

---

# **Compose Documentation**

*Release 0.1.6*

**Feature Labs, Inc.**

**Nov 04, 2019**



---

## Table of Contents

---

<b>1</b>	<b>Install</b>	<b>3</b>
<b>2</b>	<b>Getting Started</b>	<b>5</b>
2.1	Load Data . . . . .	5
2.2	Create Labeling Function . . . . .	6
2.3	Construct Label Maker . . . . .	6
2.4	Generate Labels . . . . .	6
2.5	Transform Labels . . . . .	6
2.6	Describe Labels . . . . .	7
2.7	Plot Labels . . . . .	8
<b>3</b>	<b>Main Concepts</b>	<b>11</b>
3.1	Label Maker . . . . .	11
<b>4</b>	<b>Data Slice Generator</b>	<b>13</b>
4.1	Labeling Function . . . . .	13
4.2	Data Slices . . . . .	14
4.3	Data Slice Context . . . . .	20
<b>5</b>	<b>Using Compose with Featuretools</b>	<b>21</b>
5.1	Load Data . . . . .	21
5.2	Generate Labels . . . . .	22
5.3	Generate Features . . . . .	24
5.4	Machine Learning . . . . .	26
<b>6</b>	<b>Using Label Transforms</b>	<b>31</b>
6.1	Generate Labels . . . . .	31
6.2	Threshold on Labels . . . . .	32
6.3	Lead Labels Times . . . . .	32
6.4	Bin Labels . . . . .	32
6.5	Describe Labels . . . . .	34
6.6	Sample Labels . . . . .	35
<b>7</b>	<b>Predict Next Purchase</b>	<b>37</b>
7.1	Load Data . . . . .	37
7.2	Generate Labels . . . . .	38

<b>8</b>	<b>Predict Remaining Useful Life</b>	<b>41</b>
8.1	Load Data . . . . .	41
8.2	Generate Labels . . . . .	42
8.3	Generate Features . . . . .	46
8.4	Machine Learning . . . . .	50
<b>9</b>	<b>Frequently Asked Questions</b>	<b>55</b>
9.1	I have heard of autoML and automated feature engineering, how is this different? . . . . .	55
9.2	I have used Featuretools for competing in KAGGLE, how can I use Compose? . . . . .	55
9.3	Why have I not encountered the need for Compose yet? . . . . .	55
9.4	I already have “Label times” file, do I need Compose? . . . . .	56
9.5	What is the best use of Compose? . . . . .	56
9.6	Where can I read about your technical approach in detail? . . . . .	56
9.7	Do you think Compose should be part of a data scientist’s toolkit? . . . . .	56
9.8	How can I contribute labeling functions, or use cases? . . . . .	56
9.9	I have a transaction file with the label as the last column, what are my label times? . . . . .	56
<b>10</b>	<b>API Reference</b>	<b>57</b>
10.1	Label Maker . . . . .	57
10.2	Label Times . . . . .	59
10.3	Label Plots . . . . .	64
<b>11</b>	<b>Changelog</b>	<b>67</b>
	<b>Index</b>	<b>69</b>



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

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



# CHAPTER 1

---

## Install

---

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

```
pip install compose1
```





In this example, we will generate labels on a mock dataset of transactions. For each customer, we want to label whether the total purchase amount over the next hour of transactions will exceed \$300. Additionally, we want to predict one hour in advance.

```
[1]: %matplotlib inline
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	cutoff_time	total_spent
id			
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	cutoff_time	total_spent
id			
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	cutoff_time	total_spent
id			
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
label_type         discrete
labeling_function  total_spent
minimum_data       None
num_examples_per_instance -1
target_entity      customer_id
window_size        <Hour>
```

(continues on next page)

(continued from previous page)

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

2. apply_lead
   - value:    1h
```

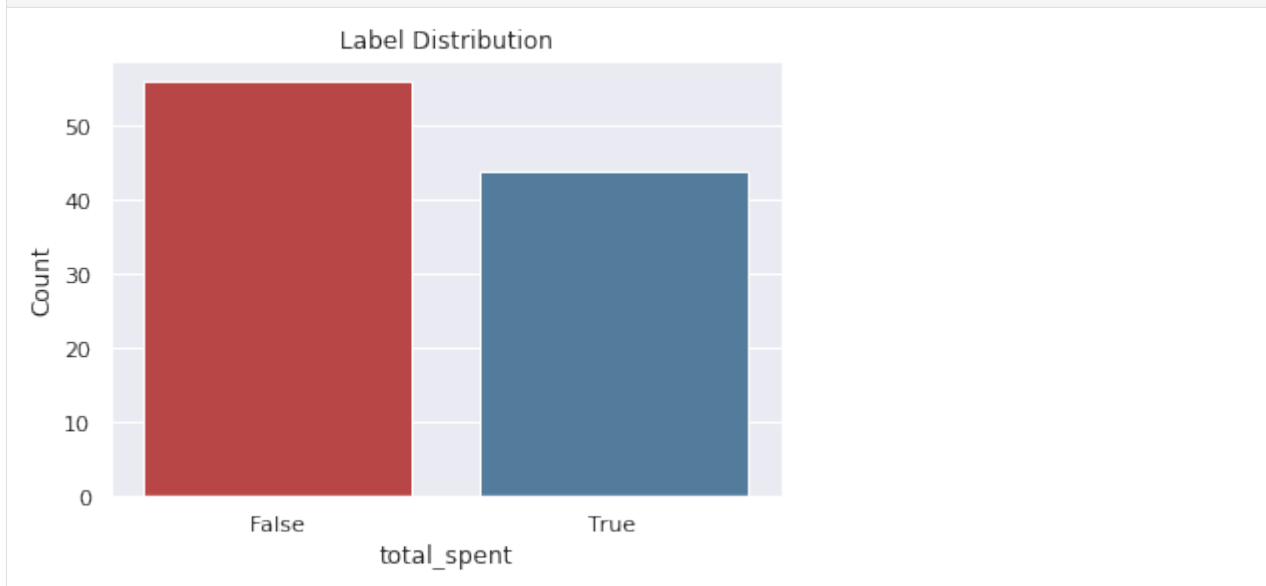
## 2.7 Plot Labels

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

### 2.7.1 Distribution

This plot shows the label distribution.

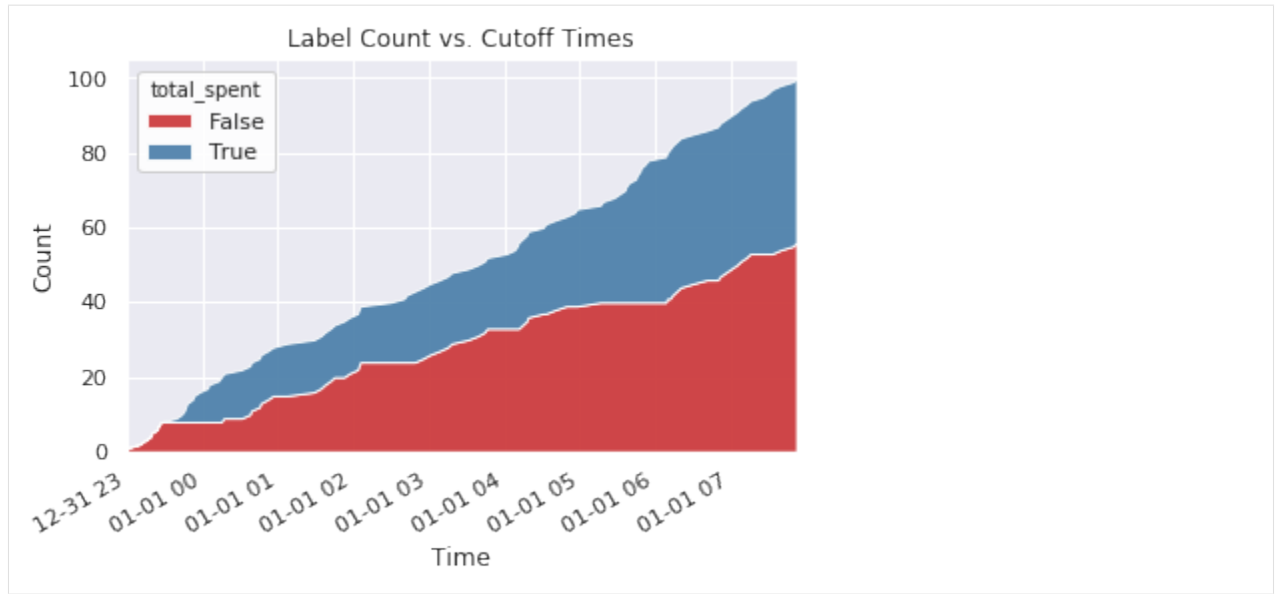
```
[9]: labels.plot.distribution();
```



### 2.7.2 Count by Time

This plot shows the label distribution across cutoff times.

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





### 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.





---

## 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):
      total = df['amount'].sum()
      return total
```

