Compose Documentation

Release 0.5.0

Feature Labs, Inc.

Aug 28, 2020

TABLE OF CONTENTS

Compose

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

ONE

INSTALL

In Python 3.6 or later, Compose can be installed by running the following commmand:

```
pip install composeml
```

CHAPTER

TWO

GETTING STARTED

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 $300. Additionally, we want to predict one hour in advance.

[1]:
```
import composeml as cp
```

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.

[2]:
```
df = cp.demos.load_transactions()

df[df.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
1            2  desktop
2            2  desktop
3            2   tablet
4            2   tablet
```

2.2 Create Labeling Function

To get started, we define the labeling function that will return the total purchase amount given a hour of transactions.

[3]:
```
def total_spent(df):
    total = df['amount'].sum()
    return total
```

2.3 Construct Label Maker

With the labeling function, we create the *LabelMaker* for our prediction problem. To process one hour of transactions for each customer, we set the `target_entity` to the customer ID and the `window_size` to one hour.

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

2.4 Generate Labels

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

[5]:
```
labels = label_maker.search(
    df.sort_values('transaction_time'),
    num_examples_per_instance=-1,
    gap=1,
    verbose=True,
)

labels.head()
```

```
Elapsed: 00:00 | Remaining: 00:00 | Progress: 100%|| customer_id: 5/5
```

[5]:

	customer_id	time	total_spent
0	1	2014-01-01 00:45:30	914.73
1	1	2014-01-01 00:46:35	806.62
2	1	2014-01-01 00:47:40	694.09
3	1	2014-01-01 00:52:00	687.80
4	1	2014-01-01 00:53:05	656.43

2.5 Transform Labels

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

2.5.1 Apply Threshold on Labels

To make the labels binary, *LabelTimes.threshold()* is applied for amounts exceeding $300.

[6]:
```
labels = labels.threshold(300)

labels.head()
```

[6]:

	customer_id	time	total_spent
0	1	2014-01-01 00:45:30	True
1	1	2014-01-01 00:46:35	True
2	1	2014-01-01 00:47:40	True
3	1	2014-01-01 00:52:00	True
4	1	2014-01-01 00:53:05	True

2.5.2 Lead Label Times

Additionally, the label times are shifted one hour earlier for predicting in advance by using *LabelTimes.apply_lead()*.

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

labels.head()
```

[7]:
```
   customer_id                time  total_spent
0            1 2013-12-31 23:45:30         True
1            1 2013-12-31 23:46:35         True
2            1 2013-12-31 23:47:40         True
3            1 2013-12-31 23:52:00         True
4            1 2013-12-31 23:53:05         True
```

2.6 Describe Labels

After transforming the labels, we can use *LabelTimes.describe()* to print out the distribution with the settings and transforms that were used to make these 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      56
True       44
Total:    100


Settings
--------
gap                                1
minimum_data                    None
num_examples_per_instance         -1
target_column            total_spent
target_entity            customer_id
target_type                 discrete
window_size                       1h


Transforms
----------
1. threshold
  - value:    300

2. apply_lead
  - value:    1h
```

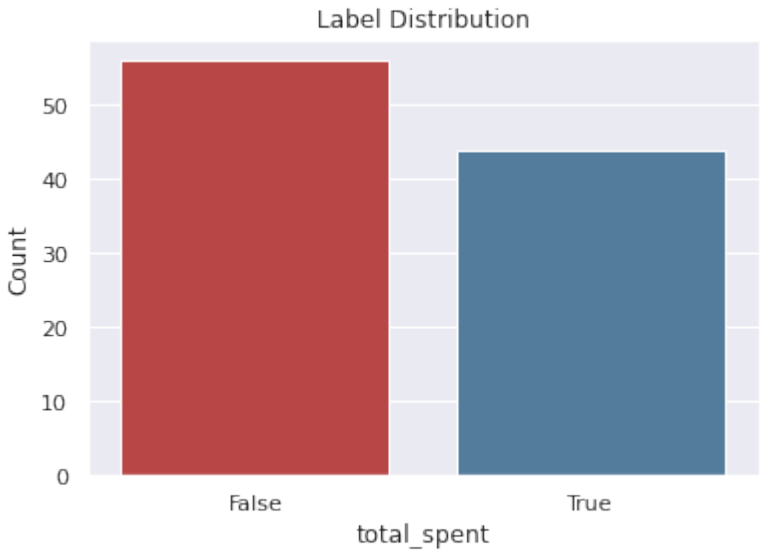
2.7 Plot Labels

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

2.7.1 Distribution

This plot shows the label distribution.

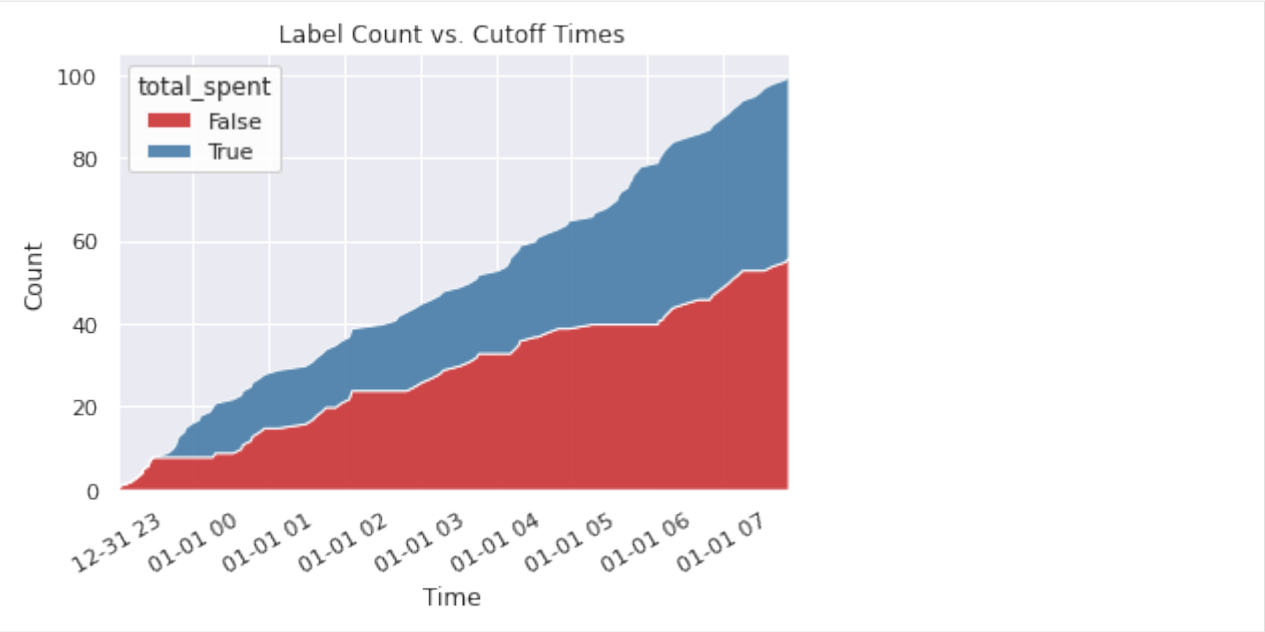
[9]:
```
%matplotlib inline
labels.plot.distribution();
```



2.7.2 Count by Time

This plot shows the label distribution across cutoff times.

[10]:
```
%matplotlib inline
labels.plot.count_by_time();
```


Label Count vs. Cutoff Times
total_spent
False
True
Count
Time
0
20
40
60
80
100
12-31 23
01-01 00
01-01 01
01-01 02
01-01 03
01-01 04
01-01 05
01-01 06
01-01 07

CHAPTER

THREE

MAIN CONCEPTS

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.

CHAPTER

FOUR

DATA SLICE GENERATOR

The data slice generator is the underlying function used to generate data slices for the labeling function. If the label maker raises an error during the search or the output labels doesn't seem right, then you will need to check the logic in the labeling function or inspect the data for any inherent errors. This is where the data slice generator can help us do both. Ideally, you also want to use the generator during the development of your labeling function for best practice. However, it is an optional step and not required to generate labels.

In this guide, we will use the data slice generator to inspect data slices and apply our labeling function. To get started, let's load a mock dataset of transactions and sample the data to see how the transactions look.

[1]:
```
import composeml as cp
```

[2]:
```
df = cp.demos.load_transactions()
df = df[df.columns[:7]]
df.sample(n=5, random_state=0)
```

[2]:
```
    transaction_id  session_id    transaction_time  product_id  amount  \
26              94          24 2014-01-01 05:55:20           5  100.42
86             274           7 2014-01-01 01:46:10           5   14.45
2              495           1 2014-01-01 00:14:05           5   69.45
55             275           4 2014-01-01 00:45:30           5  108.11
75             368          27 2014-01-01 06:36:30           5  139.43

    customer_id   device
26            5   tablet
86            3   tablet
2             2  desktop
55            1   mobile
75            1   mobile
```

4.1 Labeling Function

Let's define a labeling function that will return how much a customer spent given a slice of transactions.

[3]:
```
def total_spent(df):
    return df['amount'].sum()
```

4.2 Data Slices

The *LabelMaker.slice()* method will create the data slice generator. The parameters of this method can be passed directly to *LabelMaker.search()* to generate the labels. In the following sections, we will see how to use the data slice generator to make data slices consecutive, overlap, or spread out.

See also:

For a conceptual explanation of the process, see *Main Concepts*.

4.2.1 Consecutive

When the the gap size is equal to the window size, the data slices are consecutive. In other words, the data slices do not overlap and are not spread out (e.g. don't skip any data). This is the default value for the gap size. To demonstrate this example, let's generate data slices using these parameters.

We create a label maker with the 2-hour window size.

[4]:
```
lm = cp.LabelMaker(
    target_entity="customer_id",
    time_index="transaction_time",
    labeling_function=total_spent,
    window_size="2h",
)
```

Then, we create a data slice generator with the 2-hour gap size. The default value for the gap size is the window size.

Tip: You can directly set `minimum_data` as the first cutoff time.

[5]:
```
slices = lm.slice(
    df.sort_values('transaction_time'),
    num_examples_per_instance=-1,
    minimum_data='2014-01-01',
)
```

Consecutive - Data Slice #1

By printing this data slice, we can see that it's the first slice of transactions (denoted by the `slice_number`) for customer 1. This data slice contains all of the customer's transactions that occurred within the 2-hour window between `2014-01-01 00:00:00` and `2014-01-01 02:00:00`. We can also see that the 2-hour gap aligns the cutoff times to the window. So, the next data slice will start at the end of this data slice.

[6]:
```
ds = next(slices)
print(ds.context)
ds
```

```
customer_id                    1
slice_number                   1
slice_start      2014-01-01 00:00:00
slice_stop       2014-01-01 02:00:00
next_start       2014-01-01 02:00:00
```

[6]:
```
                     transaction_id  session_id  product_id  amount  \
transaction_time
2014-01-01 00:45:30             275           4           5  108.11
2014-01-01 00:46:35             101           4           5  112.53
2014-01-01 00:47:40              80           4           5    6.29
2014-01-01 00:52:00             163           4           5   31.37
2014-01-01 00:53:05             293           4           5   82.88
2014-01-01 00:57:25             103           4           5   20.79
2014-01-01 01:03:55             488           4           5  129.00
2014-01-01 01:05:00             413           4           5  119.98
2014-01-01 01:31:00             191           6           5  139.23
2014-01-01 01:37:30             372           6           5  114.84
2014-01-01 01:38:35             387           6           5   49.71

                     customer_id  device
transaction_time
2014-01-01 00:45:30            1  mobile
2014-01-01 00:46:35            1  mobile
2014-01-01 00:47:40            1  mobile
2014-01-01 00:52:00            1  mobile
2014-01-01 00:53:05            1  mobile
2014-01-01 00:57:25            1  mobile
2014-01-01 01:03:55            1  mobile
2014-01-01 01:05:00            1  mobile
2014-01-01 01:31:00            1  tablet
2014-01-01 01:37:30            1  tablet
2014-01-01 01:38:35            1  tablet
```

Let's apply our labeling function for the total spent on this data slice.

