Quickstart Intro

1 Contents: 3
2 Indices and tables 267
Python Module Index 269
lifelines is an implementation of survival analysis in Python. What benefits does lifelines offer over other survival analysis implementations?

- built on top of Pandas
- internal plotting methods
- simple and intuitive API
- only focus is survival analysis
- handles right, left and interval censored data
1.1 Quickstart

1.1.1 Installation

Install via pip:

```
pip install lifelines
```

1.1.2 Kaplan-Meier Nelson-Aalen, and parametric models

Note: For readers looking for an introduction to survival analysis, it’s recommended to start at Introduction to survival analysis

Let’s start by importing some data. We need the durations that individuals are observed for, and whether they “died” or not.

```python
from lifelines.datasets import load_waltons
df = load_waltons() # returns a Pandas DataFrame

print(df.head())
```

(continues on next page)
T = df['T']
E = df['E']

T is an array of durations, E is a either boolean or binary array representing whether the “death” was observed or not (alternatively an individual can be censored).

**Note:** *lifelines* assumes all “deaths” are observed unless otherwise specified.

```python
from lifelines import KaplanMeierFitter
kmf = KaplanMeierFitter()
kmf.fit(T, event_observed=E)  # or, more succinctly, kmf.fit(T, E)
```

After calling the `fit` method, we have access to new properties like `survival_function_` and methods like `plot()`. The latter is a wrapper around Panda’s internal plotting library.

```python
kmf.survival_function_
kmf.cumulative_density_
kmf.median_
kmf.plot_survival_function()  # or just kmf.plot()
```
Alternatively, you can plot the cumulative density function:

```python
kmf.plot_cumulative_density()
```
By specifying the `timeline` keyword argument in `fit`, we can change how the above models are indexed:

```python
kmf.fit(T, E, timeline=range(0, 100, 2))
kmf.survival_function_  # index is now the same as range(0, 100, 2)
kmf.confidence_interval_ # index is now the same as range(0, 100, 2)
```

Instead of the Kaplan-Meier estimator, you may be interested in a parametric model. `lifelines` has builtin parametric models. For example, Weibull, Log-Normal, Log-Logistic, and more.

```python
from lifelines import *
fig, axes = plt.subplots(2, 3, figsize=(9, 5))
kmf = KaplanMeierFitter().fit(T, E, label='KaplanMeierFitter')
wbf = WeibullFitter().fit(T, E, label='WeibullFitter')
exf = ExponentialFitter().fit(T, E, label='ExponentialFitter')
lnf = LogNormalFitter().fit(T, E, label='LogNormalFitter')
llf = LogLogisticFitter().fit(T, E, label='LogLogisticFitter')
pwf = PiecewiseExponentialFitter([40, 60]).fit(T, E, label='PiecewiseExponentialFitter')
```

(continues on next page)
wbf.plot_survival_function(ax=axes[0][0])
exf.plot_survival_function(ax=axes[0][1])
lnf.plot_survival_function(ax=axes[0][2])
kmf.plot_survival_function(ax=axes[1][0])
llf.plot_survival_function(ax=axes[1][1])
pwf.plot_survival_function(ax=axes[1][2])
Chapter 1. Contents:
Multiple groups

```python
groups = df['group']
ix = (groups == 'miR-137')

kmf.fit(T[~ix], E[~ix], label='control')
ax = kmf.plot()

kmf.fit(T[ix], E[ix], label='miR-137')
ax = kmf.plot(ax=ax)
```

Alternatively, for many more groups and more “pandas-esque”:

```python
ax = plt.subplot(111)
kmf = KaplanMeierFitter()
```

(continues on next page)
for name, grouped_df in df.groupby('group'):
    kmf.fit(grouped_df["T"], grouped_df["E"], label=name)
    kmf.plot(ax=ax)

Similar functionality exists for the NelsonAalenFitter:

```python
from lifelines import NelsonAalenFitter
naf = NelsonAalenFitter()
naf.fit(T, event_observed=E)
```

but instead of a `survival_function_` being exposed, a `cumulative_hazard_` is.

**Note:** Similar to Scikit-Learn, all statistically estimated quantities append an underscore to the property name.

**Note:** More detailed docs about estimating the survival function and cumulative hazard are available in Survival analysis with lifelines.

### 1.1.3 Getting data in the right format

Often you’ll have data that looks like::

*start_time1*, *end_time1*
*start_time2*, *end_time2*
*start_time3*, `None`
*start_time4*, *end_time4*

*lifelines* has some utility functions to transform this dataset into duration and censoring vectors. The most common one is `datetimes_to_durations`. The docs for it are [here](https://lifelines.readthedocs.io/).

```python
from lifelines.utils import datetimes_to_durations

# start_times is a vector or list of datetime objects or datetime strings
# end_times is a vector or list of (possibly missing) datetime objects or datetime strings
T, E = datetimes_to_durations(start_times, end_times, freq='h')
```

Perhaps you are interested in viewing the survival table given some durations and censoring vectors. The docs for it are [here](https://lifelines.readthedocs.io/).

```python
from lifelines.utils import survival_table_from_events

table = survival_table_from_events(T, E)
print(table.head())
```

```
    event_at removed observed censored entrance at_risk
0      0.0       0      0      0     163     163
6      6.0       1      1      0     163     163
7      7.0       2      1      1      0     162
9      9.0       3      3      0      0     160
```
1.4 Survival regression

While the above KaplanMeierFitter and NelsonAalenFitter are useful, they only give us an “average” view of the population. Often we have specific data at the individual level, either continuous or categorical, that we would like to use. For this, we turn to survival regression, specifically AalenAdditiveFitter, WeibullAFTFitter, and CoxPHFitter.

```python
from lifelines.datasets import load_regression_dataset
regression_dataset = load_regression_dataset()
regression_dataset.head()
```

The input of the fit method’s API in a regression is different. All the data, including durations, censored indicators and covariates must be contained in a Pandas DataFrame (yes, it must be a DataFrame). The duration column and event occurred column must be specified in the call to fit. Below we model our regression dataset using the Cox proportional hazard model, full docs here.

```python
from lifelines import CoxPHFitter

cph = CoxPHFitter()
cph.fit(regression_dataset, 'T', event_col='E')
cph.print_summary()
```

```
<lifelines.CoxPHFitter: fitted with 200 observations, 11 censored>
  duration col = 'T'
  event col = 'E'
  number of subjects = 200
  number of events = 189
  log-likelihood = -807.62
  time fit was run = 2019-01-27 23:11:22 UTC

  ---
  coef  exp(coef)  se(coef)     z     p  -log2(p) lower 0.95 upper 0.95
  var1  0.22     1.25     0.07   2.99 <0.005   8.49    0.08    0.37
  var2  0.05     1.05     0.08   0.61   0.54    0.89   -0.11    0.21
  var3  0.22     1.24     0.08   2.88 <0.005   7.97    0.07    0.37

  Concordance = 0.58
  Likelihood ratio test = 15.54 on 3 df, -log2(p)=9.47
```

cph.plot()
The same dataset, but with a Weibull Accelerated Failure Time model. This model was two parameters (see docs here), and we can choose to model both using our covariates or just one. Below we model just the scale parameter, \( \lambda \).

```python
from lifelines import WeibullAFTFitter

wft = WeibullAFTFitter()
wft.fit(regression_dataset, 'T', event_col='E', ancillary_df=regression_dataset)
wft.print_summary()
```

```python
<lifelines.WeibullAFTFitter: fitted with 200 observations, 11 censored>
duration col = 'T'
    event col = 'E'
number of subjects = 200
number of events = 189
log-likelihood = -504.48
time fit was run = 2019-02-19 22:07:57 UTC
```

(continues on next page)
```
---
coef  exp(coef)  se(coef)  z  p  -log2(p)  lower 0.95  upper 0.95
lambda: var1  -0.08  0.92  0.02  -3.45 <0.005  10.78  -0.13  -0.04
        var2  -0.02  0.98  0.03  -0.56  0.57  0.80  -0.07
        var3  -0.08  0.92  0.02  -3.33 <0.005  10.15  -0.13  -0.03
    _intercept  2.53  12.57  0.05  51.12 <0.005  inf  2.43  2.63
rho: _intercept  1.09  2.98  0.05  20.12 <0.005  296.66  0.99
    _intercept  1.20
---
Concordance = 0.58
Log-likelihood ratio test = 19.73 on 3 df, -log2(p)=12.34

wft.plot()
```
If we focus on Aalen’s Additive model, which has time-varying hazards:

```python
# Using Aalen's Additive model
from lifelines import AalenAdditiveFitter
aaf = AalenAdditiveFitter(fit_intercept=False)
aaf.fit(regression_dataset, 'T', event_col='E')
```

Along with `CoxPHFitter` and `WeibullAFTFitter`, after fitting you’ll have access to properties like `cumulative_hazards_` and methods like `plot`, `predict_cumulative_hazards`, and `predict_survival_function`. The latter two methods require an additional argument of individual covariates:

```python
X = regression_dataset.drop(['E', 'T'], axis=1)
aaf.predict_survival_function(X.iloc[10:12]).plot()  # get the unique survival functions of two subjects
```

Like the above estimators, there is also a built-in plotting method:

```python
aaf.plot()
```
1.2 Introduction to survival analysis

1.2.1 Applications

Traditionally, survival analysis was developed to measure lifespans of individuals. An actuary or health professional would ask questions like “how long does this population live for?”, and answer it using survival analysis. For example,
the population may be a nation’s population (for actuaries), or a population stricken by a disease (in the medical professionals case). Traditionally, sort of a morbid subject.

The analysis can be further applied to not just traditional births and deaths, but any duration. Medical professionals might be interested in the time between childbirths, where a birth in this case is the event of having a child, and a death is becoming pregnant again! (obviously, we are loose with our definitions of birth and death) Another example is users subscribing to a service: a birth is a user who joins the service, and a death is when the user leaves the service.

1.2.2 Censoring

At the time you want to make inferences about durations, it is possible, likely true, that not all the death events have occurred yet. For example, a medical professional will not wait 50 years for each individual in the study to pass away before investigating – he or she is interested in the effectiveness of improving lifetimes after only a few years, or months possibly.

The individuals in a population who have not been subject to the death event are labeled as right-censored, i.e., we did not (or can not) view the rest of their life history due to some external circumstances. All the information we have on these individuals are their current lifetime durations (which is naturally less than their actual lifetimes).

Note: There is also left-censoring and interval censoring, which are expanded on later.

A common mistake data analysts make is choosing to ignore the right-censored individuals. We will see why this is a mistake next.

Consider a case where the population is actually made up of two subpopulations, A and B. Population A has a very small lifespan, say 2 months on average, and population B enjoys a much larger lifespan, say 12 months on average. We may not know this distinction beforehand. At \( t = 10 \), we wish to investigate the average lifespan for everyone.

In the figure below, the red lines denote the lifespan of individuals where the death event has been observed, and the blue lines denote the lifespan of the right-censored individuals (deaths have not been observed). If we are asked to estimate the average lifetime of our population, and we naively decided to not included the right-censored individuals, it is clear that we would be severely underestimating the true average lifespan.

```python
from lifelines.plotting import plot_lifetimes
import numpy as np
from numpy.random import uniform, exponential

N = 25
CURRENT_TIME = 10

actual_lifetimes = np.array([exponential(12) if uniform() < 0.5 else exponential(2) for i in range(N)])
observed_lifetimes = np.minimum(actual_lifetimes, CURRENT_TIME)
death_observed = actual_lifetimes < CURRENT_TIME

ax = plot_lifetimes(observed_lifetimes, event_observed=death_observed)
ax.set_xlim(0, 25)
ax.vlines(10, 0, 30, lw=2, linestyles='--')
ax.set_xlabel("time")
ax.set_ylabel("Births and deaths of our population, at \$t=10\$")
print("Observed lifetimes at time \$t\$: \
\n\%")
```
Observed lifetimes at time 10:

[10. 1.1 8. 10. 3.43 0.63 6.28 1.03 2.37 6.17 10. 0.21 2.71 1.25 10. 3.4 0.62 1.94 0.22 7.43 6.16 10. 9.41 10. 10.]

Furthermore, if we instead simply took the mean of all observed lifespans, including the current lifespans of right-censored instances, we would still be underestimating the true average lifespan. Below we plot the actual lifetimes of all instances (recall we do not see this information at $t = 10$).

```python
ax = plot_lifetimes(actual_lifetimes, event_observed=death_observed)
ax.vlines(10, 0, 30, lw=2, linestyles='--')
ax.set_xlim(0, 25)
```
Survival analysis was originally developed to solve this type of problem, that is, to deal with estimation when our data is right-censored. Even in the case where all events have been observed, i.e. no censoring, survival analysis is still a very useful tool to understand durations.

The observations need not always start at zero, either. This was done only for understanding in the above example. Consider the example where a customer entering a store is a birth: a customer can enter at any time, and not necessarily at time zero. In survival analysis, durations are relative: individuals may start at different times. (We actually only need the duration of the observation, and not necessarily the start and end time.)

We next introduce the two fundamental objects in survival analysis, the survival function and the hazard function.

### 1.2.3 Survival function

Let $T$ be a (possibly infinite, but always non-negative) random lifetime taken from the population under study. For example, the amount of time a couple is married. Or the time it takes a user to enter a webpage (an infinite time if they never do). The survival function - $S(t)$ - of a population is defined as

\[ S(t) = Pr(T > t) \]

In plain English: the survival function defines the probability the death event has not occurred yet at time $t$, or equivalently, the probability of surviving past time $t$. Note the following properties of the survival function:

1. $0 \leq S(t) \leq 1$
2. $F_T(t) = 1 - S(t)$, where $F_T(t)$ is the CDF of $T$, which implies
3. $S(t)$ is a non-increasing function of $t$.

### 1.2.4 Hazard curve

We are also interested in the probability of the death event occurring at time $t$, given that the death event has not occurred until time $t$. Mathematically, that is:

\[ \lim_{\delta t \to 0} Pr(t \leq T \leq t + \delta t | T > t) \]

This quantity goes to 0 as $\delta t$ shrinks, so we divide this by the interval $\delta t$ (like we might do in calculus). This defines the hazard function at time $t$, $h(t)$:

\[ h(t) = \lim_{\delta t \to 0} \frac{Pr(t \leq T \leq t + \delta t | T > t)}{\delta t} \]

It can be shown that this is equal to:

\[ h(t) = \frac{-S'(t)}{S(t)} \]

and solving this differential equation (cool, it is a differential equation!), we get:

\[ S(t) = \exp \left( - \int_0^t h(z)dz \right) \]

What I love about the above equation is that it defines all survival functions. Notice that we can now speak either about the survival function, $S(t)$, or the hazard function, $h(t)$, and we can convert back and forth quite easily. It also gives us another, albeit not as useful, expression for $T$. Upon differentiation and some algebra, we recover:

\[ f_T(t) = h(t) \exp \left( - \int_0^t h(z)dz \right) = h(t)S(t) \]
Of course, we do not observe the true survival curve of a population. We must use the observed data to estimate it. There are many ways to estimate the survival function and the hazard rate, which brings us to *estimation using lifelines.*

### 1.3 Estimating univariate models

In the previous *section,* we introduced the use of survival analysis, the need, and the mathematical objects on which it relies. In this article, we will work with real data and the *lifelines* library to estimate these mathematical objects.

#### 1.3.1 Estimating the survival function using Kaplan-Meier

For this example, we will be investigating the lifetimes of political leaders around the world. A political leader, in this case, is defined by a single individual’s time in office who controls the ruling regime. This political leader could be an elected president, unelected dictator, monarch, etc. The birth event is the start of the individual’s tenure, and the death event is the retirement of the individual. Censoring can occur if they are a) still in offices at the time of dataset compilation (2008), or b) die while in power (this includes assassinations).

For example, the Bush regime began in 2000 and officially ended in 2008 upon his retirement, thus this regime’s lifespan was eight years, and there was a “death” event observed. On the other hand, the JFK regime lasted 2 years, from 1961 and 1963, and the regime’s official death event was not observed – JFK died before his official retirement.

(This is an example that has gladly redefined the birth and death events, and in fact completely flips the idea upside down by using deaths as the censoring event. This is also an example where the current time is not the only cause of censoring; there are the alternative events (e.g., death in office) that can be the cause of censoring.

To estimate the survival function, we first will use the Kaplan-Meier Estimate, defined:

\[ \hat{S}(t) = \prod_{t_i < t} \frac{n_i - d_i}{n_i} \]

where \(d_i\) are the number of death events at time \(t\) and \(n_i\) is the number of subjects at risk of death just prior to time \(t\).

Let’s bring in our dataset.

```python
from lifelines.datasets import load_dd
data = load_dd()
data.head()
```
From the lifelines library, we’ll need the KaplanMeierFitter for this exercise:

```python
from lifelines import KaplanMeierFitter
kmf = KaplanMeierFitter()
```

**Note:** Other ways to estimate the survival function in lifelines are discussed below.

For this estimation, we need the duration each leader was/has been in office, and whether or not they were observed to have left office (leaders who died in office or were in office in 2008, the latest date this data was record at, do not have observed death events)

We next use the KaplanMeierFitter method fit to fit the model to the data. (This is similar to, and inspired by, scikit-learn’s fit/predict API).

Below we fit our data with the KaplanMeierFitter:

```python
T = data['duration']
E = data['observed']

kmf.fit(T, event_observed=E)
```

After calling the fit method, the KaplanMeierFitter has a property called survival_function_ (again, we follow the styling of scikit-learn, and append an underscore to all properties that were computational estimated). The property is a Pandas DataFrame, so we can call plot on it:

```python
kmf.survival_function_.plot()
plt.title('Survival function of political regimes');
How do we interpret this? The y-axis represents the probability a leader is still around after $t$ years, where $t$ years is on the x-axis. We see that very few leaders make it past 20 years in office. Of course, like all good stats, we need to report how uncertain we are about these point estimates, i.e., we need confidence intervals. They are computed in the call to `fit`, and located under the `confidence_interval_` property. (The method uses exponential Greenwood confidence interval. The mathematics are found in these notes.)

\[ S(t) = Pr(T > t) \]

Alternatively, we can call `plot` on the `KaplanMeierFitter` itself to plot both the KM estimate and its confidence intervals:

```python
kmf.plot()
```
The median time in office, which defines the point in time where on average 1/2 of the population has expired, is a property:

```python
kmf.median_
# 4.0
```

Interesting that it is only four years. That means, around the world, elected leaders have a 50% chance of cessation in four years or less!

Let’s segment on democratic regimes vs non-democratic regimes. Calling `plot` on either the estimate itself or the fitter object will return an `axis` object, that can be used for plotting further estimates:

```python
ax = plt.subplot(111)
dem = (data["democracy"] == "Democracy")
kmf.fit(T[dem], event_observed=E[dem], label="Democratic Regimes")
kmf.plot(ax=ax)
kmf.fit(T[~dem], event_observed=E[~dem], label="Non-democratic Regimes")
kmf.plot(ax=ax)
plt.ylim(0, 1);
plt.title("Lifespans of different global regimes");
```

![Lifespans of different global regimes](image)

We might be interested in estimating the probabilities in between some points. We can do that with the `timeline`
argument. We specify the times we are interested in and are returned a DataFrame with the probabilities of survival at those points:

```python
ax = plt.subplot(111)
t = np.linspace(0, 50, 51)
kmf.fit(T[dem], event_observed=E[dem], timeline=t, label="Democratic Regimes")
ax = kmf.plot(ax=ax)
print("Median survival time of democratic: ", kmf.median_)

kmf.fit(T[~dem], event_observed=E[~dem], timeline=t, label="Non-democratic Regimes")
ax = kmf.plot(ax=ax)
print("Median survival time of non-democratic: ", kmf.median_)

plt.ylim(0, 1)
plt.title("Lifespans of different global regimes");

""
Median survival time of democratic: 3
Median survival time of non-democratic: 6
"""
It is incredible how much longer these non-democratic regimes exist for. A democratic regime does have a natural bias towards death though: both via elections and natural limits (the US imposes a strict eight-year limit). The median of a non-democratic is only about twice as large as a democratic regime, but the difference is apparent in the tails: if you’re a non-democratic leader, and you’ve made it past the 10 year mark, you probably have a long life ahead. Meanwhile, a democratic leader rarely makes it past ten years, and then have a very short lifetime past that.

Here the difference between survival functions is very obvious, and performing a statistical test seems pedantic. If the curves are more similar, or we possess less data, we may be interested in performing a statistical test. In this case, lifelines contains routines in lifelines.statistics to compare two survival curves. Below we demonstrate this routine. The function logrank_test is a common statistical test in survival analysis that compares two event series’ generators. If the value returned exceeds some pre-specified value, then we rule that the series have different generators.

```python
from lifelines.statistics import logrank_test
results = logrank_test(T[dem], T[~dem], E[dem], E[~dem], alpha=.99)
results.print_summary()
```

```markdown
```
lifeselines.StatisticalResult
t_0 = -1
null_distribution = chi squared
degrees_of_freedom = 1
    alpha = 0.99
```

```python
---
test_statistic p -log2(p)
  260.47 <0.005 192.23
```

```
```

Lets compare the different types of regimes present in the dataset:

```python
regime_types = data['regime'].unique()
for i, regime_type in enumerate(regime_types):
    ax = plt.subplot(2, 3, i + 1)
    ix = data['regime'] == regime_type
    kmf.fit(T[ix], E[ix], label=regime_type)
    kmf.plot(ax=ax, legend=False)
    plt.title(regime_type)
    plt.xlim(0, 50)
    if i==0:
        plt.ylabel('Frac. in power after $n$ years')
```

```python
plt.tight_layout()
```
There are alternative (and sometimes better) tests of survival curves, and we explain more here: Statistically compare two populations

Getting data into the right format

*lifelines* data format is consistent across all estimator class and functions: an array of individual durations, and the individuals event observation (if any). These are often denoted \( T \) and \( E \) respectively. For example:

\[
T = [0,3,3,2,1,2] \\
E = [1,1,0,0,1,1]
\]

```python
kmf.fit(T, event_observed=E)
```

The raw data is not always available in this format – *lifelines* includes some helper functions to transform data formats to *lifelines* format. These are located in the *lifelines.utils* sub-library. For example, the function `datetimes_to_durations` accepts an array or Pandas object of start times/dates, and an array or Pandas objects of end times/dates (or None if not observed):

```python
from lifelines.utils import datetimes_to_durations

start_date = ['2013-10-10 00:00:00', '2013-10-09', '2013-10-10']
end_date = ['2013-10-13', '2013-10-10', None]
T, E = datetimes_to_durations(start_date, end_date, fill_date='2013-10-15')
print('T (durations): ', T)
print('E (event_observed): ', E)
```

\[
T \text{ (durations): } [ 3. 1. 5.]
E \text{ (event_observed): } [ \text{True True False}]
\]

The function `datetimes_to_durations` is very flexible, and has many keywords to tinker with.

1.3.2 Estimating hazard rates using Nelson-Aalen

The survival curve is a great way to summarize and visualize the survival dataset, however it is not the only way. If we are curious about the hazard function \( h(t) \) of a population, we unfortunately cannot transform the Kaplan Meier estimate – statistics doesn’t work quite that well. Fortunately, there is a proper non-parametric estimator of the cumulative hazard function:

\[
H(t) = \int_0^t \lambda(z) \, dz
\]

The estimator for this quantity is called the Nelson Aalen estimator:

\[
\hat{H}(t) = \sum_{t_i \leq t} \frac{d_i}{n_i}
\]

where \( d_i \) is the number of deaths at time \( t_i \) and \( n_i \) is the number of susceptible individuals.

In *lifelines*, this estimator is available as the *NelsonAalenFitter*. Let’s use the regime dataset from above:

```python
T = data["duration"]
E = data["observed"]

from lifelines import NelsonAalenFitter
```

(continues on next page)
naf = NelsonAalenFitter()
naf.fit(T, event_observed=E)

After fitting, the class exposes the property `cumulative_hazard_` as a DataFrame:

```python
print(naf.cumulative_hazard_.head())
naf.plot()
```

<table>
<thead>
<tr>
<th>NA-estimate</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0.000000</td>
</tr>
<tr>
<td>1</td>
<td>0.325912</td>
</tr>
<tr>
<td>2</td>
<td>0.507356</td>
</tr>
<tr>
<td>3</td>
<td>0.671251</td>
</tr>
<tr>
<td>4</td>
<td>0.869867</td>
</tr>
</tbody>
</table>

[5 rows x 1 columns]
The cumulative hazard has less immediate understanding than the survival curve, but the hazard curve is the basis of more advanced techniques in survival analysis. Recall that we are estimating cumulative hazard curve, \( H(t) \). (Why? The sum of estimates is much more stable than the point-wise estimates.) Thus we know the rate of change of this curve is an estimate of the hazard function.

Looking at figure above, it looks like the hazard starts off high and gets smaller (as seen by the decreasing rate of change). Let’s break the regimes down between democratic and non-democratic, during the first 20 years:

**Note:** We are using the `loc` argument in the call to `plot` here: it accepts a slice and plots only points within that slice.

```python
naf.fit(T[dem], event_observed=E[dem], label="Democratic Regimes")
ax = naf.plot(loc=slice(0, 20))
naf.fit(T[~dem], event_observed=E[~dem], label="Non-democratic Regimes")
naf.plot(ax=ax, loc=slice(0, 20))
plt.title("Cumulative hazard function of different global regimes");
```
Looking at the rates of change, I would say that both political philosophies have a constant hazard, albeit democratic regimes have a much higher constant hazard.

**Smoothing the hazard curve**

Interpretation of the cumulative hazard function can be difficult – it is not how we usually interpret functions. On the other hand, most survival analysis is done using the cumulative hazard function, so understanding it is recommended. Alternatively, we can derive the more-interpretable hazard curve, but there is a catch. The derivation involves a kernel smoother (to smooth out the differences of the cumulative hazard curve), and this requires us to specify a bandwidth parameter that controls the amount of smoothing. This functionality is in the `smoothed_hazard_` and `hazard_confidence_intervals_` methods. Why methods? They require an argument representing the bandwidth.

There is also a `plot_hazard` function (that also requires a `bandwidth` keyword) that will plot the estimate plus the confidence intervals, similar to the traditional `plot` functionality.

```python
bandwidth = 3.

naf.fit(T[dem], event_observed=E[dem], label="Democratic Regimes")
ax = naf.plot_hazard(bandwidth=bandwidth)
naf.fit(T[~dem], event_observed=E[~dem], label="Non-democratic Regimes")
naf.plot_hazard(ax=ax, bandwidth=bandwidth)

plt.title("Hazard function of different global regimes | bandwidth=%.1f" % bandwidth);
plt.ylim(0, 0.4)
plt.xlim(0, 25);
```
1.3. Estimating univariate models
It is more clear here which group has the higher hazard, and Non-democratic regimes appear to have a constant hazard. There is no obvious way to choose a bandwidth, and different bandwidths produce different inferences, so it’s best to be very careful here. My advice: stick with the cumulative hazard function.

```python
bandwidth = 8.0

naf.fit(T[dem], event_observed=E[dem], label="Democratic Regimes")
ax = naf.plot_hazard(bandwidth=bandwidth)

naf.fit(T[~dem], event_observed=E[~dem], label="Non-democratic Regimes")
naf.plot_hazard(ax=ax, bandwidth=bandwidth)

plt.title("Hazard function of different global regimes | bandwidth=%.1f" % bandwidth);
```
1.3.3 Estimating cumulative hazards using parametric models

Fitting to a Weibull model

Note: The parameterization of the Weibull and Exponential model changed in lifelines 0.19.0, released in Feb. 2019.

Another very popular model for survival data is the Weibull model. In contrast to the Nelson-Aalen estimator, this model is a parametric model, meaning it has a functional form with parameters that we are fitting the data to. (The Nelson-Aalen estimator has no parameters to fit to). The survival function looks like:

\[ S(t) = \exp\left(-\left(\frac{t}{\lambda}\right)^\rho\right), \lambda > 0, \rho > 0, \]

A priori, we do not know what \( \lambda \) and \( \rho \) are, but we use the data on hand to estimate these parameters. We model and estimate the cumulative hazard rate instead of the survival function (this is different than the Kaplan-Meier estimator):

\[ H(t) = \left(\frac{t}{\lambda}\right)^\rho \]

In lifelines, estimation is available using the WeibullFitter class. The plot method will plot the cumulative hazard.

```python
from lifelines import WeibullFitter
from lifelines.datasets import load_waltons
data = load_waltons()
T = data['T']
E = data['E']
wf = WeibullFitter().fit(T, E)
wf.print_summary()
wf.plot()
```

```
<lifelines.WeibullFitter: fitted with 163 observations, 7 censored>
  number of subjects = 163
  number of events = 156
  log-likelihood = -672.062
  hypothesis = lambda != 1, rho != 1
---
  coef  se(coef)  lower 0.95  upper 0.95  p  -log2(p)
lambda_  0.02  0.00  0.02  0.02  <0.005  inf
rho_    3.45  0.24  2.97  3.93  <0.005  76.83
***
```
1.3. Estimating univariate models
Other parametric models: Exponential, Log-Logistic & Log-Normal

Similarly, there are other parametric models in lifelines. Generally, which parametric model to choose is determined by either knowledge of the distribution of durations, or some sort of model goodness-of-fit. Below are the built-in parametric models, and the Nelson-Aalen nonparametric model, of the same data.

```python
from lifelines import WeibullFitter
from lifelines import ExponentialFitter
from lifelines import LogNormalFitter
from lifelines import LogLogisticFitter
from lifelines import NelsonAalenFitter
from lifelines import PiecewiseExponentialFitter

from lifelines.datasets import load_waltons

data = load_waltons()

fig, axes = plt.subplots(2, 3, figsize=(9, 5))

T = data['T']
E = data['E']

wbf = WeibullFitter().fit(T, E, label='WeibullFitter')
exf = ExponentialFitter().fit(T, E, label='ExponentialFitter')
lnf = LogNormalFitter().fit(T, E, label='LogNormalFitter')
naf = NelsonAalenFitter().fit(T, E, label='NelsonAalenFitter')
llf = LogLogisticFitter().fit(T, E, label='LogLogisticFitter')
pwf = PiecewiseExponentialFitter([40, 60]).fit(T, E, label='PiecewiseExponentialFitter')

wbf.plot_cumulative_hazard(ax=axes[0][0])
exf.plot_cumulative_hazard(ax=axes[0][1])
lnf.plot_cumulative_hazard(ax=axes[0][2])
naf.plot_cumulative_hazard(ax=axes[1][0])
llf.plot_cumulative_hazard(ax=axes[1][1])
pwf.plot_cumulative_hazard(ax=axes[1][2])
```
1.3. Estimating univariate models

- **WeibullFitter**
- **NelsonAalenFitter**

![Graph comparing WeibullFitter and NelsonAalenFitter models.](image)
lifelines can also be used to define your own parametric model. There is a tutorial on this available, see Piecewise Exponential Models and Creating Custom Models.

Parametric models can also be used to create and plot the survival function, too. Below we compare the parametric models versus the non-parametric Kaplan-Meier estimate:

```python
from lifelines import KaplanMeierFitter

fig, axes = plt.subplots(2, 3, figsize=(9, 5))

T = data['T']
E = data['E']

kmf = KaplanMeierFitter().fit(T, E, label='KaplanMeierFitter')
wbf = WeibullFitter().fit(T, E, label='WeibullFitter')
exf = ExponentialFitter().fit(T, E, label='ExponentialFitter')
lnf = LogNormalFitter().fit(T, E, label='LogNormalFitter')
llf = LogLogisticFitter().fit(T, E, label='LogLogisticFitter')
pwf = PiecewiseExponentialFitter([40, 60]).fit(T, E, label='PiecewiseExponentialFitter')

wbf.plot_survival_function(ax=axes[0][0])
exf.plot_survival_function(ax=axes[0][1])
lnf.plot_survival_function(ax=axes[0][2])
kmf.plot_survival_function(ax=axes[1][0])
llf.plot_survival_function(ax=axes[1][1])
pwf.plot_survival_function(ax=axes[1][2])
```
With parametric models, we have a functional form that allows us to extend the survival function (or hazard or cumulative hazard) past our maximum observed duration. This is called extrapolation. We can do this in a few ways.

```python
timeline = np.linspace(0, 100, 200)

# directly compute the survival function, these return a pandas Series
wbf = WeibullFitter().fit(T, E)
wbf.survival_function_at_times(timeline)
wbf.hazard_at_times(timeline)
wbf.cumulative_hazard_at_times(timeline)

# use the `timeline` kwarg in `fit`
# by default, all functions and properties will use
# these values provided
wbf = WeibullFitter().fit(T, E, timeline=timeline)

wbf.plot_survival_function()
```
To aid model selection, lifelines has provided qq-plots, Selecting a parametric model using QQ plots.

### 1.3.4 Other types of censoring

#### Left censored data and non-detection

We’ve mainly been focusing on right-censoring, which describes cases where we do not observe the death event. This situation is the most common one. Alternatively, there are situations where we do not observe the birth event occurring. Consider the case where a doctor sees a delayed onset of symptoms of an underlying disease. The doctor is unsure when the disease was contracted (birth), but knows it was before the discovery.

Another situation where we have left-censored data is when measurements have only an upper bound, that is, the measurements instruments could only detect the measurement was less than some upper bound. This bound is often called the limit of detection (LOD). In practice, there could be more than one LOD. One very important statistical lesson: don’t “fill-in” this value naively. It’s tempting to use something like one-half the LOD, but this will cause lots of bias in downstream analysis. An example dataset is below:

**Note:** The recommended API for modeling left-censored data using parametric models changed in version 0.21.0. Below is the recommended API.

```python
def = load_nh4()['NH4.Orig.mg.per.L', 'NH4.mg.per.L', 'Censored']
```

```python
T, E = df['NH4.mg.per.L'], ~df['Censored']
kmf = KaplanMeierFitter()
kmf.fit_left_censoring(T, E)
```

Instead of producing a survival function, left-censored data analysis is more interested in the cumulative density function. This is available as the cumulative_density_ property after fitting the data.

```python
print (kmf.cumulative_density_.head())
kmf.plot()  #will plot the CDF
plt.xlabel("Concentration of NH_4")
```

(continues on next page)
Alternatively, you can use a parametric model to model the data. This allows for you to “peer” below the LOD, however using a parametric model means you need to correctly specify the distribution. You can use plots like qq-plots to help invalidate some distributions, see Selecting a parametric model using QQ plots.

```python
fig, axes = plt.subplots(3, 2, figsize=(9, 9))
timeline = np.linspace(0, 0.25, 100)

wf = WeibullFitter().fit_left_censoring(T, E, label="Weibull", timeline=timeline)
lnf = LogNormalFitter().fit_left_censoring(T, E, label="Log Normal", timeline=timeline)
lgf = LogLogisticFitter().fit_left_censoring(T, E, label="Log Logistic", timeline=timeline)

# plot what we just fit, along with the KMF estimate
kmf.plot_cumulative_density(ax=axes[0][0], ci_show=False)
wf.plot_cumulative_density(ax=axes[0][0], ci_show=False)
qq_plot(wf, ax=axes[0][1])
```

1.3. Estimating univariate models
kmf.plot_cumulative_density(ax=axes[1][0], ci_show=False)
lnf.plot_cumulative_density(ax=axes[1][0], ci_show=False)
qq_plot(lnf, ax=axes[1][1])

kmf.plot_cumulative_density(ax=axes[2][0], ci_show=False)
lgf.plot_cumulative_density(ax=axes[2][0], ci_show=False)
qq_plot(lgf, ax=axes[2][1])
1.3. Estimating univariate models
Based on the above, the log-normal distribution seems to fit well, and the Weibull not very well at all.

**Interval censored data**

Data can also be interval censored. An example of this is periodically recording the population of micro-organisms as they die-off. Their deaths are interval censored because you know a subject died between two observations periods. New to lifelines in version 0.21.0, all parametric models have support for interval censored data.

**Note:** The API for \texttt{fit\_interval\_censoring} is different than right and left censored data.

```python
from lifelines.datasets import load_diabetes
df = load_diabetes()
wf = WeibullFitter().fit_interval_censoring(lower_bound=df["left"], upper_bound=df["right"])
```

Another example of using lifelines for interval censored data is located here.

**Left truncated (late entry) data**

Another form of bias that is introduced into a dataset is called left-truncation (or late entry). Left-truncation can occur in many situations. One situation is when individuals may have the opportunity to die before entering into the study. For example, if you are measuring time to death of prisoners in prison, the prisoners will enter the study at different ages. So it’s possible there are some counter-factual individuals who would have entered into your study (that is, went to prison), but instead died early.

All univariate fitters, like \texttt{KaplanMeierFitter} and any parametric models, have an optional argument for \texttt{entry}, which is an array of equal size to the duration array. It describes the time between actual “birth” (or “exposure”) to entering the study.

**Note:** Nothing changes in the duration array: it still measures time from “birth” to time exited study (either by death or censoring). That is, durations refers to the absolute death time rather than a duration relative to the study entry.

Another situation with left-truncation occurs when subjects are exposed before entry into study. For example, a study of time to all-cause mortality of AIDS patients that recruited individuals previously diagnosed with AIDS, possibly years before. In our example below we will use a dataset like this, called the Multicenter Aids Cohort Study. In the figure below, we plot the lifetimes of subjects. A solid line is when the subject was under our observation, and a dashed line represents the unobserved period between diagnosis and study entry. A solid dot at the end of the line represents death.

```python
from lifelines.datasets import load_multicenter_aids_cohort_study
from lifelines.plotting import plot_lifetimes
df = load_multicenter_aids_cohort_study()
plot_lifetimes(
    df["T"] - df["W"],
    event_observed=df["D"],
    entry=df["W"],
)
```

(continues on next page)
event_observed_color="#383838",
    event_censored_color="#383838",
    left_truncated=True,
})
plt.ylabel("Patient Number")
plt.xlabel("Years from AIDS diagnosis")
So subject #77, the subject at the top, was diagnosed with AIDS 7.5 years ago, but wasn’t in our study for the first 4.5 years. From this point-of-view, why can’t we “fill in” the dashed lines and say, for example, “subject #77 lived for 7.5 years”? If we did this, we would severely underestimate chance of dying early on after diagnosis. Why? It’s possible that there were individuals who were diagnosed and then died shortly after, and never had a chance to enter our study. If we did manage to observe them however, they would have depressed the survival curve early on. Thus, “filling in” the dashed lines makes us over confident about what occurs in the early period after diagnosis. We can see this below when we model the survival curve with and without taking into account late entries.

```python
from lifelines import KaplanMeierFitter

kmf = KaplanMeierFitter()
kmf.fit(df["T"], event_observed=df["D"], entry=df["W"], label='modeling late entries')
ax = kmf.plot()

kmf.fit(df["T"], event_observed=df["D"], label='ignoring late entries')
kmf.plot(ax=ax)
```
1.4 Piecewise Exponential models and creating custom models

This section will be easier if we recall our three mathematical “creatures” and the relationships between them. First is the survival function, \( S(t) \), that represents the probability of living past some time, \( t \). Next is the always non-negative and non-decreasing cumulative hazard function, \( H(t) \). Its relation to \( S(t) \) is:

\[
S(t) = \exp(-H(t))
\]

Finally, the hazard function, \( h(t) \), is the derivative of the cumulative hazard:

\[
h(t) = \frac{dH(t)}{dt}
\]

which has the immediate relation to the survival function:

\[
S(t) = \exp\left(-\int_0^t h(s)ds\right)
\]

Notice that any of the three absolutely defines the other two. Some situations make it easier to define one vs the others. For example, in the Cox model, it’s easiest to work with the hazard, \( h(t) \). In this section on parametric univariate models, it’ll be easiest to work with the cumulative hazard. This is because of an asymmetry in math: derivatives are much easier to compute than integrals. So, if we define the cumulative hazard, both the hazard and survival function are much easier to reason about vs if we define the hazard and ask questions about the other two. At first, it may be easier to think about the hazard, and that’s fine, but so long as we are clever enough to also determine the cumulative hazard, then we can ride the computational train. This will be clear by the end of the tutorial.

First, let’s revisit some simpler parametric models.

1.4.1 The Exponential model

Recall that the Exponential model has a constant hazard, that is:

\[
h(t) = \frac{1}{\lambda}
\]

which implies that the cumulative hazard, \( H(t) \), has a pretty simple form: \( H(t) = \frac{t}{\lambda} \). Below we fit this model to some survival data.
This model does a poor job of fitting to our data. If I fit a non-parametric model, like the Nelson-Aalen model, to this data, the lack of fit is very obvious.
It should be clear that the single parameter model is just averaging the hazards over the entire time period. In reality though, the true hazard rate exhibits some complex non-linear behaviour.

### 1.4.2 Piecewise Exponential models

What if we could break out model into different time periods, and fit an exponential model to each of those? For example, we define the hazard as:

\[
    h(t) = \begin{cases} 
        \lambda_0, & \text{if } t \leq \tau_0 \\
        \lambda_1, & \text{if } \tau_0 < t \leq \tau_1 \\
        \lambda_2, & \text{if } \tau_1 < t \leq \tau_2 \\
        \vdots 
    \end{cases}
\]

This model should be flexible enough to fit better to our dataset.

The cumulative hazard is only slightly more complicated, but not too much and can still be defined in Python. In *lifelines*, univariate models are constructed such that one only needs to define the cumulative hazard model with the parameters of interest, and all the hard work of fitting, creating confidence intervals, plotting, etc. is taken care.

For example, *lifelines* has implemented the `PiecewiseExponentialFitter` model. Internally, the code is a single function that defines the cumulative hazard. The user specifies where they believe the “breaks” are, and *lifelines* estimates the best \( \lambda_i \).

[4]: from lifelines import PiecewiseExponentialFitter

    # looking at the above plot, I think there may be breaks at t=40 and t=60.
    pf = PiecewiseExponentialFitter(breakpoints=[40, 60]).fit(T, E)

    fig, axs = plt.subplots(nrows=1, ncols=2, figsize=(10, 4))

    ax = pf.plot(ax=axs[1])
    pf.plot_hazard(ax=axs[0])
ax = naf.plot(ax=ax, ci_show=False)
axs[0].set_title("hazard"); axs[1].set_title("cumulative_hazard")

pf.print_summary(3)

<lifelines.PiecewiseExponentialFitter: fitted with 163 observations, 7 censored>
number of subjects = 163
number of events = 156
log-likelihood = -647.118
  hypothesis = lambda_0_ != 1, lambda_1_ != 1, lambda_2_ != 1

<table>
<thead>
<tr>
<th></th>
<th>coef</th>
<th>se(coef)</th>
<th>lower 0.95</th>
<th>upper 0.95</th>
<th>p</th>
<th>-log2(p)</th>
</tr>
</thead>
<tbody>
<tr>
<td>lambda_0_</td>
<td>162.989</td>
<td>27.144</td>
<td>109.787</td>
<td>216.191</td>
<td>&lt;0.0005</td>
<td>28.630</td>
</tr>
<tr>
<td>lambda_1_</td>
<td>31.366</td>
<td>4.043</td>
<td>23.442</td>
<td>39.290</td>
<td>&lt;0.0005</td>
<td>43.957</td>
</tr>
<tr>
<td>lambda_2_</td>
<td>4.686</td>
<td>0.624</td>
<td>3.462</td>
<td>5.910</td>
<td>&lt;0.0005</td>
<td>28.055</td>
</tr>
</tbody>
</table>

We can see a much better fit in this model. A quantitative measure of better fit is to compare the log-likelihood of the exponential model and the piecewise exponential model (higher is better). The log-likelihood went from -772 to -647, respectively. We could keep going and add more and more breakpoints, but that would end up overfitting to the data.

1.4.3 Univariate models in lifelines

I mentioned that the PiecewiseExponentialFitter was implemented using only its cumulative hazard function. This is not a lie. lifelines has very general semantics for univariate fitters. For example, this is how the entire ExponentialFitter is implemented:

```python
class ExponentialFitter(ParametericUnivariateFitter):
    _fitted_parameter_names = ["lambda_"]

    def _cumulative_hazard(self, params, times):
        lambda_ = params[0]
        return lambda_ * times
```

60 Chapter 1. Contents:
We only need to specify the cumulative hazard function because of the 1-1 relationship between the cumulative hazard function and the survival function and the hazard rate. From there, lifelines handles the rest.

### 1.4.4 Defining our own survival models

To show off the flexibility of lifelines univariate models, we’ll create a brand new, never before seen, survival model. Looking at the Nelson-Aalen fit, the cumulative hazard looks looks like their might be an asymptote at \( t = 80 \). This may correspond to an absolute upper limit of subjects’ lives. Let’s start with that functional form.

\[
H_1(t; \alpha) = \frac{\alpha}{(80 - t)}
\]

We subscript 1 because we’ll investigate other models. In a lifelines univariate model, this is defined in the following code.

**Important:** in order to compute derivatives, you must use the numpy imported from the autograd library. This is a thin wrapper around the original numpy. See `import` below.

```python
from lifelines.fitters import ParametericUnivariateFitter
import autograd.numpy as np

class InverseTimeHazardFitter(ParametericUnivariateFitter):
    # we tell the model what we want the names of the unknown parameters to be
    _fitted_parameter_names = ['alpha_']

    # this is the only function we need to define. It always takes two arguments:
    # params: an iterable that unpacks the parameters you'll need in the order of _
    # -fitted_parameter_names
    # times: a vector of times that will be passed in.
    def _cumulative_hazard(self, params, times):
        alpha = params[0]
        return alpha / (80 - times)
```

```python
itf = InverseTimeHazardFitter()
itf.fit(T, E)
itf.print_summary()
ax = itf.plot(figsize=(8,5))
ax = naf.plot(ax=ax, ci_show=False)
plt.legend()
```

```
lifelines.InverseTimeHazardFitter: fitted with 163 observations, 7 censored
number of subjects = 163
number of events = 156
log-likelihood = -697.84
hypothesis = alpha_ != 1

<table>
<thead>
<tr>
<th></th>
<th>coef</th>
<th>se(coef)</th>
<th>lower 0.95</th>
<th>upper 0.95</th>
<th>p</th>
<th>-log2(p)</th>
</tr>
</thead>
<tbody>
<tr>
<td>alpha</td>
<td>21.51</td>
<td>1.72</td>
<td>18.13</td>
<td>24.88</td>
<td>&lt;0.005</td>
<td>106.22</td>
</tr>
</tbody>
</table>
```

1.4. Piecewise Exponential models and creating custom models 61
The best fit of the model to the data is:

\[ H_1(t) = \frac{21.51}{80 - t} \]

Our choice of 80 as an asymptote was maybe mistaken, so let’s allow the asymptote to be another parameter:

\[ H_2(t; \alpha, \beta) = \frac{\alpha}{\beta - t} \]

If we define the model this way, we need to add a bound to the values that \( \beta \) can take. Obviously it can’t be smaller than or equal to the maximum observed duration. Generally, the cumulative hazard must be positive and non-decreasing. Otherwise the model fit will hit convergence problems.

```python
[7]: class TwoParamInverseTimeHazardFitter(ParametericUnivariateFitter):
    _fitted_parameter_names = ['alpha_', 'beta_']

    # Sequence of (min, max) pairs for each element in x. None is used to specify no bound
    _bounds = [(0, None), (75.0001, None)]

    def _cumulative_hazard(self, params, times):
        a, b = params
        return a / (b - times)
```

```python
[8]: two_f = TwoParamInverseTimeHazardFitter()
two_f.fit(T, E)
two_f.print_summary()
ax = itf.plot(ci_show=False, figsize=(8,5))
ax = naf.plot(ax=ax, ci_show=False)
two_f.plot(ax=ax)
plt.legend()
```
From the output, we see that the value of 76.55 is the suggested asymptote, that is:

\[ H_2(t) = \frac{16.50}{76.55 - t} \]

The curve also appears to track against the Nelson-Aalen model better too. Let's try one additional parameter, \( \gamma \), some sort of measure of decay.

\[ H_3(t; \alpha, \beta, \gamma) = \frac{\alpha}{(\beta - t)^\gamma} \]

---

from lifelines.fitters import ParametericUnivariateFitter
class ThreeParamInverseTimeHazardFitter(ParametericUnivariateFitter):
    _fitted_parameter_names = ['alpha_', 'beta_', 'gamma_']
    _bounds = [(0, None), (75.0001, None), (0, None)]

# this is the only function we need to define. It always takes two arguments:
# params: an iterable that unpacks the parameters you'll need in the order of _fitted_parameter_names
# times: a numpy vector of times that will be passed in by the optimizer (continues on next page)
def _cumulative_hazard(self, params, times):
    a, b, c = params
    return a / (b - times) ** c

three_f = ThreeParamInverseTimeHazardFitter()
three_f.fit(T, E)
three_f.print_summary()

ax = itf.plot(ci_show=False, figsize=(8,5))
ax = naf.plot(ax=ax, ci_show=False)
ax = two_f.plot(ax=ax, ci_show=False)
ax = three_f.plot(ax=ax)
plt.legend()

Our new asymptote is at \( t \approx 100 \), c.i. = (87, 112). The model appears to fit the early times better than the previous models as well, however our \( \alpha \) parameter has more uncertainty now. Continuing to add parameters isn’t advisable, as we will overfit to the data.
Why fit parametric models anyways?

Taking a step back, we are fitting parametric models and comparing them to the non-parametric Nelson-Aalen. Why not just always use the Nelson-Aalen model?

1) Sometimes we have scientific motivations to use a parametric model. That is, using domain knowledge, we may know the system has a parametric model and we wish to fit to that model.

2) In a parametric model, we are borrowing information from all observations to determine the best parameters. To make this more clear, imagine taking a single observation and changing it’s value wildly. The fitted parameters would change as well. On the other hand, imagine doing the same for a non-parametric model. In this case, only the local survival function or hazard function would change. Because parametric models can borrow information from all observations, and there are much fewer unknowns than a non-parametric model, parametric models are said to be more statistical efficient.

3) Extrapolation: non-parametric models are not easily extended to values outside the observed data. On the other hand, parametric models have no problem with this. However, extrapolation outside observed values is a very dangerous activity.

```python
[11]: fig, axs = plt.subplots(3, figsize=(7, 8), sharex=True)
new_timeline = np.arange(0, 85)
three_f = ThreeParamInverseTimeHazardFitter().fit(T, E, timeline=new_timeline)
three_f.plot_hazard(label='hazard', ax=axs[0]).legend()
three_f.plot_cumulative_hazard(label='cumulative hazard', ax=axs[1]).legend()
three_f.plot_survival_function(label='survival function', ax=axs[2]).legend()
fig.subplots_adjust(hspace=0)
# Hide x labels and tick labels for all but bottom plot.
for ax in axs:
    ax.label_outer()
```
Inverse Gaussian distribution

The inverse Gaussian distribution is another popular model for survival analysis. Unlike other models, it’s hazard does not asymptotically converge to 0, allowing for a long tail of survival. Let’s model this, using the same parameterization from Wikipedia.

```python
import autograd.numpy as np
from autograd.scipy.stats import norm

class InverseGaussianFitter(ParametericUnivariateFitter):
    _fitted_parameter_names = ['lambda_', 'mu_']

    def _cumulative_density(self, params, times):
        mu_, lambda_ = params
        v = norm.cdf(np.sqrt(lambda_ / times) * (times / mu_ - 1), loc=0, scale=1) +
            np.exp(2 * lambda_ / mu_) * norm.cdf(-np.sqrt(lambda_ / times) * -
            (times / mu_ + 1), loc=0, scale=1)
        return v

    def _cumulative_hazard(self, params, times):
        return -np.log(1-self._cumulative_density(params, times))
```

[79]: import autograd.numpy as np
     from autograd.scipy.stats import norm
from lifelines.datasets import load_rosi
rossi = load_rosi()

igf = InverseGaussianFitter()
igf.fit(rossi['week'], rossi['arrest'], timeline=np.arange(1, 500))
igf.print_summary()
igf.plot_hazard()

Looking for more examples of what you can build?
See other unique survival models in the docs on time-lagged survival

1.5 Time-lagged conversion rates and cure models

Suppose in our population we have a subpopulation that will never experience the event of interest. Or, for some subjects the event will occur so far in the future that it’s essentially at time infinity. The survival function should not asymptotically approach zero, but some positive value. Models that describe this are sometimes called cure models or time-lagged conversion models.
There’s a serious fault in using parametric models for these types of problems that non-parametric models don’t have. The most common parametric models like Weibull, Log-Normal, etc. all have strictly increasing cumulative hazard functions, which means the corresponding survival function will always converge to 0.

Let’s look at an example of this problem. Below I generated some data that has individuals who will not experience the event, not matter how long we wait.

```python
%matplotlib inline
%config InlineBackend.figure_format = 'retina'

import matplotlib.pyplot as plt
import autograd.numpy as np
import autograd.scipy.special as expit, logit
import pandas as pd
plt.style.use('bmh')

N = 200
U = np.random.rand(N)
T = -(logit(-np.log(U) / 0.5) - np.random.exponential(2, N) - 6.00) / 0.50
E = ~np.isnan(T)
T[np.isnan(T)] = 50

from lifelines import KaplanMeierFitter
kmf = KaplanMeierFitter().fit(T, E)
kmf.plot(figsize=(8,4))
plt.ylim(0, 1);
```

It should be clear that there is an asymptote at around 0.6. The non-parametric model will always show this. If this is true, then the cumulative hazard function should have a horizontal asymptote as well. Let’s use the Nelson-Aalen model to see this.

```python
from lifelines import NelsonAalenFitter
naf = NelsonAalenFitter().fit(T, E)
naf.plot(figsize=(8,4))
```
However, when we try a parametric model, we will see that it won’t extrapolate very well. Let’s use the flexible two parameter LogLogisticFitter model.

