tolist()
[0, 11, 19]
```

__init__()
Initialize self. See help(type(self)) for accurate signature.

Methods

<code>__init__()</code>	Initialize self.
<code>generate_name(base_feature_names)</code>	
<code>generate_names(base_feature_names)</code>	
<code>get_args_string()</code>	
<code>get_arguments()</code>	
<code>get_filepath(filename)</code>	
<code>get_function()</code>	

Attributes

<code>base_of</code>
<code>base_of_exclude</code>
<code>commutative</code>
<code>default_value</code>
<code>input_types</code>
<code>max_stack_depth</code>
<code>name</code>
<code>number_output_features</code>
<code>uses_calc_time</code>
<code>uses_full_entity</code>

featuretools.primitives.Day

class featuretools.primitives.Day
Determines the day of the month from a datetime.

Examples

```
>>> from datetime import datetime
>>> dates = [datetime(2019, 3, 1),
...          datetime(2019, 3, 3),
...          datetime(2019, 3, 31)]
>>> day = Day()
>>> day(dates).tolist()
[1, 3, 31]
```

`__init__()`
Initialize self. See help(type(self)) for accurate signature.

Methods

<code>__init__()</code>	Initialize self.
<code>generate_name(base_feature_names)</code>	
<code>generate_names(base_feature_names)</code>	
<code>get_args_string()</code>	
<code>get_arguments()</code>	

Continued on next page

Table 82 – continued from previous page

get_filepath(filename)	
get_function()	
Attributes	
base_of	
base_of_exclude	
commutative	
default_value	
input_types	
max_stack_depth	
name	
number_output_features	
uses_calc_time	
uses_full_entity	

featuretools.primitives.Week

class featuretools.primitives.**Week**

Determines the week of the year from a datetime.

Description: Returns the week of the year from a datetime value. The first week of the year starts on January 1, and week numbers increment each Monday.

Examples

```
>>> from datetime import datetime
>>> dates = [datetime(2019, 1, 3),
...          datetime(2019, 6, 17, 11, 10, 50),
...          datetime(2019, 11, 30, 19, 45, 15)]
>>> week = Week()
>>> week(dates).tolist()
[1, 25, 48]
```

__init__()

Initialize self. See help(type(self)) for accurate signature.

Methods

<code>__init__()</code>	Initialize self.
<code>generate_name(base_feature_names)</code>	
<code>generate_names(base_feature_names)</code>	
<code>get_args_string()</code>	
<code>get_arguments()</code>	
<code>get_filepath(filename)</code>	
<code>get_function()</code>	

Attributes

base_of
base_of_exclude
commutative
default_value
input_types
max_stack_depth
name
number_output_features
uses_calc_time
uses_full_entity

featuretools.primitives.Month

class featuretools.primitives.**Month**
 Determines the month value of a datetime.

Examples

```
>>> from datetime import datetime
>>> dates = [datetime(2019, 3, 1),
...          datetime(2019, 6, 17, 11, 10, 50),
...          datetime(2019, 11, 30, 19, 45, 15)]
>>> month = Month()
>>> month(dates).tolist()
[3, 6, 11]
```

__init__()
 Initialize self. See help(type(self)) for accurate signature.

Methods

<code>__init__()</code>	Initialize self.
<code>generate_name(base_feature_names)</code>	
<code>generate_names(base_feature_names)</code>	
<code>get_args_string()</code>	
<code>get_arguments()</code>	
<code>get_filepath(filename)</code>	
<code>get_function()</code>	

Attributes

base_of
base_of_exclude
commutative
default_value
input_types

Continued on next page

Table 87 – continued from previous page

max_stack_depth
name
number_output_features
uses_calc_time
uses_full_entity

featuretools.primitives.Year

class featuretools.primitives.Year

Determines the year value of a datetime.

Examples

```
>>> from datetime import datetime
>>> dates = [datetime(2019, 3, 1),
...          datetime(2048, 6, 17, 11, 10, 50),
...          datetime(1950, 11, 30, 19, 45, 15)]
>>> year = Year()
>>> year(dates).tolist()
[2019, 2048, 1950]
```

__init__()

Initialize self. See help(type(self)) for accurate signature.

Methods

<code>__init__()</code>	Initialize self.
<code>generate_name(base_feature_names)</code>	
<code>generate_names(base_feature_names)</code>	
<code>get_args_string()</code>	
<code>get_arguments()</code>	
<code>get_filepath(filename)</code>	
<code>get_function()</code>	

Attributes

base_of
base_of_exclude
commutative
default_value
input_types
max_stack_depth
name
number_output_features
uses_calc_time
uses_full_entity

Cumulative Transform Primitives

<code>Diff()</code>	Compute the difference between the value in a list and the previous value in that list.
<code>TimeSincePrevious([unit])</code>	Compute the time since the previous entry in a list.
<code>CumCount()</code>	Calculates the cumulative count.
<code>CumSum()</code>	Calculates the cumulative sum.
<code>CumMean()</code>	Calculates the cumulative mean.
<code>CumMin()</code>	Calculates the cumulative minimum.
<code>CumMax()</code>	Calculates the cumulative maximum.

featuretools.primitives.Diff

class featuretools.primitives.Diff

Compute the difference between the value in a list and the previous value in that list.

Description: Given a list of values, compute the difference from the previous item in the list. The result for the first element of the list will always be *NaN*. If the values are datetimes, the output will be a *timedelta*.

Examples

```
>>> diff = Diff()
>>> values = [1, 10, 3, 4, 15]
>>> diff(values).tolist()
[nan, 9.0, -7.0, 1.0, 11.0]
```

`__init__()`

Initialize self. See `help(type(self))` for accurate signature.

Methods

<code>__init__()</code>	Initialize self.
<code>generate_name(base_feature_names)</code>	
<code>generate_names(base_feature_names)</code>	
<code>get_args_string()</code>	
<code>get_arguments()</code>	
<code>get_filepath(filename)</code>	
<code>get_function()</code>	

Attributes

<code>base_of</code>
<code>base_of_exclude</code>
<code>commutative</code>
<code>default_value</code>
<code>input_types</code>
<code>max_stack_depth</code>
<code>name</code>

Continued on next page

Table 92 – continued from previous page

number_output_features
uses_calc_time
uses_full_entity

featuretools.primitives.TimeSincePrevious

class featuretools.primitives.TimeSincePrevious (*unit='seconds'*)

Compute the time since the previous entry in a list.

Parameters *unit* (*str*) – Defines the unit of time to count from. Defaults to Seconds. Acceptable values: years, months, days, hours, minutes, seconds, milliseconds, nanoseconds

Description: Given a list of datetimes, compute the time in seconds elapsed since the previous item in the list. The result for the first item in the list will always be *NaN*.

Examples

```
>>> from datetime import datetime
>>> time_since_previous = TimeSincePrevious()
>>> dates = [datetime(2019, 3, 1, 0, 0, 0),
...          datetime(2019, 3, 1, 0, 2, 0),
...          datetime(2019, 3, 1, 0, 3, 0),
...          datetime(2019, 3, 1, 0, 2, 30),
...          datetime(2019, 3, 1, 0, 10, 0)]
>>> time_since_previous(dates).tolist()
[nan, 120.0, 60.0, -30.0, 450.0]
```

__init__ (*unit='seconds'*)

Initialize self. See help(type(self)) for accurate signature.

Methods

__init__ (<i>[unit]</i>)	Initialize self.
generate_name(<i>base_feature_names</i>)	
generate_names(<i>base_feature_names</i>)	
get_args_string()	
get_arguments()	
get_filepath(<i>filename</i>)	
get_function()	

Attributes

base_of
base_of_exclude
commutative
default_value
input_types
max_stack_depth
name

Continued on next page

Table 94 – continued from previous page

number_output_features
uses_calc_time
uses_full_entity

featuretools.primitives.CumCount**class** featuretools.primitives.CumCount

Calculates the cumulative count.

Description: Given a list of values, return the cumulative count (or running count). There is no set window, so the count at each point is calculated over all prior values. *NaN* values are counted.**Examples**

```
>>> cum_count = CumCount()
>>> cum_count([1, 2, 3, 4, None, 5]).tolist()
[1, 2, 3, 4, 5, 6]
```

__init__()

Initialize self. See help(type(self)) for accurate signature.

Methods

__init__()	Initialize self.
generate_name(base_feature_names)	
generate_names(base_feature_names)	
get_args_string()	
get_arguments()	
get_filepath(filename)	
get_function()	

Attributes

base_of
base_of_exclude
commutative
default_value
input_types
max_stack_depth
name
number_output_features
uses_calc_time
uses_full_entity

featuretools.primitives.CumSum**class** featuretools.primitives.CumSum

Calculates the cumulative sum.

Description: Given a list of values, return the cumulative sum (or running total). There is no set window, so the sum at each point is calculated over all prior values. *NaN* values will return *NaN*, but in the window of a cumulative calculation, they're ignored.

Examples

```
>>> cum_sum = CumSum()
>>> cum_sum([1, 2, 3, 4, None, 5]).tolist()
[1.0, 3.0, 6.0, 10.0, nan, 15.0]
```

`__init__()`

Initialize self. See help(type(self)) for accurate signature.

Methods

<code>__init__()</code>	Initialize self.
<code>generate_name(base_feature_names)</code>	
<code>generate_names(base_feature_names)</code>	
<code>get_args_string()</code>	
<code>get_arguments()</code>	
<code>get_filepath(filename)</code>	
<code>get_function()</code>	

Attributes

<code>base_of</code>
<code>base_of_exclude</code>
<code>commutative</code>
<code>default_value</code>
<code>input_types</code>
<code>max_stack_depth</code>
<code>name</code>
<code>number_output_features</code>
<code>uses_calc_time</code>
<code>uses_full_entity</code>

featuretools.primitives.CumMean

class featuretools.primitives.CumMean

Calculates the cumulative mean.

Description: Given a list of values, return the cumulative mean (or running mean). There is no set window, so the mean at each point is calculated over all prior values. *NaN* values will return *NaN*, but in the window of a cumulative calculation, they're treated as 0.

Examples

```
>>> cum_mean = CumMean()
>>> cum_mean([1, 2, 3, 4, None, 5]).tolist()
[1.0, 1.5, 2.0, 2.5, nan, 2.5]
```

`__init__()`

Initialize self. See help(type(self)) for accurate signature.

Methods

<code>__init__()</code>	Initialize self.
<code>generate_name(base_feature_names)</code>	
<code>generate_names(base_feature_names)</code>	
<code>get_args_string()</code>	
<code>get_arguments()</code>	
<code>get_filepath(filename)</code>	
<code>get_function()</code>	

Attributes

<code>base_of</code>
<code>base_of_exclude</code>
<code>commutative</code>
<code>default_value</code>
<code>input_types</code>
<code>max_stack_depth</code>
<code>name</code>
<code>number_output_features</code>
<code>uses_calc_time</code>
<code>uses_full_entity</code>

featuretools.primitives.CumMin

class featuretools.primitives.CumMin

Calculates the cumulative minimum.

Description: Given a list of values, return the cumulative min (or running min). There is no set window, so the min at each point is calculated over all prior values. *NaN* values will return *NaN*, but in the window of a cumulative calculation, they're ignored.

Examples

```
>>> cum_min = CumMin()
>>> cum_min([1, 2, -3, 4, None, 5]).tolist()
[1.0, 1.0, -3.0, -3.0, nan, -3.0]
```

`__init__()`

Initialize self. See help(type(self)) for accurate signature.

Methods

<code>__init__()</code>	Initialize self.
<code>generate_name(base_feature_names)</code>	
<code>generate_names(base_feature_names)</code>	
<code>get_args_string()</code>	
<code>get_arguments()</code>	
<code>get_filepath(filename)</code>	
<code>get_function()</code>	

Attributes

<code>base_of</code>
<code>base_of_exclude</code>
<code>commutative</code>
<code>default_value</code>
<code>input_types</code>
<code>max_stack_depth</code>
<code>name</code>
<code>number_output_features</code>
<code>uses_calc_time</code>
<code>uses_full_entity</code>

featuretools.primitives.CumMax

class featuretools.primitives.CumMax

Calculates the cumulative maximum.

Description: Given a list of values, return the cumulative max (or running max). There is no set window, so the max at each point is calculated over all prior values. *NaN* values will return *NaN*, but in the window of a cumulative calculation, they're ignored.

Examples

```
>>> cum_max = CumMax()
>>> cum_max([1, 2, 3, 4, None, 5]).tolist()
[1.0, 2.0, 3.0, 4.0, nan, 5.0]
```

`__init__()`
Initialize self. See `help(type(self))` for accurate signature.

Methods

<code>__init__()</code>	Initialize self.
<code>generate_name(base_feature_names)</code>	
<code>generate_names(base_feature_names)</code>	
<code>get_args_string()</code>	
<code>get_arguments()</code>	

Continued on next page

Table 103 – continued from previous page

<code>get_filepath(filename)</code>
<code>get_function()</code>

Attributes

<code>base_of</code>
<code>base_of_exclude</code>
<code>commutative</code>
<code>default_value</code>
<code>input_types</code>
<code>max_stack_depth</code>
<code>name</code>
<code>number_output_features</code>
<code>uses_calc_time</code>
<code>uses_full_entity</code>

Text Transform Primitives

<code>NumCharacters()</code>	Calculates the number of characters in a string.
<code>NumWords()</code>	Determines the number of words in a string by counting the spaces.

featuretools.primitives.NumCharacters

class featuretools.primitives.**NumCharacters**

Calculates the number of characters in a string.

Examples

```
>>> num_characters = NumCharacters()
>>> num_characters(['This is a string',
...                'second item',
...                'finall']).tolist()
[16, 11, 6]
```

__init__()
Initialize self. See `help(type(self))` for accurate signature.

Methods

<code>__init__()</code>	Initialize self.
<code>generate_name(base_feature_names)</code>	
<code>generate_names(base_feature_names)</code>	
<code>get_args_string()</code>	
<code>get_arguments()</code>	
<code>get_filepath(filename)</code>	
<code>get_function()</code>	

Attributes

base_of
base_of_exclude
commutative
default_value
input_types
max_stack_depth
name
number_output_features
uses_calc_time
uses_full_entity

featuretools.primitives.NumWords

class featuretools.primitives.NumWords

Determines the number of words in a string by counting the spaces.

Examples

```
>>> num_words = NumWords()
>>> num_words(['This is a string',
...           'Two words',
...           'no-spaces',
...           'Also works with sentences. Second sentence!']).tolist()
[4, 2, 1, 6]
```

__init__()

Initialize self. See help(type(self)) for accurate signature.

Methods

<code>__init__()</code>	Initialize self.
<code>generate_name(base_feature_names)</code>	
<code>generate_names(base_feature_names)</code>	
<code>get_args_string()</code>	
<code>get_arguments()</code>	
<code>get_filepath(filename)</code>	
<code>get_function()</code>	

Attributes

base_of
base_of_exclude
commutative
default_value
input_types
max_stack_depth

Continued on next page

Table 109 – continued from previous page

name
number_output_features
uses_calc_time
uses_full_entity

Location Transform Primitives

<i>Latitude()</i>	Returns the first tuple value in a list of LatLong tuples.
<i>Longitude()</i>	Returns the second tuple value in a list of LatLong tuples.
<i>Haversine([unit])</i>	Calculates the approximate haversine distance between two LatLong variable types.

featuretools.primitives.Latitude

class featuretools.primitives.Latitude

Returns the first tuple value in a list of LatLong tuples. For use with the LatLong variable type.

Examples