## 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)
      ds
```

```

slice_number          1
customer_id          1
window               [2014-01-01 00:00:00, 2014-01-01 02:00:00)
gap                  [2014-01-01 00:00:00, 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)
      ds

```

```

slice_number          2
customer_id          1
window               [2014-01-01 02:00:00, 2014-01-01 04:00:00)
gap                  [2014-01-01 02:00:00, 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

```

(continues on next page)

(continued from previous page)

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 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)
        ds
```

slice_number	1
customer_id	1
window	[2014-01-01 00:00:00, 2014-01-01 03:00:00)
gap	[2014-01-01 00:00:00, 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)
ds

slice_number      2
customer_id       1
window  [2014-01-01 01:00:00, 2014-01-01 04:00:00)
gap      [2014-01-01 01:00:00, 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
2014-01-01 01:37:30      372         6         5 114.84
```

(continues on next page)

(continued from previous page)

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

transaction_time	customer_id	device
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]: 821.64000000000001
```

### 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)
ds
slice_number          1
customer_id           1
window                [2014-01-01 00:00:00, 2014-01-01 01:00:00)
gap                   [2014-01-01 00:00:00, 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)
ds
slice_number          2
customer_id           1
window                [2014-01-01 03:00:00, 2014-01-01 04:00:00)
gap                   [2014-01-01 03:00:00, 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]: {'gap': (Timestamp('2014-01-01 03:00:00'), Timestamp('2014-01-01 06:00:00')),  
      'window': (Timestamp('2014-01-01 03:00:00'),  
                Timestamp('2014-01-01 04:00:00')),  
      'slice_number': 2,  
      'target_entity': 'customer_id',  
      'target_instance': 1}
```

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



---

## Using Compose with Featuretools

---

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

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

### 5.1 Load Data

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

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

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

```
[2]:
```

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

	customer_id	device
0	2	desktop

(continues on next page)

(continued from previous page)

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

## 5.2 Generate Labels

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

### 5.2.1 Create Labeling Function

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

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

### 5.2.2 Construct Label Maker

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

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

### 5.2.3 Create Labels

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

#### See also:

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

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

labels.head()

Elapsed: 00:01 | Remaining: 00:00 | Progress: 100%| customer_id: 5/5
```

```
[5]:
```

	customer_id	cutoff_time	total_spent
id			
0	1	2014-01-01 00:44:25	2880.53
1	1	2014-01-01 00:45:30	2859.18
2	1	2014-01-01 00:46:35	2751.07
3	1	2014-01-01 00:47:40	2638.54
4	1	2014-01-01 00:48:45	2632.25

## 5.2.4 Transform Labels

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

### Apply Threshold on Labels

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

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

labels.head()

[6]:
```

	customer_id	cutoff_time	total_spent
id			
0	1	2014-01-01 00:44:25	True
1	1	2014-01-01 00:45:30	True
2	1	2014-01-01 00:46:35	True
3	1	2014-01-01 00:47:40	True
4	1	2014-01-01 00:48:45	True

### Lead Label Times

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

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

labels.head()

[7]:
```

	customer_id	cutoff_time	total_spent
id			
0	1	2013-12-31 23:44:25	True
1	1	2013-12-31 23:45:30	True
2	1	2013-12-31 23:46:35	True
3	1	2013-12-31 23:47:40	True
4	1	2013-12-31 23:48:45	True

## 5.2.5 Describe Labels

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

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

```
Label Distribution
-----
True      252
False     248
Total:    500

Settings
-----
num_examples_per_instance    -1
minimum_data                 None
window_size                  <Hour>
gap                           1

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

2. apply_lead
   - value:    1h
```

## 5.3 Generate Features

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

### 5.3.1 Create Entity Set

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

#### See also:

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

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

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

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

(continues on next page)

(continued from previous page)

```

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

es.normalize_entity(
    base_entity_id='sessions',
    new_entity_id='customers',
    index='customer_id',
    make_time_index='join_date',
    additional_variables=[
        'zip_code',
        'join_date',
        'date_of_birth',
    ],
)

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

es.add_last_time_indexes()
```

### 5.3.2 Describe Entity Set

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

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

```
es.plot()
```

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

```
[10]:
```

### 5.3.3 Create Feature Matrix

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

**See also:**

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

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

Built 77 features
Elapsed: 02:32 | Progress: 100%| Remaining: 00:00
```

### 5.3.4 Describe Features

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

```
[12]: features_defs[:20]
[12]: [<Feature: zip_code>,
      <Feature: COUNT(sessions)>,
      <Feature: NUM_UNIQUE(sessions.device)>,
      <Feature: MODE(sessions.device)>,
      <Feature: SUM(transactions.amount)>,
      <Feature: STD(transactions.amount)>,
      <Feature: MAX(transactions.amount)>,
      <Feature: SKEW(transactions.amount)>,
      <Feature: MIN(transactions.amount)>,
      <Feature: MEAN(transactions.amount)>,
      <Feature: COUNT(transactions)>,
      <Feature: NUM_UNIQUE(transactions.product_id)>,
      <Feature: MODE(transactions.product_id)>,
      <Feature: DAY(join_date)>,
      <Feature: DAY(date_of_birth)>,
      <Feature: YEAR(join_date)>,
      <Feature: YEAR(date_of_birth)>,
      <Feature: MONTH(join_date)>,
      <Feature: MONTH(date_of_birth)>,
      <Feature: WEEKDAY(join_date)>]
```

## 5.4 Machine Learning

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

### 5.4.1 Preprocess Features

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

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

### 5.4.2 Split Labels and Features

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

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

### 5.4.3 Train Model

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