```python
from lifelines import LogLogisticFitter

fig, ax = plt.subplots(nrows=2, ncols=2, figsize=(10, 6))
t = np.linspace(0, 40)
llf = LogLogisticFitter().fit(T, E, timeline=t)
llf.plot_survival_function(ax=ax[0][0])
kmf.plot(ax=ax[0][0])
llf.plot_cumulative_hazard(ax=ax[0][1])
naf.plot(ax=ax[0][1])
t = np.linspace(0, 100)
llf = LogLogisticFitter().fit(T, E, timeline=t)
llf.plot_survival_function(ax=ax[1][0])
kmf.plot(ax=ax[1][0])
llf.plot_cumulative_hazard(ax=ax[1][1])
naf.plot(ax=ax[1][1])
```

1.5. Time-lagged conversion rates and cure models
The LogLogistic model does a quite terrible job of capturing not only the asymptotes, but also the intermediate values as well. If we extended the survival function out further, we would see that it eventually nears 0.

### 1.5.1 Custom parametric models to handle asymptotes

Focusing on modeling the cumulative hazard function, what we would like is a function that increases up to a limit and then tapers off to an asymptote. We can think long and hard about these (I did), but there’s a family of functions that have this property that we are very familiar with: cumulative distribution functions! By their nature, they will asymptotically approach 1. And, they are readily available in the SciPy and autograd libraries. So our new model of the cumulative hazard function is:

$$H(t; c, \theta) = cF(t; \theta)$$

where $c$ is the (unknown) horizontal asymptote, and $\theta$ is a vector of (unknown) parameters for the CDF, $F$.

We can create a custom cumulative hazard model using `ParametericUnivariateFitter` (for a tutorial on how to create custom models, see [this here](#)). Let’s choose the Normal CDF for our model.

Remember we must use the imports from `autograd` for this, i.e. `from autograd.scipy.stats import norm`.

```python
from autograd.scipy.stats import norm
from lifelines.fitters import ParametericUnivariateFitter

class UpperAsymptoteFitter(ParametericUnivariateFitter):
    _fitted_parameter_names = ["c_", "mu_", "sigma_"]
    _bounds = ((0, None), (None, None), (0, None))
```

(continues on next page)
def _cumulative_hazard(self, params, times):
    c, mu, sigma = params
    return c * norm.cdf((times - mu) / sigma, loc=0, scale=1)

[7]: uaf = UpperAsymptoteFitter().fit(T, E)
uaf.print_summary(3)
uaf.plot(figsize=(8,4))

<lifelines.UpperAsymptoteFitter: fitted with 200 observations, 118 censored>
number of subjects = 200
number of events = 82
log-likelihood = -380.473
hypothesis = c_ != 1, mu_ != 0, sigma_ != 1

--

    coef  se(coef)  lower 0.95  upper 0.95  p  -log2(p)
  c_   0.528     0.059    0.413    0.644 <0.0005  49.413
  mu_  16.678     0.542   15.615   17.740 <0.0005  688.353
  sigma_  4.833     0.379    4.090    5.575 <0.0005  77.451

[7]: <matplotlib.axes._subplots.AxesSubplot at 0x116a916a0>

We get a lovely asymptotical cumulative hazard. The summary table suggests that the best value of \(c\) is 0.586. This can be translated into the survival function asymptote by \(\exp(-0.586) \approx 0.56\).

Let’s compare this fit to the non-parametric models.

[8]: fig, ax = plt.subplots(nrows=2, ncols=2, figsize=(10, 6))
t = np.linspace(0, 40)
uaf = UpperAsymptoteFitter().fit(T, E, timeline=t)
uaf.plot_survival_function(ax=ax[0][0])
kmf.plot(ax=ax[0][0])
uaf.plot_cumulative_hazard(ax=ax[0][1])
naf.plot(ax=ax[0][1])
t = np.linspace(0, 100)
uaf = UpperAsymptoteFitter().fit(T, E, timeline=t)
uaf.plot_survival_function(ax=ax[1][0])
kmf.survival_function_.plot(ax=ax[1][0])

uaf.plot_cumulative_hazard(ax=ax[1][1])
naf.plot(ax=ax[1][1])

I wasn’t expect this good of a fit. But there it is. This was some artificial data, but let’s try this technique on some real life data.

from lifelines.datasets import load_leukemia, load_kidney_transplant
T, E = load_leukemia()['t'], load_leukemia()['status']
uaf.fit(T, E)
ax = uaf.plot_survival_function(figsize=(8,4))
uaf.print_summary()

kmf.fit(T, E).plot(ax=ax, ci_show=False)
print("---")
print("Estimated lower bound: {:.2f} ({:.2f}, {:.2f}).")
np.exp(-uaf.summary.loc['c_', 'coef']),
np.exp(-uaf.summary.loc['c_', 'upper 0.95']),
np.exp(-uaf.summary.loc['c_', 'lower 0.95']),
)

<lifelines.UpperAsymptoteFitter: fitted with 42 observations, 12 censored>
number of subjects = 42
number of events = 30
log-likelihood = -118.601
So we might expect that about 20% will not have the even in this population (but make note of the large CI bounds too!)

[11]: # Another, less obvious, dataset.

```python
T, E = load_kidney_transplant()['time'], load_kidney_transplant()['death']
uaf.fit(T, E)
ax = uaf.plot_survival_function(figsize=(8,4))
uaf.print_summary()
kmf.fit(T, E).plot(ax=ax)
print('Estimated lower bound: {:.2f} ({:.2f}, {:.2f})'.format(
    np.exp(-uaf.summary.loc['c_', 'coef']), np.exp(-uaf.summary.loc['c_', 'upper 0.95']), np.exp(-uaf.summary.loc['c_', 'lower 0.95']))
)
```

1.5. Time-lagged conversion rates and cure models

73
Using alternative functional forms

An even simpler model might look like \( c \left( 1 - \frac{1}{\lambda t + 1} \right) \), however this model cannot handle any “inflection points” like our artificial dataset has above. However, it works well for this Lung dataset.

With all cure models, one important feature is the ability to extrapolate to unseen times.

[12]: from autograd.scipy.stats import norm
    from lifelines.fitters import ParametericUnivariateFitter

    class SimpleUpperAsymptoteFitter(ParametericUnivariateFitter):
        _fitted_parameter_names = ['c_', 'lambda_']
        _bounds = ((0, None), (0, None))

        def _cumulative_hazard(self, params, times):
            c, lambda_ = params
            return c * (1 - (1 / (lambda_ * times + 1)))

    [13]: # Another, less obvious, dataset.

    saf = SimpleUpperAsymptoteFitter().fit(T, E, timeline=np.arange(1, 10000))
    ax = saf.plot_survival_function(figsize=(8,4))
    saf.print_summary(4)
    kmf.fit(T, E).plot(ax=ax)
    print("---")
    print("Estimated lower bound: \{:2f\} \([{:2f}, {:2f}]\)\).format(
        np.exp(-saf.summary.loc['c_', 'coef']),
        np.exp(-saf.summary.loc['c_', 'upper 0.95']),
        np.exp(-saf.summary.loc['c_', 'lower 0.95'])),

(continues on next page)
1.6 Survival regression

Often we have additional data aside from the duration that we want to use. The technique is called survival regression – the name implies we regress covariates (e.g., age, country, etc.) against another variable – in this case durations. Similar to the logic in the first part of this tutorial, we cannot use traditional methods like linear regression because of censoring.

There are a few popular models in survival regression: Cox’s model, accelerated failure models, and Aalen’s additive model. All models attempt to represent the hazard rate $h(t|x)$ as a function of $t$ and some covariates $x$. We explore these models next.
1.6.1 The dataset for regression

The dataset required for survival regression must be in the format of a Pandas DataFrame. Each row of the DataFrame should be an observation. There should be a column denoting the durations of the observations. There may be a column denoting the event status of each observation (1 if event occurred, 0 if censored). There are also the additional covariates you wish to regress against. Optionally, there could be columns in the DataFrame that are used for stratification, weights, and clusters which will be discussed later in this tutorial.

An example dataset we will use is the Rossi recidivism dataset, available in lifelines as datasets.load_rossi.

```python
from lifelines.datasets import load_rossi
rossi = load_rossi()
```

```
<table>
<thead>
<tr>
<th>week</th>
<th>arrest</th>
<th>fin</th>
<th>age</th>
<th>race</th>
<th>wexp</th>
<th>mar</th>
<th>paro</th>
<th>prio</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>20</td>
<td>1</td>
<td>0</td>
<td>27</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>1</td>
<td>17</td>
<td>1</td>
<td>0</td>
<td>18</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>25</td>
<td>0</td>
<td>1</td>
<td>19</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>13</td>
</tr>
<tr>
<td>3</td>
<td>52</td>
<td>0</td>
<td>1</td>
<td>23</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
```

The dataframe rossi contains 432 observations. The week column is the duration, the arrest column is the event occurred, and the other columns represent variables we wish to regress against.

If you need to first clean or transform your dataset (encode categorical variables, add interaction terms, etc.), that should happen before using lifelines. Libraries like Pandas and Patsy help with that.

1.6.2 Cox’s proportional hazard model

lifelines has an implementation of the Cox proportional hazards regression model (implemented in R as coxph). The idea behind the model is that the log-hazard of an individual is a linear function of their static covariates and a population-level baseline hazard that changes over time. Mathematically:

\[
\frac{h(t|x)}{b_0(t)} = \exp\left(\sum_{i=1}^{n} b_i(x_i - \bar{x}_i)\right)
\]

Note a few facts about this model: the only time component is in the baseline hazard, \(b_0(t)\). In the above product, the partial hazard is a time-invariant scalar factor that only increases or decreases the baseline hazard. Thus a changes in covariates will only increase or decrease the baseline hazard.

**Note:** In other regression models, a column of 1s might be added that represents that intercept or baseline. This is not necessary in the Cox model. In fact, there is no intercept in the additive Cox model - the baseline hazard represents this. lifelines will will throw warnings and may experience convergence errors if a column of 1s is present in your dataset.

**Running the regression**

The implementation of the Cox model in lifelines is called CoxPHFitter. Like R, it has a print_summary function that prints a tabular view of coefficients and related stats.
from lifelines import CoxPHFitter
from lifelines.datasets import load_rossi

rossi_dataset = load_rossi()

cph = CoxPHFitter()
cph.fit(rossi_dataset, duration_col='week', event_col='arrest', show_progress=True)
cph.print_summary()  # access the results using cph.summary

###
<lifelines.CoxPHFitter: fitted with 432 observations, 318 censored>

duration col = 'week'
event col = 'arrest'
number of subjects = 432
number of events = 114
log-likelihood = -658.75
time fit was run = 2019-01-27 23:10:15 UTC

---

<table>
<thead>
<tr>
<th>coef</th>
<th>exp(coef)</th>
<th>se(coef)</th>
<th>z</th>
<th>p</th>
<th>-log2(p)</th>
<th>lower 0.95</th>
<th>upper 0.95</th>
</tr>
</thead>
<tbody>
<tr>
<td>fin</td>
<td>-0.38</td>
<td>0.68</td>
<td>0.19</td>
<td>1.98</td>
<td>0.05</td>
<td>-0.75</td>
<td>-0.00</td>
</tr>
<tr>
<td>age</td>
<td>-0.06</td>
<td>0.94</td>
<td>0.02</td>
<td>2.61</td>
<td>0.01</td>
<td>0.79</td>
<td>-0.10</td>
</tr>
<tr>
<td>race</td>
<td>0.31</td>
<td>1.37</td>
<td>0.31</td>
<td>1.02</td>
<td>0.31</td>
<td>1.70</td>
<td>-0.29</td>
</tr>
<tr>
<td>wexp</td>
<td>-0.15</td>
<td>0.86</td>
<td>0.21</td>
<td>-0.71</td>
<td>0.48</td>
<td>1.06</td>
<td>-0.57</td>
</tr>
<tr>
<td>mar</td>
<td>-0.43</td>
<td>0.65</td>
<td>0.38</td>
<td>1.14</td>
<td>0.26</td>
<td>1.97</td>
<td>-1.18</td>
</tr>
<tr>
<td>paro</td>
<td>-0.08</td>
<td>0.92</td>
<td>0.20</td>
<td>-0.43</td>
<td>0.66</td>
<td>0.59</td>
<td>-0.47</td>
</tr>
<tr>
<td>prio</td>
<td>0.09</td>
<td>1.10</td>
<td>0.03</td>
<td>3.19</td>
<td>&lt;0.005</td>
<td>9.48</td>
<td>0.04</td>
</tr>
</tbody>
</table>

---

Concordance = 0.64
Likelihood ratio test = 33.27 on 7 df, -log2(p)=15.37

###

To access the coefficients and the baseline hazard directly, you can use cph.hazards_ and cph.
baseline_hazard_ respectively. Taking a look at these coefficients for a moment, prio (the number of prior
arrests) has a coefficient of about 0.09. Thus, a one unit increase in prio means the the baseline hazard will increase
by a factor of $\exp(0.09) = 1.10$ - about a 10% increase. Recall, in the Cox proportional hazard model, a higher hazard
means more at risk of the event occurring. The value $\exp(0.09)$ is called the hazard ratio, a name that will be clear
with another example.

Consider the coefficient of mar (whether the subject is married or not). The values in the column are binary: 0 or 1,
representing either not married or married. The value of the coefficient associated with mar, $\exp(-0.43)$, is the value
of ratio of hazards associated with being married, that is:

$$
\exp(-0.43) = \frac{\text{hazard of married subjects at time } t}{\text{hazard of unmarried subjects at time } t}
$$

Note that left-hand side is a constant (specifically, it’s independent of time, $t$), but the right-hand side has two factors
that can vary with time. The proportional assumption is that this is true in reality. That is, hazards can change over
time, but their ratio between levels remains a constant. Later we will deal with checking this assumption.

**Convergence**

Fitting the Cox model to the data involves using iterative methods. lifelines takes extra effort to help with convergence, so please be attentive to any warnings that appear. Fixing any warnings will generally help convergence and decrease the number of iterative steps required. If you wish to see the fitting, there is a show_progress parameter in
CoxPHFitter.fit function. For further help, see Problems with convergence in the Cox proportional hazard model.

After fitting, the value of the maximum log-likelihood is available using `cph._log_likelihood`. The variance matrix of the coefficients is available under `cph.variance_matrix_`.

**Goodness of fit**

After fitting, you may want to know how “good” of a fit your model was to the data. A few methods the author has found useful is to

- look at the concordance-index (see below section on Model selection in survival regression), available as `cph.score_` or in the `print_summary` as a measure of predictive accuracy.
- look at the log-likelihood test result in the `print_summary`
- check the proportional hazards assumption with the `check_assumptions` method. See section later on this page for more details.

**Prediction**

After fitting, you can use the suite of prediction methods: `.predict_partial_hazard`, `.predict_survival_function`, etc.

```python
X = rossi_dataset.drop(['week', 'arrest'], axis=1)
cph.predict_partial_hazard(X)
cph.predict_survival_function(X, times=[5., 25., 50.])
cph.predict_median(X)
```

A common use case is to predict the event time of censored subjects. This is easy to do, but we first have to calculate an important conditional probability. Let $T$ be the (random) event time for some subject, and $S(t)P(T > t)$ be their survival function. We are interested to know What is the new survival function, given I know the subject has lived past time $s$, where $s < t$? Mathematically:

$$P(T > t \mid T > s) = \frac{P(T > t \text{ and } T > s)}{P(T > s)}$$

$$= \frac{P(T > t)}{P(T > s)}$$

$$= \frac{S(t)}{S(s)}$$

Thus we scale the original survival function by the survival function at time $s$ (everything prior to $s$ should be mapped to 1.0 as well, since we are working with probabilities and we know that the subject was alive before $s$).

Back to our original problem of predicting the event time of censored individuals, we do the same thing:

```python
from lifelines import CoxPHFitter
from lifelines.datasets import load_regression_dataset
df = load_regression_dataset()
cph = CoxPHFitter().fit(df, 'T', 'E')
```
censored_subjects = df.loc[df['E'] == 0]

unconditioned_sf = cph.predict_survival_function(censored_subjects)

conditioned_sf = unconditioned_sf.apply(lambda c: (c / c.loc[df.loc[c.name, 'T']]).clip_upper(1))

# let's focus on a single subject
subject = 13
unconditioned_sf[subject].plot(ls='--', color='#A60628', label="unconditioned")
conditioned_sf[subject].plot(color='#A60628', label="conditioned on $T>10$")
plt.legend()

From here, you can pick a median or percentile as a best guess as to the subject’s event time:

```python
from lifelines.utils import median_survival_times, qth_survival_times
predictions_50 = median_survival_times(conditioned_sf)
predictions_75 = qth_survival_times(0.75, conditioned_sf)

# plotting subject 13 again
plt.hlines([0.5, 0.75], 0, 23, alpha=0.5, label="percentiles")
```

(continues on next page)
Plotting the coefficients

With a fitted model, an alternative way to view the coefficients and their ranges is to use the `plot` method.

```python
from lifelines.datasets import load_rossi
from lifelines import CoxPHFitter

rossi_dataset = load_rossi()
cph = CoxPHFitter()
cph.fit(rossi_dataset, duration_col='week', event_col='arrest', show_progress=True)
cph.plot()
```
Plotting the effect of varying a covariate

After fitting, we can plot what the survival curves look like as we vary a single covariate while holding everything else equal. This is useful to understand the impact of a covariate, *given the model*. To do this, we use the `plot_covariate_groups` method and give it the covariate of interest, and the values to display.

```python
from lifelines.datasets import load_rossi
from lifelines import CoxPHFitter

rossi_dataset = load_rossi()
cph = CoxPHFitter()
cph.fit(rossi_dataset, duration_col='week', event_col='arrest', show_progress=True)

cph.plot_covariate_groups('prio', [0, 2, 4, 6, 8, 10], cmap='coolwarm')
```
The `plot_covariate_groups` method can accept multiple covariates as well. This is useful for two purposes:

1. There are derivative features in your dataset. For example, suppose you have included `year` and `year**2` in your dataset. It doesn’t make sense to just vary `year` and leave `year**2` fixed. You’ll need to specify manually the values the covariates take on in a N-d array or list (where N is the number of covariates being varied.)

   ```python
   cph.plot_covariate_groups(['year', 'year**2'],
   [0, 0],
   [1, 1],
   [2, 4],
   [3, 9],
   [8, 64],
   cmap='coolwarm')
   ```

2. This feature is also useful for analyzing categorical variables. In your regression, you may have dummy variables (also called one-hot-encoded variables) in your DataFrame that represent some categorical variable. To simultaneously plot the survival curves of each category, all else being equal, we can use:

   ```python
   cph.plot_covariate_groups(['d1', 'd2' 'd3', 'd4', 'd5'],
   np.eye(5),
   cmap='coolwarm')
   ```

   The reason why we use `np.eye` is because we want each row of the matrix to “turn on” one category and “turn off” the others.

### Checking the proportional hazards assumption

CoxPHFitter has a `check_assumptions` method that will output violations of the proportional hazard assumption. For a tutorial on how to fix violations, see Testing the Proportional Hazard Assumptions.

Non-proportional hazards is a case of model misspecification. Suggestions are to look for ways to stratify a column (see docs below), or use a time varying model.

### Stratification

Sometimes one or more covariates may not obey the proportional hazard assumption. In this case, we can allow the covariate(s) to still be including in the model without estimating its effect. This is called stratification. At a high level, think of it as splitting the dataset into N smaller datasets, defined by the unique values of the stratifying covariate(s). Each dataset has its own baseline hazard (the non-parametric part of the model), but they all share the regression parameters (the parametric part of the model). Since covariates are the same within each dataset, there is no regression parameter for the covariates stratified on, hence they will not show up in the output. However there will be N baseline hazards under `baseline_cumulative_hazard_`.

To specify variables to be used in stratification, we define them in the call to `fit`:

```python
from lifelines.datasets import load_rossi
from lifelines import CoxPHFitter

rossi_dataset = load_rossi()
cph = CoxPHFitter()
(continues on next page)
```
cph.fit(rossi_dataset, 'week', event_col='arrest', strata=['race'], show_progress=True)

cph.print_summary()  # access the results using cph.summary

```python
< lifelines.CoxPHFitter: fitted with 432 observations, 318 censored >
  duration col = 'week'
  event col = 'arrest'
  strata = ['race']
number of subjects = 432
number of events = 114
log-likelihood = -620.56
time fit was run = 2019-01-27 23:08:35 UTC

---

<table>
<thead>
<tr>
<th>coef</th>
<th>exp(coef)</th>
<th>se(coef)</th>
<th>z</th>
<th>p</th>
<th>-log2(p)</th>
<th>lower 0.95</th>
<th>upper 0.95</th>
</tr>
</thead>
<tbody>
<tr>
<td>fin</td>
<td>-0.38</td>
<td>0.68</td>
<td>-1.98</td>
<td>0.05</td>
<td>4.39</td>
<td>-0.75</td>
<td>-0.00</td>
</tr>
<tr>
<td>age</td>
<td>-0.06</td>
<td>0.94</td>
<td>0.02</td>
<td>2.62</td>
<td>0.01</td>
<td>6.83</td>
<td>0.10</td>
</tr>
<tr>
<td>wexp</td>
<td>-0.14</td>
<td>0.87</td>
<td>0.21</td>
<td>-0.67</td>
<td>0.50</td>
<td>0.99</td>
<td>0.04</td>
</tr>
<tr>
<td>mar</td>
<td>-0.44</td>
<td>0.64</td>
<td>0.38</td>
<td>-1.15</td>
<td>0.25</td>
<td>2.00</td>
<td>-1.19</td>
</tr>
<tr>
<td>paro</td>
<td>-0.09</td>
<td>0.92</td>
<td>0.20</td>
<td>-0.44</td>
<td>0.66</td>
<td>0.60</td>
<td>0.47</td>
</tr>
<tr>
<td>prio</td>
<td>0.09</td>
<td>1.10</td>
<td>0.03</td>
<td>3.21</td>
<td>&lt;0.005</td>
<td>9.56</td>
<td>0.04</td>
</tr>
</tbody>
</table>
---

Concordance = 0.64
Likelihood ratio test = 109.63 on 6 df, -log2(p)=68.48

```

cph.baseline_cumulative_hazard_.shape
# (49, 2)

Weights & robust errors

Observations can come with weights, as well. These weights may be integer values representing some commonly occurring observation, or they may be float values representing some sampling weights (ex: inverse probability weights). In the `CoxPHFitter.fit` method, an kwarg is present for specifying which column in the DataFrame should be used as weights, ex: `CoxPHFitter(df, 'T', 'E', weights_col='weights')`.

When using sampling weights, it’s correct to also change the standard error calculations. That is done by turning on the `robust` flag in `fit`. Internally, `CoxPHFitter` will use the sandwich estimator to compute the errors.

```python
from lifelines import CoxPHFitter
df = pd.DataFrame({
    'T': [5, 3, 9, 8, 7, 4, 4, 3, 2, 5, 6, 7],
    'E': [1, 1, 1, 1, 1, 0, 1, 1, 1, 0, 1, 0],
    'weights': [1.1, 0.5, 2.0, 1.6, 1.2, 4.3, 1.4, 4.5, 3.0, 3.2, 0.4, 6.2],
    'month': [10, 3, 9, 8, 7, 4, 4, 3, 2, 5, 6, 7],
    'age': [4, 3, 9, 8, 7, 4, 4, 3, 2, 5, 6, 7],
})
cph = CoxPHFitter()
cph.fit(df, 'T', 'E', weights_col='weights', robust=True)
cph.print_summary()
```

See more examples in Adding weights to observations in a Cox model.
Clusters & correlations

Another property your dataset may have is groups of related subjects. This could be caused by:

- a single individual having multiple occurrences, and hence showing up in the dataset more than once.
- subjects that share some common property, like members of the same family or being matched on propensity scores.

We call these grouped subjects “clusters”, and assume they are designated by some column in the DataFrame (example below). When using cluster, the point estimates of the model don’t change, but the standard errors will increase. An intuitive argument for this is that 100 observations on 100 individuals provide more information than 100 observations on 10 individuals (or clusters).

```python
from lifelines import CoxPHFitter
def = pd.DataFrame({
    'T': [5, 3, 9, 8, 7, 4, 4, 3, 2, 5, 6, 7],
    'E': [1, 1, 1, 1, 1, 0, 1, 1, 1, 0, 1, 0],
    'month': [10, 3, 9, 8, 7, 4, 4, 3, 2, 5, 6, 7],
    'age': [4, 3, 9, 8, 7, 4, 4, 3, 2, 5, 6, 7],
    'id': [1, 1, 1, 1, 2, 3, 3, 4, 4, 5, 6, 7]
})
cph = CoxPHFitter()
cph.fit(def, 'T', 'E', cluster_col='id')
cph.print_summary()
```

For more examples, see Correlations between subjects in a Cox model.

Residuals

After fitting a Cox model, we can look back and compute important model residuals. These residuals can tell us about non-linearities not captured, violations of proportional hazards, and help us answer other useful modeling questions. See Assessing Cox model fit using residuals.

1.6.3 Accelerated failure time models

Suppose we have two populations, A and B, with different survival functions, \( S_A(t) \) and \( S_B(t) \), and they are related by some accelerated failure rate, \( \lambda \):

\[
S_A(t) = S_B\left(\frac{t}{\lambda}\right)
\]

This can be interpreted as slowing down or speeding up moving along the survival function. A classic example of this is that dogs age at 7 times the rate of humans, i.e. \( \lambda = \frac{1}{7} \). This model has some other nice properties: the average survival time of population B is \( \lambda \) times the average survival time of population A. Likewise with the median survival time.

More generally, we can model the \( \lambda \) as a function of covariates available, that is:

\[
S_A(t) = S_B\left(\frac{t}{\lambda(x)}\right)
\]

\[
\lambda(x) = \exp\left(b_0 + \sum_{i=1}^{n} b_i x_i\right)
\]
This model can accelerate or decelerate failure times depending on subjects’ covariates. Another nice feature of this is the ease of interpretation of the coefficients: a unit increase in \( x_i \) means the average/median survival time changes by a factor of \( \exp(b_i) \).

**Note:** An important note on interpretation: Suppose \( b_i \) was positive, then the factor \( \exp(b_i) \) is greater than 1, which will decelerate the event time since we divide time by the factor <=> increase mean/median survival. Hence, it will be a **protective effect**. Likewise, a negative \( b_i \) will hasten the event time <=> reduce the mean/median survival time. This interpretation is opposite of how the sign influences event times in the Cox model! This is standard survival analysis convention.

Next, we pick a parametric form for the survival function, \( S(t) \). The most common is the Weibull form. So if we assume the relationship above and a Weibull form, our hazard function is quite easy to write down:

\[
H(t; x) = \left( \frac{t}{\lambda(x)} \right)^\rho
\]

We call these accelerated failure time models, shortened often to just AFT models. Using *lifelines*, we can fit this model (and the unknown \( \rho \) parameter too).

**The Weibull AFT model**

The API for the Weibull AFT model is similar to the other regression models in *lifelines*. After fitting, the coefficients can be accessed using `.params_` or `.summary`, or alternatively printed using `.print_summary()`.

```python
from lifelines import WeibullAFTFitter
from lifelines.datasets import load_rossi

rossi_dataset = load_rossi()

aft = WeibullAFTFitter()
aft.fit(rossi_dataset, duration_col='week', event_col='arrest')
aft.print_summary(3)  # access the results using aft.summary
```

```
<lifelines.WeibullAFTFitter: fitted with 432 observations, 318 censored>

duration col = 'week'
event col = 'arrest'
number of subjects = 432
number of events = 114
log-likelihood = -679.917
time fit was run = 2019-02-20 17:47:19 UTC

---

<table>
<thead>
<tr>
<th>coef</th>
<th>exp(coef)</th>
<th>se(coef)</th>
<th>z</th>
<th>p</th>
<th>-log2(p)</th>
<th>lower 0.95</th>
<th>upper 0.95</th>
</tr>
</thead>
<tbody>
<tr>
<td>lambda_ fin</td>
<td>0.272</td>
<td>1.313</td>
<td>0.138</td>
<td>1.973</td>
<td>0.049</td>
<td>4.365</td>
<td>0.002</td>
</tr>
<tr>
<td></td>
<td>0.543</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>age</td>
<td>0.041</td>
<td>1.042</td>
<td>0.016</td>
<td>2.544</td>
<td>0.011</td>
<td>6.512</td>
<td>0.009</td>
</tr>
<tr>
<td></td>
<td>0.072</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>race</td>
<td>-0.225</td>
<td>0.799</td>
<td>0.220</td>
<td>-1.021</td>
<td>0.307</td>
<td>1.703</td>
<td>-0.656</td>
</tr>
<tr>
<td></td>
<td>0.207</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>wexp</td>
<td>0.107</td>
<td>1.112</td>
<td>0.152</td>
<td>0.703</td>
<td>0.482</td>
<td>1.053</td>
<td>-0.190</td>
</tr>
<tr>
<td></td>
<td>0.404</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>mar</td>
<td>0.311</td>
<td>1.365</td>
<td>0.273</td>
<td>1.139</td>
<td>0.255</td>
<td>1.973</td>
<td>-0.224</td>
</tr>
<tr>
<td></td>
<td>0.847</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

(continues on next page)
From above, we can see that `prio`, which is the number of previous incarcerations, has a large negative coefficient. This means that each addition incarceration changes a subject’s mean/median survival time by \( \exp(-0.066) = 0.936 \), approximately a 7% decrease in mean/median survival time. What is the mean/median survival time?

```python
def print(aft.median_survival_time_

# 100.325
# 118.67
```

What does the `rho__intercept` row mean in the above table? Internally, we model the log of the `rho_` parameter, so the value of \( \rho \) is the exponential of the value, so in case above it’s \( \hat{\rho} = \exp 0.339 = 1.404 \). This brings us to the next point - modelling \( \rho \) with covariates as well:

**Modeling ancillary parameters**

In the above model, we left the parameter \( \rho \) as a single unknown. We can also choose to model this parameter as well. Why might we want to do this? It can help in survival prediction to allow heterogeneity in the \( \rho \) parameter. The model is no longer an AFT model, but we can still recover and understand the influence of changing a covariate by looking at its outcome plot (see section below). To model \( \rho \), we use the `ancillary_df` keyword argument in the call to `fit`. There are four valid options:

1. **False** or **None**: explicitly do not model the `rho_` parameter (except for its intercept).
2. a Pandas DataFrame. This option will use the columns in the Pandas DataFrame as the covariates in the regression for `rho_`. This DataFrame could be a equal to, or a subset of, the original dataset using for modeling `lambda_`, or it could be a totally different dataset.
3. **True**. Passing in True will internally reuse the dataset that is being used to model `lambda_`.

```python
aft = WeibullAFTFitter()
aft.fit(rossi, duration_col='week', event_col='arrest', ancillary_df=False)
# identical to aft.fit(rossi, duration_col='week', event_col='arrest', ancillary_
→df=None)
```

```python
aft.fit(rossi, duration_col='week', event_col='arrest', ancillary_df=some_df)
```

```python
aft.fit(rossi, duration_col='week', event_col='arrest', ancillary_df=True)
# identical to aft.fit(rossi, duration_col='week', event_col='arrest', ancillary_
→df=rossi)
```
aft.print_summary()

""
<lifelines.WeibullAFTFitter: fitted with 432 observations, 318 censored>
duration col = 'week'
event col = 'arrest'
number of subjects = 432
number of events = 114
log-likelihood = -669.40
time fit was run = 2019-02-20 17:42:55 UTC
---

<table>
<thead>
<tr>
<th>coef</th>
<th>exp(coef)</th>
<th>se(coef)</th>
<th>z</th>
<th>p</th>
<th>-log2(p)</th>
<th>lower 0.95</th>
<th>upper 0.95</th>
</tr>
</thead>
<tbody>
<tr>
<td>lambda_fin</td>
<td>0.24</td>
<td>1.28</td>
<td>0.15</td>
<td>1.60</td>
<td>0.11</td>
<td>3.18</td>
<td>-0.06</td>
</tr>
<tr>
<td>age</td>
<td>0.10</td>
<td>1.10</td>
<td>0.03</td>
<td>3.43</td>
<td>&lt;0.005</td>
<td>10.69</td>
<td>0.04</td>
</tr>
<tr>
<td>race</td>
<td>0.07</td>
<td>1.07</td>
<td>0.19</td>
<td>0.36</td>
<td>0.72</td>
<td>0.48</td>
<td>-0.30</td>
</tr>
<tr>
<td>wexp</td>
<td>-0.34</td>
<td>0.71</td>
<td>0.15</td>
<td>-2.22</td>
<td>0.03</td>
<td>5.26</td>
<td>-0.64</td>
</tr>
<tr>
<td>mar</td>
<td>0.26</td>
<td>1.30</td>
<td>0.30</td>
<td>0.86</td>
<td>0.39</td>
<td>1.35</td>
<td>-0.33</td>
</tr>
<tr>
<td>paro</td>
<td>0.09</td>
<td>1.10</td>
<td>0.15</td>
<td>0.61</td>
<td>0.54</td>
<td>0.88</td>
<td>-0.21</td>
</tr>
<tr>
<td>prio</td>
<td>-0.08</td>
<td>0.92</td>
<td>0.02</td>
<td>-4.24</td>
<td>&lt;0.005</td>
<td>15.46</td>
<td>-0.12</td>
</tr>
<tr>
<td>_intercept</td>
<td>2.68</td>
<td>14.65</td>
<td>0.60</td>
<td>4.50</td>
<td>&lt;0.005</td>
<td>17.14</td>
<td>1.51</td>
</tr>
<tr>
<td>rho_fin</td>
<td>-0.01</td>
<td>0.99</td>
<td>0.15</td>
<td>-0.09</td>
<td>0.92</td>
<td>1.11</td>
<td>-0.31</td>
</tr>
<tr>
<td>age</td>
<td>-0.05</td>
<td>0.95</td>
<td>0.02</td>
<td>-3.10</td>
<td>&lt;0.005</td>
<td>9.01</td>
<td>-0.08</td>
</tr>
<tr>
<td>race</td>
<td>-0.46</td>
<td>0.63</td>
<td>0.25</td>
<td>-1.79</td>
<td>0.07</td>
<td>3.77</td>
<td>-0.95</td>
</tr>
<tr>
<td>wexp</td>
<td>0.56</td>
<td>1.74</td>
<td>0.17</td>
<td>3.32</td>
<td>&lt;0.005</td>
<td>10.13</td>
<td>0.23</td>
</tr>
<tr>
<td>mar</td>
<td>0.10</td>
<td>1.10</td>
<td>0.27</td>
<td>0.36</td>
<td>0.72</td>
<td>0.47</td>
<td>-0.44</td>
</tr>
<tr>
<td>paro</td>
<td>0.02</td>
<td>1.02</td>
<td>0.16</td>
<td>0.12</td>
<td>0.90</td>
<td>0.15</td>
<td>-0.29</td>
</tr>
<tr>
<td>prio</td>
<td>0.03</td>
<td>1.03</td>
<td>0.02</td>
<td>1.44</td>
<td>0.15</td>
<td>2.73</td>
<td>-0.01</td>
</tr>
<tr>
<td>_intercept</td>
<td>1.48</td>
<td>4.41</td>
<td>0.41</td>
<td>3.60</td>
<td>&lt;0.005</td>
<td>11.62</td>
<td>0.68</td>
</tr>
</tbody>
</table>
---
Concordance = 0.63
Log-likelihood ratio test = 54.45 on 14 df, -log2(p)=19.83
""

Plotting

The plotting API is the same as in CoxPHFitter. We can view all covariates in a forest plot:
```python
wft = WeibullAFTFitter().fit(rossi, 'week', 'arrest', ancillary_df=True)
wft.plot()
```
We can observe the influence a variable in the model by plotting the outcome (i.e. survival) of changing the variable. This is done using `plot_covariate_groups`, and this is also a nice time to observe the effects of modeling rho vs keeping it fixed. Below we fit the Weibull model to the same dataset twice, but in the first model we model rho and in the second model we don’t. We when vary the prior (which is the number of prior arrests) and observe how the survival changes.

```python
fig, ax = plt.subplots(nrows=1, ncols=2, figsize=(10, 4))

times = np.arange(0, 100)
wft_model_rho = WeibullAFTFitter().fit(rossi, 'week', 'arrest', ancillary_df=True,
                          timeline=times)
wft_model_rho.plot_covariate_groups('prio', range(0, 16, 3), cmap='coolwarm',
                          ax=ax[0])
ax[0].set_title("Modelling rho")

wft_not_model_rho = WeibullAFTFitter().fit(rossi, 'week', 'arrest', ancillary_df=False,
                           timeline=times)
wft_not_model_rho.plot_covariate_groups('prio', range(0, 16, 3), cmap='coolwarm',
                           ax=ax[1])
ax[1].set_title("Not modelling rho");
```
Comparing a few of these survival functions side by side:

```python
fig, ax = plt.subplots(nrows=1, ncols=1, figsize=(7, 4))

wft_model_rho.plot_covariate_groups('prio', range(0, 16, 5), cmap='coolwarm', ax=ax,
   lw=2, plot_baseline=False)

wft_not_model_rho.plot_covariate_groups('prio', range(0, 16, 5), cmap='coolwarm',
   ax=ax, ls='--', lw=2, plot_baseline=False)

ax.get_legend().remove()
```
You read more about and see other examples of the extensions to `plot_covariate_groups` *Plotting the effect of varying a covariate*.

**Prediction**

Given a new subject, we ask questions about their future survival? When are they likely to experience the event? What does their survival function look like? The `WeibullAFTFitter` is able to answer these. If we have modeled the ancillary covariates, we are required to include those as well:

```python
X = rossi.loc[:10]

aft.predict_cumulative_hazard(X, ancillary_X=X)

aft.predict_survival_function(X, ancillary_X=X)

aft.predict_median(X, ancillary_X=X)

aft.predict_percentile(X, ancillary_X=X)

aft.predict_expectation(X, ancillary_X=X)
```

There are two tunable parameters that can be used to to achieve a better test score. These are `penalizer` and `l1_ratio` in the call to `WeibullAFTFitter`. The penalizer is similar to scikit-learn’s `ElasticNet` model, see their docs.

```python
aft_with_elastic_penalty = WeibullAFTFitter(penalizer=4.0, l1_ratio=1.0)
aft_with_elastic_penalty.fit(rossi, 'week', 'arrest')
aft_with_elastic_penalty.predict_median(rossi)
```

```python
aft_with_elastic_penalty.print_summary()

```

```
<lifelines.WeibullAFTFitter: fitted with 432 observations, 318 censored>
  duration col = 'week'
  event col = 'arrest'
  penalizer = 4.0
  l1_ratio = 1.0
  number of subjects = 432
  number of events = 114
  log-likelihood = -2710.95
  time fit was run = 2019-02-20 19:53:29 UTC

---

    coef  exp(coef)  se(coef)     z     p  -log2(p) lower 0.95 upper 0.95
---

  lambda_fin 0.00      1.00    0.08  0.00  1.00     0.00    -0.15  0.15

    age 0.13      1.14    0.01 12.27 <0.005 112.47  0.11  0.15

    race 0.55      1.73    0.09  5.80 <0.005   27.16  0.36  0.73

    wexp 0.00      1.00    0.09  0.00  1.00     0.00   -0.17  0.17

    mar 0.00      1.00    0.14  0.01  0.99     0.01  -0.27  0.28

    paro 0.00      1.00    0.08  0.01  0.99     0.01  -0.16  0.16

    prio 0.00      1.00    0.01  0.00  1.00     0.00  -0.03  0.03

    _intercept 0.00      1.00    0.19  0.00  1.00     0.00  -0.38  0.38

(continues on next page)
```

1.6. Survival regression 95
The Log-Normal and Log-Logistic AFT model

There are also the LogNormalAFTFitter and LogLogisticAFTFitter models, which instead of assuming that the survival time distribution is Weibull, we assume it is Log-Normal or Log-Logistic, respectively. They have identical APIs to the WeibullAFTFitter, but the parameter names are different.

```python
from lifelines import LogLogisticAFTFitter
from lifelines import LogNormalAFTFitter
llf = LogLogisticAFTFitter().fit(rossi, 'week', 'arrest')
lnf = LogNormalAFTFitter().fit(rossi, 'week', 'arrest')
```

Model selection for AFT models

Often, you don’t know a priori which AFT model to use. Each model has some assumptions built-in (not implemented yet in lifelines), but a quick and effective method is to compare the log-likelihoods for each fitted model. (Technically, we are comparing the AIC, but the number of parameters for each model is the same, so we can simply and just look at the log-likelihood). Generally, given the same dataset and number of parameters, a better fitting model has a larger log-likelihood. We can look at the log-likelihood for each fitted model and select the largest one.

```python
from lifelines import LogLogisticAFTFitter, WeibullAFTFitter, LogNormalAFTFitter
from lifelines.datasets import load_rossi
rossi = load_rossi()
llf = LogLogisticAFTFitter().fit(rossi, 'week', 'arrest')
lnf = LogNormalAFTFitter().fit(rossi, 'week', 'arrest')
wf = WeibullAFTFitter().fit(rossi, 'week', 'arrest')
print(llf._log_likelihood)  # -679.938
print(lnf._log_likelihood)  # -683.234
print(wf._log_likelihood)  # -679.916, slightly the best model.

# with some heterogeneity in the ancillary parameters
ancillary_df = rossi[['prio']]
llf = LogLogisticAFTFitter().fit(rossi, 'week', 'arrest', ancillary_df=ancillary_df)
lnf = LogNormalAFTFitter().fit(rossi, 'week', 'arrest', ancillary_df=ancillary_df)
wf = WeibullAFTFitter().fit(rossi, 'week', 'arrest', ancillary_df=ancillary_df)
print(llf._log_likelihood)  # -678.94, slightly the best model.
print(lnf._log_likelihood)  # -680.39
print(wf._log_likelihood)  # -679.60
Left, right and interval censored data

The AFT models have APIs that handle left and interval censored data, too. The API for them is different than the API for fitting to right censored data. Here’s an example with interval censored data.

```python
from lifelines.datasets import load_diabetes

df = load_diabetes()
df['gender'] = df['gender'] == 'male

print(df.head())

""
left  right   gender
1     24      27    True
2     22      22    False
3     37      39    True
4     20      20    True
5     1       16    True
""

wf = WeibullAFTFitter().fit_interval_censoring(df, lower_bound_col='left', upper_bound_col='right')
wf.print_summary()

""
<lifelines.WeibullAFTFitter: fitted with 731 observations, 136 censored>
   event col = 'E'
   number of subjects = 731
   number of events = 595
   log-likelihood = -2027.20
   time fit was run = 2019-04-11 19:39:42 UTC

---
     coef  exp(coef)  se(coef)   z     p  -log2(p)  lower 0.95  upper 0.95
lambda_gender  0.05  1.05  0.03  1.66  0.10  3.38    -0.01   0.10
   _intercept  2.91  18.32  0.02 130.15 <0.005 inf    2.86   inf
   rho_ _intercept  1.04  2.83  0.03 36.91 <0.005 988.46   0.98  1.09
---
Log-likelihood ratio test = 2.74 on 1 df, -log2(p)=3.35
""
```

Another example of using lifelines for interval censored data is located here.

### 1.6.4 Aalen’s additive model

**Warning:** This implementation is still experimental.

Aalen’s Additive model is another regression model we can use. Like the Cox model, it defines the hazard rate, but instead of the linear model being multiplicative like the Cox model, the Aalen model is additive. Specifically:

\[
h(t|x) = b_0(t) + b_1(t)x_1 + ... + b_N(t)x_N
\]
Inference typically does not estimate the individual \( b_i(t) \) but instead estimates \( \int_0^t b_i(s) \, ds \) (similar to the estimate of the hazard rate using \texttt{NelsonAalenFitter}). This is important when interpreting plots produced.

For this exercise, we will use the regime dataset and include the categorical variables \texttt{un_continent_name} (eg: Asia, North America,...), the \texttt{regime} type (e.g., monarchy, civilian,...) and the year the regime started in, \texttt{start_year}. The estimator to fit unknown coefficients in Aalen’s additive model is located under \texttt{lifelines}. \texttt{AalenAdditiveFitter}.

\begin{verbatim}
from lifelines import AalenAdditiveFitter
from lifelines.datasets import load_dd

data = load_dd()
data.head()
\end{verbatim}

<table>
<thead>
<tr>
<th>ctry_name</th>
<th>cowd-code</th>
<th>poli-ty-code</th>
<th>un_region_name</th>
<th>un_continent_name</th>
<th>leaderspellreg</th>
<th>democracy</th>
<th>regime</th>
<th>start_year</th>
<th>duration</th>
<th>observed</th>
</tr>
</thead>
<tbody>
<tr>
<td>Afghanistan</td>
<td>700</td>
<td>Southern Asia</td>
<td>Asia</td>
<td>Mohammad Zahir Shah</td>
<td>Mohammad Zahir Shah.Afghanistan.1946</td>
<td>Non-democracy</td>
<td>Monarchy</td>
<td>1946</td>
<td>7</td>
<td>1</td>
</tr>
<tr>
<td>Afghanistan</td>
<td>700</td>
<td>Southern Asia</td>
<td>Asia</td>
<td>Sardar Mohammad Daoud</td>
<td>Sardar Mohammad Daoud.Afghanistan.1953</td>
<td>Non-democracy</td>
<td>Civilian Dict</td>
<td>1953</td>
<td>10</td>
<td>1</td>
</tr>
</tbody>
</table>

I’m using the lovely library \texttt{Patsy} here to create a design matrix from my original dataframe.

\begin{verbatim}
import patsy
X = patsy.dmatrix('un_continent_name + regime + start_year', data, return_type='dataframe')
X = X.rename(columns={'Intercept': 'baseline'})
print(X.columns.tolist())
\end{verbatim}

\begin{verbatim}
['baseline',
 'un_continent_name[T.Americas]',
 'un_continent_name[T.Asia]',
 'un_continent_name[T.Europe]',
 'un_continent_name[T.Oceania]',
 'regime[T.Military Dict]',
 'regime[T.Mixed Dem]',
 'regime[T.Monarchy]',

(continues on next page)
We have also included the `coef_penalizer` option. During the estimation, a linear regression is computed at each step. Often the regression can be unstable (due to high co-linearity or small sample sizes) – adding a penalizer term controls the stability. I recommend always starting with a small penalizer term – if the estimates still appear to be too unstable, try increasing it.

```python
aaf = AalenAdditiveFitter(coef_penalizer=1.0, fit_intercept=False)
```

An instance of `AalenAdditiveFitter` includes a `fit` method that performs the inference on the coefficients. This method accepts a pandas DataFrame: each row is an individual and columns are the covariates and two individual columns: a `duration` column and a boolean `event occurred` column (where event occurred refers to the event of interest - expulsion from government in this case).

```python
X['T'] = data['duration']
X['E'] = data['observed']
```

```python
aaf.fit(X, 'T', event_col='E')
```

After fitting, the instance exposes a `cumulative_hazards_` DataFrame containing the estimates of \( \int_0^t b_i(s) \, ds \):

```python
aaf.cumulative_hazards_.head()
```

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>-0.03447</td>
<td>-0.03173</td>
<td>0.06216</td>
<td>0.2058</td>
<td>-0.009559</td>
<td>0.07611</td>
<td>0.08729</td>
<td>-0.1362</td>
<td>0.04885</td>
<td>0.1285</td>
<td>0.000092</td>
</tr>
<tr>
<td>0.14278</td>
<td>0.02496</td>
<td>0.11122</td>
<td>0.2083</td>
<td>0.079042</td>
<td>0.11704</td>
<td>0.36254</td>
<td>-0.2293</td>
<td>0.17103</td>
<td>0.1238</td>
<td>0.000044</td>
</tr>
<tr>
<td>0.30154</td>
<td>0.07212</td>
<td>0.10929</td>
<td>0.1614</td>
<td>-0.063030</td>
<td>0.16553</td>
<td>0.68693</td>
<td>-0.2738</td>
<td>0.33300</td>
<td>0.1499</td>
<td>0.000004</td>
</tr>
<tr>
<td>0.37960</td>
<td>0.06853</td>
<td>0.15162</td>
<td>0.2609</td>
<td>0.185569</td>
<td>0.22695</td>
<td>0.95016</td>
<td>-0.2961</td>
<td>0.37351</td>
<td>0.4311</td>
<td>0.000032</td>
</tr>
<tr>
<td>0.36740</td>
<td>0.20201</td>
<td>0.21252</td>
<td>0.2429</td>
<td>0.188740</td>
<td>0.25127</td>
<td>1.15132</td>
<td>-0.3926</td>
<td>0.54952</td>
<td>0.7593</td>
<td>0.000000</td>
</tr>
</tbody>
</table>

`AalenAdditiveFitter` also has built in plotting:

```python
aaf.plot(columns=['regime[T.Presidential Dem]', 'baseline', 'un_continent_name[T.Europe]'], iloc=slice(1,15))
```
Regression is most interesting if we use it on data we have not yet seen, i.e., prediction! We can use what we have learned to predict individual hazard rates, survival functions, and median survival time. The dataset we are using is available up until 2008, so let's use this data to predict the duration of former Canadian Prime Minister Stephen Harper.

```python
ix = (data['ctryname'] == 'Canada') & (data['start_year'] == 2006)
harper = X.loc[ix]
print("Harper's unique data point:")
print(harper)
```

Harper's unique data point:

<table>
<thead>
<tr>
<th></th>
<th>baseline</th>
<th>un_continent_name[T.Americas]</th>
<th>un_continent_name[T.Asia]</th>
<th>...</th>
<th>start_year</th>
<th>T</th>
<th>E</th>
</tr>
</thead>
<tbody>
<tr>
<td>268</td>
<td>1.0</td>
<td>1.0</td>
<td>0.0</td>
<td>...</td>
<td>2006.</td>
<td>3</td>
<td>0</td>
</tr>
</tbody>
</table>

```python
ax = plt.subplot(2,1,1)
aaf.predict_cumulative_hazard(harper).plot(ax=ax)
ax = plt.subplot(2,1,2)
aaf.predict_survival_function(harper).plot(ax=ax);
```
Because of the nature of the model, estimated survival functions of individuals can increase. This is an expected artifact of Aalen’s additive model.

### 1.6.5 Model selection in survival regression

#### Parametric vs Semi-parametric models

Above, we’ve displayed two semi-parametric models (Cox model and Aalen’s model), and a family of parametric AFT models. Which should you choose? What are the advantages and disadvantages of either? I suggest reading the two following StackExchange answers to get a better idea of what experts think:

1. In survival analysis, why do we use semi-parametric models (Cox proportional hazards) instead of fully parametric models?
2. In survival analysis, when should we use fully parametric models over semi-parametric ones?

#### Model selection based on residuals

The sections Testing the Proportional Hazard Assumptions and Assessing Cox model fit using residuals may be useful for modelling your data better.
Model selection based on predictive power

If censoring is present, it’s not appropriate to use a loss function like mean-squared-error or mean-absolute-loss. Instead, one measure is the concordance-index, also known as the c-index. This measure evaluates the accuracy of the ordering of predicted time. It is in fact a generalization of AUC, another common loss function, and is interpreted similarly:

- 0.5 is the expected result from random predictions,
- 1.0 is perfect concordance and,
- 0.0 is perfect anti-concordance (multiply predictions with -1 to get 1.0)

Fitted survival models typically have a concordance index between 0.55 and 0.75 (this may seem bad, but even a perfect model has a lot of noise than can make a high score impossible). In lifelines, a fitted model’s concordance-index is present in the output of print_summary(), but also available under the score_ property. Generally, the measure is implemented in lifelines under lifelines.utils.concordance_index and accepts the actual times (along with any censored subjects) and the predicted times.

```python
from lifelines import CoxPHFitter
from lifelines.datasets import load_rossi

rossi = load_rossi()
cph = CoxPHFitter()
cph.fit(rossi, duration_col="week", event_col="arrest")

# Three ways to view the c-index:
# method one
cph.print_summary()

# method two
print(cph.score_)

# method three
from lifelines.utils import concordance_index
print(concordance_index(rossi['week'], -cph.predict_partial_hazard(rossi), rossi['arrest']))
```

However, there are other, arguably better, methods to measure the fit of a model. Included in print_summary is the log-likelihood, which can be used in an AIC calculation, and the log-likelihood ratio statistic. Generally, I personally loved this article by Frank Harrell, “Statistically Efficient Ways to Quantify Added Predictive Value of New Measurements”.

lifelines has an implementation of k-fold cross validation under lifelines.utils.k_fold_cross_validation. This function accepts an instance of a regression fitter (either CoxPHFitter of AalenAdditiveFitter), a dataset, plus k (the number of folds to perform, default 5). On each fold, it splits the data into a training set and a testing set fits itself on the training set and evaluates itself on the testing set (using the concordance measure by default).

```python
from lifelines import CoxPHFitter
from lifelines.datasets import load_regression_dataset
from lifelines.utils import k_fold_cross_validation

regression_dataset = load_regression_dataset()
cph = CoxPHFitter()
scores = k_fold_cross_validation(cph, regression_dataset, 'T', event_col='E', k=3)
print(scores)
```
1.7 Time varying survival regression

1.7.1 Cox’s time varying proportional hazard model

Often an individual will have a covariate change over time. An example of this is hospital patients who enter the study and, at some future time, may receive a heart transplant. We would like to know the effect of the transplant, but we cannot condition on whether they received the transplant naively. Consider that if patients needed to wait at least 1 year before getting a transplant, then everyone who dies before that year is considered as a non-transplant patient, and hence this would overestimate the hazard of not receiving a transplant.

We can incorporate changes over time into our survival analysis by using a modification of the Cox model above. The general mathematical description is:

$$ h(t|x) = b_0(t) \exp \left( \sum_{i=1}^{n} \beta_i (x_i(t) - x_i) \right) $$

Note the time-varying $x_i(t)$ to denote that covariates can change over time. This model is implemented in `lifelines` as `CoxTimeVaryingFitter`. The dataset schema required is different than previous models, so we will spend some time describing this.

**Dataset creation for time-varying regression**

`lifelines` requires that the dataset be in what is called the *long* format. This looks like one row per state change, including an ID, the left (exclusive) time point, and right (inclusive) time point. For example, the following dataset tracks three unique subjects.

<table>
<thead>
<tr>
<th>id</th>
<th>start</th>
<th>stop</th>
<th>group</th>
<th>z</th>
<th>event</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>8</td>
<td>1</td>
<td>0</td>
<td>False</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>5</td>
<td>0</td>
<td>0</td>
<td>False</td>
</tr>
<tr>
<td>2</td>
<td>5</td>
<td>8</td>
<td>0</td>
<td>1</td>
<td>True</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>3</td>
<td>1</td>
<td>0</td>
<td>False</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>12</td>
<td>1</td>
<td>1</td>
<td>True</td>
</tr>
</tbody>
</table>

In the above dataset, start and stop denote the boundaries, id is the unique identifier per subject, and event denotes if the subject died at the end of that period. For example, subject ID 2 had variable $z=0$ up to and including the end of time period 5 (we can think that measurements happen at end of the time period), after which it was set to 1. Since event is 1 in that row, we conclude that the subject died at time 8.

This desired dataset can be built up from smaller datasets. To do this we can use some helper functions provided in `lifelines`. Typically, data will be in a format that looks like it comes out of a relational database. You may have a “base” table with ids, durations alive, and a censored flag, and possibly static covariates. Ex:
We will perform a light transform to this dataset to modify it into the “long” format.

