Quickstart Intro

1  Contents: 3
2  Indices and tables 319
Python Module Index 321
Index 323
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
```

OR

Install via conda:

```
conda install -c conda-forge 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.
from lifelines.datasets import load_waltons
df = load_waltons()  # returns a Pandas DataFrame

print(df.head())

***
T   E   group
0 6   1   miR-137
1 13  1   miR-137
2 13  1   miR-137
3 13  1   miR-137
4 19  1   miR-137
***

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). We will fit a Kaplan Meier model to this, implemented as KaplanMeierFitter:

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.

kmf.survival_function_
kmf.cumulative_density_
kmf.plot_survival_function()  # or just kmf.plot()

Alternatively, you can plot the cumulative density function:
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)
```

A useful summary stat is the median survival time, which represents when 50% of the population has died:

```python
from lifelines.utils import median_survival_times
median_ = kmf.median_survival_time_
median_confidence_interval_ = median_survival_times(kmf.confidence_interval_)
```

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
import matplotlib.pyplot as plt
import numpy as np
from lifelines import *
fig, axes = plt.subplots(3, 3, figsize=(13.5, 7.5))
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')
ggf = GeneralizedGammaFitter().fit(T, E, label='GeneralizedGammaFitter')
```
sf = SplineFitter(np.percentile(T.loc[E.astype(bool)], [0, 50, 100])).fit(T, E, label='SplineFitter')

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])
ggf.plot_survival_function(ax=axes[2][0])
sf.plot_survival_function(ax=axes[2][1])
Multiple groups

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

(continues on next page)
Alternatively, for many more groups and more “pandas-esque”:

```python
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)
```

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 *lifelines.utils.datetimes_to_durations()*.

```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 function *lifelines.utils.survival_table_from_events()* will help with that:

```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        163   163
6     1      1      0        163
7     2      1      1        162
9     3      3      0        160
13    3      3      0        157
```

### 1.1.4 Survival regression

While the above *KaplanMeierFitter* model is useful, it only gives us an “average” view of the population. Often we have specific data at the individual level that we would like to use. For this, we turn to survival regression.

**Note:** More detailed documentation and tutorials are available in Survival Regression.

```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 model is different. All the data, including durations, censored indicators and covariates must be contained in a Pandas DataFrame. The duration column and event occurred...
column are 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()
```

```plaintext
<lifelines.CoxPHFitter: fitted with 200 observations, 11 censored>
  duration col = 'T'
  event col = 'E'
  number of subjects = 200
  number of events = 189
  partial log-likelihood = -807.62
  time fit was run = 2019-07-31 10:22:07 UTC

---

<table>
<thead>
<tr>
<th></th>
<th>coef</th>
<th>exp(coef)</th>
<th>se(coef)</th>
<th>coef lower 95%</th>
<th>coef upper 95%</th>
<th>exp(coef) lower 95%</th>
<th>exp(coef) upper 95%</th>
</tr>
</thead>
<tbody>
<tr>
<td>var1</td>
<td>0.22</td>
<td>1.25</td>
<td>0.07</td>
<td>0.08</td>
<td>0.37</td>
<td>1.08</td>
<td>1.44</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>var2</td>
<td>0.05</td>
<td>1.05</td>
<td>0.08</td>
<td>-0.11</td>
<td>0.21</td>
<td>0.89</td>
<td>1.24</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>var3</td>
<td>0.22</td>
<td>1.24</td>
<td>0.08</td>
<td>0.07</td>
<td>0.37</td>
<td>1.07</td>
<td>1.44</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

z    p    -log2(p)
| var1 | 2.99 | <0.005   | 8.49       |
| var2 | 0.61 | 0.54     | 0.89       |
| var3 | 2.88 | <0.005   | 7.97       |

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

```
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')
wft.print_summary()
```

```
<lifelines.WeibullAFTFitter: fitted with 200 observations, 11 censored>
  event col = 'E'
number of subjects = 200
number of events = 189
  log-likelihood = -504.48
time fit was run = 2019-07-31 10:19:07 UTC

---

<table>
<thead>
<tr>
<th></th>
<th>coef</th>
<th>exp(coef)</th>
<th>se(coef)</th>
<th>coef lower 95%</th>
<th>coef upper 95%</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \lambda )</td>
<td>-0.08</td>
<td>0.92</td>
<td>0.02</td>
<td>-0.13</td>
<td>0.04</td>
</tr>
<tr>
<td>var1</td>
<td>0.88</td>
<td>0.97</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>var2</td>
<td>-0.02</td>
<td>0.98</td>
<td>0.03</td>
<td>-0.07</td>
<td>0.04</td>
</tr>
<tr>
<td>var3</td>
<td>0.93</td>
<td>1.04</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>( \lambda )</td>
<td>-0.08</td>
<td>0.92</td>
<td>0.02</td>
<td>-0.13</td>
<td>0.03</td>
</tr>
<tr>
<td>_intercept</td>
<td>2.53</td>
<td>12.57</td>
<td>0.05</td>
<td>2.43</td>
<td>2.63</td>
</tr>
<tr>
<td>( \rho )</td>
<td>11.41</td>
<td>13.85</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>_intercept</td>
<td>1.09</td>
<td>2.98</td>
<td>0.05</td>
<td>0.99</td>
<td>1.20</td>
</tr>
<tr>
<td></td>
<td>2.68</td>
<td>3.32</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

(continues on next page)
<table>
<thead>
<tr>
<th></th>
<th>z</th>
<th>p</th>
<th>-log2(p)</th>
</tr>
</thead>
<tbody>
<tr>
<td>lambda_var1</td>
<td>-3.45</td>
<td>&lt;0.005</td>
<td>10.78</td>
</tr>
<tr>
<td>var2</td>
<td>-0.56</td>
<td>0.57</td>
<td>0.80</td>
</tr>
<tr>
<td>var3</td>
<td>-3.33</td>
<td>&lt;0.005</td>
<td>10.15</td>
</tr>
<tr>
<td>_intercept</td>
<td>51.12</td>
<td>&lt;0.005</td>
<td>inf</td>
</tr>
<tr>
<td>rho_ _intercept</td>
<td>20.12</td>
<td>&lt;0.005</td>
<td>296.66</td>
</tr>
</tbody>
</table>

---

Concordance = 0.58

Log-likelihood ratio test = 19.73 on 3 df, -log2(p)=12.34

```python
wft.plot()
```

Other AFT models are available as well, see here. An alternative regression model is 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:
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:

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 professional’s case). Traditionally, sort of a morbid subject.

But survival analysis can be applied to not only 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 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 making decisions 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 don’t know this distinction beforehand. At $t = 10$, we wish to investigate the average lifespan for the entire population.

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.

```
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_title("Births and deaths of our population, at $t=10$")
print("Observed lifetimes at time \$t=%d\$ \% (CURRENT_TIME), observed_lifetimes)
Furthermore, if we instead simply took the mean of all 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. However, even in the case where all events have been observed, i.e. there is no censoring, survival analysis is still a very useful tool to understand durations and rates.

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 three fundamental objects in survival analysis, the survival function, hazard function and the cumulative 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

$$S(t) = Pr(T > t)$$

Simply, 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 $t$.

Here’s an example of a survival function:

![Example survival function](image)

1.2.4 Hazard function

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:

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

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:

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

The integral has a more common name: the *cumulative hazard function*, denoted \( H(t) \). We can rewrite the above as:

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

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 cumulative hazard function, \( H(t) \), and we can convert back and forth quite easily.

---

**Fig. 1:** Map of the mathematical entities used in the survival analysis and the transforms between them. Don’t panic: *lifelines* does this all for you.

The two figures below represent the hazard and the cumulative hazard of the survival function in the figure above.
1.2.5 Next steps

Of course, we do not observe the true survival function of a population. We must use the observed data to estimate it. There are many ways to estimate the survival function and the hazard functions, which brings us to estimation using lifelines.

1.3 Estimating univariate models

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

$$\hat{S}(t) = \prod_{t_i \leq t} \frac{n_i - d_i}{n_i}$$

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()
```

<table>
<thead>
<tr>
<th>democracy</th>
<th>regime</th>
<th>start_year</th>
<th>observation</th>
<th>countryname</th>
<th>cowcode</th>
<th>politycode</th>
<th>un_region_name</th>
<th>un_continent_name</th>
<th>leadeर spellreg</th>
</tr>
</thead>
</table>

From the lifelines library, we’ll need the KaplanMeierFitter for this exercise:

```python
from lifelines import KaplanMeierFitter
df = 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 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, 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.) 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 50% of the population has expired, is a property:

```python
kmf.median_survival_time_
# 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! To get the confidence interval of the median, you can use:

```python
from lifelines.utils import median_survival_times
median_ci = median_survival_times(kmf.confidence_interval_)
```

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)
```

(continues on next page)
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");
```

![Lifespans of different global regimes](image)
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 functions. Below we demonstrate this routine. The function lifelines.statistics.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()

```
\begin{verbatim}
t_0 = -1
null_distribution = chi squared
degrees_of_freedom = 1
    alpha = 0.99

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

"""
\end{verbatim}

There are alternative (and sometimes better) tests of survival functions, and we explain more here: Statistically compare two populations

Let’s compare the different types of regimes present in the dataset:

```python
define

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')
```

plt.tight_layout()
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:

```python
T = [0,3,3,2,1,2]
E = [1,1,0,0,1,1]
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 0:00:00', '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 (durations): [ 3. 1. 5.]
E (event_observed): [ 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 functions 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
naf = NelsonAalenFitter()
naf.fit(T, event_observed=E)
```

After fitting, the class exposes the property `cumulative_hazard_`() as a DataFrame:
The cumulative hazard has less obvious understanding than the survival functions, but the hazard functions is the basis of more advanced techniques in survival analysis. Recall that we are estimating cumulative hazard functions, $H(t)$. (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.
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 function**

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 function, but there is a catch. The derivation involves a kernel smoother (to smooth out the differences of the cumulative hazard function), and this requires us to specify a bandwidth parameter that controls the amount of smoothing. This functionality is in the `smoothed_hazard()` and `smoothed_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.

```
bandwidth = 3.
```

(continues on next page)
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);
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 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
```
Other parametric models: Exponential, Log-Logistic, Log-Normal and Splines

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 non-parametric model, of the same data.

```python
from lifelines import (WeibullFitter, ExponentialFitter,
LogNormalFitter, LogLogisticFitter, NelsonAalenFitter,
PiecewiseExponentialFitter, GeneralizedGammaFitter, SplineFitter)

from lifelines.datasets import load_waltons
data = load_waltons()

fig, axes = plt.subplots(3, 3, figsize=(10, 7.5))

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')
```

(continues on next page)
llf = LogLogisticFitter().fit(T, E, label='LogLogisticFitter')
pwf = PiecewiseExponentialFitter([40, 60]).fit(T, E, label='PiecewiseExponentialFitter')
gg = GeneralizedGammaFitter().fit(T, E, label='GeneralizedGammaFitter')
spf = SplineFitter([6, 20, 40, 75]).fit(T, E, label='SplineFitter')

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])
gg.plot_cumulative_hazard(ax=axes[2][0])
spf.plot_cumulative_hazard(ax=axes[2][1])
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:
from lifelines import KaplanMeierFitter

fig, axes = plt.subplots(3, 3, figsize=(10, 7.5))

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')
gg = GeneralizedGammaFitter().fit(T, E, label='GeneralizedGammaFitter')
spf = SplineFitter([6, 20, 40, 75]).fit(T, E, label='SplineFitter')

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])
gg.plot_survival_function(ax=axes[2][0])
spf.plot_survival_function(ax=axes[2][1])
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)
```

(continues on next page)
Model Selection

When the underlying data generation distribution is unknown, we resort to measures of fit to tell us which model is most appropriate. *lifelines* has provided qq-plots, Selecting a parametric model using QQ plots, and also tools to compare AIC and other measures: Selecting a parametric model using AIC.

**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
from lifelines.datasets import load_nh4
df = load_nh4()[['NH4.Orig.mg.per.L', 'NH4.mg.per.L', 'Censored']]
print(df.head())

""
<table>
<thead>
<tr>
<th>NH4.Orig.mg.per.L</th>
<th>NH4.mg.per.L</th>
<th>Censored</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;0.006</td>
<td>0.006</td>
<td>True</td>
</tr>
<tr>
<td>&lt;0.006</td>
<td>0.006</td>
<td>True</td>
</tr>
<tr>
<td>0.006</td>
<td>0.006</td>
<td>False</td>
</tr>
<tr>
<td>0.016</td>
<td>0.016</td>
<td>False</td>
</tr>
<tr>
<td>&lt;0.006</td>
<td>0.006</td>
<td>True</td>
</tr>
</tbody>
</table>
""
```

```
lifelines has support for left-censored datasets in most univariate models, including the KaplanMeierFitter class, by using the lifelines.fitters.kaplan_meier_fitter.KaplanMeierFitter. fit_left_censoring() method.

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 lifelines.fitters.kaplan_meier_fitter.KaplanMeierFitter. 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")

""
<table>
<thead>
<tr>
<th>KM_estimate</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.000</td>
</tr>
<tr>
<td>0.006</td>
</tr>
<tr>
<td>0.007</td>
</tr>
<tr>
<td>0.008</td>
</tr>
<tr>
<td>0.009</td>
</tr>
</tbody>
</table>
""
```
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 and Selecting a parametric model using AIC.

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

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])

kmf.plot_cumulative_density(ax=axes[1][0], ci_show=False)
lnf.plot_cumulative_density(ax=axes[1][0], ci_show=False)
```

(continues on next page)
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])
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 `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 `KaplanMeierFitter` and any parametric models, have an optional argument for `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)
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 function 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 function 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 versus if we define the hazard and ask questions about the other two.

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.

```python
[1]: %matplotlib inline
%config InlineBackend.figure_format = 'retina'

from matplotlib import pyplot as plt
import numpy as np
import pandas as pd

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

[2]: from lifelines import ExponentialFitter

fig, ax = plt.subplots(nrows=1, ncols=2, figsize=(10, 4))
epf = ExponentialFitter().fit(T, E)
epf.plot_hazard(ax=ax[0])
epf.plot_cumulative_hazard(ax=ax[1])
ax[0].set_title("hazard"); ax[1].set_title("cumulative_hazard")
epf.print_summary(3)
```
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 Exponential’s lack of fit is very obvious.

```python
[3]: from lifelines import NelsonAalenFitter

ax = epf.plot(figsize=(8,5))

naf = NelsonAalenFitter().fit(T, E)
ax = naf.plot(ax=ax)
plt.legend()

[3]: <matplotlib.legend.Legend at 0x126f5c630>
```

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:

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

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 \( \lambda_i \).

```
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_cumulative_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)
```

We can see a much better fit in this model. A quantitative measure of fit is to compare the log-likelihood between
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(ParametricUnivariateFitter):

    _fitted_parameter_names = ["lambda_

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

We only need to specify the cumulative hazard function because of the 1:1:1 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 like there 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.

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

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. Note the `import autograd.numpy as np` below.

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

class InverseTimeHazardFitter(ParametricUnivariateFitter):

    # 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)
```

(continues on next page)
itf.print_summary()

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

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
class TwoParamInverseTimeHazardFitter(ParametricUnivariateFitter):
    _fitted_parameter_names = ['alpha_', 'beta_']

    _bounds = [(0, None), (75.0001, None)]

    def _cumulative_hazard(self, params, times):
        alpha, beta = params
        return alpha / (beta - times)
```

```
[6]: <matplotlib.legend.Legend at 0x127597198>
```

```
[7]: class TwoParamInverseTimeHazardFitter(ParametricUnivariateFitter):
    _fitted_parameter_names = ['alpha_', 'beta_']
    _bounds = [(0, None), (75.0001, None)]
    def _cumulative_hazard(self, params, times):
        alpha, beta = params
        return alpha / (beta - times)
```
From the output, we see that the value of 76.55 is the suggested asymptote, that is:

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

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.

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

```python
[9]: from lifelines.fitters import ParametricUnivariateFitter
class ThreeParamInverseTimeHazardFitter(ParametricUnivariateFitter):
    _fitted_parameter_names = ['alpha_', 'beta_', 'gamma_']
    _bounds = [(0, None), (75.0001, None), (0, None)]
```
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 statistically 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()
```

1.4. Piecewise exponential models and creating custom models 55
3-parameter Weibull distribution

We can easily extend the built-in Weibull model (lifelines.WeibullFitter) to include a new location parameter:

$$ H(t) = \left( \frac{t - \theta}{\lambda} \right)^\rho $$

(When $\theta = 0$, this is just the 2-parameter case again). In lifelines custom models, this looks like:

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

# I'm shifting this to exaggerate the effect
T_ = T + 10

class ThreeParameterWeibullFitter(ParametricUnivariateFitter):
    _fitted_parameter_names = ['lambda_', 'rho_', 'theta_']
    _bounds = [(0, None), (0, None), (0, T.min()-0.001)]

    def _cumulative_hazard(self, params, times):
        lambda_, rho_, theta_ = params
        return ((times - theta_) / lambda_) ** rho_
```

```python
tpw = ThreeParameterWeibullFitter()
tpw.fit(T_, E)
tpw.print_summary()
at = tpw.plot_cumulative_hazard(figsize=(8,5))
at = NelsonAalenFitter().fit(T_, E).plot(ax=ax, ci_show=False)
```

![Graph showing cumulative hazard with two lines: one for ThreeParameterWeibull_estimate and another for NA_estimate.](image)
Inverse Gaussian distribution

The inverse Gaussian distribution is another popular model for survival analysis. Unlike other models, its 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
from autograd.scipy.stats import norm

class InverseGaussianFitter(ParametricUnivariateFitter):
    _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-np.clip(self._cumulative_density(params, times), 1e-15, 1-1e-15))
```

```python
igf = InverseGaussianFitter()
igf.fit(T, E)
igf.print_summary()
ax = igf.plot_cumulative_hazard(figsize=(8,5))
ax = NelsonAalenFitter().fit(T, E).plot(ax=ax, ci_show=False)

<IPython.core.display.HTML object>
```

1.4. Piecewise exponential models and creating custom models 57
Gompertz

```python
[95]: class GompertzFitter(ParametricUnivariateFitter):
    # this parameterization is slightly different than wikipedia.
    _fitted_parameter_names = ['nu_', 'b_']

    def _cumulative_hazard(self, params, times):
        nu_, b_ = params
        return nu_ * (np.expm1(times * b_))
```

```python
[96]: T, E = waltons['T'], waltons['E']

ggf = GompertzFitter()
ggf.fit(T, E)
ggf.print_summary()
ax = ggf.plot_cumulative_hazard(figsize=(8,5))
ax = NelsonAalenFitter().fit(T, E).plot(ax=ax, ci_show=False)

<IPython.core.display.HTML object>
```

APGW


```python
[97]: class APGW_Fitter(ParametricUnivariateFitter):
    # this parameterization is slightly different than wikipedia.
    _fitted_parameter_names = ['kappa_', 'gamma_', 'phi_']

    def _cumulative_hazard(self, params, t):
        kappa_, phi_, gamma_ = params
        return (kappa_ + 1) / kappa_ * ((1 + ((phi_ * t) ** gamma_) / (kappa_ + 1)) ** (kappa_ -1))
```
Bounded lifetimes using the beta distribution

Maybe your data is bounded between 0 and some (unknown) upperbound M? That is, lifetimes can’t be more than M. Maybe you know M, maybe you don’t.

```python
n = 100
T = 5 * np.random.random(n)**2
T_censor = 10 * np.random.random(n)**2
E = T < T_censor
T_obs = np.minimum(T, T_censor)
```

```python
from autograd_gamma import betainc
class BetaFitter(ParametricUnivariateFitter):
    _fitted_parameter_names = ['alpha_', 'beta_', "m_""]
    _bounds = [(0, None), (0, None), (T.max(), None)]

    def _cumulative_density(self, params, times):
        alpha_, beta_, m_ = params
        return betainc(alpha_, beta_, times / m_)

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

1.4. Piecewise exponential models and creating custom models
beta_fitter = BetaFitter().fit(T_obs, E)
beta_fitter.plot()
beta_fitter.print_summary()

StatisticalWarning: The diagonal of the variance_matrix_ has negative values. This could be a problem with BetaFitter's fit to the data.

It's advisable not to trust the variances reported, and to be suspicious of the fitted parameters too. Perform plots of the cumulative hazard to help understand the latter's bias.

To fix this, try specifying an `initial_point` kwarg in `fit`.

warnings.warn(warning_text, utils.StatisticalWarning)

1.5 Discrete survival models

So far we have only been investigating continuous time survival models, where times can take on any positive value. If we want to consider discrete survival times (for example, over the positive integers), we need to make a small adjustment. With discrete survival models, there is a slightly more complicated relationship between the hazard and cumulative hazard. This is because there are two ways to define the cumulative hazard.

\[
H_1(t) = \sum_i h(t_i)
\]

\[
H_2(t) = -\log(S(t))
\]

We also no longer have the relationship that \( h(t) = \frac{dH_1(t)}{dt} \) since \( t \) is no longer continuous. Instead, depending on which version of the cumulative hazard you choose to use (inference will be the same), we have to redefine the hazard function in lifelines.

\[
h(t) = H_1(t) - H_1(t - 1)
\]
Here is an example of a discrete survival model, that may not look like a survival model at first, where we use a redefined _hazard function.

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

1.6 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, no matter how long we wait.

```python
from lifelines import KaplanMeierFitter
kmf = KaplanMeierFitter().fit(T, E)
```
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)
```

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)
```
1.6.1 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 `ParametricUnivariateFitter` (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 ParametricUnivariateFitter

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

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

```python
uaf = UpperAsymptoteFitter().fit(T, E)
uaf.print_summary(3)
uaf.plot(figsize=(8,4))
```

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.
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.

```
[16]: 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})".format(
    np.exp(-uaf.summary.loc['c_', 'coef']),
    np.exp(-uaf.summary.loc['c_', 'coef upper 95%']),
    np.exp(-uaf.summary.loc['c_', 'coef lower 95%']),
)
}

<IPython.core.display.HTML object>
<IPython.core.display.HTML object>
---
Estimated lower bound: 0.20 (0.10, 0.39)

So we might expect that about 20% will not have the event in this population (but make note of the large CI bounds too!)

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

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("---")
print("Estimated lower bound: {:.2f} ({:.2f}, {:.2f})".format(
    np.exp(-uaf.summary.loc['c_', 'coef']),
    np.exp(-uaf.summary.loc['c_', 'coef upper 95%']),
    np.exp(-uaf.summary.loc['c_', 'coef lower 95%']),
)
}

<IPython.core.display.HTML object>
<IPython.core.display.HTML object>
---
Estimated lower bound: 0.75 (0.70, 0.79)
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 unforeseen times.

```
[13]: from autograd.scipy.stats import norm
     from lifelines.fitters import ParametricUnivariateFitter

class SimpleUpperAsymptoteFitter(ParametricUnivariateFitter):
    _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))

[15]: # 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_', 'coef upper 95%']),
        np.exp(-saf.summary.loc['c_', 'coef lower 95%']),
    )
)
```

1.6. Time-lagged conversion rates and cure models
1.7 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 \( h(t \mid x) \) as a function of \( t \) and some covariates \( x \). We explore these models next.

1.7.1 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 `load_rosi()`. 

from lifelines.datasets import load_rossi

rossi = load_rossi()

"""
  week arrest fin age race wexp mar paro prio
  0   20   1   0  27   1   0   0   1   3
  1   17   1   0  18   1   0   0   1   8
  2   25   1   0  19   0   1   0   1  13
  3   52   0   1  23   1   1   1   1   1
"""

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.

1.7.2 Cox’s proportional hazard model

The idea behind Cox’s proportional hazard 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:

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

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.

Fitting the regression

The implementation of the Cox model in lifelines is under CoxPHFitter. Like R, it has a print_summary() function that prints a tabular view of coefficients and related stats.
To access the coefficients and the baseline hazard directly, you can use `params_` and `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 unmarried 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 may vary with time. The proportional hazard assumption is that relationship is true. That is, hazards can change.
over time, but their ratio between levels remains a constant. Later we will deal with checking this assumption. However, in reality, it’s very common for the hazard ratio to change over the study duration. The hazard ratio then has the interpretation of some sort of weighted average of period-specific hazard ratios. As a result, the hazard ratio may critically depend on the duration of the follow-up.

**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 more information during fitting, there is a `show_progress` parameter in `fit()` function. For further help, see *Problems with convergence in the Cox proportional hazard model*.

After fitting, the value of the maximum log-likelihood this available using `log_likelihood_`. The variance matrix of the coefficients is available under `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 `concordance_index_` or in the `print_summary()` as a measure of predictive accuracy.
- look at the log-likelihood test result in the `print_summary()` or `log_likelihood_ratio_test()`
- 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 use the suite of prediction methods: `predict_partial_hazard()`, `predict_survival_function()`, and others.

```python
x = rossi_dataset
cph.predict_partial_hazard(X)
cph.predict_survival_function(X)
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 answer the following: *What is a subject’s new survival function given I know the subject has lived past time $s$?* 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, *lifelines* has all this math and logic built in when using the `conditional_after` kwarg.

```python
# filter down to just censored subjects to predict remaining survival
censored_subjects = X.loc[~X['arrest'].astype(bool)]
censored_subjects_last_obs = censored_subjects['week']
cph.predict_survival_function(censored_subjects, times=[5., 25., 50.], conditional_after=censored_subjects_last_obs)
cph.predict_median(censored_subjects, conditional_after=censored_subjects_last_obs)
```

### Penalties and sparse regression

It’s possible to add a penalizer term to the Cox regression as well. One can use these to i) stabilize the coefficients, ii) shrink the estimates to 0, iii) encourages a Bayesian interpretation, and iv) create sparse coefficients. Regression models, including the Cox model, include both an L1 and L2 penalty:

\[
\frac{1}{2} \text{penalizer} \left((1 - \text{l1_ratio}) \cdot \|\beta\|_2^2 + \text{l1_ratio} \cdot \|\beta\|_1\right)
\]

Both the `penalizer` and `l1_ratio` are specified in the class creation:

```python
from lifelines import CoxPHFitter
from lifelines.datasets import load_rossi
rossi = load_rossi()
cph = CoxPHFitter(penalizer=0.1, l1_ratio=1.0)  # only sparse solutions
cph.fit(rossi, 'week', 'arrest')
cph.print_summary()
```

### 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')
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 \texttt{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')

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
import lifelines

# example survival object
import pandas as pd
from lifelines import KaplanMeierFitter

cph = KaplanMeierFitter()
cph.fit(df['event'], df['time'])

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
import lifelines

# example survival object
import pandas as pd
from lifelines import KaplanMeierFitter

cph = KaplanMeierFitter()
cph.fit(df['event'], df['time'])

cph.plot_covariate_groups(
    ['d1', 'd2', 'd3', 'd4', 'd5'],
```

(continues on next page)
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

To make proper inferences, we should ask if our Cox model is appropriate for our dataset. Recall from above that when using the Cox model, we are implicitly applying the proportional hazard assumption. We should ask, does our dataset obey this 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](#). Suggestions are to look for ways to `stratify` a column (see docs below), or use a time varying model.

**Note:** Checking assumptions like this is only necessary if your goal is inference or correlation. That is, you wish to understand the influence of a covariate on the survival duration & outcome. If your goal is prediction, checking model assumptions is less important since your goal is to maximize an accuracy metric, and not learn about how the model is making that prediction.

### 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()
cph.fit(rossi_dataset, 'week', event_col='arrest', strata=['race'])
cph.print_summary()  # access the results using cph.summary
```

```
<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
```

(continues on next page)
### 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 `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

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, 1, 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(df, '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.

**Baseline hazard and survival**

Normally, the Cox model is *semi-parametric*, which means that its baseline hazard, $h_0(t)$, has no functional form. This is the default for *lifelines*. However, it is sometimes valuable to produce a parametric baseline instead. There is an option to create a parametric baseline with splines:

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

rossi_dataset = load_rossi()

cph = CoxPHFitter(baseline_estimation_method="splines")
cph.fit(rossi_dataset, 'week', event_col='arrest')
```

To access the baseline hazard and baseline survival, one can use `baseline_hazard_` and `baseline_survival_` respectively.

### 1.7.3 Parametric survival models

#### 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 Weibull AFT model is implemented under `WeibullAFTFitter`. The API for the class 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

```

---

<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></td>
<td></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 additional 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
print(aft.median_survival_time_)
print(aft.mean_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 an equal to, or a subset of, the original dataset used 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)
```

```
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)

```python
aft.print_summary()
```

```python
print("")
print("<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
  ---
  coef   exp(coef)  se(coef)  z     p  -log2(p)  lower 0.95  upper 0.95
  lambda_ fin  0.24   1.28    0.15  1.60  0.11  3.18    -0.06  0.55
  age        0.10   1.10    0.03  3.43 <0.005 10.69    0.04  0.16
  race       0.07   1.07    0.19  0.36  0.72  0.48    -0.30  0.04
  wexp      -0.34   0.71    0.15 -2.22  0.03  5.26    -0.64  0.04
  mar       -0.04   0.96    0.25 -0.16  0.86  0.44    -0.30  0.30
  paro      -0.08   0.92    0.15 -4.24 <0.005 15.46    -0.12  0.12
  prio      -0.04   0.96    0.02 -4.24 <0.005 15.46    -0.12  0.12
  _intercept  2.68  14.65    0.60  4.50 <0.005 17.14    0.68  2.29
  rho_ fin   -0.01   0.99    0.02 -4.24 <0.005 15.46    -0.12  0.12
  age        -0.05   0.95    0.02 -3.10 <0.005 9.01     -0.08  0.08
  race       -0.46   0.63    0.25 -1.79  0.07  3.77    -0.95  0.25
  wexp       0.56   1.74    0.17  3.32 <0.005 10.13    0.23  3.32
  mar        0.10   1.11    0.27  0.36  0.72  0.47    -0.44  0.44
  paro       0.02   1.02    0.16  1.22  0.21  0.15    -0.29  0.29
  prio       0.03   1.03    0.02  1.44  0.15  2.73    -0.01  2.73
  _intercept  1.48   4.41    0.41  3.60 <0.005 11.62    0.68  2.29
```

(continues on next page)
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 \( \text{prio} \) (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_coxnet()`

## 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_df=X)
aft.predict_survival_function(X, ancillary_df=X)
aft.predict_median(X, ancillary_df=X)
aft.predict_percentile(X, p=0.9, ancillary_df=X)
aft.predict_expectation(X, ancillary_df=X)
```

When predicting time remaining for uncensored individuals, you can use the `conditional_after` kwarg:

```python
censored_X = rossi.loc[-rossi['arrest'].astype(bool)]
censored_subjects_last_obs = censored_X['week']
aft.predict_cumulative_hazard(censored_X, ancillary_df=censored_X, conditional_after=censored_subjects_last_obs)
aft.predict_survival_function(censored_X, ancillary_df=censored_X, conditional_after=censored_subjects_last_obs)
aft.predict_median(censored_X, ancillary_df=censored_X, conditional_after=censored_subjects_last_obs)
aft.predict_percentile(X, p=0.9, ancillary_df=censored_X, conditional_after=censored_subjects_last_obs)
```

There are two hyper-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=1e-4, l1_ratio=1.0)
aft_with_elastic_penalty.fit(rossi, 'week', 'arrest')
aft_with_elastic_penalty.predict_median(rossi)
aft_with_elastic_penalty.print_summary()
```

```python
<\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 <0.005 112.47 112.47

(continues on next page)
The log-normal and log-logistic AFT models

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')
```

The piecewise-exponential regression and generalized gamma models

Another class of parametric models involves more flexible modeling of the hazard function. The `PiecewiseExponentialRegressionFitter` can model jumps in the hazard (think: the differences in “survival-of-staying-in-school” between 1st year, 2nd year, 3rd year, and 4th year students), and constant values between jumps. The ability to specify when these jumps occur, called breakpoints, offers modelers great flexibility. An example application involving customer churn is available in this notebook.

For a flexible and smooth parametric model, there is the `GeneralizedGammaRegressionFitter`. This model is actually a generalization of all the AFT models above (that is, specific values of its parameters represent another model) - see docs for specific parameter values. The API is slightly different however, and looks more like how custom regression models are built (see next section on Custom Regression Models).

```python
from lifelines import GeneralizedGammaRegressionFitter
from lifelines.datasets import load_rossi
df = load_rossi()
df['constant'] = 1.
# this will regress df against all 3 parameters
ggf = GeneralizedGammaRegressionFitter().fit(df, 'week', 'arrest')
ggf.print_summary()
```
# if we only want to regress against the scale parameter, `mu_`
regressors = {
    'mu_': rossi.columns,
    'sigma_': ['constant'],
    'lambda_': ['constant']
}

ggf = GeneralizedGammaRegressionFitter().fit(df, 'week', 'arrest',
    regressors=regressors)
ggf.print_summary()

Model selection for parametric models

Often, you don’t know *a priori* which parametric 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_)  # -678.941, slightly the best model.
print(lnf.log_likelihood_)  # -680.329
print(wf.log_likelihood_)  # -679.611

Left, right and interval censored data

The parametric 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
   0.10
__intercept  2.91    18.32    0.02 130.15 <0.005 inf    2.86
   2.95
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.

Custom parametric regression models

*lifelines* has a very general syntax for creating your own parametric regression models. If you are looking to create your own custom models, see docs Custom Regression Models.

1.7.4 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:

\[
h(t|x) = b_0(t) + b_1(t)x_1 + \ldots + b_N(t)x_N
\]
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 NelsonAalenFitter). This is important when interpreting plots produced.