[7]:
```
total_spent(ds)
```

[7]:
```
914.7300000000001
```

Consecutive - Data Slice #2

In the second data slice, we can see the next 2 consecutive hours of transactions between `2014-01-01 02:00:00` and `2014-01-01 04:00:00`. This is useful for generating labels that will consecutively process the data only once.

[8]:
```
ds = next(slices)
print(ds.context)
ds
```

```
customer_id                    1
slice_number                   2
slice_start      2014-01-01 02:00:00
slice_stop       2014-01-01 04:00:00
next_start       2014-01-01 04:00:00
```

[8]:
```
                     transaction_id  session_id  product_id  amount  \
transaction_time
2014-01-01 02:28:25             287           9           5   50.94
2014-01-01 03:29:05             190          14           5  110.52
2014-01-01 03:39:55               7          14           5  107.42

                     customer_id   device
```

(continues on next page)

(continued from previous page)

```
transaction_time
2014-01-01 02:28:25            1  desktop
2014-01-01 03:29:05            1   tablet
2014-01-01 03:39:55            1   tablet
```

Let's apply our labeling function for the total spent on this data slice.

[9]:
```
total_spent(ds)
```

[9]:
```
268.88
```

4.2.2 Overlap

When the the gap size is less than the window size, the data slices will overlap. We can use this for rolling window based labeling processes. The amount of overlap is the difference between the window and gap size. For example, if the window size is 3 hours and the gap size is 1 hour, then 2 hours will overlap on each data slice. To demonstrate this example, let's generate data slices using these parameters.

We create a label maker with the 3-hour window size.

[10]:
```
lm = cp.LabelMaker(
    target_entity="customer_id",
    time_index="transaction_time",
    labeling_function=total_spent,
    window_size="3h",
)
```

Then, we create a data slice generator with the 1-hour gap size.

[11]:
```
slices = lm.slice(
    df.sort_values('transaction_time'),
    num_examples_per_instance=-1,
    minimum_data='2014-01-01',
    gap="1h",
)
```

Overlap - Data Slice #1

The first data slice contains all of the customer's transactions that occurred within the 3-hour window between `2014-01-01 00:00:00` and `2014-01-01 03:00:00`. We can also see that the 1-hour gap spaces apart the cutoff time of this data slice at `2014-01-01 00:00:00` from the cutoff time of the next data slice at `2014-01-01 01:00:00`.

[12]:
```
ds = next(slices)
print(ds.context)
ds
```

```
customer_id                    1
slice_number                   1
slice_start      2014-01-01 00:00:00
slice_stop       2014-01-01 03:00:00
next_start       2014-01-01 01:00:00
```

[12]:
```
                     transaction_id  session_id  product_id  amount  \
transaction_time
2014-01-01 00:45:30             275           4           5  108.11
2014-01-01 00:46:35             101           4           5  112.53
2014-01-01 00:47:40              80           4           5    6.29
2014-01-01 00:52:00             163           4           5   31.37
2014-01-01 00:53:05             293           4           5   82.88
2014-01-01 00:57:25             103           4           5   20.79
2014-01-01 01:03:55             488           4           5  129.00
2014-01-01 01:05:00             413           4           5  119.98
2014-01-01 01:31:00             191           6           5  139.23
2014-01-01 01:37:30             372           6           5  114.84
2014-01-01 01:38:35             387           6           5   49.71
2014-01-01 02:28:25             287           9           5   50.94

                     customer_id   device
transaction_time
2014-01-01 00:45:30            1   mobile
2014-01-01 00:46:35            1   mobile
2014-01-01 00:47:40            1   mobile
2014-01-01 00:52:00            1   mobile
2014-01-01 00:53:05            1   mobile
2014-01-01 00:57:25            1   mobile
2014-01-01 01:03:55            1   mobile
2014-01-01 01:05:00            1   mobile
2014-01-01 01:31:00            1   tablet
2014-01-01 01:37:30            1   tablet
2014-01-01 01:38:35            1   tablet
2014-01-01 02:28:25            1  desktop
```

Let's apply our labeling function for the total spent on this data slice.

[13]:
```
total_spent(ds)
```

[13]:
```
965.6700000000001
```

Overlap - Data Slice #2

In the second data slice, we can see that there is a 2-hour overlap on the transactions that occurred between `2014-01-01 01:00:00` and `2014-01-01 03:00:00`. By adjusting the gap size, we can set the precise amount of overlap in the data slices. This is useful for generating labels with specific overlap.

[14]:
```
ds = next(slices)
print(ds.context)
ds
```

```
customer_id                    1
slice_number                   2
slice_start      2014-01-01 01:00:00
slice_stop       2014-01-01 04:00:00
next_start       2014-01-01 02:00:00
```

[14]:
```
                     transaction_id  session_id  product_id  amount  \
transaction_time
2014-01-01 01:03:55             488           4           5  129.00
2014-01-01 01:05:00             413           4           5  119.98
2014-01-01 01:31:00             191           6           5  139.23
```

(continues on next page)

(continued from previous page)

```
2014-01-01 01:37:30             372           6           5  114.84
2014-01-01 01:38:35             387           6           5   49.71
2014-01-01 02:28:25             287           9           5   50.94
2014-01-01 03:29:05             190          14           5  110.52
2014-01-01 03:39:55               7          14           5  107.42

                     customer_id   device
transaction_time
2014-01-01 01:03:55            1   mobile
2014-01-01 01:05:00            1   mobile
2014-01-01 01:31:00            1   tablet
2014-01-01 01:37:30            1   tablet
2014-01-01 01:38:35            1   tablet
2014-01-01 02:28:25            1  desktop
2014-01-01 03:29:05            1   tablet
2014-01-01 03:39:55            1   tablet
```

Let's apply our labeling function for the total spent on this data slice.

[15]:
```
total_spent(ds)
```

[15]:
```
821.6400000000001
```

4.2.3 Spread Out

When the the gap size is greater than the window size, then there is data in-between data slices that will be skipped. We can use this for labeling data at specific intervals of time. The amount of data skipped is the difference between the gap and window size. For example, if the gap size is 3 hours and the window size is 1 hour, then 2 hours of data will be skipped in-between data slices. To demonstrate this example, let's generate data slices using these parameters.

We create a label maker with the 1-hour window size.

[16]:
```
lm = cp.LabelMaker(
    target_entity="customer_id",
    time_index="transaction_time",
    labeling_function=total_spent,
    window_size="1h",
)
```

Then, we create a data slice generator with the 3-hour gap size.

[17]:
```
slices = lm.slice(
    df.sort_values('transaction_time'),
    num_examples_per_instance=-1,
    minimum_data='2014-01-01',
    gap="3h",
)
```

Spread Out - Data Slice #1

The first data slice contains all of the customer's transactions that occurred within the 1-hour window between `2014-01-01 00:00:00` and `2014-01-01 01:00:00`. We can also see that the 3-hour gap spaces apart the cutoff time of this data slice at `2014-01-01 00:00:00` from the cutoff time of the next data slice at `2014-01-01 03:00:00`.

[18]:
```
ds = next(slices)
print(ds.context)
ds
```

```
customer_id                     1
slice_number                    1
slice_start     2014-01-01 00:00:00
slice_stop      2014-01-01 01:00:00
next_start      2014-01-01 03:00:00
```

[18]:
```
                     transaction_id  session_id  product_id  amount  \
transaction_time
2014-01-01 00:45:30             275           4           5  108.11
2014-01-01 00:46:35             101           4           5  112.53
2014-01-01 00:47:40              80           4           5    6.29
2014-01-01 00:52:00             163           4           5   31.37
2014-01-01 00:53:05             293           4           5   82.88
2014-01-01 00:57:25             103           4           5   20.79

                     customer_id  device
transaction_time
2014-01-01 00:45:30            1  mobile
2014-01-01 00:46:35            1  mobile
2014-01-01 00:47:40            1  mobile
2014-01-01 00:52:00            1  mobile
2014-01-01 00:53:05            1  mobile
2014-01-01 00:57:25            1  mobile
```

Let's apply our labeling function for the total spent on this data slice.

[19]:
```
total_spent(ds)
```

[19]:
```
361.96999999999997
```

Spread Out - Data Slice #2

In the second data slice, we can see that 2 hours of transactions were skipped between `2014-01-01 01:00:00` and `2014-01-01 03:00:00`. By adjusting the gap size, we can set the precise amount of data to skip in-between data slices. This is useful for generating labels that target specific portions of a dataset.

[20]:
```
ds = next(slices)
print(ds.context)
ds
```

```
customer_id                     1
slice_number                    2
slice_start     2014-01-01 03:00:00
slice_stop      2014-01-01 04:00:00
next_start      2014-01-01 06:00:00
```

[20]:
```
                     transaction_id  session_id  product_id  amount  \
transaction_time
2014-01-01 03:29:05             190          14           5  110.52
2014-01-01 03:39:55               7          14           5  107.42

                     customer_id  device
transaction_time
2014-01-01 03:29:05            1  tablet
2014-01-01 03:39:55            1  tablet
```

Let's apply our labeling function for the total spent on this data slice.

[21]:
```
total_spent(ds)
```

[21]:
```
217.94
```

4.3 Data Slice Context

Each data slice has a `context` attribute to access its metadata. This is useful for integrating the context with the logic in the labeling function.

[22]:
```
vars(ds.context)
```

[22]:
```
{'next_start': Timestamp('2014-01-01 06:00:00'),
 'slice_stop': Timestamp('2014-01-01 04:00:00'),
 'slice_start': Timestamp('2014-01-01 03:00:00'),
 'slice_number': 2,
 'customer_id': 1}
```

From this guide, hopefully you have a better understanding on how to use the data slice generator to develop your labeling function.

