

# Lifetimes: Measuring Customer Lifetime Value in Python

---

[dataorigami.net/blogs/napkin-folding/18868411-lifetimes-measuring-customer-lifetime-value-in-python](https://dataorigami.net/blogs/napkin-folding/18868411-lifetimes-measuring-customer-lifetime-value-in-python)

Lifetimes is my latest Python project. Below is a summary, but you can also check out the [source code on Github](#).

## Introduction

---

As emphasized by P. Fader and B. Hardie, understanding and acting on customer lifetime value (CLV) is the most important part of your business's sales efforts. And (apparently) everyone is doing it wrong. *Lifetimes* is a Python library to calculate CLV for you.

More generally, Lifetimes can be used to understand and predict future usage based on a few lax assumption:

1. Entities under study may die after some random period of time.
2. Entities interact with you when they are alive.

Lifetimes can be used to both estimate if these entities are still *alive*, and predict how much more they will interact based on their existing history. If this is too abstract, consider these situations:

- Predicting how often a visitor will return to your website.
- Understanding how frequently a patient may return to a hospital.
- Predicting individuals who gave "died" using only their usage history.

Really, "customers" is a very general term here, (similar to "birth" and "death" in survival analysis). Whenever we have individuals repeating occurrences, we can use Lifetimes to help understand behaviour.

## Installation

---

```
pip install lifetimes
```

Requirements are only Numpy, Scipy, Pandas.

## Quickstart

---

The examples below are using the `cdnow_customers.csv` located in the `datasets/` directory.

```

from lifetimes.datasets import load_cdnow
data = load_cdnow(index_col=[0])
data.head()
"""
      x    t_x    T
ID
1    2  30.43  38.86
2    1   1.71  38.86
3    0   0.00  38.86
4    0   0.00  38.86
5    0   0.00  38.86
"""

```

`x` represents the number of repeat purchases the customer has made (also called **frequency**). `T` represents the age of the customer. `t_x` represents the age of the customer when they made their most recent purchases (also called **recency**).

## Fitting models to our data

---

We'll use the **BG/NBD model** first. Interested in the model? See this [nice paper here](#).

```

from lifetimes import BetaGeoFitter

# similar API to scikit-learn and lifelines.
bgf = BetaGeoFitter()
bgf.fit(data['x'], data['t_x'], data['T'])
print bgf
"""
<lifetimes.BetaGeoFitter: fitted with 2357 customers, a: 0.79, alpha: 4.41, r: 0.24,
b: 2.43>
"""

```

After fitting, we have lots of nice methods and properties attached to the fitter object.

## Visualizing our Frequency/Recency Matrix

---

Consider: a customer bought from you every day for three weeks straight, and we haven't heard from them in months. What are the chances they are still "alive"? Pretty small. On the other hand, a customer who historically buys from you once a quarter, and bought last quarter, is likely still alive. We can visualize this relationship using the **Frequency/Recency matrix**, which computes the expected number of transactions a artificial customer is to make in the next time period, given his or her recency (age at last purchase) and frequency (the number of repeat transactions he or she has made).