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

```python
from lifelines import AalenAdditiveFitter
from lifelines.datasets import load_dd

data = load_dd()
data.head()
```

I’m using the lovely library Patsy here to create a design matrix from my original DataFrame.

```python
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())
```

```none
['baseline',
 'un_continent_name[T.Americas]',
 'un_continent_name[T.Asia]',
 'un_continent_name[T.Europe]',
 'un_continent_name[T.Oceanial]',
 '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']
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()
```

```
-0.03447 | 0.06216 | 0.2058 | -0.009559 | 0.07611 | 0.08729 | -0.1362 | 0.04885 | 0.1285 | 0.000092
0.142780 | 0.02496 | 0.11122 | 0.2083 | -0.079042 | 0.11704 | 0.36254 | -0.2293 | 0.17103 | 0.1238 | 0.000044
0.301540 | 0.07212 | 0.10929 | 0.1614 | 0.063030 | 0.16553 | 0.68693 | -0.2738 | 0.33300 | 0.1499 | 0.000004
0.379690 | 0.06853 | 0.15162 | 0.2609 | 0.185569 | 0.22695 | 0.95016 | -0.2961 | 0.37351 | 0.4311 | -0.000032
0.367490 | 0.20201 | 0.21252 | 0.2429 | 0.188740 | 0.25127 | 1.15132 | -0.3926 | 0.54952 | 0.7593 | 0.000000
```

`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:
   baseline  un_continent_name[T.Americas]  un_continent_name[T.Asia]  ...  start_year  T  E
      268    1.0                  1.0                      ...         0.0      3  0
```

```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);
```
Note: Because of the nature of the model, estimated survival functions of individuals can increase. This is an expected artifact of Aalen’s additive model.

1.7.5 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 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 modeling 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. This is because the difference between a censored value and the predicted value could be due to poor prediction or due to censoring. Below we introduce alternative ways to measure prediction performance.

In this author’s opinion, the best way to measure predictive performance is evaluating the log-likelihood on out-of-sample data. The log-likelihood correctly handles any type of censoring, and is precisely what we are maximizing in the model training. The in-sample log-likelihood is available under `log_likelihood_` of any regression model. For out-of-sample data, the `score()` method (available on all regression models) can be used. This returns the average evaluation of the out-of-sample log-likelihood. We want to maximize this.

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

rossi = load_rossi().sample(frac=1.0)
train_rossi = rossi.iloc[:400]
test_rossi = rossi.iloc[400:]
cph_l2 = CoxPHFitter(penalizer=0.1, l1_ratio=0.).fit(train_rossi, 'week', 'arrest')
cph_l1 = CoxPHFitter(penalizer=0.1, l1_ratio=1.).fit(train_rossi, 'week', 'arrest')

print(cph_l2.score(test_rossi))
print(cph_l1.score(test_rossi))  # better model
```

Another censoring-sensitive measure is the concordance-index, also known as the c-index. This measure evaluates the accuracy of the ranking 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 `score()`, but also available under the `concordance_index_` 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")

# fours ways to view the c-index:
# method one
  cph.print_summary()
```
# method two
print(cph.concordance_index_)

# method three
print(cph.score(rossi, scoring_method="concordance_index"))

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

Note: Remember, the concordance score evaluates the relative rankings of subject’s event times. Thus, it is scale and shift invariant (i.e. you can multiple by a positive constant, or add a constant, and the rankings won’t change). A model maximized for concordance-index does not necessarily give good predicted times, but will give good predicted rankings.

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).

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)
# [2.9896, -3.08810, -3.02747]

scores = k_fold_cross_validation(cph, regression_dataset, 'T', event_col='E', k=3, scoring_method="concordance_index")
print(scores)
# [0.5449, 0.5587, 0.6179]

Also, lifelines has wrappers for compatibility with scikit learn for making cross-validation and grid-search even easier.

1.8 Custom regression models

Like for univariate models, it is possible to create your own custom parametric survival models. Why might you want to do this?

- Create new / extend AFT models using known probability distributions
- Create a piecewise model using domain knowledge about subjects
• Iterate and fit a more accurate parametric model

`lifelines` has a very simple API to create custom parametric regression models. You only need to define the cumulative hazard function. For example, the cumulative hazard for the constant-hazard regression model looks like:

\[
H(t, x) = \frac{t}{\lambda(x)} \\
\lambda(x) = \exp(\vec{\beta} \cdot \vec{x}^T)
\]

where \( \beta \) are the unknowns we will optimize over.

Below are some example custom models.

```python
from lifelines.fitters import ParametricRegressionFitter
from autograd import numpy as np
from lifelines.datasets import load_rossi

class ExponentialAFTFitter(ParametricRegressionFitter):
    _fitted_parameter_names = ['lambda_']

def _cumulative_hazard(self, params, t, Xs):
    beta = params['lambda_']
    X = Xs['lambda_']
    lambda_ = np.exp(np.dot(X, beta))
    return t / lambda_

rossi = load_rossi()
rossi['intercept'] = 1.0
regressors = {
    'lambda_': rossi.columns
}
eaf = ExponentialAFTFitter().fit(rossi, 'week', 'arrest', regressors=regressors)
eaf.print_summary()
```

```python
[17]: %matplotlib inline
%config InlineBackend.figure_format = 'retina'

class DependentCompetingRisksHazard(ParametricRegressionFitter):
    
    Reference
    ------------
    Frees and Valdez, UNDERSTANDING RELATIONSHIPS USING COPULAS
    
    _fitted_parameter_names = ["lambda1", "rho1", "lambda2", "rho2", "alpha"]
```

(continues on next page)
def _cumulative_hazard(self, params, T, Xs):
    lambda1 = np.exp(np.dot(Xs["lambda1"], params["lambda1"]))
    lambda2 = np.exp(np.dot(Xs["lambda2"], params["lambda2"]))
    rho2 = np.exp(np.dot(Xs["rho2"], params["rho2"]))
    rho1 = np.exp(np.dot(Xs["rho1"], params["rho1"]))
    alpha = np.exp(np.dot(Xs["alpha"], params["alpha"]))

    return ((T / lambda1) ** rho1 + (T / lambda2) ** rho2) ** alpha

fitter = DependentCompetingRisksHazard(penalizer=0.1)
rossi = load_rossi()
rossi["intercept"] = 1.0
rossi["week"] = rossi["week"] / rossi["week"].max()  # scaling often helps with
                                           # convergence
covariates = {
    "lambda1": rossi.columns,
    "lambda2": rossi.columns,
    "rho1": ["intercept"],
    "rho2": ["intercept"],
    "alpha": ["intercept"],
}
fitter.fit(rossi, "week", event_col="arrest", regressors=covariates, timeline=np.linspace(0, 2))
fitter.print_summary(2)

ax = fitter.plot()
ax = fitter.predict_survival_function(rossi.loc[::100]).plot(figsize=(8, 4))
ax.set_title("Predicted survival functions for selected subjects")

<IPython.core.display.HTML object>

[17]: Text(0.5, 1.0, 'Predicted survival functions for selected subjects')
1.8.1 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. In this case, the survival function for an individual should not asymptically approach zero, but some positive value. Models that describe this are sometimes called cure models (i.e. the subject is “cured” of death and hence no longer susceptible) or time-lagged conversion models.

It would be nice to be able to use common survival models and have some “cure” component. Let’s suppose that for individuals that will experience the event of interest, their survival distribution is a Weibull, denoted $S_W(t)$. For a random selected individual in the population, their survival curve, $S(t)$, is:

$$S(t) = P(T > t) = P(\text{cured})P(T > t | \text{cured}) + P(\text{not cured})P(T > t | \text{not cured}) = p + (1-p)S_W(t)$$

Even though it’s in an unconventional form, we can still determine the cumulative hazard (which is the negative logarithm of the survival function):

$$H(t) = -\log(p + (1-p)S_W(t))$$

[9]: from autograd.scipy.special import expit

```python
class CureModel(ParametricRegressionFitter):
    _scipy_fit_method = "SLSQP"
    _scipy_fit_options = {"ftol": 1e-10, "maxiter": 200}
    _fitted_parameter_names = ["lambda_", "beta_", "rho_"]

    def _cumulative_hazard(self, params, T, Xs):
        c = expit(np.dot(Xs["beta_"], params["beta_"]))
        lambda_ = np.exp(np.dot(Xs["lambda_"], params["lambda_"]))
        rho_ = np.exp(np.dot(Xs["rho_"], params["rho_"]))
        sf = np.exp(-(T / lambda_) ** rho_)
```

(continues on next page)
return -np.log((1 - c) + c * sf)

cm = CureModel(penalizer=0.0)

rossi = load_rossi()
rossi["intercept"] = 1.0
covariates = {"lambda_": rossi.columns, "rho_": ["intercept"], "beta_": ["intercept", "fin"]}

cm.fit(rossi, "week", event_col="arrest", regressors=covariates, timeline=np.arange(250))

<IPython.core.display.HTML object>

[18]: cm.predict_survival_function(rossi.loc[::100]).plot(figsize=(12,6))

[18]: <matplotlib.axes._subplots.AxesSubplot at 0x1278726a0>

[11]: # what's the effect on the survival curve if I vary "age"

fig, ax = plt.subplots(figsize=(12,6))

cm.plot_covariate_groups(["age"], values=np.arange(20, 50, 5), cmap='coolwarm', ax=ax)

[11]: <matplotlib.axes._subplots.AxesSubplot at 0x11167bb38>
1.8.2 Spline models

See royston_parmar_splines.py in the examples folder: https://github.com/CamDavidsonPilon/lifelines/tree/master/examples

1.9 Compatibility with scikit-learn

New to lifelines in version 0.21.3 is a wrapper that allows you to use lifeline’s regression models with scikit-learn’s APIs.

Note: the API and functionality is still experimental. Please report any bugs or features on our Github issue list.

```python
from lifelines.utils.sklearn_adapter import sklearn_adapter
from lifelines import CoxPHFitter
from lifelines.datasets import load_rossi
X = load_rossi().drop('week', axis=1)  # keep as a dataframe
Y = load_rossi().pop('week')
CoxRegression = sklearn_adapter(CoxPHFitter, event_col='arrest')
# CoxRegression is a class like the `LinearRegression` class or `SVC` class in scikit-learn
```

(continues on next page)
sk_cph = CoxRegression(penalizer=1e-5)
sk_cph.fit(X, Y)
print(sk_cph)

"""
SklearnCoxPHFitter(alpha=0.05, penalizer=1e-5, strata=None, tie_method='Efron')
"""

sk_cph.predict(X)
sk_cph.score(X, Y)

**Note:** The X variable still needs to be a DataFrame, and should contain the event-occurred column (`event_col`) if it exists.

If needed, the original lifeline’s instance is available as the `lifelines_model` attribute.

sk_cph.lifelines_model.print_summary()

The wrapped classes can even be used in more complex scikit-learn functions (ex: `cross_val_score`) and classes (ex: `GridSearchCV`):

```python
from lifelines import WeibullAFTFitter
from sklearn.model_selection import cross_val_score

base_class = sklearn_adapter(WeibullAFTFitter, event_col='arrest')
wf = base_class()

scores = cross_val_score(wf, X, Y, cv=5)
print(scores)

"""
[0.59037328 0.503427 0.55454545 0.59689534 0.62311068]
"""

from sklearn.model_selection import GridSearchCV
clf = GridSearchCV(wf, {
    "penalizer": 10.0 ** np.arange(-2, 3),
    "l1_ratio": [0, 1/3, 2/3],
    "model_ancillary": [True, False],
}, cv=4)
clf.fit(X, Y)

print(clf.best_estimator_)

"""
SklearnWeibullAFTFitter(alpha=0.05, fit_intercept=True,
    l1_ratio=0.66666, model_ancillary=True,
    penalizer=0.01)
"""
Note: The lifelines.utils.sklearn_adapter() is currently only designed to work with right-censored data.

1.10 Time varying survival regression

1.10.1 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 must be careful if we condition on whether they received the transplant. 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. The general mathematical description is:

\[ h(t|x) = \hat{b}_0(t) \exp \left( \sum_{i=1}^{n} \beta_i (x_i(t) - x_i) \right) \]

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 it.

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:

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

```python
from lifelines.utils import add_covariate_to_timeline
def = 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>

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 `lifelines.utils.add_covariate_to_timeline()` to fold the covariate dataset into the original dataset.

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
def = 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)
```

```
 id   E1
0   1   1.0
1   2   NaN
2   3   3.0
...
```

Initially, this can't be added to our baseline DataFrame. However, using `lifelines.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)
```

```
 event id duration E1
0   1   1.0   1
1   3   3.0   1
...
```

```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:
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`, except we need to tell the `fit()` about the additional time columns.

Fitting the Cox model to the data involves an iterative 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.

1.11 Testing the proportional hazard assumptions

This Jupyter notebook is a small tutorial on how to test and fix proportional hazard problems. An important question to first ask is: *do I need to care about the proportional hazard assumption?* - often the answer is no.

The proportional hazard assumption is that *all* individuals have the same hazard function, but a unique scaling factor infront. So the *shape* of the hazard function is the same for all individuals, and only a scalar multiple changes per
individual.

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

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.

New to lifelines 0.16.0 is the \texttt{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 \texttt{lifelines} rejects the null (that is, \texttt{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 \( j \)th 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.
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    11.03 <0.005    10.12
          rank 11.45 <0.005    10.45
  fin   km    0.02    0.89    0.17
          rank 0.02    0.90    0.15
  mar   km    0.60    0.44    1.19
          rank 0.71    0.40    1.32
  paro  km    0.12    0.73    0.45
          rank 0.13    0.71    0.49
  prio  km    0.02    0.88    0.18
          rank 0.02    0.89    0.17
  race  km    1.44    0.23    2.12
          rank 1.43    0.23    2.11
  wexp  km    7.48    0.01    7.32
          rank 7.31    0.01    7.19
```

1. Variable 'age' failed the non-proportional test: p-value is 0.0007.
   - 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.

---

[A] https://lifelines.readthedocs.io/en/latest/jupyter_notebooks/Proportional\hazard\assumption.html
[B] https://lifelines.readthedocs.io/en/latest/jupyter_notebooks/Proportional\hazard\assumption.html#Bin-variable-and-stratify-on-it
[C] https://lifelines.readthedocs.io/en/latest/jupyter_notebooks/Proportional\hazard\assumption.html#Introduce-time-varying-covariates
[D] https://lifelines.readthedocs.io/en/latest/jupyter_notebooks/Proportional\hazard\assumption.html#Modify-the-functional-form
[E] https://lifelines.readthedocs.io/en/latest/jupyter_notebooks/Proportional\hazard\assumption.html#Stratification

1.11. Testing the proportional hazard assumptions

---

![Scaled Schoenfeld residuals of 'age'](image1)

![Scaled Schoenfeld residuals of 'wexp'](image2)
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")
```

```
<linelines.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 11.453 0.001 10.453
fin 0.015 0.902 0.148
mar 0.709 0.400 1.323
paro 0.134 0.714 0.148
prio 0.019 0.891 0.167
race 1.426 0.232 2.106
wexp 7.315 0.007 7.192
```

**Stratification**

In the advice above, we can see that `wexp` has small cardinality, so we can easily fix that by specifying it in the `strata`. What does the `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 *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 *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:

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

where now we have a unique baseline hazard 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 \( G \)).

```python
[6]: cph.fit(rossi, 'week', 'arrest', strata=['wexp'])
cph.print_summary(model="wexp in strata")
<IPython.core.display.HTML object>
```

```python
[7]: cph.check_assumptions(rossi, show_plots=True)
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
```
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.

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

    test_statistic  p  -log2(p)
    age km 11.29 <0.005 10.32
    rank 4.62 0.03 4.99
    fin km 0.02 0.90 0.16
    rank 0.05 0.83 0.28
    mar km 0.53 0.47 1.10
    rank 1.31 0.25 1.99
    paro km 0.09 0.76 0.40
    rank 0.00 0.97 0.05
    prio km 0.02 0.89 0.16
    rank 0.02 0.90 0.16
    race km 1.47 0.23 2.15
    rank 0.64 0.42 1.23
```

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.

---


(continues on next page)
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
tau = rossi['age'] - rossi['age'].mean()
tau2 = tau**2
tau3 = tau**3
cph.fit(rossi_higher_order_age, 'week', 'arrest', strata=['wexp'])
cph.print_summary(model="quad and cubic age terms")
```

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
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
cph.fit(rossi_higher_order_age, 'week', 'arrest', strata=['wexp'])
cph.print_summary(model="quad and cubic age terms")
```

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
    fin km                0.03 0.87 0.20
    rank                  0.02 0.90 0.15
    mar km                0.53 0.47 1.10
    rank                  0.94 0.33 1.59
    paro km               0.20 0.66 0.60
    rank                  0.01 0.93 0.10
    prio km               0.02 0.88 0.19
    rank                  0.01 0.90 0.15
    race km               1.28 0.26 1.96
    rank                  0.47 0.49 1.02

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.

```

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 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 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 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 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.

```
[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</th>
<th>age_strata</th>
</tr>
</thead>
<tbody>
<tr>
<td>27</td>
<td>(24, 27]</td>
</tr>
<tr>
<td>18</td>
<td>(15, 18]</td>
</tr>
<tr>
<td>19</td>
<td>(18, 21]</td>
</tr>
<tr>
<td>23</td>
<td>(21, 24]</td>
</tr>
<tr>
<td>19</td>
<td>(18, 21]</td>
</tr>
</tbody>
</table>
# drop the original, redundant, age column
rossi_strata_age = rossi_strata_age.drop('age', axis=1)
cph.fit(rossi_strata_age, 'week', 'arrest', strata=['age_strata', 'wexp'])

<\texttt{lifelines.CoxPHFitter}: fitted with 432 total observations, 318 right-censored observations>

\[ \texttt{cph.print_summary(3, model="stratified age and wexp")} \]
\[ \texttt{cph.plot()} \]
\[ \texttt{<\textit{IPython.core.display.HTML object}>} \]
\[ \texttt{cph.check_assumptions(rossi_strata_age)} \]

Proportional hazard assumption looks okay.

\textbf{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 \textit{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 \textit{lifelines}:

\[ \texttt{from lifelines.utils import to_episodic_format} \]
\[ \texttt{# the time_gaps parameter specifies how large or small you want the periods to be.} \]
\[ \texttt{rossi_long = to_episodic_format(rossi, duration_col='week', event_col='arrest', time_gaps=1.)} \]
\[ \texttt{rossi_long.head(25)} \]

\[ \begin{array}{cccccccccccc}
\text{stop} & \text{start} & \text{arrest} & \text{age} & \text{fin} & \text{id} & \text{mar} & \text{paro} & \text{prio} & \text{race} & \text{wexp} \\
0 & 1.0 & 0.0 & 0 & 27 & 0 & 0 & 0 & 1 & 3 & 1 & 0 \\
1 & 2.0 & 1.0 & 0 & 27 & 0 & 0 & 0 & 1 & 3 & 1 & 0 \\
\end{array} \]

(continues on next page)
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 an 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 an episodic dataset.

```
[14]: rossi_long['time*age'] = rossi_long['age'] * rossi_long['stop']
[15]: from lifelines import CoxTimeVaryingFitter
c = CoxTimeVaryingFitter()
c.fit(rossi_long,
    id_col='id',
    event_col='arrest',
    start_col='start',
    stop_col='stop',
    strata=['wexp'])
[15]: <lifelines.CoxTimeVaryingFitter: fitted with 19809 periods, 432 subjects, 114 events>
[16]: ctv.print_summary(3, model="age * time interaction")

<IPython.core.display.HTML object>

[17]: ctv.plot()
[17]: <matplotlib.axes._subplots.AxesSubplot at 0x10e37b978>"
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.

1.12 Do I need to care about the proportional hazard assumption?

You may be surprised that often you don’t need to care about the proportional hazard assumption. There are many reasons why not:

1. If your goal is survival prediction, then you don’t need to care about proportional hazards. Your goal is to maximize some score, irrelevant of how predictions are generated.

2. Given a large enough sample size, even very small violations of proportional hazards will show up.

3. There are legitimate reasons to assume that all datasets will violate the proportional hazards assumption. This is detailed well in Stensrud & Hernán’s “Why Test for Proportional Hazards?” [1].

Given the above considerations, the status quo is still to check for proportional hazards. So if you are avoiding testing for proportional hazards, be sure to understand and able to answer *why* you are avoiding testing.

1.13 More examples and recipes

This section goes through some examples and recipes to help you use lifelines.

1.13.1 Worked Examples

If you are looking for some full examples of lifelines, there are full Jupyter notebooks and scripts here and examples and ideas on the development blog.

1.13.2 Statistically compare two populations

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

Logrank test

---

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()

""

    t_0 = -1
    alpha = 0.95
null_distribution = chi squared
df = 1
use_bonferroni = True

---

    test_statistic   p
    3.528  0.00034 **

""

print(results.p_value)       # 0.46759
print(results.test_statistic) # 0.528

```

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({
```

(continues on next page)
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 `lifelines.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()
```

Subtraction and division between survival functions

If you are interested in taking the difference between two survival functions, simply trying to subtract the `survival_function_` will likely fail if the DataFrame’s indexes are not equal. Fortunately, the `lifelines.fitters.kaplan_meier_fitter.KaplanMeierFitter` and `lifelines.fitters.nelson_aalen_fitter.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.

Restricted mean survival times (RMST)

`lifelines` has a function to accurately compute the restricted mean survival time, defined as...
\[ \text{RMST}(t) = \int_0^t S(\tau) d\tau \]

This is a good metric for comparing two survival curves, as their difference represents the area between the curves (see figure below). The upper limit is often finite because the tail of the estimated survival curve has high variance and can strongly influence the integral.

```python
from lifelines.utils import restricted_mean_survival_time
from lifelines.datasets import load_waltons
df = load_waltons()
ix = df['group'] == 'miR-137'
T, E = df['T'], df['E']
time_limit = 50

kmf_exp = KaplanMeierFitter().fit(T[ix], E[ix], label='exp')
rmst_exp = restricted_mean_survival_time(kmf_exp, t=time_limit)

kmf_con = KaplanMeierFitter().fit(T[~ix], E[~ix], label='control')
rmst_con = restricted_mean_survival_time(kmf_con, t=time_limit)
```

Furthermore, there exist plotting functions to plot the RMST:

```python
from lifelines.plotting import rmst_plot

ax = plt.subplot(311)
rmst_plot(kmf_exp, t=time_limit, ax=ax)

ax = plt.subplot(312)
rmst_plot(kmf_con, t=time_limit, ax=ax)

ax = plt.subplot(313)
rmst_plot(kmf_exp, model2=kmf_con, t=time_limit, ax=ax)
```
RMST=25.000

RMST=47.978

RMST(exp) - RMST(control)= -22.978
1.13.3 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.

*lifelines* also has wrappers to use scikit-learn’s cross validation and grid search tools. See how to use lifelines with scikit learn.

1.13.4 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\_plots()} also works with left censorship as well.

### 1.13.5 Selecting a parametric model using AIC

For univariate models (later to be extended to regression models), a natural way to compare different models is the AIC:

\[
AIC(\text{model}) = -2ll + 2k
\]

where \(k\) is the number of parameters (degrees-of-freedom) of the model and \(ll\) is the maximum log-likelihood. The model with the lowest AIC is desirable, since it's a trade off between maximizing the log-likelihood with as few parameters as possible.

\texttt{lifelines} has a built in function to automate AIC comparisons between univariate parametric models:

```python
from lifelines.utils import find_best_parametric_model
from lifelines.datasets import load_lymph_node

T = load_lymph_node()['rectime']
E = load_lymph_node()['censrec']

best_model, best_aic_ = find_best_parametric_model(T, E)

print(best_model)
# <lifelines.SplineFitter: "Spline\_estimate", fitted with 686 total observations, 387 right-censored observations>

best_model.plot_hazard()
```
1.13.6 Plotting multiple figures on a plot

When `.plot` is called, an `axis` object is returned which can be passed into future calls of `.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)
```
1.13.7 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, ncensor_styles={'ms': 6, 'marker': 's'})")
kmf.plot(show_censors=True, censor_styles={'ms': 6, 'marker': 's'})
```
Hide confidence intervals

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

1.13. More examples and recipes
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
from lifelines.datasets import load_waltons

waltons = load_waltons()
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

![Kaplan-Meier plot](image)

### 1.13.8 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')

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)
```
1.13.9 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

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

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

1.13.10 Set the index/timeline of a estimate

Suppose your dataset has lifetimes grouped near time 60, thus after fitting `lifelines.fitters.kaplan_meier_fitter.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
  66  0.10
  68  0.07
  69  0.04
  70  0.02
  71  0.01
  74  0.00
```

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.

### 1.13.11 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>

1.13.12 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",
  (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)
```

```python
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:
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>1.0</td>
<td>NaN</td>
<td>2.0</td>
</tr>
<tr>
<td>1</td>
<td>NaN</td>
<td>5.0</td>
<td>NaN</td>
</tr>
<tr>
<td>2</td>
<td>3.0</td>
<td>5.0</td>
<td>7.0</td>
</tr>
</tbody>
</table>

Initially, this can't be added to our baseline time-varying dataset. Using `lifelines.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')
pd.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>5</td>
<td>5.0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>3</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>7</td>
<td>7.0</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

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

### 1.13.13 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}])
```

(continues on next page)
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)
```

```
| start | cumsum_funding | funding | stop  | id   | E
|-------|----------------|---------|-------|------|---
| 0     | 0              | 0.0     | 5.0   | FB   | False
| 5     | 30.0           | 30.0    | 10.0  | FB   | False
| 10    | 45.0           | 15.0    | 12.0  | FB   | True
| 0     | 0              | 0.0     | 6.0   | SU   | True
| 6     | 10.0           | 10.0    | 10.0  | SU   | False
| 10    | 19.0           | 9.0     | 10.0  | SU   | True
```

### 1.13.14 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
n_exp, n_con = sample_size_necessary_under_cph(desired_power, ratio_of_participants, p_exp, p_con, postulated_hazard_ratio)
# (421, 421)
```

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.

### 1.13.15 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.
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

1.13.16 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.

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

1.13.17 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.

1.13.18 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 `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`).

### 1.13.19 Serialize a lifelines model to disk

When you want to save (and later load) a *lifelines* model to disk, you can use the `loads` and `dumps` API from any popular serialization library.

```python
from dill import loads, dumps
from pickle import loads, dumps
from joblib import loads, dumps
s_cph = dumps(cph)
cph_new = loads(s_cph)
cph.summary

s_kmf = dumps(kmf)
kmf_new = loads(s_kmf)
kmf.summary
```
1.13.20 Produce a LaTeX or HTML table

New in version 0.23.1, lifelines models now have the ability to output a LaTeX or HTML table from the print_summary option:

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

rossi = load_rossi()
cph = CoxPHFitter().fit(rossi, 'week', 'arrest')

# print a LaTeX table:
cph.print_summary(style="latex")

# print a HTML summary and table:
cph.print_summary(style="html")
```

In order to use the produced table summary in LaTeX, make sure you import the package `booktabs` in your preamble (`\usepackage{booktabs}`), since it is required to display the table properly.

1.14 API Reference

1.14.1 fitters

Univariate models

AalenJohansenFitter

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

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 import AalenJohansenFitter
from lifelines.datasets import load_waltons
T, E = load_waltons()['T'], load_waltons()['E']
ajf = AalenJohansenFitter(calculate_variance=True)
ajf.fit(T, E, event_of_interest=1)
ajf.cumulative_density_
ajf.plot()
```

**References**


**conditional_time_to_event**

Return a DataFrame, with index equal to `survival_function`, 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.

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

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

**divide** *(other) → pandas.core.frame.DataFrame*

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

**Parameters**

- `other` *(same object as self)*

**fit** *(durations, event_observed, event_of_interest, timeline=None, entry=None, label=None, 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)

Alias for `fit`

**See also:**

`fit`

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

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.

**percentile** *(p: float) → float*

Return the unique time point, t, such that $S(t) = p$.

**Parameters**

- **p** *(float)*

**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.

Returns a pyplot axis object

Return type ax

plot_cumulative_density(**kwargs)
plot_cumulative_hazard(**kwargs)
plot_density(**kwargs)
plot_hazard(**kwargs)
plot_survival_function(**kwargs)

predict (times: Union[Iterable[float], float], interpolate=False) → pandas.core.series.Series
    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 (bool, 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.

subtract (other) → pandas.core.frame.DataFrame
    Subtract the [0] of two [1] objects.

Parameters other (same object as self)

survival_function_at_times (times, label=None)

BreslowFlemingHarringtonFitter

class lifelines.fitters.breslow_fleming_harrington_fitter.BreslowFlemingHarringtonFitter (alpha: float = 0.05, label: str = None) → lifelines.fitters.UnivariateFitter

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 Nelson-Aalen estimator is the negative logarithm of the Breslow-Fleming-Harrington estimator.

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, 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.

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

divide(other) \rightarrow \text{pandas.core.frame.DataFrame}

Divide the \{0\} of two \{1\} objects.

Parameters

other (same object as self)

fit(durations, event_observed=None, timeline=None, entry=None, label=None, alpha=None, ci_labels=None, weights=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 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. 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_<alpha>

Returns

Return type

self, with new properties like survival_function_.

fit_right_censoring(*args, **kwargs)

Alias for fit

See also:

fit

hazard_at_times(times, label=None)

median_survival_time_

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.

percentile(p: float) \rightarrow float

Return the unique time point, t, such that S(t) = p.

Parameters

p (float)

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.

**Returns** a pyplot axis object

**Return type** ax

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

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

```python
plot_density(**kwargs)
```

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

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

**predict** *(times: Union[Iterable[float], float], interpolate=False) → pandas.core.series.Series*

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** *(bool, optional (default=False))* – for methods that produce a stepwise solution (Kaplan-Meier, Nelson-Aalen, etc), turning this to True will use a linear interpolation method to provide a more “smooth” answer.

```python
subtract (other) → pandas.core.frame.DataFrame
```

Subtract the {0} of two {1} objects.

**Parameters** other *(same object as self)*

```python
survival_function_at_times (times, label=None) → pandas.core.series.Series
```

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

**Parameters**

• **times** *(iterable or float)*

• **label** *(str)*
ExponentialFitter

class lifelines.fitters.exponential_fitter.ExponentialFitter(*args, **kwargs)
Bases: lifelines.fitters.KnownModelParametricUnivariateFitter

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 \( \lambda \).

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

variance_matrix_
The variance matrix of the coefficients

Type numpy array
median_survival_time
  The median time to event
  Type  float

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

density
  The estimated density function (PDF) (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_, 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.

confidence_interval_
  The confidence interval of the cumulative hazard. This is an alias for confidence_interval_cumulative_hazard_.

confidence_interval_cumulative_density_
  The lower and upper confidence intervals for the cumulative density

confidence_interval_cumulative_hazard_
  The confidence interval of the cumulative hazard. This is an alias for confidence_interval_.

confidence_interval_density_
  The confidence interval of the hazard.

confidence_interval_hazard_
  The confidence interval of the hazard.
The lower and upper confidence intervals for the survival function

cumulative_density_at_times(times, label=None) → pandas.core.series.Series
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) → pandas.core.series.Series
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.

density_at_times(times, label=None) → pandas.core.series.Series
Return a Pandas series of the predicted probability density function, dCDF/dt, 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.

divide(other) → pandas.core.frame.DataFrame
Divide the {0} of two {1} objects.

Parameters
- `other` (same object as self)

event_table

fit(durations, event_observed=None, timeline=None, label=None, alpha=None, ci_labels=None, show_progress=False, entry=None, weights=None, initial_point=None) → ParametricUnivariateFitter

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: 
- `show_progress` (bool, 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
- **initial_point** *(d, numpy array, optional)* – initialize the starting point of the iterative algorithm. Default is the zero vector.

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, initial_point=None) → ParametricUnivariateFitter

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 initialzing 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** *(bool, 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
- **initial_point** *(d, numpy array, optional)* – initialize the starting point of the iterative algorithm. Default is the zero vector.

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, initial_point=None) → ParametricUnivariateFitter

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** *(bool, 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
• **initial_point** *(d, numpy array, optional)* – initialize the starting point of the iterative algorithm. Default is the zero vector.

Returns

Return type self with new properties like cumulative_hazard_, survival_function_

**fit_right_censoring** (*args, **kwargs*)
Alias for fit

See also:

fit

**hazard_at_times** *(times, label=None)* → pandas.core.series.Series
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.

**median_survival_time**_
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.

**percentile** *(p)*
Return the unique time point, t, such that S(t) = p.

Parameters **p** *(float)*

**plot** (**kwargs**)
Produce a pretty-plot of the estimate.

**plot_cumulative_density** (**kwargs**)

**plot_cumulative_hazard** (**kwargs**)

**plot_density** (**kwargs**)

148 Chapter 1. Contents:
plot_hazard(**kwargs)

plot_survival_function(**kwargs)

predict(times: Union[Iterable[float], float], interpolate=False) → pandas.core.series.Series

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 (bool, 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.

print_summary(decimals=2, style=None, **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
- style (string) – \{html, ascii, latex\}
- kwargs – print additional metadata in the output (useful to provide model names, dataset names, etc.) when comparing multiple outputs.

subtract(other) → pandas.core.frame.DataFrame

Subtract the \{0\} of two \{1\} objects.

Parameters other (same object as self)

summary

Summary statistics describing the fit.

See also:

print_summary

survival_function_at_times(times, label=None) → pandas.core.series.Series

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.

