lifelines Documentation

Release 0.8.0.1

Cam Davidson-Pilon

May 23, 2016

Contents

lifelines is a 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 (*designed for humans*)
- only does survival analysis (No unnecessary features or second-class implementations)

CHAPTER 1

Contents:

1.1 Quickstart

1.1.1 Installation

Install via `pip`:

```
pip install lifelines
```

1.1.2 Kaplan-Meier and Nelson-Aalen

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']
```

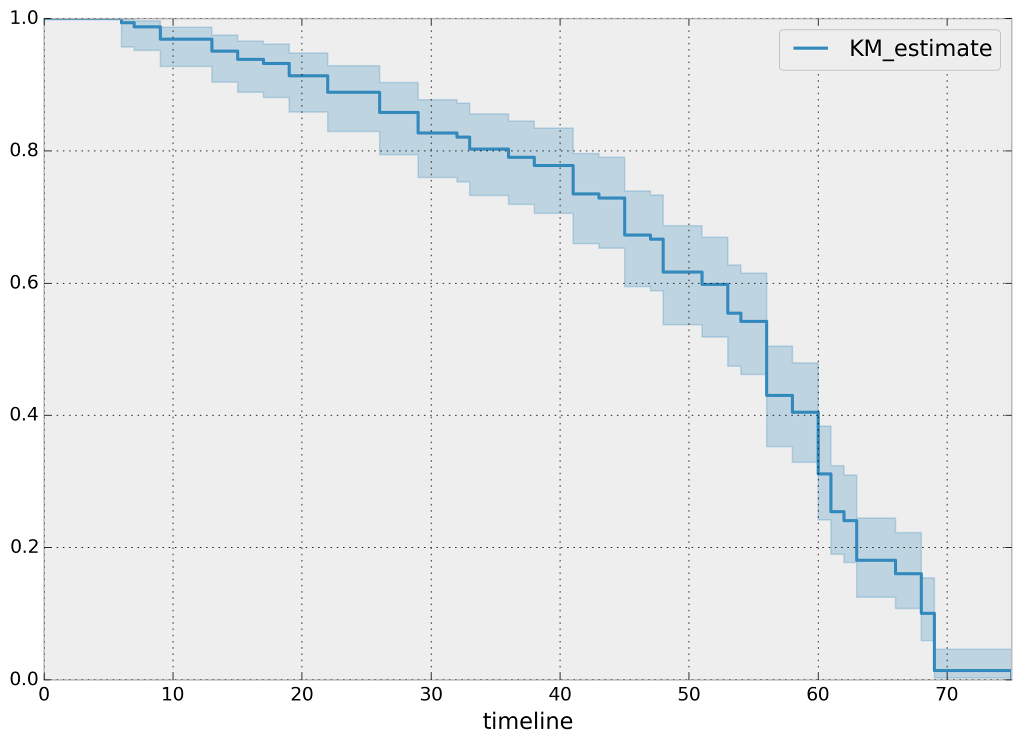
`T` is an array of durations, `E` is a either boolean or binary array representing whether the “death” was observed (alternatively an individual can be censored).

Note: By default, *lifelines* assumes all “deaths” are observed.

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

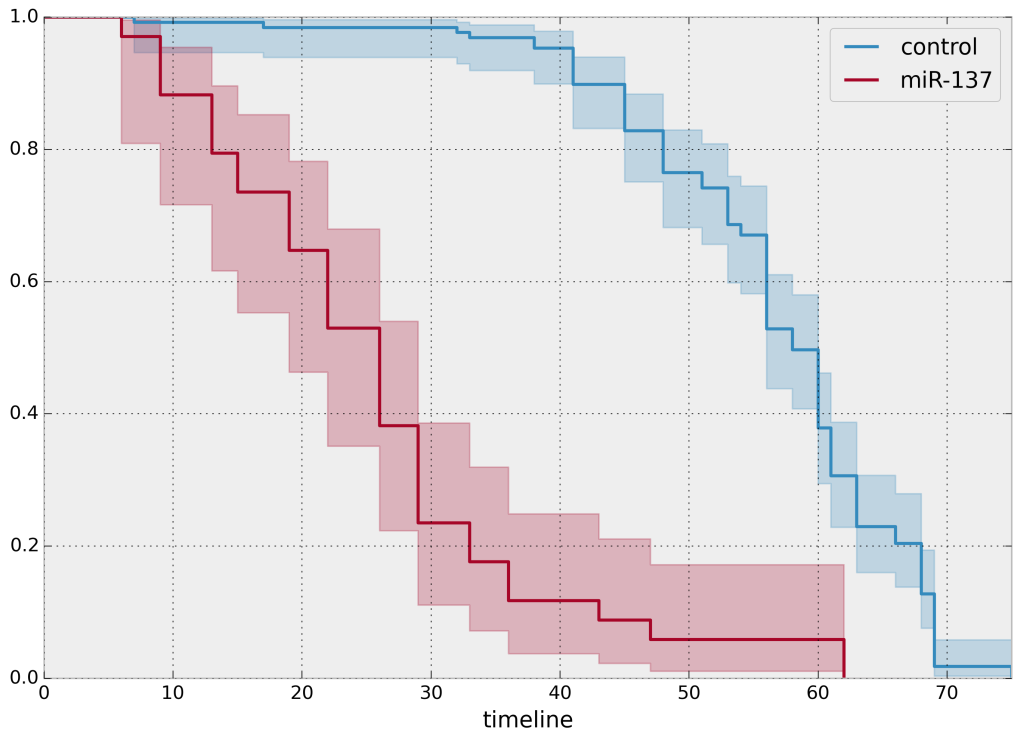
After calling the `fit` method, we have access to new properties like `survival_function_` and methods like `plot()`. The latter is a wrapper around Pandas internal plotting library.

```
kmf.survival_function_
kmf.median_
kmf.plot()
```



Multiple groups

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

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

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



Similar functionality exists for the `NelsonAalenFitter`:

```
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.

1.1.3 Getting Data in The Right Format

Often you'll have data that looks like:

start_time, *end_time*

Lifelines has some utility functions to transform this dataset into durations and censorships:

```
from lifelines.utils import datetimes_to_durations

# start_times is a vector of datetime objects
# end_times is a vector of (possibly missing) datetime objects.
T, C = datetimes_to_durations(start_times, end_times, freq='h')
```

Alternatively, perhaps you are interested in viewing the survival table given some durations and censorship vectors.

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

1.1.4 Survival Regression

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

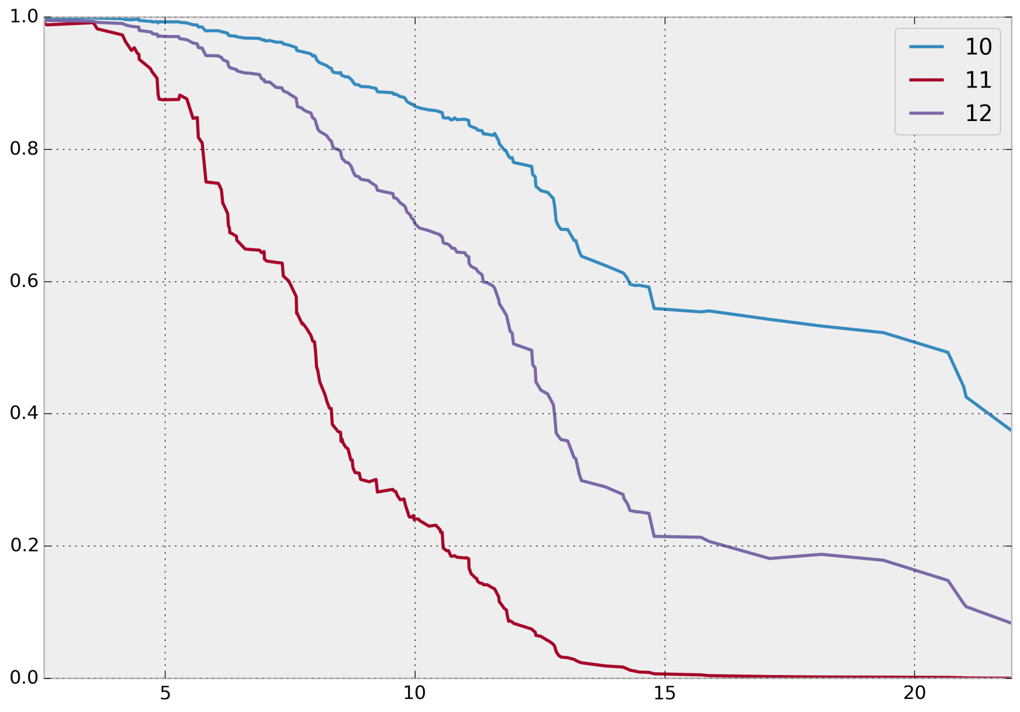
```
from lifelines.datasets import load_regression_dataset
regression_dataset = load_regression_dataset()

regression_dataset.head()
```