```
>>> latitude = Latitude()
>>> latitude([(42.4, -71.1),
...          (40.0, -122.4),
...          (41.2, -96.75)]).tolist()
[42.4, 40.0, 41.2]
```

__init__()
Initialize self. See help(type(self)) for accurate signature.

Methods

<i>__init__()</i>	Initialize self.
<i>generate_name(base_feature_names)</i>	
<i>generate_names(base_feature_names)</i>	
<i>get_args_string()</i>	
<i>get_arguments()</i>	
<i>get_filepath(filename)</i>	
<i>get_function()</i>	

Attributes

<i>base_of</i>
<i>base_of_exclude</i>
<i>commutative</i>
<i>default_value</i>
<i>input_types</i>

Continued on next page

Table 112 – continued from previous page

max_stack_depth
name
number_output_features
uses_calc_time
uses_full_entity

featuretools.primitives.Longitude

class featuretools.primitives.Longitude

Returns the second tuple value in a list of LatLong tuples. For use with the LatLong variable type.

Examples

```
>>> longitude = Longitude()
>>> longitude([(42.4, -71.1),
...           (40.0, -122.4),
...           (41.2, -96.75)]).tolist()
[-71.1, -122.4, -96.75]
```

__init__()

Initialize self. See help(type(self)) for accurate signature.

Methods

__init__()	Initialize self.
generate_name(base_feature_names)	
generate_names(base_feature_names)	
get_args_string()	
get_arguments()	
get_filepath(filename)	
get_function()	

Attributes

base_of
base_of_exclude
commutative
default_value
input_types
max_stack_depth
name
number_output_features
uses_calc_time
uses_full_entity

featuretools.primitives.Haversine**class** featuretools.primitives.**Haversine** (*unit='miles'*)

Calculates the approximate haversine distance between two LatLong variable types.

Parameters **unit** (*str*) – Determines the unit value to output. Could be *miles* or *kilometers*.
Default is *miles*.**Examples**

```
>>> haversine = Haversine()
>>> distances = haversine([(42.4, -71.1), (40.0, -122.4)],
...                       [(40.0, -122.4), (41.2, -96.75)])
>>> np.round(distances, 3).tolist()
[2631.231, 1343.289]
```

Output units can be specified

```
>>> haversine_km = Haversine(unit='kilometers')
>>> distances_km = haversine_km([(42.4, -71.1), (40.0, -122.4)],
...                             [(40.0, -122.4), (41.2, -96.75)])
>>> np.round(distances_km, 3).tolist()
[4234.555, 2161.814]
```

__init__ (*unit='miles'*)

Initialize self. See help(type(self)) for accurate signature.

Methods

__init__ (<i>[unit]</i>)	Initialize self.
generate_name (<i>base_feature_names</i>)	
generate_names (<i>base_feature_names</i>)	
get_args_string ()	
get_arguments ()	
get_filepath (<i>filename</i>)	
get_function ()	

Attributes

base_of
base_of_exclude
commutative
default_value
input_types
max_stack_depth
name
number_output_features
uses_calc_time
uses_full_entity

Natural Language Processing Primitives

Natural Language Processing primitives create features for textual data. For more information on how to use and install these primitives, see [here](#).

<code>DiversityScore()</code>	Calculates the overall complexity of the text based on the total
<code>LSA()</code>	Calculates the Latent Semantic Analysis Values of Text Input
<code>MeanCharactersPerWord()</code>	Determines the mean number of characters per word.
<code>PartOfSpeechCount()</code>	Calculates the occurrences of each different part of speech.
<code>PolarityScore()</code>	Calculates the polarity of a text on a scale from -1 (negative) to 1 (positive)
<code>PunctuationCount()</code>	Determines number of punctuation characters in a string.
<code>StopwordCount()</code>	Determines number of stopwords in a string.
<code>TitleWordCount()</code>	Determines the number of title words in a string.
<code>UniversalSentenceEncoder()</code>	Transforms a sentence or short paragraph to a vector using [tfhub model](https://tfhub.dev/google/universal-sentence-encoder/2)
<code>UpperCaseCount()</code>	Calculates the number of upper case letters in text.

`nlp_primitives.DiversityScore`

class `nlp_primitives.DiversityScore`

Calculates the overall complexity of the text based on the total number of words used in the text

Description: Given a list of strings, calculates the total number of unique words divided by the total number of words in order to give the text a score from 0-1 that indicates how unique the words used in it are. This primitive only evaluates the ‘clean’ versions of strings, so ignoring cases, punctuation, and stopwords in its evaluation.

If a string is missing, return *NaN*

Examples

```
>>> diversity_score = DiversityScore()
>>> diversity_score(["hi hi hi", "hello its me", "hey what hey what", "a dog ate_
↳ a basket"]).tolist()
[0.3333333333333333, 1.0, 0.5, 1.0]
```

`__init__()`

Initialize self. See `help(type(self))` for accurate signature.

Methods

<code>__init__()</code>	Initialize self.
<code>generate_name(base_feature_names)</code>	
<code>generate_names(base_feature_names)</code>	
<code>get_args_string()</code>	
<code>get_arguments()</code>	
<code>get_filepath(filename)</code>	
<code>get_function()</code>	

Attributes

<code>base_of</code>
<code>base_of_exclude</code>
<code>commutative</code>
<code>default_value</code>
<code>input_types</code>
<code>max_stack_depth</code>
<code>name</code>
<code>number_output_features</code>
<code>uses_calc_time</code>
<code>uses_full_entity</code>

nlp_primitives.LSA

class nlp_primitives.LSA

Calculates the Latent Semantic Analysis Values of Text Input

Description: Given a list of strings, transforms those strings using tf-idf and single value decomposition to go from a sparse matrix to a compact matrix with two values for each string. These values represent that Latent Semantic Analysis of each string. These values will represent their context with respect to (nlTK's brown sentence corpus.)[<https://www.nltk.org/book/ch02.html#brown-corpus>]

If a string is missing, return *NaN*.

Examples

```
>>> lsa = LSA()
>>> x = ["he helped her walk,", "me me me eat food", "the sentence doth long"]
>>> res = lsa(x).tolist()
>>> for i in range(len(res)): res[i] = [abs(round(x, 2)) for x in res[i]]
>>> res
[[0.0, 0.0, 0.01], [0.0, 0.0, 0.0]]
```

Now, if we change the values of the input corpus, to something that better resembles the given text, the same given input text will result in a different, more discerning, output. Also, NaN values are handled, as well as strings without words.

```
>>> lsa = LSA()
>>> x = ["the earth is round", "", np.NaN, ".,/"]
>>> res = lsa(x).tolist()
>>> for i in range(len(res)): res[i] = [abs(round(x, 2)) for x in res[i]]
>>> res
[[0.01, 0.0, nan, 0.0], [0.0, 0.0, nan, 0.0]]
```

`__init__()`
 Initialize self. See help(type(self)) for accurate signature.

Methods

<code>__init__()</code>	Initialize self.
<code>generate_name(base_feature_names)</code>	
<code>generate_names(base_feature_names)</code>	
<code>get_args_string()</code>	
<code>get_arguments()</code>	
<code>get_filepath(filename)</code>	
<code>get_function()</code>	

Attributes

<code>base_of</code>
<code>base_of_exclude</code>
<code>commutative</code>
<code>default_value</code>
<code>input_types</code>
<code>max_stack_depth</code>
<code>name</code>
<code>number_output_features</code>
<code>uses_calc_time</code>
<code>uses_full_entity</code>

nlp_primitives.MeanCharactersPerWord

class nlp_primitives.MeanCharactersPerWord

Determines the mean number of characters per word.

Description: Given list of strings, determine the mean number of characters per word in each string. A word is defined as a series of any characters not separated by white space. Punctuation is removed before counting. If a string is empty or *NaN*, return *NaN*.

Examples

```
>>> x = ['This is a test file', 'This is second line', 'third line $1,000']
>>> mean_characters_per_word = MeanCharactersPerWord()
>>> mean_characters_per_word(x).tolist()
[3.0, 4.0, 5.0]
```

`__init__()`
 Initialize self. See help(type(self)) for accurate signature.

Methods

<code>__init__()</code>	Initialize self.
<code>generate_name(base_feature_names)</code>	
<code>generate_names(base_feature_names)</code>	
<code>get_args_string()</code>	
<code>get_arguments()</code>	
<code>get_filepath(filename)</code>	
<code>get_function()</code>	

Attributes

<code>base_of</code>
<code>base_of_exclude</code>
<code>commutative</code>
<code>default_value</code>
<code>input_types</code>
<code>max_stack_depth</code>
<code>name</code>
<code>number_output_features</code>
<code>uses_calc_time</code>
<code>uses_full_entity</code>

nlp_primitives.PartOfSpeechCount

class nlp_primitives.PartOfSpeechCount

Calculates the occurrences of each different part of speech.

Description: Given a list of strings, categorize each word in the string as a different part of speech, and return the total count for each of 15 different categories of speech.

If a string is missing, return *NaN*.

Examples

```
>>> x = ['He was eating cheese', '']
>>> part_of_speech_count = PartOfSpeechCount()
>>> part_of_speech_count(x).tolist()
[[0.0, 0.0], [0.0, 0.0], [0.0, 0.0], [0.0, 0.0], [0.0, 0.0], [1.0, 0.0], [0.0, 0.0], [0.0, 0.0], [0.0, 0.0], [0.0, 0.0], [0.0, 0.0], [0.0, 0.0], [0.0, 0.0], [0.0, 0.0], [0.0, 0.0]]
```

`__init__()`
Initialize self. See `help(type(self))` for accurate signature.

Methods

<code>__init__()</code>	Initialize self.
<code>generate_name(base_feature_names)</code>	
<code>generate_names(base_feature_names)</code>	
<code>get_args_string()</code>	

Continued on next page

Table 125 – continued from previous page

<code>get_arguments()</code>
<code>get_filepath(filename)</code>
<code>get_function()</code>

Attributes

<code>base_of</code>
<code>base_of_exclude</code>
<code>commutative</code>
<code>default_value</code>
<code>input_types</code>
<code>max_stack_depth</code>
<code>name</code>
<code>number_output_features</code>
<code>uses_calc_time</code>
<code>uses_full_entity</code>

nlp_primitives.PolarityScore

class nlp_primitives.PolarityScore

Calculates the polarity of a text on a scale from -1 (negative) to 1 (positive)

Description: Given a list of strings assign a polarity score from -1 (negative text), to 0 (neutral text), to 1 (positive text). The functions returns a score for every given piece of text. If a string is missing, return 'NaN'

Examples

```
>>> x = ['He loves dogs', 'She hates cats', 'There is a dog', '']
>>> polarity_score = PolarityScore()
>>> polarity_score(x).tolist()
[0.677, -0.649, 0.0, 0.0]
```

`__init__()`

Initialize self. See help(type(self)) for accurate signature.

Methods

<code>__init__()</code>	Initialize self.
<code>generate_name(base_feature_names)</code>	
<code>generate_names(base_feature_names)</code>	
<code>get_args_string()</code>	
<code>get_arguments()</code>	
<code>get_filepath(filename)</code>	
<code>get_function()</code>	

Attributes

Table 130 – continued from previous page

input_types
max_stack_depth
name
number_output_features
uses_calc_time
uses_full_entity

nlp_primitives.StopwordCount

class nlp_primitives.**StopwordCount**

Determines number of stopwords in a string.

Description: Given list of strings, determine the number of stopwords characters in each string. Looks for any of the English stopwords defined in *nltk.corpus.stopwords*. Case insensitive.

If a string is missing, return *NaN*.

Examples

```
>>> x = ['This is a test string.', 'This is second string', 'third string']
>>> stopwords_count = StopwordCount()
>>> stopwords_count(x).tolist()
[3.0, 2.0, 0.0]
```

__init__()

Initialize self. See help(type(self)) for accurate signature.

Methods

__init__()	Initialize self.
generate_name(base_feature_names)	
generate_names(base_feature_names)	
get_args_string()	
get_arguments()	
get_filepath(filename)	
get_function()	

Attributes

base_of
base_of_exclude
commutative
default_value
input_types
max_stack_depth
name
number_output_features
uses_calc_time
uses_full_entity

nlp_primitives.TitleWordCount

class nlp_primitives.TitleWordCount

Determines the number of title words in a string.

Description: Given list of strings, determine the number of title words in each string. A title word is defined as any word starting with a capital letter. Words at the start of a sentence will be counted.

If a string is missing, return *NaN*.

Examples

```
>>> x = ['My favorite movie is Jaws.', 'this is a string', 'AAA']
>>> title_word_count = TitleWordCount()
>>> title_word_count(x).tolist()
[2.0, 0.0, 1.0]
```

__init__()

Initialize self. See help(type(self)) for accurate signature.

Methods

<code>__init__()</code>	Initialize self.
<code>generate_name(base_feature_names)</code>	
<code>generate_names(base_feature_names)</code>	
<code>get_args_string()</code>	
<code>get_arguments()</code>	
<code>get_filepath(filename)</code>	
<code>get_function()</code>	

Attributes

<code>base_of</code>
<code>base_of_exclude</code>
<code>commutative</code>
<code>default_value</code>
<code>input_types</code>
<code>max_stack_depth</code>
<code>name</code>
<code>number_output_features</code>
<code>uses_calc_time</code>
<code>uses_full_entity</code>

nlp_primitives.UniversalSentenceEncoder

class nlp_primitives.UniversalSentenceEncoder

Transforms a sentence or short paragraph to a vector using [tfhub model](<https://tfhub.dev/google/universal-sentence-encoder/2>)

Parameters None –

Examples

```

>>> universal_sentence_encoder = UniversalSentenceEncoder()
>>> sentences = ["I like to eat pizza",
...             "The roller coaster was built in 1885.",
...             ""]
>>> output = universal_sentence_encoder(sentences)
>>> len(output)
512
>>> len(output[0])
3
>>> values = output[:3, 0]
>>> [round(x, 4) for x in values]
[0.0178, 0.0616, -0.0089]

```

`__init__()`

Initialize self. See help(type(self)) for accurate signature.

Methods

<code>__init__()</code>	Initialize self.
<code>generate_name(base_feature_names)</code>	
<code>generate_names(base_feature_names)</code>	
<code>get_args_string()</code>	
<code>get_arguments()</code>	
<code>get_filepath(filename)</code>	
<code>get_function()</code>	

Attributes

<code>base_of</code>
<code>base_of_exclude</code>
<code>commutative</code>
<code>default_value</code>
<code>input_types</code>
<code>max_stack_depth</code>
<code>name</code>
<code>number_output_features</code>
<code>uses_calc_time</code>
<code>uses_full_entity</code>

nlp_primitives.UpperCaseCount

class nlp_primitives.UpperCaseCount

Calculates the number of upper case letters in text.

Description: Given a list of strings, determine the number of characters in each string that are capitalized. Counts every letter individually, not just every word that contains capitalized letters.

If a string is missing, return *NaN*

Examples

```
>>> x = ['This IS a string.', 'This is a string', 'aaa']
>>> upper_case_count = UpperCaseCount()
>>> upper_case_count(x).tolist()
[3.0, 1.0, 0.0]
```

`__init__()`

Initialize self. See help(type(self)) for accurate signature.

Methods

<code>__init__()</code>	Initialize self.
<code>generate_name(base_feature_names)</code>	
<code>generate_names(base_feature_names)</code>	
<code>get_args_string()</code>	
<code>get_arguments()</code>	
<code>get_filepath(filename)</code>	
<code>get_function()</code>	

Attributes

<code>base_of</code>
<code>base_of_exclude</code>
<code>commutative</code>
<code>default_value</code>
<code>input_types</code>
<code>max_stack_depth</code>
<code>name</code>
<code>number_output_features</code>
<code>uses_calc_time</code>
<code>uses_full_entity</code>

Feature methods

<code>FeatureBase.rename(name)</code>	Rename Feature, returns copy
<code>FeatureBase.get_depth([stop_at])</code>	Returns depth of feature

featuretools.feature_base.FeatureBase.rename

FeatureBase.**rename** (*name*)
 Rename Feature, returns copy

featuretools.feature_base.FeatureBase.get_depth

FeatureBase.**get_depth** (*stop_at=None*)
 Returns depth of feature

3.18.7 Feature calculation

<code>calculate_feature_matrix(features[,...])</code>	Calculates a matrix for a given set of instance ids and calculation times.
-------------------------------------------------------	----------------------------------------------------------------------------

featuretools.calculate_feature_matrix