GeneralizedGammaFitter

class lifelines.fitters.generalized_gamma_fitter.GeneralizedGammaFitter(*args, **kwargs)

Bases: lifelines.fitters.KnownModelParametricUnivariateFitter

This class implements a Generalized Gamma model for univariate data. The model has parameterized form:

The survival function is:

\[ S(t) = \begin{cases} 1 - \Gamma_{RL} \left( \frac{1}{\lambda^2}; \frac{\lambda \left( \log(t) - \mu \right)}{\sigma^2} \right) & \text{if } \lambda > 0 \\ \Gamma_{RL} \left( \frac{1}{\lambda^2}; \frac{\lambda \left( \log(t) - \mu \right)}{\sigma^2} \right) & \text{if } \lambda \leq 0 \end{cases} \]

where \( \Gamma_{RL} \) is the regularized lower incomplete Gamma function.
This model has the Exponential, Weibull, Gamma and Log-Normal as sub-models, and thus can be used as a way to test which model to use:

1. When $\lambda = 1$ and $\sigma = 1$, then the data is Exponential.
2. When $\lambda = 1$ then the data is Weibull.
3. When $\sigma = \lambda$ then the data is Gamma.
4. When $\lambda = 0$ then the data is Log-Normal.
5. When $\lambda = -1$ then the data is Inverse-Weibull.
6. When $\sigma = -\lambda$ then the data is Inverse-Gamma.

After calling the `.fit` method, you have access to properties like: `cumulative_hazard_`, `survival_function_`. A summary of the fit is available with the method `print_summary()`.

**Important:** The parameterization implemented has $\log \sigma$, thus there is a `ln_sigma_` in the output. Exponentiate this parameter to recover $\sigma$.

**Important:** This model is experimental. It’s API may change in the future. Also, it’s convergence is not very stable.

**Parameters**
**alpha** (*float, optional (default=0.05)*) – the level in the confidence intervals.

**Examples**

```python
def print_summary()
from lifelines import GeneralizedGammaFitter
from lifelines.datasets import load_waltons
waltons = load_waltons()

ggf = GeneralizedGammaFitter()
sggf.fit(waltons['T'], waltons['E'])
sggf.plot()
sggf.summary
```

**cumulative_hazard_**
The estimated cumulative hazard (with custom timeline if provided)

**Type** DataFrame

**hazard_**
The estimated hazard (with custom timeline if provided)

**Type** DataFrame

**survival_function_**
The estimated survival function (with custom timeline if provided)

**Type** DataFrame

**cumulative_density_**
The estimated cumulative density function (with custom timeline if provided)

**Type** DataFrame
density
   The estimated density function (PDF) (with custom timeline if provided)
   Type  DataFrame

variance_matrix_
   The variance matrix of the coefficients
   Type  numpy array

median_survival_time_
   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

alpha_
   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_, 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.

confidence_interval_
   The confidence interval of the cumulative hazard. This is an alias for confidence_interval_cumulative_hazard_.

confidence_interval_cumulative_density_
   The lower and upper confidence intervals for the cumulative density

confidence_interval_cumulative_hazard_
   The confidence interval of the cumulative hazard. This is an alias for confidence_interval_.

1.14. API Reference 151
confidence_interval_density_
The confidence interval of the hazard.

certainty_interval_hazard_
The confidence interval of the hazard.

certainty_interval_survival_function_
The lower and upper confidence intervals for the survival function.

cumulative_density_at_times(times, label=None) → pandas.core.series.Series
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 cumulative density function at.
- label (string, optional) – Rename the series returned. Useful for plotting.

cumulative_hazard_at_times(times, label=None) → pandas.core.series.Series
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.

density_at_times(times, label=None) → pandas.core.series.Series
Return a Pandas series of the predicted probability density function, dCDF/dt, at specific times.

Parameters
- times (iterable or float) – values to return the probability density function at.
- label (string, optional) – Rename the series returned. Useful for plotting.

divide(other) → pandas.core.frame.DataFrame
Divide the \(0\) of two \(1\) objects.

Parameters
- other (same object as self)

event_table

fit(durations, event_observed=None, timeline=None, label=None, alpha=None, ci_labels=None, show_progress=False, entry=None, weights=None, initial_point=None) → ParametricUnivariateFitter

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 (bool, 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

• `initial_point ((d,) numpy array, optional)` – initialize the starting point of the iterative algorithm. Default is the zero vector.

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, initial_point=None) \rightarrow ParametricUnivariateFitter

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 (n 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: 〈label〉_lower_〈alpha〉

• `show_progress (bool, 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

• `initial_point ((d,) numpy array, optional)` – initialize the starting point of the iterative algorithm. Default is the zero vector.

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, initial_point=None) → ParametricUnivariateFitter

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** (bool, 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
- **initial_point** ((d,) numpy array, optional) – initialize the starting point of the iterative algorithm. Default is the zero vector.

**Returns**

Return type self with new properties like cumulative_hazard_, survival_function_

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

Alias for fit

**See also:**

fit

**hazard_at_times** (*times, label=None) → pandas.core.series.Series

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.

**median_survival_time**

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.

**percentile** (*p*)

Return the unique time point, t, such that S(t) = p.

**Parameters**

- **p** (float)
plot(**kwargs)

Produce a pretty-plot of the estimate.

plot_cumulative_density(**kwargs)
plot_cumulative_hazard(**kwargs)
plot_density(**kwargs)
plot_hazard(**kwargs)
plot_survival_function(**kwargs)

predict(times: Union[Iterable[float], float], interpolate=False) → pandas.core.series.Series

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** (bool, 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.

print_summary(decimals=2, style=None, **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
- **style** (string) – {html, ascii, latex}
- **kwargs** – print additional metadata in the output (useful to provide model names, dataset names, etc.) when comparing multiple outputs.

subtract(other) → pandas.core.frame.DataFrame

Subtract the {0} of two {1} objects.

Parameters other (same object as self)

summary

Summary statistics describing the fit.

See also:

print_summary

survival_function_at_times(times, label=None) → pandas.core.series.Series

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.

KaplanMeierFitter

class lifelines.fitters.kaplan_meier_fitter.KaplanMeierFitter(alpha: float = 0.05, label: str = None)

Bases: lifelines.fitters.UnivariateFitter

Class for fitting the Kaplan-Meier estimate for the survival function.
Parameters

- **alpha** (float, optional (default=0.05)) – The alpha value associated with the confidence intervals.
- **label** (string, optional) – Provide a new label for the estimate - useful if looking at many groups.

Examples

```python
from lifelines import KaplanMeierFitter
from lifelines.datasets import load_waltons
waltons = load_waltons()

kmf = KaplanMeierFitter(label="waltons_data")
kmf.fit(waltons['T'], waltons['E'])
kmf.plot()
```

**survival_function**
The estimated survival function (with custom timeline if provided)

- **Type** DataFrame

**median_survival_time**
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`. Uses Greenwood’s Exponential formula (“log-log” in R).

- **Type** DataFrame

**confidence_interval_survival_function**
The lower and upper confidence intervals for the survival function. An alias of `confidence_interval`. Uses Greenwood’s Exponential formula (“log-log” in R).

- **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

**eventObserved**
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_, 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.

cumulative_density_at_times (times, label=None) \rightarrow pandas.core.series.Series
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) \rightarrow pandas.core.frame.DataFrame
    Divide the \{0\} of two \{1\} objects.

Parameters other (same object as self)

fit (durations, event_observed=None, timeline=None, entry=None, label=None, 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.

• 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_survival_time_`

Return type `KaplanMeierFitter`

`fit_left_censoring` *(durations, event_observed=None, timeline=None, entry=None, label=None, 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 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.
- `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_survival_time_`

Return type `KaplanMeierFitter`

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

Alias for `fit`

See also:

`fit`

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

`median_survival_time_`

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.

`percentile` *(p: float) → float*

Return the unique time point, t, such that S(t) = p.

Parameters p *(float)*

`plot` *(**kwargs)*

Plots a pretty figure of the model

Matplotlib plot arguments can be passed in inside the `**kwargs`
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.

Returns a pyplot axis object

Return type *ax*

**plot_cumulative_density(****kwargs)**

Plots a pretty figure of the cumulative density function.

Matplotlib plot arguments can be passed in inside the kwargs.

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.

Returns a pyplot axis object

Return type ax

plot_cumulative_hazard(**kwargs)

plot_density(**kwargs)

plot_hazard(**kwargs)

plot_loglogs(*args, **kwargs)

Plot log(S(t)) against log(t). Same arguments as .plot.

plot_survival_function(**kwargs)

Alias of plot

predict(times: Union[Iterable[float], float], interpolate=False) \rightarrow pandas.core.series.Series

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 (bool, 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.

subtract(other) \rightarrow pandas.core.frame.DataFrame

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

Parameters other (same object as self)

survival_function_at_times(times, label=None) \rightarrow pandas.core.series.Series

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

Parameters times (iterable or float)

Returns

Return type pd.Series

LogLogisticFitter

class lifelines.fitters.log_logistic_fitter.LogLogisticFitter(*args, **kwargs)

Bases: lifelines.fitters.KnownModelParametricUnivariateFitter

This class implements a Log-Logistic model for univariate data. The model has parameterized form:

\[ S(t) = \left(1 + \left(\frac{t}{\alpha}\right)^\beta\right)^{-1}, \alpha > 0, \beta > 0, \]

The \( \alpha \) (scale) parameter has an interpretation as being equal to the median lifetime of the population. The \( \beta \) parameter influences the shape of the hazard. See figure below:
The hazard rate is:

\[ h(t) = \frac{\beta (t/\alpha)^{\beta - 1}}{1 + (t/\alpha)^{\beta}} \]

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

**hazard_**

- The estimated hazard (with custom timeline if provided)
  - Type: DataFrame

**survival_function_**

- The estimated survival function (with custom timeline if provided)
  - Type: DataFrame

**cumulative_density_**

- The estimated cumulative density function (with custom timeline if provided)
  - Type: DataFrame

**density**

- The estimated density function (PDF) (with custom timeline if provided)
  - Type: DataFrame
**variance_matrix**

The variance matrix of the coefficients

*Type* numpy array

**median_survival_time**

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

*Type* array or None

**conditional_time_to_event**

Return a DataFrame, with index equal to survival_function, 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.

**confidence_interval**

The confidence interval of the cumulative hazard. This is an alias for confidence_interval_cumulative_hazard.

**confidence_interval_cumulative_density**

The lower and upper confidence intervals for the cumulative density

**confidence_interval_cumulative_hazard**

The confidence interval of the cumulative hazard. This is an alias for confidence_interval.

**confidence_interval_density**

The confidence interval of the hazard.

**confidence_interval_hazard**

The confidence interval of the hazard.

**confidence_interval_survival_function**

The lower and upper confidence intervals for the survival function
cumulative_density_at_times\((\text{times}, \text{label}=\text{None})\) → pandas.core.series.Series

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\((\text{times}, \text{label}=\text{None})\) → pandas.core.series.Series

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.

density_at_times\((\text{times}, \text{label}=\text{None})\) → pandas.core.series.Series

Return a Pandas series of the predicted probability density function, \(dCDF/dt\), 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.

divide\((\text{other})\) → pandas.core.frame.DataFrame

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

Parameters other (same object as self)

event_table

fit\((\text{durations}, \text{event_observed}=\text{None}, \text{timeline}=\text{None}, \text{label}=\text{None}, \text{alpha}=\text{None}, \text{ci_labels}=\text{None}, \text{show_progress}=\text{False}, \text{entry}=\text{None}, \text{weights}=\text{None}, \text{initial_point}=\text{None})\) → ParametricUnivariateFitter

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) – return the estimate at the values in timeline (positively increasing)
- alpha (float, optional) – a string to name the column of the estimate.
- ci_labels (list, optional) – the alpha value in the confidence intervals. Overrides the initializing alpha for this call to fit only.
- show_progress (bool, 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>\n- 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
• **initial_point** ((d,) numpy array, optional) – initialize the starting point of the iterative algorithm. Default is the zero vector.

**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, initial_point=None) → ParametricUnivariateFitter

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: `<label>_lower_<alpha>`

• **show_progress** (bool, 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

• **initial_point** ((d,) numpy array, optional) – initialize the starting point of the iterative algorithm. Default is the zero vector.

**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, initial_point=None) → ParametricUnivariateFitter

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** *(bool, 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

• **initial_point** *(d,) numpy array, optional)* – initialize the starting point of the iterative algorithm. Default is the zero vector.

Returns

Return type self with new properties like cumulative_hazard_, survival_function_

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

Alias for fit

See also:

fit

**hazard_at_times** *(times, label=None)* → pandas.core.series.Series

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.

**median_survival_time**

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.

**percentile** *(p)*

Return the unique time point, t, such that S(t) = p.

Parameters **p** *(float)*

**plot** (**kwargs**)

Produce a pretty-plot of the estimate.

**plot_cumulative_density** (**kwargs**)

**plot_cumulative_hazard** (**kwargs**)

**plot_density** (**kwargs**)

1.14. API Reference
plot_hazard(**kwargs)

plot_survival_function(**kwargs)

predict(times: Union[Iterable[float], float], interpolate=False) ➞ pandas.core.series.Series
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** (bool, 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.

print_summary(decimals=2, style=None, **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
- **style** (string) – {html, ascii, latex}
- **kwargs** – print additional metadata in the output (useful to provide model names, dataset names, etc) when comparing multiple outputs.

subtract(other) ➞ pandas.core.frame.DataFrame
Subtract the {0} of two {1} objects.

Parameters other (same object as self)

summary
Summary statistics describing the fit.

See also:

print_summary

survival_function_at_times(times, label=None) ➞ pandas.core.series.Series
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.

LogNormalFitter

class lifelines.fitters.log_normal_fitter.LogNormalFitter(*args, **kwargs)
Bases: lifelines.fitters.KnownModelParametricUnivariateFitter

This class implements an Log Normal model for univariate data. The model has parameterized form:

\[ S(t) = 1 - \Phi \left( \frac{\log(t) - \mu}{\sigma} \right), \quad \sigma > 0 \]

where \( \Phi \) is the CDF of a standard normal random variable. This implies the cumulative hazard rate is

\[ H(t) = -\log \left( 1 - \Phi \left( \frac{\log(t) - \mu}{\sigma} \right) \right) \]

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.

**cumulative_hazard**

The estimated cumulative hazard (with custom timeline if provided)

*Type* DataFrame

**hazard**

The estimated hazard (with custom timeline if provided)

*Type* DataFrame

**survival_function**

The estimated survival function (with custom timeline if provided)

*Type* DataFrame

**cumulative_density**

The estimated cumulative density function (with custom timeline if provided)

*Type* DataFrame

**density**

The estimated density function (PDF) (with custom timeline if provided)

*Type* DataFrame

**variance_matrix**

The variance matrix of the coefficients

*Type* numpy array

**median_survival_time**

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 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_, 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.

confidence_interval_
  The confidence interval of the cumulative hazard. This is an alias for confidence_interval_cumulative_hazard_.

certainty_interval_cumulative_density_
  The lower and upper confidence intervals for the cumulative density

confidence_interval_cumulative_hazard_
  The confidence interval of the cumulative hazard. This is an alias for confidence_interval_.

certainty_interval_density_
  The confidence interval of the hazard.

certainty_interval_hazard_
  The confidence interval of the hazard.

certainty_interval_survival_function_
  The lower and upper confidence intervals for the survival function

cumulative_density_at_times (times, label=None) → pandas.core.series.Series
  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) → pandas.core.series.Series
  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.

density_at_times (times, label=None) → pandas.core.series.Series
  Return a Pandas series of the predicted probability density function, dCDF/dt, 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.

divide (other) → pandas.core.frame.DataFrame
  Divide the {0} of two {1} objects.

  Parameters other (same object as self)

event_table

fit (durations, event_observed=None, timeline=None, label=None, alpha=None, ci_labels=None, show_progress=False, entry=None, weights=None, initial_point=None) → ParametricUnivariateFitter

  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 initial-izing 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** *(bool, 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

• **initial_point** *(d,) numpy array, optional)* – initialize the starting point of the iterative algorithm. Default is the zero vector.

**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, initial_point=None) → ParametricUnivariateFitter
```

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 initial-izing 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** *(bool, 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

• **initial_point** *(d,)* *(numpy array, optional)* – initialize the starting point of the iterative algorithm. Default is the zero vector.

**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, initial_point=None) → ParametricUnivariateFitter

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], [alpha]

• **show_progress** *(bool, 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

• **initial_point** *(d,)* *(numpy array, optional)* – initialize the starting point of the iterative algorithm. Default is the zero vector.

**Returns**

self with new properties like `cumulative_hazard_`, `survival_function_`

**Return type**

declare type

---

### fit_right_censoring
dur, event_observed, timeline, label, alpha, ci_labels, show_progress, entry, weights, initial_point

Alias for fit

**See also:**

fit
hazard_at_times(times, label=None) → pandas.core.series.Series
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.

median_survival_time_
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.

percentile(p) → float
Return the unique time point, t, such that S(t) = p.

Parameters p (float)

plot(**kwargs)
Produce a pretty-plot of the estimate.

plot_cumulative_density(**kwargs)
plot_cumulative_hazard(**kwargs)
plot_density(**kwargs)
plot_hazard(**kwargs)
plot_survival_function(**kwargs)
predict(times: Union[Iterable[float], float], interpolate=False) → pandas.core.series.Series
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 (bool, 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.

print_summary(decimals=2, style=None, **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
- style (string) – {html, ascii, latex}
- kwargs – print additional metadata in the output (useful to provide model names, dataset names, etc.) when comparing multiple outputs.

subtract(other) → pandas.core.frame.DataFrame
Subtract the {0} of two {1} objects.

Parameters other (same object as self)

summary
Summary statistics describing the fit.

See also:
print_summary
survival_function_at_times (times, label=None) → pandas.core.series.Series

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.

MixtureCureFitter
class lifelines.fitters.mixture_cure_fitter.MixtureCureFitter (base_fitter, *args, **kwargs)

Bases: lifelines.fitters.ParametricUnivariateFitter

This class implements a Mixture Cure Model for univariate data with a configurable distribution for the non-cure portion. The model survival function has parameterized form:

\[ S(t) = c + (1 - c) S_b(t), \quad 1 > c > 0 \]

where \( S_b(t) \) is a parametric survival function describing the non-cure portion of the population, and \( c \) is the cured fraction of the population.

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(). The parameters for both the cure portion of the model and from the base_fitter are available. The cure fraction is called cured_fraction_, and parameters from the base_fitter will be available with their own appropriate names.

Parameters

- base_fitter (ParametricUnivariateFitter, required) – an instance of a fitter that describes the non-cure portion of the population.
- alpha (float, optional (default=0.05)) – the level in the confidence intervals.

Important: The base_fitter instance is used to describe the non-cure portion of the population, but is not actually fit to the data. Some internal properties are modified, and it should not be used for any other purpose after passing it to the constructor of this class.

Examples

```python
from lifelines import MixtureCureFitter, ExponentialFitter

fitter = MixtureCureFitter(base_fitter=ExponentialFitter())
fitter.fit(T, event_observed=observed)
print(fitter.cured_fraction_) # This is available because it is a parameter of the
print(fitter.lambda_)       # This is available because it is a parameter of the
```

cumulative_hazard_
The estimated cumulative hazard (with custom timeline if provided)

Type DataFrame

cured_fraction_
The fitted parameter \( c \) in the model
Type float

hazard_
    The estimated hazard (with custom timeline if provided)
    Type DataFrame

survival_function_
    The estimated survival function (with custom timeline if provided)
    Type DataFrame

cumulative_density_
    The estimated cumulative density function (with custom timeline if provided)
    Type DataFrame

variance_matrix_
    The variance matrix of the coefficients
    Type numpy array

median_survival_time_
    The median time to event
    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_, 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.

certainty_Interval_
    The confidence interval of the cumulative hazard. This is an alias for certainty_interval_cumulative_hazard_.

confidence_interval_cumulative_density_
    The lower and upper confidence intervals for the cumulative density

confidence_interval_cumulative_hazard_
    The confidence interval of the cumulative hazard. This is an alias for confidence_interval_.

confidence_interval_density_
    The confidence interval of the hazard.
confidence_interval_hazard
The confidence interval of the hazard.

certainty_interval_survival_function
The lower and upper confidence intervals for the survival function

cumulative_density_at_times(times, label=None) → pandas.core.series.Series
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) → pandas.core.series.Series
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.

density_at_times(times, label=None) → pandas.core.series.Series
Return a Pandas series of the predicted probability density function, dCDF/dt, 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.

divide (other) → pandas.core.frame.DataFrame
Divide the {0} of two {1} objects.

Parameters other (same object as self)

event_table

fit (durations, event_observed=None, timeline=None, label=None, alpha=None, ci_labels=None, show_progress=False, entry=None, weights=None, initial_point=None) → ParametricUnivariateFitter

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 (bool, 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

• **initial_point** *((d,) numpy array, optional)* – initialize the starting point of the iterative algorithm. Default is the zero vector.

**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, initial_point=None) → ParametricUnivariateFitter*

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: <label>_lower_<alpha>

• **show_progress** *(bool, 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

• **initial_point** *((d,) numpy array, optional)* – initialize the starting point of the iterative algorithm. Default is the zero vector.

**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, initial_point=None) → ParametricUnivariateFitter*

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** *(bool, 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
- **initial_point** *((d,), numpy array, optional)* – initialize the starting point of the iterative algorithm. Default is the zero vector.

Returns

Return type self with new properties like cumulative_hazard_, survival_function_

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

Alias for fit

See also:

fit

**hazard_at_times** *(times, label=None)* → pandas.core.series.Series

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.

**median_survival_time**

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.

**percentile** *(p)*

Return the unique time point, t, such that S(t) = p.

Parameters p *(float)*

**plot** *(**kwargs)*

Produce a pretty-plot of the estimate.

**plot_cumulative_density** *(**kwargs)*
plot_cumulative_hazard(**kwargs)

plot_density(**kwargs)

plot_hazard(**kwargs)

plot_survival_function(**kwargs)

predict(times: Union[Iterable[float], float], interpolate=False) \rightarrow pandas.core.series.Series

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 (bool, 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.

print_summary(decimals=2, style=None, **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
- style (string) – {html, ascii, latex}
- kwargs – print additional metadata in the output (useful to provide model names, dataset names, etc.) when comparing multiple outputs.

subtract(other) \rightarrow pandas.core.frame.DataFrame

Subtract the \( \{0\} \) of two \( \{1\} \) objects.

Parameters other (same object as self)

summary

Summary statistics describing the fit.

See also:

print_summary

survival_function_at_times(times, label=None) \rightarrow pandas.core.series.Series

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.

NelsonAalenFitter

class lifelines.fitters.nelson_aalen_fitter.NelsonAalenFitter(alpha=0.05, nelson_aalen_smoothing=True, **kwargs)

Bases: lifelines.fitters.UnivariateFitter

Class for fitting the Nelson-Aalen estimate for the cumulative hazard.

NelsonAalenFitter(alpha=0.05, nelson_aalen_smoothing=True)
• **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_ , 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.

cumulative_density_at_times (*times, label=None*)  
cumulative_hazard_at_times (*times, label=None*)  
divide (*other*) → pandas.core.frame.DataFrame  
Divide the {0} of two {1} objects.
  
  **Parameters** *other (same object as self)*  
fit (*durations, event_observed=None, timeline=None, entry=None, label=None, 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 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*)

Alias for `fit`

**See also:**

`fit`

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

**median_survival_time_**

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.

**percentile** *(p)*

Return the unique time point, t, such that S(t) = p.

**Parameters** p *(float)*

**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.

**Returns** a pyplot axis object

**Return type** ax

- **plot_cumulative_density**(**kwargs**)
- **plot_cumulative_hazard**(**kwargs**)
- **plot_density**(**kwargs**)
- **plot_hazard**(bandwidth=None, **kwargs**)
- **plot_survival_function**(**kwargs**)

**predict**(times: Union[Iterable[float], float], interpolate=False) → pandas.core.series.Series

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** *(bool, 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.

**smoothed_hazard_**(bandwidth)

**Parameters** **bandwidth** *(float)* – the bandwith 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) → pandas.core.frame.DataFrame

Subtract the `{0}` of two `{1}` objects.

**Parameters** **other** *(same object as self)*

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

```python
class lifelines.fitters.piecewise_exponential_fitter.PiecewiseExponentialFitter(breakpoints, *args, **kwargs)
```

**Bases:** lifelines.fitters.KnownModelParametricUnivariateFitter

This class implements a Piecewise Exponential model for univariate data. The model has parameterized hazard rate:

\[
 h(t) = \begin{cases} 
 \frac{1}{\lambda_0} & \text{if } t \leq \tau_0 \\
 \frac{1}{\lambda_1} & \text{if } \tau_0 < t \leq \tau_1 \\
 \frac{1}{\lambda_2} & \text{if } \tau_1 < t \leq \tau_2 \\
 \end{cases} 
\]

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

### hazard_

The estimated hazard (with custom timeline if provided)

**Type** DataFrame

### survival_function_

The estimated survival function (with custom timeline if provided)

**Type** DataFrame

### cumulative_density_

The estimated cumulative density function (with custom timeline if provided)

**Type** DataFrame

### density

The estimated density function (PDF) (with custom timeline if provided)

**Type** DataFrame

### variance_matrix_

The variance matrix of the coefficients

**Type** numpy array

### median_survival_time_

The median time to event
**Type** float

`lambda_i_`

The fitted parameter in the model, for \( i = 0, 1 \ldots n-1 \) 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

**Type** array

`conditional_time_to_event_`

Return a DataFrame, with index equal to `survival_function_`, 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.

`confidence_interval_`

The confidence interval of the cumulative hazard. This is an alias for `confidence_interval_cumulative_hazard_`.

`confidence_interval_cumulative_density_`

The lower and upper confidence intervals for the cumulative density

`confidence_interval_cumulative_hazard_`

The confidence interval of the cumulative hazard. This is an alias for `confidence_interval_`.

`confidence_interval_density_`

The confidence interval of the hazard.

`confidence_interval_hazard_`

The confidence interval of the hazard.

`confidence_interval_survival_function_`

The lower and upper confidence intervals for the survival function

`cumulative_density_at_times(times, label=None) → pandas.core.series.Series`

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) → pandas.core.series.Series

Parameters

- **times** (iterable or float) – values to return the cumulative hazard at.
- **label** (string, optional) – Rename the series returned. Useful for plotting.

density_at_times(times, label=None) → pandas.core.series.Series

Parameters

- **times** (iterable or float) – values to return the survival function at.
- **label** (string, optional) – Rename the series returned. Useful for plotting.

divide(other) → pandas.core.frame.DataFrame

Divide the 0 of two 1 objects.

Parameters

- **other** (same object as self)

event_table

fit(durations, event_observed=None, timeline=None, label=None, alpha=None, ci_labels=None, show_progress=False, entry=None, weights=None, initial_point=None) → ParametricUnivariateFitter

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** (bool, 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
- **initial_point** ((d,) numpy array, optional) – initialize the starting point of the iterative algorithm. Default is the zero vector.

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, initial_point=None) → ParametricUnivariateFitter

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: <label>_lower_<alpha>*
- **show_progress** *(bool, 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.*
- **initial_point** *(d,) numpy array, optional) – initialize the starting point of the iterative algorithm. Default is the zero vector.*

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, initial_point=None) → ParametricUnivariateFitter

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** *(bool, 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

• **initial_point** *(d,) numpy array, optional)* – initialize the starting point of the iterative algorithm. Default is the zero vector.

Returns

Return type self with new properties like cumulative_hazard_, survival_function_

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

Alias for fit

See also:

```python
fit
```

```python
hazard_at_times(times, label=None) → pandas.core.series.Series
```

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.

```python
median_survival_time()
```

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
percentile(p: float) → float
```

Return the unique time point, t, such that S(t) = p.

Parameters **p** *(float)*

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

Produce a pretty-plot of the estimate.

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

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

```python
plot_density(**kwargs)
```

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

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

```python
predict(times: Union[Iterable[float], float], interpolate=False) → pandas.core.series.Series
```

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** (*bool, 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.

**print_summary**(*decimals=2, style=None, **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
- **style** (*string*) – {html, ascii, latex}
- **kwargs** – print additional metadata in the output (useful to provide model names, dataset names, etc.) when comparing multiple outputs.

**subtract**(*other*) → pandas.core.frame.DataFrame

Subtract the \(0\) of two \(1\) objects.

**Parameters**

- **other** (*same object as self*)

**summary**

Summary statistics describing the fit.

See also:

**print_summary**

**survival_function_at_times**(*times, label=None*) → pandas.core.series.Series

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.

**SplineFitter**

**class** lifelines.fitters.spline_fitter.SplineFitter(*knot_locations: auto-grad.numpy.numpy_wrapper.array, *args, **kwargs*)

**Bases** lifelines.fitters.KnownModelParametricUnivariateFitter, lifelines.fitters.mixins.SplineFitterMixin

Model the cumulative hazard using \(N\) cubic splines. This offers great flexibility and smoothness of the cumulative hazard.

\[
H(t) = \exp \left( \phi_0 + \phi_1 \log t + \sum_{j=2}^{N} \phi_j v_j \log t \right)
\]

where \(v_j\) are our cubic basis functions at predetermined knots. See references for exact definition.

**Parameters**

- **knot_locations** (*list, np.array*) – The locations of the cubic breakpoints. Typically, the first knot is the minimum observed death, the last knot is the maximum observed death, and the knots in between are the centiles of observed data (ex: if one additional knot, choose the 50th percentile, the median. If two additional knots, choose the 33rd and 66th percentiles).
References


Examples

```python
from lifelines import SplineFitter
from lifelines.datasets import load_waltons
waltons = load_waltons()

T, E = waltons['T'], waltons['E']
knots = np.percentile(T.loc[E.astype(bool)], [0, 50, 100])

sf = SplineFitter(knots)
sf.fit(T, E)
sf.plot()
print(sf.knots)
```

cumulative_hazard_
  The estimated cumulative hazard (with custom timeline if provided)
    Type DataFrame

hazard_
  The estimated hazard (with custom timeline if provided)
    Type DataFrame

survival_function_
  The estimated survival function (with custom timeline if provided)
    Type DataFrame

cumulative_density_
  The estimated cumulative density function (with custom timeline if provided)
    Type DataFrame

density
  The estimated density function (PDF) (with custom timeline if provided)
    Type DataFrame

variance_matrix_
  The variance matrix of the coefficients
    Type numpy array

median_survival_time_
  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
event
    The entry array provided, or None
    Type array or None
knot_locations
    The locations of the cubic breakpoints.
basis (x: numpy.array, knot: float, min_knot: float, max_knot: float)
 conditional_time_to_event_
    Return a DataFrame, with index equal to survival_function_, 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.
confidence_interval_
    The confidence interval of the cumulative hazard. This is an alias for confidence_interval_cumulative_hazard_.
confidence_interval_cumulative_density_
    The lower and upper confidence intervals for the cumulative density
confidence_interval_cumulative_hazard_
    The confidence interval of the cumulative hazard. This is an alias for confidence_interval_.
confidence_interval_density_
    The confidence interval of the hazard.
confidence_interval_hazard_
    The confidence interval of the hazard.
confidence_interval_survival_function_
    The lower and upper confidence intervals for the survival function
 cumulative_density_at_times (times, label=None) → pandas.core.series.Series
    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) → pandas.core.series.Series
    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.

density_at_times(times, label=None) → pandas.core.series.Series
   Return a Pandas series of the predicted probability density function, dCDF/dt, 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.

divide(other) → pandas.core.frame.DataFrame
   Divide the {0} of two {1} objects.

   Parameters other (same object as self)

event_table

fit(durations, event_observed=None, timeline=None, label=None, alpha=None, ci_labels=None, show_progress=False, entry=None, weights=None, initial_point=None) → ParametricUnivariateFitter

   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 initial-
     izing 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: <la-
     bel>_lower_<alpha>
   • show_progress (bool, 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
   • initial_point ((d,) numpy array, optional) – initialize the starting point of the iterative algorithm. Default is the zero vector.

   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, initial_point=None) → ParametricUnivariateFitter

   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: <label>_lower_<alpha>

• **show_progress** *(bool, 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

• **initial_point** *(d,) numpy array, optional* – initialize the starting point of the iterative algorithm. Default is the zero vector.

**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, initial_point=None) → ParametricUnivariateFitter*

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** *(bool, 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

• **initial_point** *(\(d,\) numpy array, optional)* – initialize the starting point of the iterative algorithm. Default is the zero vector.

Returns

Return type

self with new properties like `cumulative_hazard_`, `survival_function_`

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

Alias for `fit`

See also:

`fit`

**hazard_at_times** *(times, label=None)* → pandas.core.series.Series

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.

**median_survival_time_**

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.

**percentile** *(p: float)* → float

Return the unique time point, \(t\), such that \(S(t) = p\).

Parameters **p** *(float)*

**plot** *(**kwargs)*

Produce a pretty-plot of the estimate.

**plot_cumulative_density** *(**kwargs)*

**plot_cumulative_hazard** *(**kwargs)*

**plot_density** *(**kwargs)*

**plot_hazard** *(**kwargs)*

**plot_survival_function** *(**kwargs)*

**predict** *(times: Union[Iterable[float], float], interpolate=False)* → pandas.core.series.Series

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** *(bool, 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.
print_summary (decimals=2, style=None, **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
- style (string) – {html, ascii, latex}
- kwargs – print additional metadata in the output (useful to provide model names, dataset names, etc.) when comparing multiple outputs.

static relu (x: numpy.array)

subtract (other) → pandas.core.frame.DataFrame
Subtract the {0} of two {1} objects.

Parameters other (same object as self)

summary
Summary statistics describing the fit.

See also:
print_summary

survival_function_at_times (times, label=None) → pandas.core.series.Series
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.

WeibullFitter

class lifelines.fitters.weibull_fitter.WeibullFitter (*args, **kwargs)

Bases: lifelines.fitters.KnownModelParametricUnivariateFitter

This class implements a Weibull model for univariate data. The model has parameterized form:

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

The \( \lambda \) (scale) parameter has an applicable interpretation: it represent the time when 37% of the population has died. The \( \rho \) (shape) parameter controls if the cumulative hazard (see below) is convex or concave, representing accelerating or decelerating hazards.