The input of the `fit` method's API on `AalenAdditiveFitter` is different than above. All the data, including durations, censorships and covariates must be contained in **a Pandas DataFrame** (yes, it must be a DataFrame). The duration column and event occured column must be specified in the call to `fit`.

```
from lifelines import AalenAdditiveFitter, CoxPHFitter

# Using Cox Proportional Hazards model
cf = CoxPHFitter()
cf.fit(regression_dataset, 'T', event_col='E')
cf.print_summary()

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

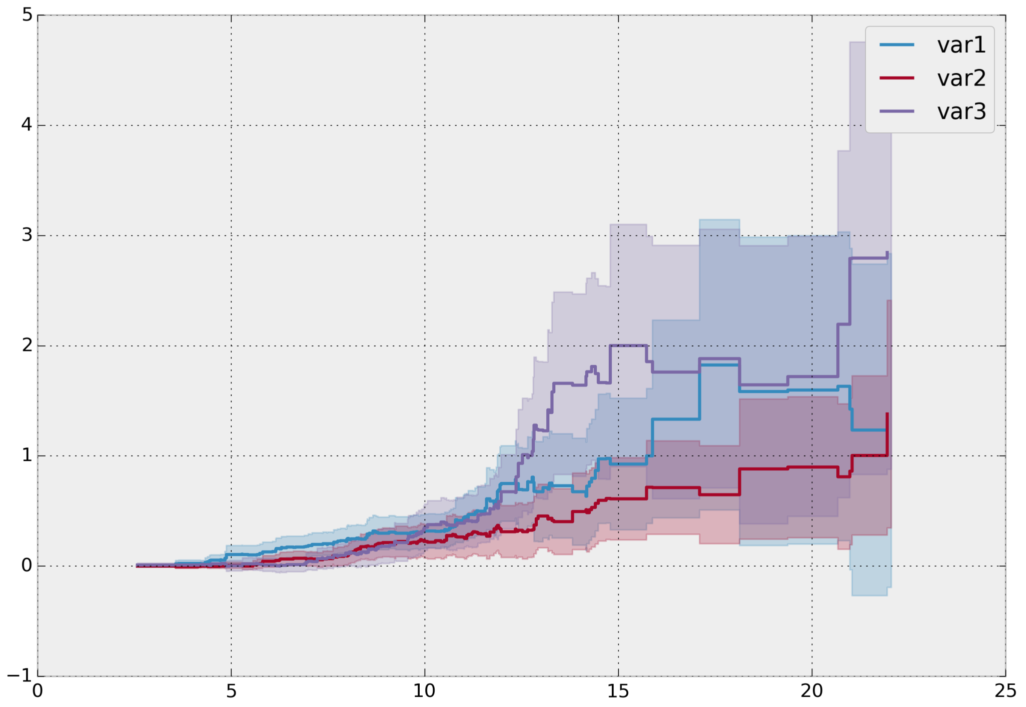
After fitting, you'll have access to properties like `cumulative_hazards_` and methods like `plot`, `predict_cumulative_hazards`, and `predict_survival_function`. The latter two methods require an additional argument of individual covariates:

```
x = regression_dataset[regression_dataset.columns - ['E','T']]
aaf.predict_survival_function(x.ix[10:12]).plot() #get the unique survival functions of the first two
```



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 sticken by a disease (in the medical professional’s case). Traditionally, sort of a morbid subject.

The analysis can be further applied to not just traditional *births and deaths*, but any duration. Medical professional 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 Censorship

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

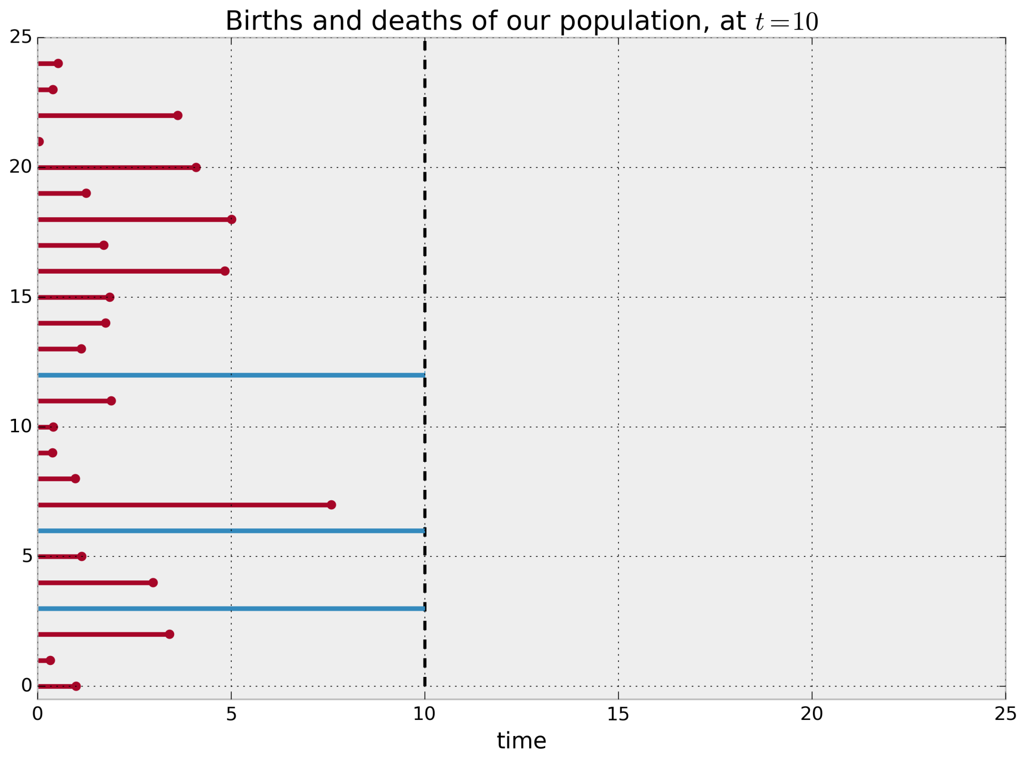
The individuals in a population who have not been subject to the death event are labeled as *right-censored*, i.e. we did not (or can not) view the rest of their life history due to some external circumstances. All the information we have on these individuals are their current lifetime durations (which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

Note: There is also left-censorship, where an individuals birth event is not seen.

A common mistake data analysts make is choosing to ignore the right-censored individuals. We shall 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 might not know this distinction before hand. At $t = 10$, we wish to investigate the average lifespan. Below is an example of such a situation.

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

N = 25
current_time = 10
actual_lifetimes = np.array([[exponential(12), exponential(2)][uniform()<0.5] for i in range(N)])
observed_lifetimes = np.minimum(actual_lifetimes,current_time)
observed= actual_lifetimes < current_time

plt.xlim(0,25)
plt.vlines(10,0,30,lw=2, linestyles="--")
plt.xlabel('time')
plt.title('Births and deaths of our population, at $t=10$')
plot_lifetimes(observed_lifetimes, event_observed=observed)
print "Observed lifetimes at time %d:\n"%(current_time), observed_lifetimes
```

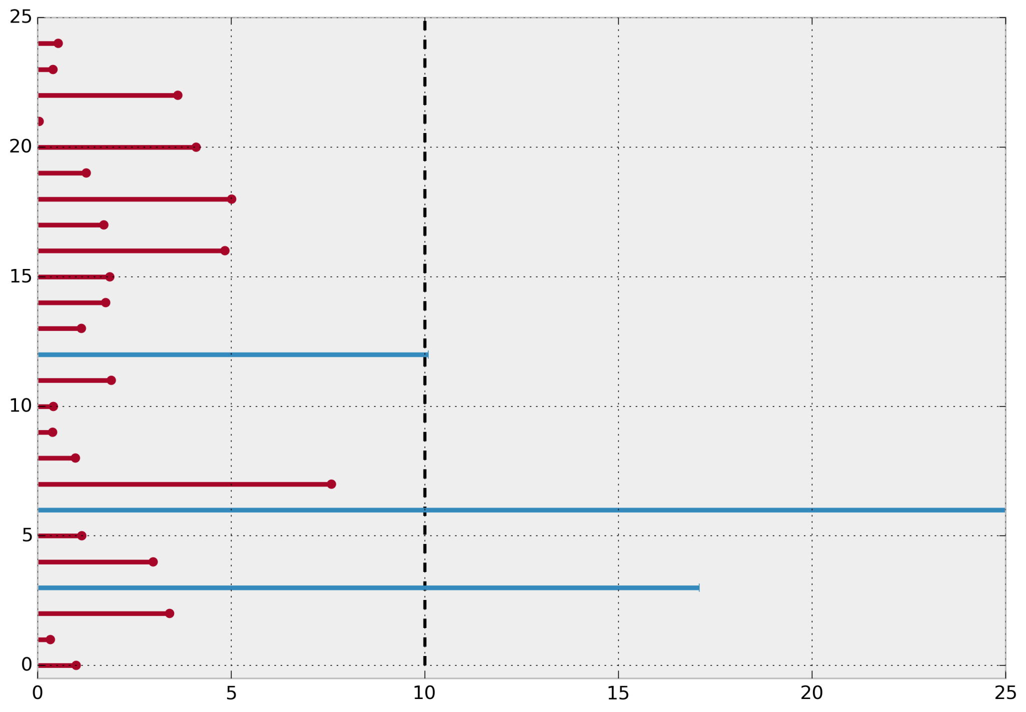


```
Observed lifetimes at time 10:
[ 10.     1.1    8.    10.     3.43   0.63   6.28   1.03   2.37   6.17  10.
   0.21   2.71   1.25  10.     3.4    0.62   1.94   0.22   7.43   6.16  10.
   9.41  10.     10.  ]
```

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 serverly underestimating the true average lifespan.

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

```
plt.xlim(0,25)
plt.vlines(10,0,30,lw=2,linestyles="--")
plot_lifetimes(actual_lifetimes, event_observed=observed)
```



Survival analysis was originally developed to solve this type of problem, that is, to deal with estimation when our data is right-censored. Even in the case where all events have been observed, i.e. no censorship, survival analysis is still a very useful to understand durations.

The observations need not always start at zero, either. This was done only for understanding in the above example. Consider the example of 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 the necessarily the start and end time.)

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

1.2.3 Survival function

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

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

In human language: the survival function defines the probability the death event has not occured yet at time t, or equivalently, the probability of surviving until atleast time t. Note the following properties of the survival function:

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

1.2.4 Hazard curve

We are also interested in the probability of dying in the next instant, given we haven't expired yet. Mathematically, that is:

$$\lim_{\delta t \to 0} \; Pr(t \le T \le t + \delta t | T > t)$$

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

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

It can be shown with quite elementary probability that this is equal to:

$$\lambda(t) = \frac{-S'(t)}{S(t)}$$

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

$$S(t) = \exp\left(-\int_0^t \lambda(z) dz\right)$$

What I love about the above equation is that it defines **all** survival functions, and because the hazard function is arbitrary (i.e. there is no parametric form), the entire function is non-parametric (this allows for very flexible curves). Notice that we can now speak either about the survival function, $S(t)$, or the hazard function, $\lambda(t)$, and we can convert back and forth quite easily. It also gives us another, albeit less useful, expression for T: Upon differentiation and some algebra, we recover:

$$f_T(t) = \lambda(t) \exp\left(-\int_0^t \lambda(z) dz\right)$$

Of course, we do not observe the true survival curve of a population. We must use the observed data to estimate it. We also want to continue to be non-parametric, that is not assume anything about how the survival curve looks. The *best* method to recreate the survival function non-parametrically from the data is known as the Kaplan-Meier estimate, which brings us to estimation using lifelines.

1.3 Introduction to using lifelines

In the previous section, we introduced how survival analysis is used, needed, and the mathematical objects that it relies on. In this article, we will work with real data and the *lifelines* library to estimate these mathematical objects.