```python
from lifelines.utils import to_long_format

base_df = to_long_format(base_df, duration_col="duration")
```

The new dataset looks like:

<table>
<thead>
<tr>
<th>id</th>
<th>start</th>
<th>stop</th>
<th>var1</th>
<th>event</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>10</td>
<td>0.1</td>
<td>True</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>12</td>
<td>0.5</td>
<td>False</td>
</tr>
</tbody>
</table>

You’ll also have secondary dataset that references future measurements. This could come in two “types”. The first is when you have a variable that changes over time (ex: administering varying medication over time, or taking a temperature over time). The second type is an event-based dataset: an event happens at some time in the future (ex: an organ transplant occurs, or an intervention). We will address this second type later. The first type of dataset may look something like:

**Example:**

<table>
<thead>
<tr>
<th>id</th>
<th>time</th>
<th>var2</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>1.4</td>
</tr>
<tr>
<td>1</td>
<td>4</td>
<td>1.2</td>
</tr>
<tr>
<td>1</td>
<td>8</td>
<td>1.5</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>1.6</td>
</tr>
</tbody>
</table>

where `time` is the duration from the entry event. Here we see subject 1 had a change in their `var2` covariate at the end of time 4 and at the end of time 8. We can use `add_covariate_to_timeline` to fold the covariate dataset into the original dataset.

```python
from lifelines.utils import add_covariate_to_timeline

df = add_covariate_to_timeline(base_df, cv, duration_col="time", id_col="id", event_col="event")
```

<table>
<thead>
<tr>
<th>id</th>
<th>start</th>
<th>stop</th>
<th>var1</th>
<th>var2</th>
<th>event</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>4</td>
<td>0.1</td>
<td>1.4</td>
<td>False</td>
</tr>
<tr>
<td>1</td>
<td>4</td>
<td>8</td>
<td>0.1</td>
<td>1.2</td>
<td>False</td>
</tr>
<tr>
<td>1</td>
<td>8</td>
<td>10</td>
<td>0.1</td>
<td>1.5</td>
<td>True</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>12</td>
<td>0.5</td>
<td>1.6</td>
<td>False</td>
</tr>
</tbody>
</table>

From the above output, we can see that subject 1 changed state twice over the observation period, finally expiring at the end of time 10. Subject 2 was a censored case, and we lost track of them after time 12.

You may have multiple covariates you wish to add, so the above could be streamlined like so:

```python
from lifelines.utils import add_covariate_to_timeline
```

(continues on next page)
df = base_df.pipe(add_covariate_to_timeline, cv1, duration_col="time", id_col="id", event_col="event")
.pipe(add_covariate_to_timeline, cv2, duration_col="time", id_col="id", event_col="event")
.pipe(add_covariate_to_timeline, cv3, duration_col="time", id_col="id", event_col="event")

If your dataset is of the second type, that is, event-based, your dataset may look something like the following, where values in the matrix denote times since the subject’s birth, and None or NaN represent the event not happening (subjects can be excluded if the event never occurred as well):

```python
print(event_df)
```

<table>
<thead>
<tr>
<th>id</th>
<th>E1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1.0</td>
</tr>
<tr>
<td>1</td>
<td>NaN</td>
</tr>
<tr>
<td>2</td>
<td>3.0</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

Initially, this can’t be added to our baseline DataFrame. However, using `utils.covariates_from_event_matrix` we can convert a DataFrame like this into one that can be easily added.

```python
from lifelines.utils import covariates_from_event_matrix
cv = covariates_from_event_matrix(event_df, id_col="id")
print(cv)
```

<table>
<thead>
<tr>
<th>event</th>
<th>id</th>
<th>duration</th>
<th>E1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>1.0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>3</td>
<td>3.0</td>
<td>1</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td>...</td>
<td></td>
</tr>
</tbody>
</table>

```python
base_df = add_covariate_to_timeline(base_df, cv, duration_col="time", id_col="id", event_col="E")
```

For an example of pulling datasets like this from a SQL-store, and other helper functions, see *Example SQL queries and transformations to get time varying data*.

### Cumulative sums

One additional flag on `add_covariate_to_timeline` that is of interest is the `cumulative_sum` flag. By default it is False, but turning it to True will perform a cumulative sum on the covariate before joining. This is useful if the covariates describe an incremental change, instead of a state update. For example, we may have measurements of drugs administered to a patient, and we want the covariate to reflect how much we have administered since the start. Event columns do make sense to cumulative sum as well. In contrast, a covariate to measure the temperature of the patient is a state update, and should not be summed. See *Example cumulative sums over time-varying covariates* to see an example of this.
Delaying time-varying covariates

`add_covariate_to_timeline` also has an option for delaying, or shifting, a covariate so it changes later than originally observed. One may ask, why should one delay a time-varying covariate? Here’s an example. Consider investigating the impact of smoking on mortality and available to us are time-varying observations of how many cigarettes are consumed each month. Unbeknownst to us, when a subject reaches critical illness levels, they are admitted to the hospital and their cigarette consumption drops to zero. Some expire while in hospital. If we used this dataset naively, we would see that not smoking leads to sudden death, and conversely, smoking helps your health! This is a case of reverse causation: the upcoming death event actually influences the covariates.

To handle this, you can delay the observations by time periods:

```python
from lifelines.utils import covariates_from_event_matrix

base_df = add_covariate_to_timeline(base_df, cv, duration_col="time", id_col="id", event_col="E", delay=14)
```

Fitting the model

Once your dataset is in the correct orientation, we can use `CoxTimeVaryingFitter` to fit the model to your data. The method is similar to `CoxPHFitter`, expect we need to tell the `fit` about the additional time columns.

Fitting the Cox model to the data involves using gradient descent. `lifelines` takes extra effort to help with convergence, so please be attentive to any warnings that appear. Fixing any warnings will generally help convergence. For further help, see *Problems with convergence in the Cox proportional hazard model*.

```python
from lifelines import CoxTimeVaryingFitter

ctv = CoxTimeVaryingFitter()
ctv.fit(df, id_col="id", event_col="event", start_col="start", stop_col="stop", show_progress=True)
ctv.print_summary()
ctv.plot()
```

Short note on prediction

Unlike the other regression models, prediction in a time-varying setting is not trivial. To predict, we would need to know the covariates values beyond the observed times, but if we knew that, we would also know if the subject was still alive or not! However, it is still possible to compute the hazard values of subjects at known observations, the baseline cumulative hazard rate, and baseline survival function. So while `CoxTimeVaryingFitter` exposes prediction methods, there are logical limitations to what these predictions mean.
import numpy as np
import pandas as pd

1.8 Testing the proportional hazard assumptions

This Jupyter notebook is a small tutorial on how to test and fix proportional hazard problems.

The proportional hazard assumption is that all individuals have the same hazard function, but a unique scaling factor in front. So the shape of the hazard function is the same for all individuals, and only a scalar in front changes.

\[ h_i(t) = a_i h(t) \]

At the core of the assumption is that \( a_i \) is not time varying, that is, \( a_i(t) = a_i \). Further more, if we take the ratio of this with another subject (called the hazard ratio):

\[ \frac{h_i(t)}{h_j(t)} = \frac{a_i h(t)}{a_j h(t)} = \frac{a_i}{a_j} \]

is constant for all \( t \). In this tutorial we will test this non-time varying assumption, and look at ways to handle violations.

from lifelines.datasets import load_rossi
rossi = load_rossi()
cph = CoxPHFitter()
cph.fit(rossi, 'week', 'arrest')

from lifelines.CoxPHFitter: fitted with 432 observations, 318 censored

duration col = 'week'
event col = 'arrest'
number of subjects = 432
number of events = 114
log-likelihood = -658.748
time fit was run = 2019-04-03 02:39:31 UTC
model = untransformed variables

--
coef exp(coef) se(coef)  z     p   -log2(p) lower 0.95 upper 0.95
fin  -0.379  0.684  0.191 -1.983 0.047  4.398  -0.755  -0.004
age  -0.057  0.944  0.022 -2.611 0.009  6.791   -0.101  -0.014
race  0.314  1.369  0.308  1.019 0.308  1.698   -0.290   0.918
wexp -0.150  0.861  0.212 -0.706 0.480  1.058   -0.566   0.266
mar  -0.434  0.648  0.382 -1.136 0.256  1.965  -1.182   0.315
paro -0.085  0.919  0.196 -0.434 0.665  0.589   -0.469   0.299
prio  0.091  1.096  0.029  3.194 0.001  9.476    0.035   0.148
--
Concordance = 0.640
Log-likelihood ratio test = 33.266 on 7 df, -log2(p)=15.370
1.8.1 Checking assumptions with check_assumptions

New to lifelines 0.16.0 is the CoxPHFitter.check_assumptions method. This method will compute statistics that check the proportional hazard assumption, produce plots to check assumptions, and more. Also included is an option to display advice to the console. Here’s a breakdown of each information displayed:

- Presented first are the results of a statistical test to test for any time-varying coefficients. A time-varying coefficient imply a covariate’s influence relative to the baseline changes over time. This implies a violation of the proportional hazard assumption. For each variable, we transform time four times (these are common transformations of time to perform). If lifelines rejects the null (that is, lifelines rejects that the coefficient is not time-varying), we report this to the user.

- Some advice is presented on how to correct the proportional hazard violation based on some summary statistics of the variable.

- As a compliment to the above statistical test, for each variable that violates the PH assumption, visual plots of the the scaled Schoenfeld residuals is presented against the four time transformations. A fitted lowess is also presented, along with 10 bootstrapped lowess lines (as an approximation to the confidence interval of the original lowess line). Ideally, this lowess line is constant (flat). Deviations away from the constant line are violations of the PH assumption.

Why the scaled Schoenfeld residuals?

This section can be skipped on first read. Let $s_{t,j}$ denote the scaled Schoenfeld residuals of variable j at time t, $\hat{\beta}_j$ denote the maximum-likelihood estimate of the jth variable, and $\beta_j(t)$ a time-varying coefficient in (fictional) alternative model that allows for time-varying coefficients. Therneau and Grambsch showed that:

$$E[s_{t,j}] + \hat{\beta}_j = \beta_j(t)$$

The proportional hazard assumption implies that $\hat{\beta}_j = \beta_j(t)$, hence $E[s_{t,j}] = 0$. This is what the above proportional hazard test is testing. Visually, plotting $s_{t,j}$ over time (or some transform of time), is a good way to see violations of $E[s_{t,j}] = 0$, along with the statistical test.

```
[4]: cph.check_assumptions(rossi, p_value_threshold=0.05, show_plots=True)
```

The `p_value_threshold` is set at 0.05. Even under the null hypothesis of no violations, some covariates will be below the threshold by chance. This is compounded when there are many covariates. Similarly, when there are lots of observations, even minor deviances from the proportional hazard assumption will be flagged.

With that in mind, it’s best to use a combination of statistical tests and visual tests to determine the most serious violations. Produce visual plots using `check_assumptions(..., show_plots=True)` and looking for non-constant lines. See link [A] below for a full example.
1. Variable ‘age’ failed the non-proportional test: p-value is 0.0009.

   Advice 1: the functional form of the variable ‘age’ might be incorrect. That is, there may be non-linear terms missing. The proportional hazard test used is very sensitive to incorrect functional forms. See documentation in link [D] below on how to specify a functional form.

   Advice 2: try binning the variable ‘age’ using pd.cut, and then specify it in `strata=['age', ...]` in the call in `.fit`. See documentation in link [B] below.

   Advice 3: try adding an interaction term with your time variable. See documentation in link [C] below.

2. Variable ‘wexp’ failed the non-proportional test: p-value is 0.0063.

   Advice: with so few unique values (only 2), you can include `strata=['wexp', ...]` in the call in `.fit`. See documentation in link [E] below.

---

Alternatively, you can use the proportional hazard test outside of `check_assumptions`:

```python
from lifelines.statistics import proportional_hazard_test
results = proportional_hazard_test(cph, rossi, time_transform='rank')
results.print_summary(decimals=3, model="untransformed variables")
```

```
<lifelines.StatisticalResult>
    test_name = proportional_hazard_test
    time_transform = rank
    null_distribution = chi squared
    degrees_of_freedom = 1
        model = untransformed variables

    --
    test_statistic  p  -log2(p)
    age    11.094  0.001  10.173
```
Stratification

In the advice above, we can see that \texttt{wexp} has small cardinality, so we can easily fix that by specifying it in the \texttt{strata}. What does the \texttt{strata} do? Let’s go back to the proportional hazard assumption.

In the introduction, we said that the proportional hazard assumption was that

\[
 h_i(t) = a_i h(t)
\]

In a simple case, it may be that there are two subgroups that have \textit{very} different baseline hazards. That is, we can split the dataset into subsamples based on some variable (we call this the stratifying variable), run the Cox model on all subsamples, and compare their baseline hazards. If these baseline hazards are \textit{very} different, then clearly the formula above is wrong - the \( h(t) \) is some weighted average of the subgroups’ baseline hazards. This ill fitting average baseline can cause \( a_i \) to have time-dependent influence. A better model might be:

\[
 h_{i \in G}(t) = a_i h_G(t)
\]

where now we have a unique baseline hazard \textit{per} subgroup \( G \). Because of the way the Cox model is designed, inference of the coefficients is identical (expect now there are more baseline hazards, and no variation of the stratifying variable within a subgroup \( G \)).

```r
[6]: cph.fit(rossi, 'week', 'arrest', strata=['wexp'])
cph.print_summary(model="wexp in strata")
< lifelines.CoxPHFitter: fitted with 432 observations, 318 censored>

duration col = 'week'
event col = 'arrest'
strata = ['wexp']
number of subjects = 432
number of events = 114
log-likelihood = -580.89
time fit was run = 2019-04-03 02:39:34 UTC
model = wexp in strata

--
   coef  exp(coef)  se(coef)     z      p  -log2(p) lower 0.95 upper 0.95
fin -0.38 0.68 0.19 -1.99 0.05  4.42 -0.76  -0.01
age -0.06 0.94 0.02 -2.64 0.01  6.91 -0.10  -0.01
race 0.31 1.36 0.31  1.00 0.32  1.65 -0.30  0.91
mar -0.45 0.64 0.38 -1.19 0.23  2.09 -1.20  0.29
paro -0.08 0.92 0.20 -0.42 0.67  0.57 -0.47  0.30
prio 0.09 1.09 0.03  3.16 <0.005  9.33  0.03  0.15
--
Concordance = 0.61
Log-likelihood ratio test = 172.71 on 6 df, -log2(p)=112.69
```
The ``p_value_threshold`` is set at 0.01. Even under the null hypothesis of no violations, some covariates will be below the threshold by chance. This is compounded when there are many covariates. Similarly, when there are lots of observations, even minor deviances from the proportional hazard assumption will be flagged.

With that in mind, it’s best to use a combination of statistical tests and visual tests to determine the most serious violations. Produce visual plots using `check_assumptions(..., show_plots=True)` and looking for non-constant lines. See link [A] below for a full example.

```
< lifelines.StatisticalResult >
    test_name = proportional_hazard_test
    null_distribution = chi squared
    degrees_of_freedom = 1
```

<table>
<thead>
<tr>
<th>test_statistic</th>
<th>p</th>
<th>-log2(p)</th>
</tr>
</thead>
<tbody>
<tr>
<td>age</td>
<td>11.29</td>
<td>&lt;0.005</td>
</tr>
<tr>
<td>fin</td>
<td>0.02</td>
<td>0.90</td>
</tr>
<tr>
<td>mar</td>
<td>0.53</td>
<td>0.47</td>
</tr>
<tr>
<td>paro</td>
<td>0.09</td>
<td>0.76</td>
</tr>
<tr>
<td>prio</td>
<td>0.02</td>
<td>0.89</td>
</tr>
<tr>
<td>race</td>
<td>1.47</td>
<td>0.23</td>
</tr>
<tr>
<td>rank</td>
<td>0.64</td>
<td>0.42</td>
</tr>
</tbody>
</table>

1. Variable ‘age’ failed the non-proportional test: p-value is 0.0008.

   Advice 1: the functional form of the variable ‘age’ might be incorrect. That is, there may be non-linear terms missing. The proportional hazard test used is very sensitive to incorrect functional forms. See documentation in link [D] below on how to specify a functional form.

   Advice 2: try binning the variable ‘age’ using pd.cut, and then specify it in `strata=['age', ...]` in the call in `.fit`. See documentation in link [B] below.

   Advice 3: try adding an interaction term with your time variable. See documentation in link [C] below.

Since age is still violating the proportional hazard assumption, we need to model it better. From the residual plots above, we can see the effect of age start to become negative over time. This will be relevant later. Below, we present three options to handle age.

**Modify the functional form**

The proportional hazard test is very sensitive (i.e. lots of false positives) when the functional form of a variable is incorrect. For example, if the association between a covariate and the log-hazard is non-linear, but the model has only a linear term included, then the proportional hazard test can raise a false positive.

The modeller can choose to add quadratic or cubic terms, i.e:

```python
rossi['age**2'] = (rossi['age'] - rossi['age'].mean())**2
rossi['age**3'] = (rossi['age'] - rossi['age'].mean())**3
```

but I think a more correct way to include non-linear terms is to use splines. Both Patsy and zEpid provide functionality for splines (tutorial incoming), but let's stick with the form above.

```python
[8]: rossi_higher_order_age = rossi.copy()
rossi_higher_order_age['age'] = rossi_higher_order_age['age'] - rossi_higher_order_age['age'].mean()
rossi_higher_order_age['age**2'] = (rossi_higher_order_age['age'] - rossi_higher_order_age['age'].mean())**2
rossi_higher_order_age['age**3'] = (rossi_higher_order_age['age'] - rossi_higher_order_age['age'].mean())**3
```

(continues on next page)
```python
cph.fit(rossi_higher_order_age, 'week', 'arrest', strata=['wexp'])
cph.print_summary(model="quad and cubic age terms"); print()
cph.check_assumptions(rossi_higher_order_age, show_plots=True, p_value_threshold=0.05)
```

<lifelines.CoxPHFitter: fitted with 432 observations, 318 censored>

duration col = 'week'
event col = 'arrest'
strata = ['wexp']

number of subjects = 432
number of events = 114
log-likelihood = -579.37
time fit was run = 2019-04-03 02:39:36 UTC
model = quad and cubic age terms

```
    coef  exp(coef)  se(coef)   z     p    -log2(p)    lower 0.95    upper 0.95
  fin  -0.37     0.69    0.19  -1.93 0.05      4.24   -0.75        0.00
  age  -0.06     0.94    0.03 -1.85 0.06      3.95   -0.13        0.00
 race  0.35     1.42    0.31  1.13 0.26      1.95   -0.26        0.95
  mar  -0.39     0.68    0.38 -1.02 0.31      1.70   -1.15        0.36
  paro 0.10     1.10    0.20 -0.52 0.60      0.74   -0.49        0.28
 race**2 0.01    1.01    0.00  1.57 0.12      3.09   -0.00        0.02
 age**2 -0.00    1.00   -0.89  0.37 0.37      1.42   -0.00        0.00
```

Concordance = 0.62
Log-likelihood ratio test = 175.75 on 8 df, -log2(p)=109.94

The `p_value_threshold` is set at 0.05. Even under the null hypothesis of no violations, some covariates will be below the threshold by chance. This is compounded when there are many covariates.

Similarly, when there are lots of observations, even minor deviances from the proportional hazard assumption will be flagged.

With that in mind, it’s best to use a combination of statistical tests and visual tests to determine the most serious violations. Produce visual plots using `check_assumptions(..., show_plots=True)` and looking for non-constant lines. See link [A] below for a full example.

```
<lifelines.StatisticalResult>
    test_name = proportional_hazard_test
    null_distribution = chi squared
degrees_of_freedom = 1
```

```
    test_statistic  p   -log2(p)
  age     km  0.96 0.33   1.62
         rank 4.09 0.04   4.54
age**2 km  1.81 0.18   2.48
          rank 0.79 0.37   1.42
age**3 km  2.33 0.13   2.98
          rank 0.03 0.87   0.19
```

(continues on next page)
<table>
<thead>
<tr>
<th>Variable</th>
<th>km</th>
<th>hr</th>
<th>rank</th>
</tr>
</thead>
<tbody>
<tr>
<td>fin</td>
<td>0.03 0.87</td>
<td>0.20</td>
<td></td>
</tr>
<tr>
<td>rank</td>
<td>0.02 0.90</td>
<td>0.15</td>
<td></td>
</tr>
<tr>
<td>mar</td>
<td>0.53 0.47</td>
<td>1.10</td>
<td></td>
</tr>
<tr>
<td>rank</td>
<td>0.94 0.33</td>
<td>1.59</td>
<td></td>
</tr>
<tr>
<td>paro</td>
<td>0.20 0.66</td>
<td>0.60</td>
<td></td>
</tr>
<tr>
<td>rank</td>
<td>0.01 0.93</td>
<td>0.10</td>
<td></td>
</tr>
<tr>
<td>prio</td>
<td>0.02 0.88</td>
<td>0.19</td>
<td></td>
</tr>
<tr>
<td>rank</td>
<td>0.01 0.90</td>
<td>0.15</td>
<td></td>
</tr>
<tr>
<td>race</td>
<td>1.28 0.26</td>
<td>1.96</td>
<td></td>
</tr>
<tr>
<td>rank</td>
<td>0.47 0.49</td>
<td>1.02</td>
<td></td>
</tr>
</tbody>
</table>

1. Variable ‘age’ failed the non-proportional test: p-value is 0.0431.

   Advice 1: the functional form of the variable ‘age’ might be incorrect. That is, there may be non-linear terms missing. The proportional hazard test used is very sensitive to incorrect functional forms. See documentation in link [D] below on how to specify a functional form.

   Advice 2: try binning the variable ‘age’ using pd.cut, and then specify it in strata=['age', ...] in the call in '.fit'. See documentation in link [B] below.

   Advice 3: try adding an interaction term with your time variable. See documentation in link [C] below.

---


1.8. Testing the proportional hazard assumptions
We see we still have potentially *some* violation, but it’s a heck of a lot less. Also, interestingly, when we include these non-linear terms for age, the \textit{wexp} proportionality violation disappears. It is not uncommon to see changing the functional form of one variable effects other’s proportional tests, usually positively. So, we could remove the \texttt{strata=['wexp']} if we wished.

### Bin variable and stratify on it

The second option proposed is to bin the variable into equal-sized bins, and stratify like we did with \textit{wexp}. There is a trade off here between estimation and information-loss. If we have large bins, we will lose information (since different values are now binned together), but we need to estimate less new baseline hazards. On the other hand, with tiny bins, we allow the \textit{age} data to have the most “wiggle room”, but must compute many baseline hazards each of which has a smaller sample size. Like most things, the optimal value is somewhere inbetween.

```python
[9]: rossi_strata_age = rossi.copy()
    rossi_strata_age['age_strata'] = pd.cut(rossi_strata_age['age'], np.arange(0, 80, 3))
    rossi_strata_age[['age', 'age_strata']].head()
```

<table>
<thead>
<tr>
<th>age_strata</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 27 (24, 27)</td>
</tr>
<tr>
<td>1 18 (15, 18)</td>
</tr>
<tr>
<td>2 19 (18, 21)</td>
</tr>
<tr>
<td>3 23 (21, 24)</td>
</tr>
<tr>
<td>4 19 (18, 21)</td>
</tr>
</tbody>
</table>

```python
[10]: # drop the original, redundant, \textit{age} column
    rossi_strata_age = rossi_strata_age.drop('age', axis=1)
    cph.fit(rossi_strata_age, 'week', 'arrest', strata=['age_strata', 'wexp'])
```

```python
[10]: <lifelines.CoxPHFitter: fitted with 432 observations, 318 censored>
```

```python
[11]: cph.print_summary(3, model="stratified age and wexp")
    cph.plot()
```

```python
<lifelines.CoxPHFitter: fitted with 432 observations, 318 censored>
```
event col = 'arrest'
strata = ['age_strata', 'wexp']
number of subjects = 432
number of events = 114
log-likelihood = -392.443
time fit was run = 2019-04-03 02:39:37 UTC
model = stratified age and wexp

--

<table>
<thead>
<tr>
<th></th>
<th>coef</th>
<th>exp(coef)</th>
<th>se(coef)</th>
<th>z</th>
<th>p</th>
<th>-log2(p)</th>
<th>lower 0.95</th>
<th>upper 0.95</th>
</tr>
</thead>
<tbody>
<tr>
<td>fin</td>
<td>-0.395</td>
<td>0.674</td>
<td>0.197</td>
<td>-2.004</td>
<td>0.045</td>
<td>4.472</td>
<td>-0.781</td>
<td>-0.009</td>
</tr>
<tr>
<td>race</td>
<td>0.280</td>
<td>1.324</td>
<td>0.313</td>
<td>0.895</td>
<td>0.371</td>
<td>1.431</td>
<td>-0.334</td>
<td>0.895</td>
</tr>
<tr>
<td>mar</td>
<td>-0.194</td>
<td>0.824</td>
<td>0.392</td>
<td>-0.494</td>
<td>0.621</td>
<td>0.687</td>
<td>-0.961</td>
<td>0.574</td>
</tr>
<tr>
<td>paro</td>
<td>-0.163</td>
<td>0.849</td>
<td>0.200</td>
<td>-0.818</td>
<td>0.413</td>
<td>1.275</td>
<td>-0.555</td>
<td>0.228</td>
</tr>
<tr>
<td>prio</td>
<td>0.080</td>
<td>1.084</td>
<td>0.028</td>
<td>2.854</td>
<td>0.004</td>
<td>7.857</td>
<td>0.025</td>
<td>0.135</td>
</tr>
</tbody>
</table>

Concordance = 0.582
Log-likelihood ratio test = 532.244 on 5 df, -log2(p)=372.252

[11]: <matplotlib.axes._subplots.AxesSubplot at 0x120e05828>

[12]: cph.check_assumptions(rossi_strata_age)
Proportional hazard assumption looks okay.

**Introduce time-varying covariates**

Our second option to correct variables that violate the proportional hazard assumption is to model the time-varying component directly. This is done in two steps. The first is to transform your dataset into *episodic format*. This means that we split a subject from a single row into $n$ new rows, and each new row represents some time period for the subject. It’s okay that the variables are static over this new time periods - we’ll introduce some time-varying covariates later.

See below for how to do this in *lifelines*:
from lifelines.utils import to_episodic_format

# the time_gaps parameter specifies how large or small you want the periods to be.
rossi_long = to_episodic_format(rossi, duration_col='week', event_col='arrest', time_gaps=1.)
rossi_long.head(25)

Each subject is given a new id (but can be specified as well if already provided in the dataframe). This id is used to track subjects over time. Notice the arrest col is 0 for all periods prior to their (possible) event as well.

Above I mentioned there were two steps to correct age. The first was to convert to a episodic format. The second is to create an interaction term between age and stop. This is a time-varying variable.

Instead of CoxPHFitter, we must use CoxTimeVaryingFitter instead since we are working with a episodic dataset.

rossi_long['time*age'] = rossi_long['age'] * rossi_long['stop']

from lifelines import CoxTimeVaryingFitter
crv = CoxTimeVaryingFitter()
crv.fit(rossi_long,
id_col='id',
event_col='arrest',
start_col='start',
stop_col='stop',
strata=['wexp'])

<lifelines.CoxTimeVaryingFitter: fitted with 19809 periods, 432 subjects, 114 events>
In the above scaled Schoenfeld residual plots for `age`, we can see there is a slight negative effect for higher time values. This is confirmed in the output of the `CoxTimeVaryingFitter`: we see that the coefficient for `time*age` is -0.005.

**Conclusion**

The point estimates and the standard errors are very close to each other using either option, we can feel confident that either approach is okay to proceed.
1.9 More examples and recipes

This section goes through some examples and recipes to help you use lifelines. If you are looking for some full example usage of lifelines, there are full Jupyter notebooks here and examples and ideas on the development blog.

1.9.1 Statistically compare two populations

Often researchers want to compare survival-ness between different populations. Here are some techniques to do that:

Subtraction and division between survival curves

If you are interested in taking the difference between two survival curves, simply trying to subtract the survival_function_ will likely fail if the DataFrame’s indexes are not equal. Fortunately, the KaplanMeierFitter and NelsonAalenFitter have a built-in subtract method:

```python
kmf1.subtract(kmf2)
```

will produce the difference at every relevant time point. A similar function exists for division: divide. However, for rigorous testing of differences, lifelines comes with a statistics library. See below.

Logrank test

Note: The logrank test has maximum power when the assumption of proportional hazards is true. As a consequence, if the survival curves cross, the logrank test will give an inaccurate assessment of differences.

The lifelines.statistics.logrank_test function compares whether the “death” generation process of the two populations are equal:

```python
from lifelines.statistics import logrank_test

results = logrank_test(T1, T2, event_observed_A=E1, event_observed_B=E2)
results.print_summary()

```
If you have more than two populations, you can use pairwise_logrank_test (which compares each pair in the same manner as above), or multivariate_logrank_test (which tests the hypothesis that all the populations have the same “death” generation process).

```python
from lifelines.statistics import multivariate_logrank_test
df = pd.DataFrame({
    'durations': [5, 3, 9, 7, 4, 4, 3, 2, 5, 6, 7, 4],
    'groups': [0, 0, 0, 1, 1, 1, 1, 2, 2, 2, 1, 1],  # could be strings too
    'events': [1, 1, 1, 1, 1, 1, 0, 0, 1, 1, 1, 0],
})
results = multivariate_logrank_test(df['durations'], df['groups'], df['events'])
results.print_summary()
```

Survival differences at a point in time

Often analysts want to compare the survival-ness of groups at specific times, rather than comparing the entire survival curves against each other. For example, analysts may be interested in 5-year survival. Statistically comparing the naive Kaplan-Meier points at a specific time actually has reduced power. By transforming the Kaplan-Meier curve, we can recover more power. The function statistics.survival_difference_at_fixed_point_in_time_test uses the log(-log) transformation implicitly and compares the survival-ness of populations at a specific point in time.

```python
from lifelines.statistics import survival_difference_at_fixed_point_in_time_test
results = survival_difference_at_fixed_point_in_time_test(point_in_time, T1, T2,  
  event_observed_A=E1, event_observed_B=E2)
results.print_summary()
```

1.9.2 Model selection using lifelines

If using lifelines for prediction work, it’s ideal that you perform some type of cross-validation scheme. This cross-validation allows you to be confident that your out-of-sample predictions will work well in practice. It also allows you
to choose between multiple models.

*lifelines* has a built-in k-fold cross-validation function. For example, consider the following example:

```python
from lifelines import AalenAdditiveFitter, CoxPHFitter
from lifelines.datasets import load_regression_dataset
from lifelines.utils import k_fold_cross_validation

df = load_regression_dataset()

# create the three models we'd like to compare.
aaf_1 = AalenAdditiveFitter(coef_penalizer=0.5)
aaf_2 = AalenAdditiveFitter(coef_penalizer=10)
cph = CoxPHFitter()

print(np.mean(k_fold_cross_validation(cph, df, duration_col='T', event_col='E')))  
print(np.mean(k_fold_cross_validation(aaf_1, df, duration_col='T', event_col='E')))  
print(np.mean(k_fold_cross_validation(aaf_2, df, duration_col='T', event_col='E')))  
```

From these results, Aalen’s Additive model with a penalizer of 10 is best model of predicting future survival times.

### 1.9.3 Selecting a parametric model using QQ plots

QQ plots normally are constructed by sorting the values. However, this isn’t appropriate when there is censored data. In *lifelines*, there are routines to still create QQ plots with censored data. These are available under *lifelines.plotting.qq_plots*, and accepts fitted a parametric lifelines model.

```python
from lifelines import *
from lifelines.plotting import qq_plot

# generate some fake log-normal data
N = 1000
T_actual = np.exp(np.random.randn(N))
C = np.exp(np.random.randn(N))
E = T_actual < C
T = np.minimum(T_actual, C)

fig, axes = plt.subplots(2, 2, figsize=(8, 6))
axes = axes.reshape(4,)

for i, model in enumerate([WeibullFitter(), LogNormalFitter(), LogLogisticFitter(), ExponentialFitter()]):
    model.fit(T, E)
    qq_plot(model, ax=axes[i])
```
This graphical test can be used to invalidate models. For example, in the above figure, we can see that only the log-normal parametric model is appropriate (we expect deviance in the tails, but not too much). Another use case is choosing the correct parametric AFT model.
The \texttt{qq\_plot} also works with left censorship as well.

### 1.9.4 Plotting multiple figures on a plot

When \texttt{.plot} is called, an \texttt{axis} object is returned which can be passed into future calls of \texttt{.plot}:

```python
kmf.fit(data1)
ax = kmf.plot()
kmf.fit(data2)
ax = kmf.plot(ax=ax)
```

If you have a pandas DataFrame with columns “group”, “\(T\)”, and “\(E\)”, then something like the following would work:

```python
from lifelines import KaplanMeierFitter
from matplotlib import pyplot as plt

ax = plt.subplot(111)
kmf = KaplanMeierFitter()
for name, grouped_df in df.groupby('group'):
    kmf.fit(grouped_df['T'], grouped_df['E'], label=name)
    kmf.plot(ax=ax)
```

### 1.9.5 Plotting options and styles

Let’s load some data:

```python
from lifelines.datasets import load_waltons
waltons = load_waltons()
T = waltons['T']
E = waltons['E']
```

**Standard**

```python
kmf = KaplanMeierFitter()
kmf.fit(T, E, label="kmf.plot()")
kmf.plot()
```
Show censors and edit markers

```python
kmf.fit(T, E, label="kmf.plot(show_censors=True, censor_styles={'ms': 6, 'marker': 's'})")
kmf.plot(show_censors=True, censor_styles={'ms': 6, 'marker': 's'})
```
Hide confidence intervals

```python
kmf.fit(T, E, label="kmf.plot(ci_show=False)")
kmf.plot(ci_show=False)
```
Invert axis

```python
tf.mfit(T, E, label="kmf.plot(invert_y_axis=True)")

kmf.plot(invert_y_axis=True)
```
Note: This is deprecated and we suggest to use `kmf.plot_cumulative_density()` instead.

Displaying at-risk counts below plots

```python
kmf.fit(T, E, label="label name")
kmf.plot(at_risk_counts=True)
```
Displaying multiple at-risk counts below plots

The function `add_at_risk_counts` in `lifelines.plotting` allows you to add At-Risk counts at the bottom of your figures. For example:

```python
from lifelines import KaplanMeierFitter
ix = waltons['group'] == 'control'
ax = plt.subplot(111)
kmf_control = KaplanMeierFitter()
ax = kmf_control.fit(waltons.loc[ix]['T'], waltons.loc[ix]['E'], label='control').plot(ax=ax)

kmf_exp = KaplanMeierFitter()
ax = kmf_exp.fit(waltons.loc[~ix]['T'], waltons.loc[~ix]['E'], label='exp').plot(ax=ax)

from lifelines.plotting import add_at_risk_counts
add_at_risk_counts(kmf_exp, kmf_control, ax=ax)
```

will display
1.9.6 Transforming survival-table data into lifelines format

Some lifelines classes are designed for lists or arrays that represent one individual per row. If you instead have data in a survival table format, there exists a utility method to get it into lifelines format.

Example: Suppose you have a CSV file with data that looks like this:

```
<table>
<thead>
<tr>
<th>time</th>
<th>observed deaths</th>
<th>censored</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>7</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>4</td>
<td>5</td>
<td>2</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
```

```python
import pandas as pd
from lifelines.utils import survival_events_from_table

df = pd.read_csv('file.csv', columns=['time', 'observed deaths', 'censored'])
df = df.set_index('time')
```

(continues on next page)
T, E, W = survival_events_from_table(df, observed_deaths_col='observed deaths',
censored_col='censored')
# weights, W, is the number of occurrences of each observation - helps with data
→ compression.
kmf = KaplanMeierFitter().fit(T, E, weights=W)

### 1.9.7 Transforming observational data into survival-table format

Perhaps you are interested in viewing the survival table given some durations and censoring vectors.

```python
from lifelines.utils import survival_table_from_events

table = survival_table_from_events(T, E)
print(table.head())
```

```
removed  observed  censored  entrance  at_risk
event_at
0       0         0         0         60       60
1       2         1         1         60       60
2       3         1         2         58       58
3       5         3         2         55       55
4       12        6         6         50       50
```

### 1.9.8 Set the index/timeline of a estimate

Suppose your dataset has lifetimes grouped near time 60, thus after fitting `KaplanMeierFitter`, your survival function might look something like:

```python
print(kmf.survival_function_)
```

```
KM-estimate
0     1.00
47    0.99
49    0.97
50    0.96
51    0.95
52    0.91
53    0.86
54    0.84
55    0.79
56    0.74
57    0.71
58    0.67
59    0.58
60    0.49
61    0.41
62    0.31
63    0.24
64    0.19
65    0.14
```
What you would like is to have a predictable and full index from 40 to 75. (Notice that in the above index, the last two
time points are not adjacent – the cause is observing no lifetimes existing for times 72 or 73). This is especially useful
for comparing multiple survival functions at specific time points. To do this, all fitter methods accept a timeline
argument:

```python
kmf.fit(T, timeline=range(40,75))
print(kmf.survival_function_)
```

```
KM-estimate
40  1.00
41  1.00
42  1.00
43  1.00
44  1.00
45  1.00
46  1.00
47  0.99
48  0.99
49  0.97
50  0.96
51  0.95
52  0.91
53  0.86
54  0.84
55  0.79
56  0.74
57  0.71
58  0.67
59  0.58
60  0.49
61  0.41
62  0.31
63  0.24
64  0.19
65  0.14
66  0.10
67  0.10
68  0.07
69  0.04
70  0.02
71  0.01
72  0.01
73  0.01
74  0.00
```

*lifelines* will intelligently forward-fill the estimates to unseen time points.
1.9.9 Example SQL query to get survival data from a table

Below is a way to get an example dataset from a relational database (this may vary depending on your database):

```sql
SELECT
    id,
    DATEDIFF('dd', started_at, COALESCE(ended_at, CURRENT_DATE)) AS "T",
    (ended_at IS NOT NULL) AS "E"
FROM table
```

**Explanation**

Each row is an id, a duration, and a boolean indicating whether the event occurred or not. Recall that we denote a “True” if the event did occur, that is, ended_at is filled in (we observed the ended_at). Ex:

<table>
<thead>
<tr>
<th>id</th>
<th>T</th>
<th>E</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>40</td>
<td>True</td>
</tr>
<tr>
<td>11</td>
<td>42</td>
<td>False</td>
</tr>
<tr>
<td>12</td>
<td>42</td>
<td>False</td>
</tr>
<tr>
<td>13</td>
<td>36</td>
<td>True</td>
</tr>
<tr>
<td>14</td>
<td>33</td>
<td>True</td>
</tr>
</tbody>
</table>

1.9.10 Example SQL queries and transformations to get time varying data

For Cox time-varying models, we discussed what the dataset should look like in *Dataset creation for time-varying regression*. Typically we have a base dataset, and then we fold in the covariate datasets. Below are some SQL queries and Python transformations from end-to-end.

**Base dataset**: `base_df`

```sql
SELECT
    id, group,
    DATEDIFF('dd', dt.started_at, COALESCE(dt.ended_at, CURRENT_DATE)) AS "T",
    (dt.ended_at IS NOT NULL) AS "E"
FROM dimension_table dt
```

**Time-varying variables**: `cv`

```sql
-- this could produce more than 1 row per subject
SELECT
    id,
    DATEDIFF('dd', dt.started_at, ft.event_at) AS "time",
    ft.var1
FROM fact_table ft
JOIN dimension_table dt
USING(id)
```
from lifelines.utils import to_long_format
from lifelines.utils import add_covariate_to_timeline

base_df = to_long_format(base_df, duration_col="T")
df = add_covariate_to_timeline(base_df, cv, duration_col="time", id_col="id", event_col="E")

---

### Event variables: event_df

Another very common operation is to add event data to our time-varying dataset. For example, a dataset/SQL table that contains information about the dates of an event (and NULLS if the event didn’t occur). An example SQL query may look like:

```sql
SELECT
  id,
  DATEDIFF('dd', dt.started_at, ft.event1_at) AS "E1",
  DATEDIFF('dd', dt.started_at, ft.event2_at) AS "E2",
  DATEDIFF('dd', dt.started_at, ft.event3_at) AS "E3"
FROM dimension_table dt
```

In Pandas, this may look like:

<table>
<thead>
<tr>
<th>id</th>
<th>E1</th>
<th>E2</th>
<th>E3</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>1.0</td>
<td>NaN</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>NaN</td>
<td>5.0</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>3.0</td>
<td>5.0</td>
</tr>
</tbody>
</table>

Initially, this can’t be added to our baseline time-varying dataset. Using `utils.covariates_from_event_matrix` we can convert a dataframe like this into one that can be easily added.

```python
from lifelines.utils import covariates_from_event_matrix
cv = covariates_from_event_matrix(event_df, id_col='id')
print(cv)
```

<table>
<thead>
<tr>
<th>id</th>
<th>duration</th>
<th>E1</th>
<th>E2</th>
<th>E3</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>1.0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>2.0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>5.0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>5</td>
<td>3.0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>5</td>
<td>5.0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
<td>7.0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

```python
base_df = add_covariate_to_timeline(base_df, cv, duration_col="time", id_col="id", event_col="E")
```

---

### 1.9.11 Example cumulative sums over time-varying covariates

Often we have either transactional covariate datasets or state covariate datasets. In a transactional dataset, it may make sense to sum up the covariates to represent administration of a treatment over time. For example, in the risky world
of start-ups, we may want to sum up the funding amount received at a certain time. We also may be interested in the
amount of the last round of funding. Below is an example to do just that:

Suppose we have an initial DataFrame of start-ups like:

```python
seed_df = pd.DataFrame([{'id': 'FB', 'E': True, 'T': 12, 'funding': 0},
                        {'id': 'SU', 'E': True, 'T': 10, 'funding': 0},
])
```

And a covariate dataframe representing funding rounds like:

```python
cv = pd.DataFrame([{'id': 'FB', 'funding': 30, 't': 5},
                    {'id': 'FB', 'funding': 15, 't': 10},
                    {'id': 'FB', 'funding': 50, 't': 15},
                    {'id': 'SU', 'funding': 10, 't': 6},
                    {'id': 'SU', 'funding': 9, 't': 10},
])
```

We can do the following to get both the cumulative funding received and the latest round of funding:

```python
from lifelines.utils import to_long_format
from lifelines.utils import add_covariate_to_timeline

df = seed_df.pipe(to_long_format, 'T')
    .pipe(add_covariate_to_timeline, cv, 'id', 't', 'E', cumulative_sum=True)
    .pipe(add_covariate_to_timeline, cv, 'id', 't', 'E', cumulative_sum=False)
```

```
<table>
<thead>
<tr>
<th>start</th>
<th>cumsum_funding</th>
<th>funding</th>
<th>stop</th>
<th>id</th>
<th>E</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0.0</td>
<td>0.0</td>
<td>5.0</td>
<td>FB</td>
<td>False</td>
</tr>
<tr>
<td>5</td>
<td>30.0</td>
<td>30.0</td>
<td>10.0</td>
<td>FB</td>
<td>False</td>
</tr>
<tr>
<td>10</td>
<td>45.0</td>
<td>15.0</td>
<td>12.0</td>
<td>FB</td>
<td>True</td>
</tr>
<tr>
<td>0</td>
<td>0.0</td>
<td>0.0</td>
<td>6.0</td>
<td>SU</td>
<td>False</td>
</tr>
<tr>
<td>6</td>
<td>10.0</td>
<td>10.0</td>
<td>10.0</td>
<td>SU</td>
<td>False</td>
</tr>
<tr>
<td>10</td>
<td>19.0</td>
<td>9.0</td>
<td>10.0</td>
<td>SU</td>
<td>True</td>
</tr>
</tbody>
</table>
```

**1.9.12 Sample size determination under a CoxPH model**

Suppose you wish to measure the hazard ratio between two populations under the CoxPH model. That is, we want
to evaluate the hypothesis $H_0$: relative hazard ratio = 1 vs $H_1$: relative hazard ratio $\neq 1$, where the relative hazard
ratio is $\exp(\beta)$ for the experiment group vs the control group. A priori, we are interested in the sample sizes of
the two groups necessary to achieve a certain statistical power. To do this in lifelines, there is the lifelines.
statistics.sample_size_necessary_under_cph function. For example:

```python
from lifelines.statistics import sample_size_necessary_under_cph

desired_power = 0.8
ratio_of_participants = 1.
p_exp = 0.25
p_con = 0.35
postulated_hazard_ratio = 0.7
```
This assumes you have estimates of the probability of event occurring for both the experiment and control group. This could be determined from previous experiments.

### 1.9.13 Power determination under a CoxPH model

Suppose you wish to measure the hazard ratio between two populations under the CoxPH model. To determine the statistical power of a hazard ratio hypothesis test, under the CoxPH model, we can use `lifelines.statistics.power_under_cph`. That is, suppose we want to know the probability that we reject the null hypothesis that the relative hazard ratio is 1, assuming the relative hazard ratio is truly different from 1. This function will give you that probability.

```python
from lifelines.statistics import power_under_cph

n_exp = 50
n_con = 100
p_exp = 0.25
p_con = 0.35
postulated_hazard_ratio = 0.5
power = power_under_cph(n_exp, n_con, p_exp, p_con, postulated_hazard_ratio)
# 0.4957
```

### 1.9.14 Problems with convergence in the Cox proportional hazard model

Since the estimation of the coefficients in the Cox proportional hazard model is done using the Newton-Raphson algorithm, there are sometimes problems with convergence. Here are some common symptoms and resolutions:

1. First check: look for `ConvergenceWarning` in the output. Most often problems in convergence are the result of problems in the dataset. `lifelines` has checks it runs against the dataset before fitting and warnings are outputted to the user.

2. `delta` contains `nan` value(s): First try adding `show_progress=True` in the `fit` function. If the values in `delta` grow unbounded, it’s possible the `step_size` is too large. Try setting it to a small value (0.1-0.5).

3. Convergence halted due to matrix inversion problems: This means that there is high collinearity in your dataset. That is, a column is equal to the linear combination of 1 or more other columns. A common cause of this error is dummying categorical variables but not dropping a column, or some hierarchical structure in your dataset. Try to find the relationship by:
   1. adding a penalizer to the model, ex: `CoxPHFitter(penalizer=0.1).fit(...)` until the model converges. In the `print_summary()`, the coefficients that have high collinearity will have large (absolute) magnitude in the `coefs` column.
   2. using the variance inflation factor (VIF) to find redundant variables.
   3. looking at the correlation matrix of your dataset, or

4. Some coefficients are many orders of magnitude larger than others, and the standard error of the coefficient is also large or there are `nan`’s in the results. This can be seen using the `print_summary` method on a fitted `CoxPHFitter` object.
1. Look for a `ConvergenceWarning` about variances being too small. The dataset may contain a constant column, which provides no information for the regression (Cox model doesn’t have a traditional “intercept” term like other regression models).

2. The data is completely separable, which means that there exists a covariate the completely determines whether an event occurred or not. For example, for all “death” events in the dataset, there exists a covariate that is constant amongst all of them. Look for a `ConvergenceWarning` after the fit call. See https://stats.stackexchange.com/questions/11109/how-to-deal-with-perfect-separation-in-logistic-regression

3. Related to above, the relationship between a covariate and the duration may be completely determined. For example, if the rank correlation between a covariate and the duration is very close to 1 or -1, then the log-likelihood can be increased arbitrarily using just that covariate. Look for a `ConvergenceWarning` after the fit call.

4. Another problem may be a collinear relationship in your dataset. See point 3. above.

5. If adding a very small penalizer significantly changes the results (`CoxPHFitter(penalizer=0.0001)`), then this probably means that the step size in the iterative algorithm is too large. Try decreasing it (`.fit(..., step_size=0.50)` or smaller), and returning the penalizer term to 0.

6. If using the `strata` argument, make sure your stratification group sizes are not too small. Try `df.groupby(strata).size()`.

### 1.9.15 Adding weights to observations in a Cox model

There are two common uses for weights in a model. The first is as a data size reduction technique (known as case weights). If the dataset has more than one subjects with identical attributes, including duration and event, then their likelihood contribution is the same as well. Thus, instead of computing the log-likelihood for each individual, we can compute it once and multiple it by the count of users with identical attributes. In practice, this involves first grouping subjects by covariates and counting. For example, using the Rossi dataset, we will use Pandas to group by the attributes (but other data processing tools, like Spark, could do this as well):

```python
from lifelines.datasets import load_rossi
rossi = load_rossi()
rossi_weights = rossi.copy()
rossi_weights['weights'] = 1.
rossi_weights = rossi_weights.groupby(rossi.columns.tolist())['weights'].sum()\ .reset_index()
```

The original dataset has 432 rows, while the grouped dataset has 387 rows plus an additional `weights` column. `CoxPHFitter` has an additional parameter to specify which column is the weight column.

```python
from lifelines import CoxPHFitter
cph = CoxPHFitter()
cph.fit(rossi_weights, 'week', 'arrest', weights_col='weights')
```

The fitting should be faster, and the results identical to the unweighted dataset. This option is also available in the `CoxTimeVaryingFitter`.

The second use of weights is sampling weights. These are typically positive, non-integer weights that represent some artificial under/over sampling of observations (ex: inverse probability of treatment weights). It is recommended to set `robust=True` in the call to the `fit` as the usual standard error is incorrect for sampling weights. The `robust` flag will use the sandwich estimator for the standard error.
Warning: The implementation of the sandwich estimator does not handle ties correctly (under the Efron handling of ties), and will give slightly or significantly different results from other software depending on the frequency of ties.

1.9.16 Correlations between subjects in a Cox model

There are cases when your dataset contains correlated subjects, which breaks the independent-and-identically-distributed assumption. What are some cases when this may happen?

1. If a subject appears more than once in the dataset (common when subjects can have the event more than once)
2. If using a matching technique, like propensity-score matching, there is a correlation between pairs.

In both cases, the reported standard errors from a unadjusted Cox model will be wrong. In order to adjust for these correlations, there is a `cluster_col` keyword in `CoxPHFitter.fit` that allows you to specify the column in the DataFrame that contains designations for correlated subjects. For example, if subjects in rows 1 & 2 are correlated, but no other subjects are correlated, then `cluster_col` column should have the same value for rows 1 & 2, and all others unique. Another example: for matched pairs, each subject in the pair should have the same value.