```
featuretools.calculate_feature_matrix(features, entityset=None, cutoff_time=None,
                                     instance_ids=None, entities=None, relationships=None,
                                     cutoff_time_in_index=False, training_window=None,
                                     approximate=None, save_progress=None, verbose=False,
                                     chunk_size=None, n_jobs=1, dask_kwargs=None,
                                     progress_callback=None)
```

Calculates a matrix for a given set of instance ids and calculation times.

Parameters

- **features** (list[FeatureBase]) – Feature definitions to be calculated.
- **entityset** (EntitySet) – An already initialized entityset. Required if *entities* and *relationships* not provided
- **cutoff_time** (pd.DataFrame or Datetime) – Specifies at which time to calculate the features for each instance. The resulting feature matrix will use data up to and including the cutoff_time. Can either be a DataFrame with ‘instance_id’ and ‘time’ columns, DataFrame with the name of the index variable in the target entity and a time column, or a single value to calculate for all instances. If the dataframe has more than two columns, any additional columns will be added to the resulting feature matrix.
- **instance_ids** (list) – List of instances to calculate features on. Only used if cutoff_time is a single datetime.
- **entities** (dict[str -> tuple(pd.DataFrame, str, str)]) – dictionary of entities. Entries take the format {entity id: (dataframe, id column, (time_column))}.
- **relationships** (list[(str, str, str, str)]) – list of relationships between entities. List items are a tuple with the format (parent entity id, parent variable, child entity id, child variable).
- **cutoff_time_in_index** (bool) – If True, return a DataFrame with a MultiIndex where the second index is the cutoff time (first is instance id). DataFrame will be sorted by (time, instance_id).
- **training_window** (Timedelta or str, optional) – Window defining how much time before the cutoff time data can be used when calculating features. If None, all data before cutoff time is used. Defaults to None.
- **approximate** (Timedelta or str) – Frequency to group instances with similar cutoff times by for features with costly calculations. For example, if bucket is 24 hours, all instances with cutoff times on the same day will use the same calculation for expensive features.
- **verbose** (bool, optional) – Print progress info. The time granularity is per chunk.
- **chunk_size** (int or float or None) – maximum number of rows of output feature matrix to calculate at time. If passed an integer greater than 0, will try to use that

many rows per chunk. If passed a float value between 0 and 1 sets the chunk size to that percentage of all rows. if None, and `n_jobs > 1` it will be set to `1/n_jobs`

- **n_jobs** (*int, optional*) – number of parallel processes to use when calculating feature matrix.
- **dask_kwargs** (*dict, optional*) – Dictionary of keyword arguments to be passed when creating the dask client and scheduler. Even if `n_jobs` is not set, using `dask_kwargs` will enable multiprocessing. Main parameters:

cluster (*str or dask.distributed.LocalCluster*): cluster or address of cluster to send tasks to. If unspecified, a cluster will be created.

diagnostics port (*int*): port number to use for web dashboard. If left unspecified, web interface will not be enabled.

Valid keyword arguments for `LocalCluster` will also be accepted.

- **save_progress** (*str, optional*) – path to save intermediate computational results.
- **progress_callback** (*callable*) – function to be called with incremental progress updates. Has the following parameters:

update: percentage change (float between 0 and 100) in progress since last call

progress_percent: percentage (float between 0 and 100) of total computation

completed time_elapsed: total time in seconds that has elapsed since start of call

3.18.8 Feature encoding

`encode_features(feature_matrix, features[, ...])` Encode categorical features

featuretools.encode_features

`featuretools.encode_features(feature_matrix, features, top_n=10, include_unknown=True, to_encode=None, inplace=False, drop_first=False, verbose=False)`

Encode categorical features

Parameters

- **feature_matrix** (*pd.DataFrame*) – Dataframe of features.
- **features** (*list[PrimitiveBase]*) – Feature definitions in `feature_matrix`.
- **top_n** (*int or dict[string -> int]*) – Number of top values to include. If `dict[string -> int]` is used, key is feature name and value is the number of top values to include for that feature. If a feature's name is not in dictionary, a default value of 10 is used.
- **include_unknown** (*pd.DataFrame*) – Add feature encoding an unknown class. defaults to True
- **to_encode** (*list[str]*) – List of feature names to encode. features not in this list are unencoded in the output matrix defaults to encode all necessary features.
- **inplace** (*bool*) – Encode `feature_matrix` in place. Defaults to False.
- **drop_first** (*bool*) – Whether to get k-1 dummies out of k categorical levels by removing the first level. defaults to False

- **verbose** (*str*) – Print progress info.

Returns encoded feature_matrix, encoded features

Return type (pd.DataFrame, list)

Example

```
In [1]: f1 = ft.Feature(es["log"]["product_id"])

In [2]: f2 = ft.Feature(es["log"]["purchased"])

In [3]: f3 = ft.Feature(es["log"]["value"])

In [4]: features = [f1, f2, f3]

In [5]: ids = [0, 1, 2, 3, 4, 5]

In [6]: feature_matrix = ft.calculate_feature_matrix(features, es,
...:                                     instance_ids=ids)
...:

In [7]: fm_encoded, f_encoded = ft.encode_features(feature_matrix,
...:                                     features)
...:

In [8]: f_encoded
Out[8]:
[<Feature: product_id = coke zero>,
 <Feature: product_id = car>,
 <Feature: product_id = toothpaste>,
 <Feature: product_id is unknown>,
 <Feature: purchased>,
 <Feature: value>]

In [9]: fm_encoded, f_encoded = ft.encode_features(feature_matrix,
...:                                     features, top_n=2)
...:

In [10]: f_encoded
Out[10]:
[<Feature: product_id = coke zero>,
 <Feature: product_id = car>,
 <Feature: product_id is unknown>,
 <Feature: purchased>,
 <Feature: value>]

In [11]: fm_encoded, f_encoded = ft.encode_features(feature_matrix, features,
...:                                     include_unknown=False)
...:

In [12]: f_encoded
Out[12]:
[<Feature: product_id = coke zero>,
 <Feature: product_id = car>,
 <Feature: product_id = toothpaste>,
 <Feature: purchased>,
```

(continues on next page)

(continued from previous page)

```

<Feature: value>]

In [13]: fm_encoded, f_encoded = ft.encode_features(feature_matrix, features,
.....:                                             to_encode=['purchased'])
.....:

In [14]: f_encoded
Out[14]: [<Feature: product_id>, <Feature: purchased>, <Feature: value>]

In [15]: fm_encoded, f_encoded = ft.encode_features(feature_matrix, features,
.....:                                             drop_first=True)
.....:

In [16]: f_encoded
Out[16]:
[<Feature: product_id = coke zero>,
 <Feature: product_id = car>,
 <Feature: product_id is unknown>,
 <Feature: purchased>,
 <Feature: value>]

```

3.18.9 Saving and Loading Features

<code>save_features(features[, location, profile_name])</code>	Saves the features list as JSON to a specified filepath/S3 path, writes to an open file, or returns the serialized features as a JSON string.
<code>load_features(features[, profile_name])</code>	Loads the features from a filepath, S3 path, URL, an open file, or a JSON formatted string.

featuretools.save_features

`featuretools.save_features` (*features*, *location=None*, *profile_name=None*)

Saves the features list as JSON to a specified filepath/S3 path, writes to an open file, or returns the serialized features as a JSON string. If no file provided, returns a string.

Parameters

- **features** (list[FeatureBase]) – List of Feature definitions.
- **location** (str or FileObject, optional) – The location of where to save the features list which must include the name of the file, or a writeable file handle to write to. If location is None, will return a JSON string of the serialized features. Default: None
- **profile_name** (*str*, *bool*) – The AWS profile specified to write to S3. Will default to None and search for AWS credentials. Set to False to use an anonymous profile.

Note: Features saved in one version of Featuretools are not guaranteed to work in another. After upgrading Featuretools, features may need to be generated again.

Example

```
f1 = ft.Feature(es["log"]["product_id"])
f2 = ft.Feature(es["log"]["purchased"])
f3 = ft.Feature(es["log"]["value"])

features = [f1, f2, f3]

filepath = os.path.join('/Home/features/', 'list.json')
ft.save_features(features, filepath)

f = open(filepath, 'w')
ft.save_features(features, f)

features_str = ft.save_features(features)
```

See also:

`load_features()`

featuretools.load_features

`featuretools.load_features(features, profile_name=None)`

Loads the features from a filepath, S3 path, URL, an open file, or a JSON formatted string.

Parameters

- **features** (str or FileObject) – The location of where features has
- **saved which this must include the name of the file, or a JSON formatted (been) –**
- **or a readable file handle where the features have been saved. (string,) –**
- **profile_name** (str, bool) – The AWS profile specified to write to S3. Will default to None and search for AWS credentials. Set to False to use an anonymous profile.

Returns Feature definitions list.

Return type features (list[FeatureBase])

Note: Features saved in one version of Featuretools or python are not guaranteed to work in another. After upgrading Featuretools or python, features may need to be generated again.

Example

```
filepath = os.path.join('/Home/features/', 'list.json')
ft.load_features(filepath)

f = open(filepath, 'r')
ft.load_features(f)

feature_str = f.read()
ft.load_features(feature_str)
```

See also:

`save_features()`

3.18.10 EntitySet, Entity, Relationship, Variable Types

Constructors

<code>EntitySet([id, entities, relationships])</code>	Stores all actual data for a entityset
<code>Entity(id, df, entityset[, variable_types, ...])</code>	Represents an entity in a Entityset, and stores relevant metadata and data
<code>Relationship(parent_variable, child_variable)</code>	Class to represent an relationship between entities

featuretools.EntitySet

class featuretools.**EntitySet** (*id=None, entities=None, relationships=None*)

Stores all actual data for a entityset

id

entity_dict

relationships

time_type

Properties: metadata

__init__ (*id=None, entities=None, relationships=None*)

Creates EntitySet

Parameters

- **id** (*str*) – Unique identifier to associate with this instance
- **entities** (*dict[str -> tuple(pd.DataFrame, str, str)]*) – Dictionary of entities. Entries take the format {entity id -> (dataframe, id column, (time_column), (variable_types))}. Note that time_column and variable_types are optional.
- **relationships** (*list[(str, str, str, str)]*) – List of relationships between entities. List items are a tuple with the format (parent entity id, parent variable, child entity id, child variable).

Example

```
entities = {
    "cards" : (card_df, "id"),
    "transactions" : (transactions_df, "id", "transaction_time")
}

relationships = [("cards", "id", "transactions", "card_id")]

ft.EntitySet("my-entity-set", entities, relationships)
```

Methods

<code>__init__([id, entities, relationships])</code>	Creates EntitySet
<code>add_interesting_values([max_values, verbose])</code>	Find interesting values for categorical variables, to be used to generate “where” clauses
<code>add_last_time_indexes([updated_entities])</code>	Calculates the last time index values for each entity (the last time an instance or children of that instance were observed).
<code>add_relationship(relationship)</code>	Add a new relationship between entities in the entityset
<code>add_relationships(relationships)</code>	Add multiple new relationships to a entityset
<code>concat(other[, inplace])</code>	Combine entityset with another to create a new entityset with the combined data of both entitysets.
<code>entity_from_dataframe(entity_id, dataframe)</code>	Load the data for a specified entity from a Pandas DataFrame.
<code>find_backward_paths(start_entity_id, ...)</code>	Generator which yields all backward paths between a start and goal entity.
<code>find_forward_paths(start_entity_id, ...)</code>	Generator which yields all forward paths between a start and goal entity.
<code>get_backward_entities(entity_id[, deep])</code>	Get entities that are in a backward relationship with entity
<code>get_backward_relationships(entity_id)</code>	get relationships where entity “entity_id” is the parent.
<code>get_forward_entities(entity_id[, deep])</code>	Get entities that are in a forward relationship with entity
<code>get_forward_relationships(entity_id)</code>	Get relationships where entity “entity_id” is the child
<code>has_unique_forward_path(start_entity_id, ...)</code>	Is the forward path from start to end unique?
<code>normalize_entity(base_entity_id, ...[, ...])</code>	Create a new entity and relationship from unique values of an existing variable.
<code>plot([to_file])</code>	Create a UML diagram-ish graph of the EntitySet.
<code>reset_data_description()</code>	
<code>to_csv(path[, sep, encoding, engine, ...])</code>	Write entityset to disk in the csv format, location specified by <i>path</i> .
<code>to_dictionary()</code>	
<code>to_parquet(path[, engine, compression, ...])</code>	Write entityset to disk in the parquet format, location specified by <i>path</i> .
<code>to_pickle(path[, compression, profile_name])</code>	Write entityset in the pickle format, location specified by <i>path</i> .

Attributes

<code>entities</code>	
<code>metadata</code>	Returns the metadata for this EntitySet.

featuretools.Entity

class featuretools.**Entity** (*id, df, entityset, variable_types=None, index=None, time_index=None, secondary_time_index=None, last_time_index=None, already_sorted=False, make_index=False, verbose=False*)

Represents an entity in a Entityset, and stores relevant metadata and data

An Entity is analogous to a table in a relational database

See also:

Relationship, *Variable*, *EntitySet*

```
__init__(id, df, entityset, variable_types=None, index=None, time_index=None, secondary_time_index=None, last_time_index=None, already_sorted=False, make_index=False, verbose=False)
Create Entity
```

Parameters

- **id** (*str*) – Id of Entity.
- **df** (*pd.DataFrame*) – Dataframe providing the data for the entity.
- **entityset** (*EntitySet*) – Entityset for this Entity.
- **variable_types** (*dict[str -> dict[str -> type]]*) – An entity’s variable_types dict maps string variable ids to types (*Variable*) or (*type*, *kwargs*) to pass keyword arguments to the *Variable*.
- **index** (*str*) – Name of id column in the dataframe.
- **time_index** (*str*) – Name of time column in the dataframe.
- **secondary_time_index** (*dict[str -> str]*) – Dictionary mapping columns in the dataframe to the time index column they are associated with.
- **last_time_index** (*pd.Series*) – Time index of the last event for each instance across all child entities.
- **make_index** (*bool, optional*) – If True, assume index does not exist as a column in dataframe, and create a new column of that name using integers the (0, len(dataframe)). Otherwise, assume index exists in dataframe.

Methods

<code>__init__(id, df, entityset[, ...])</code>	Create Entity
<code>add_interesting_values([max_values, verbose])</code>	Find interesting values for categorical variables, to be used to
<code>convert_variable_type(variable_id, new_type)</code>	Convert variable in dataframe to different type
<code>delete_variables(variable_ids)</code>	Remove variables from entity’s dataframe and from self.variables
<code>query_by_values(instance_vals[, ...])</code>	Query instances that have variable with given value
<code>set_index(variable_id[, unique])</code>	param variable_id Name of an existing variable to set as index.
<code>set_secondary_time_index(secondary_time_index)</code>	
<code>set_time_index(variable_id[, already_sorted])</code>	
<code>update_data(df[, already_sorted, ...])</code>	Update entity’s internal dataframe, optionally making sure data is sorted, reference indexes to other entities are consistent, and last_time_indexes are consistent.

Attributes

<code>df</code>	Dataframe providing the data for the entity.
<code>last_time_index</code>	Time index of the last event for each instance across all child entities.
<code>shape</code>	Shape of the entity's dataframe
<code>variable_types</code>	Dictionary mapping variable id's to variable types

featuretools.Relationship

class featuretools.**Relationship** (*parent_variable, child_variable*)

Class to represent an relationship between entities

See also:

EntitySet, Entity, Variable

`__init__` (*parent_variable, child_variable*)

Create a relationship

Parameters

- **parent_variable** (Discrete) – Instance of variable in parent entity. Must be a Discrete Variable
- **child_variable** (Discrete) – Instance of variable in child entity. Must be a Discrete Variable

Methods

<code>__init__</code> (<i>parent_variable, child_variable</i>)	Create a relationship
<code>from_dictionary</code> (<i>arguments, es</i>)	
<code>to_dictionary</code> ()	

Attributes

<code>child_entity</code>	Child entity object
<code>child_name</code>	The name of the child, relative to the parent.
<code>child_variable</code>	Instance of variable in child entity
<code>parent_entity</code>	Parent entity object
<code>parent_name</code>	The name of the parent, relative to the child.
<code>parent_variable</code>	Instance of variable in parent entity

EntitySet load and prepare data

<code>EntitySet.entity_from_dataframe</code> (<i>entity_id, ...</i>)	Load the data for a specified entity from a Pandas DataFrame.
<code>EntitySet.add_relationship</code> (<i>relationship</i>)	Add a new relationship between entities in the entityset
<code>EntitySet.normalize_entity</code> (<i>base_entity_id, ...</i>)	Create a new entity and relationship from unique values of an existing variable.

Continued on next page

Table 150 – continued from previous page

<code>EntitySet.add_interesting_values(...)</code>	Find interesting values for categorical variables, to be used to generate “where” clauses
----------------------------------------------------	-------------------------------------------------------------------------------------------

featuretools.EntitySet.entity_from_dataframe

`EntitySet.entity_from_dataframe` (*entity_id*, *dataframe*, *index=None*, *variable_types=None*, *make_index=False*, *time_index=None*, *secondary_time_index=None*, *already_sorted=False*)

Load the data for a specified entity from a Pandas DataFrame.

Parameters

- **entity_id** (*str*) – Unique id to associate with this entity.
- **dataframe** (*pandas.DataFrame*) – Dataframe containing the data.
- **index** (*str*, *optional*) – Name of the variable used to index the entity. If None, take the first column.
- **variable_types** (*dict[str -> Variable]*, *optional*) – Keys are of variable ids and values are variable types. Used to initialize an entity’s store.
- **make_index** (*bool*, *optional*) – If True, assume index does not exist as a column in dataframe, and create a new column of that name using integers. Otherwise, assume index exists.
- **time_index** (*str*, *optional*) – Name of the variable containing time data. Type must be in `variables.DateTime` or be able to be cast to `datetime` (e.g. `str`, `float`, or `numeric`.)
- **secondary_time_index** (*dict[str -> Variable]*) – Name of variable containing time data to use a second time index for the entity.
- **already_sorted** (*bool*, *optional*) – If True, assumes that input dataframe is already sorted by time. Defaults to False.

Notes

Will infer variable types from Pandas dtype

Example

