000_Intro

December 4, 2024

Oreum Industries Reference Project, 2024Q3

---

1 000_Intro.ipynb

1.0.1 Oreum Reference - Survival Regression `oreum_survival`

Demonstrate Survival Regression Modelling using Bayesian inference and a Bayesian workflow, specifically using the `pymc` & `arviz` ecosystem.

This intro can also be used for verbal presentation and discussion purposes, ideally followed by a deeper technical walkthrough of the project in a long-form style. There we evaluate the behaviour and performance of the models throughout the workflows, including several state-of-the-art methods unavailable to conventional max-likelihood / machine-learning models.

PDF version

1.0.2 *What is Survival Regression?*

We seek to create *principled* models that provide explanatory inference and predictions of the Survival Function $\widehat{S}(t)$ and Expected Time-to-Event $\widehat{\mathbb{E}}_t$ of real-world events experienced by a population, using quantified uncertainty to support real-world decision-making.

General project approach

The emphasis in this project is to build a variety of models of increasing sophistication and demonstrate their usage. We strike a balance between building up concepts & methods (similar to Betancourt's detailed article) vs practical application & worked examples in a `pmyc`-based Bayesian workflow.

We don't focus on specific analysis of the dataset, nor try to infer too much. The dataset is simply a good substrate on which to learn and demonstrate the variety of model architectures used herein.

We evaluate the behaviour and performance of the models throughout the workflows, including several state-of-the-art methods unavailable to conventional max-likelihood / machine-learning models

This series of Notebooks covers

- `000_Intro.ipynb`: Orientiation and Fundamental Concepts of Survival

- The **10x__Exponential family**: basic exponential parametric and semi-parametric (piecewise) structure
  - `100_Exponential_Univariate.ipynb`: A Parametric Univariate Model: Constant Hazard
  - `101_Exponential_Regression.ipynb`: A Parametric Regression aka Accelerated Failure Time Model: Exponential
  - `102_Exponential_CoxPH0.ipynb`: A Semi-Parametric aka Piecewise Regression Model: CoxPH
- The **20x__AFT family**: extension of parametric models with focus on Accelerated Failure Time structure
  - `200_AFT_Weibull.ipynb`: A Parametric Regression aka AFT Model: Weibull
  - `201_AFT_Gompertz.ipynb`: A Parametric Regression aka AFT Model: Gompertz
  - `202_AFT_GompertzAlt.ipynb`: A Parametric Regression aka AFT Model: Gompertz Alternative Parameterization
- The **30x__Piecewise family**: extension of semi-parametric models with focus on Hierarchical & Ordinal structure
  - `300_Piecewise_CoxPH1.ipynb`: Extending CoxPH: Hierarchical Baseline Hazard
  - `301_Piecewise_CoxPH2.ipynb`: Extending CoxPH: Ordinal Hierarchical Baseline Hazard
  - `302_Piecewise_CoxPH3.ipynb`: Extending CoxPH: Ordinal Hierarchical Baseline Hazard & Coeffs

Useful references

This also references / remixes / extends a handful published guides & notes including:

- WWS507 Chap7, online here and also saved to pdf in this project repo at `../assets/pdf/wws507_pop507_generalized_linear_statistical_models_c7.pdf`
- Michael Betancourt's article on Bayesian survival model fundamentals
- Austin Rochfort's gist and updated gist both on CoxPH, and gist on AFT
- Chris Fonnesbeck's gist
- The `pymc-examples` docs
- The `lifelines` api docs
- Monica Alexander's workflow example
- This is a comprehensive update of `appliedai_bayesiansurvivaldemo/01_BayesianSurvivalModelling` written in 2016

In this Notebook

We dive straight into **Orientation** and **Fundamental General Abstractions** with a simple real-world observational censored dataset, and then go on to demonstrate the theory and usage of an increasing sophistication of models.

1.1 Contents

---

2 Setup

2.1 Imports

```
import sys
from pathlib import Path

import lifelines as sa
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
import statsmodels.api as sm
from pyprojroot.here import here
from scipy import stats

# prepend local project src files
module_path = here('src').resolve(strict=True)
if str(module_path) not in sys.path:
    sys.path.insert(0, str(module_path))

from oreum_core import curate, eda

from engine import logger

import warnings  # isort:skip # suppress seaborn, it's far too chatty

warnings.simplefilter(action='ignore', category=FutureWarning)  # isort:skip
import seaborn as sns
```

Notebook config

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

log = logger.get_logger('000_Intro', notebook=True)
_ = logger.get_logger('oreum_core', notebook=True)
```

2.1.1 Local Functions and Global Vars

```
LOAD_FROM_CSV = True
RNG = np.random.default_rng(seed=42)
```

2.1.2 Data Connections and Helper Objects