CHAPTER

FIVE

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]:
```
import composeml as cp

def total_spent(df):
    return df['amount'].sum()

label_maker = cp.LabelMaker(
    labeling_function=total_spent,
    target_entity='customer_id',
    time_index='transaction_time',
    window_size='1h',
)

labels = label_maker.search(
    cp.demos.load_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	time	total_spent
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	time	total_spent
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	time	total_spent
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	time	total_spent
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                time  total_spent
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                time       total_spent
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                time       total_spent
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                time total_spent
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 examlpe, a description of the labels is printed after transforming the labels into discrete values.

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

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


Settings
--------
gap                                2min
minimum_data                         2h
num_examples_per_instance            10
target_column                total_spent
target_entity                customer_id
target_type                     discrete
```

(continues on next page)

(continued from previous page)

```
window_size                             1h


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

5.6 Sample Labels

LabelTimes.sample() will sample the labels based on a number or fraction. Samples can be reproduced by fixing `random_state` to an integer.

To sample 10 labels, `n` is set to 10.

[12]:
```
labels.sample(n=10, random_state=0)
```

[12]:

	customer_id	time	total_spent
2	1	2014-01-01 02:49:30	217.94
4	1	2014-01-01 02:53:30	217.94
10	2	2014-01-01 02:00:00	290.39
11	2	2014-01-01 02:02:00	290.39
22	3	2014-01-01 03:49:05	196.25
27	3	2014-01-01 03:59:05	196.25
28	3	2014-01-01 04:01:05	196.25
31	4	2014-01-01 02:41:00	343.69
38	4	2014-01-01 02:55:00	225.18
41	5	2014-01-01 03:48:25	53.22

Similarly, to sample 10% of labels, `frac` is set to 10%.

[13]:
```
labels.sample(frac=.1, random_state=0)
```

[13]:

	customer_id	time	total_spent
2	1	2014-01-01 02:49:30	217.94
10	2	2014-01-01 02:00:00	290.39
11	2	2014-01-01 02:02:00	290.39
28	3	2014-01-01 04:01:05	196.25
41	5	2014-01-01 03:48:25	53.22

5.6.1 Categorical Labels

When working with categorical labels, the number or fraction of labels for each category can be sampled by using a dictionary. Let's bin the labels into 4 bins to make categorical.

[14]:
```
categorical = labels.bin(4, labels=['A', 'B', 'C', 'D'])
```

To sample 2 labels per category, map each category to the number 2.

[15]:
```
n = {'A': 2, 'B': 2, 'C': 2, 'D': 2}
categorical.sample(n=n, random_state=0)
```

[15]:
```
    customer_id                time total_spent
6             1 2014-01-01 02:57:30           C
11            2 2014-01-01 02:02:00           D
16            2 2014-01-01 02:12:00           D
26            3 2014-01-01 03:57:05           B
38            4 2014-01-01 02:55:00           C
42            5 2014-01-01 03:50:25           A
46            5 2014-01-01 03:58:25           A
48            5 2014-01-01 04:02:25           B
```

Similarly, to sample 10% of labels per category, map each category to 10%.

[16]:
```
frac = {'A': .1, 'B': .1, 'C': .1, 'D': .1}
categorical.sample(frac=frac, random_state=0)
```

[16]:
```
    customer_id                time total_spent
6             1 2014-01-01 02:57:30           C
11            2 2014-01-01 02:02:00           D
16            2 2014-01-01 02:12:00           D
26            3 2014-01-01 03:57:05           B
46            5 2014-01-01 03:58:25           A
```

CHAPTER

SIX

PREDICT NEXT PURCHASE

In this example, we build a machine learning application to predict whether customers will purchase a product within the next shopping period. This application is structured into three important steps:

- Prediction Engineering
- Feature Engineering
- Machine Learning

In the first step, we generate new labels from the data by using Compose. In the second step, we generate features for the labels by using Featuretools. In the third step, we search for the best machine learning pipeline by using EvalML. After working through these steps, you will learn how to build machine learning applications for real-world problems like predicting consumer spending. Let's get started.

[1]:
```
from demo.next_purchase import load_sample
from matplotlib.pyplot import subplots
import composeml as cp
import featuretools as ft
import evalml
```

We will use this historical data of online grocery orders provided by Instacart.

[2]:
```
df = load_sample()

df.head()
```

[2]:
```
    order_id  product_id  add_to_cart_order  reordered  \
id
0        120       33120                 13          0
1        120       31323                  7          0
2        120        1503                  8          0
3        120       28156                 11          0
4        120       41273                  4          0

                          product_name  aisle_id  department_id  department  \
id
0                   Organic Egg Whites        86             16  dairy eggs
1        Light Wisconsin String Cheese        21             16  dairy eggs
2               Low Fat Cottage Cheese       108             16  dairy eggs
3   Total 0% Nonfat Plain Greek Yogurt       120             16  dairy eggs
4                    Broccoli Florets       123              4     produce

    user_id          order_time
id
0     23750 2015-01-11 08:00:00
```

(continues on next page)

(continued from previous page)

```
1      23750 2015-01-11 08:00:00
2      23750 2015-01-11 08:00:00
3      23750 2015-01-11 08:00:00
4      23750 2015-01-11 08:00:00
```

6.1 Prediction Engineering

> Will customers purchase a product within the next shopping period?

In this prediction problem, we have two parameters:

- The product that a customer can purchase.
- The length of the shopping period.

We can change these parameters to create different prediction problems. For example, will a customer purchase a banana within the next 5 days or an avocado within the next three weeks? These variations can be done by simply tweaking the parameters. This helps us explore different scenarios which is crucial for making better decisions.

6.1.1 Defining the Labeling Process

Let's start by defining a labeling function that checks if a customer bought a given product. We will make the product a parameter of the function.

[3]:
```
def bought_product(ds, product_name):
    return ds.product_name.str.contains(product_name).any()
```

6.1.2 Representing the Prediction Problem

Then, let's represent the prediction problem by creating a label maker with the following parameters:

- `target_entity` as the columns for the customer ID, since we want to process orders for each customer.
- `labeling_function` as the function we defined previously.
- `time_index` as the column for the order time. The shoppings periods are based on this time index.
- `window_size` as the length of a shopping period. We can easily change this parameter to create variations of the prediction problem.

[4]:
```
lm = cp.LabelMaker(
    target_entity='user_id',
    time_index='order_time',
    labeling_function=bought_product,
    window_size='3d',
)
```

6.1.3 Finding the Training Examples

Now, let's run a search to get the training examples by using the following parameters:

- The grocery orders sorted by the order time.
- `num_examples_per_instance` to find the number of training examples per customer. In this case, the search will return all existing examples.
- `product_name` as the product to check for purchases. This parameter gets passed directly to the our labeling function.
- `minimum_data` as the amount of data that will be used to make features for the first training example.

[5]:
```
lt = lm.search(
    df.sort_values('order_time'),
    num_examples_per_instance=-1,
    product_name='Banana',
    minimum_data='3d',
    verbose=False,
)

lt.head()
```

[5]:

	user_id	time	bought_product
0	2555	2015-01-22 09:00:00	False
1	3283	2015-01-08 13:00:00	True
2	5360	2015-01-21 11:00:00	False
3	5669	2015-01-09 08:00:00	False
4	5669	2015-01-12 08:00:00	True

The output from the search is a label times table with three columns:

- The customer ID associated to the orders.
- The start time of the shopping period. This is also the cutoff time for building features. Only data that existed beforehand is valid to use for predictions.
- Whether or not the product was purchased. This is calculated by our labeling function.

As a helpul reference, we can print out the search settings that were used to generate these labels. The description also shows us the label distribution which we can check for imbalanced labels.

[6]:
```
lt.describe()
```

```
Label Distribution
------------------
False     24
True      16
Total:    40


Settings
--------
gap                                 None
minimum_data                          3d
num_examples_per_instance             -1
target_column             bought_product
target_entity                    user_id
target_type                     discrete
window_size                           3d
```

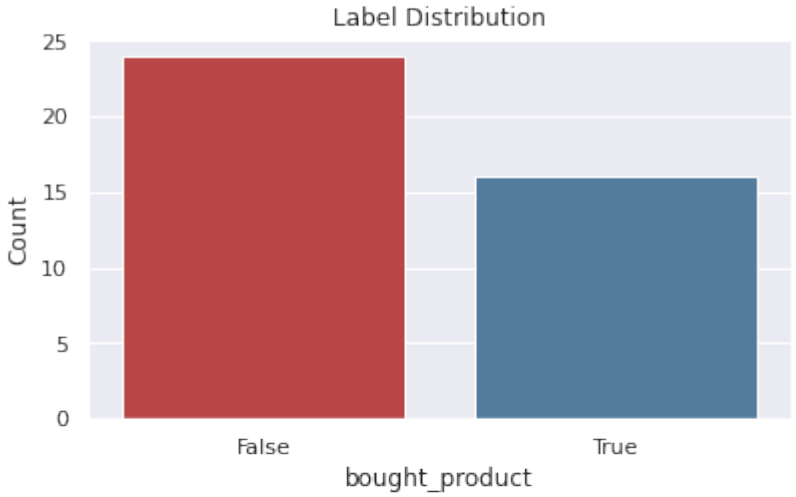
(continues on next page)

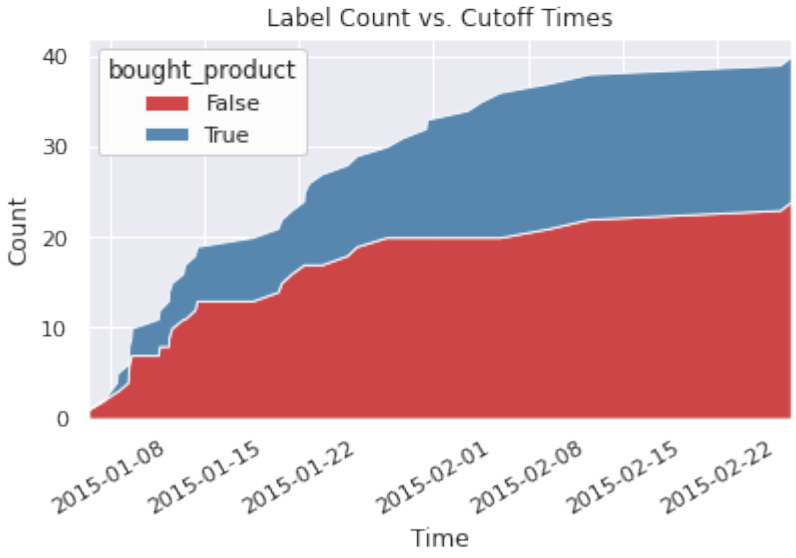
(continued from previous page)

```
Transforms
----------
No transforms applied
```

We can also get a better look at the labels by plotting the distribution and cumulative count across time.

[7]:
```
%matplotlib inline
fig, ax = subplots(nrows=2, ncols=1, figsize=(6, 8))
lt.plot.distribution(ax=ax[0])
lt.plot.count_by_time(ax=ax[1])
fig.tight_layout(pad=2)
```





6.2 Feature Engineering

In the previous step, we generated the labels. The next step is to generate the features.

6.2.1 Representing the Data

We will represent the online grocery orders using an entity set. This way, we can generate features based on the relational structure of the dataset. We currently have a single table of orders where one customer can many orders. This one-to-many relationship can be represented by normalizing an entity for customers. The same can be done for departments, aisles, and products. Let's structure the entity set.

[8]:
```
es = ft.EntitySet('instacart')

es.entity_from_dataframe(
    dataframe=df.reset_index(),
    entity_id='order_products',
    time_index='order_time',
    index='id',
)

es.normalize_entity(
    base_entity_id='order_products',
    new_entity_id='orders',
    index='order_id',
    additional_variables=['user_id'],
    make_time_index=False,
)

es.normalize_entity(
    base_entity_id='orders',
    new_entity_id='customers',
    index='user_id',
    make_time_index=False,
)

es.normalize_entity(
    base_entity_id='order_products',
    new_entity_id='products',
    index='product_id',
    additional_variables=['aisle_id', 'department_id'],
    make_time_index=False,
)

es.normalize_entity(
    base_entity_id='products',
    new_entity_id='aisles',
    index='aisle_id',
    additional_variables=['department_id'],
    make_time_index=False,
)

es.normalize_entity(
    base_entity_id='aisles',
    new_entity_id='departments',
    index='department_id',
    make_time_index=False,
```

(continues on next page)

(continued from previous page)

```
)

es["order_products"]["department"].interesting_values = ['produce']
es["order_products"]["product_name"].interesting_values = ['Banana']
es.plot()
```

[8]:

6.2.2 Calculating the Features

Now, we can generate features by using a method called Deep Feature Synthesis (DFS). This will automatically build features by stacking and applying mathematical operations called primitives across relationships in an entity set. The more structured an entity set is, the better DFS can leverage the relationships to generate better features. Let's run DFS using the following parameters:

- `entity_set` as the entity set we structured previously.
- `target_entity` as the customers, since we want to generate features for each customer.
- `cutoff_time` as the label times that we generated previously.

[9]:
```
fm, fd = ft.dfs(
    entityset=es,
    target_entity='customers',
    cutoff_time=lt,
    cutoff_time_in_index=True,
    include_cutoff_time=False,
    verbose=False,
)

fm.head()
```

[9]:
```
                            COUNT(orders)  \
user_id time
2555    2015-01-22 09:00:00              2
3283    2015-01-08 13:00:00              2
5360    2015-01-21 11:00:00              2
5669    2015-01-09 08:00:00              3
        2015-01-12 08:00:00              3

