lifetimes Documentation

Release 0.8.0.0

Cameron Davidson-Pilon

Aug 21, 2017

Contents:

1 Introduction **3**
- 1.1 Applications . . . 3
- 1.2 Specific Application: Customer Lifetime Value . . . 3

2 Installation **5**

3 Documenation and tutorials **7**

4 Questions? Comments? Requests? **9**

5 More Information **11**
- 5.1 Quickstart . . . 11
- 5.2 Saving and loading model . . . 17
- 5.3 More Examples and recipes . . . 18

6 Indices and tables **21**

CHAPTER 1

Introduction

Lifetimes can be used to analyze your users based on a few assumption:

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

I've quoted "alive" and "die" as these are the most abstract terms: feel free to use your own definition of "alive" and "die" (they are used similarly to "birth" and "death" in survival analysis). Whenever we have individuals repeating occurrences, we can use Lifetimes to help understand user behaviour.

Applications

If this is too abstract, consider these applications:

- Predicting how often a visitor will return to your website. (Alive = visiting. Die = decided the website wasn't for them)
- Understanding how frequently a patient may return to a hospital. (Alive = visiting. Die = maybe the patient moved to a new city, or became deceased.)
- Predicting individuals who have churned from an app using only their usage history. (Alive = logins. Die = removed the app)
- Predicting repeat purchases from a customer. (Alive = actively purchasing. Die = became disinterested with your product)
- Predicting the lifetime values of your customers

Specific Application: Customer Lifetime Value

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.

CHAPTER 2

Installation

```
pip install lifetimes
```

Requirements are only Numpy, Scipy, Pandas, Dill (and optionally-but-seriously matplotlib).

CHAPTER 3

Documenation and tutorials

Official documentation

CHAPTER 4

Questions? Comments? Requests?

Please create an issue in the lifetimes repository.

CHAPTER 5

More Information

1. Roberto Medri did a nice presentation on CLV at Etsy.
2. Papers, lots of papers.
3. R implementation is called BTYD (for, *Buy 'Til You Die*).

Quickstart

For the following examples, we'll use a dataset from an ecommerce provider to analyze their customers' repeat purchases. The examples below are using the `cdnow_customers.csv` located in the `datasets/` directory.

```
from lifetimes.datasets import load_cdnow_summary
data = load_cdnow_summary(index_col=[0])

print(data.head())
"""
     frequency   recency      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
"""
```

The shape of your data

For all models, the following nomenclature is used:

- `frequency` represents the number of *repeat* purchases the customer has made. This means that it's one less than the total number of purchases. This is actually slightly wrong. It's the count of time periods the customer had a purchase in. So if using days as units, then it's the count of days the customer had a purchase on.

- `T` represents the age of the customer in whatever time units chosen (weekly, in the above dataset). This is equal to the duration between a customer's first purchase and the end of the period under study.
- `recency` represents the age of the customer when they made their most recent purchases. This is equal to the duration between a customer's first purchase and their latest purchase. (Thus if they have made only 1 purchase, the recency is 0.)

If your data is not in the format (very common), there are *utility functions* in lifetimes to transform your data to look like this

Basic Frequency/Recency analysis using the BG/NBD model

We'll use the **BG/NBD model** first. There are other models which we will explore in these docs, but this is the simplest to start with.

```
from lifetimes import BetaGeoFitter

# similar API to scikit-learn and lifelines.
bgf = BetaGeoFitter(penalizer_coef=0.0)
bgf.fit(data['frequency'], data['recency'], data['T'])
print(bgf)
"""
<lifetimes.BetaGeoFitter: fitted with 2357 subjects, a: 0.79, alpha: 4.41, b: 2.43,␣
↪r: 0.24>
"""
```

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

For small samples sizes, the parameters can get implausibly large, so by adding an l2 penalty the likelihood, we can control how large these parameters can be. This is implemented as setting as positive `penalizer_coef` in the initialization of the model. In typical applications, penalizers on the order of 0.001 to 0.1 are effective.