```
csvio = curate.PandasCSVIO(here(Path('data', 'raw', 'verbatim')).
 ↪resolve(strict=True))
ppqio = curate.PandasParquetIO(here(Path('data', 'prepared')).
 ↪resolve(strict=True))
figio = eda.FigureIO(here(Path('plots')).resolve(strict=True))
staticimgio = eda.FigureIO(here(Path('assets', 'img')).resolve(strict=True))
```

---

3 Preamble: Why Bayes?

3.1 We gain massive advantage by using a Bayesian Framework

We specifically use **Bayesian Inference** rather than Frequentist Max-Likelihood methods for many reasons, including:

	Bayesian Inference	Frequentist Max-Likelihood
General formulation → Desirable Trait ↓	*Bayes' Rule* $\underbrace{P(\hat{\mathcal{H}}\|D)}_{\text{posterior}} = \frac{\overbrace{P(D\|\mathcal{H})}^{\text{likelihood}} \cdot \overbrace{P(\mathcal{H})}^{\text{prior}}}{\underbrace{P(D)}_{\text{evidence}}}$	*MLE* $\hat{\mathcal{H}}^{\text{MLE}} \propto \arg\max_{\mathcal{H}} P(D\|\mathcal{H})$
***Principled* model structure represents hypothesis about the data-generating process**	**Very strong** Can build bespoke arbitrary and hierarchical structures of parameters to map to the real-world data-generating process.	**Weak** Can only state structure under strict limited assumptions of model statistical validity.
Model parameters and their initial values represent domain expert knowledge	**Very strong** Marginal prior distributions represent real-world probability of parameter values before any data is seen.	**Very weak** No concept of priors. Lack of joint probability distribution can lead to discontinuities in parameter values.

	Bayesian Inference	Frequentist Max-Likelihood
Robust parameter fitting process	**Strong** Estimate full joint posterior probability mass distribution for parameters - more stable and representative of the expectation for the parameter values. Sampling can be a computationally expensive process.	**Weak** Estimate single-point max-aposterioi-likelihood (density) of parameters - this can be far outside the probability mass and so is prone to overfitting and only correct in the limit of infinite data. But optimization method can be computationally cheap.
Fitted parameters have meaningful summary statistics for inference	**Very strong** Full marginal probability distributions can be interpreted exactly as probabilities.	**Weak** Point estimates only have meaningful summary statistics under strict limited assumptions of model statistical validity.

continues ...

... continued

Desirable Trait	Bayesian Inference	Frequentist Max-Likelihood
Robust model evaluation process	**Strong** Use entire dataset, evaluate via Leave-One-Out Cross Validation (best theoretically possible).	**Weak** Cross-validation rarely seen in practice, even if used, rarely better than 5-fold CV. Simplistic method can be computationally cheap.
Predictions made with quantified variance	**Very strong** Predictions made using full posterior probability distributions, so predictions have full empirical probability distributions.	**Weak** Predictions using point estimates can be bootstrapped, but predictions only have interpretation under strict limited assumptions of model validity.
Handle imbalanced, high cardinality & hierarchical factor features	**Very strong** Can introduce partial-pooling to automatically balance factors through hierarchical priors.	**Weak** Difficult to introduce partial-pooling (aka mixed random effects) without affecting strict limited assumptions of model validity.
Handle skewed / multimodal / extreme value target variable	**Very strong** Represent the model likelihood as any arbitrary probability distribution, including mixture (compound) functions e.g. a zero-inflated Weibull.	**Weak** Represent model likelihood with a usually very limited set of distributions. Very difficult to create mixture compound functions.

Desirable Trait	Bayesian Inference	Frequentist Max-Likelihood
Handle small datasets	**Very strong** Bayesian concept assumes that there is a probable range of values for each parameter, and that we evidence our prior on any amount of data (even very small counts).	**Very weak** Frequentist concept assumes that there is a single true value for each parameter and that we only discover that value in the limit (of infinite observations).
Automatically impute missing data	**Very strong** Establish a prior for each datapoint, evidence on the available data within the context of the model, to automatically impute missing values.	**Very weak** No inherent method. Usually impute as a pre-processing step with weak non-modelled methods.

3.2 Practical Implementations of Bayesian Inference

We briefly referenced *Bayes Rule* above, which is a useful mnemonic when discussing Bayesian Inference, but in practice the crux of putting these advanced statistical techniques into practice is estimating the evidence $P(D)$ i.e. the probability of observing the data that we use to evidence the model

$$\underbrace{P(\hat{\mathcal{H}}|D)}_{\text{posterior}} = \frac{\overbrace{P(D|\mathcal{H})}^{\text{likelihood}} \cdot \overbrace{P(\mathcal{H})}^{\text{prior}}}{\underbrace{P(D)}_{\text{evidence}}}$$

...where:

$$P(D) \sim \int_{\Theta} P(D, \theta)\ d\theta$$

This joint probability $P(D, \theta)$ of data D and parameters θ requires an almost impossible-to-solve integral over parameter-space Θ. Rather than attempt to calculate that integral, we do something that sounds far more difficult, but given modern computing capabilities is actually practical.

3.2.1 We use a Bleeding-edge MCMC Toolkit for Bayesian Inference: `pymc` & `arviz`

We use **Markov Chain Monte-Carlo (MCMC)** sampling to take a series of *ergodic*, *partly-reversible*, *partly-randomised* samples of model parameters θ, and at each step compute the ratio of log-likelihoods $\log P(D|\mathcal{H})$ between a starting position (current values) θ_{p0} and proposed "sampled" position θ_p in parameter space, so as to reduce that log-likelihood (whilst exploring the parameter space).

This results in a posterior estimate $P(\hat{\theta}|D)$:

$$P(\hat{\theta}|D) \sim \frac{\overbrace{P(D|\theta_p)}^{\text{likelihood @ proposal}} \cdot \overbrace{P(\theta_p)}^{\text{prior @ proposal}}}{\underbrace{P(D|\theta_{p0})}_{\text{likelihood @ current}} \cdot \underbrace{P(\theta_{p0})}_{\text{prior @ current}}}$$

This is the heart of MCMC sampling: for detailed practical explanations see Betancourt, 2021 and Tweicki, 2015

We use the bleeding-edge `pymc` and `arviz` Python packages to provide the full Bayesian toolkit that we require, including advanced sampling, probabilistic programming, statistical inferences, model evaluation and comparison, and more.

