Asset Management: Tutorial 1: likelihood maximization

Luuk Pentinga MSc
l.pentinga@rug.nl

Contents

Recap of the Weibull distribution

The Weibull distribution

In the previous lecture, we found some problems with the Kaplan-Meier estimator:

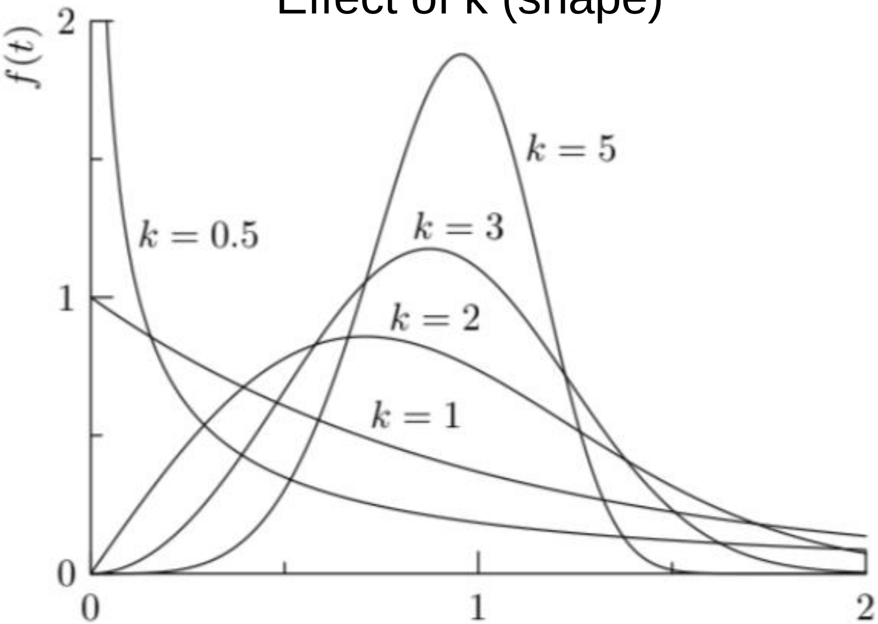
- Probabilities are given to discrete events.
- It is more difficult to do optimizations with.

So, we want to fit a continuous distribution to our data:
The Weibull distribution

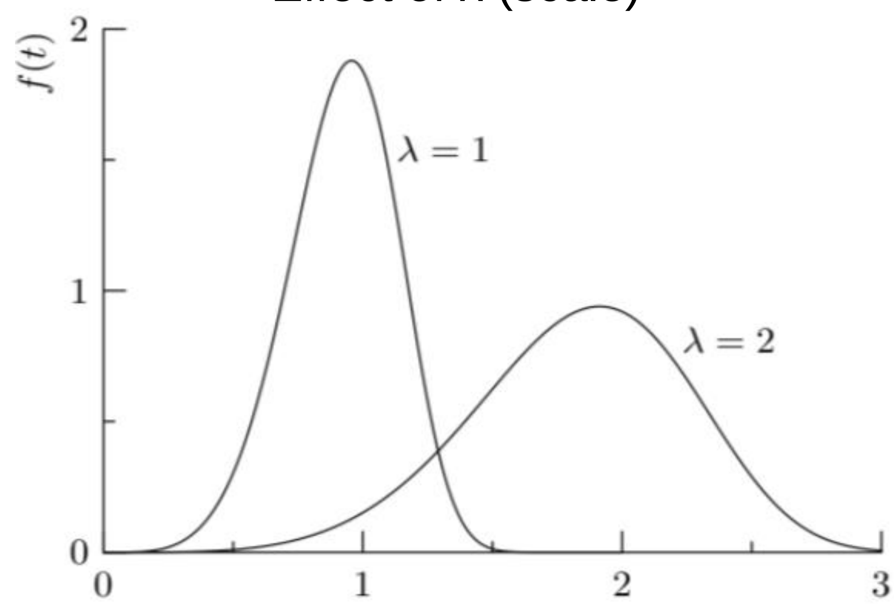
The Weibull distribution

The Weibull distribution is a **continuous** distribution, with two parameters, λ and κ.

Effect of κ (shape)

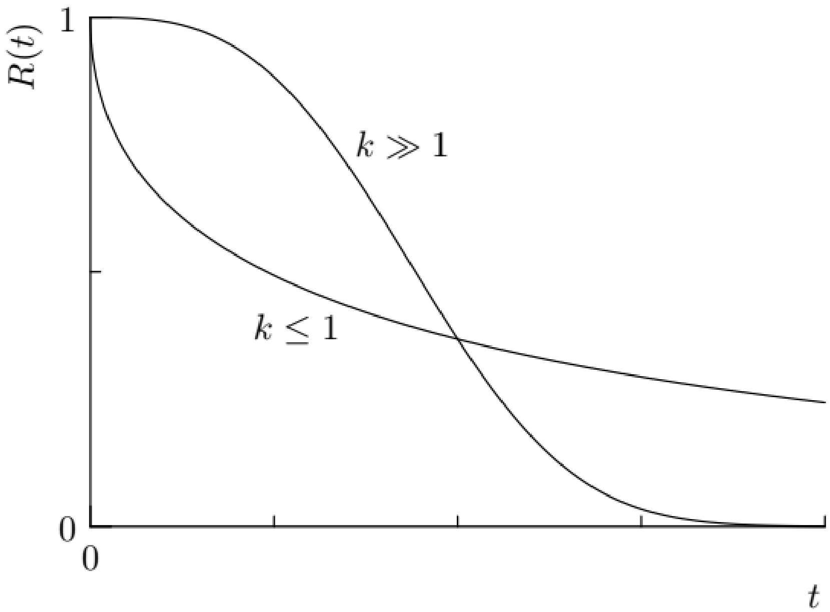


Effect of λ (scale)



The Weibull distribution

Dependent on κ, the shape of the distribution changes a lot.



κ > 1: Increasing failure rate
κ = 1: Constant failure rate
κ < 1: Decreasing failure rate

The Weibull distribution

Key functions:

F(t): Lifetime distribution function. This is the probability of a failure **before** t.

R(t): Reliability distribution function. The opposite of F(t), the probability of survival until t

f(t) : Density function. Gives the **likelihood** of a failure at time t. This is **not** equal to the probability of a failure at time t, but closely related.

The Weibull distribution

$$F(t;\lambda,k) = 1 - e^{-\left(\frac{t}{\lambda}\right)^k}, \quad t \geq 0.$$

$$R(t;\lambda,k) = 1 - F(t;\lambda,k) = e^{-\left(\frac{t}{\lambda}\right)^k}, \quad t \geq 0.$$

$$f(t;\lambda,k) = \frac{k}{\lambda}\left(\frac{t}{\lambda}\right)^{k-1} e^{-\left(\frac{t}{\lambda}\right)^k}, \quad t \geq 0.$$

Today: how to find the parameters that best explain our data.

Likelihood maximization

Maximum (log)-likelihood

Today, we will look at optimizing the parameters λ and κ.

Remember that we want to find the parameters with the highest **likelihood** for our data.

That means:

For **event durations**, the λ and κ with the highest value of f(t; λ, κ).

For **censored durations**, the λ and κ with the highest value of R(t; λ, κ).

Log-likelihood

› But, we take the log of the likelihood to prevent computational issues.

$$\begin{aligned} L(\lambda, k) &= f(4.8; \lambda, k) \times R(5.9; \lambda, k) \times f(2.5; \lambda, k) \times f(8.1; \lambda, k) \\ &\quad \times f(2.8; \lambda, k) \times R(3.4; \lambda, k) \times f(4.8; \lambda, k) \times f(9.7; \lambda, k). \end{aligned}$$

$$\begin{aligned} \log L(\lambda, k) &= \log(f(4.8; \lambda, k) \times R(5.9; \lambda, k) \times f(2.5; \lambda, k) \\ &\quad \times f(8.1; \lambda, k) \times f(2.8; \lambda, k) \times R(3.4; \lambda, k) \\ &\quad \times f(4.8; \lambda, k) \times f(9.7; \lambda, k)). \end{aligned}$$

$$\begin{aligned} \log L(\lambda, k) &= \log f(4.8; \lambda, k) + \log R(5.9; \lambda, k) + \log f(2.5; \lambda, k) \\ &\quad + \log f(8.1; \lambda, k) + \log f(2.8; \lambda, k) + \log R(3.4; \lambda, k) \\ &\quad + \log f(4.8; \lambda, k) + \log f(9.7; \lambda, k). \end{aligned}$$

How to - Maximization

We take the easiest approach, brute-force.