```
[15]: clf = RandomForestClassifier(n_estimators=10, random_state=0)
clf.fit(x_train, y_train)
[15]: RandomForestClassifier(bootstrap=True, class_weight=None, criterion='gini',
    max_depth=None, max_features='auto', max_leaf_nodes=None,
    min_impurity_decrease=0.0, min_impurity_split=None,
    min_samples_leaf=1, min_samples_split=2,
    min_weight_fraction_leaf=0.0, n_estimators=10,
    n_jobs=None, oob_score=False, random_state=0, verbose=0,
    warm_start=False)
```

### 5.4.4 Test Model

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

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

	precision	recall	f1-score	support
False	0.80	0.86	0.83	50
True	0.85	0.78	0.81	50
accuracy			0.82	100
macro avg	0.82	0.82	0.82	100
weighted avg	0.82	0.82	0.82	100

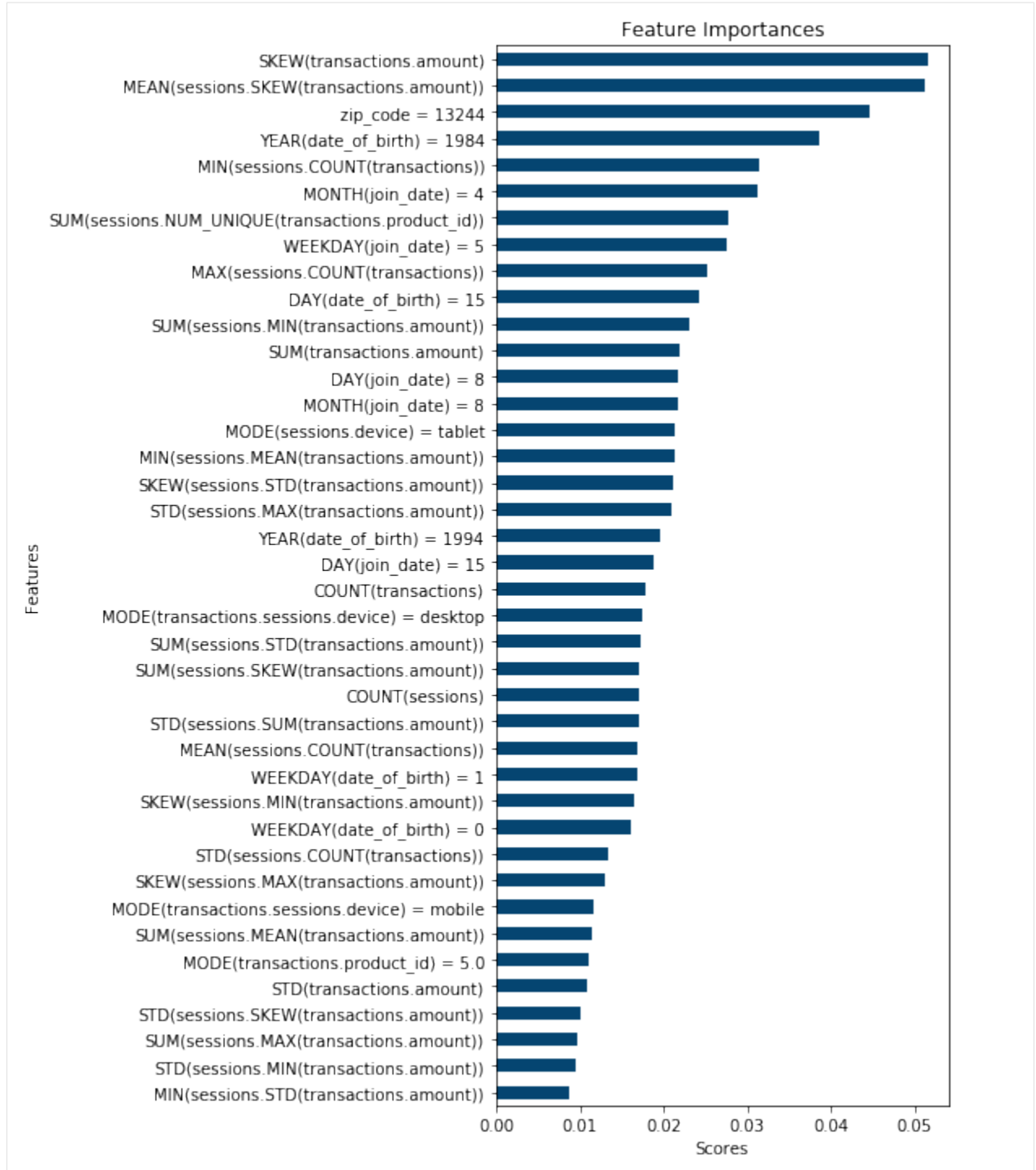
## 5.4.5 Feature Importances

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

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

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







---

## Using Label Transforms

---

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

### 6.1 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,
    label_type='continuous',
    minimum_data='2h',
    gap='2min',
    verbose=True,
)

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

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

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

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

## 6.2 Threshold on Labels

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

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

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

## 6.3 Lead Labels Times

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

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

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

## 6.4 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.

### 6.4.1 Value Based

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

## Equal Width

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

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

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

## Custom Widths

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

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

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

## 6.4.2 Quantile Based

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

### Equal Width

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

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

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

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

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

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

### Custom Widths

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

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

```
[9]:
```

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

### 6.4.3 Label Bins

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

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

```
[10]:
```

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

## 6.5 Describe Labels

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

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

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

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

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

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

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

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

```
[13]:
```

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

## 6.6.1 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      cutoff_time total_spent
id
46           5 2014-01-01 03:58:25           A
42           5 2014-01-01 03:50:25           A
26           3 2014-01-01 03:57:05           B
48           5 2014-01-01 04:02:25           B
6            1 2014-01-01 02:57:30           C
38           4 2014-01-01 02:55:00           C
11           2 2014-01-01 02:02:00           D
16           2 2014-01-01 02:12:00           D
```

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      cutoff_time total_spent
id
46           5 2014-01-01 03:58:25           A
26           3 2014-01-01 03:57:05           B
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
```



---

## Predict Next Purchase

---

In this example, we will generate labels on online grocery orders provided by Instacart using Compose. The labels can be used to train a machine learning model to predict whether a customer will buy a specific product within the next month.

If you plan to run this notebook, you can use the following command at the root directory of the repository.

```
jupyter notebook docs/source/examples/predict-next-purchase/example.ipynb
```

### 7.1 Load Data

```
[1]: %matplotlib inline
import composeml as cp
import data
```

The data hosted [here](#) will be downloaded automatically into the `data` module of this notebook unless it already exist. Once the data is in place, we can preview the grocery orders to see how they look.

```
[2]: df = data.load_orders(nrows=1000000)
```

```
df.head()
```

```
Downloaded: 206MB / 206MB - 1.04s/MB, Elapsed: 03:34, Remaining: 00:00, Progress: 100%|
↪100%|
```

```
[2]:
```

	order_id	product_id	add_to_cart_order	reordered	\
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 \
```

(continues on next page)

(continued from previous page)

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
0	23750	2015-01-11 08:00:00
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

## 7.2 Generate Labels

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

### 7.2.1 Create Labeling Function

To get started, we define the labeling function that will return whether a customer purchased the product in a given month.