```
f = figio.read(fn='../assets/img/logos',figsize=(12, 2))
```

---

4 0. Load Dataset

For this Intro Notebook we'll use a simple dataset with censoring: the `mastectomy` dataset from the `HSAUR` R package via `statsmodels`

Per the documentation this dataset represents:

> Survival times in months after mastectomy of women with breast cancer. The cancers are classified as having metastasized or not based on a histochemical marker.

- `time` is the number of months since mastectomy,
- `event` indicates whether the woman died at the corresponding `time` (if `True`) or the observation was censored (if `False`). In the context of survival analysis, censoring means that the woman survived past the corresponding time, but that her death was not observed. Censoring (and its counterpart truncation) represents a fundamental challenge in survival analysis
- `metastized` indicates whether the woman's cancer had metastasized

```
if LOAD_FROM_CSV:
    dfr = csvio.read(fn='mastectomy', index_col='rowid')
else:
    dfr = sm.datasets.get_rdataset(dataname='mastectomy', package='HSAUR',␣
 ↪cache=True).data
```

```
    _ = csvio.write(df=dfr, fn='mastectomy')
eda.display_ht(dfr)
```

```
       time  event metastized
rowid
0        23   True         no
1        47   True         no
2        69   True         no
41      212  False        yes
42      217  False        yes
43      225  False        yes
```

```
'Shape: (44, 3), Memsize 0.0 MB'
```

Correct dtypes etc

```
df = dfr.copy()
df['pid'] = ['p{}'.format(x) for x in range(len(df))]
df = df.reset_index(drop=True).set_index('pid')
df.rename(columns={'time': 'duration', 'event': 'death', 'metastized': 'met'},␣
 ↪inplace=True)
df['death'] = df['death'].astype(bool)
df['met'] = df['met'].apply(lambda x: True if str(x).strip() == 'yes' else␣
 ↪False)
eda.describe(df)
```

```
              28      3     33   dtype  count_null  count_inf  count_zero  \
ft
index: pid   p28     p3    p33  object           0        NaN         NaN
duration      68     70    109   int64           0        0.0         0.0
death       True  False  False    bool           0        NaN         NaN
met         True  False   True    bool           0        NaN         NaN

           count_unique   top freq     sum   mean    std  min    25%   50%  \
ft
index: pid           44    p0    1     NaN    NaN    NaN   p0    NaN   NaN
duration            NaN   NaN  NaN  4251.0  96.61  69.87  5.0  38.75  73.5
death                 2  True   26     NaN    NaN    NaN  NaN    NaN   NaN
met                   2  True   32     NaN    NaN    NaN  NaN    NaN   NaN

               75%    max
ft
index: pid     NaN     p9
duration    145.75  225.0
death          NaN    NaN
met            NaN    NaN
```

```
'Shape: (44, 4), Memsize 0.0 MB'
```

---

5 1. Orientation: Survival Functions and Non-Parametric Estimators

5.1 1.1 Example Frequency Tables & Lifetime Plots

5.1.1 1.1.1 Table: Death event

```
def freq_prop_table(df, ft):
    counts = df.groupby(ft).size()
    return pd.DataFrame({'freq': counts, 'prop': counts / len(df)})


freq_prop_table(df, 'death')
```

```
       freq      prop
death
False    18  0.409091
True     26  0.590909
```

Observe:

- 26 obs (59%) are observed death events within the observation period
- 18 obs (41%) are right-censored, see §1.2 for plots and discussion

5.1.2 1.1.2 Death grouped by `met`

```
piv = df.pivot_table(values='duration', index='met', columns='death',␣
 ↪aggfunc='count', margins=True)
piv
```

```
death  False  True  All
met
False      7     5   12
True      11    21   32
All       18    26   44
```

Observe: + Deaths within the metastized group are `21 / 32 = 65.6%` + Deaths within the non-metastized group are `5 / 12 = 41.7%`

5.1.3 1.1.3 Chisquare test of proportions (aka prop.test aka ztest)

```
chi2, pval, _ = sm.stats.proportions_chisquare(piv.iloc[:-1, 1], piv.iloc[:-1,␣
 ↪-1])
print(f'chi2: {chi2:.2f}\npval: {pval:.2f}')
```

```
chi2: 2.07
pval: 0.15
```

Observe + The correlation of `met` and `death` appears unlikely to be pure chance

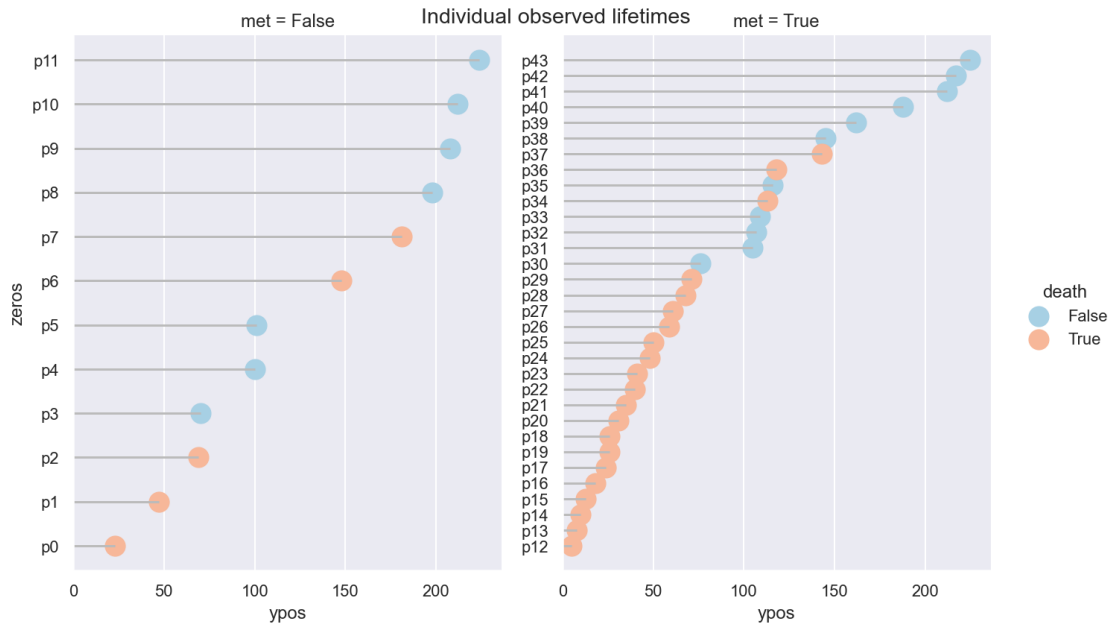
5.1.4 1.1.4 Plot individual lifetimes: `duration` x `death` (x `met`)

For each patient:

- Duration of observation
- Metastization
- Death event (if observed, allowing for censoring)