                            SUM(order_products.add_to_cart_order)  \
user_id time
2555    2015-01-22 09:00:00                                    3
3283    2015-01-08 13:00:00                                   28
5360    2015-01-21 11:00:00                                  903
5669    2015-01-09 08:00:00                                   66
        2015-01-12 08:00:00                                  157

                            SUM(order_products.reordered)  \
user_id time
2555    2015-01-22 09:00:00                             2
3283    2015-01-08 13:00:00                             6
5360    2015-01-21 11:00:00                            37
5669    2015-01-09 08:00:00                             9
        2015-01-12 08:00:00                            22

                            STD(order_products.add_to_cart_order)  \
user_id time
```

(continues on next page)

(continued from previous page)

```
2555    2015-01-22 09:00:00                                0.707107
3283    2015-01-08 13:00:00                                2.160247
5360    2015-01-21 11:00:00                               12.267844
5669    2015-01-09 08:00:00                                3.316625
        2015-01-12 08:00:00                                3.599265

                            STD(order_products.reordered)  \
user_id time
2555    2015-01-22 09:00:00                       0.000000
3283    2015-01-08 13:00:00                       0.377964
5360    2015-01-21 11:00:00                       0.327770
5669    2015-01-09 08:00:00                       0.404520
        2015-01-12 08:00:00                       0.282330

                            MAX(order_products.add_to_cart_order)  \
user_id time
2555    2015-01-22 09:00:00                                    2
3283    2015-01-08 13:00:00                                    7
5360    2015-01-21 11:00:00                                   42
5669    2015-01-09 08:00:00                                   11
        2015-01-12 08:00:00                                   13

                            MAX(order_products.reordered)  \
user_id time
2555    2015-01-22 09:00:00                             1
3283    2015-01-08 13:00:00                             1
5360    2015-01-21 11:00:00                             1
5669    2015-01-09 08:00:00                             1
        2015-01-12 08:00:00                             1

                            SKEW(order_products.add_to_cart_order)  \
user_id time
2555    2015-01-22 09:00:00                                  NaN
3283    2015-01-08 13:00:00                             0.000000
5360    2015-01-21 11:00:00                             0.000000
5669    2015-01-09 08:00:00                             0.000000
        2015-01-12 08:00:00                             0.072227

                            SKEW(order_products.reordered)  \
user_id time
2555    2015-01-22 09:00:00                         NaN
3283    2015-01-08 13:00:00                   -2.645751
5360    2015-01-21 11:00:00                   -2.440735
5669    2015-01-09 08:00:00                   -1.922718
        2015-01-12 08:00:00                   -3.219960

                            MIN(order_products.add_to_cart_order)  ...  \
user_id time                                                       ...
2555    2015-01-22 09:00:00                                    1  ...
3283    2015-01-08 13:00:00                                    1  ...
5360    2015-01-21 11:00:00                                    1  ...
5669    2015-01-09 08:00:00                                    1  ...
        2015-01-12 08:00:00                                    1  ...

                            NUM_UNIQUE(orders.MODE(order_products.product_id))  \
user_id time
2555    2015-01-22 09:00:00                                                1.0
```

(continues on next page)

(continued from previous page)

```
3283    2015-01-08 13:00:00                                                1.0
5360    2015-01-21 11:00:00                                                1.0
5669    2015-01-09 08:00:00                                                1.0
        2015-01-12 08:00:00                                                2.0

                           NUM_UNIQUE(orders.MODE(order_products.product_name))  \
user_id time
2555    2015-01-22 09:00:00                                                  1
3283    2015-01-08 13:00:00                                                  1
5360    2015-01-21 11:00:00                                                  1
5669    2015-01-09 08:00:00                                                  1
        2015-01-12 08:00:00                                                  1

                           MODE(orders.MODE(order_products.department))  \
user_id time
2555    2015-01-22 09:00:00                                  dairy eggs
3283    2015-01-08 13:00:00                                     produce
5360    2015-01-21 11:00:00                                      snacks
5669    2015-01-09 08:00:00                                     produce
        2015-01-12 08:00:00                                  dairy eggs

                           MODE(orders.MODE(order_products.product_id))  \
user_id time
2555    2015-01-22 09:00:00                                    27086.0
3283    2015-01-08 13:00:00                                    13176.0
5360    2015-01-21 11:00:00                                     4142.0
5669    2015-01-09 08:00:00                                     4097.0
        2015-01-12 08:00:00                                     4097.0

                           MODE(orders.MODE(order_products.product_name))  \
user_id time
2555    2015-01-22 09:00:00                                   Half & Half
3283    2015-01-08 13:00:00                          Bag of Organic Bananas
5360    2015-01-21 11:00:00   6 OZ LA PANZANELLA CROSTINI ORIGINAL CRACKERS
5669    2015-01-09 08:00:00           Almonds & Sea Salt in Dark Chocolate
        2015-01-12 08:00:00           Almonds & Sea Salt in Dark Chocolate

                           COUNT(order_products WHERE product_name = Banana)  \
user_id time
2555    2015-01-22 09:00:00                                              0.0
3283    2015-01-08 13:00:00                                              0.0
5360    2015-01-21 11:00:00                                              0.0
5669    2015-01-09 08:00:00                                              1.0
        2015-01-12 08:00:00                                              1.0

                           COUNT(order_products WHERE department = produce)  \
user_id time
2555    2015-01-22 09:00:00                                            0.0
3283    2015-01-08 13:00:00                                            2.0
5360    2015-01-21 11:00:00                                            2.0
5669    2015-01-09 08:00:00                                            4.0
        2015-01-12 08:00:00                                            7.0

                          NUM_UNIQUE(order_products.orders.user_id)  \
user_id time
2555    2015-01-22 09:00:00                                         1
3283    2015-01-08 13:00:00                                         1
```

(continues on next page)

(continued from previous page)

```
5360    2015-01-21 11:00:00                                          1
5669    2015-01-09 08:00:00                                          1
        2015-01-12 08:00:00                                          1

                            MODE(order_products.orders.user_id)   \
user_id time
2555    2015-01-22 09:00:00                                  2555
3283    2015-01-08 13:00:00                                  3283
5360    2015-01-21 11:00:00                                  5360
5669    2015-01-09 08:00:00                                  5669
        2015-01-12 08:00:00                                  5669

                           bought_product
user_id time
2555    2015-01-22 09:00:00          False
3283    2015-01-08 13:00:00           True
5360    2015-01-21 11:00:00          False
5669    2015-01-09 08:00:00          False
        2015-01-12 08:00:00           True

[5 rows x 116 columns]
```

There are two outputs from DFS: a feature matrix and feature definitions. The feature matrix is a table that contains the feature values based on the cutoff times from our labels. Feature definitions are features in a list that can be stored and reused later to calculate the same set of features on future data.

6.3 Machine Learning

In the previous steps, we generated the labels and features. The final step is to build the machine learning pipeline.

6.3.1 Splitting the Data

We will start by extracting the labels from the feature matrix and splitting the data into a training set and holdout set.

[10]:
```
y = fm.pop('bought_product')

splits = evalml.preprocessing.split_data(
    X=fm,
    y=y,
    test_size=0.2,
    random_state=0,
)

X_train, X_holdout, y_train, y_holdout = splits
```

6.3.2 Finding the Best Model

Then, we run a search on the training set for the best machine learning model.

[11]:
```
automl = evalml.AutoMLSearch(
    problem_type='binary',
    objective='f1',
    random_state=0,
)

automl.search(
    X=X_train,
    y=y_train,
    data_checks='disabled',
    show_iteration_plot=False,
)
```

```
Using default limit of max_pipelines=5.

Generating pipelines to search over...
*****************************
* Beginning pipeline search *
*****************************

Optimizing for F1.
Greater score is better.

Searching up to 5 pipelines.
Allowed model families: catboost, xgboost, random_forest, linear_model, extra_trees

(1/5) Mode Baseline Binary Classification P... Elapsed:00:00
        Starting cross validation
        Finished cross validation - mean F1: 0.000
(2/5) Extra Trees Classifier w/ Imputer + O... Elapsed:00:00
        Starting cross validation
        Finished cross validation - mean F1: 0.370
(3/5) Elastic Net Classifier w/ Imputer + O... Elapsed:00:02
        Starting cross validation
        Finished cross validation - mean F1: 0.000
(4/5) CatBoost Classifier w/ Imputer           Elapsed:00:04
        Starting cross validation
        Finished cross validation - mean F1: 0.422
(5/5) XGBoost Classifier w/ Imputer + One H... Elapsed:00:04
        Starting cross validation
        Finished cross validation - mean F1: 0.290

Search finished after 00:06
Best pipeline: CatBoost Classifier w/ Imputer
Best pipeline F1: 0.421958
```

Once the search is complete, we can print out information about the best pipeline found, such as the parameters in each component.

[12]:
```
automl.best_pipeline.describe()
automl.best_pipeline.graph()
```

```
*********************************
* CatBoost Classifier w/ Imputer *
*********************************
```

(continues on next page)

(continued from previous page)

```
Problem Type: Binary Classification
Model Family: CatBoost

Pipeline Steps
==============
1. Imputer
         * categorical_impute_strategy : most_frequent
         * numeric_impute_strategy : mean
         * categorical_fill_value : None
         * numeric_fill_value : None
2. CatBoost Classifier
         * n_estimators : 10
         * eta : 0.03
         * max_depth : 6
         * bootstrap_type : None
```

[12]:

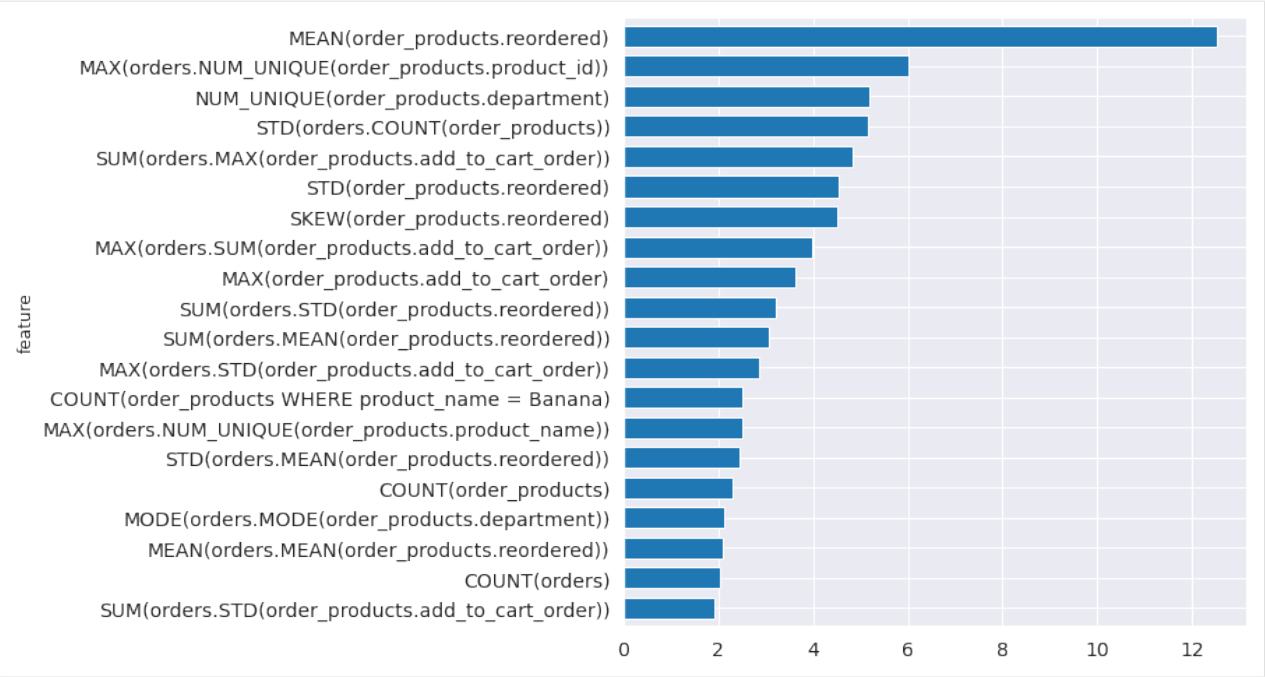
Now, let's score the model performance by evaluating predictions on the holdout set.

[13]:
```
best_pipeline = automl.best_pipeline.fit(X_train, y_train)