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

### 7.2.2 Construct Label Maker

With the labeling function, we create the label maker for our prediction problem. To process one month of orders for each customer, we set the `target_entity` to the customer ID and the `window_size` to one month. When window size is set to `1MS`, the window size will end on the first day of the next month. Alias definitions are listed here.

```
[4]: lm = cp.LabelMaker(
      target_entity='user_id',
      time_index='order_time',
      labeling_function=bought_product,
      window_size='1MS',
    )
```

### 7.2.3 Search Labels

Next, the label maker will search through the data continuously to label whether a customer bought bananas in a given month. This happens when we use `LabelMaker.search` and set the `product_name` to `bananas`. If you are running this code yourself, feel free to experiment with other products (e.g. limes, avocados, etc.) and different time frames!

```
[5]: lt = lm.search(
      df.sort_values('order_time'),
      minimum_data='2015-01-01',
      num_examples_per_instance=-1,
      product_name='Banana',
      verbose=True,
    )

lt.head()
```

```
Elapsed: 00:38 | Remaining: 00:00 | Progress: 100%|| user_id: 19477/19477
```

```
[5]:   bought_product  cutoff_time  user_id
id
0           False  2015-01-01         4
1           False  2015-01-01         7
2           False  2015-01-01        10
3           False  2015-02-01        10
4           False  2015-01-01        13
```

## 7.2.4 Describe Labels

With the generate label times, 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.

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

```
Label Distribution
-----
False      13752
True       7044
Total:     20796
```

```
Settings
```

```
-----
gap                None
label_type         discrete
labeling_function  bought_product
minimum_data       2015-01-01
num_examples_per_instance -1
target_entity      user_id
window_size        <MonthBegin>
```

```
Transforms
```

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

## 7.2.5 Plot Labels

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

## Distribution

This plot shows the label distribution.

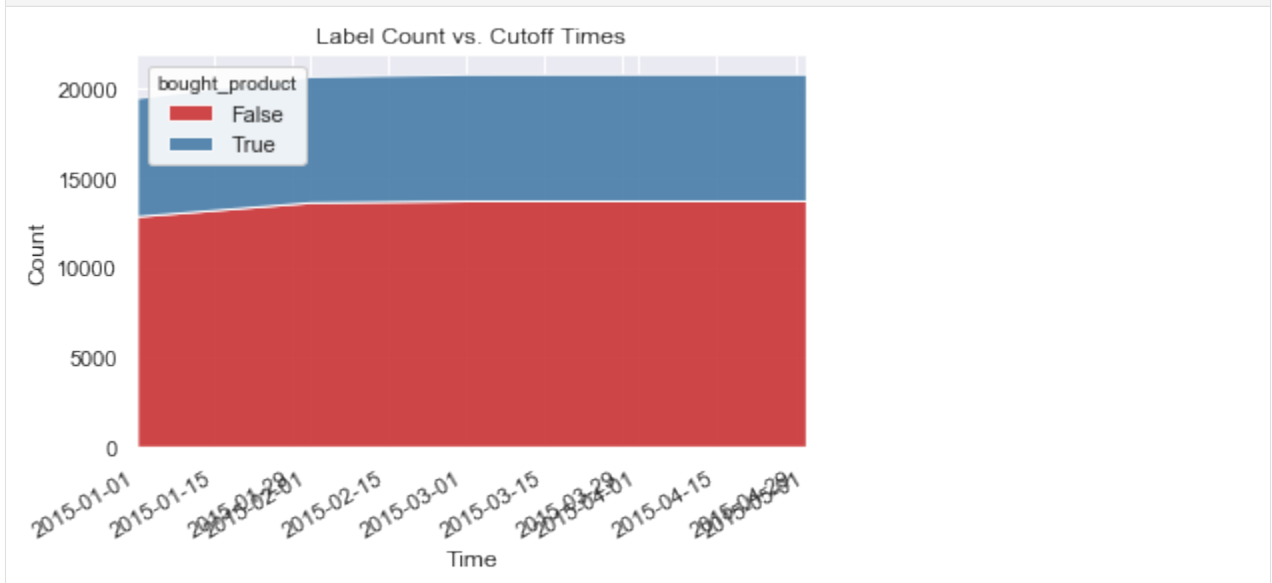
```
[7]: lt.plot.distribution();
```



## Count by Time

This plot shows the label distribution across cutoff times.

```
[8]: lt.plot.count_by_time();
```



---

## Predict Remaining Useful Life

---

In this example, we will generate labels using Compose on data provided by NASA simulating turbofan engine degradation. Then, the labels are used to generate features and train a machine learning model to predict the Remaining Useful Life (RUL) of an engine.

```
[1]: import composeml as cp
import featuretools as ft
import pandas as pd
import data

from sklearn.model_selection import train_test_split
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import classification_report
```

### 8.1 Load Data

In this dataset, we have 249 engines (`engine_no`) which are monitored over time (`time_in_cycles`). 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.

You can download the data directly from NASA [here](#). After downloading the data, you can set the `file` parameter as an absolute path to `train_FD004.txt`. With the file in place, we preview the data to get an idea on how to observations look.

```
[2]: df = data.load('data/train_FD004.txt')
df[df.columns[:7]].head()
```

```
[2]:   engine_no  time_in_cycles  operational_setting_1  operational_setting_2  \
0           1                1                42.0049                0.8400
1           1                2                20.0020                0.7002
2           1                3                42.0038                0.8409
```

(continues on next page)

(continued from previous page)

3	1	4	42.0000	0.8400
4	1	5	25.0063	0.6207
	operational_setting_3	sensor_measurement_1	sensor_measurement_2	
0	100.0	445.00	549.68	
1	100.0	491.19	606.07	
2	100.0	445.00	548.95	
3	100.0	445.00	548.70	
4	60.0	462.54	536.10	

## 8.2 Generate Labels

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

### 8.2.1 Define Labeling Function

To get started, we define the labeling function that will return the RUL given the remaining observations of an engine.

```
[3]: def remaining_useful_life(df):
      return len(df) - 1
```

### 8.2.2 Create Label Maker

With the labeling function, we create the label maker for our prediction problem. To process the RUL for each engine, we set the `target_entity` to the engine number. By default, the `window_size` is set to the total observation size to contain the remaining observations for each engine.

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

### 8.2.3 Search Labels

Let's imagine we want to make predictions on turbines that are up and running. Turbines in general don't fail before 120 cycles, so we will only make labels for engines that reach at least 100 cycles. To do this, the `minimum_data` parameter is set to 100. Using Compose, we can easily tweak this parameter as the requirements of our model changes. Additionally, we set `gap` to one to create labels on every cycle and limit the search to 10 examples for each engine.

#### See also:

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

```
[5]: lt = lm.search(
      df.sort_values('time'),
      num_examples_per_instance=10,
      minimum_data=100,
      gap=1,
      verbose=True,
```

(continues on next page)

(continued from previous page)

```
)
lt.head()
Elapsed: 00:02 | Remaining: 00:00 | Progress: 100%| engine_no: 2490/2490
[5]: engine_no      cutoff_time  remaining_useful_life
id
0          1 2000-01-01 16:40:00          220
1          1 2000-01-01 16:50:00          219
2          1 2000-01-01 17:00:00          218
3          1 2000-01-01 17:10:00          217
4          1 2000-01-01 17:20:00          216
```

## 8.2.4 Continuous Labels

The labeling function we defined returns continuous labels which can be used to train a regression model for our predictin problem. Alternatively, there are label transforms available to further process these labels into discrete values. In which case, can be used to train a classification model.