```
dfp = df.sort_values(['met', 'duration', 'death'], ascending=[1, 0, 1]).
 ↪reset_index()
# dfp = df.sort_values(['met','duration','death']).reset_index()
dfp['ypos'] = dfp['pid'].apply(lambda x: (11 - int(x.strip('p')) if int(x.
 ↪strip('p')) < 12 else 43 - int(x.strip('p'))))  # hack
dfp['color'] = dfp.apply(lambda r: sns.color_palette('RdBu_r', 2).
 ↪as_hex()[r['death']], axis=1)
dfp['zeros'] = np.zeros(len(dfp))

g = sns.FacetGrid(col='met', data=dfp, height=6, aspect=0.8, sharey=False)
_ = g.map(sns.pointplot, 'duration', 'pid', 'death', palette='RdBu_r',␣
 ↪join=False, orient='h', scale=1.6).add_legend(title='death')
_ = g.map(plt.hlines, 'ypos', 'zeros', 'duration', color='#bbbbbb')
_ = g.axes.flat[0].set_xlim(left=0)
_ = g.fig.suptitle('Individual observed lifetimes')
```

```
/Users/jon/miniforge/envs/oreum_survival/lib/python3.11/site-
packages/seaborn/axisgrid.py:712: UserWarning: Using the pointplot function
without specifying `order` is likely to produce an incorrect plot.
  warnings.warn(warning)
/Users/jon/miniforge/envs/oreum_survival/lib/python3.11/site-
packages/seaborn/axisgrid.py:717: UserWarning: Using the pointplot function
without specifying `hue_order` is likely to produce an incorrect plot.
  warnings.warn(warning)
```

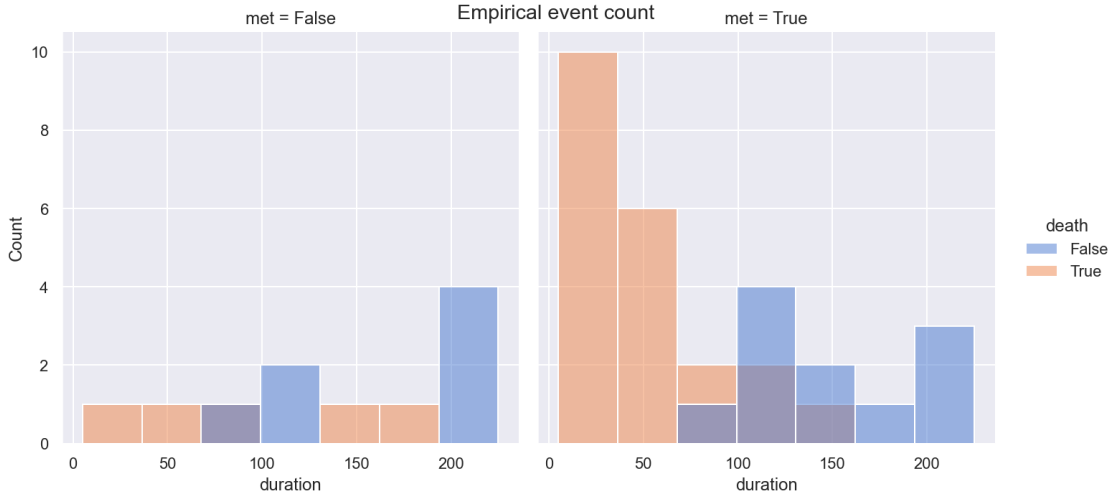


Observe:

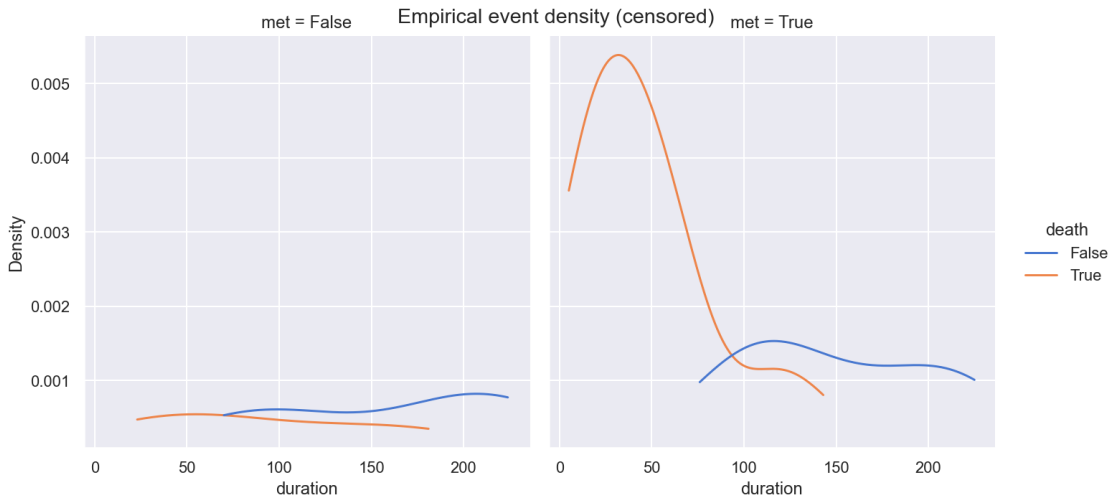
- Again we see a plausible correlation between `death` and `met`, to be investigated properly via a range of models
- Again we see right-censoring, where patients have not experienced the death event during the observation period

5.1.5 1.1.5 Empirical Event Density

```
g = sns.displot(x='duration', hue='death', data=df, col='met', kind='hist')
_ = g.fig.suptitle('Empirical event count')
```