Idea: If we calculate the log-likelihood for every pair λ and κ, we can see which is the highest.

Problem: λ and κ can be anything above 0. Infinitely my options.

Solution: We set ranges for our parameters, and set a precision level.

How to - Maximization

Decide on ranges for the parameters:

Example:

λ in 1,2,3,…,20

κ in 0.1, 0.2, 0.3,…,2

This we can calculate!

It just requires trying 20 x 20 = 400 different options.
Hence, brute-force.

How to - Maximization

Step-by-step guide for programming:

1. Decide on ranges for your parameters.
2. Put all combinations of parameters into a table.
3. Get the log-likelihood of your first observation for each pair of parameters
4. Get the log-likelihood of your second observation for each pair of parameters

   …
5. Sum all log-likelihoods for each pair of parameters.
6. Find the best pair of parameters.

Example data

92 durations
73 failures
19 PM's

	Time	Event	Durations
0	10.18	no	10.18
1	16.70	no	6.52
2	29.26	no	12.56
3	32.18	no	2.92
4	35.09	no	2.91
..	...	...	...
87	580.79	no	3.73
88	585.90	yes	5.11
89	598.02	no	12.12
90	601.68	no	3.66
91	606.30	no	4.62

How to - Maximization

Step 1: Decide on ranges for your parameters.

How?

Mostly guessing… but you might use your Kaplan-Meier estimator as an indicator.

But your precision level is more important: Setting this to 0.01 instead of 0.1 will result in 10x as many calculations. (and will take ~10x as long)

Code

We select the following ranges:

λ in 1,2,3,…,20
κ in 0.1, 0.2, 0.3,…,2

In Python, we can use the np.linspace() function:

```
l_range = np.linspace(start=1, stop=20, num=20)
k_range = np.linspace(start=0.1, stop = 2, num=20)
```

How to - Maximization

Step 2: Put all combinations of parameters into a data frame.

We can not try λ and κ separately, but we need each possible pair.

Numpy has functions that can do this, but a *nested for-loop* works very well.

```
     lambda  kappa
0       1.0    0.1
1       1.0    0.2
2       1.0    0.3
3       1.0    0.4
4       1.0    0.5
..      ...    ...
395    20.0    1.6
396    20.0    1.7
397    20.0    1.8
398    20.0    1.9
399    20.0    2.0
```

Intermediate result

How to - Maximization

Step 3: Get the log-likelihood of your first observation for each pair of parameters.

Remember that the likelihood is calculated differently for censored durations and event durations!

In case of an event: $f(t;\lambda,k) = \frac{k}{\lambda}\left(\frac{t}{\lambda}\right)^{k-1} e^{-\left(\frac{t}{\lambda}\right)^k}, \quad t \geq 0.$

In case of a censored duration: $R(t;\lambda,k) = 1 - F(t;\lambda,k) = e^{-\left(\frac{t}{\lambda}\right)^k}, \quad t \geq 0.$

Code

To code this, we first need to add a column to our data frame.

Then, for each row in your data frame:

1. Take the value of λ and κ
2. Check if the event is censored
3. Insert the correct log-likelihood value

```
#we add a column to the output dataframe
column_name = 'Observation '+ str(observation_index)
output_data[column_name] = 0

#now, we loop over each pair of lambda and kappa in output_data
for row_index in range(len(output_data)):
    #create shorter-named variable for lambda and kappa for convenience
    l = output_data.loc[row_index, "lambda"]
    k = output_data.loc[row_index, "kappa"]

    #create x variable for convenience
    x = input_data.loc[observation_index, "Durations"]
    if input_data.loc[observation_index, "Censored"] == "no":
        #log (f(x))
        fx = (k / l) * (x / l) ** (k-1) * math.exp(-(x/l)**k)
        log_likelihood = np.log(fx)
    else:
        #log (R(x)) = log ( 1- F(x))
        Rx = math.exp(-(x/l)**k)
        log_likelihood = np.log(Rx)

    output_data.loc[row_index, column_name] = log_likelihood
```

```
#we add a column to the output dataframe
column_name = 'Observation '+ str(observation_index)
output_data[column_name] = 0
```

Creates the column

```
#now, we loop over each pair of lambda and kappa in output_data
for row_index in range(len(output_data)):
    #create shorter-named variable for lambda and kappa for convenience
    l = output_data.loc[row_index, "lambda"]
    k = output_data.loc[row_index, "kappa"]

    #create x variable for convenience
    x = input_data.loc[observation_index, "Durations"]
    if input_data.loc[observation_index, "Censored"] == "no":
        #log (f(x))
        fx = (k / l) * (x / l) ** (k-1) * math.exp(-(x/l)**k)
        log_likelihood = np.log(fx)
    else:
        #log (R(x)) = log ( 1- F(x))
        Rx = math.exp(-(x/l)**k)
        log_likelihood = np.log(Rx)

    output_data.loc[row_index, column_name] = log_likelihood
```

```
#we add a column to the output dataframe
column_name = 'Observation '+ str(observation_index)
output_data[column_name] = 0

#now, we loop over each pair of lambda and kappa in output_data
for row_index in range(len(output_data)):
    #create shorter-named variable for lambda and kappa for convenience
    l = output_data.loc[row_index, "lambda"]
    k = output_data.loc[row_index, "kappa"]

    #create x variable for convenience
    x = input_data.loc[observation_index, "Durations"]
    if input_data.loc[observation_index, "Censored"] == "no":
        #log (f(x))
        fx = (k / l) * (x / l) ** (k-1) * math.exp(-(x/l)**k)
        log_likelihood = np.log(fx)
    else:
        #log (R(x)) = log ( 1- F(x))
        Rx = math.exp(-(x/l)**k)
        log_likelihood = np.log(Rx)

    output_data.loc[row_index, column_name] = log_likelihood
```

Loop through the rows

```
#we add a column to the output dataframe
column_name = 'Observation '+ str(observation_index)
output_data[column_name] = 0

#now, we loop over each pair of lambda and kappa in output_data
for row_index in range(len(output_data)):
    #create shorter-named variable for lambda and kappa for convenience
    l = output_data.loc[row_index, "lambda"]
    k = output_data.loc[row_index, "kappa"]

    #create x variable for convenience
    x = input_data.loc[observation_index, "Durations"]
    if input_data.loc[observation_index, "Censored"] == "no":
        #log (f(x))
        fx = (k / l) * (x / l) ** (k-1) * math.exp(-(x/l)**k)
        log_likelihood = np.log(fx)
    else:
        #log (R(x)) = log ( 1- F(x))
        Rx = math.exp(-(x/l)**k)
        log_likelihood = np.log(Rx)

    output_data.loc[row_index, column_name] = log_likelihood
```

Get lambda and kappa from the first two columns

```
#we add a column to the output dataframe
column_name = 'Observation '+ str(observation_index)
output_data[column_name] = 0

#now, we loop over each pair of lambda and kappa in output_data
for row_index in range(len(output_data)):
    #create shorter-named variable for lambda and kappa for convenience
    l = output_data.loc[row_index, "lambda"]
    k = output_data.loc[row_index, "kappa"]

    #create x variable for convenience
    x = input_data.loc[observation_index, "Durations"]
    if input_data.loc[observation_index, "Censored"] == "no":
        #log (f(x))
        fx = (k / l) * (x / l) ** (k-1) * math.exp(-(x/l)**k)
        log_likelihood = np.log(fx)
    else:
        #log (R(x)) = log ( 1- F(x))
        Rx = math.exp(-(x/l)**k)
        log_likelihood = np.log(Rx)

    output_data.loc[row_index, column_name] = log_likelihood
```

Current duration

```
#we add a column to the output dataframe
column_name = 'Observation '+ str(observation_index)
output_data[column_name] = 0

#now, we loop over each pair of lambda and kappa in output_data
for row_index in range(len(output_data)):
    #create shorter-named variable for lambda and kappa for convenience
    l = output_data.loc[row_index, "lambda"]
    k = output_data.loc[row_index, "kappa"]

    #create x variable for convenience
    x = input_data.loc[observation_index, "Durations"]
    if input_data.loc[observation_index, "Censored"] == "no":
        #log (f(x))
        fx = (k / l) * (x / l) ** (k-1) * math.exp(-(x/l)**k)
        log_likelihood = np.log(fx)
    else:
        #log (R(x)) = log ( 1- F(x))
        Rx = math.exp(-(x/l)**k)
        log_likelihood = np.log(Rx)

    output_data.loc[row_index, column_name] = log_likelihood
```