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_**
  - The estimated cumulative hazard (with custom timeline if provided)
  - Type: `DataFrame`

- **hazard_**
  - The estimated hazard (with custom timeline if provided)
  - Type: `DataFrame`

- **survival_function_**
  - The estimated survival function (with custom timeline if provided)
  - Type: `DataFrame`

- **cumulative_density_**
  - The estimated cumulative density function (with custom timeline if provided)
  - Type: `DataFrame`

- **density_**
  - The estimated density function (PDF) (with custom timeline if provided)
  - Type: `DataFrame`

- **variance_matrix_**
  - The variance matrix of the coefficients
  - Type: `numpy array`

- **median_survival_time_**
  - 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

Notes
Looking for a 3-parameter Weibull model? See notes here.

conditional_time_to_event_
Return a DataFrame, with index equal to survival_function_, 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.

confidence_interval_
The confidence interval of the cumulative hazard. This is an alias for confidence_interval_cumulative_hazard_.

confidence_interval_cumulative_density_
The lower and upper confidence intervals for the cumulative density

confidence_interval_cumulative_hazard_
The confidence interval of the cumulative hazard. This is an alias for confidence_interval_.

confidence_interval_density_
The confidence interval of the hazard.

confidence_interval_hazard_
The confidence interval of the hazard.

confidence_interval_survival_function_
The lower and upper confidence intervals for the survival function

cumulative_density_at_times(times, label=None) → pandas.core.series.Series
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) \rightarrow \text{pandas.core.series.Series}

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.

density_at_times (times, label=None) \rightarrow \text{pandas.core.series.Series}

Return a Pandas series of the predicted probability density function, dCDF/dt, 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.

divide (other) \rightarrow \text{pandas.core.frame.DataFrame}

Divide the \{0\} of two \{1\} objects.

Parameters

other (same object as self)

event_table

fit (durations, event_observed=None, timeline=None, label=None, alpha=None, ci_labels=None, show_progress=False, entry=None, weights=None, initial_point=None) \rightarrow \text{ParametricUnivariateFitter}

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) – return the estimate at the values in timeline (positively increasing)

• alpha (float, optional) – a string to name the column of the estimate.

• ci_labels (list, 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 (bool, 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

• initial_point ((d,) numpy array, optional) – initialize the starting point of the iterative algorithm. Default is the zero vector.

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, initial_point=None) → ParametricUnivariateFitter

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** *(bool, 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
- **initial_point** *(d,) numpy array, optional)* – initialize the starting point of the iterative algorithm. Default is the zero vector.

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, initial_point=None) → ParametricUnivariateFitter

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 initial-
izing alpha for this call to fit only.

• **ci_labels (list, optional)** – add custom column names to the generated confidence in-
tervals as a length-2 list: [<lower-bound name>, <upper-bound name>]. Default: <la-
bel>_lower_<alpha>

• **show_progress (bool, 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

• **initial_point ((d,) numpy array, optional)** – initialize the starting point of the iterative
algorithm. Default is the zero vector.

**Returns**

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

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

Alias for fit

**See also:**

• fit

**hazard_at_times (times, label=None) → pandas.core.series.Series**

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.

**median_survival_time**

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.

**percentile (p) → float**

Return the unique time point, t, such that S(t) = p.

**Parameters** p (float)

**plot (**kwargs**)**

Produce a pretty-plot of the estimate.

**plot_cumulative_density (**kwargs**)**

**plot_cumulative_hazard (**kwargs**)**

**plot_density (**kwargs**)**

**plot_hazard (**kwargs**)**

**plot_survival_function (**kwargs**)**

**predict (times: Union[Iterable[float], float], interpolate=False) → pandas.core.series.Series**

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** *(bool, 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.

**print_summary** *(decimals=2, style=None, **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

• **style** *(string)* – {html, ascii, latex}

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

**subtract** *(other) → pandas.core.frame.DataFrame)*
Subtract the {0} of two {1} objects.

Parameters **other** *(same object as self)*

**summary**
Summary statistics describing the fit.

See also:
print_summary

**survival_function_at_times** *(times, label=None) → pandas.core.series.Series)*
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.

**Regression models**

**AalenAdditiveFitter**

class lifelines.fitters.aalen_additive_fitter.AalenAdditiveFitter *(fit_intercept=True, alpha=0.05, coef_penalizer=0.0, smooth_penalizer=0.0)*

Bases: lifelines.fitters.RegressionFitter

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

Note: This class was rewritten in lifelines 0.17.0 to focus solely on static datasets. There is no guarantee of backwards compatibility.
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 magnitude 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 magnitude 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

**event_observed**

The event_observed variable provided

Type array

**weights**

The event_observed variable provided

Type array

**compute_residuals** *(training_dataframe: pandas.core.frame.DataFrame, kind: str) → pandas.core.frame.DataFrame*

Compute the residuals the model.

Parameters

- **training_dataframe** *(DataFrame) – the same training DataFrame given in fit*


**fit** *(df, duration_col=None, 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** *(bool, 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, 0, 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],
    })

aaf = AalenAdditiveFitter()
aaf.fit(df, 'T', 'E')
aaf.predict_median(df)
aaf.print_summary()
```

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

Alias for `fit`

**See also:**

- `fit`

**plot** *(columns=None, loc=None, iloc=None, ax=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 \( (X) \rightarrow \text{pandas.core.series.Series} \\
\) 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 \( (X) \rightarrow \text{pandas.core.series.Series} \\
\) 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 \( (X, p=0.5) \rightarrow \text{pandas.core.series.Series} \\
\) Returns the median lifetimes for the individuals. [http://stats.stackexchange.com/questions/102986/percentile-loss-functions](http://stats.stackexchange.com/questions/102986/percentile-loss-functions)

**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 \( (X, \text{times=None}) \rightarrow \text{pandas.core.series.Series} \\
\) 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.

- \( \text{times} \) – Not implemented yet

print_summary \( (\text{decimals=2, style=None, **kwargs)} \rightarrow \) Print summary statistics describing the fit, the coefficients, and the error bounds.

**Parameters**

- \( \text{decimals} \) (int, optional (default=2)) – specify the number of decimal places to show

- \( \text{style} \) (string) – {html, ascii, latex}

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

score \( (df: \text{pandas.core.frame.DataFrame, scoring_method: str = 'log_likelihood'} \rightarrow \text{float} \\
\) Score the data in \( df \) on the fitted model. With default scoring method, returns the average partial log-likelihood.

**Parameters**

- \( df \) (DataFrame) – the dataframe with duration col, event col, etc.

- \( \text{scoring_method} \) (str) – one of {‘log_likelihood’, ‘concordance_index’} log_likelihood: returns the average unpenalized partial log-likelihood. concordance_index: returns the concordance-index
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

CoxPHFitter

class lifelines.fitters.coxph_fitter.CoxPHFitter (baseline_estimation_method: str = 'breslow', penalizer: float = 0.0, strata: Union[List[str], str, None] = None, ll_ratio: float = 0.0, n_baseline_knots: int = 1, **kwargs)

Bases: lifelines.fitters.RegressionFitter, lifelines.fitters.mixins.ProportionalHazardMixin

This class implements fitting Cox’s proportional hazard model using Efron’s method for ties.

\[ h(t|x) = h_0(t) \exp((x - \pi)\beta) \]

The baseline hazard, \( h_0(t) \) can be modeled non-parametrically (using Breslow’s method) or parametrically (using cubic splines). This is specified using the baseline_estimation_method parameter.

Parameters

- alpha (float, optional (default=0.05)) – the level in the confidence intervals.
- baseline_estimation_method (string, optional) – specify how the fitter should estimate the baseline. "breslow" or "spline"
- penalizer (float, optional (default=0.0)) – Attach a penalty 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 magnitude value of \( \beta_i \). See ll_ratio below. The penalty term is \( \frac{1}{2} \)penalizer \((1 - \)

Examples

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

params_
The estimated coefficients. Changed in version 0.22.0: use to be .hazards_
hazard_ratios_
The \(\exp(\text{coefficients})\)

confidence_intervals_
The lower and upper confidence intervals for the hazard coefficients

durations
The durations provided

event_observed
The event\(_{\text{observed}}\) variable provided

weights
The event\(_{\text{observed}}\) variable provided

variance_matrix_
The variance matrix of the coefficients

strata
the strata provided

standard_errors_
the standard errors of the estimates

baseline_hazard_
Type DataFrame

baseline_cumulative_hazard_
Type DataFrame

baseline_survival_
Type DataFrame

check_assumptions (training\_df: pandas.core.frame.DataFrame, advice: bool = True, show_plots: bool = False, p\_value\_threshold: float = 0.01, plot\_n\_bootstraps: int = 10, columns: Optional[List[str]] = None) \(\rightarrow\) None

Use this function to test the proportional hazards assumption. See usage example at https://lifelines.readthedocs.io/en/latest/jupyter_notebooks/Proportional\_hazard\_assumption.html

Parameters
- \textbf{training\_df} (DataFrame) – the original DataFrame used in the call to \texttt{fit(...) or a sub-sampled version.}
- \textbf{advice} (bool, optional) – display advice as output to the user’s screen
• **show_plots** *(bool, 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.

• **columns** *(list, optional)* – specify a subset of columns to test.

### 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


### compute_followup_hazard_ratios

```python
def compute_followup_hazard_ratios(training_df: pandas.core.frame.DataFrame, followup_times: Iterable[T_co]) -> pandas.core.frame.DataFrame
```

Recompute the hazard ratio at different follow-up times (lifelines handles accounting for updated censoring and updated durations). This is useful because we need to remember that the hazard ratio is actually a weighted-average of period-specific hazard ratios.

#### Parameters

- **training_df** *(pd.DataFrame)* – The same dataframe used to train the model
- **followup_times** *(Iterable)* – a list/array of follow-up times to recompute the hazard ratio at.

### compute_residuals

```python
def compute_residuals(training_dataframe: pandas.core.frame.DataFrame, kind: str) -> pandas.core.frame.DataFrame
```

Compute the residuals the model.

#### Parameters
• **training_dataframe** *(DataFrame)* – the same training DataFrame given in `fit`

• **kind** *(string)* – {'schoenfeld', 'score', 'delta_beta', 'deviance', 'martingale', 'scaled_schoenfeld'}

**concordance_index**

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 censoring.

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


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** *(bool, 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 n 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** *(bool, 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

- **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')
cph.print_summary()
cph.predict_median(df)
```

```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],
     '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],
    })

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)*

Alias for fit

**See also:**

fit
hazard_ratios_

log_likelihood_ratio_test() → lifelines.statistics.StatisticalResult
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, hazard_ratios=False, ax=None, **errorbar_kwargs)
Produces a visual representation of the coefficients (i.e. log hazard ratios), including their standard errors and magnitudes.

Parameters
- columns (list, optional) – specify a subset of the columns to plot
- hazard_ratios (bool, optional) – by default, plot will present the log-hazard ratios (the coefficients). However, by turning this flag to True, the hazard ratios are presented instead.
- errorbar_kwargs – pass in additional plotting commands to matplotlib errorbar command

Examples

```python
from lifelines import datasets, CoxPHFitter
rossi = datasets.load_rossi()
cph = CoxPHFitter().fit(rossi, 'week', 'arrest')
cph.plot(hazard_ratios=True)
```

Returns ax – the matplotlib axis that be edited.

Return type matplotlib axis

plot_covariate_groups(covariates, values, plot_baseline=True, **kwargs)
Produces a plot 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 specific 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, 3), 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 do the following to see the effect of all the categories on one plot.
cph.plot_covariate_groups(["dummy1", "dummy2", "dummy3"], values=[[1, 0, 0],
→[0, 1, 0], [0, 0, 1]])
# same as:
cph.plot_covariate_groups(["dummy1", "dummy2", "dummy3"], values=np.eye(3))

## predict_cumulative_hazard

\[
\text{predict\_cumulative\_hazard}(X: \text{Union[pandas.core.series.Series, pandas.core.frame.DataFrame]}, \\
times: \text{Union[numpy.ndarray, List[float], None]} = \text{None}, \text{conditional\_after}: \text{Optional[List[int]]} = \text{None}) \rightarrow \text{pandas.core.frame.DataFrame}
\]

**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.
- **conditional_after** (iterable, optional) – Must be equal is size to X.shape[0] (denoted n above). An iterable (array, list, series) of possibly non-zero values that represent how long the subject has already lived for. Ex: if T is the unknown event time, then this represents \(T|T > s\). This is useful for knowing the remaining hazard/survival of censored subjects. The new timeline is the remaining duration of the subject, i.e. normalized back to starting at 0.

## predict_expectation

\[
\text{predict\_expectation}(X: \text{pandas.core.frame.DataFrame}, \text{conditional\_after}: \text{Optional[numpy.ndarray]} = \text{None}) \rightarrow \text{pandas.core.series.Series}
\]

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:** If the survival function doesn’t converge to 0, then 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.
- **conditional_after** (iterable, optional) – Must be equal is size to X.shape[0] (denoted n above). An iterable (array, list, series) of possibly non-zero values that represent how long the subject has already lived for. Ex: if T is the unknown event time, then this represents \(T|T > s\). This is useful for knowing the remaining hazard/survival of censored subjects. The new timeline is the remaining duration of the subject, i.e. normalized back to starting at 0.

## 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:

\texttt{predict\_median()}, \texttt{predict\_percentile()}

\texttt{predict\_log\_partial\_hazard (X: Union[numpy.ndarray, pandas.core.frame.DataFrame]) \rightarrow pandas.core.series.Series}

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 $(x - \text{mean}(x_{\text{train}}))\beta$

\textbf{Parameters} \textit{X} (\textit{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.

\textbf{Notes}

If \textit{X} is a DataFrame, the order of the columns do not matter. But if \textit{X} is an array, then the column ordering is assumed to be the same as the training dataset.

\texttt{predict\_median (X: pandas.core.frame.DataFrame, conditional\_after: Optional[numpy.ndarray] = None) \rightarrow pandas.core.series.Series}

Predict the median lifetimes for the individuals. If the survival curve of an individual does not cross 0.5, then the result is infinity.

\textbf{Parameters}

\begin{itemize}
  \item \textit{X} (\textit{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.
  \item \textit{conditional\_after} (iterable, optional) – Must be equal is size to \textit{X}.shape[0] (denoted \textit{n} above). An iterable (array, list, series) of possibly non-zero values that represent how long the subject has already lived for. Ex: if \textit{T} is the unknown event time, then this represents $T|T > s$. This is useful for knowing the remaining hazard/survival of censored subjects. The new timeline is the remaining duration of the subject, i.e. normalized back to starting at 0.
\end{itemize}

See also:

\texttt{predict\_percentile()}  

\texttt{predict\_partial\_hazard (X: Union[numpy.ndarray, pandas.core.frame.DataFrame]) \rightarrow pandas.core.series.Series}

Returns the partial hazard for the individuals, partial since the baseline hazard is not included. Equal to $\exp(x - \text{mean}(x_{\text{train}}))\beta$

\textbf{Parameters} \textit{X} (\textit{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.

\textbf{Notes}

If \textit{X} is a DataFrame, the order of the columns do not matter. But if \textit{X} is an array, then the column ordering is assumed to be the same as the training dataset.

\texttt{predict\_percentile (X: pandas.core.frame.DataFrame, p: float = 0.5, conditional\_after: Optional[numpy.ndarray] = None) \rightarrow pandas.core.series.Series}

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. \url{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.*

- **conditional_after** *(iterable, optional)* – Must be equal is size to X.shape[0] (denoted n above). An iterable (array, list, series) of possibly non-zero values that represent how long the subject has already lived for. Ex: if \( T \) is the unknown event time, then this represents \( T | T > s \). This is useful for knowing the remaining hazard/survival of censored subjects. The new timeline is the remaining duration of the subject, i.e. normalized back to starting at 0.

See also:

`predict_median()`

`predict_survival_function (X: Union[pandas.core.series.Series, pandas.core.frame.DataFrame], times: Union[numpy.ndarray, List[float], None] = None, conditional_after: Optional[List[int]] = None) → pandas.core.frame.DataFrame`

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.

- **conditional_after** *(iterable, optional)* – Must be equal is size to X.shape[0] (denoted n above). An iterable (array, list, series) of possibly non-zero values that represent how long the subject has already lived for. Ex: if \( T \) is the unknown event time, then this represents \( T | T > s \). This is useful for knowing the remaining hazard/survival of censored subjects. The new timeline is the remaining duration of the subject, i.e. normalized back to starting at 0.

`print_summary (decimals: int = 2, style: Optional[str] = None, **kwargs) → None`

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*

- **style** *(string)* – {html, ascii, latex}

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

`sco** (df: pandas.core.frame.DataFrame, scoring_method: str = 'log_likelihood') → float`

Score the data in df on the fitted model. With default scoring method, returns the average partial log-likelihood.

Parameters

- **df** *(DataFrame)* – the dataframe with duration col, event col, etc.
• **scoring_method** *(str)* – one of {‘log_likelihood’, ‘concordance_index’} *log_likelihood*: returns the average unpenalized partial log-likelihood. *concordance_index*: returns the concordance-index

**Examples**

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

rossi_train = load_rossi().loc[:400]
rossi_test = load_rossi().loc[400:]
cph = CoxPHFitter().fit(rossi_train, 'week', 'arrest')
cph.score(rossi_train)
cph.score(rossi_test)
```

**summary**

Summary statistics describing the fit.

Returns *df*

Return type *DataFrame*

**CoxTimeVaryingFitter**

```python
class lifelines.fitters.cox_time_varying_fitter.CoxTimeVaryingFitter(alpha=0.05, penalizer=0.0, l1_ratio: float = 0.0, strata=None)
```

Bases: *lifelines.fitters.RegressionFitter*, *lifelines.fitters.mixins.ProportionalHazardMixin*

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

\[ h(t|x(t)) = h_0(t) \exp((x(t) - \bar{x})'\beta) \]

**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

**params**

The estimated coefficients. Changed in version 0.22.0: use to be .hazards_

Type *Series*

**hazard_ratios**

The exp(coefficients)

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

baseline_cumulative_hazard_
      Type DataFrame

baseline_survival_
      Type DataFrame

check_assumptions (training_df: pandas.core.frame.DataFrame, advice: bool = True, show_plots: bool = False, p_value_threshold: float = 0.01, plot_n_bootstraps: int = 10, columns: Optional[List[str]] = None) → None

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 (bool, optional) – display advice as output to the user’s screen

• show_plots (bool, 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.

• columns (list, optional) – specify a subset of columns to test.
Examples

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


```python
def compute_followup_hazard_ratios(training_df: pandas.core.frame.DataFrame,
                                    followup_times: Iterable[T_co]) → pandas.core.frame.DataFrame
```

Recompute the hazard ratio at different follow-up times (lifelines handles accounting for updated censoring and updated durations). This is useful because we need to remember that the hazard ratio is actually a weighted-average of period-specific hazard ratios.

Parameters

- `training_df` (pd.DataFrame) – The same dataframe used to train the model
- `followup_times` (Iterable) – a list/array of follow-up times to recompute the hazard ratio at.

```python
def compute_residuals(training_dataframe: pandas.core.frame.DataFrame,
                      kind: str) → pandas.core.frame.DataFrame
```

Compute the residuals the model.

Parameters

- `training_dataframe` (DataFrame) – the same training DataFrame given in `fit`
- `kind` (string) – {'schoenfeld', 'score', 'delta_beta', 'deviance', 'martingale', 'scaled_schoenfeld'}

```python
fit(df, event_col='start', stop_col='stop', weights_col=None, id_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).

• 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.

• id_col (string, optional) – A subject could have multiple rows in the DataFrame. This column contains the unique identifier per subject. If not provided, it’s up to the user to make sure that there are no violations.

• show_progress (since the fitter is iterative, show convergence) – diagnostics.

• robust (bool, 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 n 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.

• 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)

Alias for fit.

See also:

fit

hazard_ratios_

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.

Conveniently, we can actually use CoxPHFitter class to do most of the work.

plot (columns=None, ax=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
lifelines Documentation, Release 0.24.3

Returns \( \text{ax} \) – the matplotlib axis that be edited.

Return type matplotlib axis

\textbf{predict_expectation} \( (df, \text{conditional\_after=None}) \rightarrow \text{pandas.core.series.Series} \)

\textbf{predict_log_partial_hazard} \( (X) \rightarrow \text{pandas.core.series.Series} \)

This is equivalent to R’s \texttt{linear.predictors}. Returns the log of the partial hazard for the individuals, partial since the baseline hazard is not included. Equal to \( (x - \bar{x})' \beta \)

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

Return type DataFrame

\textbf{predict_median} \( (df, \text{conditional\_after=None}) \rightarrow \text{pandas.core.series.Series} \)

\textbf{predict_partial_hazard} \( (X) \rightarrow \text{pandas.core.series.Series} \)

Returns the partial hazard for the individuals, partial since the baseline hazard is not included. Equal to \( \exp (x - \bar{x})' \beta \)

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

Return type DataFrame

\textbf{predict_percentile} \( (df, *, p=0.5, \text{conditional\_after=None}) \rightarrow \text{pandas.core.series.Series} \)

\textbf{print_summary} \( (\text{decimals=2, style=None, **kwargs}) \)

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

Parameters

- **decimals** \( \text{int, optional (default=2)} \) – specify the number of decimal places to show
- **style** \( \text{string} \) – {html, ascii, latex}
- **kwargs** – print additional meta data in the output (useful to provide model names, dataset names, etc.) when comparing multiple outputs.

\textbf{score} \( (df: \text{pandas.core.frame.DataFrame, scoring\_method: str = 'log\_likelihood'}) \rightarrow \text{float} \)

\textbf{summary}

Summary statistics describing the fit.

Returns \( df \) – Contains columns \text{coef, np.exp(coef), se(coef), z, p, lower, upper}.

Return type DataFrame
GeneralizedGammaRegressionFitter

class lifelines.fitters.generalized_gamma_regression_fitter.GeneralizedGammaRegressionFitter

Bases: lifelines.fitters.ParametricRegressionFitter

This class implements a Generalized Gamma model for regression data. The model has parameterized form:

The survival function is:

\[
S(t; x) = \begin{cases} 
1 - \frac{\Gamma_{RL} \left( \frac{1}{\lambda^2}; \frac{\log(t) - \mu}{\sigma^2} \right)}{\Gamma_{RL} \left( \frac{1}{\lambda^2}; \frac{\log(t) - \mu}{\sigma^2} \right)} & \text{if } \lambda > 0 \\
\frac{\Gamma_{RL} \left( \frac{1}{\lambda^2}; \frac{\log(t) - \mu}{\sigma^2} \right)}{\lambda^2} & \text{if } \lambda \leq 0 
\end{cases}
\]

where \( \Gamma_{RL} \) is the regularized lower incomplete Gamma function, and \( \sigma = \sigma(x) = \exp(\alpha x^T), \lambda = \lambda(x) = \beta x^T, \mu = \mu(x) = \gamma x^T \).

This model has the Exponential, Weibull, Gamma and Log-Normal as sub-models, and thus can be used as a way to test which model to use:

1. When \( \lambda = 1 \) and \( \sigma = 1 \), then the data is Exponential.
2. When \( \lambda = 1 \) then the data is Weibull.
3. When \( \sigma = \lambda \) then the data is Gamma.
4. When \( \lambda = 0 \) then the data is Log-Normal.
5. When \( \lambda = -1 \) then the data is Inverse-Weibull.
6. When \( -\sigma = \lambda \) then the data is Inverse-Gamma.

After calling the .fit method, you have access to properties like: cumulative_hazard_, survival_function_,. A summary of the fit is available with the method print_summary().

**Important:** The parameterization implemented has \( \log \sigma \), thus there is a ln_sigma_ in the output. Exponentiate this parameter to recover \( \sigma \).

**Important:** This model is experimental. It’s API may change in the future. Also, it’s convergence is not very stable.

**Parameters**

\textbf{alpha} (float, optional (default=0.05)) – the level in the confidence intervals.

**Examples**

```python
from lifelines import GeneralizedGammaFitter
from lifelines.datasets import load_waltons
waltons = load_waltons()

ggf = GeneralizedGammaFitter()
ggf.fit(waltons['T'], waltons['E'])
```

(continues on next page)
<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>cumulative_hazard_</code></td>
<td>The estimated cumulative hazard (with custom timeline if provided)</td>
<td>DataFrame</td>
</tr>
<tr>
<td><code>hazard_</code></td>
<td>The estimated hazard (with custom timeline if provided)</td>
<td>DataFrame</td>
</tr>
<tr>
<td><code>survival_function_</code></td>
<td>The estimated survival function (with custom timeline if provided)</td>
<td>DataFrame</td>
</tr>
<tr>
<td><code>cumulative_density_</code></td>
<td>The estimated cumulative density function (with custom timeline if provided)</td>
<td>DataFrame</td>
</tr>
<tr>
<td><code>density</code></td>
<td>The estimated density function (PDF) (with custom timeline if provided)</td>
<td>DataFrame</td>
</tr>
<tr>
<td><code>variance_matrix_</code></td>
<td>The variance matrix of the coefficients</td>
<td>numpy array</td>
</tr>
<tr>
<td><code>median_</code></td>
<td>The median time to event</td>
<td>float</td>
</tr>
<tr>
<td><code>lambda_</code></td>
<td>The fitted parameter in the model</td>
<td>float</td>
</tr>
<tr>
<td><code>rho_</code></td>
<td>The fitted parameter in the model</td>
<td>float</td>
</tr>
<tr>
<td><code>alpha_</code></td>
<td>The fitted parameter in the model</td>
<td>float</td>
</tr>
<tr>
<td><code>durations</code></td>
<td>The durations provided</td>
<td>array</td>
</tr>
<tr>
<td><code>event_observed</code></td>
<td>The event_observed variable provided</td>
<td>array</td>
</tr>
<tr>
<td><code>timeline</code></td>
<td>The time line to use for plotting and indexing</td>
<td>array</td>
</tr>
</tbody>
</table>
Type  array

**entry**
The entry array provided, or None

**Type**  array or None

**compute_residuals** *(training_dataframe: pandas.core.frame.DataFrame, kind: str) → pandas.core.frame.DataFrame*

Compute the residuals the model.

**Parameters**

- **training_dataframe** *(Dataframe)* – the same training DataFrame given in fit
- **kind** *(string)* – {'schoenfeld', 'score', 'delta_beta', 'deviance', 'martingale', 'scaled_schoenfeld'}

**concordance_index**
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 concordance_index is a measure of the predictive accuracy of the fitted model onto the training dataset.

**fit** *(df, duration_col=None, event_col=None, regressors=None, show_progress=False, timeline=None, weights_col=None, robust=False, initial_point=None, entry_col=None) → ParametricRegressionFitter*

Fit the regression 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** *(bool, optional (default=False))* – since the fitter is iterative, show convergence diagnostics. Useful if convergence is failing.
- **regressors** *(dict, optional)* – a dictionary of parameter names -> list of column names that maps model parameters to a linear combination of variables. If left as None, all variables will be used for all parameters.
- **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** *(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** *(string)* – 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

**Return type**  print_summary, params_, confidence_intervals_ and more
fit_interval_censoring(df, lower_bound_col, upper_bound_col, event_col=None, ancillary_df=None, regressors=None, show_progress=False, timeline=None, weights_col=None, robust=False, initial_point=None, entry_col=None) → ParametricRegressionFitter

Fit the regression 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).

- **lower_bound_col** (string) – the name of the column in DataFrame that contains the lower bounds of the intervals.

- **upper_bound_col** (string) – the name of the column in DataFrame that contains the upper bounds of the intervals.

- **event_col** (string, optional) – the name of the column in DataFrame that contains the subjects’ death observation. If left as None, this is inferred based on the upper and lower interval limits (equal implies observed death.)

- **show_progress** (bool, optional (default=False)) – since the fitter is iterative, show convergence diagnostics. Useful if convergence is failing.

- **regressors** (dict, optional) – a dictionary of parameter names -> list of column names that maps model parameters to a linear combination of variables. If left as None, all variables will be used for all parameters.

- **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** (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** (string) – 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

**Return type** print_summary, params_, confidence_intervals_ and more

fit_left_censoring(df, duration_col=None, event_col=None, regressors=None, fit_intercept=None, show_progress=False, timeline=None, weights_col=None, robust=False, initial_point=None, entry_col=None) → ParametricRegressionFitter

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.

• **fit_intercept** *(bool, optional)* – If true, add a constant column to the regression. Overrides value set in class instantiation.

• **show_progress** *(bool, optional (default=False))* – since the fitter is iterative, show convergence diagnostics. Useful if convergence is failing.

• **regressors** *(dict, optional)* – a dictionary of parameter names -> list of column names that maps model parameters to a linear combination of variables. If left as None, all variables will be used for all parameters.

• **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** *(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** *(str)* – specify a column in the DataFrame that denotes any late-entries (left truncation) that occurred. See the docs on left truncation

**Returns**

**Return type** *self* with additional new properties *print_summary, params_, confidence_intervals_ and more*

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

Alias for **fit**

 **See also:**

**fit**

**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**

The mean survival time of the average subject in the training dataset.

**median_survival_time**

The median survival time of the average subject in the training dataset.

**plot** *(columns=None, parameter=None, ax=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, ax=None, times=None, **kwargs)*

Produces a plot 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.
- **times** – pass in a times to plot
- **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, 3), cmap='coolwarm')
```

```none
fitters/regression/images/plot_covariate_example3.png
```

```python
# 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** *(df, *, times=None, conditional_after=None)*

Predict the cumulative hazard for individuals, given their covariates.

Parameters

- **df** *(DataFrame)* – a (n,d) DataFrame. If a DataFrame, columns can be in any order.
- **times** *(iterable, optional)* – an iterable (array, list, series) of increasing times to predict the cumulative hazard at. Default is the set of all durations in the training dataset (observed and unobserved).
- **conditional_after** *(iterable, optional)* – Must be equal is size to (df.shape[0],) (n above). An iterable (array, list, series) of possibly non-zero values that represent how long the subject has already lived for. Ex: if $T$ is the unknown event time, then this represents $T | T > s$. This is useful for knowing the remaining hazard/survival of censored subjects. The new timeline is the remaining duration of the subject, i.e. normalized back to starting at 0.
Returns the cumulative hazards of individuals over the timeline

Return type  DataFrame

**predict_expectation** *(X, conditional_after=None) → pandas.core.series.Series*

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 trapizoidal rule to approximate the integral.

**Caution:** 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.

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_hazard** *(df, *, times=None)*

Predict the hazard for individuals, given their covariates.

Parameters

- *df* *(DataFrame)* – a \((n,d)\) DataFrame. If a DataFrame, columns can be in any order.

- *times* *(iterable, optional)* – an iterable (array, list, series) of increasing times to predict the cumulative hazard at. Default is the set of all durations in the training dataset (observed and unobserved).

- *conditional_after* – Not implemented yet.

Returns  the hazards of individuals over the timeline

Return type  DataFrame

**predict_median** *(df, *, conditional_after=None) → pandas.core.frame.DataFrame*

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.

- *conditional_after* *(iterable, optional)* – Must be equal is size to \( df\.shape[0] \) (denoted \( n \) above). An iterable (array, list, series) of possibly non-zero values that represent how long the subject has already lived for. Ex: if \( T \) is the unknown event time, then this represents \( T | T > s \). This is useful for knowing the remaining hazard/survival of censored subjects.
The new timeline is the remaining duration of the subject, i.e. normalized back to starting at 0.

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(df, *, p=0.5, conditional_after=None) → pandas.core.series.Series

predict_survival_function(df, times=None, conditional_after=None) → pandas.core.frame.DataFrame

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

- df (DataFrame) – a (n,d) DataFrame. If a DataFrame, columns can be in any order.
- 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.
- conditional_after (iterable, optional) – Must be equal is size to df.shape[0] (denoted n above). An iterable (array, list, series) of possibly non-zero values that represent how long the subject has already lived for. Ex: if \( T \) is the unknown event time, then this represents \( T | T > s \). This is useful for knowing the remaining hazard/survival of censored subjects.

Returns survival_function – the survival probabilities of individuals over the timeline

Return type DataFrame

print_summary (decimals=2, style=None, **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
- style (string) – {html, ascii, latex}
- kwargs – print additional metadata in the output (useful to provide model names, dataset names, etc.) when comparing multiple outputs.

score (df: pandas.core.frame.DataFrame, scoring_method: str = 'log_likelihood') → float

Score the data in df on the fitted model. With default scoring method, returns the _average log-likelihood_.

Parameters

- df (DataFrame) – the dataframe with duration col, event col, etc.
- scoring_method (str) – one of {‘log_likelihood’, ‘concordance_index’} log_likelihood: returns the average unpenalized log-likelihood. concordance_index: returns the concordance-index

Examples
from lifelines import WeibullAFTFitter
from lifelines.datasets import load_rossi

rossi_train = load_rossi().loc[:400]
rossi_test = load_rossi().loc[400:]
wf = WeibullAFTFitter().fit(rossi_train, 'week', 'arrest')
wf.score(rossi_train)
wf.score(rossi_test)

summary
Summary statistics describing the fit.

See also:
print_summary

LogLogisticAFTFitter
class lifelines.fitters.log_logistic_aft_fitter.LogLogisticAFTFitter(alpha=0.05, penalizer=0.0, l1_ratio=0.0, fit_intercept=True, model_ancillary=False)

Bases: lifelines.fitters.ParametericAFTRegressionFitter

This class implements a Log-Logistic AFT model. The model has parameterized form, with \( \alpha(x) = \exp(a_0 + a_1 x_1 + \ldots + a_n x_n) \), and optionally, \( \beta(y) = \exp(b_0 + b_1 y_1 + \ldots + b_m y_m) \).

The cumulative hazard rate is

\[
H(t; x, y) = \log \left( 1 + \left( \frac{t}{\alpha(x)} \right)^{\beta(y)} \right)
\]

The \( \alpha \) (scale) parameter has an interpretation as being equal to the median lifetime. The \( \beta \) parameter influences the shape of the hazard.

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
- **model_ancillary** (optional (default=False)) – set the model instance to always model the ancillary parameter with the supplied Dataframe. This is useful for grid-search optimization.
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
compute_residuals (df)
Compute the residuals the model.

Parameters
• training_dataframe (DataFrame) – the same training DataFrame given in fit

concordance_index_
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 concordance_index_ is a measure of the predictive accuracy of the fitted model onto the training dataset.

fit (df, duration_col=None, event_col=None, ancillary_df=None, fit_intercept=None, show_progress=False, timeline=None, weights_col=None, robust=False, initial_point=None, entry_col=None) → ParametricAFTRegressionFitter
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** *(bool, 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.

- **fit_intercept** *(bool, optional)* – If true, add a constant column to the regression. Overrides value set in class instantiation.

- **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** *(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** *(string)* – specify a column in the DataFrame that denotes any late-entries (left truncation) that occurred. See the docs on left truncation

**Returns**

**Return type** self with additional new properties print_summary, params_, confidence_intervals_ and more

**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, 1, 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, fit_intercept=None, show_progress=False, timeline=None, weights_col=None, robust=False, initial_point=None, entry_col=None) → ParametericAFTRegressionFitter

Fit the accelerated failure time model to a interval-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)

- **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`.

- **fit_intercept** (*bool, optional*) – If true, add a constant column to the regression. Overrides value set in class instantiation.

- **show_progress** (*bool, 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 DataFrame that specifies weights per observation.

- **robust** (*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** (*str*) – specify a column in the DataFrame that denotes any late-entries (left truncation) that occurred. See the docs on left truncation

Returns

- **Return type** self with additional new properties `print_summary`, `params_`, `confidence_intervals_` and more

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
     'event': [1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],  # 0 = censored
     'ancillary': [1, 0, 1, 0, 1, 0, 0, 1, 0, 1, 0, 0])

gf = WeibullAFTFitter()
    .fit_interval_censoring(df, 'start', 'stop', 'event')
```

(continues on next page)
{'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, fit_intercept=None, show_progress=False, timeline=None, weights_col=None, robust=False, initial_point=None, entry_col=None) → ParametericAFTRegressionFitter

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.