1.3.1 Estimating the Survival function using Kaplan-Meier

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

For example, the Bush regime began in 2000 and officially ended in 2008 upon his retirement, thus this regime's lifespan was 8 years and the "death" event was 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 infact completely flips the idea upside down by using deaths as the censorship event. This is also an example where the current time is not the only cause of censorship – there are alternative events (eg: death in office) that can censor.)

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

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

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

Let's bring in our dataset.

```
import pandas as pd
import lifelines

data = lifelines.datasets.load_dd()
```

```
data.head()
#the boolean columns `observed` refers to whether the death (leaving office)
#was observed or not.
```

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

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

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, another popular Python library scikit-learn's fit/predict API)

```
KaplanMeierFitter.fit(event_times, event_observed=None,
                      timeline=None, label='KM-estimate',
                      alpha=None)
Parameters:
  event_times: an array, or pd.Series, of length n of times that
         the death event occured at
  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
  timeline: set the index of the survival curve to this postively increasing array.
  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.

Returns:
  self, with new properties like 'survival_function_'.
```

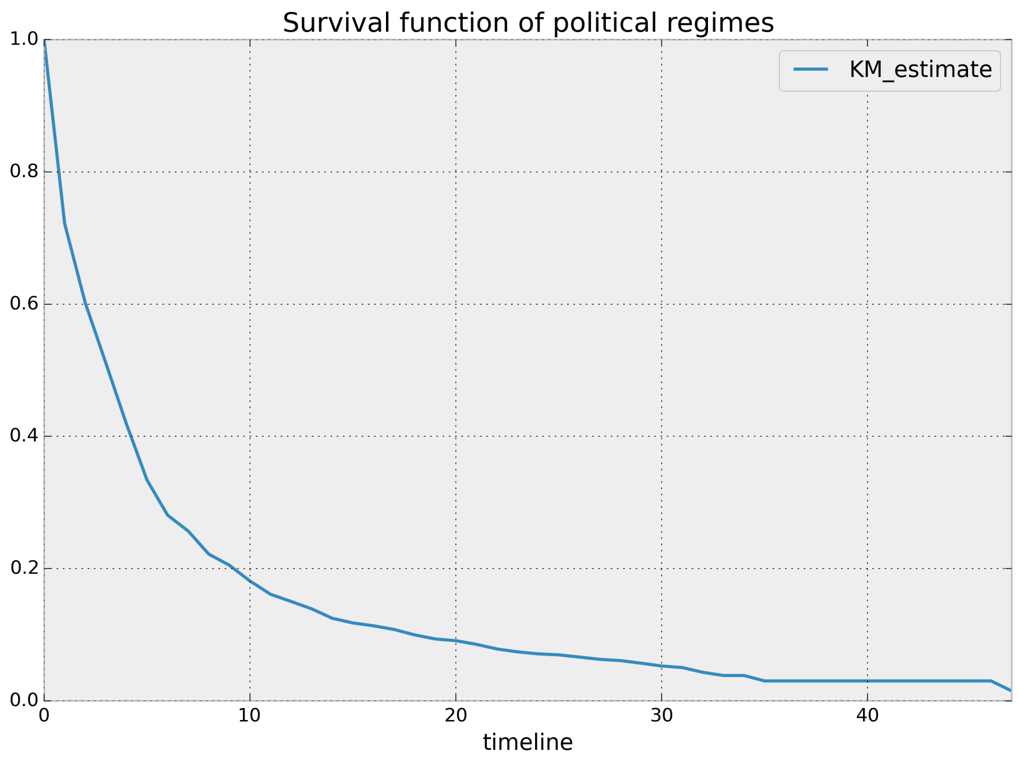
Below we fit our data to the fitter:

```
T = data["duration"]
C = data["observed"]

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

```
<lifelines.KaplanMeierFitter: fitted with 1808 observations, 340 censored>
```

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

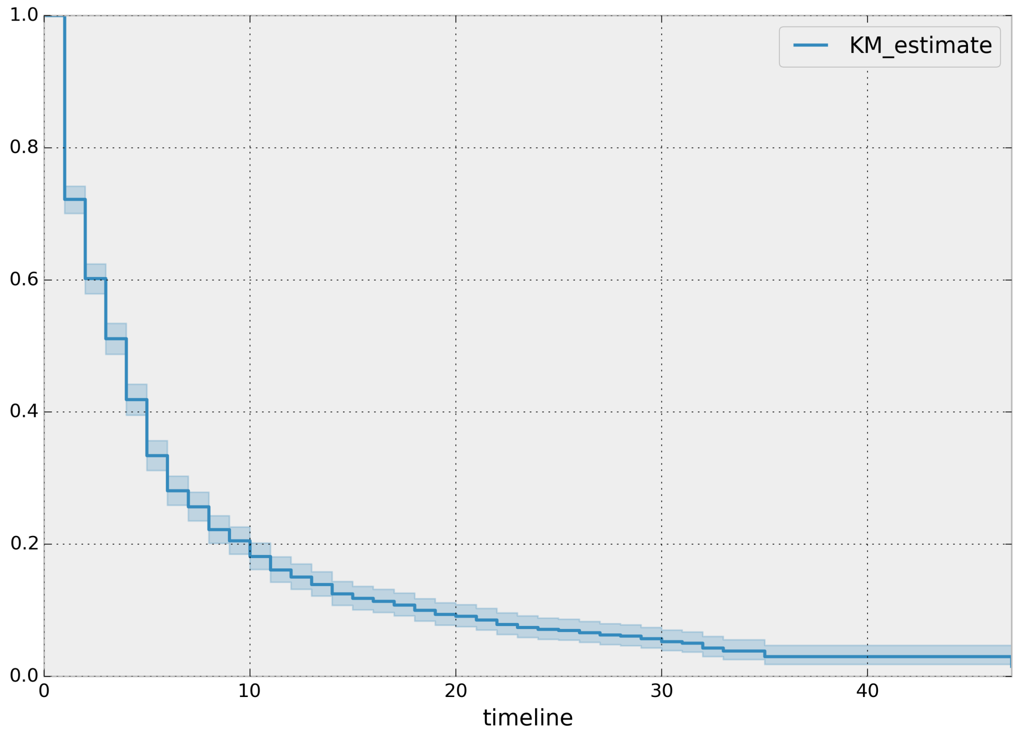
```
kmf.survival_function_.plot()
plt.title('Survival function of political regimes');
```



How do we interpret this? The y-axis represents the probability a leader is still around after t years, where t years is on the x-axis. We see that very few leaders make it past 20 years in office. Of course, like all good stats, we need to report how uncertain we are about these point estimates, i.e. we need confidence intervals. They are computed on the call to `fit`, and are located under the `confidence_interval_` property.

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

```
kmf.plot()
```



Note: Don't like the shaded area for confidence intervals? See below for examples on how to change this.

The median time in office, which defines the point in time where on average 1/2 of the population has expired, is a property:

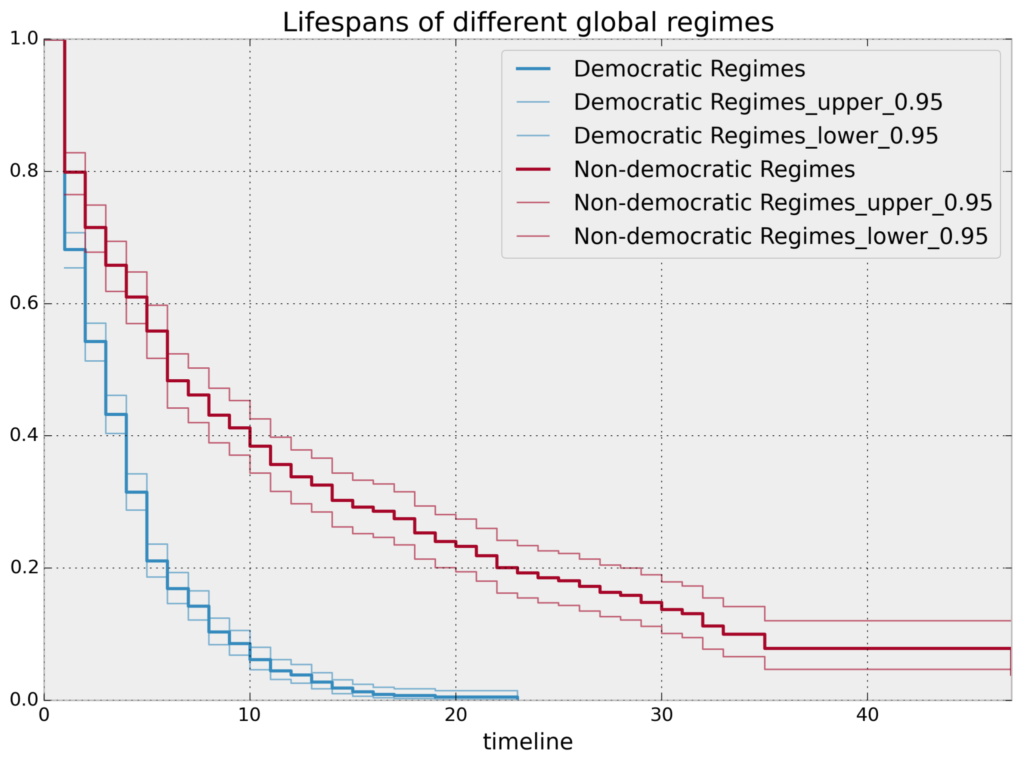
```
kmf.median_

#    4
#
```

Interesting that it is only 3 years. That means, around the world, when a leader is elected there is a 50% chance he or she will be gone in 3 years!

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:

```
ax = plt.subplot(111)

dem = (data["democracy"] == "Democracy")
kmf.fit(T[dem], event_observed=C[dem], label="Democratic Regimes")
kmf.plot(ax=ax, ci_force_lines=True)
kmf.fit(T[~dem], event_observed=C[~dem], label="Non-democratic Regimes")
kmf.plot(ax=ax, ci_force_lines=True)
```

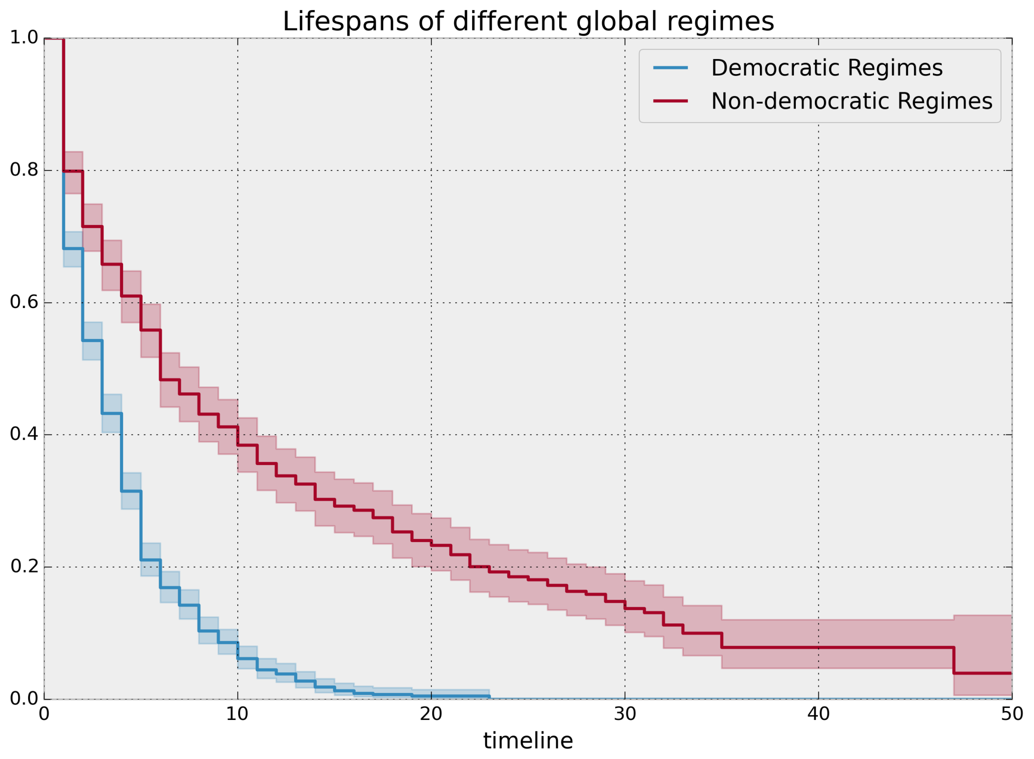
```
plt.ylim(0,1);
plt.title("Lifespans of different global regimes");
```



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 probabilties of survival at those points:

```
ax = subplot(111)

t = np.linspace(0,50,51)
kmf.fit(T[dem], event_observed=C[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=C[~dem], timeline=t, label="Non-democratic Regimes")
ax = kmf.plot(ax=ax)
print "Median survival time of non-democratic:", kmf.median_

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

```
Median survival time of democratic: Democratic Regimes    3
dtype: float64
Median survival time of non-democratic: Non-democratic Regimes    6
dtype: float64
```



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 8 year limit). The median of a non-democractic is only about twice as large as a democratic regime, but the difference is really 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 10 years, and then have a very short lifetime past that.

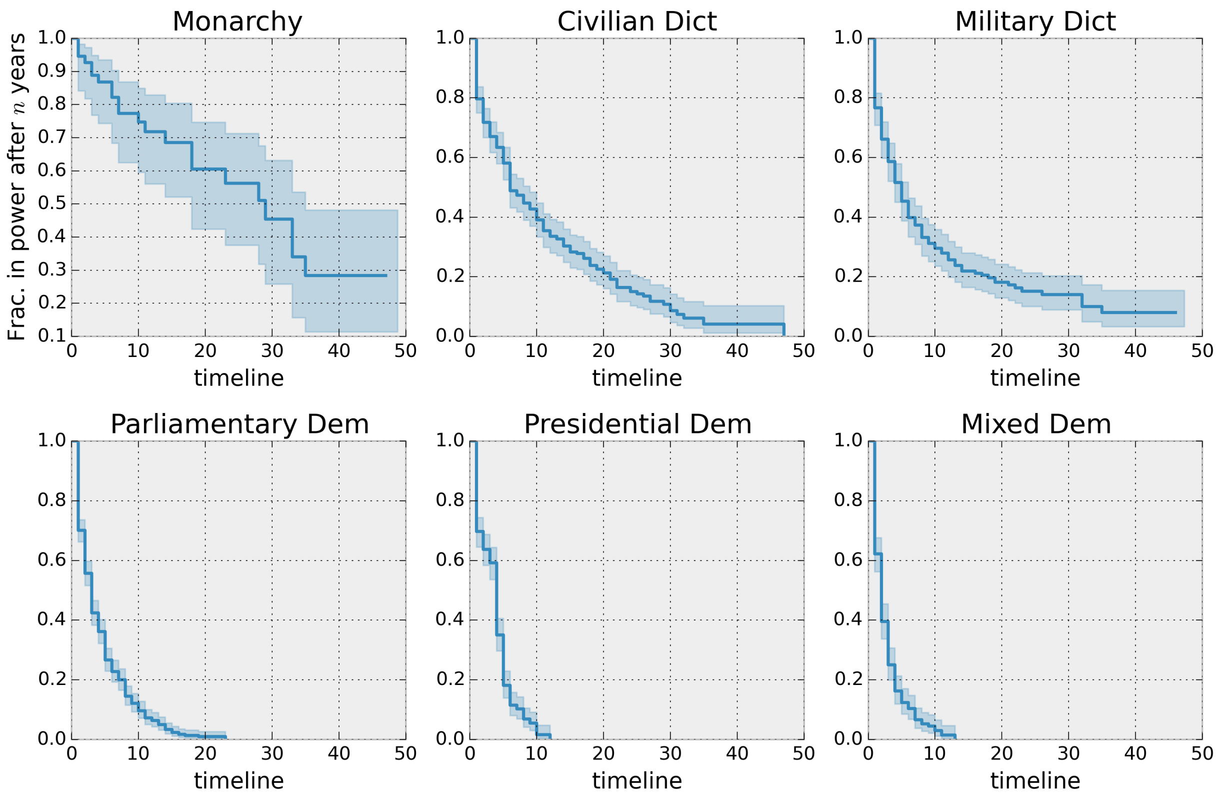
Here the difference between survival functions is very obvious, and performing a statistical test seems pendantic. If the curves are more similar, or we possess less data, we may be interested in performing a statistical test. In this case, *lifelines* contains routines in `lifelines.statistics` to compare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

```
from lifelines.statistics import logrank_test

results = logrank_test(T[dem], T[~dem], C[dem], C[~dem], alpha=.99 )

results.print_summary()
```

```
Results
   df: 1
   alpha: 0.99
   t 0: -1
   test: logrank
   null distribution: chi squared

   __ p-value ___|__ test statistic __|____ test results ____|__ significant __
         0.00000 |              208.306 |       Reject Null      |      True
```

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

```
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], C[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 `C` respectively. For example:

```
T = [0,3,3,2,1,2]
C = [1,1,0,0,1,1]
kmf.fit(T, event_observed=C )
```

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` sublibrary. 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):

```
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,C = datetimes_to_durations(start_date, end_date, fill_date='2013-10-15')
print 'T (durations): ', T
print 'C (event_observed): ',C
```

```
T (durations):  [ 3.  1.  5.]
C (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 curve is a great way to summarize and visualize the lifetime data, however it is not the only way. If we are curious about the hazard function $\lambda(t)$ of a population, we unfortunately cannot transform the Kaplan Meier estimate – statistics doesn't work quite that well. Fortunately, there is a proper estimator of the *cumulative* hazard function:

$$\Lambda(t) = \int_0^t \lambda(z) \; dz$$

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

$$\hat{\Lambda}(t) = \sum_{t_i \le t} \frac{d_i}{n_i}$$

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

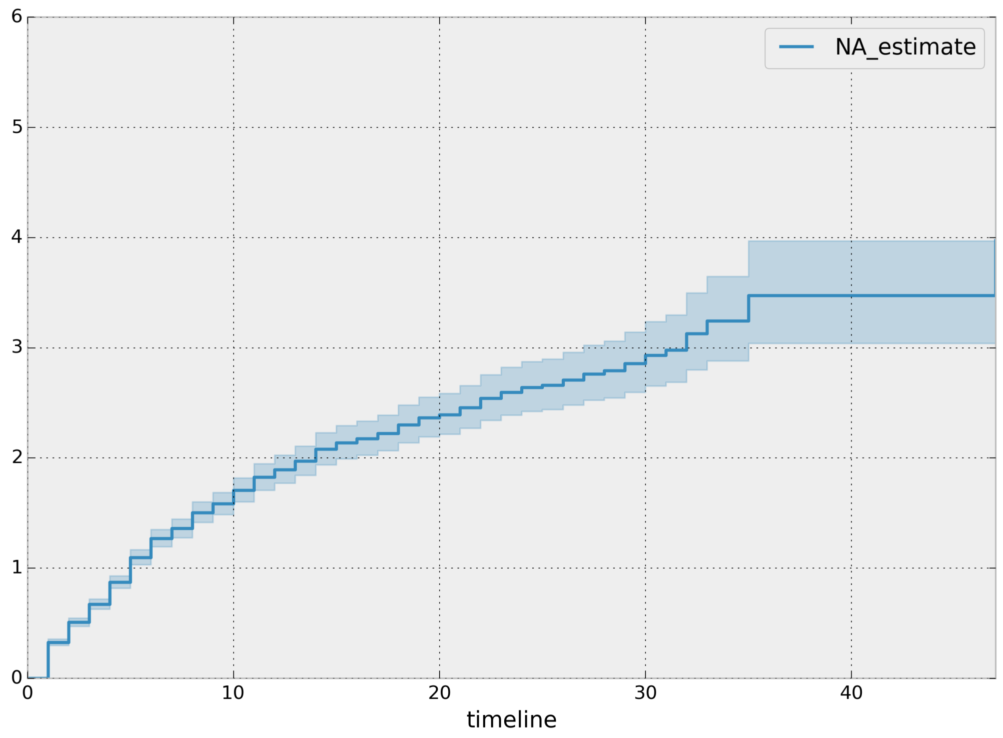
In *lifelines*, this estimator is available as the `NelsonAalenFitter`. Let's use the regime dataset from above:

```
T = data["duration"]
C = data["observed"]

from lifelines import NelsonAalenFitter
naf = NelsonAalenFitter()

naf.fit(T,event_observed=C)
```

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

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

```
   NA-estimate
0     0.000000
1     0.325912
2     0.507356
3     0.671251
4     0.869867

[5 rows x 1 columns]
```

