SurPyval

Aug 04, 2021

Quickstart Intro

surpyval is an implementation of survival analysis in Python. The intent of this was to see if I could actually make it, and therefore learn a lot about survival analysis along the way, but also so that each time a model is created, it can be reused by other planned projects for monte carlo simulations (used in reliability engineering) and optimisations.

One feature of surpyval that separates it from other survival analysis packages is the intuitive way with which you can pass data to the fit methods. There are many different formats that can be used for survival analysis; surpyval handles many of the conceivable ways you can have your data stored. This is discussed in the data format tab.

Surpyval is also unique in the way in which it lets you estimate the parameters. With surpyval, you can use any of the following methods to estimate the parameters of you distribution of interest:

Table 1: SurPyval Modelling Methods

Method	Para/Non-Para	Observed	Censored	Truncated
Maximum Likelihood (MLE)	Parametric	Yes	Yes	Yes
Probability Plotting (MPP)	Parametric	Yes	Yes	Limited
Mean Square Error (MSE)	Parametric	Yes	Yes	Limited
Method of Moments (MOM)	Parametric	Yes	No	No
Maximum Product Spacing (MPS)	Parametric	Yes	Yes	No (planned)
Kaplan-Meier	Non-Parametric	Yes	Right only	Left only
Nelson-Aalen	Non-Parametric	Yes	Right only	Left only
Fleming-Harrington	Non-Parametric	Yes	Right only	Left only
Turnbull	Non-Parametric	Yes	Yes	Yes

Most other survival analysis packages focus on just using the MLE, or maybe the Probability Plotting. This package grew out of replicating the historically used probability plotting method from engineering, and as it progressed, it was discovered that there are many many ways parameters of distributions can be estimated. The product spacing estimator is particularly useful for offset distributions or finitely bounded distributions.

SurPyval attempts to use the combination of these methods to make parameter estimation possible for any distribution with arbitrary combnations of observations, censoring, and truncation.

Becoming a competent survival analyst depends strongly on having a very strong understanding of censoring, truncation, and observations in conjunction with a solid understanding of different types of distributions. Knowing and being able to identify situations as being censored or truncated in real applications will ensure you do not make an errors in your analysis. This can be very difficult to do. This documention can be used as a reference to understand the types of censoring and truncation so that you can identify these situations in your work. Further, having a deep understanding of the types of distributions used in survival analysis will allow you to identify the process that is generating your data. This will then allow you to select an appropriate distribution, if any, to solve your problem. Survival analysis is an extremely powerful, and thoroughly interesting tool, so don't give up, or if you do give up, do the survival statistics on it.

CHAPTER 1

Contents:

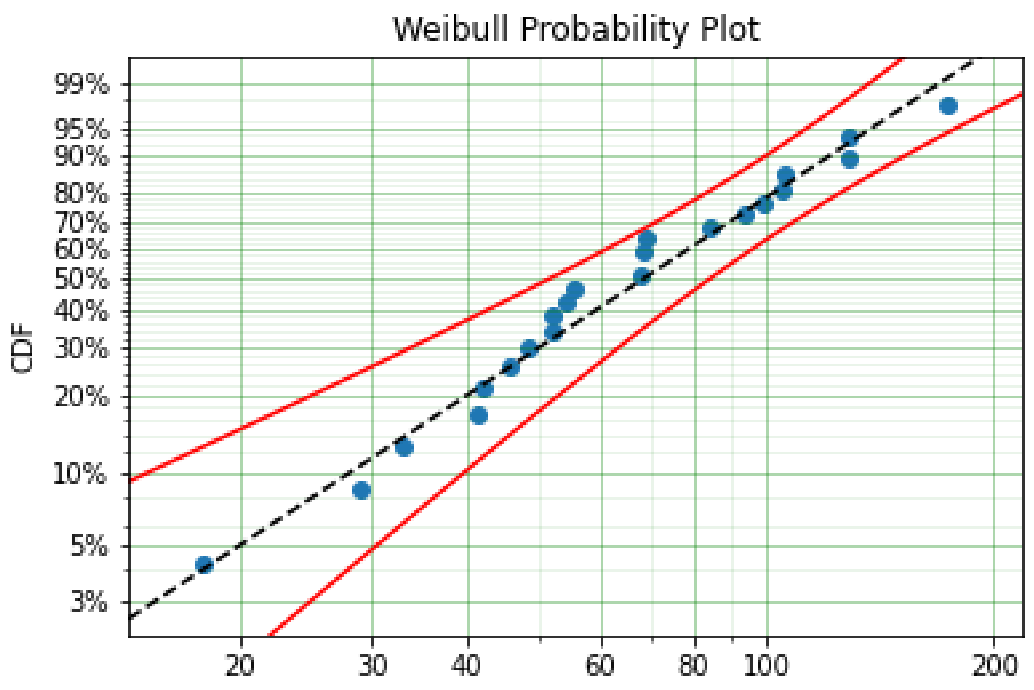
1.1 Quickstart

So, you know what survival analysis is and you just want to see what this can do.

Alas

```
import surpyval as surv

x = [17.88, 28.92, 33, 41.52, 42.12, 45.6, 48.4, 51.84,
 51.96, 54.12, 55.56, 67.8, 68.64, 68.64, 68.88, 84.12,
 93.12, 98.64, 105.12, 105.84, 127.92, 128.04, 173.4]

model = surv.Weibull.fit(x)
model.plot()
```

This gives us the Weibull plot



1.2 Types of Data

Survival analysis is the statistics about durations. To understand durations, or time to events, we must have data that captures how long something lasts. This is the start of survival analysis where we have data in the form of a list of durations of some time to event. This time to event can be engineering failure data, health data on time to death from a given disease, economic data on the duration of a recession or time between recessions, or it could be race times for a group of athletes in a triathlon.

Survival analysis is unique in statistics because of the types of data that we encounter, specifically censoring and truncation. The purpose of this section is to explain these types of data and the scenarios under which they are generated so that you can understand when you might need to use the different flags in surpyval in your analysis.

1.2.1 Exactly Observed

The first type of data is exactly observed data. This is the type of data where we know exactly when the death or failure occurred. For example, if I run a test on how long some light bulbs will last, I get 5 and turn them on and watch them continuously. Then as each fail I record their failure times 983, 1321, 1889, 1923, and 2932 hours. Each of these times is exact because I saw the exact moment at which they failed. So the task is then understanding the distribution of these exact failures.

1.2.2 Censored Data

Say that I got bored of sitting and looking at light bulbs. And because of this boredome I stopped looking at the light bulbs at 1900 hours. I would therefore not have seen two of the light bulbs fail, the failures that would have occured at 1923 and 2932 hours. All we would know about these two light bulbs is that they failed sometime *after* 1,900 hours. That is, we know that these two light bulbs would have failed if they continued the test but that this faliure time is greater than 1,900. This is to say that the failure time has been *censored*. Specifically the failure has been right censored. 'Right' is used because if we consider a (horizontal) timeline with time progressing along the line from left to right, we know that the failure would have occured to the right of the time at which we stopped our observation. Hence, the observation is right censored.

If on the other hand, I also knew it would take some time for the bulbs to start failing so instead of waiting from the very start of the test I did not sit there for the first 1,000 hours. That is, the test continues to run but is not being observed for the first 1,000 hours, then after the first 1,000 hours I return to the test and start my observations. When I return I find that a bulb has failed. From the original data, I see that there was a bulb that failed at 983 hours. But if I was not observing for the first 1,000 hours all I would know about this failure is that it occurred sometime *before* 1,000 hours. Using the timeline concept again, I know that the failure would have occurred to the left of the 1,000 hour mark. Therefore, we say that the failure is *left* censored.

Finally, had I not been patient enough to sit down for any extended period of time and instead inspected the light bulbs at different times to see if any had failed. So say I inspect the bulbs every 100 hours from 1000 hours till 2,000 hours. The first and last failures would be left and right censored. But the middle failures would be known to fail between inspections. So the second failure would have occurred between the 1300 and 1400 hours inspections, the third between 1800 and 1900, and the second last failure would have happened between the 1900 and 2000 hours inspections. These failures are said to be *intervally* censored. That is because they are known to have happened in a given interval on a timeline.

Survival analysis has several methods for handling censored data in the parametric and non-parametric analysis. Surpyval is able to handle an input that has an arbitrary combination of observed and left, right, and intervally censored failure data. Although, not all methods can handle all types of data. This is covered in the sections on each of the estimation and fitting methods.

Surpyval uses a convention regarding censoring. Specifically, surpyval takes as input, with a list of failure times 'x', an optional censoring flag array 'c'. If no flaggin array is provided, it is assumed that all the data are exact observations, i.e. that they are not censored. But if the 'c' array is provided, it must have a value for each value in the x input. That is, they must be the same length. The possible values of c are -1, 0, 1, and 2. The convention tries to illustrate the concept of left, right, and interval censoring on the timeline. That is, -1 is the flag for left censoring because it is to the left of an observed failure. With an observed failure at 0. 1 is used to flag a value as right censored. Finally, 2 is used to flag a value as being intervally censored because it has 2 data points, a left and right point. In practice this will therefore look like:

```python
import surpyval

x = [3, 3, 3, 4, 4, [4, 6], [6, 8], 8]
c = [-1, -1, -1, 0, 0, 2, 2, 1]

model = surpyval.Weibull.fit(x=x, c=c)
```

This example shows the flexibility surpyval offers. It allows users to analyse data that has any arbitrary combination of the different types of censoring. The surpyval format is even more powerful, because the above example can be condensed even further through using the 'n' value.

```python
import surpyval

x = [3, 4, [4, 6], [6, 8], 8]
c = [-1, 0, 2, 2, 1]
n = [3, 2, 1, 1, 1]

model = surpyval.Weibull.fit(x=x, c=c, n=n)
```

The first step of the fit method actually wrangles the input data into the densest form possible. So internally, the example without the n value, will be condensed to be the second example without you seeing it. But it shows the capability of how data can be input to surpyval if you have different formats. But we are getting away from data types. . .

1.2.3 Truncated Data

For my light bulb test, let's say I test a different manufacturers bulbs. This time, I know that the bulbs from this manufacturer have been tested for 500 hours prior to shipping them. This situation needs to be treated differently because we know that in this circumstance we only have the bulbs because they survived more than 500 hours. If there were any failures prior to 500 hours the bulb would not have been shipped and therefore would not be being tested by me. This is to say, that my observation of the distribution of the light bulb failures has been *truncated*. In this regime there is no way I can have any observation below 500 hours because of the testing then discarding done by the manufacturer. The astute reader might have observed that this data is in fact *left* truncated. This is because the truncation occurs to the left of the observation on a timeline. In this example, all the bulbs are left truncated at the 500 hour mark.

In biostatistics left truncation is known as 'late-entry', this is because in clinical trials a participant can enter a trial later than other participant. Therefore this participant was at risk of not being present in the trial. This is because they could have died prior to entering the trial. Morbid, yes, but the estimate of the distribution needs to account for this risk otherwise the estimate will overestimate the true risk of the event.

Right truncated data is when you only observe a value because it happened below some time. For example, in the light bulb experiment, I received some of the bulbs that passed the burn in test. That is, I received some of the bulbs that survived the original 500 hours of testing. But if the failed bulbs were then given to an engineering team to investigate possible design changes that will improve reliability; they will have a series of failure times that must be below 500 hours. That is, from their perspective, they have data that is right truncated. There is one condition to this situation, they must not know how many other bulbs were tested. If they knew how many other bulbs were tested, they would know how many would fail after 500 hours. That is, they would know that all the other bulbs are right censored. So for our engineers investigating the failed bulbs, they must be ignorant of how many other bulbs were actually tested for the right truncation to work for them. In many applications we do know how many were under test and therefore right truncation become right censoring, but from our engineers circumstance, we can see that they are right censored.

Parametric and non-parametric analysis can both handle left truncated data. This is explained further in the estimation methods for both these methods. Right truncation can only be handles in surpyval with parametric analysis, specifically, with Maximum Likelihood Estimation and in limited cases with Minimum Product Spacing. This is also explained in their respective sections of these notes.

In surpyval, passing truncated data to the fitting method looks like:

```
import surpyval

x  = [674, 792, 1153, 1450, 1555, 1923, 2019]
tl = 500

model = surpyval.Weibull.fit(x=x, tl=tl)
```

1.2.4 Concluding Points

Having read through the above explanation you might be thinking how often these scenarios appear in real data, if ever. The vast majority of data used in survival analysis is observed or right censored. This is what happens when you observe a whole population but finish the observation before the event happens on all the items being observed.

Right truncation is extremely rare because it only happens if you do not know the size of the whole population under test. It can happen with scientific instruments where say, a camera is limited in the frequencies of light it can capture. So if we were to try capture a distribution of light of an object, say a star, this distribution could be truncated above and below certain frequencies. Meeker and Escobar provide an example in their book on reliability statistics for warranty analysis, similar to the contrived example provided above. If you have some returns of products from the field, these are right-truncated because you do not know what has been bought and used in the field. A more realistic example could be the estimation of race finish times at a triathlon or marathon. If I arrive at the finish line of a race and record the times of participants as they cross the line during that window I will have truncated data. I do not know how many people started the race (presumably) and I only stay and watch for a given period of time, therefore all the observations I make are truncated within the window of my observation time. In conclusion though, right truncation in survival analysis is rare.

Left truncation is common in insurance studies. If an insurance company wants to estimate the distribution of losses due to property crime based on policy payouts they need to consider the impact of 'excess'. Excess is the cost of making a claim on an insurance policy. So if I have an insurance policy with an excess of $500, if I lose $20,000 worth of peoperty in a robbery I will have to pay $500 to be paid $20,000. Because of this, it is clear that if I lost $400 in a robbery I would not pay the $500 excess to make a claim. Therefore the distribution of property crime will be truncated by the value of the excesses on the policies. Actuaries need to consider this in their calculaitons of policy fees.

Insurance is also a good example of right censoring. An insurance policy will also have a maximum payout. So if calculating the distribution of the value of property crime an analyst will need to consider that those payouts that are at the maximum of that policy value are in fact censored. That is, the value of the loss or damage was greater than the actual payout and therefore the payout is a censored value. In the classic Boston housing pricing data there is censored data! A histogram of the values of houses shows that there is a large number of houses at the highest price. This can be understood because a limit was set on the highest possible value, therefore these house prices are actually censored, not exact observations.

1.3 Conventions

1.3.1 Variable Names

Before discussing the formats, the conventions for vairable names needs to be clarified. These conventions are:

- x = The random variable (time, stress etc.) array
- xl = The random variable (time, stress etc.) array for the left interval of interval censored data.
- xr = The random variable (time, stress etc.) array for the right interval of interval censored data.
- c = Refers to the censoring data array associated with x
- n = The count array associated with x
- t = the truncation values for the left and right truncation at x (must be two dim, or use tl and tr instead)
- r = risk set at x
- d = failures/deaths at x
- t = two dimensional array of the truncation interval at x
- tl = one dimensional array of the value at which x is left truncated
- tr = one dimensional array of the value at which x is right truncated

1.3.2 Data Formats

The conventional formats use in surpyval are:

- xcnt = x variables, with c as the censoring scheme, n as the counts, and t as the truncation
- xrd = x variables, with the risk set, r, at x and the deaths, d, also at x

All functions in surpyval have default handling conditions for c and n. That is, if these variables aren't passed, it is assumed that there was one observation and it was a failure for every x.

Surpyval fit() functions use the xcnt format. But the package has handlers for other formats to rearrange it to the needed format. Other formats are:

wranglers for formats:

- fs = failure time array, f, and right censored time array, s
- fsl = fs format plus an array for left censored times.

1.3.3 Censoring conventions

For the censoring values, surpyval uses the convention used in Meeker and Escobar, that is:

- -1 = left
- 0 = failure / event
- 1 = right
- 2 = interval censoring. Must have left and right value in x

This convention gives an intuitive feel for the placement of the data on a timeline.

1.3.4 Function Conventions

The conventions for SurPyval are that each object returned from a `fit()` call has the ability to compute the following:

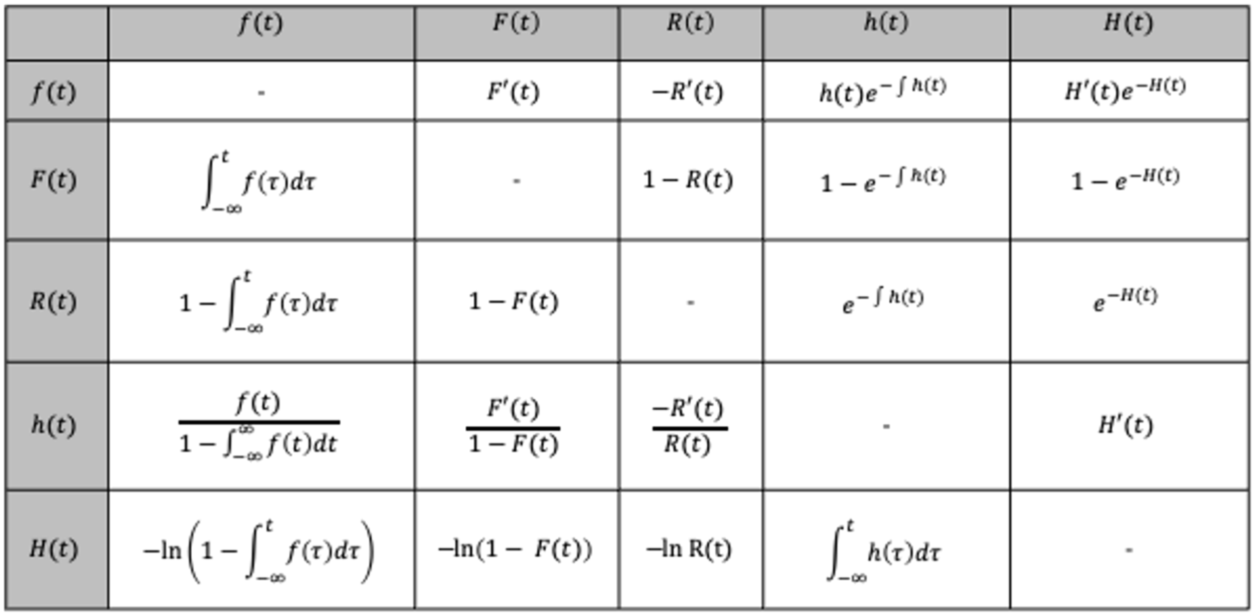
- `df()` - The density function
- `ff()` - The CDF
- `sf()` - The survival function, or reliability function
- `hf()` - The (instantaneous) hazard function
- `Hf()` - The cumulative hazard function

These functions can be used to plot or even in optimisers so that you can optimize decisions that you are guiding with your survival analysis.

1.4 Handy References - Aide-mémoire

1.4.1 Relationship between functions of a probability distribution

There exists a relationship between each of the functions of a distribution and the others. This can be very useful to keep in mind when understanding how surpyval works. For example, the Nelson-Aalen estimator is used to estimate the cumulative hazard function (Hf), the below relationships is how distribution for this can be used to estimate the survival function, or the cdf.



	$f(t)$	$F(t)$	$R(t)$	$h(t)$	$H(t)$
$f(t)$	-	$F'(t)$	$-R'(t)$	$h(t)e^{-\int h(t)}$	$H'(t)e^{-H(t)}$
$F(t)$	$\int_{-\infty}^{t} f(\tau)d\tau$	-	$1-R(t)$	$1-e^{-\int h(t)}$	$1-e^{-H(t)}$
$R(t)$	$1-\int_{-\infty}^{t} f(\tau)d\tau$	$1-F(t)$	-	$e^{-\int h(t)}$	$e^{-H(t)}$
$h(t)$	$\frac{f(t)}{1-\int_{-\infty}^{\infty} f(t)dt}$	$\frac{F'(t)}{1-F(t)}$	$\frac{-R'(t)}{R(t)}$	-	$H'(t)$
$H(t)$	$-\ln\left(1-\int_{-\infty}^{t} f(\tau)d\tau\right)$	$-\ln(1-F(t))$	$-\ln R(t)$	$\int_{-\infty}^{t} h(\tau)d\tau$	-

The above table shows how the function on the left, can be described by the function along the top row (I leave out the function describing itself as it is simply itself…). So, an interesting one is that the reliability or survival function, R(t), is simply the exponentiated negative of the cumulative hazard function! This relationship holds for **every** distribution.

1.4.2 AFT, AL, or PH?

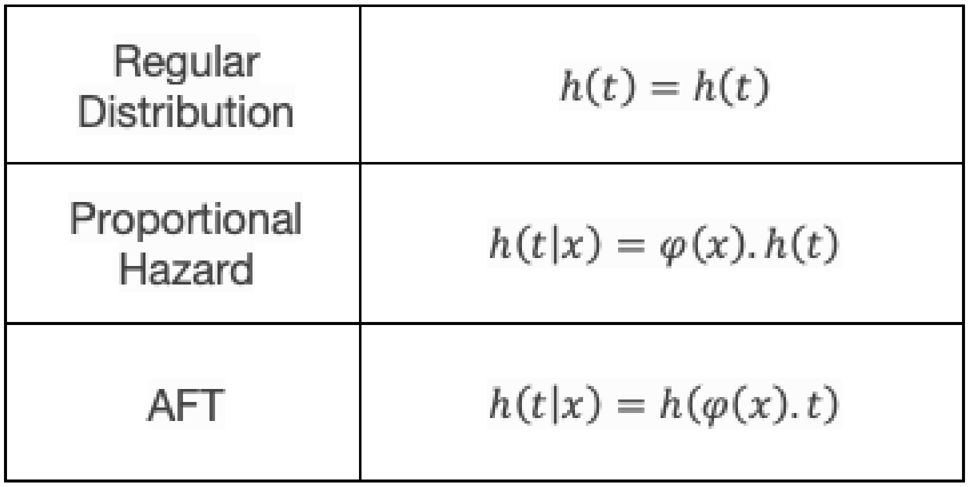
What is the difference, if any, between an Accelerated Failure Time model, an Accelerated Life model, and a Proportional Hazard model? SurPyval uses the distinctions defined in [Bagdonavicius]. The explanation of these are:

- ALT is an accelerated life model. That is, a model where the 'characteristic life' of the distribution is a function of the stress or stresses applied to the system. Another way to describe it is that, for two different stresses and two different times, t1 and t2, if the probability of failure at the times is the same.
- AFT is an Accelerated Failure Time model. This is simply a distsribution where the time is multiplied by a function of covariates. This has the effect of 'accelerating' the time. Concretely, for a function $f(t)$ it can be accelerated with a function to give $f(\phi(x)t)$.
- PH is a proportional hazard model. In a proportional hazard model, the hazard function is multiplied by some function of covariates. Hence if a function has a hazard rate of $h(x)$ then the proportional hazard model will give simply $\phi(x)h(t)$.

SurPyval has implementations, and even a general constructor, for AFT, AL, and PH models. Each of which can handle arbitrary censoring (truncation coming).

1.4.3 How an AFT and PH Model Relate to a regular distribution

An AFT, or accelerated failure time, model does exactly that. It 'accelerates' the actual time by multiplying the time in the hazard function by a function of factors, $\phi(x)$. This factor can be any function. A Proportional Hazard model also does exactly what it says, if changes the hazard rate by a particular proportion.



Regular Distribution	$h(t) = h(t)$
Proportional Hazard	$h(t\|x) = \varphi(x).h(t)$
AFT	$h(t\|x) = h(\varphi(x).t)$

Given the relationship between variables and a distribution with either the PH or AFT models, you can see, using the above relationships that the survival, failure, and density functions can all be determined. This relationship is good to know to understand how AFT and PH models work.

1.4.4 References

1.5 Data Wrangling Examples

Lets just say we have a list of right censored data and a list of failures. How can we wrangle these into data for the `fit()` method to accept?

```
import surpyval as surv

# Failure data
f = [2, 3, 4, 5, 6, 7, 8, 8, 9]
# 'suspended' or right censored data
s = [1, 2, 10]

# convert to xcn format!
x, c, n = surv.fs_to_xcn(f, s)
print(x, c, n)

model = surv.Weibull.fit(x, c, n)
print(model)
```

```
[ 1  2  2  3  4  5  6  7  8  9 10] [1 0 1 0 0 0 0 0 0 0 1] [1 1 1 1 1 1 1 1 2 1 1]
Parametric Surpyval model with Weibull distribution fitted by MLE yielding parameters␣
↪(7.200723109183674, 2.474773882227539)
```

You can even bring in your left censored data as well:

```
# Failure data
f = [2, 3, 4, 5, 6, 7, 8, 8, 9]
# 'suspended' or right censored data
s = [1, 2, 10]
# left censored data
```

(continues on next page)

(continued from previous page)

```
l = [7, 8, 9]

# convert to xcn format!
x, c, n = surv.fsl_to_xcn(f, s, l)
print(x, c, n)

model = surv.Weibull.fit(x, c, n)
print(model)
```

```
[ 1  2  2  3  4  5  6  7  7  8  8  9  9 10] [ 1  0  1  0  0  0  0 -1  0 -1  0 -1  0 ␣
↪1] [1 1 1 1 1 1 1 1 1 1 2 1 1 1]
Parametric Surpyval model with Weibull distribution fitted by MLE yielding parameters␣
↪(6.814750943874994, 2.4708983791967163)
```

Another common type of data that is provided is in a simple text list with "+" indicating that the observation was censored at that point. Using some simple python list comprehensions can help.

```
# Example provided data
data = "1, 2, 3+, 5, 6, 8, 10, 3+, 5+"

f = [float(x) for x in data.split(',') if "+" not in x]
s = [float(x[0:-1]) for x in data.split(',') if "+" in x]

data = surv.fs_to_xcn(f, s)

model = surv.Weibull.fit(*data)
```

```
Parametric Surpyval model with Weibull distribution fitted by MLE yielding parameters␣
↪(6.737537377506333, 1.9245506420162473)
```

Again, this can be extended to left censored data as well:

```
data = "1, 2, 3+, 5, 6, 8, 10, 3+, 5+, 15-, 16-, 17-"
split_data = data.split(',')

f = [float(x) for x in split_data if ("+" not in x) & ("-" not in x)]
s = [float(x[0:-1]) for x in split_data if "+" in x]
l = [float(x[0:-1]) for x in split_data if "-" in x]

# Create the x, c, n data
data = surv.fsl_to_xcn(f, s, l)

model = surv.Weibull.fit(*data)
```

Surpyval also offers the ability to use a pandas DataFrame as an input. All you need to do is tell it which columns to look at for x, c, n, and t. Columns for c, n, and t are optional. Further, if you have interval censored data you can use the 'xl' and 'xr' column names instead. If you have mixed interval and observed or censored data, just make sure the value in the 'xl' column is the value of the observation or left or right censoring.

```
xr = [2, 4, 6, 8, 10]
xl = [1, 2, 3, 4, 5]
df = pd.DataFrame({'xl' : xl, 'xr' : xr})

model = surv.Weibull.fit_from_df(df)
print(model)
```

```
Parametric Surpyval model with Weibull distribution fitted by MLE yielding parameters␣
↪(4.694329418712716, 2.4106930022962714)
```

1.6 Datasets

1.6.1 Ball Bearing Failures

```
>>> from surpyval.datasets import Bearing

>>> Bearing.df.info()
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 23 entries, 0 to 22
Data columns (total 1 columns):
 #   Column                        Non-Null Count  Dtype
---  ------                        --------------  -----
 0   Cycles to Failure (millions)  23 non-null     float64
dtypes: float64(1)
memory usage: 312.0 bytes
```

	Cycles to Failure (millions)
0	17.88
1	28.92
2	33
3	41.52
4	42.12

1.7 Non-Parametric Estimation

Non-parametric survival analysis is the attempt to capture the distribution of survival data without making any assumptions about the shape of the distribution. That is, non-parametric analysis, unlike parametric analysis, does not assume that the survival data was Weibull distributed or that it was Normally distributed etc. Concretely, non-parametric estimation does not attempt to estimate the parameters of a distribution, therefore "non-parametric." Parametric analysis is covered in more detail in the section covering parametric estimation but it is important to contrast non-parametric estimation against what it is not. So what exactly is non-parametric analysis?

Survival analysis is using statistics to answer the question 'what is the probability that the thing survived to a particular time?' Non-parametric analysis answers this by estimating the probability from the proportion failed upto a given time. This can be done by either estimating the probability of surviving a particular segment or estimating the hazard rate.

Non-parametric estimation is well understood by appreciating the data format used to estimate the CDF. Specifically, the 'xrd' format and particularly understanding the r and d sets of that format.

The number of components at risk, r, at a given time, x, is the number of things at risk just prior to time x. The number of deaths, d, is the number of the at risk items that died (or failed) at time x. So for completely observed data the number at risk counts down for every death. So r would count down, e.g. 6, 5, 4, 3... for each death, 1, 1, 1, 1, ... So in this example there were 6 items at risk at one death at the first time. Then, because there was 1 death at the first time the number of items at risk has decreased to 5, therefore for the next death there are only 5 at risk. This continues further until there are no more items at risk because they have all died, i.e. there is 1 at risk and 1 death.

This can be extended to more than one death. For example, the risk set could be 8, 6, 5, 3, 2, 1. with an accompanying death set of 2, 1, 2, 1, 1, 1. In this example there were times where there were 2 deaths and therefore the number at risk decreased by 2 after that number of deaths.

So a complete example of this format is:

```
x = [1, 2, 3, 4, 5, 6]
r = [7, 5, 4, 3, 2, 1]
d = [2, 1, 1, 1, 1, 1]
```

This format for data is not how survival data is usually provided in text books or papers. Survival data is usually displayed with the simple list of failure times such as “1, 3, 6, 7, 10, 16”. The first step surpyval does for non-parametric analysis is to transform data into the xrd format. All the `fit()` methods for surpyval take as input the xcnt format, see more at the data types docs. So if you provide surpyval with the data “1, 2, 3, 4, 5, 6” it will assume that each of them are one death, and then create the risk set from the death counts resulting in the xrd format from above.

Given we now understand the format of the data we can estimate the probability of survival to some time with non-parametric methods. The first method we will visit is the Kaplan-Meier.

1.7.1 Kaplan-Meier Estimation

Kaplan-Meier [KM] is a very popular method for estimating survival curves for populations. The insight for this method is that for each time there is a death, we can estimate the probability of having survived since the previous deaths. Using the data from above as an example, at time 1, there are 7 items at risk and there are 2 deaths. We can therefore say that the probability of surviving this period was (7 - 2)/7, i.e. 5/7. Then the next time there is a death, the probability of having survived that extra time is (5 - 1)/5, i.e. 4/5.

To be clear, this is the chance of survival between each death. Therefore the chance of surviving up to a given time is the chance of surviving each segment. Therefore the probability of surviving up to any given time is the probability of surviving through all the previous segments. The probability of surviving multiple outcomes is the multiplication of each of the survival probabilities. Surviving through three sections is equal to the probability that I survive the first, then multiply this by the probability of surviving the second, then multiplying this result with the probability of surviving the third. So continuing our example from above, the probability of surviving the first two segments is (5/7) x (4/5) = 4/7.

Therefore using the at risk count, r, and the death count, d, can be used to estimate the segment survival probabilities and the survival probability to any point can be found by multiplying these probabilities. Formally, this has the following formula:

$$R(x) = \prod_{i:x_i \leq x} \left(1 - \frac{d_i}{r_i}\right)$$

1.7.2 Nelson-Aalen Estimation

The Nelson-Aalen estimator [NA] (also known as the Breslow estimator), instead of finding the probability, estimates the cumulative hazard function, and given that we know the relationship between the cumulative hazard function and the reliability function, the Nelson-Aalen cumulative hazard estimate can be converted to a survival curve.

The first step in computing the NA estimate is to convert your data to the x, r, d format. Once in this format the instantaneous hazard rate is found by:

$$h(x) = \frac{d_x}{r_x}$$

This estimate of the instantaneous hazard rate is the proportion of deaths/failures at a value, x. Then to find the cumulative hazard rate for any x we simply take the sum of the instantaneous hazard rates for all the values below x.

Mathematically:

$$H(x) = \sum_{i:x_i \leq x} \frac{d_i}{r_i}$$

Then, since we know that the reliability, or survival function, is related to the cumulative hazard function, we can easily compute it.

$$R(x) = e^{-H(x)}$$

So we now have the survival/reliability function. One benefit of the Nelson-Aalen estimator is that it does not estimate a probability of 0 for the highest value (in a completely observed data set). This means that for a completely observed data set the whole estimation can be plotted on a transformed y-axis. For this reason SurPyval uses the Nelson-Aalen as the default plotting position.

1.7.3 Fleming-Harrington Estimation

The Fleming-Harrington estimator [FH], uses the same principal as the Nelson-Aalen estimator. That is, it finds the cumulative hazard function and then converts that to the reliability/survival estimate. However, the NA estimate assumes, for any given step that the number of items at risk is equal for each death, the FH estimate changes this. Mathematically, the hazard rate is calculated with:

$$h(x) = \frac{1}{r_x} + \frac{1}{r_x - 1} + \frac{1}{r_x - 2} + ... + \frac{1}{r_x - (d_x - 1)}$$

Which can be summarised as:

$$h(x) = \sum_{i=0}^{d_x - 1} \frac{1}{r_x - i}$$

The cumulative hazard rate therefore becomes:

$$H(x) = \sum_{i:x_i \leq x} \sum_{i=0}^{d_x - 1} \frac{1}{r_x - i}$$

You can see that the cumulative hazard rate will be slightly higher than the NA estimate since:

$$\frac{1}{r_x} + ... + \frac{1}{r_x} \leq \frac{1}{r_x} + ... + \frac{1}{r_x - (d_x - 1)}$$

The above is less than or equal for the case where there is one death/failure. The Fleming-Harrington and Nelson-Aalen estimates are particularly useful for small samples, see [FH].

1.7.4 Turnbull Estimation

The Turnbull estimator is a remarkable non-parametric estimation method for data that can handle arbitrary censoring and truncation [TB]. The Turnbull estimator can be found with a procedure of finding the most likely survival curve from the data, for that reason it is also known as the Non-Parametric Maximum Likelihood Estimator. The Kaplan-Meier is also known as the Maximum Likelihood estimator, so is there a contradiciton? No, the Turnbull estimator is the same as the Kaplan-Meier for fully observed data.

The Turnbull estimate is really an estimate of the observed failures given censoring, and then the ‘ghost’ failures (as Turnbull describes it) due to truncation. Turnbull’s estimate converts all failures to interval failures regardless of the censoring. This is because a left censored point is equivalent to an intervally censored observation in the interval -Inf to x, and a right censored point is equivalent to an intervally censored observation in the interval x to Inf. Then for all the intervals between negative infinity we find how many failures happened in that interval. This value need not be a whole number since a single observation could have failed across several intervals. To estimate the failures, we use:

$$\mu_{ij}(s) = \frac{\alpha_{ij} s_j}{\sum_{k=1}^{m} \alpha_{ik} s_k}$$

Where μ_{ij} is the probability of the i-th observation failing in the j-th interval, α_{ij} is a flag to indicate if the i-th failure was at risk in interval j, (1 if at risk and 0 if not), and s_j is the probability of failure in an interval. That is, s_j is the survival function we are trying to estimate.

If an observation is truncated, it was only a possible observation among others that would have been seen had the observation not been limited. To estimate the additional at risk items outside of the domain for which an observation is truncated we use:

$$\nu_{ij}(s) = \frac{(1 - \beta_{ij}) s_j}{\sum_{k=1}^{m} \alpha_{ik} s_k}$$

Where ν_{ij} is the probability of the i-th observation failing in the j-th interval and β_{ij} is a flag to indicate if the i-th failure was observable in interval j, (1 if at risk and 0 if not).

This formula then finds the number of failures outside the truncated interval for a given observation.

We can then estimate the probability of failure in a given interval using the total failures in each interval divided by the total number of failures:

$$s_j = \frac{\sum_{i=1}^{N} \mu_{ij} + \nu_{ij}}{M(s)}$$

where

$$M(s) = \frac{\sum_{i=1}^{N} \sum_{j=1}^{m} \mu_{ij} + \nu_{ij}}{M(s)}$$

Using this estimate of the survival function, it can be input to the start of this procedure and it done again. This can then be repeated over and over until the values do not change. At this point we have reached the NPMLE estimate of the survival function!

The Turnbull estimation is the only non-parametric method that can be used with truncated and left censored data. Therefore it must be used when using the plotting methods in the parametric package when you have truncated or left censored data.

1.7.5 References

1.8 Parametric Estimation

Parametric modelling is the process of estimating the parameters of a particular distribution from a set of data. This is distinct from non-parametric modelling where we make no assumptions about the shape of the distribution. In parametric modelling we make some assumptions, explicit or implied, about the shape of the data that we have.

For this segment I will use the Weibull distribution as the example distribution. The Weibull distribution is a very useful distribution for one interesting reason. It is the distribution for the 'weakest link.' As the normal distribution is the limiting distribution of averages, the Weibull distribution is the limiting distribution for minimums. What does that mean? If we have a large number of sets of samples from something that is normally distributed the average of these sets will also be normally distributed but the minimums of these sets of samples will be Weibull distributed. This is analagous to a chain. It is common wisdom that a chain is only as strong as it's weakest link. The Weibull distribution enables us to model the strength of a chain based on the strength of the links.

The Weibull distribution can then be used in scenarios where we assume that the shape of the distribution will be due to a weakest link effect. This assumption holds in many scenarios, the strength of materials, the fielded life of equipment, the lifetime of animals, the time until another recession, or the time until germination of seeds. This example makes clear the assumption that we can make when using the Weibull distribution. Other distributions have differing processes that can result in their generation. If we know and understand these processes we can check them against the scenario we are analysing and chose a distribution from them. For example, a lognormal distribution can arise due to the combined effect of the product of random variables so in petroleum engineering the total recoverable oil is a product of the height, width, depth, features of the rock and an infinitude of other variables of the field. Therefore fields can be lognormally distributed. Similar considerations can be applied for many other types of distributions. Finally, If we don't know, or mind, what distribution we have, we can simply find the best fit amongst a set of distributions.

SurPyval offers users several methods for estimating parameters, these are:

- Method of Moments (MOM)
- Method of Probability Plotting (MPP)
- Mean Square Error (MSE)
- Maximum Likelihood Estimation (MLE)
- Minimum Product Spacing (MPS)

There are other methods that can be used, e.g. L-moments or generalised method of moments. These are interesting, and may be added in future, but for now surpyval offers the above estimation methods. Surpyval is unique in the capability to provide the estimation technique. Most other survival analysis methods do not allow for specifying different methods. The advantage of this flexibility will become apparent.

1.8.1 Method of Moments (MOM)

This method is the simplest (and least accurate) method to find parameters of a distribution. The intent of the Method of Moments (MOM) is to find the closest match of a distributions moments, to those of the moments of a sample of data.

For a given data set, or sample, the kth moment is defined as:

$$M_k = \frac{1}{n}\sum_{i=1}^{n} X_i^k$$

If the distribution has only one parameter, like the exponential distribution, then the method of moments is simply equates the sample moment to the dsitribution moment. For a continuous distribution the kth moment is defined as:

$$M_k = \int_{-\infty}^{\infty} x^k f(x)dx$$

Where f(x) is the density function of that distribution. Therefore, for the exponential distribution, the moments can be computed (with some working) to be:

$$E[X^k] = \frac{k!}{\lambda^k}$$

Because there is only one parameter of the exponential distribution, we need to only match the first moment of the distribution (k=1) to the first moment of the sample. Therefore we get:

$$\frac{1}{n}\sum_{i=1}^{n} X_i = \frac{1}{\lambda}$$

This is to say that the method of moments solution for the parameter of the exponential is simply the inverse of the average. This is an easy result. When we extend to other distributions with more than one parameter, such simple analytical solutions are not available, so numeric optimisation is needed. SurPyval uses numeric optimisation to compute the parameters for these distributions.

The method of moments, although interesting, can produce incorrect results, and it can only be used with observed data, so it cannot account for truncation or censoring. But it is good to understand as it is one of the oldest methods used to estimate the parameters of a distribution.

1.8.2 Method of Probability Plotting (MPP)

Probability plotting is an extremely simple way to find the parameters of a distribution. This method has a long history because it is a simple activity to do while providing an easy to understand graphic. Further, probability plotting produces a good estimate for the parameters even with few data points. All this combined with the fact that probability plotting can be used for all types of data, observed, censored, and truncated, it is easy to understand why it is widely used.

SurPyval uses the MPP method as an initial guess, when not provided, because it is the only method that does not require an initial guess of the parameters. This is because numeric optimisers require an initial guess, however, when using a probability plotting method, an initial guess is not needed. It therefore provides an excellent method to get an initial guess for subsequent optimisation. But the method itself can be sufficient enough for the majority of applications.

So how does it work?

Probability plotting works of the idea that a distributions CDF can be made into a straight line if the data is transformed. This can be shown by rearranging the CDF of a dsitribution. For the Weibull:

$$F(x) = 1 - e^{-(\frac{x}{\alpha})^{\beta}}$$

If we negate, add one, and then take the log of each side we get:

$$\ln(1 - F(x)) = -(\frac{x}{\alpha})^{\beta}$$

Then take the log again:

$$\ln(-\ln(1 - F(x))) = \beta \ln(x) - \beta \ln(\alpha)$$

From here, we can see that there is a relationship between the CDF and x. That is, the log of the log of (1 - CDF) has a linear relationship with the log of x. Therefore, if we take the log of x, and take the log of the negative log of 1 minus the CDF and plot these, we will get a straight line. To make this work, we therefore need a method to estimate the CDF empirically. Traditionally, there have been heuristics used to create the CDF. However, we can also use the non-parametric estimate as discussed in the non-parametric session. Concretely, we can use the Kaplan-Meier, the Nelson-Aalen, Fleming-Harrington, or Turnbull estimates to approximate the CDF, F(x), transform it, plot, and then do the linear regression. SurPyval uses as a default, the Nelson-Aalen estimator for the plotting point.

Other methods are available. The simplest estimate, for complete data, is the empirical CDF:

$$\hat{F}(x) = \frac{1}{n} \sum_{i=1}^{n} 1_{X_i \leq x}$$

This equation says, that (for a fully observed data set) for any given value, x, the estimate of the CDF at that value is simply the sum of all the observations that occurred below that value divided by the total number of observations. This is a simple percentage estimate that has failed at any given point. This equation will therefore make a step function that increases from 0 to 1.

One issues with this is that the highest value is always 1. But if this is transformed as above, this will be an undefined number. As such, you can adjust the value with a simple change:

$$\hat{F}(x) = \frac{1}{n+1} \sum_{i=1}^{n} 1_{X_i \leq x}$$

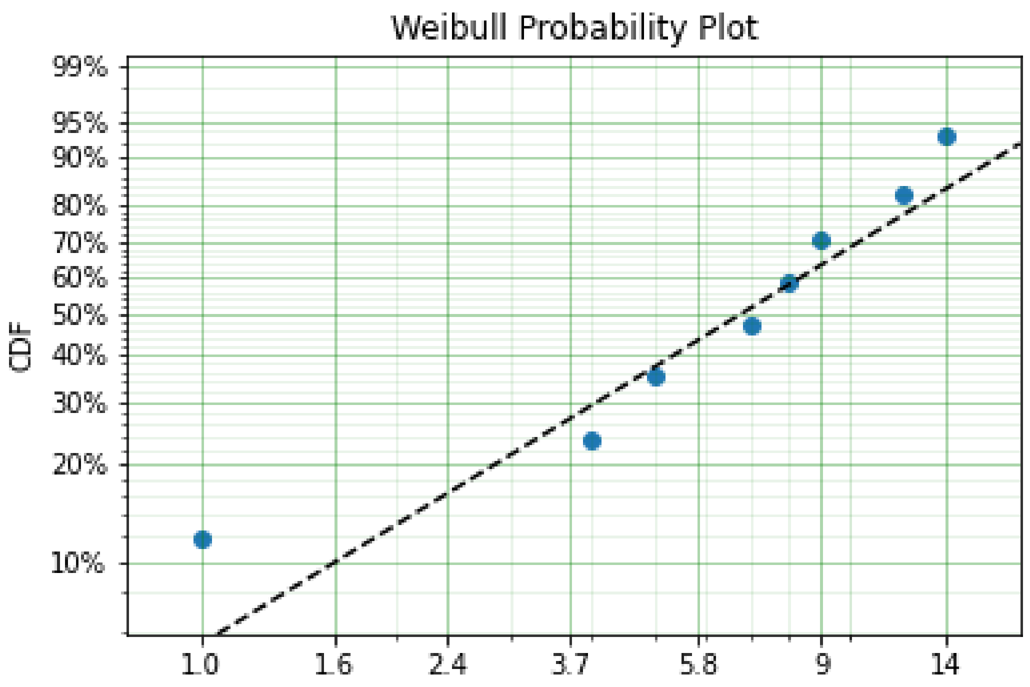
By using this simple change, the highest value will not be 1, and will therefore be plottable, and not undefined. There are many different methods used to adjust the simple ECDF to be used with a plotting method to estimate the parameters of a distribution. For example, consider Blom's method:

$$\hat{F}_k = (k - 0.375)/(n + 0.25)$$

Where k is the rank of an observation k is in (1, 2, 3, 4.... n) for n observations. Using these methods we can therefore plot the linearised version above.

Combining this all together is simple witht surpyval.

```
x = [1, 4, 5, 7, 8, 9, 12, 14]
model = surv.Weibull.fit(x, how='MPP', heuristic='Blom')
model.plot()
```



In this example we have used the probability plotting method with the Blom heuristic to estimate the parameters of the distribution. SurPyval has the option to use many different plotting methods, including the regular KM, NA, and FH non-parametric estimates. All you need to do is change the 'heuristic' parameter; SurPyval includes:

Table 1: SurPyval Modelling Methods

Method	A	B
Blom	0.375	0.25
Median	0.3	0.4
ECDF	0	0
ECDF_Adj	0	1
Mean	0	1
Weibull	0	1
Modal	1	-1
DPW	1	0
Midpoint	0.5	0
Benard	0.3	0.2
Beard	0.31	0.38
Hazen	0.5	0
Gringorten	0.44	0.12
Larsen	0.567	-0.134
Larsen	1/3	1/3
None	0	0

Which is used with the general formula to estimate the plotting position heuristic:

$$\hat{F}_k = (k - A)/(n + B)$$

One final option available is that of the Filliben estimate:

1.8.3 Mean Square Error (MSE)

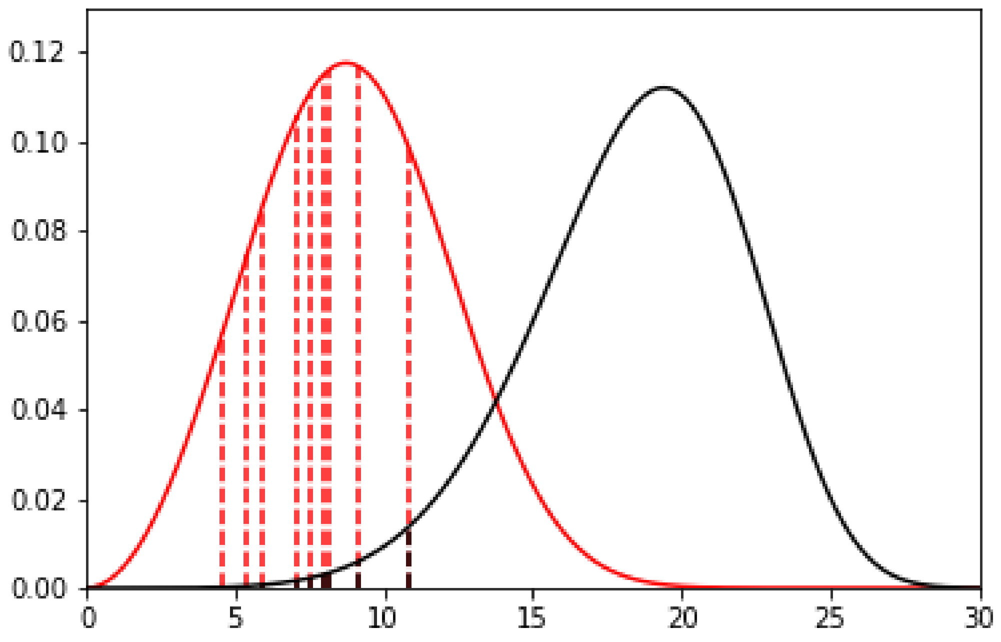
MSE is essentially the same as probability plotting. Instead of finding the minimum against the transformed data in the x and y axes. The parameters are found by minimising the distance to the non-parametric estimate without transforming the data to be linear. Mathematically, MSE find the parameters by minimising:

$$\Sigma \left(\hat{F} - F(x;\theta) \right)^2$$

This is the difference between the, untransformed, empirical estimate of the CDF and the parametric distribution.

1.8.4 Maximum Likelihood Estimation (MLE)

Maximum Likelihood Estimation (MLE) is the most widely used, and most flexible of all the estimation methods. It's relative simplicity (because of modern computing power) makes it the reasonable first choice for parametric estimation. What does it do? Essentially MLE asks what parameters of a distribution are 'most likely' given the data that we have seen. Consider the following data and distributions:



The solid lines are the densities of two different Weibull distributions. The dashed lines represent the data we have observed, their height is the density of the two distributions at the x value for each observation. Given the data and the two distributions, which one seems to explain the distribution of the data better? That is, which distribution is more likely to produce, if sampled, the dashed lines? It should be fairly intuitive that the red distribution is more likely to do so. For example, the observation just above 10, you can see the height to the black line and the heigth to the red line. The red line is taller than the black line, therefore this observation is more 'likely' to have come from the red distribution than the black one. Conversely, the value near 15 is more likely to have come from the black distribution than the red one because the height to the black line is greater than the height to the red line. To find the distribution of best fit then we need to find the parameters that best averages the height of all these lines.

MLE formalises this concept by saying that the most likely distribution is the one that has the highest (geometric) mean of the height of the density function for each sample of data. The height of the density at a particular observation is known as the likelihood. Mathematically, (for uncensored data) MLE then maximises the following:

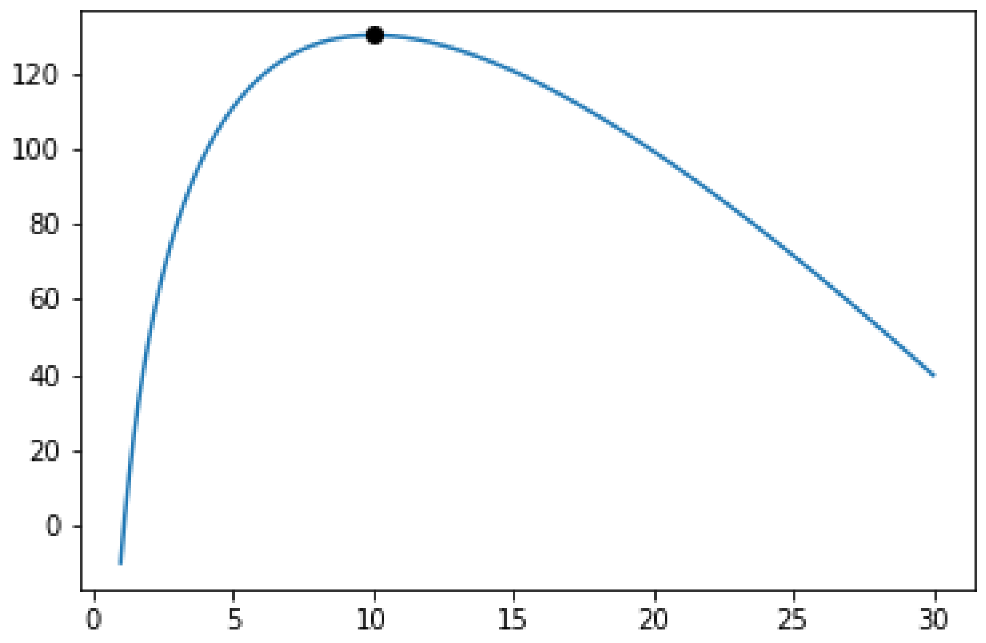
$$L = \left(\prod_{i=1}^{n} f(x_i|\theta) \right)^{1/n}$$

f is the pdf of the distribution being estimated, x is the observed value, theta is the parameter vector, and L is the geometric mean of all the values. This is complicated, but a simplification is available by taking the log of this product yielding:

$$l = \frac{1}{n} \sum_{i=1}^{n} \ln f(x_i|\theta)$$

Therefore MLE simply finds the parameters of the distribution that maximise the average of the log of the likelihood for each point… One final transform that is used in optimisers is that we take the negative of the above equation so that we find the minimum of the negative log-likelihood.