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)
```

We can see that if a customer has bought 25 times from you, and their latest purchase was when they were 35 weeks old (given the individual is 35 weeks old), then they are your best customer (bottom-right). Your coldest customers are those that are 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.

Another interesting matrix to look at is the probability of still being *alive*:

```
from lifetimes.plotting import plot_probability_alive_matrix

plot_probability_alive_matrix(bgf)
```

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'] = bgf.conditional_expected_number_of_purchases_up_to_
↪time(t, data['frequency'], data['recency'], data['T'])
data.sort_values(by='predicted_purchases').tail(5)
"""
       frequency  recency      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()
print(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      0.0  298.0
1          0.0      0.0  224.0
2          6.0    142.0  292.0
3          0.0      0.0  147.0
4          2.0      9.0  183.0
"""

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

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          0.0  177.0                0.0               121
1             0.0          0.0  103.0                0.0               121
2             6.0        142.0  171.0                0.0               121
3             0.0          0.0   26.0                0.0               121
4             2.0          9.0   62.0                0.0               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

Based 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 is an alias to `bfg.conditional_expected_number_of_purchases_up_
↪to_time`
```

```
bgf.predict(t, individual['frequency'], individual['recency'], individual['T'])
# 0.0576511
```

Customer Probability Histories

Given a customer transaction history, we can calculate their historical probability of being alive, according to our trained model. For example:

```
from lifetimes.plotting import plot_history_alive

id = 35
days_since_birth = 200
sp_trans = transaction_data.loc[transaction_data['id'] == id]
plot_history_alive(bgf, days_since_birth, sp_trans, 'date')
```

Estimating customer lifetime value using the Gamma-Gamma model

For this whole time we didn't take into account the economic value of each transaction and we focused mainly on transactions' occurrences. To estimate this we can use the Gamma-Gamma submodel. But first we need to create summary data from transactional data also containing economic values for each transaction (i.e. profits or revenues).

```
from lifetimes.datasets import load_cdnow_summary_data_with_monetary_value

summary_with_money_value = load_cdnow_summary_data_with_monetary_value()
summary_with_money_value.head()
returning_customers_summary = summary_with_money_value[summary_with_money_value[
↪'frequency']>0]

print(returning_customers_summary.head())
"""
             frequency  recency      T  monetary_value
customer_id
1                    2    30.43  38.86           22.35
2                    1     1.71  38.86           11.77
6                    7    29.43  38.86           73.74
7                    1     5.00  38.86           11.77
9                    2    35.71  38.86           25.55
"""
```

The Gamma-Gamma model and the independence assumption

The model we are going to use to estimate the CLV for our userbase is called the Gamma-Gamma submodel, which relies upon an important assumption. The Gamma-Gamma submodel, in fact, assumes that there is no relationship between the monetary value and the purchase frequency. In practice we need to check whether the Pearson correlation between the two vectors is close to 0 in order to use this model.

```
returning_customers_summary[['monetary_value', 'frequency']].corr()
"""
                monetary_value  frequency
monetary_value        1.000000   0.113884
frequency             0.113884   1.000000
"""
```

At this point we can train our Gamma-Gamma submodel and predict the conditional, expected average lifetime value of our customers.

```
from lifetimes import GammaGammaFitter

ggf = GammaGammaFitter(penalizer_coef = 0)
ggf.fit(returning_customers_summary['frequency'],
        returning_customers_summary['monetary_value'])
print(ggf)
"""
<lifetimes.GammaGammaFitter: fitted with 946 subjects, p: 6.25, q: 3.74, v: 15.45>
"""
```

We can now estimate the average transaction value:

```
print(ggf.conditional_expected_average_profit(
        summary_with_money_value['frequency'],
        summary_with_money_value['monetary_value']
    ).head(10))
"""
customer_id
1     24.658619
2     18.911489
3     35.170981
4     35.170981
5     35.170981
6     71.462843
7     18.911489
8     35.170981
9     27.282408
10    35.170981
dtype: float64
"""

print("Expected conditional average profit: %s, Average profit: %s" % (
    ggf.conditional_expected_average_profit(
        summary_with_money_value['frequency'],
        summary_with_money_value['monetary_value']
    ).mean(),
    summary_with_money_value[summary_with_money_value['frequency']>0]['monetary_value
↪'].mean()
))
"""
Expected conditional average profit: 35.2529588256, Average profit: 35.078551797
"""
```