- **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.

- **fit_intercept** (bool, optional) – If true, add a constant column to the regression. Overrides value set in class instantiation.

- **show_progress** (bool, 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 DataFrame that specifies weights per observation.

- **robust** (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** (str) – specify a column in the DataFrame that denotes any late-entries (left truncation) that occurred. See the docs on left truncation

Returns **self**
Return type self with additional new properties `print_summary`, `params_`, `confidence_intervals_` and more

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, 1, 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)`

Alias for `fit`

See also:

`fit`

`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_`

The mean survival time of the average subject in the training dataset.

`median_survival_time_`

The median survival time of the average subject in the training dataset.

`plot(columns=None, parameter=None, ax=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, ax=None, times=None, **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.
- **times** *(iterable)* – pass in a times to plot
- **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** *(df, *, ancillary_df=None, times=None, conditional_after=None)*

→ **pandas.core.frame.DataFrame**

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

- **df** *(DataFrame)* – a (n,d) covariate numpy array or DataFrame. If a DataFrame, columns can be in any order. If a numpay 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).
- **conditional_after** *(iterable, optional)* – Must be equal is size to df.shape[0] (denoted n above). An iterable (array, list, series) of possibly non-zero values that represent how long the subject has already lived for. Ex: if $T$ is the unknown event time, then this represents $T \mid T > s$. This is useful for knowing the remaining hazard/survival of censored subjects.

The new timeline is the remaining duration of the subject, i.e. normalized back to starting at 0.

See also:

**predict_percentile**, **predict_expectation**, **predict_survival_function**

**predict_expectation** *(df, ancillary_df=None)* → **pandas.core.series.Series**

Predict the expectation of lifetimes, $E[T | x]$. 

Parameters
• **X** (*DataFrame*) – a (n,d) 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** (*DataFrame, optional*) – a (n,d) 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_hazard(*df*, *ancillary_df=None, times=None, conditional_after=None) → pandas.core.frame.DataFrame

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**

• **df** (*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).

• **conditional_after** (*iterable, optional*) – Not implemented yet

**See also:**

predict_percentile(), predict_expectation(), predict_survival_function()

predict_median(*df*, *ancillary_df=None, conditional_after=None) → pandas.core.frame.DataFrame

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**

• **df** (*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.

• **conditional_after** (*iterable, optional*) – Must be equal is size to df.shape[0] (denoted n above). An iterable (array, list, series) of possibly non-zero values that represent how long the subject has already lived for. Ex: if \(T\) is the unknown event time, then this represents \(T \mid T > s\). This is useful for knowing the remaining hazard/survival of censored subjects. The new timeline is the remaining duration of the subject, i.e. normalized back to starting at 0.

**See also:**

predict_percentile(), predict_expectation()

predict_percentile(*df*, *ancillary_df=None, p=0.5, conditional_after=None) → pandas.core.series.Series

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

**Parameters**
• **X** (*DataFrame*) – a (n,d) 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** (*DataFrame, optional*) – a (n,d) 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** (*df*, *times=None*, *conditional_after=None*, *ancillary_df=None*) → `pandas.core.frame.DataFrame`

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 survival function at. Default is the set of all durations (observed and unobserved).

• **conditional_after** (*iterable, optional*) – Must be equal is size to df.shape[0] (denoted n above). An iterable (array, list, series) of possibly non-zero values that represent how long the subject has already lived for. Ex: if \( T \) is the unknown event time, then this represents \( T \mid T > s \). This is useful for knowing the remaining hazard/survival of censored subjects. The new timeline is the remaining duration of the subject, i.e. normalized back to starting at 0.

**print_summary** (*decimals=2, style=None, **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

• **style** (*string*) – {html, ascii, latex}

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

**score** (*df: pandas.core.frame.DataFrame*, *scoring_method: str = ‘log_likelihood’*) → float

Score the data in df on the fitted model. With default scoring method, returns the _average log-likelihood_.

**Parameters**

• **df** (*DataFrame*) – the dataframe with duration col, event col, etc.

• **scoring_method** (*str*) – one of {'log_likelihood', 'concordance_index'} log_likelihood: returns the average unpenalized log-likelihood. concordance_index: returns the concordance-index
Examples

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

rossi_train = load_rossi().loc[:400]
rossi_test = load_rossi().loc[400:]
wf = WeibullAFTFitter().fit(rossi_train, 'week', 'arrest')
wf.score(rossi_train)
wf.score(rossi_test)
```

summary

Summary statistics describing the fit.

See also:

print_summary

LogNormalAFTFitter

```python
class lifelines.fitters.log_normal_aft_fitter.LogNormalAFTFitter (alpha=0.05, penalizer=0.0, l1_ratio=0.0, fit_intercept=True, model_ancillary=False)
```

Bases: lifelines.fitters.ParametericAFTRegressionFitter

This class implements a Log-Normal AFT model. The model has parameterized form, with \( \mu(x) = a_0 + a_1 x_1 + \ldots + a_n x_n \), and optionally, \( \sigma(y) = \exp(b_0 + b_1 y_1 + \ldots + b_m y_m) \).

The cumulative hazard rate is

\[
H(t; x, y) = -\log \left( 1 - \Phi \left( \frac{\log(T) - \mu(x)}{\sigma(y)} \right) \right)
\]

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** (bool, 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 \( \lambda \). 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

\[
\text{penalizer} \times \text{l1_ratio} \times ||w||_1 + 0.5 \times \text{penalizer} \times (1 - \text{l1_ratio}) \times ||w||^2_2
\]

- **model_ancillary** (optional (default=False)) – set the model instance to always model the ancillary parameter with the supplied DataFrame. This is useful for grid-search optimization.
**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

**compute_residuals(df)**
Compute the residuals the model.

Parameters

- training_dataframe (DataFrame) – the same training DataFrame given in fit

**concordance_index_**
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 concordance_index_ is a measure of the predictive accuracy of the fitted model onto the training dataset.

**fit(df, duration_col=None, event_col=None, ancillary_df=None, fit_intercept=None, show_progress=False, timeline=None, weights_col=None, robust=False, initial_point=None, entry_col=None)** → ParametericAFTRegressionFitter
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** *(bool, 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.

• **fit_intercept** *(bool, optional)* – If true, add a constant column to the regression. Overrides value set in class instantiation.

• **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** *(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** *(string)* – specify a column in the DataFrame that denotes any late-entries (left truncation) that occurred. See the docs on left truncation

**Returns**

**Return type** self with additional new properties print_summary, params_, confidence_intervals_ and more

**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, 0, 1, 0],
    'var': [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
    '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, fit_intercept=None, show_progress=False, timeline=None, weights_col=None, robust=False, initial_point=None, entry_col=None) → ParametericAFTRegressionFitter

Fit the accelerated failure time model to an interval-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)
- 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.
- fit_intercept (bool, optional) – If true, add a constant column to the regression. Overrides value set in class instantiation.
- show_progress (bool, 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 DataFrame that specifies weights per observation.
- robust (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 (str) – specify a column in the DataFrame that denotes any late-entries (left truncation) that occurred. See the docs on left truncation

Returns

Return type self with additional new properties print_summary, params_, confidence_intervals_ and more

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],
'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, fit_intercept=None, show_progress=False, timeline=None, weights_col=None, robust=False, initial_point=None, entry_col=None) →
ParametericAFTRegressionFitter
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.

- **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.

- **fit_intercept** (bool, optional) – If true, add a constant column to the regression. Overrides value set in class instantiation.

- **show_progress** (bool, 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 DataFrame that specifies weights per observation.

- **robust** (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** (str) – specify a column in the DataFrame that denotes any late-entries (left truncation) that occurred. See the docs on left truncation

Returns self
Return type self with additional new properties print_summary, params_,
  confidence_intervals_ and more

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, 1, 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)

Alias for fit

See also:

fit

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_  
The mean survival time of the average subject in the training dataset.

median_survival_time_  
The median survival time of the average subject in the training dataset.

plot columns=None, parameter=None, ax=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, ax=None, times=None, **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.
- **times** *(iterable)* – pass in a times to plot
- **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**(df: `pandas.core.frame.DataFrame`, ancillary_df: `Optional[pandas.core.frame.DataFrame]` = `None`, times: `None`, conditional_after: `None`) ➔ `pandas.core.frame.DataFrame`

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

- **df** *(DataFrame)* – a (n,d) covariate numpy array or DataFrame. If a DataFrame, columns can be in any order. If a numpay 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).
- **conditional_after** *(iterable, optional)* – Must be equal is size to df.shape[0] (denoted n above). An iterable (array, list, series) of possibly non-zero values that represent how long the subject has already lived for. Ex: if $T$ is the unknown event time, then this represents $T|T > s$. This is useful for knowing the remaining hazard/survival of censored subjects. The new timeline is the remaining duration of the subject, i.e. normalized back to starting at 0.

See also:

 predict_percentile(), predict_expectation(), predict_survival_function()


Predict the expectation of lifetimes, $E[T|\mathbf{x}]$. 

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.
- \( \text{ancillary}_X \) (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 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()
ual does not cross $p$, 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.

- **conditional_after** *(iterable, optional)* – Must be equal is size to df.shape[0] (denoted $n$ above). An iterable (array, list, series) of possibly non-zero values that represent how long the subject has already lived for. Ex: if $T$ is the unknown event time, then this represents $T|T > s$. This is useful for knowing the remaining hazard/survival of censored subjects. The new timeline is the remaining duration of the subject, i.e. normalized back to starting at 0.

Returns **percentiles**

Return type **DataFrame**

See also:

predict_median()

**predict_survival_function** *(df, times=None, conditional_after=None, ancillary_df=None) → pandas.core.frame.DataFrame*

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 survival function at. Default is the set of all durations (observed and unobserved).

- **conditional_after** *(iterable, optional)* – Must be equal is size to df.shape[0] (denoted $n$ above). An iterable (array, list, series) of possibly non-zero values that represent how long the subject has already lived for. Ex: if $T$ is the unknown event time, then this represents $T|T > s$. This is useful for knowing the remaining hazard/survival of censored subjects. The new timeline is the remaining duration of the subject, i.e. normalized back to starting at 0.

**print_summary** *(decimals=2, style=None, **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

- **style** *(string)* – {html, ascii, latex}
• **kwargs** – print additional metadata in the output (useful to provide model names, dataset names, etc.) when comparing multiple outputs.

**score**(df: pandas.core.frame.DataFrame, scoring_method: str = 'log_likelihood') → float
Score the data in df on the fitted model. With default scoring method, returns the _average log-likelihood_.

**Parameters**
- **df** (DataFrame) – the dataframe with duration col, event col, etc.
- **scoring_method** (str) – one of {'log_likelihood', 'concordance_index'} log_likelihood: returns the average unpenalized log-likelihood. concordance_index: returns the concordance-index

**Examples**

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

rossi_train = load_rossi().loc[:400]
rossi_test = load_rossi().loc[400:]
wf = WeibullAFTFitter().fit(rossi_train, 'week', 'arrest')
wf.score(rossi_train)
wf.score(rossi_test)
```

**summary**
Summary statistics describing the fit.

See also:
print_summary

**PiecewiseExponentialRegressionFitter**

**class** lifelines.fitters.piecewise_exponential_regression_fitter.PiecewiseExponentialRegressionFitter

**Bases:** lifelines.fitters.ParametricRegressionFitter

This implements a piecewise constant-hazard model at pre-specified break points.

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

where \(\lambda_i(x) = \exp \beta_i x\).

**Parameters**
- **breakpoints** (list) – a list of times when a new exponential model is constructed.
- **penalizer** (float) – penalize the variance of the \(\lambda_i\). See blog post below.
- **alpha** (float, optional (default=0.05)) – the level in the confidence intervals.
Examples

See blog post here and paper replication here.

```python
compute_residuals(training_dataframe: pandas.core.frame.DataFrame, kind: str) \rightarrow pandas.core.frame.DataFrame
```
Compute the residuals the model.

Parameters

- **training_dataframe** (*DataFrame*) – the same training DataFrame given in `fit`
- **kind** (*string*) – `{'schoenfeld', 'score', 'delta_beta', 'deviance', 'martingale', 'scaled_schoenfeld'}`

```python
concordance_index_
```
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 `concordance_index_` is a measure of the predictive accuracy of the fitted model onto the training dataset.

```python
fit(df, duration_col, event_col=None, regressors=None, show_progress=False, timeline=None, weights_col=None, robust=False, initial_point=None, entry_col=None)
```
Fit the regression 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** (*bool, optional (default=False)*) – since the fitter is iterative, show convergence diagnostics. Useful if convergence is failing.
- **regressors** (*dict, optional*) – a dictionary of parameter names -> list of column names that maps model parameters to a linear combination of variables. If left as None, all variables will be used for all parameters.
- **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** (*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** (*string*) – 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

Return type `print_summary, params_; confidence_intervals_and more`
fit_interval_censoring\( (df, \text{lower\_bound\_col}, \text{upper\_bound\_col}, \text{event\_col}=\text{None}, \text{ancillary\_df}=\text{None}, \text{regressors}=\text{None}, \text{show\_progress}=\text{False}, \text{timeline}=\text{None}, \text{weights\_col}=\text{None}, \text{robust}=\text{False}, \text{initial\_point}=\text{None}, \text{entry\_col}=\text{None}) \rightarrow \text{ParametricRegressionFitter} \)

Fit the regression 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).
- **lower_bound_col** (*string*) – the name of the column in DataFrame that contains the lower bounds of the intervals.
- **upper_bound_col** (*string*) – the name of the column in DataFrame that contains the upper bounds of the intervals.
- **event_col** (*string, optional*) – the name of the column in DataFrame that contains the subjects’ death observation. If left as None, this is inferred based on the upper and lower interval limits (equal implies observed death.)
- **show_progress** (*bool, optional (default=False]*) – since the fitter is iterative, show convergence diagnostics. Useful if convergence is failing.
- **regressors** (*dict, optional*) – a dictionary of parameter names -> list of column names that maps model parameters to a linear combination of variables. If left as None, all variables will be used for all parameters.
- **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** (*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** (*string*) – 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

Return type **print_summary**, **params**, **confidence_intervals** and more

fit_left_censoring\( (df, \text{duration\_col}=\text{None}, \text{event\_col}=\text{None}, \text{regressors}=\text{None}, \text{fit\_intercept}=\text{None}, \text{show\_progress}=\text{False}, \text{timeline}=\text{None}, \text{weights\_col}=\text{None}, \text{robust}=\text{False}, \text{initial\_point}=\text{None}, \text{entry\_col}=\text{None}) \rightarrow \text{ParametricRegressionFitter} \)

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.

• **fit_intercept** *(bool, optional)* – If true, add a constant column to the regression. Overrides value set in class instantiation.

• **show_progress** *(bool, optional (default=False))* – since the fitter is iterative, show convergence diagnostics. Useful if convergence is failing.

• **regressors** *(dict, optional)* – a dictionary of parameter names -> list of column names that maps model parameters to a linear combination of variables. If left as None, all variables will be used for all parameters.

• **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** *(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** *(str)* – specify a column in the DataFrame that denotes any late-entries (left truncation) that occurred. See the docs on left truncation.

Returns

Return type self with additional new properties print_summary, params_,
confidence_intervals_ and more

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

Alias for **fit**

See also:

**fit**

**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_**

The mean survival time of the average subject in the training dataset.

**median_survival_time_**

The median survival time of the average subject in the training dataset.

**plot** *(columns=None, parameter=None, ax=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, ax=None, times=None, **kwargs)*

Produces a plot 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.
- **times** – pass in a times to plot
- **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
datasets.load_rossi()
wf = WeibullAFTFitter().fit(rossi, 'week', 'arrest')
wf.plot_covariate_groups('prio', values=np.arange(0, 15, 3), cmap='coolwarm')
```

```python
# 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** *(df, times=None, conditional_after=None) → pandas.core.frame.DataFrame*

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.

Returns **cumulative_hazard** – the cumulative hazard of individuals over the timeline

Return type **DataFrame**
predict_expectation \( (X, \text{conditional}_{\text{after}}=\text{None}) \) → pandas.core.series.Series
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 trapizoidal rule to approximate the integral.

Caution: 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.

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_hazard \( (df, *, \text{times} = \text{None}) \)
Predict the hazard for individuals, given their covariates.

Parameters
- **df** (DataFrame) – a (n,d) DataFrame. If a DataFrame, columns can be in any order.
- **times** (iterable, optional) – an iterable (array, list, series) of increasing times to predict the cumulative hazard at. Default is the set of all durations in the training dataset (observed and unobserved).
- **conditional_after** – Not implemented yet.

Returns the hazards of individuals over the timeline

Return type DataFrame

predict_median \( (df, *, \text{conditional}_{\text{after}}=\text{None}) \) → pandas.core.frame.DataFrame
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.
- **conditional_after** (iterable, optional) – Must be equal is size to df.shape[0] (denoted \( n \) above). An iterable (array, list, series) of possibly non-zero values that represent how long the subject has already lived for. Ex: if \( T \) is the unknown event time, then this represents \( T|T > s \). This is useful for knowing the remaining hazard/survival of censored subjects. The new timeline is the remaining duration of the subject, i.e. normalized back to starting at 0.
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()

\[\text{predict_percentile}(df, *, p=0.5, \text{conditional_after=None}) \rightarrow \text{pandas.core.series.Series}\]

\[\text{predict_survival_function}(df, \text{times=None, conditional_after=None}) \rightarrow \text{pandas.core.frame.DataFrame}\]

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

• df (DataFrame) – a (n,d) DataFrame. If a DataFrame, columns can be in any order.

• 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.

• conditional_after (iterable, optional) – Must be equal is size to df.shape[0] (denoted \(n\) above). An iterable (array, list, series) of possibly non-zero values that represent how long the subject has already lived for. Ex: if \(T\) is the unknown event time, then this represents \(T|T > s\). This is useful for knowing the remaining hazard/survival of censored subjects.

Returns survival_function – the survival probabilities of individuals over the timeline

Return type DataFrame

\[\text{print_summary}(\text{decimals}=2, \text{style=None, **kwargs})\]

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

• style (string) – {html, ascii, latex}

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

\[\text{score}(df: \text{pandas.core.frame.DataFrame}, \text{scoring_method: str = 'log_likelihood'}) \rightarrow \text{float}\]

Score the data in df on the fitted model. With default scoring method, returns the _average log-likelihood_.

Parameters

• df (DataFrame) – the dataframe with duration col, event col, etc.

• scoring_method (str) – one of {‘log_likelihood’, ‘concordance_index’} log_likelihood: returns the average unpenalized log-likelihood. concordance_index: returns the concordance-index

Examples

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

rossi_train = load_rossi().loc[:400]
rossi_test = load_rossi().loc[400:]
```
```python
wf = WeibullAFTFitter().fit(rossi_train, 'week', 'arrest')
wf.score(rossi_train)
wf.score(rossi_test)

summary
Summary statistics describing the fit.
See also:
print_summary

WeibullAFTFitter

class lifelines.fitters.weibull_aft_fitter.WeibullAFTFitter(alpha: float = 0.05, penalizer: float = 0.0, l1_ratio: float = 0.0, fit_intercept: bool = True, model_ancillary: bool = False)


This class implements a Weibull AFT model. The model has parameterized form, with $\lambda(x) = \exp(\beta_0 + \beta_1 x_1 + \ldots + \beta_n x_n)$, and optionally, $\rho(y) = \exp(\alpha_0 + \alpha_1 y_1 + \ldots + \alpha_m y_m)$,

$$S(t; x, y) = \exp\left(-\left(\frac{t}{\lambda(x)}\right)^{\rho(y)}\right),$$

With no covariates, the Weibull model’s parameters has the following interpretations: The $\lambda$ (scale) parameter has an applicable interpretation: it represent the time when 37% of the population has died. The $\rho$ (shape) parameter controls if the cumulative hazard (see below) is convex or concave, representing accelerating or decelerating hazards.

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 $\text{penalizer} \times \text{l1_ratio} \times ||w||_1 + 0.5 \times \text{penalizer} \times (1 - \text{l1_ratio}) \times ||w||^2_2$
• **model_ancillary** (*optional (default=False)*) – set the model instance to always model the ancillary parameter with the supplied Dataframe. This is useful for grid-search optimization.

**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

**check_assumptions** (*training_df: pandas.core.frame.DataFrame, advice: bool = True, show_plots: bool = False, p_value_threshold: float = 0.01, plot_n_bootstraps: int = 10, columns: Optional[List[str]] = None) → None*

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** (*bool, optional*) – display advice as output to the user’s screen

• **show_plots** (*bool, 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.

• **columns** (*list, optional*) – specify a subset of columns to test.
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 deviations 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


```python
def compute_followup_hazard_ratios(training_df: pandas.core.frame.DataFrame, followup_times: Iterable[T_co]) -> pandas.core.frame.DataFrame
```

Recompute the hazard ratio at different follow-up times (lifelines handles accounting for updated censoring and updated durations). This is useful because we need to remember that the hazard ratio is actually a weighted-average of period-specific hazard ratios.

Parameters

- `training_df` (`pd.DataFrame`) – The same dataframe used to train the model
- `followup_times` (`Iterable`) – a list/array of follow-up times to recompute the hazard ratio at.

```python
def compute_residuals(df)
```

Compute the residuals the model.

Parameters

- `training_dataframe` (`DataFrame`) – the same training DataFrame given in `fit`

```python
def concordance_index_
```

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 `concordance_index_` is a measure of the predictive accuracy of the fitted model onto the training dataset.
**fit** *(df, duration_col=None, ancillary_df=None, fit_intercept=None, show_progress=False, timeline=None, weights_col=None, robust=False, initial_point=None, entry_col=None) → ParametericAFTRegressionFitter*

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** *(bool, 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`.

- **fit_intercept** *(bool, optional)* – If true, add a constant column to the regression. Overrides value set in class instantiation.

- **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** *(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** *(string)* – specify a column in the DataFrame that denotes any late-entries (left truncation) that occurred. See the docs on left truncation

**Returns**

**Return type** self with additional new properties `print_summary`, `params_`, `confidence_intervals_` and more

**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, 2, 2, 2],
    'age': [4, 3, 9, 8, 7, 4, 4, 3, 2, 5, 6, 7],
})

```

(continues on next page)
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, fit_intercept=None, show_progress=False, timeline=None, weights_col=None, robust=False, initial_point=None, entry_col=None) → ParametericAFTRegressionFitter

Fit the accelerated failure time model to an interval-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).

- **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.

- **fit_intercept** (*bool, optional*) – If true, add a constant column to the regression. Overrides value set in class instantiation.

- **show_progress** (*bool, 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 DataFrame that specifies weights per observation.

- **robust** (*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** (*str*) – specify a column in the DataFrame that denotes any late-entries (left truncation) that occurred. See the docs on left truncation

Returns

**Return type** self with additional new properties `print_summary`, `params_`, `confidence_intervals_` and more
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, 1, 0],
    'var': [0, 0, 0, 0, 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_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, fit_intercept=None, show_progress=False, timeline=None, weights_col=None, robust=False, initial_point=None, entry_col=None)` → `ParametericAFTRegressionFitter`

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.

- **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.

- **fit_intercept** *(bool, optional)* – If true, add a constant column to the regression. Overrides value set in class instantiation.

- **show_progress** *(bool, 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 DataFrame that specifies weights per observation.

- **robust** *(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** *(str)* – specify a column in the DataFrame that denotes any late-entries (left truncation) that occurred. See the docs on left truncation

**Returns** self

**Return type**

self with additional new properties print_summary, params_, confidence_intervals_ and more

### Examples

```python
from lifelines import WeibullAFTFitter, LogNormalAFTFitter
from lifelines import 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, 0, 1, 1],
     'var': [0, 0, 0, 0, 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)*

Alias for *fit*

**See also:**

fit

**hazard_ratios_**

**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_**

The mean survival time of the average subject in the training dataset.

**median_survival_time_**

The median survival time of the average subject in the training dataset.

**plot** *(columns=None, parameter=None, ax=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
**plot_covariate_groups**

Returns `ax` – the matplotlib axis that be edited.

Return type matplotlib axis

```python
plot_covariate_groups(covariates, values, plot_baseline=True, ax=None, times=None, **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.
- **times** (iterable) – pass in a times to plot
- **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, 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**

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**

- **df** (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).
- **conditional_after** (iterable, optional) – Must be equal is size to df.shape[0] (denoted n above). An iterable (array, list, series) of possibly non-zero values that represent how long the subject has already lived for. Ex: if $T$ is the unknown event time, then this represents $T|T > s$. This is useful for knowing the remaining hazard/survival of censored subjects.
The new timeline is the remaining duration of the subject, i.e. normalized back to starting at 0.

**See also:**

predict_percentile(), predict_expectation(), predict_survival_function()

**predict_expectation**

```python
def predict_expectation(df: pandas.core.frame.DataFrame, ancillary_df: Optional[pandas.core.frame.DataFrame] = None) → pandas.core.series.Series
```

Predict the expectation of lifetimes, \( E[T|x] \).

**Parameters**

- **df** (*DataFrame*) – a (n,d) 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_df** (*DataFrame, optional*) – a (n,d) 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**

```python
def predict_median(df, *, ancillary_df=None, conditional_after=None) → pandas.core.frame.DataFrame
```

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**

- **df** (*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.
- **conditional_after** (*iterable, optional*) – Must be equal is size to df.shape[0] (denoted \( n \) above). An iterable (array, list, series) of possibly non-zero values that represent how long the subject has already lived for. Ex: if \( T \) is the unknown event time, then this represents \( T|T > s \). This is useful for knowing the remaining hazard/survival of censored subjects.

**See also:**

predict_percentile(), predict_expectation(), predict_survival_function()
The new timeline is the remaining duration of the subject, i.e. normalized back to starting at 0.

See also:

predict_percentile(), predict_expectation()


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

Parameters

- **df** *(DataFrame)* – a (n,d) 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_df** *(DataFrame, optional)* – a (n,d) 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** *(df, times=None, conditional_after=None, ancillary_df=None) → pandas.core.frame.DataFrame*

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 survival function at. Default is the set of all durations (observed and unobserved).

- **conditional_after** *(iterable, optional)* – Must be equal is size to df.shape[0] (denoted \( n \) above). An iterable (array, list, series) of possibly non-zero values that represent how long the subject has already lived for. Ex: if \( T \) is the unknown event time, then this represents \( T | T > s \). This is useful for knowing the remaining hazard/survival of censored subjects.

The new timeline is the remaining duration of the subject, i.e. normalized back to starting at 0.

**print_summary** *(decimals=2, style=None, **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
**score** *(df: pandas.core.frame.DataFrame, scoring_method: str = 'log_likelihood') → float*

Score the data in df on the fitted model. With default scoring method, returns the *average log-likelihood*.

**Parameters**

- **df** *(DataFrame)* – the dataframe with duration col, event col, etc.
- **scoring_method** *(str)* – one of {'log_likelihood', 'concordance_index'}
  - log_likelihood: returns the average unpenalized log-likelihood.
  - concordance_index: returns the concordance-index

**Examples**

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

rossi_train = load_rossi().loc[:400]
rossi_test = load_rossi().loc[400:]
wf = WeibullAFTFitter().fit(rossi_train, 'week', 'arrest')

wf.score(rossi_train)
wf.score(rossi_test)
```

**summary**

Summary statistics describing the fit.

**See also:**

print_summary

### 1.14.2 utils

**lifelines.utils.qth_survival_time** *(q: float, model_or_survival_function) → float*

Returns the time when a single survival function reaches the qth percentile, that is, solves \( q = S(t) \) for \( t \).

**Parameters**

- **q** *(float)* – value between 0 and 1.
- **model_or_survival_function** *(Series, single-column DataFrame, or lifelines model)*

**See also:**

qth_survival_times(), median_survival_times()

**lifelines.utils.restricted_mean_survival_time** *(model_or_survival_function, t: float = inf, return_variance=False) → Union[float, Tuple[float, float]]*

Compute the restricted mean survival time, RMST, of a survival function. This is defined as

\[
RMST(t) = \int_0^t S(\tau)d\tau
\]

For reason why we use an upper bound and not always \( \infty \) is because the tail of a survival function has high variance and strongly affects the RMST.
Parameters

- **model_or_survival_function** (*lifelines model or DataFrame*) – This can be a univariate model, or a pandas DataFrame. The former will provide a more accurate estimate however.
- **t** (*float*) – The upper limit of the integration in the RMST.

Example

```python
from lifelines import KaplanMeierFitter, WeibullFitter
from lifelines.utils import restricted_mean_survival_time

kmf = KaplanMeierFitter().fit(T, E)
restricted_mean_survival_time(kmf, t=3.5)
restricted_mean_survival_time(kmf.survival_function_, t=3.5)

wf = WeibullFitter().fit(T, E)
restricted_mean_survival_time(wf)
restricted_mean_survival_time(wf.survival_function_)
```

References

https://bmcmedresmethodol.biomedcentral.com/articles/10.1186/1471-2288-13-152#Sec27

lifelines.utils.median_survival_times(*model_or_survival_function*) → float

Compute the median survival time of survival function(s).

Parameters

- **model_or_survival_function** (*lifelines model or DataFrame*) – This can be a univariate lifelines model, or a DataFrame of one or more survival functions.

lifelines.utils.qth_survival_times(*q*, *survival_functions*) → Union[pandas.core.frame.DataFrame, float]

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.

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.survival_table_from_events(*death_times, event_observed, birth_times=None, columns=['removed', 'observed', 'censored', 'entrance', 'at_risk'], weights=None, collapse=False, intervals=None) → pandas.core.frame.DataFrame

Create a survival table from right-censored dataset.

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** *(bool, 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**

<table>
<thead>
<tr>
<th>#Uncollapsed output</th>
<th>removed</th>
<th>observed</th>
<th>censored</th>
<th>entrance</th>
<th>at_risk</th>
</tr>
</thead>
<tbody>
<tr>
<td>event_at</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>11</td>
<td>11</td>
</tr>
<tr>
<td>6</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>11</td>
</tr>
<tr>
<td>7</td>
<td>2</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>10</td>
</tr>
<tr>
<td>9</td>
<td>3</td>
<td>3</td>
<td>0</td>
<td>0</td>
<td>8</td>
</tr>
<tr>
<td>13</td>
<td>3</td>
<td>3</td>
<td>0</td>
<td>0</td>
<td>5</td>
</tr>
<tr>
<td>15</td>
<td>2</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>#Collapsed output</th>
<th>removed</th>
<th>observed</th>
<th>censored</th>
<th>at_risk</th>
</tr>
</thead>
<tbody>
<tr>
<td>event_at</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(0, 2]</td>
<td>34</td>
<td>33</td>
<td>1</td>
<td>312</td>
</tr>
<tr>
<td>(2, 4]</td>
<td>84</td>
<td>42</td>
<td>42</td>
<td>278</td>
</tr>
<tr>
<td>(4, 6]</td>
<td>64</td>
<td>17</td>
<td>47</td>
<td>194</td>
</tr>
<tr>
<td>(6, 8]</td>
<td>63</td>
<td>16</td>
<td>47</td>
<td>130</td>
</tr>
<tr>
<td>(8, 10]</td>
<td>35</td>
<td>12</td>
<td>23</td>
<td>67</td>
</tr>
<tr>
<td>(10, 12]</td>
<td>24</td>
<td>5</td>
<td>19</td>
<td>32</td>
</tr>
</tbody>
</table>

**See also:**

`lifelines.utils.group_survival_table_from_events()`

Joins multiple event series together into DataFrames. A generalization of `survival_table_from_events` to data with groups.