Armed with the log likelihood we can then search for the parameter where the log likelihood is maximised. Using an Exponential distribution as an example, we can see the change in the value of the log likelihood as the exponential parameter changes. The following is a random sample of 100 observations with a parameter of 10. Then changing the value of the parameter 'lambda' from low to high we can see what the log-likelihood is and find the value at which it is maximized.



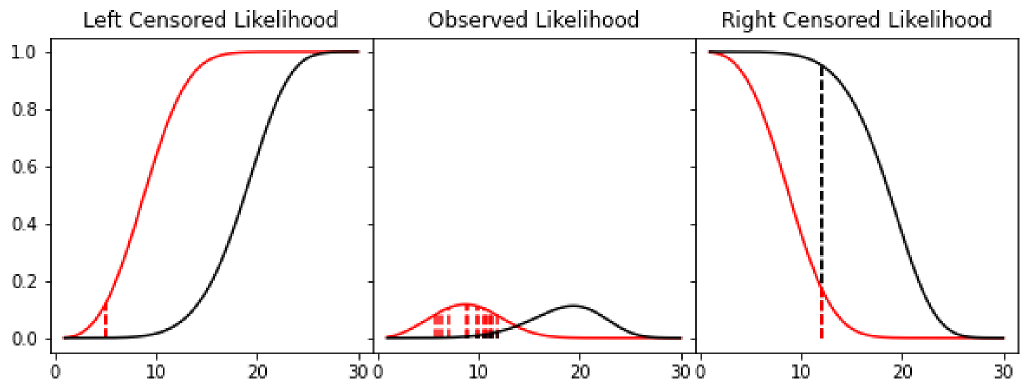
On the chart above you can see that the maximum is near 10. As we would expect given that we know that the answer is 10. It is this simple and intuitive approach that allows the parameters of distributions are estimated with the MLE.

What about censored data?

All the equations above are for observed data. Handling the likelihood of censored data also has an intuitive understanding. What we know about the point when the data point is censored is that we know it is above or below the value at which we observed. So for a right censored data point, we want to maximize the probability that we saw this observation, concretely we want a censored points contribution to the likelihood function is the probability that the point was left or right censored. This is simply the probability of failure (CDF) for left censored and the probability of surviving to that time (survival function). Formally:

$$l = \frac{1}{n_o}\sum_{i=1}^{n_o} \ln f(x_{o_i}|\theta) + \frac{1}{n_r}\sum_{i=1}^{n_r} \ln R(x_{r_i}|\theta) + \frac{1}{n_l}\sum_{i=1}^{n_l} \ln F(x_{l_i}|\theta)$$

An easy and intuitive way to understand this is to compare these two possibilities. With some randomly generated data with a few values made to be left censored, and a few to be right censored. We get:



In this example, again, we need to consider whether the red or black distribution is a more likely description of the observations, including some censored ones. Althought the right censored point for the black distribuiton is very likely, this does not mean it is a good fit because the 'average' across all observations is poor. Therefore, it should be obvious that the red distribution is the better fit.

But what about truncated data

1.8.5 Maximum Product of Spacings (MPS)

Coming soon

1.9 Regression Analysis

Regression analysis is the process of capturing the effect that other factors, i.e. covariates, will have on the survival probability. That is, we use data on other factors to 'regress' onto the survival distribution.

SurPyval

1.9.1 Accelerated Life

1.10 Non-Parametric SurPyval Modelling

To get started, let's import some useful packages, as such, for the rest of this page we will assume the following imports have occurred:

```python
import surpyval as surv
import numpy as np
from matplotlib import pyplot as plt
```

Survival modelling with *surpyval* is very easy. This page will take you through a series of scenarios that can show you how to use the features of *surpyval* to get you the answers you need. The first example is if you simply have a list of event times and need to find the distribution of best fit.

In each of the examples below, each of the `KaplanMeier`, `NelsonAalen`, or `FlemingHarrington` can be substituted with any of the others. It is the choice of the analyst which should be used. The `Turnbull` estimator has additional capabilities that can be used when you have right truncated, left censored, or interval censored data.

1.10.1 Complete Data

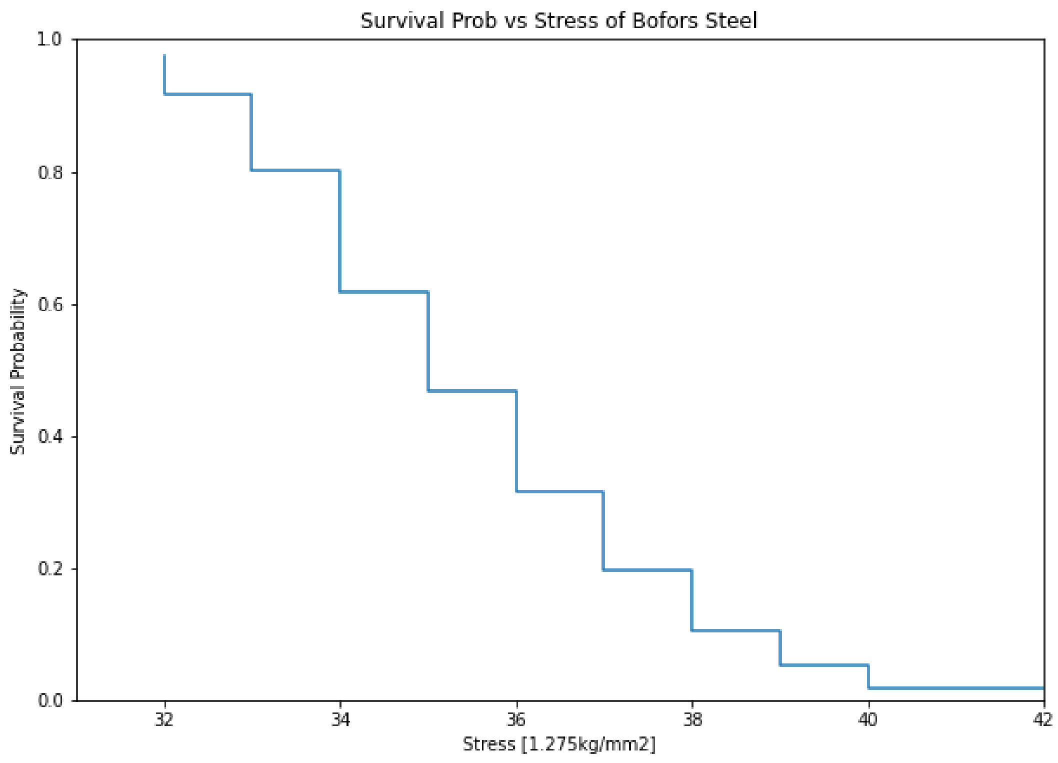
Using data of the stress of Bofors steel from Weibull's original paper we can esimtate the reliability, that is, the probability that a sample of steel will survive up to a given applied stress. So what does that mean?

We can find when the steel will break. This is particularly useful when we know the application.

For this example, lets say that the maximum tensile stress our design will see during use is 34 units. Lets try and estimate the proportion that will fail during operation.

For this we can use the Nelson-Aalen estimator of the hazard rate, then convert it to the reliability. This is all done with one easy call.

```python
import surpyval as surv
import numpy as np
from matplotlib import pyplot as plt

x = np.array([32, 33, 34, 35, 36, 37, 38, 39, 40, 42])
n = np.array([10, 33, 81, 161, 224, 289, 336, 369, 383, 389])

# Weibull's measurements are cumulative so we need to tranasform them
n = np.concatenate([[n[0]], np.diff(n)])

bofors_steel_na = surv.NelsonAalen.fit(x, n=n)

plt.figure(figsize=(10, 7));
plt.ylabel('Survival Probability')
plt.xlabel('Stress [1.275kg/mm2]')
plt.ylim([0, 1])
plt.xlim([31, 42])
plt.step(bofors_steel_na.x, bofors_steel_na.R)
plt.title('Survival Prob vs Stress of Bofors Steel');
```



So what purpose is this?

With our non-parametric model of the Bofors steel. We can use this model to estimate the reliability in our application. Let's say that our application uses Bofors steel up to 34. What is our estimate of the number of failures?

```
print(str(bofors_steel_na.sf(34).round(4).item() * 100) + "%")
```

Which gives:

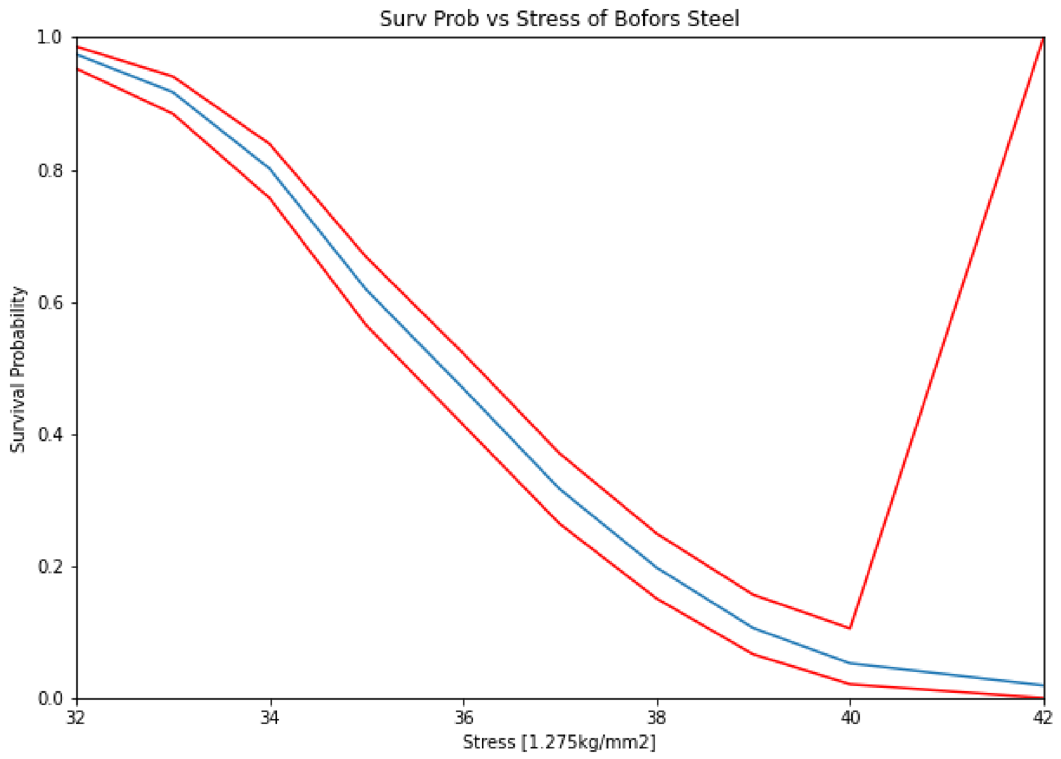
```
80.15%
```

The above shows that approximately 80% will survive up to a stress of 34. Therefore we will have an approximately 20% chance of our component failing in the design.

It is up to the designer to determine whether this is acceptable.

What if we want to take into account our uncertainty about the reliability. The non-parametric class automatically computes the Greenwood variance and uses that to compute the upper and lower confidence intervals. Let's plot the intervals to see.

```
plt.figure(figsize=(10, 7))
bofors_steel_na.plot(how='interp')
plt.xlabel('Stress [1.275kg/mm2]')
plt.ylabel('Survival Probability')
plt.ylim([0, 1])
plt.xlim([32, 42])
plt.title('Surv Prob vs Stress of Bofors Steel')
```



The confidence bounds can also be used to estimate the probability of survival up to some point with some degree of confidence. For example:

```
print(str(bofors_steel_na.R_cb(34, bound='lower', how='interp', confidence=0.95).
↪round(4).item() * 100) + "%")
```

```
76.46%
```

Therefore we can be 95% confident that the reliability at 34 is above 76%. You can also see that the confidence interval stretches the entire span of the possible [0, 1] interval at the higest value. This is because the variance at the final value is infinite using the Greenwood confidence interval.

1.10.2 Right Censored Data

Non-Parametric estimation can handle right censored, this is possible because at the point of censoring the item is removed from the at risk group without couting a death/failure.

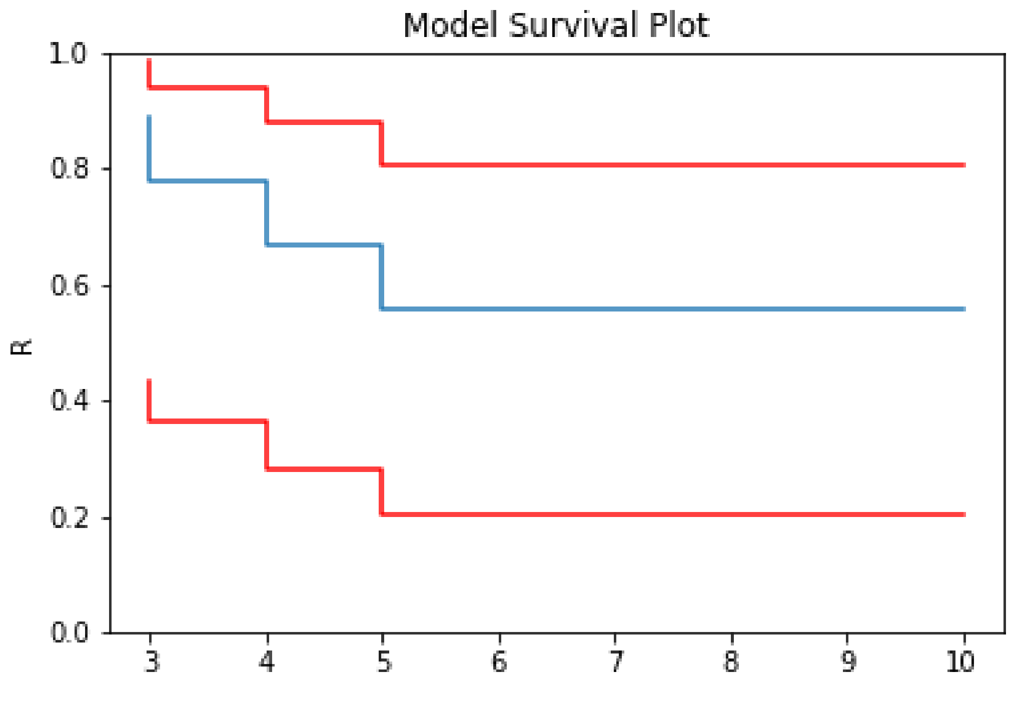
```
import numpy as np
from surpyval import KaplanMeier as KM

x = np.array([3, 4, 5, 6, 10])
c = np.array([0, 0, 0, 0, 1])
n = np.array([1, 1, 1, 1, 5])

model = KM.fit(x=x, c=c, n=n)
```

(continues on next page)

(continued from previous page)

```
model.plot()
model.R
```

```
array([0.88888889, 0.77777778, 0.66666667, 0.55555556, 0.55555556])
```

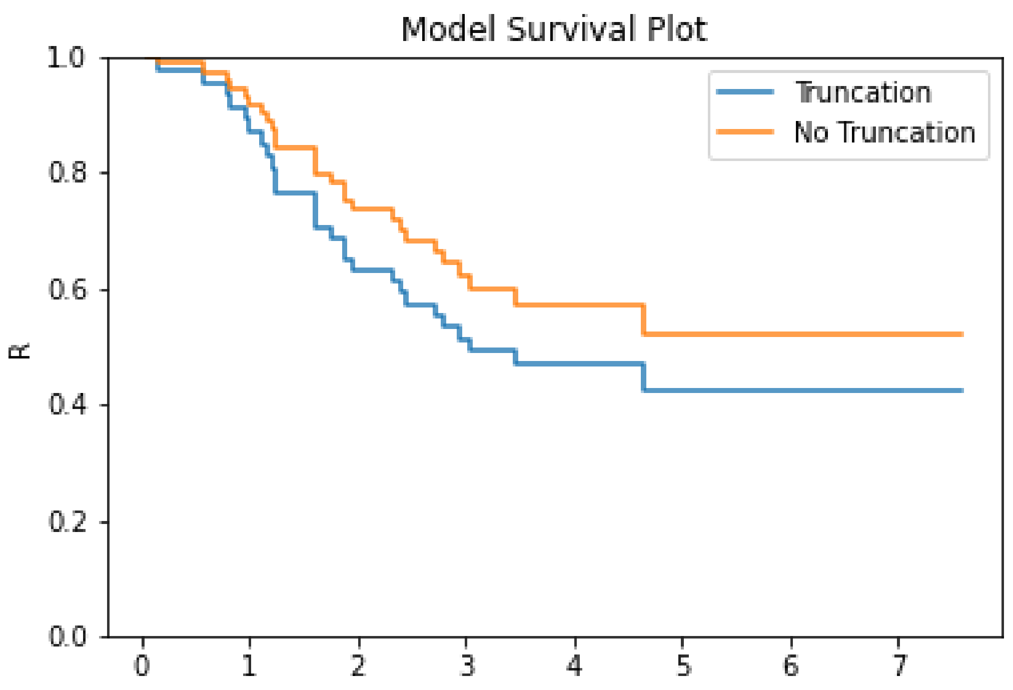


In this example, we have included right censored data. This example can be done for the Nelson-Aalen, Fleming-Harrington, and Turnbull estimators as well.

1.10.3 Left Truncated Data

In some instances you will need to account for left truncated data. These data can be passed stright to the same KM, NA, and FH fitters, Using one (of the many) excellent data sets from the lifelines. package:

```
from surpyval import KaplanMeier as KM
from lifelines.datasets import load_multicenter_aids_cohort_study
df = load_multicenter_aids_cohort_study()

x = df["T"].values
c = 1. - df["D"].values
tl = df["W"].values

model = KM.fit(x=x, c=c, tl=tl)
model_no_trunc = KM.fit(x=x, c=c)

model.plot(plot_bounds=False)
model_no_trunc.plot(plot_bounds=False)
plt.legend(['Truncation', 'No Truncation'])
```

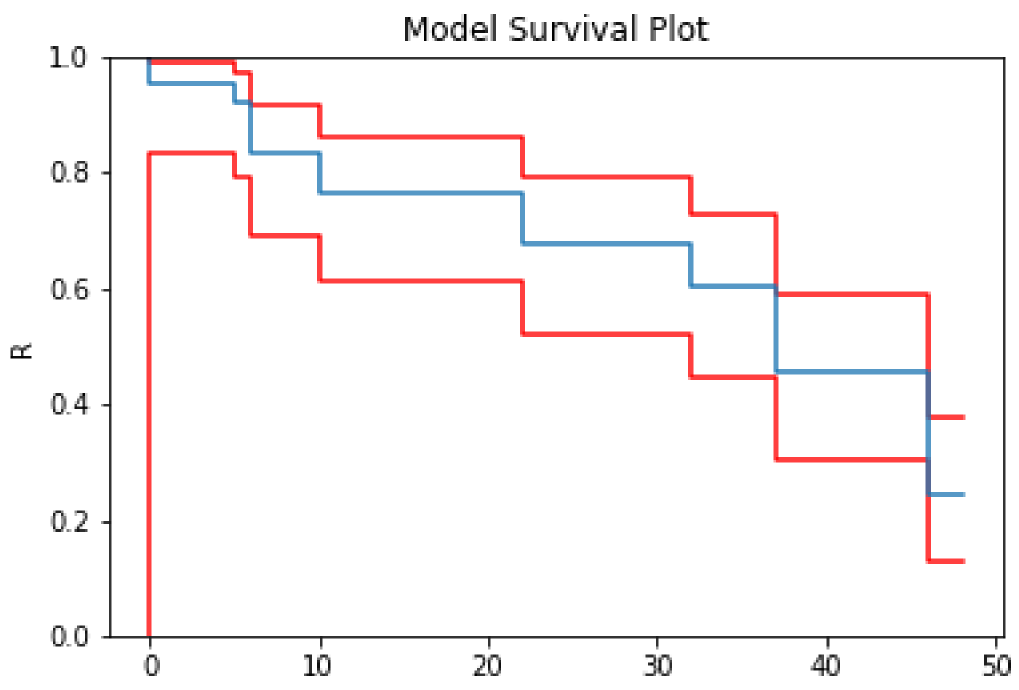


The image above shows that if you fail to take into account the left truncation (using the `tl` keyword) you will overstate the survival probability. This can be used with any of the other non-parametric fitters.

1.10.4 Arbitrarily Truncated and Censored Data

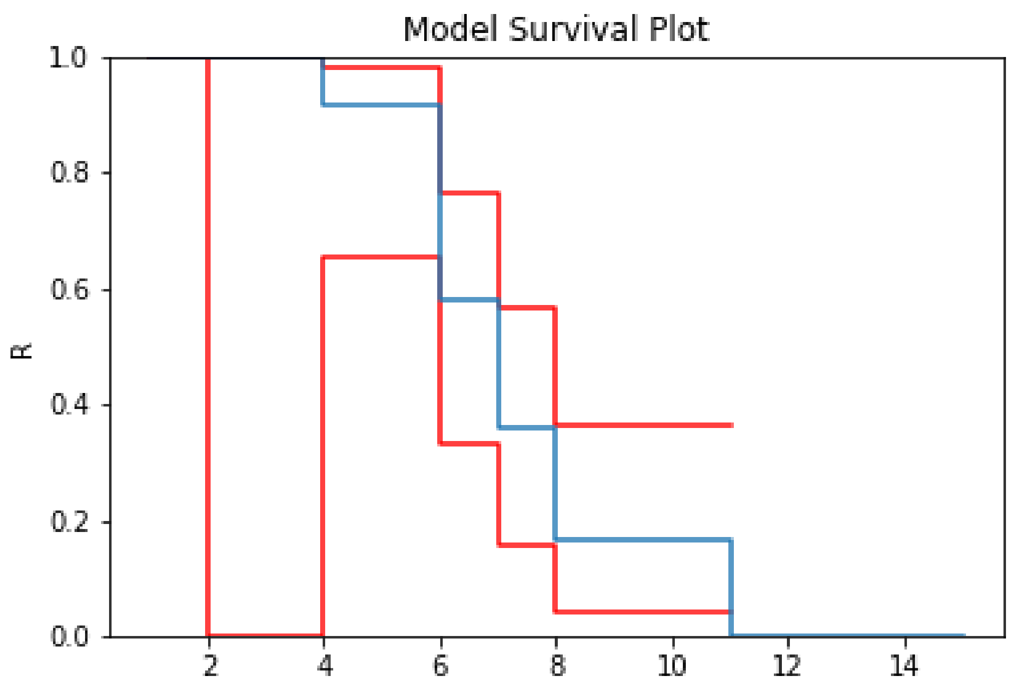
In the event you have data that has interval, left, or right censoring with no, left, or right truncation, the previous estimators will not work. Enter the `Turnbull` estimator. First an interval estimation example:

```
low = np.array([0, 0, 0, 4, 5, 5, 6, 7, 7, 11, 11, 15, 17, 17,
                17, 18, 19, 18, 22, 24, 24, 25, 26, 27, 32, 33,
                34, 36, 36, 36, 36, 37, 37, 37, 37, 38, 40, 45,
                46, 46, 46, 46, 46, 46, 46, 46])
upp = np.array([7, 8, 5, 11, 12, 11, 10, 16, 14, 15, 18, np.inf,
                np.inf, 25, 25, np.inf, 35, 26, np.inf, np.inf,
                np.inf, 37, 40, 34, np.inf, np.inf, np.inf, 44,
                48, np.inf, np.inf, 44, np.inf, np.inf, np.inf,
                np.inf, np.inf, np.inf, np.inf, np.inf, np.inf,
                np.inf, np.inf, np.inf, np.inf, np.inf])

x = np.array([low, upp]).T
model = TB.fit(x)
model.plot()
```



And finally, an example with arbitrary censoring and truncation:

```
from surpyval import Turnbull as TB

x = [1, 2, [3, 6], 7, 8, 9, [5, 9], [4, 10], [7, 10], 11, 12]
c = [1, 1, 2, 0, 0, 0, 2, 2, 2, -1, 0]
n = [1, 2, 1, 3, 2, 2, 1, 1, 2, 1, 1]
tl = [0, 0, 0, 0, 0, 2, 3, 3, 1, 1, 5]
tr = [np.inf, np.inf, 10, 10, 10, 10, np.inf, np.inf, np.inf, 15, 15]


model = TB.fit(x=x, c=c, n=n, tl=tl, tr=tr)
model.plot()
```



With a completely arbitrary set of data we have created a non-parametric estimate of the survival curve that can be used to estimate probabilities.

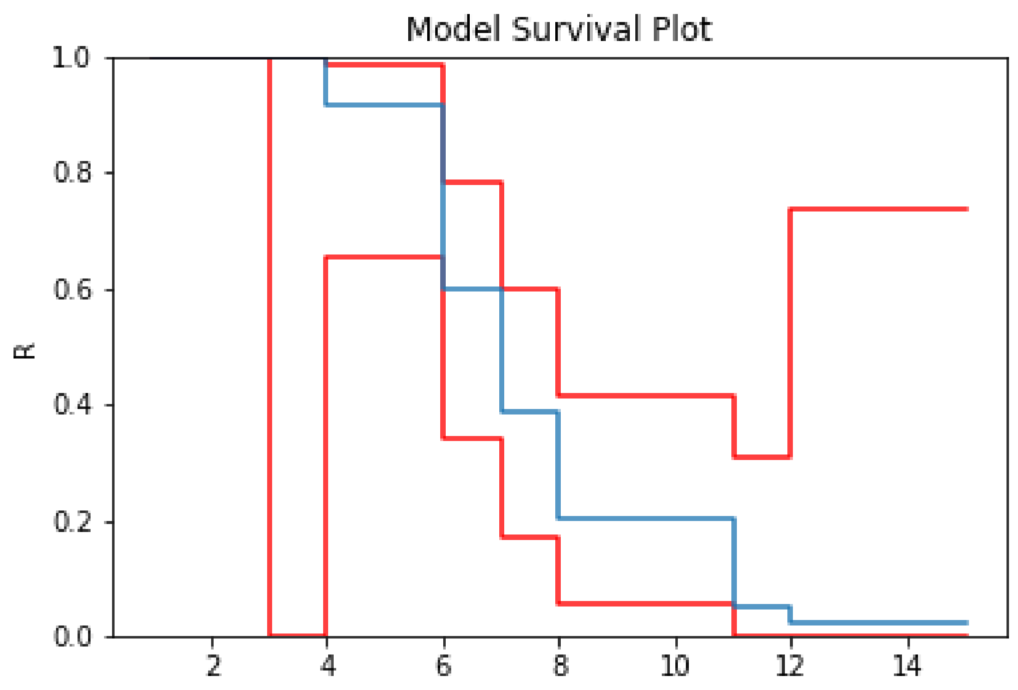
What is interesting about the Turbull estimate is that it first finds the data in the 'xrd' format. This is done even though we might not have a complete failure occur in an interval. This can be seen by looking at the number of deaths/failures occur at each value.

```
model.d
```

```
array([0.00000000e+00, 0.00000000e+00, 0.00000000e+00, 2.76875496e-02,
       1.58808369e+00, 0.00000000e+00, 5.81471061e+00, 4.10951885e+00,
       3.54383160e+00, 7.67984832e-02, 3.93153047e-15, 3.09598691e+00,
       1.66794197e+00])
```

You can see that some values are 0 (or essentially 0) or that there is an interval where there were 4.1095188 failures. But because the Turbull estimate finds the x, r, d format we can actually elect to use the Nelson-Aalen or Kaplan-Meier estimate with the Turnbull estimates of x, r, and d.

```
model = TB.fit(x=x, c=c, n=n, tl=tl, tr=tr, estimator='Nelson-Aalen')
model.plot()
```



The Greenwood confidence intervals do give us a strange set of bounds. But you can see that using the Nelson-Aalen estimator instead of the Kaplan-Meier gives us a better approximation for the tail end of the distribution.

Some Issues with the Turnbull Estimate

Caution must be given when using the Turnbull estimate when all values are truncated by some left and/or right value. This will be shown below in the methods for estimating parameters with truncated values. But essentially the Turnbull method cannot make any assumptions about the probability by which the smallest value if left truncated should be adjusted. This is because there is no information available with the non-parametric method below this smallest value. The same is true for the largest value if it is also right truncated, there is no information available about the probability of its observation. Therefore the Turnbull method makes an implicit assumption that the first value, if left truncated has 100% chance of observation, and the highest value, if right truncated also has 100% chance of being observed.

The implications of this are detailed in the Parametric section, because the only way to gain an understanding of these situations is by assuming a shape of the distribution. That is, by doing parametric analysis. This is possible since if the distribution within the truncated ends has a shape that matches to a particular distribution you can then extrapolate beyond the observed values. Parametric analysis is therefore incredibly powerful for prediction / extrapolation.

1.10.5 Confidence Intervals

Coming soon

1.11 Parametric SurPyval Modelling

The parametric API is essentially the exact same as the non-parametric API. All models are fit by a call to the `fit()` method. However, the parametric models have more options that are only applicable to parametric modelling. The inputs of `x` for the random variable, `c` for the censoring flag, `n` for count of each `x`, `xl` and `xr` for intervally censored data (can't be used with `x`) `t` for the truncation matrix, `tl` for the left truncation scalar or array, and `tr` for the right truncation scalar or array all remain.

1.11.1 Complete Data

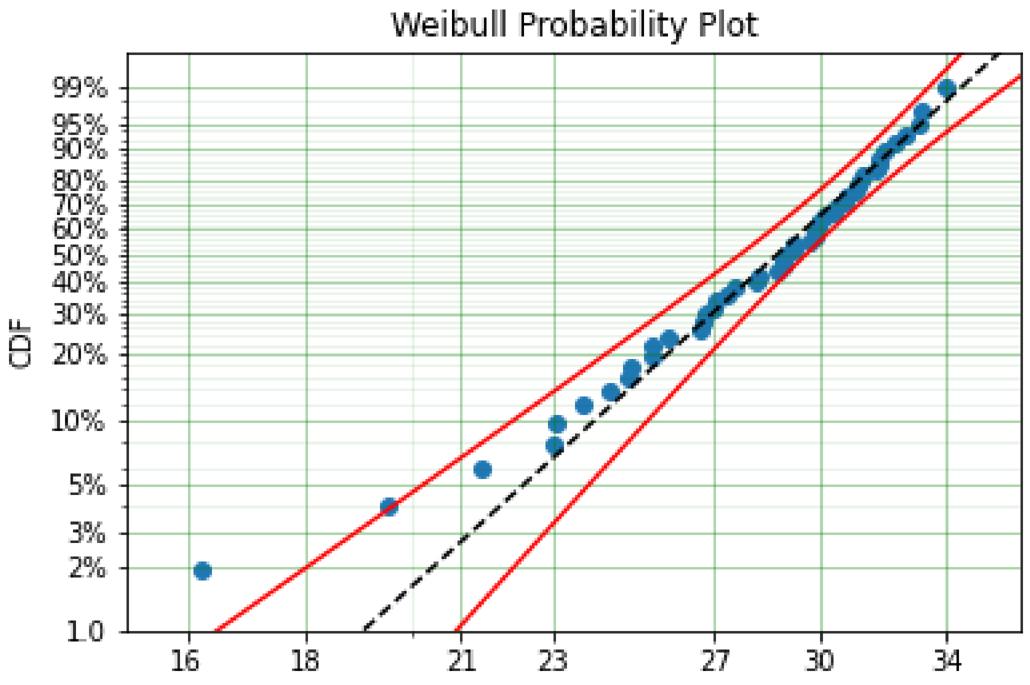
The easiest and simplest case is that when you have a dataset of exactly observed data. that is, you have one array of data with the values at which they failed. Fitting a parametric distribution to the data can be done with a simple call to the `fit()` method:

```
import surpyval as surv
import numpy as np

np.random.seed(10)
x = surv.Weibull.random(50, 30., 9.)
model = surv.Weibull.fit(x)
print(model)
model.plot();
```

```
Parametric SurPyval Model
=========================
Distribution        : Weibull
Fitted by           : MLE
Parameters          :
     alpha: 29.805137406871953
      beta: 10.296037991991037
```

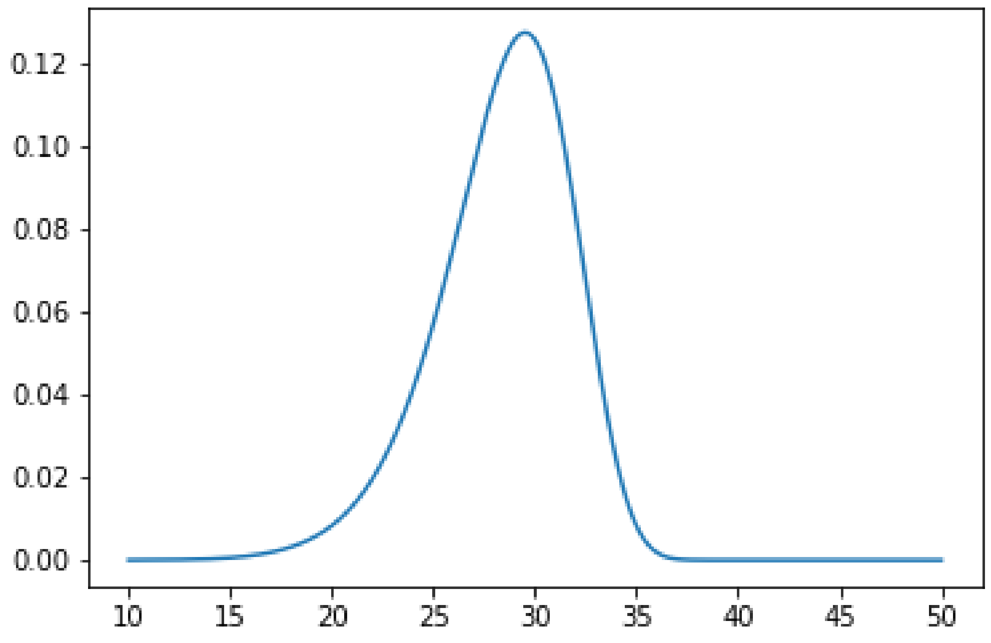
To visualise the outcome of this fit we can inspect the results on a probability plot:

```
model.plot()
```



The `model` object from the above example can be used to calculate the density of the distribution with the parameters found with the best fit from above. This is very easy to do:

```
x = np.linspace(10, 50, 1000)
f = model.df(x)

plt.plot(x, f)
```



The CDF `ff()`, Survival (or Reliability) `sf()`, hazard rate `hf()`, or cumulative hazard rate `Hf()` can be computed as well. This functionality makes it very easy to work with surpyval models to determine risks or to pass the function to other libraries to find optimal trade-offs.

1.11.2 Using censored data

Right Censored

A common complication in survival analysis is that all the data is not observed up to the point of failure (or death). In this case the data is right censored, see the types of data section for a more detailed discussion, surpyval offers a very clean and easy way to model this. First, let's create a simulated data set:

```
import surpyval as surv
import numpy as np

np.random.seed(10)
x = surv.Weibull.random(50, 30, 2.)

observation_limit = 40
# Censoring flag
```

(continues on next page)

(continued from previous page)

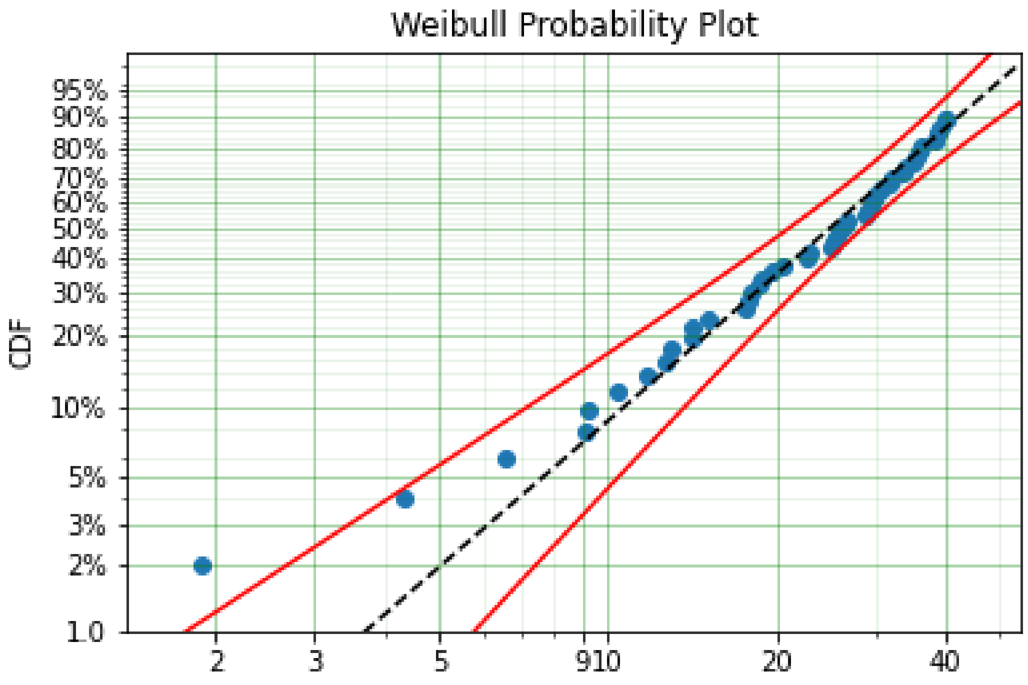
```
c = (x >= observation_limit).astype(int)
x[x >= observation_limit] = observation_limit
```

In this example, we created 50 random Weibull distributed values with alpha = 30 and beta = 2. For this example the observation window has been set to 40. This value is where we stopped observing the events. For all the randomly generated values that are above this limit we create the censoring flag array c. This array has zeros where the event time was observed, and a 1 where the value is above the recorded value. For all the values in the data that are above 40 we set them to 40. This is a common occurence in survival analysis and surpyval is designed to accept this input with a simple call:

```
model = surv.Weibull.fit(x, c)
print(model)
model.plot()
```

```
Parametric SurPyval Model
=========================
Distribution        : Weibull
Fitted by           : MLE
Parameters          :
     alpha: 29.249243175049152
      beta: 2.2291485877426354
```

The plot for this can be seen to be:



The results from this model are very close to the data we input, and with only 50 samples.

Left Censored

The above example can be extended to another kind of censoring; left censored data. This is the case where the values are known to fall below a particular value. We can change our example data set to have a start observation time for which we will left censor all the data below that:

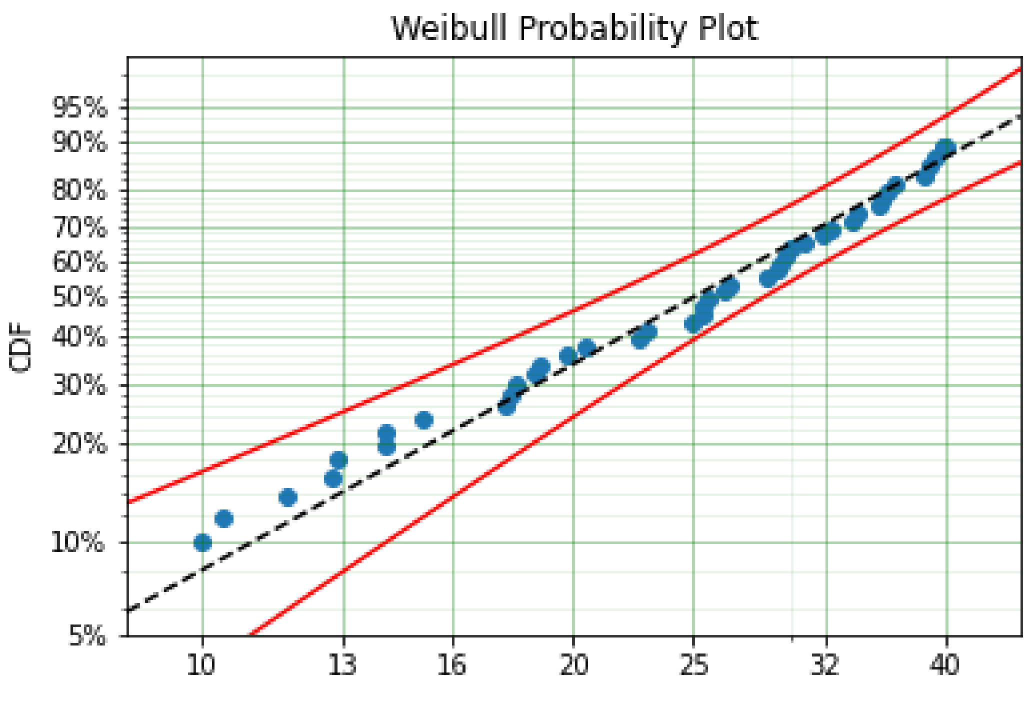
```
observation_start = 10
# Censoring flag
c[x <= observation_start] = -1
x[x <= observation_start] = observation_start
```

That is, we set the start of the observations at 10 and flag that all the values at or below this are left censored. We can then use the updated values of x and c:

```
model = surv.Weibull.fit(x, c)
print(model)
model.plot()
```

```
Parametric SurPyval Model
=========================
Distribution        : Weibull
Fitted by           : MLE
Parameters          :
     alpha: 29.34709766238127
      beta: 2.3049027909575903
```

The values did not substantially change, although the plot does look different as there are no values below 10.



Intervally Censored

The next type of censoring that is naturally handled by surpyval is interval censoring. Creating another example data set:

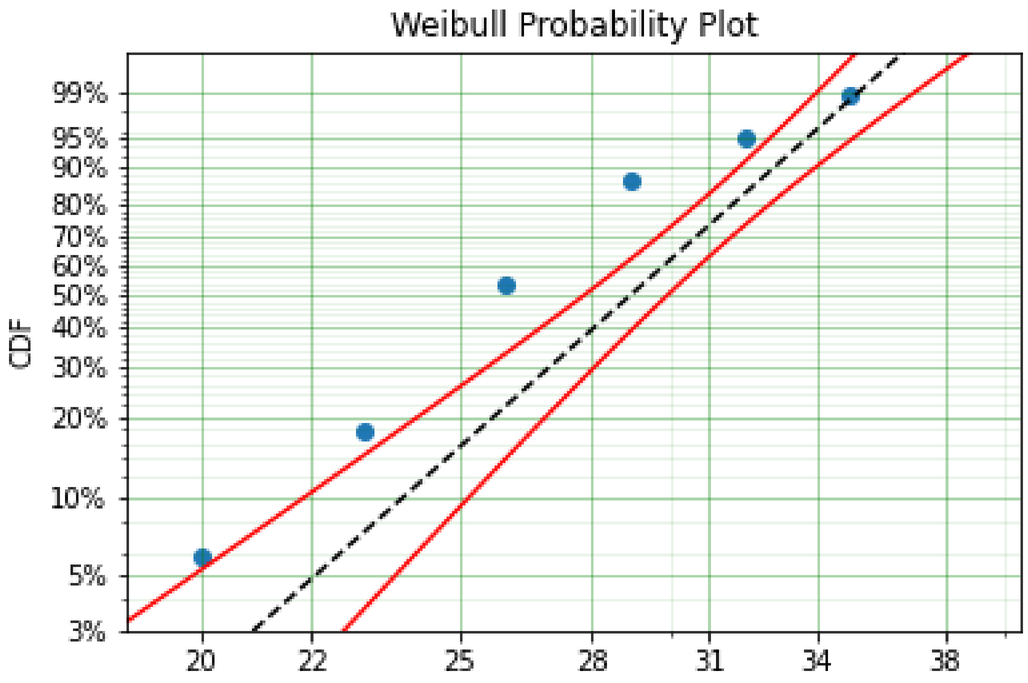
```
import surpyval as surv
import numpy as np

np.random.seed(30)
x = surv.Weibull.random(50, 30, 10.)
n, xx = np.histogram(x, bins=[20, 23, 26, 29, 32, 35, 38])
x = np.vstack([xx[0:-1], xx[1:]]).T
```

In this example we have created the varable x with a matrix of the intervals within which each of the obervations have failed. That is each exact observation has been binned into a window and the x array has an entry [left, right] within which the event failed. We also have the n array that has the count of the failures within the window. With these two values we can make the simple surpyval call:

```
model = surv.Weibull.fit(x, n=n)
print(model)
```

```
Parametric SurPyval Model
=========================
Distribution        : Weibull
Fitted by           : MLE
Parameters          :
     alpha: 30.074154903683105
      beta: 9.637405285678362
```

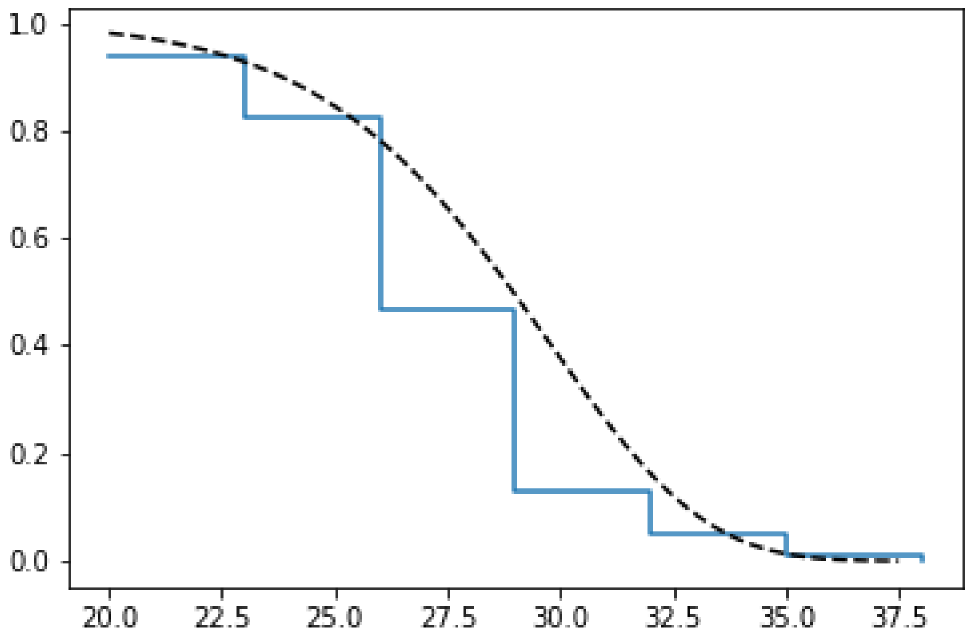
Again, we have a result that is very close to the original parameters. SurPyval can take as input an arbitrary combination of censored data. If we plot the data we will see:



This does not look to be such a good fit. This is because the Turbull estimator finds the probability of failing in a window, not at a given point. So if we align the model plot to the end of the window instead of start with:

```
np_model = surv.Turnbull.fit(x, n=n)
plt.step(np_model.x, np_model.R, where='post')
x_plot = np.linspace(20, 37.5, 1000)
plt.plot(x_plot, model.sf(x_plot), color='k', linestyle='dashed')
```

We get:



Which is, visually, clearly a better fit. You need to be careful when using the Turnbull plotting points to estimate the parameters of a distribution. This is because it is not known where in the intervals a death has actually occurred. However it is good to check the start and end of the window (changing 'where' betweek 'pre' and 'post' or 'mid') to see the goodness-of-fit.

Mixed Censoring

Mixed censoring, or arbitrary censoring is easily handled by SurPyval. So no matter the combination of the data that you have, SurPyval will be able to fit a distribution to it.

```
import surpyval as surv

x  = [0, 1, 2, [3, 4], [6, 10], [4, 8], 5, 19, 10, 13, 15]
c  = [0, 0, 1, 2, 2, 2, 0, -1, 0, 1, 0]
surv.Gumbel.fit(x, c=c)
```

```
Parametric SurPyval Model
=========================
Distribution        : Gumbel
```

(continues on next page)

(continued from previous page)

```
Fitted by            : MLE
Parameters           :
        mu: 9.912232006272871
     sigma: 4.95952392045353
```

1.11.3 Using truncated data

Left truncated

Surpyval has the capacity to handle arbitrary truncated data. A common occurence of this is in the insurance industry data. When customers make a claim on their policies they have to pay an 'excess' which is a charge to submit a claim for processing. If say, the excess on a set of policies in an area is $250, then it would not be logical for a customer to submit a claim for a loss of less than that number. Therefore there will be no claims under $250. This can also happen in engineering where a part may be tested up to some limit prior to be sold, therefore, as a customer you need to make sure you take into account the fact that some parts would have been rejected at the end of the line which you may not have seen. So a washing machine may run through 25 cycles prior to shipping. This is similar to, but distinct from censoring. When something is left censored, we know there was a failure or event below the threshold. Whereas with truncation, we do not see any variables below the threshold. A simulated example may explain this better:

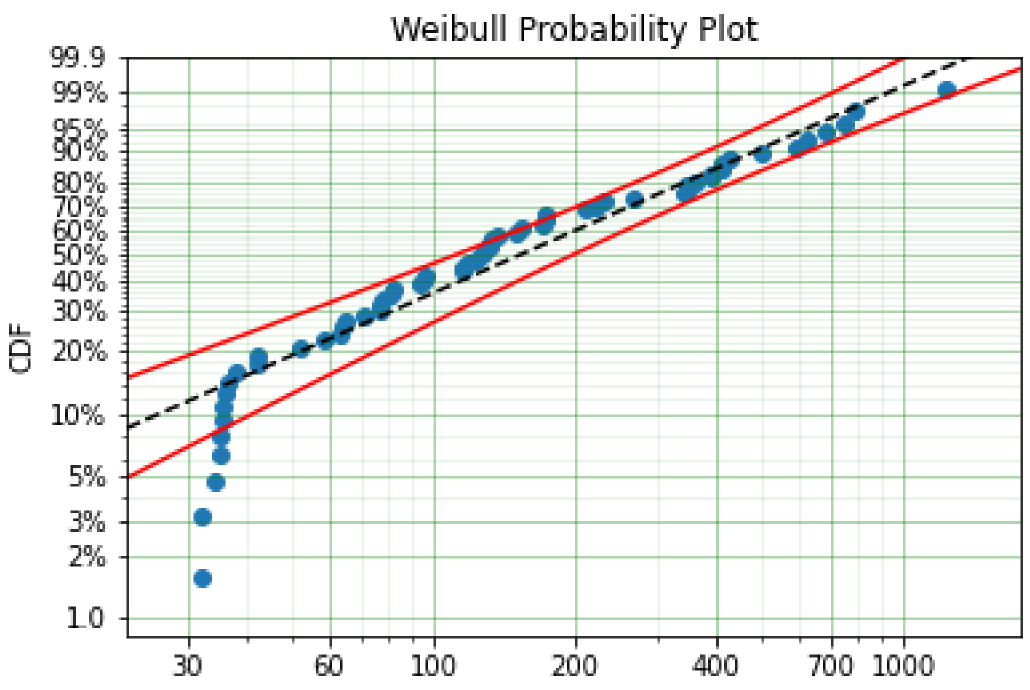
```
import numpy as np
import surpyval as surv

np.random.seed(10)
x = surv.Weibull.random(100, alpha=100, beta=0.6)
# Keep only those values greater than 250
threshold = 25
x = x[x > threshold]
```

We have therefore simulated a scenario where we have taken 100 random samples from a fat tailed Weibull distribution. We then filter to keep only those records that are above the threshold. In this case we assume we haven't seen the data for the washing machines with less than 25 cycles. To understand what could go wrong if we ignore this, what do we get if we assume all the data are failures and there is no truncation?

```
model = surv.Weibull.fit(x=x)
print(model.params)
```

```
[218.39245675   1.0507186 ]
```

With a plot that looks like:



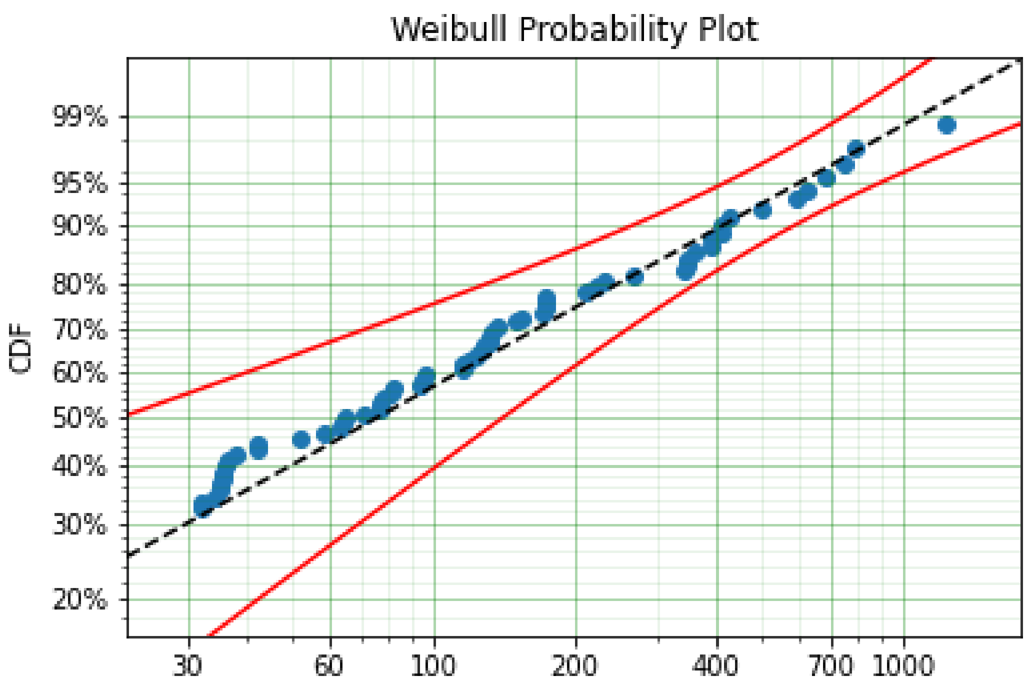
Looking at the parameters of the distribution, you can see that the beta value is greater than 1. Although only slightly, this implies that this distribution has an increasing hazard rate. If you were the operator of the washing machines (e.g. a hotel or a laundromat) and any downtime had a cost, you would conclude from this that replacing the machines after a fixed time would be a good policy.