While for computing the total CLV using the DCF method (https://en.wikipedia.org/wiki/Discounted_cash_flow) adjusting for cost of capital:

```
# refit the BG model to the summary_with_money_value dataset
bgf.fit(summary_with_money_value['frequency'], summary_with_money_value['recency'],␣
↪summary_with_money_value['T'])

print(ggf.customer_lifetime_value(
    bgf, #the model to use to predict the number of future transactions
    summary_with_money_value['frequency'],
    summary_with_money_value['recency'],
    summary_with_money_value['T'],
```

```
    summary_with_money_value['monetary_value'],
    time=12, # months
    discount_rate=0.01 # monthly discount rate ~ 12.7% annually
).head(10))
"""
customer_id
1       140.096211
2        18.943467
3        38.180574
4        38.180574
5        38.180574
6      1003.868107
7        28.109683
8        38.180574
9       167.418216
10       38.180574
Name: clv, dtype: float64
"""
```

Saving and loading model

When you have lots of data and training takes a lot of time option with saving and loading model could be useful. First you need to fit the model, then save it and load.

Fit model

```
from lifetimes import BetaGeoFitter
from lifetimes.datasets import load_cdnow_summary

data = load_cdnow_summary(index_col=[0])
bgf = BetaGeoFitter()
bgf.fit(data['frequency'], data['recency'], data['T'])
bgf
"""<lifetimes.BetaGeoFitter: fitted with 2357 subjects, a: 0.79, alpha: 4.41, b: 2.43,
↪ r: 0.24>"""
```

Saving model

Model will be saved with dill to pickle object. Optional parameters `save_data` and `save_generate_data_method` are present to reduce final pickle object size for big dataframes. Optional parameters:

- `save_data` is used for saving data from model or not (default: `True`).
- `save_generate_data_method` is used for saving `generate_new_data` method from model or not (default: `True`)

```
bgf.save_model('bgf.pkl')
```

or to save only model with minumum size without `data` and `generate_new_data`:

```
bgf.save_model('bgf_small_size.pkl', save_data=False, save_generate_data_method=False)
```

Loading model

Before loading you should initialize the model first and then use method `load_model`

```
bgf_loaded = BetaGeoFitter()
bgf_loaded.load_model('bgf.pkl')
bgf_loaded
"""<lifetimes.BetaGeoFitter: fitted with 2357 subjects, a: 0.79, alpha: 4.41, b: 2.43,
↪ r: 0.24>"""
```

More Examples and recipes

Example SQL statement to transform transactional data into RFM data

Let's review what our variables mean:

- `frequency` represents the number of *repeat* purchases the customer has made. This means that it's one less than the total number of purchases. This is actually slightly wrong. It's the count of distinct time periods the customer had a purchase in. So if using days as units, then it's the count of distinct days the customer had a purchase on.
- `T` represents the age of the customer in whatever time units chosen. This is equal to the duration between a customer's first purchase and the end of the period under study.
- `recency` represents the age of the customer when they made their most recent purchases. This is equal to the duration between a customer's first purchase and their latest purchase. (Thus if they have made only 1 purchase, the recency is 0.)

Thus, executing a query against a transactional dataset, called `orders`, in a SQL-store may look like:

```
SELECT
  customer_id,
  COUNT(distinct date(transaction_at)) - 1 as frequency,
  datediff('day', MIN(transaction_at), MAX(transaction_at)) as recency,
  datediff('day', CURRENT_DATE, MIN(transaction_at)) as T
FROM orders
GROUP BY customer_id
```

Create table with RFM summary matrix with holdout

Variables `frequency`, `T` and `recency` have the same meaning as in previous section.

Two variables to set before executing:

- `duration_holdout` - holdout duration in days.
- `CURRENT_DATE` - current date, could be changed to final date of the transactional data.

```
select
    a.*,
    COALESCE(b.frequency_holdout, 0) as frequency_holdout,
```

```
    duration_holdout as duration_holdout
from (
    select
        customer_id,
        datediff(max(event_date), min(event_date)) as recency,
        count(*) - 1 as frequency,
        datediff(date_sub(CURRENT_DATE, duration_holdout), min(event_date)) as T
    from orders
    where event_date < date_sub(CURRENT_DATE, duration_holdout)
    group by customer_id
) a
left join (
    select
        customer_id,
        count(*) as frequency_holdout
    from orders
    where event_date >= date_sub(CURRENT_DATE, duration_holdout)
      and event_date < CURRENT_DATE
    group by customer_id
) b
on a.customer_id = b.customer_id
```

CHAPTER 6

Indices and tables

- genindex
- modindex
- search