
Compose Documentation

Release 0.1.4

Feature Labs, Inc.

Aug 23, 2019

Table of Contents

1	Install	3
2	Getting Started	5
2.1	Load Data	5
2.2	Create Labeling Function	6
2.3	Construct Label Maker	6
2.4	Generate Labels	6
2.5	Transform Labels	6
2.6	Describe Labels	7
2.7	Plot Labels	8
3	Main Concepts	11
3.1	Label Maker	11
4	Using Compose with Featuretools	13
4.1	Load Data	13
4.2	Generate Labels	14
4.3	Generate Features	16
4.4	Apply Machine Learning	18
5	Using Label Transforms	23
5.1	Generate Labels	23
5.2	Threshold on Labels	24
5.3	Lead Labels Times	24
5.4	Bin Labels	24
5.5	Describe Labels	26
6	API Reference	29
6.1	Label Maker	29
6.2	Label Times	31
7	Changelog	37
	Index	39



Creating labels from raw data for a machine learning problem is difficult and time consuming. This is where *Compose* helps by making it easier to quickly generate complex labels from raw data.

Compose is advanced software for automating the prediction engineering process. Compose enables you to systematically define prediction problems by automatically extracting historical training examples to train machine learning algorithms.

CHAPTER 1

Install

Compose is available for Python ≥ 3.5 . To install, use `pip` by running the following command:

```
pip install compose1
```


In this example, we will generate labels on a mock dataset of transactions. For each customer, we want to label whether the total purchase amount over the next hour of transactions will exceed 100. Additionally, we want to predict one hour in advance.

2.1 Load Data

With the package installed, we load in the data. To get an idea on how the transactions looks, we preview the data frame.

```
[1]: import composeml as cp

df = cp.datasets.transactions()

df[df.columns[:7]].head()
```

```
[1]:
```

	transaction_id	session_id	product_id	amount	\
transaction_time					
2014-01-01 03:13:51	190	14	5	120.52	
2014-01-01 11:04:42	350	19	3	65.43	
2014-01-02 11:44:35	254	11	5	128.51	
2014-01-02 17:12:39	337	16	2	105.15	
2014-01-02 17:46:20	177	29	5	65.11	

	customer_id
transaction_time	
2014-01-01 03:13:51	1
2014-01-01 11:04:42	3
2014-01-02 11:44:35	4
2014-01-02 17:12:39	2
2014-01-02 17:46:20	1

2.2 Create Labeling Function

First, we define the function that will return the total purchase amount given a hour of transactions.

```
[2]: def my_labeling_function(df_slice):
      label = df_slice["amount"].sum()
      return label
```

2.3 Construct Label Maker

With the labeling function, we create the *LabelMaker* for our prediction problem. We need an hour of transactions for each label, so we set `window_size` to one hour.

```
[3]: label_maker = cp.LabelMaker(
      target_entity="customer_id",
      time_index="transaction_time",
      labeling_function=my_labeling_function,
      window_size="1h",
  )
```

2.4 Generate Labels

Next, we automatically search and extract the labels by using *LabelMaker.search()*.

```
[4]: labels = label_maker.search(
      df,
      minimum_data="1h",
      num_examples_per_instance=25,
      gap=1,
      verbose=True,
  )
```

```
labels.head()
```

```
Elapsed: 00:00 | Remaining: 00:00 | Progress: 100%|| customer_id: 125/125
```

```
[4]:
```

	customer_id	cutoff_time	my_labeling_function
label_id			
0	1	2014-01-01 04:13:51	65.11
1	1	2014-01-03 15:41:34	101.08
2	1	2014-01-05 11:46:10	16.78
3	1	2014-01-06 09:54:58	108.16
4	1	2014-01-08 08:54:02	48.33

2.5 Transform Labels

With the generated *LabelTimes*, we will apply specific transforms for our prediction problem.

2.5.1 Apply Threshold on Labels

We apply `LabelTimes.threshold()` to make the labels binary for totaled amounts exceeding 100.

```
[5]: labels = labels.threshold(100)

labels.head()

[5]:
```

label_id	customer_id	cutoff_time	my_labeling_function
0	1	2014-01-01 04:13:51	False
1	1	2014-01-03 15:41:34	True
2	1	2014-01-05 11:46:10	False
3	1	2014-01-06 09:54:58	True
4	1	2014-01-08 08:54:02	False

2.5.2 Lead Label Times

Lastly, we use `LabelTimes.apply_lead()` to shift the label times 1 hour earlier for predicting in advance.

```
[6]: labels = labels.apply_lead('1h')

labels.head()

[6]:
```

label_id	customer_id	cutoff_time	my_labeling_function
0	1	2014-01-01 03:13:51	False
1	1	2014-01-03 14:41:34	True
2	1	2014-01-05 10:46:10	False
3	1	2014-01-06 08:54:58	True
4	1	2014-01-08 07:54:02	False

2.6 Describe Labels

We could use `LabelTimes.describe()` to get the steps and settings used to make the labels.

```
[7]: labels.describe()

Label Distribution
-----
False          75
True           50
Total:        125

Settings
-----
num_examples_per_instance    25
minimum_data                 1h
window_size                  1h
gap                           1

Transforms
-----
```