score = best_pipeline.score(
    X=X_holdout,
    y=y_holdout,
    objectives=['f1'],
)

dict(score)
```

[13]:
```
{'F1': 0.8}
```

From the pipeline, we can see which features are most important for predictions.

[14]:
```
feature_importance = best_pipeline.feature_importance
feature_importance = feature_importance.set_index('feature')['importance']
top_k = feature_importance.abs().sort_values().tail(20).index
feature_importance[top_k].plot.barh(figsize=(8, 8), fontsize=14, width=.7);
```



6.3.3 Next Steps

At this point, we have completed the machine learning application. We can revisit each step to explore and fine-tune with different parameters until the model is ready for deployment. For more information on how to work with the features produced by Featuretools, take a look at the Featuretools documentation. For more information on how to work with the models produced by EvalML, take a look at the EvalML documentation.

CHAPTER

SEVEN

PREDICT TURBOFAN DEGRADATION

In this example, we build a machine learning application to predict turbofan engine degradation. This application is structured into three important steps:

- Prediction Engineering
- Feature Engineering
- Machine Learning

In the first step, we generate new labels from the data by using Compose. In the second step, we generate features for the labels by using Featuretools. In the third step, we search for the best machine learning pipeline by using EvalML. After working through these steps, you will learn how to build machine learning applications for real-world problems like predictive maintenance. Let's get started.

[1]:
```
from demo.turbofan_degredation import load_sample
from matplotlib.pyplot import subplots
import composeml as cp
import featuretools as ft
import evalml
```

We will use a dataset provided by NASA simulating turbofan engine degradation. In this dataset, we have engines which are monitored over time. Each engine had operational settings and sensor measurements recorded for each cycle. The remaining useful life (RUL) is the amount of cycles an engine has left before it needs maintenance. What makes this dataset special is that the engines run all the way until failure, giving us precise RUL information for every engine at every point in time.

[2]:
```
records = load_sample()

records.head()
```

[2]:

id	engine_no	time_in_cycles	operational_setting_1	operational_setting_2	operational_setting_3	sensor_measurement_1	sensor_measurement_2
0	1	1	42.0049	0.8400	100.0	445.00	549.68
1	1	2	20.0020	0.7002	100.0	491.19	606.07
2	1	3	42.0038	0.8409	100.0	445.00	548.95
3	1	4	42.0000	0.8400	100.0	445.00	548.70
4	1	5	25.0063	0.6207	60.0	462.54	536.10

(continues on next page)

(continued from previous page)

```
    sensor_measurement_3  sensor_measurement_4  sensor_measurement_5  ...  \
id                                                                      ...
0                1343.43               1112.93                  3.91  ...
1                1477.61               1237.50                  9.35  ...
2                1343.12               1117.05                  3.91  ...
3                1341.24               1118.03                  3.91  ...
4                1255.23               1033.59                  7.05  ...

    sensor_measurement_13  sensor_measurement_14  sensor_measurement_15  \
id
0                 2387.99                8074.83                 9.3335
1                 2387.73                8046.13                 9.1913
2                 2387.97                8066.62                 9.4007
3                 2388.02                8076.05                 9.3369
4                 2028.08                7865.80                10.8366

    sensor_measurement_16  sensor_measurement_17  sensor_measurement_18  \
id
0                    0.02                    330                   2212
1                    0.02                    361                   2324
2                    0.02                    329                   2212
3                    0.02                    328                   2212
4                    0.02                    305                   1915

    sensor_measurement_19  sensor_measurement_20  sensor_measurement_21  \
id
0                  100.00                  10.62                 6.3670
1                  100.00                  24.37                14.6552
2                  100.00                  10.48                 6.4213
3                  100.00                  10.54                 6.4176
4                   84.93                  14.03                 8.6754

                   time
id
0  2000-01-01 00:00:00
1  2000-01-01 00:10:00
2  2000-01-01 00:20:00
3  2000-01-01 00:30:00
4  2000-01-01 00:40:00

[5 rows x 27 columns]
```

7.1 Prediction Engineering

> Which range is the RUL of a turbofan engine in?

In this prediction problem, we want to group the RUL into ranges. Then, predict which range the RUL is in. We can make variations of the ranges to create different prediction problems. For example, the ranges can be manually defined (0 - 150, 150 - 300, etc.) or based on the quartiles from historical observations. These variations can be done by simply binning the RUL. This helps us explore different scenarios which is crucial for making better decisions.

7.1.1 Defining the Labeling Process

Let's start by defining the labeling function of an engine that calculates the RUL. Given that engines run all the way until failure, the RUL is just the remaining number of observations.

[3]:
```
def rul(ds):
    return len(ds) - 1
```

7.1.2 Representing the Prediction Problem

Then, let's represent the prediction problem by creating a label maker with the following parameters:

- The `target_entity` as the column for the engine ID, since we want to process records for each engine.
- The `labeling_function` as the function we defined previously.
- The `time_index` as the column for the event time.

[4]:
```
lm = cp.LabelMaker(
    target_entity='engine_no',
    labeling_function=rul,
    time_index='time',
)
```

7.1.3 Finding the Training Examples

Now, let's run a search to get the training examples by using the following parameters:

- The data sorted by the event time.
- `num_examples_per_instance` as the number of training examples to find for each engine.
- `minimum_data` as the amount of data that will be used to make features for the first training example.
- `gap` as the number of rows to skip between examples. This is done to cover different points in time of an engine.

We can easily tweak these parameters and run more searches for training examples as the requirements of our model changes.

[5]:
```
lt = lm.search(
    records.sort_values('time'),
    num_examples_per_instance=20,
    minimum_data=5,
    gap=20,
    verbose=False,
)

lt.head()
```

[5]:

	engine_no	time	rul
0	1	2000-01-01 00:50:00	315
1	1	2000-01-01 04:10:00	295
2	1	2000-01-01 07:30:00	275
3	1	2000-01-01 10:50:00	255
4	1	2000-01-01 14:10:00	235

The output from the search is a label times table with three columns:

- The engine ID associated to the records.

- The event time of the engine. This is also known as a cutoff time for building features. Only data that existed beforehand is valid to use for predictions.
- The value of the RUL. This is calculated by our labeling function.

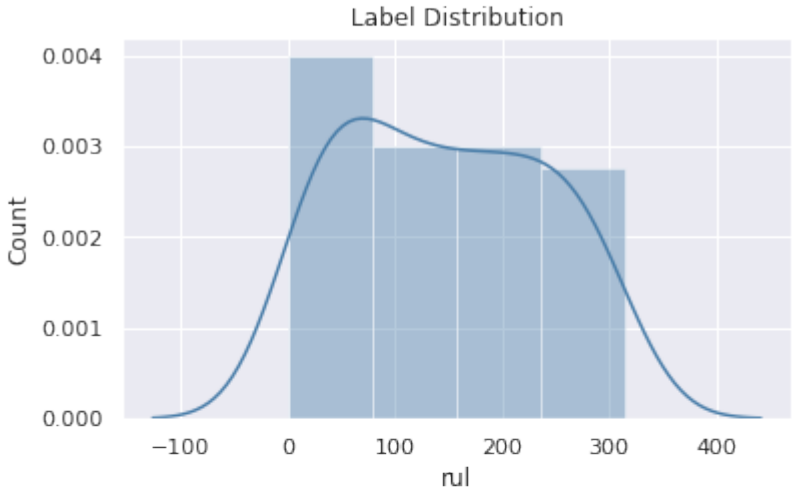
At this point, we only have continuous values of the RUL. As a helpul reference, we can print out the search settings that were used to generate these labels.

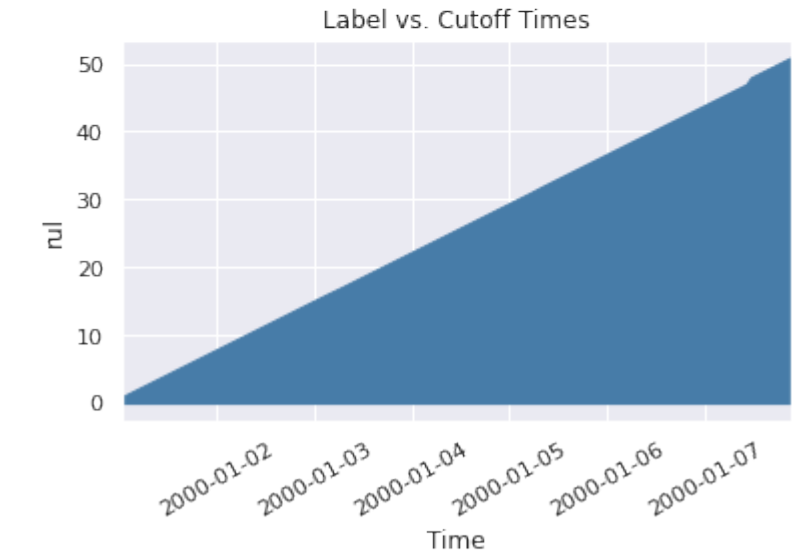
[6]:
```
lt.describe()
```

```
Settings
--------
gap                                 20
minimum_data                         5
num_examples_per_instance           20
target_column                      rul
target_entity                engine_no
target_type                 continuous
window_size                       None


Transforms
----------
No transforms applied
```

We can also get a better look at the values by plotting the distribution and the cumulative count across time.

[7]:
```
%matplotlib inline
fig, ax = subplots(nrows=2, ncols=1, figsize=(6, 8))
lt.plot.distribution(ax=ax[0])
lt.plot.count_by_time(ax=ax[1])
fig.tight_layout(pad=2)
```





With the continuous values, we can explore different ranges without running the search again. In this case, we will just use quartiles to bin the values into ranges.

[8]:
```
lt = lt.bin(4, quantiles=True, precision=0)
```

When we print out the settings again, we can now see that the description of the labels has been updated and reflects the latest changes.

[9]:
```
lt.describe()
```