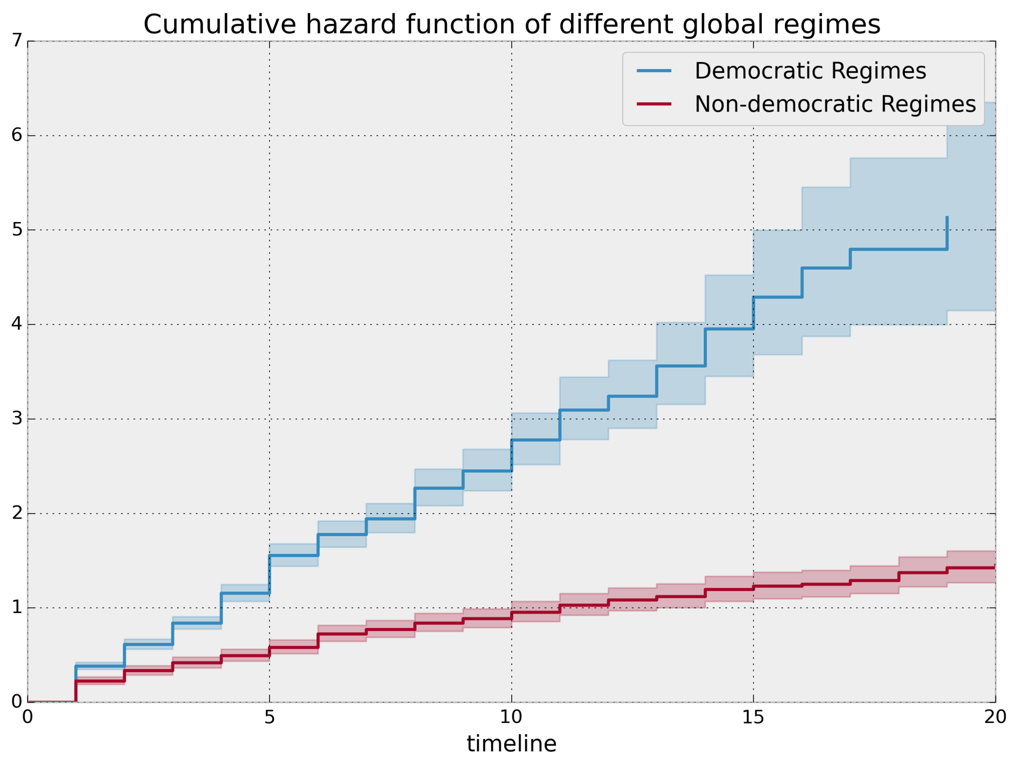


The cumulative hazard has less immediate understanding than the survival curve, but the hazard curve is the basis of more advanced techniques in survival analysis. Recall that we are estimating *cumulative hazard curve*, $\Lambda(t)$. (Why? The sum of estimates is much more stable than the point-wise 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 `ix` argument in the call to `plot` here: it accepts a `slice` and plots only points within that slice.

```
naf.fit(T[dem], event_observed=C[dem], label="Democratic Regimes")
ax = naf.plot(ix=slice(0,20))
naf.fit(T[~dem], event_observed=C[~dem], label="Non-democratic Regimes")
naf.plot(ax=ax, ix=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. So why did the combination of both regimes have a *decreasing* hazard? This is the effect of *frailty*, a topic we will discuss later.

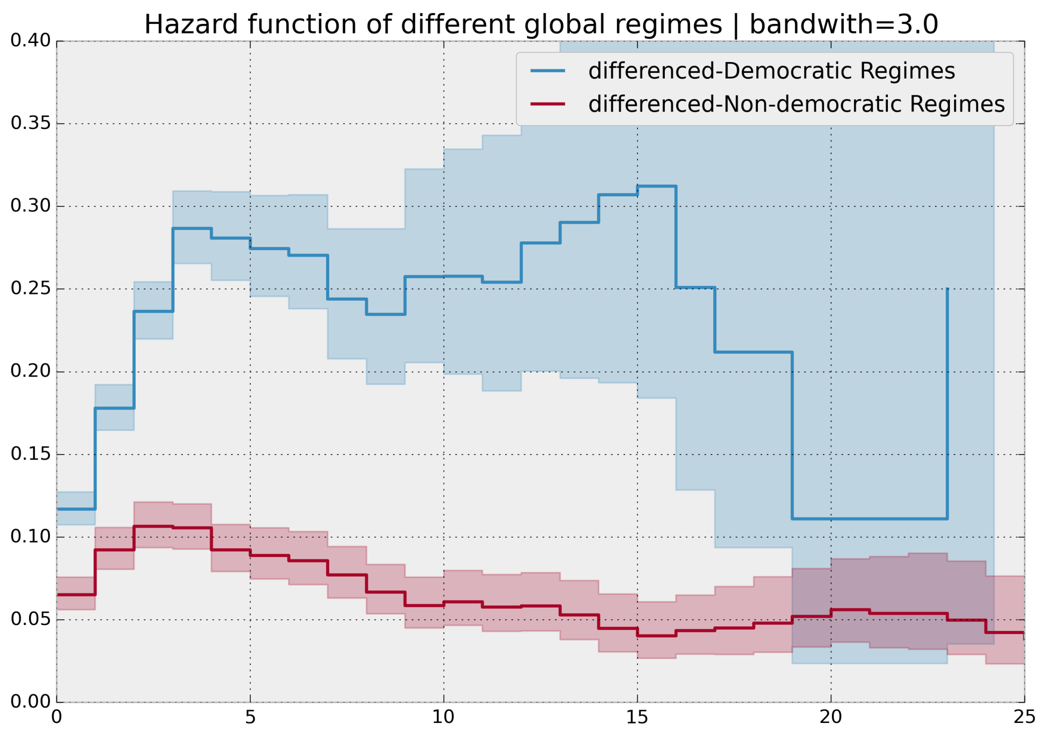
Smoothing the hazard curve

Interpretation of the cumulative hazard function can be difficult – it is not how we usually interpret functions. (On the other hand, most survival analysis is done using the cumulative hazard fuction, so understanding it is recommended).

Alternatively, we can derive the more-interpretable hazard curve, but there is a catch. The derivation involves a kernel smoother (to smooth out the differences of the cumulative hazard curve) , and this requires us to specify a bandwidth parameter that controls the amount of smoothing. This functionality is provided in the `smoothed_hazard_` and `hazard_confidence_intervals_` methods. (Why methods? They require an argument representing the bandwidth).

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

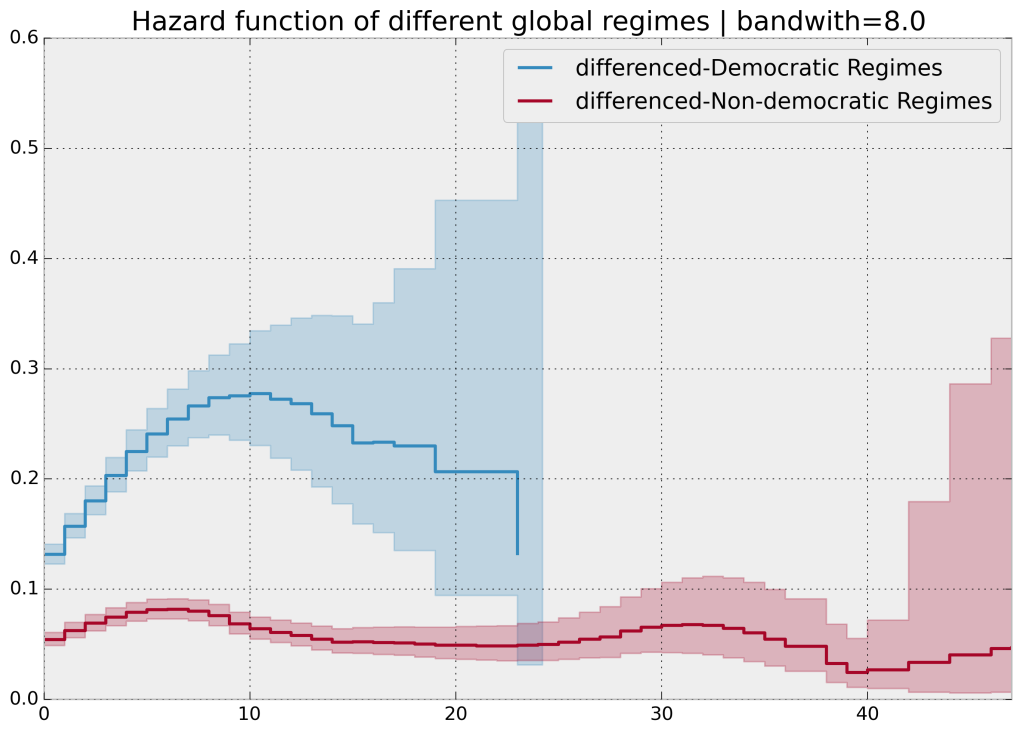
```
b = 3.
naf.fit(T[dem], event_observed=C[dem], label="Democratic Regimes")
ax = naf.plot_hazard(bandwidth=b)
naf.fit(T[~dem], event_observed=C[~dem], label="Non-democratic Regimes")
naf.plot_hazard(ax=ax, bandwidth=b)
plt.title("Hazard function of different global regimes | bandwith=%.1f"%b);
plt.ylim(0,0.4)
plt.xlim(0,25);
```



It is more clear here which group has the higher hazard, and like hypothesized above, both hazard rates are close to being constant.

There is no obvious way to choose a bandwith, and different bandwidth can produce different inferences, so best to be very careful here. (My advice: stick with the cumulative hazard function.)