(continues on next page)

(continued from previous page)

```
1. threshold
   - value: 100

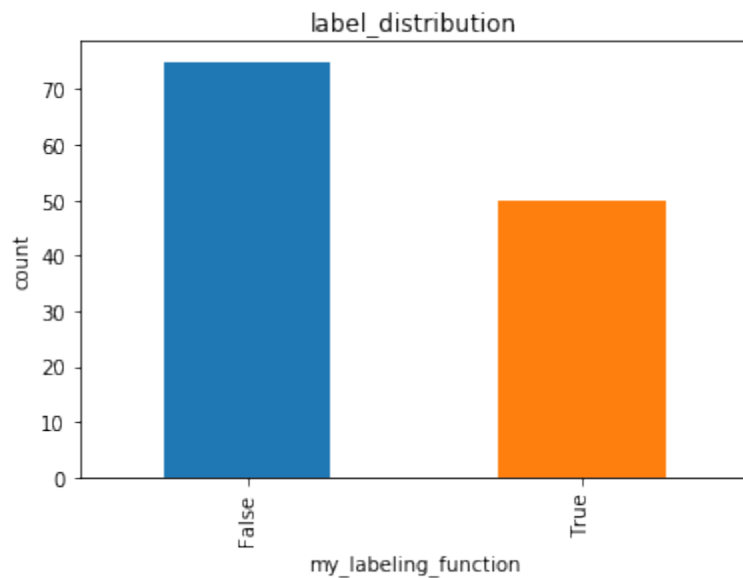
2. apply_lead
   - value: 1h
```

2.7 Plot Labels

Also, there are plots available for insight to the labels.

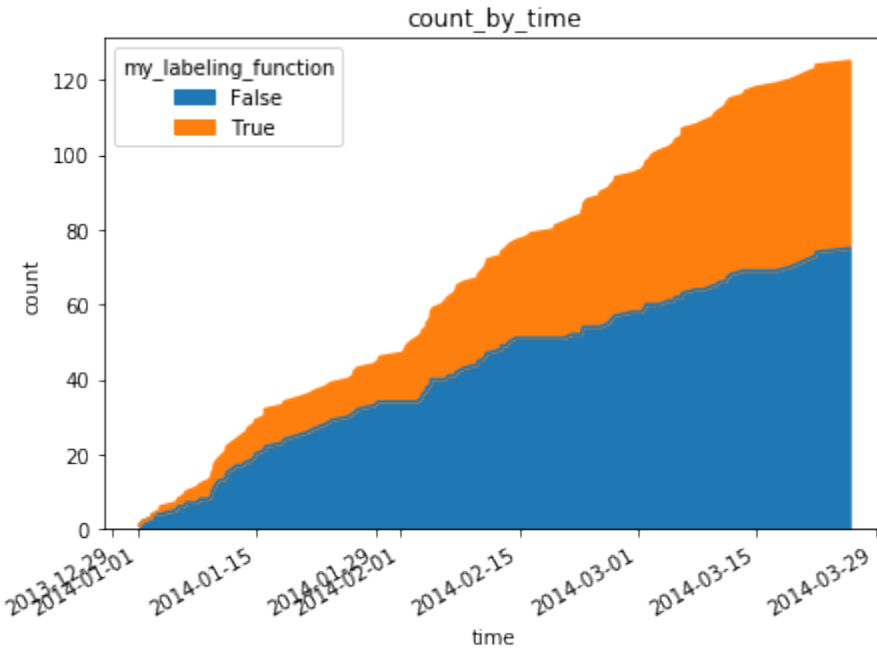
2.7.1 Label Distribution

```
labels.plot.distribution()
```



2.7.2 Label Count vs. Time

```
labels.plot.count_by_time(figsize=(7, 5))
```



3.1 Label Maker

The label maker automatically extracts data along the time index to generate labels. The process starts by setting the first cutoff time after the minimum amount of data. Then subsequent cutoff times are spaced apart using **gaps**. Starting from each cutoff time, a window determines the amount of data, also referred to as a **data slice**, to pass into a labeling function.

The labeling function will then transform the extracted data slice into a label.

In cases where the labeling function returned continuous values, there are label transforms available to further process the labels into discrete values.

Using Compose with Featuretools

In this guide, we will generate labels and features on a mock dataset of transactions using Compose and Featuretools. Then create a machine learning model for predicting one hour in advance whether customers will spend over \$1200 within the next hour of transactions.

```
[1]: %matplotlib inline
import composeml as cp
import featuretools as ft
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import classification_report
```

4.1 Load Data

To get an idea on how the transactions looks, we preview the data frame.

```
[2]: transactions = ft.demo.load_mock_customer(
    return_single_table=True,
    random_seed=0,
)

transactions[transactions.columns[:7]].head()
```

```
[2]: 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
0            2  desktop
```

(continues on next page)

(continued from previous page)

```
1         2 desktop
2         2 desktop
3         2 tablet
4         2 tablet
```

4.2 Generate Labels

Now with the transactions loaded, we are ready to generate labels for our prediction problem.