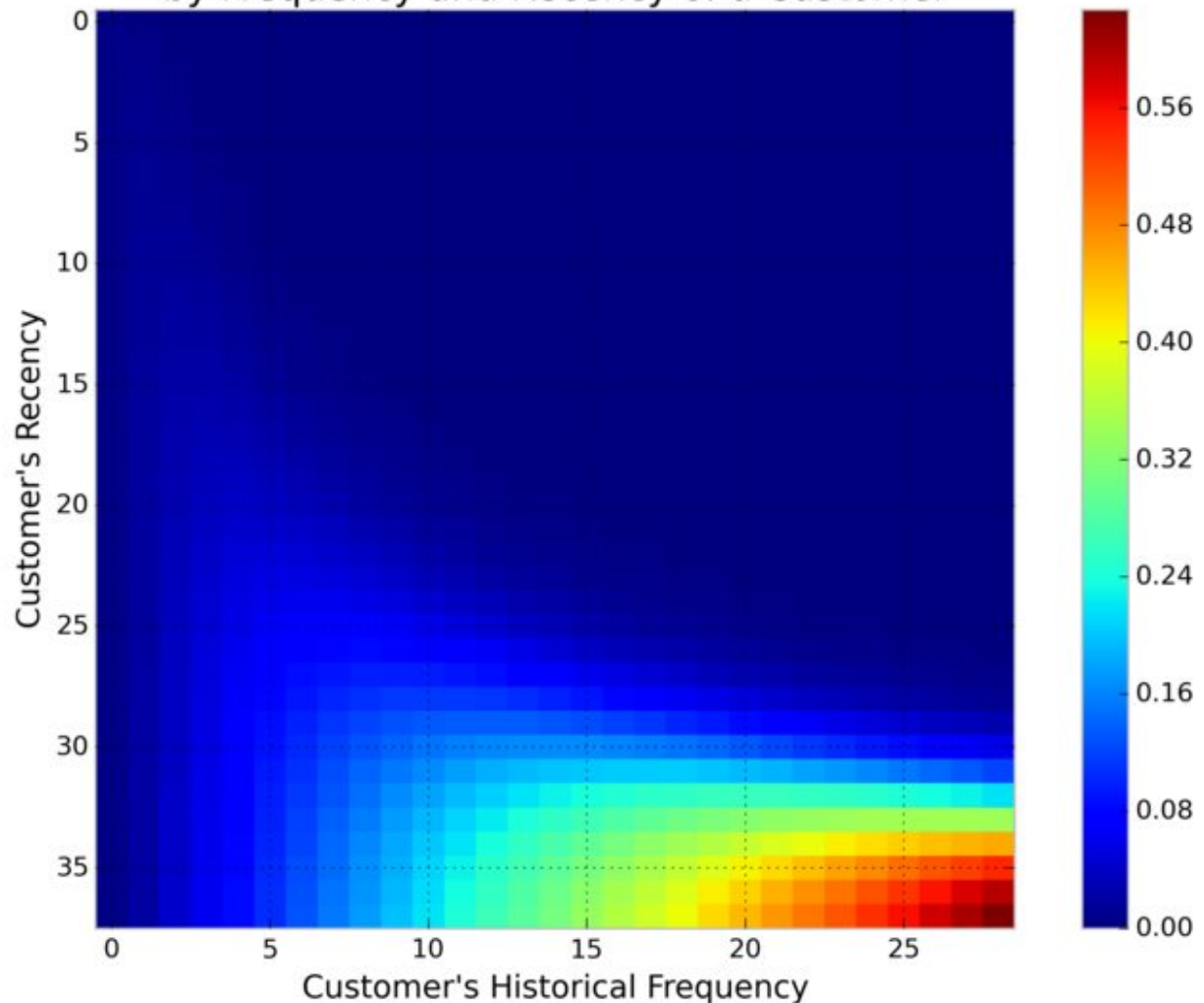
```

from lifetimes.plotting import plot_frequency_recency_matrix

plot_frequency_recency_matrix(bgf)

```

Expected Number of Future Purchases over 1 Unit of Time,  
by Frequency and Recency of a Customer



We can see that if a customer has bought 25 times from you, and their lastest purchase was when they were 35 weeks old (given the individual is 35 weeks old), then they are you best customer (bottom-right). You coldest customers are those that in the top-right corner: they bought a lot quickly, and we haven't seen them in weeks.

There's also that beautiful "tail" around (5,25). That represents the customer who buys infrequently, but we've seen him or her recently, so they *might* buy again - we're not sure if they are dead or just between purchases.

### Ranking customers from best to worst

Let's return to our customers and rank them from "highest expected purchases in the next period" to lowest. Models expose a method that will predict a customer's expected purchases in the next period using their history.

```

t = 1
data['predicted_purchases'] = data.apply(lambda r:
bgf.conditional_expected_number_of_purchases_up_to_time(t, r['x'], r['t_x'],
r['T']), axis=1)
data.sort('predicted_purchases').tail(5)
"""
      x    t_x    T predicted_purchases
ID
509  18  35.14  35.86             0.424877
841  19  34.00  34.14             0.474738
1981 17  28.43  28.86             0.486526
157   29  37.71  38.00             0.662396
1516 26  30.86  31.00             0.710623
"""

```

Great, we can see that the customer who has made 26 purchases, and bought very recently from us, is probably going to buy again in the next period.