Calculate log-likelihood

```
#we add a column to the output dataframe
column_name = 'Observation '+ str(observation_index)
output_data[column_name] = 0

#now, we loop over each pair of lambda and kappa in output_data
for row_index in range(len(output_data)):
    #create shorter-named variable for lambda and kappa for convenience
    l = output_data.loc[row_index, "lambda"]
    k = output_data.loc[row_index, "kappa"]

    #create x variable for convenience
    x = input_data.loc[observation_index, "Durations"]
    if input_data.loc[observation_index, "Censored"] == "no":
        #log (f(x))
        fx = (k / l) * (x / l) ** (k-1) * math.exp(-(x/l)**k)
        log_likelihood = np.log(fx)
    else:
        #log (R(x)) = log ( 1- F(x))
        Rx = math.exp(-(x/l)**k)
        log_likelihood = np.log(Rx)

    output_data.loc[row_index, column_name] = log_likelihood
```

Insert result in data frame

How to - Maximization

Step 4,…: Get the log-likelihood of your other observations for each pair of parameters.

For this, you can put the code in the previous slides in a loop.

	lambda	kappa	Observation 0	Observation 1	...	Observation 88	Observation 89	Observation 90	Observation 91
0	1.0	0.1	-5.652141	-5.196187	...	-1.177178	-5.831322	-4.608841	-4.845311
1	1.0	0.2	-5.056336	-4.564292	...	-1.385748	-5.252350	-3.943681	-4.191843
2	1.0	0.3	-4.834240	-4.271373	...	-1.631272	-5.064109	-3.588054	-3.857927
3	1.0	0.4	-4.838421	-4.158109	...	-1.920297	-5.125900	-3.375090	-3.678935
4	1.0	0.5	-5.043971	-4.184013	...	-2.260531	-5.421955	-3.254991	-3.607763
..	...	...	...	...	...	...	...	...	...
395	20.0	1.6	-3.270342	-3.364641	...	-0.112674	-3.274954	-3.610748	-3.500822
396	20.0	1.7	-3.255083	-3.398459	...	-0.098302	-3.242496	-3.709632	-3.573661
397	20.0	1.8	-3.244737	-3.437614	...	-0.085764	-3.214575	-3.813595	-3.651748
398	20.0	1.9	-3.238836	-3.481532	...	-0.074824	-3.190765	-3.922008	-3.734464
399	20.0	2.0	-3.236973	-3.529719	...	-0.065280	-3.170696	-4.034343	-3.821284

Intermediate result

How to - Maximization

Step 5: Add the sums of the log-likelihoods in the last column.

```
[400 rows x 94 columns]
    lambda kappa  Observation 0  Observation 1  ...  Observation 89  Observation 90  Observation 91  Loglikelihood_sum
0      1.0   0.1      -5.652141      -5.196187  ...       -5.831322       -4.608841       -4.845311        -392.223868
1      1.0   0.2      -5.056336      -4.564292  ...       -5.252350       -3.943681       -4.191843        -348.455416
2      1.0   0.3      -4.834240      -4.271373  ...       -5.064109       -3.588054       -3.857927        -330.057118
3      1.0   0.4      -4.838421      -4.158109  ...       -5.125900       -3.375090       -3.678935        -325.971539
4      1.0   0.5      -5.043971      -4.184013  ...       -5.421955       -3.254991       -3.607763        -333.857772
..     ...   ...            ...            ...  ...             ...             ...             ...                ...
395   20.0   1.6      -3.270342      -3.364641  ...       -3.274954       -3.610748       -3.500822        -250.655808
396   20.0   1.7      -3.255083      -3.398459  ...       -3.242496       -3.709632       -3.573661        -253.354121
397   20.0   1.8      -3.244737      -3.437614  ...       -3.214575       -3.813595       -3.651748        -256.418408
398   20.0   1.9      -3.238836      -3.481532  ...       -3.190765       -3.922008       -3.734464        -259.807312
399   20.0   2.0      -3.236973      -3.529719  ...       -3.170696       -4.034343       -3.821284        -263.485765
```

How to - Maximization

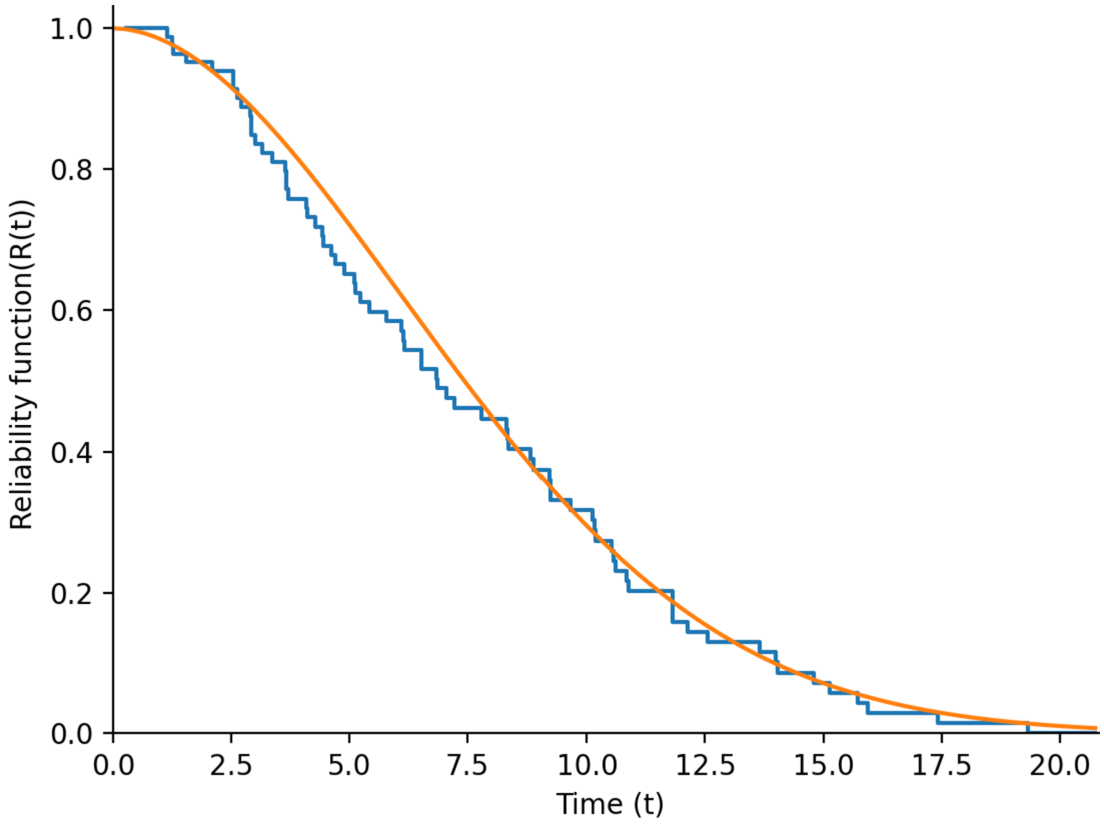
Step 6: Find the best pair of parameters.

Of course, these are in the row with the highest sum of log-likelihoods.

For my data, this is $\lambda = 9$ and $\kappa = 1.8$

And we are done!