But if you take the truncation into account:

```
model = surv.Weibull.fit(x=x, tl=threshold)
print(model.params)
```

```
[127.32704868   0.71053572]
```

With the plot:



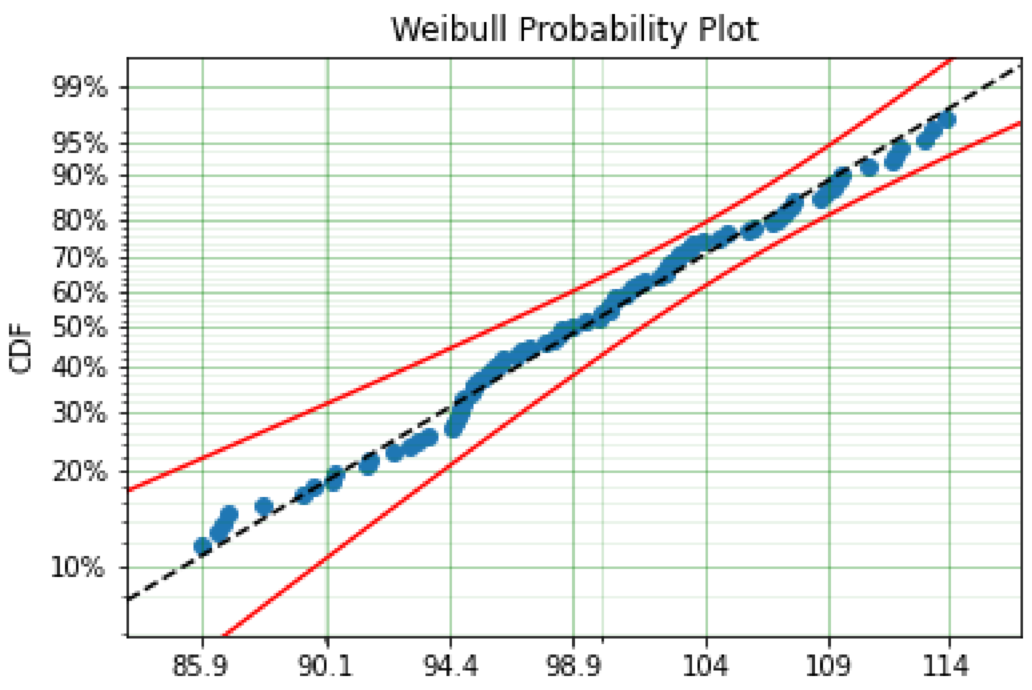
You can see now that the model fits the data much better, but also that the beta parameter is actually below 1. This shows that ignoring the left-truncated data in parametric estimation can lead to errors in prediction.

Right truncated

The example from above can be continued for right-truncated data as well.

```
import numpy as np
import surpyval as surv

np.random.seed(10)
x = surv.Normal.random(100, mu=100, sigma=10)
# Keep only those values greater than 250
tl = 85
tr = 115
# Truncate the data
x = x[(x > tl) & (x < tr)]

model = surv.Weibull.fit(x=x, tl=tl, tr=tr)
print(model.params)
```

```
[102.27078401  12.47906136]
```



From the output above, the number of data points we have has been reduced from the simulated 100, downt to 87. Then with the 87 samples we now have we estimated the parameters to be quite close to the parameters used in the simulation. Further, the plot looks as though the parametric distribution fits the non-parametric distribution quite well.

In the cases above we used a scalar value for the truncation values. But some data has individual values for left truncation. This is seen in trials where someone may join the trial as a late entry. Therefore each data point as an entry time. For example:

```
import surpyval as surv

x  = [3, 4, 6, 7, 9, 10]
tl = [0, 0, 0, 0, 5, 2]

model = surv.Weibull.fit(x, tl=tl)
print(model.params)
```

```
[7.05854717 2.70096672]
```

Intervally and Arbitrarily truncated

Surpyval can even work with arbitrary left and right truncation:

```
import surpyval as surv

x  = [3, 4, 6, 7, 9, 10]
tl = [0, 0, 0, 0, 5, 2]
tr = [10, 9, 8, 10, 15]

model = surv.Weibull.fit(x, tl=tl, tr=tr)
print(model.params)
```

```
[8.12377602 2.56917036]
```

In the above example we used both the tl and tr. However, surpyval has a flexible API where it can take the truncation data as a two dimensional array:

```
import surpyval as surv

x  = [3, 4, 6, 7, 9, 10]
t =   [[ 0, 10],
       [ 0,  9],
       [ 0,  8],
       [ 0, 10],
       [ 5, 15],
       [ 2, 15]]

model = surv.Weibull.fit(x, t=t)
print(model.params)
```

```
[8.12377602 2.56917036]
```

Which, obviously, gives the same result. This shows the flexibility of the surpyval API, you can use scalar, array, or matrix values for the truncations using the t, tl, and tr keywords with the fit method and surpyval does the rest.

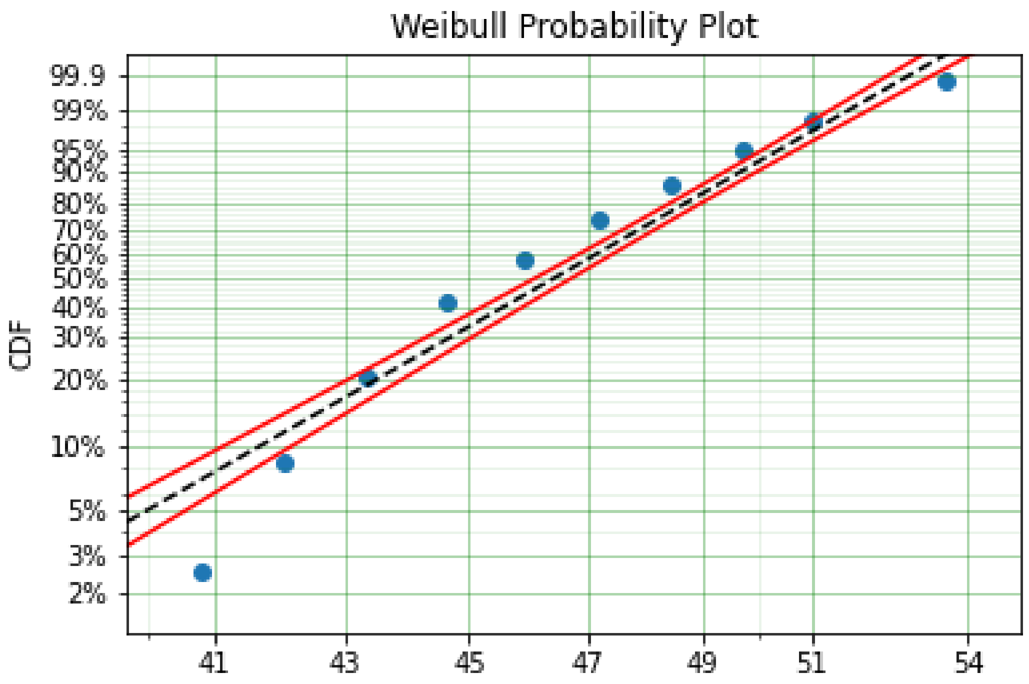
1.11.4 Offsets

Another common feature in survival analysis is a requirement to fit a distribution with an offset. These distributions are sometimes referred to as the two-parameter (e.g. two parameter exponential) three parameter, (e.g., the three three parameter Weibull), or four parameter (e.g four parameter Exponentiated Weibull distribution). SurPyval however just uses an `offset` to increase the numbers of parameters and allow the distribution to be shifted.

Using data from Weibull's original paper for the strenght of Bofor's steel shows when this might be necessary.

```
import surpyval as surv
from surpyval.datasets import BoforsSteel

df = BoforsSteel.df
x = df['x']
n = df['n']

model = surv.Weibull.fit(x=x, n=n)
print(model.params)
model.plot()
```

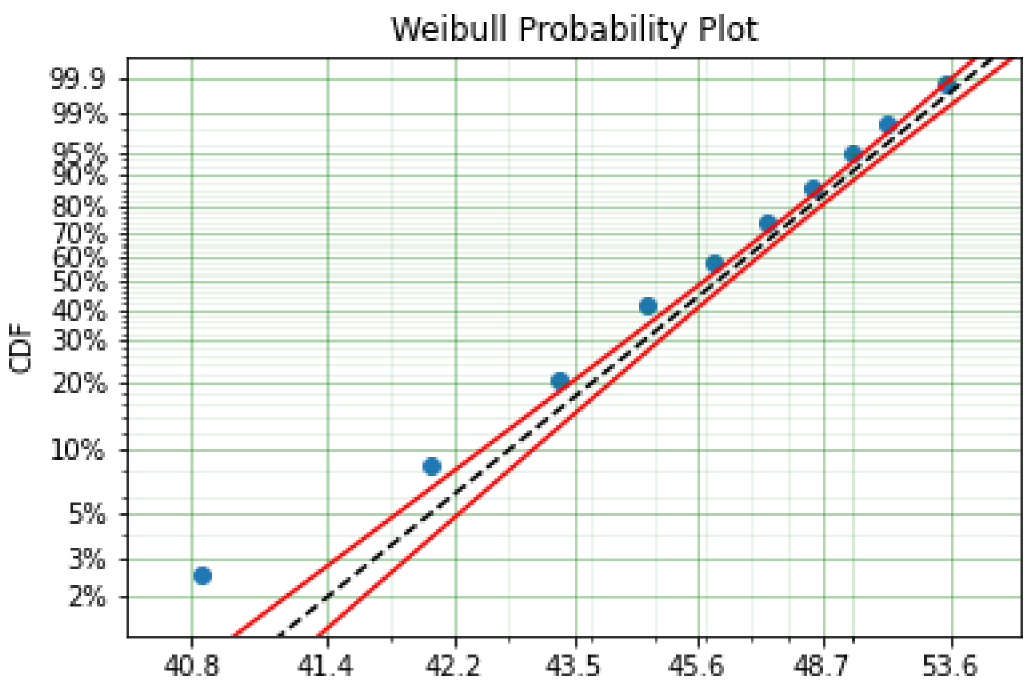
```
[47.36735846 17.5713195 ]
```



The above plot does not look to be a good fit. However, if we use an offset we can use the three parameter Weibull distribution to attempt to get a better fit. Using offset values with surpyval is very easy:

```
import surpyval as surv
from surpyval.datasets import BoforsSteel

df = BoforsSteel.df
x = df['x']
n = df['n']

model = surv.Weibull.fit(x=x, n=n, offset=True)
print(model)
model.plot()
```

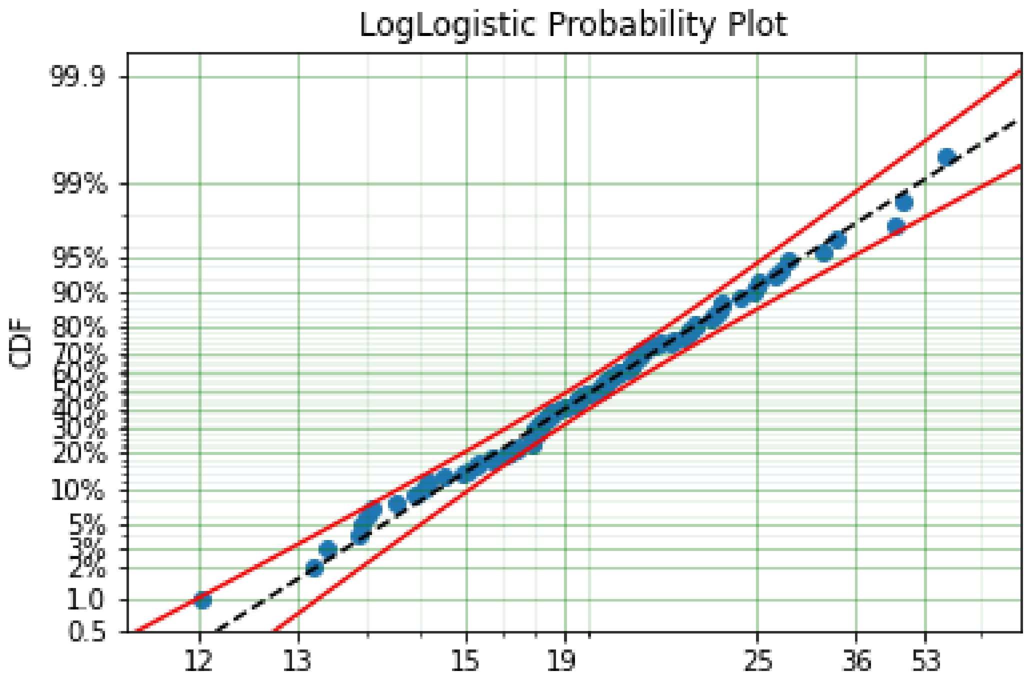
```
Parametric SurPyval Model
=========================
Distribution        : Weibull
Fitted by           : MLE
Offset (gamma)      : 39.76562962867477
Parameters          :
     alpha: 7.141925216146524
      beta: 2.6204524040137844
```



This is evidently a much better fit! The offset value for an offset distribution is saved as `gamma` in the model object. Offsets can be used for any distribution supported on the half real line. Currently, this is the Weibull, Gamma, LogNormal, LogLogistic, and Exponential. For example:

```python
import surpyval as surv
import numpy as np

np.random.seed(10)
x = surv.LogLogistic.random(100, 10, 3) + 10
model = surv.LogLogistic.fit(x, offset=True, how='MLE')
print(model)
model.plot()
```

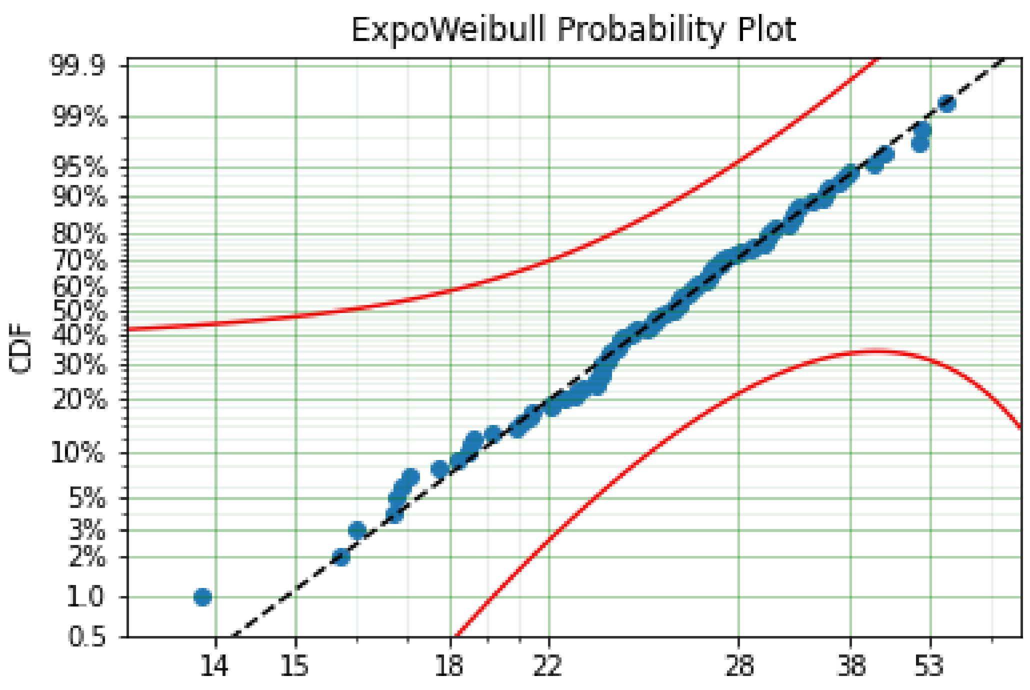
```
Parametric SurPyval Model
=========================
Distribution        : LogLogistic
Fitted by           : MLE
Offset (gamma)      : 9.56270794050046
Parameters          :
     alpha: 10.18946967467503
      beta: 3.407325975660712
```



A four parameter exponentiated Weibull can also be found:

```
import surpyval as surv
import numpy as np

np.random.seed(10)
x = surv.ExpoWeibull.random(100, 10, 1.2, 4) + 10
model = surv.ExpoWeibull.fit(x, offset=True)
print(model)
model.plot(plot_bounds=False)
```

```
Parametric SurPyval Model
=========================
Distribution        : ExpoWeibull
Fitted by           : MLE
Offset (gamma)      : 10.701280166551431
Parameters          :
     alpha: 11.47511146192537
      beta: 1.3969785125819283
        mu: 2.845307244239084
```

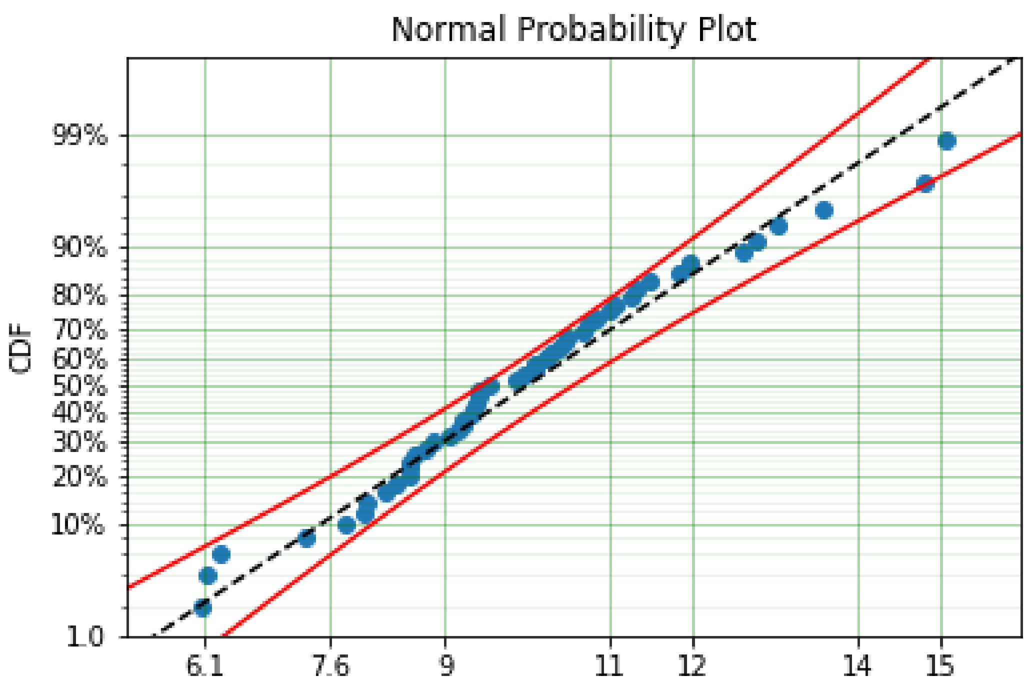


1.11.5 Fixing parameters

Another useful feature of surpyval is the ability to easily fix parameters. For example:

```
import surpyval as surv
import numpy as np

np.random.seed(30)
x = surv.Normal.random(50, 10., 2)
model = surv.Normal.fit(x, fixed={'mu' : 10})
print(model)
model.plot()
```

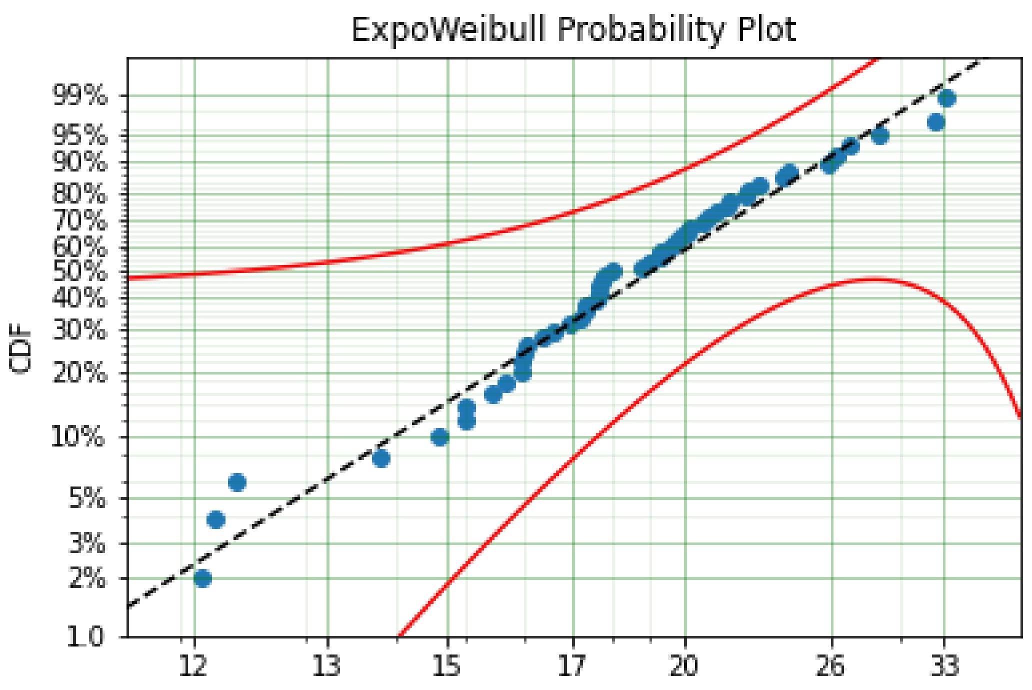
```
Parametric SurPyval Model
=========================
Distribution        : Normal
Fitted by           : MLE
Parameters          :
        mu: 10.0
     sigma: 1.9353643871136006
```



You can see that the mu parameter has been fixed at 10. This can work for distribuitons with many more parameters, including the offset.

```
import surpyval as surv
import numpy as np

np.random.seed(30)
x = surv.ExpoWeibull.random(50, 10., 2, 4) + 10
model = surv.ExpoWeibull.fit(x, offset=True, fixed={'mu' : 4, 'gamma' : 10, 'alpha' :␣
↪10})
print(model)
model.plot()
```

```
Parametric SurPyval Model
=========================
Distribution        : ExpoWeibull
Fitted by           : MLE
Offset (gamma)      : 10.0
Parameters          :
     alpha: 10.0
      beta: 1.9986073390210994
        mu: 1.2
```



We have fit three of the four parameters for an offset exponentiated-Weibull distribution!

1.11.6 Modelling with arbitrary input

The surpyval API is extremely flexible. All the unique examples provided above can all be used at once. That is, data can be censored, truncated, and directly observed with offsets and fixing parameters. The API is completely flexible. This makes surpyval an extremely useful tool for analysts where the data is gathered in a manner where it's cleanliness is not guaranteed.

```python
import surpyval as surv

x  = [0, 1, 2, [3, 4], [6, 10], [4, 8], 5, 19, 10, 13, 15]
c  = [0, 0, 1, 2, 2, 2, 0, -1, 0, 1, 0]
tl = [-1, 0, 0, 0, 0, 0, 2, 2, -np.inf, 0, 0]
tr = 25
model = surv.Normal.fit(x, c=c, tl=tl, tr=tr, fixed={'mu' : 1.})
print(model)
```

```
Parametric SurPyval Model
=========================
Distribution        : Normal
Fitted by           : MLE
Parameters          :
        mu: 1.0
     sigma: 9.131202240846182
```

1.11.7 Using alternate estimation methods

Surpyval's API is very flexible because you can change which method is used to estimate parameters. This is useful when a more appropriate method is needed or the method you are using fails.

The default parametric method for surpyval is the maximum likelihood estimation (MLE), this is because it can take any arbitrary input. However, the MLE is not always the best estimator. Consider an example with the uniform distribution:

```python
import surpyval as surv
import numpy as np

np.random.seed(5)
x = surv.Uniform.random(20, 5, 10)
print(x.min(), x.max())

mle_model = surv.Uniform.fit(x)
print(*mle_model.params)
```

```
5.9386061433062585 9.593054539689607
5.9386061433062585 9.593054539689607
```

You can see that the results are the same. This is because the maximum likelihood estimate of the parameters of a uniform distriubtion are just the smallest and largest values in the sample. If however we use the 'Maximum Product Spacing' method we get:

```python
mps_model = surv.Uniform.fit(x, how='MPS')
print(*mps_model.params)
```

```
5.532556321486052 9.999104361509815
```

You can see that using the MPS method we have parameters that are closer to the real values. This is because the MPS method can 'look outside' the existing values to estimate where the real value lies. See the details of this method in the 'Parametric Estimation' section. But the MPS method is useful when you need to estimate the point at which a distribution's support starts or for any disttribution that has unknown support. Concretely, this includes any offset distribution or a distribution with a finite upper and lower support (Uniform, Generalised Beta, Triangle)

The other important use case is when, for some reason, an alternate estimation method just does not work. For example:

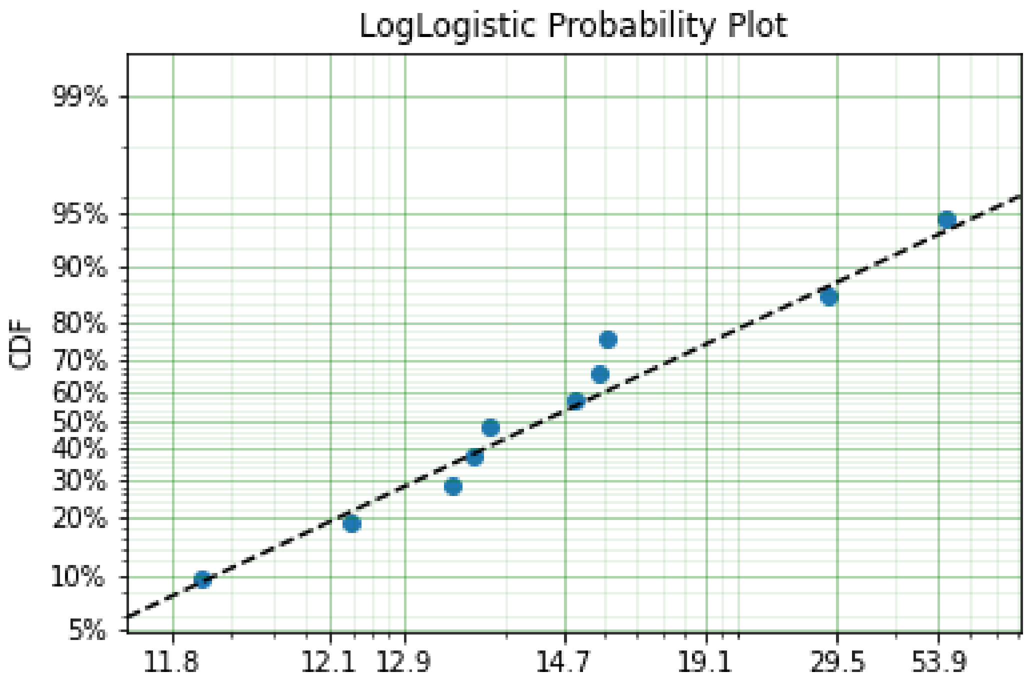
```python
import surpyval as surv
import numpy as np

np.random.seed(30)
x = surv.LogLogistic.random(10, 4., 2) + 10
model = surv.LogLogistic.fit(x, how='MLE', offset=True)
```

```
Precision was lost, try:
    - Using alternate fitting method
    - visually checking model fit
    - change data to be closer to 1.
```

This shows, that the Maximum Likelihood Estimation may have failed for this data. However, because we have access to other methods, we can use an alternate estimation method:

```python
import surpyval as surv
import numpy as np

np.random.seed(30)
x = surv.LogLogistic.random(10, 4., 2) + 10
model = surv.LogLogistic.fit(x, how='MPS', offset=True)
print(model)
model.plot()
```

```
Parametric SurPyval Model
=========================
Distribution        : LogLogistic
Fitted by           : MPS
Offset (gamma)      : 11.524905733806891
Parameters          :
     alpha: 2.631868521887908
      beta: 0.9657662293516666
```



Our estimation has worked! Even though we used the MPS estimate for the parameters, we can still call all the same functions with the created variable to find the density `df()`, hazard `hf()`, CDF `ff()`, SF `sf()` etc. So regardless of the estimation method, we can still use the model.

This shows the power of the flexible API that surpyval offers, because if your modelling fails using one estimation method, you can use another. In this case, the MPS method is quite good at handling offset distributions. It is therefore a good approach to use when using offset distributions.

As stated in the Non-Parametric section, there is a risk that using the Turnbull estimator when all values are tunctated by the same values. We will now show what happens. First, some example data:

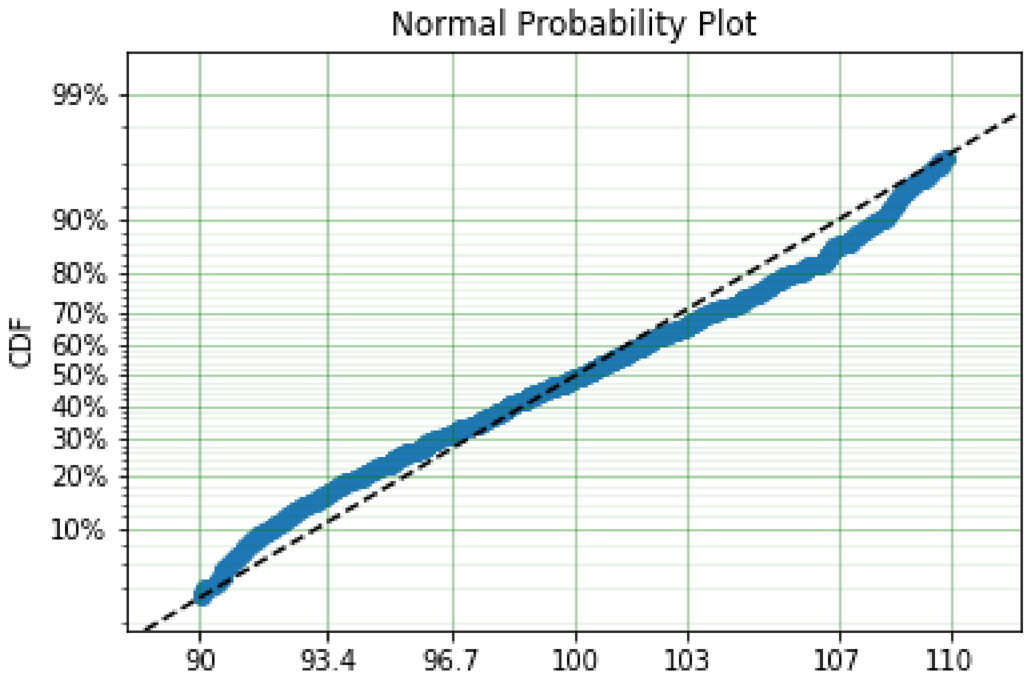
```python
import surpyval as surv
```

(continues on next page)

(continued from previous page)

```
import numpy as np

np.random.seed(1)
x = surv.Normal.random(1000, 100, 10)
tl = 90
tr = 110
x = x[x > tl]
x = x[x < tr]

mpp_model = surv.Normal.fit(x, tl=tl, tr=tr, how='MPP')
mpp_model.plot()
mpp_model
```

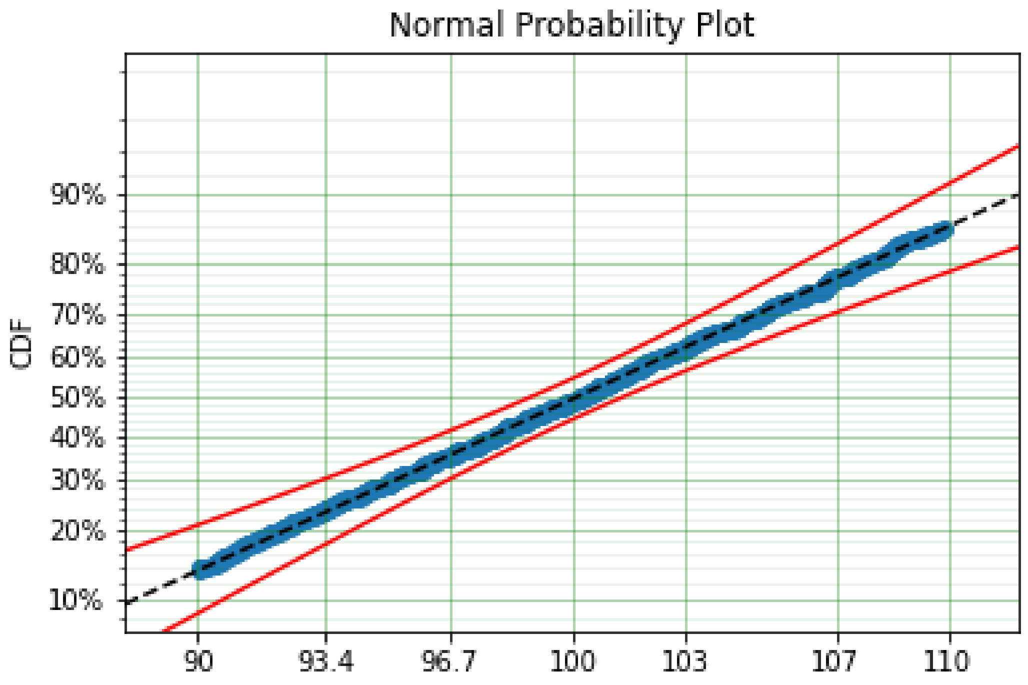
```
Parametric SurPyval Model
=========================
Distribution        : Normal
Fitted by           : MPP
Parameters          :
        mu: 100.03108440743388
     sigma: 5.432878735738111
```



You can see that there is a strange match between the Turnbull estimate of the CDF and the parametric model. Also, you can see that the CDF at 90 is near 0% and the CDF at 110 is near 100%. This shows that it has not taken into account the truncation. Instead, if we use MLE we get:

```
model = surv.Normal.fit(x, tl=tl, tr=tr, how='MLE')
model.plot()
model
```

```
Parametric SurPyval Model
=========================
Distribution        : Normal
Fitted by           : MLE
Parameters          :
        mu: 100.13045397963812
     sigma: 9.17784957390746
```



We can see that the MLE method is a much better fit to this data, further, the MLE estimate of the σ parameter is much closer. The plotting points for the MLE plot have been adjusted in accordance with the truncation that the MLE model has estimated at the first entry. This is because it is known to be truncated and needs to be adjusted. This is not possible with the MPP method because the Turnbull estimator cannot adjust the truncation at the first and last value as it can make no assumptions about the truncation at those points.

This is just a word of warning for when using Truncation and the MPP method, make sure not all values are truncated by the same value, otherwise it will give a poor fit.

1.11.8 Mixture Models

On occasion, it can appear as though there are one, or two different distributions in the data you are using. On these occasions it can be useful to use a different type of distribuiton; or really, distributions. A mixture model is a distribution made from the partial combination of several distributions. Intuitively, it can be understood as a distribution where there is a proportion that fail for each kind of distribution. So 60% may come from a Weibull(3, 4) distribution but then another 40% come from a Weibull(19, 2) distribution.

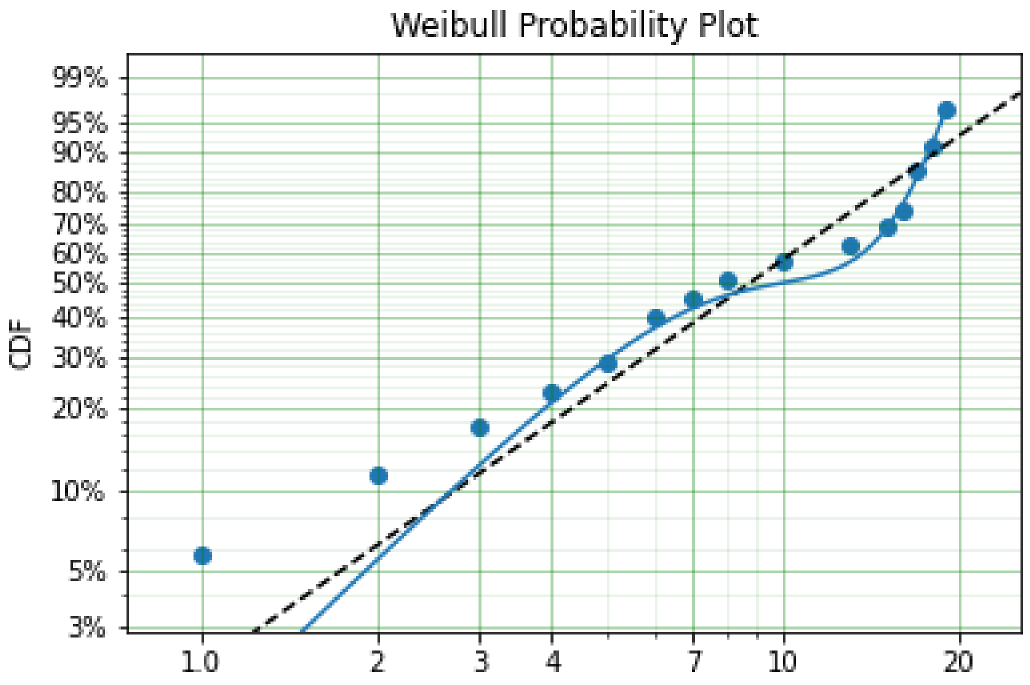
SurPyval uses Expectation-Maximisation to

```
import surpyval as surv
import numpy as np
```

(continues on next page)

(continued from previous page)

```
from matplotlib import pyplot as plt

x = [1, 2, 3, 4, 5, 6, 6, 7, 8, 10, 13, 15, 16, 17 ,17, 18, 19]
x_ = np.linspace(np.min(x), np.max(x))

model = surv.Weibull.fit(x)
wmm = surv.MixtureModel(x=x, dist=surv.Weibull, m=2)

model.plot(plot_bounds=False)
plt.plot(x_, wmm.ff(x_))
```



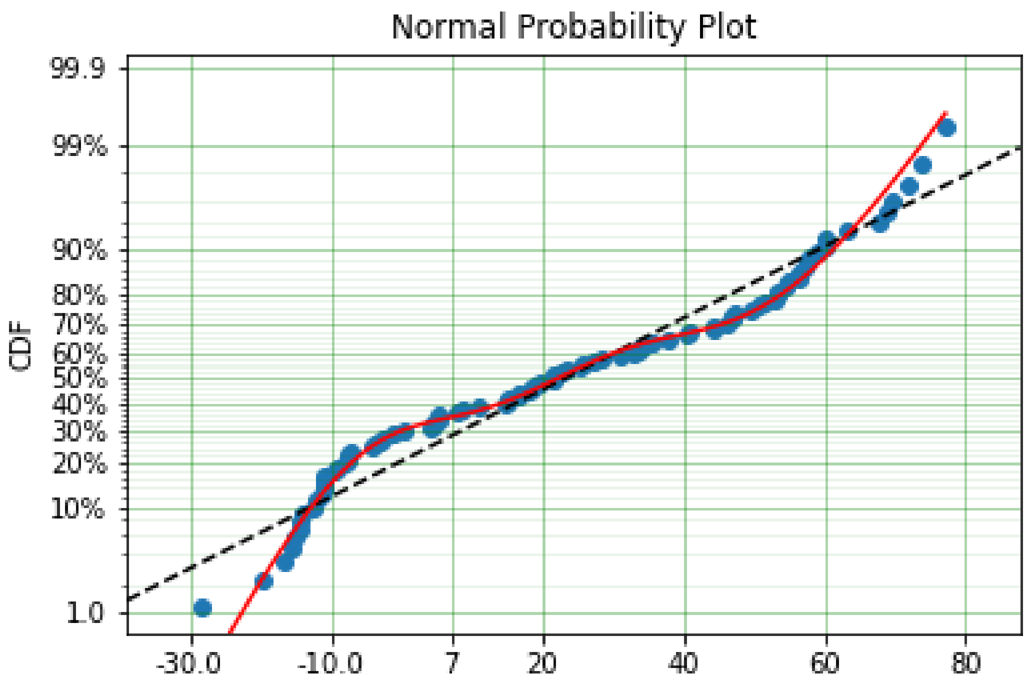
You can see that the mixture model, in blue, tracks the data more closely than does the single model. SurPyval has incredible flexibility. The number of distributions can be changed by simply changing the value of `m`, and, the distribution passed to `dist` in the mixture can also be changed. Consider:

```
import surpyval as surv
import numpy as np
from matplotlib import pyplot as plt

np.random.seed(1)
x1 = surv.Normal.random(20, -10, 5)
x2 = surv.Normal.random(30, 10, 10)
x3 = surv.Normal.random(40, 50, 15)
x = np.concatenate([x1, x2, x3])
np.random.shuffle(x)
x_ = np.linspace(np.min(x), np.max(x))

normal = surv.Normal.fit(x)
gmm = surv.MixtureModel(x=x, dist=surv.Normal, m=3)
```

(continues on next page)

(continued from previous page)

```
normal.plot(plot_bounds=False)
plt.plot(x_, gmm.ff(x_), color='red')
```



It was that simple to create a gaussian mixture model using `m=3` and the `dist=surv.Normal` parameters. SurPuyval does default to 2 Weibull distributions if neither parameters are provided, but it can take any distribution in SurPyval as an input distribution.

Finally, mixture models can take counts and censoring flags as input (but not, yet, truncation). This makes SurPyval a truly powerful package for your survival analysis.

1.11.9 Limited Failure Population

Another kind of model that is useful in survival analysis is when a population has a limited number of items in the population that are susceptible to the failure. This is also known as a 'Defective Subpopulation' model. As such, no matter how long a test continues, it will not be possible for all items to fail (with the particular death/failure).

As an example, we can created a Defective Subpopulation Weibull, also known as a Limited Failure Population Model using a Weibull distribution:

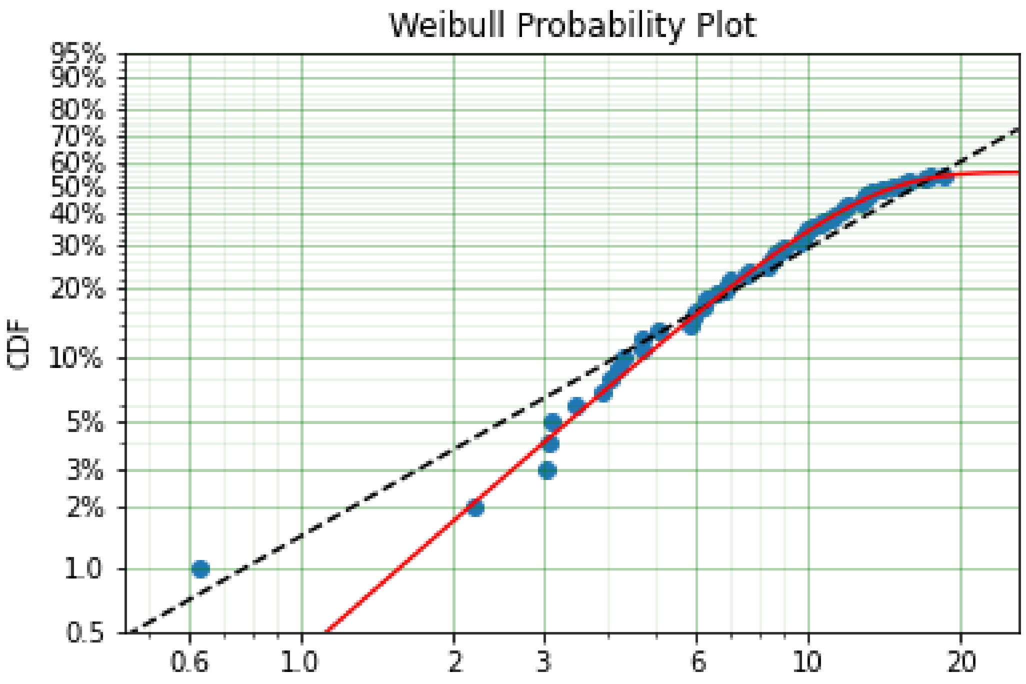
```
import surpyval as surv
import numpy as np
from matplotlib import pyplot as plt

lfp_weibull = surv.Weibull.from_params([10, 2], p=0.6)
np.random.seed(10)
# LFP Model outputs x, c, and n from `random()`
x, c, n = lfp_weibull.random(100)
```

(continues on next page)

(continued from previous page)

```
# Fit regular Weibull
model = surv.Weibull.fit(x=x, c=c, n=n)
model.plot(plot_bounds=False)

# Set LFP to be `True`
lfp_model = surv.Weibull.fit(x=x, c=c, n=n, lfp=True)
print(lfp_model)
xx = np.linspace(np.min(x), np.max(x)*2)
plt.plot(xx, lfp_model.ff(xx), color='red')
```

```
Parametric SurPyval Model
=========================
Distribution        : Weibull
Fitted by           : MLE
Max Proportion (p)  : 0.5553951704157292
Parameters          :
     alpha: 10.180334244350309
      beta: 2.1358575854287265
```



This API works with any distribution so simply changing `Weibull` to `Exponential` would create a Defective Subpopulation Exponential / Limited Failure Population Exponential model. Further, if it was changed to `Gamma` it would create a Defective Subpopulation Gamma model / Limited Failure Population Gamma.

LFP models can only (as yet) work with `MLE`. It cannot (yet) work with the other estimation methods. The `MSE` is a good candidate for implementation.

1.11.10 Zero-Inflated Modelling

In survival analysis you might have the scenario where many failure times are 0, known as being dead on arrival. In this case we need a model that can account for the fact that many will be failed at 0, this is a situation that cannot be handled by regular distribuitons, since most have a 0% chance of failing at 0. Therefore what we need is something that is symmetrical to the LFP/DS case, where a proportion of the failures occur at 0 instead of there being a proportion that will never fail.

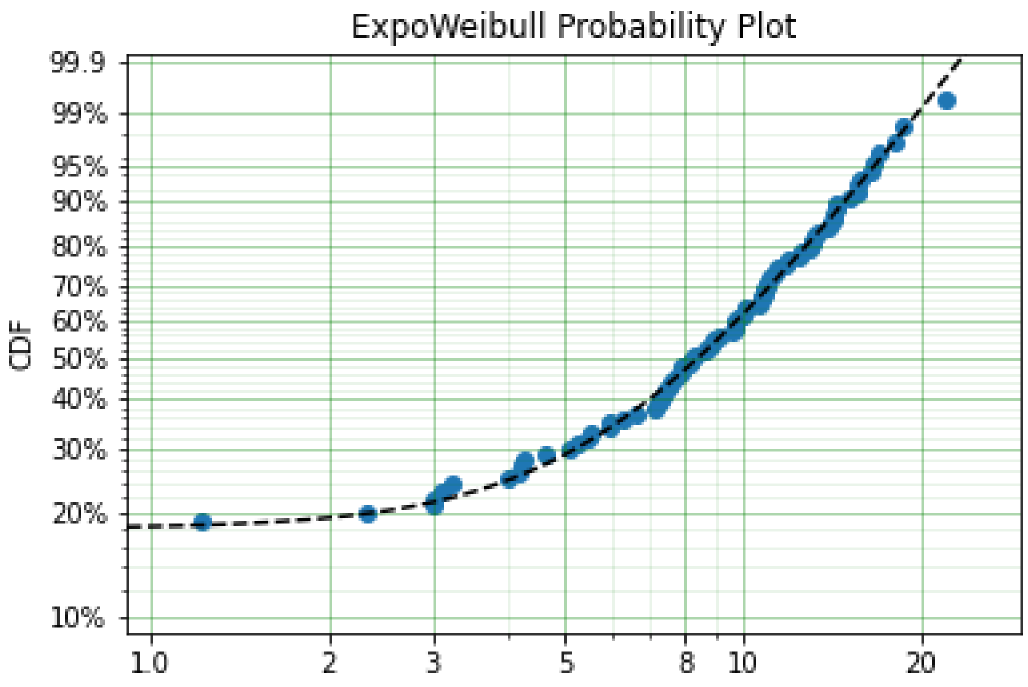
```python
import surpyval as surv
from autograd import numpy as np

dist = surv.ExpoWeibull
model = dist.from_params([10.2, 2., 1.3], f0=0.15)
np.random.seed(10)
x = model.random(100)
model
```

```
Parametric SurPyval Model
=========================
Distribution        : ExpoWeibull
Fitted by           : given parameters
Zero-Inflation (f0) : 0.15
Parameters          :
     alpha: 10.2
      beta: 2.0
        mu: 1.3
```

Using this random data, we can make a fitted model (with the added convenience not offered in the real world of knowing exactly what parameters we are aiming toward).

```python
fitted_model = dist.fit(x, zi=True)
print(fitted_model)
fitted_model.plot()
```

```
Parametric SurPyval Model
=========================
Distribution        : ExpoWeibull
Fitted by           : MLE
Zero-Inflation (f0) : 0.1799999522942094
Parameters          :
     alpha: 11.723925167019866
      beta: 2.769781748379123
        mu: 0.8437868556785479
```

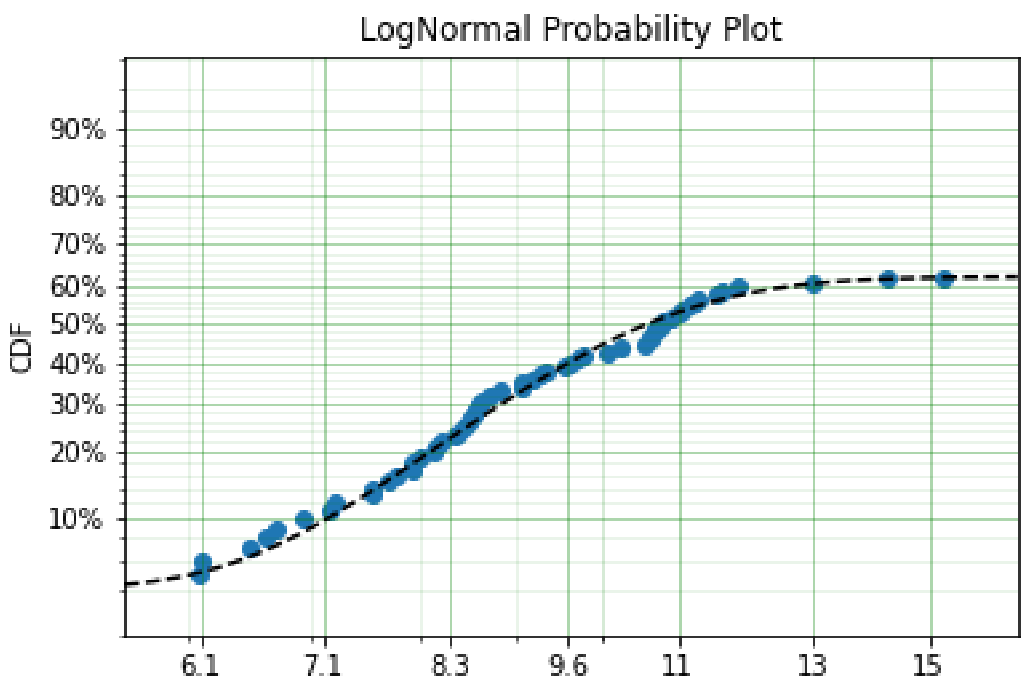


We can see that we have made a good fit!