```python
from lifelines.datasets import load_rossi
from lifelines import CoxPHFitter

rossi = load_rossi()

# this may come from a database, or other libraries that specialize in matching
matched_pairs = [
    (156, 230),
    (275, 228),
    (61, 252),
    (364, 201),
    (54, 340),
    (130, 33),
    (183, 145),
    (268, 140),
    (332, 259),
    (314, 413),
    (330, 211),
    (372, 255),
    # ...
]

rossi['id'] = None  # we will populate this column

for i, pair in enumerate(matched_pairs):
    subjectA, subjectB = pair
    rossi.loc[subjectA, 'id'] = i
    rossi.loc[subjectB, 'id'] = i

rossi = rossi.dropna(subset=['id'])

cph = CoxPHFitter()
cph.fit(rossi, 'week', 'arrest', cluster_col='id')
```

Specifying `cluster_col` will handle correlations, and invoke the robust sandwich estimator for standard errors (the same as setting `robust=True`).
1.10 Reference library for *lifelines*

1.10.1 *lifelines*.fitters

*lifelines*.fitters.aalen_additive_fitter module

```python
class lifelines.fitters.aalen_additive_fitter.AalenAdditiveFitter (fit_intercept=True, alpha=0.05, coef_penalizer=0.0, smoothing_penalizer=0.0)
```

Bases: *lifelines*.fitters.BaseFitter

This class fits the regression model:

\[
h(t|x) = b_0(t) + b_1(t)x_1 + \ldots + b_N(t)x_N
\]

that is, the hazard rate is a linear function of the covariates with time-varying coefficients. This implementation assumes non-time-varying covariates, see TODO: name

**Parameters**

- **fit_intercept** *(bool, optional (default: True))* – If False, do not attach an intercept (column of ones) to the covariate matrix. The intercept, \(b_0(t)\) acts as a baseline hazard.
- **alpha** *(float, optional (default=0.05))* – the level in the confidence intervals.
- **coef_penalizer** *(float, optional (default: 0))* – Attach a L2 penalizer to the size of the coefficients during regression. This improves stability of the estimates and controls for high correlation between covariates. For example, this shrinks the absolute value of \(c_{i,t}\).
- **smoothing_penalizer** *(float, optional (default: 0))* – Attach a L2 penalizer to difference between adjacent (over time) coefficients. For example, this shrinks the absolute value of \(c_{i,t} - c_{i,t+1}\).

**cumulative_hazards_**

The estimated cumulative hazard  

**Type** DataFrame

**hazards_**

The estimated hazards  

**Type** DataFrame

**confidence_intervals_**

The lower and upper confidence intervals for the cumulative hazard  

**Type** DataFrame

**durations**

The durations provided  

**Type** array
The `event_observed` variable provided

**Type** array

The `weights` variable provided

**Type** array

**fit** *(df, duration_col, event_col=None, weights_col=None, show_progress=False)*

**Parameters** Fit the Aalen Additive model to a dataset.

**Parameters**

- **df** *(DataFrame)* – a Pandas DataFrame with necessary columns `duration_col` and `event_col` (see below), covariates columns, and special columns (weights). `duration_col` refers to the lifetimes of the subjects. `event_col` refers to whether the ‘death’ events was observed: 1 if observed, 0 else (censored).

- **duration_col** *(string)* – the name of the column in DataFrame that contains the subjects’ lifetimes.

- **event_col** *(string, optional)* – the name of the column in DataFrame that contains the subjects’ death observation. If left as None, assume all individuals are uncensored.

- **weights_col** *(string, optional)* – an optional column in the DataFrame, df, that denotes the weight per subject. This column is expelled and not used as a covariate, but as a weight in the final regression. Default weight is 1. This can be used for case-weights. For example, a weight of 2 means there were two subjects with identical observations. This can be used for sampling weights.

- **show_progress** *(boolean, optional (default=False))* – Since the fitter is iterative, show iteration number.

**Returns** self – self with additional new properties: `cumulative_hazards_`, etc.

**Return type** AalenAdditiveFitter

Examples

```python
>>> from lifelines import AalenAdditiveFitter
>>> df = pd.DataFrame({
    'T': [5, 3, 9, 8, 7, 4, 4, 3, 2, 5, 6, 7],
    'E': [1, 1, 1, 1, 1, 0, 0, 1, 1, 1, 1, 0],
    'var': [0, 0, 0, 1, 1, 1, 1, 1, 2, 2, 2, 2],
    'age': [4, 3, 9, 8, 7, 4, 4, 3, 2, 5, 6, 7],
})
>>> aaf = AalenAdditiveFitter()
>>> aaf.fit(df, 'T', 'E')
>>> aaf.predict_median(df)
>>> aaf.print_summary()
```

**fit_right_censoring** *(*args, **kwargs)*

**plot** *(columns=None, loc=None, iloc=None, **kwargs)*

"A wrapper around plotting. Matplotlib plot arguments can be passed in, plus:

**Parameters**
• **columns** *(string or list-like, optional)* – If not empty, plot a subset of columns from the `cumulative_hazards_`. Default all.

• **loc**

• **iloc** *(slice, optional)* –

  *specify a location-based subsection of the curves to plot, ex:*.  
  *plot(iloc=slice(0,10)) will plot the first 10 time points.*

**predict_cumulative_hazard** *(X)*

Returns the hazard rates for the individuals  

**Parameters**  
X *(a (n,d) covariate numpy array or DataFrame)* – If a DataFrame, columns can be in any order. If a numpy array, columns must be in the same order as the training data.

**predict_expectation** *(X)*

Compute the expected lifetime, E[T], using covariates X.  

**Parameters**

• **X** *(a (n,d) covariate numpy array or DataFrame)* – If a DataFrame, columns can be in any order. If a numpy array, columns must be in the same order as the training data.

• **Returns the expected lifetimes for the individuals**

**predict_median** *(X)*

**Parameters**

• **X** *(a (n,d) covariate numpy array or DataFrame)* – If a DataFrame, columns can be in any order. If a numpy array, columns must be in the same order as the training data.

• **Returns the median lifetimes for the individuals**

**predict_percentile** *(X, p=0.5)*

Returns the median lifetimes for the individuals.  

**Parameters**

• **X** *(a (n,d) covariate numpy array or DataFrame)* – If a DataFrame, columns can be in any order. If a numpy array, columns must be in the same order as the training data.

• **p** *(float)* – default: 0.5

**predict_survival_function** *(X)*

Returns the survival functions for the individuals  

**Parameters**

• **X** *(a (n,d) covariate numpy array or DataFrame)* – If a DataFrame, columns can be in any order. If a numpy array, columns must be in the same order as the training data.

**print_summary** *(decimals=2, **kwargs)*

Print summary statistics describing the fit, the coefficients, and the error bounds.  

**Parameters**

• **decimals** *(int, optional (default=2))* – specify the number of decimal places to show

• **kwargs** – print additional meta data in the output (useful to provide model names, dataset names, etc.) when comparing multiple outputs.

**score**

The concordance score (also known as the c-index) of the fit. The c-index is a generalization of the ROC AUC to survival data, including censorships.
For this purpose, the `score_` is a measure of the predictive accuracy of the fitted model onto the training dataset. It’s analogous to the R^2 in linear models.

`smoothed_hazards_ (bandwidth=1)`
Using the epanechnikov kernel to smooth the hazard function, with sigma/bandwidth

`summary`
Summary statistics describing the fit.

Returns df
Return type DataFrame

lifelines.fitters.aalen_johansen_fitter module

class lifelines.fitters.aalen_johansen_fitter.AalenJohansenFitter (jitter_level=0.0001, seed=None, alpha=0.05, calculate_variance=True)

Bases: lifelines.fitters.UnivariateFitter

Class for fitting the Aalen-Johansen estimate for the cumulative incidence function in a competing risks framework. Treating competing risks as censoring can result in over-estimated cumulative density functions. Using the Kaplan Meier estimator with competing risks as censored is akin to estimating the cumulative density if all competing risks had been prevented.

Aalen-Johansen cannot deal with tied times. We can get around this by randomly jittering the event times slightly. This will be done automatically and generates a warning.

Parameters

- `alpha` (`float`, option (default=0.05)) – The alpha value associated with the confidence intervals.
- `jitter_level` (`float`, option (default=0.00001)) – If tied event times are detected, event times are randomly changed by this factor.
- `seed` (`int`, option (default=None)) – To produce replicate results with tied event times, the numpy.random.seed can be specified in the function.
- `calculate_variance` (`bool`, option (default=True)) – By default, AalenJohansenFitter calculates the variance and corresponding confidence intervals. Due to how the variance is calculated, the variance must be calculated for each event time individually. This is computationally intensive. For some procedures, like bootstrapping, the variance is not necessary. To reduce computation time during these procedures, `calculate_variance` can be set to `False` to skip the variance calculation.

Example

```python
>>> from lifelines.datasets import load_waltons
>>> T, E = load_waltons()['T'], load_waltons()['E']
>>> ajf = AalenJohansenFitter(calculate_variance=True)
>>> ajf.fit(T, E)
>>> ajf.cumulative_density_
>>> ajf.plot()
```
References


conditional_time_to_event_

Return a DataFrame, with index equal to survival_function_’s index, that estimates the median duration remaining until the death event, given survival up until time t. For example, if an individual exists until age 1, their expected life remaining given they lived to time 1 might be 9 years.

Returns conditional_time_to_

Return type DataFrame

cumulative_density_at_times(times, label=None)
cumulative_hazard_at_times(times, label=None)
divide(other)

Divide the [0] of two [1] objects.

Parameters other (an [1] fitted instance.)

fit(durations, event_observed, event_of_interest, timeline=None, entry=None, label='AJ_estimate', alpha=None, ci_labels=None, weights=None)

Parameters

• durations (an array or pd.Series of length n – duration of subject was observed for)

• event_observed (an array, or pd.Series, of length n. Integer indicator of distinct events. Must be) – only positive integers, where 0 indicates censoring.

• event_of_interest (integer – indicator for event of interest. All other integers are considered competing events) – Ex) event_observed contains 0, 1, 2 where 0:censored, 1:lung cancer, and 2:death. If event_of_interest=1, then death (2) is considered a competing event. The returned cumulative incidence function corresponds to risk of lung cancer

• timeline (return the best estimate at the values in timelines (positively increasing))

• entry (an array, or pd.Series, of length n – relative time when a subject entered the study. This is) – useful for left-truncated (not left-censored) observations. If None, all members of the population were born at time 0.

• label (a string to name the column of the estimate.)

• alpha (the alpha value in the confidence intervals. Overrides the initializing) – alpha for this call to fit only.

• ci_labels (add custom column names to the generated confidence intervals) – as a length-2 list: [lower-bound name>, <upper-bound name>]. Default: <label>_lower_<1-alpha/2>

• weights (n array, or pd.Series, of length n, if providing a weighted dataset. For example, instead) – of providing every subject as a single element of durations and event_observed, one could weigh subject differently.

Returns self – self, with new properties like cumulative_incidence_.

Return type AalenJohansenFitter

fit_right_censoring(*args, **kwargs)

hazard_at_times(times, label=None)
```python
plot(**kwargs)
```

Plots a pretty figure of the model

Matplotlib plot arguments can be passed in inside the kwargs, plus

**Parameters**

- **show_censors** *(bool)* – place markers at censorship events. Default: False
- **censor_styles** *(dict)* – If show_censors, this dictionary will be passed into the plot call.
- **ci_alpha** *(float)* – the transparency level of the confidence interval. Default: 0.3
- **ci_force_lines** *(bool)* – force the confidence intervals to be line plots (versus default shaded areas). Default: False
- **ci_show** *(bool)* – show confidence intervals. Default: True
- **ci_legend** *(bool)* – if ci_force_lines is True, this is a boolean flag to add the lines’ labels to the legend. Default: False
- **at_risk_counts** *(bool)* – show group sizes at time points. See function `add_at_risk_counts` for details. Default: False
- **loc** *(slice)* – specify a time-based subsection of the curves to plot, ex:

  ```
  >>> model.plot(loc=slice(0.,10.))
  ```

  will plot the time values between t=0. and t=10.

- **iloc** *(slice)* – specify a location-based subsection of the curves to plot, ex:

  ```
  >>> model.plot(iloc=slice(0,10))
  ```

  will plot the first 10 time points.

- **invert_y_axis** *(bool)* – boolean to invert the y-axis, useful to show cumulative graphs instead of survival graphs. (Deprecated, use `plot_cumulative_density()`)  

**Returns** a pyplot axis object

**Return type** `ax`

```python
plot_cumulative_density(**kwargs)
```

```python
plot_cumulative_hazard(**kwargs)
```

```python
plot_hazard(**kwargs)
```

```python
plot_survival_function(**kwargs)
```

```python
predict(times, interpolate=False)
```

Predict the {0} at certain point in time. Uses a linear interpolation if points in time are not in the index.

**Parameters**

- **times** *(scalar, or array)* – a scalar or an array of times to predict the value of {0} at.
- **interpolate** *(boolean, optional (default=False))* – for methods that produce a stepwise solution (Kaplan-Meier, Nelson-Aalen, etc), turning this to True will use an linear interpolation method to provide a more “smooth” answer.

**Returns** predictions

**Return type** a scalar if time is a scalar, a numpy array if time in an array.
subtract (other)
    Subtract the {0} of two {1} objects.

Parameters other (an {1} fitted instance.)

survival_function_at_times (times, label=None)

lifelines.fitters.breslow_fleming_harrington_fitter module

class lifelines.fitters.breslow_fleming_harrington_fitter.BreslowFlemingHarringtonFitter (alpha=0.05)

Bases: lifelines.fitters.UnivariateFitter

Class for fitting the Breslow-Fleming-Harrington estimate for the survival function. This estimator is a biased estimator of the survival function but is more stable when the population is small and there are too few early truncation times, it may happen that is the number of patients at risk and the number of deaths is the same. Mathematically, the NAF estimator is the negative logarithm of the BFH estimator.

BreslowFlemingHarringtonFitter(alpha=0.05)

Parameters alpha (float, optional (default=0.05)) – The alpha value associated with the confidence intervals.

conditional_time_to_event_
    Return a DataFrame, with index equal to survival_function_’s index, that estimates the median duration remaining until the death event, given survival up until time t. For example, if an individual exists until age 1, their expected life remaining given they lived to time 1 might be 9 years.

Returns conditional_time_to_

Return type DataFrame

cumulative_density_at_times (times, label=None)

cumulative_hazard_at_times (times, label=None)

divide (other)
    Divide the {0} of two {1} objects.

Parameters other (an {1} fitted instance.)

fit (durations, event_observed=None, timeline=None, entry=None, label='BFH_estimate', alpha=None, ci_labels=None)

Parameters
    • durations (an array, or pd.Series, of length n) – duration subject was observed for
    • timeline – return the best estimate at the values in timelines (positively increasing)
    • event_observed (an array, or pd.Series, of length n) – True if the the death was observed, False if the event was lost (right-censored). Defaults all True if event_observed==None
    • entry (an array, or pd.Series, of length n) – relative time when a subject entered the study. This is useful for left-truncated observations, i.e the birth event was not observed. If None, defaults to all 0 (all birth events observed.)
    • label (string) – a string to name the column of the estimate.
    • alpha (float, optional (default=0.05)) – the alpha value in the confidence intervals. Over-rides the initializing alpha for this call to fit only.
• **ci_labels** (*iterable*) – add custom column names to the generated confidence intervals as a length-2 list: [<lower-bound name>, <upper-bound name>]. Default: `<label>_lower_<alpha>`

**Returns**

**Return type**  self, with new properties like `survival_function_`

---

```python
fit_right_censoring(*args, **kwargs)
```

```python
hazard_at_times(times, label=None)
```

```python
plot(**kwargs)
```

Plots a pretty figure of the model

Matplotlib plot arguments can be passed in inside the kwargs, plus

**Parameters**

- **show_censors** (*bool*) – place markers at censorship events. Default: False
- **censor_styles** (*dict*) – If show_censors, this dictionary will be passed into the plot call.
- **ci_alpha** (*float*) – the transparency level of the confidence interval. Default: 0.3
- **ci_force_lines** (*bool*) – force the confidence intervals to be line plots (versus default shaded areas). Default: False
- **ci_show** (*bool*) – show confidence intervals. Default: True
- **ci_legend** (*bool*) – if `ci_force_lines` is True, this is a boolean flag to add the lines’ labels to the legend. Default: False
- **at_risk_counts** (*bool*) – show group sizes at time points. See function `add_at_risk_counts` for details. Default: False
- **loc** (*slice*) – specify a time-based subsection of the curves to plot, ex:

```python
>>> model.plot(loc=slice(0., 10.))
```

will plot the time values between t=0. and t=10.

- **iloc** (*slice*) – specify a location-based subsection of the curves to plot, ex:

```python
>>> model.plot(iloc=slice(0, 10))
```

will plot the first 10 time points.

- **invert_y_axis** (*bool*) – boolean to invert the y-axis, useful to show cumulative graphs instead of survival graphs. (Deprecated, use `plot_cumulative_density()`)

**Returns**  a pyplot axis object

**Return type**  ax

---

```python
plot_cumulative_density(**kwargs)
```

```python
plot_cumulative_hazard(**kwargs)
```

```python
plot_hazard(**kwargs)
```

```python
plot_survival_function(**kwargs)
```

```python
predict(times, interpolate=False)
```

Predict the `{0}` at certain point in time. Uses a linear interpolation if points in time are not in the index.

**Parameters**

- **times** *(scalar, or array)* – a scalar or an array of times to predict the value of \( t \) at.
- **interpolate** *(boolean, optional (default=False))* – for methods that produce a stepwise solution (Kaplan-Meier, Nelson-Aalen, etc), turning this to True will use an linear interpolation method to provide a more “smooth” answer.

**Returns** predictions

**Return type** a scalar if time is a scalar, a numpy array if time in an array.

**subtract** *(other)*

Subtract the \( t \) of two \( t \) objects.

**Parameters** other *(an \( t \) fitted instance.)*

**survival_function_at_times** *(times, label=None)*

Return a Pandas series of the predicted survival value at specific times

**Parameters** times *(iterable or float)*

**Returns**

**Return type** pd.Series

### lifelines.fitters.cox_time_varying_fitter module

```python
class lifelines.fitters.cox_time_varying_fitter.CoxTimeVaryingFitter(
    alpha=0.05,
    penalizer=0.0,
    strata=None
)
```

**Bases:** lifelines.fitters.BaseFitter

This class implements fitting Cox’s time-varying proportional hazard model:

\[
h(t|x(t)) = h_0(t) \exp(x(t)^T \beta)\]

**Parameters**

- **alpha** *(float, optional (default=0.05))* – the level in the confidence intervals.
- **penalizer** *(float, optional)* – the coefficient of an L2 penalizer in the regression

**hazards_**

The estimated hazards

**Type** Series

**confidence_intervals_**

The lower and upper confidence intervals for the hazard coefficients

**Type** DataFrame

**event_observed**

The event_observed variable provided

**Type** Series

**weights**

The event_observed variable provided

**Type** Series
variance_matrix
  The variance matrix of the coefficients
  Type  numpy array
strata
  the strata provided
  Type  list
standard_errors
  the standard errors of the estimates
  Type  Series
score
  the concordance index of the model.
  Type  float
baseline_cumulative_hazard
  Type  DataFrame
baseline_survival
  Type  DataFrame

fit(df, id_col, event_col='start', stop_col='stop', weights_col=None, show_progress=False, step_size=None, robust=False, strata=None, initial_point=None)
Fit the Cox Proportional Hazard model to a time varying dataset. Tied survival times are handled using Efron’s tie-method.

Parameters

• df (DataFrame) – a Pandas DataFrame with necessary columns duration_col and event_col, plus other covariates. duration_col refers to the lifetimes of the subjects. event_col refers to whether the ‘death’ events was observed: 1 if observed, 0 else (censored).
• id_col (string) – A subject could have multiple rows in the DataFrame. This column contains the unique identifier per subject.
• event_col (string) – the column in DataFrame that contains the subjects’ death observation. If left as None, assume all individuals are non-censored.
• start_col (string) – the column that contains the start of a subject’s time period.
• stop_col (string) – the column that contains the end of a subject’s time period.
• weights_col (string, optional) – the column that contains (possibly time-varying) weight of each subject-period row.
• show_progress (since the fitter is iterative, show convergence) – diagnostics.
• robust (boolean, optional (default: True)) – Compute the robust errors using the Huber sandwich estimator, aka Wei-Lin estimate. This does not handle ties, so if there are high number of ties, results may significantly differ. See “The Robust Inference for the Cox Proportional Hazards Model”, Journal of the American Statistical Association, Vol. 84, No. 408 (Dec., 1989), pp. 1074- 1078
• step_size (float, optional) – set an initial step size for the fitting algorithm.
• **strata** *(list or string, optional)* – specify a column or list of columns to use in stratification. This is useful if a categorical covariate does not obey the proportional hazard assumption. This is used similar to the `strata` expression in R. See [http://courses.washington.edu/b515/l17.pdf](http://courses.washington.edu/b515/l17.pdf).

• **initial_point** *(d,) numpy array, optional* – initialize the starting point of the iterative algorithm. Default is the zero vector.

Returns **self** – self, with additional properties like `hazards_` and `print_summary`

Return type **CoxTimeVaryingFitter**

**fit_right_censoring** *(*args, **kwargs)*

This function computes the likelihood ratio test for the Cox model. We compare the existing model (with all the covariates) to the trivial model of no covariates.

Conveniently, we can actually use CoxPHFitter class to do most of the work.

**plot** *(columns=None, **errorbar_kwargs)*

Produces a visual representation of the coefficients, including their standard errors and magnitudes.

Parameters

• **columns** *(list, optional)* – specify a subset of the columns to plot

• **errorbar_kwargs** – pass in additional plotting commands to matplotlib errorbar command

Returns **ax** – the matplotlib axis that be edited.

Return type **matplotlib axis**

**predict_log_partial_hazard** *(X)*

This is equivalent to R’s `linear.predictors`. Returns the log of the partial hazard for the individuals, partial since the baseline hazard is not included. Equal to :math:`(x - \bar{x})'\beta`

Parameters **X** *(numpy array or DataFrame)* – a (n,d) covariate numpy array or DataFrame. If a DataFrame, columns can be in any order. If a numpy array, columns must be in the same order as the training data.

Returns **DataFrame**

Note: If X is a DataFrame, the order of the columns do not matter. But if X is an array, then the column ordering is assumed to be the same as the training dataset.

**predict_partial_hazard** *(X)*

Returns the partial hazard for the individuals, partial since the baseline hazard is not included. Equal to :math:`\exp(x - \bar{x})'\beta`

Parameters **X** *(numpy array or DataFrame)* – a (n,d) covariate numpy array or DataFrame. If a DataFrame, columns can be in any order. If a numpy array, columns must be in the same order as the training data.

Returns **DataFrame**

1.10. Reference library for *lifelines*
**Note:** If X is a DataFrame, the order of the columns do not matter. But if X is an array, then the column ordering is assumed to be the same as the training dataset.

```python
print_summary(decimals=2, **kwargs)
```

Print summary statistics describing the fit, the coefficients, and the error bounds.

**Parameters**

- **decimals** *(int, optional (default=2)) – specify the number of decimal places to show*
- **kwargs** – print additional meta data in the output (useful to provide model names, dataset names, etc.) when comparing multiple outputs.

**summary**

Summary statistics describing the fit.

**Returns**

- **df** – contains columns coef, exp(coef), se(coef), z, p, lower, upper
- **Return type** DataFrame

---

 lifelines.fitters.coxph_fitter module

```python
class lifelines.fitters.coxph_fitter.CoxPHFitter(alpha=0.05, tie_method='Efron', penalizer=0.0, strata=None)
```

**Bases:** lifelines.fitters.BaseFitter

This class implements fitting Cox’s proportional hazard model:

\[
h(t|x) = h_0(t) \exp((x - \bar{x})'\beta)
\]

**Parameters**

- **alpha** *(float, optional (default=0.05)) – the level in the confidence intervals.*
- **tie_method** *(string, optional) – specify how the fitter should deal with ties. Currently only ‘Efron’ is available.*
- **penalizer** *(float, optional (default=0.0)) – Attach an L2 penalizer to the size of the coefficients during regression. This improves stability of the estimates and controls for high correlation between covariates. For example, this shrinks the absolute value of \( \beta_i \). The penalty is \( \frac{1}{2} \text{penalizer} ||\beta||^2 \).*
- **strata** *(list, optional) – specify a list of columns to use in stratification. This is useful if a categorical covariate does not obey the proportional hazard assumption. This is used similar to the strata expression in R. See http://courses.washington.edu/b515/l17.pdf.*

**Examples**

```python
>>> from lifelines.datasets import load_rossi
>>> from lifelines import CoxPHFitter
>>> rossi = load_rossi()
>>> cph = CoxPHFitter()
>>> cph.fit(rossi, 'week', 'arrest')
>>> cph.print_summary()
```
hazards_  
The estimated hazards
Type  Series

confidence_intervals_  
The lower and upper confidence intervals for the hazard coefficients
Type  DataFrame

durations  
The durations provided
Type  Series
event_observed  
The event_observed variable provided
Type  Series

weights  
The event_observed variable provided
Type  Series

variance_matrix_  
The variance matrix of the coefficients
Type  numpy array

strata  
the strata provided
Type  list

standard_errors_  
the standard errors of the estimates
Type  Series

score_  
the concordance index of the model.
Type  float

baseline_hazard_  
Type  DataFrame

baseline_cumulative_hazard_  
Type  DataFrame

baseline_survival_  
Type  DataFrame

check_assumptions  
Use this function to test the proportional hazards assumption. See usage example at https://lifelines.readthedocs.io/en/latest/jupyter_notebooks/Proportional%20hazard%20assumption.html

Parameters

- training_df (DataFrame) – the original DataFrame used in the call to `fit(...)` or a sub-sampled version.
- advice (boolean, optional) – display advice as output to the user’s screen
• **show_plots** *(boolean, optional)* – display plots of the scaled schoenfeld residuals and loess curves. This is an eyeball test for violations. This will slow down the function significantly.

• **p_value_threshold** *(float, optional)* – the threshold to use to alert the user of violations. See note below.

• **plot_n_bootstraps** – in the plots displayed, also display plot_n_bootstraps bootstrapped loess curves. This will slow down the function significantly.

### Examples

```python
>>> from lifelines.datasets import load_rossi
>>> from lifelines import CoxPHFitter

>>> rossi = load_rossi()
>>> cph = CoxPHFitter().fit(rossi, 'week', 'arrest')
>>> cph.check_assumptions(rossi)
```

### Notes

The **p_value_threshold** is arbitrarily set at 0.01. Under the null, some covariates will be below the threshold (i.e. by chance). This is compounded when there are many covariates.

Similarly, when there are lots of observations, even minor deviances from the proportional hazard assumption will be flagged.

With that in mind, it’s best to use a combination of statistical tests and eyeball tests to determine the most serious violations.

### References

- [http://eprints.lse.ac.uk/84988/1/06_ParkHendry2015-ReassessingSchoenfeldTests_Final.pdf](http://eprints.lse.ac.uk/84988/1/06_ParkHendry2015-ReassessingSchoenfeldTests_Final.pdf)

### compute_residuals *(training_dataframe, kind)*

#### Parameters

- **training_dataframe** *(pandas DataFrame)* – the same training DataFrame given in *fit*


#### fit *(df, duration_col=None, event_col=None, show_progress=False, initial_point=None, strata=None, step_size=None, weights_col=None, cluster_col=None, robust=False, batch_mode=None)*

Fit the Cox proportional hazard model to a dataset.

#### Parameters

- **df** *(DataFrame)* – a Pandas DataFrame with necessary columns duration_col and event_col (see below), covariates columns, and special columns (weights, strata). duration_col refers to the lifetimes of the subjects. event_col refers to whether the ‘death’ events was observed: 1 if observed, 0 else (censored).

- **duration_col** *(string)* – the name of the column in DataFrame that contains the subjects’ lifetimes.
• **event_col** *(string, optional)* – the name of the column in DataFrame that contains the subjects’ death observation. If left as None, assume all individuals are uncensored.

• **weights_col** *(string, optional)* – an optional column in the DataFrame, df, that denotes the weight per subject. This column is expelled and not used as a covariate, but as a weight in the final regression. Default weight is 1. This can be used for case-weights. For example, a weight of 2 means there were two subjects with identical observations. This can be used for sampling weights. In that case, use robust=True to get more accurate standard errors.

• **show_progress** *(boolean, optional (default=False))* – since the fitter is iterative, show convergence diagnostics. Useful if convergence is failing.

• **initial_point** *((d, ) numpy array, optional)* – initialize the starting point of the iterative algorithm. Default is the zero vector.

• **strata** *(list or string, optional)* – specify a column or list of columns to use in stratification. This is useful if a categorical covariate does not obey the proportional hazard assumption. This is used similar to the strata expression in R. See http://courses.washington.edu/b515/l17.pdf.

• **step_size** *(float, optional)* – set an initial step size for the fitting algorithm. Setting to 1.0 may improve performance, but could also hurt convergence.

• **robust** *(boolean, optional (default=False))* – Compute the robust errors using the Huber sandwich estimator, aka Wei-Lin estimate. This does not handle ties, so if there are high number of ties, results may significantly differ. See “The Robust Inference for the Cox Proportional Hazards Model”, Journal of the American Statistical Association, Vol. 84, No. 408 (Dec., 1989), pp. 1074- 1078

• **cluster_col** *(string, optional)* – specifies what column has unique identifiers for clustering covariances. Using this forces the sandwich estimator (robust variance estimator) to be used.

• **batch_mode** *(bool, optional)* – enabling batch_mode can be faster for datasets with a large number of ties. If left as None, lifelines will choose the best option.

Returns **self** – self with additional new properties: print_summary, hazards_, confidence_intervals_, baseline_survival_, etc.

**Return type** *CoxPHFitter*

**Note:** Tied survival times are handled using Efron’s tie-method.

**Examples**

```python
>>> from lifelines import CoxPHFitter

>>> df = pd.DataFrame({
    'T': [5, 3, 9, 8, 7, 4, 4, 3, 2, 5, 6, 7],
    'E': [1, 1, 1, 1, 1, 0, 0, 1, 1, 1, 1, 0],
    'var': [0, 0, 0, 0, 1, 1, 1, 1, 1, 2, 2, 2],
    'age': [4, 3, 9, 8, 7, 4, 4, 3, 2, 5, 6, 7],
})

>>> cph = CoxPHFitter()
>>> cph.fit(df, 'T', 'E')
```

(continues on next page)
>>> cph.print_summary()
>>> cph.predict_median(df)

>>> from lifelines import CoxPHFitter

>>> df = pd.DataFrame({
    'T': [5, 3, 9, 8, 4, 4, 3, 2, 5, 6, 7],
    'E': [1, 1, 1, 1, 0, 0, 1, 1, 1, 1, 1],
    'var': [0, 0, 0, 0, 1, 1, 1, 1, 2, 2, 2],
    'weights': [1.1, 0.5, 2.0, 1.6, 1.2, 4.3, 1.4, 4.5, 3.0, 3.2, 0.4, 6.2],
    'month': [10, 3, 9, 8, 7, 4, 4, 3, 2, 5, 6, 7],
    'age': [4, 3, 9, 8, 7, 4, 4, 3, 2, 5, 6, 7],
})

>>> cph = CoxPHFitter()
>>> cph.fit(df, 'T', 'E', strata=['month', 'age'], robust=True, weights_col='weights')
>>> cph.print_summary()
>>> cph.predict_median(df)

fit_right_censoring(*args, **kwargs)

log_likelihood_ratio_test()
This function computes the likelihood ratio test for the Cox model. We compare the existing model (with all the covariates) to the trivial model of no covariates.

plot (columns=None, **errorbar_kwargs)
Produces a visual representation of the coefficients, including their standard errors and magnitudes.

Parameters

- **columns** (list, optional) – specify a subset of the columns to plot

- **errorbar_kwargs** – pass in additional plotting commands to matplotlib errorbar command

Returns ax – the matplotlib axis that be edited.

Return type matplotlib axis

plot_covariate_groups (covariates, values, plot_baseline=True, **kwargs)
Produces a visual representation comparing the baseline survival curve of the model versus what happens when a covariate(s) is varied over values in a group. This is useful to compare subjects’ survival as we vary covariate(s), all else being held equal. The baseline survival curve is equal to the predicted survival curve at all average values in the original dataset.

Parameters

- **covariates** (string or list) – a string (or list of strings) of the covariate(s) in the original dataset that we wish to vary.

- **values** (1d or 2d iterable) – an iterable of the values we wish the covariate(s) to take on.

- **plot_baseline** (bool) – also display the baseline survival, defined as the survival at the mean of the original dataset.

- **kwargs** – pass in additional plotting commands.

Returns ax – the matplotlib axis that be edited.

Return type matplotlib axis, or list of axis’
Examples

```python
>>> from lifelines import datasets, CoxPHFitter
>>> rossi = datasets.load_rossi()
>>> cph = CoxPHFitter().fit(rossi, 'week', 'arrest')
>>> cph.plot_covariate_groups('prio', values=np.arange(0, 15), cmap='coolwarm')

>>> # multiple variables at once
>>> cph.plot_covariate_groups(['prio', 'paro'], values=[[0, 0], [5, 0], [10, 0], [0, 1], [5, 1], [10, 1]], cmap='coolwarm')

>>> # if you have categorical variables, you can simply things:
>>> cph.plot_covariate_groups(['dummy1', 'dummy2', 'dummy3'], values=np.eye(3))
```

`predict_cumulative_hazard(X, times=None)`

Parameters

- **X** *(numpy array or DataFrame)* – a (n,d) covariate numpy array or DataFrame. If a DataFrame, columns can be in any order. If a numpy array, columns must be in the same order as the training data.
- **times** *(iterable, optional)* – an iterable of increasing times to predict the cumulative hazard at. Default is the set of all durations (observed and unobserved). Uses a linear interpolation if points in time are not in the index.

Returns **cumulative_hazard_** – the cumulative hazard of individuals over the timeline

Return type DataFrame

`predict_expectation(X)`

Compute the expected lifetime, \( \mathbb{E}[T] \), using covariates X. This algorithm to compute the expectation is to use the fact that \( \mathbb{E}[T] = \int_0^\infty P(T > t) dt = \int_0^\infty S(t) dt \). To compute the integral, we use the trapizoidal rule to approximate the integral.

**Caution:** However, if the survival function doesn’t converge to 0, the the expectation is really infinity and the returned values are meaningless/too large. In that case, using `predict_median` or `predict_percentile` would be better.

Parameters **X** *(numpy array or DataFrame)* – a (n,d) covariate numpy array or DataFrame. If a DataFrame, columns can be in any order. If a numpy array, columns must be in the same order as the training data.

Returns **expectations**

Return type DataFrame

Notes

If X is a DataFrame, the order of the columns do not matter. But if X is an array, then the column ordering is assumed to be the same as the training dataset.

See also:

- `predict_median()`, `predict_percentile()`
**predict_log_partial_hazard** \((X)\)

This is equivalent to R’s `linear.predictors`. Returns the log of the partial hazard for the individuals, partial since the baseline hazard is not included. Equal to :math:`\ilog{(x - \text{mean}(x_{\text{train}}))'\beta}`

**Parameters**

- \(X\) *(numpy array or DataFrame)* – a \((n,d)\) covariate numpy array or DataFrame. If a DataFrame, columns can be in any order. If a numpy array, columns must be in the same order as the training data.

**Returns**

log_partial_hazard

**Return type**

DataFrame

**Notes**

If \(X\) is a DataFrame, the order of the columns do not matter. But if \(X\) is an array, then the column ordering is assumed to be the same as the training dataset.

**predict_median** \((X)\)

Predict the median lifetimes for the individuals. If the survival curve of an individual does not cross 0.5, then the result is infinity.

**Parameters**

- \(X\) *(numpy array or DataFrame)* – a \((n,d)\) covariate numpy array or DataFrame. If a DataFrame, columns can be in any order. If a numpy array, columns must be in the same order as the training data.

**Returns**

percentiles – the median lifetimes for the individuals. If the survival curve of an individual does not cross 0.5, then the result is infinity.

**Return type**

DataFrame

**See also:**

* predict_percentile *

**predict_partial_hazard** \((X)\)

**Parameters**

- \(X\) *(numpy array or DataFrame)* – a \((n,d)\) covariate numpy array or DataFrame. If a DataFrame, columns can be in any order. If a numpy array, columns must be in the same order as the training data.

**Returns**

partial_hazard – Returns the partial hazard for the individuals, partial since the baseline hazard is not included. Equal to \(\exp{(x - \text{mean}(x_{\text{train}}))'\beta}\)

**Return type**

DataFrame

**Notes**

If \(X\) is a DataFrame, the order of the columns do not matter. But if \(X\) is an array, then the column ordering is assumed to be the same as the training dataset.

**predict_percentile** \((X, p=0.5)\)

Returns the median lifetimes for the individuals, by default. If the survival curve of an individual does not cross 0.5, then the result is infinity. [http://stats.stackexchange.com/questions/102986/percentile-loss-functions](http://stats.stackexchange.com/questions/102986/percentile-loss-functions)

**Parameters**

- \(X\) *(numpy array or DataFrame)* – a \((n,d)\) covariate numpy array or DataFrame. If a DataFrame, columns can be in any order. If a numpy array, columns must be in the same order as the training data.
• p (float, optional (default=0.5)) – the percentile, must be between 0 and 1.

Returns percentiles
Return type DataFrame

See also:

predict_median()

predict_survival_function (X, times=None)
Predict the survival function for individuals, given their covariates. This assumes that the individual just entered the study (that is, we do not condition on how long they have already lived for.)

Parameters

• X (numpy array or DataFrame) – a (n,d) covariate numpy array or DataFrame. If a DataFrame, columns can be in any order. If a numpy array, columns must be in the same order as the training data.

• times (iterable, optional) – an iterable of increasing times to predict the cumulative hazard at. Default is the set of all durations (observed and unobserved). Uses a linear interpolation if points in time are not in the index.

Returns survival_function – the survival probabilities of individuals over the timeline

Return type DataFrame

print_summary (decimals=2, **kwargs)
Print summary statistics describing the fit, the coefficients, and the error bounds.

Parameters

• decimals (int, optional (default=2)) – specify the number of decimal places to show

• kwargs – print additional metadata in the output (useful to provide model names, dataset names, etc.) when comparing multiple outputs.

score_
The concordance score (also known as the c-index) of the fit. The c-index is a generalization of the ROC AUC to survival data, including censorships.

For this purpose, the score_ is a measure of the predictive accuracy of the fitted model onto the training dataset.

References

https://stats.stackexchange.com/questions/133817/stratified-concordance-index-survivalsurvconcordance

summary
Summary statistics describing the fit. Set alpha property in the object before calling.

Returns df – Contains columns coef, np.exp(coef), se(coef), z, p, lower, upper

Return type DataFrame

lifelines.fitters.exponential_fitter module

class lifelines.fitters.exponential_fitter.ExponentialFitter(*args, **kwargs)
Bases: lifelines.fitters.KnownModelParametericUnivariateFitter

1.10. Reference library for lifelines
This class implements an Exponential model for univariate data. The model has parameterized form:

\[ S(t) = \exp\left(-\frac{t}{\lambda}\right), \lambda > 0 \]

which implies the cumulative hazard rate is

\[ H(t) = \frac{t}{\lambda} \]

and the hazard rate is:

\[ h(t) = \frac{1}{\lambda} \]

After calling the `.fit` method, you have access to properties like: `survival_function_`, `lambda_`, `cumulative_hazard_`. A summary of the fit is available with the method `print_summary()`

**Parameters** alpha (float, optional (default=0.05)) – the level in the confidence intervals.

---

**Important:** The parameterization of this model changed in lifelines 0.19.0. Previously, the cumulative hazard looked like \( \lambda t \). The parameterization is now the reciprocal of \( \lambda \).
**lambda_**

The fitted parameter in the model

*Type* float

**durations**

The durations provided

*Type* array

**event_observed**

The event_observed variable provided

*Type* array

**timeline**

The time line to use for plotting and indexing

*Type* array

**entry**

The entry array provided, or None

*Type* array or None

**cumulative_density_**

The estimated cumulative density function (with custom timeline if provided)

*Type* DataFrame

**confidence_interval_cumulative_density_**

The lower and upper confidence intervals for the cumulative density

*Type* DataFrame

**conditional_time_to_event_**

Return a DataFrame, with index equal to `survival_function_`’s index, that estimates the median duration remaining until the death event, given survival up until time \( t \). For example, if an individual exists until age 1, their expected life remaining *given they lived to time 1* might be 9 years.

*Returns* `conditional_time_to_`

*Return type* DataFrame

**confidence_interval_**

The confidence interval of the cumulative hazard. This is an alias for `confidence_interval_cumulative_hazard_`

**confidence_interval_cumulative_density_**

The confidence interval of the cumulative density

**confidence_interval_cumulative_hazard_**

The confidence interval of the cumulative hazard. This is an alias for `confidence_interval_`

**confidence_interval_hazard_**

The confidence interval of the hazard

**confidence_interval_survival_function_**

The confidence interval of the survival function.

**cumulative_density_at_times** *(times, label=None)*

Return a Pandas series of the predicted cumulative density function (1-survival function) at specific times.

*Parameters*

- **times** *(iterable or float)* – values to return the survival function at.
• **label** *(string, optional)* – Rename the series returned. Useful for plotting.

**Returns**

**Return type** `pd.Series`

`cumulative_hazard_at_times` *(times, label=None)*

Return a Pandas series of the predicted cumulative hazard value at specific times.

**Parameters**

• **times** *(iterable or float)* – values to return the cumulative hazard at.

• **label** *(string, optional)* – Rename the series returned. Useful for plotting.

**Returns**

**Return type** `pd.Series`

`divide` *(other)*

Divide the `{0}` of two `{1}` objects.

**Parameters**

other *(an `{1}` fitted instance.)*

`fit` *(durations, event_observed=None, timeline=None, label=None, alpha=None, ci_labels=None, show_progress=False, entry=None, weights=None, left_censorship=False)*

**Parameters**

• **durations** *(an array, or pd.Series)* – length n, duration subject was observed for

• **event_observed** *(numpy array or pd.Series, optional)* – length n, True if the the death was observed, False if the event was lost (right-censored). Defaults all True if `event_observed==None`

• **timeline** *(list, optional)* – return the estimate at the values in timeline (positively increasing)

• **label** *(string, optional)* – a string to name the column of the estimate.

• **alpha** *(float, optional)* – the alpha value in the confidence intervals. Overrides the initializing alpha for this call to fit only.

• **ci_labels** *(list, optional)* – add custom column names to the generated confidence intervals as a length-2 list: `[<lower-bound name>, <upper-bound name>]`. Default: `<label>_lower_<alpha>`

• **show_progress** *(boolean, optional)* – since this is an iterative fitting algorithm, switching this to True will display some iteration details.

• **entry** *(an array, or pd.Series, of length n)* – relative time when a subject entered the study. This is useful for left-truncated (not left-censored) observations. If None, all members of the population entered study when they were “born”: time zero.

• **weights** *(an array, or pd.Series, of length n)* – integer weights per observation

**Returns**

self with new properties like `cumulative_hazard`, `survival_function_`

**Return type** `self`

`fit_interval_censoring` *(lower_bound, upper_bound, event_observed=None, timeline=None, label=None, alpha=None, ci_labels=None, show_progress=False, entry=None, weights=None)*

Fit the model to an interval censored dataset.

**Parameters**
• **lower_bound** *(an array, or pd.Series)* – length n, the start of the period the subject experienced the event in.

• **upper_bound** *(an array, or pd.Series)* – length n, the end of the period the subject experienced the event in. If the value is equal to the corresponding value in lower_bound, then the individual’s event was observed (not censored).

• **event_observed** *(numpy array or pd.Series, optional)* – length n, if left optional, infer from lower_bound and upper_cound (if lower_bound==upper_bound then event observed, if lower_bound < upper_bound, then event censored)

• **timeline** *(list, optional)* – return the estimate at the values in timeline (positively increasing)

• **label** *(string, optional)* – a string to name the column of the estimate.

• **alpha** *(float, optional)* – the alpha value in the confidence intervals. Overrides the initializing alpha for this call to fit only.

• **ci_labels** *(list, optional)* – add custom column names to the generated confidence intervals as a length-2 list: [<lower-bound name>, <upper-bound name>]. Default: <label>_lower_<alpha>

• **show_progress** *(boolean, optional)* – since this is an iterative fitting algorithm, switching this to True will display some iteration details.

• **entry** *(an array, or pd.Series, of length n)* – relative time when a subject entered the study. This is useful for left-truncated (not left-censored) observations. If None, all members of the population entered study when they were “born”: time zero.

• **weights** *(an array, or pd.Series, of length n)* – integer weights per observation

**Returns** self with new properties like cumulative_hazard_, survival_function_

**Return type** self

**fit_left_censoring** *(durations, event_observed=None, timeline=None, label=None, alpha=None, ci_labels=None, show_progress=False, entry=None, weights=None)*

Fit the model to a left-censored dataset

**Parameters**

• **durations** *(an array, or pd.Series)* – length n, duration subject was observed for

• **event_observed** *(numpy array or pd.Series, optional)* – length n, True if the the death was observed, False if the event was lost (right-censored). Defaults all True if event_observed==None

• **timeline** *(list, optional)* – return the estimate at the values in timeline (positively increasing)

• **label** *(string, optional)* – a string to name the column of the estimate.

• **alpha** *(float, optional)* – the alpha value in the confidence intervals. Overrides the initializing alpha for this call to fit only.

• **ci_labels** *(list, optional)* – add custom column names to the generated confidence intervals as a length-2 list: [<lower-bound name>, <upper-bound name>]. Default: <label>_lower_<alpha>

• **show_progress** *(boolean, optional)* – since this is an iterative fitting algorithm, switching this to True will display some iteration details.
• **entry** *(an array, or pd.Series, of length n)* – relative time when a subject entered the study. This is useful for left-truncated (not left-censored) observations. If None, all members of the population entered study when they were “born”: time zero.

• **weights** *(an array, or pd.Series, of length n)* – integer weights per observation

**Returns** self with new properties like `cumulative_hazard_`, `survival_function_`

**Return type** self

```python
fit_right_censoring(*args, **kwargs)
```

```python
hazard_at_times(times, label=None)
```

Return a Pandas series of the predicted hazard at specific times.

**Parameters**

- **times** *(iterable or float)* – values to return the hazard at.
- **label** *(string, optional)* – Rename the series returned. Useful for plotting.

**Returns**

**Return type** pd.Series

```python
median_
```

Return the unique time point, t, such that S(t) = 0.5. This is the “half-life” of the population, and a robust summary statistic for the population, if it exists.

```python
plot(**kwargs)
```

Produce a pretty-plot of the estimate.

```python
plot_cumulative_density(**kwargs)
```

```python
plot_cumulative_hazard(**kwargs)
```

```python
plot_hazard(**kwargs)
```

```python
plot_survival_function(**kwargs)
```

```python
predict(times, interpolate=False)
```

Predict the {0} at certain point in time. Uses a linear interpolation if points in time are not in the index.

**Parameters**

- **times** *(scalar, or array)* – a scalar or an array of times to predict the value of {0} at.
- **interpolate** *(boolean, optional (default=False))* – for methods that produce a stepwise solution (Kaplan-Meier, Nelson-Aalen, etc), turning this to True will use an linear interpolation method to provide a more “smooth” answer.

**Returns predictions**

**Return type** a scalar if time is a scalar, a numpy array if time in an array.

```python
print_summary(decimals=2, **kwargs)
```

Print summary statistics describing the fit, the coefficients, and the error bounds.

**Parameters**

- **decimals** *(int, optional (default=2))* – specify the number of decimal places to show
- **kwargs** – print additional metadata in the output (useful to provide model names, dataset names, etc.) when comparing multiple outputs.

```python
subtract(other)
```

Subtract the {0} of two {1} objects.
Parameters other (an fitted instance.)

summary
Summary statistics describing the fit.

Returns df – Contains columns coef, exp(coef), se(coef), z, p, lower, upper

Return type pd.DataFrame

See also:

print_summary
survival_function_at_times (times, label=None)
Return a Pandas series of the predicted survival value at specific times.

Parameters

• times (iterable or float) – values to return the survival function at.

• label (string, optional) – Rename the series returned. Useful for plotting.

Returns

Return type pd.Series

lifelines.fitters.kaplan_meier_fitter module

class lifelines.fitters.kaplan_meier_fitter.KaplanMeierFitter (alpha=0.05)
Bases: lifelines.fitters.UnivariateFitter

Class for fitting the Kaplan-Meier estimate for the survival function.

Parameters alpha (float, option (default=0.05)) – The alpha value associated with the confidence intervals.

Examples

```python
>>> from lifelines import KaplanMeierFitter
>>> from lifelines.datasets import load_waltons
>>> waltons = load_waltons()
>>> kmf = KaplanMeierFitter()
>>> kmf.fit(waltons['T'], waltons['E'])
>>> kmf.plot()
```

survival_function_
The estimated survival function (with custom timeline if provided)

Type DataFrame

median_
The estimated median time to event. np.inf if doesn’t exist.

Type float

confidence_interval_
The lower and upper confidence intervals for the survival function. An alias of

confidence_interval_survival_function_

Type DataFrame
confidence_interval_survival_function_
   The lower and upper confidence intervals for the survival function. An alias of confidence_interval_
   Type DataFrame

cumulative_density_
   The estimated cumulative density function (with custom timeline if provided)
   Type DataFrame

confidence_interval_cumulative_density_
   The lower and upper confidence intervals for the cumulative density
   Type DataFrame
durations
   The durations provided
   Type array
event_observed
   The event_observed variable provided
   Type array
timeline
   The time line to use for plotting and indexing
   Type array
tenry
   The entry array provided, or None
   Type array or None
event_table
   A summary of the life table
   Type DataFrame

conditional_time_to_event_
   Return a DataFrame, with index equal to survival_function_'s index, that estimates the median duration remaining until the death event, given survival up until time t. For example, if an individual exists until age 1, their expected life remaining given they lived to time 1 might be 9 years.
   Returns conditional_time_to_
   Return type DataFrame
cumulative_density_at_times(times, label=None)
   Return a Pandas series of the predicted cumulative density at specific times
   Parameters times (iterable or float)
   Returns
   Return type pd.Series
cumulative_hazard_at_times(times, label=None)
divide (other)
   Divide the {0} of two {1} objects.
   Parameters other (an {1} fitted instance.)

164 Chapter 1. Contents:
fit (durations, event_observed=None, timeline=None, entry=None, label='KM_estimate', left_censorship=False, alpha=None, ci_labels=None, weights=None)
Fit the model to a right-censored dataset

Parameters

- durations (an array, list, pd.DataFrame or pd.Series) – length n – duration subject was observed for
- event_observed (an array, list, pd.DataFrame, or pd.Series, optional) – True if the the death was observed, False if the event was lost (right-censored). Defaults all True if event_observed==None
- timeline (an array, list, pd.DataFrame, or pd.Series, optional) – return the best estimate at the values in timelines (positively increasing)
- entry (an array, list, pd.DataFrame, or pd.Series, optional) – relative time when a subject entered the study. This is useful for left-truncated (not left-censored) observations. If None, all members of the population entered study when they were “born”.
- label (string, optional) – a string to name the column of the estimate.
- alpha (float, optional) – the alpha value in the confidence intervals. Overrides the initializing alpha for this call to fit only.
- left_censorship (bool, optional (default=False)) – Deprecated, use fit_left_censoring
- ci_labels (tuple, optional) – add custom column names to the generated confidence intervals as a length-2 list: [lower-bound name], [upper-bound name]. Default: <label>_lower_<1-alpha/2>
- weights (an array, list, pd.DataFrame, or pd.Series, optional) – if providing a weighted dataset. For example, instead of providing every subject as a single element of durations and event_observed, one could weigh subject differently.

Returns self – self with new properties like survival_function_.plot(), median

Return type KaplanMeierFitter

fit_left_censoring (durations, event_observed=None, timeline=None, entry=None, label='KM_estimate', alpha=None, ci_labels=None, weights=None)
Fit the model to a left-censored dataset

Parameters

- durations (an array, list, pd.DataFrame or pd.Series) – length n – duration subject was observed for
- event_observed (an array, list, pd.DataFrame, or pd.Series, optional) – True if the the death was observed, False if the event was lost (right-censored). Defaults all True if event_observed==None
- timeline (an array, list, pd.DataFrame, or pd.Series, optional) – return the best estimate at the values in timelines (positively increasing)
- entry (an array, list, pd.DataFrame, or pd.Series, optional) – relative time when a subject entered the study. This is useful for left-truncated (not left-censored) observations. If None, all members of the population entered study when they were “born”.
- label (string, optional) – a string to name the column of the estimate.
- alpha (float, optional) – the alpha value in the confidence intervals. Overrides the initializing alpha for this call to fit only.
• **left_censorship** *(bool, optional (default=False)) – Deprecated, use fit_left_censoring*

• **ci_labels** *(tuple, optional) – add custom column names to the generated confidence intervals as a length-2 list: [<lower-bound name>, <upper-bound name>]. Default: <label>_lower_<1-alpha/2>*

• **weights** *(an array, list, pd.DataFrame, or pd.Series, optional) – if providing a weighted dataset. For example, instead of providing every subject as a single element of durations and event_observed, one could weigh subject differently.*

Returns **self** – self with new properties like survival_function_, plot(), median

Return type **KaplanMeierFitter**

**fit_right_censoring** *(args, **kwargs)*

**hazard_at_times** *(times, label=None)*

**plot** *(**kwargs)*

Plots a pretty figure of the model

Matplotlib plot arguments can be passed in inside the kwargs, plus

**Parameters**

• **show_censors** *(bool) – place markers at censorship events. Default: False*

• **censor_styles** *(dict) – If show_censors, this dictionary will be passed into the plot call.*

• **ci_alpha** *(float) – the transparency level of the confidence interval. Default: 0.3*

• **ci_force_lines** *(bool) – force the confidence intervals to be line plots (versus default shaded areas). Default: False*

• **ci_show** *(bool) – show confidence intervals. Default: True*

• **ci_legend** *(bool) – if ci_force_lines is True, this is a boolean flag to add the lines’ labels to the legend. Default: False*

• **at_risk_counts** *(bool) – show group sizes at time points. See function add_at_risk_counts for details. Default: False*

• **loc** *(slice) – specify a time-based subsection of the curves to plot, ex:*

```python
>>> model.plot(loc=slice(0.,10.))
```

will plot the time values between t=0. and t=10.

• **iloc** *(slice) – specify a location-based subsection of the curves to plot, ex:*

```python
>>> model.plot(iloc=slice(0,10))
```

will plot the first 10 time points.

• **invert_y_axis** *(bool) – boolean to invert the y-axis, useful to show cumulative graphs instead of survival graphs. (Deprecated, use plot_cumulative_density())

Returns a pyplot axis object

Return type **ax**

**plot_cumulative_density** *(**kwargs)*

Plots a pretty figure of {0}.{1}

Matplotlib plot arguments can be passed in inside the kwargs, plus
Parameters

- **show_censors** *(bool)* – place markers at censorship events. Default: False
- **censor_styles** *(bool)* – If show_censors, this dictionary will be passed into the plot call.
- **ci_alpha** *(bool)* – the transparency level of the confidence interval. Default: 0.3
- **ci_force_lines** *(bool)* – force the confidence intervals to be line plots (versus default shaded areas). Default: False
- **ci_show** *(bool)* – show confidence intervals. Default: True
- **ci_legend** *(bool)* – if ci_force_lines is True, this is a boolean flag to add the lines’ labels to the legend. Default: False
- **at_risk_counts** *(bool)* – show group sizes at time points. See function `add_at_risk_counts` for details. Default: False
- **loc** *(slice)* – specify a time-based subsection of the curves to plot, ex:

  ```
  >>> model.plot(loc=slice(0.,10.))
  ```

  will plot the time values between t=0. and t=10.
- **iloc** *(slice)* – specify a location-based subsection of the curves to plot, ex:

  ```
  >>> model.plot(iloc=slice(0,10))
  ```

  will plot the first 10 time points.
- **invert_y_axis** *(bool)* – boolean to invert the y-axis, useful to show cumulative graphs instead of survival graphs. (Deprecated, use `plot_cumulative_density`)

Returns

a pyplot axis object

Return type

ax

- `plot_cumulative_hazard(**kwargs)`
- `plot_hazard(**kwargs)`
- `plot_loglogs(*args, **kwargs)`
  Plot \( \log(S(t)) \) against \( \log(t) \)
- `plot_survival_function(**kwargs)`
  Alias of `plot`
- `predict(times, interpolate=False)`
  Predict the \( \{0\} \) at certain point in time. Uses a linear interpolation if points in time are not in the index.

Parameters

- **times** *(scalar, or array)* – a scalar or an array of times to predict the value of \( \{0\} \) at.
- **interpolate** *(boolean, optional (default=False))* – for methods that produce a stepwise solution (Kaplan-Meier, Nelson-Aalen, etc), turning this to True will use an linear interpolation method to provide a more “smooth” answer.

Returns predictions

Return type

a scalar if time is a scalar, a numpy array if time in an array.

- `subtract(other)`
  Subtract the \( \{0\} \) of two \( \{1\} \) objects.
Parameters other (an fitted instance.)

**survival_function_at_times** *(times, label=None)*

Return a Pandas series of the predicted survival value at specific times

Parameters **times** *(iterable or float)*

Returns

Return type pd.Series

**lifelines.fitters.log_logistic_fitter module**

```python
class lifelines.fitters.log_logistic_fitter.LogLogisticFitter(*args, **kwargs)
Bases: lifelines.fitters.KnownModelParametericUnivariateFitter
```

This class implements a Log-Logistic model for univariate data. The model has parameterized form:

\[
S(t) = \left(1 + \left(\frac{t}{\alpha}\right)^\beta\right)^{-1}, \alpha > 0, \beta > 0,
\]

and the hazard rate is:

\[
h(t) = \frac{\beta}{\alpha} \left(\frac{t}{\alpha}\right)^{\beta-1} \left(1 + \left(\frac{t}{\alpha}\right)^\beta\right)^{-1}
\]

and the cumulative hazard is:

\[
H(t) = \log \left(\left(\frac{t}{\alpha}\right)^\beta + 1\right)
\]

After calling the `.fit` method, you have access to properties like: `cumulative_hazard_`, `plot`, `survival_function_`, `alpha_` and `beta_`. A summary of the fit is available with the method `print_summary()`

Parameters **alpha** *(float, optional (default=0.05)) – the level in the confidence intervals.

**Examples**