```
Label Distribution
------------------
(0.0, 64.0]        13
(141.0, 227.0]     12
(227.0, 315.0]     13
(64.0, 141.0]      13
Total:             51
```

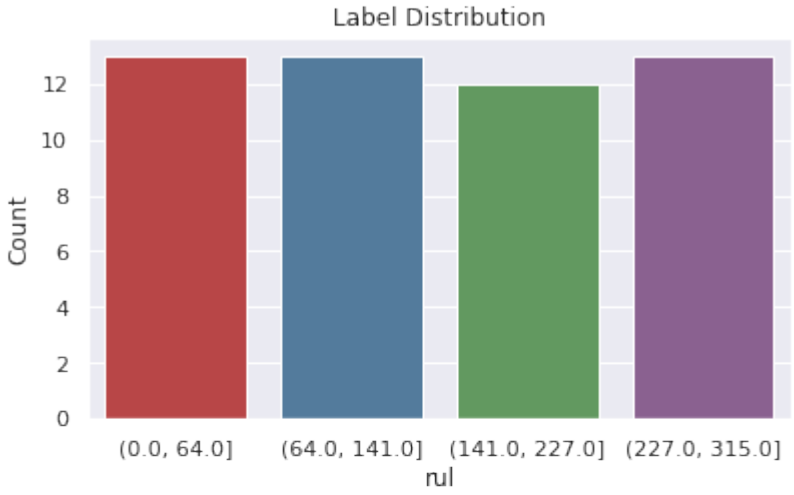
(continues on next page)

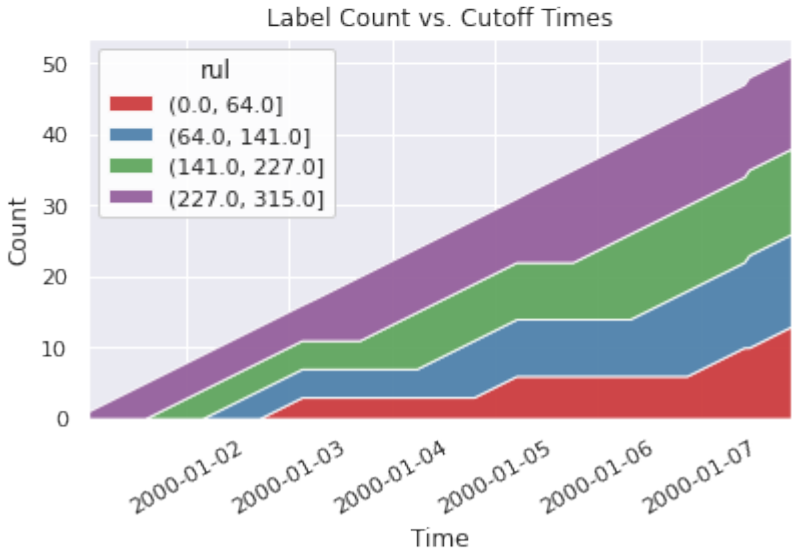
(continued from previous page)

```
Settings
--------
gap                                 20
minimum_data                         5
num_examples_per_instance           20
target_column                      rul
target_entity                engine_no
target_type                   discrete
window_size                       None


Transforms
----------
1. bin
  - bins:            4
  - labels:       None
  - precision:       0
  - quantiles:    True
  - right:        True
```

Let's have a look at the new label distribution and cumulative count across time.

[10]:
```
fig, ax = subplots(nrows=2, ncols=1, figsize=(6, 8))
lt.plot.distribution(ax=ax[0])
lt.plot.count_by_time(ax=ax[1])
fig.tight_layout(pad=2)
```





7.2 Feature Engineering

In the previous step, we generated the labels. The next step is to generate the features.

7.2.1 Representing the Data

We will represent the data using an entity set. We currently have a single table of records where one engine can many records. This one-to-many relationship can be represented in an entity set by normalizing an entity for the engines. The same can be done for engine cycles. Let's start by structuring the entity set.

[11]:
```
es = ft.EntitySet('observations')

es.entity_from_dataframe(
    dataframe=records.reset_index(),
    entity_id='records',
```

(continues on next page)

(continued from previous page)

```
    index='id',
    time_index='time',
)

es.normalize_entity(
    base_entity_id='records',
    new_entity_id='engines',
    index='engine_no',
)

es.normalize_entity(
    base_entity_id='records',
    new_entity_id='cycles',
    index='time_in_cycles',
)

es.plot()
```

[11]:

7.2.2 Calculating the Features

Now, we can generate features by using a method called Deep Feature Synthesis (DFS). This will automatically build features by stacking and applying mathematical operations called primitives across relationships in an entity set. The more structured an entity set is, the better DFS can leverage the relationships to generate better features. Let's run DFS using the following parameters:

- `entity_set` as the entity set we structured previously.
- `target_entity` as the engines, since we want to generate features for each engine.
- `cutoff_time` as the label times that we generated in the previous step.

[12]:
```
fm, fd = ft.dfs(
    entityset=es,
    target_entity='engines',
    agg_primitives=['sum'],
    trans_primitives=[],
    cutoff_time=lt,
    cutoff_time_in_index=True,
    include_cutoff_time=False,
    verbose=False,
)

fm.head()
```

[12]:
```
                             SUM(records.operational_setting_1)  \
engine_no time
1         2000-01-01 00:50:00                            171.0170
          2000-01-01 04:10:00                            639.0537
          2000-01-01 07:30:00                           1128.1028
          2000-01-01 10:50:00                           1600.1473
          2000-01-01 14:10:00                           2134.2039

                             SUM(records.operational_setting_2)  \
engine_no time
1         2000-01-01 00:50:00                              3.8418
          2000-01-01 04:10:00                             15.8218
```

(continues on next page)

(continued from previous page)

```
           2000-01-01 07:30:00                                28.0260
           2000-01-01 10:50:00                                39.2949
           2000-01-01 14:10:00                                51.8739

                              SUM(records.operational_setting_3)  \
engine_no time
1          2000-01-01 00:50:00                                 460.0
           2000-01-01 04:10:00                                2300.0
           2000-01-01 07:30:00                                4060.0
           2000-01-01 10:50:00                                5980.0
           2000-01-01 14:10:00                                7940.0

                              SUM(records.sensor_measurement_1)  \
engine_no time
1          2000-01-01 00:50:00                             2288.73
           2000-01-01 04:10:00                            11809.97
           2000-01-01 07:30:00                            21211.42
           2000-01-01 10:50:00                            30722.92
           2000-01-01 14:10:00                            40118.03

                              SUM(records.sensor_measurement_10)  \
engine_no time
1          2000-01-01 00:50:00                                5.04
           2000-01-01 04:10:00                               26.48
           2000-01-01 07:30:00                               47.58
           2000-01-01 10:50:00                               69.59
           2000-01-01 14:10:00                               91.37

                              SUM(records.sensor_measurement_11)  \
engine_no time
1          2000-01-01 00:50:00                              205.45
           2000-01-01 04:10:00                             1049.86
           2000-01-01 07:30:00                             1874.82
           2000-01-01 10:50:00                             2734.53
           2000-01-01 14:10:00                             3591.92

                              SUM(records.sensor_measurement_12)  \
engine_no time
1          2000-01-01 00:50:00                              865.90
           2000-01-01 04:10:00                             6212.29
           2000-01-01 07:30:00                            11083.94
           2000-01-01 10:50:00                            16627.16
           2000-01-01 14:10:00                            21653.49

                              SUM(records.sensor_measurement_13)  \
engine_no time
1          2000-01-01 00:50:00                            11579.79
           2000-01-01 04:10:00                            57897.83
           2000-01-01 07:30:00                           103495.65
           2000-01-01 10:50:00                           150532.67
           2000-01-01 14:10:00                           197930.92

                              SUM(records.sensor_measurement_14)  \
engine_no time
1          2000-01-01 00:50:00                            40129.43
           2000-01-01 04:10:00                           200818.30
           2000-01-01 07:30:00                           361091.30
```

(continues on next page)

(continued from previous page)

```
          2000-01-01 10:50:00                            522348.94
          2000-01-01 14:10:00                            683766.02

                                SUM(records.sensor_measurement_15)  ...  \
engine_no time                                                      ...
1         2000-01-01 00:50:00                              48.0990  ...
          2000-01-01 04:10:00                             236.7149  ...
          2000-01-01 07:30:00                             428.9520  ...
          2000-01-01 10:50:00                             613.1760  ...
          2000-01-01 14:10:00                             797.0065  ...

                                SUM(records.sensor_measurement_20)  \
engine_no time
1         2000-01-01 00:50:00                               70.04
          2000-01-01 04:10:00                              490.85
          2000-01-01 07:30:00                              880.67
          2000-01-01 10:50:00                             1312.52
          2000-01-01 14:10:00                             1706.80

                                SUM(records.sensor_measurement_21)  \
engine_no time
1         2000-01-01 00:50:00                               42.5365
          2000-01-01 04:10:00                             295.2742
          2000-01-01 07:30:00                             528.9393
          2000-01-01 10:50:00                             788.4988
          2000-01-01 14:10:00                            1025.8136

                                SUM(records.sensor_measurement_3)  \
engine_no time
1         2000-01-01 00:50:00                              6760.63
          2000-01-01 04:10:00                             34941.71
          2000-01-01 07:30:00                             62509.49
          2000-01-01 10:50:00                             90981.87
          2000-01-01 14:10:00                            119254.98

                                SUM(records.sensor_measurement_4)  \
engine_no time
1         2000-01-01 00:50:00                              5619.10
          2000-01-01 04:10:00                             29358.68
          2000-01-01 07:30:00                             52491.99
          2000-01-01 10:50:00                             76598.79
          2000-01-01 14:10:00                            100389.55

                                SUM(records.sensor_measurement_5)  \
engine_no time
1         2000-01-01 00:50:00                                28.13
          2000-01-01 04:10:00                               193.41
          2000-01-01 07:30:00                               349.26
          2000-01-01 10:50:00                               514.70
          2000-01-01 14:10:00                               663.76

                                SUM(records.sensor_measurement_6)  \
engine_no time
1         2000-01-01 00:50:00                               39.70
          2000-01-01 04:10:00                              276.33
          2000-01-01 07:30:00                              496.56
          2000-01-01 10:50:00                              736.42
```

(continues on next page)

(continued from previous page)

```
           2000-01-01 14:10:00                              953.22

                                SUM(records.sensor_measurement_7)  \
engine_no time
1          2000-01-01 00:50:00                              920.44
           2000-01-01 04:10:00                             6605.79
           2000-01-01 07:30:00                            11781.82
           2000-01-01 10:50:00                            17673.33
           2000-01-01 14:10:00                            23012.97

                                SUM(records.sensor_measurement_8)  \
engine_no time
1          2000-01-01 00:50:00                            10874.54
           2000-01-01 04:10:00                            55252.70
           2000-01-01 07:30:00                            98664.69
           2000-01-01 10:50:00                           143694.65
           2000-01-01 14:10:00                           188788.43

                                SUM(records.sensor_measurement_9)  \
engine_no time
1          2000-01-01 00:50:00                            41638.90
           2000-01-01 04:10:00                           211714.13
           2000-01-01 07:30:00                           379760.09
           2000-01-01 10:50:00                           550823.50
           2000-01-01 14:10:00                           721240.61

                                          rul
engine_no time
1          2000-01-01 00:50:00  (227.0, 315.0]
           2000-01-01 04:10:00  (227.0, 315.0]
           2000-01-01 07:30:00  (227.0, 315.0]
           2000-01-01 10:50:00  (227.0, 315.0]
           2000-01-01 14:10:00  (227.0, 315.0]

[5 rows x 25 columns]
```

There are two outputs from DFS: a feature matrix and feature definitions. The feature matrix is a table that contains the feature values based on the cutoff times from our labels. Feature definitions are features in a list that can be stored and reused later to calculate the same set of features on future data.

7.3 Machine Learning

In the previous steps, we generated the labels and features. The final step is to build the machine learning pipeline.

7.3.1 Splitting the Data

We will start by extracting the labels from the feature matrix and splitting the data into a training set and holdout set.

[13]:
```
y = fm.pop('rul').cat.codes

splits = evalml.preprocessing.split_data(
    X=fm,
    y=y,
    test_size=0.2,
    random_state=0,
)

X_train, X_holdout, y_train, y_holdout = splits
```

7.3.2 Finding the Best Model

Then, let's run a search on the training set for the best machine learning pipeline.

[14]:
```
automl = evalml.AutoMLSearch(
    problem_type='multiclass',
    objective='f1_macro',
)

automl.search(
    X_train,
    y_train,
    data_checks='disabled',
    show_iteration_plot=False,
)
```

```
Using default limit of max_pipelines=5.