4.2.1 Create Labeling Function

First, we define the function that will return the total purchase amount given a hour of transactions.

```
[3]: def total_spent(df_slice):
      label = df_slice["amount"].sum()
      return label
```

4.2.2 Construct Label Maker

With our labeling function, we create the `LabelMaker` for the transactions. The `target_entity` is set to `customer_id` so that the labels are generated for each customer. The `window_size` is set to one hour to process one hour of transactions for a given customer.

```
[4]: label_maker = cp.LabelMaker(
      target_entity='customer_id',
      time_index='transaction_time',
      labeling_function=total_spent,
      window_size='1h',
      )
```

4.2.3 Create Labels

Next, we automatically search and extract the labels by using `LabelMaker.search()`.

See also:

For more details on how the label maker works, see [Main Concepts](#).

```
[5]: labels = label_maker.search(
      transactions,
      minimum_data='2h',
      num_examples_per_instance=50,
      gap='2min',
      )

labels.head()

Elapsed: 00:01 | Remaining: 00:00 | Progress: 100%| customer_id: 250/250
```

```
[5]:
customer_id      cutoff_time  total_spent
label_id
0              1 2014-01-01 02:44:25      1968.91
1              1 2014-01-01 03:30:00      1767.31
2              1 2014-01-01 03:32:10      1628.60
3              1 2014-01-01 03:34:20      1482.73
4              1 2014-01-01 03:36:30      1258.71
```

4.2.4 Transform Labels

With the generated *LabelTimes*, we will apply specific transforms for our prediction problem.

Apply Threshold on Labels

We apply *LabelTimes.threshold()* to make the labels binary for total amounts exceeding \$1200.

```
[6]: labels = labels.threshold(1200)

labels.head()

[6]:
customer_id      cutoff_time  total_spent
label_id
0              1 2014-01-01 02:44:25          True
1              1 2014-01-01 03:30:00          True
2              1 2014-01-01 03:32:10          True
3              1 2014-01-01 03:34:20          True
4              1 2014-01-01 03:36:30          True
```

Lead Label Times

We also use *LabelTimes.apply_lead()* to shift the label times 1 hour earlier for predicting in advance.

```
[7]: labels = labels.apply_lead('1h')

labels.head()

[7]:
customer_id      cutoff_time  total_spent
label_id
0              1 2014-01-01 01:44:25          True
1              1 2014-01-01 02:30:00          True
2              1 2014-01-01 02:32:10          True
3              1 2014-01-01 02:34:20          True
4              1 2014-01-01 02:36:30          True
```

4.2.5 Describe Labels

After transforming the labels, we could use *LabelTimes.describe()* to print out the distribution with the settings and transforms that were used to make the labels. This is useful as a reference for understanding how the labels were generated from raw data. Also, the label distribution is helpful for determining if we have imbalanced labels.

```
[8]: labels.describe()
```

```
Label Distribution
-----
False      93
True       91
Total:    184

Settings
-----
num_examples_per_instance      50
minimum_data                   2h
window_size                    1h
gap                             2min

Transforms
-----
1. threshold
   - value:    1200

2. apply_lead
   - value:    1h
```

4.3 Generate Features

Now with the generated labels, we are ready to generate features for our prediction problem.

4.3.1 Create Entity Set

Let's construct an `EntitySet` and load the transactions as an entity by using `EntitySet.entity_from_dataframe()`. Then extract additional entities by using `EntitySet.normalize_entity()`.

See also:

For more details on working with entity sets, see [Representing Data with EntitySets](#).

```
[9]: es = ft.EntitySet('transactions')

es.entity_from_dataframe(
    'transactions',
    transactions,
    index='transaction_id',
    time_index='transaction_time',
)

es.normalize_entity(
    base_entity_id='transactions',
    new_entity_id='sessions',
    index='session_id',
    make_time_index='session_start',
    additional_variables=[
        'device',
```

(continues on next page)

(continued from previous page)

```

        'customer_id',
        'zip_code',
        'session_start',
        'join_date',
        'date_of_birth',
    ],
)

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',
        'date_of_birth',
    ],
)

es.normalize_entity(
    base_entity_id='transactions',
    new_entity_id='products',
    index='product_id',
    additional_variables=['brand'],
    make_time_index=False,
)

es.add_last_time_indexes()
```

4.3.2 Describe Entity Set

To get information on how the entity set is structured, we could print the entity set and use `EntitySet.plot()` to create a diagram.

```
[10]: print(es, end='\n\n')
```

```
es.plot()
```

```

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

```
[10]:
```

4.3.3 Create Feature Matrix

Next, we generate features that correspond to the labels created previously by using `dfs()`. The `target_entity` is set to `customers` so that features are only calculated for customers. The `cutoff_time` is set to the labels so that features are calculated only using data up to and including the label cutoff times. Notice that the output of Compose integrates easily with Featuretools.

See also:

For more details on calculating features using cutoff times, see [Handling Time](#).

```
[11]: feature_matrix, features_defs = ft.dfs(  
      entityset=es,  
      target_entity='customers',  
      cutoff_time=labels,  
      cutoff_time_in_index=True,  
      verbose=True,  
      )
```

```
Built 73 features
```

```
Elapsed: 00:50 | Remaining: 00:00 | Progress: 100%| Calculated: 11/11 chunks
```

4.3.4 Describe Features

To get an idea on how the generated features look, we preview the feature definitions.