### Describe Labels

Let's print out the settings and transforms that were used to make the continuous labels. This is useful as a reference for understanding how the labels were generated from raw data.

```
[6]: lt.describe()
Settings
-----
label_type                continuous
labeling_function         remaining_useful_life
num_examples_per_instance      10
minimum_data              100
window_size               61249
gap                        1

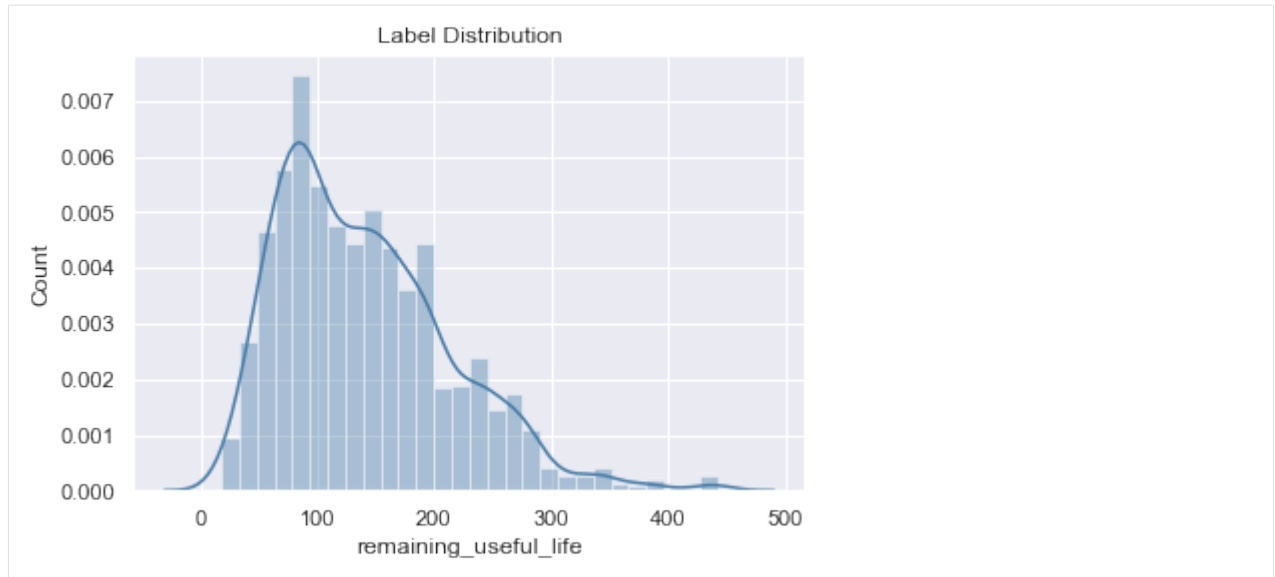
Transforms
-----
No transforms applied
```

Let's plot the labels to get additional insight of the RUL.

### Label Distribution

This plot shows the continuous label distribution.

```
[7]: lt.plot.distribution();
```



## 8.2.5 Discrete Labels

Let's further process the labels into discrete values. We divide the RUL into quartile bins to predict which range an engine's RUL will fall in.

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

### Describe Labels

Next, let's print out the settings and transforms that were used to make the discrete labels. This time we can see the label distribution which is useful for determining if we have imbalanced labels. Also, we can see that the label type changed from continuous to discrete and the binning transform used in the previous step is included below.

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

Label Distribution
-----
(128.0, 187.0]      629
(17.999, 83.0]     625
(83.0, 128.0]      622
(187.0, 442.0]     614
Total:              2490

Settings
-----
label_type                discrete
labeling_function          remaining_useful_life
num_examples_per_instance          10
minimum_data                100
window_size                 61249
gap                           1
```

(continues on next page)



(continued from previous page)

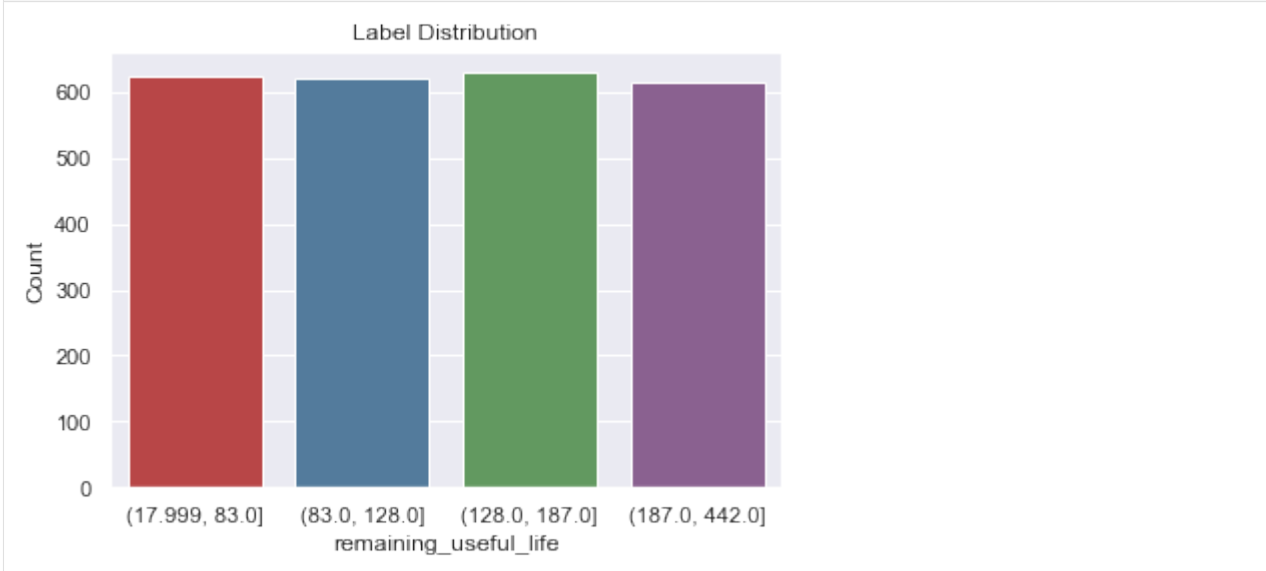
```
Transforms
-----
1. bin
  - bins:          4
  - quantiles:     True
  - labels:        None
  - right:         True
```

Let's plot the labels to get additional insight of the RUL.

## Label Distribution

This plot shows the discrete label distribution.

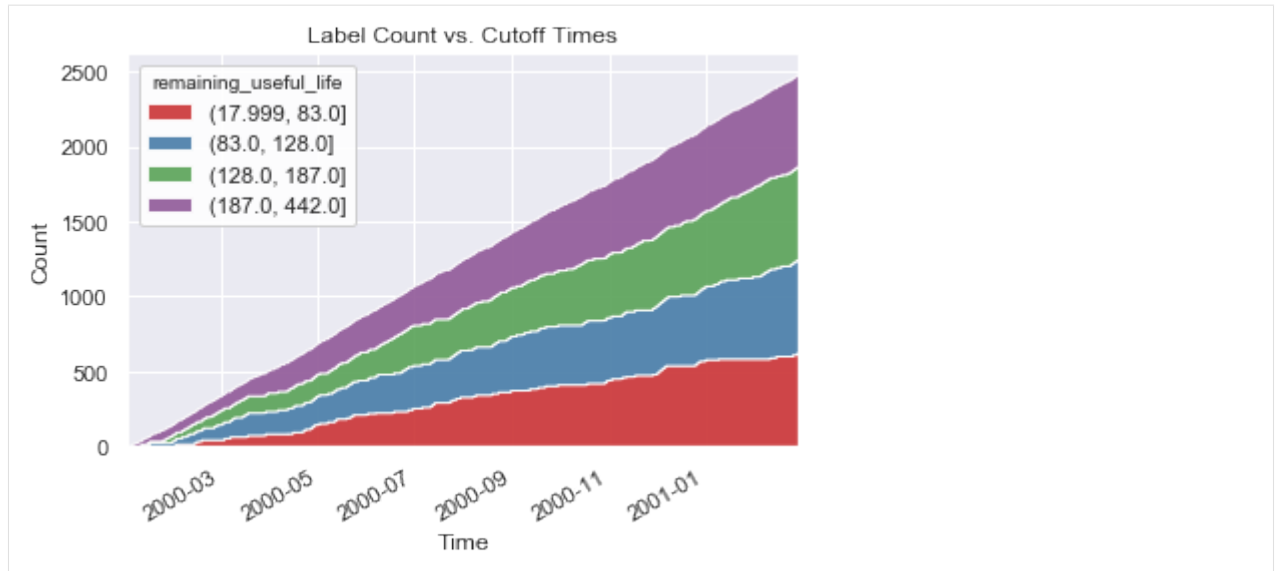
```
[10]: lt.plot.distribution();
```



## Count by Time

This plot shows the label count accumulated across cutoff times.