**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 [birth times + duration observed]. Normally set to all zeros, but can be positive or negative.

**limit**

**Returns**

• **unique_groups** *(np.array)* – array of all the unique groups present
• **removed** *(DataFrame)* – DataFrame of removal count data at event_times for each group, column names are ‘removed:<group name>’
• **observed** *(DataFrame)* – DataFrame of observed count data at event_times for each group, column names are ‘observed:<group name>’
• **censored** *( DataFrame)* – DataFrame of censored count data at event_times for each group, column names are ‘censored:<group name>’

**Example**

```python
#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:

```
<table>
<thead>
<tr>
<th>index</th>
<th>observed</th>
<th>censored</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>
```

# 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])

See also:

- survival_table_from_events

lifelines.utils.datetimes_to_durations(start_times, 
end_times, 
fill_date=datetime.datetime(2020, 3, 31, 23, 26, 2, 907701), 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 (bool, 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) → float

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

• 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
References


Examples

```python
from lifelines.utils import concordance_index
cph = CoxPHFitter().fit(df, 'T', 'E')
concordance_index(df['T'], -cph.predict_partial_hazard(df), df['E'])
```

```python
lifelines.utils.k_fold_cross_validation(fitters, df, duration_col, event_col=None, k=5, scoring_method='log_likelihood', 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** *(array)* – the column in DataFrame that contains the subjects lifetimes.

- **event_col** *(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). Default: `statistics.concordance_index`: (C-index)

- **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_kwarg)`

- **fitter_kwarg** *(keyword args)* – keyword args to pass into fitter.fit method.

Returns **results** *(k,l) list of scores for each fold. The scores can be anything.*

Return type **list**

See also:

```python
lifelines.utils.sklearn_adapter.sklearn_adapter()
```

```python
lifelines.utils.to_long_format(df, duration_col) → pandas.core.frame.DataFrame
```

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 each 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()`

lifelines.utils.to_episodic_format(df, duration_col, event_col, id_col=None, time_gaps=1) → pandas.core.frame.DataFrame

This function takes a “flat” dataset (that is, non-time-varying), and converts it into a time-varying dataset with static variables.

Useful if your dataset has variables that do not satisfy the proportional hazard assumption, and you need to create a time-varying dataset to include interaction terms with time.

Parameters

• **df** (*DataFrame*) – a DataFrame of the static dataset.

• **duration_col** (*string*) – string representing the column in df that represents the durations of each subject.

• **event_col** (*string*) – string representing the column in df that represents whether the subject experienced the event or not.

• **id_col** (*string, optional*) – Specify the column that represents an id, else lifelines creates an auto-incrementing one.

• **time_gaps** (*float or int*) – Specify a desired time_gap. For example, if time_gap is 2 and a subject lives for 10.5 units of time, then the final long form will have 5 + 1 rows for that subject: (0, 2], (2, 4], (4, 6], (6, 8], (8, 10], (10, 10.5] Smaller time_gaps will produce larger DataFrames, and larger time_gaps will produce smaller DataFrames. In the limit, the long DataFrame will be identical to the original DataFrame.

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.)

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(long_form_df, cv, id_col, duration_col, event_col, start_col='start', stop_col='stop', add_enum=False, overwrite=True, cumulative_sum=False, cumulative_sum_prefix='cumsum_', delay=0) → pandas.core.frame.DataFrame

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/Time%20varying%20survival%20regression.html#dataset-creation-for-time-varying-regression](https://lifelines.readthedocs.io/en/latest/Time%20varying%20survival%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 id_col, ‘start’, ‘stop’, and event_col. See function 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 id_col and duration_col. duration_col represents time since the start of the subject’s life.

- **id_col** (string) – the column in long_form_df and cv representing a unique identifier for subjects.

- **duration_col** (string) – the column in cv that represents the time-since-birth the observation occurred at.

- **event_col** (string) – the column in df that represents if the event-of-interest occurred

- **add_enum** (bool, 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** (bool, optional) – if True, covariate values in long_form_df will be overwritten by covariate values in cv if the column exists in both cv and long_form_df and the timestamps are identical. If False, the default behavior will be to sum the values together.

- **cumulative_sum** (bool, 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 **long_form_df** – A DataFrame with updated rows to reflect the novel times slices (if any) being added from cv, and novel (or updated) columns of new covariates from cv

Return type DataFrame

See also:

to_episodic_format(), to_long_format(), covariates_from_event_matrix()

lifelines.utils.covariates_from_event_matrix(df, id_col) → pandas.core.frame.DataFrame

This is a helper function to handle binary event datastreams in a specific format and convert it to a format that add_covariate_to_timeline will accept. For example, suppose you have a dataset that looks like:
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)
```

To quickly determine the best\(^1\) univariate model, this function will iterate through each parametric model avail-
able in lifelines and select the one that minimizes a particular measure of fit.

\(^1\)Best, according to the measure of fit.

**Parameters**

- **event_times (list, np.array, pd.Series)** – a (n,) array of observed survival times.
- **event_observed (list, np.array, pd.Series)** – a (n,) array of censored flags, 1 if observed, 0
  if not. Default None assumes all observed.
- **scoring_method (string)** – one of {"AIC", "BIC"}
- **additional_models (list)** – list of other parametric models that implement the lifelines API.

**Returns**

- **Return type**  tuple of fitted best_model and best_score

### 1.14.3 statistics

**class lifelines.statistics.StatisticalResult**

```python
class lifelines.statistics.StatisticalResult(p_value, test_statistic, name=None,
test_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.
- **test_name** *(string)* – the test that was used. Lifelines should set this.
- **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()`.

```python
ascii_print(decimals=2, **kwargs)
html_print(decimals=2, **kwargs)
html_print_inside_jupyter(decimals=2, **kwargs)
lifelines.statistics.logrank_test(durations_A, durations_B, event_observed_A=None, event_observed_B=None, t_0=-1, **kwargs) → lifelines.statistics.StatisticalResultMeasures
latex_print(decimals=2, **kwargs)
print_specific_style(style, decimals=2, **kwargs)
print_summary(decimals=2, style=None, **kwargs)
```

**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.
- This implementation is a special case of the function `multivariate_logrank_test`, which is used internally. See Survival and Event Analysis, page 108.
• There are only disadvantages to using the log-rank test versus using the Cox regression. See more here for a discussion. To convert to using the Cox regression:

```python
from lifelines import CoxPHFitter
dfA = pd.DataFrame({'E': event_observed_A, 'T': durations_A, 'groupA': 1})
dfB = pd.DataFrame({'E': event_observed_B, 'T': durations_B, 'groupA': 0})
df = pd.concat([dfA, dfB])
cph = CoxPHFitter().fit(df, 'T', 'E')
cph.print_summary()
```

Parameters

- **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.
- **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()
p(results.p_value)  # 0.7676
p(results.test_statistic)  # 0.0872
```

See also:

- multivariate_logrank_test()
- pairwise_logrank_test()
- survival_difference_at_fixed_point_in_time_test()

lifelines.statistics.multivariate_logrank_test(event_durations, event_observed=None, t_0=-1, **kwargs) → lifelines.statistics.StatisticalResult
This test is a generalization of the logrank_test: it can deal with $n > 2$ populations (and should be equal when $n=2$):

$$H_0 : h_1(t) = h_2(t) = h_3(t) = \ldots = h_n(t)$$

$$H_A : \text{there exist at least one group that differs from the other}$$

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, 1, 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, 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
```

See also:

- `pairwise_logrank_test()`, `logrank_test()`

```
lifelines.statistics.pairwise_logrank_test(event_durations, groups, event_observed=None, t_0=-1, **kwargs) \rightarrow StatisticalResult
```

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)

→ lifelines.statistics.StatisticalResult

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 : 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(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


lifelines.statistics.proportional_hazard_test(fitted_cox_model, training_df, time_transform='rank', precomputed_residuals=None, **kwargs)

→ lifelines.statistics.StatisticalResult

Test whether any variable in a Cox model breaks the proportional hazard assumption.

Parameters

- **fitted_cox_model** (*CoxPHFitter*) – the fitted Cox model, fitted with *training_df*, you wish to test. Currently only the CoxPHFitter is supported, but later CoxTimeVaryingFitter, too.
- **training_df** (*DataFrame*) – the DataFrame used in the call to the Cox model’s *fit*.
- **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.
- **precomputed_residuals** (*DataFrame, optional*) – specify the scaled Schoenfeld residuals, if already computed.
- **kwargs** – additional parameters to add to the StatisticalResult

Notes

R uses the default *km*, we use *rank*, as this performs well versus other transforms. See http://eprints.lse.ac.uk/84988/1/06_ParkHendry2015-ReassessingSchoenfeldTests_Final.pdf

lifelines.statistics.power_under_cph(n_exp, n_con, p_exp, p_con, postulated_hazard_ratio, alpha=0.05) → float

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

- **n_exp** (*integer*) – size of the experiment group.
- **n_con** (*integer*) – size of the control group.
- **p_exp** (*float*) – probability of failure in experimental group over period of study.
- **p_con** (*float*) – probability of failure in control group over period of study
- **postulated_hazard_ratio** (*float*)
- the postulated hazard ratio
- **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:

```python
calculate_sample_size_necessary_under_cph()
```

lifelines.statistics.calculate_sample_size_necessary_under_cph(power, ratio_of_participants, p_exp, p_con, postulated_hazard_ratio, alpha=0.05)

This computes the sample size for needed power to compare two groups under a Cox Proportional Hazard model.

**Parameters**

- **power** (*float*) – power to detect the magnitude of the hazard ratio as small as that specified by postulated_hazard_ratio.
- **ratio_of_participants** (*ratio of participants in experimental group over control group.*)
- **p_exp** (*float*) – probability of failure in experimental group over period of study.
- **p_con** (*float*) – probability of failure in control group over period of study
- **postulated_hazard_ratio** (*float*) – the postulated hazard ratio
- **alpha** (*float, optional (default=0.05)*) – type I error rate

**Returns**

- **n_exp** (*integer*) – the samples sizes need for the experiment to achieve desired power
- **n_con** (*integer*) – the samples sizes need for the control group to achieve desired power

**Examples**

```python
from lifelines.statistics import calculate_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 = calculate_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()

1.14.4 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:

The axes which was used.

Return type:

ax

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', ax=None, **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 \n  – offsetting the births away from \(t=0\). This could be from left-truncation, or delayed entry into study.

• left_truncated \((\text{boolean})\) \n  – if entry is provided, and the data is left-truncated, this will display additional information in the plot to reflect this.

• sort_by_duration \((\text{boolean})\) \n  – sort by the duration vector

• event_observed_color \((\text{str})\) \n  – default: “\#A60628”

• event_censored_color \((\text{str})\) \n  – 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.\texttt{qq\_plot} \((\text{model, ax=None, **plot\_kwargs})\)

Produces a quantile-quantile plot of the empirical CDF against the fitted parametric CDF. Large deviations away from the line \(y=x\) can invalidate a model (though we expect some natural deviance in the tails).

Parameters

• model \((\text{obj})\) \n  – A fitted lifelines univariate parametric model, like \texttt{WeibullFitter}

• plot_kwargs – kwargs for the plot.

Returns The axes which was used.

Return type ax

Examples

```python
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.\texttt{cdf\_plot} \((\text{model, timeline=None, ax=None, **plot\_kwargs})\)

lifelines.plotting.\texttt{rmst\_plot} \((\text{model, model2=None, \texttt{t=inf, ax=None, text\_position=None, **plot\_kwargs}})\)

This functions plots the survival function of the model plus it’s area-under-the-curve (AUC) up until the point \(t\). The AUC is known as the restricted mean survival time (RMST).

To compare the difference between two models’ survival curves, you can supply an additional model in \texttt{model2}.

Parameters

• model \((\text{lifelines.UnivariateFitter})\)

• model2 \((\text{lifelines.UnivariateFitter, optional})\) – used to compute the delta RMST of two models
• `t (float)` – the upper bound of the expectation

• `ax (axis)`

• `text_position (tuple)` – move the text position of the RMST.

### Examples

```python
from lifelines.utils import restricted_mean_survival_time
from lifelines.datasets import load_waltons
from lifelines.plotting import rmst_plot

df = load_waltons()
ix = df['group'] == 'miR-137'
T, E = df['T'], df['E']
time_limit = 50

kmf_exp = KaplanMeierFitter().fit(T[ix], E[ix], label='exp')
kmf_con = KaplanMeierFitter().fit(T[~ix], E[~ix], label='control')

ax = plt.subplot(311)
rmst_plot(kmf_exp, t=time_limit, ax=ax)

ax = plt.subplot(312)
rmst_plot(kmf_con, t=time_limit, ax=ax)

ax = plt.subplot(313)
rmst_plot(kmf_exp, model2=kmf_con, t=time_limit, ax=ax)
```

```python
lifelines.plotting.loglogs_plot(cls, loc=None, iloc=None, show_censors=False, censor_styles=None, ax=None, **kwargs)
```

Specifies a plot of the log(-log(SV)) versus log(time) where SV is the estimated survival function.

### 1.14.5 datasets

```python
lifelines.datasets.load_c_botulinum_lag_phase(**kwargs)
```

A dataset from [1] that represents the duration of the lag phase for C. botulinum, measured in days. The data is left and right censored.

### References


```python
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>
</tbody>
</table>

(continues on next page)
lifelines.datasets.load_dd(**kwargs)

Classification of political regimes as democracy and dictatorship. Classification of democracies as parliamen-
tary, 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>start_date</th>
<th>end_date</th>
<th>reason</th>
<th>diff_days</th>
<th>observed</th>
</tr>
</thead>
<tbody>
<tr>
<td>1893-10-30</td>
<td>1893-10-30</td>
<td>Death</td>
<td>2363</td>
<td>True</td>
</tr>
</tbody>
</table>

Size: (1808, 12)
Example:

- ctryname: Afghanistan
- cowcode2: 700
- politycode: 700
- un_region_name: Southern Asia
- un_continent_name: Asia
- ehead: Mohammad Zahir Shah
- leaderspellreg: Mohammad Zahir Shah.Afghanistan.1946.1952.Mona...
- democracy: Non-democracy
- regime: Monarchy
- start_year: 1946
- duration: 7
- observed: 1

References

Cheibub, José Antonio, Jennifer Gandhi, and James Raymond Vreeland. 2010. “Democracy and Dictatorship

lifelines.datasets.load_dfcv()
A toy example of a time dependent dataset.

Size: (14, 6)
Example:

<table>
<thead>
<tr>
<th>start</th>
<th>group</th>
<th>z</th>
<th>stop</th>
<th>id</th>
<th>event</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.0</td>
<td>1.0</td>
<td>0</td>
<td>3.0</td>
<td>1</td>
<td>True</td>
</tr>
<tr>
<td>0.0</td>
<td>1.0</td>
<td>0</td>
<td>5.0</td>
<td>2</td>
<td>False</td>
</tr>
<tr>
<td>0.0</td>
<td>1.0</td>
<td>1</td>
<td>5.0</td>
<td>3</td>
<td>True</td>
</tr>
<tr>
<td>0.0</td>
<td>1.0</td>
<td>0</td>
<td>6.0</td>
<td>4</td>
<td>True</td>
</tr>
</tbody>
</table>

References


lifelines.datasets.load_diabetes(**kwargs)
An interval censored dataset.

References

I (insulin-dependent) diabetes mellitus.” Diabetologia, 28, 590-596.
lifelines.datasets.load_g3(**kwargs)

Size: (731, 3)
Example:

<table>
<thead>
<tr>
<th>left</th>
<th>right</th>
<th>gender</th>
</tr>
</thead>
<tbody>
<tr>
<td>24</td>
<td>27</td>
<td>male</td>
</tr>
<tr>
<td>22</td>
<td>22</td>
<td>female</td>
</tr>
<tr>
<td>37</td>
<td>39</td>
<td>male</td>
</tr>
<tr>
<td>20</td>
<td>20</td>
<td>male</td>
</tr>
<tr>
<td>1</td>
<td>16</td>
<td>male</td>
</tr>
<tr>
<td>8</td>
<td>20</td>
<td>female</td>
</tr>
<tr>
<td>14</td>
<td>14</td>
<td>male</td>
</tr>
</tbody>
</table>

lifelines.datasets.load_gbsg2(**kwargs)

A data frame containing the observations from the GBSG2 study of 686 women:

Size: (686,10)
Example:

<table>
<thead>
<tr>
<th>horTh</th>
<th>age</th>
<th>menostat</th>
<th>tsize</th>
<th>tgrade</th>
<th>pnodes</th>
<th>progres</th>
<th>estrec</th>
<th>time</th>
<th>cens</th>
</tr>
</thead>
<tbody>
<tr>
<td>yes</td>
<td>56</td>
<td>Post</td>
<td>12</td>
<td>II</td>
<td>7</td>
<td>61</td>
<td>77</td>
<td>2018</td>
<td>1</td>
</tr>
</tbody>
</table>

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.
lifelines.datasets.load_kidney_transplant(**kwargs)

Size: (863, 6)
Example:

time
death
age
black_male
white_male
black_female

lifelines.datasets.load_larynx(**kwargs)

Size: (89, 6)
Example:

time
age
deadh
Stage II
Stage III
Stage IV
0.6
1.3
2.4
2.5
3.2

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.


Size: (104, 3)
Example:

C
T
group
0
1
class
alluvial_fan
alluvial_fan
alluvial_fan
alluvial_fan
alluvial_fan

lifelines.datasets.load_leukemia(**kwargs)

Leukemia dataset.

Size: (42, 5)
Example:

t
status
sex
logWBC
Rx
0
1
2
### References

From http://web1.sph.emory.edu/dkleinb/allDatasets/surv2datasets/anderson.dat

**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.

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>32</td>
<td>0</td>
<td>1</td>
<td>2.53</td>
</tr>
<tr>
<td>4</td>
<td>25</td>
<td>0</td>
<td>1</td>
<td>1.78</td>
</tr>
</tbody>
</table>

**lifelines.datasets.load_lupus(**kwargs**)

See https://projecteuclid.org/download/pdf_1/euclid.aos/1176345693

**lifelines.datasets.load_lymph_node(**kwargs**)

References


Notes

In lifelines v0.23.7, two rows were updated with more correct data (transcription problems originally.)


randomized 2 x 2 trial evaluating hormonal treatment and the duration of chemotherapy in node-positive breast cancer patients. Journal of Clinical Oncology, 12, 2086-2093.


```python
lifelines.datasets.load_lymphoma(**kwargs)
```

<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

```python
lifelines.datasets.load_multicenter_aids_cohort_study(**kwargs)
```

Originally in [1]:

<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


```python
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.


<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_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)


<table>
<thead>
<tr>
<th>ID</th>
<th>Subject Identification</th>
<th>Age</th>
<th>TREAT</th>
<th>TIME0</th>
<th>TIME1</th>
<th>CENSOR</th>
<th>EVENT</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1 - 400</td>
<td>years</td>
<td>0 = New</td>
<td>Days</td>
<td>Days</td>
<td>1 = Episode Occurred</td>
<td>0 to at most 4</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td>1 = Old</td>
<td></td>
<td></td>
<td>at TIME1</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td>or censoring</td>
<td></td>
<td></td>
<td>0 = Censored</td>
<td></td>
</tr>
</tbody>
</table>

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)
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:

Size: (432, 9)
Example:

week  20
arrest  1
fin  0
age  27
race  1
wexp  0
mar  0
paro  1
prio  3

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:

Size: (172, 8)
Example:

start stop event age year surgery transplant id
0.0  50.0  1  -17.155373  0.123203  0  0  1
0.0   6.0  1   3.835729  0.254620  0  0  2
0.0   1.0  0   6.297057  0.265572  0  0  3
1.0  16.0  1   6.297057  0.265572  0  1  3
0.0  36.0  0  -7.737166  0.490075  0  0  4

References


lifelines.datasets.load_static_test(**kwargs)

Size: (7, 5)
Example:

id  t  E  var1  var2
1  4  1  -1  -1

(continues on next page)
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:
   T  E  group
  6 1  miR-137
 13 1  miR-137
 13 1  miR-137
 13 1  miR-137
 19 1  miR-137
```

### 1.15 Changelog

#### 1.15.1 0.24.3 - 2020-03-25

**New features**

- new `logx` kwarg in plotting curves
- PH models have `compute_followup_hazard_ratios` for simulating what the hazard ratio would be at previous times. This is useful because the final hazard ratio is some weighted average of these.

**Bug fixes**

- Fixed error in HTML printer that was hiding concordance index information.

#### 1.15.2 0.24.2 - 2020-03-15

**Bug fixes**

- Fixed bug when no covariates were passed into `CoxPHFitter`. See #975
- Fixed error in `StatisticalResult` where the test name was not displayed correctly.
- Fixed a keyword bug in `plot_covariate_groups` for parametric models.

#### 1.15.3 0.24.1 - 2020-03-05
New features

• Stability improvements for GeneralizedGammaRegressionFitter and CoxPHFitter with spline estimation.

Bug fixes

• Fixed bug with plotting hazards in NelsonAalenFitter.

1.15.4 0.24.0 - 2020-02-20

This version and future versions of lifelines no longer support py35. Pandas 1.0 is fully supported, along with previous versions. Minimum Scipy has been bumped to 1.2.0.

New features

• CoxPHFitter and CoxTimeVaryingFitter has support for an elastic net penalty, which includes L1 and L2 regression.

• CoxPHFitter has new baseline survival estimation methods. Specifically, spline now estimates the coefficients and baseline survival using splines. The traditional method, breslow, is still the default however.

• Regression models have a new score method that will score your model against a dataset (ex: a testing or validation dataset). The default is to evaluate the log-likelihood, but also the concordance index can be chose.

• New MixtureCureFitter for quickly creating univariate mixture models.

• Univariate parametric models have a plot_density, density_at_times, and property density_ that computes the probability density function estimates.

• new dataset for interval regression involving C. Botulinum.

• new lifelines.fitters.mixins.ProportionalHazardMixin that implements proportional hazard checks.

API Changes

• Models’ prediction method that return a single array now return a Series (use to return a DataFrame). This includes predict_median, predict_percentile, predict_expectation, predict_log_partial_hazard, and possibly others.

• The penalty in Cox models is now scaled by the number of observations. This makes it invariant to changing sample sizes. This change also make the penalty magnitude behave the same as any parametric regression model.

• score_ on models has been renamed concordance_index_

• models’.variance_matrix_ is now a DataFrame.

• CoxTimeVaryingFitter no longer requires an id_col. It’s optional, and some checks may be done for integrity if provided.

• Significant changes to utils.k_fold_cross_validation.

• removed automatically adding inf from PiecewiseExponentialRegressionFitter. breakpoints and PiecewiseExponentialFitter.breakpoints

• tie_method was dropped from Cox models (it was always Efron anyways...)
- Mixins are moved to `lifelines.fitters.mixins`
- `find_best_parametric_model` evaluation kwarg has been changed to `scoring_method`.
- removed `_score_` and `path` from Cox model.

**Bug fixes**

- Fixed `show_censors` with `KaplanMeierFitter.plot_cumulative_density` see issue #940.
- Fixed error in "BIC" code path in `find_best_parametric_model`
- Fixed a bug where left censoring in AFT models was not converging well
- Cox models now incorporate any penalizers in their `log_likelihood_`

**1.15.5 0.23.9 - 2020-01-28**

**Bug fixes**

- fixed important error when a parametric regression model would not assign the correct labels to fitted parameters’ variances. See more here: https://github.com/CamDavidsonPilon/lifelines/issues/931. Users of `GeneralizedGammaRegressionFitter` and any custom regression models should update their code as soon as possible.

**1.15.6 0.23.8 - 2020-01-21**

**Bug fixes**

- fixed important error when a parametric regression model would not assign the correct labels to fitted parameters. See more here: https://github.com/CamDavidsonPilon/lifelines/issues/931. Users of `GeneralizedGammaRegressionFitter` and any custom regression models should update their code as soon as possible.

**1.15.7 0.23.7 - 2020-01-14**

Bug fixes for py3.5.

**1.15.8 0.23.6 - 2020-01-07**

**New features**

- New univariate model, `SplineFitter`, that uses cubic splines to model the cumulative hazard.
- To aid users with selecting the best parametric model, there is a new `lifelines.utils.find_best_parametric_model` function that will iterate through the models and return the model with the lowest AIC (by default).
- custom parametric regression models can now do left and interval censoring.
1.15.9 0.23.5 - 2020-01-05

New features

• New predict_hazard for parametric regression models.
• New lymph node cancer dataset, originally from *H.F. for the German Breast Cancer Study Group (GBSG) (1994)*

Bug fixes

• fixes error thrown when converge of regression models fails.
• kwargs is now used in plot_covariate_groups
• fixed bug where large exponential numbers in print_summary were not being suppressed correctly.

1.15.10 0.23.4 - 2019-12-15

• Bug fix for PyPI

1.15.11 0.23.3 - 2019-12-11

New features

• StatisticalResult.print_summary supports html output.

Bug fixes

• fix import in printer.py
• fix html printing with Univariate models.

1.15.12 0.23.2 - 2019-12-07

New features

• new lifelines.plotting.rmst_plot for pretty figures of survival curves and RMSTs.
• new variance calculations for lifelines.utils.restricted_mean_survival_time
• performance improvements on regression models’ preprocessing. Should make datasets with high number of columns more performant.

Bug fixes

• fixed print_summary for AAF class.
• fixed repr for sklearn_adapter classes.
• fixed conditional_after in Cox model with strata was used.
1.15.13 0.23.1 - 2019-11-27

New features

- `new print_summary option style` to print HTML, LaTeX or ASCII output
- performance improvements for `CoxPHFitter` - up to 30% performance improvements for some datasets.

Bug fixes

- fixed bug where computed statistics were not being shown in `print_summary` for HTML output.
- fixed bug where “None” was displayed in models’ `__repr__`
- fixed bug in `StatisticalResult.print_summary`
- fixed bug when using `print_summary` with left censored models.
- lots of minor bug fixes.

1.15.14 0.23.0 - 2019-11-17

New features

- `new print_summary abstraction` that allows HTML printing in Jupyter notebooks!
- silenced some warnings.

Bug fixes

- The “comparison” value of some parametric univariate models wasn’t standard, so the null hypothesis p-value may have been wrong. This is now fixed.
- fixed a NaN error in confidence intervals for KaplanMeierFitter

API Changes

- To align values across models, the column names for the confidence intervals in parametric univariate models have changed.
- Fixed typo in `ParametricUnivariateFitter` name.
- `median_` has been removed in favour of `median_survival_time_`.
- `left_censorship in fit` has been removed in favour of `fit_left_censoring`.

1.15.15 0.22.10 - 2019-11-08

The tests were re-factored to be shipped with the package. Let me know if this causes problems.

Bug fixes

- fixed error in plotting models with “lower” or “upper” was in the label name.
- fixed bug in `plot_covariate_groups` for AFT models when >1d arrays were used for values arg.
1.15.16  0.22.9 - 2019-10-30

Bug fixes

- fixed `predict_` methods in AFT models when `timeline` was not specified.
- fixed error in `qq_plot`
- fixed error when submitting a model in `qth_survival_time`
- `CoxPHFitter` now displays correct columns values when changing alpha param.

1.15.17  0.22.8 - 2019-10-06

New features

- Serializing lifelines is better supported. Packages like joblib and pickle are now supported. Thanks @Abdeal-iJK!
- `conditional_after` now available in `CoxPHFitter.predict_median`
- Suppressed some unimportant warnings.

Bug fixes

- fixed `initial_point` being ignored in AFT models.

1.15.18  0.22.7 - 2019-09-29

New features

- new `ApproximationWarning` to tell you if the package is making an potentially mislead approximation.

Bug fixes

- fixed a bug in parametric prediction for interval censored data.
- realigned values in `print_summary`.
- fixed bug in `survival_difference_at_fixed_point_in_time_test`

API Changes

- `utils.qth_survival_time` no longer takes a `cdf` argument - users should take the compliment (1-cdf).
- Some previous `StatisticalWarnings` have been replaced by `ApproximationWarning`

1.15.19  0.22.6 - 2019-09-25

New features

- `conditional_after` works for `CoxPHFitter` prediction models
Bug fixes

API Changes

- CoxPHFitter.baseline_cumulative_hazard_'s column is renamed "baseline cumulative hazard" - previously it was "baseline hazard". (Only applies if the model has no strata.)
- utils.dataframe_interpolate_at_times renamed to utils.interpolate_at_times_and_return_pandas.

1.15.20 0.22.5 - 2019-09-20

New features

- Improvements to the repr of models that takes into accounts weights.
- Better support for predicting on Pandas Series

Bug fixes

- Fixed issue where fit_interval_censoring wouldn’t accept lists.
- Fixed an issue with AalenJohansenFitter failing to plot confidence intervals.

API Changes

- _get_initial_value in parametric univariate models is renamed _create_initial_point

1.15.21 0.22.4 - 2019-09-04

New features

- Some performance improvements to regression models.
- lifelines will avoid penalizing the intercept (aka bias) variables in regression models.
- new utils.restricted_mean_survival_time that approximates the RMST using numerical integration against survival functions.

API changes

- KaplanMeierFitter.survival_function_'s' index is no longer given the name “timeline”.

Bug fixes

- Fixed issue where concordance_index would never exit if NaNs in dataset.
**1.15.22 0.22.3 - 2019-08-08**

**New features**

- model’s now expose a `log_likelihood_` property.
- new `conditional_after` argument on `predict_*` methods that make prediction on censored subjects easier.
- new `lifelines.utils.safe_exp` to make `exp` overflows easier to handle.
- smarter initial conditions for parametric regression models.
- New regression model: `GeneralizedGammaRegressionFitter`

**API changes**

- removed `lifelines.utils.gamma` - use `autograd_gamma` library instead.
- removed bottleneck as a dependency. It offered slight performance gains only in Cox models, and only a small fraction of the API was being used.

**Bug fixes**

- AFT log-likelihood ratio test was not using weights correctly.
- corrected (by bumping) scipy and autograd dependencies
- convergence is improved for most models, and many `exp` overflow warnings have been eliminated.
- Fixed an error in the `predict_percentile` of `LogLogisticAFTFitter`. New tests have been added around this.

**1.15.23 0.22.2 - 2019-07-25**

**New features**

- lifelines is now compatible with `scipy>=1.3.0`

**Bug fixes**

- fixed printing error when using robust=True in regression models
- `GeneralizedGammaFitter` is more stable, maybe.
- lifelines was allowing old version of numpy (1.6), but this caused errors when using the library. The correctly numpy has been pinned (to 1.14.0+)

**1.15.24 0.22.1 - 2019-07-14**

**New features**

- New univariate model, `GeneralizedGammaFitter`. This model contains many sub-models, so it is a good model to check fits.
• added a warning when a time-varying dataset had instantaneous deaths.
• added a `initial_point` option in univariate parametric fitters.
• `initial_point` kwarg is present in parametric univariate fitters `.fit`
• `event_table` is now an attribute on all univariate fitters (if right censoring)
• improvements to `lifelines.utils.gamma`

API changes

• In AFT models, the column names in `confidence_intervals_` has changed to include the alpha value.
• In AFT models, some column names in `.summary` and `.print_summary` has changed to include the alpha value.
• In AFT models, some column names in `.summary` and `.print_summary` includes confidence intervals for the exponential of the value.

Bug fixes

• when using `censors_show` in plotting functions, the censor ticks are now reactive to the estimate being shown.
• fixed an overflow bug in `KaplanMeierFitter` confidence intervals
• improvements in data validation for `CoxTimeVaryingFitter`

1.15.25  0.22.0 - 2019-07-03

New features

• Ability to create custom parametric regression models by specifying the cumulative hazard. This enables new and extensions of AFT models.
• `percentile(p)` method added to univariate models that solves the equation \( p = S(t) \) for \( t \)
• for parametric univariate models, the `conditional_time_to_event_` is now exact instead of an approximation.

API changes

• In Cox models, the attribute `hazards_` has been renamed to `params_`. This aligns better with the other regression models, and is more clear (what is a hazard anyways?)
• In Cox models, a new `hazard_ratios_` attribute is available which is the exponentiation of `params_`
• In Cox models, the column names in `confidence_intervals_` has changed to include the alpha value.
• In Cox models, some column names in `.summary` and `.print_summary` has changed to include the alpha value.
• In Cox models, some column names in `.summary` and `.print_summary` includes confidence intervals for the exponential of the value.
• Significant changes to internal AFT code.
• A change to how fit_intercept works in AFT models. Previously one could set fit_intercept to False and not have to set ancillary_df - now one must specify a DataFrame.

Bug fixes

• for parametric univariate models, the conditional_time_to_event_ is now exact instead of an approximation.
• fixed a name error bug in CoxTimeVaryingFitter.plot

1.15.26 0.21.5 - 2019-06-22

I’m skipping 0.21.4 version because of deployment issues.

New features

• scoring_method now a kwarg on sklearn_adapter

Bug fixes

• fixed an implicit import of scikit-learn. scikit-learn is an optional package.
• fixed visual bug that misaligned x-axis ticks and at-risk counts. Thanks @christopherahern!

1.15.27 0.21.3 - 2019-06-04

New features

• include in lifelines is a scikit-learn adapter so lifeline’s models can be used with scikit-learn’s API. See documentation here.
• CoxPHFitter.plot now accepts a hazard_ratios (boolean) parameter that will plot the hazard ratios (and CIs) instead of the log-hazard ratios.
• CoxPHFitter.check_assumptions now accepts a columns parameter to specify only checking a subset of columns.

Bug fixes

• covariates_from_event_matrix handle nulls better

1.15.28 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

1.15.29 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!

1.15.30 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

1.15.31 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`

1.15.32 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 *`. 
1.15.33 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_.

1.15.34 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

1.15.35 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.

1.15.36 0.20.0 - 2019-03-05

• Starting with 0.20.0, only Python3 will be supported. Over 75% of recent installs where 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`.

1.15.37 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.

1.15.38 0.19.4 - 2019-02-25

Bug fixes

• remove some bad print statements in `CoxPHFitter`.
1.15.39 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.

1.15.40 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.

1.15.41 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 PiecewiseExponential to the same as ExponentialFitter (from $\lambda \times t$ to $t / \lambda$).

1.15.42 0.19.0 - 2019-02-20

New features

- New regression model WeibullAFTFitter for fitting accelerated failure time models. Docs have been added to our documentation about how to use WeibullAFTFitter (spoiler: it’s API is similar to the other regression models) and how to interpret the output.
- CoxPHFitter performance improvements (about 10%)
- CoxTimeVaryingFitter performance improvements (about 10%)

1.15. Changelog
API changes

- **Important**: we changed the .hazards_ and .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 .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 WeibullFitter and 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 AalenAdditiveModel), the name of the added column has been changed from baseline to _intercept

- **Important**: the meaning of alpha in all fitters has changed to be the standard interpretation of alpha in confidence intervals. That means that the default for alpha is set to 0.05 in the latest lifelines, instead of 0.95 in previous versions.

Bug Fixes

- Fixed a bug in the _log_likelihood_ property of 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 _print_summary_, 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 _event_ columns.

- Fixed a calculation bug in the concordance index for stratified Cox models. Thanks @airanmehr!

- Fixed some Pandas <0.24 bugs.

1.15.43  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_.

1.15.44  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 edgcase 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!
1.15.45 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

1.15.46 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.

1.15.47 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.

1.15.48 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.

1.15.49 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 astericks 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.

1.15.50 0.17.5 - 2019-01-25
• more bugs in plot_covariate_groups fixed when using non-numeric strata.

1.15.51 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

1.15.52 0.17.3 - 2019-01-24
• Fix in compute_residuals when using schoenfeld and the minimum duration has only censored subjects.

1.15.53 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

1.15.54 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.
1.15.55 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

1.15.56 0.16.3 - 2019-01-03

• More CoxPHFitter performance improvements. Up to a 40% reduction vs 0.16.2 for some datasets.

1.15.57 0.16.2 - 2019-01-02

• Fixed CoxTimeVaryingFitter to allow more than one variable to be stratified
• Significant performance improvements for CoxPHFitter with dataset has lots of duplicate times. See https://github.com/CamDavidsonPilon/lifelines/issues/591

1.15.58 0.16.1 - 2019-01-01

• Fixed py2 division error in concordance method.

1.15.59 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 thesholds 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.

1.15.60 0.15.4
• bug fix for the Cox model likelihood ratio test when using non-trivial weights.

1.15.61 0.15.3 - 2018-12-18
• Only allow matplotlib less than 3.0.

1.15.62 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.

1.15.63 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.

1.15.64 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 `CoxTimeVaryFitter` 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.

1.15.65 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.

1.15.66 0.14.5 - 2018-06-29
• fix for n > 2 groups in `multivariate_logrank_test`
• fix weights in `KaplanMeierFitter` when using a pandas Series.
1.15.67 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`

1.15.68 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.

1.15.69 0.14.2 - 2018-05-18

- some quality of life improvements for working with `CoxTimeVaryingFitter` including new `predict_*` methods.

1.15.70 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.

1.15.71 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.

1.15.72 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.

1.15.73 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 many 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.

1.15.74 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.
1.15.75 0.11.2

- Changing license to valilla MIT.
- Speed up `NelsonAalenFitter.fit` considerably.

1.15.76 0.11.1 - 2017-06-22

- Pythom3 fix for `CoxPHFitter.plot`.

1.15.77 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`.

1.15.78 0.10.1 - 2017-06-05

- fix in internal normalization for `CoxPHFitter` predict methods.

1.15.79 0.10.0

- 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`

1.15.80 0.9.4

- adding `plot_loglogs` 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`. 

312 Chapter 1. Contents:
1.15.81 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.

1.15.82 0.9.1

- add two summary functions to Weibull and Exponential fitter, solves #224

1.15.83 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

1.15.84 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.

1.15.85 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.
1.15.86 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.

1.15.87 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.

1.15.88 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.

1.15.89 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.
1.15.90 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.

1.15.91 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.

1.15.92 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.

1.15.93 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 \mid 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.

1.15.94 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

1.15.95 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`.

1.15.96 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 =)
1.15.97 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.

1.16 Contributing to lifelines

1.16.1 Questions about survival analysis?

If you are using lifelines for survival analysis and have a question about “how do I do X?” or “what does Y do?”, the best place to ask that is either in our gitter channel or at stats.stackexchange.com.

1.16.2 Submitting bugs or other errors observed

We appreciate all bug reports submitted, as this will help the entire community get a better product. Please open up an issue in the Github Repository. If possible, please provide a code snippet, and what version of lifelines you are using.

1.16.3 Submitting new feature requests

Please open up an issue in the Github Repository with as much context as possible about the feature you would like to see. Also useful is to link to other libraries/software that have that feature.

1.16.4 Submitting code, or other changes

If you are interested in contributing to lifelines (and we thank you for the interest!), we recommend first opening up an issue in the GitHub repository to discuss the changes. From there, we can together plan how to execute the changes. See the Development section below for how to setup a local environment.

1.17 Development

1.17.1 Setting up a lifelines development environment

1. From the root directory of lifelines activate your virtual environment (if you plan to use one).
2. Install the development requirements and pre-commit hooks. If you are on Mac, Linux, or Windows WSL you can use the provided Makefile. Just type make into the console and you’re ready to start developing. This will also install the dev-requirements.

1.17.2 Formatting

lifelines uses the black python formatter. There are 3 different ways to format your code.

1. Use the Makefile.
   
   make lint
2. Call `black` directly and pass the correct line length.
   
   ```bash
   black . -l 120
   ```

3. Have your code formatted automatically during commit with the `pre-commit` hook.
   
   • Stage and commit your unformatted changes:
     
     ```bash
     git commit -m "your_commit_message"
     ```
   
   • Code that needs to be formatted will “fail” the commit hooks and be formatted for you.
   
   • Stage the newly formatted python code:
     
     ```bash
     git add *.py
     ```
   
   • Recall your original commit command and commit again:
     
     ```bash
     git commit -m "your_commit_message"
     ```

### 1.17.3 Running the tests

You can optionally run the test suite after install with