```
[12]: features_defs[:20]
```

```
[12]: [<Feature: zip_code>,  
      <Feature: COUNT(sessions)>,  
      <Feature: NUM_UNIQUE(sessions.device)>,  
      <Feature: MODE(sessions.device)>,  
      <Feature: SUM(transactions.amount)>,  
      <Feature: STD(transactions.amount)>,  
      <Feature: MAX(transactions.amount)>,  
      <Feature: SKEW(transactions.amount)>,  
      <Feature: MIN(transactions.amount)>,  
      <Feature: MEAN(transactions.amount)>,  
      <Feature: COUNT(transactions)>,  
      <Feature: NUM_UNIQUE(transactions.product_id)>,  
      <Feature: MODE(transactions.product_id)>,  
      <Feature: DAY(join_date)>,  
      <Feature: DAY(date_of_birth)>,  
      <Feature: YEAR(join_date)>,  
      <Feature: YEAR(date_of_birth)>,  
      <Feature: MONTH(join_date)>,  
      <Feature: MONTH(date_of_birth)>,  
      <Feature: WEEKDAY(join_date)>]
```

4.4 Apply Machine Learning

Now with the generated labels and features, we are ready to create a machine learning model for our prediction problem.

4.4.1 Preprocess Features

In the feature matrix, let's extract the labels and fill any missing values with zeros. Then, one-hot encode all categorical features by using `encode_features()`.

```
[13]: y = feature_matrix.pop(labels.name)
x = feature_matrix.fillna(0)
x, features_enc = ft.encode_features(x, features_defs)
```

4.4.2 Split Labels and Features

After preprocessing, we split the features and corresponding labels each into training and testing sets.

```
[14]: x_train, x_test, y_train, y_test = train_test_split(
    x,
    y,
    train_size=.8,
    test_size=.2,
    random_state=0,
)
```

4.4.3 Train Model

Next, we train a random forest classifier on the training set.

```
[15]: clf = RandomForestClassifier(n_estimators=10, random_state=0)
clf.fit(x_train, y_train)
[15]: RandomForestClassifier(bootstrap=True, class_weight=None, criterion='gini',
    max_depth=None, max_features='auto', max_leaf_nodes=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=10,
    n_jobs=None, oob_score=False, random_state=0, verbose=0,
    warm_start=False)
```

4.4.4 Test Model

Lastly, we test the model performance by evaluating predictions on the testing set.

```
[16]: y_hat = clf.predict(x_test)
print(classification_report(y_test, y_hat))
```

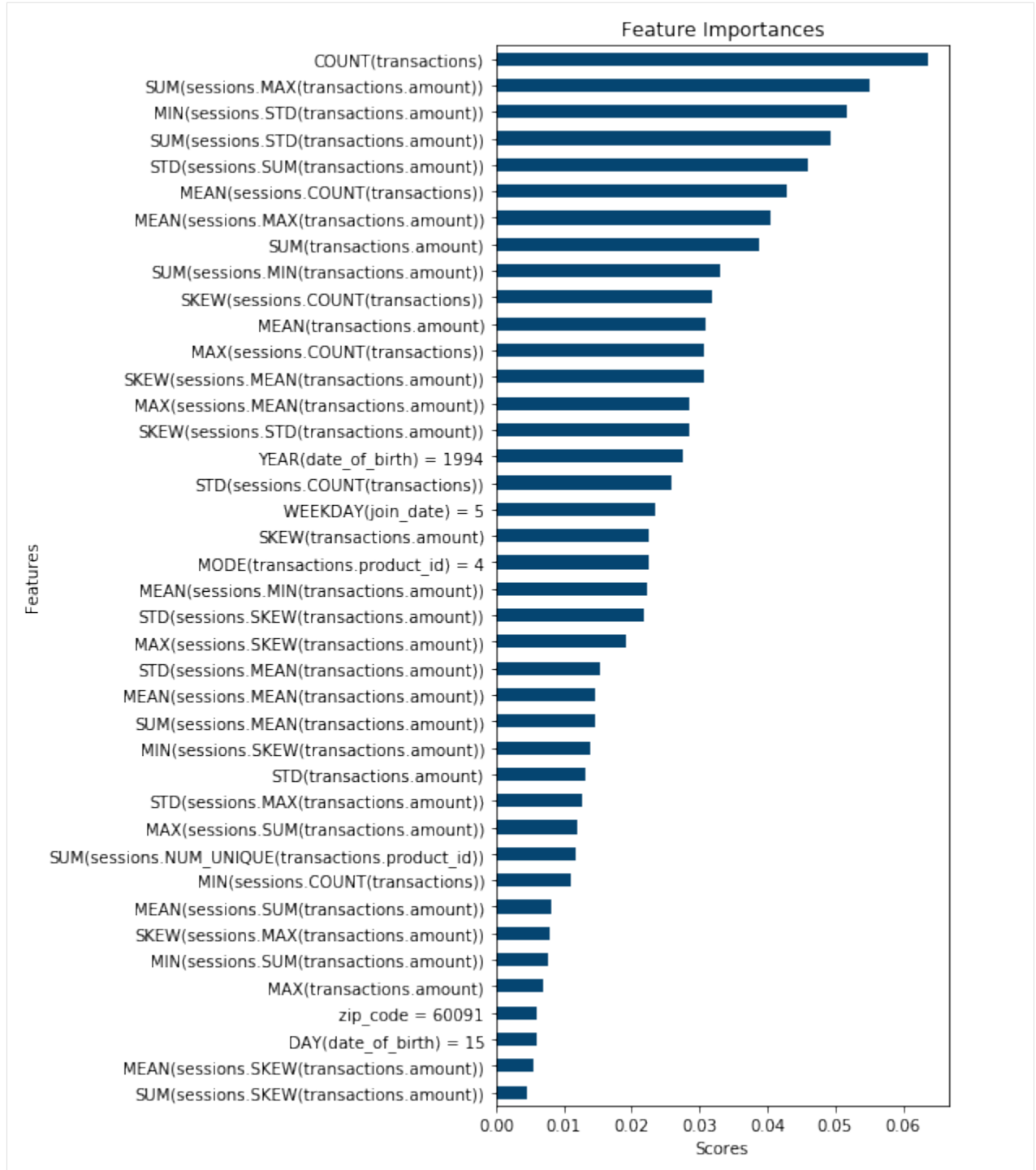
	precision	recall	f1-score	support
False	0.79	0.96	0.86	23
True	0.89	0.57	0.70	14
accuracy			0.81	37
macro avg	0.84	0.76	0.78	37
weighted avg	0.82	0.81	0.80	37

4.4.5 Feature Importances

This plot is based on scores obtained by the model to illustrate which features are considered important for predictions.