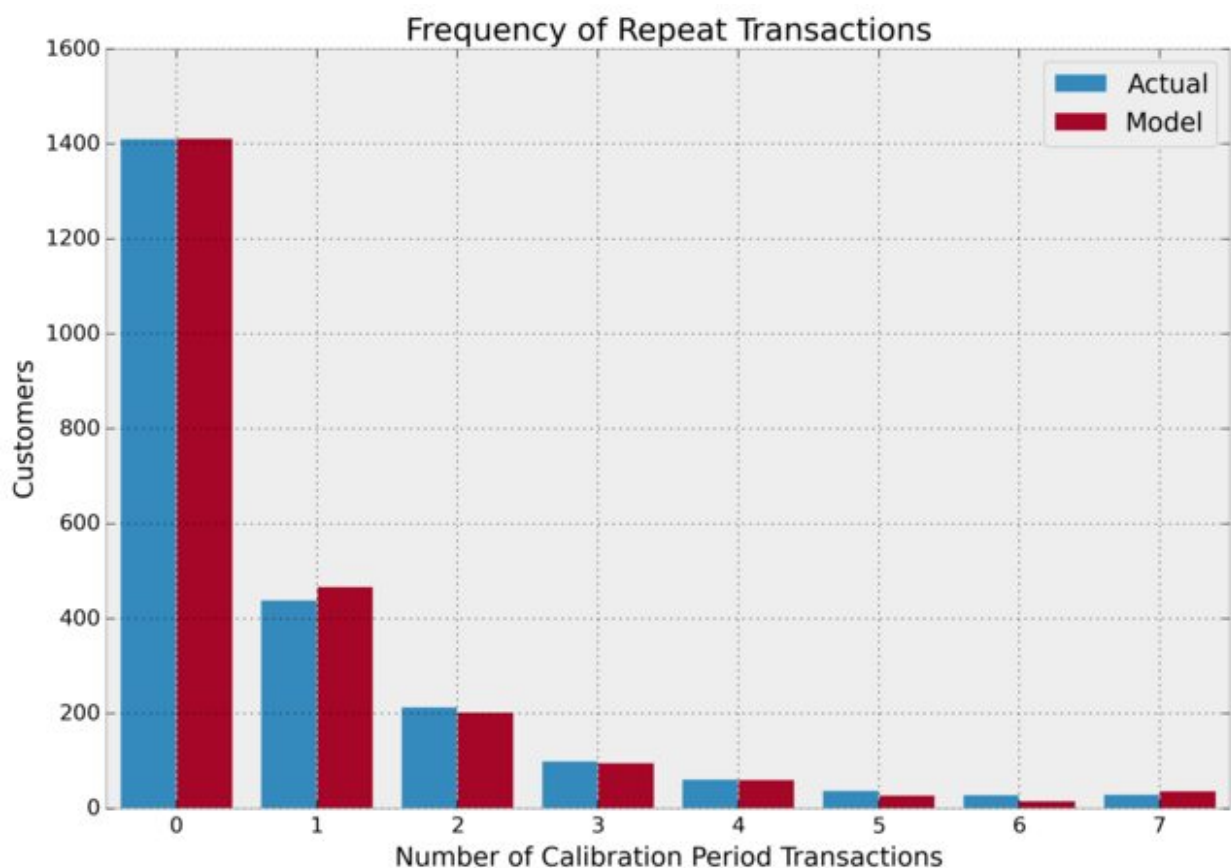
## Assessing model fit

Ok, we can predict and we can visualize our customers' behaviour, but is our model correct? There are a few ways to assess the model's correctness. The first is to compare your data versus artificial data simulated with your fitted model's parameters.

```

from lifetimes.plotting import plot_period_transactions
plot_period_transactions(bgf)

```



We can see that our actual data and our simulated data line up well. This proves that our model doesn't suck.

## Example using transactional datasets

Most often, the dataset you have at hand will be at the transaction level. Lifetimes has some utility functions to transform that transactional data (one row per purchase) into summary data (a frequency, recency and age dataset).

```
from lifetimes.datasets import load_transaction_data
from lifetimes.utils import summary_data_from_transaction_data

transaction_data = load_transaction_data()
transaction_data.head()
"""
           date  id
0  2014-03-08 00:00:00  0
1  2014-05-21 00:00:00  1
2  2014-03-14 00:00:00  2
3  2014-04-09 00:00:00  2
4  2014-05-21 00:00:00  2
"""

summary = summary_data_from_transaction_data(transaction_data, 'id', 'date',
observation_period_end='2014-12-31')

print summary.head()
"""
      frequency  recency    T
id
0             0         0  298
1             0         0  224
2             6        142  292
3             0         0  147
4             2          9  183
"""

bgf.fit(summary['frequency'], summary['recency'], summary['T'])
# <lifetimes.BetaGeoFitter: fitted with 5000 customers, a: 1.85, alpha: 1.86, r:
0.16, b: 3.18>
```

## More model fitting

---

With transactional data, we can partition the dataset into a calibration period dataset and a holdout dataset. This is important as we want to test how our model performs on data not yet seen (think cross-validation in standard machine learning literature). Lifetimes has a function to partition our dataset like this:

```

from lifetimes.utils import calibration_and_holdout_data

summary_cal_holdout = calibration_and_holdout_data(transaction_data, 'id', 'date',
                                                    calibration_period_end='2014-09-01',
                                                    observation_period_end='2014-12-31' )

print summary_cal_holdout.head()
"""
    frequency_cal  recency_cal  T_cal  frequency_holdout  duration_holdout
id
0                0          0    177                0        121
1                0          0    103                0        121
2                6        142    171                0        121
3                0          0     26                0        121
4                2          9     62                0        121
"""