```
In [1]: import featuretools as ft
In [2]: import pandas as pd
In [3]: transactions_df = pd.DataFrame({"id": [1, 2, 3, 4, 5, 6],
...:                                 "session_id": [1, 2, 1, 3, 4, 5],
...:                                 "amount": [100.40, 20.63, 33.32, 13.12, 67.22, 1.00],
...:                                 "transaction_time": pd.date_range(start="10:00", periods=6, freq="10s"),
...:                                 "fraud": [True, False, True, False, True, True]})
```

(continues on next page)

(continued from previous page)

```

In [4]: es = ft.EntitySet("example")

In [5]: es.entity_from_dataframe(entity_id="transactions",
...:                             index="id",
...:                             time_index="transaction_time",
...:                             dataframe=transactions_df)
...:
Out [5]:
Entityset: example
Entities:
  transactions [Rows: 6, Columns: 5]
Relationships:
  No relationships

In [6]: es["transactions"]
~~~~~
↳
Entity: transactions
Variables:
  id (dtype: index)
  session_id (dtype: numeric)
  amount (dtype: numeric)
  transaction_time (dtype: datetime_time_index)
  fraud (dtype: boolean)
Shape:
  (Rows: 6, Columns: 5)

In [7]: es["transactions"].df
~~~~~
↳
   id  session_id  amount  transaction_time  fraud
1   1            1  100.40  2020-02-01 10:00:00  True
2   2            2   20.63  2020-02-01 10:00:10  False
3   3            1   33.32  2020-02-01 10:00:20  True
4   4            3   13.12  2020-02-01 10:00:30  False
5   5            4   67.22  2020-02-01 10:00:40  True
6   6            5    1.00  2020-02-01 10:00:50  True

```

featuretools.EntitySet.add_relationship

`EntitySet.add_relationship` (*relationship*)
 Add a new relationship between entities in the entityset

Parameters `relationship` (`Relationship`) – Instance of new relationship to be added.

featuretools.EntitySet.normalize_entity

`EntitySet.normalize_entity` (*base_entity_id*, *new_entity_id*, *index*, *additional_variables=None*,
copy_variables=None, *make_time_index=None*,
make_secondary_time_index=None, *new_entity_time_index=None*,
new_entity_secondary_time_index=None)

Create a new entity and relationship from unique values of an existing variable.

Parameters

- **base_entity_id** (*str*) – Entity id from which to split.
- **new_entity_id** (*str*) – Id of the new entity.
- **index** (*str*) – Variable in old entity that will become index of new entity. Relationship will be created across this variable.
- **additional_variables** (*list[str]*) – List of variable ids to remove from base_entity and move to new entity.
- **copy_variables** (*list[str]*) – List of variable ids to copy from old entity and move to new entity.
- **make_time_index** (*bool or str, optional*) – Create time index for new entity based on time index in base_entity, optionally specifying which variable in base_entity to use for time_index. If specified as True without a specific variable, uses the primary time index. Defaults to True if base entity has a time index.
- **make_secondary_time_index** (*dict[str -> list[str]], optional*) – Create a secondary time index from key. Values of dictionary are the variables to associate with the secondary time index. Only one secondary time index is allowed. If None, only associate the time index.
- **new_entity_time_index** (*str, optional*) – Rename new entity time index.
- **new_entity_secondary_time_index** (*str, optional*) – Rename new entity secondary time index.

featuretools.EntitySet.add_interesting_values

EntitySet.add_interesting_values (*max_values=5, verbose=False*)

Find interesting values for categorical variables, to be used to generate “where” clauses

Parameters

- **max_values** (*int*) – Maximum number of values per variable to add.
- **verbose** (*bool*) – If True, print summary of interesting values found.

Returns None

EntitySet serialization

<code>read_entityset(path[, profile_name])</code>	Read entityset from disk, S3 path, or URL.
---------------------------------------------------	--------------------------------------------

featuretools.read_entityset

featuretools.read_entityset (*path, profile_name=None, **kwargs*)

Read entityset from disk, S3 path, or URL.

Parameters

- **path** (*str*) – Directory on disk, S3 path, or URL to read *data_description.json*.
- **profile_name** (*str, bool*) – The AWS profile specified to write to S3. Will default to None and search for AWS credentials. Set to False to use an anonymous profile.
- **kwargs** (*keywords*) – Additional keyword arguments to pass as keyword arguments to the underlying deserialization method.

<code>EntitySet.to_csv(path[, sep, encoding, ...])</code>	Write entityset to disk in the csv format, location specified by <i>path</i> .
<code>EntitySet.to_pickle(path[, compression, ...])</code>	Write entityset in the pickle format, location specified by <i>path</i> .
<code>EntitySet.to_parquet(path[, engine, ...])</code>	Write entityset to disk in the parquet format, location specified by <i>path</i> .

featuretools.entityset.EntitySet.to_csv

`EntitySet.to_csv(path, sep=', ', encoding='utf-8', engine='python', compression=None, profile_name=None)`

Write entityset to disk in the csv format, location specified by *path*. Path could be a local path or a S3 path. If writing to S3 a tar archive of files will be written.

Parameters

- **path** (*str*) – Location on disk to write to (will be created as a directory)
- **sep** (*str*) – String of length 1. Field delimiter for the output file.
- **encoding** (*str*) – A string representing the encoding to use in the output file, defaults to 'utf-8'.
- **engine** (*str*) – Name of the engine to use. Possible values are: {'c', 'python'}.
- **compression** (*str*) – Name of the compression to use. Possible values are: {'gzip', 'bz2', 'zip', 'xz', None}.
- **profile_name** (*str*) – Name of AWS profile to use, False to use an anonymous profile, or None.

featuretools.entityset.EntitySet.to_pickle

`EntitySet.to_pickle(path, compression=None, profile_name=None)`

Write entityset in the pickle format, location specified by *path*. Path could be a local path or a S3 path. If writing to S3 a tar archive of files will be written.

Parameters

- **path** (*str*) – location on disk to write to (will be created as a directory)
- **compression** (*str*) – Name of the compression to use. Possible values are: {'gzip', 'bz2', 'zip', 'xz', None}.
- **profile_name** (*str*) – Name of AWS profile to use, False to use an anonymous profile, or None.

featuretools.entityset.EntitySet.to_parquet

`EntitySet.to_parquet(path, engine='auto', compression=None, profile_name=None)`

Write entityset to disk in the parquet format, location specified by *path*. Path could be a local path or a S3 path. If writing to S3 a tar archive of files will be written.

Parameters

- **path** (*str*) – location on disk to write to (will be created as a directory)

- **engine** (*str*) – Name of the engine to use. Possible values are: {‘auto’, ‘pyarrow’, ‘fastparquet’}.
- **compression** (*str*) – Name of the compression to use. Possible values are: {‘snappy’, ‘gzip’, ‘brotli’, None}.
- **profile_name** (*str*) – Name of AWS profile to use, False to use an anonymous profile, or None.

EntitySet query methods

<code>EntitySet.__getitem__(entity_id)</code>	Get entity instance from entityset
<code>EntitySet.find_backward_paths(...)</code>	Generator which yields all backward paths between a start and goal entity.
<code>EntitySet.find_forward_paths(...)</code>	Generator which yields all forward paths between a start and goal entity.
<code>EntitySet.get_forward_entities(entity_id[, deep])</code>	Get entities that are in a forward relationship with entity
<code>EntitySet.get_backward_entities(entity_id[, ...])</code>	Get entities that are in a backward relationship with entity

featuretools.entityset.EntitySet.__getitem__

`EntitySet.__getitem__(entity_id)`

Get entity instance from entityset

Parameters `entity_id` (*str*) – Id of entity.

Returns

Instance of entity. None if entity doesn’t exist.

Return type `Entity`

featuretools.entityset.EntitySet.find_backward_paths

`EntitySet.find_backward_paths(start_entity_id, goal_entity_id)`

Generator which yields all backward paths between a start and goal entity. Does not include paths which contain cycles.

Parameters

- **start_entity_id** (*str*) – Id of entity to start the search from.
- **goal_entity_id** (*str*) – Id of entity to find backward path to.

See also:

`BaseEntitySet.find_forward_paths()`

featuretools.entityset.EntitySet.find_forward_paths

`EntitySet.find_forward_paths(start_entity_id, goal_entity_id)`

Generator which yields all forward paths between a start and goal entity. Does not include paths which contain cycles.

Parameters

- **start_entity_id** (*str*) – id of entity to start the search from
- **goal_entity_id** (*str*) – id of entity to find forward path to

See also:

`BaseEntitySet.find_backward_paths()`

featuretools.entityset.EntitySet.get_forward_entities

`EntitySet.get_forward_entities(entity_id, deep=False)`

Get entities that are in a forward relationship with entity

Parameters

- **entity_id** (*str*) – Id entity of entity to search from.
- **deep** (*bool*) – if True, recursively find forward entities.

Yields a tuple of (descendent_id, path from entity_id to descendant).

featuretools.entityset.EntitySet.get_backward_entities

`EntitySet.get_backward_entities(entity_id, deep=False)`

Get entities that are in a backward relationship with entity

Parameters

- **entity_id** (*str*) – Id entity of entity to search from.
- **deep** (*bool*) – if True, recursively find backward entities.

Yields a tuple of (descendent_id, path from entity_id to descendant).

EntitySet visualization

`EntitySet.plot([to_file])`

Create a UML diagram-ish graph of the EntitySet.

featuretools.entityset.EntitySet.plot

`EntitySet.plot(to_file=None)`

Create a UML diagram-ish graph of the EntitySet.

Parameters **to_file** (*str, optional*) – Path to where the plot should be saved. If set to None (as by default), the plot will not be saved.

Returns

Graph object that can directly be displayed in Jupyter notebooks.

Return type `graphviz.Digraph`

Entity methods

<code>Entity.convert_variable_type(variable_id, ...)</code>	Convert variable in dataframe to different type
<code>Entity.add_interesting_values([max_values, ...])</code>	Find interesting values for categorical variables, to be used to

featuretools.entityset.Entity.convert_variable_type

`Entity.convert_variable_type` (*variable_id*, *new_type*, *convert_data=True*, ***kwargs*)
 Convert variable in dataframe to different type

Parameters

- **variable_id** (*str*) – Id of variable to convert.
- **new_type** (subclass of *Variable*) – Type of variable to convert to.
- **entityset** (*BaseEntitySet*) – EntitySet associated with this entity.
- **convert_data** (*bool*) – If True, convert underlying data in the EntitySet.

Raises `RuntimeError` – Raises if it cannot convert the underlying data

Examples