```
[17]: feature_importances = zip(x_train.columns, clf.feature_importances_)
feature_importances = pd.Series(dict(feature_importances))
feature_importances = feature_importances.rename_axis('Features')
feature_importances = feature_importances.sort_values()

top_features = feature_importances.tail(40)
plot = top_features.plot(kind='barh', figsize=(5, 12), color='#054571')
plot.set_title('Feature Importances')
plot.set_xlabel('Scores');
```

Using Label Transforms

In this guide, we will demonstrate how to use the transforms that are available on *LabelTimes*. Each transform will return a copy of the label times. This is useful for trying out multiple transforms in different settings without having to recalculate the labels. As a result, we could see which labels give a better performance in less time.

5.1 Generate Labels

Let's start by generating labels on a mock dataset of transactions. Each label is defined as the total spent by a customer given one hour of transactions.

```
[1]: from composeml import datasets, LabelMaker, LabelTimes
```

```
def total_spent(df):  
    return df['amount'].sum()  
  
label_maker = LabelMaker(  
    labeling_function=total_spent,  
    target_entity='customer_id',  
    time_index='transaction_time',  
    window_size='1h',  
)  
  
labels = label_maker.search(  
    datasets.transactions(),  
    num_examples_per_instance=10,  
    minimum_data='2h',  
    gap='2min',  
    verbose=True,  
)
```

```
Elapsed: 00:00 | Remaining: 00:00 | Progress: 100%| customer_id: 50/50
```

To get an idea on how the labels looks, we preview the data frame.

```
[2]: labels.head()
[2]:
```

	customer_id	cutoff_time	total_spent
id			
0	1	2014-01-01 02:45:30	217.94
1	1	2014-01-01 02:47:30	217.94
2	1	2014-01-01 02:49:30	217.94
3	1	2014-01-01 02:51:30	217.94
4	1	2014-01-01 02:53:30	217.94

5.2 Threshold on Labels

LabelTimes.threshold() will create binary labels by testing if label values are above a threshold. In this example, a threshold is applied to determine which customers spent over 100.

```
[3]: labels.threshold(100).head()
[3]:
```

	customer_id	cutoff_time	total_spent
id			
0	1	2014-01-01 02:45:30	True
1	1	2014-01-01 02:47:30	True
2	1	2014-01-01 02:49:30	True
3	1	2014-01-01 02:51:30	True
4	1	2014-01-01 02:53:30	True

5.3 Lead Labels Times

LabelTimes.apply_lead() will shift the label time earlier. This is useful for training a model to predict in advance. In this example, a one hour lead is applied to the label times.

```
[4]: labels.apply_lead('1h').head()
[4]:
```

	customer_id	cutoff_time	total_spent
id			
0	1	2014-01-01 01:45:30	217.94
1	1	2014-01-01 01:47:30	217.94
2	1	2014-01-01 01:49:30	217.94
3	1	2014-01-01 01:51:30	217.94
4	1	2014-01-01 01:53:30	217.94

5.4 Bin Labels

LabelTimes.bin() will bin the labels into discrete intervals. There are two types of bins. Bins could either be based on values or quantiles. Additionally, the widths of the bins could either be defined by the user or divided equally. The following examples will go through each type.

5.4.1 Value Based

To use bins based on values, `quantiles` should be set to `False` which is the default value.

Equal Width

To group values into bins of equal width, set `bins` as a scalar value. In this example, the total spent is grouped into bins of equal width.