```python
>>> from lifelines import LogLogisticFitter
>>> from lifelines.datasets import load_waltons
>>> waltons = load_waltons()
>>> llf = LogLogisticFitter()
>>> llf.fit(waltons['T'], waltons['E'])
>>> llf.plot()
>>> print(llf.alpha_)
```

**cumulative_hazard_**

The estimated cumulative hazard (with custom timeline if provided)

Type DataFrame

**confidence_interval_cumulative_hazard_**

The lower and upper confidence intervals for the cumulative hazard

Type DataFrame
hazard_
    The estimated hazard (with custom timeline if provided)
    Type DataFrame

certainty_interval_hazard_
    The lower and upper confidence intervals for the hazard
    Type DataFrame

survival_function_
    The estimated survival function (with custom timeline if provided)
    Type DataFrame

certainty_interval_survival_function_
    The lower and upper confidence intervals for the survival function
    Type DataFrame

cumulative_density_
    The estimated cumulative density function (with custom timeline if provided)
    Type DataFrame

certainty_interval_cumulative_density_
    The lower and upper confidence intervals for the cumulative density
    Type DataFrame

variance_matrix_
    The variance matrix of the coefficients
    Type numpy array

median_
    The median time to event
    Type float

alpha_
    The fitted parameter in the model
    Type float

beta_
    The fitted parameter in the model
    Type float

durations
    The durations provided
    Type array

event_observed
    The event_observed variable provided
    Type array

timeline
    The time line to use for plotting and indexing
    Type array

entry
    The entry array provided, or None

1.10. Reference library for lifelines
Type  array or None

**conditional_time_to_event**
Return a DataFrame, with index equal to \texttt{survival_function}'s index, that estimates the median duration remaining until the death event, given survival up until time t. For example, if an individual exists until age 1, their expected life remaining \textit{given they lived to time 1} might be 9 years.

**Returns**  \texttt{conditional_time_to_event}

**Return type**  DataFrame

**confidence_interval**
The confidence interval of the cumulative hazard. This is an alias for \texttt{confidence_interval_cumulative_hazard}.

**confidence_interval_cumulative_density**
The confidence interval of the survival function.

**confidence_interval_cumulative_hazard**
The confidence interval of the cumulative hazard. This is an alias for \texttt{confidence_interval}.

**confidence_interval_hazard**
The confidence interval of the hazard.

**confidence_interval_survival_function**
The confidence interval of the survival function.

**cumulative_density_at_times** \texttt{(times, label=None)}
Return a Pandas series of the predicted cumulative density function (1-survival function) at specific times.

**Parameters**
- \texttt{times (iterable or float)} – values to return the survival function at.
- \texttt{label (string, optional)} – Rename the series returned. Useful for plotting.

**Returns**

**Return type**  pd.Series

**cumulative_hazard_at_times** \texttt{(times, label=None)}
Return a Pandas series of the predicted cumulative hazard value at specific times.

**Parameters**
- \texttt{times (iterable or float)} – values to return the cumulative hazard at.
- \texttt{label (string, optional)} – Rename the series returned. Useful for plotting.

**Returns**

**Return type**  pd.Series

**divide** \texttt{(other)}
Divide the \{0\} of two \{1\} objects.

**Parameters**
- \texttt{other (an \{1\} fitted instance.)}

**fit** \texttt{(durations, event_observed=None, timeline=None, label=None, alpha=None, ci_labels=None, show_progress=False, entry=None, weights=None, left_censorship=False)}

**Parameters**
- \texttt{durations (an array, or pd.Series)} – length n, duration subject was observed for
- **event_observed** *(numpy array or pd.Series, optional)* – length n, True if the death was observed, False if the event was lost (right-censored). Defaults all True if event_observed==None

- **timeline** *(list, optional)* – return the estimate at the values in timeline (positively increasing)

- **label** *(string, optional)* – a string to name the column of the estimate.

- **alpha** *(float, optional)* – the alpha value in the confidence intervals. Overrides the initializing alpha for this call to fit only.

- **ci_labels** *(list, optional)* – add custom column names to the generated confidence intervals as a length-2 list: [lower-bound name>, <upper-bound name>]. Default: <label>_lower_<alpha>

- **show_progress** *(boolean, optional)* – since this is an iterative fitting algorithm, switching this to True will display some iteration details.

- **entry** *(an array, or pd.Series, of length n)* – relative time when a subject entered the study. This is useful for left-truncated (not left-censored) observations. If None, all members of the population entered study when they were “born”: time zero.

- **weights** *(an array, or pd.Series, of length n)* – integer weights per observation

Returns self with new properties like `cumulative_hazard_`, `survival_function_`

Return type self

**fit_interval_censoring** *(lower_bound, upper_bound, event_observed=None, timeline=None, label=None, alpha=None, ci_labels=None, show_progress=False, entry=None, weights=None)*

Fit the model to an interval censored dataset.

Parameters

- **lower_bound** *(an array, or pd.Series)* – length n, the start of the period the subject experienced the event in.

- **upper_bound** *(an array, or pd.Series)* – length n, the end of the period the subject experienced the event in. If the value is equal to the corresponding value in lower_bound, then the individual’s event was observed (not censored).

- **event_observed** *(numpy array or pd.Series, optional)* – length n, if left optional, infer from lower_bound and upper_cound (if lower_bound==upper_bound then event observed, if lower_bound < upper_bound, then event censored)

- **timeline** *(list, optional)* – return the estimate at the values in timeline (positively increasing)

- **label** *(string, optional)* – a string to name the column of the estimate.

- **alpha** *(float, optional)* – the alpha value in the confidence intervals. Overrides the initializing alpha for this call to fit only.

- **ci_labels** *(list, optional)* – add custom column names to the generated confidence intervals as a length-2 list: [lower-bound name>, <upper-bound name>]. Default: <label>_lower_<alpha>

- **show_progress** *(boolean, optional)* – since this is an iterative fitting algorithm, switching this to True will display some iteration details.
• **entry** *(an array, or pd.Series, of length n)* – relative time when a subject entered the study. This is useful for left-truncated (not left-censored) observations. If None, all members of the population entered study when they were “born”: time zero.

• **weights** *(an array, or pd.Series, of length n)* – integer weights per observation

**Returns** self with new properties like `cumulative_hazard_`, `survival_function_`

**Return type** `self`

`fit_left_censoring` *(durations, event_observed=None, timeline=None, label=None, alpha=None, ci_labels=None, show_progress=False, entry=None, weights=None)*

Fit the model to a left-censored dataset

**Parameters**

• **durations** *(an array, or pd.Series)* – length n, duration subject was observed for

• **event_observed** *(numpy array or pd.Series, optional)* – length n, True if the the death was observed, False if the event was lost (right-censored). Defaults all True if event_observed==None

• **timeline** *(list, optional)* – return the estimate at the values in timeline (positively increasing)

• **label** *(string, optional)* – a string to name the column of the estimate.

• **alpha** *(float, optional)* – the alpha value in the confidence intervals. Overrides the initializing alpha for this call to fit only.

• **ci_labels** *(list, optional)* – add custom column names to the generated confidence intervals as a length-2 list: [lower-bound name>, <upper-bound name>]. Default: <label>_lower_<alpha>

• **show_progress** *(boolean, optional)* – since this is an iterative fitting algorithm, switching this to True will display some iteration details.

• **entry** *(an array, or pd.Series, of length n)* – relative time when a subject entered the study. This is useful for left-truncated (not left-censored) observations. If None, all members of the population entered study when they were “born”: time zero.

• **weights** *(an array, or pd.Series, of length n)* – integer weights per observation

**Returns** self with new properties like `cumulative_hazard_`, `survival_function_`

**Return type** `self`

`fit_right_censoring` *(args, **kwargs)*

`hazard_at_times` *(times, label=None)*

Return a Pandas series of the predicted hazard at specific times.

**Parameters**

• **times** *(iterable or float)* – values to return the hazard at.

• **label** *(string, optional)* – Rename the series returned. Useful for plotting.

**Returns**

**Return type** `pd.Series`

`median_`

Return the unique time point, t, such that S(t) = 0.5. This is the “half-life” of the population, and a robust summary statistic for the population, if it exists.
plot (**kwargs)

Produce a pretty-plot of the estimate.

plot_cumulative_density (**kwargs)

plot_cumulative_hazard (**kwargs)

plot_hazard (**kwargs)

plot_survival_function (**kwargs)

predict (times, interpolate=False)

Predict the {0} at certain point in time. Uses a linear interpolation if points in time are not in the index.

Parameters

- times (scalar, or array) – a scalar or an array of times to predict the value of {0} at.
- interpolate (boolean, optional (default=False)) – for methods that produce a stepwise solution (Kaplan-Meier, Nelson-Aalen, etc), turning this to True will use an linear interpolation method to provide a more “smooth” answer.

Returns predictions

Return type a scalar if time is a scalar, a numpy array if time in an array.

print_summary (decimals=2, **kwargs)

Print summary statistics describing the fit, the coefficients, and the error bounds.

Parameters

- decimals (int, optional (default=2)) – specify the number of decimal places to show
- kwargs – print additional metadata in the output (useful to provide model names, dataset names, etc.) when comparing multiple outputs.

subtract (other)

Subtract the {0} of two {1} objects.

Parameters other (an {1} fitted instance.)

summary

Summary statistics describing the fit.

Returns df – Contains columns coef, exp(coef), se(coef), z, p, lower, upper

Return type pd.DataFrame

See also:

print_summary

survival_function_at_times (times, label=None)

Return a Pandas series of the predicted survival value at specific times.

Parameters

- times (iterable or float) – values to return the survival function at.
- label (string, optional) – Rename the series returned. Useful for plotting.

Returns

Return type pd.Series
This class implements an Log Normal model for univariate data. The model has parameterized form:

\[ S(t) = 1 - \Phi((\log(t) - \mu)/\sigma), \sigma > 0 \]

where \( \Phi \) is the CDF of a standard normal random variable. This implies the cumulative hazard rate is

\[ H(t) = -\log(1 - \Phi((\log(t) - \mu)/\sigma)) \]

After calling the `.fit` method, you have access to properties like: `survival_function_`, `mu_`, `sigma_`. A summary of the fit is available with the method `print_summary()`

**Parameters**

- `alpha` *(float, optional (default=0.05))* – the level in the confidence intervals.

**Attributes**

- `cumulative_hazard_` *(DataFrame)*
  - The estimated cumulative hazard (with custom timeline if provided)

- `confidence_interval_cumulative_hazard_` *(DataFrame)*
  - The lower and upper confidence intervals for the cumulative hazard

- `hazard_` *(DataFrame)*
  - The estimated hazard (with custom timeline if provided)

- `confidence_interval_hazard_` *(DataFrame)*
  - The lower and upper confidence intervals for the hazard

- `survival_function_` *(DataFrame)*
  - The estimated survival function (with custom timeline if provided)

- `confidence_interval_survival_function_` *(DataFrame)*
  - The lower and upper confidence intervals for the survival function

- `cumulative_density_` *(DataFrame)*
  - The estimated cumulative density function (with custom timeline if provided)

- `confidence_interval_cumulative_density_` *(DataFrame)*
  - The lower and upper confidence intervals for the cumulative density

- `variance_matrix_` *(numpy array)*
  - The variance matrix of the coefficients

- `median_` *(numpy array)*
  - The median time to event
Type float

mu_
The fitted parameter in the model
Type float

sigma_
The fitted parameter in the model
Type float
durations
The durations provided
Type array
event_observed
The event_observed variable provided
Type array	
timeline
The timeline to use for plotting and indexing
Type array
time
event
The entry array provided, or None
Type array or None
conditional_time_to_event_
Return a DataFrame, with index equal to survival_function_’s index, that estimates the median duration remaining until the death event, given survival up until time t. For example, if an individual exists until age 1, their expected life remaining given they lived to time 1 might be 9 years.

Returns conditional_time_to_
Return type DataFrame

certainty_interval_
The confidence interval of the cumulative hazard. This is an alias for confidence_interval_cumulative_hazard_.
certainty_interval_cumulative_density_
The confidence interval of the survival function.
certainty_interval_cumulative_hazard_
The confidence interval of the cumulative hazard. This is an alias for confidence_interval_.
certainty_interval_hazard_
The confidence interval of the hazard.
certainty_interval_survival_function_
The confidence interval of the survival function.
cumuatuve_density at times(times, label=None)
Return a Pandas series of the predicted cumulative density function (1-survival function) at specific times.

Parameters

- times (iterable or float) – values to return the survival function at.
- label (string, optional) – Rename the series returned. Useful for plotting.
cumulative_hazard_at_times(times, label=None)

Return a Pandas series of the predicted cumulative hazard value at specific times.

Parameters
- times (iterable or float) – values to return the cumulative hazard at.
- label (string, optional) – Rename the series returned. Useful for plotting.

Returns
Return type pd.Series
divide(other)
Divide the [0] of two [1] objects.

Parameters other (an [1] fitted instance.)

fit(durations, event_observed=None, timeline=None, label=None, alpha=None, ci_labels=None, show_progress=False, entry=None, weights=None, left_censorship=False)

Parameters
- durations (an array, or pd.Series) – length n, duration subject was observed for
- event_observed (numpy array or pd.Series, optional) – length n, True if the the death was observed, False if the event was lost (right-censored). Defaults all True if event_observed==None
- timeline (list, optional) – return the estimate at the values in timeline (positively increasing)
- label (string, optional) – a string to name the column of the estimate.
- alpha (float, optional) – the alpha value in the confidence intervals. Overrides the initializing alpha for this call to fit only.
- ci_labels (list, optional) – add custom column names to the generated confidence intervals as a length-2 list: [<lower-bound name>, <upper-bound name>]. Default: <label>_lower_<alpha>
- show_progress (boolean, optional) – since this is an iterative fitting algorithm, switching this to True will display some iteration details.
- entry (an array, or pd.Series, of length n) – relative time when a subject entered the study. This is useful for left-truncated (not left-censored) observations. If None, all members of the population entered study when they were “born”: time zero.
- weights (an array, or pd.Series, of length n) – integer weights per observation

Returns self with new properties like cumulative_hazard_, survival_function_

Return type self

fit_interval_censoring(lower_bound, upper_bound, event_observed=None, timeline=None, label=None, alpha=None, ci_labels=None, show_progress=False, entry=None, weights=None)

Fit the model to an interval censored dataset.

Parameters
- lower_bound (an array, or pd.Series) – length n, the start of the period the subject experienced the event in.
• **upper_bound** *(an array, or pd.Series)* – length n, the end of the period the subject experienced the event in. If the value is equal to the corresponding value in lower_bound, then the individual’s event was observed (not censored).

• **event_observed** *(numpy array or pd.Series, optional)* – length n, if left optional, infer from lower_bound and upper_cound (if lower_bound==upper_bound then event observed, if lower_bound < upper_bound, then event censored)

• **timeline** *(list, optional)* – return the estimate at the values in timeline (positively increasing)

• **label** *(string, optional)* – a string to name the column of the estimate.

• **alpha** *(float, optional)* – the alpha value in the confidence intervals. Overrides the initializing alpha for this call to fit only.

• **ci_labels** *(list, optional)* – add custom column names to the generated confidence intervals as a length-2 list: [lower-bound name, upper-bound name]. Default: <label>_lower_<alpha>

• **show_progress** *(boolean, optional)* – since this is an iterative fitting algorithm, switching this to True will display some iteration details.

• **entry** *(an array, or pd.Series, of length n)* – relative time when a subject entered the study. This is useful for left-truncated (not left-censored) observations. If None, all members of the population entered study when they were “born”: time zero.

• **weights** *(an array, or pd.Series, of length n)* – integer weights per observation

Returns self with new properties like cumulative_hazard_, survival_function_

Return type self

fit_left_censoring*(durations, event_observed=None, timeline=None, label=None, alpha=None, ci_labels=None, show_progress=False, entry=None, weights=None)*

Fit the model to a left-censored dataset

Parameters

• **durations** *(an array, or pd.Series)* – length n, duration subject was observed for

• **event_observed** *(numpy array or pd.Series, optional)* – length n, True if the the death was observed, False if the event was lost (right-censored). Defaults all True if event_observed==None

• **timeline** *(list, optional)* – return the estimate at the values in timeline (positively increasing)

• **label** *(string, optional)* – a string to name the column of the estimate.

• **alpha** *(float, optional)* – the alpha value in the confidence intervals. Overrides the initializing alpha for this call to fit only.

• **ci_labels** *(list, optional)* – add custom column names to the generated confidence intervals as a length-2 list: [lower-bound name, upper-bound name]. Default: <label>_lower_<alpha>

• **show_progress** *(boolean, optional)* – since this is an iterative fitting algorithm, switching this to True will display some iteration details.

• **entry** *(an array, or pd.Series, of length n)* – relative time when a subject entered the study. This is useful for left-truncated (not left-censored) observations. If None, all members of the population entered study when they were “born”: time zero.

• **weights** *(an array, or pd.Series, of length n)* – integer weights per observation
Returns self with new properties like cumulative_hazard_, survival_function_

Return type self

fit_right_censoring (*args, **kwargs)

hazard_at_times (times, label=None)
    Return a Pandas series of the predicted hazard at specific times.

Parameters
    • times (iterable or float) – values to return the hazard at.
    • label (string, optional) – Rename the series returned. Useful for plotting.

Returns
    Return type pd.Series

median_
    Return the unique time point, t, such that S(t) = 0.5. This is the “half-life” of the population, and a robust
    summary statistic for the population, if it exists.

plot (**kwargs)
    Produce a pretty-plot of the estimate.

plot_cumulative_density (**kwargs)

plot_cumulative_hazard (**kwargs)

plot_hazard (**kwargs)

plot_survival_function (**kwargs)

predict (times, interpolate=False)
    Predict the {0} at certain point in time. Uses a linear interpolation if points in time are not in the index.

Parameters
    • times (scalar, or array) – a scalar or an array of times to predict the value of {0} at.
    • interpolate (boolean, optional (default=False) – for methods that produce a stepwise
        solution (Kaplan-Meier, Nelson-Aalen, etc), turning this to True will use an linear inter-
        polation method to provide a more “smooth” answer.

Returns predictions
    Return type a scalar if time is a scalar, a numpy array if time in an array.

print_summary (decimals=2, **kwargs)
    Print summary statistics describing the fit, the coefficients, and the error bounds.

Parameters
    • decimals (int, optional (default=2)) – specify the number of decimal places to show
    • kwargs – print additional metadata in the output (useful to provide model names, dataset
        names, etc.) when comparing multiple outputs.

subtract (other)
    Subtract the {0} of two {1} objects.

Parameters other (an {1} fitted instance.)

summary
    Summary statistics describing the fit.

Returns df – Contains columns coef, exp(coef), se(coef), z, p, lower, upper
Return type pd.DataFrame

See also:

print_summary

**survival_function_at_times**(times, label=None)

Return a Pandas series of the predicted survival value at specific times.

**Parameters**

- **times** (iterable or float) – values to return the survival function at.
- **label** (string, optional) – Rename the series returned. Useful for plotting.

**Returns**

Return type pd.Series

**lifelines.fitters.nelson_aalen_fitter module**

**class** lifelines.fitters.nelson_aalen_fitter.NelsonAalenFitter(alpha=0.05, nelson_aalen_smoothing=True)

**Bases:** lifelines.fitters.UnivariateFitter

Class for fitting the Nelson-Aalen estimate for the cumulative hazard.

NelsonAalenFitter(alpha=0.05, nelson_aalen_smoothing=True)

**Parameters**

- **alpha** (float, optional (default=0.05)) – The alpha value associated with the confidence intervals.
- **nelson_aalen_smoothing** (bool, optional) – If the event times are naturally discrete (like discrete years, minutes, etc.) then it is advisable to turn this parameter to False. See [1], pg.84.

**Notes**


**cumulative_hazard**

The estimated cumulative hazard (with custom timeline if provided)

Type DataFrame

**confidence_interval**

The lower and upper confidence intervals for the cumulative hazard

Type DataFrame

**durations**

The durations provided

Type array

**event_observed**

The event_observed variable provided

Type array

**timeline**

The time line to use for plotting and indexing
Type array

**entry**
The entry array provided, or None

Type array or None

**event_table**
A summary of the life table

Type DataFrame

**conditional_time_to_event**
Return a DataFrame, with index equal to survival_function_'s index, that estimates the median duration remaining until the death event, given survival up until time t. For example, if an individual exists until age 1, their expected life remaining given they lived to time 1 might be 9 years.

Returns conditional_time_to_event

Return type DataFrame

**cumulative_density_at_times**(times, label=None)

**cumulative_hazard_at_times**(times, label=None)

**divide**(other)
Divide the [0] of two [1] objects.

Parameters other *(an [1] fitted instance.)*

**fit**(durations, event_observed=None, timeline=None, entry=None, label='NA_estimate', alpha=None, ci_labels=None, weights=None)

Parameters

• **durations** *(an array, or pd.Series, of length n)* – duration subject was observed for

• **timeline** *(iterable)* – return the best estimate at the values in timelines (positively increasing)

• **event_observed** *(an array, or pd.Series, of length n)* – True if the the death was observed, False if the event was lost (right-censored). Defaults all True if event_observed==None

• **entry** *(an array, or pd.Series, of length n)* – relative time when a subject entered the study. This is useful for left-truncated observations, i.e the birth event was not observed. If None, defaults to all 0 (all birth events observed.)

• **label** *(string)* – a string to name the column of the estimate.

• **alpha** *(float)* – the alpha value in the confidence intervals. Overrides the initializing alpha for this call to fit only.

• **ci_labels** *(iterable)* – add custom column names to the generated confidence intervals as a length-2 list: [<lower-bound name>, <upper-bound name>]. Default: <label>_lower_<1-alpha/2>

• **weights** *(n array, or pd.Series, of length n)* – if providing a weighted dataset. For example, instead of providing every subject as a single element of durations and event_observed, one could weigh subject differently.

Returns

Return type self, with new properties like cumulative_hazard_.

**fit_right_censoring**(args, **kwargs)

**hazard_at_times**(times, label=None)
**plot(**kwargs**)

Plots a pretty figure of the model

Matplotlib plot arguments can be passed in inside the kwargs, plus

**Parameters**

- **show_censors** (bool) – place markers at censorship events. Default: False
- **censor_styles** (dict) – If show_censors, this dictionary will be passed into the plot call.
- **ci_alpha** (float) – the transparency level of the confidence interval. Default: 0.3
- **ci_force_lines** (bool) – force the confidence intervals to be line plots (versus default shaded areas). Default: False
- **ci_show** (bool) – show confidence intervals. Default: True
- **ci_legend** (bool) – if ci_force_lines is True, this is a boolean flag to add the lines’ labels to the legend. Default: False
- **at_risk_counts** (bool) – show group sizes at time points. See function `add_at_risk_counts` for details. Default: False
- **loc** (slice) – specify a time-based subsection of the curves to plot, ex:

  ```python
  >>> model.plot(loc=slice(0.,10.))
  ```

  will plot the time values between t=0. and t=10.

- **iloc** (slice) – specify a location-based subsection of the curves to plot, ex:

  ```python
  >>> model.plot(iloc=slice(0,10))
  ```

  will plot the first 10 time points.

- **invert_y_axis** (bool) – boolean to invert the y-axis, useful to show cumulative graphs instead of survival graphs. (Deprecated, use `plot_cumulative_density()`)

**Returns** a pyplot axis object

**Return type** ax

**plot_cumulative_density(**kwargs**)

**plot_cumulative_hazard(**kwargs**)

**plot_hazard(**kwargs**)

**plot_survival_function(**kwargs**)

**predict**(times, interpolate=False)

Predict the \{0\} at certain point in time. Uses a linear interpolation if points in time are not in the index.

**Parameters**

- **times** (scalar, or array) – a scalar or an array of times to predict the value of \{0\} at.
- **interpolate** (boolean, optional (default=False)) – for methods that produce a stepwise solution (Kaplan-Meier, Nelson-Aalen, etc), turning this to True will use an linear interpolation method to provide a more “smooth” answer.

**Returns** predictions

**Return type** a scalar if time is a scalar, a numpy array if time in an array.
Parameters `bandwidth` *(float)* – the bandwidth used in the Epanechnikov kernel.

Returns a DataFrame of the smoothed hazard

Return type DataFrame

`smoothed_hazard_confidence_intervals_` *(bandwidth, hazard=None)*

Parameters

- `bandwidth` *(float)* – the bandwidth to use in the Epanechnikov kernel. > 0
- `hazard_` *(numpy array)* – a computed (n,) numpy array of estimated hazard rates. If none, uses `smoothed_hazard_`

`subtract` *(other)*

Subtract the {0} of two {1} objects.

Parameters `other` *(an {1} fitted instance.)*

`survival_function_at_times` *(times, label=None)*

lifelines.fitters.piecewise_exponential_fitter module

**class** lifelines.fitters.piecewise_exponential_fitter.PiecewiseExponentialFitter *(breakpoints, *args, **kwargs)*

Bases: lifelines.fitters.KnownModelParametericUnivariateFitter

This class implements an Piecewise Exponential model for univariate data. The model has parameterized hazard rate:

\[
   h(t) = \begin{cases} 
   1/\lambda_0, & \text{if } t \leq \tau_0 \\
   1/\lambda_1, & \text{if } \tau_0 < t \leq \tau_1 \\
   1/\lambda_2, & \text{if } \tau_1 < t \leq \tau_2 \\
   \vdots
   \end{cases}
\]

You specify the breakpoints, \(\tau_i\), and lifelines will find the optional values for the parameters.

After calling the `.fit` method, you have access to properties like: `survival_function_`, plot, `cumulative_hazard_`. A summary of the fit is available with the method `print_summary()`

Parameters

- `breakpoints` *(list)* – a list of times when a new exponential model is constructed.
- `alpha` *(float, optional (default=0.05))* – the level in the confidence intervals.

**Important:** The parameterization of this model changed in lifelines 0.19.1. Previously, the cumulative hazard looked like \(\lambda_i t\). The parameterization is now the reciprocal of \(\lambda_i\).

`cumulative_hazard_`

The estimated cumulative hazard (with custom timeline if provided)

Type DataFrame

`confidence_interval_cumulative_hazard_`

The lower and upper confidence intervals for the cumulative hazard

Type DataFrame
hazard
   The estimated hazard (with custom timeline if provided)
   Type DataFrame

confidence_interval_hazard
   The lower and upper confidence intervals for the hazard
   Type DataFrame

survival_function
   The estimated survival function (with custom timeline if provided)
   Type DataFrame

confidence_interval_survival_function
   The lower and upper confidence intervals for the survival function
   Type DataFrame

cumulative_density
   The estimated cumulative density function (with custom timeline if provided)
   Type DataFrame

confidence_interval_cumulative_density
   The lower and upper confidence intervals for the cumulative density
   Type DataFrame

variance_matrix
   The variance matrix of the coefficients
   Type numpy array

median
   The median time to event
   Type float

lambda_i
   The fitted parameter in the model, for i = 0, 1 ... n-1 breakpoints
   Type float

durations
   The durations provided
   Type array

event_observed
   The event_observed variable provided
   Type array

timeline
   The time line to use for plotting and indexing
   Type array

entry
   The entry array provided, or None
   Type array or None

breakpoints
   The provided breakpoints
conditional_time_to_event_
Return a DataFrame, with index equal to survival_function_’s index, that estimates the median duration remaining until the death event, given survival up until time t. For example, if an individual exists until age 1, their expected life remaining given they lived to time 1 might be 9 years.

Returns conditional_time_to_

Return type DataFrame
classification_interval_
The confidence interval of the cumulative hazard. This is an alias for confidence_interval_cumulative_hazard_.
classification_interval_cumulative_density_
The confidence interval of the survival function.
classification_interval_cumulative_hazard_
The confidence interval of the cumulative hazard. This is an alias for confidence_interval_.
classification_interval_hazard_
The confidence interval of the hazard.
classification_interval_survival_function_
The confidence interval of the survival function.
cumulative_density_at_times (times, label=None)
Return a Pandas series of the predicted cumulative density function (1-survival function) at specific times.

Parameters

- times (iterable or float) – values to return the survival function at.
- label (string, optional) – Rename the series returned. Useful for plotting.

Returns
classification type pd.Series
cumulative_hazard_at_times (times, label=None)
Return a Pandas series of the predicted cumulative hazard value at specific times.

Parameters

- times (iterable or float) – values to return the cumulative hazard at.
- label (string, optional) – Rename the series returned. Useful for plotting.

Returns
classification type pd.Series
divide (other)
Divide the [0] of two [1] objects.

Parameters other (an [1] fitted instance.)

fit (durations, event_observed=None, timeline=None, label=None, alpha=None, ci_labels=None, show_progress=False, entry=None, weights=None, left_censorship=False)

Parameters

- durations (an array, or pd.Series) – length n, duration subject was observed for
- **event_observed** *(numpy array or pd.Series, optional)* – length n, True if the death was observed, False if the event was lost (right-censored). Defaults all True if event_observed==None

- **timeline** *(list, optional)* – return the estimate at the values in timeline (positively increasing)

- **label** *(string, optional)* – a string to name the column of the estimate.

- **alpha** *(float, optional)* – the alpha value in the confidence intervals. Overrides the initializing alpha for this call to fit only.

- **ci_labels** *(list, optional)* – add custom column names to the generated confidence intervals as a length-2 list: [lower-bound name, upper-bound name]. Default: \(<\text{label}\textunderscore lower\textunderscore <alpha>\)

- **show_progress** *(boolean, optional)* – since this is an iterative fitting algorithm, switching this to True will display some iteration details.

- **entry** *(an array, or pd.Series, of length n)* – relative time when a subject entered the study. This is useful for left-truncated (not left-censored) observations. If None, all members of the population entered study when they were “born”: time zero.

- **weights** *(an array, or pd.Series, of length n)* – integer weights per observation

**Returns** self with new properties like `cumulative_hazard_`, `survival_function_`

**Return type** self

**fit_interval_censoring** *(lower_bound, upper_bound, event_observed=None, timeline=None, label=None, alpha=None, ci_labels=None, show_progress=False, entry=None, weights=None)*

Fit the model to an interval censored dataset.

**Parameters**

- **lower_bound** *(an array, or pd.Series)* – length n, the start of the period the subject experienced the event in.

- **upper_bound** *(an array, or pd.Series)* – length n, the end of the period the subject experienced the event in. If the value is equal to the corresponding value in lower_bound, then the individual’s event was observed (not censored).

- **event_observed** *(numpy array or pd.Series, optional)* – length n, if left optional, infer from lower_bound and upper_bound (if lower_bound==upper_bound then event observed, if lower_bound < upper_bound, then event censored)

- **timeline** *(list, optional)* – return the estimate at the values in timeline (positively increasing)

- **label** *(string, optional)* – a string to name the column of the estimate.

- **alpha** *(float, optional)* – the alpha value in the confidence intervals. Overrides the initializing alpha for this call to fit only.

- **ci_labels** *(list, optional)* – add custom column names to the generated confidence intervals as a length-2 list: [lower-bound name, upper-bound name]. Default: \(<\text{label}\textunderscore lower\textunderscore <alpha>\)

- **show_progress** *(boolean, optional)* – since this is an iterative fitting algorithm, switching this to True will display some iteration details.
• **entry** *(an array, or pd.Series, of length n)* – relative time when a subject entered the study. This is useful for left-truncated (not left-censored) observations. If None, all members of the population entered study when they were “born”: time zero.

• **weights** *(an array, or pd.Series, of length n)* – integer weights per observation

**Returns**  self with new properties like `cumulative_hazard_`, `survival_function_`

**Return type**  self

`fit_left_censoring` *(durations, event_observed=None, timeline=None, label=None, alpha=None, ci_labels=None, show_progress=False, entry=None, weights=None)*

Fit the model to a left-censored dataset

**Parameters**

• **durations** *(an array, or pd.Series)* – length n, duration subject was observed for

• **event_observed** *(numpy array or pd.Series, optional)* – length n, True if the death was observed, False if the event was lost (right-censored). Defaults all True if `event_observed==None`

• **timeline** *(list, optional)* – return the estimate at the values in timeline (positively increasing)

• **label** *(string, optional)* – a string to name the column of the estimate.

• **alpha** *(float, optional)* – the alpha value in the confidence intervals. Overrides the initializing alpha for this call to fit only.

• **ci_labels** *(list, optional)* – add custom column names to the generated confidence intervals as a length-2 list: `[<lower-bound name>, <upper-bound name>]`. Default: `<label>_lower_<alpha>`

• **show_progress** *(boolean, optional)* – since this is an iterative fitting algorithm, switching this to True will display some iteration details.

• **entry** *(an array, or pd.Series, of length n)* – relative time when a subject entered the study. This is useful for left-truncated (not left-censored) observations. If None, all members of the population entered study when they were “born”: time zero.

• **weights** *(an array, or pd.Series, of length n)* – integer weights per observation

**Returns**  self with new properties like `cumulative_hazard_`, `survival_function_`

**Return type**  self

`fit_right_censoring` *(args, **kwargs)*

`hazard_at_times` *(times, label=None)*

Return a Pandas series of the predicted hazard at specific times.

**Parameters**

• **times** *(iterable or float)* – values to return the hazard at.

• **label** *(string, optional)* – Rename the series returned. Useful for plotting.

**Returns**

**Return type**  pd.Series

`median_`

Return the unique time point, t, such that \( S(t) = 0.5 \). This is the “half-life” of the population, and a robust summary statistic for the population, if it exists.
plot(**kwargs)
    Produce a pretty-plot of the estimate.

plot_cumulative_density(**kwargs)

plot_cumulative_hazard(**kwargs)

plot_hazard(**kwargs)

plot_survival_function(**kwargs)

predict(times, interpolate=False)
    Predict the {0} at certain point in time. Uses a linear interpolation if points in time are not in the index.

Parameters
    • times (scalar, or array) – a scalar or an array of times to predict the value of {0} at.
    • interpolate (boolean, optional (default=False)) – for methods that produce a stepwise solution (Kaplan-Meier, Nelson-Aalen, etc), turning this to True will use an linear interpolation method to provide a more “smooth” answer.

Returns predictions

Return type a scalar if time is a scalar, a numpy array if time in an array.

print_summary(decimals=2, **kwargs)
    Print summary statistics describing the fit, the coefficients, and the error bounds.

Parameters
    • decimals (int, optional (default=2)) – specify the number of decimal places to show
    • kwargs – print additional metadata in the output (useful to provide model names, dataset names, etc.) when comparing multiple outputs.

subtract(other)
    Subtract the {0} of two {1} objects.

Parameters other (an {1} fitted instance.)

summary
    Summary statistics describing the fit.

Returns df – Contains columns coef, exp(coef), se(coef), z, p, lower, upper

Return type pd.DataFrame

See also:

print_summary

survival_function_at_times(times, label=None)
    Return a Pandas series of the predicted survival value at specific times.

Parameters
    • times (iterable or float) – values to return the survival function at.
    • label (string, optional) – Rename the series returned. Useful for plotting.

Returns

Return type pd.Series
The `WeibullFitter` class implements a Weibull model for univariate data. The model has parameterized form:

\[ S(t) = \exp\left(-\left(\frac{t}{\lambda}\right)^{\rho}\right), \lambda > 0, \rho > 0, \]

which implies the cumulative hazard rate is

\[ H(t) = \left(\frac{t}{\lambda}\right)^{\rho}, \]

and the hazard rate is:

\[ h(t) = \frac{\rho}{\lambda} \left(\frac{t}{\lambda}\right)^{\rho - 1} \]

After calling the `.fit` method, you have access to properties like: `cumulative_hazard_`, `survival_function_`, `lambda_` and `rho_`. A summary of the fit is available with the method `print_summary()`.

**Parameters**

- `alpha` *(float, optional (default=0.05)) – the level in the confidence intervals.*

**Important:** The parameterization of this model changed in lifelines 0.19.0. Previously, the cumulative hazard looked like \((\lambda t)^{\rho}\). The parameterization is now the reciprocal of \(\lambda\).

**Examples**

```python
>>> from lifelines import WeibullFitter
>>> from lifelines.datasets import load_waltons
>>> waltons = load_waltons()
>>> wbf = WeibullFitter()
>>> wbf.fit(waltons['T'], waltons['E'])
>>> wbf.plot()
>>> print(wbf.lambda_)
```

- `cumulative_hazard_` *(DataFrame)*
  - The estimated cumulative hazard (with custom timeline if provided)

- `confidence_interval_cumulative_hazard_` *(DataFrame)*
  - The lower and upper confidence intervals for the cumulative hazard

- `hazard_` *(DataFrame)*
  - The estimated hazard (with custom timeline if provided)

- `confidence_interval_hazard_` *(DataFrame)*
  - The lower and upper confidence intervals for the hazard
**survival_function**
  The estimated survival function (with custom timeline if provided)
  
  **Type** DataFrame

**confidence_interval_survival_function**
  The lower and upper confidence intervals for the survival function
  
  **Type** DataFrame

**cumulative_density**
  The estimated cumulative density function (with custom timeline if provided)
  
  **Type** DataFrame

**confidence_interval_cumulative_density**
  The lower and upper confidence intervals for the cumulative density
  
  **Type** DataFrame

**variance_matrix**
  The variance matrix of the coefficients
  
  **Type** numpy array

**median**
  The median time to event
  
  **Type** float

**lambda_**
  The fitted parameter in the model
  
  **Type** float

**rho_**
  The fitted parameter in the model
  
  **Type** float

**durations**
  The durations provided
  
  **Type** array

**event_observed**
  The event_observed variable provided
  
  **Type** array

**timeline**
  The time line to use for plotting and indexing
  
  **Type** array

**entry**
  The entry array provided, or None
  
  **Type** array or None

**conditional_time_to_event**
  Return a DataFrame, with index equal to `survival_function`'s index, that estimates the median duration remaining until the death event, given survival up until time t. For example, if an individual exists until age 1, their expected life remaining *given they lived to time 1* might be 9 years.
  
  **Returns** conditional_time_to_
Return type DataFrame

certainty_interval_
The confidence interval of the cumulative hazard. This is an alias for certainty_interval_cumulative_hazard_.

certainty_interval_cumulative_density_
The confidence interval of the survival function.

certainty_interval_cumulative_hazard_
The confidence interval of the cumulative hazard. This is an alias for certainty_interval_.

certainty_interval_hazard_
The confidence interval of the hazard.

certainty_interval_survival_function_
The confidence interval of the survival function.

cumulative_density_at_times(times, label=None)
Return a Pandas series of the predicted cumulative density function (1-survival function) at specific times.

Parameters
- times (iterable or float) – values to return the survival function at.
- label (string, optional) – Rename the series returned. Useful for plotting.

Returns
Return type pd.Series

cumulative_hazard_at_times(times, label=None)
Return a Pandas series of the predicted cumulative hazard value at specific times.

Parameters
- times (iterable or float) – values to return the cumulative hazard at.
- label (string, optional) – Rename the series returned. Useful for plotting.

Returns
Return type pd.Series

divide(other)
Divide the {0} of two {1} objects.

Parameters other (an {1} fitted instance.)

fit(durations, event_observed=None, timeline=None, label=None, alpha=None, ci_labels=None, show_progress=False, entry=None, weights=None, left_censorship=False)

Parameters
- durations (an array, or pd.Series) – length n, duration subject was observed for
- event_observed (numpy array or pd.Series, optional) – length n, True if the death was observed, False if the event was lost (right-censored). Defaults all True if event_observed==None
- timeline (list, optional) – return the estimate at the values in timeline (positively increasing)
- label (string, optional) – a string to name the column of the estimate.
- alpha (float, optional) – the alpha value in the confidence intervals. Overrides the initializing alpha for this call to fit only.
• **ci_labels** *(list, optional)* – add custom column names to the generated confidence intervals as a length-2 list: [lower-bound name, upper-bound name]. Default: \_<label>_\_<alpha>

• **show_progress** *(boolean, optional)* – since this is an iterative fitting algorithm, switching this to True will display some iteration details.

• **entry** *(an array, or pd.Series, of length n)* – relative time when a subject entered the study. This is useful for left-truncated (not left-censored) observations. If None, all members of the population entered study when they were “born”: time zero.

• **weights** *(an array, or pd.Series, of length n)* – integer weights per observation

Returns self with new properties like \_cumulative_hazard_\_\_, \_survival_function_\_

Return type self

---

### fit_interval_censoring

\[\text{lower_bound, upper_bound, event_observed=\_None, timeline=\_None, label=\_None, alpha=\_None, ci_labels=\_None, show_progress=\_False, entry=\_None, weights=\_None}\]

Fit the model to an interval censored dataset.

**Parameters**

• **lower_bound** *(an array, or pd.Series)* – length n, the start of the period the subject experienced the event in.

• **upper_bound** *(an array, or pd.Series)* – length n, the end of the period the subject experienced the event in. If the value is equal to the corresponding value in lower_bound, then the individual’s event was observed (not censored).

• **event_observed** *(numpy array or pd.Series, optional)* – length n, if left optional, infer from lower_bound and upper_cound (if lower_bound==upper_bound then event observed, if lower_bound < upper_bound, then event censored)

• **timeline** *(list, optional)* – return the estimate at the values in timeline (positively increasing)

• **label** *(string, optional)* – a string to name the column of the estimate.

• **alpha** *(float, optional)* – the alpha value in the confidence intervals. Overrides the initializing alpha for this call to fit only.

• **ci_labels** *(list, optional)* – add custom column names to the generated confidence intervals as a length-2 list: [lower-bound name, upper-bound name]. Default: \_<label>_\_<alpha>

• **show_progress** *(boolean, optional)* – since this is an iterative fitting algorithm, switching this to True will display some iteration details.

• **entry** *(an array, or pd.Series, of length n)* – relative time when a subject entered the study. This is useful for left-truncated (not left-censored) observations. If None, all members of the population entered study when they were “born”: time zero.

• **weights** *(an array, or pd.Series, of length n)* – integer weights per observation

Returns self with new properties like \_cumulative_hazard_\_\_, \_survival_function_\_

Return type self

---

### fit_left_censoring

\[\text{durations, event_observed=\_None, timeline=\_None, label=\_None, alpha=\_None, ci_labels=\_None, show_progress=\_False, entry=\_None, weights=\_None}\]

Fit the model to a left-censored dataset.

**Parameters**

---

1.10. Reference library for *lifelines*
• **durations** *(an array, or pd.Series)* — length n, duration subject was observed for

• **event_observed** *(numpy array or pd.Series, optional)* — length n, True if the the death was observed, False if the event was lost (right-censored). Defaults all True if `event_observed==None`

• **timeline** *(list, optional)* — return the estimate at the values in timeline (positively increasing)

• **label** *(string, optional)* — a string to name the column of the estimate.

• **alpha** *(float, optional)* — the alpha value in the confidence intervals. Overrides the initial-

• **ci_labels** *(list, optional)* — add custom column names to the generated confidence in-

• **show_progress** *(boolean, optional)* — since this is an iterative fitting algorithm, switching this to True will display some iteration details.

• **entry** *(an array, or pd.Series, of length n)* — relative time when a subject entered the study. This is useful for left-truncated (not left-censored) observations. If None, all members of the population entered study when they were “born”: time zero.

• **weights** *(an array, or pd.Series, of length n)* — integer weights per observation

**Returns**  
self with new properties like `cumulative_hazard_`, `survival_function_`

**Return type**  
self

**fit_right_censoring** *(args, **kwargs)*

**hazard_at_times** *(times, label=None)*

Return a Pandas series of the predicted hazard at specific times.

**Parameters**

• **times** *(iterable or float)* — values to return the hazard at.

• **label** *(string, optional)* — Rename the series returned. Useful for plotting.

**Returns**

**Return type**  
pd.Series

**median**

Return the unique time point, t, such that S(t) = 0.5. This is the “half-life” of the population, and a robust summary statistic for the population, if it exists.

**plot** *(**kwargs)*

Produce a pretty-plot of the estimate.

**plot_cumulative_density** *(**kwargs)*

**plot_cumulative_hazard** *(**kwargs)*

**plot_hazard** *(**kwargs)*

**plot_survival_function** *(**kwargs)*

**predict** *(times, interpolate=False)*

Predict the {0} at certain point in time. Uses a linear interpolation if points in time are not in the index.

**Parameters**

• **times** *(scalar, or array)* — a scalar or an array of times to predict the value of {0} at.
• **interpolate** *(boolean, optional (default=False))* – for methods that produce a stepwise solution (Kaplan-Meier, Nelson-Aalen, etc), turning this to True will use an linear interpolation method to provide a more “smooth” answer.

**Returns predictions**

**Return type** a scalar if time is a scalar, a numpy array if time in an array.

**print_summary**(decimals=2, **kwargs)

Print summary statistics describing the fit, the coefficients, and the error bounds.

**Parameters**

- **decimals** *(int, optional (default=2))* – specify the number of decimal places to show
- **kwargs** – print additional metadata in the output (useful to provide model names, dataset names, etc.) when comparing multiple outputs.

**subtract**(other)

Subtract the 0 of two 1 objects.

**Parameters** other *(an 1 fitted instance.)*

**summary**

Summary statistics describing the fit.

**Returns** df – Contains columns coef, exp(coef), se(coef), z, p, lower, upper

**Return type** pd.DataFrame

See also:

print_summary

**survival_function_at_times**(times, label=None)

Return a Pandas series of the predicted survival value at specific times.

**Parameters**

- **times** *(iterable or float)* – values to return the survival function at.
- **label** *(string, optional)* – Rename the series returned. Useful for plotting.

**Returns**

**Return type** pd.Series

**lifelines.fitters.weibull_aft_fitter module**

**class lifelines.fitters.weibull_aft_fitter.WeibullAFTFitter**(alpha=0.05, penalizer=0.0, l1_ratio=0.0, fit_intercept=True)

**Bases:** lifelines.fitters.ParametericAFTRegressionFitter

This class implements a Weibull AFT model. The model has parameterized form, with \( \lambda(x) = \exp(\beta_0 + \beta_1 x_1 + ... + \beta_n x_n) \), and optionally, \( \rho(y) = \exp(\alpha_0 + \alpha_1 y_1 + ... + \alpha_m y_m) \),

\[
S(t; x, y) = \exp \left( - \left( \frac{t}{\lambda(x)} \right)^{\rho(y)} \right),
\]
which implies the cumulative hazard rate is

\[ H(t; x, y) = \left( \frac{t}{\lambda(x)} \right)^\rho(y) \],

After calling the `.fit` method, you have access to properties like: `params_`, `print_summary()`. A summary of the fit is available with the method `print_summary()`.

**Parameters**

- **alpha** (`float`, optional (default=0.05)) – the level in the confidence intervals.
- **fit_intercept** (`boolean`, optional (default=True)) – Allow lifelines to add an intercept column of 1s to `df`, and `ancillary_df` if applicable.
- **penalizer** (`float`, optional (default=0.0)) – the penalizer coefficient to the size of the coefficients. See `l1_ratio`. Must be equal to or greater than 0.
- **l1_ratio** (`float`, optional (default=0.0)) – how much of the penalizer should be attributed to an L1 penalty (otherwise an L2 penalty). The penalty function looks like `penalizer * l1_ratio * ||w||_1 + 0.5 * penalizer * (1 - l1_ratio) * ||w||^2_2`

**params_**

The estimated coefficients

**Type** `DataFrame`

**confidence_intervals_**

The lower and upper confidence intervals for the coefficients

**Type** `DataFrame`

**durations**

The event_observed variable provided

**Type** `Series`

**event_observed**

The event_observed variable provided

**Type** `Series`

**weights**

The event_observed variable provided

**Type** `Series`

**variance_matrix_**

The variance matrix of the coefficients

**Type** `numpy array`

**standard_errors_**

the standard errors of the estimates

**Type** `Series`

**score_**

the concordance index of the model.

**Type** `float`

**fit**(df, duration_col=None, event_col=None, ancillary_df=None, show_progress=False, timeline=None, weights_col=None, robust=False, initial_point=None, entry_col=None)

Fit the accelerated failure time model to a right-censored dataset.
Parameters

- **df (DataFrame)** – a Pandas DataFrame with necessary columns `duration_col` and `event_col` (see below), covariates columns, and special columns (weights). `duration_col` refers to the lifetimes of the subjects. `event_col` refers to whether the ‘death’ events was observed: 1 if observed, 0 else (censored).

- **duration_col (string)** – the name of the column in DataFrame that contains the subjects’ lifetimes.

- **event_col (string, optional)** – the name of the column in DataFrame that contains the subjects’ death observation. If left as None, assume all individuals are uncensored.

- **show_progress (boolean, optional (default=False))** – since the fitter is iterative, show convergence diagnostics. Useful if convergence is failing.

- **ancillary_df (None, boolean, or DataFrame, optional (default=None))** – Choose to model the ancillary parameters. If None or False, explicitly do not fit the ancillary parameters using any covariates. If True, model the ancillary parameters with the same covariates as df. If DataFrame, provide covariates to model the ancillary parameters. Must be the same row count as df.

- **timeline (array, optional)** – Specify a timeline that will be used for plotting and prediction

- **weights_col (string)** – the column in DataFrame that specifies weights per observation.

- **robust (boolean, optional (default=False))** – Compute the robust errors using the Huber sandwich estimator.

- **initial_point ((d,) numpy array, optional)** – initialize the starting point of the iterative algorithm. Default is the zero vector.

- **entry_col (specify a column in the DataFrame that denotes any late-entries (left truncation) that occurred. See) – the docs on left truncation

Returns self with additional new properties: `print_summary`, `params_`, `confidence_intervals` and more

Return type self

Examples

```python
>>> from lifelines import WeibullAFTFitter, LogNormalAFTFitter,
   LogLogisticAFTFitter
>>> df = pd.DataFrame(
   {'T': [5, 3, 9, 8, 7, 4, 4, 3, 2, 5, 6, 7],
   'E': [1, 1, 1, 1, 1, 0, 1, 1, 1, 2, 2, 2],
   'var': [0, 0, 0, 1, 1, 1, 1, 1, 2, 2, 2, 2],
   'age': [4, 3, 9, 8, 7, 4, 4, 3, 2, 5, 6, 7],
})
>>> aft = WeibullAFTFitter()
>>> aft.fit(df, 'T', 'E')
>>> aft.print_summary()
>>> aft.predict_median(df)
>>> aft = WeibullAFTFitter()
>>> aft.fit(df, 'T', 'E', ancillary_df=df)
```
```python
>>> aft.print_summary()
>>> aft.predict_median(df)
```

`fit_interval_censoring`(df, lower_bound_col, upper_bound_col, event_col=None, ancillary_df=None, show_progress=False, timeline=None, weights_col=None, robust=False, initial_point=None, entry_col=None)

Fit the accelerated failure time model to a left-censored dataset.

**Parameters**

- **df** (*DataFrame*) – a Pandas DataFrame with necessary columns `lower_bound_col`, `upper_bound_col` (see below), and any other covariates or weights.
- **lower_bound_col** (*string*) – the name of the column in DataFrame that contains the subjects’ left-most observation.
- **upper_bound_col** (*string*) – the name of the column in DataFrame that contains the subjects’ right-most observation. Values can be np.inf (and should be if the subject is right-censored).
- **event_col** (*string, optional*) – the name of the column in DataFrame that contains the subjects’ death observation. If left as None, will be inferred from the start and stop columns (lower_bound==upper_bound means uncensored)
- **show_progress** (*boolean, optional (default=False)*) – since the fitter is iterative, show convergence diagnostics. Useful if convergence is failing.
- **ancillary_df** (*None, boolean, or DataFrame, optional (default=None)*) – Choose to model the ancillary parameters. If None or False, explicitly do not fit the ancillary parameters using any covariates. If True, model the ancillary parameters with the same covariates as `df`. If DataFrame, provide covariates to model the ancillary parameters. Must be the same row count as `df`.
- **timeline** (*array, optional*) – Specify a timeline that will be used for plotting and prediction
- **weights_col** (*string*) – the column in DataFrame that specifies weights per observation.
- **robust** (*boolean, optional (default=False)*) – Compute the robust errors using the Huber sandwich estimator.
- **initial_point** (*((d,) numpy array, optional)* – initialize the starting point of the iterative algorithm. Default is the zero vector.
- **entry_col** (*specify a column in the DataFrame that denotes any late-entries (left truncation) that occurred, See an explanation in the docs on left truncation*)

**Returns**

Self with additional new properties: `print_summary`, `params_`, `confidence_intervals_` and more

**Return type**

`self`

**Examples**

```python
>>> from lifelines import WeibullAFTFitter, LogNormalAFTFitter,
→LogLogisticAFTFitter

>>> df = pd.DataFrame({

    'start': [5, 3, 9, 8, 7, 4, 4, 3, 2, 5, 6, 7],
```

(continues on next page)
```python
>>> 'stop': [5, 3, 9, 8, 7, 4, 8, 5, 2, 5, 6, np.inf],  # this last
    subject is right-censored.
>>> 'E': [1, 1, 1, 1, 1, 1, 0, 0, 1, 1, 1, 0],
>>> 'var': [0, 0, 0, 0, 1, 1, 1, 1, 1, 2, 2, 2],
>>> 'age': [4, 3, 9, 8, 7, 4, 4, 3, 2, 5, 6, 7],
>>> }
>>> aft = WeibullAFTFitter()
aft.fit_interval_censoring(df, 'start', 'stop', 'E')
aft.print_summary()
aft.predict_median(df)
>>> aft = WeibullAFTFitter()
aft.fit_interval_censoring(df, 'start', 'stop', 'E', ancillary_df=df)
aft.print_summary()
aft.predict_median(df)
```

### fit_left_censoring
df, duration_col=None, event_col=None, ancillary_df=None, show_progress=False, timeline=None, weights_col=None, robust=False, initial_point=None, entry_col=None

Fit the accelerated failure time model to a left-censored dataset.

**Parameters**

- **df** ([DataFrame](https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.html)) – a Pandas DataFrame with necessary columns `duration_col` and `event_col` (see below), covariates columns, and special columns (weights). `duration_col` refers to the lifetimes of the subjects. `event_col` refers to whether the ‘death’ events was observed: 1 if observed, 0 else (censored).

- **duration_col** ([string](https://docs.python.org/3/library/stdtypes.html#str)) – the name of the column in DataFrame that contains the subjects’ lifetimes/measurements/etc. This column contains the (possibly) left-censored data.

- **event_col** ([string](https://docs.python.org/3/library/stdtypes.html#str), optional) – the name of the column in DataFrame that contains the subjects’ death observation. If left as None, assume all individuals are uncensored.

- **show_progress** ([boolean](https://docs.python.org/3/library/stdtypes.html#bool), optional (default=False)) – since the fitter is iterative, show convergence diagnostics. Useful if convergence is failing.

- **ancillary_df** ([None](https://docs.python.org/3/library/NoneType.html), [boolean](https://docs.python.org/3/library/stdtypes.html#bool), or [DataFrame](https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.html), optional (default=None)) – Choose to model the ancillary parameters. If None or False, explicitly do not fit the ancillary parameters using any covariates. If True, model the ancillary parameters with the same covariates as `df`. If DataFrame, provide covariates to model the ancillary parameters. Must be the same row count as `df`.

- **timeline** ([array](https://docs.python.org/3/library/array.html#module-array), optional) – Specify a timeline that will be used for plotting and prediction

- **weights_col** ([string](https://docs.python.org/3/library/stdtypes.html#str)) – the column in DataFrame that specifies weights per observation.

- **robust** ([boolean](https://docs.python.org/3/library/stdtypes.html#bool), optional (default=False)) – Compute the robust errors using the Huber sandwich estimator.

- **initial_point** ([([d,] numpy array], optional) – initialize the starting point of the iterative algorithm. Default is the zero vector.

- **entry_col** (specify a column in the DataFrame that denotes any late-entries (left truncation) that occurred. See) – the docs on left truncation

**Returns** `self` with additional new properties: `print_summary`, `params_`, `confidence_intervals` and more

**Return type** `self`
Examples