Generating pipelines to search over...
*****************************
* Beginning pipeline search *
*****************************

Optimizing for F1 Macro.
Greater score is better.

Searching up to 5 pipelines.
Allowed model families: random_forest, linear_model, catboost, xgboost, extra_trees

(1/5) Mode Baseline Multiclass Classificati... Elapsed:00:00
        Starting cross validation
        Finished cross validation - mean F1 Macro: 0.092
(2/5) Extra Trees Classifier w/ Imputer        Elapsed:00:00
        Starting cross validation
        Finished cross validation - mean F1 Macro: 0.735
(3/5) Elastic Net Classifier w/ Imputer + S... Elapsed:00:01
        Starting cross validation
        Finished cross validation - mean F1 Macro: 0.190
(4/5) CatBoost Classifier w/ Imputer           Elapsed:00:01
        Starting cross validation
        Finished cross validation - mean F1 Macro: 0.839
(5/5) XGBoost Classifier w/ Imputer            Elapsed:00:01
```

(continues on next page)

(continued from previous page)

```
        Starting cross validation
        Finished cross validation - mean F1 Macro: 0.702

Search finished after 00:02
Best pipeline: CatBoost Classifier w/ Imputer
Best pipeline F1 Macro: 0.838823
```

Once the search is complete, we can print out information about the best pipeline found, such as the parameters in each component.

[15]:
```
automl.best_pipeline.describe()
automl.best_pipeline.graph()
```

```
*********************************
* CatBoost Classifier w/ Imputer *
*********************************

Problem Type: Multiclass Classification
Model Family: CatBoost

Pipeline Steps
==============
1. Imputer
         * categorical_impute_strategy : most_frequent
         * numeric_impute_strategy : mean
         * categorical_fill_value : None
         * numeric_fill_value : None
2. CatBoost Classifier
         * n_estimators : 10
         * eta : 0.03
         * max_depth : 6
         * bootstrap_type : None
```

[15]:

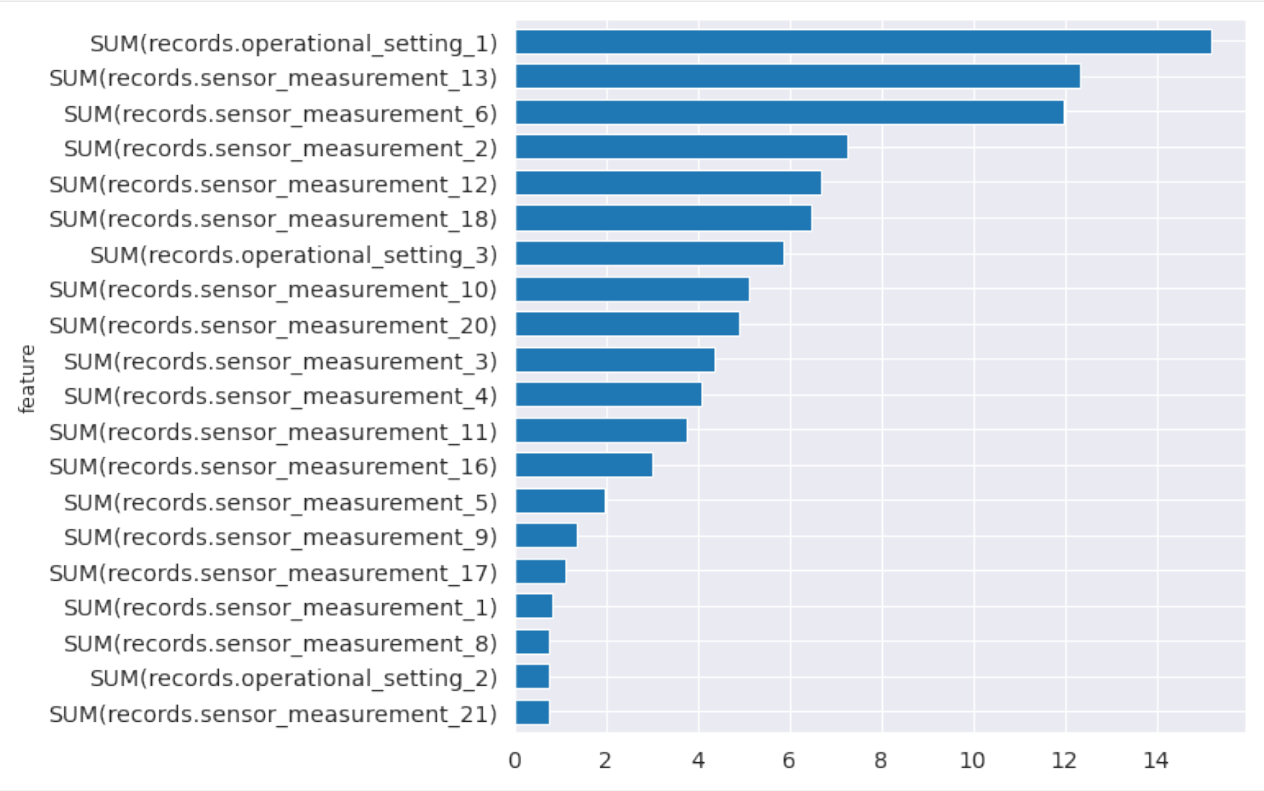
Now, let's score the model performance by evaluating predictions on the holdout set.

[16]:
```
best_pipeline = automl.best_pipeline.fit(X_train, y_train)

score = best_pipeline.score(
    X=X_holdout,
    y=y_holdout,
    objectives=['f1_macro'],
)

dict(score)
```

[16]:
```
{'F1 Macro': 0.8142857142857144}
```

From the pipeline, we can see which features are most important for predictions.

[17]:
```
feature_importance = best_pipeline.feature_importance
feature_importance = feature_importance.set_index('feature')['importance']
top_k = feature_importance.abs().sort_values().tail(20).index
feature_importance[top_k].plot.barh(figsize=(8, 8), fontsize=14, width=.7);
```



7.3.3 Next Steps

At this point, we have completed the machine learning application. We can revisit each step to explore and fine-tune with different parameters until the model is ready for deployment. For more information on how to work with the features produced by Featuretools, take a look at the Featuretools documentation. For more information on how to work with the models produced by EvalML, take a look at the EvalML documentation.

CHAPTER

EIGHT

FREQUENTLY ASKED QUESTIONS

8.1 I have heard of autoML and automated feature engineering, how is this different?

AutoML targets solving the problem once the labels or targets one wants to predict are well defined and are already available. Feature engineering focuses on generating features given a dataset, labels or targets. Both assume that the target a user wants to predict is already defined and computed. In most real world scenarios, this is something a data scientist has to do - define an outcome to predict and create labeled training examples. We structured this process and called it prediction engineering (a play on an already well defined process - feature engineering). This library provides an easy way for a user to define the target outcome and generate training examples automatically - from relational, temporal, multi entity datasets.

8.2 I have used Featuretools for competing in KAGGLE, how can I use Compose?

In most KAGGLE competitions the target to predict is already defined. In many cases, they follow the same way to represent training examples as us - "label times" (see here and here). Compose is a step prior to where KAGGLE starts. Indeed, it is a step that KAGGLE or the company sponsoring the competition may have to do or would have done before publishing the competition.

8.3 Why have I not encountered the need for Compose yet?

In many cases, setting up prediction problem is done independently before even getting started on the machine learning. This has resulted in a very skewed availability of datasets with already defined prediction problems and labels. A number of times it also results in a data scientist not knowing how the label was defined. For example, when given a list of , the data scientist does not know how the churn was defined. In opening up this part of the process, we are enabling data scientists to more flexibly define problems, explore more problems and solve problems to maximize the end goal - ROI.

8.4 I already have "Label times" file, do I need Compose?

If you already have label times you don't need LabelMaker and search. However, you could use the label transforms functionality of Compose, to apply lead, threshold, balance labels and all the other cool things that are yet to come.

8.5 What is the best use of Compose?

Since we have automated feature engineering and autoML, the best recommended use for Compose is to closely couple *LabelMaker* and *Search* functionality of Compose with the rest of the machine learning pipeline. Certain parameters used in *Search*, and *LabelMaker* and *label transforms* can be tuned alongside machine learning model. We have an end to end demo on this here.

8.6 Where can I read about your technical approach in detail?

You can read about prediction engineering, the way we defined the search algorithm and technical details in this peer reviewed paper published in IEEE international conference on data science and advanced analytics. If you're interested, you can also watch a video here. Please note that some of our thinking and terminology has evolved as we built this library and applied Compose to different industrial scale problems.

8.7 Do you think Compose should be part of a data scientist's toolkit?

Yes. As we mentioned above, extracting value out of your data is dependent on how you set the prediction problem. Currently, data scientists do not iterate through the setting up of the prediction problem because there is no structured way of doing it or algorithms and library to help do it. We believe that prediction engineering should be taken even more seriously than any other part of actually solving a problem.

8.8 How can I contribute labeling functions, or use cases?

We are happy for anyone who can provide interesting labeling functions. To contribute an interesting new use case and labeling function, we request you create a representative synthetic data set, a labeling function and the parameters for label maker. Once you have these three, you can write a brief explanation about the use case and do a pull request. To get a template for the pull request please see here.

8.9 I have a transaction file with the label as the last column, what are my label times?

Your label times is the . However, when such a data set is given one should ask for how that label was generated. It could be one of very many cases: a human could have assigned it based on their assessment/analysis, it could have been automatically generated by a system, or it could have been computed using some data. If it is the third case one should ask for the function that computed the label or rewrite it. If it is (1), one should note that the ref_time would be slightly after the transaction timestamp.

CHAPTER

NINE

API REFERENCE

9.1 Label Maker

LabelMaker	Automatically makes labels for prediction problems.

9.1.1 composeml.LabelMaker

class composeml.**LabelMaker**(*target_entity*, *time_index*, *labeling_function=None*, *window_size=None*)

Automatically makes labels for prediction problems.

Methods

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

composeml.LabelMaker.__init__

LabelMaker.**__init__**(*target_entity*, *time_index*, *labeling_function=None*, *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 or list(function) or dict(str=function)*) – Function, list of functions, or dictionary of functions that transform a data slice. When set as a dictionary, the key is used as the name of the labeling function.
- **window_size** (*str or int*) – Size of the data slices. As a string, the value can be a timedelta or a column in the data frame to group by. As an integer, the value can be the number of rows. Default value is all future data.

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 or dict*) – The expected number of examples to return from each entity group. A dictionary can be used to further specify the expected number of examples to return from each label.
- **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 lt (*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 df (DataFrame)

composeml.LabelMaker.slice

LabelMaker.**slice**(*df*, *num_examples_per_instance*, *minimum_data=None*, *gap=None*, *drop_empty=True*)

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.
- **drop_empty** (*bool*) – Whether to drop empty slices. Default value is True.

Returns Returns a generator of data slices.

Return type ds (generator)

9.2 Label Times

LabelTimes	The data frame that contains labels and cutoff times for the target entity.

9.2.1 composeml.LabelTimes

class composeml.**LabelTimes**(*data=None*, *target_entity=None*, *target_types=None*, *target_columns=None*, *search_settings=None*, *transforms=None*, **args*, ***kwargs*)

The data frame that contains labels and cutoff times for the target entity.

Methods

__init__	Initialize self.
apply_lead	Shifts the label times earlier for predicting in advance.
bin	Bin labels into discrete intervals.
copy	Make a copy of this object's indices and data.
describe	Prints out the settings used to make the label times.
equals	Determines if two label time objects are the same.
sample	Return a random sample of labels.
select	Selects one of the target variables.
threshold	Creates binary labels by testing if labels are above threshold.
to_csv	Write label times in csv format to disk.
to_parquet	Write label times in parquet format to disk.
to_pickle	Write label times in pickle format to disk.

composeml.LabelTimes.__init__

LabelTimes.**__init__**(*data=None*, *target_entity=None*, *target_types=None*, *target_columns=None*, *search_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*, *precision=3*)

Bin labels into discrete intervals.

Parameters

- **bins** (*int or array*) – The criteria to bin by. As an integer, the value can be the number of equal-width or quantile-based bins. If `quantiles` is False, the value is defined as 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, the value is defined as the number of quantiles (e.g. 10 for deciles, 4 for quartiles, etc.) As an array, the value can be custom or quantile-based edges. If `quantiles` is False, the value is defined as bin edges allowing for non-uniform width. No extension is done. If `quantiles` is True, the value is defined as bin edges usings 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.
- **precision** (*int*) – The precision at which to store and display the bins labels. Default value is 3.

Returns Instance of labels.

Return type *LabelTimes*

Examples

These are the target values for the examples.