```
[11]: lt.plot.count_by_time();
```



## 8.3 Generate Features

Now, we are ready to generate features for our prediction problem.

### 8.3.1 Create Entity Set

To get started, let's create an `EntitySet` for the observations.

**See also:**

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

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

es.entity_from_dataframe(
    dataframe=df,
    entity_id='recordings',
    index='id',
    time_index='time',
    make_index=True,
)

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

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

```
[12]: Entityset: observations
      Entities:
        recordings [Rows: 61249, Columns: 28]
        engines [Rows: 249, Columns: 2]
        cycles [Rows: 543, Columns: 2]
      Relationships:
        recordings.engine_no -> engines.engine_no
        recordings.time_in_cycles -> cycles.time_in_cycles
```

### 8.3.2 Describe Entity Set

To get an idea on how the entity set is structured, we can plot a diagram.

```
[13]: es.plot()
```

```
[13]:
```

### 8.3.3 Create Feature Matrix

To simplify the calculation for the feature matrix, we only use 20 percent of the labels.

```
[14]: lt = lt.sample(frac=.2, random_state=0)
```

Let's generate features that correspond to the labels. To do this, we set the `target_entity` to `engines` and the `cutoff_time` to our labels so that the features are calculated for each engine only using data up to and including the cutoff time of each label. Notice that the output of Compose integrates easily with Featuretools.

#### See also:

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

```
[15]: fm, fd = ft.dfs(
      entityset=es,
      target_entity='engines',
      agg_primitives=['last', 'max', 'min'],
      trans_primitives=[],
      cutoff_time=lt,
      cutoff_time_in_index=True,
      max_depth=3,
      verbose=True,
    )
```

```
fm.head()
```

```
Built 292 features
Elapsed: 08:17 | Progress: 100%| Remaining: 00:00
```

```
[15]:
```

engine_no	time	LAST(recordings.id)
1	2000-01-01 16:50:00	101
	2000-01-01 18:10:00	109
2	2000-01-03 22:10:00	421
	2000-01-03 22:50:00	425
	2000-01-03 23:20:00	428

```

      LAST(recordings.time_in_cycles) \
engine_no time
```

(continues on next page)

(continued from previous page)

1	2000-01-01 16:50:00	102
	2000-01-01 18:10:00	110
2	2000-01-03 22:10:00	101
	2000-01-03 22:50:00	105
	2000-01-03 23:20:00	108
LAST(recordings.operational_setting_1) \		
engine_no	time	
1	2000-01-01 16:50:00	42.0057
	2000-01-01 18:10:00	25.0070
2	2000-01-03 22:10:00	34.9989
	2000-01-03 22:50:00	42.0015
	2000-01-03 23:20:00	20.0061
LAST(recordings.operational_setting_2) \		
engine_no	time	
1	2000-01-01 16:50:00	0.8400
	2000-01-01 18:10:00	0.6200
2	2000-01-03 22:10:00	0.8402
	2000-01-03 22:50:00	0.8400
	2000-01-03 23:20:00	0.7015
LAST(recordings.operational_setting_3) \		
engine_no	time	
1	2000-01-01 16:50:00	100.0
	2000-01-01 18:10:00	60.0
2	2000-01-03 22:10:00	100.0
	2000-01-03 22:50:00	100.0
	2000-01-03 23:20:00	100.0
LAST(recordings.sensor_measurement_1) \		
engine_no	time	
1	2000-01-01 16:50:00	445.00
	2000-01-01 18:10:00	462.54
2	2000-01-03 22:10:00	449.44
	2000-01-03 22:50:00	445.00
	2000-01-03 23:20:00	491.19
LAST(recordings.sensor_measurement_2) \		
engine_no	time	
1	2000-01-01 16:50:00	549.11
	2000-01-01 18:10:00	535.85
2	2000-01-03 22:10:00	555.49
	2000-01-03 22:50:00	549.47
	2000-01-03 23:20:00	607.45
LAST(recordings.sensor_measurement_3) \		
engine_no	time	
1	2000-01-01 16:50:00	1341.25
	2000-01-01 18:10:00	1255.17
2	2000-01-03 22:10:00	1359.56
	2000-01-03 22:50:00	1349.80
	2000-01-03 23:20:00	1478.40
LAST(recordings.sensor_measurement_4) \		
engine_no	time	
1	2000-01-01 16:50:00	1120.01

(continues on next page)

(continued from previous page)

	2000-01-01 18:10:00	1044.68
2	2000-01-03 22:10:00	1128.23
	2000-01-03 22:50:00	1113.55
	2000-01-03 23:20:00	1255.95
	LAST(recordings.sensor_measurement_5) ... \	
engine_no	time	...
1	2000-01-01 16:50:00	3.91 ...
	2000-01-01 18:10:00	7.05 ...
2	2000-01-03 22:10:00	5.48 ...
	2000-01-03 22:50:00	3.91 ...
	2000-01-03 23:20:00	9.35 ...
	MIN(recordings.cycles.MIN(recordings.sensor_	
	↪measurement_13)) \	
engine_no	time	
1	2000-01-01 16:50:00	2028.08
	2000-01-01 18:10:00	2028.08
2	2000-01-03 22:10:00	2028.08
	2000-01-03 22:50:00	2028.08
	2000-01-03 23:20:00	2028.08
	MIN(recordings.cycles.MIN(recordings.sensor_	
	↪measurement_14)) \	
engine_no	time	
1	2000-01-01 16:50:00	7860.14
	2000-01-01 18:10:00	7860.14
2	2000-01-03 22:10:00	7860.14
	2000-01-03 22:50:00	7860.14
	2000-01-03 23:20:00	7860.14
	MIN(recordings.cycles.MIN(recordings.sensor_	
	↪measurement_15)) \	
engine_no	time	
1	2000-01-01 16:50:00	8.3885
	2000-01-01 18:10:00	8.3885
2	2000-01-03 22:10:00	8.3646
	2000-01-03 22:50:00	8.3646
	2000-01-03 23:20:00	8.3646
	MIN(recordings.cycles.MIN(recordings.sensor_	
	↪measurement_16)) \	
engine_no	time	
1	2000-01-01 16:50:00	0.02
	2000-01-01 18:10:00	0.02
2	2000-01-03 22:10:00	0.02
	2000-01-03 22:50:00	0.02
	2000-01-03 23:20:00	0.02
	MIN(recordings.cycles.MIN(recordings.sensor_	
	↪measurement_17)) \	
engine_no	time	
1	2000-01-01 16:50:00	303
	2000-01-01 18:10:00	303
2	2000-01-03 22:10:00	303
	2000-01-03 22:50:00	303
	2000-01-03 23:20:00	303

(continues on next page)

(continued from previous page)