Result


1.0
0.8
0.6
0.4
0.2
0.0
Reliability function(R(t))
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
20.0
Time (t)

Goodness-of-fit

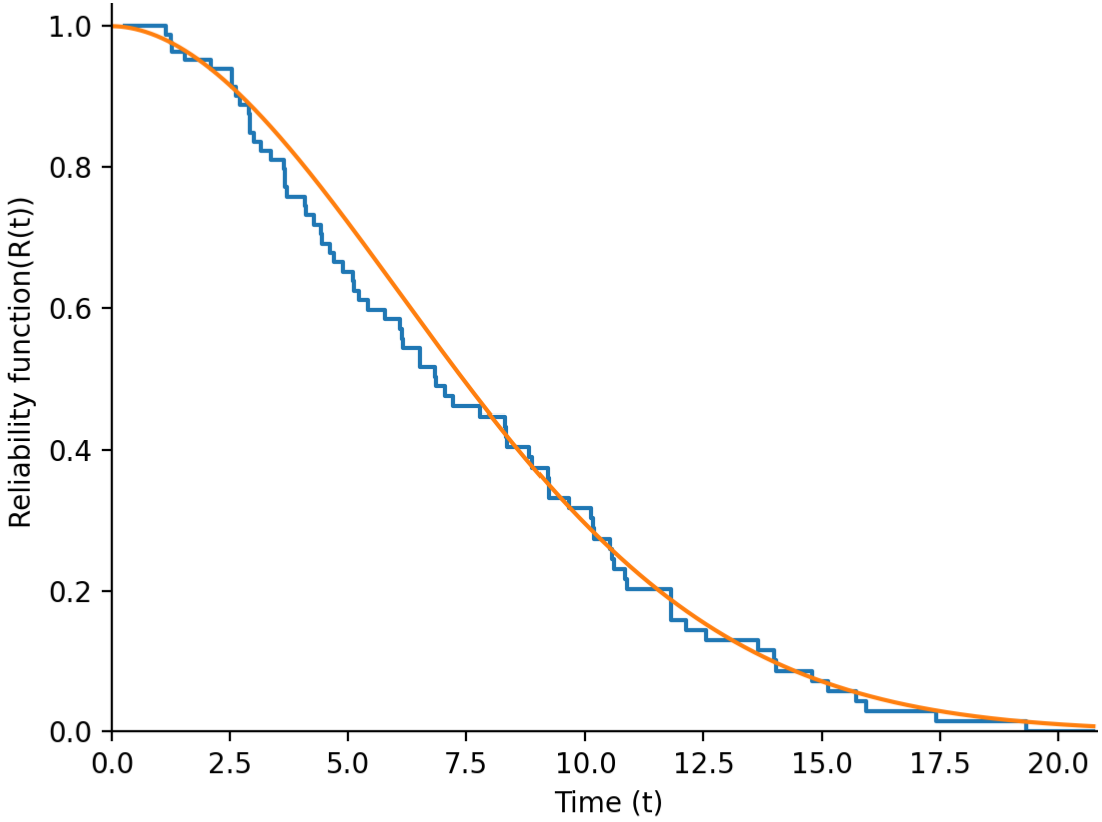
Goodness-of-fit

When your maximum likelihood estimator is done, you need to verify if it is reasonable.

This is easily done by comparing it to a Kaplan-Meier estimator.

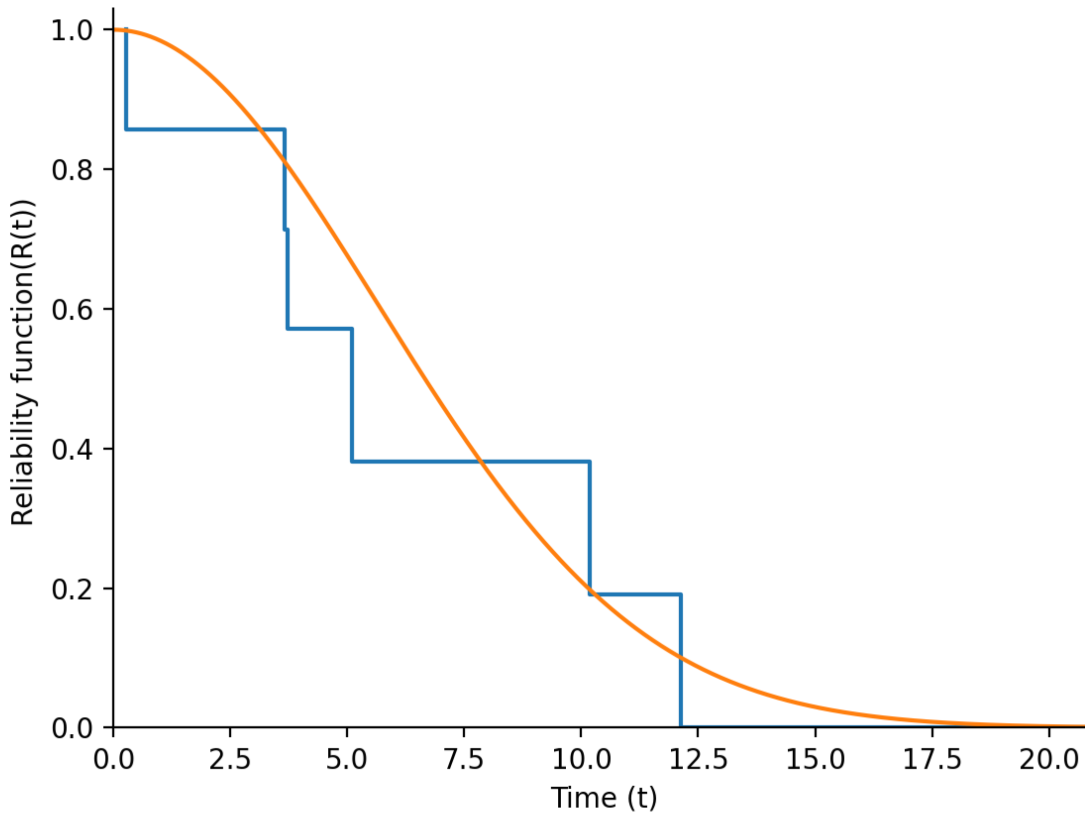
Some examples:

Goodness-of-fit



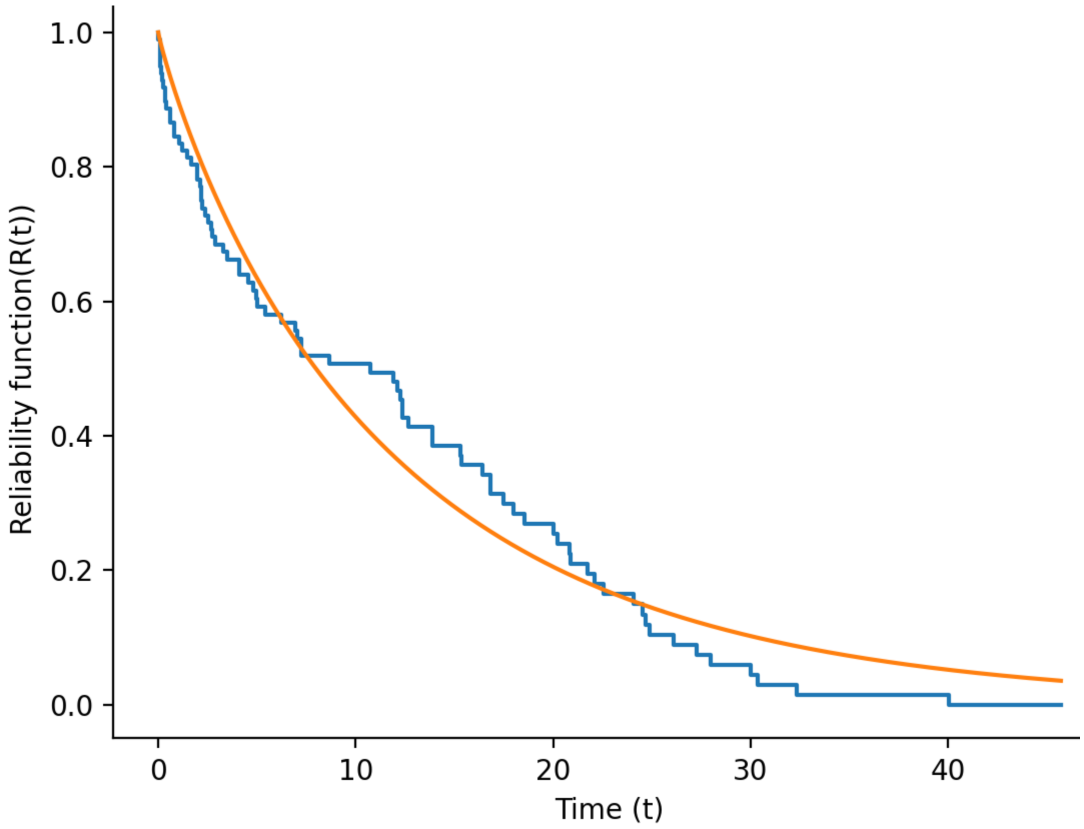
Looks fine!

Goodness-of-fit



Not enough datapoints to judge

Goodness-of-fit



Fit is relatively poor

Goodness-of-fit

What causes bad fit?