```python
>>> from lifelines import WeibullAFTFitter, LogNormalAFTFitter,
                    LogLogisticAFTFitter

>>> df = pd.DataFrame(
    {'T': [5, 3, 9, 8, 7, 4, 4, 3, 2, 5, 6, 7],
     'E': [1, 1, 1, 1, 1, 0, 0, 1, 1, 1, 1, 0],
     'var': [0, 0, 0, 1, 1, 1, 1, 1, 2, 2, 2, 2],
     'age': [4, 3, 9, 8, 7, 4, 4, 3, 2, 5, 6, 7]},
)

>>> aft = WeibullAFTFitter()
>>> aft.fit_left_censoring(df, 'T', 'E')
>>> aft.print_summary()
>>> aft.predict_median(df)

>>> aft = WeibullAFTFitter()
>>> aft.fit_left_censoring(df, 'T', 'E', ancillary_df=df)
>>> aft.print_summary()
>>> aft.predict_median(df)
```

**fit_right_censoring** (*args, **kwargs)

**log_likelihood_ratio_test** ()

This function computes the likelihood ratio test for the model. We compare the existing model (with all the covariates) to the trivial model of no covariates.

**mean_survival_time_**

**median_survival_time_**

**plot** (columns=None, parameter=None, **errorbar_kwargs)**

Produces a visual representation of the coefficients, including their standard errors and magnitudes.

Parameters

- **columns** *(list, optional)* – specify a subset of the columns to plot
- **errorbar_kwargs** – pass in additional plotting commands to matplotlib errorbar command

Returns **ax** – the matplotlib axis that be edited.

Return type matplotlib axis

**plot_covariate_groups** *(covariates, values, plot_baseline=True, **kwargs)*

Produces a visual representation comparing the baseline survival curve of the model versus what happens when a covariate(s) is varied over values in a group. This is useful to compare subjects’ survival as we vary covariate(s), all else being held equal. The baseline survival curve is equal to the predicted survival curve at all average values in the original dataset.

Parameters

- **covariates** *(string or list)* – a string (or list of strings) of the covariate in the original dataset that we wish to vary.
- **values** *(1d or 2d iterable)* – an iterable of the values we wish the covariate to take on.
- **plot_baseline** *(bool)* – also display the baseline survival, defined as the survival at the mean of the original dataset.
- **kwargs** – pass in additional plotting commands
Returns `ax` – the matplotlib axis that be edited.

Return type matplotlib axis, or list of axis’

Examples

```python
>>> from lifelines import datasets, WeibullAFTFitter
>>> rossi = datasets.load_rossi()
>>> wf = WeibullAFTFitter().fit(rossi, 'week', 'arrest')
>>> wf.plot_covariate_groups('prio', values=np.arange(0, 15), cmap='coolwarm')

# multiple variables at once
>>> wf.plot_covariate_groups(['prio', 'paro'], values=[[0, 0], [5, 0], [10, 0], [0, 1], [5, 1], [10, 1]], cmap='coolwarm')

# if you have categorical variables, you can simply things:
>>> wf.plot_covariate_groups(['dummy1', 'dummy2', 'dummy3'], values=np.eye(3))
```

`predict_cumulative_hazard` *(X, times=None, ancillary_X=None)*

Return the cumulative hazard rate of subjects in X at time points.

Parameters

- `X` *(numpy array or DataFrame)* – a (n,d) covariate numpy array or DataFrame. If a DataFrame, columns can be in any order. If a numpy array, columns must be in the same order as the training data.

- `times` *(iterable, optional)* – an iterable of increasing times to predict the cumulative hazard at. Default is the set of all durations (observed and unobserved). Uses a linear interpolation if points in time are not in the index.

- `ancillary_X` *(numpy array or DataFrame, optional)* – a (n,d) covariate numpy array or DataFrame. If a DataFrame, columns can be in any order. If a numpy array, columns must be in the same order as the training data.

Returns `cumulative_hazard_` – the cumulative hazard of individuals over the timeline

Return type DataFrame

`predict_expectation` *(X, ancillary_X=None)*

Predict the expectation of lifetimes, \( E[T|x] \).

Parameters

- `X` *(numpy array or DataFrame)* – a (n,d) covariate numpy array or DataFrame. If a DataFrame, columns can be in any order. If a numpy array, columns must be in the same order as the training data.

- `ancillary_X` *(numpy array or DataFrame, optional)* – a (n,d) covariate numpy array or DataFrame. If a DataFrame, columns can be in any order. If a numpy array, columns must be in the same order as the training data.

Returns `percentiles` – the median lifetimes for the individuals. If the survival curve of an individual does not cross 0.5, then the result is infinity.

Return type DataFrame

See also:

`predict_median()`
**predict_median** *(X, ancillary_X=None)*

Predict the median lifetimes for the individuals. If the survival curve of an individual does not cross 0.5, then the result is infinity.

**Parameters**

- **X** *(numpy array or DataFrame)* – a (n,d) covariate numpy array or DataFrame. If a DataFrame, columns can be in any order. If a numpy array, columns must be in the same order as the training data.

- **ancillary_X** *(numpy array or DataFrame, optional)* – a (n,d) covariate numpy array or DataFrame. If a DataFrame, columns can be in any order. If a numpy array, columns must be in the same order as the training data.

**Returns percentiles** – the median lifetimes for the individuals. If the survival curve of an individual does not cross 0.5, then the result is infinity.

**Return type** DataFrame

**See also:**

`predict_percentile()` , `predict_expectation()`

**predict_percentile** *(X, ancillary_X=None, p=0.5)*

Returns the median lifetimes for the individuals, by default. If the survival curve of an individual does not cross 0.5, then the result is infinity. [http://stats.stackexchange.com/questions/102986/percentile-loss-functions](http://stats.stackexchange.com/questions/102986/percentile-loss-functions)

**Parameters**

- **X** *(numpy array or DataFrame)* – a (n,d) covariate numpy array or DataFrame. If a DataFrame, columns can be in any order. If a numpy array, columns must be in the same order as the training data.

- **ancillary_X** *(numpy array or DataFrame, optional)* – a (n,d) covariate numpy array or DataFrame. If a DataFrame, columns can be in any order. If a numpy array, columns must be in the same order as the training data.

- **p** *(float, optional (default=0.5))* – the percentile, must be between 0 and 1.

**Returns percentiles**

**Return type** DataFrame

**See also:**

`predict_median()`

**predict_survival_function** *(X, times=None, ancillary_X=None)*

Predict the survival function for individuals, given their covariates. This assumes that the individual just entered the study (that is, we do not condition on how long they have already lived for.)

**Parameters**

- **X** *(numpy array or DataFrame)* – a (n,d) covariate numpy array or DataFrame. If a DataFrame, columns can be in any order. If a numpy array, columns must be in the same order as the training data.

- **ancillary_X** *(numpy array or DataFrame, optional)* – a (n,d) covariate numpy array or DataFrame. If a DataFrame, columns can be in any order. If a numpy array, columns must be in the same order as the training data.
• **times** *(iterable, optional)* – an iterable of increasing times to predict the cumulative hazard at. Default is the set of all durations (observed and unobserved). Uses a linear interpolation if points in time are not in the index.

**Returns** `survival_function` – the survival probabilities of individuals over the timeline

**Return type** DataFrame

**print_summary** *(decimals=2, **kwargs)*

Print summary statistics describing the fit, the coefficients, and the error bounds.

**Parameters**

• **decimals** *(int, optional (default=2))* – specify the number of decimal places to show

• **alpha** *(float or iterable)* – specify confidence intervals to show

• **kwargs** – print additional metadata in the output (useful to provide model names, dataset names, etc.) when comparing multiple outputs.

**score_**

The concordance score (also known as the c-index) of the fit. The c-index is a generalization of the ROC AUC to survival data, including censorships.

For this purpose, the `score_` is a measure of the predictive accuracy of the fitted model onto the training dataset.

**summary**

Summary statistics describing the fit.

**Returns** `df` – Contains columns `coef`, `np.exp(coef)`, `se(coef)`, `z`, `p`, `lower`, `upper`

**Return type** DataFrame

### `lifelines.fitters.log_normal_aft_fitter` module

**class** `lifelines.fitters.log_normal_aft_fitter.LogNormalAFTFitter` *(alpha=0.05, penalizer=0.0, l1_ratio=0.0, fit_intercept=True)*

**Bases:** `lifelines.fitters.ParametericAFTRegressionFitter`

This class implements a Log-Normal AFT model. The model has parameterized form, with \( \mu(x) = \exp(a_0 + a_1 x_1 + \ldots + a_n x_n) \), and optionally, \( \sigma(y) = \exp(b_0 + b_1 y_1 + \ldots + b_m y_m) \).

The cumulative hazard rate is

\[
H(t; x, y) = -\log(1 - \Phi\left(\frac{\log(T) - \mu(x)}{\sigma(y)}\right))
\]

After calling the `.fit` method, you have access to properties like: `params_`, `print_summary()`. A summary of the fit is available with the method `print_summary()`.

**Parameters**

• **alpha** *(float, optional (default=0.05))* – the level in the confidence intervals.

• **fit_intercept** *(boolean, optional (default=True))* – Allow lifelines to add an intercept column of 1s to df, and ancillary_df if applicable.

• **penalizer** *(float, optional (default=0.0))* – the penalizer coefficient to the size of the coefficients. See `l1_ratio`. Must be equal to or greater than 0.
• \texttt{l1\_ratio} (\texttt{float, optional (default=0.0)}) – how much of the penalizer should be attributed to an l1 penalty (otherwise an l2 penalty). The penalty function looks like \(\text{penalizer} \times \text{l1\_ratio} \times ||w||_1 + 0.5 \times \text{penalizer} \times (1 - \text{l1\_ratio}) \times ||w||^2_2\)

\texttt{params\_}
The estimated coefficients
\texttt{Type} DataFrame

\texttt{confidence\_intervals\_}
The lower and upper confidence intervals for the coefficients
\texttt{Type} DataFrame

\texttt{durations}
The event\_observed variable provided
\texttt{Type} Series

\texttt{event\_observed}
The event\_observed variable provided
\texttt{Type} Series

\texttt{weights}
The event\_observed variable provided
\texttt{Type} Series

\texttt{variance\_matrix\_}
The variance matrix of the coefficients
\texttt{Type} numpy array

\texttt{standard\_errors\_}
the standard errors of the estimates
\texttt{Type} Series

\texttt{score\_}
the concordance index of the model.
\texttt{Type} float

\texttt{fit (df, duration\_col, event\_col=None, ancillary\_df=None, show\_progress=False, timeline=None, weights\_col=None, robust=False, initial\_point=None, entry\_col=None)}
Fit the accelerated failure time model to a right-censored dataset.

\textbf{Parameters}

• \texttt{df} (\texttt{DataFrame}) – a Pandas DataFrame with necessary columns duration\_col and event\_col (see below), covariates columns, and special columns (weights). duration\_col refers to the lifetimes of the subjects. event\_col refers to whether the ‘death’ events was observed: 1 if observed, 0 else (censored).

• \texttt{duration\_col} (\texttt{string}) – the name of the column in DataFrame that contains the subjects’ lifetimes.

• \texttt{event\_col} (\texttt{string, optional}) – the name of the column in DataFrame that contains the subjects’ death observation. If left as None, assume all individuals are uncensored.

• \texttt{show\_progress} (\texttt{boolean, optional (default=False)}) – since the fitter is iterative, show convergence diagnostics. Useful if convergence is failing.
• **ancillary_df** (*None*, boolean, or DataFrame, optional (default=None)) – Choose to model the ancillary parameters. If None or False, explicitly do not fit the ancillary parameters using any covariates. If True, model the ancillary parameters with the same covariates as df. If DataFrame, provide covariates to model the ancillary parameters. Must be the same row count as df.

• **timeline** (*array, optional*) – Specify a timeline that will be used for plotting and prediction

• **weights_col** (*string*) – the column in DataFrame that specifies weights per observation.

• **robust** (boolean, optional (default=False)) – Compute the robust errors using the Huber sandwich estimator.

• **initial_point** (*(d,) numpy array, optional*) – initialize the starting point of the iterative algorithm. Default is the zero vector.

• **entry_col** (specify a column in the DataFrame that denotes any late-entries (left truncation) that occurred. See) – the docs on left truncation

**Returns** self with additional new properties: print_summary, params_, confidence_intervals and more

**Return type** self

**Examples**

```python
df = pd.DataFrame({
    'T': [5, 3, 9, 8, 7, 4, 4, 3, 2, 5, 6, 7],
    'E': [1, 1, 1, 1, 1, 0, 0, 1, 1, 1, 1, 0],
    'var': [0, 0, 0, 1, 1, 1, 1, 1, 2, 2, 2],
    'age': [4, 3, 9, 8, 7, 4, 4, 3, 2, 5, 6, 7],
})
```

```python
from lifelines import WeibullAFTFitter, LogNormalAFTFitter, LogLogisticAFTFitter
```

```python
aft = WeibullAFTFitter()
aft.fit(df, 'T', 'E')
aft.print_summary()
aft.predict_median(df)
```

```python
aft = WeibullAFTFitter()
aft.fit(df, 'T', 'E', ancillary_df=df)
aft.print_summary()
aft.predict_median(df)
```
• **upper_bound_col** *(string)* – the name of the column in DataFrame that contains the subjects’ right-most observation. Values can be np.inf (and should be if the subject is right-censored).

• **event_col** *(string, optional)* – the name of the column in DataFrame that contains the subjects’ death observation. If left as None, will be inferred from the start and stop columns (lower_bound==upper_bound means uncensored).

• **show_progress** *(boolean, optional (default=False))* – since the fitter is iterative, show convergence diagnostics. Useful if convergence is failing.

• **ancillary_df** *(None, boolean, or DataFrame, optional (default=None))* – Choose to model the ancillary parameters. If None or False, explicitly do not fit the ancillary parameters using any covariates. If True, model the ancillary parameters with the same covariates as df. If DataFrame, provide covariates to model the ancillary parameters. Must be the same row count as df.

• **timeline** *(array, optional)* – Specify a timeline that will be used for plotting and prediction.

• **weights_col** *(string)* – the column in DataFrame that specifies weights per observation.

• **robust** *(boolean, optional (default=False))* – Compute the robust errors using the Huber sandwich estimator.

• **initial_point** *(\(d,\) numpy array, optional)* – initialize the starting point of the iterative algorithm. Default is the zero vector.

• **entry_col** *(specify a column in the DataFrame that denotes any late-entries (left truncation) that occurred. See)* – the docs on left truncation.

Returns `self` with additional new properties: `print_summary`, `params_`, `confidence_intervals` and more.

**Return type** `self`

### Examples

```python
>>> from lifelines import WeibullAFTFitter, LogNormalAFTFitter,
  \LogLogisticAFTFitter
>>> df = pd.DataFrame({
...     'start': [5, 3, 9, 8, 7, 4, 4, 3, 2, 5, 6, 7],
...     'stop': [5, 3, 9, 8, 7, 4, 8, 5, 2, 5, 6, np.inf], # this last subject is right-censored.
...     'E': [1, 1, 1, 1, 1, 0, 0, 1, 1, 1, 0, 0],
...     'var': [0, 0, 0, 1, 1, 1, 1, 2, 2, 2, 2, 2],
...     'age': [4, 3, 9, 8, 7, 4, 4, 3, 2, 5, 6, 7],
... })
>>> aft = WeibullAFTFitter()
>>> aft.fit_interval_censoring(df, 'start', 'stop', 'E')
>>> aft.print_summary()
>>> aft.predict_median(df)
```
fit_left_censoring(df, duration_col=None, event_col=None, ancillary_df=None, show_progress=False, timeline=None, weights_col=None, robust=False, initial_point=None, entry_col=None)

Fit the accelerated failure time model to a left-censored dataset.

Parameters

- **df** (DataFrame) – a Pandas DataFrame with necessary columns `duration_col` and `event_col` (see below), covariates columns, and special columns (weights). `duration_col` refers to the lifetimes of the subjects. `event_col` refers to whether the ‘death’ events was observed: 1 if observed, 0 else (censored).

- **duration_col** (string) – the name of the column in DataFrame that contains the subjects’ lifetimes/measurements/etc. This column contains the (possibly) left-censored data.

- **event_col** (string, optional) – the name of the column in DataFrame that contains the subjects’ death observation. If left as None, assume all individuals are uncensored.

- **show_progress** (boolean, optional (default=False)) – since the fitter is iterative, show convergence diagnostics. Useful if convergence is failing.

- **ancillary_df** (None, boolean, or DataFrame, optional (default=None)) – Choose to model the ancillary parameters. If None or False, explicitly do not fit the ancillary parameters using any covariates. If True, model the ancillary parameters with the same covariates as df. If DataFrame, provide covariates to model the ancillary parameters. Must be the same row count as df.

- **timeline** (array, optional) – Specify a timeline that will be used for plotting and prediction

- **weights_col** (string) – the column in DataFrame that specifies weights per observation.

- **robust** (boolean, optional (default=False)) – Compute the robust errors using the Huber sandwich estimator.

- **initial_point** ((d,) numpy array, optional) – initialize the starting point of the iterative algorithm. Default is the zero vector.

- **entry_col** (specify a column in the DataFrame that denotes any late-entries (left truncation) that occurred. See the docs on left truncation

Returns self with additional new properties: print_summary, params_, confidence_intervals_and more

Return type self

Examples

```python
>>> from lifelines import WeibullAFTFitter, LogNormalAFTFitter, LogLogisticAFTFitter
>>> df = pd.DataFrame({
...     'T': [5, 3, 9, 8, 7, 4, 4, 3, 2, 5, 6, 7],
...     'E': [1, 1, 1, 1, 1, 0, 0, 1, 1, 1, 1, 0],
...     'var': [0, 0, 0, 0, 1, 1, 1, 1, 2, 2, 2, 1],
...     'age': [4, 3, 9, 8, 7, 4, 4, 3, 2, 5, 6, 7],
... })
>>> afl = WeibullAFTFitter()
>>> afl.fit_left_censoring(df, 'T', 'E')
>>> afl.print_summary()
```

(continues on next page)
```python
>>> aft.predict_median(df)
>>> aft = WeibullAFTFitter()
>>> aft.fit_left_censoring(df, 'T', 'E', ancillary_df=df)
>>> aft.print_summary()
>>> aft.predict_median(df)
```

### fit_right_censoring(*args, **kwargs)

### log_likelihood_ratio_test()

This function computes the likelihood ratio test for the model. We compare the existing model (with all the covariates) to the trivial model of no covariates.

### mean_survival_time_

### median_survival_time_

### plot(columns=None, parameter=None, **errorbar_kwargs)

Produces a visual representation of the coefficients, including their standard errors and magnitudes.

**Parameters**

- **columns** (list, optional) – specify a subset of the columns to plot
- **errorbar_kwargs** – pass in additional plotting commands to matplotlib errorbar command

**Returns**

- **ax** – the matplotlib axis that be edited.

**Return type**

matplotlib axis

### plot_covariate_groups(covariates, values, plot_baseline=True, **kwargs)

Produces a visual representation comparing the baseline survival curve of the model versus what happens when a covariate(s) is varied over values in a group. This is useful to compare subjects’ survival as we vary covariate(s), all else being held equal. The baseline survival curve is equal to the predicted survival curve at all average values in the original dataset.

**Parameters**

- **covariates** (string or list) – a string (or list of strings) of the covariate in the original dataset that we wish to vary.
- **values** (1d or 2d iterable) – an iterable of the values we wish the covariate to take on.
- **plot_baseline** (bool) – also display the baseline survival, defined as the survival at the mean of the original dataset.
- **kwargs** – pass in additional plotting commands

**Returns**

- **ax** – the matplotlib axis that be edited.

**Return type**

matplotlib axis, or list of axis’

### Examples

```python
>>> from lifelines import datasets, WeibullAFTFitter
>>> rossi = datasets.load_rossi()
>>> wf = WeibullAFTFitter().fit(rossi, 'week', 'arrest')
>>> wf.plot_covariate_groups('prio', values=np.arange(0, 15), cmap='coolwarm')
```
predict_cumulative_hazard (X, times=None, ancillary_X=None)

Return the cumulative hazard rate of subjects in X at time points.

Parameters

- X (numpy array or DataFrame) – a (n,d) covariate numpy array or DataFrame. If a
  DataFrame, columns can be in any order. If a numpy array, columns must be in the same
  order as the training data.
- times (iterable, optional) – an iterable of increasing times to predict the cumulative hazard
  at. Default is the set of all durations (observed and unobserved). Uses a linear interpolation
  if points in time are not in the index.
- ancillary_X (numpy array or DataFrame, optional) – a (n,d) covariate numpy array or
  DataFrame. If a DataFrame, columns can be in any order. If a numpy array, columns must
  be in the same order as the training data.

Returns cumulative_hazard_ – the cumulative hazard of individuals over the timeline

Return type DataFrame

predict_expectation (X, ancillary_X=None)

Predict the expectation of lifetimes, \( \mathbb{E}[T|x] \).

Parameters

- X (numpy array or DataFrame) – a (n,d) covariate numpy array or DataFrame. If a
  DataFrame, columns can be in any order. If a numpy array, columns must be in the same
  order as the training data.
- ancillary_X (numpy array or DataFrame, optional) – a (n,d) covariate numpy array or
  DataFrame. If a DataFrame, columns can be in any order. If a numpy array, columns must
  be in the same order as the training data.

Returns percentiles – the median lifetimes for the individuals. If the survival curve of an indi-
  vidual does not cross 0.5, then the result is infinity.

Return type DataFrame

See also:

predict_median()

predict_median (X, ancillary_X=None)

Returns the median lifetimes for the individuals. If the survival curve of an individual does not cross 0.5,
then the result is infinity. http://stats.stackexchange.com/questions/102986/percentile-loss-functions

Parameters

- X (numpy array or DataFrame) – a (n,d) covariate numpy array or DataFrame. If a
  DataFrame, columns can be in any order. If a numpy array, columns must be in the same
  order as the training data.
• **ancillary_X** (*numpy array or DataFrame, optional*) – a (n,d) covariate numpy array or DataFrame. If a DataFrame, columns can be in any order. If a numpy array, columns must be in the same order as the training data.

• **p** (*float, optional (default=0.5)*) – the percentile, must be between 0 and 1.

Returns

Return type DataFrame

See also:

`predict_percentile()`

**predict_percentile** (*X, ancillary_X=None, p=0.5*)

Returns the median lifetimes for the individuals, by default. If the survival curve of an individual does not cross p, then the result is infinity. [http://stats.stackexchange.com/questions/102986/percentile-loss-functions](http://stats.stackexchange.com/questions/102986/percentile-loss-functions)

Parameters

• **X** (*numpy array or DataFrame*) – a (n,d) covariate numpy array or DataFrame. If a DataFrame, columns can be in any order. If a numpy array, columns must be in the same order as the training data.

• **ancillary_X** (*numpy array or DataFrame, optional*) – a (n,d) covariate numpy array or DataFrame. If a DataFrame, columns can be in any order. If a numpy array, columns must be in the same order as the training data.

• **p** (*float, optional (default=0.5)*) – the percentile, must be between 0 and 1.

Returns percentiles

Return type DataFrame

See also:

`predict_median()`

**predict_survival_function** (*X, times=None, ancillary_X=None*)

Predict the survival function for individuals, given their covariates. This assumes that the individual just entered the study (that is, we do not condition on how long they have already lived for.)

Parameters

• **X** (*numpy array or DataFrame*) – a (n,d) covariate numpy array or DataFrame. If a DataFrame, columns can be in any order. If a numpy array, columns must be in the same order as the training data.

• **ancillary_X** (*numpy array or DataFrame, optional*) – a (n,d) covariate numpy array or DataFrame. If a DataFrame, columns can be in any order. If a numpy array, columns must be in the same order as the training data.

• **times** (*iterable, optional*) – an iterable of increasing times to predict the cumulative hazard at. Default is the set of all durations (observed and unobserved). Uses a linear interpolation if points in time are not in the index.

Returns survival_function – the survival probabilities of individuals over the timeline

Return type DataFrame

**print_summary** (*decimals=2, **kwargs*)

Print summary statistics describing the fit, the coefficients, and the error bounds.

Parameters
• **decimals** *(int, optional (default=2)) – specify the number of decimal places to show*

• **alpha** *(float or iterable) – specify confidence intervals to show*

• **kwargs** – print additional metadata in the output (useful to provide model names, dataset names, etc.) when comparing multiple outputs.

**score_**
The concordance score (also known as the c-index) of the fit. The c-index is a generalization of the ROC AUC to survival data, including censorships.

For this purpose, the score_ is a measure of the predictive accuracy of the fitted model onto the training dataset.

**summary**
Summary statistics describing the fit.

Returns df – Contains columns coef, np.exp(coef), se(coef), z, p, lower, upper

Return type DataFrame

---

**lifelines.fitters.log_logistic_aft_fitter module**

class lifelines.fitters.log_logistic_aft_fitter.LogisticAFTFitter(alpha=0.05, penalizer=0.0, l1_ratio=0.0, fit_intercept=True)

Bases: lifelines.fitters.ParametericAFTRegressionFitter

This class implements a Log-Logistic AFT model. The model has parameterized form, with \( \alpha(x) = \exp(a_0 + a_1 x_1 + ... + a_n x_n) \), and optionally, \( \beta(y) = \exp(b_0 + b_1 y_1 + ... + b_m y_m) \).

The cumulative hazard rate is

\[
H(t; x, y) = \log \left( 1 + \left( \frac{t}{\alpha(x)} \right)^\beta(y) \right)
\]

After calling the .fit method, you have access to properties like: params_, print_summary(). A summary of the fit is available with the method print_summary().

**Parameters**

• **alpha** *(float, optional (default=0.05)) – the level in the confidence intervals.*

• **fit_intercept** *(boolean, optional (default=True)) – Allow lifelines to add an intercept column of 1s to df, and ancillary_df if applicable.*

• **penalizer** *(float, optional (default=0.0)) – the penalizer coefficient to the size of the coefficients. See l1_ratio. Must be equal to or greater than 0.*

• **l1_ratio** *(float, optional (default=0.0)) – how much of the penalizer should be attributed to an l1 penalty (otherwise an l2 penalty). The penalty function looks like penalizer * l1_ratio * \|w\|_1 + 0.5 * penalizer * (1 - l1_ratio) * \|w\|^2_2

**params_**
The estimated coefficients

Type DataFrame
confidence_intervals_
The lower and upper confidence intervals for the coefficients

Type DataFrame

durations
The event_observed variable provided

Type Series
event_observed
The event_observed variable provided

Type Series
weights
The event_observed variable provided

Type Series

variance_matrix_
The variance matrix of the coefficients

Type numpy array

standard_errors_
the standard errors of the estimates

Type Series

score_
the concordance index of the model.

Type float

fit (df, duration_col=None, event_col=None, ancillary_df=None, show_progress=False, timeline=None, weights_col=None, robust=False, initial_point=None, entry_col=None)
Fit the accelerated failure time model to a right-censored dataset.

Parameters

• df (DataFrame) – a Pandas DataFrame with necessary columns duration_col and event_col (see below), covariates columns, and special columns (weights). duration_col refers to the lifetimes of the subjects. event_col refers to whether the ‘death’ events was observed: 1 if observed, 0 else (censored).

• duration_col (string) – the name of the column in DataFrame that contains the subjects’ lifetimes.

• event_col (string, optional) – the name of the column in DataFrame that contains the subjects’ death observation. If left as None, assume all individuals are uncensored.

• show_progress (boolean, optional (default=False)) – since the fitter is iterative, show convergence diagnostics. Useful if convergence is failing.

• ancillary_df (None, boolean, or DataFrame, optional (default=None)) – Choose to model the ancillary parameters. If None or False, explicitly do not fit the ancillary parameters using any covariates. If True, model the ancillary parameters with the same covariates as df. If DataFrame, provide covariates to model the ancillary parameters. Must be the same row count as df.

• timeline (array, optional) – Specify a timeline that will be used for plotting and prediction

• weights_col (string) – the column in DataFrame that specifies weights per observation.
• **robust** *(boolean, optional (default=False)) – Compute the robust errors using the Huber sandwich estimator.*

• **initial_point** *(d,) numpy array, optional) – initialize the starting point of the iterative algorithm. Default is the zero vector.*

• **entry_col** *(specify a column in the DataFrame that denotes any late-entries (left truncation) that occurred. See) – the docs on left truncation*

Returns self with additional new properties: print_summary, params_, confidence_intervals_ and more

Return type self

Examples

```python
>>> from lifelines import WeibullAFTFitter, LogNormalAFTFitter, LogLogisticAFTFitter

>>> df = pd.DataFrame(
>>>    {'T': [5, 3, 9, 8, 7, 4, 4, 3, 2, 5, 6, 7],
>>>     'E': [1, 1, 1, 1, 1, 0, 0, 1, 1, 1, 1, 0],
>>>     'var': [0, 0, 0, 0, 1, 0, 1, 1, 2, 2, 2, 2],
>>>     'age': [4, 3, 9, 8, 7, 4, 4, 3, 2, 5, 6, 7],
>>> )

>>> aft = WeibullAFTFitter()
>>> aft.fit(df, 'T', 'E')
>>> aft.print_summary()
>>> aft.predict_median(df)

>>> aft = WeibullAFTFitter()
>>> aft.fit(df, 'T', 'E', ancillary_df=df)
>>> aft.print_summary()
>>> aft.predict_median(df)
```

**fit_interval_censoring** *(df, lower_bound_col, upper_bound_col, event_col=None, ancillary_df=None, show_progress=False, timeline=None, weights_col=None, robust=False, initial_point=None, entry_col=None)*

Fit the accelerated failure time model to a left-censored dataset.

Parameters

• **df** *(DataFrame) – a Pandas DataFrame with necessary columns lower_bound_col, upper_bound_col (see below), and any other covariates or weights.*

• **lower_bound_col** *(string) – the name of the column in DataFrame that contains the subjects’ left-most observation.*

• **upper_bound_col** *(string) – the name of the column in DataFrame that contains the subjects’ right-most observation. Values can be np.inf (and should be if the subject is right-censored).*

• **event_col** *(string, optional) – the name of the column in DataFrame that contains the subjects’ death observation. If left as None, will be inferred from the start and stop columns (lower_bound==upper_bound means uncensored)*

• **show_progress** *(boolean, optional (default=False)) – since the fitter is iterative, show convergence diagnostics. Useful if convergence is failing.*
• **ancillary_df** *(None, boolean, or DataFrame, optional (default=None))* – Choose to model the ancillary parameters. If None or False, explicitly do not fit the ancillary parameters using any covariates. If True, model the ancillary parameters with the same covariates as df. If DataFrame, provide covariates to model the ancillary parameters. Must be the same row count as df.

• **timeline** *(array, optional)* – Specify a timeline that will be used for plotting and prediction

• **weights_col** *(string)* – the column in DataFrame that specifies weights per observation.

• **robust** *(boolean, optional (default=False))* – Compute the robust errors using the Huber sandwich estimator.

• **initial_point** *((d,) numpy array, optional)* – initialize the starting point of the iterative algorithm. Default is the zero vector.

• **entry_col** *(specify a column in the DataFrame that denotes any late-entries (left truncation) that occurred. See)* – the docs on left truncation

Returns self with additional new properties: `print_summary`, `params_`, `confidence_intervals_` and more

Return type self

Examples

```python
>>> from lifelines import WeibullAFTFitter, LogNormalAFTFitter, LogLogisticAFTFitter

aft = WeibullAFTFitter()
aft.fit_interval_censoring(df, 'start', 'stop', 'E')
aft.print_summary()
aft.predict_median(df)

aft = WeibullAFTFitter()
aft.fit_interval_censoring(df, 'start', 'stop', 'E', ancillary_df=df)
aft.print_summary()
aft.predict_median(df)
```

`fit_left_censoring` *(df, duration_col=None, event_col=None, ancillary_df=None, show_progress=False, timeline=None, weights_col=None, robust=False, initial_point=None, entry_col=None)*

Fit the accelerated failure time model to a left-censored dataset.

Parameters

• **df** *(DataFrame)* – a Pandas DataFrame with necessary columns `duration_col` and `event_col` (see below), covariates columns, and special columns (weights). `duration_col` refers to the lifetimes of the subjects. `event_col` refers to whether the ‘death’ events was observed: 1 if observed, 0 else (censored).
• duration_col (string) – the name of the column in DataFrame that contains the subjects’ lifetimes/measurements/etc. This column contains the (possibly) left-censored data.

• event_col (string, optional) – the name of the column in DataFrame that contains the subjects’ death observation. If left as None, assume all individuals are uncensored.

• show_progress (boolean, optional (default=False)) – since the fitter is iterative, show convergence diagnostics. Useful if convergence is failing.

• ancillary_df (None, boolean, or DataFrame, optional (default=None)) – Choose to model the ancillary parameters. If None or False, explicitly do not fit the ancillary parameters using any covariates. If True, model the ancillary parameters with the same covariates as df. If DataFrame, provide covariates to model the ancillary parameters. Must be the same row count as df.

• timeline (array, optional) – Specify a timeline that will be used for plotting and prediction

• weights_col (string) – the column in DataFrame that specifies weights per observation.

• robust (boolean, optional (default=False)) – Compute the robust errors using the Huber sandwich estimator.

• initial_point ((d,) numpy array, optional) – initialize the starting point of the iterative algorithm. Default is the zero vector.

• entry_col (specify a column in the DataFrame that denotes any late-entries (left truncation) that occurred. See) – the docs on left truncation

Returns self with additional new properties: print_summary, params_, confidence_intervals_and more

Return type self

Examples

```python
>>> from lifelines import WeibullAFTFitter, LogNormalAFTFitter,
   LogLogisticAFTFitter
   ...
   ...
   ...
   df = pd.DataFrame({
   ...
   'T': [5.3, 9.8, 7.4, 4.3, 2.5, 6.7],
   ...
   'E': [1.0, 1.0, 1.0, 1.0, 0.0, 1.0, 1.0, 0.0, 1.0, 2.2, 1.1, 2.2],
   ...
   'var': [0.0, 0.0, 0.1, 1.1, 1.1, 2.2, 2.2],
   ...
   'age': [4.3, 9.8, 7.4, 4.3, 2.5, 6.7], ...
   ...
   ...
   })
   ...
   aft = WeibullAFTFitter()
   ...
   aft.fit_left_censoring(df, 'T', 'E')
   ...
   aft.print_summary()
   ...
   aft.predict_median(df)
   ...
   aft = WeibullAFTFitter()
   ...
   aft.fit_left_censoring(df, 'T', 'E', ancillary_df=df)
   ...
   aft.print_summary()
   ...
   aft.predict_median(df)
   ...
   ...
```

fit_right_censoring (*args, **kwargs)

log_likelihood_ratio_test()

This function computes the likelihood ratio test for the model. We compare the existing model (with all the covariates) to the trivial model of no covariates.
**mean_survival_time_**

**median_survival_time_**

**plot** (*columns=None, parameter=None, **errorbar_kwargs*)

Produces a visual representation of the coefficients, including their standard errors and magnitudes.

**Parameters**

- **columns** *(list, optional)* – specify a subset of the columns to plot
- **errorbar_kwargs** – pass in additional plotting commands to matplotlib errorbar command

**Returns**

- **ax** – the matplotlib axis that be edited.

**Return type**

matplotlib axis

**plot_covariate_groups** *(covariates, values, plot_baseline=True, **kwargs]*)

Produces a visual representation comparing the baseline survival curve of the model versus what happens when a covariate(s) is varied over values in a group. This is useful to compare subjects’ survival as we vary covariate(s), all else being held equal. The baseline survival curve is equal to the predicted survival curve at all average values in the original dataset.

**Parameters**

- **covariates** *(string or list)* – a string (or list of strings) of the covariate in the original dataset that we wish to vary.
- **values** *(1d or 2d iterable)* – an iterable of the values we wish the covariate to take on.
- **plot_baseline** *(bool)* – also display the baseline survival, defined as the survival at the mean of the original dataset.
- **kwargs** – pass in additional plotting commands

**Returns**

- **ax** – the matplotlib axis that be edited.

**Return type**

matplotlib axis, or list of axis’

**Examples**

```python
>>> from lifelines import datasets, WeibullAFTFitter
>>> rossi = datasets.load_rossi()
>>> wf = WeibullAFTFitter().fit(rossi, 'week', 'arrest')
>>> wf.plot_covariate_groups('prio', values=np.arange(0, 15), cmap='coolwarm')

>>> # multiple variables at once
>>> wf.plot_covariate_groups(['prio', 'paro'], values=[[0, 0], [5, 0], [10, 0], [0, 1], [5, 1], [10, 1]], cmap='coolwarm')

>>> # if you have categorical variables, you can simply things:
>>> wf.plot_covariate_groups(['dummy1', 'dummy2', 'dummy3'], values=np.eye(3))
```

**predict_cumulative_hazard** *(X, times=None, ancillary_X=None)*

Return the cumulative hazard rate of subjects in X at time points.

**Parameters**

- **X** *(numpy array or DataFrame)* – a (n,d) covariate numpy array or DataFrame. If a DataFrame, columns can be in any order. If a numpy array, columns must be in the same order as the training data.
• **times** *(iterable, optional)* – an iterable of increasing times to predict the cumulative hazard at. Default is the set of all durations (observed and unobserved). Uses a linear interpolation if points in time are not in the index.

• **ancillary_X** *(numpy array or DataFrame, optional)* – a (n,d) covariate numpy array or DataFrame. If a DataFrame, columns can be in any order. If a numpy array, columns must be in the same order as the training data.

Returns **cumulative_hazard_** – the cumulative hazard of individuals over the timeline

Return type DataFrame

**predict_expectation** *(X, ancillary_X=None)*

Predict the expectation of lifetimes, $E[T|x]$.

Parameters

• **X** *(numpy array or DataFrame)* – a (n,d) covariate numpy array or DataFrame. If a DataFrame, columns can be in any order. If a numpy array, columns must be in the same order as the training data.

• **ancillary_X** *(numpy array or DataFrame, optional)* – a (n,d) covariate numpy array or DataFrame. If a DataFrame, columns can be in any order. If a numpy array, columns must be in the same order as the training data.

Returns **percentiles** – the median lifetimes for the individuals. If the survival curve of an individual does not cross $0.5$, then the result is infinity.

Return type DataFrame

See also:

**predict_median()**

**predict_median** *(X, ancillary_X=None)*

Predict the median lifetimes for the individuals. If the survival curve of an individual does not cross $0.5$, then the result is infinity.

Parameters

• **X** *(numpy array or DataFrame)* – a (n,d) covariate numpy array or DataFrame. If a DataFrame, columns can be in any order. If a numpy array, columns must be in the same order as the training data.

• **ancillary_X** *(numpy array or DataFrame, optional)* – a (n,d) covariate numpy array or DataFrame. If a DataFrame, columns can be in any order. If a numpy array, columns must be in the same order as the training data.

Returns **percentiles** – the median lifetimes for the individuals. If the survival curve of an individual does not cross $0.5$, then the result is infinity.

Return type DataFrame

See also:

**predict_percentile(), predict_expectation()**

**predict_percentile** *(X, ancillary_X=None, p=0.5)*

Returns the median lifetimes for the individuals, by default. If the survival curve of an individual does not cross $p$, then the result is infinity. [Link](http://stats.stackexchange.com/questions/102986/percentile-loss-functions)

Parameters
• X (numpy array or DataFrame) – a (n,d) covariate numpy array or DataFrame. If a DataFrame, columns can be in any order. If a numpy array, columns must be in the same order as the training data.

• ancillary_X (numpy array or DataFrame, optional) – a (n,d) covariate numpy array or DataFrame. If a DataFrame, columns can be in any order. If a numpy array, columns must be in the same order as the training data.

• p (float, optional (default=0.5)) – the percentile, must be between 0 and 1.

Returns percentiles

Return type DataFrame

See also:

predict_median()

predict_survival_function (X, times=None, ancillary_X=None)
Predict the survival function for individuals, given their covariates. This assumes that the individual just entered the study (that is, we do not condition on how long they have already lived for.)

Parameters

• X (numpy array or DataFrame) – a (n,d) covariate numpy array or DataFrame. If a DataFrame, columns can be in any order. If a numpy array, columns must be in the same order as the training data.

• ancillary_X (numpy array or DataFrame, optional) – a (n,d) covariate numpy array or DataFrame. If a DataFrame, columns can be in any order. If a numpy array, columns must be in the same order as the training data.

• times (iterable, optional) – an iterable of increasing times to predict the cumulative hazard at. Default is the set of all durations (observed and unobserved). Uses a linear interpolation if points in time are not in the index.

Returns survival_function – the survival probabilities of individuals over the timeline

Return type DataFrame

print_summary (decimals=2, **kwargs)
Print summary statistics describing the fit, the coefficients, and the error bounds.

Parameters

• decimals (int, optional (default=2)) – specify the number of decimal places to show

• alpha (float or iterable) – specify confidence intervals to show

• kwargs – print additional metadata in the output (useful to provide model names, dataset names, etc.) when comparing multiple outputs.

score_
The concordance score (also known as the c-index) of the fit. The c-index is a generalization of the ROC AUC to survival data, including censorships.

For this purpose, the score_ is a measure of the predictive accuracy of the fitted model onto the training dataset.

summary
Summary statistics describing the fit.

Returns df – Contains columns coef, np.exp(coef), se(coef), z, p, lower, upper

Return type DataFrame
This class implements an Piecewise Exponential model for univariate data. The model has parameterized hazard rate:

$$h(t \mid x) = \begin{cases} \frac{1}{\lambda_0(x)}, & \text{if } t \leq \tau_0 \\ \frac{1}{\lambda_1(x)}, & \text{if } \tau_0 < t \leq \tau_1 \\ \frac{1}{\lambda_2(x)}, & \text{if } \tau_1 < t \leq \tau_2 \\ \vdots \end{cases}$$

and $\lambda_i(x) = \exp(\mathbf{beta}_i \cdot x^T)$, $\mathbf{beta}_i = (\beta_{i,1}, \beta_{i,2}, \ldots)$. That is, each period has a hazard rate, $\lambda_i(x)$ is the exponential of a linear model. The parameters of each linear model are unique to that period - different periods have different parameters (later we will generalize this).

Why do I want a model like this? Well, it offers lots of flexibility (at the cost of efficiency though), but importantly I can see:

1. Influence of variables over time.
2. Looking at important variables at specific “drops” (or regime changes). For example, what variables cause the large drop at the start? What variables prevent death at the second billing?
3. Predictive power: since we model the hazard more accurately (we hope) than a simpler parametric form, we have better estimates of a subjects survival curve.

After calling the .fit method, you have access to properties like: 

- **params_** A summary of the fit is available with the method print_summary()

**Parameters**

- **breakpoints (list)** – a list of times when a new exponential model is constructed.
- **alpha (float, optional (default=0.05))** – the level in the confidence intervals.
- **fit_intercept (boolean, optional (default=True))** – Allow lifelines to add an intercept column of 1s to df, and ancillary_dff if applicable.
- **penalizer (float, optional (default=0.0))** – the penalizer coefficient to the size of the coefficients. See $L_1$ ratio. Must be equal to or greater than 0.

**params_**

The estimated coefficients

Type DataFrame

**confidence_intervals_**

The lower and upper confidence intervals for the coefficients

Type DataFrame
durations
  The event_observed variable provided
    Type Series

event_observed
  The event_observed variable provided
    Type Series

weights
  The event_observed variable provided
    Type Series

variance_matrix
  The variance matrix of the coefficients
    Type numpy array

standard_errors
  the standard errors of the estimates
    Type Series

score
  the concordance index of the model.
    Type float

fit (df, duration_col=None, event_col=None, show_progress=False, timeline=None, weights_col=None, robust=False, initial_point=None)
Fit the accelerated failure time model to a dataset.

Parameters

• df (DataFrame) – a Pandas DataFrame with necessary columns duration_col and event_col (see below), covariates columns, and special columns (weights). duration_col refers to the lifetimes of the subjects. event_col refers to whether the ‘death’ events was observed: 1 if observed, 0 else (censored).

• duration_col (string) – the name of the column in DataFrame that contains the subjects’ lifetimes.

• event_col (string, optional) – the name of the column in DataFrame that contains the subjects’ death observation. If left as None, assume all individuals are uncensored.

• show_progress (boolean, optional (default=False)) – since the fitter is iterative, show convergence diagnostics. Useful if convergence is failing.

• timeline (array, optional) – Specify a timeline that will be used for plotting and prediction

• weights_col (string) – the column in df that specifies weights per observation.

• robust (boolean, optional (default=False)) – Compute the robust errors using the Huber sandwich estimator.

• initial_point (d,) numpy array, optional – initialize the starting point of the iterative algorithm. Default is the zero vector.

Returns self with additional new properties: print_summary, params_, confidence_intervals and more

Return type self
Examples

```python
>>> N, d = 80000, 2
>>> # some numbers take from http://statwonk.com/parametric-survival.html
>>> breakpoints = (1, 31, 34, 62, 65)
>>> betas = np.array([
... [1.0, -0.2, np.log(15)],
... [5.0, -0.4, np.log(333)],
... [9.0, -0.6, np.log(18)],
... [5.0, -0.8, np.log(500)],
... [2.0, -1.0, np.log(20)],
... [1.0, -1.2, np.log(500)],
... ]
>>> X = 0.1 * np.random.exponential(size=(N, d))
>>> X = np.c_[X, np.ones(N)]
>>> T = np.empty(N)
>>> for i in range(N):
...    lambdas = np.exp(-betas.dot(X[i, :]))
...    T[i] = piecewise_exponential_survival_data(1, breakpoints, lambdas)[0]
...    T_censor = np.minimum(T.mean() * np.random.exponential(size=N), 110)
... # 110 is the end of observation, eg. current time.
>>> df = pd.DataFrame(X[:, :-1], columns=["var1", "var2"])
>>> df["T"] = np.round(np.maximum(np.minimum(T, T_censor), 0.1), 1)
>>> df["E"] = T <= T_censor
>>> pew = PiecewiseExponentialRegressionFitter(breakpoints=breakpoints,
... penalizer=0.0001).fit(df, "T", "E")
>>> pew.print_summary()
>>> pew.plot()
```

`fit_right_censoring` (df, duration_col=None, event_col=None, show_progress=False, timeline=None, weights_col=None, robust=False, initial_point=None)

Fit the accelerated failure time model to a dataset.

Parameters

- **df** (DataFrame) – a Pandas DataFrame with necessary columns `duration_col` and `event_col` (see below), covariates columns, and special columns (weights). `duration_col` refers to the lifetimes of the subjects. `event_col` refers to whether the ‘death’ events was observed: 1 if observed, 0 else (censored).
- **duration_col** (string) – the name of the column in DataFrame that contains the subjects’ lifetimes.
- **event_col** (string, optional) – the name of the column in DataFrame that contains the subjects’ death observation. If left as None, assume all individuals are uncensored.
- **show_progress** (boolean, optional (default=False)) – since the fitter is iterative, show convergence diagnostics. Useful if convergence is failing.
- **timeline** (array, optional) – Specify a timeline that will be used for plotting and prediction
- **weights_col** (string) – the column in df that specifies weights per observation.
- **robust** (boolean, optional (default=False)) – Compute the robust errors using the Huber sandwich estimator.
- **initial_point** *(shape, numpy array, optional)* – initialize the starting point of the iterative algorithm. Default is the zero vector.

**Returns**

self with additional new properties: `print_summary`, `params_`, `confidence_intervals` and more

**Return type**

self

**Examples**

```python
code_snippet
```

```bash
mean_survival_time_
median_survival_time_
plot(columns=None, parameters=None, **errorbar_kwargs)
```

Produces a visual representation of the coefficients, including their standard errors and magnitudes.

**Parameters**

- **columns** *(list, optional)* – specify a subset of the columns (variables from the training data) to plot

- **parameter** *(list, optional)* – specify a subset of the parameters to plot

- **errorbar_kwargs** – pass in additional plotting commands to matplotlib errorbar command

**Returns**

ax – the matplotlib axis that be edited.
Return type matplotlib axis

predict_cumulative_hazard \((X, \text{times}=\text{None})\)
Return the cumulative hazard rate of subjects in \(X\) at time points.

Parameters

- \(X\) (numpy array or DataFrame) – a (n,d) covariate numpy array or DataFrame. If a DataFrame, columns can be in any order. If a numpy array, columns must be in the same order as the training data.
- \(\text{times}\) (iterable, optional) – an iterable of increasing times to predict the cumulative hazard at. Default is the set of all durations (observed and unobserved). Uses a linear interpolation if points in time are not in the index.
- \(\text{ancillary}\_X\) (numpy array or DataFrame, optional) – a (n,d) covariate numpy array or DataFrame. If a DataFrame, columns can be in any order. If a numpy array, columns must be in the same order as the training data.

Returns cumulative_hazard_ – the cumulative hazard of individuals over the timeline

Return type DataFrame

predict_expectation \((X)\)
Compute the expected lifetime, \(E[T]\), using covariates \(X\). This algorithm to compute the expectation is to use the fact that \(E[T] = \int_0^{\infty} P(T > t) dt = \int_0^{\infty} S(t) dt\). To compute the integral, we use the trapezoidal rule to approximate the integral.

Caution: However, if the survival function doesn’t converge to 0, the the expectation is really infinity and the returned values are meaningless/too large. In that case, using predict_median or predict_percentile would be better.

Parameters \(X\) (numpy array or DataFrame) – a (n,d) covariate numpy array or DataFrame. If a DataFrame, columns can be in any order. If a numpy array, columns must be in the same order as the training data.

Returns expectations

Return type DataFrame

Notes

If \(X\) is a DataFrame, the order of the columns do not matter. But if \(X\) is an array, then the column ordering is assumed to be the same as the training dataset.

See also:
predict_median(), predict_percentile()

predict_median \((X)\)
Predict the median lifetimes for the individuals. If the survival curve of an individual does not cross 0.5, then the result is infinity.

Parameters \(X\) (numpy array or DataFrame) – a (n,d) covariate numpy array or DataFrame. If a DataFrame, columns can be in any order. If a numpy array, columns must be in the same order as the training data.

Returns percentiles – the median lifetimes for the individuals. If the survival curve of an individual does not cross 0.5, then the result is infinity.
Return type: DataFrame

See also: predict_percentile(), predict_expectation()

predict_percentile(X, p=0.5)

Returns the median lifetimes for the individuals, by default. If the survival curve of an individual does not cross 0.5 in the timeline (set in fit), then the result is infinity. http://stats.stackexchange.com/questions/102986/percentile-loss-functions

Parameters

- **X** (numpy array or DataFrame) – a (n,d) covariate numpy array or DataFrame. If a DataFrame, columns can be in any order. If a numpy array, columns must be in the same order as the training data.
- **p** (float, optional (default=0.5)) – the percentile, must be between 0 and 1.

Returns: percentiles

Return type: DataFrame

See also: predict_median()

predict_survival_function(X, times=None, ancillary_X=None)

Predict the survival function for individuals, given their covariates. This assumes that the individual just entered the study (that is, we do not condition on how long they have already lived for.)

Parameters

- **X** (numpy array or DataFrame) – a (n,d) covariate numpy array or DataFrame. If a DataFrame, columns can be in any order. If a numpy array, columns must be in the same order as the training data.
- **ancillary_X** (numpy array or DataFrame, optional) – a (n,d) covariate numpy array or DataFrame. If a DataFrame, columns can be in any order. If a numpy array, columns must be in the same order as the training data.
- **times** (iterable, optional) – an iterable of increasing times to predict the cumulative hazard at. Default is the set of all durations (observed and unobserved). Uses a linear interpolation if points in time are not in the index.

Returns: survival_function – the survival probabilities of individuals over the timeline

Return type: DataFrame

print_summary(decimals=2, **kwargs)

Print summary statistics describing the fit, the coefficients, and the error bounds.

Parameters

- **decimals** (int, optional (default=2)) – specify the number of decimal places to show
- **alpha** (float or iterable) – specify confidence intervals to show
- **kwargs** – print additional metadata in the output (useful to provide model names, dataset names, etc.) when comparing multiple outputs.

score_

The concordance score (also known as the c-index) of the fit. The c-index is a generalization of the ROC AUC to survival data, including censorships.
For this purpose, the `score_` is a measure of the predictive accuracy of the fitted model onto the training dataset.

### summary

Summary statistics describing the fit.

- **Returns** df – Contains columns `coef`, `np.exp(coef)`, `se(coef)`, `z`, `p`, `lower`, `upper`
- **Return type** DataFrame

## 1.10.2 lifelines.utils

### lifelines.utils.qth_survival_times

Find the times when one or more survival functions reach the qth percentile.

#### Parameters

- `q` *(float or array)* – A float between 0 and 1 that represents the time when the survival function hits the qth percentile.
- `survival_functions` *(a (n,d) DataFrame or numpy array.)* – If DataFrame, will return index values (actual times) If numpy array, will return indices.
- `cdf` *(boolean, optional)* – When doing left-censored data, `cdf=True` is used.

#### Returns

- If `d==1`, returns a float, `np.inf` if infinity. If `d > 1`, an DataFrame containing the first times the value was crossed.
- **Return type** float, or DataFrame

See also:

- `qth_survival_time()`, `median_survival_times()`

### lifelines.utils.qth_survival_time

Returns the time when a single survival function reaches the qth percentile.

#### Parameters

- `q` *(float)* – A float between 0 and 1 that represents the time when the survival function hit’s the qth percentile.
- `survival_function` *(Series or single-column DataFrame.)*
- `cdf` *(boolean, optional)* – When doing left-censored data, `cdf=True` is used.

#### Returns

- **Return type** float

See also:

- `qth_survival_times()`, `median_survival_times()`

### lifelines.utils.median_survival_times

#### Parameters

- `density_or_survival_function` *(Density or survival function, left_censorship=False)*

### lifelines.utils.survival_table_from_events

#### Parameters
• **death_times** *(n,)* array – represent the event times

• **event_observed** *(n,)* array – 1 if observed event, 0 is censored event.

• **birth_times** *(a (n,)* array, optional)* – representing when the subject was first observed. A subject’s death event is then at [birth times + duration observed]. If None (default), birth_times are set to be the first observation or 0, which ever is smaller.

• **columns** *(iterable, optional)* – a 3-length array to call the, in order, removed individuals, observed deaths and censorships.

• **weights** *(n,1)* array, optional – Optional argument to use weights for individuals. Assumes weights of 1 if not provided.

• **collapse** *(boolean, optional (default=False))* – If True, collapses survival table into lifetable to show events in interval bins

• **intervals** *(iterable, optional)* – Default None, otherwise a list/(n,1) array of interval edge measures. If left as None while collapse=True, then Freedman-Diaconis rule for histogram bins will be used to determine intervals.

**Returns** Pandas DataFrame with index as the unique times or intervals in event_times. The columns named ‘removed’ refers to the number of individuals who were removed from the population by the end of the period. The column ‘observed’ refers to the number of removed individuals who were observed to have died (i.e. not censored.) The column ‘censored’ is defined as ‘removed’ - ‘observed’ (the number of individuals who left the population due to event_observed)

**Return type** DataFrame

**Example**