```
g = sns.displot(x='duration', hue='death', data=df, col='met', kind='kde',␣
↪cut=0)
_ = g.fig.suptitle('Empirical event density (censored)')
```



5.2 1.2 The Kaplan Meier Estimator (empirical Survival function)

We can manually calculate a Kaplan Meier survival curve on our censored dataset to motivate further understanding.

This empirical calculation (not really a model per-se) is a univariate max-likelihood estimate of the survival function $\hat{S}(t)$

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

where:

- d is the count of 'death' events
- n is the count of individuals at risk at timestep i

The cumulative product gives us a non-increasing curve where we can read off, at any timestep during the study, the estimated probability of survival from the start to that timestep. We can also compute the estimated survival time or median survival time (half-life) as shown above.

Calc manually

```
# TODO return to this to clean it up
dfg = (df.reset_index().groupby(['duration', 'death']).agg(nobs=pd.
↪NamedAgg(column='pid', aggfunc='count'))\
```

```
        .unstack('death').droplevel(0, axis=1).rename(columns={0:␣
 ↪'censored_csum', 1: 'died_csum'})\
        .cumsum(axis=0).fillna(method='ffill').fillna(0))
dfg.columns.name = None
dfg['survival'] = (len(df) - dfg['died_csum']) / len(df)
dfo = pd.DataFrame.from_dict([0, 0, 1]).T
dfo.columns = dfg.columns
dfg = pd.concat((dfo, dfg), axis=0, ignore_index=False)
dfp = dfg.reindex(np.arange(dfg.index.max() + 1)).fillna(method='ffill').
 ↪reset_index().rename(columns={'index': 'duration'})
eda.display_ht(dfp)
```

	duration	censored_csum	died_csum	survival
0	0	0.0	0.0	1.00
1	1	0.0	0.0	1.00
2	2	0.0	0.0	1.00
223	223	16.0	26.0	0.41
224	224	17.0	26.0	0.41
225	225	18.0	26.0	0.41

'Shape: (226, 4), Memsize 0.0 MB'

Calc using `lifelines.KaplanMeierFitter.survival_function_`

```
km = sa.KaplanMeierFitter(alpha=0.05)
km.fit(durations=df['duration'], event_observed=df['death'])
```

<lifelines.KaplanMeierFitter:"KM_estimate", fitted with 44 total observations, 18 right-censored observations>

```
eda.display_ht(km.event_table)
```

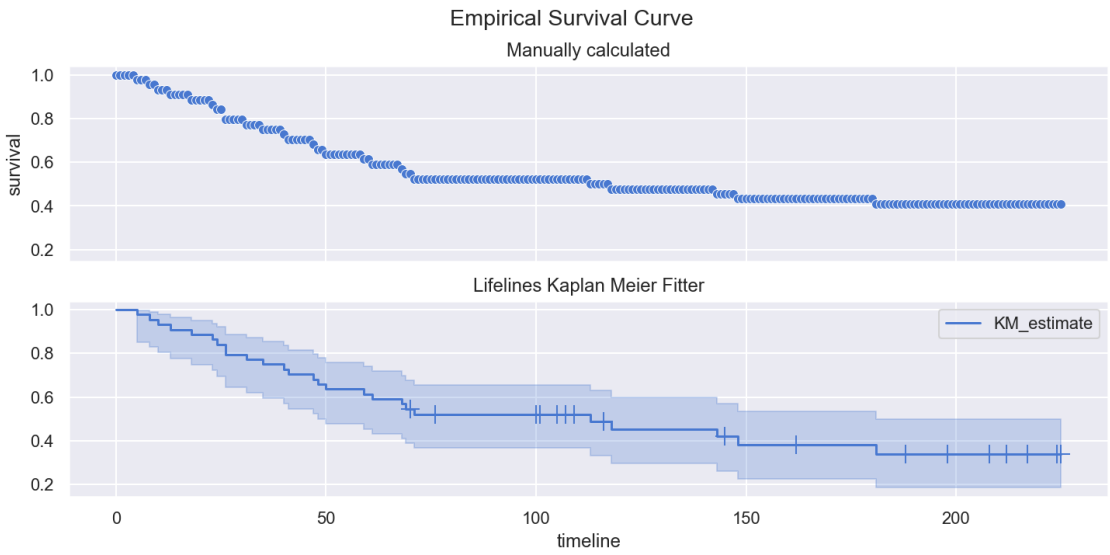
event_at	removed	observed	censored	entrance	at_risk
0.0	0	0	0	44	44
5.0	1	1	0	0	44
8.0	1	1	0	0	43
217.0	1	0	1	0	3
224.0	1	0	1	0	2
225.0	1	0	1	0	1

'Shape: (43, 5), Memsize 0.0 MB'

5.2.1 1.2.1 Plot Survival function

```
f, axs = plt.subplots(2, 1, figsize=(12, 5), sharey=True, sharex=True)
ax0 = sns.scatterplot(x='duration', y='survival', data=dfp, ax=axs[0])  # ,␣
 ↪color='C1')
ax1 = km.plot_survival_function(ax=axs[1], show_censors=True)
```

```
_ = ax0.set_title('Manually calculated')
_ = ax1.set_title('Lifelines Kaplan Meier Fitter')
_ = f.suptitle('Empirical Survival Curve')
```



Observe:

- Looks reasonable
- Same results, though `lifelines` much simpler to use than manual calculations
- Note we're showing CI on KM fit but KM is deterministic, for details see
- here, and here for more

5.2.2 1.2.2 Calc median and mean survival times

NOTE:

- Median is simply where the survival function crosses 50% (we can't call this "half-life" since KM is non-parametric)
- Mean is the area under the survival curve, which can be infinite in the case of censoring, so we use a non-parametric approach

```
med = km.median_survival_time_
mn = np.trapz(y=km.survival_function_.values.flatten(), x=km.survival_function_.
  ↪index)
print(f'med: {med:.1f}\nmean: {mn:.1f}')
```