```
>>> data = [226.93, 47.95, 283.46, 31.54]
>>> lt = LabelTimes({'target': data})
>>> lt
   target
0  226.93
1   47.95
```

(continues on next page)

(continued from previous page)

```
2  283.46
3   31.54
```

Bin values using equal-widths.

```
>>> lt.bin(2)
            target
0  (157.5, 283.46]
1  (31.288, 157.5]
2  (157.5, 283.46]
3  (31.288, 157.5]
```

Bin values using custom-widths.

```
>>> lt.bin([0, 200, 400])
       target
0  (200, 400]
1    (0, 200]
2  (200, 400]
3    (0, 200]
```

Bin values using infinite edges.

```
>>> lt.bin(['-inf', 100, 'inf'])
          target
0   (100.0, inf]
1  (-inf, 100.0]
2   (100.0, inf]
3  (-inf, 100.0]
```

Bin values using quartiles.

```
>>> lt.bin(4, quantiles=True)
                         target
0              (137.44, 241.062]
1               (43.848, 137.44]
2              (241.062, 283.46]
3  (31.538999999999998, 43.848]
```

Bin values using custom quantiles with precision.

```
>>> lt.bin([0, .5, 1], quantiles=True, precision=1)
           target
0  (137.4, 283.5]
1   (31.4, 137.4]
2  (137.4, 283.5]
3   (31.4, 137.4]
```

Assign labels to bins.

```
>>> lt.bin(2, labels=['low', 'high'])
  target
0   high
1    low
2   high
3    low
```

composeml.LabelTimes.copy

LabelTimes.**copy**(*deep=True*)
Make a copy of this object's indices and data.

Parameters **deep** (*bool*) – Make a deep copy, including a copy of the data and the indices. With deep=False neither the indices nor the data are copied. Default is True.

Returns A copy of the label times object.

Return type lt (*LabelTimes*)

composeml.LabelTimes.describe

LabelTimes.**describe**()
Prints out the settings used to make the label times.

composeml.LabelTimes.equals

LabelTimes.**equals**(*other*, ***kwargs*)
Determines if two label time objects are the same.

Parameters

- **other** (LabelTimes) – Other label time object for comparison.
- ****kwargs** – Keyword arguments to pass to underlying pandas.DataFrame.equals method

Returns Whether label time objects are the same.

Return type bool

composeml.LabelTimes.sample

LabelTimes.**sample**(*n=None*, *frac=None*, *random_state=None*, *replace=False*, *per_instance=False*)
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.
- **replace** (*bool*) – Sample with or without replacement. Default value is False.
- **per_instance** (*bool*) – Whether to apply sampling to each group. Default is False.

Returns Random sample of labels.

Return type *LabelTimes*

Examples

Create a label times object.

```
>>> entity = [0, 0, 1, 1]
>>> labels = [True, False, True, False]
>>> data = {'entity': entity, 'labels': labels}
>>> lt = LabelTimes(data=data, target_entity='entity', target_columns=['labels
↪'])
>>> lt
   entity  labels
0       0    True
1       0   False
2       1    True
3       1   False
```

Sample a number of the examples.

```
>>> lt.sample(n=3, random_state=0)
   entity  labels
1       0   False
2       1    True
3       1   False
```

Sample a fraction of the examples.

```
>>> lt.sample(frac=.25, random_state=0)
   entity  labels
2       1    True
```

Sample a number of the examples for specific labels.

```
>>> n = {True: 1, False: 1}
>>> lt.sample(n=n, random_state=0)
   entity  labels
2       1    True
3       1   False
```

Sample a fraction of the examples for specific labels.

```
>>> frac = {True: .5, False: .5}
>>> lt.sample(frac=frac, random_state=0)
   entity  labels
2       1    True
3       1   False
```

Sample a number of the examples from each entity group.

```
>>> lt.sample(n={True: 1}, per_instance=True, random_state=0)
   entity  labels
0       0    True
2       1    True
```

Sample a fraction of the examples from each entity group.

```
>>> lt.sample(frac=.5, per_instance=True, random_state=0)
   entity  labels
```

(continues on next page)

(continued from previous page)

```
1       0   False
3       1   False
```

composeml.LabelTimes.select

LabelTimes.**select**(*target*)

Selects one of the target variables.

Parameters **target** (*str*) – The name of the target column.

Returns A label times object that contains a single target.

Return type lt (*LabelTimes*)

Examples

Create a label times object that contains multiple target variables.

```
>>> entity = [0, 0, 1, 1]
>>> labels = [True, False, True, False]
>>> time = ['2020-01-01', '2020-01-02', '2020-01-03', '2020-01-04']
>>> data = {'entity': entity, 'time': time, 'A': labels, 'B': labels}
>>> lt = LabelTimes(data=data, target_entity='entity', target_columns=['A', 'B
↪'])
>>> lt
   entity        time      A      B
0       0  2020-01-01   True   True
1       0  2020-01-02  False  False
2       1  2020-01-03   True   True
3       1  2020-01-04  False  False
```

Select a single target from the label times.

```
>>> lt.select('B')
   entity        time      B
0       0  2020-01-01   True
1       0  2020-01-02  False
2       1  2020-01-03   True
3       1  2020-01-04  False
```

composeml.LabelTimes.threshold

LabelTimes.**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*)

composeml.LabelTimes.to_csv

LabelTimes.**to_csv**(*path*, *save_settings=True*, ***kwargs*)
Write label times in csv format to disk.

Parameters

- **path** (*str*) – Location on disk to write to (will be created as a directory).
- **save_settings** (*bool*) – Whether to save the settings used to make the label times.
- ****kwargs** – Keyword arguments to pass to underlying pandas.DataFrame.to_csv method

composeml.LabelTimes.to_parquet

LabelTimes.**to_parquet**(*path*, *save_settings=True*, ***kwargs*)
Write label times in parquet format to disk.

Parameters

- **path** (*str*) – Location on disk to write to (will be created as a directory).
- **save_settings** (*bool*) – Whether to save the settings used to make the label times.
- ****kwargs** – Keyword arguments to pass to underlying pandas.DataFrame.to_parquet method

composeml.LabelTimes.to_pickle

LabelTimes.**to_pickle**(*path*, *save_settings=True*, ***kwargs*)
Write label times in pickle format to disk.

Parameters

- **path** (*str*) – Location on disk to write to (will be created as a directory).
- **save_settings** (*bool*) – Whether to save the settings used to make the label times.
- ****kwargs** – Keyword arguments to pass to underlying pandas.DataFrame.to_pickle method

9.2.2 Transform Methods

LabelTimes.apply_lead	Shifts the label times earlier for predicting in advance.
LabelTimes.bin	Bin labels into discrete intervals.
LabelTimes.sample	Return a random sample of labels.
LabelTimes.threshold	Creates binary labels by testing if labels are above threshold.

9.3 Label Plots

LabelPlots	Creates plots for Label Times.

9.3.1 composeml.label_times.plots.LabelPlots

class composeml.label_times.plots.**LabelPlots**(*label_times*)

Creates plots for Label Times.

Methods

__init__	Initializes Label Plots.
count_by_time	Plots the label distribution across cutoff times.
distribution	Plots the label distribution.

composeml.label_times.plots.LabelPlots.__init__

LabelPlots.**__init__**(*label_times*)

Initializes Label Plots.

Parameters **label_times** (LabelTimes) – instance of Label Times

composeml.label_times.plots.LabelPlots.count_by_time

LabelPlots.**count_by_time**(*ax=None*, ***kwargs*)

Plots the label distribution across cutoff times.

composeml.label_times.plots.LabelPlots.distribution

LabelPlots.**distribution**(***kwargs*)

Plots the label distribution.

9.3.2 Plotting Methods

LabelPlots.count_by_time	Plots the label distribution across cutoff times.
LabelPlots.distribution	Plots the label distribution.

CHAPTER

TEN

CHANGELOG

v0.5.0 August 28, 2020

- **Enhancements**
  - Added Column-Based Windows (#151).
- **Changes**
  - Refactored Data Slice Generator (#150).
- **Documentation Changes**
  - Updated README (#164).
  - Updated Predict Next Purchase Demo (#154).
  - Updated Predict Turbofan Degradation Demo (#154).

Breaking Changes

- Attributes of the data slice context have changed. Inside a labeling function, the timestamps of a data slice can be referenced by `ds.context.slice_start` and `ds.context.slice_stop`. For more details, see the Data Slice Generator Guide.

v0.4.0 July 2, 2020

- **Enhancements**
  - Target values can be sampled from each group (#138).
  - One of multiple targets can be selected (#147).
  - Labels can be binned using infinite edges represented as string (#133).
- **Changes**
  - The label times object was refactored to improve design and structure (#135).

Breaking Changes

- Loading label times from previous versions will result in an error.

v0.3.0 June 1, 2020

- **Enhancements**
  - Label Search for Multiple Targets (#130)
- **Changes**
  - Column renamed from `cutoff_time` to `time` (#139)

v0.2.0 April 23, 2020

- **Changes**
  - Dropped Support for Python 3.5 (#128)
  - Rename LabelTimes.name to LabelTimes.label_name (#126)
  - Support keyword arguments in Pandas methods. (#121)
- **Documentation Changes**
  - Improved data download in Predict Next Purchase (#76)
- **Testing Changes**
  - Added tests that use Python 3.8 in CirlceCI (#128)

Breaking Changes

- `LabelTimes.name` has been renamed to `LabelTimes.label_name`

v0.1.8 March 11, 2020

- **Fixes**
  - Support for Pandas 1.0

v0.1.7 January 31, 2020

- **Enhancements**
  - Added higher-level mappings to offsets.
  - Track settings for sample transforms.
- **Fixes**
  - Pinned Pandas version.
- **Testing Changes**
  - Moved Featuretools to test requirements.

v0.1.6 October 22, 2019

- **Enhancements**
  - Serialization for Label Times
- **Fixes**
  - Matplotlib Backend Fix
  - Sampling Label Times
- **Documentation Changes**
  - Added Data Slice Generator Guide
- **Testing Changes**
  - Integration Tests for Python Versions 3.6 and 3.7

v0.1.5 September 16, 2019

- **Enhancements**
  - Added Slice Generator
  - Added Seaborn Plots
  - Added Data Slice Context

    - Added Count per Group
- **Documentation Changes**
    - Updated README
    - Added Example: Predict Next Purchase
    - Added Example: Predict RUL

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

INDEX

Symbols

A

B

C

D

E

L

S

T