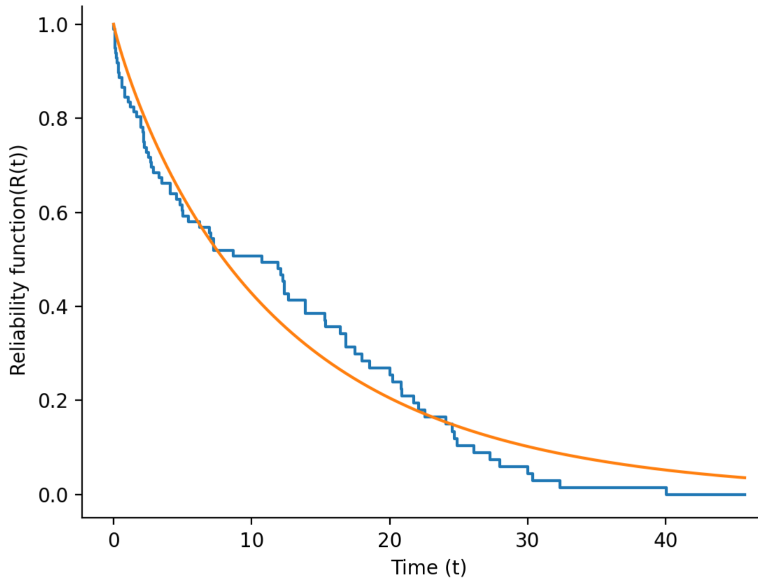
Methodological:

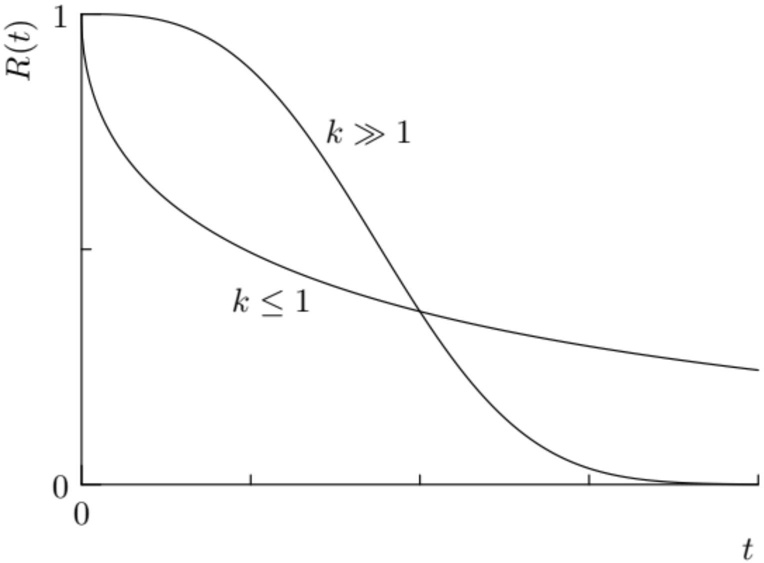
› Our range of parameters was insufficient.
› The optimization method was flawed.

But, if your analysis was correct:

› The Weibull distribution is a poor choice.

Goodness-of-fit





The shape of the Kaplan-Meier estimator does not seem to match any of the shapes the Weibull reliability function can take.

Suggests another distribution might be more appropriate.

Distributions

Recall that we selected the Weibull distribution for a variety of reasons:

- It is a continuous distribution (like the normal distribution)
- It is flexible, with two parameters.
- It does not result in negative durations.

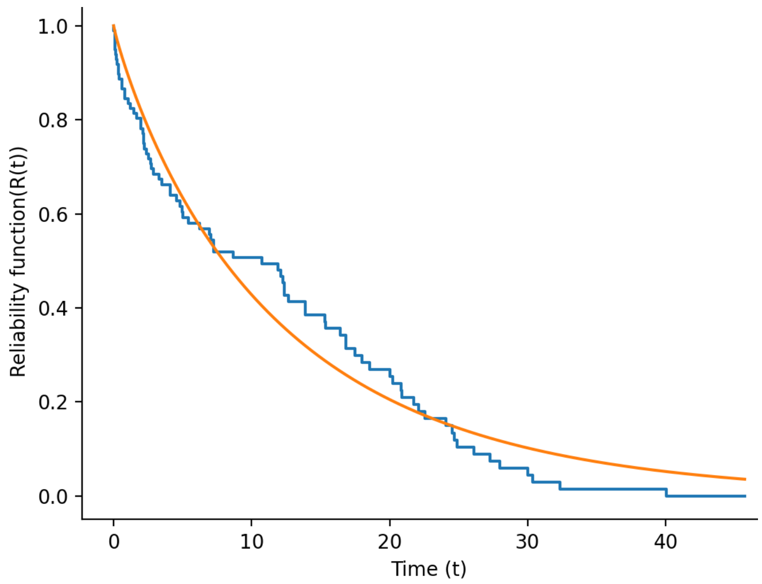
There are more distributions that might fit (e.g. log-normal, gamma, logistic, inverse gaussian, etc.)

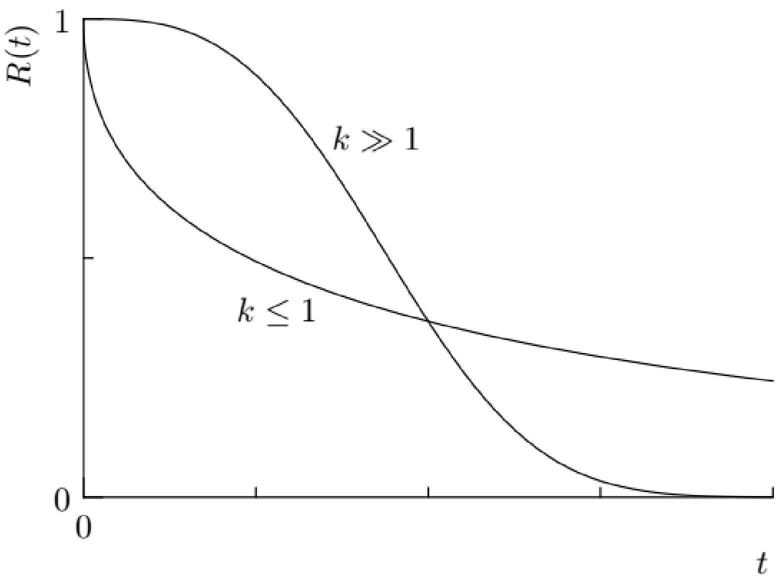
But, we can also use **mixtures of distributions**.

Mixtures of Weibull distributions

Mixtures of Weibull distributions

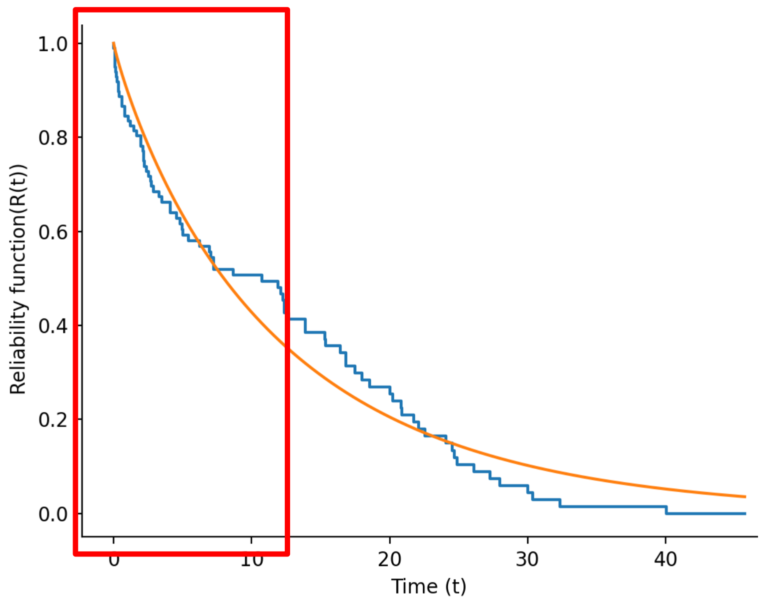
Lets again look at our more complex dataset.