To showcase the SurPyval API again, and to demonstrate the flexibility, it is trivial to have Defective Subpopulation Zero Inflated (DSZI) model / Limited Failure Population and Zero Inflated model.

```
import surpyval as surv
import numpy as np

dist = surv.LogNormal
model = dist.from_params([2.2, .2], f0=0.05, p=0.6)
np.random.seed(10)
# Random values from LFP models come in xcn format!!!!!
x, c, n = model.random(100)


fitted_model = dist.fit(x, c, n, zi=True, lfp=True)
print(fitted_model)
fitted_model.plot()
```

```
Parametric SurPyval Model
=========================
Distribution         : LogNormal
Fitted by            : MLE
Max Proportion (p)   : 0.6061204729747632
Zero-Inflation (f0)  : 0.040000034963115105
Parameters           :
        mu: 2.2060270833195372
     sigma: 0.19060910628572927
```



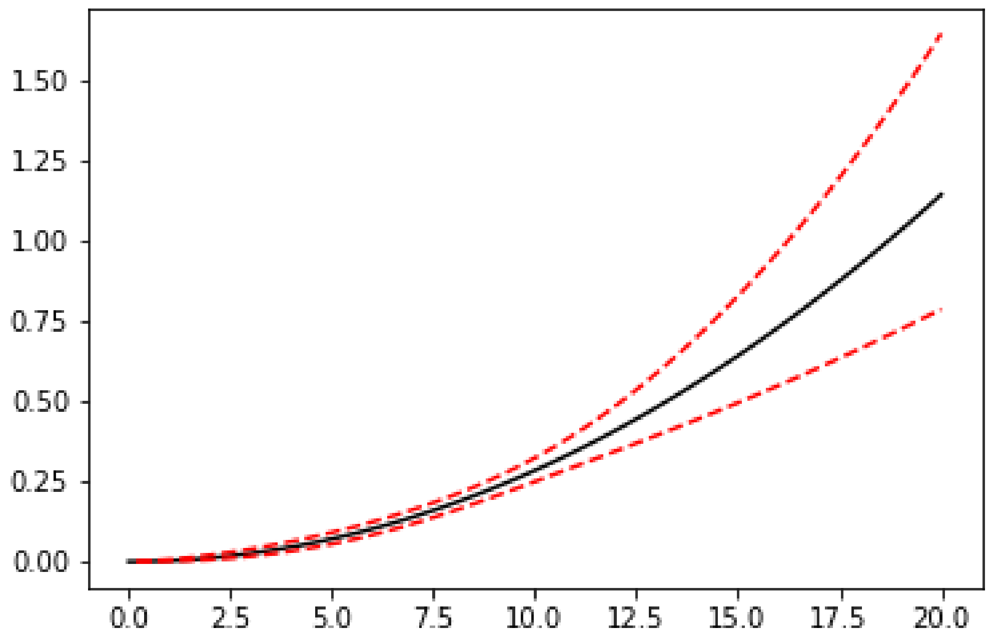
Using a `LogNormal` distribution we were able to easily capture the DS/LFP and ZI behaviour of the data.

1.11.11 Confidence Intervals

SurPyval can be used to compute the confidence interval for any of the functions of a distribution. That is, *SurPyval* can compute the confidence interval for `ff()`, `sf()`, `hf()`, `Hf()`, and `df()`.

Once you have a model, this can easily be computed with the `cb()` method.

```
from surpyval import Weibull
import numpy as np
from matplotlib import pyplot as plt

x = Weibull.random(100, 10, 3)

model = Weibull.fit(x)

x_plot = np.linspace(0, 20, 100)
plt.plot(x_plot, model.sf(x_plot), color='black')
plt.plot(x_plot, model.cb(x_plot, on='sf', alpha_ci=0.1), color='red', linestyle='--')
```



This shows that we can change the confidence level with `alpha_ci` and that we can change the function for which we want the confidence interval. That is, the `on` keyword can be any of `sf`, `ff`, `df`, `hf`, or `Hf`. This will work with models that you create as well, so even a user defined Distribution will be able to have the confidence intervals computed. Creating these models is discussed in the section below.

1.11.12 Creating a custom Distribution

Given the implementation in SurPyval, it is possible to create a new distribution and use all the previously listed techniques. For example, the Gompertz distribution is not implemented in the surpyval API, this however can be quickly overcome. First, we set up a random number generator. Because SurPyval works based on the autograd numpy implementation, it is essential that you use the autograd numpy import to make this work.

```
import surpyval as surv
# IMPORTANT - Will not work with regular numpy
from autograd import numpy as np

def qf(p, mu, b):
    return (np.log(((-np.log(p)/mu))) + 1)/b

# Generate random values from Gompertz distribution
np.random.seed(1)
x = qf(np.random.uniform(0, 1, 100), .3, 1.1)
```

Now that we have our random data set, we can fit a Gompertz distribution to it. To do so, we need to create a Gompertz distribution class, and to do this we need the cumulative hazard function, the names of the parameters, the bounds of the parameters, and the distribution support.

```
name = 'Gompertz'
```

(continues on next page)

(continued from previous page)

```
def Hf(x, *params):
    return params[0] * np.exp(params[1] * x - 1)

param_names = ['nu', 'b']
bounds = ((0, None), (0, None))
support = (-np.inf, np.inf)
Gompertz = surv.parametric.Distribution(name, Hf, param_names, bounds, support)
```

With this now created, all the calls to the regular surpyval API can be used.

```
Gompertz.fit(x)
```

```
Parametric SurPyval Model
=========================
Distribution        : Gompertz
Fitted by           : MLE
Parameters          :
        nu: 1.15060014910275
         b: 1.8973107004872167
```

If we transform the data slightly, we can show that this can be used with censored and truncated data as well.

```
c = np.zeros_like(x)
# Right censor all values above 2
c[x > 2] = 1
x[x > 2] = 2
# Left truncate all values below 0
tl = 0
c = c[x > tl]
x = x[x > tl]

model = Gompertz.fit(x=x, c=c, tl=tl)
model
```

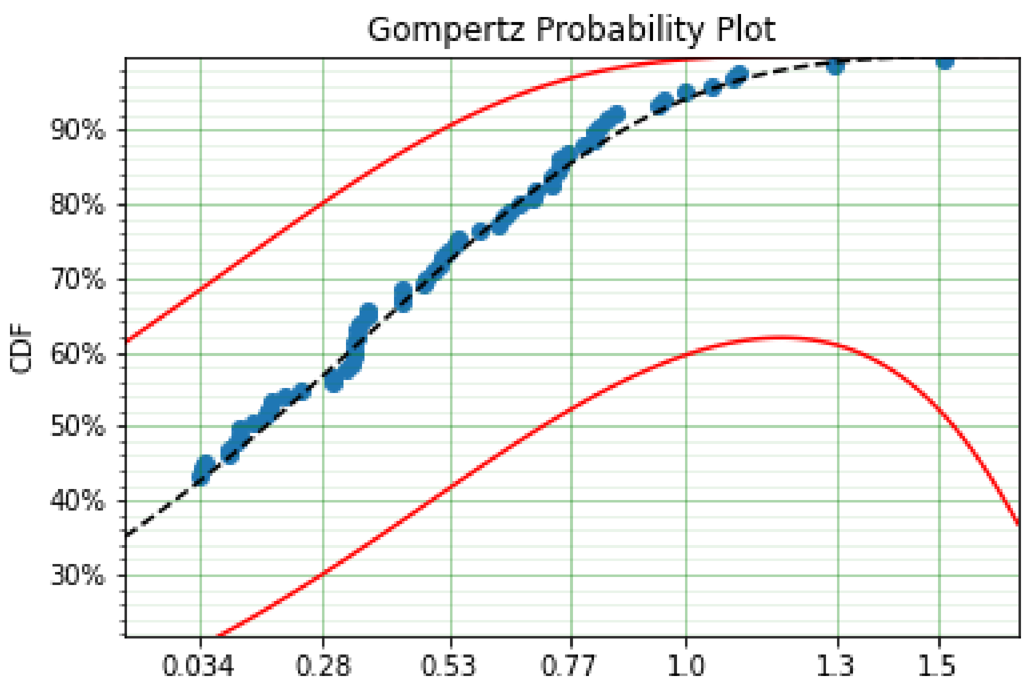
```
Parametric SurPyval Model
=========================
Distribution        : Gompertz
Fitted by           : MLE
Parameters          :
        nu: 1.4228615499353794
         b: 1.688152800158132
```

This is extraordinary! We have created a new distribution using only the cumulative hazard function, but are able to handle arbitrary censoring and truncation. It shows the power of the SurPyval API and functionality.

Credit for this idea must be given to the creators of the *lifelines* package. *lifelines* is capable of receiving a cumulative hazard function that can then be used as a distribution to fit parameters. However, at the time of writing it could not handle arbitrarily censored or truncated data.

Even with a user defined `Hf()` we can still use the confidence bounds as well. The results of this can be seen by simply calling the plot function:

```
model.plot(alpha_ci=0.5)
```



You can see that the distribution is not linearised. This is because the Hf is not readily convertible into the transformation function needed to do the linearisation of the CDF. The defaults are a simple linear scale for both the x and y axis and it shows that the confidence bounds have worked nicely.

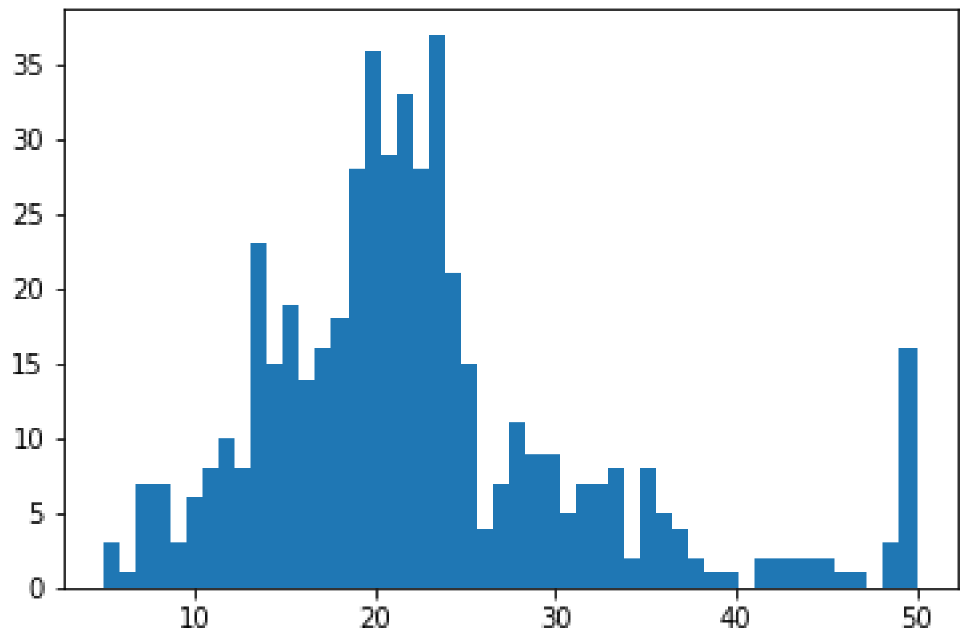
Warning: Due to the implementation of confidence bounds in surpyval it can result in numeric overflows which results in incredulous bounds. Please take caution when using the cb with non surpyval implemented distributions.

1.12 Example Applications

This section documents some of the applications that SurPyval as a survival analysis toolkit can be useful to you, no matter what discipline you need it for.

1.12.1 Boston House Prices

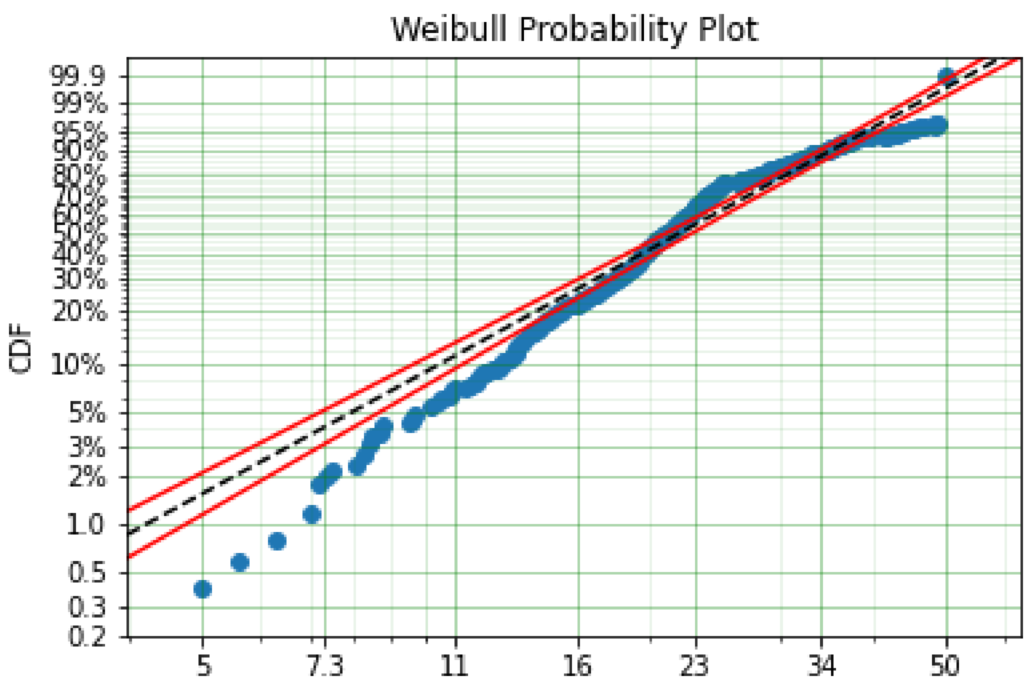
No statistical analysis package can avoid doing the 'hello world' task of analysing the 'Boston House Pricing' dataset. What might surprise some readers is that this would even be considered… The myriad blogs and Kaggle posts looking into this problem can not surely be improved upon. I agree, however, it is a good example of why one needs to be aware of censoring and how flexible the SurPyval API is when dealing with it. Looking at the boston house pricing dataset you can see that there is a suspicous number of houses at the top end that all have the same price:



On consideration, one can see that there are no houses above $50,000 and that the density at that point is much higher than we would expect because we would expect some form of a 'fat-tail.' That is, we should expect a decreasing number of houses at the highest costs. It is therefore safe to conclude that all values above $50,000 have been set to $50,000; which is to say that the sale price is right censored! And because it is a censored observation we will need to use an analysis tool that can handle censored observations, lest we may be wrong in our estimates of the distribution of housing prices. So lets load the data and see what results we can get, starting with the raw data.

```python
import surpyval as surv
x = Boston.df['medv'].values
x, c, n = surv.xcn_handler(x)

model = surv.Weibull.fit(x, c, n)
print(model)
model.plot()
```

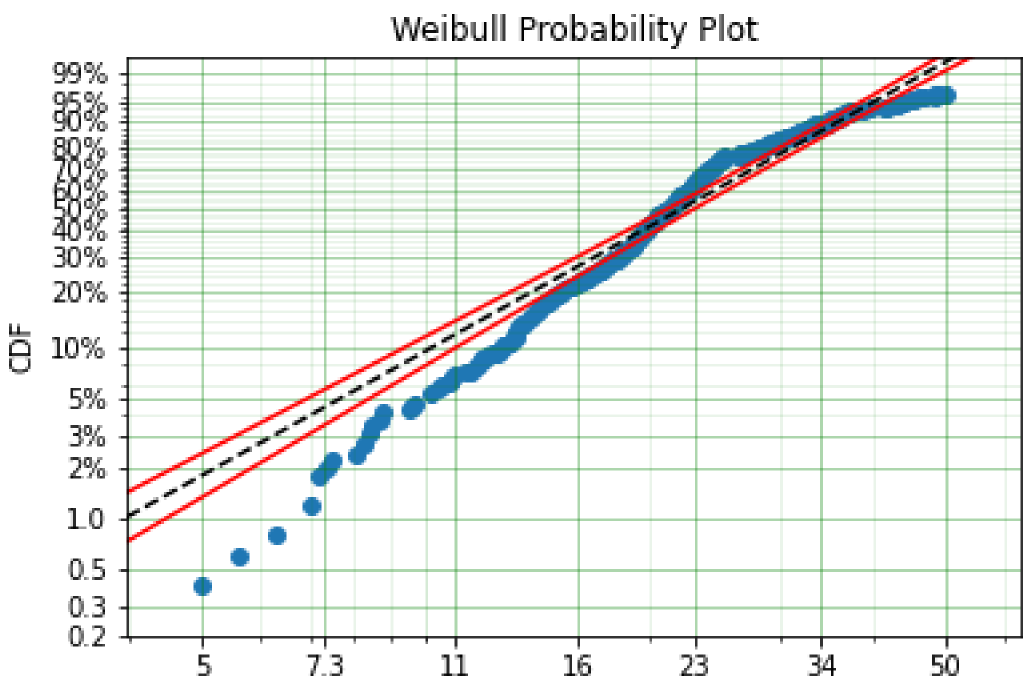
```
Parametric SurPyval Model
=========================
Distribution        : Weibull
Fitted by           : MLE
Parameters          :
     alpha: 25.386952832397032
      beta: 2.5651903209947684
```



From the above plot you can see that near 50, the parametric model diverges substantially from the actual data. So we can see that having not censored the highest values means that our model could be improved by doing so. Let’s see:

```
import surpyval as surv
x = Boston.df['medv'].values
x, c, n = surv.xcn_handler(x)
# Right censor the highest value
c[-1] = 1

model = surv.Weibull.fit(x, c, n)
print(model)
model.plot()
```

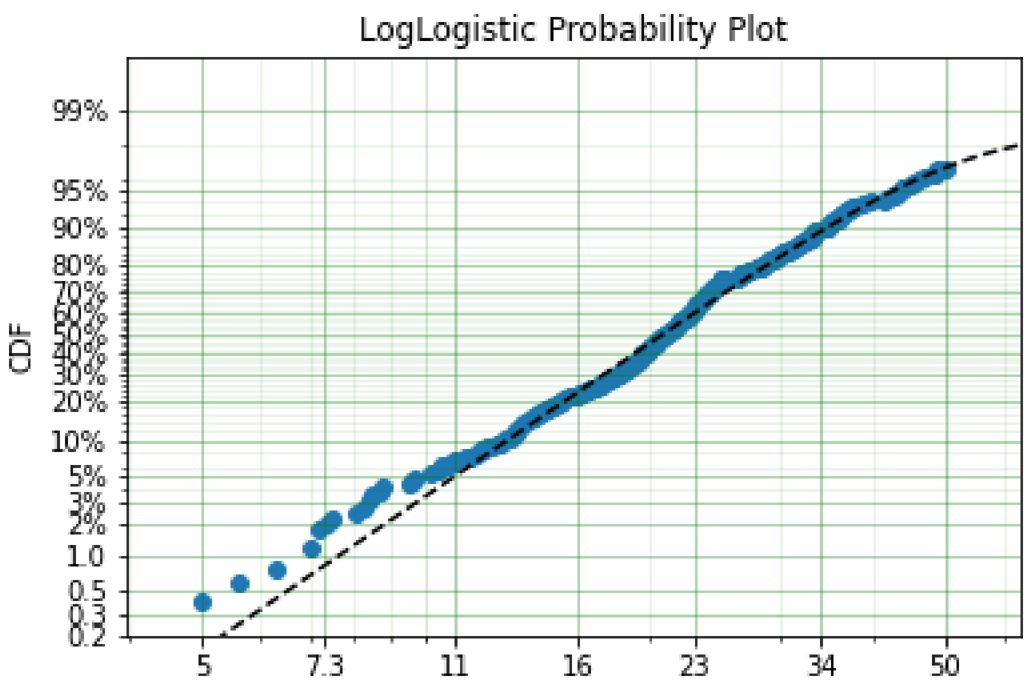
```
Parametric SurPyval Model
=========================
Distribution        : Weibull
Fitted by           : MLE
Parameters          :
     alpha: 25.53669601993307
      beta: 2.469159446459548
```



We can see that the model has changed slightly, however, there appears to be a ‘disconnect’ near 24 that makes the model a poor fit above and below the value. Let’s see if a different distribution will improve the fit:

```python
import surpyval as surv
x = Boston.df['medv'].values
x, c, n = surv.xcn_handler(x)
# Right censor the highest value
c[-1] = 1

model = surv.LogLogistic.fit(x=x, c=c, n=n, lfp=True)
print(model)
model.plot()
```

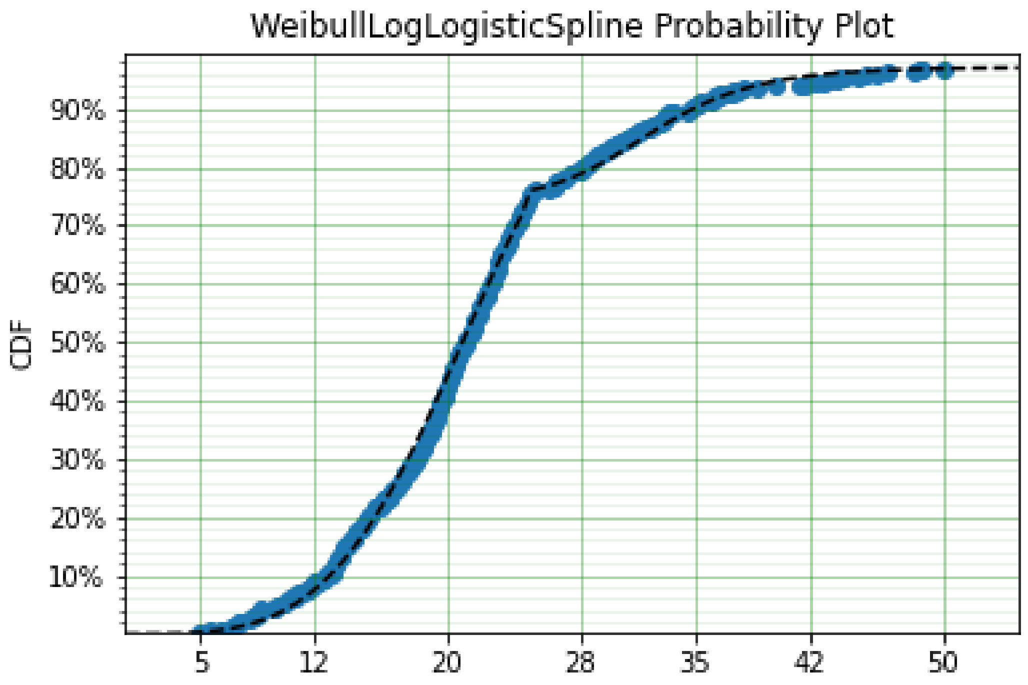
```
Parametric SurPyval Model
=========================
Distribution        : LogLogistic
Fitted by           : MLE
Max Proportion (p)  : 0.9861133787129936
Parameters          :
     alpha: 20.804405478058186
      beta: 4.56190516414644
```



This appears to be a much better fit, however, there is still quite a bit of difference between the data and the model in the middle of the distribution. Lets create a custom spline to see if we can perfect the fit.

```python
import surpyval as surv
x = Boston.df['medv'].values
x, c, n = surv.xcn_handler(x)
# Right censor the highest value
c[-1] = 1

def Hf(x, *params):
    x = np.array(x)
    Hf = np.zeros_like(x)
    knot = params[0]
    params = params[1:]
    dist1 = surv.Weibull
    dist2 = surv.LogLogistic
    Hf = np.where(x < knot, dist1.Hf(x, *params[0:2]), Hf)
    Hf = np.where(x >= knot, (dist1.Hf(knot, *params[0:2])
                              + dist2.Hf(x, *params[2::])), Hf)
    return Hf
bounds = ((0, 50), (0, None), (0, None), (0, None), (0, None),)
param_names = ['knot', 'alpha_w', 'beta_w', 'alpha_ll', 'beta_ll']
name = 'WeibullLogLogisticSpline'
support = (0, np.inf)

WeibullLogLogisticSpline = surv.Distribution(name, Hf, param_names, bounds, support)

model = WeibullLogLogisticSpline.fit(x=x, c=c, n=n, lfp=True)

print(model)
model.plot()
```

```
Parametric SurPyval Model
=========================
Distribution        : WeibullLogLogisticSpline
Fitted by           : MLE
Max Proportion (p)  : 0.9711459340639835
Parameters          :
      knot: 25.0000103742294
   alpha_w: 22.735658691657452
    beta_w: 3.926996942307611
  alpha_ll: 32.2716411336919
   beta_ll: 10.120540049344006
```



Much better!

It must be said that this is a bit ‘hacky’. There is no theory that we are using to guide the choice of the spline model, we are simply finding the best fit to the data. For example, this model would not able to be used for extrapolation too far beyond $50,0000, this is because the model is limited to 97.1% of houses. A separate spline would be needed to model those data. However, the example shows the importance of censoring and the power of the surpyval API!

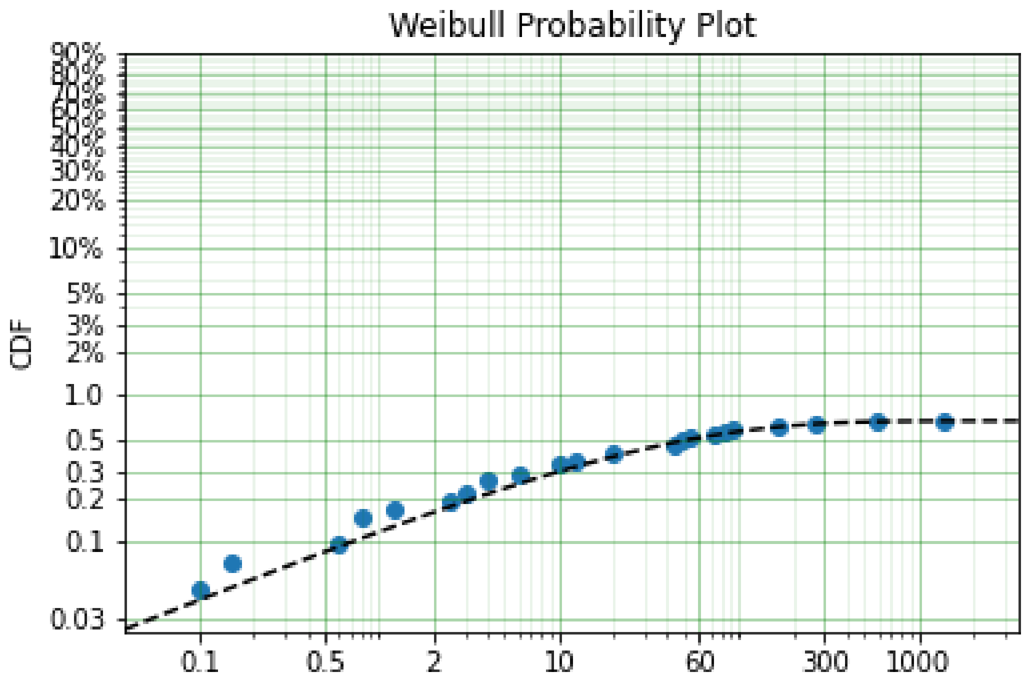
1.12.2 Applided Reliability Engineering

In reliability engineering we might be interested in the proportion of a population that will experience a particular failure mode. We do not want to ship the items that will fail so that our customers do not have a poor experience. But, we will want to determine the minimum duration of a test that can establish whether a component will fail. This is because a test that is too long we will waste time and money in testing and if a test is too short we will ship too many items that will fail in the field. We need to optimise this interval the minimize the cost of testing but also the number of items at risk in the field.

Using data from the paper that introduced the Limited Failure Population model (also known as the Defective Subpopulation) to the reliability engineering world [Meeker] we can show how surpyval can be used in part to calculate an optimal ‘burn-in’ test duration.

```
import surpyval as surv

f = [.1, .1, .15, .6, .8, .8, 1.2, 2.5, 3., 4., 4., 6., 10., 10.,
     12.5, 20., 20., 43., 43., 48., 48., 54., 74., 84., 94., 168., 263., 593.]
s = [1370.] * 4128

x, c, n = surv.fs_to_xcn(f, s)
model = surv.Weibull.fit(x, c, n, lfp=True)
print(model)
model.plot()
```

```
Parametric SurPyval Model
=========================
Distribution        : Weibull
Fitted by           : MLE
Max Proportion (p)  : 0.006744450944727198
Parameters          :
     alpha: 28.367193779799038
      beta: 0.4959762140288241
```



We can see from these results that at maximum we will have approximately 0.67% fail. If the company accepts a 0.1% probability of their products failing in the field then we can calculate the interval at which the difference between the total population and the proportion failed in the test is 0.1%.

```
from scipy.optimize import minimize
fun = lambda x : (0.001 - np.abs(model.p - model.ff(x)))**2

res = minimize(fun, 10, tol=1e-50)
print(res.x)
```

```
[104.43741352]
```

Therefore we should do a burn in test up to approximately 104.4 to make sure we minimize the number of items shipped that are defective while also minimizing the duration of the test. We can simply change the value of `0.001` in the above code to any value we may wish to use.

1.12.3 Demographics / Actuarial

In demographics and actuarial studies, the distribution of the life of a population is of interest. For the demographer, it is necessary to understand how a population might change, in particular, how the expected lifespan is changing over time. The same applies to an actuary, an actuary is interested in lifetimes to understand the risk of payouts among those who own a life insurance policy.

The Gompertz-Makeham is a distribution used in demography and actuarial studies to estimate the lifetime of a population. This can be implemented in surpyval with relative ease.

```python
import surpyval as surv
from autograd import numpy as np
from matplotlib import pyplot as plt
from scipy.special import lambertw

bounds = ((0, None), (0, None), (0, None),)
support = (0, np.inf)
param_names = ['lambda', 'alpha', 'beta']
def Hf(x, *params):
    Hf = params[0] * x + (params[1]/params[2])*(np.exp(params[2]*x))
    return Hf

GompertzMakeham = surv.Distribution('GompertzMakeham', Hf, param_names, bounds,␣
↪support)
```

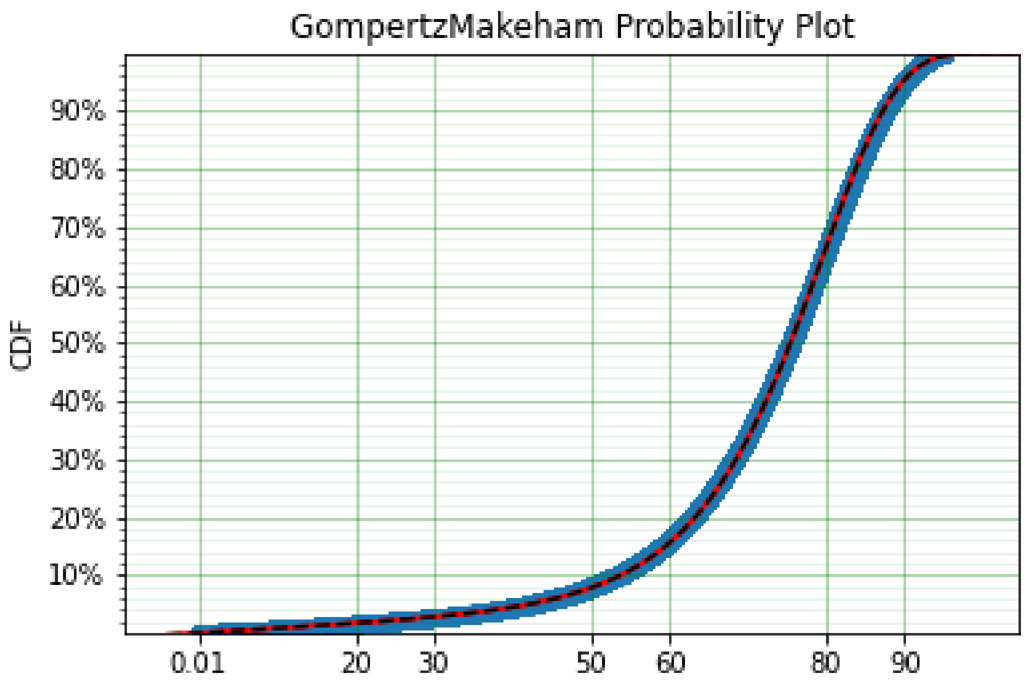
We now have a GM distribution object that can be used to fit data. But we need some data:

```python
# GM qf()
def qf(p, params):
    lambda_ = params[0]
    alpha = params[1]
    beta = params[2]
    return (alpha/(lambda_ * beta) - (1./lambda_)*np.log(1 - p)
            - (1./beta)*lambertw((alpha*np.exp(alpha/lambda_)*(1 - p)**(-(beta/lambda_
↪)))/(lambda_))).real


np.random.seed(1)
params = np.array([.68, 28.7e-3, 102.3])/1000
x = qf(np.random.uniform(0, 1, 100_000), params)
# Filter out some numeric overflows.
x = x[np.isfinite(x)]
```

The parameters for the distribution come from [Gavrilov], specifically the parameters for the lifespans of the 1974-1978 data. So in this case we have (simulated) data on the lifespans of 100,000 thousand people and we need to determine the GM parameters. This can be compared to the historic parameters to see if the age related mortality has changed or has remained roughly constant. To do so, all we need do with surpyval is to put the data to the `fit()` method.

```
model = GompertzMakeham.fit(x)
model.plot(alpha_ci=0.99, heuristic='Nelson-Aalen')
model
```

```
Parametric SurPyval Model
=========================
Distribution        : GompertzMakeham
Fitted by           : MLE
Parameters          :
    lambda: 0.0007827108147066848
     alpha: 2.1199267751549727e-05
      beta: 0.10690878152126947
```



You can see that the model is a good fit to the data. Using the model we can determine the probability of death in a given term for a random individual from the population. This is useful to price the premium of a life insurance policy. For example, if a 60 year old was to take out a two year policy, what premium should we charge them for the policy. First, we need to determine the probability of death:

```
p_death = model.ff(62) - model.ff(60)
policy_payout = 100_000
expected_loss = policy_payout * p_death
print(p_death, expected_loss)
```

```
0.025337351289907883 2533.7351289907883
```

From the results above, you can see that the probability of death over the two year interval is approximately 2.5%. Given the contract is to payout $100,000 in this event, the expected loss is therefore $2,533.74. Therefore, to make a profit, the policy will need to cost more than $2,533.74. So say the company has a strategy of making 10% from each policy, the policy cost to the individual would therefore be $2,787.11. If we divide this payment scheme into a per month basis over the two years we get a monthly payment of $116.13 for two years (in the case of death the amount owing can be subtracted from the payout).

Although this is a basic example, as insurance companies would have much more sophisticated models, it shows the basics of how demographic and actuarial data can be used. This shows the application of surpyval to actuarial and demogrphic studies.

1.12.4 Applied Reliability Engineering - 2

In reliability engineering you can come across the case where a new product has been built that is similar in design to a previous, but has better materias, geometry, seals.. etc. You have data from the tests of the old product and new results for the same test on the new product. The only problem, the new product only had one failure in the test! What will you do?

Given the similarities, it is common to use the same shape parameter, the β value, from a similar product as an initial estimate. In this case, we may need to know the reliability of the item in the field. We can create a model of this new product, but first the old product:

```
import surpyval as surv

x_old = [ 5.2, 10.7, 16.3, 22. , 32.9, 38.6, 42.1, 58.7, 92.8, 93.8]
old = surv.Weibull.fit(x_old)
print(old)
```

```
Parametric SurPyval Model
=========================
Distribution        : Weibull
Fitted by           : MLE
Parameters          :
     alpha: 45.27418484669478
      beta: 1.377623372184365
```

We can use the above value of beta with the new data:

```
x_new = [87, 100]
c_new = [0, 1]
n_new = [1, 9]

surv.Weibull.fit(x_new, c_new, n_new, fixed={'beta' : 1.3776}, init=[100.])
```

```
Parametric SurPyval Model
=========================
Distribution        : Weibull
Fitted by           : MLE
Parameters          :
     alpha: 525.1398140084557
      beta: 1.3776
```

The characteristic life of the new bearing is over 10 times higher! Quite an improved new design. This new model can be used as part of the sales of the new product (10x more life!) and to provide recommendations for maintenance.

1.12.5 Social Science / Criminology

Another application of surpyval is when encountering extreme values. The Weibull distribution is one of the limiting cases of the Generalized Extreme Value distribution. In other words, the Weibull distribution is the distribution that can model the strength of a chain because it can model the extreme value, in this case the minimum, of a collection of distributions. A chain is as only as strong as it's weakest link. If there are many many links in a chain (which is a fair assumption) then links of which follow a known strength distribution, then the strength of the chain will will follow a Weibull distribution. It is for this reason that the Weibull distribution is so widely used.

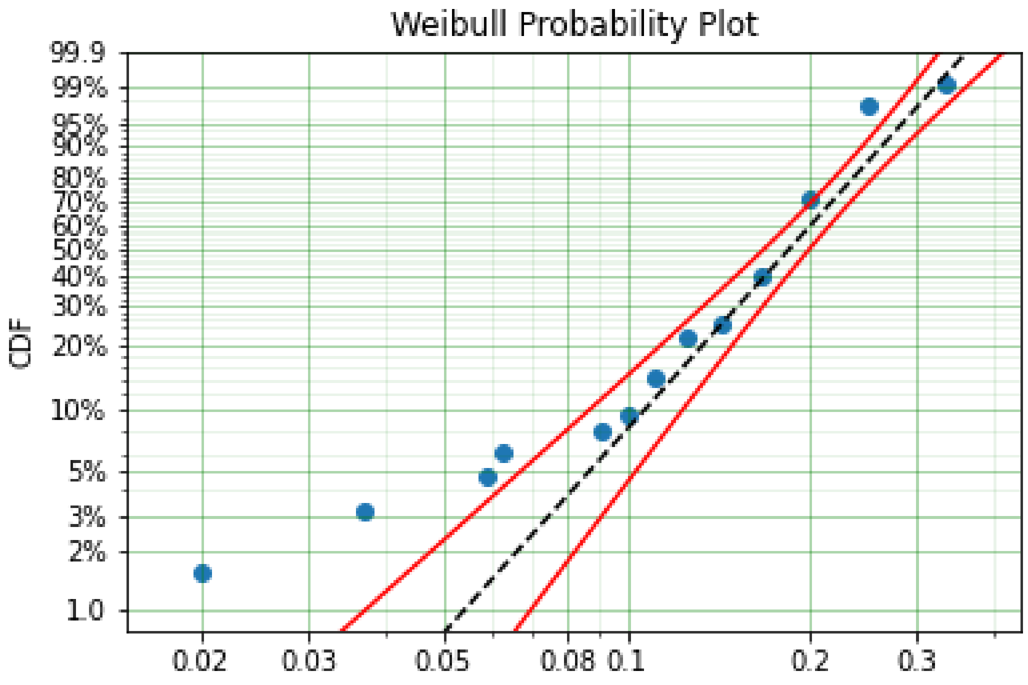
Another extreme value is the maximum. The maximum extreme value distribution is the Frechet distribution. But, if you simply inverse a minimum, you can get a maxmimum. Therefore, if we know our data is following a process of finding a maximum, then we can use the Weibull distribution to model the phenonmena.

Warning: This may be a distressing topic for some readers.

Social scientists and criminologists are interested in understanding the phenomena of mass shootings in an effort to eliminate the scourge from society. A mass shooting is an extreme event, and an extreme event can be modelled to understand the risks of future occurence, and with that understanding, the effect of interventions can also be understood.

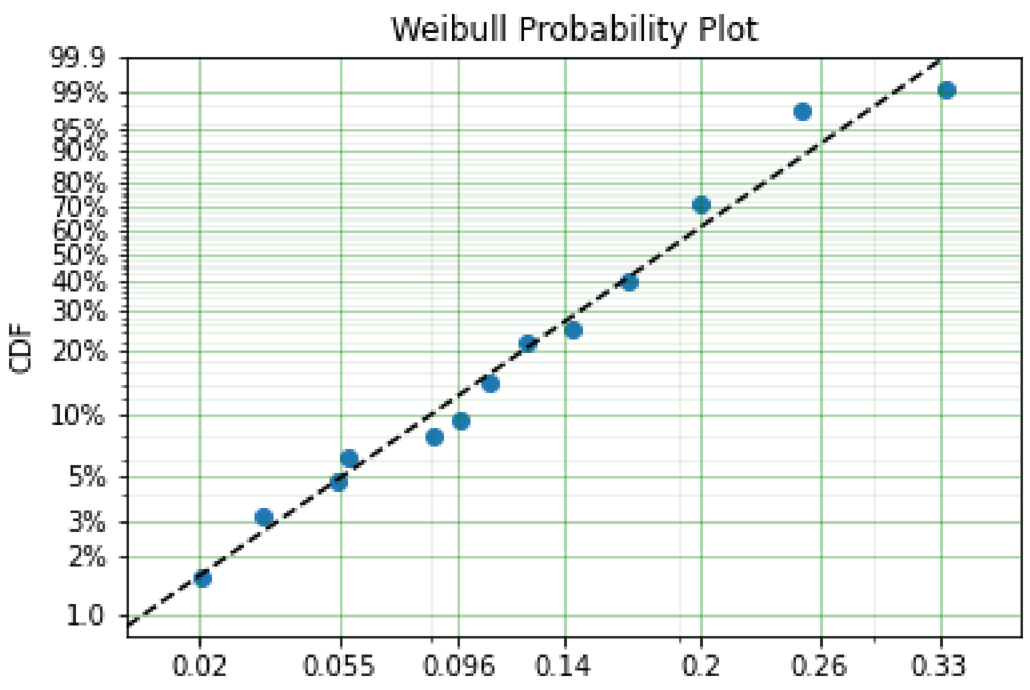
Using the gun violence data from Kaggle we can model the process. That is, if we take the maximum number of deaths in a given month over several years, we have data that can be used to estimate the probability of something even worse occuring. This data covers the period from 2013 to 2018, see Kaggle for more details.

```
import surpyval as surv
import pandas as pd

# Data not in surpyval, available at https://www.kaggle.com/jameslko/gun-violence-data
gun_violence_df = pd.read_csv('../gun-violence-data_01-2013_03-2018.csv', parse_
↪dates=['date'])

# Find the maximum number of people killed each month
gun_violence_df = gun_violence_df.groupby(pd.Grouper(key='date', freq='M')).agg({'n_
↪killed' : 'max'})

x = df['n_killed'].values

# Inverse the data to get the maximum
model = surv.Weibull.fit(1./x)
model.plot()
```



It is worth reminding that since we have taken the inverse, it is the lower values that represent more victims. And it is the extremes that we are trying to capture. You can see from the above plot that the model does not fit the data from 0.02 to 0.1 very well. We can try using a different approach

```
# Inverse the data to get the maximum
mpp_model = surv.Weibull.fit(1./x, how='MPP', offset=True)
mpp_model.plot()
```



You can see that this model is a much better description of the data. However, the problem is that it cannot have a real interpretation. Because the offset is negative, that means there is a non-zero probability of 0, which because the data was inversed, means that there is a non-zero probability of having a shooting with infinite victims. This model is therefore not a good option for such extreme extrapolations. The model can however, be used to estimate the probabiltiy of having a shooting as bad or worse than the most extreme event up to 2040.

```
p_happening = mpp_model.ff(1./50)
p_not_happening = 1 - p_happening
# Months from 2022 to 2040
months = 12 * (2040 - 2022)
p_not_happening_before_2040 = (p_not_happening)**(months)
(1 - p_not_happening_before_2040)*100
```

```
(1.6077640040390584, 96.98325003600236)
```

The model estimates that there is an approximately 1.6% chance of an event killing 50 or more people in a given month, which may seem low, however, because there are 216 months between 2022 and 2040 the chances of not having as extreme an event over that time period becomes horrifyingly small. The model suggests that the probability of having a month in which an event with more than 50 people will be killed, has a 97.0% chance of happening from 2022 to 2040. Chilling.

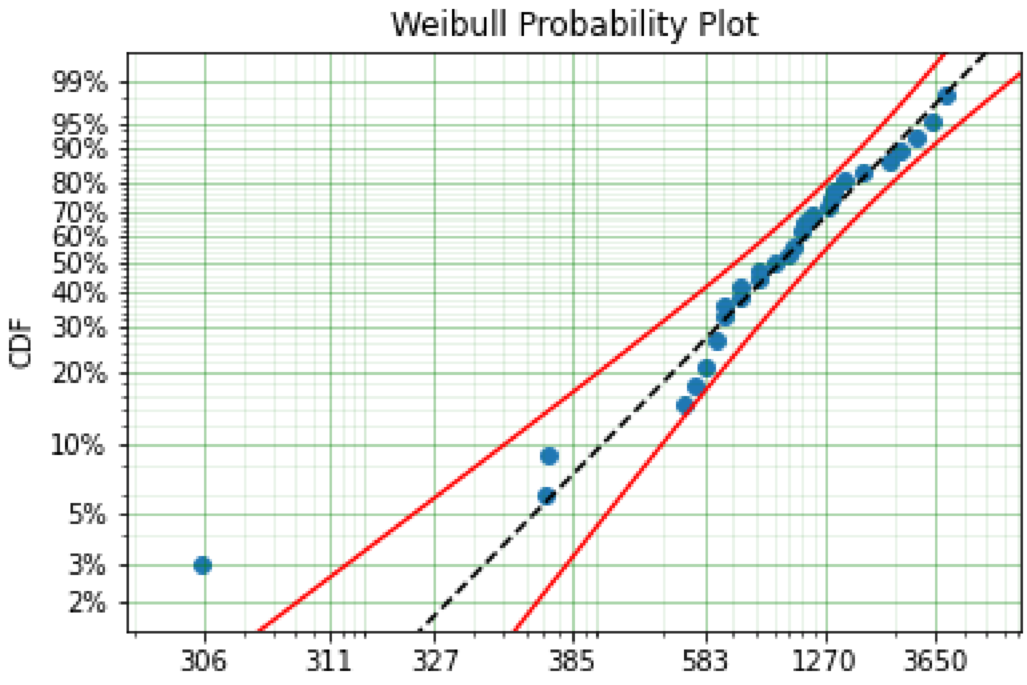
This is a bit higher than other reports of the same prediction, see [Duwe] who report at 35% probability, which is some, but not even close to complete, relief.

1.12.6 Economics

Economists are interested in the times between recessions. This information helps them formulate policy proscriptions that may (or may not) reduce the duration of a recession, or the time between recessions. Using data from Tadeu Cristino et al. [TC] we can use real data to esimate the probability of a recession.

```
import pandas as pd
import surpyval as surv

start = [np.nan, "June 1857", "October 1860", "April 1865", "June 1869",
         "October 1873", "March 1882", "March 1887", "July 1890", "January 1893",
         "December 1895", "June 1899", "September 1902", "May 1907", "January 1910",
         "January 1913", "August 1918", "January 1920", "May 1923", "October 1926",
         "August 1929", "May 1937", "February 1945", "November 1948", "July 1953",
         "August 1957", "April 1960", "December 1969", "November 1973",
         "January 1980", "July 1981", "July 1990", "March 2001", "December 2007"]


end = [
    "December 1854", "December 1858", "June 1861", "December 1867", "December 1870",
    "March 1879", "May 1885", "April 1888", "May 1891", "June 1894", "June 1897",
    "December 1900", "August 1904", "June 1908", "January 1912", "December 1914",
    "March 1919", "July 1921", "July 1924", "November 1927", "March 1933",
    "June 1938", "October 1945", "October 1949", "May 1954", "April 1958",
    "February 1961", "November 1970", "March 1975", "July 1980", "November 1982",
    "March 1991", "November 2001", "June 2009"
]

df = pd.DataFrame({'start' : pd.to_datetime(start),
                   'end' : pd.to_datetime(end)})

# Compute time from end of last recession to peak of next.
x = (df.start - df.end.shift(1)).dropna().dt.days.values

model = surv.Weibull.fit(x, offset=True)
print(model)
model.plot()
```

```
Parametric SurPyval Model
=========================
Distribution         : Weibull
Fitted by            : MLE
Offset (gamma)       : 304.0659320899125
Parameters           :
     alpha: 895.3220718605215
      beta: 1.0629492868473804
```



You can see from the above the data is a good fit to the model! Great. So now what?

We can communicate what the expected time between recessions is:

```
model.mean()
```

```
1178.2499033086633
```

Therefore the average growth period is 1,178 days, or about 3.2 years between recessions.

1.12.7 References

1.13 API

1.13.1 Non-Parametric

class surpyval.nonparametric.nonparametric.**NonParametric**

Bases: `object`

Result of `.fit()` method for every non-parametric surpyval distribution. This means that each of the methods in this class can be called with a model created from the `NelsonAalen`, `KaplanMeier`, `FlemingHarrington`, or `Turnbull` estimators.

Hf (*x*, *interp='step'*)

Cumulative hazard rate with the non-parametric estimates from the data. This is calculated using the relationship between the hazard function and the density:

$$H(x) = -\ln(R(x))$$

Parameters **x** (*array like or scalar*) – The values of the random variables at which the survival function will be calculated

Returns **Hf** – The value(s) of the density function at x

Return type scalar or numpy array

Examples

```
>>> from surpyval import NelsonAalen
>>> x = np.array([1, 2, 3, 4, 5])
>>> model = NelsonAalen.fit(x)
>>> model.Hf(2)
array([0.45])
>>> model.df([1., 1.5, 2., 2.5])
model.Hf([1., 1.5, 2., 2.5])
```

cb (*x*, *on='sf'*, *bound='two-sided'*, *interp='step'*, *alpha_ci=0.05*, *bound_type='exp'*, *dist='z'*)

Confidence bounds of the `on` function at the `alpa_ci` level of significance. Can be the upper, lower, or two-sided confidence by changing value of `bound`. Can change the bound type to be regular or exponential using either the 't' or 'z' statistic.