```
med: 113.0
mean: 118.2
```

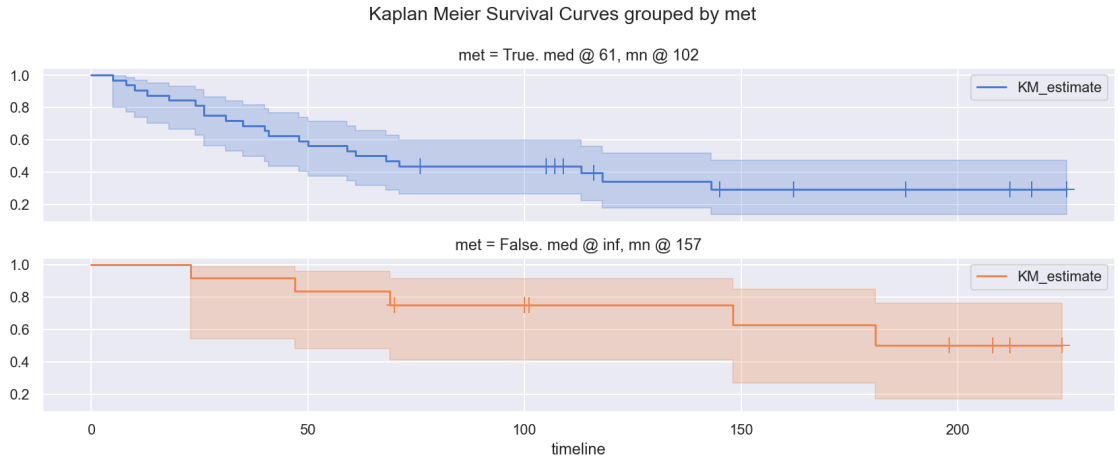
Observe:

- Median and mean quite close (for this overall, non-stratified dataset)

- More stable to take the med $\sim$ 113 days, but more transferable to take the mean $\sim$ 118 days
- Again we can't call this "half-life" since KM is non-parametric

5.2.3 1.2.3 Kaplan Meier Survival Curve grouped by `met`

```
def get_km(g):
    """Convenience get and fit km per group"""
    km = sa.KaplanMeierFitter(alpha=0.05)
    km.fit(durations=g['duration'], event_observed=g['death'])
    return km


kmf = df.groupby('met').apply(get_km, include_groups=False)
t_med_km = kmf[True].median_survival_time_
t_mn_km = np.trapz(y=kmf[True].survival_function_.values.flatten(),
                   x=kmf[True].survival_function_.index)
f_med_km = kmf[False].median_survival_time_
f_mn_km = np.trapz(y=kmf[False].survival_function_.values.flatten(),
                   x=kmf[False].survival_function_.index)

f, axs = plt.subplots(2, 1, figsize=(12, 5), sharey=True, sharex=True)
ax0 = kmf[True].plot_survival_function(ax=axs[0], show_censors=True)
ax1 = kmf[False].plot_survival_function(ax=axs[1], show_censors=True,␣
 ↪color='C1')
_ = ax0.set_title(f'met = True. med @ {t_med_km:.0f}' + f', mn @ {t_mn_km:.0f}')
_ = ax1.set_title(f'met = False. med @ {f_med_km:.0f}' + f', mn @ {f_mn_km:.
 ↪0f}')
_ = f.suptitle('Kaplan Meier Survival Curves grouped by met')
_ = f.tight_layout()
```



Observe:

- Larger rate of decay (shorter survival) for `met=True` (which we saw in plot 1.2.1 above)
- For `met=True`
  - Median survival @ 61
  - Mean (expected) survival @ 102 very different, due to high variance (low counts) in the small data
- For `met=False`
  - Median survival not observed, a weakness in the approach
  - Mean (expected) survival @ 157 again with high variance (low counts) in the small data

5.3 1.3 The Nelson Aalen Estimator (empirical Cumulative Hazard function)

Given the fundamental abstractions noted above, we can consider a close alternative estimator to the Kaplan-Meier method: the **Nelson-Aalen model**. This is also univariate, and estimates the **cumulative hazard function** $\Lambda(t) = -logS(t)$, and is more stable since it has a summation form:

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

where again:

- d is the count of 'death' events
- n is the count of individuals at risk at timestep i

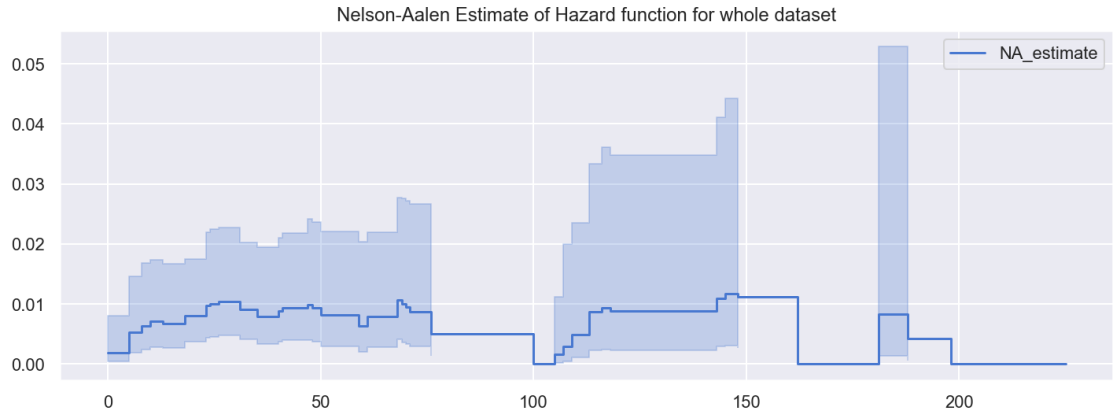
NOTE we will use our real-world censored dataset again