```
>>> from featuretools.tests.testing_utils import make_ecommerce_entityset
>>> es = make_ecommerce_entityset()
>>> es["customers"].convert_variable_type("engagement_level", vtypes.Categorical)
```

featuretools.entityset.Entity.add_interesting_values

`Entity.add_interesting_values` (*max_values=5*, *verbose=False*)

Find interesting values for categorical variables, to be used to generate “where” clauses

Parameters

- **max_values** (*int*) – Maximum number of values per variable to add.
- **verbose** (*bool*) – If True, print summary of interesting values found.

Returns None

Relationship attributes

<code>Relationship.parent_variable</code>	Instance of variable in parent entity
<code>Relationship.child_variable</code>	Instance of variable in child entity
<code>Relationship.parent_entity</code>	Parent entity object
<code>Relationship.child_entity</code>	Child entity object

featuretools.entityset.Relationship.parent_variable

`Relationship.parent_variable`
 Instance of variable in parent entity

featuretools.entityset.Relationship.child_variable

Relationship.**child_variable**
 Instance of variable in child entity

featuretools.entityset.Relationship.parent_entity

Relationship.**parent_entity**
 Parent entity object

featuretools.entityset.Relationship.child_entity

Relationship.**child_entity**
 Child entity object

Variable types

<i>Index</i> (id, entity[, name])	Represents variables that uniquely identify an instance of an entity
<i>Id</i> (id, entity[, name, categories])	Represents variables that identify another entity
<i>TimeIndex</i> (id, entity[, name])	Represents time index of entity
<i>DatetimeTimeIndex</i> (id, entity[, name, format])	Represents time index of entity that is a datetime
<i>NumericTimeIndex</i> (id, entity[, name, range, ...])	Represents time index of entity that is numeric
<i>Datetime</i> (id, entity[, name, format])	Represents variables that are points in time
<i>Numeric</i> (id, entity[, name, range, ...])	Represents variables that contain numeric values
<i>Categorical</i> (id, entity[, name, categories])	Represents variables that can take an unordered discrete values
<i>Ordinal</i> (id, entity[, name])	Represents variables that take on an ordered discrete value
<i>Boolean</i> (id, entity[, name, true_values, ...])	Represents variables that take on one of two values
<i>Text</i> (id, entity[, name])	Represents variables that are arbitrary strings
<i>LatLong</i> (id, entity[, name])	Represents an ordered pair (Latitude, Longitude) To make a latlong in a dataframe do <code>data['latlong'] = data[['latitude', 'longitude']].apply(tuple, axis=1)</code>
<i>ZIPCode</i> (id, entity[, name, categories])	Represents a postal address in the United States.
<i>IPAddress</i> (id, entity[, name])	Represents a computer network address.
<i>FullName</i> (id, entity[, name])	Represents a person's full name.
<i>EmailAddress</i> (id, entity[, name])	Represents an email box to which email message are sent.
<i>URL</i> (id, entity[, name])	Represents a valid web url (with or without http/www)
<i>PhoneNumber</i> (id, entity[, name])	Represents any valid phone number.
<i>DateOfBirth</i> (id, entity[, name, format])	Represents a date of birth as a datetime
<i>CountryCode</i> (id, entity[, name, categories])	Represents an ISO-3166 standard country code.
<i>SubRegionCode</i> (id, entity[, name, categories])	Represents an ISO-3166 standard sub-region code.
<i>FilePath</i> (id, entity[, name])	Represents a valid filepath, absolute or relative

featuretools.variable_types.Index

class featuretools.variable_types.**Index** (*id, entity, name=None*)

Represents variables that uniquely identify an instance of an entity

count

Type int

__init__ (*id, entity, name=None*)

Initialize self. See help(type(self)) for accurate signature.

Methods

<code>__init__(id, entity[, name])</code>	Initialize self.
<code>create_from(variable)</code>	Create new variable this type from existing
<code>to_data_description()</code>	

Attributes

<code>dtype</code>
<code>entityset</code>
<code>interesting_values</code>
<code>name</code>
<code>series</code>
<code>type_string</code>

featuretools.variable_types.Id

class featuretools.variable_types.**Id** (*id, entity, name=None, categories=None*)

Represents variables that identify another entity

__init__ (*id, entity, name=None, categories=None*)

Initialize self. See help(type(self)) for accurate signature.

Methods

<code>__init__(id, entity[, name, categories])</code>	Initialize self.
<code>create_from(variable)</code>	Create new variable this type from existing
<code>to_data_description()</code>	

Attributes

<code>dtype</code>
<code>entityset</code>
<code>interesting_values</code>
<code>name</code>
<code>series</code>

Continued on next page

Table 161 – continued from previous page

type_string

featuretools.variable_types.TimeIndex

class featuretools.variable_types.**TimeIndex** (*id, entity, name=None*)

Represents time index of entity

__init__ (*id, entity, name=None*)

Initialize self. See help(type(self)) for accurate signature.

Methods

<code>__init__(id, entity[, name])</code>	Initialize self.
<code>create_from(variable)</code>	Create new variable this type from existing
<code>to_data_description()</code>	

Attributes

<code>dtype</code>
<code>entityset</code>
<code>interesting_values</code>
<code>name</code>
<code>series</code>
<code>type_string</code>

featuretools.variable_types.DatetimeTimeIndex

class featuretools.variable_types.**DatetimeTimeIndex** (*id, entity, name=None, format=None*)

Represents time index of entity that is a datetime

__init__ (*id, entity, name=None, format=None*)

Initialize self. See help(type(self)) for accurate signature.

Methods

<code>__init__(id, entity[, name, format])</code>	Initialize self.
<code>create_from(variable)</code>	Create new variable this type from existing
<code>to_data_description()</code>	

Attributes

<code>dtype</code>
<code>entityset</code>
<code>interesting_values</code>
<code>name</code>

Continued on next page

Table 165 – continued from previous page

```
series
type_string
```

featuretools.variable_types.NumericTimeIndex

```
class featuretools.variable_types.NumericTimeIndex (id, entity, name=None,
                                                    range=None,
                                                    start_inclusive=True,
                                                    end_inclusive=False)
```

Represents time index of entity that is numeric

```
__init__ (id, entity, name=None, range=None, start_inclusive=True, end_inclusive=False)
    Initialize self. See help(type(self)) for accurate signature.
```

Methods

<code>__init__</code> (id, entity[, name, range, ...])	Initialize self.
<code>create_from</code> (variable)	Create new variable this type from existing
<code>to_data_description</code> ()	

Attributes

```
dtype
entityset
interesting_values
name
series
type_string
```

featuretools.variable_types.Datetime

```
class featuretools.variable_types.Datetime (id, entity, name=None, format=None)
    Represents variables that are points in time
```

Parameters `format` (*str*) – Python datetime format string documented [here](#).

```
__init__ (id, entity, name=None, format=None)
    Initialize self. See help(type(self)) for accurate signature.
```

Methods

<code>__init__</code> (id, entity[, name, format])	Initialize self.
<code>create_from</code> (variable)	Create new variable this type from existing
<code>to_data_description</code> ()	

Attributes

dtype
entityset
interesting_values
name
series
type_string

featuretools.variable_types.Numeric

class featuretools.variable_types.Numeric(*id*, *entity*, *name=None*, *range=None*, *start_inclusive=True*, *end_inclusive=False*)

Represents variables that contain numeric values

Parameters

- **range** (*list*, *optional*) – List of start and end. Can use inf and -inf to represent infinity. Unconstrained if not specified.
- **start_inclusive** (*bool*, *optional*) – Whether or not range includes the start value.
- **end_inclusive** (*bool*, *optional*) – Whether or not range includes the end value

max

Type float

min

Type float

std

Type float

mean

Type float

__init__ (*id*, *entity*, *name=None*, *range=None*, *start_inclusive=True*, *end_inclusive=False*)
Initialize self. See help(type(self)) for accurate signature.

Methods

<code>__init__(id, entity[, name, range, ...])</code>	Initialize self.
<code>create_from(variable)</code>	Create new variable this type from existing
<code>to_data_description()</code>	

Attributes

dtype
entityset
interesting_values
name
series

Continued on next page

Table 171 – continued from previous page

type_string

featuretools.variable_types.Categorical
class featuretools.variable_types.**Categorical** (*id, entity, name=None, categories=None*)

Represents variables that can take an unordered discrete values

Parameters **categories** (*list*) – List of categories. If left blank, inferred from data.

__init__ (*id, entity, name=None, categories=None*)

Initialize self. See help(type(self)) for accurate signature.

Methods

__init__ (<i>id, entity[, name, categories]</i>)	Initialize self.
create_from (<i>variable</i>)	Create new variable this type from existing
to_data_description ()	

Attributes

<i>dtype</i>
<i>entityset</i>
<i>interesting_values</i>
<i>name</i>
<i>series</i>
<i>type_string</i>

featuretools.variable_types.Ordinal
class featuretools.variable_types.**Ordinal** (*id, entity, name=None*)

Represents variables that take on an ordered discrete value

__init__ (*id, entity, name=None*)

Initialize self. See help(type(self)) for accurate signature.

Methods

__init__ (<i>id, entity[, name]</i>)	Initialize self.
create_from (<i>variable</i>)	Create new variable this type from existing
to_data_description ()	

Attributes

<i>dtype</i>
<i>entityset</i>
<i>interesting_values</i>
<i>name</i>

Continued on next page

Table 175 – continued from previous page

series
type_string

featuretools.variable_types.Boolean

class featuretools.variable_types.**Boolean** (*id, entity, name=None, true_values=None, false_values=None*)

Represents variables that take on one of two values

Parameters

- **true_values** (*list*) – List of valued true values. Defaults to [1, True, “true”, “True”, “yes”, “t”, “T”]
- **false_values** (*list*) – List of valued false values. Defaults to [0, False, “false”, “False”, “no”, “F”, “F”]

__init__ (*id, entity, name=None, true_values=None, false_values=None*)

Initialize self. See help(type(self)) for accurate signature.

Methods

__init__ (<i>id, entity[, name, true_values, ...]</i>)	Initialize self.
create_from (<i>variable</i>)	Create new variable this type from existing
to_data_description ()	

Attributes

dtype
entityset
interesting_values
name
series
type_string

featuretools.variable_types.Text

class featuretools.variable_types.**Text** (*id, entity, name=None*)

Represents variables that are arbitrary strings

__init__ (*id, entity, name=None*)

Initialize self. See help(type(self)) for accurate signature.

Methods

__init__ (<i>id, entity[, name]</i>)	Initialize self.
create_from (<i>variable</i>)	Create new variable this type from existing
to_data_description ()	

Attributes

dtype
entityset
interesting_values
name
series
type_string

featuretools.variable_types.LatLong

class featuretools.variable_types.LatLong (*id, entity, name=None*)

Represents an ordered pair (Latitude, Longitude) To make a latlong in a dataframe do `data['latlong'] = data[['latitude', 'longitude']].apply(tuple, axis=1)`

__init__ (*id, entity, name=None*)

Initialize self. See help(type(self)) for accurate signature.

Methods

__init__ (id, entity[, name])	Initialize self.
create_from (variable)	Create new variable this type from existing
to_data_description ()	

Attributes

dtype
entityset
interesting_values
name
series
type_string

featuretools.variable_types.ZIPCode

class featuretools.variable_types.ZIPCode (*id, entity, name=None, categories=None*)

Represents a postal address in the United States. Consists of a series of digits which are casts as string. Five digit and 9 digit zipcodes are supported.

__init__ (*id, entity, name=None, categories=None*)

Initialize self. See help(type(self)) for accurate signature.

Methods

__init__ (id, entity[, name, categories])	Initialize self.
create_from (variable)	Create new variable this type from existing
to_data_description ()	

Attributes

dtype
entityset
interesting_values
name
series
type_string

featuretools.variable_types.IPAddress

class featuretools.variable_types.**IPAddress** (*id, entity, name=None*)

Represents a computer network address. Represented in dotted-decimal notation. IPv4 and IPv6 are supported.

__init__ (*id, entity, name=None*)

Initialize self. See help(type(self)) for accurate signature.

Methods

__init__ (id, entity[, name])	Initialize self.
create_from(variable)	Create new variable this type from existing
to_data_description()	

Attributes

dtype
entityset
interesting_values
name
series
type_string

featuretools.variable_types.FullName

class featuretools.variable_types.**FullName** (*id, entity, name=None*)

Represents a person's full name. May consist of a first name, last name, and a title.

__init__ (*id, entity, name=None*)

Initialize self. See help(type(self)) for accurate signature.

Methods

__init__ (id, entity[, name])	Initialize self.
create_from(variable)	Create new variable this type from existing
to_data_description()	

Attributes

dtype

entityset

interesting_values

name

series

type_string

featuretools.variable_types.EmailAddress

class featuretools.variable_types.**EmailAddress** (*id, entity, name=None*)

Represents an email box to which email message are sent. Consists of a local-part, an @ symbol, and a domain.

__init__ (*id, entity, name=None*)

Initialize self. See help(type(self)) for accurate signature.

Methods

__init__ (id, entity[, name])	Initialize self.
create_from (variable)	Create new variable this type from existing
to_data_description ()	

Attributes

dtype

entityset

interesting_values

name

series

type_string

featuretools.variable_types.URL

class featuretools.variable_types.**URL** (*id, entity, name=None*)

Represents a valid web url (with or without http/www)

__init__ (*id, entity, name=None*)

Initialize self. See help(type(self)) for accurate signature.

Methods

__init__ (id, entity[, name])	Initialize self.
create_from (variable)	Create new variable this type from existing
to_data_description ()	

Attributes

<code>dtype</code>
<code>entityset</code>
<code>interesting_values</code>
<code>name</code>
<code>series</code>
<code>type_string</code>

featuretools.variable_types.PhoneNumber

class featuretools.variable_types.**PhoneNumber** (*id, entity, name=None*)

Represents any valid phone number. Can be with/without parenthesis. Can be with/without area/country codes.

`__init__` (*id, entity, name=None*)

Initialize self. See help(type(self)) for accurate signature.

Methods

<code>__init__(id, entity[, name])</code>	Initialize self.
<code>create_from(variable)</code>	Create new variable this type from existing
<code>to_data_description()</code>	

Attributes

<code>dtype</code>
<code>entityset</code>
<code>interesting_values</code>
<code>name</code>
<code>series</code>
<code>type_string</code>

featuretools.variable_types.DateOfBirth

class featuretools.variable_types.**DateOfBirth** (*id, entity, name=None, format=None*)

Represents a date of birth as a datetime

`__init__` (*id, entity, name=None, format=None*)

Initialize self. See help(type(self)) for accurate signature.

Methods

<code>__init__(id, entity[, name, format])</code>	Initialize self.
<code>create_from(variable)</code>	Create new variable this type from existing
<code>to_data_description()</code>	

Attributes

dtype
entityset
interesting_values
name
series
type_string

featuretools.variable_types.CountryCode

class featuretools.variable_types.**CountryCode** (*id, entity, name=None, categories=None*)

Represents an ISO-3166 standard country code. ISO 3166-1 (countries) are supported. These codes should be in the Alpha-2 format. e.g. United States of America = US

__init__ (*id, entity, name=None, categories=None*)
 Initialize self. See help(type(self)) for accurate signature.

Methods

__init__ (id, entity[, name, categories])	Initialize self.
create_from(variable)	Create new variable this type from existing
to_data_description()	

Attributes

dtype
entityset
interesting_values
name
series
type_string

featuretools.variable_types.SubRegionCode

class featuretools.variable_types.**SubRegionCode** (*id, entity, name=None, categories=None*)

Represents an ISO-3166 standard sub-region code. ISO 3166-2 codes (sub-regions) are supported. These codes should be in the Alpha-2 format. e.g. United States of America, Arizona = US-AZ

__init__ (*id, entity, name=None, categories=None*)
 Initialize self. See help(type(self)) for accurate signature.

Methods

__init__ (id, entity[, name, categories])	Initialize self.
create_from(variable)	Create new variable this type from existing
to_data_description()	

Attributes

dtype
entityset
interesting_values
name
series
type_string

featuretools.variable_types.FilePath

class featuretools.variable_types.**FilePath** (*id, entity, name=None*)

Represents a valid filepath, absolute or relative

`__init__` (*id, entity, name=None*)

Initialize self. See help(type(self)) for accurate signature.

Methods

<code>__init__</code> (<i>id, entity[, name]</i>)	Initialize self.
<code>create_from</code> (<i>variable</i>)	Create new variable this type from existing
<code>to_data_description</code> ()	

Attributes

dtype
entityset
interesting_values
name
series
type_string

Feature Selection

`remove_low_information_features` (*feature_matrix*) Select features that have at least 2 unique values and that are not all null

featuretools.selection.remove_low_information_features

featuretools.selection.**remove_low_information_features** (*feature_matrix, features=None*)

Select features that have at least 2 unique values and that are not all null

Parameters

- **feature_matrix** (pd.DataFrame) – DataFrame whose columns are feature names and rows are instances
- **features** (list[featuretools.FeatureBase] or list[str], optional) – List of features to select

Returns (feature_matrix, features)

3.19 Changelog

v0.13.2 Jan 31, 2020

- **Enhancements**
 - Support for Pandas 1.0.0 (GH#844)
- **Changes**
 - Remove dependency on s3fs library for anonymous downloads from S3 (GH#825)
- **Testing Changes**
 - Added GitHub Action to automatically run performance tests (GH#840)

Thanks to the following people for contributing to this release: @frances-h, @rwedge

v0.13.1 Dec 28, 2019

- **Fixes**
 - Raise error when given wrong input for ignore_variables (GH#826)
 - Fix multi-output features not created when there is no child data (GH#834)
 - Removing type casting in Equals and NotEquals primitives (GH#504)
- **Changes**
 - Replace pd.timedelta time units that were deprecated (GH#822)
 - Move sklearn wrapper to separate library (GH#835, GH#837)
- **Testing Changes**
 - Run unit tests in windows environment (GH#790)
 - Update boto3 version requirement for tests (GH#838)

Thanks to the following people for contributing to this release: @jeffzi, @kmax12, @rwedge, @systemshift

v0.13.0 Nov 30, 2019

- **Enhancements**
 - Added GitHub Action to auto upload releases to PyPI (GH#816)
- **Fixes**
 - Fix issue where some primitive options would not be applied (GH#807)
 - Fix issue with converting to pickle or parquet after adding interesting features (GH#798, GH#823)
 - Diff primitive now calculates using all available data (GH#824)
 - Prevent DFS from creating Identity Features of globally ignored variables (GH#819)
- **Changes**
 - Remove python 2.7 support from serialize.py (GH#812)
 - Make smart_open, boto3, and s3fs optional dependencies (GH#827)
- **Documentation Changes**

- remove python 2.7 support and add 3.7 in install.rst (GH#805)
- Fix import error in docs (GH#803)
- Fix release title formatting in changelog (GH#806)

- **Testing Changes**

- Use multiple CPUS to run tests on CI (GH#811)
- Refactor test entityset creation to avoid saving to disk (GH#813, GH#821)
- Remove get_values() from test_es.py to remove warnings (GH#820)

Thanks to the following people for contributing to this release: @frances-h, @jeff-hernandez, @rwedge, @systemshift

Breaking Changes

- The libraries used for downloading or uploading from S3 or URLs are now optional and will no longer be installed by default. To use this functionality they will need to be installed separately.
- The fix to how the Diff primitive is calculated may slow down the overall calculation time of feature lists that use this primitive.

v0.12.0 Oct 31, 2019

- **Enhancements**

- Added First primitive (GH#770)
- Added Entropy aggregation primitive (GH#779)
- Allow custom naming for multi-output primitives (GH#780)

- **Fixes**

- Prevents user from removing base entity time index using additional_variables (GH#768)
- Fixes error when a multioutput primitive was supplied to dfs as a groupby trans primitive (GH#786)

- **Changes**

- Drop Python 2 support (GH#759)
- Add unit parameter to AvgTimeBetween (GH#771)
- Require Pandas 0.24.1 or higher (GH#787)

- **Documentation Changes**

- Update featuretools slack link (GH#765)
- Set up repo to use Read the Docs (GH#776)
- Add First primitive to API reference docs (GH#782)

- **Testing Changes**

- CircleCI fixes (GH#774)
- Disable PIP progress bars (GH#775)

Thanks to the following people for contributing to this release: @ablacke-ayx, @BoopBoopBeepBoop, @jeffzi, @kmax12, @rwedge, @thhomebrewnerd, @twjobson

v0.11.0 Sep 30, 2019

Warning: The next non-bugfix release of Featuretools will not support Python 2

- **Enhancements**

- Improve how files are copied and written (GH#721)
- Add number of rows to graph in entityset.plot (GH#727)
- Added support for pandas DateOffsets in DFS and Timedelta (GH#732)
- Enable feature-specific top_n value using a dictionary in encode_features (GH#735)
- Added progress_callback parameter to dfs() and calculate_feature_matrix() (GH#739, GH#745)
- Enable specifying primitives on a per column or per entity basis (GH#748)

- **Fixes**

- Fixed entity set deserialization (GH#720)
- Added error message when DateTimeIndex is a variable but not set as the time_index (GH#723)
- Fixed CumCount and other group-by transform primitives that take ID as input (GH#733, GH#754)
- Fix progress bar undercounting (GH#743)
- Updated training_window error assertion to only check against observations (GH#728)
- Don't delete the whole destination folder while saving entityset (GH#717)

- **Changes**

- Raise warning and not error on schema version mismatch (GH#718)
- Change feature calculation to return in order of instance ids provided (GH#676)
- Removed time remaining from displayed progress bar in dfs() and calculate_feature_matrix() (GH#739)
- Raise warning in normalize_entity() when time_index of base_entity has an invalid type (GH#749)
- Remove toolz as a direct dependency (GH#755)
- Allow boolean variable types to be used in the Multiply primitive (GH#756)

- **Documentation Changes**

- Updated URL for Compose (GH#716)

- **Testing Changes**

- Update dependencies (GH#738, GH#741, GH#747)

Thanks to the following people for contributing to this release: @angela97lin, @chidauri, @christopherbunn, @frances-h, @jeff-hernandez, @kmax12, @MarcoGorelli, @rwedge, @thhomebrewnerd

Breaking Changes

- Feature calculations will return in the order of instance ids provided instead of the order of time points instances are calculated at.

v0.10.1 Aug 25, 2019

- **Fixes**

- Fix serialized LatLong data being loaded as strings (GH#712)

- **Documentation Changes**

- Fixed FAQ cell output (GH#710)

Thanks to the following people for contributing to this release: @gsheni, @rwwedge

v0.10.0 Aug 19, 2019

Warning: The next non-bugfix release of Featuretools will not support Python 2

- **Enhancements**

- Give more frequent progress bar updates and update chunk size behavior (GH#631, GH#696)
- Added drop_first as param in encode_features (GH#647)
- Added support for stacking multi-output primitives (GH#679)
- Generate transform features of direct features (GH#623)
- Added serializing and deserializing from S3 and deserializing from URLs (GH#685)
- Added nlp_primitives as an add-on library (GH#704)
- Added AutoNormalize to Featuretools plugins (GH#699)
- Added functionality for relative units (month/year) in Timedelta (GH#692)
- Added categorical-encoding as an add-on library (GH#700)

- **Fixes**

- Fix performance regression in DFS (GH#637)
- Fix deserialization of feature relationship path (GH#665)
- Set index after adding ancestor relationship variables (GH#668)
- Fix user-supplied variable_types modification in Entity init (GH#675)
- Don't calculate dependencies of unnecessary features (GH#667)
- Prevent normalize entity's new entity having same index as base entity (GH#681)
- Update variable type inference to better check for string values (GH#683)

- **Changes**

- Moved dask, distributed imports (GH#634)

- **Documentation Changes**

- Miscellaneous changes (GH#641, GH#658)
- Modified doc_string of top_n in encoding (GH#648)
- Hyperlinked ComposeML (GH#653)
- Added FAQ (GH#620, GH#677)
- Fixed FAQ question with multiple question marks (GH#673)

- **Testing Changes**

- Add master, and release tests for premium primitives (GH#660, GH#669)
- Miscellaneous changes (GH#672, GH#674)

Thanks to the following people for contributing to this release: @alexjwang, @allisonportis, @ayushpatidar, @CJStadler, @ctduffy, @gsheni, @jeff-hernandez, @jeremyliweishih, @kmax12, @rwedge, @zhxt95,

v0.9.1 July 3, 2019

- **Enhancements**

- Speedup groupby transform calculations (GH#609)
- Generate features along all paths when there are multiple paths between entities (GH#600, GH#608)

- **Fixes**

- Select columns of dataframe using a list (GH#615)
- Change type of features calculated on Index features to Categorical (GH#602)
- Filter dataframes through forward relationships (GH#625)
- Specify Dask version in requirements for python 2 (GH#627)
- Keep dataframe sorted by time during feature calculation (GH#626)
- Fix bug in encode_features that created duplicate columns of features with multiple outputs (GH#622)

- **Changes**

- Remove unused variance_selection.py file (GH#613)
- Remove Timedelta data param (GH#619)
- Remove DaysSince primitive (GH#628)

- **Documentation Changes**

- Add installation instructions for add-on libraries (GH#617)
- Clarification of Multi Output Feature Creation (GH#638)
- Miscellaneous changes (GH#632, GH#639)

- **Testing Changes**

- Miscellaneous changes (GH#595, GH#612)

Thanks to the following people for contributing to this release: @CJStadler, @kmax12, @rwedge, @gsheni, @kkleidal, @ctduffy

v0.9.0 June 19, 2019

- **Enhancements**

- Add unit parameter to timesince primitives (GH#558)
- Add ability to install optional add on libraries (GH#551)
- Load and save features from open files and strings (GH#566)
- Support custom variable types (GH#571)
- Support entitysets which have multiple paths between two entities (GH#572, GH#544)
- Added show_info function, more output information added to CLI *featuretools info* (GH#525)

- **Fixes**

- Normalize_entity specifies error when 'make_time_index' is an invalid string (GH#550)

- Schema version added for entityset serialization (GH#586)
- Renamed features have names correctly serialized (GH#585)
- Improved error message for index/time_index being the same column in normalize_entity and entity_from_dataframe (GH#583)
- Removed all mentions of allow_where (GH#587, GH#588)
- Removed unused variable in normalize entity (GH#589)
- Change time since return type to numeric (GH#606)
- **Changes**
 - Refactor get_pandas_data_slice to take single entity (GH#547)
 - Updates TimeSincePrevious and Diff Primitives (GH#561)
 - Remove unnecessary time_last variable (GH#546)
- **Documentation Changes**
 - Add Featuretools Enterprise to documentation (GH#563)
 - Miscellaneous changes (GH#552, GH#573, GH#577, GH#599)
- **Testing Changes**
 - Miscellaneous changes (GH#559, GH#569, GH#570, GH#574, GH#584, GH#590)

Thanks to the following people for contributing to this release: @alexjwang, @allisonportis, @CJStadler, @ct-duffy, @gsheni, @kmax12, @rwwedge

v0.8.0 May 17, 2019

- Rename NUnique to NumUnique (GH#510)
- Serialize features as JSON (GH#532)
- Drop all variables at once in normalize_entity (GH#533)
- Remove unnecessary sorting from normalize_entity (GH#535)
- Features cache their names (GH#536)
- Only calculate features for instances before cutoff (GH#523)
- Remove all relative imports (GH#530)
- Added FullName Variable Type (GH#506)
- Add error message when target entity does not exist (GH#520)
- New demo links (GH#542)
- Remove duplicate features check in DFS (GH#538)
- featuretools_primitives entry point expects list of primitive classes (GH#529)
- Update ALL_VARIABLE_TYPES list (GH#526)
- More Informative N Jobs Prints and Warnings (GH#511)
- Update sklearn version requirements (GH#541)
- Update Makefile (GH#519)
- Remove unused parameter in Entity._handle_time (GH#524)
- Remove build_ext code from setup.py (GH#513)

- Documentation updates (GH#512, GH#514, GH#515, GH#521, GH#522, GH#527, GH#545)
- Testing updates (GH#509, GH#516, GH#517, GH#539)

Thanks to the following people for contributing to this release: @bphi, @CharlesBradshaw, @CJStadler, @glentennis, @gsheni, @kmax12, @rwedge

Breaking Changes

- NUnique has been renamed to NumUnique.

Previous behavior