Parameters

- **x** (*array like or scalar*) – The values of the random variables at which the confidence bounds will be calculated
- **on** (*('sf', 'ff', 'Hf'), optional*) – The function on which the confidence bound will be calculated.
- **bound** (*('two-sided', 'upper', 'lower'), str, optional*) – Compute either the two-sided, upper or lower confidence bound(s). Defaults to two-sided.
- **interp** (*('step', 'linear', 'cubic'), optional*) – How to interpolate the values between observations. Survival statistics traditionally uses step functions, but can use interpolated values if desired. Defaults to step.
- **alpha_ci** (*scalar, optional*) – The level of significance at which the bound will be computed.
- **bound_type** (*('exp', 'regular'), str, optional*) – The method with which the bounds will be calculated. Using regular will allow for the bounds to exceed 1 or be less than 0. Defaults to exp as this ensures the bounds are within 0 and 1.
- **dist** (*('t', 'z'), str, optional*) – The statistic to use in calculating the bounds (student-t or normal). Defaults to the normal (z).

Returns **cb** – The value(s) of the upper, lower, or both confidence bound(s) of the selected function at x

Return type scalar or numpy array

Examples

```
>>> from surpyval import NelsonAalen
>>> x = np.array([1, 2, 3, 4, 5])
>>> model = NelsonAalen.fit(x)
>>> model.cb([1., 1.5, 2., 2.5], bound='lower', dist='t')
```

(continues on next page)

(continued from previous page)

```
array([0.11434813, 0.11434813, 0.04794404, 0.04794404])
>>> model.cb([1., 1.5, 2., 2.5])
array([[0.97789387, 0.16706394],
       [0.97789387, 0.16706394],
       [0.91235117, 0.10996882],
       [0.91235117, 0.10996882]])
```

References

http://reliawiki.org/index.php/Non-Parametric_Life_Data_Analysis

df (*x*, *interp='step'*)

Density function with the non-parametric estimates from the data. This is calculated using the relationship between the hazard function and the density:

$$f(x) = h(x)e^{-H(x)}$$

Parameters x (*array like or scalar*) – The values of the random variables at which the survival function will be calculated

Returns df – The value(s) of the density function at x

Return type scalar or numpy array

Examples

```
>>> from surpyval import NelsonAalen
>>> x = np.array([1, 2, 3, 4, 5])
>>> model = NelsonAalen.fit(x)
>>> model.df(2)
array([0.28693267])
>>> model.df([1., 1.5, 2., 2.5])
array([0.16374615, 0.        , 0.15940704, 0.        ])
```

ff (*x*, *interp='step'*)

CDF (failure or unreliability) function with the non-parametric estimates from the data

Parameters x (*array like or scalar*) – The values of the random variables at which the survival function will be calculated

Returns ff – The value(s) of the failure function at each x

Return type scalar or numpy array

Examples

```
>>> from surpyval import NelsonAalen
>>> x = np.array([1, 2, 3, 4, 5])
>>> model = NelsonAalen.fit(x)
>>> model.ff(2)
array([0.36237185])
>>> model.ff([1., 1.5, 2., 2.5])
array([0.18126925, 0.18126925, 0.36237185, 0.36237185])
```

hf (*x*, *interp='step'*)

Instantaneous hazard function with the non-parametric estimates from the data. This is calculated using simply the difference between consecutive H(x).

Parameters **x** (*array like or scalar*) – The values of the random variables at which the survival function will be calculated

Returns **hf** – The value(s) of the failure function at each x

Return type scalar or numpy array

Examples

```
>>> from surpyval import NelsonAalen
>>> x = np.array([1, 2, 3, 4, 5])
>>> model = NelsonAalen.fit(x)
>>> model.ff(2)
array([0.36237185])
>>> model.ff([1., 1.5, 2., 2.5])
array([0.18126925, 0.18126925, 0.36237185, 0.36237185])
```

plot (***kwargs*)

Creates a plot of the survival function.

sf (*x*, *interp='step'*)

Surival (or Reliability) function with the non-parametric estimates from the data

Parameters **x** (*array like or scalar*) – The values of the random variables at which the survival function will be calculated

Returns **sf** – The value(s) of the survival function at each x

Return type scalar or numpy array

Examples

```
>>> from surpyval import NelsonAalen
>>> x = np.array([1, 2, 3, 4, 5])
>>> model = NelsonAalen.fit(x)
>>> model.sf(2)
array([0.63762815])
>>> model.sf([1., 1.5, 2., 2.5])
array([0.81873075, 0.81873075, 0.63762815, 0.63762815])
```

class surpyval.nonparametric.kaplan_meier.**KaplanMeier_**

Bases: surpyval.nonparametric.nonparametric_fitter.NonParametricFitter

Kaplan-Meier estimator class. Calculates the Non-Parametric estimate of the survival function using:

$$R(x) = \prod_{i:x_i \leq x} \left(1 - \frac{d_i}{r_i}\right)$$

Examples

```
>>> import numpy as np
>>> from surpyval import KaplanMeier
>>> x = np.array([1, 2, 3, 4, 5])
>>> model = KaplanMeier.fit(x)
>>> model.R
array([0.8, 0.6, 0.4, 0.2, 0. ])
```

fit(*x=None*, *c=None*, *n=None*, *t=None*, *xl=None*, *xr=None*, *tl=None*, *tr=None*, *turnbull_estimator='Fleming-Harrington'*)

The central feature to SurPyval's capability. This function aimed to have an API to mimic the simplicity of the scipy API. That is, to use a simple `fit()` call, with as many or as few parameters as is needed.

Parameters

- **x** (*array like, optional*) – Array of observations of the random variables. If x is `None`, xl and xr must be provided.
- **c** (*array like, optional*) – Array of censoring flag. -1 is left censored, 0 is observed, 1 is right censored, and 2 is intervally censored. If not provided will assume all values are observed.
- **n** (*array like, optional*) – Array of counts for each x. If data is proivded as counts, then this can be provided. If `None` will assume each observation is 1.
- **t** (*2D-array like, optional*) – 2D array like of the left and right values at which the respective observation was truncated. If not provided it assumes that no truncation occurs.
- **tl** (*array like or scalar, optional*) – Values of left truncation for observations. If it is a scalar value assumes each observation is left truncated at the value. If an array, it is the respective 'late entry' of the observation
- **tr** (*array like or scalar, optional*) – Values of right truncation for observations. If it is a scalar value assumes each observation is right truncated at the value. If an array, it is the respective right truncation value for each observation
- **xl** (*array like, optional*) – Array like of the left array for 2-dimensional input of x. This is useful for data that is all intervally censored. Must be used with the `xr` input.
- **xr** (*array like, optional*) – Array like of the right array for 2-dimensional input of x. This is useful for data that is all intervally censored. Must be used with the `xl` input.
- **turnbull_estimator** (*('Nelson-Aalen', 'Kaplan-Meier', or 'Fleming-Harrington'), str, optional*) – If using the Turnbull heuristic, you can elect to use either the KM, NA, or FH estimator with the Turnbull estimates of r, and d. Defaults to FH.

Returns **model** – A parametric model with the fitted parameters and methods for all functions of the distribution using the fitted parameters.

Return type *NonParametric*

Examples

```
>>> from surpyval import NelsonAalen, Weibull, Turnbull
>>> import numpy as np
>>> x = Weibull.random(100, 10, 4)
>>> model = NelsonAalen.fit(x)
>>> print(model)
```

(continues on next page)

(continued from previous page)

```
Non-Parametric SurPyval Model
=============================
Model              : Nelson-Aalen
>>> Turnbull.fit(x, turnbull_estimator='Kaplan-Meier')
Non-Parametric SurPyval Model
=============================
Model              : Turnbull
Estimator          : Kaplan-Meier
```

class `surpyval.nonparametric.nelson_aalen.`**`NelsonAalen_`**

Bases: `surpyval.nonparametric.nonparametric_fitter.NonParametricFitter`

Nelson-Aalen estimator class. Returns a *NonParametric* object from method `fit()` Calculates the Non-Parametric estimate of the survival function using:

$$R(x) = e^{-\sum_{i:x_{i} \leq x} \frac{d_{i}}{r_{i}}}$$

Examples

```
>>> import numpy as np
>>> from surpyval import NelsonAalen
>>> x = np.array([1, 2, 3, 4, 5])
>>> model = NelsonAalen.fit(x)
>>> model.R
array([0.81873075, 0.63762815, 0.45688054, 0.27711205, 0.10194383])
```

fit(*x=None, c=None, n=None, t=None, xl=None, xr=None, tl=None, tr=None, turnbull_estimator='Fleming-Harrington'*)

The central feature to SurPyval's capability. This function aimed to have an API to mimic the simplicity of the scipy API. That is, to use a simple `fit()` call, with as many or as few parameters as is needed.

Parameters

- **x** (*array like, optional*) – Array of observations of the random variables. If x is `None`, xl and xr must be provided.
- **c** (*array like, optional*) – Array of censoring flag. -1 is left censored, 0 is observed, 1 is right censored, and 2 is intervally censored. If not provided will assume all values are observed.
- **n** (*array like, optional*) – Array of counts for each x. If data is proivded as counts, then this can be provided. If `None` will assume each observation is 1.
- **t** (*2D-array like, optional*) – 2D array like of the left and right values at which the respective observation was truncated. If not provided it assumes that no truncation occurs.
- **tl** (*array like or scalar, optional*) – Values of left truncation for observations. If it is a scalar value assumes each observation is left truncated at the value. If an array, it is the respective 'late entry' of the observation
- **tr** (*array like or scalar, optional*) – Values of right truncation for observations. If it is a scalar value assumes each observation is right truncated at the value. If an array, it is the respective right truncation value for each observation
- **xl** (*array like, optional*) – Array like of the left array for 2-dimensional input of x. This is useful for data that is all intervally censored. Must be used with the `xr` input.

- **xr** (*array like, optional*) – Array like of the right array for 2-dimensional input of x. This is useful for data that is all intervally censored. Must be used with the `xl` input.
- **turnbull_estimator** (*('Nelson-Aalen', 'Kaplan-Meier', or 'Fleming-Harrington'), str, optional*) – If using the Turnbull heuristic, you can elect to use either the KM, NA, or FH estimator with the Turnbull estimates of r, and d. Defaults to FH.

Returns **model** – A parametric model with the fitted parameters and methods for all functions of the distribution using the fitted parameters.

Return type *NonParametric*

Examples

```
>>> from surpyval import NelsonAalen, Weibull, Turnbull
>>> import numpy as np
>>> x = Weibull.random(100, 10, 4)
>>> model = NelsonAalen.fit(x)
>>> print(model)
Non-Parametric SurPyval Model
=============================
Model                : Nelson-Aalen
>>> Turnbull.fit(x, turnbull_estimator='Kaplan-Meier')
Non-Parametric SurPyval Model
=============================
Model                : Turnbull
Estimator            : Kaplan-Meier
```

class `surpyval.nonparametric.fleming_harrington.`**`FlemingHarrington_`**

Bases: `surpyval.nonparametric.nonparametric_fitter.NonParametricFitter`

Fleming-Harrington estimation of survival distribution. Returns a *NonParametric* object from method `fit()` Calculates the Non-Parametric estimate of the survival function using:

$$R = e^{-\sum_{i:x_i \leq x} \sum_{i=0}^{d_x-1} \frac{1}{r_x - i}}$$

See 'NonParametric section for detailed estimate of how H is computed.'

Examples

```
>>> import numpy as np
>>> from surpyval import FlemingHarrington
>>> x = np.array([1, 2, 3, 4, 5])
>>> model = FlemingHarrington.fit(x)
>>> model.R
array([0.81873075, 0.63762815, 0.45688054, 0.27711205, 0.10194383])
```

fit(*x=None, c=None, n=None, t=None, xl=None, xr=None, tl=None, tr=None, turnbull_estimator='Fleming-Harrington'*)

The central feature to SurPyval's capability. This function aimed to have an API to mimic the simplicity of the scipy API. That is, to use a simple `fit()` call, with as many or as few parameters as is needed.

Parameters

- **x** (*array like, optional*) – Array of observations of the random variables. If x is `None`, xl and xr must be provided.

- **c** (*array like, optional*) – Array of censoring flag. -1 is left censored, 0 is observed, 1 is right censored, and 2 is intervally censored. If not provided will assume all values are observed.
- **n** (*array like, optional*) – Array of counts for each x. If data is proivded as counts, then this can be provided. If `None` will assume each observation is 1.
- **t** (*2D-array like, optional*) – 2D array like of the left and right values at which the respective observation was truncated. If not provided it assumes that no truncation occurs.
- **tl** (*array like or scalar, optional*) – Values of left truncation for observations. If it is a scalar value assumes each observation is left truncated at the value. If an array, it is the respective 'late entry' of the observation
- **tr** (*array like or scalar, optional*) – Values of right truncation for observations. If it is a scalar value assumes each observation is right truncated at the value. If an array, it is the respective right truncation value for each observation
- **xl** (*array like, optional*) – Array like of the left array for 2-dimensional input of x. This is useful for data that is all intervally censored. Must be used with the `xr` input.
- **xr** (*array like, optional*) – Array like of the right array for 2-dimensional input of x. This is useful for data that is all intervally censored. Must be used with the `xl` input.
- **turnbull_estimator** (*('Nelson-Aalen', 'Kaplan-Meier', or 'Fleming-Harrington'), str, optional*) – If using the Turnbull heuristic, you can elect to use either the KM, NA, or FH estimator with the Turnbull estimates of r, and d. Defaults to FH.

Returns model – A parametric model with the fitted parameters and methods for all functions of the distribution using the fitted parameters.

Return type *NonParametric*

Examples

```
>>> from surpyval import NelsonAalen, Weibull, Turnbull
>>> import numpy as np
>>> x = Weibull.random(100, 10, 4)
>>> model = NelsonAalen.fit(x)
>>> print(model)
Non-Parametric SurPyval Model
=============================
Model              : Nelson-Aalen
>>> Turnbull.fit(x, turnbull_estimator='Kaplan-Meier')
Non-Parametric SurPyval Model
=============================
Model              : Turnbull
Estimator          : Kaplan-Meier
```

class `surpyval.nonparametric.turnbull.`**`Turnbull_`**

Bases: `surpyval.nonparametric.nonparametric_fitter.NonParametricFitter`

Turnbull estimator class. Returns a *NonParametric* object from method `fit()`. Calculates the Non-Parametric estimate of the survival function using the Turnbull NPMLE

Examples

```
>>> import numpy as np
>>> from surpyval import Turnbull
>>> x = np.array([[1, 5], [2, 3], [3, 6], [1, 8], [9, 10]])
>>> model = Turnbull.fit(x)
>>> model.R
array([1.        , 0.59999999, 0.20000002, 0.2       , 0.2       ,
       0.2       , 0.        , 0.        ])
```

fit (*x=None*, *c=None*, *n=None*, *t=None*, *xl=None*, *xr=None*, *tl=None*, *tr=None*, *turnbull_estimator='Fleming-Harrington'*)

The central feature to SurPyval's capability. This function aimed to have an API to mimic the simplicity of the scipy API. That is, to use a simple `fit()` call, with as many or as few parameters as is needed.

Parameters

- **x** (*array like, optional*) – Array of observations of the random variables. If x is `None`, xl and xr must be provided.
- **c** (*array like, optional*) – Array of censoring flag. -1 is left censored, 0 is observed, 1 is right censored, and 2 is intervally censored. If not provided will assume all values are observed.
- **n** (*array like, optional*) – Array of counts for each x. If data is proivded as counts, then this can be provided. If `None` will assume each observation is 1.
- **t** (*2D-array like, optional*) – 2D array like of the left and right values at which the respective observation was truncated. If not provided it assumes that no truncation occurs.
- **tl** (*array like or scalar, optional*) – Values of left truncation for observations. If it is a scalar value assumes each observation is left truncated at the value. If an array, it is the respective 'late entry' of the observation
- **tr** (*array like or scalar, optional*) – Values of right truncation for observations. If it is a scalar value assumes each observation is right truncated at the value. If an array, it is the respective right truncation value for each observation
- **xl** (*array like, optional*) – Array like of the left array for 2-dimensional input of x. This is useful for data that is all intervally censored. Must be used with the `xr` input.
- **xr** (*array like, optional*) – Array like of the right array for 2-dimensional input of x. This is useful for data that is all intervally censored. Must be used with the `xl` input.
- **turnbull_estimator** (*('Nelson-Aalen', 'Kaplan-Meier', or 'Fleming-Harrington'), str, optional*) – If using the Turnbull heuristic, you can elect to use either the KM, NA, or FH estimator with the Turnbull estimates of r, and d. Defaults to FH.

Returns **model** – A parametric model with the fitted parameters and methods for all functions of the distribution using the fitted parameters.

Return type *NonParametric*

Examples

```
>>> from surpyval import NelsonAalen, Weibull, Turnbull
>>> import numpy as np
>>> x = Weibull.random(100, 10, 4)
>>> model = NelsonAalen.fit(x)
>>> print(model)
Non-Parametric SurPyval Model
=============================
Model              : Nelson-Aalen
>>> Turnbull.fit(x, turnbull_estimator='Kaplan-Meier')
Non-Parametric SurPyval Model
=============================
Model              : Turnbull
Estimator          : Kaplan-Meier
```

`surpyval.nonparametric.success_run.`**`success_run`**(*n*, *confidence=0.95*, *alpha=None*)

Function that can be used to estimte the confidence given n samples all survive a test.

`surpyval.nonparametric.plotting_positions.`**`plotting_positions`**(*x*, *c=None*, *n=None*, *t=None*, *heuristic='Blom'*, *turnbull_estimator='Fleming-Harrington'*)

This function takes in data in the xcnt format and outputs an approximation of the CDF. This function can be used to produce estimates of F using the Nelson-Aalen, Kaplan-Meier, Fleming-Harrington, and the Turnbull estimates. Additionally, it can be used to create 'plotting heuristics.'

Plotting heuristics are the values that are used to plot on probability paper and can be used to estiamte the parameters of a distribution. The use of probability plots is one of the traditional ways to estimate the parameters of a distribution.

If right censored data can be used by the regular plotting positions. If there is right censored data this method adjusts the ranks of the values using the mean order number.

Parameters

- **x** (*array like, optional*) – Array of observations of the random variables. If x is `None`, xl and xr must be provided.
- **c** (*array like, optional*) – Array of censoring flag. -1 is left censored, 0 is observed, 1 is right censored, and 2 is intervally censored. If not provided will assume all values are observed.
- **n** (*array like, optional*) – Array of counts for each x. If data is proivded as counts, then this can be provided. If `None` will assume each observation is 1.
- **t** (*2D-array like, optional*) – 2D array like of the left and right values at which the respective observation was truncated. If not provided it assumes that no truncation occurs.
- **heuristic** (*("Blom", "Median", "ECDF", "ECDF_Adj", "Modal", "Midpoint", "Mean", "Weibull", "Benard", "Beard", "Hazen", "Gringorten", "None", "Larsen", "Tukey", "DPW"), str, optional*) – Method to use to compute the heuristic of F. See details of each heursitic in the probability plotting section.
- **turnbull_estimator** (*('Nelson-Aalen', 'Kaplan-Meier'), str, optional*) – If using the Turnbull heuristic, you can elect to use the NA or KM method to compute R with the Turnbull estimates of the risk and deat sets.

Returns

- **x** (*numpy array*) – x values for the plotting points
- **r** (*numpy array*) – risk set at each x
- **d** (*numpy array*) – death set at each x
- **F** (*numpy array*) – estimate of F to use in plotting positions.

Examples

```
>>> from surpyval.nonparametric import plotting_positions
>>> import numpy as np
>>> x = np.array([1, 2, 3, 4, 5, 6, 7, 8])
>>> x, r, d, F = plotting_positions(x, heuristic="Filliben")
>>> F
array([0.08299596, 0.20113568, 0.32068141, 0.44022714, 0.55977286,
       0.67931859, 0.79886432, 0.91700404])
```

1.13.2 Parametric

Distribution Classes

Exponential

class `surpyval.parametric.exponential.`**`Exponential_`**(*name*)

Bases: *`surpyval.parametric.parametric_fitter.ParametricFitter`*

Class used to generate the Exponential class.

```
from surpyval import Exponential
```

`Hf`(*x*, *failure_rate*)

Cumulative hazard rate for the Exponential Distribution.

$$f(x) = \lambda x$$

Parameters

- **`x`** (*`numpy array or scalar`*) – The values at which the function will be calculated
- **`failure_rate`** (*`numpy array or scalar`*) – The scale parameter for the Exponential distribution

Returns **hf** – The cumulative hazard rate of the Exponential distribution for each value of x.

Return type scalar or numpy array

Examples

```
>>> import numpy as np
>>> from surpyval import Exponential
>>> x = np.array([1, 2, 3, 4, 5])
>>> Exponential.Hf(x, 3)
array([ 3,  6,  9, 12, 15])
```

cs (*x*, *X*, *failure_rate*)

Conditional survival function for the Exponential Distribution:

$$R(x) = e^{-\lambda x}$$

The Exponential distribution is memoryless, and hence is the same as the regular survival distribution.

Parameters

- **x** (*numpy array or scalar*) – The value(s) at which the function will be calculated
- **X** (*numpy array or scalar*) – The value(s) at which each value(s) in x was known to have survived
- **failure_rate** (*numpy array or scalar*) – The scale parameter for the Exponential distribution

Returns **cs** – the conditional survival probability.

Return type scalar or numpy array

Examples

```
>>> import numpy as np
>>> from surpyval import Exponential
>>> x = np.array([1, 2, 3, 4, 5])
>>> Exponential.cs(x, 5, 3)
array([4.97870684e-02, 2.47875218e-03, 1.23409804e-04, 6.14421235e-06,
       3.05902321e-07])
```

df (*x*, *failure_rate*)

Density function for the Exponential Distribution:

$$f(x) = \lambda e^{-\lambda x}$$

Parameters

- **x** (*numpy array or scalar*) – The values at which the function will be calculated
- **failure_rate** (*numpy array or scalar*) – The scale parameter for the Exponential distribution

Returns **df** – The density of the Exponential distribution for each value of x.

Return type scalar or numpy array

Examples

```
>>> import numpy as np
>>> from surpyval import Exponential
>>> x = np.array([1, 2, 3, 4, 5])
>>> Exponential.df(x, 3)
array([1.49361205e-01, 7.43625653e-03, 3.70229412e-04, 1.84326371e-05,
       9.17706962e-07])
```

entropy (*failure_rate*)

Calculates the entropy of the Exponential distribution.

$$S = 1 - \ln(\lambda)$$

Parameters **failure_rate** (*numpy array or scalar*) – The scale parameter for the Exponential distribution

Returns **entropy** – The entropy(ies) of the Exponential distribution

Return type scalar or numpy array

Examples

```
>>> from surpyval import Exponential
>>> Exponential.entropy(3)
-0.09861228866810978
```

ff (*x*, *failure_rate*)
CDF (or unreliability or failure) function for the Exponential Distribution:

$$F(x) = 1 - e^{-\lambda x}$$

Parameters

- **x** (*numpy array or scalar*) – The values at which the function will be calculated
- **failure_rate** (*numpy array or scalar*) – The scale parameter for the Exponential distribution

Returns **ff** – The value(s) of the CDF for each value of x.

Return type scalar or numpy array

Examples

```
>>> import numpy as np
>>> from surpyval import Exponential
>>> x = np.array([1, 2, 3, 4, 5])
>>> Exponential.ff(x, 3)
array([0.95021293, 0.99752125, 0.99987659, 0.99999386, 0.99999969])
```

fit (*x=None*, *c=None*, *n=None*, *t=None*, *how='MLE'*, *offset=False*, *zi=False*, *lfp=False*, *tl=None*, *tr=None*, *xl=None*, *xr=None*, *fixed=None*, *heuristic='Turnbull'*, *init=[]*, *rr='y'*, *on_d_is_0=False*, *turnbull_estimator='Fleming-Harrington'*)
The central feature to SurPyval's capability. This function aimed to have an API to mimic the simplicity of the scipy API. That is, to use a simple `fit()` call, with as many or as few parameters as is needed.

Parameters

- **x** (*array like, optional*) – Array of observations of the random variables. If x is `None`, xl and xr must be provided.
- **c** (*array like, optional*) – Array of censoring flag. -1 is left censored, 0 is observed, 1 is right censored, and 2 is intervally censored. If not provided will assume all values are observed.
- **n** (*array like, optional*) – Array of counts for each x. If data is proivded as counts, then this can be provided. If `None` will assume each observation is 1.
- **t** (*2D-array like, optional*) – 2D array like of the left and right values at which the respective observation was truncated. If not provided it assumes that no truncation occurs.

- **how** (*{'MLE', 'MPP', 'MOM', 'MSE', 'MPS'}, optional*) – Method to estimate parameters, these are:
  - MLE : Maximum Likelihood Estimation
  - MPP : Method of Probability Plotting
  - MOM : Method of Moments
  - MSE : Mean Square Error
  - MPS : Maximum Product Spacing
- **offset** (*boolean, optional*) – If `True` finds the shifted distribution. If not provided assumes not a shifted distribution. Only works with distributions that are supported on the half-real line.
- **tl** (*array like or scalar, optional*) – Values of left truncation for observations. If it is a scalar value assumes each observation is left truncated at the value. If an array, it is the respective 'late entry' of the observation
- **tr** (*array like or scalar, optional*) – Values of right truncation for observations. If it is a scalar value assumes each observation is right truncated at the value. If an array, it is the respective right truncation value for each observation
- **xl** (*array like, optional*) – Array like of the left array for 2-dimensional input of x. This is useful for data that is all intervally censored. Must be used with the `xr` input.
- **xr** (*array like, optional*) – Array like of the right array for 2-dimensional input of x. This is useful for data that is all intervally censored. Must be used with the `xl` input.
- **fixed** (*dict, optional*) – Dictionary of parameters and their values to fix. Fixes parameter by name.
- **heuristic** (*{'"Blom", "Median", "ECDF", "Modal", "Midpoint", "Mean", "Weibull", "Benard", "Beard", "Hazen", "Gringorten", "None", "Tukey", "DPW", "Fleming-Harrington", "Kaplan-Meier", "Nelson-Aalen", "Filliben", "Larsen", "Turnbull"}*) – Plotting method to use, if using the probability plotting, MPP, method.
- **init** (*array like, optional*) – initial guess of parameters. Useful if method is failing.
- **rr** (*('y', 'x')*) – The dimension on which to minimise the spacing between the line and the observation. If 'y' the mean square error between the line and vertical distance to each point is minimised. If 'x' the mean square error between the line and horizontal distance to each point is minimised.
- **on_d_is_0** (*boolean, optional*) – For the case when using MPP and the highest value is right censored, you can choosed to include this value into the regression analysis or not. That is, if `False`, all values where there are 0 deaths are excluded from the regression. If `True` all values regardless of whether there is a death or not are included in the regression.
- **turnbull_estimator** (*('Nelson-Aalen', 'Kaplan-Meier', or 'Fleming-Harrington'), str, optional*) – If using the Turnbull heuristic, you can elect to use either the KM, NA, or FH estimator with the Turnbull estimates of r, and d. Defaults to FH.

Returns **model** – A parametric model with the fitted parameters and methods for all functions of the distribution using the fitted parameters.

Return type *Parametric*

Examples

```
>>> from surpyval import Weibull
>>> import numpy as np
>>> x = Weibull.random(100, 10, 4)
>>> model = Weibull.fit(x)
>>> print(model)
Parametric SurPyval Model
=========================
Distribution        : Weibull
Fitted by           : MLE
Parameters          :
     alpha: 10.551521182640098
      beta: 3.792549834495306
>>> Weibull.fit(x, how='MPS', fixed={'alpha' : 10})
Parametric SurPyval Model
=========================
Distribution        : Weibull
Fitted by           : MPS
Parameters          :
     alpha: 10.0
      beta: 3.4314657446866836
>>> Weibull.fit(xl=x-1, xr=x+1, how='MPP')
Parametric SurPyval Model
=========================
Distribution        : Weibull
Fitted by           : MPP
Parameters          :
     alpha: 9.943092756713078
      beta: 8.613016934518258
>>> c = np.zeros_like(x)
>>> c[x > 13] = 1
>>> x[x > 13] = 13
>>> c = c[x > 6]
>>> x = x[x > 6]
>>> Weibull.fit(x=x, c=c, tl=6)
Parametric SurPyval Model
=========================
Distribution        : Weibull
Fitted by           : MLE
Parameters          :
     alpha: 10.363725328793413
      beta: 4.9886821457305865
```

fit_from_df (*df*, *x=None*, *c=None*, *n=None*, *xl=None*, *xr=None*, *tl=None*, *tr=None*, ***fit_options*)

The central feature to SurPyval's capability. This function aimed to have an API to mimic the simplicity of the scipy API. That is, to use a simple `fit()` call, with as many or as few parameters as is needed.

Parameters

- **df** (*DataFrame*) – DataFrame of data to be used to create surpyval model
- **x** (*string, optional*) – column name for the column in df containing the variable data. If not provided must provide both xl and xr
- **c** (*string, optional*) – column name for the column in df containing the censor

flag of x. If not provided assumes all values of x are observed.

- **n** (*string, optional*) – column name in for the column in df containing the counts of x. If not provided assumes each x is one observation.
- **tl** (*string or scalar, optional*) – If string, column name in for the column in df containing the left truncation data. If scalar assumes each x is left truncated by that value. If not provided assumes x is not left truncated.
- **tr** (*string or scalar, optional*) – If string, column name in for the column in df containing the right truncation data. If scalar assumes each x is right truncated by that value. If not provided assumes x is not right truncated.
- **xl** (*string, optional*) – column name for the column in df containing the left interval for interval censored data. If left interval is -Inf, assumes left censored. If xl[i] == xr[i] assumes observed. Cannot be provided with x, must be provided with xr.
- **xr** (*string, optional*) – column name for the column in df containing the right interval for interval censored data. If right interval is Inf, assumes right censored. If xl[i] == xr[i] assumes observed. Cannot be provided with x, must be provided with xl.
- **fit_options** (*dict, optional*) – dictionary of fit options that will be passed to the `fit` method, see that method for options.

Returns **model** – A parametric model with the fitted parameters and methods for all functions of the distribution using the fitted parameters.

Return type *Parametric*

Examples

```
>>> import surpyval as surv
>>> df = surv.datasets.BoforsSteel.df
>>> model = surv.Weibull.fit_from_df(df, x='x', n='n', offset=True)
>>> print(model)
Parametric SurPyval Model
=========================
Distribution        : Weibull
Fitted by           : MLE
Offset (gamma)      : 39.76562962867477
Parameters          :
     alpha: 7.141925216146524
      beta: 2.6204524040137844
```

from_params (*params, gamma=None, p=None, f0=None*)
Creating a SurPyval Parametric class with provided parameters.

Parameters

- **params** (*array like*) – array of the parameters of the distribution.
- **gamma** (*scalar, optional*) – offset value for the distribution. If not provided will fit a regular, unshifted/not offset, distribution.

Returns **model** – A parametric model with the fitted parameters and methods for all functions of the distribution using the fitted parameters.

Return type *Parametric*

Examples

```
>>> from surpyval import Weibull
>>> model = Weibull.from_params([10, 4])
>>> print(model)
Parametric SurPyval Model
=========================
Distribution        : Weibull
Fitted by           : given parameters
Parameters          :
     alpha: 10
      beta: 4
>>> model = Weibull.from_params([10, 4], gamma=2)
>>> print(model)
Parametric SurPyval Model
=========================
Distribution        : Weibull
Fitted by           : given parameters
Offset (gamma)      : 2
Parameters          :
     alpha: 10
      beta: 4
```

hf (*x*, *failure_rate*)
Instantaneous hazard rate for the Exponential Distribution.

$$f(x) = \lambda$$

The failure rate for the exponential distribution is constant. So this function only returns the input failure rate in the same shape as x.

Parameters

- **x** (*numpy array or scalar*) – The values at which the function will be calculated
- **failure_rate** (*numpy array or scalar*) – The scale parameter for the Exponential distribution

Returns **hf** – The instantaneous hazard rate of the Exponential distribution for each value of x.

Return type scalar or numpy array

Examples

```
>>> import numpy as np
>>> from surpyval import Exponential
>>> x = np.array([1, 2, 3, 4, 5])
>>> Exponential.hf(x, 3)
array([3, 3, 3, 3, 3])
```

mean (*failure_rate*)
Calculates the mean of the Exponential distribution with given parameters.

$$E = \frac{1}{\lambda}$$

Parameters **failure_rate** (*numpy array or scalar*) – The scale parameter for the Exponential distribution

Returns **mean** – The mean(s) of the Exponential distribution

Return type scalar or numpy array

Examples

```
>>> from surpyval import Exponential
>>> Exponential.mean(3)
0.3333333333333333
```

moment (*n*, *failure_rate*)

Calculates the n-th moment of the Exponential distribution.

$$E = \frac{n!}{\lambda^n}$$

Parameters

- **n** (*integer or numpy array of integers*) – The ordinal of the moment to calculate
- **failure_rate** (*numpy array or scalar*) – The scale parameter for the Exponential distribution

Returns **moment** – The moment(s) of the Exponential distribution

Return type scalar or numpy array

Examples

```
>>> from surpyval import Exponential
>>> Exponential.moment(2, 3)
0.2222222222222222
```

qf (*p*, *failure_rate*)

Quantile function for the Exponential Distribution:

$$q(p) = \frac{-\ln(p)}{\lambda}$$

Parameters

- **p** (*numpy array or scalar*) – The percentiles at which the quantile will be calculated
- **failure_rate** (*numpy array or scalar*) – The scale parameter for the Exponential distribution

Returns **q** – The quantiles for the Exponential distribution at each value p.

Return type scalar or numpy array

Examples

```
>>> import numpy as np
>>> from surpyval import Exponential
>>> p = np.array([.1, .2, .3, .4, .5])
>>> Exponential.qf(p, 3)
array([0.76752836, 0.5364793 , 0.40132427, 0.30543024, 0.23104906])
```

random (*size*, *failure_rate*)
Draws random samples from the distribution in shape *size*

Parameters

- **size** (*integer or tuple of positive integers*) – Shape or size of the random draw
- **failure_rate** (*numpy array or scalar*) – The scale parameter for the Exponential distribution

Returns **random** – Random values drawn from the distribution in shape *size*

Return type scalar or numpy array

Examples

```
>>> import numpy as np
>>> from surpyval import Exponential
>>> Exponential.random(10, 3)
array([0.32480264, 0.03186663, 0.41807108, 0.74221745, 0.06133774,
       0.2128422 , 0.36299424, 0.12250138, 0.61431089, 0.02266754])
>>> Exponential.random((5, 5), 3)
array([[0.25425552, 0.16867629, 0.21692401, 0.07020826, 0.03676643],
       [0.65528908, 0.20774767, 0.00625475, 0.04122388, 0.07089254],
       [1.22844679, 0.36199751, 0.564159  , 1.86811492, 0.08132478],
       [0.33541878, 0.38614518, 0.09285907, 0.33422975, 0.32515494],
       [0.03529228, 0.63134988, 0.45528738, 0.05037512, 0.7338039 ]])
```

sf (*x*, *failure_rate*)
Surival (or Reliability) function for the Exponential Distribution:

$$R(x) = e^{-\lambda x}$$

Parameters

- **x** (*numpy array or scalar*) – The values at which the function will be calculated
- **failure_rate** (*numpy array or scalar*) – The scale parameter for the Exponential distribution

Returns **sf** – The scalar value of the survival function of the distribution if a scalar was passed. If an array like object was passed then a numpy array is returned with the value of the survival function at each corresponding value in the input array.

Return type scalar or numpy array

Examples

```
>>> import numpy as np
>>> from surpyval import Exponential
>>> x = np.array([1, 2, 3, 4, 5])
>>> Exponential.sf(x, 3)
array([4.97870684e-02, 2.47875218e-03, 1.23409804e-04, 6.14421235e-06,
       3.05902321e-07])
```

Gamma

class surpyval.parametric.gamma.**Gamma_**(*name*)

Bases: *surpyval.parametric.parametric_fitter.ParametricFitter*

Class used to generate the Gamma class.

```
from surpyval import Gamma
```

Hf(*x*, *alpha*, *beta*)

Cumulative hazard rate for the Gamma Distribution:

$$H(x) = -\ln(1 - \frac{\gamma\left(\alpha, \beta x\right)}{\Gamma\left(\alpha\right)})$$

Parameters

- **x** (*numpy array or scalar*) – The values at which the function will be calculated
- **alpha** (*numpy array or scalar*) – The shape parameter for the Gamma distribution
- **beta** (*numpy array or scalar*) – The scale parameter for the Gamma distribution

Returns **Hf** – The cumulative hazard rate of the distribution at each x

Return type scalar or numpy array

Examples

```
>>> import numpy as np
>>> from surpyval import Gamma
>>> x = np.array([1, 2, 3, 4, 5])
>>> Gamma.Hf(x, 3, 2)
array([0.39056209, 1.43505064, 2.78112418, 4.28642793, 5.88912614])
```

cs(*x*, *X*, *alpha*, *beta*)

Conditional survival function for the Gamma Distribution:

$$R(x) = e^{-\lambda x}$$

Parameters

- **x** (*numpy array or scalar*) – The value(s) at which the function will be calculated
- **X** (*numpy array or scalar*) – The value(s) at which each value(s) in x was known to have survived
- **alpha** (*numpy array or scalar*) – The shape parameter for the Gamma distribution
- **beta** (*numpy array or scalar*) – The scale parameter for the Gamma distribution

Returns **cs** – the conditional survival probability.

Return type scalar or numpy array

Examples

```
>>> import numpy as np
>>> from surpyval import Gamma
>>> x = np.array([1, 2, 3, 4, 5])
>>> Gamma.cs(x, 5, 3, 4)
array([2.59402488e-02, 6.39048747e-04, 1.51519143e-05, 3.48776510e-07,
       7.79933496e-09])
```

df (*x*, *alpha*, *beta*)

Density function for the Gamma Distribution:

$$f(x) = \frac{\beta^\alpha}{\Gamma\left(\alpha\right)}x^{\alpha-1}e^{-\beta x}$$

Parameters

- **x** (*numpy array or scalar*) – The values at which the function will be calculated
- **alpha** (*numpy array or scalar*) – The shape parameter for the Gamma distribution
- **beta** (*numpy array or scalar*) – The scale parameter for the Gamma distribution

Returns **df** – The density of the distribution at each x

Return type scalar or numpy array

Examples

```
>>> import numpy as np
>>> from surpyval import Gamma
>>> x = np.array([1, 2, 3, 4, 5])
>>> Gamma.df(x, 3, 2)
array([0.54134113, 0.29305022, 0.08923508, 0.02146961, 0.00453999])
```

ff (*x*, *alpha*, *beta*)

CDF (or unreliability or failure) function for the Gamma Distribution:

$$F(x) = \frac{\gamma\left(\alpha, \beta x\right)}{\Gamma\left(\alpha\right)}$$

Parameters

- **x** (*numpy array or scalar*) – The values at which the function will be calculated
- **alpha** (*numpy array or scalar*) – The shape parameter for the Gamma distribution
- **beta** (*numpy array or scalar*) – The scale parameter for the Gamma distribution

Returns **ff** – The value(s) for the CDF at each x

Return type scalar or numpy array

Examples

```
>>> import numpy as np
>>> from surpyval import Gamma
>>> x = np.array([1, 2, 3, 4, 5])
>>> Gamma.ff(x, 3, 2)
array([0.32332358, 0.76189669, 0.9380312 , 0.98624603, 0.9972306 ])
```

fit(*x=None*, *c=None*, *n=None*, *t=None*, *how='MLE'*, *offset=False*, *zi=False*, *lfp=False*, *tl=None*, *tr=None*, *xl=None*, *xr=None*, *fixed=None*, *heuristic='Turnbull'*, *init=[]*, *rr='y'*, *on_d_is_0=False*, *turnbull_estimator='Fleming-Harrington'*)

The central feature to SurPyval's capability. This function aimed to have an API to mimic the simplicity of the scipy API. That is, to use a simple `fit()` call, with as many or as few parameters as is needed.

Parameters

- **x** (*array like, optional*) – Array of observations of the random variables. If x is `None`, xl and xr must be provided.
- **c** (*array like, optional*) – Array of censoring flag. -1 is left censored, 0 is observed, 1 is right censored, and 2 is intervally censored. If not provided will assume all values are observed.
- **n** (*array like, optional*) – Array of counts for each x. If data is proivded as counts, then this can be provided. If `None` will assume each observation is 1.
- **t** (*2D-array like, optional*) – 2D array like of the left and right values at which the respective observation was truncated. If not provided it assumes that no truncation occurs.
- **how** (*{'MLE', 'MPP', 'MOM', 'MSE', 'MPS'}, optional*) – Method to estimate parameters, these are:
  - MLE : Maximum Likelihood Estimation
  - MPP : Method of Probability Plotting
  - MOM : Method of Moments
  - MSE : Mean Square Error
  - MPS : Maximum Product Spacing
- **offset** (*boolean, optional*) – If `True` finds the shifted distribution. If not provided assumes not a shifted distribution. Only works with distributions that are supported on the half-real line.
- **tl** (*array like or scalar, optional*) – Values of left truncation for observations. If it is a scalar value assumes each observation is left truncated at the value. If an array, it is the respective 'late entry' of the observation
- **tr** (*array like or scalar, optional*) – Values of right truncation for observations. If it is a scalar value assumes each observation is right truncated at the value. If an array, it is the respective right truncation value for each observation
- **xl** (*array like, optional*) – Array like of the left array for 2-dimensional input of x. This is useful for data that is all intervally censored. Must be used with the `xr` input.
- **xr** (*array like, optional*) – Array like of the right array for 2-dimensional input of x. This is useful for data that is all intervally censored. Must be used with the `xl` input.
- **fixed** (*dict, optional*) – Dictionary of parameters and their values to fix. Fixes parameter by name.
- **heuristic** (*{'"Blom", "Median", "ECDF", "Modal", "Midpoint", "Mean", "Weibull", "Benard", "Beard", "Hazen", "Gringorten", "None", "Tukey", "DPW", "Fleming-Harrington", "Kaplan-Meier", "Nelson-Aalen", "Filliben", "Larsen", "Turnbull"}*) – Plotting method to use, if using the probability plotting, MPP, method.

- **init** (*array like, optional*) – initial guess of parameters. Useful if method is failing.
- **rr** (*('y', 'x')*) – The dimension on which to minimise the spacing between the line and the observation. If 'y' the mean square error between the line and vertical distance to each point is minimised. If 'x' the mean square error between the line and horizontal distance to each point is minimised.
- **on_d_is_0** (*boolean, optional*) – For the case when using MPP and the highest value is right censored, you can choosed to include this value into the regression analysis or not. That is, if `False`, all values where there are 0 deaths are excluded from the regression. If `True` all values regardless of whether there is a death or not are included in the regression.
- **turnbull_estimator** (*('Nelson-Aalen', 'Kaplan-Meier', or 'Fleming-Harrington'), str, optional*) – If using the Turnbull heuristic, you can elect to use either the KM, NA, or FH estimator with the Turnbull estimates of r, and d. Defaults to FH.

Returns **model** – A parametric model with the fitted parameters and methods for all functions of the distribution using the fitted parameters.

Return type *Parametric*

Examples

```
>>> from surpyval import Weibull
>>> import numpy as np
>>> x = Weibull.random(100, 10, 4)
>>> model = Weibull.fit(x)
>>> print(model)
Parametric SurPyval Model
=========================
Distribution        : Weibull
Fitted by           : MLE
Parameters          :
     alpha: 10.551521182640098
      beta: 3.792549834495306
>>> Weibull.fit(x, how='MPS', fixed={'alpha' : 10})
Parametric SurPyval Model
=========================
Distribution        : Weibull
Fitted by           : MPS
Parameters          :
     alpha: 10.0
      beta: 3.4314657446866836
>>> Weibull.fit(xl=x-1, xr=x+1, how='MPP')
Parametric SurPyval Model
=========================
Distribution        : Weibull
Fitted by           : MPP
Parameters          :
     alpha: 9.943092756713078
      beta: 8.613016934518258
>>> c = np.zeros_like(x)
>>> c[x > 13] = 1
>>> x[x > 13] = 13
```

(continues on next page)

(continued from previous page)

```
>>> c = c[x > 6]
>>> x = x[x > 6]
>>> Weibull.fit(x=x, c=c, tl=6)
Parametric SurPyval Model
=========================
Distribution        : Weibull
Fitted by           : MLE
Parameters          :
     alpha: 10.363725328793413
      beta: 4.9886821457305865
```

fit_from_df (*df*, *x=None*, *c=None*, *n=None*, *xl=None*, *xr=None*, *tl=None*, *tr=None*, ***fit_options*)

The central feature to SurPyval's capability. This function aimed to have an API to mimic the simplicity of the scipy API. That is, to use a simple `fit()` call, with as many or as few parameters as is needed.

Parameters

- **df** (*DataFrame*) – DataFrame of data to be used to create surpyval model
- **x** (*string, optional*) – column name for the column in df containing the variable data. If not provided must provide both xl and xr
- **c** (*string, optional*) – column name for the column in df containing the censor flag of x. If not provided assumes all values of x are observed.
- **n** (*string, optional*) – column name in for the column in df containing the counts of x. If not provided assumes each x is one observation.
- **tl** (*string or scalar, optional*) – If string, column name in for the column in df containing the left truncation data. If scalar assumes each x is left truncated by that value. If not provided assumes x is not left truncated.
- **tr** (*string or scalar, optional*) – If string, column name in for the column in df containing the right truncation data. If scalar assumes each x is right truncated by that value. If not provided assumes x is not right truncated.
- **xl** (*string, optional*) – column name for the column in df containing the left interval for interval censored data. If left interval is -Inf, assumes left censored. If xl[i] == xr[i] assumes observed. Cannot be provided with x, must be provided with xr.
- **xr** (*string, optional*) – column name for the column in df containing the right interval for interval censored data. If right interval is Inf, assumes right censored. If xl[i] == xr[i] assumes observed. Cannot be provided with x, must be provided with xl.
- **fit_options** (*dict, optional*) – dictionary of fit options that will be passed to the `fit` method, see that method for options.

Returns **model** – A parametric model with the fitted parameters and methods for all functions of the distribution using the fitted parameters.

Return type *Parametric*

Examples

```
>>> import surpyval as surv
>>> df = surv.datasets.BoforsSteel.df
>>> model = surv.Weibull.fit_from_df(df, x='x', n='n', offset=True)
>>> print(model)
```

(continues on next page)

(continued from previous page)

```
Parametric SurPyval Model
=========================
Distribution        : Weibull
Fitted by           : MLE
Offset (gamma)      : 39.76562962867477
Parameters          :
     alpha: 7.141925216146524
      beta: 2.6204524040137844
```

from_params (*params*, *gamma=None*, *p=None*, *f0=None*)

Creating a SurPyval Parametric class with provided parameters.

Parameters

- **params** (*array like*) – array of the parameters of the distribution.
- **gamma** (*scalar, optional*) – offset value for the distribution. If not provided will fit a regular, unshifted/not offset, distribution.

Returns **model** – A parametric model with the fitted parameters and methods for all functions of the distribution using the fitted parameters.

Return type *Parametric*

Examples

```
>>> from surpyval import Weibull
>>> model = Weibull.from_params([10, 4])
>>> print(model)
Parametric SurPyval Model
=========================
Distribution        : Weibull
Fitted by           : given parameters
Parameters          :
     alpha: 10
      beta: 4
>>> model = Weibull.from_params([10, 4], gamma=2)
>>> print(model)
Parametric SurPyval Model
=========================
Distribution        : Weibull
Fitted by           : given parameters
Offset (gamma)      : 2
Parameters          :
     alpha: 10
      beta: 4
```

hf (*x*, *alpha*, *beta*)

Instantaneous hazard rate for the Gamma Distribution:

$$h(x) = \frac{\frac{\beta^\alpha}{\Gamma(\alpha)} x^{\alpha-1} e^{-\beta x}}{1 - \frac{\gamma(\alpha, \beta x)}{\Gamma(\alpha)}}$$

Parameters

- **x** (*numpy array or scalar*) – The values at which the function will be calculated

- **alpha** (*numpy array or scalar*) – The shape parameter for the Gamma distribution
- **beta** (*numpy array or scalar*) – The scale parameter for the Gamma distribution

Returns **Hf** – The instantaneous hazard rate of the distribution at each x

Return type scalar or numpy array

Examples

```
>>> import numpy as np
>>> from surpyval import Gamma
>>> x = np.array([1, 2, 3, 4, 5])
>>> Gamma.hf(x, 3, 2)
array([0.8        , 1.23076923, 1.44       , 1.56097561, 1.63934426])
```

mean (*alpha*, *beta*)

Calculates the mean of the Gamma distribution with given parameters.

$$E = \frac{\alpha}{\beta}$$

Parameters

- **alpha** (*numpy array or scalar*) – The shape parameter for the Gamma distribution
- **beta** (*numpy array or scalar*) – The scale parameter for the Gamma distribution

Returns **mean** – The mean(s) of the Gamma distribution

Return type scalar or numpy array

Examples

```
>>> from surpyval import Gamma
>>> Gamma.mean(3, 4)
0.75
```

moment (*n*, *alpha*, *beta*)

Calculates the n-th moment of the Gamma distribution with given parameters.

$$E = \frac{\Gamma(n+\alpha)}{\beta^n \Gamma(\alpha)}$$

Parameters

- **n** (*integer or numpy array of integers*) – The ordinal of the moment to calculate
- **alpha** (*numpy array or scalar*) – The shape parameter for the Gamma distribution
- **beta** (*numpy array or scalar*) – The scale parameter for the Gamma distribution

Returns **mean** – The moment(s) of the Gamma distribution

Return type scalar or numpy array

Examples

```
>>> from surpyval import Gamma
>>> Gamma.moment(3, 3, 4)
0.9375
```

qf (*p*, *alpha*, *beta*)
Quantile function for the Gamma Distribution:

$$q(p) = \frac{-\ln(p)}{\lambda}$$

Parameters

- **p** (*numpy array or scalar*) – The percentiles at which the quantile will be calculated
- **alpha** (*numpy array or scalar*) – The shape parameter for the Gamma distribution
- **beta** (*numpy array or scalar*) – The scale parameter for the Gamma distribution

Returns **q** – The quantiles for the Gamma distribution at each value p.