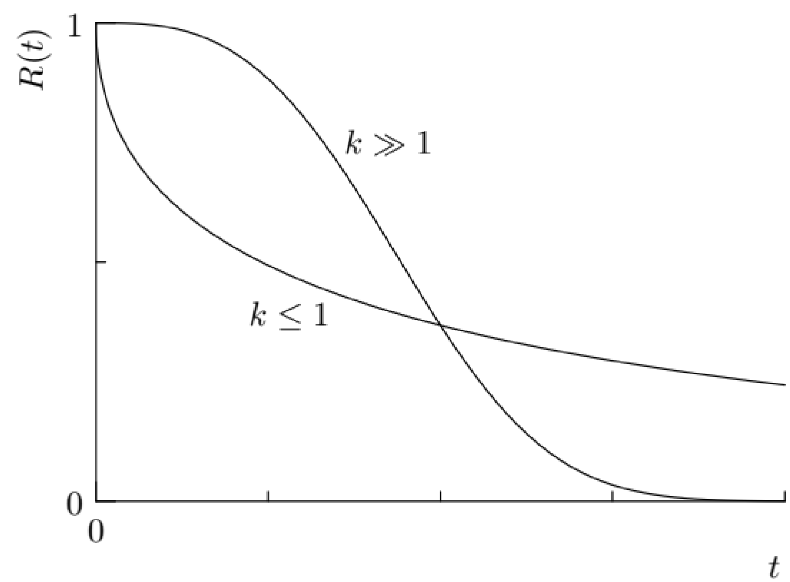




Mixtures of Weibull distributions

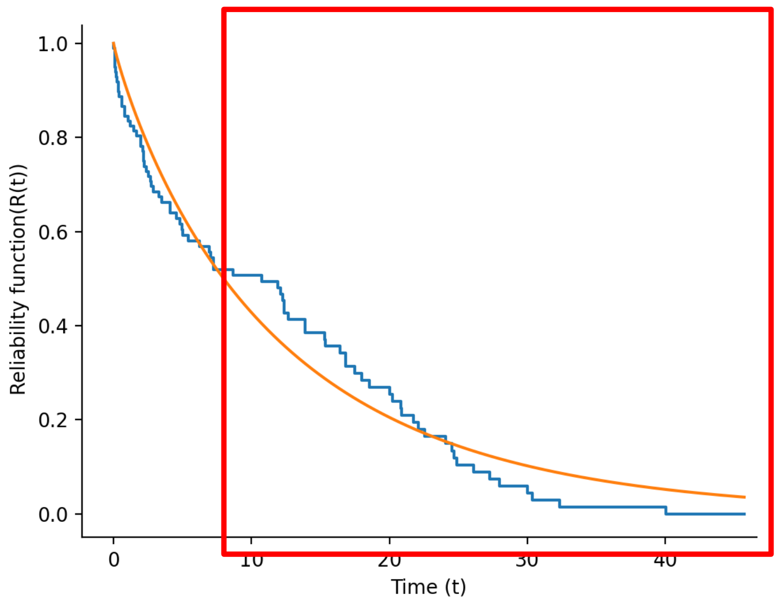
For smaller values of t, it is shaped like k < 1.

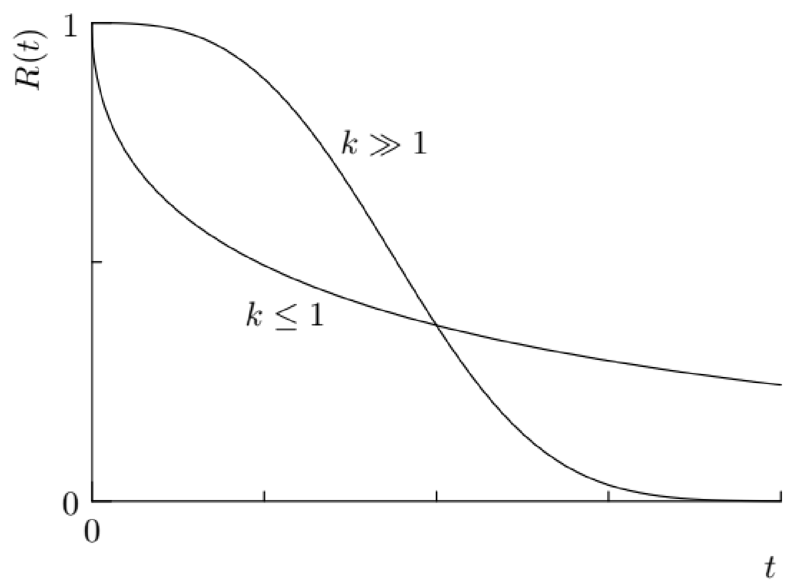




Mixtures of Weibull distributions

For larger values of t, it is shaped like k > 1.





Mixtures of Weibull distributions

This suggests that we can accurately model the data with a mixture of two Weibull distributions.

- With probability *p*, a time until failure comes from the first distribution, with λ_1 and κ_1 as its parameters.
- With probability 1-*p*, a time until failure comes from the second distribution, with λ_2 and κ_2 as its parameters.

A mixture of two Weibull distributions thus has five parameters:

p, λ_1, κ_1, λ_2, κ_2

Mixtures of Weibull distributions

As with the normal Weibull distribution, we can use three relevant functions:

Density function: $pf(t;\lambda_1,\kappa_1)+(1-p)f(t;\lambda_2,\kappa_2)$

Lifetime distribution function: $pF(t;\lambda_1,\kappa_1)+(1-p)F(t;\lambda_2,\kappa_2)$

Reliability function: $pR(t;\lambda_1,\kappa_1)+(1-p)R(t;\lambda_2,\kappa_2)$

Mixtures of Weibull distributions

Of course, we want to find the best values of
p, λ_1, κ_1, λ_2, κ_2 for our data.

The good news:
We can still use the maximum likelihood method.

The bad news:
We have to optimize 5 parameters. With similar ranges as before:
20x20x20x20x20 = 3.2 million options.

Optimization of larger problems

Clearly, we need a more scalable alternative.

Python does offer options for this, through the *scipy.optimize* package.

This package provides a *minimize()* function, which we can use to maximize the loglikelihood.

Likelihood maximization in Python

The *scipy.optimize.minimize()* function takes multiple (sometimes optional) inputs.

1. The function to minimize.
2. An initial guess for the solution.
3. Additional arguments for the function to minimize.
4. Bounds on your solution.

Likelihood maximization in Python

We first need to make a function that calculates the log-likelihood for any set of parameters.

This is not the same as what we did with the brute-force algorithm, where we calculated the log-likelihood for all parameters at the same time!

```python
def mix_weibull_likelihood(parameters, input_data):
    #separate the parameters
    p, l1, k1, l2, k2 = parameters
    #get a place to store the likelihood values
    loglikelihoods = np.zeros(len(input_data))
    #loop
    for observation_index in range(len(input_data)):
        x = input_data.loc[observation_index, "Duration"]

        if input_data.loc[observation_index, "Censored"] == "no":
            #log (f(x))
            likelihood = p * (k1 / l1) * (x / l1) ** (k1-1) * math.exp(-(x/l1)**k1) + \
                (1-p) * (k2 / l2) * (x / l2) ** (k2-1) * math.exp(-(x/l2)**k2)
            loglikelihoods[observation_index] = np.log(likelihood)
        else:
            #log (R(x)) = log ( 1- F(x))
            likelihood = p * math.exp(-(x/l1)**k1) + (1-p) * math.exp(-(x/l2)**k2)
            loglikelihoods[observation_index] = np.log(likelihood)

    return -np.sum(loglikelihoods)
```

```
def mix_weibull_likelihood(parameters, input_data):
    #separate the parameters
    p, l1, k1, l2, k2 = parameters
    #get a place to store the likelihood values
    loglikelihoods = np.zeros(len(input_data))
    #loop
    for observation_index in range(len(input_data)):
        x = input_data.loc[observation_index, "Duration"]

        if input_data.loc[observation_index, "Censored"] == "no":
            #log (f(x))
            likelihood = p * (k1 / l1) * (x / l1) ** (k1-1) * math.exp(-(x/l1)**k1) + \
                (1-p) * (k2 / l2) * (x / l2) ** (k2-1) * math.exp(-(x/l2)**k2)
            loglikelihoods[observation_index] = np.log(likelihood)
        else:
            #log (R(x)) = log ( 1- F(x))
            likelihood = p * math.exp(-(x/l1)**k1) + (1-p) * math.exp(-(x/l2)**k2)
            loglikelihoods[observation_index] = np.log(likelihood)

    return -np.sum(loglikelihoods)
```