```bash
py.test
```

### 1.18 Installation

```bash
pip install lifelines
```

or

```bash
conda install -c conda-forge lifelines
```

### 1.19 Source code and issue tracker

Available on Github, [CamDavidsonPilon/lifelines](https://github.com/CamDavidsonPilon/lifelines). Please report bugs, issues and feature extensions there. We also have [Gitter channel](https://gitter.im/lifelines/lifelines) available to discuss survival analysis and *lifelines*:

### 1.20 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
\begin{itemize}
\item lifelines.datasets, 281
\item lifelines.fitters.aalen_additive_fitter, 198
\item lifelines.fitters.aalen_johansen_fitter, 138
\item lifelines.fitters.breslow_fleming_harrington_fitter, 141
\item lifelines.fitters.cox_time_varying_fitter, 215
\item lifelines.fitters.coxph_fitter, 202
\item lifelines.fitters.exponential_fitter, 144
\item lifelines.fitters.generalized_gamma_fitter, 149
\item lifelines.fitters.generalized_gamma_regression_fitter, 220
\item lifelines.fitters.kaplan_meier_fitter, 155
\item lifelines.fitters.log_logistic_aft_fitter, 228
\item lifelines.fitters.log_logistic_fitter, 160
\item lifelines.fitters.log_normal_aft_fitter, 237
\item lifelines.fitters.log_normal_fitter, 166
\item lifelines.fitters.mixture_cure_fitter, 172
\item lifelines.fitters.nelson_aalen_fitter, 177
\item lifelines.fitters.piecewise_exponential_fitter, 181
\item lifelines.fitters.piecewise_exponential_regression_fitter, 246
\item lifelines.fitters.spline_fitter, 186
\item lifelines.fitters.weibull_aft_fitter, 253
\item lifelines.fitters.weibull_fitter, 192
\item lifelines.plotting, 279
\item lifelines.statistics, 272
\item lifelines.utils, 263
\end{itemize}
Index

A

AalenAdditiveFitter (class in lifelines.fitters.aalen_additive_fitter), 198
AalenJohansenFitter (class in lifelines.fitters.aalen_johansen_fitter), 138
add_at_risk_counts() (in module lifelines.plotting), 279
add_covariate_to_timeline() (in module lifelines.utils), 270
alpha_ (lifelines.fitters.generalized_gamma_fitter.GeneralizedGammaFitter attribute), 151
alpha_ (lifelines.fitters.generalized_gamma_regression_fitter.GeneralizedGammaRegressionFitter attribute), 221
alpha_ (lifelines.fitters.log_logistic_fitter.LogLogisticFitter attribute), 162
ascii_print () (lifelines.statistics.StatisticalResult method), 273
B

baseline_cumulative_hazard_ (lifelines.fitters.cox_time_varying_fitter.CoxTimeVaryingFitter attribute), 216
baseline_cumulative_hazard_ (lifelines.fitters.coxph_fitter.CoxPHFitter attribute), 203
baseline_hazard_ (lifelines.fitters.coxph_fitter.CoxPHFitter attribute), 203
baseline_survival_ (lifelines.fitters.cox_time_varying_fitter.CoxTimeVaryingFitter attribute), 216
baseline_survival_ (lifelines.fitters.coxph_fitter.CoxPHFitter attribute), 203
basis() (lifelines.fitters.spline_fitter.SplineFitter method), 188
beta_ (lifelines.fitters.log_logistic_fitter.LogLogisticFitter attribute), 162
breakpoints (lifelines.fitters.piecewise_exponential_fitter.PiecewiseExponentialFitter)
BreslowFlemingHarringtonFitter (class in lifelines.fitters.breslow_fleming_harrington_fitter), 141
BreslowFlemingHarringtonFitter (class in life- lines.fitters.breslow_fleming_harrington_fitter), 141
check_assumptions() (lifelines.fitters.cox_time_varying_fitter.CoxTimeVaryingFitter method), 216
check_assumptions() (lifelines.fitters.coxph_fitter.CoxPHFitter method), 203
check_assumptions() (lifelines.fitters.weibull_aft_fitter.WeibullAFTFitter method), 254
cdf_plot() (in module lifelines.plotting), 280
compute_followup_hazard_ratios() (lifelines.fitters.cox_time_varying_fitter.CoxTimeVaryingFitter method), 217
compute_followup_hazard_ratios() (lifelines.fitters.coxph_fitter.CoxPHFitter method), 204
compute_followup_hazard_ratios() (lifelines.fitters.weibull_aft_fitter.WeibullAFTFitter method), 255
compute_residuals() (lifelines.fitters.aalen_additive_fitter.AalenAdditiveFitter method), 199
compute_residuals() (lifelines.fitters.cox_time_varying_fitter.CoxTimeVaryingFitter method), 217
compute_residuals() (lifelines.fitters.coxph_fitter.CoxPHFitter method), 204
compute_residuals() (lifelines.fitters.generalized_gamma_regression_fitter.GeneralizedGammaRegressionFitter)
compute_residuals() (lifelines.fitters.log_logistic_aft_fitter.LogLogisticAFTFitter)

323
method), 229

compute_residuals() (lifelines.fitters.log_normal_aft_fitter.LogNormalAFTFitter method), 238

compute_residuals() (lifelines.fitters.piecewise_exponential_regression_fitter.PiecewiseExponentialRegressionFitter method), 247

compute_residuals() (lifelines.fitters.weibull_aft_fitter.WeibullAFTFitter method), 255

concordance_index() (in module lifelines.utils), 268

concordance_index (lifelines.fitters.coxph_fitter.CoxPHFitter attribute), 205

concordance_index (lifelines.fitters.generalized_gamma_regression_fitter.GeneralizedGammaRegressionFitter attribute), 222

concordance_index (lifelines.fitters.log_logistic_aft_fitter.LogLogisticAFTFitter attribute), 229

concordance_index (lifelines.fitters.log_normal_aft_fitter.LogNormalAFTFitter attribute), 238

concordance_index (lifelines.fitters.piecewise_exponential_regression_fitter.PiecewiseExponentialRegressionFitter attribute), 247

concordance_index (lifelines.fitters.weibull_aft_fitter.WeibullAFTFitter attribute), 255

conditional_time_to_event (lifelines.fitters.aalen_johansen_fitter.AalenJohansenFitter attribute), 139

conditional_time_to_event (lifelines.fitters.breslow_fleming_harrington_fitter.BreslowFlemingHarringtonFitter attribute), 141

conditional_time_to_event (lifelines.fitters.exponential_fitter.ExponentialFitter attribute), 145

conditional_time_to_event (lifelines.fitters.generalized_gamma_fitter.GeneralizedGammaFitter attribute), 151

conditional_time_to_event (lifelines.fitters.kaplan_meier_fitter.KaplanMeierFitter attribute), 156

conditional_time_to_event (lifelines.fitters.log_logistic_fitter.LogLogisticFitter attribute), 162

conditional_time_to_event (lifelines.fitters.log_normal_fitter.LogNormalFitter attribute), 168

conditional_time_to_event (lifelines.fitters.mixture_cure_fitter.MixtureCureFitter attribute), 173

conditional_time_to_event (lifelines.fitters.nelson_aalen_fitter.NelsonAalenFitter attribute), 178

conditional_time_to_event (lifelines.fitters.piecewise_exponential_fitter.PiecewiseExponentialFitter attribute), 182

conditional_time_to_event (lifelines.fitters.spline_fitter.SplineFitter attribute), 188

conditional_time_to_event (lifelines.fitters.weibull_fitter.WeibullFitter attribute), 194

confidence_interval (lifelines.fitters.exponential_fitter.ExponentialFitter attribute), 145

confidence_interval (lifelines.fitters.generalized_gamma_fitter.GeneralizedGammaFitter attribute), 151

confidence_interval (lifelines.fitters.kaplan_meier_fitter.KaplanMeierFitter attribute), 156

confidence_interval (lifelines.fitters.log_logistic_fitter.LogLogisticFitter attribute), 162

confidence_interval (lifelines.fitters.log_normal_fitter.LogNormalFitter attribute), 168

confidence_interval (lifelines.fitters.mixture_cure_fitter.MixtureCureFitter attribute), 173

confidence_interval (lifelines.fitters.nelson_aalen_fitter.NelsonAalenFitter attribute), 178

confidence_interval (lifelines.fitters.piecewise_exponential_fitter.PiecewiseExponentialFitter attribute), 182

confidence_interval (lifelines.fitters.spline_fitter.SplineFitter attribute), 188

confidence_interval (lifelines.fitters.weibull_fitter.WeibullFitter attribute), 194

confidence_interval_cumulative_density (lifelines.fitters.exponential_fitter.ExponentialFitter attribute), 145

confidence_interval_cumulative_density (lifelines.fitters.generalized_gamma_fitter.GeneralizedGammaFitter attribute), 151

confidence_interval_cumulative_density (lifelines.fitters.kaplan_meier_fitter.KaplanMeierFitter attribute), 156

confidence_interval_cumulative_density (lifelines.fitters.log_logistic_fitter.LogLogisticFitter attribute), 162

confidence_interval_cumulative_density (lifelines.fitters.log_normal_fitter.LogNormalFitter attribute), 168

confidence_interval_cumulative_density (lifelines.fitters.mixture_cure_fitter.MixtureCureFitter attribute), 173

324 Index
<table>
<thead>
<tr>
<th>Function/Attribute</th>
<th>Module/Class</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>confidence_interval_survival_function</code></td>
<td>(lifelines.fitters.spline_fitter.SplineFitter attribute)</td>
<td>188</td>
</tr>
<tr>
<td><code>confidence_interval_survival_function</code></td>
<td>(lifelines.fitters.weibull_fitter.WeibullFitter attribute)</td>
<td>194</td>
</tr>
<tr>
<td><code>confidence_intervals</code></td>
<td>(lifelines.fitters.aalen_additive_fitter.AalenAdditiveFitter attribute)</td>
<td>199</td>
</tr>
<tr>
<td><code>confidence_intervals</code></td>
<td>(lifelines.fitters.cox_time_varying_fitter.CoxTimeVaryingFitter attribute)</td>
<td>215</td>
</tr>
<tr>
<td><code>confidence_intervals</code></td>
<td>(lifelines.fitters.coxph_fitter.CoxPHFitter attribute)</td>
<td>203</td>
</tr>
<tr>
<td><code>confidence_intervals</code></td>
<td>(lifelines.fitters.log_logistic_aft_fitter.LogLogisticAFTFitter attribute)</td>
<td>229</td>
</tr>
<tr>
<td><code>confidence_intervals</code></td>
<td>(lifelines.fitters.log_normal_aft_fitter.LogNormalAFTFitter attribute)</td>
<td>238</td>
</tr>
<tr>
<td><code>confidence_intervals</code></td>
<td>(lifelines.fitters.weibull_aft_fitter.WeibullAFTFitter attribute)</td>
<td>254</td>
</tr>
<tr>
<td><code>covariates_from_event_matrix()</code></td>
<td>(in module lifelines.utils)</td>
<td>271</td>
</tr>
<tr>
<td><code>CoxPHFitter</code></td>
<td>(class in lifelines.fitters.coxph_fitter)</td>
<td>202</td>
</tr>
<tr>
<td><code>CoxTimeVaryingFitter</code></td>
<td>(class in lifelines.fitters.cox_time_varying_fitter)</td>
<td>215</td>
</tr>
<tr>
<td><code>cumulative_density_</code></td>
<td>(lifelines.fitters.exponential_fitter.ExponentialFitter attribute)</td>
<td>145</td>
</tr>
<tr>
<td><code>cumulative_density_</code></td>
<td>(lifelines.fitters.generalized_gamma_fitter.GeneralizedGammaFitter attribute)</td>
<td>150</td>
</tr>
<tr>
<td><code>cumulative_density_</code></td>
<td>(lifelines.fitters.generalized_gamma_regression_fitter.GeneralizedGammaRegressionFitter attribute)</td>
<td>221</td>
</tr>
<tr>
<td><code>cumulative_density_</code></td>
<td>(lifelines.fitters.kaplan_meier_fitter.KaplanMeierFitter attribute)</td>
<td>157</td>
</tr>
<tr>
<td><code>cumulative_density_</code></td>
<td>(lifelines.fitters.nelson_aalen_fitter.NelsonAalenFitter attribute)</td>
<td>178</td>
</tr>
<tr>
<td><code>cumulative_density_</code></td>
<td>(lifelines.fitters.piecewise_exponential_fitter.PiecewiseExponentialFitter attribute)</td>
<td>182</td>
</tr>
<tr>
<td><code>cumulative_density_</code></td>
<td>(lifelines.fitters.spline_fitter.SplineFitter attribute)</td>
<td>188</td>
</tr>
<tr>
<td><code>cumulative_density_</code></td>
<td>(lifelines.fitters.weibull_fitter.WeibullFitter attribute)</td>
<td>194</td>
</tr>
<tr>
<td><code>cumulative_hazard_</code></td>
<td>(lifelines.fitters.exponential_fitter.ExponentialFitter attribute)</td>
<td>144</td>
</tr>
<tr>
<td><code>cumulative_hazard_</code></td>
<td>(lifelines.fitters.generalized_gamma_fitter.GeneralizedGammaFitter attribute)</td>
<td>150</td>
</tr>
<tr>
<td><code>cumulative_hazard_</code></td>
<td>(lifelines.fitters.piecewise_exponential_fitter.PiecewiseExponentialFitter attribute)</td>
<td>188</td>
</tr>
<tr>
<td><code>cumulative_hazard_</code></td>
<td>(lifelines.fitters.spline_fitter.SplineFitter attribute)</td>
<td>187</td>
</tr>
<tr>
<td><code>cumulative_hazard_</code></td>
<td>(lifelines.fitters.weibull_fitter.WeibullFitter attribute)</td>
<td>193</td>
</tr>
<tr>
<td><code>cumulative_density_at_times()</code></td>
<td>(lifelines.fitters.aalen_johansen_fitter.AalenJohansenFitter method)</td>
<td>139</td>
</tr>
<tr>
<td><code>cumulative_density_at_times()</code></td>
<td>(lifelines.fitters.breslow_fleming_harrington_fitter.BreslowFlemingHarringtonFitter method)</td>
<td>141</td>
</tr>
<tr>
<td><code>cumulative_density_at_times()</code></td>
<td>(lifelines.fitters.exponential_fitter.ExponentialFitter method)</td>
<td>146</td>
</tr>
<tr>
<td><code>cumulative_density_at_times()</code></td>
<td>(lifelines.fitters.generalized_gamma_fitter.GeneralizedGammaFitter method)</td>
<td>152</td>
</tr>
<tr>
<td><code>cumulative_density_at_times()</code></td>
<td>(lifelines.fitters.kaplan_meier_fitter.KaplanMeierFitter method)</td>
<td>157</td>
</tr>
<tr>
<td><code>cumulative_density_at_times()</code></td>
<td>(lifelines.fitters.nelson_aalen_fitter.NelsonAalenFitter method)</td>
<td>178</td>
</tr>
<tr>
<td><code>cumulative_density_at_times()</code></td>
<td>(lifelines.fitters.piecewise_exponential_fitter.PiecewiseExponentialFitter method)</td>
<td>182</td>
</tr>
<tr>
<td><code>cumulative_density_at_times()</code></td>
<td>(lifelines.fitters.spline_fitter.SplineFitter method)</td>
<td>187</td>
</tr>
<tr>
<td><code>cumulative_density_at_times()</code></td>
<td>(lifelines.fitters.weibull_fitter.WeibullFitter method)</td>
<td>194</td>
</tr>
<tr>
<td><code>cumulative_hazard_at_times()</code></td>
<td>(lifelines.fitters.exponential_fitter.ExponentialFitter attribute)</td>
<td>144</td>
</tr>
<tr>
<td><code>cumulative_hazard_at_times()</code></td>
<td>(lifelines.fitters.generalized_gamma_fitter.GeneralizedGammaFitter attribute)</td>
<td>150</td>
</tr>
<tr>
<td><code>cumulative_hazard_at_times()</code></td>
<td>(lifelines.fitters.piecewise_exponential_fitter.PiecewiseExponentialFitter attribute)</td>
<td>188</td>
</tr>
<tr>
<td><code>cumulative_hazard_at_times()</code></td>
<td>(lifelines.fitters.spline_fitter.SplineFitter attribute)</td>
<td>187</td>
</tr>
<tr>
<td><code>cumulative_hazard_at_times()</code></td>
<td>(lifelines.fitters.weibull_fitter.WeibullFitter attribute)</td>
<td>193</td>
</tr>
</tbody>
</table>
cumulative_hazard_(life-lines.fitters.log_normal_fitter.LogNormalFitter attribute), 167

cumulative_hazard_(life-lines.fitters.mixture_cure_fitter.MixtureCureFitter attribute), 172

cumulative_hazard_(life-lines.fitters.nelson_aalen_fitter.NelsonAalenFitter attribute), 178

cumulative_hazard_(life-lines.fitters.piecewise_exponential_fitter.PiecewiseExponentialFitter attribute), 181

cumulative_hazard_(life-lines.fitters.spline_fitter.SplineFitter attribute), 187

cumulative_hazard_(life-lines.fitters.weibull_fitter.WeibullFitter attribute), 193

cumulative_hazards_(life-lines.fitters.aalen_additive_fitter.AalenAdditiveFitter attribute), 199

cured_fraction_(life-lines.fitters.mixture_cure_fitter.MixtureCureFitter attribute), 172

datetimes_to_durations() (in module life-lines.utils), 267

density (life-lines.fitters.exponential_fitter.ExponentialFitter attribute), 145

density (life-lines.fitters.generalized_gamma_fitter.GeneralizedGammaFitter attribute), 150

density (life-lines.fitters.generalized_gamma_regression_fitter.GeneralizedGammaRegressionFitter attribute), 221

density (life-lines.fitters.log_logistic_fitter.LogLogisticFitter attribute), 161

density (life-lines.fitters.log_normal_fitter.LogNormalFitter attribute), 167

density (life-lines.fitters.piecewise_exponential_fitter.PiecewiseExponentialFitter attribute), 181

density (life-lines.fitters.spline_fitter.SplineFitter attribute), 187

density (life-lines.fitters.weibull_fitter.WeibullFitter attribute), 193

density_at_times() (life-lines.fitters.exponential_fitter.ExponentialFitter method), 146

density_at_times() (life-lines.fitters.generalized_gamma_fitter.GeneralizedGammaFitter method), 146

density_at_times() (life-lines.fitters.generalized_gamma_regression_fitter.GeneralizedGammaRegressionFitter method), 146

density_at_times() (life-lines.fitters.log_logistic_fitter.LogLogisticFitter method), 168

density_at_times() (life-lines.fitters.log_normal_fitter.LogNormalFitter method), 168

density_at_times() (life-lines.fitters.mixture_cure_fitter.MixtureCureFitter method), 174

density_at_times() (life-lines.fitters.nelson_aalen_fitter.NelsonAalenFitter method), 178

density_at_times() (life-lines.fitters.piecewise_exponential_fitter.PiecewiseExponentialFitter method), 182

density_at_times() (life-lines.fitters.spline_fitter.SplineFitter method), 188

density_at_times() (life-lines.fitters.weibull_fitter.WeibullFitter method), 195

divide() (life-lines.fitters.aalen_johansen_fitter.AalenJohansenFitter method), 139

divide() (life-lines.fitters.breslow_fleming_harrington_fitter.BreslowFlemingHarringtonFitter method), 142

divide() (life-lines.fitters.exp_regression.ExpRegressionFitter method), 147

divide() (life-lines.fitters.exponential_fitter.ExponentialFitter method), 146

divide() (life-lines.fitters.generalized_gamma_regression_fitter.GeneralizedGammaRegressionFitter method), 221

divide() (life-lines.fitters.log_logistic_fitter.LogLogisticFitter method), 162

divide() (life-lines.fitters.log_normal_fitter.LogNormalFitter method), 168

divide() (life-lines.fitters.piecewise_exponential_fitter.PiecewiseExponentialFitter method), 183

divide() (life-lines.fitters.spline_fitter.SplineFitter method), 189

divide() (life-lines.fitters.spline_regression.SplineRegressionFitter method), 191

divide() (life-lines.fitters.weibull_fitter.WeibullFitter method), 195

dv_  

Index 327
divide() (lifelines.fitters.exponential_fitter.ExponentialFitter method), 146
divide() (lifelines.fitters.generalized_gamma_fitter.GeneralizedGammaFitter method), 152
divide() (lifelines.fitters.kaplan_meier_fitter.KaplanMeierFitter method), 157
divide() (lifelines.fitters.log_logistic_fitter.LogLogisticFitter method), 163
divide() (lifelines.fitters.log_normal_fitter.LogNormalFitter method), 168
divide() (lifelines.fitters.lognormal_aft_fitter.LogNormalAFTFitter method), 174
divide() (lifelines.fitters.piecewise_exponential_fitter.PiecewiseExponentialFitter method), 183
divide() (lifelines.fitters.spline_fitter.SplineFitter method), 189
divide() (lifelines.fitters.weibull_fitter.WeibullFitter method), 195
durations (lifelines.fitters.aalen_additive_fitter.AalenAdditiveFitter attribute), 199
durations (lifelines.fitters.coxph_fitter.CoxPHFitter attribute), 203
durations (lifelines.fitters.exponential_fitter.ExponentialFitter attribute), 145
durations (lifelines.fitters.generalized_gamma_fitter.GeneralizedGammaFitter attribute), 151
durations (lifelines.fitters.generalized_gamma_regression_fitter.GeneralizedGammaRegressionFitter attribute), 221
durations (lifelines.fitters.kaplan_meier_fitter.KaplanMeierFitter attribute), 156
durations (lifelines.fitters.log_normal_aft_fitter.LogNormalAFTFitter attribute), 162
durations (lifelines.fitters.lognormal_aft_fitter.LogNormalAFTFitter attribute), 238
durations (lifelines.fitters.mixture_cure_fitter.MixtureCureFitter attribute), 173
durations (lifelines.fitters.mixture_cure_fitter.MixtureCureFitter attribute), 178
durations (lifelines.fitters.piecewise_exponential_fitter.PiecewiseExponentialFitter attribute), 182
durations (lifelines.fitters.piecewise_exponential_fitter.PiecewiseExponentialFitter attribute), 183
durations (lifelines.fitters.piecewise_exponential_fitter.PiecewiseExponentialFitter attribute), 199
durations (lifelines.fitters.pareto_fitter.ParetoFitter attribute), 203
durations (lifelines.fitters.spline_fitter.SplineFitter attribute), 238

event_observed (lifelines.fitters.exponential_fitter.ExponentialFitter attribute), 145
event_observed (lifelines.fitters.generalized_gamma_fitter.GeneralizedGammaFitter attribute), 151
event_observed (lifelines.fitters.log_logistic_fitter.LogLogisticFitter attribute), 162
event_observed (lifelines.fitters.log_normal_fitter.LogNormalFitter attribute), 167
event_observed (lifelines.fitters.lognormal_aft_fitter.LogNormalAFTFitter attribute), 151
event_observed (lifelines.fitters.lognormal_aft_fitter.LogNormalAFTFitter attribute), 156
event_observed (lifelines.fitters.lognormal_aft_fitter.LogNormalAFTFitter attribute), 182
event_observed (lifelines.fitters.lognormal_aft_fitter.LogNormalAFTFitter attribute), 229
event_observed (lifelines.fitters.log_normal_fitter.LogNormalFitter attribute), 162
event_observed (lifelines.fitters.log_normal_aft_fitter.LogNormalAFTFitter attribute), 238
event_table
(event_table
(event_table
(event_table
(event_table
(event_table
(event_table
(event_table
(event_table
(event_table

fit_interval_censoring()

ExponentialFitter (class in lifelines.fitters.exponential_fitter), 144

fit_interval_censoring()

fit_interval_censoring()

fit_interval_censoring()

fit_interval_censoring()

fit_interval_censoring()

fit_interval_censoring()

find_best_parametric_model() (in module lifelines.utils), 272

fit() (lifelines.fitters.aalen_additive_fitter.AalenAdditiveFitter method), 199

fit() (lifelines.fitters.aalen_johansen_fitter.AalenJohansenFitter method), 139

fit() (lifelines.fitters.breslow_fleming_harrington_fitter.BreslowFlemingHarringtonFitter method), 142

fit() (lifelines.fitters.cox_time_varying_fitter.CoxTimeVaryingFitter method), 217
Index
fit_right_censoring( )  (lifelines.fitters.spline_fitter.SplineFitter method), 191
fit_right_censoring( )  (lifelines.fitters.weibull_aft_fitter.WeibullAFTFitter method), 259
fit_right_censoring( )  (lifelines.fitters.weibull_fitter.WeibullFitter method), 197

G
GeneralizedGammaFitter  (class in lifelines.fitters.generalized_gamma_fitter), 149
GeneralizedGammaRegressionFitter  (class in lifelines.fitters.generalized_gamma_regression_fitter), 220
group_survival_table_from_events()  (in module lifelines.utils), 265

H
hazard_  (lifelines.fitters.exponential_fitter.ExponentialFitter attribute), 144
hazard_  (lifelines.fitters.generalized_gamma_fitter.GeneralizedGammaFitter attribute), 150
hazard_  (lifelines.fitters.generalized_gamma_regression_fitter.GeneralizedGammaRegressionFitter attribute), 221
hazard_  (lifelines.fitters.log_logistic_fitter.LogLogisticFitter attribute), 161
hazard_  (lifelines.fitters.log_normal_fitter.LogNormalFitter attribute), 167
hazard_  (lifelines.fitters.mixture_cure_fitter.MixtureCureFitter attribute), 173
hazard_  (lifelines.fitters.piecewise_exponential_fitter.PiecewiseExponentialFitter attribute), 181
hazard_  (lifelines.fitters.spline_fitter.SplineFitter attribute), 187
hazard_  (lifelines.fitters.weibull_fitter.WeibullFitter attribute), 193
hazard_at_times()  (lifelines.fitters.aalen_johansen_fitter.AalenJohansenFitter method), 140
hazard_at_times()  (lifelines.fitters.breslow_fleming_harrington_fitter.BreslowFlemingHarringtonFitter method), 142
hazard_at_times()  (lifelines.fitters.exponential_fitter.ExponentialFitter method), 148
hazard_at_times()  (lifelines.fitters.generalized_gamma_fitter.GeneralizedGammaFitter attribute), 154

K
KaplanMeierFitter  (class in lifelines.fitters.kaplan_meier_fitter), 155
k_fold_cross_validation()  (in module lifelines.utils), 269

L
Index 331
lambda__ (lifelines.fitters.generalized_gamma_fitter. GeneralizedGammaFitter method), 272
lifelines.datasets (module), 281
lifelines.datasets.load_canadian_senators() (in module lifelines.datasets), 281
lifelines.datasets.load_dd() (in module lifelines.datasets), 282
lifelines.datasets.load_dfcv() (in module lifelines.datasets), 282
lifelines.datasets.load_gbsq2() (in module lifelines.datasets), 283
lifelines.datasets.load_holly_molly_polly() (in module lifelines.datasets), 283
lifelines.datasets.load_kidney_transplant() (in module lifelines.datasets), 284
lifelines.datasets.load_larynx() (in module lifelines.datasets), 284
lifelines.datasets.load_lcd() (in module lifelines.datasets), 284
lifelines.datasets.load_g3() (in module lifelines.datasets), 283
lifelines.datasets.load_rossi() (in module lifelines.datasets), 283
lifelines.datasets.load_regression_dataset() (in module lifelines.datasets), 284
lifelines.datasets.load_recour() (in module lifelines.datasets), 287
lifelines.datasets.load_rh() (in module lifelines.datasets), 288
lifelines.datasets.load_saint_laurent() (in module lifelines.datasets), 288
lifelines.datasets.load_stanford_heart_transplants() (in module lifelines.datasets), 287
lifelines.datasets.loadstatic_test() (in module lifelines.datasets), 288
lifelines.datasets.load_static_test() (in module lifelines.datasets), 288
lifelines.datasets.load_waltons() (in module lifelines.datasets), 289
lifelines.datasets.load_yale() (in module lifelines.datasets), 289
lifelines.datasets.log_likelihood_ratio_test() (lifelines.fitters.cox_time_varying_fitter.CoxTimeVaryingFitter method), 218
lifelines.datasets.logLikelihoodRatioTest() (lifelines.fitters.coxph_fitter.CoxPHFitter method), 207
lifelines.datasets.loglikelihood_ratio_test() (lifelines.fitters.generalized_gamma_regression_fitter.GeneralizedGammaRegressionFitter method), 224
lifelines.datasets.log_likelihood_ratio_test() (lifelines.fitters.log_logistic_aft_fitter.LogLogisticAFTFitter method), 233
lifelines.datasets.loglikelihood_ratio_test() (lifelines.fitters.log_normal_aft_fitter.LogNormalAFTFitter method), 242
log_likelihood_ratio_test() (lifelines.fitters.piecewise_exponential_regression_fitter.PiecewiseExponentialRegressionFitter method), 249

log_likelihood_ratio_test() (lifelines.fitters.weibull_aft_fitter.WeibullAFTFitter method), 259

LogLogisticAFTFitter (class in lifelines.fitters.log_logistic_aft_fitter), 233

LogLogisticFitter (class in lifelines.fitters.log_logistic_fitter), 162, 165

LogNormalAFTFitter (class in lifelines.fitters.log_normal_aft_fitter), 242

LogNormalFitter (class in lifelines.fitters.log_normal_fitter), 167, 171

Logrank_test() (in module lifelines.statistics), 259