```
>>> #Uncollapsed output
>>> event_at removed observed censored entrance at_risk
>>> 0 0 0 0 0 0 11 11
>>> 6 1 1 0 0 0 11
>>> 7 2 2 0 0 0 10
>>> 9 3 3 0 0 0 8
>>> 13 3 3 0 0 0 5
>>> 15 2 2 0 0 2

>>> #Collapsed output
>>> event_at removed observed censored at_risk
>>> sum sum sum max

>>> 0, 2] 34 33 1 11 312
>>> 2, 4] 84 42 42 278
>>> 4, 6] 64 17 47 194
>>> 6, 8] 63 16 47 130
>>> 8, 10] 35 12 23 67
>>> 10, 12] 24 5 19 32
```

**See also:**

*group_survival_table_from_events()*

lifelines.utils.*group_survival_table_from_events*(groups, durations, event_observed, birth_times=0, limit=-1)

Joins multiple event series together into DataFrames. A generalization of *survival_table_from_events* to data with groups. Previously called *group_event_series* pre 0.2.3.
Parameters

- **groups** \((a \, (n,) \, array)\) – individuals’ group ids.
- **durations** \((a \, (n,) \, array)\) – durations of each individual
- **event_observed** \((a \, (n,) \, array)\) – event observations, 1 if observed, 0 else.
- **birth_times** \((a \, (n,) \, array)\) – when the subject was first observed. A subject’s death event is then at \([\text{birth times} + \text{duration observed}]\). Normally set to all zeros, but can be positive or negative.

Returns

- **unique_groups** \((\text{np.array})\) – array of all the unique groups present
- **removed** \((\text{DataFrame})\) – DataFrame of removal count data at event_times for each group, column names are ‘removed:<group name>’
- **observed** \((\text{DataFrame})\) – DataFrame of observed count data at event_times for each group, column names are ‘observed:<group name>’
- **censored** \((\text{DataFrame})\) – DataFrame of censored count data at event_times for each group, column names are ‘censored:<group name>’

Example

```python
doctest
>>> #input
>>> group_survival_table_from_events(waltonG, waltonT, np.ones_like(waltonT))
->#data available in test_suite.py
>>> #output
>>> {  
>>>     array(['control', 'miR-137'], dtype=object),
>>>     removed:control  removed:miR-137
>>>     event_at
>>>     6 0 1
>>>     7 2 0
>>>     9 0 3
>>>     13 0 3
>>>     15 0 2
>>> ,
>>>     observed:control  observed:miR-137
>>>     event_at
>>>     6 0 1
>>>     7 2 0
>>>     9 0 3
>>>     13 0 3
>>>     15 0 2
>>> ,
>>>     censored:control  censored:miR-137
>>>     event_at
>>>     6 0 0
>>>     7 0 0
>>>     9 0 0
>>> ,
>>> }
```

See also:
survival_table_from_events() 

lifelines.utils.survival_events_from_table(survival_table, observed_deaths_col='observed', censored_col='censored')

This is the inverse of the function survival_table_from_events.

Parameters

- **survival_table** *(DataFrame)* - a pandas DataFrame with index as the durations and columns “observed” and “censored”, referring to the number of individuals that died and were censored at time t.

- **observed_deaths_col** *(str, optional (default: “observed”))* - the column in the survival table that represents the number of subjects that were observed to die at a specific time

- **censored_col** *(str, optional (default: “censored”))* - the column in the survival table that represents the number of subjects that were censored at a specific time

Returns

- **T** *(array)* - durations of observation – one element for observed time

- **E** *(array)* - event observations – 1 if observed, 0 else.

- **W** *(array)* - weights - integer weights to “condense” the data

Example

```python
>>> # Ex: The survival table, as a pandas DataFrame:
>>> observed  censored
>>> index
>>> 1         1        0
>>> 2         0        1
>>> 3         1        0
>>> 4         1        1
>>> 5         0        1

>>> # would return
>>> T = np.array([ 1., 2., 3., 4., 4., 5.]),
>>> E = np.array([ 1., 0., 1., 1., 0., 0.])
>>> W = np.array([ 1, 1, 1, 1, 1, 1])
```

lifelines.utils.datetimes_to_durations(start_times, end_times, fill_date=datetime.datetime(2019, 5, 16, 13, 0, 27, 32153), freq='D', dayfirst=False, na_values=None)

This is a very flexible function for transforming arrays of start_times and end_times to the proper format for lifelines: duration and event observation arrays.

Parameters

- **start_times** *(an array, Series or DataFrame)* - iterable representing start times. These can be strings, or datetime objects.

- **end_times** *(an array, Series or DataFrame)* - iterable representing end times. These can be strings, or datetimes. These values can be None, or an empty string, which corresponds to censorship.
• **fill_date** *(datetime, optional (default=datetime.Today()))* – the date to use if end_times is a None or empty string. This corresponds to last date of observation. Anything after this date is also censored.

• **freq** *(string, optional (default='D'))* – the units of time to use. See Pandas ‘freq’. Default ‘D’ for days.

• **dayfirst** *(boolean, optional (default=False))* – convert assuming European-style dates, i.e. day/month/year.

• **na_values** *(list, optional)* – list of values to recognize as NA/NaN. Ex: ["", 'NaT']

Returns

• T *(numpy array)* – array of floats representing the durations with time units given by freq.

• C *(numpy array)* – boolean array of event observations: 1 if death observed, 0 else.

Examples

```python
>>> from lifelines.utils import datetimes_to_durations

>>> start_dates = ['2015-01-01', '2015-04-01', '2014-04-05']
>>> end_dates = ['2016-02-02', None, '2014-05-06']

>>> T, E = datetimes_to_durations(start_dates, end_dates, freq="D")
>>> T
# array([ 397., 1414., 31.])
>>> E
# array([ True, False, True])
```

**lifelines.utils.concordance_index** *(event_times, predicted_scores, event_observed=None)*

Calculates the concordance index (C-index) between two series of event times. The first is the real survival times from the experimental data, and the other is the predicted survival times from a model of some kind.

The c-index is the average of how often a model says X is greater than Y when, in the observed data, X is indeed greater than Y. The c-index also handles how to handle censored values (obviously, if Y is censored, it's hard to know if X is truly greater than Y).

The concordance index is a value between 0 and 1 where:

• 0.5 is the expected result from random predictions,

• 1.0 is perfect concordance and,

• 0.0 is perfect anti-concordance (multiply predictions with -1 to get 1.0)

Parameters

• **event_times** *(iterable)* – a length-n iterable of observed survival times.

• **predicted_scores** *(iterable)* – a length-n iterable of predicted scores - these could be survival times, or hazards, etc. See https://stats.stackexchange.com/questions/352183/use-median-survival-time-to-calculate-cph-c-statistic/352435#352435

• **event_observed** *(iterable, optional)* – a length-n iterable censorship flags, 1 if observed, 0 if not. Default None assumes all observed.

Returns **c-index** – a value between 0 and 1.

Return type float

1.10. Reference library for *lifelines* 227
References


Examples

```python
cph = CoxPHFitter().fit(df, 'T', 'E')
concordance_index(df['T'], -cph.predict_partial_hazard(df), df['E'])
```

```
lifelines.utils.k_fold_cross_validation(fitters, df, duration_col, event_col=None, k=5, evaluation_measure=concordance_index, predictor='predict_expectation', predictor_kwargs={}, fitter_kwargs={})
```

Perform cross validation on a dataset. If multiple models are provided, all models will train on each of the k subsets.

**Parameters**

- **fitters (model)** – one or several objects which possess a method: `fit(self, data, duration_col, event_col)` Note that the last two arguments will be given as keyword arguments, and that event_col is optional. The objects must also have the “predictor” method defined below.

- **df (DataFrame)** – a Pandas DataFrame with necessary columns duration_col and (optional) event_col, plus other covariates. duration_col refers to the lifetimes of the subjects. event_col refers to whether the ‘death’ events was observed: 1 if observed, 0 else (censored).

- **duration_col (n,) array** – the column in DataFrame that contains the subjects lifetimes.

- **event_col (n,) array** – the column in DataFrame that contains the subject’s death observation. If left as None, assumes all individuals are non-censored.

- **k (int)** – the number of folds to perform. n/k data will be withheld for testing on.

- **evaluation_measure (function)** – a function that accepts either (event_times, predicted_event_times), or (event_times, predicted_event_times, event_observed) and returns something (could be anything). Default: statistics.concordance_index: (C-index) between two series of event times

- **predictor (string)** – a string that matches a prediction method on the fitter instances. For example, predict_expectation or predict_percentile. Default is “predict_expectation” The interface for the method is: `predict(self, data, **optional_kwargs)`

- **fitter_kwargs** – keyword args to pass into fitter.fit method

- **predictor_kwargs** – keyword args to pass into predictor-method.

**Returns**

- **results** – (k,1) list of scores for each fold. The scores can be anything.

**Return type**

```
lifelines.utils.to_long_format(df, duration_col)
```

This function converts a survival analysis DataFrame to a lifelines “long” format. The lifelines “long” format is used in a common next function, `add_covariate_to_timeline`.

**Parameters**
• **df** (*DataFrame*) – a DataFrame in the standard survival analysis form (one for per observation, with covariates, duration and event flag)

• **duration_col** (*string*) – string representing the column in df that represents the durations of each subject.

Returns **long_form_df** – A DataFrame with new columns. This can be fed into `add_covariate_to_timeline`

Return type DataFrame

See also:

`to_episodic_format()`, `add_covariate_to_timeline()`

```python
from lifelines.datasets import load_rossi
from lifelines.utils import to_episodic_format

rossi = load_rossi()
long_rossi = to_episodic_format(rossi, 'week', 'arrest', time_gaps=2.0)

ctv = CoxTimeVaryingFitter()
ctv.fit(long_rossi, id_col='id', event_col='arrest', show_progress=True)
ctv.print_summary()
```

Example

```python
>>> from lifelines.datasets import load_rossi
>>> from lifelines.utils import to_episodic_format
>>> rossi = load_rossi()
>>> long_rossi = to_episodic_format(rossi, 'week', 'arrest', time_gaps=2.0)

>>> from lifelines import CoxTimeVaryingFitter
>>> ctv = CoxTimeVaryingFitter()
>>> # age variable violates proportional hazard
>>> long_rossi['time * age'] = long_rossi['stop'] * long_rossi['age']
>>> ctv.fit(long_rossi, id_col='id', event_col='arrest', show_progress=True)
>>> ctv.print_summary()
```

See also:

`add_covariate_to_timeline()`, `to_long_format()`
lifelines.utils.add_covariate_to_timeline\( (\text{long\_form\_df}, \ cv, \ \text{id\_col}, \ \text{duration\_col}, \ \text{event\_col}, \ \text{start\_col}='start', \ \
\text{stop\_col}='stop', \ \text{add\_enum}=\text{False}, \ \text{overwrite}=\text{True}, \ \text{cumulative\_sum}=\text{False}, \ \text{cumulative\_sum\_prefix}='\text{cumsum}_', \ \text{delay}=0) \)\n
This is a util function to help create a long form table tracking subjects’ covariate changes over time. It is meant to be used iteratively as one adds more and more covariates to track over time. Before using this function, it is recommended to view the documentation at https://lifelines.readthedocs.io/en/latest/Survival\%20Regression.html#dataset-creation-for-time-varying-regression.

**Parameters**

- **long\_form\_df** (*DataFrame*) – a DataFrame that has the initial or intermediate “long” form of time-varying observations. Must contain columns \text{id\_col}, ‘start’, ‘stop’, and \text{event\_col}. See function \text{to\_long\_format} to transform data into long form.

- **cv** (*DataFrame*) – a DataFrame that contains (possibly more than) one covariate to track over time. Must contain columns \text{id\_col} and \text{duration\_col}. \text{duration\_col} represents time since the start of the subject’s life.

- **id\_col** (*string*) – the column in \text{long\_form\_df} and \text{cv} representing a unique identifier for subjects.

- **duration\_col** (*string*) – the column in \text{cv} that represents the time-since-birth the observation occurred at.

- **event\_col** (*string*) – the column in \text{df} that represents if the event-of-interest occurred

- **add\_enum** (*boolean, optional*) – a Boolean flag to denote whether to add a column enumerating rows per subject. Useful to specify a specific observation, ex: df[df[‘enum’] == 1] will grab the first observations per subject.

- **overwrite** (*boolean, optional*) – if True, covariate values in \text{long\_form\_df} will be overwritten by covariate values in \text{cv} if the column exists in both \text{cv} and \text{long\_form\_df} and the timestamps are identical. If False, the default behavior will be to sum the values together.

- **cumulative\_sum** (*boolean, optional*) – sum over time the new covariates. Makes sense if the covariates are new additions, and not state changes (ex: administering more drugs vs taking a temperature.)

- **cumulative\_sum\_prefix** (*string, optional*) – a prefix to add to calculated cumulative sum columns

- **delay** (*int, optional*) – add a delay to covariates (useful for checking for reverse causality in analysis)

**Returns** \text{long\_form\_df} – A DataFrame with updated rows to reflect the novel times slices (if any) being added from \text{cv}, and novel (or updated) columns of new covariates from \text{cv}

**Return type** DataFrame

See also:

- \text{to\_episodic\_format()}, \text{to\_long\_format()}, \text{covariates\_from\_event\_matrix()}

lifelines.utils.covariates_from_event_matrix\( (\text{df}, \ \text{id\_col}) \)\n
This is a helper function to handle binary event datastreams in a specific format and convert it to a format that \text{add\_covariate\_to\_timeline} will accept. For example, suppose you have a dataset that looks like:

<table>
<thead>
<tr>
<th>id</th>
<th>promotion</th>
<th>movement</th>
<th>event</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>1.0</td>
<td>NaN</td>
</tr>
</tbody>
</table>

(continues on next page)
where the values (aside from the id column) represent when an event occurred for a specific user, relative to the subject's birth/entry. This is a common way format to pull data from a SQL table. We call this a duration matrix, and we want to convert this DataFrame to a format that can be included in a long form DataFrame (see add_covariate_to_timeline for more details on this).

The duration matrix should have 1 row per subject (but not necessarily all subjects).

Parameters

- **df** *(DataFrame)* – the DataFrame we want to transform
- **id_col** *(string)* – the column in long_form_df and cv representing a unique identifier for subjects.

Example

```python
>>> cv = covariates_from_event_matrix(duration_df, 'id')
>>> long_form_df = add_covariate_to_timeline(long_form_df, cv, 'id', 'duration', ‘e’, cumulative_sum=True)
```

## 1.10.3 lifelines.statistics

### class lifelines.statistics.StatisticalResult

```python
class lifelines.statistics.StatisticalResult(p_value, test_statistic, name=None, **kwargs)
```

Bases: object

This class holds the result of statistical tests with a nice printer wrapper to display the results.

**Note:** This class’ API changed in version 0.16.0.

Parameters

- **p_value** *(iterable or float)* – the p-values of a statistical test(s)
- **test_statistic** *(iterable or float)* – the test statistics of a statistical test(s). Must be the same size as p-values if iterable.
- **name** *(iterable or string)* – if this class holds multiple results (ex: from a pairwise comparison), this can hold the names. Must be the same size as p-values if iterable.
- **kwargs** – additional information to attach to the object and display in print_summary().

### print_summary

```python
print_summary(decimals=2, **kwargs)
```

Print summary statistics describing the fit, the coefficients, and the error bounds.

Parameters

- **decimals** *(int, optional (default=2))* – specify the number of decimal places to show
- **kwargs** – print additional meta data in the output (useful to provide model names, dataset names, etc.) when comparing multiple outputs.
lifelines Documentation, Release 0.21.2

summary
returns: a DataFrame containing the test statistics and the p-value :rtype: DataFrame

lifelines.statistics.logrank_test(durations_A, durations_B, event_observed_A=None, event_observed_B=None, t_0=-1, **kwargs)

Measures and reports on whether two intensity processes are different. That is, given two event series, determines whether the data generating processes are statistically different. The test-statistic is chi-squared under the null hypothesis. Let \( h_i(t) \) be the hazard ratio of group \( i \) at time \( t \), then:

\[
H_0: h_1(t) = h_2(t) \\
H_A: h_1(t) = ch_2(t), \ c \neq 1
\]

This implicitly uses the log-rank weights.

Note: The logrank test has maximum power when the assumption of proportional hazards is true. As a consequence, if the survival curves cross, the logrank test will give an inaccurate assessment of differences.

Parameters
- **durations_A** (iterable) – a (n,) list-like of event durations (birth to death,\ldots) for the first population.
- **durations_B** (iterable) – a (n,) list-like of event durations (birth to death,\ldots) for the second population.
- **event_observed_A** (iterable, optional) – a (n,) list-like of censorship flags, (1 if observed, 0 if not), for the first population. Default assumes all observed.
- **event_observed_B** (iterable, optional) – a (n,) list-like of censorship flags, (1 if observed, 0 if not), for the second population. Default assumes all observed.
- **t_0** (float, optional (default=-1)) – the final time period under observation, -1 for all time.
- **kwargs** – add keywords and meta-data to the experiment summary

Returns a StatisticalResult object with properties p_value, summary, test_statistic, print_summary

Return type StatisticalResult

Examples

```python
>>> T1 = [1, 4, 10, 12, 12, 3, 5.4]
>>> E1 = [1, 0, 1, 0, 1, 1, 1]
>>> T2 = [4, 5, 7, 11, 14, 20, 8, 8]
>>> E2 = [1, 1, 1, 1, 1, 1, 1, 1]

>>> from lifelines.statistics import logrank_test
>>> results = logrank_test(T1, T2, event_observed_A=E1, event_observed_B=E2)
>>> results.print_summary()
>>> print(results.p_value)  # 0.7676
>>> print(results.test_statistic)  # 0.0872
```
Notes

This is a special case of the function `multivariate_logrank_test`, which is used internally. See Survival and Event Analysis, page 108.

See also:

- `multivariate_logrank_test()`, `pairwise_logrank_test()`, `survival_difference_at_fixed_point_in_time_test()`

This test is a generalization of the logrank_test: it can deal with n>2 populations (and should be equal when n=2):

\[
H_0 : h_1(t) = h_2(t) = h_3(t) = ... = h_n(t)
\]

\[
H_A : \text{there exist at least one group that differs from the other}
\]

Parameters

- `event_durations` *(iterable)* – a (n,) list-like representing the (possibly partial) durations of all individuals
- `groups` *(iterable)* – a (n,) list-like of unique group labels for each individual.
- `event_observed` *(iterable, optional)* – a (n,) list-like of event_observed events: 1 if observed death, 0 if censored. Defaults to all observed.
- `t_0` *(float, optional (default=-1))* – the period under observation, -1 for all time.
- `kwargs` – add keywords and meta-data to the experiment summary.

Returns a `StatisticalResult` object with properties `p_value`, `summary`, `test_statistic`, `print_summary`

Return type `StatisticalResult`

Examples

```python
>>> df = pd.DataFrame({
...     'durations': [5, 3, 9, 8, 7, 4, 4, 3, 2, 5, 6, 7],
...     'events': [1, 1, 1, 1, 1, 0, 0, 1, 1, 1, 1, 0],
...     'groups': [0, 0, 0, 0, 1, 1, 1, 1, 1, 2, 2, 2]
... })
>>> result = multivariate_logrank_test(df['durations'], df['groups'], df['events'])
>>> result.test_statistic
>>> result.p_value
>>> result.print_summary()
```

```
>>> # numpy example
>>> G = [0, 0, 0, 0, 1, 1, 1, 1, 2, 2, 2]
>>> T = [5, 3, 9, 8, 7, 4, 4, 3, 2, 5, 6, 7]
>>> E = [1, 1, 1, 1, 1, 0, 0, 1, 1, 1, 1, 0]
>>> result = multivariate_logrank_test(T, G, E)
>>> result.test_statistic
```

1.10. Reference library for lifelines 233
See also:

`pairwise_logrank_test()`, `logrank_test()`

**lifelines.statistics.pairwise_logrank_test** *(event_durations, groups, event_observed=None, t_0=-1, **kwargs)*

Perform the logrank test pairwise for all $n \geq 2$ unique groups.

**Parameters**

- `event_durations` *(iterable)* – a (n,) list-like representing the (possibly partial) durations of all individuals.
- `groups` *(iterable)* – a (n,) list-like of unique group labels for each individual.
- `event_observed` *(iterable, optional)* – a (n,) list-like of event_observed events: 1 if observed death, 0 if censored. Defaults to all observed.
- `t_0` *(float, optional (default=-1))* – the period under observation, -1 for all time.
- `kwargs` – add keywords and meta-data to the experiment summary.

**Returns** a `StatisticalResult` object that contains all the pairwise comparisons (try `StatisticalResult.summary` or `StatisticalResult.print_summary`)

**Return type** `StatisticalResult`

See also: `multivariate_logrank_test()`, `logrank_test()`

**lifelines.statistics.survival_difference_at_fixed_point_in_time_test** *(point_in_time, durations_A, durations_B, event_observed_A=None, event_observed_B=None, **kwargs)*

Often analysts want to compare the survival-ness of groups at specific times, rather than comparing the entire survival curves against each other. For example, analysts may be interested in 5-year survival. Statistically comparing the naive Kaplan-Meier points at a specific time actually has reduced power (see [1]). By transforming the Kaplan-Meier curve, we can recover more power. This function uses the log(-log) transformation.

**Parameters**

- `point_in_time` *(float,)* – the point in time to analyze the survival curves at.
- `durations_A` *(iterable)* – a (n,) list-like of event durations (birth to death,...) for the first population.
- `durations_B` *(iterable)* – a (n,) list-like of event durations (birth to death,...) for the second population.
- `event_observed_A` *(iterable, optional)* – a (n,) list-like of censorship flags, (1 if observed, 0 if not), for the first population. Default assumes all observed.
- `event_observed_B` *(iterable, optional)* – a (n,) list-like of censorship flags, (1 if observed, 0 if not), for the second population. Default assumes all observed.
- `kwargs` – add keywords and meta-data to the experiment summary

**Returns** a `StatisticalResult` object with properties `p_value`, `summary`, `test_statistic`, `print_summary`
Return type  

\textit{StatisticalResult}

Examples

```python
>>> T1 = [1, 4, 10, 12, 12, 3, 5.4]
>>> E1 = [1, 0, 1, 0, 1, 1, 1]

>>> T2 = [4, 5, 7, 11, 14, 20, 8, 8]
>>> E2 = [1, 1, 1, 1, 1, 1, 1, 1]

>>> from lifelines.statistics import survival_difference_at_fixed_point_in_time_test
>>> results = survival_difference_at_fixed_point_in_time_test(12, T1, T2, event_observed_A=E1, event_observed_B=E2)

>>> results.print_summary()
>>> print(results.p_value)  # 0.893
>>> print(results.test_statistic)  # 0.017
```

Notes

Other transformations are possible, but Klein et al. [1] showed that the log(-log(c)) transform has the most desirable statistical properties.

References


\texttt{lifelines.statistics.proportional_hazard_test} \(\texttt{(fitted\_cox\_model, training\_df, time\_transform='rank', precomputed\_residuals=None, **kwargs)}\)

Test whether any variable in a Cox model breaks the proportional hazard assumption.

Parameters

- \texttt{fitted\_cox\_model} (\texttt{CoxPHFitter}) – the fitted Cox model, fitted with \texttt{training\_df}, you wish to test. Currently only the CoxPHFitter is supported, but later CoxTimeVaryingFitter, too.

- \texttt{training\_df} (\texttt{DataFrame}) – the DataFrame used in the call to the Cox model’s \texttt{fit}.

- \texttt{time\_transform} (vectorized function, list, or string, optional (default=’rank’)) – \{'all’, ‘km’, ‘rank’, ‘identity’, ‘log’\} One of the strings above, a list of strings, or a function to transform the time (must accept (time, durations, weights) however). ‘all’ will present all the transforms.

- \texttt{precomputed\_residuals} (\texttt{DataFrame, optional}) – specify the residuals, if already computed.

- \texttt{kwags} – additional parameters to add to the StatisticalResult

Returns

Return type  \textit{StatisticalResult}
Notes

R uses the default \texttt{km}, we use \texttt{rank}, as this performs well versus other transforms. See http://eprints.lse.ac.uk/84988/1/06_ParkHendry2015-ReassessingSchoenfeldTests_Final.pdf

\begin{verbatim}
lifelines.statistics.power_under_cph(n_exp, n_con, p_exp, p_con, postulated_hazard_ratio, alpha=0.05)
\end{verbatim}

This computes the power of the hypothesis test that the two groups, experiment and control, have different hazards (that is, the relative hazard ratio is different from 1.)

Parameters

- \texttt{n_exp (integer)} – size of the experiment group.
- \texttt{n_con (integer)} – size of the control group.
- \texttt{p_exp (float)} – probability of failure in experimental group over period of study.
- \texttt{p_con (float)} – probability of failure in control group over period of study
- \texttt{postulated_hazard_ratio (float)}
- \texttt{the postulated hazard ratio}
- \texttt{alpha (float, optional (default=0.05))} – type I error rate

Returns power to detect the magnitude of the hazard ratio as small as that specified by postulated_hazard_ratio.

Return type float

Notes

Reference.

See also:

\begin{verbatim}
lifelines.statistics.sample_size_necessary_under_cph()
\end{verbatim}

\begin{verbatim}
lifelines.statistics.sample_size_necessary_under_cph(power, ratio_of_participants, p_exp, p_con, postulated_hazard_ratio, alpha=0.05)
\end{verbatim}

This computes the sample size for needed power to compare two groups under a Cox Proportional Hazard model.

Parameters

- \texttt{power (float)} – power to detect the magnitude of the hazard ratio as small as that specified by postulated_hazard_ratio.
- \texttt{ratio_of_participants (ratio of participants in experimental group over control group.)}
- \texttt{p_exp (float)} – probability of failure in experimental group over period of study.
- \texttt{p_con (float)} – probability of failure in control group over period of study
- \texttt{postulated_hazard_ratio (float)} – the postulated hazard ratio
- \texttt{alpha (float, optional (default=0.05))} – type I error rate

Returns

- \texttt{n_exp (integer)} – the samples sizes need for the experiment to achieve desired power
- \texttt{n_con (integer)} – the samples sizes need for the control group to achieve desired power
Examples

```python
>>> from lifelines.statistics import sample_size_necessary_under_cph
>>> desired_power = 0.8
>>> ratio_of_participants = 1.
>>> p_exp = 0.25
>>> p_con = 0.35
>>> postulated_hazard_ratio = 0.7
>>> n_exp, n_con = sample_size_necessary_under_cph(desired_power, ratio_of_participants, p_exp, p_con, postulated_hazard_ratio)
>>> # (421, 421)
```

References

https://cran.r-project.org/web/packages/powerSurvEpi/powerSurvEpi.pdf

See also:

`power_under_cph()`

1.10.4 lifelines.plotting

`lifelines.plotting.add_at_risk_counts(*fitters, **kwargs)`

Add counts showing how many individuals were at risk at each time point in survival/hazard plots.

Parameters

- **fitters** – One or several fitters, for example KaplanMeierFitter, NelsonAalenFitter, etc...

Returns

- **ax**

Return type

The axes which was used.

Examples

```python
>>> # First train some fitters and plot them
>>> fig = plt.figure()
>>> ax = plt.subplot(111)
>>> f1 = KaplanMeierFitter()
>>> f1.fit(data)
>>> f1.plot(ax=ax)
>>> f2 = KaplanMeierFitter()
>>> f2.fit(data)
>>> f2.plot(ax=ax)
>>> # There are equivalent
>>> add_at_risk_counts(f1, f2)
>>> add_at_risk_counts(f1, f2, ax=ax, fig=fig)
>>> # This overrides the labels
>>> add_at_risk_counts(f1, f2, labels=['fitter one', 'fitter two'])
```
>>> # This hides the labels
>>> add_at_risk_counts(f1, f2, labels=None)

lifelines.plotting.plot_lifetimes(durations, event_observed=None, entry=None,
left_truncated=False, sort_by_duration=True,
event_observed_color='#A60628',
event_censored_color='#348ABD', **kwargs)

Returns a lifetime plot, see examples: https://lifelines.readthedocs.io/en/latest/Survival%20Analysis%20intro.html#Censoring

Parameters

- durations (n,) numpy array or pd.Series – duration subject was observed for.
- event_observed (n,) numpy array or pd.Series – array of booleans: True if event observed, else False.
- entry (n,) numpy array or pd.Series – offsetting the births away from t=0. This could be from left-truncation, or delayed entry into study.
- left_truncated (boolean) – if entry is provided, and the data is left-truncated, this will display additional information in the plot to reflect this.
- sort_by_duration (boolean) – sort by the duration vector
- event_observed_color (str) – default: “#A60628”
- event_censored_color (str) – default: “#348ABD”

Returns

Return type ax

Examples

```python
>>> from lifelines.datasets import load_waltons
>>> from lifelines.plotting import plot_lifetimes
>>> T, E = load_waltons()['T'], load_waltons()['E']
>>> ax = plot_lifetimes(T.loc[:50], event_observed=E.loc[:50])
```

lifelines.plotting.qq_plot(model, **plot_kwargs)

Produces a quantile-quantile plot of the empirical CDF against the fitted parametric CDF. Large deviances away from the line y=x can invalidate a model (though we expect some natural deviance in the tails).

Parameters

- model (obj) – A fitted lifelines univariate parametric model, like WeibullFitter
- plot_kwargs – kwargs for the plot.

Returns ax

Return type axis object

Examples
>>> from lifelines import *
>>> from lifelines.plotting import qq_plot
>>> from lifelines.datasets import load_rossi

>>> df = load_rossi()

>>> wf = WeibullFitter().fit(df['week'], df['arrest'])

>>> qq_plot(wf)

lifelines.plotting.cdf_plot(model, timeline=None, **plot_kwargs)

1.10.5 lifelines.datasets

lifelines.datasets.load_canadian_senators(**kwargs)

A history of Canadian senators in office:

<table>
<thead>
<tr>
<th>Size: (933,10)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Example:</td>
</tr>
<tr>
<td>Name</td>
</tr>
<tr>
<td>Political Affiliation at Appointment</td>
</tr>
<tr>
<td>Province / Territory</td>
</tr>
<tr>
<td>Appointed on the advice of</td>
</tr>
<tr>
<td>Term (yyyy.mm.dd)</td>
</tr>
<tr>
<td>start_date</td>
</tr>
<tr>
<td>end_date</td>
</tr>
<tr>
<td>reason</td>
</tr>
<tr>
<td>diff_days</td>
</tr>
<tr>
<td>observed</td>
</tr>
</tbody>
</table>

lifelines.datasets.load_dd(**kwargs)

Classification of political regimes as democracy and dictatorship. Classification of democracies as parliamentary, semi-presidential (mixed) and presidential. Classification of dictatorships as military, civilian and royal. Coverage: 202 countries, from 1946 or year of independence to 2008:

<table>
<thead>
<tr>
<th>Size: (1808, 12)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Example:</td>
</tr>
<tr>
<td>ctryname</td>
</tr>
<tr>
<td>cowcode2</td>
</tr>
<tr>
<td>politycode</td>
</tr>
<tr>
<td>un_region_name</td>
</tr>
<tr>
<td>un_continent_name</td>
</tr>
<tr>
<td>ehead</td>
</tr>
<tr>
<td>leaderspellreg</td>
</tr>
<tr>
<td>democracy</td>
</tr>
<tr>
<td>regime</td>
</tr>
<tr>
<td>start_year</td>
</tr>
<tr>
<td>duration</td>
</tr>
<tr>
<td>observed</td>
</tr>
</tbody>
</table>

References


lifelines.datasets.load_dfcv()

A toy example of a time dependent dataset.
Size: \( (14, 6) \)
Example:

```
start  group  z   stop  id   event
 0   1.0  0   3.0  1   True
 0   1.0  0   5.0  2   False
 0   1.0  1   5.0  3   True
 0   1.0  0   6.0  4   True
```

References


`lifelines.datasets.load_diabetes(**kwargs)`

An interval censored dataset.

References


Size: \( (731, 3) \)
Example:

```
left  right  gender
 24   27   male
 22   22   female
 37   39   male
 20   20   male
 1    16   male
 8    20   female
 14   14   male
```

`lifelines.datasets.load_g3(**kwargs)`

Size: \( (17, 7) \)
Example:

```
no.  age  sex  histology  group  event  time
 1   41  Female  Grade3   RIT    True   53
```

`lifelines.datasets.load_gbsg2(**kwargs)`

A data frame containing the observations from the GBSG2 study of 686 women.
tgrade   II
pnodes   7
progresc 61
estrec   77
time     2018
cens     1

References


lifelines.datasets.load_holly_molly_polly(**kwargs)
From https://stat.ethz.ch/education/semesters/ss2011/seminar/contents/presentation_10.pdf Used as a toy example for CoxPH in recurrent SA:

<table>
<thead>
<tr>
<th>ID</th>
<th>Status</th>
<th>Stratum</th>
<th>Start(days)</th>
<th>Stop(days)</th>
<th>tx</th>
<th>T</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>M</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>100</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>M</td>
<td>1</td>
<td>2</td>
<td>100</td>
<td>105</td>
<td>5</td>
</tr>
<tr>
<td>2</td>
<td>H</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>30</td>
<td>30</td>
</tr>
<tr>
<td>3</td>
<td>H</td>
<td>1</td>
<td>2</td>
<td>30</td>
<td>50</td>
<td>20</td>
</tr>
<tr>
<td>4</td>
<td>P</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>20</td>
<td>20</td>
</tr>
</tbody>
</table>

lifelines.datasets.load_kidney_transplant(**kwargs)

Size: (863, 6)
Example:

time 5
dead 0
age 51
black_male 0
white_male 1
black_female 0

lifelines.datasets.load_larynx(**kwargs)

Size: (89, 6)
Example:

time 77 1 0 0 0
1.3 53 1 0 0 0
2.4 45 1 0 0 0
2.5 57 0 0 0 0
3.2 58 1 0 0 0

lifelines.datasets.load_lcd(**kwargs)
Copper concentrations (µg/L) in shallow groundwater samples from two different geological zones in the San
Joaquin Valley, California. The alluvial fan data include four different detection limits and the basin trough data include five different detection limits.


```
lifelines.datasets.load_leukemia(**kwargs)
Leukemia dataset:
```

```
Example:
C T group
0 1 alluvial_fan
0 1 alluvial_fan
0 1 alluvial_fan
0 1 alluvial_fan
1 1 alluvial_fan
```

```
lifelines.datasets.load_lung(**kwargs)
Survival in patients with advanced lung cancer from the North Central Cancer Treatment Group. Performance scores rate how well the patient can perform usual daily activities.
```

```
Example:
```
```

```
References

From http://web1.sph.emory.edu/dkleinb/allDatasets/surv2datasets/anderson.dat
```

```
lifelines.datasets.load_lupus(**kwargs)
See https://projecteuclid.org/download/pdf_1/euclid.aos/1176345693
```

Note: I transcribed this from the original paper, and highly suspect there are differences.
References


lifelines.datasets.load_lymphoma(**kwargs)

Size: (80, 3)
Example:

<table>
<thead>
<tr>
<th>Stage_group</th>
<th>Time</th>
<th>Censor</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>6</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>19</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>32</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>42</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>42</td>
<td>1</td>
</tr>
</tbody>
</table>

References

From https://www.statsdirect.com/help/content/survival_analysis/logrank.htm

lifelines.datasets.load_multicenter_aids_cohort_study(**kwargs)

Originally in [1]:

Size: (78, 4)

AIDSY: date of AIDS diagnosis
W: years from AIDS diagnosis to study entry
T: years from AIDS diagnosis to minimum of death or censoring
D: indicator of death during follow up

<table>
<thead>
<tr>
<th>i</th>
<th>AIDSY</th>
<th>W</th>
<th>T</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1990.425</td>
<td>4.575</td>
<td>7.575</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>1991.250</td>
<td>3.750</td>
<td>6.750</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>1992.014</td>
<td>2.986</td>
<td>5.986</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>1992.030</td>
<td>2.970</td>
<td>5.970</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>1992.072</td>
<td>2.928</td>
<td>5.928</td>
<td>0</td>
</tr>
<tr>
<td>6</td>
<td>1992.220</td>
<td>2.780</td>
<td>4.688</td>
<td>1</td>
</tr>
</tbody>
</table>

References


lifelines.datasets.load_nh4(**kwargs)

Ammonium (NH4) concentration (mg/L) in precipitation measured at Olympic National Park, Hoh Ranger Station (WA14), weekly or every other week from January 6, 2009 through December 20, 2011.


Size: (104, 3)
lifelines.datasets.load_panel_test(**kwargs)

Size: (28,5)
Example:
   id  t  E  var1  var2
   1  1  0  0.0  1
   1  2  0  0.0  1
   1  3  0  4.0  3
   1  4  1  8.0  4
   2  1  0  1.2  1

lifelines.datasets.load_psychiatric_patients(**kwargs)

Size: (26,4)
Example:
   Age  T  C  sex
   51  1  1  2
   58  1  1  2
   55  2  1  2
   28  22  1  2
   21  30  0  1

lifelines.datasets.load_recur(**kwargs)


ID  Subject Identification  1 - 400
AGE  Age  years
TREAT  Treatment Assignment  0 = New
       1 = Old
TIME0  Day of Previous Episode  Days
TIME1  Day of New Episode  Days
       or censoring
CENSOR  Indicator for Soreness  1 = Episode Occurred
       at TIME1
       0 = Censored
EVENT  Soreness Episode Number  0 to at most 4

Size: (1296, 7)
Example:
   ID,AGE,TREAT,TIME0,TIME1,CENSOR,EVENT
   1,43,0,9,56,1,3
   1,43,0,56,88,1,4
   1,43,0,6,1,1
   1,43,0,6,9,1,2

lifelines.datasets.load_regression_dataset(**kwargs)

Artificial regression dataset. Useful since there are no ties in this dataset. Slightly edit in v0.15.0 to achieve this, however:

Size: (200,5)
Example:
   var1  var2  var3  T  E
   0.595170 1.143472 1.571079 14.785479 1
   0.209325 0.184677 0.356980 7.336734 1

(continues on next page)
0.693919 0.071893 0.557960 5.271527 1
0.443804 1.364646 0.374221 11.684168 1
1.613324 0.125566 1.921325 7.637764 1

**lifelines.datasets.load_rossi(****kwargs**)**

This data set is originally from Rossi et al. (1980), and is used as an example in Allison (1995). The data pertain to 432 convicts who were released from Maryland state prisons in the 1970s and who were followed up for one year after release. Half the released convicts were assigned at random to an experimental treatment in which they were given financial aid; half did not receive aid.:

**References**


**lifelines.datasets.load_stanford_heart_transplants(****kwargs**)**

This is a classic dataset for survival regression with time varying covariates. The original dataset is from [1], and this dataset is from R’s survival library.:  

**References**


**lifelines.datasets.load_static_test(****kwargs**)**

**References**

1.10. Reference library for lifelines
lifelines.datasets.load_waltons(**kwargs)

Genotypes and number of days survived in Drosophila. Since we work with flies, we don’t need to worry about left-censoring. We know the birth date of all flies. We do have issues with accidentally killing some or if some escape. These would be right-censored as we do not actually observe their death due to “natural” causes.:  

Size: (163,3)
Example:

<table>
<thead>
<tr>
<th>T</th>
<th>E</th>
<th>group</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>1</td>
<td>miR-137</td>
</tr>
<tr>
<td>13</td>
<td>1</td>
<td>miR-137</td>
</tr>
<tr>
<td>13</td>
<td>1</td>
<td>miR-137</td>
</tr>
<tr>
<td>13</td>
<td>1</td>
<td>miR-137</td>
</tr>
<tr>
<td>19</td>
<td>1</td>
<td>miR-137</td>
</tr>
</tbody>
</table>

1.11 Changelog

1.11.1 0.21.2 - 2019-05-16

New features

- New regression model: PiecewiseExponentialRegressionFitter is available. See blog post here: https://dataorigami.net/blogs/napkin-folding/churn
- Regression models have a new method log_likelihood_ratio_test that computes, you guessed it, the log-likelihood ratio test. Previously this was an internal API that is being exposed.

API changes

- The default behavior of the predict method on non-parametric estimators (KaplanMeierFitter, etc.) has changed from (previous) linear interpolation to (new) return last value. Linear interpolation is still possible with the interpolate flag.
- removing _compute_likelihood_ratio_test on regression models. Use log_likelihood_ratio_test now.

Bug fixes

1.11.2 0.21.1 - 2019-04-26

New features

- users can provided their own start and stop column names in add_covariate_to_timeline
- PiecewiseExponentialFitter now allows numpy arrays as breakpoints
API changes

- output of `survival_table_from_events` when collapsing rows to intervals now removes the “aggregate” column multi-index.

Bug fixes

- fixed bug in CoxTimeVaryingFitter when ax is provided, thanks @j-i-l!

1.11.3 0.21.0 - 2019-04-12

New features

- weights is now a optional kwarg for parametric univariate models.
- all univariate and multivariate parametric models now have ability to handle left, right and interval censored data (the former two being special cases of the latter). Users can use the `fit_right_censoring` (which is an alias for `fit`), `fit_left_censoring` and `fit_interval_censoring`.
- a new interval censored dataset is available under `lifelines.datasets.load_diabetes`

API changes

- `left_censorship` on all univariate fitters has been deprecated. Please use the new api model. `fit_left_censoring(...)`. 
- `invert_y_axis` in `model.plot(...) has been removed.`
- `entries` property in multivariate parametric models has a new Series name: `entry`

Bug fixes

- lifelines was silently converting any NaNs in the event vector to True. An error is now thrown instead.
- Fixed an error that didn’t let users use Numpy arrays in prediction for AFT models

1.11.4 0.20.5 - 2019-04-08

New features

- performance improvements for `print_summary`.

API changes

- `utils.survival_events_from_table` returns an integer weight vector as well as durations and censoring vector.
- in `AalenJohansenFitter`, the `variance` parameter is renamed to `variance_` to align with the usual `lifelines` convention.
Bug fixes

- Fixed an error in the CoxTimeVaryingFitter's likelihood ratio test when using strata.
- Fixed some plotting bugs with AalenJohansenFitter

1.11.5 0.20.4 - 2019-03-27

New features

- left-truncation support in AFT models, using the entry_col kwarg in fit()
- generate_datasets.piecewise_exponential_survival_data for generating piecewise exp. data
- Faster print_summary for AFT models.

API changes

- Pandas is now correctly pinned to >= 0.23.0. This was always the case, but not specified in setup.py correctly.

Bug fixes

- Better handling for extremely large numbers in print_summary
- PiecewiseExponentialFitter is available with from lifelines import *.

1.11.6 0.20.3 - 2019-03-23

New features

- Now cumulative_density_ & survival_function_ are always present on a fitted KaplanMeierFitter.
- New attributes/methods on KaplanMeierFitter: plot_cumulative_density(), confidence_interval_cumulative_density_, plot_survival_function and confidence_interval_survival_function_.

1.11.7 0.20.2 - 2019-03-21

New features

- Left censoring is now supported in univariate parametric models: .fit(..., left_censorship=True). Examples are in the docs.
- new dataset: lifelines.datasets.load_nh4()
- Univariate parametric models now include, by default, support for the cumulative density function: .cumulative_density_, .confidence_interval_cumulative_density_, plot_cumulative_density(), cumulative_density_at_times(t).
- add a lifelines.plotting.qq_plot for univariate parametric models that handles censored data.
API changes

- `plot_lifetimes` no longer reverses the order when plotting. Thanks @vpolimenov!
- The C column in `load_lcd` dataset is renamed to E.

Bug fixes

- Fixed a naming error in `KaplanMeierFitter` when `left_censorship` was set to True, `plot_cumulative_density_()` is now `plot_cumulative_density()`.
- Added some error handling when passing in timedeltas. Ideally, users don’t pass in timedeltas, as the scale is ambiguous. However, the error message before was not obvious, so we do some conversion, warn the user, and pass it through.
- `qth_survival_times` for a truncated CDF would return `np.inf` if the q parameter was below the truncation limit. This should have been `-np.inf`.

1.11.8 0.20.1 - 2019-03-16

- Some performance improvements to `CoxPHFitter` (about 30%). I know it may seem silly, but we are now about the same or slightly faster than the Cox model in R’s `survival` package (for some testing datasets and some configurations). This is a big deal, because 1) lifelines does more error checking prior, 2) R’s cox model is written in C, and we are still pure Python/NumPy, 3) R’s cox model has decades of development.
- Suppressed unimportant warnings.

API changes

- Previously, lifelines always added a 0 row to `cph.baseline_hazard_`, even if there were no event at this time. This is no longer the case. A 0 will still be added if there is a duration (observed or not) at 0 occurs however.

1.11.9 0.20.0 - 2019-03-05

- Starting with 0.20.0, only Python3 will be supported. Over 75% of recent installs were Py3.
- Updated minimum dependencies, specifically Matplotlib and Pandas.

New features

- Smarter initialization for AFT models which should improve convergence.

API changes

- `initial_beta` in Cox model’s `.fit` is now `initial_point`.
- `initial_point` is now available in AFT models and `CoxTimeVaryingFitter`
- The DataFrame `confidence_intervals_` for univariate models is transposed now (previous parameters where columns, now parameters are rows).
Bug fixes

- Fixed a bug with plotting and `check_assumptions`.

1.11.10 0.19.5 - 2019-02-26

New features

- `plot_covariate_group` can accept multiple covariates to plot. This is useful for columns that have implicit correlation like polynomial features or categorical variables.
- Convergence improvements for AFT models.

1.11.11 0.19.4 - 2019-02-25

Bug fixes

- remove some bad print statements in `CoxPHFitter`.

1.11.12 0.19.3 - 2019-02-25

New features

- new AFT models: `LogNormalAFTFitter` and `LogLogisticAFTFitter`.
- AFT models now accept a `weights_col` argument to `fit`.
- Robust errors (sandwich errors) are now available in AFT models using the `robust=True` kwarg in `fit`.
- Performance increase to `print_summary` in the `CoxPHFitter` and `CoxTimeVaryingFitter` model.

1.11.13 0.19.2 - 2019-02-22

New features

- `ParametricUnivariateFitters`, like `WeibullFitter`, have smoothed plots when plotting (vs stepped plots)

Bug fixes

- The `ExponentialFitter` log likelihood value was incorrect - inference was correct however.
- Univariate fitters are more flexible and can allow 2-d and DataFrames as inputs.

1.11.14 0.19.1 - 2019-02-21

New features

- improved stability of `LogNormalFitter`
- Matplotlib for Python3 users are not longer forced to use 2.x.
API changes

- **Important**: we changed the parameterization of the \texttt{PiecewiseExponential} to the same as \texttt{ExponentialFitter} (from $\lambda \times t$ to $t / \lambda$).

1.11.15 0.19.0 - 2019-02-20

New features

- New regression model \texttt{WeibullAFTFitter} for fitting accelerated failure time models. Docs have been added to our documentation about how to use \texttt{WeibullAFTFitter} (spoiler: it's API is similar to the other regression models) and how to interpret the output.
- \texttt{CoxPHFitter} performance improvements (about 10%)
- \texttt{CoxTimeVaryingFitter} performance improvements (about 10%)

API changes

- **Important**: we changed the \texttt{.hazards_} and \texttt{.standard_errors_} on Cox models to be pandas Series (instead of Dataframes). This felt like a more natural representation of them. You may need to update your code to reflect this. See notes here: https://github.com/CamDavidsonPilon/lifelines/issues/636
- **Important**: we changed the \texttt{.confidence_intervals_} on Cox models to be transposed. This felt like a more natural representation of them. You may need to update your code to reflect this. See notes here: https://github.com/CamDavidsonPilon/lifelines/issues/636
- **Important**: we changed the parameterization of the \texttt{WeibullFitter} and \texttt{ExponentialFitter} from $\lambda \times t$ to $t / \lambda$. This was for a few reasons: 1) it is a more common parameterization in literature, 2) it helps in convergence.
- **Important**: in models where we add an intercept (currently only \texttt{AalenAdditiveModel}), the name of the added column has been changed from \texttt{baseline} to \texttt{_intercept}
- **Important**: the meaning of \texttt{alpha} in all fitters has changed to be the standard interpretation of \texttt{alpha} in confidence intervals. That means that the \texttt{default} for \texttt{alpha} is set to 0.05 in the latest lifelines, instead of 0.95 in previous versions.

Bug Fixes

- Fixed a bug in the \texttt{_log_likelihood_} property of \texttt{ParametericUnivariateFitter} models. It was showing the “average” log-likelihood (i.e. scaled by $1/n$) instead of the total. It now displays the total.
- In model \texttt{print_summaries}, correct a label erroring. Instead of “Likelihood test”, it should have read “Log-likelihood test”.
- Fixed a bug that was too frequently rejecting the dtype of \texttt{event} columns.
- Fixed a calculation bug in the concordance index for stratified Cox models. Thanks @airanmehr!
- Fixed some Pandas <0.24 bugs.
1.11.16 0.18.6 - 2019-02-13

- some improvements to the output of check_assumptions. show_plots is turned to False by default now. It only shows rank and km p-values now.
- some performance improvements to qth_survival_time.

1.11.17 0.18.5 - 2019-02-11

- added new plotting methods to parametric univariate models: plot_survival_function, plot_hazard and plot_cumulative_hazard. The last one is an alias for plot.
- added new properties to parametric univariate models: confidence_interval_survival_function_, confidence_interval_hazard_, confidence_interval_cumulative_hazard_. The last one is an alias for confidence_interval_.
- Fixed some overflow issues with AalenJohansenFitter’s variance calculations when using large datasets.
- Fixed an edgecase in AalenJohansenFitter that causing some datasets with to be jittered too often.
- Add a new kwarg to AalenJohansenFitter, calculate_variance that can be used to turn off variance calculations since this can take a long time for large datasets. Thanks @pzivich!

1.11.18 0.18.4 - 2019-02-10

- fixed confidence intervals in cumulative hazards for parametric univariate models. They were previously serverly depressed.
- adding left-truncation support to parametric univariate models with the entry kwarg in .fit

1.11.19 0.18.3 - 2019-02-07

- Some performance improvements to parametric univariate models.
- Suppressing some irrelevant NumPy and autograd warnings, so lifeline warnings are more noticeable.
- Improved some warning and error messages.

1.11.20 0.18.2 - 2019-02-05

- New univariate fitter PiecewiseExponentialFitter for creating a stepwise hazard model. See docs online.
- Ability to create novel parametric univariate models using the new ParametericUnivariateFitter super class. See docs online for how to do this.
- Unfortunately, parametric univariate fitters are not serializable with pickle. The library dill is still useable.
- Complete overhaul of all internals for parametric univariate fitters. Moved them all (most) to use autograd.
- LogNormalFitter no longer models log_sigma.
1.11.21 0.18.1 2019-02-02

- bug fixes in LogNormalFitter variance estimates
- improve convergence of LogNormalFitter. We now model the log of sigma internally, but still expose sigma externally.
- use the autograd lib to help with gradients.
- New LogLogisticFitter univariate fitter available.

1.11.22 0.18.0 - 2019-01-31

- LogNormalFitter is a new univariate fitter you can use.
- WeibullFitter now correctly returns the confidence intervals (previously returned only NaNs)
- WeibullFitter.print_summary() displays p-values associated with its parameters not equal to 1.0 - previously this was (implicitly) comparing against 0, which is trivially always true (the parameters must be greater than 0)
- ExponentialFitter.print_summary() displays p-values associated with its parameters not equal to 1.0 - previously this was (implicitly) comparing against 0, which is trivially always true (the parameters must be greater than 0)
- ExponentialFitter.plot now displays the cumulative hazard, instead of the survival function. This is to make it easier to compare to WeibullFitter and LogNormalFitter
- Univariate fitters' cumulative_hazard_at_times, hazard_at_times, survival_function_at_times return pandas Series now (use to be numpy arrays)
- remove alpha keyword from all statistical functions. This was never being used.
- Gone are asterisks and dots in print_summary functions that represent significance thresholds.
- In models' summary (including print_summary), the log(p) term has changed to -log2(p). This is known as the s-value. See https://lesslikely.com/statistics/s-values/
- introduce new statistical tests between univariate datasets: survival_difference_at_fixed_point_in_time_test,
- new warning message when Cox models detects possible non-unique solutions to maximum likelihood.
- Generally: clean up lifelines exception handling. Ex: catch LinAlgError: Matrix is singular. and report back to the user advice.

1.11.23 0.17.5 - 2019-01-25

- more bugs in plot_covariate_groups fixed when using non-numeric strata.