```
na = sa.NelsonAalenFitter(alpha=0.05, nelson_aalen_smoothing=False)
na.fit(durations=df['duration'], event_observed=df['death'])
```

```
<lifelines.NelsonAalenFitter:"NA_estimate", fitted with 44 total observations,
18 right-censored observations>
```

Now we can plot the hazard function across time

```
ax = na.plot_hazard(bandwidth=10)
_ = ax.set_title('Nelson-Aalen Estimate of Hazard function for whole dataset')
```

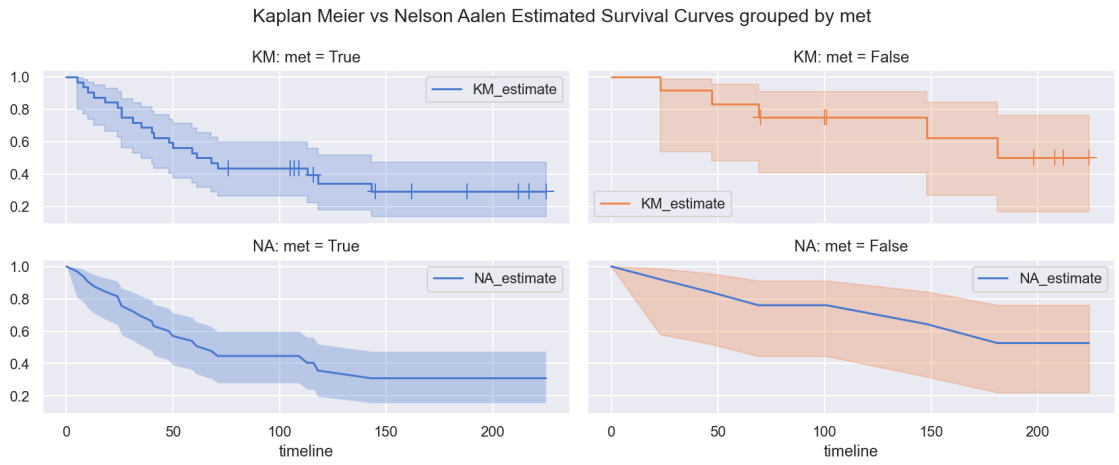


Observe:

- The hazard function $\lambda(t)$ is instantaneous, non-monotonic, and bounded $[0, \infty)$ so it's a convenient measure to estimate
- The cumulative hazard function $\Lambda(t)$ is more stable

  lets compare with the KM estimator output on the subgrouped data

```
def get_na(g):
    """Convenice get and fit na per group"""
    na = sa.NelsonAalenFitter(alpha=0.05, nelson_aalen_smoothing=False)
    na.fit(durations=g['duration'], event_observed=g['death'])
    return na


naf = df.groupby('met').apply(get_na, include_groups=False)

f, axs = plt.subplots(2, 2, figsize=(12, 5), sharey=True, sharex=True)
ax00 = kmf[True].plot_survival_function(ax=axs[0][0], show_censors=True)
ax01 = kmf[False].plot_survival_function(ax=axs[0][1], show_censors=True,␣
 ↪color='C1')
_ = ax00.set_title('KM: met = True')
_ = ax01.set_title('KM: met = False')

Lam = naf[True].cumulative_hazard_
Lam_ci = naf[True].confidence_interval_
s = np.exp(-Lam)
s_ci = np.exp(-Lam_ci)
ax10 = sns.lineplot(s, ax=axs[1][0])
_ = ax10.fill_between(x=s_ci.index, y1=s_ci.iloc[:, 0], y2=s_ci.iloc[:, 1],␣
 ↪alpha=0.3)

Lam = naf[False].cumulative_hazard_
Lam_ci = naf[False].confidence_interval_
s = np.exp(-Lam)
s_ci = np.exp(-Lam_ci)
ax11 = sns.lineplot(s, ax=axs[1][1], color='C1')
_ = ax11.fill_between(x=s_ci.index, y1=s_ci.iloc[:, 0], y2=s_ci.iloc[:, 1],␣
 ↪alpha=0.3, color='C1')