```

MIN(recordings.cycles.MIN(recordings.sensor_
↪measurement_18)) \
engine_no time
1      2000-01-01 16:50:00      1915
      2000-01-01 18:10:00      1915
2      2000-01-03 22:10:00      1915
      2000-01-03 22:50:00      1915
      2000-01-03 23:20:00      1915

MIN(recordings.cycles.MIN(recordings.sensor_
↪measurement_19)) \
engine_no time
1      2000-01-01 16:50:00      84.93
      2000-01-01 18:10:00      84.93
2      2000-01-03 22:10:00      84.93
      2000-01-03 22:50:00      84.93
      2000-01-03 23:20:00      84.93

MIN(recordings.cycles.MIN(recordings.sensor_
↪measurement_20)) \
engine_no time
1      2000-01-01 16:50:00      10.36
      2000-01-01 18:10:00      10.36
2      2000-01-03 22:10:00      10.36
      2000-01-03 22:50:00      10.36
      2000-01-03 23:20:00      10.36

MIN(recordings.cycles.MIN(recordings.sensor_
↪measurement_21)) \
engine_no time
1      2000-01-01 16:50:00      6.2607
      2000-01-01 18:10:00      6.2607
2      2000-01-03 22:10:00      6.2534
      2000-01-03 22:50:00      6.2534
      2000-01-03 23:20:00      6.2534

remaining_useful_life
engine_no time
1      2000-01-01 16:50:00      (187.0, 442.0]
      2000-01-01 18:10:00      (187.0, 442.0]
2      2000-01-03 22:10:00      (187.0, 442.0]
      2000-01-03 22:50:00      (187.0, 442.0]
      2000-01-03 23:20:00      (187.0, 442.0]

[5 rows x 293 columns]

```

## 8.4 Machine Learning

Now, we are ready to create a machine learning model for our prediction problem.

### 8.4.1 Preprocess Features

Let's extract the labels from the feature matrix and fill any missing values with zeros. Additionally, the categorical features are one-hot encoded.

```
[16]: y = fm.pop('lt.name')
      y = y.astype('str')

      x = fm.fillna(0)
      x, fe = ft.encode_features(x, fd)
```

### 8.4.2 Split Labels and Features

Then, we split the labels and features each into training and testing sets.

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

### 8.4.3 Train Model

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

```
[18]: clf = RandomForestClassifier(n_estimators=10, random_state=0)
      clf.fit(x_train, y_train)