```

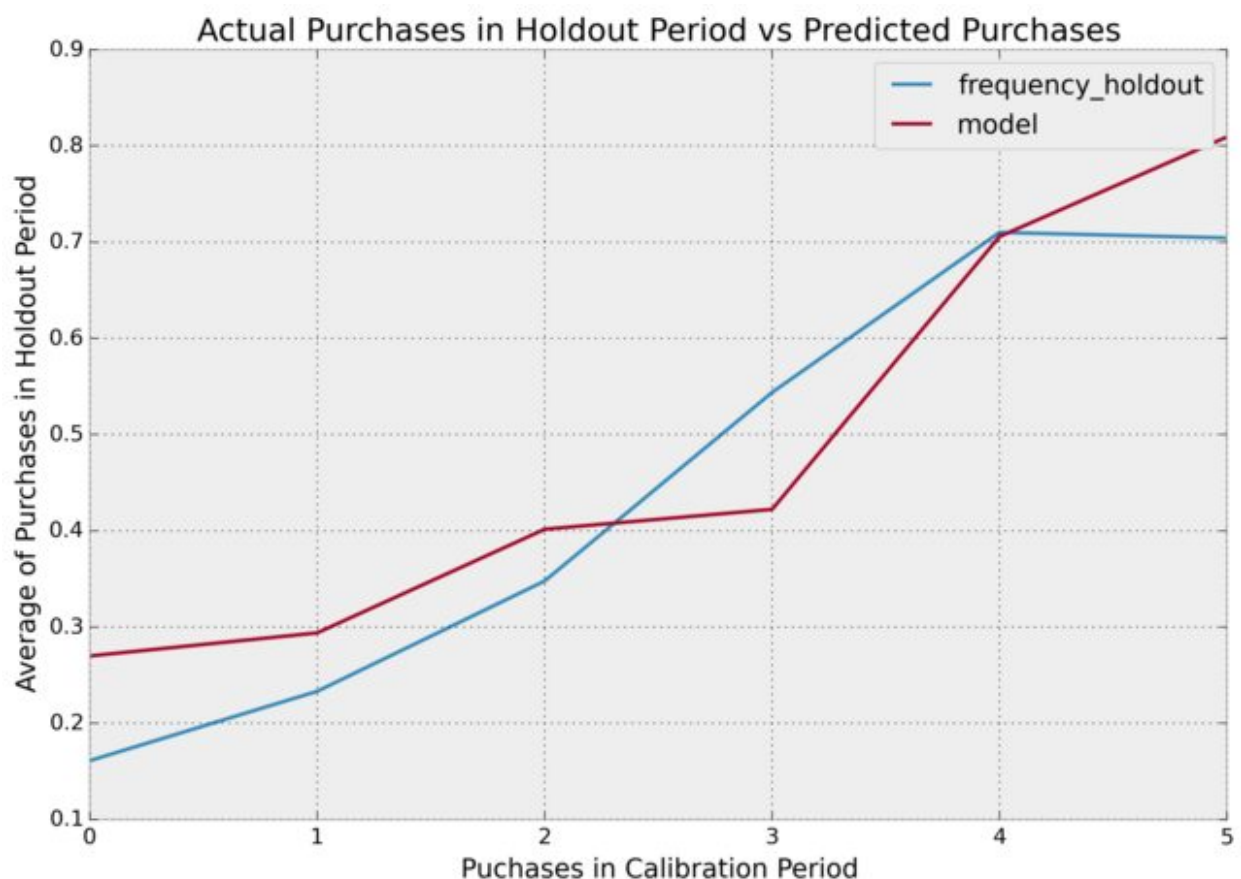
With this dataset, we can perform fitting on the `_cal` columns, and test on the `_holdout` columns:

```

from lifetimes.plotting import plot_calibration_purchases_vs_holdout_purchases

bgf.fit(summary_cal_holdout['frequency_cal'], summary_cal_holdout['recency_cal'],
summary_cal_holdout['T_cal'])
plot_calibration_purchases_vs_holdout_purchases(bgf, summary_cal_holdout)

```



## Customer Predictions

Basic on customer history, we can predict what an individuals future purchases might look like:

```
t = 10 #predict purchases in 10 periods
individual = summary.iloc[20]
# The below function may be renamed to `predict` in a future version of lifetimes
bgf.conditional_expected_number_of_purchases_up_to_time(t, individual['frequency'],
individual['recency'], individual['T'])
# 0.0576511
```

Drop me a line at [@cmrn\\_dp!](#)

## More Information

---

### Related Posts

---

- [How Shopify Merchants Can Measure Retention \[x-post from Shopify Engineering Blog\]](#)  
How Shopify Merchants Can Measure Retention