```
[5]: labels.bin(4, quantiles=False).head()
[5]:
```

	customer_id	cutoff_time	total_spent
id			
0	1	2014-01-01 02:45:30	(198.455, 271.072]
1	1	2014-01-01 02:47:30	(198.455, 271.072]
2	1	2014-01-01 02:49:30	(198.455, 271.072]
3	1	2014-01-01 02:51:30	(198.455, 271.072]
4	1	2014-01-01 02:53:30	(198.455, 271.072]

Custom Widths

To group values into bins of custom widths, set `bins` as an array of values to define edges. In this example, the total spent is grouped into bins of custom widths.

```
[6]: inf = float('inf')
edges = [-inf, 34, 50, 67, inf]
labels.bin(edges, quantiles=False).head()
[6]:
```

	customer_id	cutoff_time	total_spent
id			
0	1	2014-01-01 02:45:30	(67.0, inf]
1	1	2014-01-01 02:47:30	(67.0, inf]
2	1	2014-01-01 02:49:30	(67.0, inf]
3	1	2014-01-01 02:51:30	(67.0, inf]
4	1	2014-01-01 02:53:30	(67.0, inf]

5.4.2 Quantile Based

To use bins based on quantiles, `quantiles` should be set to `True`.

Equal Width

To group values into quantile bins of equal width, set `bins` to the number of quantiles as a scalar value (e.g. 4 for quartiles, 10 for deciles, etc.). In this example, the total spent is grouped into bins based on the quartiles.

```
[7]: labels.bin(4, quantiles=True).head()
[7]:
```

	customer_id	cutoff_time	total_spent
id			
0	1	2014-01-01 02:45:30	(196.25, 217.94]
1	1	2014-01-01 02:47:30	(196.25, 217.94]
2	1	2014-01-01 02:49:30	(196.25, 217.94]
3	1	2014-01-01 02:51:30	(196.25, 217.94]
4	1	2014-01-01 02:53:30	(196.25, 217.94]

To verify quartile values, we could check the descriptive statistics.

```
[8]: stats = labels.total_spent.describe()
stats = stats.round(3).to_string()
print(stats)
```

```
count      50.000
mean       215.182
std         90.518
min         53.220
25%        196.250
50%        217.940
75%        290.390
max        343.690
```

Custom Widths

To group values into quantile bins of custom widths, set `bins` as an array of quantiles. In this example, the total spent is grouped into quantile bins of custom widths.

```
[9]: quantiles = [0, .34, .5, .67, 1]
labels.bin(quantiles, quantiles=True).head()
```

```
[9]:
```

	customer_id	cutoff_time	total_spent
id			
0	1	2014-01-01 02:45:30	(196.25, 217.94]
1	1	2014-01-01 02:47:30	(196.25, 217.94]
2	1	2014-01-01 02:49:30	(196.25, 217.94]
3	1	2014-01-01 02:51:30	(196.25, 217.94]
4	1	2014-01-01 02:53:30	(196.25, 217.94]

5.4.3 Label Bins

To assign bins with custom labels, set `labels` to the array of values. The number of labels need to match the number of bins. In this example, the total spent is grouped into bins with custom labels.

```
[10]: values = ['low', 'medium', 'high']
labels.bin(3, labels=values).head()
```

```
[10]:
```

	customer_id	cutoff_time	total_spent
id			
0	1	2014-01-01 02:45:30	medium
1	1	2014-01-01 02:47:30	medium
2	1	2014-01-01 02:49:30	medium
3	1	2014-01-01 02:51:30	medium
4	1	2014-01-01 02:53:30	medium

5.5 Describe Labels

`LabelTimes.describe()` will print out the distribution with the settings and transforms that were used to make the labels. This is useful as a reference for understanding how the labels were generated from raw data. Also, the label distribution is helpful for determining if we have imbalanced labels. In this example, a description of the labels is printed after transforming the labels into discrete values.

```
[11]: labels.threshold(100).describe()
```

```
Label Distribution
```

```
-----  
True      42  
False     8  
Total:    50
```

```
Settings
```

```
-----  
gap                2min  
minimum_data      2h  
num_examples_per_instance 10  
window_size       <Hour>
```