[18]: RandomForestClassifier(bootstrap=True, class_weight=None, criterion='gini',
      max_depth=None, max_features='auto', max_leaf_nodes=None,
      min_impurity_decrease=0.0, min_impurity_split=None,
      min_samples_leaf=1, min_samples_split=2,
      min_weight_fraction_leaf=0.0, n_estimators=10,
      n_jobs=None, oob_score=False, random_state=0, verbose=0,
      warm_start=False)
```

### 8.4.4 Test Model

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

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

	precision	recall	f1-score	support
(128.0, 187.0]	0.73	0.68	0.70	28
(17.999, 83.0]	0.62	0.80	0.70	25
(187.0, 442.0]	0.88	0.81	0.84	26
(83.0, 128.0]	0.67	0.57	0.62	21
accuracy			0.72	100

(continues on next page)

(continued from previous page)

macro avg	0.72	0.71	0.72	100
weighted avg	0.73	0.72	0.72	100

## 8.4.5 Feature Importances

This plot is based on scores from the model to show which features are important for predictions.

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

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







---

### Frequently Asked Questions

---

#### **9.1 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.

#### **9.2 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.

#### **9.3 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.

## 9.4 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.

## 9.5 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.

## 9.6 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.

## 9.7 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.

## 9.8 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.

## 9.9 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.

## 10.1 Label Maker

---

*LabelMaker*

Automatically makes labels for prediction problems.

---

### 10.1.1 `composeml.LabelMaker`

**class** `composeml.LabelMaker` (*target\_entity*, *time\_index*, *labeling\_function*, *window\_size=None*, *label\_type=None*)  
Automatically makes labels for prediction problems.

#### Methods

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

#### `composeml.LabelMaker.__init__`

`LabelMaker.__init__` (*target\_entity*, *time\_index*, *labeling\_function*, *window\_size=None*, *label\_type=None*)  
Creates an instance of label maker.

#### Parameters

- **target\_entity** (*str*) – Entity on which to make labels.
- **time\_index** (*str*) – Name of time column in the data frame.

- **labeling\_function** (*function*) – Function that transforms a data slice to a label.
- **window\_size** (*str or int*) – Duration of each data slice. The default value for window size is all future data.

### **composeml.LabelMaker.search**

`LabelMaker.search(df, num_examples_per_instance, minimum_data=None, gap=None, drop_empty=True, label_type=None, verbose=True, *args, **kwargs)`  
Searches the data to calculates labels.

#### **Parameters**

- **df** (*DataFrame*) – Data frame to search and extract labels.
- **num\_examples\_per\_instance** (*int*) – Number of examples per unique instance of target entity.
- **minimum\_data** (*str*) – Minimum data before starting search. Default value is first time of index.
- **gap** (*str or int*) – Time between examples. Default value is window size. If an integer, search will start on the first event after the minimum data.
- **drop\_empty** (*bool*) – Whether to drop empty slices. Default value is True.
- **label\_type** (*str*) – The label type can be “continuous” or “categorical”. Default value is the inferred label type.
- **verbose** (*bool*) – Whether to render progress bar. Default value is True.
- **\*args** – Positional arguments for labeling function.
- **\*\*kwargs** – Keyword arguments for labeling function.

**Returns** Calculated labels with cutoff times.

**Return type** *LabelTimes*

### **composeml.LabelMaker.set\_index**

`LabelMaker.set_index(df)`  
Sets the time index in a data frame (if not already set).

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

**Returns** Data frame with time index set.

**Return type** *DataFrame*

### **composeml.LabelMaker.slice**

`LabelMaker.slice(df, num_examples_per_instance, minimum_data=None, gap=None, drop_empty=True, verbose=False)`  
Generates data slices of target entity.

#### **Parameters**

- **df** (*DataFrame*) – Data frame to create slices on.
- **num\_examples\_per\_instance** (*int*) – Number of examples per unique instance of target entity.

- **minimum\_data** (*str*) – Minimum data before starting search. Default value is first time of index.
- **gap** (*str or int*) – Time between examples. Default value is window size. If an integer, search will start on the first event after the minimum data.
- **drop\_empty** (*bool*) – Whether to drop empty slices. Default value is True.
- **verbose** (*bool*) – Whether to print metadata about slice. Default value is False.

**Returns** Returns data slice.

**Return type** DataSlice

## 10.2 Label Times

---

*LabelTimes*

A data frame containing labels made by a label maker.

---

### 10.2.1 composeml.LabelTimes

**class** `composeml.LabelTimes` (*data=None, target\_entity=None, name=None, label\_type=None, settings=None, \*args, \*\*kwargs*)

A data frame containing labels made by a label maker.

**settings**

**Methods**

<code>__init__</code>	Initialize self.
<code>apply_lead</code>	Shifts the label times earlier for predicting in advance.
<code>bin</code>	Bin labels into discrete intervals.
<code>copy</code>	Makes a copy of this object.
<code>describe</code>	Prints out label info with transform settings that reproduce labels.
<code>equals</code>	Determines if two label time objects are the same.
<code>infer_type</code>	Infer label type.
<code>sample</code>	Return a random sample of labels.
<code>threshold</code>	Creates binary labels by testing if labels are above threshold.
<code>to_csv</code>	Write label times in csv format to disk.
<code>to_parquet</code>	Write label times in parquet format to disk.
<code>to_pickle</code>	Write label times in pickle format to disk.

**composeml.LabelTimes.\_\_init\_\_**

`LabelTimes.__init__` (*data=None, target\_entity=None, name=None, label\_type=None, settings=None, \*args, \*\*kwargs*)

Initialize self. See `help(type(self))` for accurate signature.

### composeml.LabelTimes.apply\_lead

LabelTimes.**apply\_lead**(*value*, *inplace=False*)

Shifts the label times earlier for predicting in advance.

#### Parameters

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

**Returns** Instance of labels.

**Return type** labels (*LabelTimes*)

### composeml.LabelTimes.bin

LabelTimes.**bin**(*bins*, *quantiles=False*, *labels=None*, *right=True*)

Bin labels into discrete intervals.

#### Parameters

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

**Returns** Instance of labels.

**Return type** *LabelTimes*

### Examples

Using bins of *equal-widths*:

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

Using bins of *custom-widths*:



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

Using *quantile-based* bins:

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

Assigning *labels* to bins:

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

### `composeml.LabelTimes.copy`

`LabelTimes.copy()`

Makes a copy of this object.

**Returns** Copy of label times.

**Return type** *LabelTimes*

### `composeml.LabelTimes.describe`

`LabelTimes.describe()`

Prints out label info with transform settings that reproduce labels.

### `composeml.LabelTimes.equals`

`LabelTimes.equals(other)`

Determines if two label time objects are the same.

**Parameters** **other** (*LabelTimes*) – Other label time object for comparison.

**Returns** Whether label time objects are the same.

**Return type** `bool`

### `composeml.LabelTimes.infer_type`

`LabelTimes.infer_type()`

Infer label type.

**Returns** Inferred label type. Either “continuous” or “discrete”.

**Return type** `str`

### `composeml.LabelTimes.sample`

`LabelTimes.sample` (*n=None, frac=None, random\_state=None, replace=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`.

**Returns** Random sample of labels.

**Return type** `LabelTimes`

### Examples

Create mock data:

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

Sample number of labels:

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

Sample number per label:

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

Sample fraction of labels:

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

Sample fraction per label:

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

### 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*, *filename='label\_times.csv'*, *save\_settings=True*)

Write label times in csv format to disk.

#### Parameters

- **path** (*str*) – Location on disk to write to (will be created as a directory).
- **filename** (*str*) – Filename for label times. Default value is *label\_times.csv*.
- **save\_settings** (*bool*) – Whether to save the settings used to make the label times.

### composeml.LabelTimes.to\_parquet

`LabelTimes.to_parquet` (*path*, *filename='label\_times.parquet'*, *save\_settings=True*)

Write label times in parquet format to disk.

#### Parameters

- **path** (*str*) – Location on disk to write to (will be created as a directory).
- **filename** (*str*) – Filename for label times. Default value is *label\_times.parquet*.
- **save\_settings** (*bool*) – Whether to save the settings used to make the label times.

### composeml.LabelTimes.to\_pickle

`LabelTimes.to_pickle` (*path*, *filename*='label\_times.pickle', *save\_settings*=True)  
 Write label times in pickle format to disk.

#### Parameters

- **path** (*str*) – Location on disk to write to (will be created as a directory).
- **filename** (*str*) – Filename for label times. Default value is *label\_times.pickle*.
- **save\_settings** (*bool*) – Whether to save the settings used to make the label times.

## 10.2.2 Transform Methods

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

## 10.3 Label Plots

<code>LabelPlots</code>	Creates plots for Label Times.
-------------------------	--------------------------------

### 10.3.1 composeml.label\_plots.LabelPlots

**class** `composeml.label_plots.LabelPlots` (*label\_times*)  
 Creates plots for Label Times.

#### Methods

<code>__init__</code>	Initializes Label Plots.
<code>count_by_time</code>	Plots the label distribution across cutoff times.
<code>distribution</code>	Plots the label distribution.

#### `composeml.label_plots.LabelPlots.__init__`

`LabelPlots.__init__` (*label\_times*)  
 Initializes Label Plots.

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

#### `composeml.label_plots.LabelPlots.count_by_time`

`LabelPlots.count_by_time` (*ax*=None, *\*\*kwargs*)  
 Plots the label distribution across cutoff times.

**composeml.label\_plots.LabelPlots.distribution**

`LabelPlots.distribution(**kwargs)`  
Plots the label distribution.

**10.3.2 Plotting Methods**

---

<code>LabelPlots.count_by_time</code>	Plots the label distribution across cutoff times.
<code>LabelPlots.distribution</code>	Plots the label distribution.

---



### **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



---

## Symbols

`__init__()` (*composeml.LabelMaker* method), 57

`__init__()` (*composeml.LabelTimes* method), 59

`__init__()` (*composeml.label\_plots.LabelPlots* method), 64

## A

`apply_lead()` (*composeml.LabelTimes* method), 60

## B

`bin()` (*composeml.LabelTimes* method), 60

## C

`copy()` (*composeml.LabelTimes* method), 61

`count_by_time()` (*composeml.label\_plots.LabelPlots* method), 64

## D

`describe()` (*composeml.LabelTimes* method), 61

`distribution()` (*composeml.label\_plots.LabelPlots* method), 65

## E

`equals()` (*composeml.LabelTimes* method), 61

## I

`infer_type()` (*composeml.LabelTimes* method), 61

## L

`LabelMaker` (class in *composeml*), 57

`LabelPlots` (class in *composeml.label\_plots*), 64

`LabelTimes` (class in *composeml*), 59

## S

`sample()` (*composeml.LabelTimes* method), 62

`search()` (*composeml.LabelMaker* method), 58

`set_index()` (*composeml.LabelMaker* method), 58

`settings` (*composeml.LabelTimes* attribute), 59

`slice()` (*composeml.LabelMaker* method), 58

## T

`threshold()` (*composeml.LabelTimes* method), 63

`to_csv()` (*composeml.LabelTimes* method), 63

`to_parquet()` (*composeml.LabelTimes* method), 63

`to_pickle()` (*composeml.LabelTimes* method), 64