Parameters are bundled for the minimize() function.

```python
def mix_weibull_likelihood(parameters, input_data):
    #separate the parameters
    p, l1, k1, l2, k2 = parameters
    #get a place to store the likelihood values
    loglikelihoods = np.zeros(len(input_data))
    #loop
    for observation_index in range(len(input_data)):
        x = input_data.loc[observation_index, "Duration"]

        if input_data.loc[observation_index, "Censored"] == "no":
            #log (f(x))
            likelihood = p * (k1 / l1) * (x / l1) ** (k1-1) * math.exp(-(x/l1)**k1) + \
                (1-p) * (k2 / l2) * (x / l2) ** (k2-1) * math.exp(-(x/l2)**k2)
            loglikelihoods[observation_index] = np.log(likelihood)
        else:
            #log (R(x)) = log ( 1- F(x))
            likelihood = p * math.exp(-(x/l1)**k1) + (1-p) * math.exp(-(x/l2)**k2)
            loglikelihoods[observation_index] = np.log(likelihood)

    return -np.sum(loglikelihoods)
```

Loop through the observations, storing the duration in *x*

```python
def mix_weibull_likelihood(parameters, input_data):
    #separate the parameters
    p, l1, k1, l2, k2 = parameters
    #get a place to store the likelihood values
    loglikelihoods = np.zeros(len(input_data))
    #loop
    for observation_index in range(len(input_data)):
        x = input_data.loc[observation_index, "Duration"]

        if input_data.loc[observation_index, "Censored"] == "no":
            #log (f(x))
            likelihood = p * (k1 / l1) * (x / l1) ** (k1-1) * math.exp(-(x/l1)**k1) + \
                (1-p) * (k2 / l2) * (x / l2) ** (k2-1) * math.exp(-(x/l2)**k2)
            loglikelihoods[observation_index] = np.log(likelihood)
        else:
            #log (R(x)) = log ( 1- F(x))
            likelihood = p * math.exp(-(x/l1)**k1) + (1-p) * math.exp(-(x/l2)**k2)
            loglikelihoods[observation_index] = np.log(likelihood)

    return -np.sum(loglikelihoods)
```

Calculate the mixed-Weibull log-likelihood values

```python
def mix_weibull_likelihood(parameters, input_data):
    #separate the parameters
    p, l1, k1, l2, k2 = parameters
    #get a place to store the likelihood values
    loglikelihoods = np.zeros(len(input_data))
    #loop
    for observation_index in range(len(input_data)):
        x = input_data.loc[observation_index, "Duration"]

        if input_data.loc[observation_index, "Censored"] == "no":
            #log (f(x))
            likelihood = p * (k1 / l1) * (x / l1) ** (k1-1) * math.exp(-(x/l1)**k1) + \
                (1-p) * (k2 / l2) * (x / l2) ** (k2-1) * math.exp(-(x/l2)**k2)
            loglikelihoods[observation_index] = np.log(likelihood)
        else:
            #log (R(x)) = log ( 1- F(x))
            likelihood = p * math.exp(-(x/l1)**k1) + (1-p) * math.exp(-(x/l2)**k2)
            loglikelihoods[observation_index] = np.log(likelihood)

    return -np.sum(loglikelihoods)
```

We want to maximize, but Python minimizes by default!

Likelihood maximization in Python

Next, we decide on an initial guess.

We decide on the values for p, λ_1, κ_1, λ_2, κ_2 where the algorithm will start to search.

```
#define an initial guess: (p, l1, k1, l2, k2)
guess = [0.5, 4, 0.5, 15, 1.5]
```

Likelihood maximization in Python

Next, we give the additional arguments for the function. In our case, this is the historical data.

```
[100 rows x 3 columns]
      Time Censored  Duration
0    13.89       no     13.89
1    24.19      yes     10.30
2    36.57       no     12.38
3    41.97       no      5.40
4    46.77       no      4.80
..     ...      ...       ...
95  974.87       no      2.14
96  979.31       no      4.44
97  982.21       no      2.90
98  987.19       no      4.98
99  987.20       no      0.01
```

Likelihood maximization in Python

Finally, set some bounds on the values that p, λ_1, κ_1, λ_2, κ_2 can take.

This is similar to the brute-force method, where we decided on a search range beforehand.

Each parameter has a lower and upper bound.

```
#decide on a search area for your variables
bounds = [(0,1), (1,30), (0.1,4), (1,30), (0.1,4)]
```

Likelihood maximization in Python

Now we can run the function!

```
#this function does the optimization
result = scipy.optimize.minimize(mix_weibull_likelihood, guess, input_data, bounds=bounds)
```

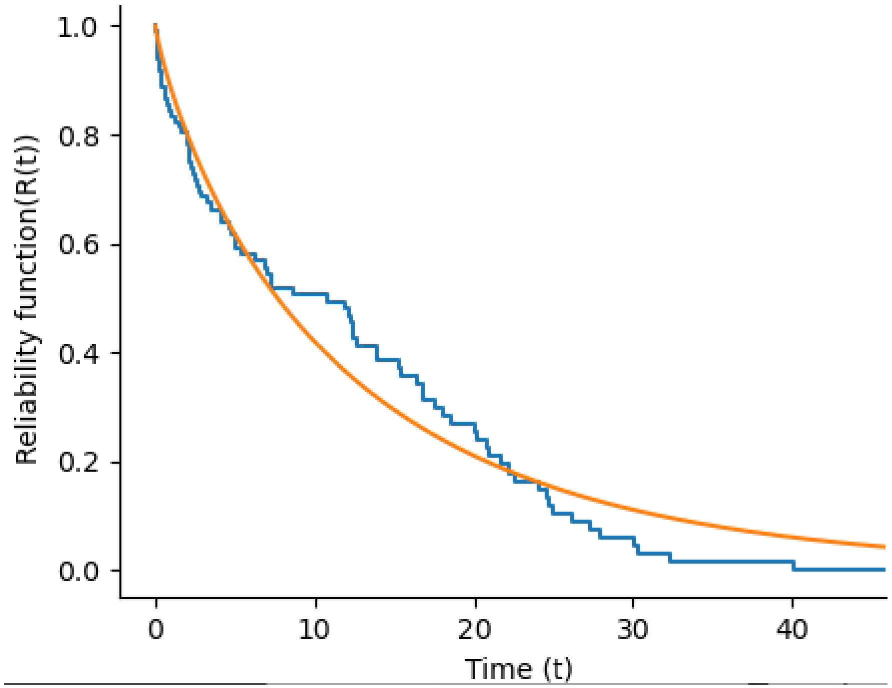
The result variable will contain all the information related to the minimization.

We care about *result.x*:

```
    x: array([ 0.45176582,  2.89643374,  0.82928152, 22.47452599,  2.358543  ])
```

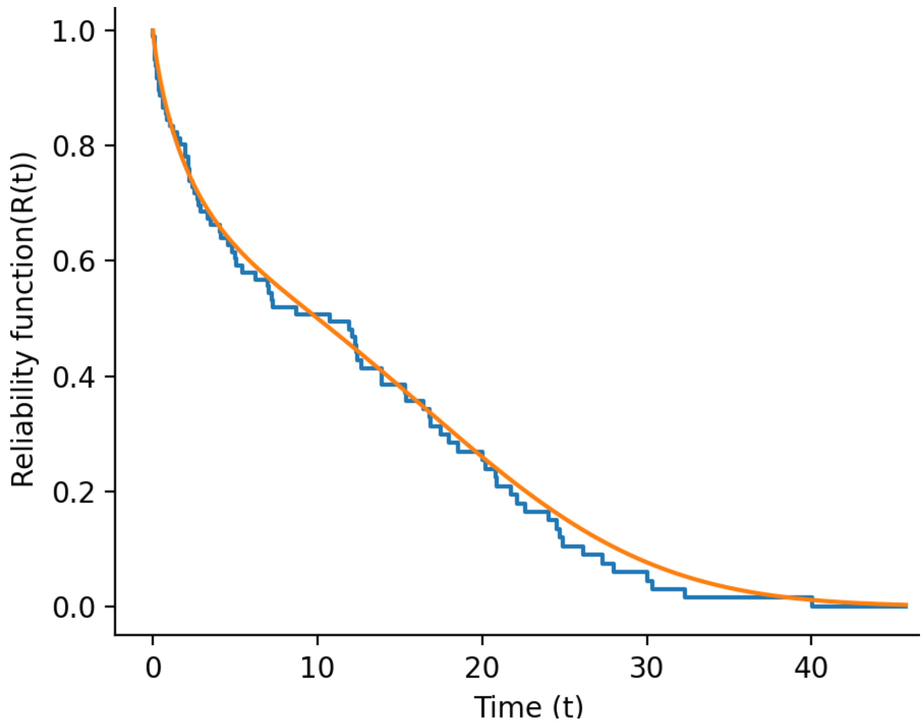
Likelihood maximization in Python

Your initial solution can have a massive impact on the result!
For example: starting with (p, λ_1, κ_1, λ_2, κ_2)= (0.5, 10, 1, 10, 1)



Likelihood maximization in Python

Instead, starting with (p, λ_1, κ_1, λ_2, κ_2)= (0.5, 5, .5, 15, 1.5):



Likelihood maximization in Python

So, to recap:

When we want to optimize a mixture of two Weibull distributions, we need more complex algorithms.