1.11.24 0.17.4 -2019-01-25

- Fix bug in plot_covariate_groups that wasn’t allowing for strata to be used.
- change name of multicenter_aids_cohort_study to load_multicenter_aids_cohort_study
- groups is now called values in CoxPHFitter.plot_covariate_groups
1.11.25 0.17.3 - 2019-01-24

- Fix in compute_residuals when using schoenfeld and the minimum duration has only censored subjects.

1.11.26 0.17.2 2019-01-22

- Another round of serious performance improvements for the Cox models. Up to 2x faster for CoxPHFitter and CoxTimeVaryingFitter. This was mostly the result of using NumPy’s einsum to simplify a previous for loop. The downside is the code is more esoteric now. I’ve added comments as necessary though.

1.11.27 0.17.1 - 2019-01-20

- Adding bottleneck as a dependency. This library is highly-recommended by Pandas, and in lifelines we see some nice performance improvements with it too. (~15% for CoxPHFitter)
- There was a small bug in CoxPHFitter when using batch_mode that was causing coefficients to deviate from their MLE value. This bug eluded tests, which means that it’s discrepancy was less than 0.0001 difference. It’s fixed now, and even more accurate tests are added.
- Faster CoxPHFitter._compute_likelihood_ratio_test()
- Fixes a Pandas performance warning in CoxTimeVaryingFitter.
- Performances improvements to CoxTimeVaryingFitter.

1.11.28 0.17.0 - 2019-01-11

- Corrected behaviour in CoxPHFitter where score_ was not being refreshed on every new fit.
- Reimplementation of AalenAdditiveFitter. There were significant changes to it:
  - implementation is at least 10x faster, and possibly up to 100x faster for some datasets.
  - memory consumption is way down
  - removed the time-varying component from AalenAdditiveFitter. This will return in a future release.
  - new print_summary
  - weights_col is added
  - nn_cumulative_hazard is removed (may add back)
- Some plotting improvements to plotting.plot_lifetimes

1.11.29 0.16.3 - 2019-01-03

- More CoxPHFitter performance improvements. Up to a 40% reduction vs 0.16.2 for some datasets.
1.11.30 0.16.2 - 2019-01-02

- Fixed CoxTimeVaryingFitter to allow more than one variable to be stratafied
- Significant performance improvements for CoxPHFitter with dataset has lots of duplicate times. See https://github.com/CamDavidsonPilon/lifelines/issues/591

1.11.31 0.16.1 - 2019-01-01

- Fixed py2 division error in concordance method.

1.11.32 0.16.0 - 2019-01-01

- Drop Python 3.4 support.
- introduction of residual calculations in CoxPHFitter.compute_residuals. Residuals include “schoenfeld”, “score”, “delta_beta”, “deviance”, “martingale”, and “scaled_schoenfeld”.
- removes estimation namespace for fitters. Should be using from lifelines import xFitter now. Thanks @usmanatron
- removes predict_log_hazard_relative_to_mean from Cox model. Thanks @usmanatron
- StatisticalResult has be generalized to allow for multiple results (ex: from pairwise comparisons). This means a slightly changed API that is mostly backwards compatible. See doc string for how to use it.
- statistics.pairwise_logrank_test now returns a StatisticalResult object instead of a nasty NxN DataFrame
- Display log(p-values) as well as p-values in print_summary. Also, p-values below thresholds will be truncated. The original p-values are still recoverable using .summary.
- Floats print_summary is now displayed to 2 decimal points. This can be changed using the decimal kwarg.
- removed standardized from Cox model plotting. It was confusing.
- visual improvements to Cox models .plot
- print_summary methods accepts kwargs to also be displayed.
- CoxPHFitter has a new human-readable method, check_assumptions, to check the assumptions of your Cox proportional hazard model.
- A new helper util to “expand” static datasets into long-form: lifelines.utils.to_episodic_format.
- CoxTimeVaryingFitter now accepts strata.

1.11.33 0.15.4

- bug fix for the Cox model likelihood ratio test when using non-trivial weights.

1.11.34 0.15.3 - 2018-12-18

- Only allow matplotlib less than 3.0.
1.11.35 0.15.2 - 2018-11-23

- API changes to `plotting.plot_lifetimes`
- `cluster_col` and `strata` can be used together in `CoxPHFitter`
- removed entry from `ExponentialFitter` and `WeibullFitter` as it was doing nothing.

1.11.36 0.15.1 - 2018-11-23

- Bug fixes for v0.15.0
- Raise `NotImplementedError` if the `robust` flag is used in `CoxTimeVaryingFitter` - that's not ready yet.

1.11.37 0.15.0 - 2018-11-22

- adding `robust` params to `CoxPHFitter`'s `fit`. This enables atleast i) using non-integer weights in the model (these could be sampling weights like IPTW), and ii) mis-specified models (ex: non-proportional hazards). Under the hood it's a sandwich estimator. This does not handle ties, so if there are high number of ties, results may significantly differ from other software.

- `standard_errors_` is now a property on fitted `CoxPHFitter` which describes the standard errors of the coefficients.

- `variance_matrix_` is now a property on fitted `CoxPHFitter` which describes the variance matrix of the coefficients.

- new criteria for convergence of `CoxPHFitter` and `CoxTimeVaryingFitter` called the Newton-decrement. Tests show it is as accurate (w.r.t to previous coefficients) and typically shaves off a single step, resulting in generally faster convergence. See https://www.cs.cmu.edu/~pradeepr/convexopt/Lecture_Slides/Newton_methods.pdf. Details about the Newton-decrement are added to the `show_progress` statements.

- Minimum support for scipy is 1.0

- Convergence errors in models that use Newton-Rhapson methods now throw a `ConvergenceError`, instead of a `ValueError` (the former is a subclass of the latter, however).

- `AalenAdditiveModel` raises `ConvergenceWarning` instead of printing a warning.

- `KaplanMeierFitter` now has a cumulative plot option. Example `kmf.plot(invert_y_axis=True)`

- a `weights_col` option has been added to `CoxTimeVaryingFitter` that allows for time-varying weights.

- `WeibullFitter` has a new `show_progress` param and additional information if the convergence fails.

- `CoxPHFitter`, `ExponentialFitter`, `WeibullFitter` and `CoxTimeVaryingFitter` method `print_summary` is updated with new fields.

- `WeibullFitter` has renamed the incorrect `_jacobian` to `_hessian_`

- `variance_matrix_` is now a property on fitted `WeibullFitter` which describes the variance matrix of the parameters.

- The default `WeibullFitter().timeline` has changed from integers between the min and max duration to n floats between the max and min durations, where n is the number of observations.

- Performance improvements for `CoxPHFitter` (~20% faster)

- Performance improvements for `CoxTimeVaryingFitter` (~100% faster)

- In Python3, Univariate models are now serialisable with `pickle`. Thanks @dwilson1988 for the contribution. For Python2, `dill` is still the preferred method.
- `baseline_cumulative_hazard_` (and derivatives of that) on `CoxPHFitter` now correctly incorporate the `weights_col`.
- Fixed a bug in `KaplanMeierFitter` when late entry times lined up with death events. Thanks @pzivich
- Adding `cluster_col` argument to `CoxPHFitter` so users can specify groups of subjects/rows that may be correlated.
- Shifting the “significance codes” for p-values down an order of magnitude. (Example, p-values between 0.1 and 0.05 are not noted at all and p-values between 0.05 and 0.1 are noted with ., etc.). This deviates with how they are presented in other software. There is an argument to be made to remove p-values from lifelines altogether *(become the changes you want to see in the world lol)*, but I worry that people could compute the p-values by hand incorrectly, a worse outcome I think. So, this is my stance. P-values between 0.1 and 0.05 offer very little information, so they are removed. There is a growing movement in statistics to shift “significant” findings to p-values less than 0.01 anyways.
- New fitter for cumulative incidence of multiple risks `AalenJohansenFitter`. Thanks @pzivich! See “Methodologic Issues When Estimating Risks in Pharmacoepidemiology” for a nice overview of the model.

1.11.38 0.14.6 - 2018-07-02

- fix for n > 2 groups in `multivariate_logrank_test` (again).
- fix bug for when `event_observed` column was not boolean.

1.11.39 0.14.5 - 2018-06-29

- fix for n > 2 groups in `multivariate_logrank_test`
- fix weights in `KaplanMeierFitter` when using a pandas Series.

1.11.40 0.14.4 - 2018-06-14

- Adds `baseline_cumulative_hazard_` and `baseline_survival_` to `CoxTimeVaryingFitter`. Because of this, new prediction methods are available.
- fixed a bug in `add_covariate_to_timeline` when using `cumulative_sum` with multiple columns.
- Added Likelihood ratio test to `CoxPHFitter.print_summary` and `CoxTimeVaryingFitter.print_summary`
- New checks in `CoxTimeVaryingFitter` that check for immediate deaths and redundant rows.
- New delay parameter in `add_covariate_to_timeline`
- removed `two_sided_z_test` from `statistics`

1.11.41 0.14.3 - 2018-05-24

- fixes a bug when subtracting or dividing two `UnivariateFitters` with labels.
- fixes an import error with using `CoxTimeVaryingFitter` predict methods.
- adds a column argument to `CoxTimeVaryingFitter` and `CoxPHFitter` plot method to plot only a subset of columns.
1.11.42 0.14.2 - 2018-05-18

- some quality of life improvements for working with CoxTimeVaryingFitter including new predict_ methods.

1.11.43 0.14.1 - 2018-04-01

- fixed bug with using weights and strata in CoxPHFitter
- fixed bug in using non-integer weights in KaplanMeierFitter
- Performance optimizations in CoxPHFitter for up to 40% faster completion of fit.
  - even smarter step_size calculations for iterative optimizations.
  - simple code optimizations & cleanup in specific hot spots.
- Performance optimizations in AalenAdditiveFitter for up to 50% faster completion of fit for large dataframes, and up to 10% faster for small dataframes.

1.11.44 0.14.0 - 2018-03-03

- adding plot_covariate_groups to CoxPHFitter to visualize what happens to survival as we vary a covariate, all else being equal.
- utils functions like qth_survival_times and median_survival_times now return the transpose of the DataFrame compared to previous version of lifelines. The reason for this is that we often treat survival curves as columns in DataFrames, and functions of the survival curve as index (ex: KaplanMeierFitter.survival_function_ returns a survival curve at time t).
- KaplanMeierFitter.fit and NelsonAalenFitter.fit accept a weights vector that can be used for pre-aggregated datasets. See this issue.
- Convergence errors now return a custom ConvergenceWarning instead of a RuntimeWarning
- New checks for complete separation in the dataset for regressions.

1.11.45 0.13.0 - 2017-12-22

- removes is_significant and test_result from StatisticalResult. Users can instead choose their significance level by comparing to p_value. The string representation of this class has changed aswell.
- CoxPHFitter and AalenAdditiveFitter now have a score_ property that is the concordance-index of the dataset to the fitted model.
- CoxPHFitter and AalenAdditiveFitter no longer have the data property. It was an almost duplicate of the training data, but was causing the model to be very large when serialized.
- Implements a new fitter CoxTimeVaryingFitter available under the lifelines namespace. This model implements the Cox model for time-varying covariates.
- Utils for creating time varying datasets available in utils.
- less noisy check for complete separation.
- removed datasets namespace from the main lifelines namespace
- CoxPHFitter has a slightly more intelligent (barely...) way to pick a step size, so convergence should generally be faster.
• CoxPHFitter.fit now has accepts a weight_col kwarg so one can pass in weights per observation. This is very useful if you have many subjects, and the space of covariates is not large. Thus you can group the same subjects together and give that observation a weight equal to the count. Altogether, this means a much faster regression.

1.11.46 0.12.0
• removes include_likelihood from CoxPHFitter.fit - it was not slowing things down much (empirically), and often I wanted it for debugging (I suppose others do too). It’s also another exit condition, so we may exit from the NR iterations faster.
• added step_size param to CoxPHFitter.fit - the default is good, but for extremely large or small datasets this may want to be set manually.
• added a warning to CoxPHFitter to check for complete seperation: https://stats.idre.ucla.edu/other/mult-pkg/faq/general/faqwhat-is-complete-or-quasi-complete-separation-in-logisticprobit-regression-and-how-do-we-deal-with-them/
• Additional functionality to utils.survival_table_from_events to bin the index to make the resulting table more readable.

1.11.47 0.11.3
• No longer support matplotlib 1.X
• Adding times argument to CoxPHFitter's predict_survival_function and predict_cumulative_hazard to predict the estimates at, instead uses the default times of observation or censorship.
• More accurate prediction methods parametrics univariate models.

1.11.48 0.11.2
• Changing liscense to valilla MIT.
• Speed up NelsonAalenFitter.fit considerably.

1.11.49 0.11.1 - 2017-06-22
• Python3 fix for CoxPHFitter.plot.

1.11.50 0.11.0 - 2017-06-21
• fixes regression in KaplanMeierFitter.plot when using Seaborn and lifelines.
• introduce a new .plot function to a fitted CoxPHFitter instance. This plots the hazard coefficients and their confidence intervals.
• in all plot methods, the ix kwarg has been deprecated in favour of a new loc kwarg. This is to align with Pandas deprecating ix.

1.11.51 0.10.1 - 2017-06-05
• fix in internal normalization for CoxPHFitter predict methods.
• corrected bug that was returning the wrong baseline survival and hazard values in CoxPHFitter when normalize=True.
• removed normalize kwarg in CoxPHFitter. This was causing lots of confusion for users, and added code complexity. It’s really nice to be able to remove it.
• correcting column name in CoxPHFitter.baseline_survival_.
• CoxPHFitter.baseline_cumulative_hazard_ is always centered, to mimic R’s basehaz API.
• new predict_log_partial_hazards to CoxPHFitter

1.11.53 0.9.4

• adding plot_loglogrs to KaplanMeierFitter
• added a (correct) check to see if some columns in a dataset will cause convergence problems.
• removing flat argument in plot methods. It was causing confusion. To replicate it, one can set ci_force_lines=True and show_censors=True.
• adding strata keyword argument to CoxPHFitter on initialization (ex: CoxPHFitter(strata=['v1', 'v2'])). Why? Fitters initialized with strata can now be passed into k_fold_cross_validation, plus it makes unit testing strata fitters easier.
• If using strata in CoxPHFitter, access to strata specific baseline hazards and survival functions are available (previously it was a blended value). Prediction also uses the specific baseline hazards/survivals.
• performance improvements in CoxPHFitter - should see at least a 10% speed improvement in fit.

1.11.54 0.9.2

• deprecates Pandas versions before 0.18.
• throw an error if no admissable pairs in the c-index calculation. Previously a NaN was returned.

1.11.55 0.9.1

• add two summary functions to Weibull and Exponential fitter, solves #224

1.11.56 0.9.0

• new prediction function in CoxPHFitter, predict_log_hazard_relative_to_mean, that mimics what R’s predict.coxph does.
• removing the predict method in CoxPHFitter and AalenAdditiveFitter. This is because the choice of predict_median as a default was causing too much confusion, and no other natural choice as a default was available. All other predict_methods remain.
• Default predict method in k_fold_cross_validation is now predict_expectation
1.11.57 0.8.1 - 2015-08-01

- supports matplotlib 1.5.
- introduction of a param `nn_cumulative_hazards` in `AalenAdditiveModel`'s `__init__` (default True). This parameter will truncate all non-negative cumulative hazards in prediction methods to 0.
- bug fixes including:
  - fixed issue where the while loop in `_newton_rhaphson` would break too early causing a variable not to be set properly.
  - scaling of smooth hazards in `NelsonAalenFitter` was off by a factor of 0.5.

1.11.58 0.8.0

- reorganized lifelines directories:
  - moved test files out of main directory.
  - moved `utils.py` into it's own directory.
  - moved all estimators fitters directory.

- added a `at_risk` column to the output of `group_survival_table_from_events` and `survival_table_from_events`

- added sample size and power calculations for statistical tests. See `lifeline.statistics`. `sample_size_necessary_under_cph` and `lifelines.statistics.power_under_cph`.

- fixed a bug when using `KaplanMeierFitter` for left-censored data.

1.11.59 0.7.1

- addition of a l2 penalizer to `CoxPHFitter`.
- dropped Fortran implementation of efficient Python version. Lifelines is pure python once again!
- addition of `strata` keyword argument to `CoxPHFitter` to allow for stratification of a single or set of categorical variables in your dataset.
- `datetimes_to_durations` now accepts a list as `na_values`, so multiple values can be checked.
- fixed a bug in `datetimes_to_durations` where `fill_date` was not properly being applied.
- Changed warning in `datetimes_to_durations` to be correct.
- refactor each fitter into it's own submodule. For now, the tests are still in the same file. This will also _not_ break the API.

1.11.60 0.7.0 - 2015-03-01

- allow for multiple fitters to be passed into `k_fold_cross_validation`.
- statistical tests in `lifelines.statistics` now return a `StatisticalResult` object with properties like `p_value`, `test_results`, and `summary`.
- fixed a bug in how log-rank statistical tests are performed. The covariance matrix was not being correctly calculated. This resulted in slightly different p-values.
WeibullFitter, ExponentialFitter, KaplanMeierFitter and BreslowFlemingHarringtonFitter all have a conditional_time_to_event_ property that measures the median duration remaining until the death event, given survival up until time $t$.

1.11.61 0.6.1

- addition of median_ property to WeibullFitter and ExponentialFitter.
- WeibullFitter and ExponentialFitter will use integer timelines instead of float provided by linspace. This is so if your work is to sum up the survival function (for expected values or something similar), it’s more difficult to make a mistake.

1.11.62 0.6.0 - 2015-02-04

- Inclusion of the univariate fitters WeibullFitter and ExponentialFitter.
- Removing BayesianFitter from lifelines.
- Added new penalization scheme to AalenAdditiveFitter. You can now add a smoothing penalizer that will try to keep subsequent values of a hazard curve close together. The penalizing coefficient is smoothing_penalizer.
- Changed penalizer keyword arg to coef_penalizer in AalenAdditiveFitter.
- new ridge_regression function in utils.py to perform linear regression with l2 penalizer terms.
- Matplotlib is no longer a mandatory dependency.
- .predict(time) method on univariate fitters can now accept a scalar (and returns a scalar) and an iterable (and returns a numpy array)
- In KaplanMeierFitter, epsilon has been renamed to precision.

1.11.63 0.5.1 - 2014-12-24

- New API for CoxPHFitter and AalenAdditiveFitter: the default arguments for event_col and duration_col. duration_col is now mandatory, and event_col now accepts a column, or by default, None, which assumes all events are observed (non-censored).
- Fix statistical tests.
- Allow negative durations in Fitters.
- New API in survival_table_from_events: min_observations is replaced by birth_times (default None).
- New API in CoxPHFitter for summary: summary will return a dataframe with statistics, print_summary() will print the dataframe (plus some other statistics) in a pretty manner.
- Adding “At Risk” counts option to univariate fitter plot methods, .plot(at_risk_counts=True), and the function lifelines.plotting.add_at_risk_counts.
- Fix bug Epanechnikov kernel.
1.11.64 0.5.0 - 2014-12-07

- move testing to py.test
- refactor tests into smaller files
- make test_pairwise_logrank_test_with_identical_data_returns_inconclusive a better test
- add test for summary()
- Alternate metrics can be used for k_fold_cross_validation.

1.11.65 0.4.4 - 2014-11-27

- Lots of improvements to numerical stability (but something things still need work)
- Additions to summary in CoxPHFitter.
- Make all prediction methods output a DataFrame
- Fixes bug in 1-d input not returning in CoxPHFitter
- Lots of new tests.

1.11.66 0.4.3 - 2014-07-23

- refactoring of qth_survival_times: it can now accept an iterable (or a scalar still) of probabilities in the q argument, and will return a DataFrame with these as columns. If len(q)==1 and a single survival function is given, will return a scalar, not a DataFrame. Also some good speed improvements.
- KaplanMeierFitter and NelsonAalenFitter now have a _label property that is passed in during the fit.
- KaplanMeierFitter/NelsonAalenFitter's initial alpha value is overwritten if a new alpha value is passed in during the fit.
- New method for KaplanMeierFitter: conditional_time_to. This returns a DataFrame of the estimate: med(S(t | T>s)) - s, human readable: the estimated time left of living, given an individual is aged s.
- Adds option include_likelihood to CoxPHFitter fit method to save the final log-likelihood value.

1.11.67 0.4.2 - 2014-06-19

- Massive speed improvements to CoxPHFitter.
- Additional prediction method: predict_percentile is available on CoxPHFitter and AalenAdditiveFitter. Given a percentile, p, this function returns the value t such that S(t | x) = p. It is a generalization of predict_median.
- Additional kwargs in k_fold_cross_validation that will accept different prediction methods (default is predict_median).
- Bug fix in CoxPHFitter predict_expectation function.
- Correct spelling mistake in newton-rhapson algorithm.
- datasets now contains functions for generating the respective datasets, ex: generate_waltons_dataset.
- Bumping up the number of samples in statistical tests to prevent them from failing so often (this a stop-gap)
• pep8 everything

1.11.68 0.4.1.1

• Ability to specify default printing in statistical tests with the suppress_print keyword argument (default False).
• For the multivariate log rank test, the inverse step has been replaced with the generalized inverse. This seems to be what other packages use.
• Adding more robust cross validation scheme based on issue #67.
• fixing regression_dataset in datasets.

1.11.69 0.4.1 - 2014-06-11

• CoxFitter is now known as CoxPHFitter
• refactoring some tests that used redundant data from lifelines.datasets.
• Adding cross validation: in utils is a new k_fold_cross_validation for model selection in regression problems.
• Change CoxPHFitter’s fit method’s display_output to False.
• fixing bug in CoxPHFitter’s _compute_baseline_hazard that errored when sending Series objects to survival_table_from_events.
• CoxPHFitter’s fit now looks to columns with too low variance, and halts NR algorithm if a NaN is found.
• Adding a Changelog.
• more sanitizing for the statistical tests =)

1.11.70 0.4.0 - 2014-06-08

• CoxFitter implements Cox Proportional Hazards model in lifelines.
• lifelines moves the wheels distributions.
• tests in the statistics module now prints the summary (and still return the regular values)
• new BaseFitter class is inherited from all fitters.

1.12 Installation

```
pip install lifelines
```

1.13 Source code and issue tracker

Available on Github, CamDavidsonPilon/lifelines. Please report bugs, issues and feature extensions there. We also have Gitter channel available to discuss survival analysis and lifelines:
1.14 Citing *lifelines*

The following link will bring you to a page where you can find the latest citation for *lifelines*:

Citation for lifelines
CHAPTER 2

Indices and tables

- genindex
- modindex
- search
lifelines.datasets, 239
lifelines.fitters, 223
lifelines.fitters.aalen_additive_fitter, 139
lifelines.fitters.aalen_johansen_fitter, 142
lifelines.fitters.breslow_fleming_harrington_fitter, 145
lifelines.fitters.cox_time_varying_fitter, 147
lifelines.fitters.coxph_fitter, 150
lifelines.fitters.exponential_fitter, 157
lifelines.fitters.kaplan_meier_fitter, 163
lifelines.fitters.log_logistic_aft_fitter, 209
lifelines.fitters.log_logistic_fitter, 168
lifelines.fitters.log_normal_aft_fitter, 201
lifelines.fitters.log_normal_fitter, 174
lifelines.fitters.nelson_aalen_fitter, 179
lifelines.fitters.piecewise_exponential_fitter, 182
lifelines.fitters.piecewise_exponential_regression_fitter, 217
lifelines.fitters.weibull_aft_fitter, 193
lifelines.fitters.weibull_fitter, 188
lifelines.plotting, 237
lifelines.statistics, 231
lifelines.utils, 223
Index

A
AalenAdditiveFitter (class in lifelines.fitters.aalen_additive_fitter), 139
AalenJohansenFitter (class in lifelines.fitters.aalen_johansen_fitter), 142
add_at_risk_counts() (in module lifelines.plotting), 237
add_covariate_to_timeline() (in module lifelines.utils), 229
alpha_ (lifelines.fitters.log_logistic_fitter.LogLogisticFitter attribute), 169

B
baseline_cumulative_hazard_(life-lines.fitters.cox_time_varying_fitter.CoxTimeVaryingFitter attribute), 148
baseline_cumulative_hazard_ (life-lines.fitters.coxph_fitter.CoxPHFitter attribute), 151
baseline_hazard_ (life-lines.fitters.coxph_fitter.CoxPHFitter attribute), 151
baseline_survival_ (life-lines.fitters.cox_time_varying_fitter.CoxTimeVaryingFitter attribute), 148
baseline_survival_ (life-lines.fitters.coxph_fitter.CoxPHFitter attribute), 151
beta_ (lifelines.fitters.log_logistic_fitter.LogLogisticFitter attribute), 169
breakpoints (lifelines.fitters.piecewise_exponential_fitter.PiecewiseExponentialFitter attribute), 183
BreslowFlemingHarringtonFitter (class in lifelines.fitters.breslow_fleming_harrington_fitter), 145

C
cdf_plot() (in module lifelines.plotting), 239
check_assumptions() (life-lines.fitters.coxph_fitter.CoxPHFitter method), 151
compute_residuals() (life-lines.fitters.coxph_fitter.CoxPHFitter method), 152
concordance_index() (in module lifelines.utils), 227
conditional_time_to_event_ (life-lines.fitters.aalen_johansen_fitter.AalenJohansenFitter attribute), 143
conditional_time_to_event_ (life-lines.fitters.breslow_fleming_harrington_fitter.BreslowFlemingHarringtonFitter attribute), 145
conditional_time_to_event_ (life-lines.fitters.exponential_fitter.ExponentialFitter attribute), 159
conditional_time_to_event_ (life-lines.fitters.kaplan_meier_fitter.KaplanMeierFitter attribute), 164
conditional_time_to_event_ (life-lines.fitters.log_logistic_fitter.LogLogisticFitter attribute), 170
conditional_time_to_event_ (life-lines.fitters.log_normal_fitter.LogNormalFitter attribute), 175
conditional_time_to_event_ (life-lines.fitters.nelson_aalen_fitter.NelsonAalenFitter attribute), 180
conditional_time_to_event_ (life-lines.fitters.piecewise_exponential_fitter.PiecewiseExponentialFitter attribute), 184
conditional_time_to_event_ (life-lines.fitters.weibull_fitter.WeibullFitter attribute), 189
confidence_interval_ (life-lines.fitters.exponential_fitter.ExponentialFitter attribute), 159
confidence_interval_ (life-lines.fitters.kaplan_meier_fitter.KaplanMeierFitter attribute)
confidence_interval_
(lifelines.fitters.log_logistic_fitter.LogLogisticFitter attribute), 169
confidence_interval_cumulative_density_
(lifelines.fitters.log_normal_fitter.LogNormalFitter attribute), 174
confidence_interval_cumulative_density_
(lifelines.fitters.piecewise_exponential_fitter.PiecewiseExponentialFitter attribute), 183
confidence_interval_hazard_
(lifelines.fitters.exponential_fitter.ExponentialFitter attribute), 159, 160
confidence_interval_hazard_
(lifelines.fitters.log_logistic_fitter.LogLogisticFitter attribute), 169, 170
confidence_interval_hazard_
(lifelines.fitters.log_normal_fitter.LogNormalFitter attribute), 174, 175
confidence_interval_hazard_
(lifelines.fitters.piecewise_exponential_fitter.PiecewiseExponentialFitter attribute), 183, 184
confidence_interval_survival_function_
(lifelines.fitters.exponential_fitter.ExponentialFitter attribute), 158, 159
confidence_interval_survival_function_
(lifelines.fitters.log_logistic_fitter.LogLogisticFitter attribute), 169, 170
confidence_interval_survival_function_
(lifelines.fitters.log_normal_fitter.LogNormalFitter attribute), 174, 175
confidence_interval_survival_function_
(lifelines.fitters.piecewise_exponential_fitter.PiecewiseExponentialFitter attribute), 183, 184
confidence_intervals_
(lifelines.fitters.aalen_additive_fitter.AalenAdditiveFitter attribute), 139
confidence_intervals_
(lifelines.fitters.cox_time_varying_fitter.CoxTimeVaryingFitter attribute), 147
confidence_intervals_
(lifelines.fitters.coxph_fitter.CoxPHFitter attribute), 151
confidence_intervals_
(lifelines.fitters.log_logistic_fitter.LogLogisticFitter attribute), 159
confidence_intervals_
(lifelines.fitters.log_logistic_aft_fitter.LogLogisticAFTFitter attribute), 164
confidențe_intervals_ (life-lines.fitters.log_normal_aft_fitter.LogNormalAFTFitter
attribute), 209
cumulative_hazard_ (life-lines.fitters.piecewise_exponential_fitter.PiecewiseExponentialFitter
attribute), 179
cumulative_hazard_ (life-lines.fitters.log_logistic_fitter.LogLogisticFitter
attribute), 179
cumulative_hazard_ (life-lines.fitters.log_normal_fitter.LogNormalFitter
attribute), 179
cumulative_hazard_ (life-lines.fitters.piecewise_exponential_regression_fitter.PiecewiseExponentialRegressionFitter
attribute), 217
cumulative_hazard_at_times() (life-lines.fitters.aalen_johansen_fitter.AalenJohansenFitter
method), 143
cumulative_hazard_at_times() (life-lines.fitters.breslow_fleming_harrington_fitter.BreslowFlemingHarringtonFitter
method), 145
cumulative_hazard_at_times() (life-lines.fitters.exponential_fitter.ExponentialFitter
method), 160
cumulative_hazard_at_times() (life-lines.fitters.log_logistic_fitter.LogLogisticFitter
method), 170
cumulative_hazard_at_times() (life-lines.fitters.log_normal_fitter.LogNormalFitter
method), 176
cumulative_hazard_at_times() (life-lines.fitters.piecewise_exponential_fitter.PiecewiseExponentialFitter
method), 184
cumulative_hazard_at_times() (life-lines.fitters.weibull_fitter.WeibullFitter
method), 190
cumulative_hazard_at_times() (life-lines.fitters.aalen_additive_fitter.AalenAdditiveFitter
method), 139

cumulative_density_at_times() (life-lines.fitters.exponential_fitter.ExponentialFitter
method), 159
cumulative_density_at_times() (life-lines.fitters.log_logistic_fitter.LogLogisticFitter
method), 169
cumulative_density_at_times() (life-lines.fitters.log_normal_fitter.LogNormalFitter
method), 174
cumulative_density_at_times() (life-lines.fitters.piecewise_exponential_fitter.PiecewiseExponentialFitter
method), 184
cumulative_density_at_times() (life-lines.fitters.weibull_fitter.WeibullFitter
method), 190

cumulative_density_at_times() (life-lines.fitters.exponential_fitter.ExponentialFitter
method), 158
cumulative_density_at_times() (life-lines.fitters.log_logistic_fitter.LogLogisticFitter
method), 168
cumulative_density_at_times() (life-lines.fitters.log_normal_fitter.LogNormalFitter
method), 174
cumulative_density_at_times() (life-lines.fitters.piecewise_exponential_fitter.PiecewiseExponentialFitter
method), 183

cumulative_density_at_times() (life-lines.fitters.nelson_aalen_fitter.NelsonAalenFitter
method), 180
cumulative_density_at_times() (life-lines.fitters.piecewise_exponential_fitter.PiecewiseExponentialFitter
method), 184

cumulative_hazards_ (life-lines.fitters.nelson_aalen_fitter.NelsonAalenFitter
attribute), 180

cumulative_density_ (life-lines.fitters.exponential_fitter.ExponentialFitter
attribute), 159

cumulative_density_ (life-lines.fitters.kaplan_meier_fitter.KaplanMeierFitter
attribute), 164

cumulative_density_ (life-lines.fitters.log_logistic_fitter.LogLogisticFitter
attribute), 169

cumulative_density_ (life-lines.fitters.log_normal_fitter.LogNormalFitter
attribute), 174

cumulative_density_ (life-lines.fitters.piecewise_exponential_fitter.PiecewiseExponentialFitter
attribute), 183

cumulative_density_ (life-lines.fitters.nelson_aalen_fitter.NelsonAalenFitter
attribute), 180

cumulative_density_ (life-lines.fitters.weibull_fitter.WeibullFitter
attribute), 190
D

datetimes_to_durations() (in module lifelines.utils), 226
divide() (lifelines.fitters.aalen_johansen_fitter.AalenJohansenFitter method), 143
divide() (lifelines.fitters.breslow_fleming_harrington_fitter.BreslowFlemingHarringtonFitter method), 145
divide() (lifelines.fitters.exponential_fitter.ExponentialFitter method), 160
divide() (lifelines.fitters.kaplan_meier_fitter.KaplanMeierFitter method), 164
divide() (lifelines.fitters.log_logistic_fitter.LogLogisticFitter method), 170
divide() (lifelines.fitters.log_normal_fitter.LogNormalFitter method), 176
divide() (lifelines.fitters.nelson_aalen_fitter.NelsonAalenFitter method), 180
divide() (lifelines.fitters.piecewise_exponential_fitter.PiecewiseExponentialFitter method), 184
divide() (lifelines.fitters.weibull_fitter.WeibullFitter method), 190
durations (lifelines.fitters.aalen_additive_fitter.AalenAdditiveFitter attribute), 139
durations (lifelines.fitters.coxph_fitter.CoxPHFitter attribute), 151
durations (lifelines.fitters.exponential_fitter.ExponentialFitter attribute), 159
durations (lifelines.fitters.kaplan_meier_fitter.KaplanMeierFitter attribute), 164
durations (lifelines.fitters.log_logistic_aft_fitter.LogLogisticAFTFitter attribute), 210
durations (lifelines.fitters.log_normal_aft_fitter.LogNormalAFTFitter attribute), 175

event_observed (lifelines.fitters.aalen_additive_fitter.AalenAdditiveFitter attribute), 139
event_observed (lifelines.fitters.exponential_fitter.ExponentialFitter attribute), 159
event_observed (lifelines.fitters.kaplan_meier_fitter.KaplanMeierFitter attribute), 164
event_observed (lifelines.fitters.log_logistic_fitter.LogLogisticFitter attribute), 169
event_observed (lifelines.fitters.log_normal_fitter.LogNormalFitter attribute), 175

event_observed (lifelines.fitters.nelson_aalen_fitter.NelsonAalenFitter attribute), 179

event_observed (lifelines.fitters.piecewise_exponential_fitter.PiecewiseExponentialFitter attribute), 183

event_observed (lifelines.fitters.piecewise_exponential_regression_fitter.PiecewiseExponentialRegressionFitter attribute), 183

event_observed (lifelines.fitters.weibull_aft_fitter.WeibullAFTFitter attribute), 218

event_observed (lifelines.fitters.weibull_fitter.WeibullFitter attribute), 194

E

entry (lifelines.fitters.exponential_fitter.ExponentialFitter attribute), 159

entry (lifelines.fitters.kaplan_meier_fitter.KaplanMeierFitter attribute), 164
entry (lifelines.fitters.log_logistic_fitter.LogLogisticFitter attribute), 169
entry (lifelines.fitters.log_normal_fitter.LogNormalFitter attribute), 175
entry (lifelines.fitters.nelson_aalen_fitter.NelsonAalenFitter attribute), 179
entry (lifelines.fitters.piecewise_exponential_fitter.PiecewiseExponentialFitter attribute), 183
entry (lifelines.fitters.piecewise_exponential_regression_fitter.PiecewiseExponentialRegressionFitter attribute), 183
entry (lifelines.fitters.weibull_aft_fitter.WeibullAFTFitter attribute), 194
entry (lifelines.fitters.weibull_fitter.WeibullFitter attribute), 189

Index 274
event_table (lifelines.fitters.kaplan_meier_fitter.KaplanMeierFitter, attribute), 164

fit_interval_censoring () (lifelines.fitters.kaplan_meier_fitter.KaplanMeierFitter, method), 165

ExponentialFitter (class in lifelines.fitters.exponential_fitter), 157

ExponentialFitter (method), 160

fit () (lifelines.fitters.aalen_additive_fitter.AalenAdditiveFitter, method), 140

fit () (lifelines.fitters.aalen_johansen_fitter.AalenJohansenFitter, method), 143

fit () (lifelines.fitters.breslow_fleming_harrington_fitter.BreslowFlemingHarringtonFitter, method), 145

fit () (lifelines.fitters.cox_time_varying_fitter.CoxTimeVaryingFitter, method), 149

fit () (lifelines.fitters.coxph_fitter.CoxPHFitter, method), 154

fit () (lifelines.fitters.exponential_fitter.ExponentialFitter, method), 157

fit () (lifelines.fitters.log_logistic_aft_fitter.LogLogisticAFTFitter, method), 202

fit () (lifelines.fitters.log_logistic_fitter.LogLogisticFitter, method), 170

fit () (lifelines.fitters.log_normal_aft_fitter.LogNormalAFTFitter, method), 196

fit () (lifelines.fitters.log_normal_fitter.LogNormalFitter, method), 176

fit () (lifelines.fitters.piecewise_exponential_fitter.PiecewiseExponentialFitter, method), 218

fit () (lifelines.fitters.piecewise_exponential_regression_fitter.PiecewiseExponentialRegressionFitter, method), 204

fit () (lifelines.fitters.weibull_aft_fitter.WeibullAFTFitter, method), 194

fit () (lifelines.fitters.weibull_fitter.WeibullFitter, method), 190

fit_interval_censoring () (lifelines.fitters.exponential_fitter.ExponentialFitter, method), 160

fit_interval_censoring () (lifelines.fitters.log_logistic_aft_fitter.LogLogisticAFTFitter, method), 204

fit_interval_censoring () (lifelines.fitters.log_normal_aft_fitter.LogNormalAFTFitter, method), 196

fit_interval_censoring () (lifelines.fitters.log_normal_fitter.LogNormalFitter, method), 176

fit_interval_censoring () (lifelines.fitters.piecewise_exponential_fitter.PiecewiseExponentialFitter, method), 185

fit_interval_censoring () (lifelines.fitters.piecewise_exponential_regression_fitter.PiecewiseExponentialRegressionFitter, method), 218

fit_interval_censoring () (lifelines.fitters.weibull_aft_fitter.WeibullAFTFitter, method), 194

fit_interval_censoring () (lifelines.fitters.weibull_fitter.WeibullFitter, method), 190

fit_interval_censoring () (lifelines.fitters.exponential_fitter.ExponentialFitter, method), 164

fit_interval_censoring () (lifelines.fitters.log_logistic_aft_fitter.LogLogisticAFTFitter, method), 202

fit_interval_censoring () (lifelines.fitters.log_normal_aft_fitter.LogNormalAFTFitter, method), 196

Index 275
median_survival_time_ (lifelines.fitters.log_logistic_aft_fitter.LogisticAFTFitter attribute), 214
median_survival_time_ (lifelines.fitters.log_normal_aft_fitter.LogNormalAFTFitter attribute), 206
median_survival_time_ (lifelines.fitters.piecewise_exponential_regression_fitter.PiecewiseExponentialRegressionFitter attribute), 220
median_survival_time_ (lifelines.fitters.weibull_aft_fitter.WeibullAFTFitter attribute), 198
median_survival_times() (in module lifelines.utils), 223
mu_ (lifelines.fitters.log_normal_fitter.LogNormalFitter attribute), 175
multivariate_logrank_test() (in module lifelines.statistics), 233

N
NelsonAalenFitter (class in lifelines.fitters.nelson_aalen_fitter), 179

P
pairwise_logrank_test() (in module lifelines.statistics), 234
params_ (lifelines.fitters.logistic_aft_fitter.LogisticAFTFitter attribute), 209
params_ (lifelines.fitters.log_normal_aft_fitter.LogNormalAFTFitter attribute), 202
params_ (lifelines.fitters.piecewise_exponential_regression_fitter.PiecewiseExponentialRegressionFitter attribute), 217
params_ (lifelines.fitters.weibull_aft_fitter.WeibullAFTFitter attribute), 194
PiecewiseExponentialFitter (class in lifelines.fitters.piecewise_exponential_fitter), 182
PiecewiseExponentialRegressionFitter (class in lifelines.fitters.piecewise_exponential_regression_fitter), 217
plot() (lifelines.fitters.aalen_additive_fitter.AalenAdditiveFitter method), 140
plot() (lifelines.fitters.aalen_johansen_fitter.AalenJohansenFitter method), 143
plot() (lifelines.fitters.breslow_fleming_harrington_fitter.BreslowFlemingHarringtonFitter method), 146
plot() (lifelines.fitters.cox_time_varying_fitter.CoxTimeVaryingFitter method), 149
plot() (lifelines.fitters.coxph_fitter.CoxPHFitter method), 154
plot() (lifelines.fitters.exponential_fitter.ExponentialFitter method), 162
plot() (lifelines.fitters.kaplan_meier_fitter.KaplanMeierFitter method), 166
plot_cumulative_density() (lifelines.fitters.weibull_fitter.WeibullFitter method), 192
plot_cumulative_hazard() (lifelines.fitters.aalen_johansen_fitter.AalenJohansenFitter method), 144
plot_cumulative_hazard() (lifelines.fitters.breslow_fleming_harrington_fitter.BreslowFlemingHarringtonFitter method), 146
plot_cumulative_hazard() (lifelines.fitters.exponential_fitter.ExponentialFitter method), 162
plot_cumulative_hazard() (lifelines.fitters.kaplan_meier_fitter.KaplanMeierFitter method), 167
plot_cumulative_hazard() (lifelines.fitters.log_logistic_fitter.LogLogisticFitter method), 173
plot_cumulative_hazard() (lifelines.fitters.log_normal_fitter.LogNormalFitter method), 178
plot_cumulative_hazard() (lifelines.fitters.nelson_aalen_fitter.NelsonAalenFitter method), 181
plot_cumulative_hazard() (lifelines.fitters.piecewise_exponential_fitter.PiecewiseExponentialFitter method), 187
plot_cumulative_hazard() (lifelines.fitters.weibull_fitter.WeibullFitter method), 192
plot_logloss() (lifelines.fitters.kaplan_meier_fitter.KaplanMeierFitter method), 146
plot_cumulative_hazard() (lifelines.fitters.exponential_fitter.ExponentialFitter method), 162
plot_cumulative_hazard() (lifelines.fitters.kaplan_meier_fitter.KaplanMeierFitter method), 167
plot_cumulative_hazard() (lifelines.fitters.log_logistic_fitter.LogLogisticFitter method), 173
plot_cumulative_hazard() (lifelines.fitters.log_normal_fitter.LogNormalFitter method), 178
plot_cumulative_hazard() (lifelines.fitters.nelson_aalen_fitter.NelsonAalenFitter method), 181
plot_cumulative_hazard() (lifelines.fitters.piecewise_exponential_fitter.PiecewiseExponentialFitter method), 187
plot_cumulative_hazard() (lifelines.fitters.weibull_fitter.WeibullFitter method), 192
plot_hazard() (lifelines.fitters.aalen_johansen_fitter.AalenJohansenFitter method), 144
plot_hazard() (lifelines.fitters.breslow_fleming_harrington_fitter.BreslowFlemingHarringtonFitter method), 146
plot_hazard() (lifelines.fitters.exponential_fitter.ExponentialFitter method), 162
plot_hazard() (lifelines.fitters.kaplan_meier_fitter.KaplanMeierFitter method), 167
plot_hazard() (lifelines.fitters.log_logistic_fitter.LogLogisticFitter method), 173
plot_hazard() (lifelines.fitters.log_normal_fitter.LogNormalFitter method), 178
plot_hazard() (lifelines.fitters.nelson_aalen_fitter.NelsonAalenFitter method), 181
plot_hazard() (lifelines.fitters.piecewise_exponential_fitter.PiecewiseExponentialFitter method), 187
plot_hazard() (lifelines.fitters.weibull_fitter.WeibullFitter method), 192
plot_lifetimes() (in module lifelines.plotting), 238
plot_loglogs() (lifelines.fitters.kaplan_meier_fitter.KaplanMeierFitter method), 167
plot_cumulative_hazard() (lifelines.fitters.exponential_fitter.ExponentialFitter method), 162
plot_cumulative_hazard() (lifelines.fitters.kaplan_meier_fitter.KaplanMeierFitter method), 167
plot_cumulative_hazard() (lifelines.fitters.log_logistic_fitter.LogLogisticFitter method), 173
plot_cumulative_hazard() (lifelines.fitters.log_normal_fitter.LogNormalFitter method), 178
plot_cumulative_hazard() (lifelines.fitters.nelson_aalen_fitter.NelsonAalenFitter method), 181
plot_cumulative_hazard() (lifelines.fitters.piecewise_exponential_fitter.PiecewiseExponentialFitter method), 187
plot_cumulative_hazard() (lifelines.fitters.weibull_fitter.WeibullFitter method), 192
plot_survival_function() (lifelines.fitters.aalen_johansen_fitter.AalenJohansenFitter method), 144
plot_survival_function() (lifelines.fitters.breslow_fleming_harrington_fitter.BreslowFlemingHarringtonFitter method), 146
plot_survival_function() (lifelines.fitters.exponential_fitter.ExponentialFitter method), 162
plot_survival_function() (lifelines.fitters.kaplan_meier_fitter.KaplanMeierFitter method), 167
plot_survival_function() (lifelines.fitters.log_logistic_fitter.LogLogisticFitter method), 173
plot_survival_function() (lifelines.fitters.log_normal_fitter.LogNormalFitter method), 178
plot_survival_function() (lifelines.fitters.nelson_aalen_fitter.NelsonAalenFitter method), 181
plot_survival_function() (lifelines.fitters.piecewise_exponential_fitter.PiecewiseExponentialFitter method), 187
plot_survival_function() (lifelines.fitters.weibull_fitter.WeibullFitter method), 192
power_under_cph() (in module lifelines.statistics), 236
predict() (lifelines.fitters.aalen_johansen_fitter.AalenJohansenFitter method), 144
predict() (lifelines.fitters.breslow_fleming_harrington_fitter.BreslowFlemingHarringtonFitter method), 146
predict() (lifelines.fitters.exponential_fitter.ExponentialFitter method), 162
predict() (lifelines.fitters.kaplan_meier_fitter.KaplanMeierFitter method), 167
predict() (lifelines.fitters.log_logistic_fitter.LogLogisticFitter method), 173
predict() (lifelines.fitters.log_normal_fitter.LogNormalFitter method), 178
predict() (lifelines.fitters.nelson_aalen_fitter.NelsonAalenFitter method), 181
predict() (lifelines.fitters.piecewise_exponential_fitter.PiecewiseExponentialFitter method), 187
predict() (lifelines.fitters.weibull_fitter.WeibullFitter method), 192
predict_cumulative_hazard() (lifelines.fitters.aalen_additive_fitter.AalenAdditiveFitter method), 192
predict_median() (lifelines.fitters.piecewise_exponential_regression_fitter.PiecewiseExponentialRegressionFitter method), 207
predict_cumulative_hazard() (lifelines.fitters.coxph_fitter.CoxPHFitter method), 141
predict_median() (lifelines.fitters.piecewise_exponential_regression_fitter.PiecewiseExponentialRegressionFitter method), 221
predict_cumulative_hazard() (lifelines.fitters.cox_time_varying_fitter.CoxTimeVaryingFitter method), 149
predict_median() (lifelines.fitters.piecewise_exponential_regression_fitter.PiecewiseExponentialRegressionFitter method), 222
predict_partial_hazard() (lifelines.fitters.cox_time_varying_fitter.CoxTimeVaryingFitter method), 149
predict_percentile() (lifelines.fitters.piecewise_exponential_regression_fitter.PiecewiseExponentialRegressionFitter method), 222
predict_survival_function() (lifelines.fitters.piecewise_exponential_regression_fitter.PiecewiseExponentialRegressionFitter method), 222
print_summary() (lifelines.fitters.aalen_additive_fitter.AalenAdditiveFitter method), 141
print_summary() (lifelines.fitters.piecewise_exponential_regression_fitter.PiecewiseExponentialRegressionFitter method), 141
print_summary() (lifelines.fitters.cox_time_varying_fitter.CoxTimeVaryingFitter method), 149
print_summary() (lifelines.fitters.piecewise_exponential_regression_fitter.PiecewiseExponentialRegressionFitter method), 141
predict_expectation() (lifelines.fitters.aalen_additive_fitter.AalenAdditiveFitter method), 141
predict_expectation() (lifelines.fitters.cox_time_varying_fitter.CoxTimeVaryingFitter method), 155
predict_expectation() (lifelines.fitters.coxph_fitter.CoxPHFitter method), 156
predict_expectation() (lifelines.fitters.log_logistic_aft_fitter.LogLogisticAFTFitter method), 199
predict_expectation() (lifelines.fitters.log_normal_aft_fitter.LogNormalAFTFitter method), 207
predict_expectation() (lifelines.fitters.piecewise_exponential_regression_fitter.PiecewiseExponentialRegressionFitter method), 221
predict_expectation() (lifelines.fitters.weibull_aft_fitter.WeibullAFTFitter method), 199
predict_expectation() (lifelines.fitters.log_logistic_aft_fitter.LogLogisticAFTFitter method), 215
predict_expectation() (lifelines.fitters.log_normal_aft_fitter.LogNormalAFTFitter method), 208
predict_expectation() (lifelines.fitters.piecewise_exponential_regression_fitter.PiecewiseExponentialRegressionFitter method), 222
predict_expectation() (lifelines.fitters.weibull_aft_fitter.WeibullAFTFitter method), 200
predict_percentile() (lifelines.fitters.aalen_additive_fitter.AalenAdditiveFitter method), 141
predict_percentile() (lifelines.fitters.cox_time_varying_fitter.CoxTimeVaryingFitter method), 149
predict_percentile() (lifelines.fitters.coxph_fitter.CoxPHFitter method), 156
predict_percentile() (lifelines.fitters.log_logistic_aft_fitter.LogLogisticAFTFitter method), 216
predict_percentile() (lifelines.fitters.log_normal_aft_fitter.LogNormalAFTFitter method), 208
predict_percentile() (lifelines.fitters.piecewise_exponential_regression_fitter.PiecewiseExponentialRegressionFitter method), 222
predict_percentile() (lifelines.fitters.weibull_aft_fitter.WeibullAFTFitter method), 200
predict_survival_function() (lifelines.fitters.aalen_additive_fitter.AalenAdditiveFitter method), 141
predict_survival_function() (lifelines.fitters.cox_time_varying_fitter.CoxTimeVaryingFitter method), 149
predict_survival_function() (lifelines.fitters.coxph_fitter.CoxPHFitter method), 156
predict_survival_function() (lifelines.fitters.log_logistic_aft_fitter.LogLogisticAFTFitter method), 216
predict_survival_function() (lifelines.fitters.log_normal_aft_fitter.LogNormalAFTFitter method), 208
predict_survival_function() (lifelines.fitters.piecewise_exponential_regression_fitter.PiecewiseExponentialRegressionFitter method), 222
predict_survival_function() (lifelines.fitters.weibull_aft_fitter.WeibullAFTFitter method), 200
predict_log_partial_hazard() (lifelines.fitters.cox_time_varying_fitter.CoxTimeVaryingFitter method), 149
predict_log_partial_hazard() (lifelines.fitters.coxph_fitter.CoxPHFitter method), 156
predict_log_partial_hazard() (lifelines.fitters.log_logistic_aft_fitter.LogLogisticAFTFitter method), 216
predict_log_partial_hazard() (lifelines.fitters.log_normal_aft_fitter.LogNormalAFTFitter method), 208
predict_log_partial_hazard() (lifelines.fitters.piecewise_exponential_regression_fitter.PiecewiseExponentialRegressionFitter method), 222
predict_log_partial_hazard() (lifelines.fitters.weibull_aft_fitter.WeibullAFTFitter method), 200
predict_median() (lifelines.fitters.aalen_additive_fitter.AalenAdditiveFitter method), 141
predict_median() (lifelines.fitters.cox_time_varying_fitter.CoxTimeVaryingFitter method), 149
predict_median() (lifelines.fitters.coxph_fitter.CoxPHFitter method), 156
predict_median() (lifelines.fitters.log_logistic_aft_fitter.LogLogisticAFTFitter method), 216
predict_median() (lifelines.fitters.log_normal_aft_fitter.LogNormalAFTFitter method), 208
predict_median() (lifelines.fitters.piecewise_exponential_regression_fitter.PiecewiseExponentialRegressionFitter method), 222
predict_median() (lifelines.fitters.weibull_aft_fitter.WeibullAFTFitter method), 200
V

variance_matrix_ (lifelines.fitters.cox_time_varying_fitter.CoxTimeVaryingFitter attribute), 147
variance_matrix_ (lifelines.fitters.coxph_fitter.CoxPHFitter attribute), 151
variance_matrix_ (lifelines.fitters.exponential_fitter.ExponentialFitter attribute), 158
variance_matrix_ (lifelines.fitters.log_logistic_aft_fitter.LogLogisticAFTFitter attribute), 210
variance_matrix_ (lifelines.fitters.log_logistic_fitter.LogLogisticFitter attribute), 169
variance_matrix_ (lifelines.fitters.log_normal_aft_fitter.LogNormalAFTFitter attribute), 202
variance_matrix_ (lifelines.fitters.log_normal_fitter.LogNormalFitter attribute), 174
variance_matrix_ (lifelines.fitters.piecewise_exponential_fitter.PiecewiseExponentialFitter attribute), 183
variance_matrix_ (lifelines.fitters.piecewise_exponential_regression_fitter.PiecewiseExponentialRegressionFitter attribute), 218
variance_matrix_ (lifelines.fitters.weibull_aft_fitter.WeibullAFTFitter attribute), 194
variance_matrix_ (lifelines.fitters.weibull_fitter.WeibullFitter attribute), 189

W

WeibullAFTFitter (class in lifelines.fitters.weibull_aft_fitter), 193
WeibullFitter (class in lifelines.fitters.weibull_fitter), 188
weights (lifelines.fitters.aalen_additive_fitter.AalenAdditiveFitter attribute), 140
weights (lifelines.fitters.cox_time_varying_fitter.CoxTimeVaryingFitter attribute), 147
weights (lifelines.fitters.coxph_fitter.CoxPHFitter attribute), 151
weights (lifelines.fitters.log_logistic_aft_fitter.LogLogisticAFTFitter attribute), 210
weights (lifelines.fitters.log_normal_aft_fitter.LogNormalAFTFitter attribute), 202
weights (lifelines.fitters.piecewise_exponential_regression_fitter.PiecewiseExponentialRegressionFitter attribute), 218
weights (lifelines.fitters.weibull_aft_fitter.WeibullAFTFitter attribute), 194