Return type scalar or numpy array

Examples

```
>>> import numpy as np
>>> from surpyval import Gamma
>>> p = np.array([.1, .2, .3, .4, .5])
>>> Gamma.qf(p, 3, 4)
array([0.27551633, 0.38376105, 0.47844395, 0.57126923, 0.66851508])
```

random (*size*, *alpha*, *beta*)
Draws random samples from the Gamma distribution in shape *size*

Parameters

- **alpha** (*numpy array or scalar*) – The shape parameter for the Gamma distribution
- **beta** (*numpy array or scalar*) – The scale parameter for the Gamma distribution

Returns **random** – Random values drawn from the distribution in shape *size*

Return type scalar or numpy array

Examples

```
>>> import numpy as np
>>> from surpyval import Gamma
>>> Gamma.random(10, 3, 4)
array([0.22856155, 1.69542468, 0.70894789, 0.75552168, 0.76634128,
       0.58624638, 1.03288812, 0.85768925, 0.75071764, 0.91979151])
>>> Gamma.random((5, 5), 3, 4)
array([[0.55481976, 1.02867642, 1.25525161, 0.5141736 , 0.7227451 ],
       [1.59192864, 1.22897457, 0.80820007, 0.39872068, 0.53656654],
```

(continues on next page)

(continued from previous page)

```
       [0.80703614, 0.75406597, 0.87307426, 1.88748737, 0.78115455],
       [1.3233755 , 0.29908068, 1.88304902, 2.65690385, 0.51018073],
       [0.36070265, 0.48834586, 0.45623895, 0.30104303, 0.49942908]])
```

sf (*x*, *alpha*, *beta*)

Surival (or Reliability) function for the Gamma Distribution:

$$R(x) = 1 - \frac{\gamma\left(\alpha, \beta x\right)}{\Gamma\left(\alpha\right)}$$

Parameters

- **x** (*numpy array or scalar*) – The values at which the function will be calculated
- **alpha** (*numpy array or scalar*) – The shape parameter for the Gamma distribution
- **beta** (*numpy array or scalar*) – The scale parameter for the Gamma distribution

Returns **sf** – The value(s) for the survival function at each x

Return type scalar or numpy array

Examples

```
>>> import numpy as np
>>> from surpyval import Gamma
>>> x = np.array([1, 2, 3, 4, 5])
>>> Gamma.sf(x, 3, 2)
array([0.67667642, 0.23810331, 0.0619688 , 0.01375397, 0.0027694 ])
```

Gumbel

class surpyval.parametric.gumbel.**Gumbel_**(*name*)

Bases: *surpyval.parametric.parametric_fitter.ParametricFitter*

Hf (*x*, *mu*, *sigma*)

Cumulative hazard rate for the Gumbel Distribution:

$$H(x) = e^{\frac{x-\mu}{\sigma}}$$

Parameters

- **x** (*numpy array or scalar*) – The values of the random variables at which the survival function will be calculated
- **mu** (*numpy array like or scalar*) – The location parameter of the distribution
- **sigma** (*numpy array like or scalar*) – The scale parameter of the distribution

Returns **Hf** – The value(s) for the cumulative hazard rate for the Gumbel distribution.

Return type scalar or numpy array

Examples

```
>>> import numpy as np
>>> from surpyval import Gumbel
>>> x = np.array([1, 2, 3, 4, 5])
>>> Gumbel.Hf(x, 3, 2)
array([0.36787944, 0.60653066, 1.        , 1.64872127, 2.71828183])
```

df (*x*, *mu*, *sigma*)
Density function (pdf) for the Gumbel Distribution:

$$f(x) = \frac{1}{\sigma}e^{\left(\frac{x-\mu}{\sigma} - e^{\frac{x-\mu}{\sigma}}\right)}$$

Parameters

- **x** (*numpy array or scalar*) – The values of the random variables at which the survival function will be calculated
- **mu** (*numpy array like or scalar*) – The location parameter of the distribution
- **sigma** (*numpy array like or scalar*) – The scale parameter of the distribution

Returns **df** – The scalar value of the density of the distribution if a scalar was passed. If an array like object was passed then a numpy array is returned with the value of the density at each corresponding value in the input array.

Return type scalar or numpy array

Examples

```
>>> import numpy as np
>>> from surpyval import Gumbel
>>> x = np.array([1, 2, 3, 4, 5])
>>> Gumbel.df(x, 3, 2)
array([0.12732319, 0.16535215, 0.18393972, 0.15852096, 0.08968704])
```

ff (*x*, *mu*, *sigma*)
CDF (or Failure) function for the Gumbel Distribution:

$$F(x) = e^{e^{-(x-\mu)/\sigma}}$$

Parameters

- **x** (*numpy array or scalar*) – The values of the random variables at which the survival function will be calculated
- **mu** (*numpy array like or scalar*) – The location parameter of the distribution
- **sigma** (*numpy array like or scalar*) – The scale parameter of the distribution

Returns **ff** – The scalar value of the failure function of the distribution if a scalar was passed. If an array like object was passed then a numpy array is returned with the value of the failure function at each corresponding value in the input array.

Return type scalar or numpy array

Examples

```
>>> import numpy as np
>>> from surpyval import Gumbel
>>> x = np.array([1, 2, 3, 4, 5])
>>> Gumbel.ff(x, 3, 2)
array([0.30779937, 0.45476079, 0.63212056, 0.80770435, 0.93401196])
```

fit (*x=None*, *c=None*, *n=None*, *t=None*, *how='MLE'*, *offset=False*, *zi=False*, *lfp=False*, *tl=None*, *tr=None*, *xl=None*, *xr=None*, *fixed=None*, *heuristic='Turnbull'*, *init=[]*, *rr='y'*, *on_d_is_0=False*, *turnbull_estimator='Fleming-Harrington'*)

The central feature to SurPyval's capability. This function aimed to have an API to mimic the simplicity of the scipy API. That is, to use a simple `fit()` call, with as many or as few parameters as is needed.

Parameters

- **x** (*array like, optional*) – Array of observations of the random variables. If x is `None`, xl and xr must be provided.
- **c** (*array like, optional*) – Array of censoring flag. -1 is left censored, 0 is observed, 1 is right censored, and 2 is intervally censored. If not provided will assume all values are observed.
- **n** (*array like, optional*) – Array of counts for each x. If data is proivded as counts, then this can be provided. If `None` will assume each observation is 1.
- **t** (*2D-array like, optional*) – 2D array like of the left and right values at which the respective observation was truncated. If not provided it assumes that no truncation occurs.
- **how** (*{'MLE', 'MPP', 'MOM', 'MSE', 'MPS'}, optional*) – Method to estimate parameters, these are:
  - MLE : Maximum Likelihood Estimation
  - MPP : Method of Probability Plotting
  - MOM : Method of Moments
  - MSE : Mean Square Error
  - MPS : Maximum Product Spacing
- **offset** (*boolean, optional*) – If `True` finds the shifted distribution. If not provided assumes not a shifted distribution. Only works with distributions that are supported on the half-real line.
- **tl** (*array like or scalar, optional*) – Values of left truncation for observations. If it is a scalar value assumes each observation is left truncated at the value. If an array, it is the respective 'late entry' of the observation
- **tr** (*array like or scalar, optional*) – Values of right truncation for observations. If it is a scalar value assumes each observation is right truncated at the value. If an array, it is the respective right truncation value for each observation
- **xl** (*array like, optional*) – Array like of the left array for 2-dimensional input of x. This is useful for data that is all intervally censored. Must be used with the `xr` input.
- **xr** (*array like, optional*) – Array like of the right array for 2-dimensional input of x. This is useful for data that is all intervally censored. Must be used with the `xl` input.
- **fixed** (*dict, optional*) – Dictionary of parameters and their values to fix. Fixes parameter by name.

- **heuristic** (*{'"Blom", "Median", "ECDF", "Modal", "Midpoint", "Mean", "Weibull", "Benard", "Beard", "Hazen", "Gringorten", "None", "Tukey", "DPW", "Fleming-Harrington", "Kaplan-Meier", "Nelson-Aalen", "Filliben", "Larsen", "Turnbull"}*) – Plotting method to use, if using the probability plotting, MPP, method.
- **init** (*array like, optional*) – initial guess of parameters. Useful if method is failing.
- **rr** (*('y', 'x')*) – The dimension on which to minimise the spacing between the line and the observation. If 'y' the mean square error between the line and vertical distance to each point is minimised. If 'x' the mean square error between the line and horizontal distance to each point is minimised.
- **on_d_is_0** (*boolean, optional*) – For the case when using MPP and the highest value is right censored, you can choosed to include this value into the regression analysis or not. That is, if `False`, all values where there are 0 deaths are excluded from the regression. If `True` all values regardless of whether there is a death or not are included in the regression.
- **turnbull_estimator** (*('Nelson-Aalen', 'Kaplan-Meier', or 'Fleming-Harrington'), str, optional*) – If using the Turnbull heuristic, you can elect to use either the KM, NA, or FH estimator with the Turnbull estimates of r, and d. Defaults to FH.

Returns model – A parametric model with the fitted parameters and methods for all functions of the distribution using the fitted parameters.

Return type *Parametric*

Examples

```
>>> from surpyval import Weibull
>>> import numpy as np
>>> x = Weibull.random(100, 10, 4)
>>> model = Weibull.fit(x)
>>> print(model)
Parametric SurPyval Model
=========================
Distribution        : Weibull
Fitted by           : MLE
Parameters          :
     alpha: 10.551521182640098
      beta: 3.792549834495306
>>> Weibull.fit(x, how='MPS', fixed={'alpha' : 10})
Parametric SurPyval Model
=========================
Distribution        : Weibull
Fitted by           : MPS
Parameters          :
     alpha: 10.0
      beta: 3.4314657446866836
>>> Weibull.fit(xl=x-1, xr=x+1, how='MPP')
Parametric SurPyval Model
=========================
Distribution        : Weibull
```

(continues on next page)

(continued from previous page)

```
Fitted by           : MPP
Parameters          :
     alpha: 9.943092756713078
      beta: 8.613016934518258
>>> c = np.zeros_like(x)
>>> c[x > 13] = 1
>>> x[x > 13] = 13
>>> c = c[x > 6]
>>> x = x[x > 6]
>>> Weibull.fit(x=x, c=c, tl=6)
Parametric SurPyval Model
=========================
Distribution        : Weibull
Fitted by           : MLE
Parameters          :
     alpha: 10.363725328793413
      beta: 4.9886821457305865
```

fit_from_df (*df*, *x=None*, *c=None*, *n=None*, *xl=None*, *xr=None*, *tl=None*, *tr=None*, ***fit_options*)
The central feature to SurPyval's capability. This function aimed to have an API to mimic the simplicity of the scipy API. That is, to use a simple `fit()` call, with as many or as few parameters as is needed.

Parameters

- **df** (*DataFrame*) – DataFrame of data to be used to create surpyval model
- **x** (*string, optional*) – column name for the column in df containing the variable data. If not provided must provide both xl and xr
- **c** (*string, optional*) – column name for the column in df containing the censor flag of x. If not provided assumes all values of x are observed.
- **n** (*string, optional*) – column name in for the column in df containing the counts of x. If not provided assumes each x is one observation.
- **tl** (*string or scalar, optional*) – If string, column name in for the column in df containing the left truncation data. If scalar assumes each x is left truncated by that value. If not provided assumes x is not left truncated.
- **tr** (*string or scalar, optional*) – If string, column name in for the column in df containing the right truncation data. If scalar assumes each x is right truncated by that value. If not provided assumes x is not right truncated.
- **xl** (*string, optional*) – column name for the column in df containing the left interval for interval censored data. If left interval is -Inf, assumes left censored. If xl[i] == xr[i] assumes observed. Cannot be provided with x, must be provided with xr.
- **xr** (*string, optional*) – column name for the column in df containing the right interval for interval censored data. If right interval is Inf, assumes right censored. If xl[i] == xr[i] assumes observed. Cannot be provided with x, must be provided with xl.
- **fit_options** (*dict, optional*) – dictionary of fit options that will be passed to the `fit` method, see that method for options.

Returns **model** – A parametric model with the fitted parameters and methods for all functions of the distribution using the fitted parameters.

Return type *Parametric*

Examples

```
>>> import surpyval as surv
>>> df = surv.datasets.BoforsSteel.df
>>> model = surv.Weibull.fit_from_df(df, x='x', n='n', offset=True)
>>> print(model)
Parametric SurPyval Model
=========================
Distribution        : Weibull
Fitted by           : MLE
Offset (gamma)      : 39.76562962867477
Parameters          :
     alpha: 7.141925216146524
      beta: 2.6204524040137844
```

from_params (*params*, *gamma=None*, *p=None*, *f0=None*)
Creating a SurPyval Parametric class with provided parameters.

Parameters

- **params** (*array like*) – array of the parameters of the distribution.
- **gamma** (*scalar, optional*) – offset value for the distribution. If not provided will fit a regular, unshifted/not offset, distribution.

Returns **model** – A parametric model with the fitted parameters and methods for all functions of the distribution using the fitted parameters.

Return type *Parametric*

Examples

```
>>> from surpyval import Weibull
>>> model = Weibull.from_params([10, 4])
>>> print(model)
Parametric SurPyval Model
=========================
Distribution        : Weibull
Fitted by           : given parameters
Parameters          :
     alpha: 10
      beta: 4
>>> model = Weibull.from_params([10, 4], gamma=2)
>>> print(model)
Parametric SurPyval Model
=========================
Distribution        : Weibull
Fitted by           : given parameters
Offset (gamma)      : 2
Parameters          :
     alpha: 10
      beta: 4
```

hf (*x*, *mu*, *sigma*)
Instantaneous hazard rate for the Gumbel Distribution:

$$h(x) = \frac{1}{\sigma} e^{\frac{x-\mu}{\sigma}}$$

Parameters

- **x** (*numpy array or scalar*) – The values of the random variables at which the survival function will be calculated
- **mu** (*numpy array like or scalar*) – The location parameter of the distribution
- **sigma** (*numpy array like or scalar*) – The scale parameter of the distribution

Returns **hf** – The value(s) for the instantaneous hazard rate for the Gumbel distribution.

Return type scalar or numpy array

Examples

```
>>> import numpy as np
>>> from surpyval import Gumbel
>>> x = np.array([1, 2, 3, 4, 5])
>>> Gumbel.hf(x, 3, 2)
array([0.18393972, 0.30326533, 0.5       , 0.82436064, 1.35914091])
```

mean (*mu*, *sigma*)

Calculates the mean of the Gumbel distribution with given parameters.

$$E = \mu + \sigma\gamma$$

Where gamma is the Euler-Mascheroni constant

Parameters

- **mu** (*numpy array like or scalar*) – The location parameter(s) of the distribution
- **sigma** (*numpy array like or scalar*) – The scale parameter(s) of the distribution

Returns **mean** – The mean(s) of the Gumbel distribution

Return type scalar or numpy array

Examples

```
>>> from surpyval import Gumbel
>>> Gumbel.mean(3, 2)
4.1544313298030655
```

qf (*p*, *mu*, *sigma*)

Quantile function for the Gumbel Distribution:

$$q(p) = \mu + \sigma \ln\left(-\ln\left(1 - p\right)\right)$$

Parameters

- **p** (*numpy array or scalar*) – The percentiles at which the quantile will be calculated
- **mu** (*numpy array like or scalar*) – The location parameter(s) of the distribution

- **sigma** (*numpy array like or scalar*) – The scale parameter(s) of the distribution

Returns **q** – The quantiles for the Gumbel distribution at each value p.

Return type scalar or numpy array

Examples

```
>>> import numpy as np
>>> from surpyval import Gumbel
>>> p = np.array([.1, .2, .3, .4, .5])
>>> Gumbel.qf(p, 3, 2)
array([-1.50073465e+00,  1.20026481e-04,  9.38139134e-01,  1.65654602e+00,
2.26697416e+00])
```

random (*size*, *mu*, *sigma*)
Draws random samples from the distribution in shape *size*

Parameters

- **size** (*integer or tuple of positive integers*) – Shape or size of the random draw
- **mu** (*numpy array like or scalar*) – The location parameter(s) of the distribution
- **sigma** (*numpy array like or scalar*) – The scale parameter(s) of the distribution

Returns **random** – Random values drawn from the distribution in shape *size*

Return type scalar or numpy array

Examples

```
>>> import numpy as np
>>> from surpyval import Gumbel
>>> Gumbel.random(10, 3, 2)
array([1.50706388, 3.3098799 , 4.32358009, 2.9914246 , 4.47216839,
       3.56676358, 4.19781514, 4.49123942, 7.29849677, 6.32996653])
>>> Gumbel.random((5, 5), 3, 2)
array([[ 5.97265715,  5.89177067,  2.95883424,  2.46315557,  5.15250379],
       [ 2.33808212,  7.42817091,  0.90560051,  8.05897841,  6.30714544],
       [ 6.13076426,  6.31925048,  4.34031705,  3.01309504, -0.70053049],
       [ 5.84888474,  5.95097491,  6.23960618,  6.24830057,  4.89655192],
       [ 6.29507963,  4.21798292,  4.22835474,  5.23521822,  2.76053242]])
```

sf (*x*, *mu*, *sigma*)
Surival (or Reliability) function for the Gumbel Distribution:

$$R(x) = 1 - e^{e^{-(x-\mu)/\sigma}}$$

Parameters

- **x** (*numpy array or scalar*) – The values of the random variables at which the survival function will be calculated
- **mu** (*numpy array like or scalar*) – The location parameter of the distribution

- **sigma** (*numpy array like or scalar*) – The scale parameter of the distribution

Returns **sf** – The scalar value of the survival function of the distribution if a scalar was passed. If an array like object was passed then a numpy array is returned with the value of the survival function at each corresponding value in the input array.

Return type scalar or numpy array

Examples

```
>>> import numpy as np
>>> from surpyval import Gumbel
>>> x = np.array([1, 2, 3, 4, 5])
>>> Gumbel.sf(x, 3, 2)
array([0.69220063, 0.54523921, 0.36787944, 0.19229565, 0.06598804])
```

Logistic

class surpyval.parametric.logistic.**Logistic_**(*name*)

Bases: *surpyval.parametric.parametric_fitter.ParametricFitter*

Hf(*x, mu, sigma*)

Cumulative hazard rate for the Logistic distribution:

$$h(x) = -\ln\left(R(x)\right)$$

Parameters

- **x** (*numpy array or scalar*) – The values at which the function will be calculated
- **mu** (*numpy array or scalar*) – The location parameter for the Logistic distribution
- **sigma** (*numpy array or scalar*) – The scale parameter for the Logistic distribution

Returns **hf** – The value(s) of the cumulative hazard rate at x.

Return type scalar or numpy array

Examples

```
>>> import numpy as np
>>> from surpyval import Logistic
>>> x = np.array([1, 2, 3, 4, 5])
>>> Logistic.Hf(x, 3, 4)
array([0.47407698, 0.57593942, 0.69314718, 0.82593942, 0.97407698])
```

df(*x, mu, sigma*)

Failure (CDF or unreliability) function for the Logistic Distribution:

$$f(x) = \frac{e^{-(x-\mu)/\sigma}}{\sigma\left(1+e^{-(x-\mu)/\sigma}\right)^2}$$

Parameters

- **x** (*numpy array or scalar*) – The values at which the function will be calculated
- **mu** (*numpy array or scalar*) – The location parameter for the Logistic distribution
- **sigma** (*numpy array or scalar*) – The scale parameter for the Logistic distribution

Returns **df** – The value(s) of the density function at x.

Return type scalar or numpy array

Examples

```
>>> import numpy as np
>>> from surpyval import Logistic
>>> x = np.array([1, 2, 3, 4, 5])
>>> Logistic.df(x, 3, 4)
array([0.05875093, 0.06153352, 0.0625    , 0.06153352, 0.05875093])
```

ff (*x*, *mu*, *sigma*)

Failure (CDF or unreliability) function for the Logistic Distribution:

$$F(x) = \frac{1}{1 + e^{-(x-\mu)/\sigma}}$$

Parameters

- **x** (*numpy array or scalar*) – The values at which the function will be calculated
- **mu** (*numpy array or scalar*) – The location parameter for the Logistic distribution
- **sigma** (*numpy array or scalar*) – The scale parameter for the Logistic distribution

Returns **ff** – The value(s) of the failure function at x.

Return type scalar or numpy array

Examples

```
>>> import numpy as np
>>> from surpyval import Logistic
>>> x = np.array([1, 2, 3, 4, 5])
>>> Logistic.ff(x, 3, 4)
array([0.37754067, 0.4378235 , 0.5       , 0.5621765 , 0.62245933])
```

fit (*x=None*, *c=None*, *n=None*, *t=None*, *how='MLE'*, *offset=False*, *zi=False*, *lfp=False*, *tl=None*, *tr=None*, *xl=None*, *xr=None*, *fixed=None*, *heuristic='Turnbull'*, *init=[]*, *rr='y'*, *on_d_is_0=False*, *turnbull_estimator='Fleming-Harrington'*)

The central feature to SurPyval's capability. This function aimed to have an API to mimic the simplicity of the scipy API. That is, to use a simple `fit()` call, with as many or as few parameters as is needed.

Parameters

- **x** (*array like, optional*) – Array of observations of the random variables. If x is `None`, xl and xr must be provided.

- **c** (*array like, optional*) – Array of censoring flag. -1 is left censored, 0 is observed, 1 is right censored, and 2 is intervally censored. If not provided will assume all values are observed.
- **n** (*array like, optional*) – Array of counts for each x. If data is proivded as counts, then this can be provided. If `None` will assume each observation is 1.
- **t** (*2D-array like, optional*) – 2D array like of the left and right values at which the respective observation was truncated. If not provided it assumes that no truncation occurs.
- **how** (*{'MLE', 'MPP', 'MOM', 'MSE', 'MPS'}, optional*) – Method to estimate parameters, these are:
  - MLE : Maximum Likelihood Estimation
  - MPP : Method of Probability Plotting
  - MOM : Method of Moments
  - MSE : Mean Square Error
  - MPS : Maximum Product Spacing
- **offset** (*boolean, optional*) – If `True` finds the shifted distribution. If not provided assumes not a shifted distribution. Only works with distributions that are supported on the half-real line.
- **tl** (*array like or scalar, optional*) – Values of left truncation for observations. If it is a scalar value assumes each observation is left truncated at the value. If an array, it is the respective 'late entry' of the observation
- **tr** (*array like or scalar, optional*) – Values of right truncation for observations. If it is a scalar value assumes each observation is right truncated at the value. If an array, it is the respective right truncation value for each observation
- **xl** (*array like, optional*) – Array like of the left array for 2-dimensional input of x. This is useful for data that is all intervally censored. Must be used with the `xr` input.
- **xr** (*array like, optional*) – Array like of the right array for 2-dimensional input of x. This is useful for data that is all intervally censored. Must be used with the `xl` input.
- **fixed** (*dict, optional*) – Dictionary of parameters and their values to fix. Fixes parameter by name.
- **heuristic** (*{'"Blom", "Median", "ECDF", "Modal", "Midpoint", "Mean", "Weibull", "Benard", "Beard", "Hazen", "Gringorten", "None", "Tukey", "DPW", "Fleming-Harrington", "Kaplan-Meier", "Nelson-Aalen", "Filliben", "Larsen", "Turnbull"}*) – Plotting method to use, if using the probability plotting, MPP, method.
- **init** (*array like, optional*) – initial guess of parameters. Useful if method is failing.
- **rr** (*('y', 'x')*) – The dimension on which to minimise the spacing between the line and the observation. If 'y' the mean square error between the line and vertical distance to each point is minimised. If 'x' the mean square error between the line and horizontal distance to each point is minimised.
- **on_d_is_0** (*boolean, optional*) – For the case when using MPP and the highest value is right censored, you can choosed to include this value into the regression analysis or not. That is, if `False`, all values where there are 0 deaths are excluded from the

regression. If `True` all values regardless of whether there is a death or not are included in the regression.

- **turnbull_estimator** (*('Nelson-Aalen', 'Kaplan-Meier', or 'Fleming-Harrington'), str, optional*) – If using the Turnbull heuristic, you can elect to use either the KM, NA, or FH estimator with the Turnbull estimates of r, and d. Defaults to FH.

Returns **model** – A parametric model with the fitted parameters and methods for all functions of the distribution using the fitted parameters.

Return type *Parametric*

Examples

```
>>> from surpyval import Weibull
>>> import numpy as np
>>> x = Weibull.random(100, 10, 4)
>>> model = Weibull.fit(x)
>>> print(model)
Parametric SurPyval Model
=========================
Distribution        : Weibull
Fitted by           : MLE
Parameters          :
     alpha: 10.551521182640098
      beta: 3.792549834495306
>>> Weibull.fit(x, how='MPS', fixed={'alpha' : 10})
Parametric SurPyval Model
=========================
Distribution        : Weibull
Fitted by           : MPS
Parameters          :
     alpha: 10.0
      beta: 3.4314657446866836
>>> Weibull.fit(xl=x-1, xr=x+1, how='MPP')
Parametric SurPyval Model
=========================
Distribution        : Weibull
Fitted by           : MPP
Parameters          :
     alpha: 9.943092756713078
      beta: 8.613016934518258
>>> c = np.zeros_like(x)
>>> c[x > 13] = 1
>>> x[x > 13] = 13
>>> c = c[x > 6]
>>> x = x[x > 6]
>>> Weibull.fit(x=x, c=c, tl=6)
Parametric SurPyval Model
=========================
Distribution        : Weibull
Fitted by           : MLE
Parameters          :
     alpha: 10.363725328793413
      beta: 4.9886821457305865
```

fit_from_df(*df, x=None, c=None, n=None, xl=None, xr=None, tl=None, tr=None, **fit_options*)

The central feature to SurPyval's capability. This function aimed to have an API to mimic the simplicity of the scipy API. That is, to use a simple `fit()` call, with as many or as few parameters as is needed.

Parameters

- **df** (*DataFrame*) – DataFrame of data to be used to create surpyval model
- **x** (*string, optional*) – column name for the column in df containing the variable data. If not provided must provide both xl and xr
- **c** (*string, optional*) – column name for the column in df containing the censor flag of x. If not provided assumes all values of x are observed.
- **n** (*string, optional*) – column name in for the column in df containing the counts of x. If not provided assumes each x is one observation.
- **tl** (*string or scalar, optional*) – If string, column name in for the column in df containing the left truncation data. If scalar assumes each x is left truncated by that value. If not provided assumes x is not left truncated.
- **tr** (*string or scalar, optional*) – If string, column name in for the column in df containing the right truncation data. If scalar assumes each x is right truncated by that value. If not provided assumes x is not right truncated.
- **xl** (*string, optional*) – column name for the column in df containing the left interval for interval censored data. If left interval is -Inf, assumes left censored. If xl[i] == xr[i] assumes observed. Cannot be provided with x, must be provided with xr.
- **xr** (*string, optional*) – column name for the column in df containing the right interval for interval censored data. If right interval is Inf, assumes right censored. If xl[i] == xr[i] assumes observed. Cannot be provided with x, must be provided with xl.
- **fit_options** (*dict, optional*) – dictionary of fit options that will be passed to the `fit` method, see that method for options.

Returns **model** – A parametric model with the fitted parameters and methods for all functions of the distribution using the fitted parameters.

Return type *Parametric*

Examples

```
>>> import surpyval as surv
>>> df = surv.datasets.BoforsSteel.df
>>> model = surv.Weibull.fit_from_df(df, x='x', n='n', offset=True)
>>> print(model)
Parametric SurPyval Model
=========================
Distribution        : Weibull
Fitted by           : MLE
Offset (gamma)      : 39.76562962867477
Parameters          :
     alpha: 7.141925216146524
      beta: 2.6204524040137844
```

from_params (*params*, *gamma=None*, *p=None*, *f0=None*)

Creating a SurPyval Parametric class with provided parameters.

Parameters

- **params** (*array like*) – array of the parameters of the distribution.

- **gamma** (*scalar, optional*) – offset value for the distribution. If not provided will fit a regular, unshifted/not offset, distribution.

Returns model – A parametric model with the fitted parameters and methods for all functions of the distribution using the fitted parameters.

Return type *Parametric*

Examples

```
>>> from surpyval import Weibull
>>> model = Weibull.from_params([10, 4])
>>> print(model)
Parametric SurPyval Model
=========================
Distribution        : Weibull
Fitted by           : given parameters
Parameters          :
     alpha: 10
      beta: 4
>>> model = Weibull.from_params([10, 4], gamma=2)
>>> print(model)
Parametric SurPyval Model
=========================
Distribution        : Weibull
Fitted by           : given parameters
Offset (gamma)      : 2
Parameters          :
     alpha: 10
      beta: 4
```

hf (*x, mu, sigma*)

Instantaneous hazard rate for the Logistic Distribution:

$$h(x) = \frac{f(x)}{R(x)}$$

Parameters

- **x** (*numpy array or scalar*) – The values at which the function will be calculated
- **mu** (*numpy array or scalar*) – The location parameter for the Logistic distribution
- **sigma** (*numpy array or scalar*) – The scale parameter for the Logistic distribution

Returns hf – The value(s) of the instantaneous hazard rate at x.

Return type scalar or numpy array

Examples

```
>>> import numpy as np
>>> from surpyval import Logistic
>>> x = np.array([1, 2, 3, 4, 5])
>>> Logistic.hf(x, 3, 4)
array([0.09438517, 0.10945587, 0.125     , 0.14054413, 0.15561483])
```

mean (*mu*, *sigma*)

Mean of the Logistic distribution

$$E = \mu$$

Parameters

- **mu** (*numpy array or scalar*) – The location parameter for the Logistic distribution
- **sigma** (*numpy array or scalar*) – The scale parameter for the Logistic distribution

Returns **mu** – The mean(s) of the Logistic distribution

Return type scalar or numpy array

Examples

```
>>> from surpyval import Logistic
>>> Logistic.mean(3, 4)
3
```

qf (*p*, *mu*, *sigma*)

Quantile function for the Logistic distribution:

$$q(p) = \mu + \sigma \ln \left(\frac{p}{1-p} \right)$$

Parameters

- **p** (*numpy array or scalar*) – The percentiles at which the quantile will be calculated
- **mu** (*numpy array or scalar*) – The location parameter for the Logistic distribution
- **sigma** (*numpy array or scalar*) – The scale parameter for the Logistic distribution

Returns **q** – The quantiles for the Logistic distribution at each value p

Return type scalar or numpy array

Examples

```
>>> import numpy as np
>>> from surpyval import Logistic
>>> p = np.array([.1, .2, .3, .4, .5])
>>> Logistic.qf(p, 3, 4)
array([-5.78889831, -2.54517744, -0.38919144,  1.37813957,  3.        ])
```

random (*size*, *mu*, *sigma*)

Draws random samples from the distribution in shape *size*

Parameters

- **size** (*integer or tuple of positive integers*) – Shape or size of the random draw

- **mu** (*numpy array or scalar*) – The location parameter for the Logistic distribution
- **sigma** (*numpy array or scalar*) – The scale parameter for the Logistic distribution

Returns **random** – Random values drawn from the distribution in shape *size*

Return type scalar or numpy array

Examples

```
>>> import numpy as np
>>> from surpyval import Logistic
>>> Logistic.random(10, 3, 4)
array([-8.03085073, -1.69001847,  5.25971637,  4.49119392,  3.92027233,
       -0.8320818 , -7.08778338,  5.01180405,  0.82373259,  8.51506487])
>>> Logistic.random((5, 5), 3, 4)
array([[ 7.11691946, 14.31662627,  8.5383889 ,  1.26608344,  0.97633704],
       [-7.11229405,  8.56748118,  1.5959416 , -3.89229554, -2.44623464],
       [ 5.58805039, -0.11768336, -0.55000158,  8.5302643 ,  6.92591024],
       [-2.88281091, -9.79724128, -3.80713019,  1.74120972, 15.37924263],
       [-4.42521443, -0.69577732,  3.54658395,  2.82310964,  3.95850831]])
```

sf (*x*, *mu*, *sigma*)

Survival (or reliability) function for the Logistic Distribution:

$$R(x) = 1 - \frac{1}{1 + e^{-(x-\mu)/\sigma}}$$

Parameters

- **x** (*numpy array or scalar*) – The values at which the function will be calculated
- **mu** (*numpy array or scalar*) – The location parameter for the Logistic distribution
- **sigma** (*numpy array or scalar*) – The scale parameter for the Logistic distribution

Returns **sf** – The value(s) of the reliability function at x.

Return type scalar or numpy array

Examples

```
>>> import numpy as np
>>> from surpyval import Logistic
>>> x = np.array([1, 2, 3, 4, 5])
>>> Logistic.sf(x, 3, 4)
array([0.62245933, 0.5621765 , 0.5       , 0.4378235 , 0.37754067])
```

LogLogistic

class surpyval.parametric.loglogistic.**LogLogistic_**(*name*)

Bases: *surpyval.parametric.parametric_fitter.ParametricFitter*

Hf (*x*, *alpha*, *beta*)

Cumulative hazard rate for the LogLogistic Distribution:

$$H(x) = -\ln\left(R(x)\right)$$

Parameters

- **x** (*numpy array or scalar*) – The values at which the function will be calculated
- **alpha** (*numpy array or scalar*) – scale parameter for the LogLogistic distribution
- **beta** (*numpy array or scalar*) – shape parameter for the LogLogistic distribution

Returns **Hf** – The value(s) of the cumulative hazard rate at x.

Return type scalar or numpy array

Examples

```
>>> import numpy as np
>>> from surpyval import LogLogistic
>>> x = np.array([1, 2, 3, 4, 5])
>>> LogLogistic.Hf(x, 3, 4)
array([0.01227009, 0.18026182, 0.69314718, 1.42563378, 2.16516608])
```

cs (*x*, *X*, *alpha*, *beta*)

Conditional survival function for the LogLogistic Distribution:

$$R(x, X) = \frac{R(x+X)}{R(X)}$$

Parameters

- **x** (*numpy array or scalar*) – The values at which the function will be calculated
- **X** (*numpy array or scalar*) – The value(s) at which each value(s) in x was known to have survived
- **alpha** (*numpy array or scalar*) – scale parameter for the LogLogistic distribution
- **beta** (*numpy array or scalar*) – shape parameter for the LogLogistic distribution

Returns **cs** – The value(s) of the conditional survival function at x.

Return type scalar or numpy array

Examples

```
>>> import numpy as np
>>> from surpyval import LogLogistic
>>> x = np.array([1, 2, 3, 4, 5])
>>> LogLogistic.cs(x, 5, 3, 4)
array([0.51270879, 0.28444803, 0.16902083, 0.10629329, 0.07003273])
```

df (*x*, *alpha*, *beta*)

Density function for the LogLogistic Distribution:

$$f(x) = \frac{(\beta/\alpha)(x/\alpha)^{\beta-1}}{\left(1+(x/\alpha)^{-\beta}\right)^2}$$

Parameters

- **x** (*numpy array or scalar*) – The values at which the function will be calculated
- **alpha** (*numpy array or scalar*) – scale parameter for the LogLogistic distribution
- **beta** (*numpy array or scalar*) – shape parameter for the LogLogistic distribution

Returns **df** – The value(s) of the failure function at x.

Return type scalar or numpy array

Examples

```
>>> import numpy as np
>>> from surpyval import LogLogistic
>>> x = np.array([1, 2, 3, 4, 5])
>>> LogLogistic.df(x, 3, 4)
array([0.0481856 , 0.27548092, 0.33333333, 0.18258504, 0.08125416])
```

ff (*x*, *alpha*, *beta*)

Failure (CDF or unreliability) function for the LogLogistic Distribution:

$$F(x) = \frac{1}{1+(x/\alpha)^{-\beta}}$$

Parameters

- **x** (*numpy array or scalar*) – The values at which the function will be calculated
- **alpha** (*numpy array or scalar*) – scale parameter for the LogLogistic distribution
- **beta** (*numpy array or scalar*) – shape parameter for the LogLogistic distribution

Returns **ff** – The value(s) of the failure function at x.

Return type scalar or numpy array

Examples

```
>>> import numpy as np
>>> from surpyval import LogLogistic
>>> x = np.array([1, 2, 3, 4, 5])
>>> LogLogistic.ff(x, 3, 4)
array([0.01219512, 0.16494845, 0.5       , 0.75964392, 0.88526912])
```

fit(*x=None*, *c=None*, *n=None*, *t=None*, *how='MLE'*, *offset=False*, *zi=False*, *lfp=False*, *tl=None*, *tr=None*, *xl=None*, *xr=None*, *fixed=None*, *heuristic='Turnbull'*, *init=[]*, *rr='y'*, *on_d_is_0=False*, *turnbull_estimator='Fleming-Harrington'*)

The central feature to SurPyval's capability. This function aimed to have an API to mimic the simplicity of the scipy API. That is, to use a simple `fit()` call, with as many or as few parameters as is needed.

Parameters

- **x** (*array like, optional*) – Array of observations of the random variables. If x is `None`, xl and xr must be provided.
- **c** (*array like, optional*) – Array of censoring flag. -1 is left censored, 0 is observed, 1 is right censored, and 2 is intervally censored. If not provided will assume all values are observed.
- **n** (*array like, optional*) – Array of counts for each x. If data is proivded as counts, then this can be provided. If `None` will assume each observation is 1.
- **t** (*2D-array like, optional*) – 2D array like of the left and right values at which the respective observation was truncated. If not provided it assumes that no truncation occurs.
- **how** (*{'MLE', 'MPP', 'MOM', 'MSE', 'MPS'}, optional*) – Method to estimate parameters, these are:
  - MLE : Maximum Likelihood Estimation
  - MPP : Method of Probability Plotting
  - MOM : Method of Moments
  - MSE : Mean Square Error
  - MPS : Maximum Product Spacing
- **offset** (*boolean, optional*) – If `True` finds the shifted distribution. If not provided assumes not a shifted distribution. Only works with distributions that are supported on the half-real line.
- **tl** (*array like or scalar, optional*) – Values of left truncation for observations. If it is a scalar value assumes each observation is left truncated at the value. If an array, it is the respective 'late entry' of the observation
- **tr** (*array like or scalar, optional*) – Values of right truncation for observations. If it is a scalar value assumes each observation is right truncated at the value. If an array, it is the respective right truncation value for each observation
- **xl** (*array like, optional*) – Array like of the left array for 2-dimensional input of x. This is useful for data that is all intervally censored. Must be used with the `xr` input.
- **xr** (*array like, optional*) – Array like of the right array for 2-dimensional input of x. This is useful for data that is all intervally censored. Must be used with the `xl` input.
- **fixed** (*dict, optional*) – Dictionary of parameters and their values to fix. Fixes parameter by name.
- **heuristic** (*{'"Blom", "Median", "ECDF", "Modal", "Midpoint", "Mean", "Weibull", "Benard", "Beard", "Hazen", "Gringorten", "None", "Tukey", "DPW", "Fleming-Harrington", "Kaplan-Meier", "Nelson-Aalen", "Filliben", "Larsen", "Turnbull"}*) – Plotting method to use, if using the probability plotting, MPP, method.

- **init** (*array like, optional*) – initial guess of parameters. Useful if method is failing.
- **rr** (*('y', 'x')*) – The dimension on which to minimise the spacing between the line and the observation. If 'y' the mean square error between the line and vertical distance to each point is minimised. If 'x' the mean square error between the line and horizontal distance to each point is minimised.
- **on_d_is_0** (*boolean, optional*) – For the case when using MPP and the highest value is right censored, you can choosed to include this value into the regression analysis or not. That is, if `False`, all values where there are 0 deaths are excluded from the regression. If `True` all values regardless of whether there is a death or not are included in the regression.
- **turnbull_estimator** (*('Nelson-Aalen', 'Kaplan-Meier', or 'Fleming-Harrington'), str, optional*) – If using the Turnbull heuristic, you can elect to use either the KM, NA, or FH estimator with the Turnbull estimates of r, and d. Defaults to FH.

Returns **model** – A parametric model with the fitted parameters and methods for all functions of the distribution using the fitted parameters.

Return type *Parametric*

Examples

```
>>> from surpyval import Weibull
>>> import numpy as np
>>> x = Weibull.random(100, 10, 4)
>>> model = Weibull.fit(x)
>>> print(model)
Parametric SurPyval Model
=========================
Distribution        : Weibull
Fitted by           : MLE
Parameters          :
     alpha: 10.551521182640098
      beta: 3.792549834495306
>>> Weibull.fit(x, how='MPS', fixed={'alpha' : 10})
Parametric SurPyval Model
=========================
Distribution        : Weibull
Fitted by           : MPS
Parameters          :
     alpha: 10.0
      beta: 3.4314657446866836
>>> Weibull.fit(xl=x-1, xr=x+1, how='MPP')
Parametric SurPyval Model
=========================
Distribution        : Weibull
Fitted by           : MPP
Parameters          :
     alpha: 9.943092756713078
      beta: 8.613016934518258
>>> c = np.zeros_like(x)
>>> c[x > 13] = 1
>>> x[x > 13] = 13
```

(continues on next page)

(continued from previous page)

```
>>> c = c[x > 6]
>>> x = x[x > 6]
>>> Weibull.fit(x=x, c=c, tl=6)
Parametric SurPyval Model
=========================
Distribution        : Weibull
Fitted by           : MLE
Parameters          :
     alpha: 10.363725328793413
      beta: 4.9886821457305865
```

fit_from_df (*df*, *x=None*, *c=None*, *n=None*, *xl=None*, *xr=None*, *tl=None*, *tr=None*, ***fit_options*)

The central feature to SurPyval's capability. This function aimed to have an API to mimic the simplicity of the scipy API. That is, to use a simple `fit()` call, with as many or as few parameters as is needed.

Parameters

- **df** (*DataFrame*) – DataFrame of data to be used to create surpyval model
- **x** (*string, optional*) – column name for the column in df containing the variable data. If not provided must provide both xl and xr
- **c** (*string, optional*) – column name for the column in df containing the censor flag of x. If not provided assumes all values of x are observed.
- **n** (*string, optional*) – column name in for the column in df containing the counts of x. If not provided assumes each x is one observation.
- **tl** (*string or scalar, optional*) – If string, column name in for the column in df containing the left truncation data. If scalar assumes each x is left truncated by that value. If not provided assumes x is not left truncated.
- **tr** (*string or scalar, optional*) – If string, column name in for the column in df containing the right truncation data. If scalar assumes each x is right truncated by that value. If not provided assumes x is not right truncated.
- **xl** (*string, optional*) – column name for the column in df containing the left interval for interval censored data. If left interval is -Inf, assumes left censored. If xl[i] == xr[i] assumes observed. Cannot be provided with x, must be provided with xr.
- **xr** (*string, optional*) – column name for the column in df containing the right interval for interval censored data. If right interval is Inf, assumes right censored. If xl[i] == xr[i] assumes observed. Cannot be provided with x, must be provided with xl.
- **fit_options** (*dict, optional*) – dictionary of fit options that will be passed to the `fit` method, see that method for options.

Returns **model** – A parametric model with the fitted parameters and methods for all functions of the distribution using the fitted parameters.

Return type *Parametric*

Examples

```
>>> import surpyval as surv
>>> df = surv.datasets.BoforsSteel.df
>>> model = surv.Weibull.fit_from_df(df, x='x', n='n', offset=True)
>>> print(model)
```

(continues on next page)

(continued from previous page)

```
Parametric SurPyval Model
=========================
Distribution        : Weibull
Fitted by           : MLE
Offset (gamma)      : 39.76562962867477
Parameters          :
     alpha: 7.141925216146524
      beta: 2.6204524040137844
```

from_params (*params*, *gamma=None*, *p=None*, *f0=None*)
Creating a SurPyval Parametric class with provided parameters.

Parameters

- **params** (`array like`) – array of the parameters of the distribution.
- **gamma** (`scalar, optional`) – offset value for the distribution. If not provided will fit a regular, unshifted/not offset, distribution.

Returns model – A parametric model with the fitted parameters and methods for all functions of the distribution using the fitted parameters.

Return type *Parametric*

Examples

```
>>> from surpyval import Weibull
>>> model = Weibull.from_params([10, 4])
>>> print(model)
Parametric SurPyval Model
=========================
Distribution        : Weibull
Fitted by           : given parameters
Parameters          :
     alpha: 10
      beta: 4
>>> model = Weibull.from_params([10, 4], gamma=2)
>>> print(model)
Parametric SurPyval Model
=========================
Distribution        : Weibull
Fitted by           : given parameters
Offset (gamma)      : 2
Parameters          :
     alpha: 10
      beta: 4
```

hf (*x*, *alpha*, *beta*)
Instantaneous hazard rate for the LogLogistic Distribution:

$$h(x) = \frac{f(x)}{R(x)}$$

Parameters

- **x** (`numpy array or scalar`) – The values at which the function will be calculated
- **alpha** (`numpy array or scalar`) – scale parameter for the LogLogistic distribution

- **beta** (*numpy array or scalar*) – shape parameter for the LogLogistic distribution

Returns **hf** – The value(s) of the instantaneous hazard rate at x.

Return type scalar or numpy array

Examples

```
>>> import numpy as np
>>> from surpyval import LogLogistic
>>> x = np.array([1, 2, 3, 4, 5])
>>> LogLogistic.hf(x, 3, 4)
array([0.04878049, 0.32989691, 0.66666667, 0.75964392, 0.7082153 ])
```

mean (*alpha*, *beta*)
Mean of the LogLogistic distribution

$$E = \frac{\alpha\pi/\beta}{sin\left(\pi/\beta\right)}$$

Parameters

- **alpha** (*numpy array or scalar*) – scale parameter for the LogLogistic distribution
- **beta** (*numpy array or scalar*) – shape parameter for the LogLogistic distribution

Returns **mean** – The mean(s) of the LogLogistic distribution

Return type scalar or numpy array

Examples

```
>>> from surpyval import LogLogistic
>>> LogLogistic.mean(3, 4)
3
```

qf (*p*, *alpha*, *beta*)
Quantile function for the LogLogistic distribution:

$$q(p) = \alpha \left(\frac{p}{1-p} \right)^{\frac{1}{\beta}}$$

Parameters

- **p** (*numpy array or scalar*) – The percentiles at which the quantile will be calculated
- **alpha** (*numpy array or scalar*) – scale parameter for the LogLogistic distribution
- **beta** (*numpy array or scalar*) – shape parameter for the LogLogistic distribution

Returns **q** – The quantiles for the LogLogistic distribution at each value p

Return type scalar or numpy array

Examples

```
>>> import numpy as np
>>> from surpyval import LogLogistic
>>> p = np.array([.1, .2, .3, .4, .5])
>>> LogLogistic.qf(p, 3, 4)
array([1.73205081, 2.12132034, 2.42732013, 2.71080601, 3.        ])
```

random (*size*, *alpha*, *beta*)

Draws random samples from the distribution in shape *size*

Parameters

- **size** (*integer or tuple of positive integers*) – Shape or size of the random draw
- **alpha** (*numpy array or scalar*) – scale parameter for the LogLogistic distribution
- **beta** (*numpy array or scalar*) – shape parameter for the LogLogistic distribution

Returns **random** – Random values drawn from the distribution in shape *size*

Return type scalar or numpy array

Examples

```
>>> import numpy as np
>>> from surpyval import LogLogistic
>>> LogLogistic.random(10, 3, 4)
array([4.46072122, 2.1336253 , 2.74159711, 2.90125715, 3.2390347 ,
       5.45223664, 4.28281376, 2.7017541 , 3.023811  , 2.16225601])
>>> LogLogistic.random((5, 5), 3, 4)
array([[1.97744499, 4.02823921, 1.95761719, 1.20481591, 3.7166738 ],
       [2.94863864, 3.02609811, 3.30563774, 2.39100075, 3.24937459],
       [3.16102391, 1.77003533, 4.73831093, 0.36936215, 1.41566853],
       [3.88505024, 2.88183095, 2.43977804, 2.62385959, 3.40881857],
       [1.2349273 , 1.83914641, 3.68502568, 6.49834769, 8.62995574]])
```

sf (*x*, *alpha*, *beta*)

Survival (or reliability) function for the LogLogistic Distribution:

$$R(x) = 1 - \frac{1}{1 + (x/\alpha)^{-\beta}}$$

Parameters

- **x** (*numpy array or scalar*) – The values at which the function will be calculated
- **alpha** (*numpy array or scalar*) – scale parameter for the LogLogistic distribution
- **beta** (*numpy array or scalar*) – shape parameter for the LogLogistic distribution

Returns **sf** – The value(s) of the reliability function at x.

Return type scalar or numpy array

Examples

```
>>> import numpy as np
>>> from surpyval import LogLogistic
>>> x = np.array([1, 2, 3, 4, 5])
>>> LogLogistic.sf(x, 3, 4)
array([0.62245933, 0.5621765 , 0.5       , 0.4378235 , 0.37754067])
```

LogNormal

class surpyval.parametric.lognormal.**LogNormal_**(*name*)

Bases: *surpyval.parametric.parametric_fitter.ParametricFitter*

Hf(*x, mu, sigma*)

Cumulative hazard rate for the LogNormal Distribution:

$$H(x) = -\ln\left(R(x)\right)$$

Parameters

- **x** (*numpy array or scalar*) – The values at which the function will be calculated
- **mu** (*numpy array or scalar*) – The location parameter for the LogNormal distribution
- **sigma** (*numpy array or scalar*) – The scale parameter for the LogNormal distribution

Returns **Hf** – The value(s) of the cumulative hazard rate at x.

Return type scalar or numpy array

Examples

```
>>> import numpy as np
>>> from surpyval import LogNormal
>>> x = np.array([1, 2, 3, 4, 5])
>>> LogNormal.Hf(x, 3, 4)
array([0.25699427, 0.33137848, 0.3816556 , 0.4205543 , 0.45264333])
```

cs(*x, X, mu, sigma*)

Conditional survival function for the LogNormal Distribution:

$$R(x, X) = \frac{R(x + X)}{R(X)}$$

Parameters

- **x** (*numpy array or scalar*) – The value(s) at which the function will be calculated
- **X** (*numpy array or scalar*) – The value(s) at which each value(s) in x was known to have survived
- **mu** (*numpy array or scalar*) – The location parameter for the LogNormal distribution
- **sigma** (*numpy array or scalar*) – The scale parameter for the LogNormal distribution

Returns **cs** – the conditional survival probability at x

Return type scalar or numpy array

Examples

```
>>> import numpy as np
>>> from surpyval import LogNormal
>>> x = np.array([1, 2, 3, 4, 5])
>>> LogNormal.cs(x, 5, 3, 4)
array([0.97287811, 0.9496515 , 0.92933892, 0.91129122, 0.89505592])
```

df (*x*, *mu*, *sigma*)

Density function for the LogNormal Distribution:

$$f(x) = \frac{1}{x\sigma\sqrt{2\pi}}e^{-\frac{1}{2}\left(\frac{\ln x - \mu}{\sigma}\right)^2}$$

Parameters

- **x** (*numpy array or scalar*) – The values at which the function will be calculated
- **mu** (*numpy array or scalar*) – The location parameter for the LogNormal distribution
- **sigma** (*numpy array or scalar*) – The scale parameter for the LogNormal distribution

Returns **df** – The value(s) of the density function at x.

Return type scalar or numpy array

Examples

```
>>> import numpy as np
>>> from surpyval import LogNormal
>>> x = np.array([1, 2, 3, 4, 5])
>>> LogNormal.df(x, 3, 4)
array([0.07528436, 0.04222769, 0.02969364, 0.02298522, 0.01877747])
```

ff (*x*, *mu*, *sigma*)

Failure (CDF or unreliability) function for the LogNormal Distribution:

$$F(x) = \Phi\left(\frac{\ln(x) - \mu}{\sigma}\right)$$

Parameters

- **x** (*numpy array or scalar*) – The values at which the function will be calculated
- **mu** (*numpy array or scalar*) – The location parameter for the LogNormal distribution
- **sigma** (*numpy array or scalar*) – The scale parameter for the LogNormal distribution

Returns **ff** – The value(s) of the failure function at x.

Return type scalar or numpy array

Examples

```
>>> import numpy as np
>>> from surpyval import LogNormal
>>> x = np.array([1, 2, 3, 4, 5])
>>> LogNormal.ff(x, 3, 4)
array([0.22662735, 0.28206661, 0.31726986, 0.34331728, 0.36405509])
```

fit(*x=None*, *c=None*, *n=None*, *t=None*, *how='MLE'*, *offset=False*, *zi=False*, *lfp=False*, *tl=None*, *tr=None*, *xl=None*, *xr=None*, *fixed=None*, *heuristic='Turnbull'*, *init=[]*, *rr='y'*, *on_d_is_0=False*, *turnbull_estimator='Fleming-Harrington'*)

The central feature to SurPyval's capability. This function aimed to have an API to mimic the simplicity of the scipy API. That is, to use a simple `fit()` call, with as many or as few parameters as is needed.