```
from featuretools.primitives import NUnique
```

New behavior

```
from featuretools.primitives import NumUnique
```

v0.7.1 Apr 24, 2019

- Automatically generate feature name for controllable primitives (GH#481)
- Primitive docstring updates (GH#489, GH#492, GH#494, GH#495)
- Change primitive functions that returned strings to return functions (GH#499)
- CLI customizable via entrypoints (GH#493)
- Improve calculation of aggregation features on grandchildren (GH#479)
- Refactor entrypoints to use decorator (GH#483)
- Include doctests in testing suite (GH#491)
- Documentation updates (GH#490)
- Update how standard primitives are imported internally (GH#482)

Thanks to the following people for contributing to this release: @bukosabino, @CharlesBradshaw, @glentennis, @gsheni, @jeff-hernandez, @kmax12, @minkvsky, @rwedge, @thehomebrewnerd

v0.7.0 Mar 29, 2019

- Improve Entity Set Serialization (GH#361)
- Support calling a primitive instance's function directly (GH#461, GH#468)
- Support other libraries extending featuretools functionality via entrypoints (GH#452)
- Remove featuretools install command (GH#475)
- Add GroupByTransformFeature (GH#455, GH#472, GH#476)
- Update Haversine Primitive (GH#435, GH#462)
- Add commutative argument to SubtractNumeric and DivideNumeric primitives (GH#457)
- Add FilePath variable_type (GH#470)
- Add PhoneNumber, DateOfBirth, URL variable types (GH#447)
- Generalize infer_variable_type, convert_variable_data and convert_all_variable_data methods (GH#423)
- Documentation updates (GH#438, GH#446, GH#458, GH#469)
- Testing updates (GH#440, GH#444, GH#445, GH#459)

Thanks to the following people for contributing to this release: @bukosabino, @CharlesBradshaw, @ColCarroll, @glentennis, @grayskripko, @gsheni, @jeff-hernandez, @jrkinley, @kmax12, @RogerTangos, @rwedge

Breaking Changes

- `ft.dfs` now has a `groupby_trans_primitives` parameter that DFS uses to automatically construct features that group by an ID column and then apply a transform primitive to search group. This change applies to the following primitives: `CumSum`, `CumCount`, `CumMean`, `CumMin`, and `CumMax`.

Previous behavior

```
ft.dfs(entityset=es,
      target_entity='customers',
      trans_primitives=["cum_mean"])
```

New behavior

```
ft.dfs(entityset=es,
      target_entity='customers',
      groupby_trans_primitives=["cum_mean"])
```

- Related to the above change, cumulative transform features are now defined using a new feature class, `GroupByTransformFeature`.

Previous behavior

```
ft.Feature([base_feature, groupby_feature],
          primitive=CumulativePrimitive)
```

New behavior