```
b = 8.
naf.fit(T[dem], event_observed=C[dem], label="Democratic Regimes")
ax = naf.plot_hazard(bandwidth=b)
naf.fit(T[~dem], event_observed=C[~dem], label="Non-democratic Regimes")
naf.plot_hazard(ax=ax, bandwidth=b)
plt.title("Hazard function of different global regimes | bandwith=%.1f"%b);
```



1.3.3 Other types of censorship

Left Censored Data

We've mainly been focusing on *right-censorship*, 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.

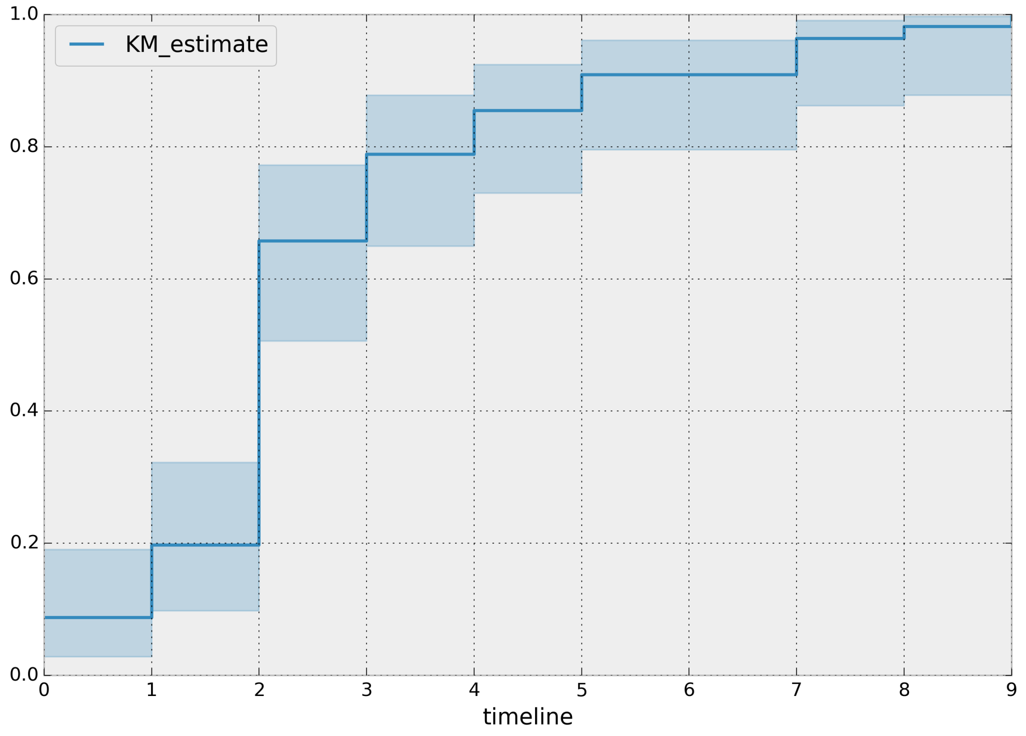
Another situation where we have left censored data is when measurements have only an upperbound, that is, the measurements instruments could only detect the measurement was *less* than some upperbound.

lifelines has support for left-censored datasets in the `KaplanMeierFitter` class, by adding the keyword `left_censorship=True` (default `False`) to the call to `fit`.

```
from lifelines.datasets import load_lcd
lcd_dataset = load_lcd()

ix = lcd_dataset['group'] == 'alluvial_fan'
T = lcd_dataset[ix]['T']
C = lcd_dataset[ix]['C'] #boolean array, True if observed.

kmf = KaplanMeierFitter()
kmf.fit(T,C, left_censorship=True)
```

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

```
kmf.cumulative_density_
kmf.plot() #will plot the CDF
```



Left Truncated Data

Another form of bias that can be introduced into a dataset is called left-truncation. (Also a form of censorship). This occurs when individuals may die even before ever entrying into the study. Both `KaplanMeierFitter` and `NelsonAalenFitter` have an optional arugment for `entry`, which is an array of equal size to the duration array. It describes the offset from birth to entering the study. This is also useful when subjects enter the study at different points in their lifetime. For example, if you are measuring time to death of prisoners in prison, the prisoners will enter the study at different ages.

Note: Nothing changes in the duration array: it still measures time from entry of study to time left study (either by death or censorship)

Note: Other types of censorship, like interval-censorship, are not implemented in *lifelines* yet.

1.4 Survival Regression

Often we have additional data aside from the durations, and if applicable any censorships that occurred. In the regime dataset, we have the type of government the political leader was part of, the country they were head of, and the year they were elected. Can we use this data in survival analysis?

Yes, the technique is called *survival regression* – the name implies we regress covariates (eg: year elected, country, etc.) against a another variable – in this case durations and lifetimes. Similar to the logic in the first part of this tutorial, we cannot use traditional methods like linear regression.

There are two popular competing techniques in survival regression: Cox's model and Aalen's additive model. Both models attempt to represent the hazard rate $\lambda(t)$. In Cox's model, the relationship is defined:

$$\lambda(t) = b_0(t) \exp\left(b_1 x_1 + ... + b_N x_n\right)$$

On the other hand, Aalen's additive model assumes the following form:

$$\lambda(t) = b_0(t) + b_1(t)x_1 + ... + b_N(t)x_T$$

Warning: These are still experimental.

1.4.1 Aalen's Additive model

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

Aalens additive model typically does not estimate the individual $b_i(t)$ but instead estimates $\int_0^t b_i(s) \; ds$ (similar to estimate of the hazard rate using `NelsonAalenFitter` above). This is important to keep in mind when analzying the output.

```
from lifelines import AalenAdditiveFitter
data.head()
```

I'm using the lovely library `patsy` <https://github.com/pydata/patsy>'__ here to create a covariance matrix from my original dataframe.

```
import patsy
# the '-1' term
# refers to not adding an intercept column (a column of all 1s).
# It can be added to the Fitter class.
X = patsy.dmatrix('un_continent_name + regime + start_year -1', data, return_type='dataframe')
```

```
X.columns
```

```
['un_continent_name[Africa]',
 'un_continent_name[Americas]',
 'un_continent_name[Asia]',
 'un_continent_name[Europe]',
 'un_continent_name[Oceania]',
 'regime[T.Military Dict]',
 'regime[T.Mixed Dem]',
 'regime[T.Monarchy]',
 'regime[T.Parliamentary Dem]',
 'regime[T.Presidential Dem]',
 'start_year']
```

Below we create our Fitter class. Since we did not supply an intercept column in our matrix we have included the keyword `fit_intercept=True` (`True` by default) which will append the column of ones to our matrix. (Sidenote: the intercept term, $b_0(t)$ in survival regression is often referred to as the *baseline* hazard.)

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.

```
aaf = AalenAdditiveFitter(coef_penalizer=1.0, fit_intercept=True)
```

Like the API syntax above, 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 special columns: a *duration* column and a boolean *event occured* column (where event occured refers to the event of interest - expulsion from government in this case)

```
data = lifelines.datasets.load_dd()

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

The api for .fit was different prior to lifelines 0.3, below refers to the 0.3+ versions

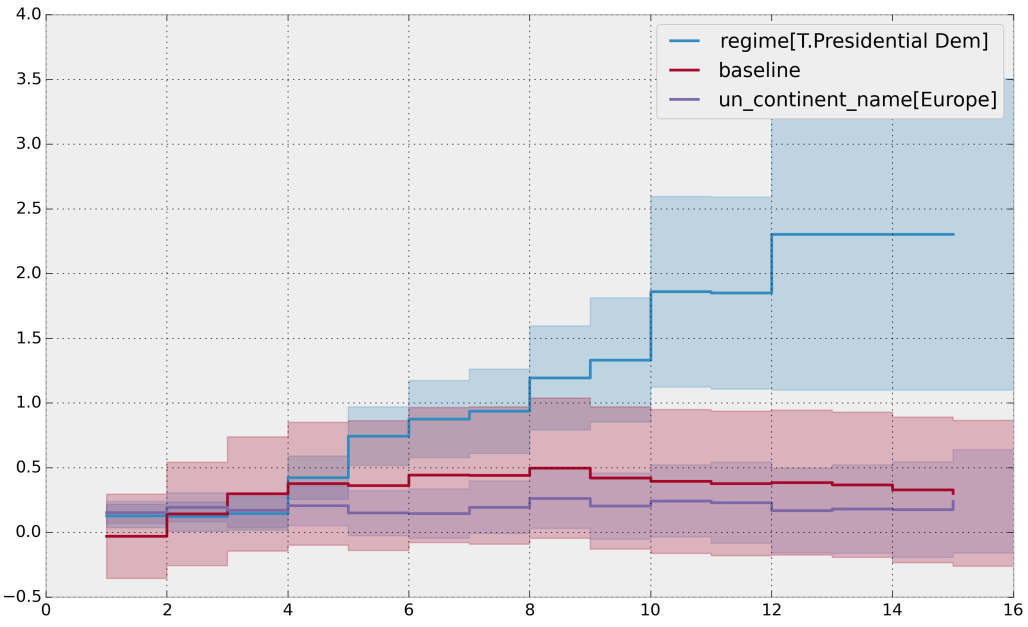
```
aaf.fit(X, 'T', event_col='E')
```

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

```
figsize(12.5,8)
aaf.cumulative_hazards_.head()
```

`AalenAdditiveFitter` also has built in plotting:

```
aaf.plot( columns=[ 'regime[T.Presidential Dem]', 'baseline', 'un_continent_name[Europe]' ], ix=slice
```



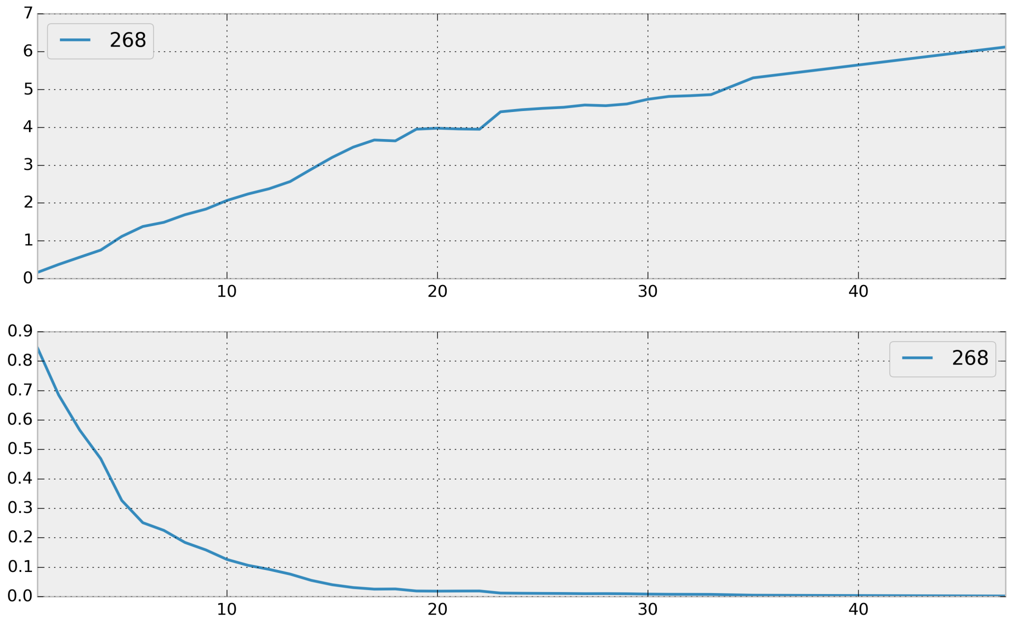
Regression is most interesting if we use it on data we have not yet seen, i.e. prediction! We can use what we have learned to predict individual hazard rates, survival functions, and median survival time. The dataset we are using is limited to 2008, so let's use this data to predict the (though already partly seen) possible duration of Canadian Prime Minister Stephen Harper.

```
ix = (data['ctryname'] == 'Canada') * (data['start_year'] == 2006)
harper = X.ix[ix]
print "Harper's unique data point", harper
```

```
Harper's unique data point
```

```
array([[    0.,      0.,      1.,      0.,      0.,      0.,      0.,      1.,
            0.,      0.,   2003.]])
```

```
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);
```



1.4.2 Cox's Proportional Hazard model

New in 0.4.0 is the implementation of the Propotional Hazard's regression model (implemented in R under `coxph`). It has a similar API to Aalen's Additive model. Like R, it has a `print_summary` function that prints a tabuluar view of coefficients and related stats.