Parameters

- **x** (*array like, optional*) – Array of observations of the random variables. If x is `None`, xl and xr must be provided.
- **c** (*array like, optional*) – Array of censoring flag. -1 is left censored, 0 is observed, 1 is right censored, and 2 is intervally censored. If not provided will assume all values are observed.
- **n** (*array like, optional*) – Array of counts for each x. If data is proivded as counts, then this can be provided. If `None` will assume each observation is 1.
- **t** (*2D-array like, optional*) – 2D array like of the left and right values at which the respective observation was truncated. If not provided it assumes that no truncation occurs.
- **how** (*{'MLE', 'MPP', 'MOM', 'MSE', 'MPS'}, optional*) – Method to estimate parameters, these are:
  – MLE : Maximum Likelihood Estimation
  – MPP : Method of Probability Plotting
  – MOM : Method of Moments
  – MSE : Mean Square Error
  – MPS : Maximum Product Spacing
- **offset** (*boolean, optional*) – If `True` finds the shifted distribution. If not provided assumes not a shifted distribution. Only works with distributions that are supported on the half-real line.
- **tl** (*array like or scalar, optional*) – Values of left truncation for observations. If it is a scalar value assumes each observation is left truncated at the value. If an array, it is the respective 'late entry' of the observation
- **tr** (*array like or scalar, optional*) – Values of right truncation for observations. If it is a scalar value assumes each observation is right truncated at the value. If an array, it is the respective right truncation value for each observation
- **xl** (*array like, optional*) – Array like of the left array for 2-dimensional input of x. This is useful for data that is all intervally censored. Must be used with the `xr` input.
- **xr** (*array like, optional*) – Array like of the right array for 2-dimensional input of x. This is useful for data that is all intervally censored. Must be used with the `xl` input.
- **fixed** (*dict, optional*) – Dictionary of parameters and their values to fix. Fixes parameter by name.

- **heuristic** (*{'"Blom", "Median", "ECDF", "Modal", "Midpoint", "Mean", "Weibull", "Benard", "Beard", "Hazen", "Gringorten", "None", "Tukey", "DPW", "Fleming-Harrington", "Kaplan-Meier", "Nelson-Aalen", "Filliben", "Larsen", "Turnbull"}*) – Plotting method to use, if using the probability plotting, MPP, method.
- **init** (*array like, optional*) – initial guess of parameters. Useful if method is failing.
- **rr** (*('y', 'x')*) – The dimension on which to minimise the spacing between the line and the observation. If 'y' the mean square error between the line and vertical distance to each point is minimised. If 'x' the mean square error between the line and horizontal distance to each point is minimised.
- **on_d_is_0** (*boolean, optional*) – For the case when using MPP and the highest value is right censored, you can choosed to include this value into the regression analysis or not. That is, if `False`, all values where there are 0 deaths are excluded from the regression. If `True` all values regardless of whether there is a death or not are included in the regression.
- **turnbull_estimator** (*('Nelson-Aalen', 'Kaplan-Meier', or 'Fleming-Harrington'), str, optional*) – If using the Turnbull heuristic, you can elect to use either the KM, NA, or FH estimator with the Turnbull estimates of r, and d. Defaults to FH.

Returns **model** – A parametric model with the fitted parameters and methods for all functions of the distribution using the fitted parameters.

Return type *Parametric*

Examples

```
>>> from surpyval import Weibull
>>> import numpy as np
>>> x = Weibull.random(100, 10, 4)
>>> model = Weibull.fit(x)
>>> print(model)
Parametric SurPyval Model
=========================
Distribution        : Weibull
Fitted by           : MLE
Parameters          :
     alpha: 10.551521182640098
      beta: 3.792549834495306
>>> Weibull.fit(x, how='MPS', fixed={'alpha' : 10})
Parametric SurPyval Model
=========================
Distribution        : Weibull
Fitted by           : MPS
Parameters          :
     alpha: 10.0
      beta: 3.4314657446866836
>>> Weibull.fit(xl=x-1, xr=x+1, how='MPP')
Parametric SurPyval Model
=========================
Distribution        : Weibull
```

(continues on next page)

(continued from previous page)

```
Fitted by           : MPP
Parameters          :
     alpha: 9.943092756713078
      beta: 8.613016934518258
>>> c = np.zeros_like(x)
>>> c[x > 13] = 1
>>> x[x > 13] = 13
>>> c = c[x > 6]
>>> x = x[x > 6]
>>> Weibull.fit(x=x, c=c, tl=6)
Parametric SurPyval Model
=========================
Distribution        : Weibull
Fitted by           : MLE
Parameters          :
     alpha: 10.363725328793413
      beta: 4.9886821457305865
```

fit_from_df(*df*, *x=None*, *c=None*, *n=None*, *xl=None*, *xr=None*, *tl=None*, *tr=None*, ***fit_options*)

The central feature to SurPyval's capability. This function aimed to have an API to mimic the simplicity of the scipy API. That is, to use a simple `fit()` call, with as many or as few parameters as is needed.

Parameters

- **df** (*DataFrame*) – DataFrame of data to be used to create surpyval model
- **x** (*string, optional*) – column name for the column in df containing the variable data. If not provided must provide both xl and xr
- **c** (*string, optional*) – column name for the column in df containing the censor flag of x. If not provided assumes all values of x are observed.
- **n** (*string, optional*) – column name in for the column in df containing the counts of x. If not provided assumes each x is one observation.
- **tl** (*string or scalar, optional*) – If string, column name in for the column in df containing the left truncation data. If scalar assumes each x is left truncated by that value. If not provided assumes x is not left truncated.
- **tr** (*string or scalar, optional*) – If string, column name in for the column in df containing the right truncation data. If scalar assumes each x is right truncated by that value. If not provided assumes x is not right truncated.
- **xl** (*string, optional*) – column name for the column in df containing the left interval for interval censored data. If left interval is -Inf, assumes left censored. If xl[i] == xr[i] assumes observed. Cannot be provided with x, must be provided with xr.
- **xr** (*string, optional*) – column name for the column in df containing the right interval for interval censored data. If right interval is Inf, assumes right censored. If xl[i] == xr[i] assumes observed. Cannot be provided with x, must be provided with xl.
- **fit_options** (*dict, optional*) – dictionary of fit options that will be passed to the `fit` method, see that method for options.

Returns **model** – A parametric model with the fitted parameters and methods for all functions of the distribution using the fitted parameters.

Return type *Parametric*

Examples

```
>>> import surpyval as surv
>>> df = surv.datasets.BoforsSteel.df
>>> model = surv.Weibull.fit_from_df(df, x='x', n='n', offset=True)
>>> print(model)
Parametric SurPyval Model
=========================
Distribution        : Weibull
Fitted by           : MLE
Offset (gamma)      : 39.76562962867477
Parameters          :
     alpha: 7.141925216146524
      beta: 2.6204524040137844
```

from_params (*params*, *gamma=None*, *p=None*, *f0=None*)
Creating a SurPyval Parametric class with provided parameters.

Parameters

- **params** (*array like*) – array of the parameters of the distribution.
- **gamma** (*scalar, optional*) – offset value for the distribution. If not provided will fit a regular, unshifted/not offset, distribution.

Returns **model** – A parametric model with the fitted parameters and methods for all functions of the distribution using the fitted parameters.

Return type *Parametric*

Examples

```
>>> from surpyval import Weibull
>>> model = Weibull.from_params([10, 4])
>>> print(model)
Parametric SurPyval Model
=========================
Distribution        : Weibull
Fitted by           : given parameters
Parameters          :
     alpha: 10
      beta: 4
>>> model = Weibull.from_params([10, 4], gamma=2)
>>> print(model)
Parametric SurPyval Model
=========================
Distribution        : Weibull
Fitted by           : given parameters
Offset (gamma)      : 2
Parameters          :
     alpha: 10
      beta: 4
```

hf (*x*, *mu*, *sigma*)
Instantaneous hazard rate for the LogNormal Distribution:

$$h(x) = \frac{f(x)}{R(x)}$$

Parameters

- **x** (*numpy array or scalar*) – The values at which the function will be calculated
- **mu** (*numpy array or scalar*) – The location parameter for the LogNormal distribution
- **sigma** (*numpy array or scalar*) – The scale parameter for the LogNormal distribution

Returns **hf** – The value(s) of the instantaneous hazard rate at x.

Return type scalar or numpy array

Examples

```
>>> import numpy as np
>>> from surpyval import LogNormal
>>> x = np.array([1, 2, 3, 4, 5])
>>> LogNormal.hf(x, 3, 4)
array([0.09734551, 0.05881839, 0.04349249, 0.03500202, 0.02952687])
```

mean (*mu, sigma*)

Quantile function for the LogNormal Distribution:

$$E = e^{\mu + \frac{\sigma^2}{2}}$$

Parameters

- **p** (*numpy array or scalar*) – The percentiles at which the quantile will be calculated
- **mu** (*numpy array or scalar*) – The location parameter for the LogNormal distribution
- **sigma** (*numpy array or scalar*) – The scale parameter for the LogNormal distribution

Returns **q** – The quantiles for the LogNormal distribution at each value p.

Return type scalar or numpy array

Examples

```
>>> from surpyval import LogNormal
>>> LogNormal.mean(3, 4)
59874.14171519782
```

moment (*n, mu, sigma*)

n-th (non central) moment of the LogNormal distribution

$$E = ...complicated.$$

Parameters

- **n** (*integer or numpy array of integers*) – The ordinal of the moment to calculate

- **mu** (*numpy array or scalar*) – The location parameter for the LogNormal distribution
- **sigma** (*numpy array or scalar*) – The scale parameter for the LogNormal distribution

Returns **moment** – The moment(s) of the LogNormal distribution

Return type scalar or numpy array

Examples

```
>>> from surpyval import LogNormal
>>> LogNormal.moment(2, 3, 4)
3.1855931757113756e+16
```

qf (*p*, *mu*, *sigma*)

Quantile function for the LogNormal Distribution:

$$q(p) = e^{\mu + \sigma\Phi^{-1}(p)}$$

Parameters

- **p** (*numpy array or scalar*) – The percentiles at which the quantile will be calculated
- **mu** (*numpy array or scalar*) – The location parameter for the LogNormal distribution
- **sigma** (*numpy array or scalar*) – The scale parameter for the LogNormal distribution

Returns **q** – The quantiles for the LogNormal distribution at each value p.

Return type scalar or numpy array

Examples

```
>>> import numpy as np
>>> from surpyval import LogNormal
>>> p = np.array([.1, .2, .3, .4, .5])
>>> LogNormal.qf(p, 3, 4)
array([ 0.11928899,  0.69316658,  2.46550819,  7.29078766, 20.08553692])
```

random (*size*, *mu*, *sigma*)

Draws random samples from the distribution in shape *size*

Parameters

- **size** (*integer or tuple of positive integers*) – Shape or size of the random draw
- **mu** (*numpy array or scalar*) – The location parameter for the LogNormal distribution
- **sigma** (*numpy array or scalar*) – The scale parameter for the LogNormal distribution

Returns **random** – Random values drawn from the distribution in shape *size*

Return type scalar or numpy array

Examples

```
>>> import numpy as np
>>> from surpyval import LogNormal
>>> LogNormal.random(10, 3, 4)
array([1.74605298e+00, 1.90729963e+02, 1.90090366e+03, 2.59154042e-02,
       3.71460694e-02, 3.38580771e+03, 7.58826512e+04, 7.23252303e+00,
       1.21226718e+03, 4.15054624e+00])
>>> LogNormal.random((5, 5), 3, 4)
array([[4.59689256e+00, 2.91472936e-01, 4.66833783e+02, 9.88539048e+01,
        3.88094471e+01],
       [7.10705735e-01, 5.00788529e-02, 2.49032431e+01, 2.19196376e+01,
        2.05043988e+02],
       [1.32193999e+03, 7.38943238e-01, 5.16503535e-01, 9.09249819e+02,
        2.69407879e+03],
       [7.29473033e+00, 5.68246498e+03, 1.74464896e+00, 1.26043004e+00,
        3.84009666e+03],
       [1.47997384e+00, 2.21809242e+02, 1.32564564e+02, 8.06883052e-02,
        1.05118538e+02]])
```

sf (*x*, *mu*, *sigma*)

Surival (or Reliability) function for the LogNormal Distribution:

$$R(x) = 1 - \Phi\left(\frac{\ln(x) - \mu}{\sigma}\right)$$

Parameters

- **x** (*numpy array or scalar*) – The values at which the function will be calculated
- **mu** (*numpy array or scalar*) – The location parameter for the LogNormal distribution
- **sigma** (*numpy array or scalar*) – The scale parameter for the LogNormal distribution

Returns **sf** – The value(s) of the survival function at x.

Return type scalar or numpy array

Examples

```
>>> import numpy as np
>>> from surpyval import LogNormal
>>> x = np.array([1, 2, 3, 4, 5])
>>> LogNormal.sf(x, 3, 4)
array([0.77337265, 0.71793339, 0.68273014, 0.65668272, 0.63594491])
```

Normal

class surpyval.parametric.normal.**Normal_**(*name*)

Bases: *surpyval.parametric.parametric_fitter.ParametricFitter*

Class used to generate the Normal (Gauss) class.

```
from surpyval import Normal
```

Hf (*x*, *mu*, *sigma*)
Cumulative hazard rate for the Normal Distribution:

$$H(x) = -\ln\left(1 - \Phi\left(\frac{x-\mu}{\sigma}\right)\right)$$

Parameters

- **x** (*numpy array or scalar*) – The values at which the function will be calculated
- **mu** (*numpy array or scalar*) – The location parameter for the Normal distribution
- **sigma** (*numpy array or scalar*) – The scale parameter for the Normal distribution

Returns **ff** – The value(s) of the cumulative hazard rate function at x.

Return type scalar or numpy array

Examples

```
>>> import numpy as np
>>> from surpyval import Normal
>>> x = np.array([1, 2, 3, 4, 5])
>>> Normal.Hf(x, 3, 4)
array([0.36894642, 0.51298408, 0.69314718, 0.91306176, 1.17591176])
```

cs (*x*, *X*, *mu*, *sigma*)
Conditional survival function for the Normal Distribution:

$$R(x, X) = \frac{R(x+X)}{R(X)}$$

Parameters

- **x** (*numpy array or scalar*) – The value(s) at which the function will be calculated
- **X** (*numpy array or scalar*) – The value(s) at which each value(s) in x was known to have survived
- **mu** (*numpy array or scalar*) – The location parameter for the Normal distribution
- **sigma** (*numpy array or scalar*) – The scale parameter for the Normal distribution

Returns **cs** – the conditional survival probability at x

Return type scalar or numpy array

Examples

```
>>> import numpy as np
>>> from surpyval import Normal
>>> x = np.array([1, 2, 3, 4, 5])
>>> Normal.cs(x, 5, 3, 4)
array([0.73452116, 0.51421702, 0.34242113, 0.2165286 , 0.1298356 ])
```

df (*x*, *mu*, *sigma*)

Density function for the Normal Distribution:

$$f(x) = \frac{1}{\sigma\sqrt{2\pi}}e^{-\frac{1}{2}\left(\frac{x-\mu}{\sigma}\right)^2}$$

Parameters

- **x** (*numpy array or scalar*) – The values at which the function will be calculated
- **mu** (*numpy array or scalar*) – The location parameter for the Normal distribution
- **sigma** (*numpy array or scalar*) – The scale parameter for the Normal distribution

Returns **df** – The value(s) of the density function at x.

Return type scalar or numpy array

Examples

```
>>> import numpy as np
>>> from surpyval import Normal
>>> x = np.array([1, 2, 3, 4, 5])
>>> Normal.df(x, 3, 4)
array([0.08801633, 0.09666703, 0.09973557, 0.09666703, 0.08801633])
```

ff (*x*, *mu*, *sigma*)

CDF (or unreliability or failure) function for the Normal Distribution:

$$F(x) = \Phi\left(\frac{x-\mu}{\sigma}\right)$$

Parameters

- **x** (*numpy array or scalar*) – The values at which the function will be calculated
- **mu** (*numpy array or scalar*) – The location parameter for the Normal distribution
- **sigma** (*numpy array or scalar*) – The scale parameter for the Normal distribution

Returns **ff** – The value(s) of the failure function at x.

Return type scalar or numpy array

Examples

```
>>> import numpy as np
>>> from surpyval import Normal
>>> x = np.array([1, 2, 3, 4, 5])
>>> Normal.ff(x, 3, 4)
array([0.30853754, 0.40129367, 0.5       , 0.59870633, 0.69146246])
```

fit (*x=None*, *c=None*, *n=None*, *t=None*, *how='MLE'*, *offset=False*, *zi=False*, *lfp=False*, *tl=None*, *tr=None*, *xl=None*, *xr=None*, *fixed=None*, *heuristic='Turnbull'*, *init=[]*, *rr='y'*, *on_d_is_0=False*, *turnbull_estimator='Fleming-Harrington'*)

The central feature to SurPyval's capability. This function aimed to have an API to mimic the simplicity of the scipy API. That is, to use a simple `fit()` call, with as many or as few parameters as is needed.

Parameters

- **x** (*array like, optional*) – Array of observations of the random variables. If x is `None`, xl and xr must be provided.
- **c** (*array like, optional*) – Array of censoring flag. -1 is left censored, 0 is observed, 1 is right censored, and 2 is intervally censored. If not provided will assume all values are observed.
- **n** (*array like, optional*) – Array of counts for each x. If data is proivded as counts, then this can be provided. If `None` will assume each observation is 1.
- **t** (*2D-array like, optional*) – 2D array like of the left and right values at which the respective observation was truncated. If not provided it assumes that no truncation occurs.
- **how** (*{'MLE', 'MPP', 'MOM', 'MSE', 'MPS'}, optional*) – Method to estimate parameters, these are:
  - MLE : Maximum Likelihood Estimation
  - MPP : Method of Probability Plotting
  - MOM : Method of Moments
  - MSE : Mean Square Error
  - MPS : Maximum Product Spacing
- **offset** (*boolean, optional*) – If `True` finds the shifted distribution. If not provided assumes not a shifted distribution. Only works with distributions that are supported on the half-real line.
- **tl** (*array like or scalar, optional*) – Values of left truncation for observations. If it is a scalar value assumes each observation is left truncated at the value. If an array, it is the respective 'late entry' of the observation
- **tr** (*array like or scalar, optional*) – Values of right truncation for observations. If it is a scalar value assumes each observation is right truncated at the value. If an array, it is the respective right truncation value for each observation
- **xl** (*array like, optional*) – Array like of the left array for 2-dimensional input of x. This is useful for data that is all intervally censored. Must be used with the `xr` input.
- **xr** (*array like, optional*) – Array like of the right array for 2-dimensional input of x. This is useful for data that is all intervally censored. Must be used with the `xl` input.
- **fixed** (*dict, optional*) – Dictionary of parameters and their values to fix. Fixes parameter by name.
- **heuristic** (*{'"Blom", "Median", "ECDF", "Modal", "Midpoint", "Mean", "Weibull", "Benard", "Beard", "Hazen", "Gringorten", "None", "Tukey", "DPW", "Fleming-Harrington", "Kaplan-Meier", "Nelson-Aalen", "Filliben", "Larsen", "Turnbull"}*) – Plotting method to use, if using the probability plotting, MPP, method.
- **init** (*array like, optional*) – initial guess of parameters. Useful if method is failing.
- **rr** (*('y', 'x')*) – The dimension on which to minimise the spacing between the line and the observation. If 'y' the mean square error between the line and vertical distance

to each point is minimised. If 'x' the mean square error between the line and horizontal distance to each point is minimised.

- **on_d_is_0** (*boolean, optional*) – For the case when using MPP and the highest value is right censored, you can choosed to include this value into the regression analysis or not. That is, if `False`, all values where there are 0 deaths are excluded from the regression. If `True` all values regardless of whether there is a death or not are included in the regression.
- **turnbull_estimator** (*('Nelson-Aalen', 'Kaplan-Meier', or 'Fleming-Harrington'), str, optional*) – If using the Turnbull heuristic, you can elect to use either the KM, NA, or FH estimator with the Turnbull estimates of r, and d. Defaults to FH.

Returns model – A parametric model with the fitted parameters and methods for all functions of the distribution using the fitted parameters.

Return type *Parametric*

Examples

```
>>> from surpyval import Weibull
>>> import numpy as np
>>> x = Weibull.random(100, 10, 4)
>>> model = Weibull.fit(x)
>>> print(model)
Parametric SurPyval Model
=========================
Distribution        : Weibull
Fitted by           : MLE
Parameters          :
     alpha: 10.551521182640098
      beta: 3.792549834495306
>>> Weibull.fit(x, how='MPS', fixed={'alpha' : 10})
Parametric SurPyval Model
=========================
Distribution        : Weibull
Fitted by           : MPS
Parameters          :
     alpha: 10.0
      beta: 3.4314657446866836
>>> Weibull.fit(xl=x-1, xr=x+1, how='MPP')
Parametric SurPyval Model
=========================
Distribution        : Weibull
Fitted by           : MPP
Parameters          :
     alpha: 9.943092756713078
      beta: 8.613016934518258
>>> c = np.zeros_like(x)
>>> c[x > 13] = 1
>>> x[x > 13] = 13
>>> c = c[x > 6]
>>> x = x[x > 6]
>>> Weibull.fit(x=x, c=c, tl=6)
Parametric SurPyval Model
=========================
```

(continues on next page)

(continued from previous page)

```
Distribution        : Weibull
Fitted by           : MLE
Parameters          :
     alpha: 10.363725328793413
      beta: 4.9886821457305865
```

fit_from_df (*df*, *x=None*, *c=None*, *n=None*, *xl=None*, *xr=None*, *tl=None*, *tr=None*, ***fit_options*)

The central feature to SurPyval's capability. This function aimed to have an API to mimic the simplicity of the scipy API. That is, to use a simple `fit()` call, with as many or as few parameters as is needed.

Parameters

- **df** (*DataFrame*) – DataFrame of data to be used to create surpyval model
- **x** (*string, optional*) – column name for the column in df containing the variable data. If not provided must provide both xl and xr
- **c** (*string, optional*) – column name for the column in df containing the censor flag of x. If not provided assumes all values of x are observed.
- **n** (*string, optional*) – column name in for the column in df containing the counts of x. If not provided assumes each x is one observation.
- **tl** (*string or scalar, optional*) – If string, column name in for the column in df containing the left truncation data. If scalar assumes each x is left truncated by that value. If not provided assumes x is not left truncated.
- **tr** (*string or scalar, optional*) – If string, column name in for the column in df containing the right truncation data. If scalar assumes each x is right truncated by that value. If not provided assumes x is not right truncated.
- **xl** (*string, optional*) – column name for the column in df containing the left interval for interval censored data. If left interval is -Inf, assumes left censored. If xl[i] == xr[i] assumes observed. Cannot be provided with x, must be provided with xr.
- **xr** (*string, optional*) – column name for the column in df containing the right interval for interval censored data. If right interval is Inf, assumes right censored. If xl[i] == xr[i] assumes observed. Cannot be provided with x, must be provided with xl.
- **fit_options** (*dict, optional*) – dictionary of fit options that will be passed to the `fit` method, see that method for options.

Returns **model** – A parametric model with the fitted parameters and methods for all functions of the distribution using the fitted parameters.

Return type *Parametric*

Examples

```
>>> import surpyval as surv
>>> df = surv.datasets.BoforsSteel.df
>>> model = surv.Weibull.fit_from_df(df, x='x', n='n', offset=True)
>>> print(model)
Parametric SurPyval Model
=========================
Distribution        : Weibull
Fitted by           : MLE
Offset (gamma)      : 39.76562962867477
```

(continues on next page)

(continued from previous page)

```
Parameters           :
     alpha: 7.141925216146524
      beta: 2.6204524040137844
```

from_params (*params*, *gamma=None*, *p=None*, *f0=None*)
Creating a SurPyval Parametric class with provided parameters.

Parameters

- **params** (*array like*) – array of the parameters of the distribution.
- **gamma** (*scalar, optional*) – offset value for the distribution. If not provided will fit a regular, unshifted/not offset, distribution.

Returns model – A parametric model with the fitted parameters and methods for all functions of the distribution using the fitted parameters.

Return type *Parametric*

Examples

```
>>> from surpyval import Weibull
>>> model = Weibull.from_params([10, 4])
>>> print(model)
Parametric SurPyval Model
=========================
Distribution         : Weibull
Fitted by            : given parameters
Parameters           :
     alpha: 10
      beta: 4
>>> model = Weibull.from_params([10, 4], gamma=2)
>>> print(model)
Parametric SurPyval Model
=========================
Distribution         : Weibull
Fitted by            : given parameters
Offset (gamma)       : 2
Parameters           :
     alpha: 10
      beta: 4
```

hf (*x*, *mu*, *sigma*)
Instantaneous hazard rate for the Normal Distribution:

$$h(x) = \frac{\frac{1}{\sigma\sqrt{2\pi}}e^{-\frac{1}{2}\left(\frac{x-\mu}{\sigma}\right)^2}}{1-\Phi\left(\frac{x-\mu}{\sigma}\right)}$$

Parameters

- **x** (*numpy array or scalar*) – The values at which the function will be calculated
- **mu** (*numpy array or scalar*) – The location parameter for the Normal distribution
- **sigma** (*numpy array or scalar*) – The scale parameter for the Normal distribution

Returns hf – The value(s) of the instantaneous hazard rate function at x.

Return type scalar or numpy array

Examples

```
>>> import numpy as np
>>> from surpyval import Normal
>>> x = np.array([1, 2, 3, 4, 5])
>>> Normal.hf(x, 3, 4)
array([0.12729011, 0.16145984, 0.19947114, 0.24088849, 0.28526944])
```

`mean` (*mu*, *sigma*)
Mean of the Normal distribution

$$E = \mu$$

Parameters

- **mu** (*numpy array or scalar*) – The location parameter for the Normal distribution
- **sigma** (*numpy array or scalar*) – The scale parameter for the Normal distribution

Returns **mu** – The mean(s) of the Normal distribution

Return type scalar or numpy array

Examples

```
>>> from surpyval import Normal
>>> Normal.mean(3, 4)
3
```

`moment` (*n*, *mu*, *sigma*)
n-th (non central) moment of the Normal distribution

$$E = ...complicated.$$

Parameters

- **n** (*integer or numpy array of integers*) – The ordinal of the moment to calculate
- **mu** (*numpy array or scalar*) – The location parameter for the Normal distribution
- **sigma** (*numpy array or scalar*) – The scale parameter for the Normal distribution

Returns **moment** – The moment(s) of the Normal distribution

Return type scalar or numpy array

Examples

```
>>> from surpyval import Normal
>>> Normal.moment(2, 3, 4)
25.0
```

qf (*p*, *mu*, *sigma*)

Quantile function for the Normal Distribution:

$$q(p) = \Phi^{-1}(p)$$

Parameters

- **p** (*numpy array or scalar*) – The percentiles at which the quantile will be calculated
- **mu** (*numpy array or scalar*) – The location parameter for the Normal distribution
- **sigma** (*numpy array or scalar*) – The scale parameter for the Normal distribution

Returns **q** – The quantiles for the Normal distribution at each value p.

Return type scalar or numpy array

Examples

```
>>> import numpy as np
>>> from surpyval import Normal
>>> p = np.array([.1, .2, .3, .4, .5])
>>> Normal.qf(p, 3, 4)
array([-2.12620626, -0.36648493,  0.90239795,  1.98661159,  3.        ])
```

random (*size*, *mu*, *sigma*)

Draws random samples from the distribution in shape *size*

Parameters

- **size** (*integer or tuple of positive integers*) – Shape or size of the random draw
- **mu** (*numpy array or scalar*) – The location parameter for the Normal distribution
- **sigma** (*numpy array or scalar*) – The scale parameter for the Normal distribution

Returns **random** – Random values drawn from the distribution in shape *size*

Return type scalar or numpy array

Examples

```
>>> import numpy as np
>>> from surpyval import Normal
>>> Normal.random(10, 3, 4)
array([-1.28484969, -1.68138703,  0.13414348,  6.53416927, -1.95649712,
        3.09951162,  6.90469836,  4.90063467,  1.11075072,  4.97841115])
>>> Normal.random((5, 5), 3, 4)
array([[ 1.57569952,  4.98472487,  3.19475597,  5.12581251, -0.98020861],
       [ 6.73877217,  1.08561611,  3.07634125,  3.54656313, 13.32064634],
       [-0.45094731,  2.52588422, -1.61414841,  8.39084564, -1.35261631],
       [ 1.98090151,  8.22151826,  5.59184063, -2.62221656,  0.20879673],
       [-2.0790734 ,  2.67886095,  2.54115153,  5.49853925,  4.57056015]])
```

sf (*x*, *mu*, *sigma*)

Surival (or Reliability) function for the Normal Distribution:

$$R(x) = 1 - \Phi\left(\frac{x-\mu}{\sigma}\right)$$

Parameters

- **x** (*numpy array or scalar*) – The values at which the function will be calculated
- **mu** (*numpy array or scalar*) – The location parameter for the Normal distribution
- **sigma** (*numpy array or scalar*) – The scale parameter for the Normal distribution

Returns **sf** – The value(s) of the survival function at x.

Return type scalar or numpy array

Examples

```
>>> import numpy as np
>>> from surpyval import Normal
>>> x = np.array([1, 2, 3, 4, 5])
>>> Normal.sf(x, 3, 4)
array([0.69146246, 0.59870633, 0.5       , 0.40129367, 0.30853754])
```

Uniform

class surpyval.parametric.uniform.**Uniform_**(*name*)

Bases: *surpyval.parametric.parametric_fitter.ParametricFitter*

Hf (*x*, *a*, *b*)

Instantaneous hazard rate for the Uniform Distribution:

$$H(x) = \ln(b-a) - \ln(b-x)$$

Parameters

- **x** (*numpy array or scalar*) – The values at which the function will be calculated
- **a** (*numpy array or scalar*) – The lower parameter for the Uniform distribution
- **b** (*numpy array or scalar*) – The upper parameter for the Uniform distribution

Returns **hf** – The value(s) of the instantaneous hazard rate at x.

Return type scalar or numpy array

Examples

```
>>> import numpy as np
>>> from surpyval import Uniform
>>> x = np.array([1, 2, 3, 4, 5])
>>> Uniform.Hf(x, 0, 6)
array([0.18232156, 0.40546511, 0.69314718, 1.09861229, 1.79175947])
```

cs (*x*, *X*, *a*, *b*)

Surival (or Reliability) function for the Uniform Distribution:

$$R(x, X) = \frac{R(x + X)}{R(X)}$$

Parameters

- **x** (*numpy array or scalar*) – The values at which the function will be calculated
- **a** (*numpy array or scalar*) – The lower parameter for the Uniform distribution
- **b** (*numpy array or scalar*) – The upper parameter for the Uniform distribution

Returns **cs** – The value(s) of the conditional survival function at x.

Return type scalar or numpy array

Examples

```
>>> import numpy as np
>>> from surpyval import Uniform
>>> x = np.array([1, 2, 3, 4, 5])
>>> Uniform.cs(x, 4, 0, 10)
array([0.83333333, 0.66666667, 0.5       , 0.33333333, 0.16666667])
```

df (*x*, *a*, *b*)

Failure (CDF or unreliability) function for the Uniform Distribution:

$$f(x) = \frac{1}{b - a}$$

Parameters

- **x** (*numpy array or scalar*) – The values at which the function will be calculated
- **a** (*numpy array or scalar*) – The lower parameter for the Uniform distribution
- **b** (*numpy array or scalar*) – The upper parameter for the Uniform distribution

Returns **df** – The value(s) of the density function at x.

Return type scalar or numpy array

Examples

```
>>> import numpy as np
>>> from surpyval import Uniform
>>> x = np.array([1, 2, 3, 4, 5])
>>> Uniform.df(x, 0, 6)
array([0.16666667, 0.16666667, 0.16666667, 0.16666667, 0.16666667])
```

ff (*x*, *a*, *b*)

Failure (CDF or unreliability) function for the Uniform Distribution:

$$F(x) = \frac{x - a}{b - a}$$

Parameters

- **x** (*numpy array or scalar*) – The values at which the function will be calculated

- **a** (*numpy array or scalar*) – The lower parameter for the Uniform distribution
- **b** (*numpy array or scalar*) – The upper parameter for the Uniform distribution

Returns **ff** – The value(s) of the failure function at x.

Return type scalar or numpy array

Examples

```
>>> import numpy as np
>>> from surpyval import Uniform
>>> x = np.array([1, 2, 3, 4, 5])
>>> Uniform.sf(x, 0, 6)
array([0.16666667, 0.33333333, 0.5       , 0.66666667, 0.83333333])
```

fit (*x=None*, *c=None*, *n=None*, *t=None*, *how='MLE'*, *offset=False*, *zi=False*, *lfp=False*, *tl=None*, *tr=None*, *xl=None*, *xr=None*, *fixed=None*, *heuristic='Turnbull'*, *init=[]*, *rr='y'*, *on_d_is_0=False*, *turnbull_estimator='Fleming-Harrington'*)

The central feature to SurPyval's capability. This function aimed to have an API to mimic the simplicity of the scipy API. That is, to use a simple `fit()` call, with as many or as few parameters as is needed.

Parameters

- **x** (*array like, optional*) – Array of observations of the random variables. If x is `None`, xl and xr must be provided.
- **c** (*array like, optional*) – Array of censoring flag. -1 is left censored, 0 is observed, 1 is right censored, and 2 is intervally censored. If not provided will assume all values are observed.
- **n** (*array like, optional*) – Array of counts for each x. If data is proivded as counts, then this can be provided. If `None` will assume each observation is 1.
- **t** (*2D-array like, optional*) – 2D array like of the left and right values at which the respective observation was truncated. If not provided it assumes that no truncation occurs.
- **how** (*{'MLE', 'MPP', 'MOM', 'MSE', 'MPS'}, optional*) – Method to estimate parameters, these are:
  - MLE : Maximum Likelihood Estimation
  - MPP : Method of Probability Plotting
  - MOM : Method of Moments
  - MSE : Mean Square Error
  - MPS : Maximum Product Spacing
- **offset** (*boolean, optional*) – If `True` finds the shifted distribution. If not provided assumes not a shifted distribution. Only works with distributions that are supported on the half-real line.
- **tl** (*array like or scalar, optional*) – Values of left truncation for observations. If it is a scalar value assumes each observation is left truncated at the value. If an array, it is the respective 'late entry' of the observation
- **tr** (*array like or scalar, optional*) – Values of right truncation for observations. If it is a scalar value assumes each observation is right truncated at the value. If an array, it is the respective right truncation value for each observation

- **xl** (*array like, optional*) – Array like of the left array for 2-dimensional input of x. This is useful for data that is all intervally censored. Must be used with the `xr` input.
- **xr** (*array like, optional*) – Array like of the right array for 2-dimensional input of x. This is useful for data that is all intervally censored. Must be used with the `xl` input.
- **fixed** (*dict, optional*) – Dictionary of parameters and their values to fix. Fixes parameter by name.
- **heuristic** (*{'"Blom", "Median", "ECDF", "Modal", "Midpoint", "Mean", "Weibull", "Benard", "Beard", "Hazen", "Gringorten", "None", "Tukey", "DPW", "Fleming-Harrington", "Kaplan-Meier", "Nelson-Aalen", "Filliben", "Larsen", "Turnbull"}*) – Plotting method to use, if using the probability plotting, MPP, method.
- **init** (*array like, optional*) – initial guess of parameters. Useful if method is failing.
- **rr** (*('y', 'x')*) – The dimension on which to minimise the spacing between the line and the observation. If 'y' the mean square error between the line and vertical distance to each point is minimised. If 'x' the mean square error between the line and horizontal distance to each point is minimised.
- **on_d_is_0** (*boolean, optional*) – For the case when using MPP and the highest value is right censored, you can choosed to include this value into the regression analysis or not. That is, if `False`, all values where there are 0 deaths are excluded from the regression. If `True` all values regardless of whether there is a death or not are included in the regression.
- **turnbull_estimator** (*('Nelson-Aalen', 'Kaplan-Meier', or 'Fleming-Harrington'), str, optional*) – If using the Turnbull heuristic, you can elect to use either the KM, NA, or FH estimator with the Turnbull estimates of r, and d. Defaults to FH.

Returns **model** – A parametric model with the fitted parameters and methods for all functions of the distribution using the fitted parameters.

Return type *Parametric*

Examples

```
>>> from surpyval import Weibull
>>> import numpy as np
>>> x = Weibull.random(100, 10, 4)
>>> model = Weibull.fit(x)
>>> print(model)
Parametric SurPyval Model
=========================
Distribution        : Weibull
Fitted by           : MLE
Parameters          :
     alpha: 10.551521182640098
      beta: 3.792549834495306
>>> Weibull.fit(x, how='MPS', fixed={'alpha' : 10})
Parametric SurPyval Model
=========================
Distribution        : Weibull
```

(continues on next page)

(continued from previous page)

```
Fitted by           : MPS
Parameters          :
     alpha: 10.0
      beta: 3.4314657446866836
>>> Weibull.fit(xl=x-1, xr=x+1, how='MPP')
Parametric SurPyval Model
=========================
Distribution        : Weibull
Fitted by           : MPP
Parameters          :
     alpha: 9.943092756713078
      beta: 8.613016934518258
>>> c = np.zeros_like(x)
>>> c[x > 13] = 1
>>> x[x > 13] = 13
>>> c = c[x > 6]
>>> x = x[x > 6]
>>> Weibull.fit(x=x, c=c, tl=6)
Parametric SurPyval Model
=========================
Distribution        : Weibull
Fitted by           : MLE
Parameters          :
     alpha: 10.363725328793413
      beta: 4.9886821457305865
```

fit_from_df(*df*, *x=None*, *c=None*, *n=None*, *xl=None*, *xr=None*, *tl=None*, *tr=None*, ***fit_options*)

The central feature to SurPyval's capability. This function aimed to have an API to mimic the simplicity of the scipy API. That is, to use a simple `fit()` call, with as many or as few parameters as is needed.

Parameters

- **df** (*DataFrame*) – DataFrame of data to be used to create surpyval model
- **x** (*string, optional*) – column name for the column in df containing the variable data. If not provided must provide both xl and xr
- **c** (*string, optional*) – column name for the column in df containing the censor flag of x. If not provided assumes all values of x are observed.
- **n** (*string, optional*) – column name in for the column in df containing the counts of x. If not provided assumes each x is one observation.
- **tl** (*string or scalar, optional*) – If string, column name in for the column in df containing the left truncation data. If scalar assumes each x is left truncated by that value. If not provided assumes x is not left truncated.
- **tr** (*string or scalar, optional*) – If string, column name in for the column in df containing the right truncation data. If scalar assumes each x is right truncated by that value. If not provided assumes x is not right truncated.
- **xl** (*string, optional*) – column name for the column in df containing the left interval for interval censored data. If left interval is -Inf, assumes left censored. If xl[i] == xr[i] assumes observed. Cannot be provided with x, must be provided with xr.
- **xr** (*string, optional*) – column name for the column in df containing the right interval for interval censored data. If right interval is Inf, assumes right censored. If xl[i] == xr[i] assumes observed. Cannot be provided with x, must be provided with xl.

- **fit_options** (*dict, optional*) – dictionary of fit options that will be passed to the `fit` method, see that method for options.

Returns model – A parametric model with the fitted parameters and methods for all functions of the distribution using the fitted parameters.

Return type *Parametric*

Examples

```
>>> import surpyval as surv
>>> df = surv.datasets.BoforsSteel.df
>>> model = surv.Weibull.fit_from_df(df, x='x', n='n', offset=True)
>>> print(model)
Parametric SurPyval Model
=========================
Distribution        : Weibull
Fitted by           : MLE
Offset (gamma)      : 39.76562962867477
Parameters          :
     alpha: 7.141925216146524
      beta: 2.6204524040137844
```

from_params (*params*, *gamma=None*, *p=None*, *f0=None*)
Creating a SurPyval Parametric class with provided parameters.

Parameters

- **params** (*array like*) – array of the parameters of the distribution.
- **gamma** (*scalar, optional*) – offset value for the distribution. If not provided will fit a regular, unshifted/not offset, distribution.

Returns model – A parametric model with the fitted parameters and methods for all functions of the distribution using the fitted parameters.

Return type *Parametric*

Examples

```
>>> from surpyval import Weibull
>>> model = Weibull.from_params([10, 4])
>>> print(model)
Parametric SurPyval Model
=========================
Distribution        : Weibull
Fitted by           : given parameters
Parameters          :
     alpha: 10
      beta: 4
>>> model = Weibull.from_params([10, 4], gamma=2)
>>> print(model)
Parametric SurPyval Model
=========================
Distribution        : Weibull
Fitted by           : given parameters
Offset (gamma)      : 2
```

(continues on next page)

(continued from previous page)

```
Parameters          :
     alpha: 10
      beta: 4
```

hf (*x*, *a*, *b*)

Instantaneous hazard rate for the Uniform Distribution:

$$h(x) = \frac{1}{b - x}$$

Parameters

- **x** (*numpy array or scalar*) – The values at which the function will be calculated
- **a** (*numpy array or scalar*) – The lower parameter for the Uniform distribution
- **b** (*numpy array or scalar*) – The upper parameter for the Uniform distribution

Returns **hf** – The value(s) of the instantaneous hazard rate at x.

Return type scalar or numpy array

Examples

```
>>> import numpy as np
>>> from surpyval import Uniform
>>> x = np.array([1, 2, 3, 4, 5])
>>> Uniform.hf(x, 0, 6)
array([0.2       , 0.25      , 0.33333333, 0.5       , 1.        ])
```

mean (*a*, *b*)

Mean of the Uniform distribution

$$E = \frac{1}{2}(a + b)$$

Parameters

- **a** (*numpy array or scalar*) – The lower parameter for the Uniform distribution
- **b** (*numpy array or scalar*) – The upper parameter for the Uniform distribution

Returns **mean** – The mean(s) of the Uniform distribution

Return type scalar or numpy array

Examples

```
>>> from surpyval import Uniform
>>> Uniform.mean(0, 6)
3.0
```

moment (*n*, *a*, *b*)

n-th (non central) moment of the Uniform distribution

$$M(n) = \frac{1}{n+1}\sum_{i=0}^{n} a^i b^{n-i}$$

Parameters

- **n** (*integer or numpy array of integers*) – The ordinal of the moment to calculate
- **a** (*numpy array or scalar*) – The lower parameter for the Uniform distribution
- **b** (*numpy array or scalar*) – The upper parameter for the Uniform distribution

Returns **moment** – The moment(s) of the Uniform distribution

Return type scalar or numpy array

Examples

```
>>> from surpyval import Uniform
>>> Uniform.moment(2, 0, 6)
12.0
```

qf (*p*, *a*, *b*)

Quantile function for the Uniform Distribution:

$$q(p) = a + p(b - a)$$

Parameters

- **p** (*numpy array or scalar*) – The percentiles at which the quantile will be calculated
- **a** (*numpy array or scalar*) – The lower parameter for the Uniform distribution
- **b** (*numpy array or scalar*) – The upper parameter for the Uniform distribution

Returns **q** – The quantiles for the Uniform distribution at each value p.

Return type scalar or numpy array

Examples

```
>>> import numpy as np
>>> from surpyval import Uniform
>>> p = np.array([.1, .2, .3, .4, .5])
>>> Uniform.qf(p, 0, 6)
array([0.6, 1.2, 1.8, 2.4, 3. ])
```

random (*size*, *a*, *b*)

Draws random samples from the distribution in shape *size*

Parameters

- **size** (*integer or tuple of positive integers*) – Shape or size of the random draw
- **a** (*numpy array or scalar*) – The lower parameter for the Uniform distribution
- **b** (*numpy array or scalar*) – The upper parameter for the Uniform distribution

Returns **random** – Random values drawn from the distribution in shape *size*

Return type scalar or numpy array

Examples

```
>>> import numpy as np
>>> from surpyval import Uniform
>>> Uniform.random(10, 0, 6)
array([3.50214341, 3.7978912 , 5.12238656, 4.27185221, 3.05507685,
       2.71236199, 4.89311322, 1.11373047, 4.90549424, 1.76321338])
>>> Uniform.random((5, 5), 0, 6)
array([[4.76809829, 4.42155933, 2.59469997, 4.31525748, 5.53469545],
       [0.06222942, 1.26267164, 1.74188626, 1.05235807, 0.92461476],
       [2.06215303, 0.02184135, 0.97058002, 3.02219656, 3.22137982],
       [2.14951891, 3.18096661, 2.37105309, 0.65710124, 0.68828779],
       [0.58827207, 3.7633596 , 5.62330526, 5.24481753, 4.23162212]])
```

sf (*x*, *a*, *b*)
Surival (or Reliability) function for the Uniform Distribution:

$$R(x) = \frac{b - x}{b - a}$$

Parameters

- **x** (*numpy array or scalar*) – The values at which the function will be calculated
- **a** (*numpy array or scalar*) – The lower parameter for the Uniform distribution
- **b** (*numpy array or scalar*) – The upper parameter for the Uniform distribution

Returns **sf** – The value(s) of the survival function at x.

Return type scalar or numpy array

Examples

```
>>> import numpy as np
>>> from surpyval import Uniform
>>> x = np.array([1, 2, 3, 4, 5])
>>> Uniform.sf(x, 0, 6)
array([0.83333333, 0.66666667, 0.5       , 0.33333333, 0.16666667])
```

Weibull

class surpyval.parametric.weibull.**Weibull_**(*name*)
Bases: *surpyval.parametric.parametric_fitter.ParametricFitter*

Hf (*x*, *alpha*, *beta*)
Cumulative hazard rate for the Weibull Distribution:

$$h(x) = \frac{x}{\alpha}^{\beta}$$

Parameters

- **x** (*numpy array or scalar*) – The values at which the function will be calculated
- **alpha** (*numpy array or scalar*) – scale parameter for the Weibull distribution
- **beta** (*numpy array or scalar*) – shape parameter for the Weibull distribution

Returns **df** – The value(s) of the cumulative hazard rate at x.

Return type scalar or numpy array

Examples

```
>>> import numpy as np
>>> from surpyval import Weibull
>>> x = np.array([1, 2, 3, 4, 5])
>>> Weibull.Hf(x, 3, 4)
array([0.01234568, 0.19753086, 1.        , 3.16049383, 7.71604938])
```

`cs` (*x*, *X*, *alpha*, *beta*)
Conditional survival function for the Weibull Distribution:

$$R(x, X) = \frac{R(x+X)}{R(X)}$$

Parameters

- **x** (*numpy array or scalar*) – The values at which the function will be calculated
- **alpha** (*numpy array or scalar*) – scale parameter for the Weibull distribution
- **beta** (*numpy array or scalar*) – shape parameter for the Weibull distribution

Returns **cs** – The value(s) of the conditional survival function at x.

Return type scalar or numpy array

Examples

```
>>> import numpy as np
>>> from surpyval import Weibull
>>> x = np.array([1, 2, 3, 4, 5])
>>> Weibull.cs(x, 5, 3, 4)
array([2.21654222e+03, 1.84183662e+03, 8.25549630e+02, 9.51596070e+01,
       1.00000000e+00])
```

`df` (*x*, *alpha*, *beta*)
Density function for the Weibull Distribution:

$$f(x) = \frac{\beta}{\alpha}\frac{x}{\alpha}^{\beta-1}e^{-\left(\frac{x}{\alpha}\right)^{\beta}}$$

Parameters

- **x** (*numpy array or scalar*) – The values at which the function will be calculated
- **alpha** (*numpy array or scalar*) – scale parameter for the Weibull distribution
- **beta** (*numpy array or scalar*) – shape parameter for the Weibull distribution

Returns **df** – The value(s) of the conditional survival function at x.

Return type scalar or numpy array

Examples

```
>>> import numpy as np
>>> from surpyval import Weibull
>>> x = np.array([1, 2, 3, 4, 5])
>>> Weibull.df(x, 5, 3, 4)
array([0.0487768 , 0.32424881, 0.49050592, 0.13402009, 0.00275073])
```

ff (*x*, *alpha*, *beta*)

Failure (CDF or unreliability) function for the Weibull Distribution:

$$F(x) = 1 - e^{-\left(\frac{x}{\alpha}\right)^{\beta}}$$

Parameters

- **x** (*numpy array or scalar*) – The values at which the function will be calculated
- **alpha** (*numpy array or scalar*) – scale parameter for the Weibull distribution
- **beta** (*numpy array or scalar*) – shape parameter for the Weibull distribution

Returns **sf** – The value(s) of the failure function at x.

Return type scalar or numpy array

Examples

```
>>> import numpy as np
>>> from surpyval import Weibull
>>> x = np.array([1, 2, 3, 4, 5])
>>> Weibull.ff(x, 3, 4)
array([0.01226978, 0.17924519, 0.63212056, 0.9575952 , 0.99955438])
```

fit (*x=None*, *c=None*, *n=None*, *t=None*, *how='MLE'*, *offset=False*, *zi=False*, *lfp=False*, *tl=None*, *tr=None*, *xl=None*, *xr=None*, *fixed=None*, *heuristic='Turnbull'*, *init=[]*, *rr='y'*, *on_d_is_0=False*, *turnbull_estimator='Fleming-Harrington'*)

The central feature to SurPyval's capability. This function aimed to have an API to mimic the simplicity of the scipy API. That is, to use a simple `fit()` call, with as many or as few parameters as is needed.

Parameters

- **x** (*array like, optional*) – Array of observations of the random variables. If x is `None`, xl and xr must be provided.
- **c** (*array like, optional*) – Array of censoring flag. -1 is left censored, 0 is observed, 1 is right censored, and 2 is intervally censored. If not provided will assume all values are observed.
- **n** (*array like, optional*) – Array of counts for each x. If data is proivded as counts, then this can be provided. If `None` will assume each observation is 1.
- **t** (*2D-array like, optional*) – 2D array like of the left and right values at which the respective observation was truncated. If not provided it assumes that no truncation occurs.
- **how** (*{'MLE', 'MPP', 'MOM', 'MSE', 'MPS'}, optional*) – Method to estimate parameters, these are:
  - MLE : Maximum Likelihood Estimation

  - MPP : Method of Probability Plotting
  - MOM : Method of Moments
  - MSE : Mean Square Error
  - MPS : Maximum Product Spacing

- **offset** (*boolean, optional*) – If `True` finds the shifted distribution. If not provided assumes not a shifted distribution. Only works with distributions that are supported on the half-real line.
- **tl** (*array like or scalar, optional*) – Values of left truncation for observations. If it is a scalar value assumes each observation is left truncated at the value. If an array, it is the respective 'late entry' of the observation
- **tr** (*array like or scalar, optional*) – Values of right truncation for observations. If it is a scalar value assumes each observation is right truncated at the value. If an array, it is the respective right truncation value for each observation
- **xl** (*array like, optional*) – Array like of the left array for 2-dimensional input of x. This is useful for data that is all intervally censored. Must be used with the `xr` input.
- **xr** (*array like, optional*) – Array like of the right array for 2-dimensional input of x. This is useful for data that is all intervally censored. Must be used with the `xl` input.
- **fixed** (*dict, optional*) – Dictionary of parameters and their values to fix. Fixes parameter by name.
- **heuristic** (*{'"Blom", "Median", "ECDF", "Modal", "Midpoint", "Mean", "Weibull", "Benard", "Beard", "Hazen", "Gringorten", "None", "Tukey", "DPW", "Fleming-Harrington", "Kaplan-Meier", "Nelson-Aalen", "Filliben", "Larsen", "Turnbull"}*) – Plotting method to use, if using the probability plotting, MPP, method.
- **init** (*array like, optional*) – initial guess of parameters. Useful if method is failing.
- **rr** (*('y', 'x')*) – The dimension on which to minimise the spacing between the line and the observation. If 'y' the mean square error between the line and vertical distance to each point is minimised. If 'x' the mean square error between the line and horizontal distance to each point is minimised.
- **on_d_is_0** (*boolean, optional*) – For the case when using MPP and the highest value is right censored, you can choosed to include this value into the regression analysis or not. That is, if `False`, all values where there are 0 deaths are excluded from the regression. If `True` all values regardless of whether there is a death or not are included in the regression.
- **turnbull_estimator** (*('Nelson-Aalen', 'Kaplan-Meier', or 'Fleming-Harrington'), str, optional*) – If using the Turnbull heuristic, you can elect to use either the KM, NA, or FH estimator with the Turnbull estimates of r, and d. Defaults to FH.