```
ft.Feature(base_feature, groupby=groupby_feature,
          primitive=CumulativePrimitive)
```

v0.6.1 Feb 15, 2019

- Cumulative primitives (GH#410)
- `Entity.query_by_values` now preserves row order of underlying data (GH#428)
- Implementing Country Code and Sub Region Codes as variable types (GH#430)
- Added `IPAddress` and `EmailAddress` variable types (GH#426)
- Install data and dependencies (GH#403)
- Add `TimeSinceFirst`, fix `TimeSinceLast` (GH#388)
- Allow user to pass in desired feature return types (GH#372)
- Add new configuration object (GH#401)
- Replace `NUnique` `get_function` (GH#434)
- `_calculate_identity_features` now only returns the features asked for, instead of the entire entity (GH#429)
- Primitive function name uniqueness (GH#424)
- Update `NumCharacters` and `NumWords` primitives (GH#419)
- Removed `Variable.dtype` (GH#416, GH#433)
- Change to `zipcode` rep, str for pandas (GH#418)
- Remove pandas version upper bound (GH#408)

- Make S3 dependencies optional (GH#404)
- Check that `agg_primitives` and `trans_primitives` are right primitive type (GH#397)
- Mean primitive changes (GH#395)
- Fix transform stacking on multi-output aggregation (GH#394)
- Fix `list_primitives` (GH#391)
- Handle `graphviz` dependency (GH#389, GH#396, GH#398)
- Testing updates (GH#402, GH#417, GH#433)
- Documentation updates (GH#400, GH#409, GH#415, GH#417, GH#420, GH#421, GH#422, GH#431)

Thanks to the following people for contributing to this release: @CharlesBradshaw, @csala, @floscha, @gsheni, @jxwolstenholme, @kmax12, @RogerTangos, @rwedge

v0.6.0 Jan 30, 2018

- Primitive refactor (GH#364)
- Mean ignore NaNs (GH#379)
- Plotting entitysets (GH#382)
- Add seed features later in DFS process (GH#357)
- Multiple output column features (GH#376)
- Add ZipCode Variable Type (GH#367)
- Add `primitive.get_filepath` and example of primitive loading data from external files (GH#380)
- Transform primitives take series as input (GH#385)
- Update dependency requirements (GH#378, GH#383, GH#386)
- Add `modulo` to override tests (GH#384)
- Update documentation (GH#368, GH#377)
- Update README.md (GH#366, GH#373)
- Update CI tests (GH#359, GH#360, GH#375)

Thanks to the following people for contributing to this release: @floscha, @gsheni, @kmax12, @RogerTangos, @rwedge

v0.5.1 Dec 17, 2018

- Add missing dependencies (GH#353)
- Move comment to note in documentation (GH#352)

v0.5.0 Dec 17, 2018

- Add specific error for duplicate `additional/copy_variables` in `normalize_entity` (GH#348)
- Removed `EntitySet._import_from_dataframe` (GH#346)
- Removed `time_index_reduce` parameter (GH#344)
- Allow installation of additional primitives (GH#326)
- Fix `DatetimeIndex` variable conversion (GH#342)
- Update Sklearn DFS Transformer (GH#343)
- Clean up entity creation logic (GH#336)

- remove casting to list in transform feature calculation (GH#330)
- Fix sklearn wrapper (GH#335)
- Add readme to pypi
- Update conda docs after move to conda-forge (GH#334)
- Add wrapper for scikit-learn Pipelines (GH#323)
- Remove parse_date_cols parameter from EntitySet._import_from_dataframe (GH#333)

Thanks to the following people for contributing to this release: @bukosabino, @georgewambold, @gsheni, @jeff-herandez, @kmax12, and @rwwedge.

v0.4.1 Nov 29, 2018

- Resolve bug preventing using first column as index by default (GH#308)
- Handle return type when creating features from Id variables (GH#318)
- Make id an optional parameter of EntitySet constructor (GH#324)
- Handle primitives with same function being applied to same column (GH#321)
- Update requirements (GH#328)
- Clean up DFS arguments (GH#319)
- Clean up Pandas Backend (GH#302)
- Update properties of cumulative transform primitives (GH#320)
- Feature stability between versions documentation (GH#316)
- Add download count to GitHub readme (GH#310)
- Fixed #297 update tests to check error strings (GH#303)
- Remove usage of fixtures in agg primitive tests (GH#325)

v0.4.0 Oct 31, 2018

- Remove ft.utils.gen_utils.getsize and make pynpler a test requirement (GH#299)
- Update requirements.txt (GH#298)
- Refactor EntitySet.find_path(...) (GH#295)
- Clean up unused methods (GH#293)
- Remove unused parents property of Entity (GH#283)
- Removed relationships parameter (GH#284)
- Improve time index validation (GH#285)
- Encode features with “unknown” class in categorical (GH#287)
- Allow where clauses on direct features in Deep Feature Synthesis (GH#279)
- Change to fullargspec (GH#288)
- Parallel verbose fixes (GH#282)
- Update tests for python 3.7 (GH#277)
- Check duplicate rows cutoff times (GH#276)
- Load retail demo data using compressed file (GH#271)

v0.3.1 Sept 28, 2018

- Handling time rewrite (GH#245)
- Update deep_feature_synthesis.py (GH#249)
- Handling return type when creating features from DatetimeTimeIndex (GH#266)
- Update retail.py (GH#259)
- Improve Consistency of Transform Primitives (GH#236)
- Update demo docstrings (GH#268)
- Handle non-string column names (GH#255)
- Clean up merging of aggregation primitives (GH#250)
- Add tests for Entity methods (GH#262)
- Handle no child data when calculating aggregation features with multiple arguments (GH#264)
- Add *is_string* utils function (GH#260)
- Update python versions to match docker container (GH#261)
- Handle where clause when no child data (GH#258)
- No longer cache demo csvs, remove config file (GH#257)
- Avoid stacking “expanding” primitives (GH#238)
- Use randomly generated names in retail csv (GH#233)
- Update README.md (GH#243)

v0.3.0 Aug 27, 2018

- Improve performance of all feature calculations (GH#224)
- Update agg primitives to use more efficient functions (GH#215)
- Optimize metadata calculation (GH#229)
- More robust handling when no data at a cutoff time (GH#234)
- Workaround categorical merge (GH#231)
- Switch which CSV is associated with which variable (GH#228)
- Remove unused kwargs from query_by_values, filter_and_sort (GH#225)
- Remove convert_links_to_integers (GH#219)
- Add conda install instructions (GH#223, GH#227)
- Add example of using Dask to parallelize to docs (GH#221)

v0.2.2 Aug 20, 2018

- Remove unnecessary check no related instances call and refactor (GH#209)
- Improve memory usage through support for pandas categorical types (GH#196)
- Bump minimum pandas version from 0.20.3 to 0.23.0 (GH#216)
- Better parallel memory warnings (GH#208, GH#214)
- Update demo datasets (GH#187, GH#201, GH#207)
- Make primitive lookup case insensitive (GH#213)

- Use capital name (GH#211)
- Set class name for Min (GH#206)
- Remove `variable_types` from normalize entity (GH#205)
- Handle parquet serialization with last time index (GH#204)
- Reset index of cutoff times in calculate feature matrix (GH#198)
- Check argument types for `.normalize_entity` (GH#195)
- Type checking ignore entities. (GH#193)

v0.2.1 July 2, 2018

- Cpu count fix (GH#176)
- Update flight (GH#175)
- Move feature matrix calculation helper functions to separate file (GH#177)

v0.2.0 June 22, 2018

- Multiprocessing (GH#170)
- Handle unicode encoding in repr throughout Featuretools (GH#161)
- Clean up EntitySet class (GH#145)
- Add support for building and uploading conda package (GH#167)
- Parquet serialization (GH#152)
- Remove variable stats (GH#171)
- Make sure index variable comes first (GH#168)
- No last time index update on normalize (GH#169)
- Remove list of times as an option for `cutoff_time` in `calculate_feature_matrix` (GH#165)
- Config does error checking to see if it can write to disk (GH#162)

v0.1.21 May 30, 2018

- Support Pandas 0.23.0 (GH#153, GH#154, GH#155, GH#159)
- No EntitySet required in loading/saving features (GH#141)
- Use s3 demo csv with better column names (GH#139)
- more reasonable start parameter (GH#149)
- add issue template (GH#133)
- Improve tests (GH#136, GH#137, GH#144, GH#147)
- Remove unused functions (GH#140, GH#143, GH#146)
- Update documentation after recent changes / removals (GH#157)
- Rename demo retail csv file (GH#148)
- Add names for binary (GH#142)
- EntitySet repr to use `get_name` rather than `id` (GH#134)
- Ensure config dir is writable (GH#135)

v0.1.20 Apr 13, 2018

- Primitives as strings in DFS parameters (GH#129)
- Integer time index bugfixes (GH#128)
- Add `make_temporal_cutoffs` utility function (GH#126)
- Show all entities, switch shape display to row/col (GH#124)
- Improved chunking when calculating feature matrices (GH#121)
- fixed num characters nan fix (GH#118)
- modify `ignore_variables` docstring (GH#117)

v0.1.19 Mar 21, 2018

- More descriptive DFS progress bar (GH#69)
- Convert text variable to string before `NumWords` (GH#106)
- `EntitySet.concat()` reindexes relationships (GH#96)
- Keep non-feature columns when encoding feature matrix (GH#111)
- Uses full entity update for dependencies of `uses_full_entity` features (GH#110)
- Update column names in retail demo (GH#104)
- Handle Transform features that need access to all values of entity (GH#91)

v0.1.18 Feb 27, 2018

- fixes related instances bug (GH#97)
- Adding non-feature columns to calculated feature matrix (GH#78)
- Relax numpy version req (GH#82)
- Remove `entity_from_csv`, tests, and lint (GH#71)

v0.1.17 Jan 18, 2018

- LatLong type (GH#57)
- Last time index fixes (GH#70)
- Make median agg primitives ignore nans by default (GH#61)
- Remove Python 3.4 support (GH#64)
- Change `normalize_entity` to update `secondary_time_index` (GH#59)
- Unpin requirements (GH#53)
- associative -> commutative (GH#56)
- Add Words and Chars primitives (GH#51)

v0.1.16 Dec 19, 2017

- fix `EntitySet.combine_variables` and standardize `encode_features` (GH#47)
- Python 3 compatibility (GH#16)

v0.1.15 Dec 18, 2017

- Fix variable type in demo data (GH#37)
- Custom primitive kwarg fix (GH#38)
- Changed order and text of arguments in `make_trans_primitive` docstring (GH#42)

v0.1.14 November 20, 2017

- Last time index (GH#33)
- Update Scipy version to 1.0.0 (GH#31)

v0.1.13 November 1, 2017

- Add MANIFEST.in (GH#26)

v0.1.11 October 31, 2017

- Package linting (GH#7)
- Custom primitive creation functions (GH#13)
- Split requirements to separate files and pin to latest versions (GH#15)
- Select low information features (GH#18)
- Fix docs typos (GH#19)
- Fixed Diff primitive for rare nan case (GH#21)
- added some missing doc strings (GH#23)
- Trend fix (GH#22)
- Remove as_dir=False option from EntitySet.to_pickle() (GH#20)
- Entity Normalization Preserves Types of Copy & Additional Variables (GH#25)

v0.1.10 October 12, 2017

- NumTrue primitive added and docstring of other primitives updated (GH#11)
- fixed hash issue with same base features (GH#8)
- Head fix (GH#9)
- Fix training window (GH#10)
- Add associative attribute to primitives (GH#3)
- Add status badges, fix license in setup.py (GH#1)
- fixed head printout and flight demo index (GH#2)

v0.1.9 September 8, 2017

- Documentation improvements
- New `featuretools.demo.load_mock_customer` function

v0.1.8 September 1, 2017

- Bug fixes
- Added `Percentile` transform primitive

v0.1.7 August 17, 2017

- Performance improvements for approximate in `calculate_feature_matrix` and `dfs`
- Added `Week` transform primitive

v0.1.6 July 26, 2017

- Added `load_features` and `save_features` to persist and reload features
- Added `save_progress` argument to `calculate_feature_matrix`

- Added approximate parameter to `calculate_feature_matrix` and `dfs`
- Added `load_flight` to `ft.demo`

v0.1.5 July 11, 2017

- Windows support

v0.1.3 July 10, 2017

- Renamed feature submodule to `primitives`
- Renamed `prediction_entity` arguments to `target_entity`
- Added `training_window` parameter to `calculate_feature_matrix`

v0.1.2 July 3rd, 2017

- Initial release

3.20 Feature types

Featuretools groups features into four general types:

- *Identity features*
- *Transform and Cumulative Transform features*
- *Aggregation features*
- *Direct features*

3.20.1 Identity Features

In Featuretools, each feature is defined as a combination of other features. At the lowest level are `IdentityFeature` features which are equal to the value of a single variable.

Most of the time, identity features will be defined transparently for you, such as in the transform feature example below. They may also be defined explicitly:

```
In [1]: time_feature = ft.Feature(es["transactions"]["transaction_time"])
In [2]: time_feature
Out [2]: <Feature: transaction_time>
```

3.20.2 Direct Features

Direct features are used to “inherit” feature values from a parent to a child entity. Suppose each event is associated with a single instance of the entity *products*. This entity has metadata about different products, such as brand, price, etc. We can pull the brand of the product into a feature of the event entity by including the event entity as an argument to `Feature`. In this case, `Feature` is an alias for `primitives.DirectFeature`:

```
In [3]: brand = ft.Feature(es["products"]["brand"], entity=es["transactions"])
In [4]: brand
Out [4]: <Feature: products.brand>
```

3.20.3 Transform Features

Transform features take one or more features on an *Entity* and create a single new feature for that same entity. For example, we may want to take a fine-grained “timestamp” feature and convert it into the hour of the day in which it occurred.

```
In [5]: from featuretools.primitives import Hour

In [6]: ft.Feature(time_feature, primitive=Hour)
Out[6]: <Feature: HOUR(transaction_time)>
```

Using algebraic and boolean operations, transform features can combine other features into arbitrary expressions. For example, to determine if a given event happened in the afternoon, we can write:

```
In [7]: hour_feature = ft.Feature(time_feature, primitive=Hour)

In [8]: after_twelve = hour_feature > 12

In [9]: after_twelve
Out[9]: <Feature: HOUR(transaction_time) > 12>

In [10]: at_twelve = hour_feature == 12

In [11]: before_five = hour_feature <= 17

In [12]: is_afternoon = after_twelve & before_five

In [13]: is_afternoon
Out[13]: <Feature: AND(HOUR(transaction_time) > 12, HOUR(transaction_time) <= 17)>
```

3.20.4 Aggregation Features

Aggregation features are used to create features for a *parent entity* by summarizing data from a *child entity*. For example, we can create a *Count* feature which counts the total number of events for each customer:

```
In [14]: from featuretools.primitives import Count

In [15]: total_events = ft.Feature(es["transactions"]["transaction_id"], parent_
->entity=es["customers"], primitive=Count)

In [16]: fm = ft.calculate_feature_matrix([total_events], es)

In [17]: fm.head()
Out[17]:
```

customer_id	COUNT(transactions)
5	79
4	109
1	126
3	93
2	93

Note: For users who have written aggregations in SQL, this concept will be familiar. One key difference in featuretools is that `GROUP BY` and `JOIN` are implicit. Since the parent and child entities are specified, featuretools can infer

how to group the child entity and then join the resulting aggregation back to the parent entity.

Often times, we only want to aggregate using a certain amount of previous data. For example, we might only want to count events from the past 30 days. In this case, we can provide the `use_previous` parameter:

```
In [18]: total_events_last_30_days = ft.Feature(es["transactions"]["transaction_id"],
.....:                                     parent_entity=es["customers"],
.....:                                     use_previous="30 days",
.....:                                     primitive=Count)
.....:

In [19]: fm = ft.calculate_feature_matrix([total_events_last_30_days], es)

In [20]: fm.head()
Out [20]:
```

customer_id	COUNT(transactions, Last 30 Days)
5	0.0
4	0.0
1	0.0
3	0.0
2	0.0

Unlike with cumulative transform features, the `use_previous` parameter here is evaluated relative to instances of the parent entity, not the child entity. The above feature translates roughly to the following: “For each customer, count the events which occurred in the 30 days preceding the customer’s timestamp.”

Find the list of the supported aggregation features [here](#).

3.20.5 Where clauses

When defining aggregation or cumulative transform features, we can provide a `where` parameter to filter the instances we are aggregating over. Using the `is_afternoon` feature from *earlier*, we can count the total number of events which occurred in the afternoon:

```
In [21]: afternoon_events = ft.Feature(es["transactions"]["transaction_id"],
.....:                               parent_entity=es["customers"],
.....:                               where=is_afternoon,
.....:                               primitive=Count).rename("afternoon_events")
.....:

In [22]: fm = ft.calculate_feature_matrix([afternoon_events], es)

In [23]: fm.head()
Out [23]:
```

customer_id	afternoon_events
5	0.0
4	0.0
1	0.0
3	0.0
2	0.0

The `where` argument can be any previously-defined boolean feature. Only instances for which the `where` feature is `True` are included in the final calculation.

3.20.6 Aggregations of Direct Feature

Composing multiple feature types is an extremely powerful abstraction that Featuretools makes simple. For instance, we can aggregate direct features on a child entity from a different parent entity. For example, to calculate the most common brand a customer interacted with:

```
In [24]: from featuretools.primitives import Mode

In [25]: brand = ft.Feature(es["products"]["brand"], entity=es["transactions"])

In [26]: favorite_brand = ft.Feature(brand, parent_entity=es["customers"],
↳primitive=Mode)

In [27]: fm = ft.calculate_feature_matrix([favorite_brand], es)

In [28]: fm.head()
Out[28]:
           MODE(transactions.products.brand)
customer_id
5                B
4                B
1                B
3                B
2                B
```

Side note: Feature equality overrides default equality

Because we can check if two features are equal (or a feature is equal to a value), we override Python's equals (==) operator. This means to check if two feature objects are equal (instead of their computed values in the feature matrix), we need to compare their hashes:

```
In [29]: hour_feature.hash() == hour_feature.hash()
Out[29]: True

In [30]: hour_feature.hash() != hour_feature.hash()
Out[30]: False
```

dictionaries and sets use equality underneath, so those keys need to be hashes as well

```
In [31]: myset = set()

In [32]: myset.add(hour_feature.hash())

In [33]: hour_feature.hash() in myset
Out[33]: True

In [34]: mydict = dict()

In [35]: mydict[hour_feature.hash()] = hour_feature

In [36]: hour_feature.hash() in mydict
Out[36]: True
```

3.21 Save Intermediate Feature Matrix Results

In this tutorial, we will go over the how to save intermediate results when computing the feature matrix.

```
[1]: import featuretools as ft
```

In this example, we will use a dataset of retail data of customers from a UK website from December 2010 to December 2011.

```
[2]: es = ft.demo.load_retail(nrows=10000)
```

let's use a simple feature for this example.

```
[3]: region = ft.Feature(es["customers"]["Country"])
```

We can supply “cutoff times” to specify that we want to calculate features one year after a customer’s first invoice.

```
[4]: import pandas as pd
cutoff_times = es["customers"].df[["CustomerID", "first_invoices_time"]].rename(
    columns={"CustomerID": "instance_id", "first_invoices_time": "time"})
cutoff_times["time"] = cutoff_times["time"] + pd.Timedelta("365 days")
```

Here is what some of the cutoff times look like.