This example data is from the paper here.

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

rossi_dataset = load_rossi()
cf = CoxPHFitter()
cf.fit(rossi_dataset, 'week', event_col='arrest')

cf.print_summary()  # access the results using cf.summary
```

```
"""
n=432, number of events=114

           coef  exp(coef)  se(coef)          z          p  lower 0.95  upper 0.95
fin  -1.897e-01  8.272e-01 9.579e-02 -1.981e+00 4.763e-02  -3.775e-01  -1.938e-03   *
age  -3.500e-01  7.047e-01 1.344e-01 -2.604e+00 9.210e-03  -6.134e-01  -8.651e-02  **
race  1.032e-01  1.109e+00 1.012e-01  1.020e+00 3.078e-01  -9.516e-02   3.015e-01
wexp -7.486e-02  9.279e-01 1.051e-01 -7.124e-01 4.762e-01  -2.809e-01   1.311e-01
mar  -1.421e-01  8.675e-01 1.254e-01 -1.134e+00 2.570e-01  -3.880e-01   1.037e-01
paro -4.134e-02  9.595e-01 9.522e-02 -4.341e-01 6.642e-01  -2.280e-01   1.453e-01
prio  2.639e-01  1.302e+00 8.291e-02  3.182e+00 1.460e-03   1.013e-01   4.264e-01  **
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Concordance = 0.642
"""
```

To access the coefficients and the baseline hazard, you can use `cf.hazards_` and `cf.baseline_hazard_` respectively. After fitting, you can use use the suite of prediction methods (similar to Aalen's additve model above): `.predict_hazard(X)`, `.predict_survival_function(X)`, etc.

Stratification

Sometimes a covariate may not obey the proportional hazard assumption. In this case, we can allow a factor to be adjusted for without estimating its effect. To specify categorical variables to be used in stratification, we specify them in the call to `fit`:

```
cf.fit(rossi_dataset, 'week', event_col='arrest', strata=['race'])

cf.print_summary()  # access the results using cf.summary
"""
n=432, number of events=114

           coef  exp(coef)  se(coef)          z          p  lower 0.95  upper 0.95
fin  -1.890e-01  8.278e-01 9.576e-02 -1.973e+00 4.848e-02  -3.767e-01  -1.218e-03   *
age  -3.503e-01  7.045e-01 1.343e-01 -2.608e+00 9.106e-03  -6.137e-01  -8.700e-02  **
wexp -7.107e-02  9.314e-01 1.053e-01 -6.746e-01 4.999e-01  -2.776e-01   1.355e-01
mar  -1.452e-01  8.649e-01 1.255e-01 -1.157e+00 2.473e-01  -3.911e-01   1.008e-01
paro -4.079e-02  9.600e-01 9.524e-02 -4.283e-01 6.684e-01  -2.275e-01   1.459e-01
prio  2.661e-01  1.305e+00 8.319e-02  3.198e+00 1.381e-03   1.030e-01   4.292e-01  **
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Concordance = 0.638
"""
```

1.4.3 Model Selection in Survival Regression

With censorship, it's not correct to use a loss function like mean-squared-error or mean-absolute-loss. Instead, one measure is the c-index, or concordance-index. This measure evaluates the ordering of predicted times: how correct is the ordering? It is infact a generalization of AUC, another common loss function, and is interpretted 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)

The measure is implemented in lifelines under *lifelines.statistics.concordance_index* and accepts the actual times (along with any censorships), and the predicted times.

Cross Validation

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).

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

regression_dataset = load_regression_dataset()
cf = CoxPHFitter()
scores = k_fold_cross_validation(cf, regression_dataset, 'T', event_col='E', k=3)
print scores
print np.mean(scores)
print np.std(scores)

#[ 0.5896  0.5358  0.5028]
# 0.542
# 0.035
```

1.5 More Examples and Recipes

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

1.5.1 Compare two populations statistically

(though this applies just as well to Nelson-Aalen estimates). Often researchers want to compare survival curves between different populations. Here are some techniques to do that:

Subtract the difference between survival curves

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

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

will produce the difference at every relevant time point. A similar function exists for division: `divide`.

Compare using a hypothesis test

For rigorous testing of differences, *lifelines* comes with a statistics library. The `logrank_test` function compares whether the "death" generation process of the two populations are equal:

```
 from lifelines.statistics import logrank_test

 results = logrank_test(T1, T2, event_observed_A=C1, event_observed_B=C2)
 results.print_summary()

 """
 Results
     df: 1
    alpha: 0.95
    t 0: -1
    test: logrank
    null distribution: chi squared

    __ p-value ___|__ test statistic __|____ test results ____|__ significant __
          0.46759 |              0.528 |  Cannot Reject Null  |      False
"""

print results.p_value     # 0.46759
print results.test_statistic # 0.528
print results.is_significant # False
```

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).

1.5.2 Model selection using *lifelines*

If using *lifelines* for prediction work, it's ideal that you perform some sort of cross-validation scheme. This 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:

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

df = load_regression_dataset()

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

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

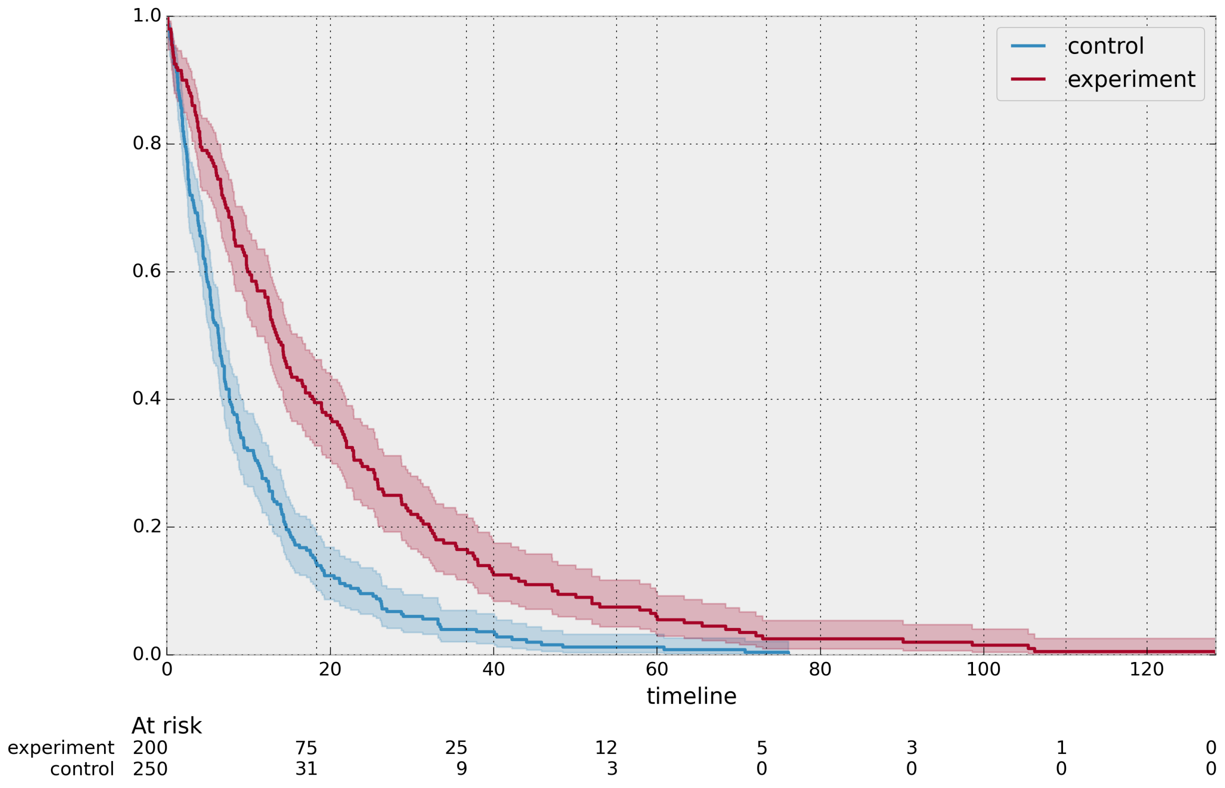
From these results, Aalen's Additive model with a penalizer of 10 is best model of predicting future survival times.

1.5.3 Displaying 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:

```
from numpy.random import exponential
T_control = exponential(10, size=250)
T_experiment = exponential(20, size=200)
ax = plt.subplot(111)

from lifelines import KaplanMeierFitter

kmf_control = KaplanMeierFitter()
ax = kmf_control.fit(T_control, label='control').plot(ax=ax)

kmf_exp = KaplanMeierFitter()
ax = kmf_exp.fit(T_experiment, label='experiment').plot(ax=ax)


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

will display



At risk

experiment	200	75	25	12	5	3	1	0
control	250	31	9	3	0	0	0	0

Alternatively, you can add this at the call to `plot`: `kmf.plot(at_risk_counts=True)`

1.5.4 Getting survival-table data into *lifelines* format

lifelines classes are designed for lists or arrays that represent one individual per element. 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:

time (months, days, ...)	observed deaths	censored
0	7	0
1	1	1
2	2	0
3	1	2
4	5	2
...	...	...