mean_survival_time_ (lifelines.fitters.generalized_gamma_regression_fitter.GeneralizedGammaRegressionFitter attribute), 221

mean_survival_time_ (lifelines.fitters.log_logistic_aft_fitter.LogLogisticAFTFitter attribute), 233

mean_survival_time_ (lifelines.fitters.log_normal_aft_fitter.LogNormalAFTFitter attribute), 173, 176

median_survival_time_ (lifelines.fitters.breslow_fleming_harrington_fitter.BreslowFlemingHarringtonFitter attribute), 140

median_survival_time_ (lifelines.fitters.cox_time_varying_fitter.CoxTimeVaryingFitter attribute), 215

median_survival_time_ (lifelines.fitters.exponential_fitter.ExponentialFitter attribute), 144, 148

median_survival_time_ (lifelines.fitters.generalized_gamma_fitter.GeneralizedGammaFitter attribute), 151, 154

median_survival_time_ (lifelines.fitters.generalized_gamma_regression_fitter.GeneralizedGammaRegressionFitter attribute), 224

median_survival_time_ (lifelines.fitters.kaplan_meier_fitter.KaplanMeierFitter attribute), 156, 158

median_survival_time_ (lifelines.fitters.nelson_aalen_fitter.NelsonAalenFitter attribute), 179

median_survival_time_ (lifelines.fitters.piecewise_exponential_regression_fitter.PiecewiseExponentialRegressionFitter attribute), 181, 185

median_survival_time_ (lifelines.fitters.piecewise_exponential_regression_fitter.PiecewiseExponentialFitter attribute), 249

median_survival_time_ (lifelines.fitters.spline_fitter.SplineFitter attribute), 187, 191

multivariate_logrank_test() (in module lifelines.statistics), 274

NelsonAalenFitter (class in lifelines.fitters.nelson_aalen_fitter), 177

mu_ (lifelines.fitters.log_normal_fitter.LogNormalFitter attribute), 167

NestCureFitter (class in lifelines.fitters.mixture_cure_fitter), 172

pairwise_logrank_test() (in module lifelines.statistics), 202

params_ (lifelines.fitters.cox_time_varying_fitter.CoxTimeVaryingFitter attribute), 215

params_ (lifelines.fitters.coxph_fitter.CoxPHFitter attribute), 202

Index 333
params_ (lifelines.fitters.log_logistic_aft_fitter.LogLogisticAFTFitter method), 218
params_ (lifelines.fitters.log_normal_aft_fitter.LogNormalAFTFitter method), 207
params_ (lifelines.fitters.weibull_aft_fitter.WeibullAFTFitter method), 254

percentile() (lifelines.fitters.aalen_johansen_fitter.AalenJohansenFitter method), 140
percentile() (lifelines.fitters.breslow_fleming_harrington_fitter.BreslowFlemingHarringtonFitter method), 142
percentile() (lifelines.fitters.exponential_fitter.ExponentialFitter method), 148
percentile() (lifelines.fitters.generalized_gamma_fitter.GeneralizedGammaFitter method), 154
percentile() (lifelines.fitters.kaplan_meier_fitter.KaplanMeierFitter method), 158
percentile() (lifelines.fitters.log_logistic_aft_fitter.LogLogisticAFTFitter method), 165
percentile() (lifelines.fitters.log_normal_aft_fitter.LogNormalAFTFitter method), 242
percentile() (lifelines.fitters.piecewise_exponential_fitter.PiecewiseExponentialFitter method), 179
percentile() (lifelines.fitters.piecewise_exponential_regression_fitter.PiecewiseExponentialRegressionFitter method), 185
percentile() (lifelines.fitters.spline_fitter.SplineFitter method), 191
percentile() (lifelines.fitters.weibull_aft_fitter.WeibullAFTFitter method), 197

PiecewiseExponentialFitter (class in lifelines.fitters.piecewise_exponential_fitter), 181
PiecewiseExponentialRegressionFitter (class in lifelines.fitters.piecewise_exponential_regression_fitter), 246

plot() (lifelines.fitters.aalen_additive_fitter.AalenAdditiveFitter method), 200
plot() (lifelines.fitters.aalen_johansen_fitter.AalenJohansenFitter method), 140
plot() (lifelines.fitters.breslow_fleming_harrington_fitter.BreslowFlemingHarringtonFitter method), 142
plot_cumulative_density() (lifelines.fitters.breslow_fleming_harrington_fitter.BreslowFlemingHarringtonFitter
(method), 143)

plot_cumulative_density() (lifelines.fitters.exponential_fitter.ExponentialFitter
(method), 148)

plot_cumulative_density() (lifelines.fitters.generalized_gamma_fitter.GeneralizedGammaFitter
(method), 155)

plot_cumulative_density() (lifelines.fitters.kaplan_meier_fitter.KaplanMeierFitter
(method), 159)

plot_cumulative_density() (lifelines.fitters.log_logistic_fitter.LogLogisticFitter
(method), 165)

plot_cumulative_density() (lifelines.fitters.log_normal_fitter.LogNormalFitter
(method), 171)

plot_cumulative_density() (lifelines.fitters.mixture_cure_fitter.MixtureCureFitter
(method), 176)

plot_cumulative_density() (lifelines.fitters.nelson_aalen_fitter.NelsonAalenFitter
(method), 180)

plot_cumulative_density() (lifelines.fitters.piecewise_exponential_fitter.PiecewiseExponentialFitter
(method), 185)

plot_cumulative_density() (lifelines.fitters.spline_fitter.SplineFitter method), 191

plot_cumulative_density() (lifelines.fitters.weibull_fitter.WeibullFitter
(method), 197)

plot_cumulative_hazard() (lifelines.fitters.aalen_johansen_fitter.AalenJohansenFitter
(method), 141)

plot_cumulative_hazard() (lifelines.fitters.breslow_fleming_harrington_fitter.BreslowFlemingHarringtonFitter
(method), 143)

plot_cumulative_hazard() (lifelines.fitters.exponential_fitter.ExponentialFitter
(method), 148)

plot_cumulative_hazard() (lifelines.fitters.generalized_gamma_fitter.GeneralizedGammaFitter
(method), 155)

plot_cumulative_hazard() (lifelines.fitters.kaplan_meier_fitter.KaplanMeierFitter
(method), 160)

plot_cumulative_hazard() (lifelines.fitters.log_logistic_fitter.LogLogisticFitter
(method), 165)

plot_cumulative_hazard() (lifelines.fitters.log_normal_fitter.LogNormalFitter
(method), 171)

plot_cumulative_hazard() (lifelines.fitters.mixture_cure_fitter.MixtureCureFitter
(method), 177)

plot_cumulative_hazard() (lifelines.fitters.nelson_aalen_fitter.NelsonAalenFitter
(method), 180)

plot_cumulative_hazard() (lifelines.fitters.piecewise_exponential_fitter.PiecewiseExponentialFitter
(method), 185)

plot_cumulative_hazard() (lifelines.fitters.spline_fitter.SplineFitter method), 191

plot_cumulative_hazard() (lifelines.fitters.weibull_fitter.WeibullFitter
(method), 197)

plot_density() (lifelines.fitters.aalen_johansen_fitter.AalenJohansenFitter
(method), 141)

plot_density() (lifelines.fitters.breslow_fleming_harrington_fitter.BreslowFlemingHarringtonFitter
(method), 143)

plot_density() (lifelines.fitters.exponential_fitter.ExponentialFitter
(method), 148)

plot_density() (lifelines.fitters.generalized_gamma_fitter.GeneralizedGammaFitter
(method), 155)

plot_density() (lifelines.fitters.kaplan_meier_fitter.KaplanMeierFitter
(method), 160)

plot_density() (lifelines.fitters.log_logistic_fitter.LogLogisticFitter
(method), 165)

plot_density() (lifelines.fitters.log_normal_fitter.LogNormalFitter
(method), 171)

plot_density() (lifelines.fitters.mixture_cure_fitter.MixtureCureFitter
(method), 177)

plot_density() (lifelines.fitters.nelson_aalen_fitter.NelsonAalenFitter
(method), 180)

plot_density() (lifelines.fitters.piecewise_exponential_fitter.PiecewiseExponentialFitter
(method), 185)

plot_density() (lifelines.fitters.spline_fitter.SplineFitter method), 191

plot_density() (lifelines.fitters.weibull_fitter.WeibullFitter
(method), 197)

plot_hazard() (lifelines.fitters.aalen_johansen_fitter.AalenJohansenFitter
(method), 141)

plot_hazard() (lifelines.fitters.breslow_fleming_harrington_fitter.BreslowFlemingHarringtonFitter
(method), 143)

plot_hazard() (lifelines.fitters.exponential_fitter.ExponentialFitter
(method), 148)

plot_hazard() (lifelines.fitters.generalized_gamma_fitter.GeneralizedGammaFitter
(method), 155)

plot_hazard() (lifelines.fitters.kaplan_meier_fitter.KaplanMeierFitter
(method), 160)

plot_hazard() (lifelines.fitters.log_logistic_fitter.LogLogisticFitter
(method), 165)

plot_hazard() (lifelines.fitters.log_normal_fitter.LogNormalFitter
(method), 171)

plot_hazard() (lifelines.fitters.mixture_cure_fitter.MixtureCureFitter
(method), 177)

plot_hazard() (lifelines.fitters.nelson_aalen_fitter.NelsonAalenFitter
(method), 180)

plot_hazard() (lifelines.fitters.piecewise_exponential_fitter.PiecewiseExponentialFitter
(method), 185)

plot_hazard() (lifelines.fitters.spline_fitter.SplineFitter method), 191

plot_hazard() (lifelines.fitters.weibull_fitter.WeibullFitter
(method), 197)
<table>
<thead>
<tr>
<th>Method</th>
<th>Class</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>plot_hazard()</td>
<td>life-plotting</td>
<td></td>
</tr>
<tr>
<td>plot_hazard()</td>
<td>lifelines.fitters.exponential_fitter.ExponentialFitter</td>
<td></td>
</tr>
<tr>
<td>plot_hazard()</td>
<td>lifelines.fitters.generalized_gamma_fitter.GeneralizedGammaFitter</td>
<td></td>
</tr>
<tr>
<td>plot_hazard()</td>
<td>lifelines.fitters.kaplan_meier_fitter.KaplanMeierFitter</td>
<td></td>
</tr>
<tr>
<td>plot_hazard()</td>
<td>lifelines.fitters.log_logistic_fitter.LogLogisticFitter</td>
<td></td>
</tr>
<tr>
<td>plot_hazard()</td>
<td>lifelines.fitters.log_normal_fitter.LogNormalFitter</td>
<td></td>
</tr>
<tr>
<td>plot_hazard()</td>
<td>lifelines.fitters.mixture_cure_fitter.MixtureCureFitter</td>
<td></td>
</tr>
<tr>
<td>plot_hazard()</td>
<td>lifelines.fitters.piecewise_exponential_fitter.PiecewiseExponentialFitter</td>
<td></td>
</tr>
<tr>
<td>plot_hazard()</td>
<td>lifelines.fitters.spline_fitter.SplineFitter</td>
<td></td>
</tr>
<tr>
<td>plot_hazard()</td>
<td>lifelines.fitters.weibull_fitter.WeibullFitter</td>
<td></td>
</tr>
<tr>
<td>predict()</td>
<td>lifelines.fitters.aalen_johansen_fitter.AalenJohansenFitter</td>
<td></td>
</tr>
<tr>
<td>predict()</td>
<td>lifelines.fitters.piecewise_exponential_fitter.PiecewiseExponentialFitter</td>
<td></td>
</tr>
<tr>
<td>predict()</td>
<td>lifelines.fitters.spline_fitter.SplineFitter</td>
<td></td>
</tr>
<tr>
<td>predict()</td>
<td>lifelines.fitters.weibull_fitter.WeibullFitter</td>
<td></td>
</tr>
<tr>
<td>predict_cumulative_hazard()</td>
<td>lifelines.fitters.aalen_johansen_fitter.AalenJohansenFitter</td>
<td></td>
</tr>
<tr>
<td>predict_cumulative_hazard()</td>
<td>lifelines.fitters.piecewise_exponential_fitter.PiecewiseExponentialFitter</td>
<td></td>
</tr>
<tr>
<td>predict_cumulative_hazard()</td>
<td>lifelines.fitters.spline_fitter.SplineFitter</td>
<td></td>
</tr>
<tr>
<td>predict_cumulative_hazard()</td>
<td>lifelines.fitters.weibull_fitter.WeibullFitter</td>
<td></td>
</tr>
<tr>
<td>predict_cumulative_hazard()</td>
<td>lifelines.fitters.aalen_johansen_fitter.AalenJohansenFitter</td>
<td></td>
</tr>
<tr>
<td>predict_cumulative_hazard()</td>
<td>lifelines.fitters.piecewise_exponential_fitter.PiecewiseExponentialFitter</td>
<td></td>
</tr>
<tr>
<td>predict_cumulative_hazard()</td>
<td>lifelines.fitters.spline_fitter.SplineFitter</td>
<td></td>
</tr>
<tr>
<td>predict_cumulative_hazard()</td>
<td>lifelines.fitters.weibull_fitter.WeibullFitter</td>
<td></td>
</tr>
<tr>
<td>predict_cumulative_hazard()</td>
<td>lifelines.fitters.aalen_johansen_fitter.AalenJohansenFitter</td>
<td></td>
</tr>
<tr>
<td>predict_cumulative_hazard()</td>
<td>lifelines.fitters.piecewise_exponential_fitter.PiecewiseExponentialFitter</td>
<td></td>
</tr>
<tr>
<td>predict_cumulative_hazard()</td>
<td>lifelines.fitters.spline_fitter.SplineFitter</td>
<td></td>
</tr>
<tr>
<td>predict_cumulative_hazard()</td>
<td>lifelines.fitters.weibull_fitter.WeibullFitter</td>
<td></td>
</tr>
<tr>
<td>predict_cumulative_hazard()</td>
<td>lifelines.fitters.aalen_johansen_fitter.AalenJohansenFitter</td>
<td></td>
</tr>
<tr>
<td>predict_cumulative_hazard()</td>
<td>lifelines.fitters.piecewise_exponential_fitter.PiecewiseExponentialFitter</td>
<td></td>
</tr>
<tr>
<td>predict_cumulative_hazard()</td>
<td>lifelines.fitters.spline_fitter.SplineFitter</td>
<td></td>
</tr>
<tr>
<td>predict_cumulative_hazard()</td>
<td>lifelines.fitters.weibull_fitter.WeibullFitter</td>
<td></td>
</tr>
<tr>
<td>predict_cumulative_hazard()</td>
<td>lifelines.fitters.aalen_johansen_fitter.AalenJohansenFitter</td>
<td></td>
</tr>
<tr>
<td>predict_cumulative_hazard()</td>
<td>lifelines.fitters.piecewise_exponential_fitter.PiecewiseExponentialFitter</td>
<td></td>
</tr>
<tr>
<td>predict_cumulative_hazard()</td>
<td>lifelines.fitters.spline_fitter.SplineFitter</td>
<td></td>
</tr>
<tr>
<td>predict_cumulative_hazard()</td>
<td>lifelines.fitters.weibull_fitter.WeibullFitter</td>
<td></td>
</tr>
<tr>
<td>predict_cumulative_hazard()</td>
<td>lifelines.fitters.aalen_johansen_fitter.AalenJohansenFitter</td>
<td></td>
</tr>
<tr>
<td>predict_cumulative_hazard()</td>
<td>lifelines.fitters.piecewise_exponential_fitter.PiecewiseExponentialFitter</td>
<td></td>
</tr>
<tr>
<td>predict_cumulative_hazard()</td>
<td>lifelines.fitters.spline_fitter.SplineFitter</td>
<td></td>
</tr>
<tr>
<td>predict_cumulative_hazard()</td>
<td>lifelines.fitters.weibull_fitter.WeibullFitter</td>
<td></td>
</tr>
<tr>
<td>predict_cumulative_hazard()</td>
<td>lifelines.fitters.aalen_johansen_fitter.AalenJohansenFitter</td>
<td></td>
</tr>
<tr>
<td>predict_cumulative_hazard()</td>
<td>lifelines.fitters.piecewise_exponential_fitter.PiecewiseExponentialFitter</td>
<td></td>
</tr>
</tbody>
</table>
predict_cumulative_hazard() (lifelines.fitters.generalized_gamma_regression_fitter.GeneralizedGammaRegressionFitter method), 225
predict_cumulative_hazard() (lifelines.fitters.log_logistic_aft_fitter.LogLogisticAFTFitter method), 234
predict_cumulative_hazard() (lifelines.fitters.log_normal_aft_fitter.LogNormalAFTFitter method), 243
predict_cumulative_hazard() (lifelines.fitters.piecewise_exponential_regression_fitter.PiecewiseExponentialRegressionFitter method), 250
predict_cumulative_hazard() (lifelines.fitters.weibull_aft_fitter.WeibullAFTFitter method), 260
predict_expectation() (lifelines.fitters.aalen_additive_fitter.AalenAdditiveFitter method), 201
predict_expectation() (lifelines.fitters.cox_time_varying_fitter.CoxTimeVaryingFitter method), 219
predict_expectation() (lifelines.fitters.coxph_fitter.CoxPHFitter method), 213
predict_expectation() (lifelines.fitters.generalized_gamma_regression_fitter.GeneralizedGammaRegressionFitter method), 226
predict_expectation() (lifelines.fitters.log_logistic_aft_fitter.LogLogisticAFTFitter method), 235
predict_expectation() (lifelines.fitters.log_normal_aft_fitter.LogNormalAFTFitter method), 244
predict_expectation() (lifelines.fitters.piecewise_exponential_regression_fitter.PiecewiseExponentialRegressionFitter method), 251
predict_expectation() (lifelines.fitters.weibull_aft_fitter.WeibullAFTFitter method), 261
predict_hazard() (lifelines.fitters.generalized_gamma_regression_fitter.GeneralizedGammaRegressionFitter method), 226
predict_hazard() (lifelines.fitters.log_logistic_aft_fitter.LogLogisticAFTFitter method), 235
predict_hazard() (lifelines.fitters.log_normal_aft_fitter.LogNormalAFTFitter method), 244
predict_hazard() (lifelines.fitters.piecewise_exponential_regression_fitter.PiecewiseExponentialRegressionFitter method), 251
predict_hazard() (lifelines.fitters.weibull_aft_fitter.WeibullAFTFitter method), 261
predict_log_partial_hazard() (lifelines.fitters.cox_time_varying_fitter.CoxTimeVaryingFitter method), 219
predict_log_partial_hazard() (lifelines.fitters.coxph_fitter.CoxPHFitter method), 213
predict_log_partial_hazard() (lifelines.fitters.generalized_gamma_regression_fitter.GeneralizedGammaRegressionFitter method), 226
predict_log_partial_hazard() (lifelines.fitters.log_logistic_aft_fitter.LogLogisticAFTFitter method), 235
predict_log_partial_hazard() (lifelines.fitters.log_normal_aft_fitter.LogNormalAFTFitter method), 244
predict_log_partial_hazard() (lifelines.fitters.piecewise_exponential_regression_fitter.PiecewiseExponentialRegressionFitter method), 251
predict_log_partial_hazard() (lifelines.fitters.weibull_aft_fitter.WeibullAFTFitter method), 261
predict_percentile()

predict_survival_function()

print_specific_style()

print_summary()

print_summary()

print_summary()

print_summary()

print_summary()

print_summary()

print_summary()

print_summary()

print_summary()

print_summary()
score() (lifelines.fitters.aalen_additive_fitter.AalenAdditiveFitter, 201)
score() (lifelines.fitters.cox_time_varying_fitter.CoxTimeVaryingFitter attribute, 216)
score() (lifelines.fitters.cox_time_varying_fitter.CoxTimeVaryingFitter method, 219)
score() (lifelines.fitters.coxph_fitter.CoxPHFitter subtract () (lifelines.fitters.aalen_johansen_fitter.AalenJohansenFitter method), 214)
score() (lifelines.fitters.coxph_fitter.CoxPHFitter subtract () (lifelines.fitters.cox_time_varying_fitter.CoxTimeVaryingFitter method), 214)
score() (lifelines.fitters.generalized_gamma_regression_fitter.GeneralizedGammaFitter subtract () (lifelines.fitters.logistic_fitter.LogisticFitter method), 227)
score() (lifelines.fitters.log_normal_fitter.LogNormalFitter subtract () (lifelines.fitters.exponential_fitter.ExponentialFitter method), 236)
score() (lifelines.fitters.log_normal_fitter.LogNormalFitter subtract () (lifelines.fitters.generalized_gamma_regression_fitter.GeneralizedGammaFitter method), 246)
score() (lifelines.fitters.piecewise_exponential_regression_fitter.PiecewiseExponentialRegressionFitter subtract () (lifelines.fitters.cox_time_varying_fitter.CoxTimeVaryingFitter method), 252)
score() (lifelines.fitters.weibull_aft_fitter.WeibullAFTFitter subtract () (lifelines.fitters.log_normal_fitter.LogNormalFitter method), 263)
score_ (lifelines.fitters.aalen_additive_fitter.AalenAdditiveFitter subtract () (lifelines.fitters.log_normal_fitter.LogNormalFitter method), 201)
score_ (lifelines.fitters.log_normal_fitter.LogNormalFitter subtract () (lifelines.fitters.piecewise_exponential_fitter.PiecewiseExponentialFitter method), 238)
sigma_ (lifelines.fitters.log_normal_fitter.LogNormalFitter subtract () (lifelines.fitters.piecewise_exponential_regression_fitter.PiecewiseExponentialRegressionFitter method), 167)
sigma_ (lifelines.fitters.piecewise_exponential_fitter.PiecewiseExponentialFitter subtract () (lifelines.fitters.piecewise_exponential_regression_fitter.PiecewiseExponentialRegressionFitter method), 180)
SplineFitter (class in lifelines.fitters.spline_fitter), 203
standard_errors_ (lifelines.fitters.cox_time_varying_fitter.CoxTimeVaryingFitter attribute, 216)
standard_errors_ (lifelines.fitters.coxph_fitter.CoxPHFitter attribute, 203)
standard_errors_ (lifelines.fitters.log_normal_fitter.LogNormalFitter attribute, 229)
standard_errors_ (lifelines.fitters.log_normal_fitter.LogNormalAFTFitter attribute, 238)
standard_errors_ (lifelines.fitters.weibull_aft_fitter.WeibullAFTFitter attribute, 254)
StatisticalResult (class in lifelines.statistics), 272
<table>
<thead>
<tr>
<th>Function/Method</th>
<th>Module/Class</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>summary (lifelines.fitters.spline_fitter.SplineFitter attribute)</td>
<td>192</td>
<td></td>
</tr>
<tr>
<td>summary (lifelines.fitters.weibull aft_fitter.WeibullAFTFitter attribute)</td>
<td>263</td>
<td></td>
</tr>
<tr>
<td>summary (lifelines.fitters.weibull_fitter.WeibullFitter attribute)</td>
<td>198</td>
<td></td>
</tr>
<tr>
<td>summary (lifelines.statistics.StatisticalResult attribute)</td>
<td>273</td>
<td></td>
</tr>
<tr>
<td>survival_difference_at_fixed_point_in_time_test()</td>
<td>(in module lifelines.statistics)</td>
<td>276</td>
</tr>
<tr>
<td>survival_events_from_table()</td>
<td>(in module lifelines.utils)</td>
<td>266</td>
</tr>
<tr>
<td>survival_function_</td>
<td>(lifelines.fitters.exponential_fitter.ExponentialFitter attribute)</td>
<td>144</td>
</tr>
<tr>
<td>survival_function_</td>
<td>(lifelines.fitters.generalized_gamma_fitter.GeneralizedGammaFitter attribute)</td>
<td>150</td>
</tr>
<tr>
<td>survival_function_</td>
<td>(lifelines.fitters.generalized_gamma_regression_fitter.GeneralizedGammaRegressionFitter attribute)</td>
<td>221</td>
</tr>
<tr>
<td>survival_function_</td>
<td>(lifelines.fitters.kaplan_meier_fitter.KaplanMeierFitter attribute)</td>
<td>156</td>
</tr>
<tr>
<td>survival_function_</td>
<td>(lifelines.fitters.log_logistic_fitter.LogLogisticFitter attribute)</td>
<td>161</td>
</tr>
<tr>
<td>survival_function_</td>
<td>(lifelines.fitters.log_normal_fitter.LogNormalFitter attribute)</td>
<td>167</td>
</tr>
<tr>
<td>survival_function_</td>
<td>(lifelines.fitters.mixture_cure_fitter.MixtureCureFitter method)</td>
<td>173</td>
</tr>
<tr>
<td>survival_function_</td>
<td>(lifelines.fitters.piecewise_exponential_fitter.PiecewiseExponentialFitter attribute)</td>
<td>181</td>
</tr>
<tr>
<td>survival_function_</td>
<td>(lifelines.fitters.spline_fitter.SplineFitter attribute)</td>
<td>187</td>
</tr>
<tr>
<td>survival_function_</td>
<td>(lifelines.fitters.weibull_fitter.WeibullFitter attribute)</td>
<td>193</td>
</tr>
<tr>
<td>survival_function_at_times()</td>
<td>(lifelines.fitters.aalen_johansen_fitter.AalenJohansenFitter method)</td>
<td>141</td>
</tr>
<tr>
<td>survival_function_at_times()</td>
<td>(lifelines.fitters.breslow_fleming_harrington_fitter.BreslowFlemingHarringtonFitter method)</td>
<td>143</td>
</tr>
<tr>
<td>survival_function_at_times()</td>
<td>(lifelines.fitters.exponential_fitter.ExponentialFitter method)</td>
<td>149</td>
</tr>
<tr>
<td>survival_function_at_times()</td>
<td>(lifelines.fitters.generalized_gamma_fitter.GeneralizedGammaFitter method)</td>
<td>155</td>
</tr>
<tr>
<td>survival_function_at_times()</td>
<td>(lifelines.fitters.kaplan_meier_fitter.KaplanMeierFitter method)</td>
<td>160</td>
</tr>
<tr>
<td>survival_function_at_times()</td>
<td>(lifelines.fitters.log_logistic_fitter.LogLogisticFitter method)</td>
<td>166</td>
</tr>
<tr>
<td>survival_function_at_times()</td>
<td>(lifelines.fitters.log_normal_fitter.LogNormalFitter method)</td>
<td>171</td>
</tr>
<tr>
<td>survival_function_at_times()</td>
<td>(lifelines.fitters.mixture_cure_fitter.MixtureCureFitter method)</td>
<td>177</td>
</tr>
<tr>
<td>survival_function_at_times()</td>
<td>(lifelines.fitters.nelson_aalen_fitter.NelsonAalenFitter method)</td>
<td>180</td>
</tr>
<tr>
<td>survival_function_at_times()</td>
<td>(lifelines.fitters.piecewise_exponential_fitter.PiecewiseExponentialFitter method)</td>
<td>186</td>
</tr>
<tr>
<td>survival_function_at_times()</td>
<td>(lifelines.fitters.spline_fitter.SplineFitter method)</td>
<td>192</td>
</tr>
<tr>
<td>survival_function_at_times()</td>
<td>(lifelines.fitters.weibull_fitter.WeibullFitter method)</td>
<td>198</td>
</tr>
<tr>
<td>survival_table_from_events()</td>
<td>(in module lifelines.utils)</td>
<td>264</td>
</tr>
<tr>
<td>to_ascii()</td>
<td>(lifelines.statistics.StatisticalResult method)</td>
<td>273</td>
</tr>
<tr>
<td>to_episodic_format()</td>
<td>(in module lifelines.utils)</td>
<td>270</td>
</tr>
</tbody>
</table>

T

<table>
<thead>
<tr>
<th>Function/Method</th>
<th>Module/Class</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>timeline (lifelines.fitters.exponential_fitter.ExponentialFitter attribute)</td>
<td>145</td>
<td></td>
</tr>
<tr>
<td>timeline (lifelines.fitters.generalized_gamma_fitter.GeneralizedGammaFitter attribute)</td>
<td>151</td>
<td></td>
</tr>
<tr>
<td>timeline (lifelines.fitters.generalized_gamma_regression_fitter.GeneralizedGammaRegressionFitter attribute)</td>
<td>221</td>
<td></td>
</tr>
<tr>
<td>timeline (lifelines.fitters.kaplan_meier_fitter.KaplanMeierFitter attribute)</td>
<td>156</td>
<td></td>
</tr>
<tr>
<td>timeline (lifelines.fitters.log_logistic_fitter.LogLogisticFitter attribute)</td>
<td>162</td>
<td></td>
</tr>
<tr>
<td>timeline (lifelines.fitters.log_normal_fitter.LogNormalFitter attribute)</td>
<td>167</td>
<td></td>
</tr>
<tr>
<td>timeline (lifelines.fitters.mixture_cure_fitter.MixtureCureFitter method)</td>
<td>173</td>
<td></td>
</tr>
<tr>
<td>timeline (lifelines.fitters.piecewise_exponential_fitter.PiecewiseExponentialFitter attribute)</td>
<td>182</td>
<td></td>
</tr>
<tr>
<td>timeline (lifelines.fitters.spline_fitter.SplineFitter attribute)</td>
<td>188</td>
<td></td>
</tr>
<tr>
<td>timeline (lifelines.fitters.weibull_fitter.WeibullFitter attribute)</td>
<td>194</td>
<td></td>
</tr>
<tr>
<td>to_ascii()</td>
<td>(lifelines.statistics.StatisticalResult method)</td>
<td>273</td>
</tr>
<tr>
<td>to_episodic_format()</td>
<td>(in module lifelines.utils)</td>
<td>270</td>
</tr>
</tbody>
</table>
to_html() (lifelines.statistics.StatisticalResult method), 273

to_latex() (lifelines.statistics.StatisticalResult method), 273

to_long_format() (in module lifelines.utils), 269

V

variance_matrix_(lifelines.fitters.cox_time_varying_fitter.CoxTimeVaryingFitter attribute), 216

variance_matrix_(lifelines.fitters.coxph_fitter.CoxPHFitter attribute), 203

variance_matrix_(lifelines.fitters.exponential_fitter.ExponentialFitter attribute), 144

variance_matrix_(lifelines.fitters.generalized_gamma_fitter.GeneralizedGammaFitter attribute), 151

variance_matrix_(lifelines.fitters.generalized_gamma_regression_fitter.GeneralizedGammaRegressionFitter attribute), 221

variance_matrix_(lifelines.fitters.log_logistic_aft_fitter.LogLogisticAFTFitter attribute), 229

variance_matrix_(lifelines.fitters.log_logistic_fitter.LogLogisticFitter attribute), 161

variance_matrix_(lifelines.fitters.log_normal_aft_fitter.LogNormalAFTFitter attribute), 238

variance_matrix_(lifelines.fitters.log_normal_fitter.LogNormalFitter attribute), 193

variance_matrix_(lifelines.fitters.mixture_cure_fitter.MixtureCureFitter attribute), 173

variance_matrix_(lifelines.fitters.piecewise_exponential_fitter.PiecewiseExponentialFitter attribute), 181

variance_matrix_(lifelines.fitters.spline_fitter.SplineFitter attribute), 187

variance_matrix_(lifelines.fitters.weibull_aft_fitter.WeibullAFTFitter attribute), 254

variance_matrix_(lifelines.fitters.weibull_fitter.WeibullFitter attribute), 193

W

WeibullAFTFitter (class in lifelines.fitters.weibull_fitter), 253