```
Transforms
```

```
-----  
1. threshold  
  - value: 100
```


6.1 Label Maker

<i>LabelMaker</i>	Automatically makes labels for prediction problems.
-------------------	---

6.1.1 composeml.LabelMaker

class `composeml.LabelMaker` (*target_entity*, *time_index*, *labeling_function*, *window_size=None*)
Automatically makes labels for prediction problems.

Methods

<code>__init__</code>	Creates an instance of label maker.
<code>print_slice</code>	Print metadata about slice.
<code>search</code>	Searches the data to calculates labels.
<code>set_index</code>	Sets the time index in a data frame (if not already set).
<code>slice</code>	Generates data slices of target entity.

`composeml.LabelMaker.__init__`

`LabelMaker.__init__` (*target_entity*, *time_index*, *labeling_function*, *window_size=None*)
Creates an instance of label maker.

Parameters

- **target_entity** (*str*) – Entity on which to make labels.
- **time_index** (*str*) – Name of time column in the data frame.
- **labeling_function** (*function*) – Function that transforms a data slice to a label.

- **window_size** (*str or int*) – Duration of each data slice. The default value for window size is all future data.

composeml.LabelMaker.print_slice

`LabelMaker.print_slice` (*metadata*)

Print metadata about slice.

Parameters `metadata` (*dict*) – metadata about slice

composeml.LabelMaker.search

`LabelMaker.search` (*df, num_examples_per_instance, minimum_data=None, gap=None, drop_empty=True, verbose=True, *args, **kwargs*)

Searches the data to calculates labels.

Parameters

- **df** (*DataFrame*) – Data frame to search and extract labels.
- **num_examples_per_instance** (*int*) – Number of examples per unique instance of target entity.
- **minimum_data** (*str*) – Minimum data before starting search. Default value is first time of index.
- **gap** (*str or int*) – Time between examples. Default value is window size. If an integer, search will start on the first event after the minimum data.
- **drop_empty** (*bool*) – Whether to drop empty slices. Default value is True.
- **verbose** (*bool*) – Whether to render progress bar. Default value is True.
- ***args** – Positional arguments for labeling function.
- ****kwargs** – Keyword arguments for labeling function.

Returns Calculated labels with cutoff times.

Return type *LabelTimes*

composeml.LabelMaker.set_index

`LabelMaker.set_index` (*df*)

Sets the time index in a data frame (if not already set).

Parameters `df` (*DataFrame*) – Data frame to set time index in.

Returns Data frame with time index set.

Return type *DataFrame*

composeml.LabelMaker.slice

`LabelMaker.slice` (*df, num_examples_per_instance, minimum_data=None, gap=None, meta-data=False, drop_empty=True, verbose=False*)

Generates data slices of target entity.

Parameters

- **df** (*DataFrame*) – Data frame to create slices on.
- **num_examples_per_instance** (*int*) – Number of examples per unique instance of target entity.
- **minimum_data** (*str*) – Minimum data before starting search. Default value is first time of index.
- **gap** (*str or int*) – Time between examples. Default value is window size. If an integer, search will start on the first event after the minimum data.
- **metadata** (*bool*) – Whether to return metadata about the data slice. Default value is False.
- **drop_empty** (*bool*) – Whether to drop empty slices. Default value is True.
- **verbose** (*bool*) – Whether to print metadata about slice. Default value is False.

Returns Slice of data.

Return type DataFrame

6.2 Label Times

LabelTimes

A data frame containing labels made by a label maker.

6.2.1 composeml.LabelTimes

class `composeml.LabelTimes` (*data=None, name=None, target_entity=None, settings=None, transforms=None, *args, **kwargs*)

A data frame containing labels made by a label maker.

name

target_entity

transforms

Methods

<code>__init__</code>	Initialize self.
<code>apply_lead</code>	Shifts the label times earlier for predicting in advance.
<code>bin</code>	Bin labels into discrete intervals.
<code>copy</code>	Makes a copy of this instance.
<code>describe</code>	Prints out label info with transform settings that reproduce labels.
<code>sample</code>	Return a random sample of labels.
<code>threshold</code>	Creates binary labels by testing if labels are above threshold.

`composeml.LabelTimes.__init__`

`LabelTimes.__init__(data=None, name=None, target_entity=None, settings=None, transforms=None, *args, **kwargs)`
Initialize self. See `help(type(self))` for accurate signature.

`composeml.LabelTimes.apply_lead`

`LabelTimes.apply_lead(value, inplace=False)`
Shifts the label times earlier for predicting in advance.

Parameters

- **value** (*str*) – Time to shift earlier.
- **inplace** (*bool*) – Modify labels in place.

Returns Instance of labels.

Return type labels (*LabelTimes*)

`composeml.LabelTimes.bin`

`LabelTimes.bin(bins, quantiles=False, labels=None, right=True)`
Bin labels into discrete intervals.

Parameters

- **bins** (*int or array*) – The criteria to bin by.
 - **bins** (*int*) [Number of bins either equal-width or quantile-based.] If *quantiles* is *False*, defines the number of equal-width bins. The range is extended by .1% on each side to include the minimum and maximum values. If *quantiles* is *True*, defines the number of quantiles (e.g. 10 for deciles, 4 for quartiles, etc.)
 - **bins** (*array*) [Bin edges either user defined or quantile-based.] If *quantiles* is *False*, defines the bin edges allowing for non-uniform width. No extension is done. If *quantiles* is *True*, defines the bin edges using an array of quantiles (e.g. [0, .25, .5, .75, 1.] for quartiles)
- **quantiles** (*bool*) – Determines whether to use a quantile-based discretization function.
- **labels** (*array*) – Specifies the labels for the returned bins. Must be the same length as the resulting bins.
- **right** (*bool*) – Indicates whether bins includes the rightmost edge or not. Does not apply to quantile-based bins.

Returns Instance of labels.

Return type *LabelTimes*

Examples

Using bins of *equal-widths*:

```
>>> labels.bin(2).head(2).T
label_id                0                1
customer_id            1                1
cutoff_time            2014-01-01 00:45:00  2014-01-01 00:48:00
my_labeling_function    (157.5, 283.46]    (31.288, 157.5]
```

Using bins of *custom-widths*:

```
>>> values = labels.bin([0, 200, 400])
>>> values.head(2).T
label_id                0                1
customer_id            1                1
cutoff_time            2014-01-01 00:45:00  2014-01-01 00:48:00
my_labeling_function    (200, 400]        (0, 200]
```

Using *quantile-based* bins:

```
>>> values = labels.bin(4, quantiles=True) # (i.e. quartiles)
>>> values.head(2).T
label_id                0                1
customer_id            1                1
cutoff_time            2014-01-01 00:45:00  2014-01-01 00:48:00
my_labeling_function    (137.44, 241.062]  (43.848, 137.44]
```

Assigning *labels* to bins:

```
>>> values = labels.bin(3, labels=['low', 'medium', 'high'])
>>> values.head(2).T
label_id                0                1
customer_id            1                1
cutoff_time            2014-01-01 00:45:00  2014-01-01 00:48:00
my_labeling_function    high                low
```

composeml.LabelTimes.copy

LabelTimes.**copy**()

Makes a copy of this instance.

Returns Copy of labels.

Return type labels (*LabelTimes*)

composeml.LabelTimes.describe

LabelTimes.**describe**()

Prints out label info with transform settings that reproduce labels.

composeml.LabelTimes.sample

LabelTimes.**sample**(*n=None, frac=None, random_state=None*)

Return a random sample of labels.

Parameters

- **n** (*int or dict*) – Sample number of labels. A dictionary returns the number of samples to each label. Cannot be used with `frac`.
- **frac** (*float or dict*) – Sample fraction of labels. A dictionary returns the sample fraction to each label. Cannot be used with `n`.
- **random_state** (*int*) – Seed for the random number generator.

Returns Random sample of labels.

Return type *LabelTimes*

Examples

Create mock data:

```
>>> labels = {'labels': list('AABBBAA')}
>>> labels = LabelTimes(labels, name='labels')
>>> labels
  labels
0      A
1      A
2      B
3      B
4      B
5      A
6      A
```

Sample number of labels:

```
>>> labels.sample(n=3, random_state=0)
  labels
6      A
2      B
1      A
```

Sample number per label:

```
>>> n_per_label = {'A': 1, 'B': 2}
>>> labels.sample(n=n_per_label, random_state=0)
  labels
5      A
4      B
3      B
```

Sample fraction of labels:

```
>>> labels.sample(frac=.4, random_state=2)
  labels
4      B
1      A
3      B
```

Sample fraction per label:

```
>>> frac_per_label = {'A': .5, 'B': .34}
>>> labels.sample(frac=frac_per_label, random_state=2)
  labels
```

(continues on next page)

(continued from previous page)

5	A
6	A
4	B

composeml.LabelTimes.thresholdLabelTimes.**threshold** (*value*, *inplace=False*)

Creates binary labels by testing if labels are above threshold.

Parameters

- **value** (*float*) – Value of threshold.
- **inplace** (*bool*) – Modify labels in place.

Returns Instance of labels.**Return type** labels (*LabelTimes*)**6.2.2 Transform Methods**

<i>LabelTimes.apply_lead</i>	Shifts the label times earlier for predicting in advance.
<i>LabelTimes.bin</i>	Bin labels into discrete intervals.
<i>LabelTimes.threshold</i>	Creates binary labels by testing if labels are above threshold.

6.2.3 Plotting Methods

LabelTimes.plot.distribution	Plot the label distribution.
LabelTimes.plot.count_by_time	Plot the label count vs. time.

v0.1.4 August 7, 2019

- **Enhancements**
 - Added Sample Transform
 - Improved Progress Bar
 - Improved Label Times description

v0.1.3 July 9, 2019

- **Enhancements**
 - Improved documentation
 - Added testing for Featuretools compatibility
 - Improved description of Label Times
 - Refactored search in Label Maker
 - Improved testing for Label Transforms

v0.1.2 June 19, 2019

- **Enhancements**
 - Add dynamic progress bar
 - Add label transform for binning labels
 - Improve code coverage
 - Update documentation

v0.1.1 May 31, 2019

- Initial Release

Symbols

`__init__()` (*composeml.LabelMaker method*), 29

`__init__()` (*composeml.LabelTimes method*), 32

A

`apply_lead()` (*composeml.LabelTimes method*), 32

B

`bin()` (*composeml.LabelTimes method*), 32

C

`copy()` (*composeml.LabelTimes method*), 33

D

`describe()` (*composeml.LabelTimes method*), 33

L

`LabelMaker` (*class in composeml*), 29

`LabelTimes` (*class in composeml*), 31

N

`name` (*composeml.LabelTimes attribute*), 31

P

`print_slice()` (*composeml.LabelMaker method*), 30

S

`sample()` (*composeml.LabelTimes method*), 33

`search()` (*composeml.LabelMaker method*), 30

`set_index()` (*composeml.LabelMaker method*), 30

`slice()` (*composeml.LabelMaker method*), 30

T

`target_entity` (*composeml.LabelTimes attribute*),
31

`threshold()` (*composeml.LabelTimes method*), 35

`transforms` (*composeml.LabelTimes attribute*), 31