```
[5]: cutoff_times.head(10)
```

```
[5]:
```

CustomerID	instance_id	time
17850.0	17850.0	2011-12-01 08:26:00
13047.0	13047.0	2011-12-01 08:34:00
12583.0	12583.0	2011-12-01 08:45:00
13748.0	13748.0	2011-12-01 09:00:00
15100.0	15100.0	2011-12-01 09:09:00
15291.0	15291.0	2011-12-01 09:32:00
14688.0	14688.0	2011-12-01 09:37:00
14527.0	14527.0	2011-12-01 09:41:00
15311.0	15311.0	2011-12-01 09:41:00
17809.0	17809.0	2011-12-01 09:41:00

If you want to save intermediate computations as CSVs, simply pass the location of a directory of where the computation should be saved. For example, if you pass a directory called “ft_temp”, CSV files will be output to the directory, named according to the timestamp that it represents.

```
[6]: import os
save_progress = os.path.join(os.getcwd(), 'ft_temp')
if not os.path.exists(save_progress):
    os.makedirs(save_progress)
```

```
[7]: fm_save = ft.calculate_feature_matrix([region],
    entityset=es,
    cutoff_time=cutoff_times.sample(10),
    save_progress=save_progress)
```

As seen below, there are now files in the directory, named by timestamp.

```
[8]: % ls ft_temp/
```

```
ft_2011_12_01_03-08-00-000000.csv ft_2011_12_02_05-03-00-000000.csv
ft_2011_12_01_09-00-00-000000.csv ft_2011_12_02_05-19-00-000000.csv
ft_2011_12_01_12-43-00-000000.csv ft_2011_12_02_12-07-00-000000.csv
ft_2011_12_01_12-51-00-000000.csv ft_2011_12_02_12-18-00-000000.csv
ft_2011_12_02_03-19-00-000000.csv ft_2011_12_03_12-57-00-000000.csv
```

```
[9]: import shutil
      shutil.rmtree(save_progress)
```

OTHER LINKS

- [genindex](#)
- [search](#)

Symbols

- `__getitem__()` (*featuretools.entityset.EntitySet method*), 215
- `__init__()` (*featuretools.Entity method*), 209
- `__init__()` (*featuretools.EntitySet method*), 207
- `__init__()` (*featuretools.Relationship method*), 210
- `__init__()` (*featuretools.Timedelta method*), 142
- `__init__()` (*featuretools.primitives.Absolute method*), 170
- `__init__()` (*featuretools.primitives.AggregationPrimitive method*), 144
- `__init__()` (*featuretools.primitives.All method*), 160
- `__init__()` (*featuretools.primitives.And method*), 167
- `__init__()` (*featuretools.primitives.Any method*), 161
- `__init__()` (*featuretools.primitives.AvgTimeBetween method*), 155
- `__init__()` (*featuretools.primitives.Count method*), 148
- `__init__()` (*featuretools.primitives.CumCount method*), 183
- `__init__()` (*featuretools.primitives.CumMax method*), 186
- `__init__()` (*featuretools.primitives.CumMean method*), 185
- `__init__()` (*featuretools.primitives.CumMin method*), 185
- `__init__()` (*featuretools.primitives.CumSum method*), 184
- `__init__()` (*featuretools.primitives.Day method*), 177
- `__init__()` (*featuretools.primitives.Diff method*), 181
- `__init__()` (*featuretools.primitives.Entropy method*), 165
- `__init__()` (*featuretools.primitives.First method*), 162
- `__init__()` (*featuretools.primitives.Haversine method*), 191
- `__init__()` (*featuretools.primitives.Hour method*), 176
- `__init__()` (*featuretools.primitives.IsIn method*), 166
- `__init__()` (*featuretools.primitives.IsWeekend method*), 176
- `__init__()` (*featuretools.primitives.Last method*), 163
- `__init__()` (*featuretools.primitives.Latitude method*), 189
- `__init__()` (*featuretools.primitives.Longitude method*), 190
- `__init__()` (*featuretools.primitives.Max method*), 151
- `__init__()` (*featuretools.primitives.Mean method*), 149
- `__init__()` (*featuretools.primitives.Median method*), 153
- `__init__()` (*featuretools.primitives.Min method*), 151
- `__init__()` (*featuretools.primitives.Minute method*), 174
- `__init__()` (*featuretools.primitives.Mode method*), 154
- `__init__()` (*featuretools.primitives.Month method*), 179
- `__init__()` (*featuretools.primitives.Not method*), 169
- `__init__()` (*featuretools.primitives.NumCharacters method*), 187
- `__init__()` (*featuretools.primitives.NumUnique method*), 158
- `__init__()` (*featuretools.primitives.NumWords method*), 188
- `__init__()` (*featuretools.primitives.Or method*), 168
- `__init__()` (*featuretools.primitives.PercentTrue method*), 159
- `__init__()` (*featuretools.primitives.Percentile method*), 171
- `__init__()` (*featuretools.primitives.Second method*), 173
- `__init__()` (*featuretools.primitives.Skew method*), 163
- `__init__()` (*featuretools.primitives.Std method*), 152
- `__init__()` (*featuretools.primitives.Sum method*), 150
- `__init__()` (*featuretools.primitives.TimeSince method*), 172
- `__init__()` (*featuretools.primitives.TimeSinceFirst method*), 157
- `__init__()` (*featuretools.primitives.TimeSinceLast method*), 156
- `__init__()` (*feature-*

tools.primitives.TimeSincePrevious method), 182
 __init__() (*featuretools.primitives.TransformPrimitive* method), 143
 __init__() (*featuretools.primitives.Trend* method), 164
 __init__() (*featuretools.primitives.Week* method), 178
 __init__() (*featuretools.primitives.Weekday* method), 175
 __init__() (*featuretools.primitives.Year* method), 180
 __init__() (*featuretools.variable_types.Boolean* method), 224
 __init__() (*featuretools.variable_types.Categorical* method), 223
 __init__() (*featuretools.variable_types.CountryCode* method), 229
 __init__() (*featuretools.variable_types.DateOfBirth* method), 228
 __init__() (*featuretools.variable_types.Datetime* method), 221
 __init__() (*featuretools.variable_types.DatetimeTimeIndex* method), 220
 __init__() (*featuretools.variable_types.EmailAddress* method), 227
 __init__() (*featuretools.variable_types.FilePath* method), 230
 __init__() (*featuretools.variable_types.FullName* method), 226
 __init__() (*featuretools.variable_types.IPAddress* method), 226
 __init__() (*featuretools.variable_types.Id* method), 219
 __init__() (*featuretools.variable_types.Index* method), 219
 __init__() (*featuretools.variable_types.LatLong* method), 225
 __init__() (*featuretools.variable_types.Numeric* method), 222
 __init__() (*featuretools.variable_types.NumericTimeIndex* method), 221
 __init__() (*featuretools.variable_types.Ordinal* method), 223
 __init__() (*featuretools.variable_types.PhoneNumber* method), 228
 __init__() (*featuretools.variable_types.SubRegionCode* method), 229
 __init__() (*featuretools.variable_types.Text* method), 224
 __init__() (*featuretools.variable_types.TimeIndex* method), 220
 __init__() (*featuretools.variable_types.URL* method), 227
 __init__() (*featuretools.variable_types.ZIPCode* method), 225
 __init__() (*featuretools.wrappers.DFSTransformer* method), 138
 __init__() (*nlp_primitives.DiversityScore* method), 192
 __init__() (*nlp_primitives.LSA* method), 193
 __init__() (*nlp_primitives.MeanCharactersPerWord* method), 194
 __init__() (*nlp_primitives.PartOfSpeechCount* method), 195
 __init__() (*nlp_primitives.PolarityScore* method), 196
 __init__() (*nlp_primitives.PunctuationCount* method), 197
 __init__() (*nlp_primitives.StopwordCount* method), 198
 __init__() (*nlp_primitives.TitleWordCount* method), 199
 __init__() (*nlp_primitives.UniversalSentenceEncoder* method), 200
 __init__() (*nlp_primitives.UpperCaseCount* method), 201

A

Absolute (*class in featuretools.primitives*), 170
 add_interesting_values() (*featuretools.EntitySet* method), 213
 add_interesting_values() (*featuretools.entityset.Entity* method), 217
 add_relationship() (*featuretools.EntitySet* method), 212
 AggregationPrimitive (*class in featuretools.primitives*), 144
 All (*class in featuretools.primitives*), 160
 And (*class in featuretools.primitives*), 167
 Any (*class in featuretools.primitives*), 161
 AvgTimeBetween (*class in featuretools.primitives*), 155

B

Boolean (*class in featuretools.variable_types*), 224

C

calculate_feature_matrix() (*in module featuretools*), 202

- Categorical (class in *featuretools.variable_types*), 223
- child entity, **130**
- child_entity (*featuretools.entityset.Relationship attribute*), 218
- child_variable (*featuretools.entityset.Relationship attribute*), 218
- convert_variable_type() (*featuretools.entityset.Entity method*), 217
- Count (class in *featuretools.primitives*), 148
- count (*featuretools.variable_types.Index attribute*), 219
- CountryCode (class in *featuretools.variable_types*), 229
- CumCount (class in *featuretools.primitives*), 183
- CumMax (class in *featuretools.primitives*), 186
- CumMean (class in *featuretools.primitives*), 184
- CumMin (class in *featuretools.primitives*), 185
- CumSum (class in *featuretools.primitives*), 183
- cutoff time, **130**
- ## D
- DateOfBirth (class in *featuretools.variable_types*), 228
- Datetime (class in *featuretools.variable_types*), 221
- DatetimeTimeIndex (class in *featuretools.variable_types*), 220
- Day (class in *featuretools.primitives*), 177
- dfs() (in module *featuretools*), 135
- DFSTransformer (class in *featuretools.wrappers*), 138
- Diff (class in *featuretools.primitives*), 181
- DiversityScore (class in *nlp_primitives*), 192
- ## E
- EmailAddress (class in *featuretools.variable_types*), 227
- encode_features() (in module *featuretools*), 203
- entity, **130**
- Entity (class in *featuretools*), 208
- entity_dict (*featuretools.EntitySet attribute*), 207
- entity_from_dataframe() (*featuretools.EntitySet method*), 211
- EntitySet, **130**
- EntitySet (class in *featuretools*), 207
- Entropy (class in *featuretools.primitives*), 165
- ## F
- feature, **131**
- feature engineering, **131**
- FilePath (class in *featuretools.variable_types*), 230
- find_backward_paths() (*featuretools.entityset.EntitySet method*), 215
- find_forward_paths() (*featuretools.entityset.EntitySet method*), 215
- First (class in *featuretools.primitives*), 162
- FullName (class in *featuretools.variable_types*), 226
- ## G
- get_backward_entities() (*featuretools.entityset.EntitySet method*), 216
- get_depth() (*featuretools.feature_base.FeatureBase method*), 201
- get_forward_entities() (*featuretools.entityset.EntitySet method*), 216
- ## H
- Haversine (class in *featuretools.primitives*), 191
- Hour (class in *featuretools.primitives*), 176
- ## I
- Id (class in *featuretools.variable_types*), 219
- id (*featuretools.EntitySet attribute*), 207
- Index (class in *featuretools.variable_types*), 219
- instance, **131**
- IPAddress (class in *featuretools.variable_types*), 226
- IsIn (class in *featuretools.primitives*), 166
- IsWeekend (class in *featuretools.primitives*), 175
- ## L
- Last (class in *featuretools.primitives*), 162
- Latitude (class in *featuretools.primitives*), 189
- LatLong (class in *featuretools.variable_types*), 225
- load_features() (in module *featuretools*), 206
- load_flight() (in module *featuretools.demo*), 134
- load_mock_customer() (in module *featuretools.demo*), 134
- load_retail() (in module *featuretools.demo*), 133
- Longitude (class in *featuretools.primitives*), 190
- LSA (class in *nlp_primitives*), 193
- ## M
- make_agg_primitive() (in module *featuretools.primitives*), 145
- make_temporal_cutoffs() (in module *featuretools*), 143
- make_trans_primitive() (in module *featuretools.primitives*), 146
- Max (class in *featuretools.primitives*), 151
- max (*featuretools.variable_types.Numeric attribute*), 222
- Mean (class in *featuretools.primitives*), 149
- mean (*featuretools.variable_types.Numeric attribute*), 222
- MeanCharactersPerWord (class in *nlp_primitives*), 194
- Median (class in *featuretools.primitives*), 153
- Min (class in *featuretools.primitives*), 150
- min (*featuretools.variable_types.Numeric attribute*), 222

Minute (class in *featuretools.primitives*), 174
 Mode (class in *featuretools.primitives*), 154
 Month (class in *featuretools.primitives*), 179

N

normalize_entity() (featuretools.EntitySet method), 212
 Not (class in *featuretools.primitives*), 169
 NumCharacters (class in *featuretools.primitives*), 187
 Numeric (class in *featuretools.variable_types*), 222
 NumericTimeIndex (class in *featuretools.variable_types*), 221
 NumUnique (class in *featuretools.primitives*), 158
 NumWords (class in *featuretools.primitives*), 188

O

Or (class in *featuretools.primitives*), 168
 Ordinal (class in *featuretools.variable_types*), 223

P

parent entity, 131
 parent_entity (featuretools.entityset.Relationship attribute), 218
 parent_variable (featuretools.entityset.Relationship attribute), 217
 PartOfSpeechCount (class in *nlp_primitives*), 195
 Percentile (class in *featuretools.primitives*), 170
 PercentTrue (class in *featuretools.primitives*), 159
 PhoneNumber (class in *featuretools.variable_types*), 228
 plot() (featuretools.entityset.EntitySet method), 216
 PolarityScore (class in *nlp_primitives*), 196
 PunctuationCount (class in *nlp_primitives*), 197

R

read_entityset() (in module *featuretools*), 213
 relationship, 131
 Relationship (class in *featuretools*), 210
 relationships (featuretools.EntitySet attribute), 207
 remove_low_information_features() (in module *featuretools.selection*), 230
 rename() (featuretools.feature_base.FeatureBase method), 201

S

save_features() (in module *featuretools*), 205
 Second (class in *featuretools.primitives*), 173
 Skew (class in *featuretools.primitives*), 163
 Std (class in *featuretools.primitives*), 152
 std (featuretools.variable_types.Numeric attribute), 222
 StopwordCount (class in *nlp_primitives*), 198
 SubRegionCode (class in *featuretools.variable_types*), 229

Sum (class in *featuretools.primitives*), 150

T

target entity, 131
 Text (class in *featuretools.variable_types*), 224
 time_type (featuretools.EntitySet attribute), 207
 Timedelta (class in *featuretools*), 141
 TimeIndex (class in *featuretools.variable_types*), 220
 TimeSince (class in *featuretools.primitives*), 171
 TimeSinceFirst (class in *featuretools.primitives*), 157
 TimeSinceLast (class in *featuretools.primitives*), 156
 TimeSincePrevious (class in *featuretools.primitives*), 182
 TitleWordCount (class in *nlp_primitives*), 199
 to_csv() (featuretools.entityset.EntitySet method), 214
 to_parquet() (featuretools.entityset.EntitySet method), 214
 to_pickle() (featuretools.entityset.EntitySet method), 214
 TransformPrimitive (class in *featuretools.primitives*), 143
 Trend (class in *featuretools.primitives*), 164

U

UniversalSentenceEncoder (class in *nlp_primitives*), 199
 UpperCaseCount (class in *nlp_primitives*), 200
 URL (class in *featuretools.variable_types*), 227

V

variable, 131

W

Week (class in *featuretools.primitives*), 178
 Weekday (class in *featuretools.primitives*), 174

Y

Year (class in *featuretools.primitives*), 180

Z

ZIPCode (class in *featuretools.variable_types*), 225