Returns **model** – A parametric model with the fitted parameters and methods for all functions of the distribution using the fitted parameters.

Return type *Parametric*

Examples

```
>>> from surpyval import Weibull
>>> import numpy as np
>>> x = Weibull.random(100, 10, 4)
>>> model = Weibull.fit(x)
>>> print(model)
Parametric SurPyval Model
=========================
Distribution        : Weibull
Fitted by           : MLE
Parameters          :
     alpha: 10.551521182640098
      beta: 3.792549834495306
>>> Weibull.fit(x, how='MPS', fixed={'alpha' : 10})
Parametric SurPyval Model
=========================
Distribution        : Weibull
Fitted by           : MPS
Parameters          :
     alpha: 10.0
      beta: 3.4314657446866836
>>> Weibull.fit(xl=x-1, xr=x+1, how='MPP')
Parametric SurPyval Model
=========================
Distribution        : Weibull
Fitted by           : MPP
Parameters          :
     alpha: 9.943092756713078
      beta: 8.613016934518258
>>> c = np.zeros_like(x)
>>> c[x > 13] = 1
>>> x[x > 13] = 13
>>> c = c[x > 6]
>>> x = x[x > 6]
>>> Weibull.fit(x=x, c=c, tl=6)
Parametric SurPyval Model
=========================
Distribution        : Weibull
Fitted by           : MLE
Parameters          :
     alpha: 10.363725328793413
      beta: 4.9886821457305865
```

`fit_from_df` (*df*, *x=None*, *c=None*, *n=None*, *xl=None*, *xr=None*, *tl=None*, *tr=None*, ***fit_options*)
The central feature to SurPyval's capability. This function aimed to have an API to mimic the simplicity of the scipy API. That is, to use a simple `fit()` call, with as many or as few parameters as is needed.

Parameters

- **df** (*DataFrame*) – DataFrame of data to be used to create surpyval model
- **x** (*string, optional*) – column name for the column in df containing the variable data. If not provided must provide both xl and xr
- **c** (*string, optional*) – column name for the column in df containing the censor flag of x. If not provided assumes all values of x are observed.
- **n** (*string, optional*) – column name in for the column in df containing the counts of x. If not provided assumes each x is one observation.

- **tl** (*string or scalar, optional*) – If string, column name in for the column in df containing the left truncation data. If scalar assumes each x is left truncated by that value. If not provided assumes x is not left truncated.
- **tr** (*string or scalar, optional*) – If string, column name in for the column in df containing the right truncation data. If scalar assumes each x is right truncated by that value. If not provided assumes x is not right truncated.
- **xl** (*string, optional*) – column name for the column in df containing the left interval for interval censored data. If left interval is -Inf, assumes left censored. If xl[i] == xr[i] assumes observed. Cannot be provided with x, must be provided with xr.
- **xr** (*string, optional*) – column name for the column in df containing the right interval for interval censored data. If right interval is Inf, assumes right censored. If xl[i] == xr[i] assumes observed. Cannot be provided with x, must be provided with xl.
- **fit_options** (*dict, optional*) – dictionary of fit options that will be passed to the `fit` method, see that method for options.

Returns **model** – A parametric model with the fitted parameters and methods for all functions of the distribution using the fitted parameters.

Return type *Parametric*

Examples

```
>>> import surpyval as surv
>>> df = surv.datasets.BoforsSteel.df
>>> model = surv.Weibull.fit_from_df(df, x='x', n='n', offset=True)
>>> print(model)
Parametric SurPyval Model
=========================
Distribution        : Weibull
Fitted by           : MLE
Offset (gamma)      : 39.76562962867477
Parameters          :
     alpha: 7.141925216146524
      beta: 2.6204524040137844
```

from_params (*params*, *gamma=None*, *p=None*, *f0=None*)

Creating a SurPyval Parametric class with provided parameters.

Parameters

- **params** (*array like*) – array of the parameters of the distribution.
- **gamma** (*scalar, optional*) – offset value for the distribution. If not provided will fit a regular, unshifted/not offset, distribution.

Returns **model** – A parametric model with the fitted parameters and methods for all functions of the distribution using the fitted parameters.

Return type *Parametric*

Examples

```
>>> from surpyval import Weibull
>>> model = Weibull.from_params([10, 4])
>>> print(model)
Parametric SurPyval Model
=========================
Distribution        : Weibull
Fitted by           : given parameters
Parameters          :
     alpha: 10
      beta: 4
>>> model = Weibull.from_params([10, 4], gamma=2)
>>> print(model)
Parametric SurPyval Model
=========================
Distribution        : Weibull
Fitted by           : given parameters
Offset (gamma)      : 2
Parameters          :
     alpha: 10
      beta: 4
```

hf (*x*, *alpha*, *beta*)

Instantaneous hazard rate for the Weibull Distribution:

$$h(x) = \frac{\beta}{\alpha}\left(\frac{x}{\alpha}\right)^{\beta-1}$$

Parameters

- **x** (*numpy array or scalar*) – The values at which the function will be calculated
- **alpha** (*numpy array or scalar*) – scale parameter for the Weibull distribution
- **beta** (*numpy array or scalar*) – shape parameter for the Weibull distribution

Returns **df** – The value(s) of the instantaneous hazard rate at x.

Return type scalar or numpy array

Examples

```
>>> import numpy as np
>>> from surpyval import Weibull
>>> x = np.array([1, 2, 3, 4, 5])
>>> Weibull.hf(x, 3, 4)
array([0.04938272, 0.39506173, 1.33333333, 3.16049383, 6.17283951])
```

mean (*alpha*, *beta*)

Mean of the Weibull distribution

$$E = \alpha\Gamma\left(1 + \frac{1}{\beta}\right)$$

Parameters

- **alpha** (*numpy array or scalar*) – scale parameter for the Weibull distribution
- **beta** (*numpy array or scalar*) – shape parameter for the Weibull distribution

Returns **mean** – The mean(s) of the Weibull distribution

Return type scalar or numpy array

Examples

```
>>> from surpyval import Weibull
>>> Weibull.mean(3, 4)
2.7192074311664314
```

moment (*n*, *alpha*, *beta*)
n-th moment of the Weibull distribution

$$M(n) = \alpha^n \Gamma\left(1 + \frac{n}{\beta}\right)$$

Parameters

- **n** (*integer or numpy array of integers*) – The ordinal of the moment to calculate
- **alpha** (*numpy array or scalar*) – scale parameter for the Weibull distribution
- **beta** (*numpy array or scalar*) – shape parameter for the Weibull distribution

Returns **mean** – The moment(s) of the Weibull distribution

Return type scalar or numpy array

Examples

```
>>> from surpyval import Weibull
>>> Weibull.moment(2, 3, 4)
7.976042329074821
```

qf (*p*, *alpha*, *beta*)
Quantile function for the Weibull distribution:

$$q(p) = \alpha\left(-\ln\left(1 - p\right)\right)^{1/\beta}$$

Parameters

- **p** (*numpy array or scalar*) – The percentiles at which the quantile will be calculated
- **alpha** (*numpy array or scalar*) – scale parameter for the Weibull distribution
- **beta** (*numpy array or scalar*) – shape parameter for the Weibull distribution

Returns **q** – The quantiles for the Weibull distribution at each value p

Return type scalar or numpy array

Examples

```
>>> import numpy as np
>>> from surpyval import Weibull
>>> p = np.array([.1, .2, .3, .4, .5])
>>> Weibull.qf(p, 3, 4)
array([1.70919151, 2.06189877, 2.31840554, 2.5362346 , 2.73733292])
```

random (*size*, *alpha*, *beta*)
Draws random samples from the distribution in shape *size*

Parameters

- **size** (*integer or tuple of positive integers*) – Shape or size of the random draw
- **alpha** (*numpy array or scalar*) – scale parameter for the Weibull distribution
- **beta** (*numpy array or scalar*) – shape parameter for the Weibull distribution

Returns **random** – Random values drawn from the distribution in shape *size*

Return type scalar or numpy array

Examples

```
>>> import numpy as np
>>> from surpyval import Weibull
>>> Weibull.random(10, 3, 4)
array([1.79782451, 1.7143211 , 2.84778674, 3.12226231, 2.61000839,
       3.05456332, 3.00280851, 2.61910071, 1.37991527, 4.17488394])
>>> Weibull.random((5, 5), 3, 4)
array([[1.64782514, 2.79157632, 1.85500681, 2.91908736, 2.46089933],
       [1.85880127, 0.96787742, 2.29677031, 2.42394129, 2.63889601],
       [2.14351859, 3.90677225, 2.24013855, 2.49467774, 3.43755278],
       [3.24417396, 1.40775181, 2.49584969, 3.07603353, 2.54679499],
       [1.98330076, 2.95002633, 3.35402601, 3.11429283, 3.45706789]])
```

sf (*x*, *alpha*, *beta*)

Survival (or reliability) function for the Weibull Distribution:

$$R(x) = e^{-\left(\frac{x}{\alpha}\right)^\beta}$$

Parameters

- **x** (*numpy array or scalar*) – The values at which the function will be calculated
- **alpha** (*numpy array or scalar*) – scale parameter for the Weibull distribution
- **beta** (*numpy array or scalar*) – shape parameter for the Weibull distribution

Returns **sf** – The value(s) of the reliability function at x.

Return type scalar or numpy array

Examples

```
>>> import numpy as np
>>> from surpyval import Weibull
>>> x = np.array([1, 2, 3, 4, 5])
>>> Weibull.sf(x, 3, 4)
array([9.87730216e-01, 8.20754808e-01, 3.67879441e-01, 4.24047953e-02,
       4.45617596e-04])
```

Exponentiated Weibull

class surpyval.parametric.expo_weibull.**ExpoWeibull_**(*name*)

Bases: *surpyval.parametric.parametric_fitter.ParametricFitter*

Hf (*x*, *alpha*, *beta*, *mu*)

Instantaneous hazard rate for the ExpoWeibull Distribution:

$$H(x) = -\ln\left(R(x)\right)$$

Parameters

- **x** (*numpy array or scalar*) – The values at which the function will be calculated
- **alpha** (*numpy array or scalar*) – scale parameter for the ExpoWeibull distribution
- **beta** (*numpy array or scalar*) – shape parameter for the ExpoWeibull distribution
- **mu** (*numpy array or scalar*) – shape parameter for the ExpoWeibull distribution

Returns **Hf** – The value(s) of the cumulative hazard rate at x.

Return type scalar or numpy array

Examples

```
>>> import numpy as np
>>> from surpyval import ExpoWeibull
>>> x = np.array([1, 2, 3, 4, 5])
>>> ExpoWeibull.Hf(x, 3, 4, 1.2)
array([5.10166141e-03, 1.35931416e-01, 8.59705336e-01, 2.98247086e+00,
       7.53377239e+00])
```

cs (*x*, *X*, *alpha*, *beta*, *mu*)

Conditional survival (or reliability) function for the ExpoWeibull Distribution:

$$R(x, X) = \frac{R(x + X)}{R(X)}$$

Parameters

- **x** (*numpy array or scalar*) – The values at which the function will be calculated
- **alpha** (*numpy array or scalar*) – scale parameter for the ExpoWeibull distribution
- **beta** (*numpy array or scalar*) – shape parameter for the ExpoWeibull distribution
- **mu** (*numpy array or scalar*) – shape parameter for the ExpoWeibull distribution

Returns **sf** – The value(s) of the reliability function at x.

Return type scalar or numpy array

Examples

```
>>> import numpy as np
>>> from surpyval import ExpoWeibull
>>> x = np.array([1, 2, 3, 4, 5])
>>> ExpoWeibull.sf(x, 1, 3, 4, 1.2)
array([8.77367129e-01, 4.25451775e-01, 5.09266354e-02, 5.37452200e-04,
       1.35732908e-07])
```

df (*x*, *alpha*, *beta*, *mu*)

Density function for the ExpoWeibull Distribution:

$$f(x) = \mu \left(\frac{\beta}{\alpha} \right) \left(\frac{x}{\alpha} \right)^{\beta - 1} \left[1 - e^{-\left(\frac{x}{\alpha} \right)^\beta} \right]^{\mu - 1} e^{-\left(\frac{x}{\alpha} \right)^\beta}$$

Parameters

- **x** (*numpy array or scalar*) – The values at which the function will be calculated
- **alpha** (*numpy array or scalar*) – scale parameter for the ExpoWeibull distribution
- **beta** (*numpy array or scalar*) – shape parameter for the ExpoWeibull distribution
- **mu** (*numpy array or scalar*) – shape parameter for the ExpoWeibull distribution

Returns **df** – The value(s) of the density function at x.

Return type scalar or numpy array

Examples

```
>>> import numpy as np
>>> from surpyval import ExpoWeibull
>>> x = np.array([1, 2, 3, 4, 5])
>>> ExpoWeibull.df(x, 3, 4, 1.2)
array([0.02427515, 0.27589838, 0.53701385, 0.15943643, 0.00330058])
```

ff (*x*, *alpha*, *beta*, *mu*)

Failure (CDF or unreliability) function for the ExpoWeibull Distribution:

$$F(x) = \left[1 - e^{-\left(\frac{x}{\alpha} \right)^\beta} \right]^{\mu}$$

Parameters

- **x** (*numpy array or scalar*) – The values at which the function will be calculated
- **alpha** (*numpy array or scalar*) – scale parameter for the ExpoWeibull distribution
- **beta** (*numpy array or scalar*) – shape parameter for the ExpoWeibull distribution
- **mu** (*numpy array or scalar*) – shape parameter for the ExpoWeibull distribution

Returns **sf** – The value(s) of the failure function at x.

Return type scalar or numpy array

Examples

```
>>> import numpy as np
>>> from surpyval import ExpoWeibull
>>> x = np.array([1, 2, 3, 4, 5])
>>> ExpoWeibull.ff(x, 3, 4, 1.2)
array([0.00508867, 0.1270975 , 0.57671321, 0.94933251, 0.99946528])
```

fit(*x=None*, *c=None*, *n=None*, *t=None*, *how='MLE'*, *offset=False*, *zi=False*, *lfp=False*, *tl=None*, *tr=None*, *xl=None*, *xr=None*, *fixed=None*, *heuristic='Turnbull'*, *init=[]*, *rr='y'*, *on_d_is_0=False*, *turnbull_estimator='Fleming-Harrington'*)

The central feature to SurPyval's capability. This function aimed to have an API to mimic the simplicity of the scipy API. That is, to use a simple `fit()` call, with as many or as few parameters as is needed.

Parameters

- **x** (*array like, optional*) – Array of observations of the random variables. If x is `None`, xl and xr must be provided.
- **c** (*array like, optional*) – Array of censoring flag. -1 is left censored, 0 is observed, 1 is right censored, and 2 is intervally censored. If not provided will assume all values are observed.
- **n** (*array like, optional*) – Array of counts for each x. If data is proivded as counts, then this can be provided. If `None` will assume each observation is 1.
- **t** (*2D-array like, optional*) – 2D array like of the left and right values at which the respective observation was truncated. If not provided it assumes that no truncation occurs.
- **how** (*{'MLE', 'MPP', 'MOM', 'MSE', 'MPS'}, optional*) – Method to estimate parameters, these are:
  - MLE : Maximum Likelihood Estimation
  - MPP : Method of Probability Plotting
  - MOM : Method of Moments
  - MSE : Mean Square Error
  - MPS : Maximum Product Spacing
- **offset** (*boolean, optional*) – If `True` finds the shifted distribution. If not provided assumes not a shifted distribution. Only works with distributions that are supported on the half-real line.
- **tl** (*array like or scalar, optional*) – Values of left truncation for observations. If it is a scalar value assumes each observation is left truncated at the value. If an array, it is the respective 'late entry' of the observation
- **tr** (*array like or scalar, optional*) – Values of right truncation for observations. If it is a scalar value assumes each observation is right truncated at the value. If an array, it is the respective right truncation value for each observation
- **xl** (*array like, optional*) – Array like of the left array for 2-dimensional input of x. This is useful for data that is all intervally censored. Must be used with the `xr` input.
- **xr** (*array like, optional*) – Array like of the right array for 2-dimensional input of x. This is useful for data that is all intervally censored. Must be used with the `xl` input.
- **fixed** (*dict, optional*) – Dictionary of parameters and their values to fix. Fixes parameter by name.
- **heuristic** (*{'"Blom", "Median", "ECDF", "Modal", "Midpoint", "Mean", "Weibull", "Benard", "Beard", "Hazen", "Gringorten", "None", "Tukey", "DPW", "Fleming-Harrington", "Kaplan-Meier", "Nelson-Aalen", "Filliben", "Larsen", "Turnbull"}*) – Plotting method to use, if using the probability plotting, MPP, method.

- **init** (*array like, optional*) – initial guess of parameters. Useful if method is failing.
- **rr** (*('y', 'x')*) – The dimension on which to minimise the spacing between the line and the observation. If 'y' the mean square error between the line and vertical distance to each point is minimised. If 'x' the mean square error between the line and horizontal distance to each point is minimised.
- **on_d_is_0** (*boolean, optional*) – For the case when using MPP and the highest value is right censored, you can choosed to include this value into the regression analysis or not. That is, if `False`, all values where there are 0 deaths are excluded from the regression. If `True` all values regardless of whether there is a death or not are included in the regression.
- **turnbull_estimator** (*('Nelson-Aalen', 'Kaplan-Meier', or 'Fleming-Harrington'), str, optional*) – If using the Turnbull heuristic, you can elect to use either the KM, NA, or FH estimator with the Turnbull estimates of r, and d. Defaults to FH.

Returns **model** – A parametric model with the fitted parameters and methods for all functions of the distribution using the fitted parameters.

Return type *Parametric*

Examples

```
>>> from surpyval import Weibull
>>> import numpy as np
>>> x = Weibull.random(100, 10, 4)
>>> model = Weibull.fit(x)
>>> print(model)
Parametric SurPyval Model
=========================
Distribution        : Weibull
Fitted by           : MLE
Parameters          :
     alpha: 10.551521182640098
      beta: 3.792549834495306
>>> Weibull.fit(x, how='MPS', fixed={'alpha' : 10})
Parametric SurPyval Model
=========================
Distribution        : Weibull
Fitted by           : MPS
Parameters          :
     alpha: 10.0
      beta: 3.4314657446866836
>>> Weibull.fit(xl=x-1, xr=x+1, how='MPP')
Parametric SurPyval Model
=========================
Distribution        : Weibull
Fitted by           : MPP
Parameters          :
     alpha: 9.943092756713078
      beta: 8.613016934518258
>>> c = np.zeros_like(x)
>>> c[x > 13] = 1
>>> x[x > 13] = 13
```

(continues on next page)

(continued from previous page)

```
>>> c = c[x > 6]
>>> x = x[x > 6]
>>> Weibull.fit(x=x, c=c, tl=6)
Parametric SurPyval Model
=========================
Distribution        : Weibull
Fitted by           : MLE
Parameters          :
     alpha: 10.363725328793413
      beta: 4.9886821457305865
```

fit_from_df (*df*, *x=None*, *c=None*, *n=None*, *xl=None*, *xr=None*, *tl=None*, *tr=None*, ***fit_options*)

The central feature to SurPyval's capability. This function aimed to have an API to mimic the simplicity of the scipy API. That is, to use a simple `fit()` call, with as many or as few parameters as is needed.

Parameters

- **df** (*DataFrame*) – DataFrame of data to be used to create surpyval model
- **x** (*string, optional*) – column name for the column in df containing the variable data. If not provided must provide both xl and xr
- **c** (*string, optional*) – column name for the column in df containing the censor flag of x. If not provided assumes all values of x are observed.
- **n** (*string, optional*) – column name in for the column in df containing the counts of x. If not provided assumes each x is one observation.
- **tl** (*string or scalar, optional*) – If string, column name in for the column in df containing the left truncation data. If scalar assumes each x is left truncated by that value. If not provided assumes x is not left truncated.
- **tr** (*string or scalar, optional*) – If string, column name in for the column in df containing the right truncation data. If scalar assumes each x is right truncated by that value. If not provided assumes x is not right truncated.
- **xl** (*string, optional*) – column name for the column in df containing the left interval for interval censored data. If left interval is -Inf, assumes left censored. If xl[i] == xr[i] assumes observed. Cannot be provided with x, must be provided with xr.
- **xr** (*string, optional*) – column name for the column in df containing the right interval for interval censored data. If right interval is Inf, assumes right censored. If xl[i] == xr[i] assumes observed. Cannot be provided with x, must be provided with xl.
- **fit_options** (*dict, optional*) – dictionary of fit options that will be passed to the `fit` method, see that method for options.

Returns **model** – A parametric model with the fitted parameters and methods for all functions of the distribution using the fitted parameters.

Return type *Parametric*

Examples

```
>>> import surpyval as surv
>>> df = surv.datasets.BoforsSteel.df
>>> model = surv.Weibull.fit_from_df(df, x='x', n='n', offset=True)
>>> print(model)
```

(continues on next page)

(continued from previous page)

```
Parametric SurPyval Model
=========================
Distribution        : Weibull
Fitted by           : MLE
Offset (gamma)      : 39.76562962867477
Parameters          :
     alpha: 7.141925216146524
      beta: 2.6204524040137844
```

from_params (*params*, *gamma=None*, *p=None*, *f0=None*)
Creating a SurPyval Parametric class with provided parameters.

Parameters

- **params** (*array like*) – array of the parameters of the distribution.
- **gamma** (*scalar, optional*) – offset value for the distribution. If not provided will fit a regular, unshifted/not offset, distribution.

Returns model – A parametric model with the fitted parameters and methods for all functions of the distribution using the fitted parameters.

Return type *Parametric*

Examples

```
>>> from surpyval import Weibull
>>> model = Weibull.from_params([10, 4])
>>> print(model)
Parametric SurPyval Model
=========================
Distribution        : Weibull
Fitted by           : given parameters
Parameters          :
     alpha: 10
      beta: 4
>>> model = Weibull.from_params([10, 4], gamma=2)
>>> print(model)
Parametric SurPyval Model
=========================
Distribution        : Weibull
Fitted by           : given parameters
Offset (gamma)      : 2
Parameters          :
     alpha: 10
      beta: 4
```

hf (*x*, *alpha*, *beta*, *mu*)
Instantaneous hazard rate for the ExpoWeibull Distribution:

$$h(x) = \frac{f(x)}{R(x)}$$

Parameters

- **x** (*numpy array or scalar*) – The values at which the function will be calculated
- **alpha** (*numpy array or scalar*) – scale parameter for the ExpoWeibull distribution

- **beta** (*numpy array or scalar*) – shape parameter for the ExpoWeibull distribution
- **mu** (*numpy array or scalar*) – shape parameter for the ExpoWeibull distribution

Returns **hf** – The value(s) of the instantaneous hazard rate at x.

Return type scalar or numpy array

Examples

```
>>> import numpy as np
>>> from surpyval import ExpoWeibull
>>> x = np.array([1, 2, 3, 4, 5])
>>> ExpoWeibull.hf(x, 3, 4, 1.2)
array([0.02439931, 0.3160701 , 1.26867613, 3.14672068, 6.17256436])
```

qf (*p*, *alpha*, *beta*, *mu*)

Instantaneous hazard rate for the ExpoWeibull Distribution:

$$q(p) =$$

Parameters

- **p** (*numpy array or scalar*) – The percentiles at which the quantile will be calculated
- **alpha** (*numpy array or scalar*) – scale parameter for the ExpoWeibull distribution
- **beta** (*numpy array or scalar*) – shape parameter for the ExpoWeibull distribution
- **mu** (*numpy array or scalar*) – shape parameter for the ExpoWeibull distribution

Returns **Q** – The quantiles for the Weibull distribution at each value p

Return type scalar or numpy array

Examples

```
>>> import numpy as np
>>> from surpyval import ExpoWeibull
>>> p = np.array([.1, .2, .3, .4, .5])
>>> ExpoWeibull.qf(p, 3, 4, 1.2)
array([1.89361341, 2.2261045 , 2.46627621, 2.66992747, 2.85807988])
```

sf (*x*, *alpha*, *beta*, *mu*)

Survival (or reliability) function for the ExpoWeibull Distribution:

$$R(x) = 1 - \left[1 - e^{-\left(\frac{x}{\alpha}\right)^{\beta}}\right]^{\mu}$$

Parameters

- **x** (*numpy array or scalar*) – The values at which the function will be calculated
- **alpha** (*numpy array or scalar*) – scale parameter for the ExpoWeibull distribution

- **beta** (*numpy array or scalar*) – shape parameter for the ExpoWeibull distribution
- **mu** (*numpy array or scalar*) – shape parameter for the ExpoWeibull distribution

Returns **sf** – The value(s) of the reliability function at x.

Return type scalar or numpy array

Examples

```
>>> import numpy as np
>>> from surpyval import ExpoWeibull
>>> x = np.array([1, 2, 3, 4, 5])
>>> ExpoWeibull.sf(x, 3, 4, 1.2)
array([9.94911330e-01, 8.72902497e-01, 4.23286791e-01, 5.06674866e-02,
       5.34717283e-04])
```

Parametric Class

class surpyval.parametric.parametric.**Parametric**(*dist*, *method*, *data*, *offset*, *lfp*, *zi*)
Bases: `object`

Result of `.fit()` or `.from_params()` method for every parametric surpyval distribution.

Instances of this class are very useful when a user needs the other functions of a distribution for plotting, optimizations, monte carlo analysis and numeric integration.

Hf(*x*)
The cumulative hazard function for a distribution using the parameters found in the `.params` attribute.

Parameters **x** (*array like or scalar*) – The values of the random variables at which the cumulative hazard function will be calculated

Returns **Hf** – The scalar value of the cumulative hazard function of the distribution if a scalar was passed. If an array like object was passed then a numpy array is returned with the value of the cumulative hazard function at each corresponding value in the input array.

Return type scalar or numpy array

Examples

```
>>> from surpyval import Weibull
>>> model = Weibull.from_params([10, 3])
>>> model.Hf(2)
0.008000000000000002
>>> model.Hf([1, 2, 3, 4, 5])
array([0.001, 0.008, 0.027, 0.064, 0.125])
```

aic()
The the Aikake Information Criterion (AIC) for the model, if it was fit with the `fit()` method. Not available if fit with the `from_params()` method.

Parameters **None** –

Returns **aic** – The AIC of the model

Return type float

Examples

```
>>> from surpyval import Weibull
>>> import numpy as np
>>> np.random.seed(1)
>>> x = Weibull.random(100, 10, 3)
>>> model = Weibull.fit(x)
>>> model.aic()
529.0537128477147
```

aic_c()

The the Corrected Aikake Information Criterion (AIC) for the model, if it was fit with the `fit()` method. Not available if fit with the `from_params()` method.

Parameters **`None`** –

Returns **aic_c** – The Corrected AIC of the model

Return type float

Examples

```
>>> from surpyval import Weibull
>>> import numpy as np
>>> np.random.seed(1)
>>> x = Weibull.random(100, 10, 3)
>>> model = Weibull.fit(x)
>>> model.aic()
529.1774241879209
```

bic()

The the Bayesian Information Criterion (BIC) for the model, if it was fit with the `fit()` method. Not available if fit with the `from_params()` method.

Parameters **`None`** –

Returns **bic** – The BIC of the model

Return type float

Examples

```
>>> from surpyval import Weibull
>>> import numpy as np
>>> np.random.seed(1)
>>> x = Weibull.random(100, 10, 3)
>>> model = Weibull.fit(x)
>>> model.bic()
534.2640532196908
```

Bayesian Information Criterion for Censored Survival Models.

cb(*t*, *on='R'*, *alpha_ci=0.05*, *bound='two-sided'*)

Confidence bounds of the `on` function at the `alpa_ci` level of significance. Can be the upper, lower, or two-sided confidence by changing value of `bound`.

Parameters

- **x** (*array like or scalar*) – The values of the random variables at which the confidence bounds will be calculated
- **on** (*('sf', 'ff', 'Hf'), optional*) – The function on which the confidence bound will be calculated.
- **bound** (*('two-sided', 'upper', 'lower'), str, optional*) – Compute either the two-sided, upper or lower confidence bound(s). Defaults to two-sided.
- **alpha_ci** (*scalar, optional*) – The level of significance at which the bound will be computed.

Returns **cb** – The value(s) of the upper, lower, or both confidence bound(s) of the selected function at x

Return type scalar or numpy array

cs (*x*, *X*)

The conditional survival of the model.

Parameters

- **x** (*array like or scalar*) – The values at which conditional survival is to be calculated.
- **X** (*array like or scalar*) – The value(s) at which it is known the item has survived

Returns **cs** – The conditional survival probability.

Return type array

Examples

```
>>> from surpyval import Weibull
>>> model = Weibull.from_params([10, 3])
>>> model.cs(11, 10)
0.00025840046151723767
```

df (*x*)

The density function for a distribution using the parameters found in the `.params` attribute.

Parameters **x** (*array like or scalar*) – The values of the random variables at which the density function will be calculated

Returns **df** – The scalar value of the density function of the distribution if a scalar was passed. If an array like object was passed then a numpy array is returned with the value of the density function at each corresponding value in the input array.

Return type scalar or numpy array

Examples

```
>>> from surpyval import Weibull
>>> model = Weibull.from_params([10, 3])
>>> model.df(2)
0.01190438297804473
>>> model.df([1, 2, 3, 4, 5])
array([0.002997  , 0.01190438, 0.02628075, 0.04502424, 0.06618727])
```

entropy()

A method to draw random samples from the distributions using the parameters found in the `.params` attribute.

Parameters **None** –

Returns **entropy** – Returns entropy of the distribution

Return type float

References

ENTROPY REF

Examples

```
>>> from surpyval import Normal
>>> model = Normal.from_params([10, 3])
>>> model.entropy()
2.588783247593625
```

ff(*x*)

The cumulative distribution function, or failure function, for a distribution using the parameters found in the `.params` attribute.

Parameters **x** (*array like or scalar*) – The values of the random variables at which the failure function (CDF) will be calculated

Returns **ff** – The scalar value of the CDF of the distribution if a scalar was passed. If an array like object was passed then a numpy array is returned with the value of the CDF at each corresponding value in the input array.

Return type scalar or numpy array

Examples

```
>>> from surpyval import Weibull
>>> model = Weibull.from_params([10, 3])
>>> model.ff(2)
0.007968085162939342
>>> model.ff([1, 2, 3, 4, 5])
array([0.0009995 , 0.00796809, 0.02663876, 0.061995  , 0.1175031 ])
```

get_plot_data(*heuristic='Turnbull'*, *alpha_ci=0.05*)

A method to gather plot data

Parameters

- **heuristic** (*{'Blom', 'Median', 'ECDF', 'Modal', 'Midpoint', 'Mean', 'Weibull', 'Benard', 'Beard', 'Hazen', 'Gringorten', 'None', 'Tukey', 'DPW', 'Fleming-Harrington', 'Kaplan-Meier', 'Nelson-Aalen', 'Filliben', 'Larsen', 'Turnbull'}, optional*) – The method that the plotting point on the problility plot will be calculated.
- **alpha_ci** (*float, optional*) – The confidence with which the confidence bounds, if able, will be calculated. Defaults to 0.95.

Returns **data** – Returns dictionary containing the data needed to do a plot.

Return type dict

Examples

```
>>> from surpyval import Weibull
>>> x = Weibull.random(100, 10, 3)
>>> model = Weibull.fit(x)
>>> data = model.get_plot_data()
```

hf(*x*)

The instantaneous hazard function for a distribution using the parameters found in the `.params` attribute.

Parameters **x** (*array like or scalar*) – The values of the random variables at which the instantaneous hazard function will be calculated

Returns **hf** – The scalar value of the instantaneous hazard function of the distribution if a scalar was passed. If an array like object was passed then a numpy array is returned with the value of the instantaneous hazard function at each corresponding value in the input array.

Return type scalar or numpy array

Examples

```
>>> from surpyval import Weibull
>>> model = Weibull.from_params([10, 3])
>>> model.hf(2)
0.012000000000000002
>>> model.hf([1, 2, 3, 4, 5])
array([0.003, 0.012, 0.027, 0.048, 0.075])
```

mean()

A method to draw random samples from the distributions using the parameters found in the `.params` attribute.

Parameters **None** –

Returns **mean** – Returns the mean of the distribution.

Return type float

Examples

```
>>> from surpyval import Weibull
>>> model = Weibull.from_params([10, 3])
>>> model.mean()
8.929795115692489
```

moment(*n*)

A method to draw random samples from the distributions using the parameters found in the `.params` attribute.

Parameters **n** (*integer*) – The degree of the moment to be computed

Returns **moment[n]** – Returns the n-th moment of the distribution

Return type float

References

INSERT WIKIPEDIA HERE

Examples

```
>>> from surpyval import Normal
>>> model = Normal.from_params([10, 3])
>>> model.moment(1)
10.0
>>> model.moment(5)
202150.0
```

neg_ll()

The the negative log-likelihood for the model, if it was fit with the `fit()` method. Not available if fit with the `from_params()` method.

Parameters **None** –

Returns **neg_ll** – The negative log-likelihood of the model

Return type float

Examples

```
>>> from surpyval import Weibull
>>> import numpy as np
>>> np.random.seed(1)
>>> x = Weibull.random(100, 10, 3)
>>> model = Weibull.fit(x)
>>> model.neg_ll()
262.52685642385734
```

plot(*heuristic='Turnbull'*, *plot_bounds=True*, *alpha_ci=0.05*)

A method to do a probability plot

Parameters

- **heuristic** (*{'Blom', 'Median', 'ECDF', 'Modal', 'Midpoint', 'Mean', 'Weibull', 'Benard', 'Beard', 'Hazen', 'Gringorten', 'None', 'Tukey', 'DPW', 'Fleming-Harrington', 'Kaplan-Meier', 'Nelson-Aalen', 'Filliben', 'Larsen', 'Turnbull'}, optional*) – The method that the plotting point on the probablility plot will be calculated.
- **plot_bounds** (*Boolean, optional*) – A Boolean value to indicate whehter you want the probability bounds to be calculated.
- **alpha_ci** (*float, optional*) – The confidence with which the confidence bounds, if able, will be calculated. Defaults to 0.95.

Returns **plot** – list of a matplotlib plot object

Return type list

Examples

```
>>> from surpyval import Weibull
>>> x = Weibull.random(100, 10, 3)
>>> model = Weibull.fit(x)
>>> model.plot()
```

qf (*p*)
The quantile function for a distribution using the parameters found in the `.params` attribute.

Parameters **p** (*array like or scalar*) – The values, which must be between 0 and 1, at which the the quantile will be calculated

Returns **qf** – The scalar value of the quantile of the distribution if a scalar was passed. If an array like object was passed then a numpy array is returned with the value of the quantile at each corresponding value in the input array.

Return type scalar or numpy array

Examples

```
>>> from surpyval import Weibull
>>> model = Weibull.from_params([10, 3])
>>> model.qf(0.2)
6.06542793124108
>>> model.qf([.1, .2, .3, .4, .5])
array([4.72308719, 6.06542793, 7.09181722, 7.99387877, 8.84997045])
```

random (*size*)
A method to draw random samples from the distributions using the parameters found in the `.params` attribute.

Parameters **size** (*int*) – The number of random samples to be drawn from the distribution.

Returns **random** – Returns a numpy array of size `size` with random values drawn from the distribution.

Return type numpy array

Examples

```
>>> from surpyval import Weibull
>>> model = Weibull.from_params([10, 3])
>>> np.random.seed(1)
>>> model.random(1)
array([8.14127103])
>>> model.random(10)
array([10.84103403,  0.48542084,  7.11387062,  5.41420125,  4.59286657,
        5.90703589,  7.5124326 ,  7.96575225,  9.18134126,  8.16000438])
```

sf (*x*)
Surival (or Reliability) function for a distribution using the parameters found in the `.params` attribute.

Parameters **x** (*array like or scalar*) – The values of the random variables at which the survival function will be calculated

Returns **sf** – The scalar value of the survival function of the distribution if a scalar was passed. If an array like object was passed then a numpy array is returned with the value of the survival function at each corresponding value in the input array.

Return type scalar or numpy array

Examples

```
>>> from surpyval import Weibull
>>> model = Weibull.from_params([10, 3])
>>> model.sf(2)
0.9920319148370607
>>> model.sf([1, 2, 3, 4, 5])
array([0.9990005 , 0.99203191, 0.97336124, 0.938005  , 0.8824969 ])
```

Parametric Fitter

class `surpyval.parametric.parametric_fitter.`**`ParametricFitter`**

Bases: `object`

`fit`(*x=None*, *c=None*, *n=None*, *t=None*, *how='MLE'*, *offset=False*, *zi=False*, *lfp=False*, *tl=None*, *tr=None*, *xl=None*, *xr=None*, *fixed=None*, *heuristic='Turnbull'*, *init=[]*, *rr='y'*, *on_d_is_0=False*, *turnbull_estimator='Fleming-Harrington'*)

The central feature to SurPyval's capability. This function aimed to have an API to mimic the simplicity of the scipy API. That is, to use a simple `fit()` call, with as many or as few parameters as is needed.

Parameters

- **x** (*array like, optional*) – Array of observations of the random variables. If x is `None`, xl and xr must be provided.
- **c** (*array like, optional*) – Array of censoring flag. -1 is left censored, 0 is observed, 1 is right censored, and 2 is intervally censored. If not provided will assume all values are observed.
- **n** (*array like, optional*) – Array of counts for each x. If data is proivded as counts, then this can be provided. If `None` will assume each observation is 1.
- **t** (*2D-array like, optional*) – 2D array like of the left and right values at which the respective observation was truncated. If not provided it assumes that no truncation occurs.
- **how** (*{'MLE', 'MPP', 'MOM', 'MSE', 'MPS'}, optional*) – Method to estimate parameters, these are:
  - MLE : Maximum Likelihood Estimation
  - MPP : Method of Probability Plotting
  - MOM : Method of Moments
  - MSE : Mean Square Error
  - MPS : Maximum Product Spacing
- **offset** (*boolean, optional*) – If `True` finds the shifted distribution. If not provided assumes not a shifted distribution. Only works with distributions that are supported on the half-real line.

- **tl** (*array like or scalar, optional*) – Values of left truncation for observations. If it is a scalar value assumes each observation is left truncated at the value. If an array, it is the respective 'late entry' of the observation
- **tr** (*array like or scalar, optional*) – Values of right truncation for observations. If it is a scalar value assumes each observation is right truncated at the value. If an array, it is the respective right truncation value for each observation
- **xl** (*array like, optional*) – Array like of the left array for 2-dimensional input of x. This is useful for data that is all intervally censored. Must be used with the `xr` input.
- **xr** (*array like, optional*) – Array like of the right array for 2-dimensional input of x. This is useful for data that is all intervally censored. Must be used with the `xl` input.
- **fixed** (*dict, optional*) – Dictionary of parameters and their values to fix. Fixes parameter by name.
- **heuristic** (*{'"Blom", "Median", "ECDF", "Modal", "Midpoint", "Mean", "Weibull", "Benard", "Beard", "Hazen", "Gringorten", "None", "Tukey", "DPW", "Fleming-Harrington", "Kaplan-Meier", "Nelson-Aalen", "Filliben", "Larsen", "Turnbull"}*) – Plotting method to use, if using the probability plotting, MPP, method.
- **init** (*array like, optional*) – initial guess of parameters. Useful if method is failing.
- **rr** (*('y', 'x')*) – The dimension on which to minimise the spacing between the line and the observation. If 'y' the mean square error between the line and vertical distance to each point is minimised. If 'x' the mean square error between the line and horizontal distance to each point is minimised.
- **on_d_is_0** (*boolean, optional*) – For the case when using MPP and the highest value is right censored, you can choosed to include this value into the regression analysis or not. That is, if `False`, all values where there are 0 deaths are excluded from the regression. If `True` all values regardless of whether there is a death or not are included in the regression.
- **turnbull_estimator** (*('Nelson-Aalen', 'Kaplan-Meier', or 'Fleming-Harrington'), str, optional*) – If using the Turnbull heuristic, you can elect to use either the KM, NA, or FH estimator with the Turnbull estimates of r, and d. Defaults to FH.

Returns **model** – A parametric model with the fitted parameters and methods for all functions of the distribution using the fitted parameters.

Return type *Parametric*

Examples

```
>>> from surpyval import Weibull
>>> import numpy as np
>>> x = Weibull.random(100, 10, 4)
>>> model = Weibull.fit(x)
>>> print(model)
Parametric SurPyval Model
=========================
Distribution        : Weibull
```

(continues on next page)

(continued from previous page)

```
Fitted by            : MLE
Parameters           :
     alpha: 10.551521182640098
      beta: 3.792549834495306
>>> Weibull.fit(x, how='MPS', fixed={'alpha' : 10})
Parametric SurPyval Model
=========================
Distribution         : Weibull
Fitted by            : MPS
Parameters           :
     alpha: 10.0
      beta: 3.4314657446866836
>>> Weibull.fit(xl=x-1, xr=x+1, how='MPP')
Parametric SurPyval Model
=========================
Distribution         : Weibull
Fitted by            : MPP
Parameters           :
     alpha: 9.943092756713078
      beta: 8.613016934518258
>>> c = np.zeros_like(x)
>>> c[x > 13] = 1
>>> x[x > 13] = 13
>>> c = c[x > 6]
>>> x = x[x > 6]
>>> Weibull.fit(x=x, c=c, tl=6)
Parametric SurPyval Model
=========================
Distribution         : Weibull
Fitted by            : MLE
Parameters           :
     alpha: 10.363725328793413
      beta: 4.9886821457305865
```

fit_from_df (*df*, *x=None*, *c=None*, *n=None*, *xl=None*, *xr=None*, *tl=None*, *tr=None*, ***fit_options*)

The central feature to SurPyval's capability. This function aimed to have an API to mimic the simplicity of the scipy API. That is, to use a simple `fit()` call, with as many or as few parameters as is needed.

Parameters

- **df** (*DataFrame*) – DataFrame of data to be used to create surpyval model
- **x** (*string, optional*) – column name for the column in df containing the variable data. If not provided must provide both xl and xr
- **c** (*string, optional*) – column name for the column in df containing the censor flag of x. If not provided assumes all values of x are observed.
- **n** (*string, optional*) – column name in for the column in df containing the counts of x. If not provided assumes each x is one observation.
- **tl** (*string or scalar, optional*) – If string, column name in for the column in df containing the left truncation data. If scalar assumes each x is left truncated by that value. If not provided assumes x is not left truncated.
- **tr** (*string or scalar, optional*) – If string, column name in for the column in df containing the right truncation data. If scalar assumes each x is right truncated by that value. If not provided assumes x is not right truncated.

- **xl** (*string, optional*) – column name for the column in df containing the left interval for interval censored data. If left interval is -Inf, assumes left censored. If xl[i] == xr[i] assumes observed. Cannot be provided with x, must be provided with xr.
- **xr** (*string, optional*) – column name for the column in df containing the right interval for interval censored data. If right interval is Inf, assumes right censored. If xl[i] == xr[i] assumes observed. Cannot be provided with x, must be provided with xl.
- **fit_options** (*dict, optional*) – dictionary of fit options that will be passed to the `fit` method, see that method for options.

Returns model – A parametric model with the fitted parameters and methods for all functions of the distribution using the fitted parameters.

Return type *Parametric*

Examples

```
>>> import surpyval as surv
>>> df = surv.datasets.BoforsSteel.df
>>> model = surv.Weibull.fit_from_df(df, x='x', n='n', offset=True)
>>> print(model)
Parametric SurPyval Model
=========================
Distribution        : Weibull
Fitted by           : MLE
Offset (gamma)      : 39.76562962867477
Parameters          :
     alpha: 7.141925216146524
      beta: 2.6204524040137844
```

from_params (*params*, *gamma=None*, *p=None*, *f0=None*)
Creating a SurPyval Parametric class with provided parameters.

Parameters

- **params** (*array like*) – array of the parameters of the distribution.
- **gamma** (*scalar, optional*) – offset value for the distribution. If not provided will fit a regular, unshifted/not offset, distribution.

Returns model – A parametric model with the fitted parameters and methods for all functions of the distribution using the fitted parameters.

Return type *Parametric*

Examples

```
>>> from surpyval import Weibull
>>> model = Weibull.from_params([10, 4])
>>> print(model)
Parametric SurPyval Model
=========================
Distribution        : Weibull
Fitted by           : given parameters
Parameters          :
     alpha: 10
```

(continues on next page)

(continued from previous page)

```
      beta: 4
>>> model = Weibull.from_params([10, 4], gamma=2)
>>> print(model)
Parametric SurPyval Model
=========================
Distribution        : Weibull
Fitted by           : given parameters
Offset (gamma)      : 2
Parameters          :
     alpha: 10
      beta: 4
```

Parametric Mixture Model

class surpyval.parametric.mixture_model.**MixtureModel**(*x*, *dist=<surpyval.parametric.weibull.Weibull_ object>*, ***kwargs*)

Bases: `object`

Generalised from algorithm found here https://www.sciencedirect.com/science/article/pii/S0307904X12002545

1.13.3 Datasets

class surpyval.datasets.**Bearing_**

Bases: `object`

class surpyval.datasets.**BoforsSteel_**

Bases: `object`

A Class with a Pandas DataFrame containing the data of the tensile strenght of Bofors Steel from the Weibull paper[1].

df

A Pandas DataFrame containing the data of the tensile strenght of Bofors Steel from the Weibull paper.

Type DataFrame

Examples

```
>>> from surpyval import BoforsSteel
>>> df = BoforsSteel.df
>>> df.head(5)
```

	x	n
0	40.800	10
1	42.075	23
2	43.350	48
3	44.625	80
4	45.900	63

[1] Weibull, W., A statistical distribution function of wide applicability, Journal of applied mechanics, Vol. 18, No. 3, pp 293-297 (1951).

References

class surpyval.datasets.**Boston_**

Bases: `object`

1.14 Changelog

1.14.1 v0.10.0 (planned)

- General ALT fitter full release
- General PH fitter full release
- Formulas
- Add more than Breslow to the CoxPH methods.
- Parameter confidence bound
- Document the rationale behind using Fleming-Harrington as the default.
- Docs on how to integrate with Pandas
- Docs for CoxPH
- Docs for Accelerated Life fitters
- Create a `RegressionFitter` class. I keep copying code across the three fitters.
- Allow truncation with zi and lfp models.
- Allow truncation with regression

1.14.2 v0.9.0 (5 Aug 2021)

- Better initial estimates in the `_parameter_initialiser` for the lfp data (use max F from nonp estimate…)
- issue #13 - Better failures when insufficient data provided.
- issue #12 - Created `fsli_to_xcn` helper function.
- Fixed bug in confidence bounds implementation for offset distributions. CBs were not using the offset and were therefore way out. Now fixed.
- Created a `NonParametric.cb()` method to match `Parametric` API for confidence bounds.
- Cleaned up NonParametric code (removed some technical debt and duplicated code).
- Changed the `__repr__` function in `NonParametric` to be aligned to `Parametric`
- Updated the docstring for `fit()` for `NonParametric`
- Fixed bug in `NonParametric` that required the `x` input to be in order for the functions (e.g. `df` etc.).
- `CoxPH` released.
- General AL fitter in beta
- General PH fitter in beta
- Created `Linear`, `Power`, `InversePower`, `Exponential`, `InverseExponential`, `Eyring`, `InverseEyring`, `DualPower`, `PowerExponential`, `DualExponential` life models.

- Created `GeneralLogLinear` life model for variable stress count input.
- For each combination of a SurPyval distribution and life model, there is an instance to use `fit()`. For example there are `WeibullDualExponential`, `LogNormalPower`, `ExponentialExponential` etc.
- **Docs Updates:**
  - **Add application examples to docs:**
    - Reliability Engineering
    - Actuary / Demography
    - Social Science/Criminology
    - Boston Housing
    - Medical science
    - Economics
    - Biology - Ware, J.H., Demets, D.L.: Reanalysis of some baboon descent data. Biometrics 459–463 (1976).

1.14.3 v0.8.0 (27 July 2021)

- Made backwards incompatible changes to `LFP` models, these are now created with the `lfp=True` keyword in the `fit()` method
- Created ability to fit zero-inflated models. Simply pass the `zi=True` option to the `fit()` method.
- Chanages to `utils.xcnt_handler` to ensure `x`, `xl`, and `xr` are handled consistently.
- changed the way __`repr`__ displays a Parametric object.
- Changed the default for plotting to be `Fleming-Harrington`. This was a result of seeing how poorly the `Nelson-Aalen` method fits zero inflated models. FH therefore offers the best performance of a Non-Parametric estimate at the low values of the survival function (as KM reaches 0 for fully observed data) and at high values (KM is good but NA is poor).
- Added a Fleming-Harrington method to the Turnbull class.
- Improved stability with dedicated `log_sf`, `log_ff`, and `log_df` functions. Less chance of overflows and therefore better convergence.
- Changed interpolation method of `NonParametric`. Allows for use of cubic interpolation
- Changed `from_params` to accept lfp and zi (or any combo)
- Changed `random()` in `Parametric` so that lfp or zi models can be simulated!
- Improved the way surpyval fails
- Substantial docs updates.

1.14.4 v0.7.0 (19 July 2021)

- Major changes to the confidence bounds for `Parametric` models. Now use the `cb()` method for every bound.
- Removed the `OffsetParametric` class and made `Parametric` class now work with (or without) an offset.
- Minor doc updates.

1.15 Support

If you need help with survival analysis, please ask a question on stats.stackexchange.

If you've searched the surpyval documentation for what you've been looking for and can't find it, please add as suggestion for a feature on GitHub. SurPyval is a growing tool. Or, if you need help with surpyval feel free to email Derryn at derryn.knife@gmail.com.

1.16 Contributing

If you want to contribute to SurPyval, please do! Please review the current open feature reqeusts to see if your desired feature is in the requests. If not, please raise a new one to notify the community. We can assign you feature for you to branch and develop.

SurPyval is in the process of complying with the PEP8 standard so please make all contributions as per that standard.

1.17 Installation

surpyval can be installed easily with the pip command:

```
$ pip install surpyval
```

CHAPTER 2

Indices and tables

- genindex
- modindex
- search

Bibliography

[Bagdonavicius] Bagdonavicius, V., & Nikulin, M. (2001). Accelerated life models: modeling and statistical analysis. CRC press.

[KM] Kaplan, E. L., & Meier, P. (1958). Nonparametric estimation from incomplete observations. Journal of the American statistical association, 53(282), 457-481.

[NA] Nelson, Wayne (1969). Hazard plotting for incomplete failure data. Journal of Quality Technology, 1(1), 27-52.

[FH] Fleming, Thomas R and Harrington, David P (1984). Nonparametric estimation of the survival distribution in censored data. Communications in Statistics-Theory and Methods, 13(20), 2469-2486.

[TB] Turnbull, Bruce W (1976). The empirical distribution function with arbitrarily grouped, censored and truncated data. Journal of the Royal Statistical Society: Series B (Methodological), 38(3), 290-295.

[TC] Tadeu Cristino, C., Żebrowski, P., & Wildemeersch, M. (2020). Assessing the time intervals between economic recessions. PloS one, 15(5), e0232615.

[Cole] Cole SR, Hudgens MG. Survival analysis in infectious disease research: describing events in time. AIDS. 2010;24(16):2423-31.

[Duwe] Duwe, G., Sanders, N. E., Rocque, M., & Fox, J. A. (2021). Forecasting the Severity of Mass Public Shootings in the United States. Journal of Quantitative Criminology, 1-39.

[Gavrilov] Gavrilov, L. A., Gavrilova, N. S., & Nosov, V. N. (1983). Human life span stopped increasing: why?. Gerontology, 29(3), 176-180.

[Meeker] William Q. Meeker (1987) Limited Failure Population Life Tests: Application to Integrated Circuit Reliability, Technometrics, 29(1), 51-65

Python Module Index

Index