```
import pandas as pd

# your argument in the function call below will be different
df = pd.read_csv('file.csv', index_cols=[0], columns = ['observed deaths', 'censored'] )

from lifelines.utils import survival_events_from_table

T,C = survival_events_from_table(df, observed_deaths_col='observed deaths', censored_col='censored')
print T # np.array([0,0,0,0,0,0,0,1,2,2, ...])
print C # np.array([1,1,1,1,1,1,1,0,1,1, ...])
```

Alternatively, perhaps you are interested in viewing the survival table given some durations and censorship vectors.

```
from lifelines.utils import survival_table_from_events

table = survival_table_from_events(T, C)
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
"""
```

1.5.5 Plotting multiple figures on an plot

When *.plot* is called, an *axis* object is returned which can be passed into future calls of *.plot*:

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

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

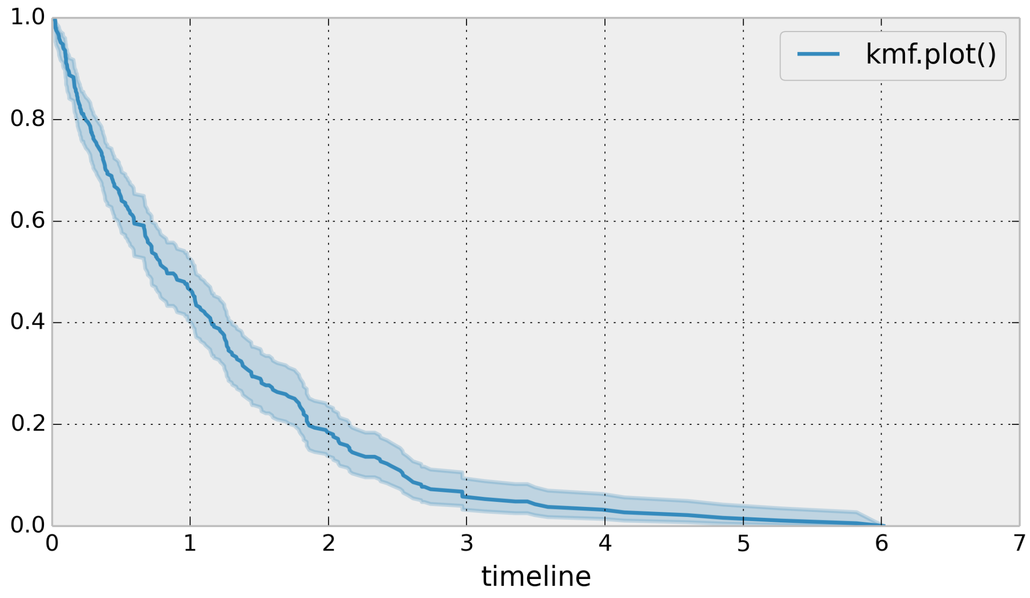
If you have a pandas *DataFrame* with columns “group”, “T”, and “C”, then something like the following would work:

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

ax = plt.subplot(111)

kmf = KaplanMeierFitter()
for group in df['group'].unique():
    data = grouped_data.get_group(group)
    kmf.fit(data["T"], data["C"], label=group)
    kmf.plot(ax=ax)
```

1.5.6 Plotting options and styles

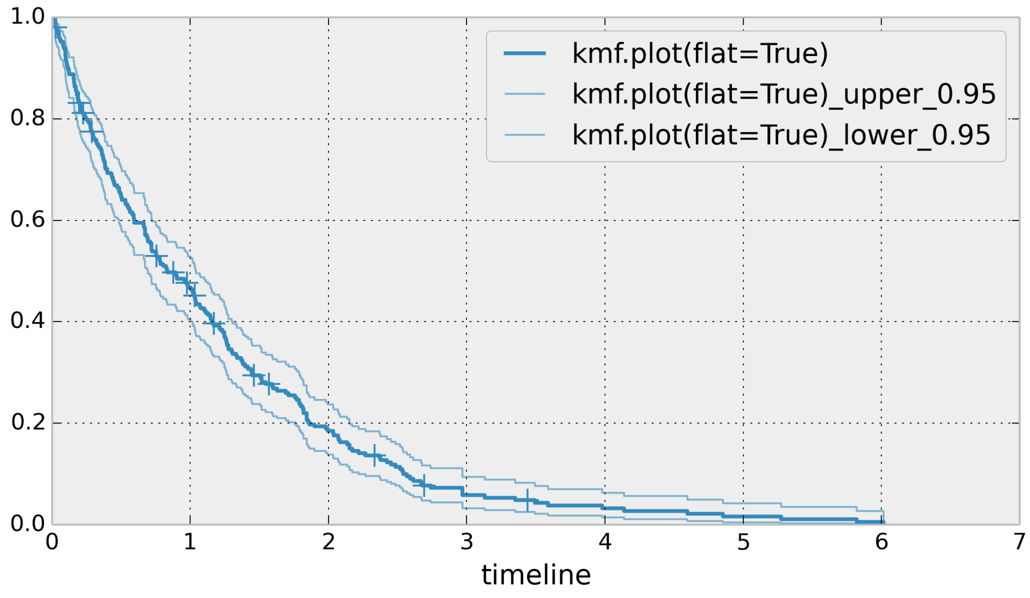
Standard

```
kmf = KaplanMeierFitter()
kmf.fit(T,C,label="kmf.plot()")
kmf.plot()
```



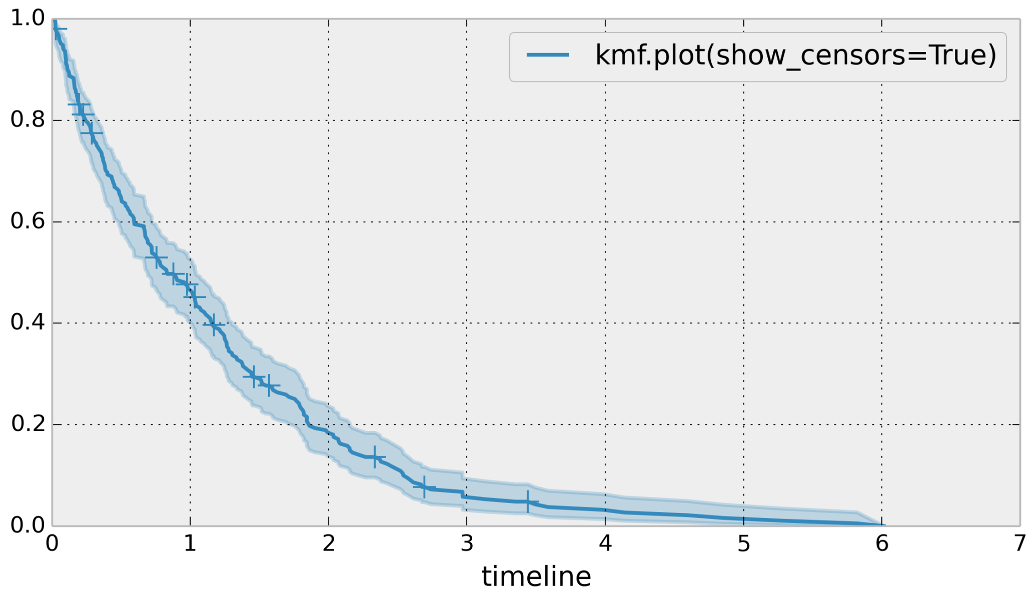
R-style

```
kmf.fit(T,C,label="kmf.plot(flat=True)")
kmf.plot(flat=True)
```



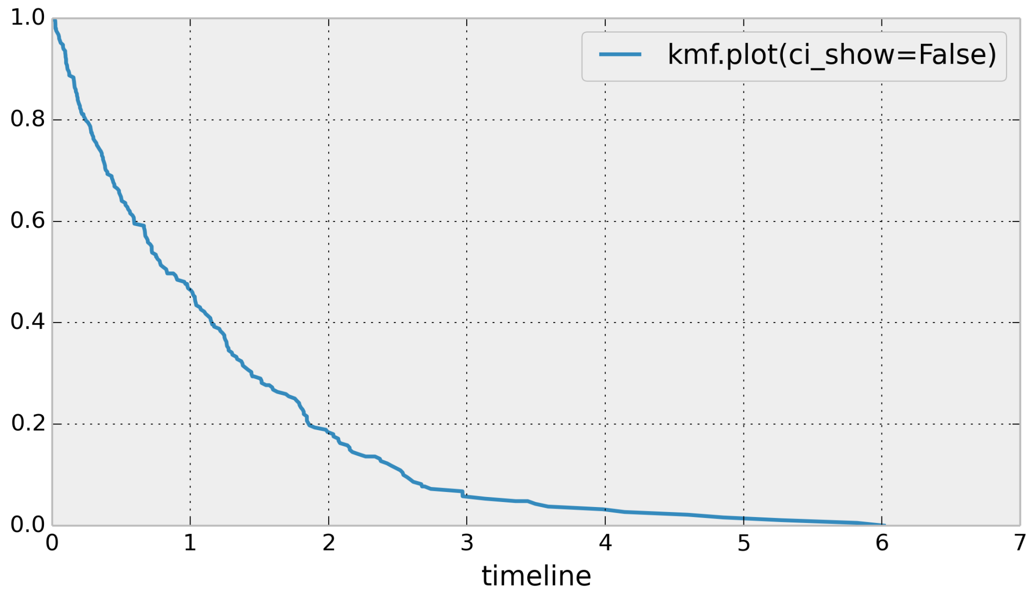
Show censorships

```
kmf.fit(T,C,label="kmf.plot(show_censors=True)")
kmf.plot(show_censors=True)
```



Hide confidence intervals

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



1.5.7 Set the index/timeline of a estimate

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

```
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 really 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 – this is caused by 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:

```
naf.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.

1.5.8 Example SQL query to get data from a table

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

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

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:

id	T	C
10	40	True
11	42	False
12	42	False
13	36	True
14	33	True

Installation

Dependencies are from the typical Python data-stack: Numpy, Pandas, Scipy, and optionally Matplotlib. Install using:

```
pip install lifelines
```

CHAPTER 3

Source code and Issue Tracker

Available on Github, CamDavidsonPilon/lifelines Please report bugs, issues and feature extensions there.

CHAPTER 4

Indices and tables

- genindex
- modindex
- search