Python can do this, but we don’t really know what is going on anymore -> **More verification is needed here.**

Interpreting a mixed Weibull

As a solution we found:

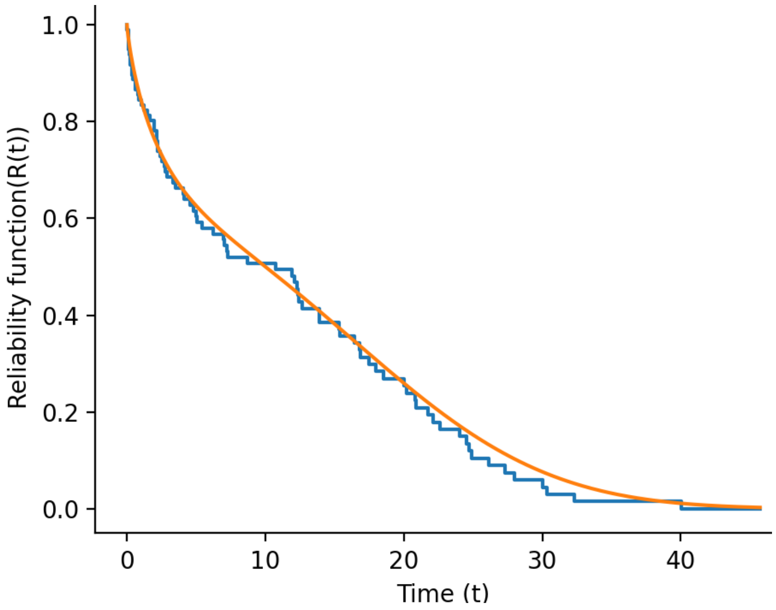
$(p, \lambda_1, \kappa_1, \lambda_2, \kappa_2) = (0.45, 2.90, 0.83, 22.47, 2.36)$

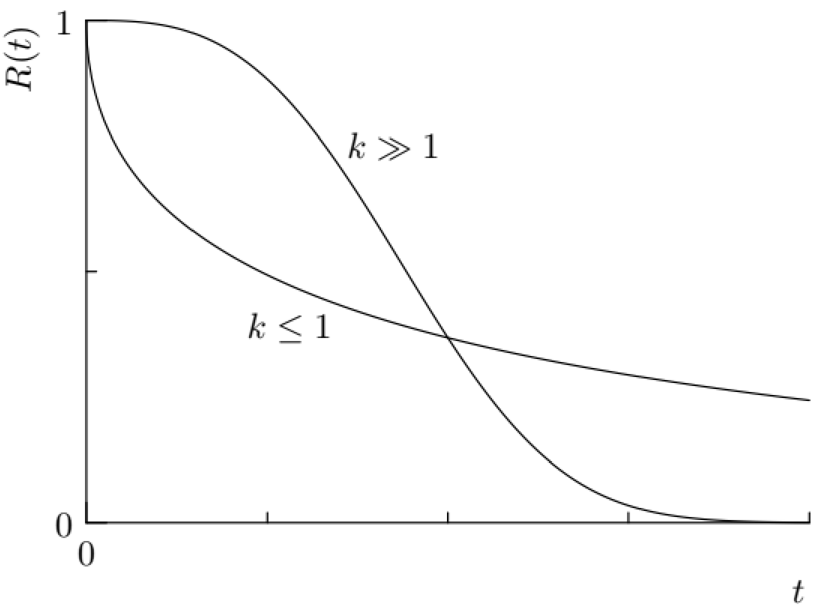
So, we have one Weibull distribution with $\kappa_1<1$, and a smaller λ_1. -> **Decreasing failure rate at small durations.**

The other Weibull distribution instead has a $\kappa_2>1$, and a larger λ_2. -> **Increasing failure rate at larger durations.**

Interpreting a mixed Weibull

If we recall our discussion of the shapes from earlier, this make sense.





Interpreting a mixed Weibull

What does this mean for maintenance policies?

Remember:

Increasing failure rates -> Preventive maintenance makes sense.

Decreasing failure rates -> No preventive maintenance.

In this case, we might do preventive maintenance in the second half of the reliability function.

Interpreting a mixed Weibull

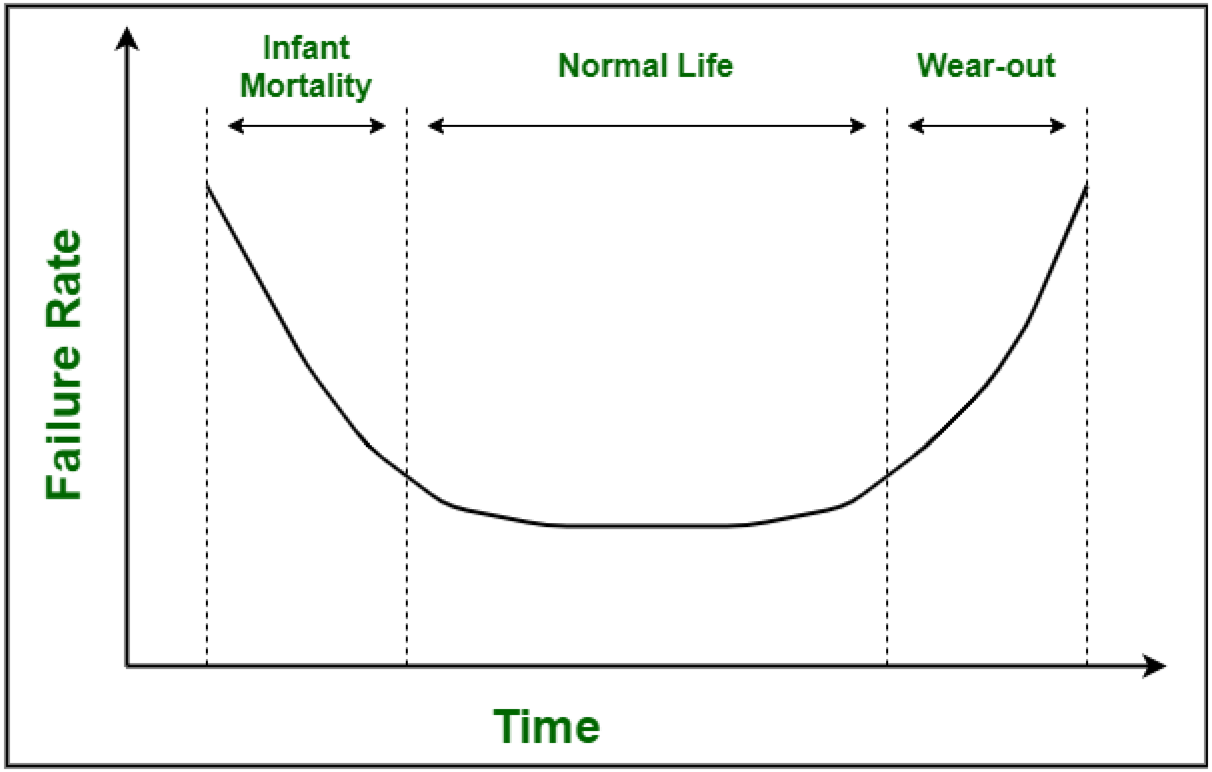
Systems where the failure rate is first decreasing and then increasing are actually quite common:

Failures are common at the start due to defects, production errors, poor maintenance jobs.

Failures are common after a long time due to wear.

We call this a **bathtub shaped** failure rate.

Example of a bathtub shape



Bathtub Curve

The end

Today should have discussed everything (and more) that you need to know when fitting a Weibull-distribution.

Next week: Condition-based maintenance