_ = ax10.set_title('NA: met = True')
_ = ax11.set_title('NA: met = False')
_ = f.suptitle('Kaplan Meier vs Nelson Aalen Estimated Survival Curves grouped␣
 ↪by met')
_ = f.tight_layout()
```



Observe:

- Very similar-looking survival curve estimates, and confidence intervals too
- NA gives interpolation between points
- NA still has most of the issues of KM: non-parametric summary of data, rather than a principled model estimator, but the hazard function can be a useful measure.

---

6 2. Fundamental General Abstrations

So far, we've only considered an empirical survival function, which by definition occupies the range $[0, 1]$ and can only monotonically decrease.

We could make a further assumption that the event (death) *must* happen within our observation period, i.e. the observation period is infinite and there is no censoring. This is an unrealistic assumption in practice, and we will quickly return to considering censored data, but allows further derivations.

6.1 2.1 The Core Functions

6.1.1 2.1.1 Survival function $S(t)$, cumulative hazard function $\Lambda(t)$, hazard function $\lambda(t)$

Due to these properties, we can alteratively state the **survival function** $S(t)$ as a log-transformation of a monotonically increasing value (range $[0, \infty)$) that we call the **cumulative hazard function** $\Lambda(t)$, itself a measure of the total instantaneous **hazard function** $\lambda(t)$ up to time t:

$$\begin{aligned}\log S(t) &= -\Lambda(t) \\ &= -\int_0^t dt\ \lambda(t)\end{aligned}$$

This approach of estimating a hazard function is the basis for many more methods, including parametric regression models e.g. the Exponential decay model and CoxPH discussed later.

6.1.2 2.1.2 Event Density function $\pi(t)$

We can also relate the **event density function** $\pi(t)$ aka time-to-event using the cumulative value $\Pi(t)$ which quantifies the probability that the event (death) will occur anytime **before** time t …

$$\Pi(t) = \int_0^t dt\ \pi(t)$$

… and the complementary value $S(t)$ which quantifies the probability that the event (death) will occur anytime **after** time t, or equivalently the probability that the event (death) **will not occur anytime before** time t, aka that the item **survives until** time t:

$$\begin{aligned}S(t) &= \int_t^\infty dt\ \pi(t) \\ &= 1 - \Pi(t)\end{aligned}$$

So we can work backwards from the survival function $S(t)$ to the event density $\pi(t)$ via differentiation:

$$\begin{aligned}\pi(t) &= -\frac{d}{dt}S(t) \\ &= -\frac{d}{dt}\exp\big(-\int_0^t dt\ \lambda(t)\big) \\ &= \frac{d}{dt}\int_0^t dt\lambda(t)\cdot\exp\big(-\int_0^t dt\ \lambda(t)\big) \\ &= \lambda(t)\cdot\exp\big(-\int_0^t dt\ \lambda(t)\big) \\ &= \lambda(t)\cdot\exp\big(-\Lambda(t)\big) \\ &= \lambda(t)\cdot S(t)\end{aligned}$$

… and there's the hazard function $\lambda(t)$ again.

6.1.3 2.1.3 Expected time-to-event $\mathbb{E}_t$

Another useful estimate is the expected duration of survival $\mathbb{E}_t$. This is the area under the survival curve, which we can estimate via integration, or if timesteps are equal, by summing and scaling:

$$\mathbb{E}_t \sim \int_0^\infty dt\ \pi(t)\ t$$
$$\sim \int_0^\infty dt\ S(t)$$

6.1.4 2.1.4 Event times t

Finally, note that given a Survival function (whether parametric or non-parametric), we can use a variant of the inverse cumulative distribution function (aka quantile function) to produce event times t by passing a uniform distribution through the **inverted survival function** S^{-1}. This comes in handy later when we specify our Bayesian models to produce prior and posterior predictive observations.

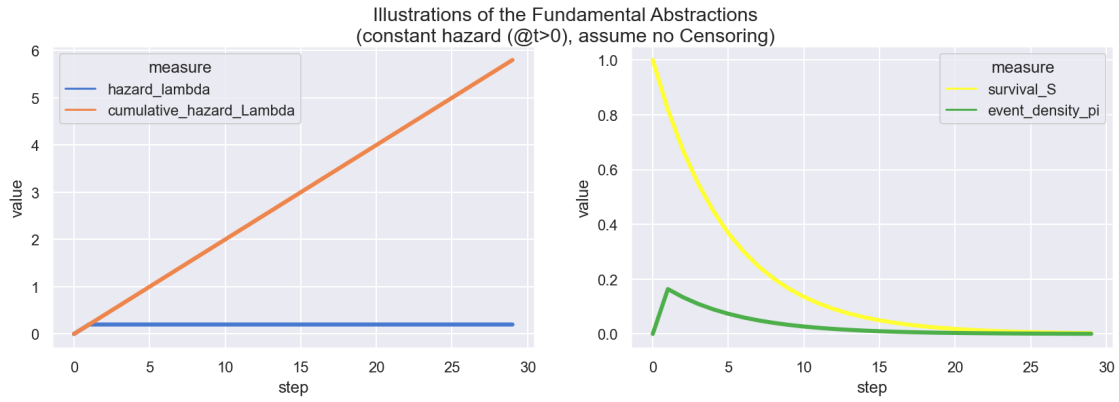
$$p \sim \text{Uniform}(0, 1)$$
$$t \sim S^{-1}(p)$$

Let's plot all the above to aid interpretation

6.2 2.2 Illustrative Plots of fundamental abstractions (ignoring censoring)

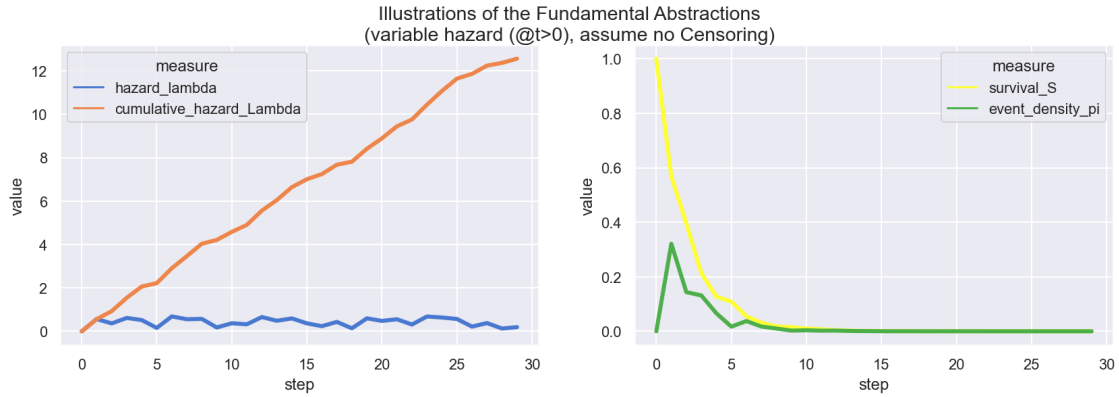
6.2.1 2.2.1 Constant hazard λ

```
steps = 30
lam = np.concatenate((np.zeros(1), np.ones(steps - 1) * 0.2))
dfi = pd.DataFrame({'hazard_lambda': lam}, index=np.arange(steps))
dfi.index.name = 'step'
dfi['cumulative_hazard_Lambda'] = dfi['hazard_lambda'].cumsum()
dfi['survival_S'] = np.exp(-dfi['cumulative_hazard_Lambda'])
dfi['event_density_pi'] = dfi[['hazard_lambda', 'survival_S']].prod(axis=1)
# eda.display_ht(dfi)
dfm0 = pd.melt(dfi[['hazard_lambda', 'cumulative_hazard_Lambda']].
 ↪reset_index(),id_vars='step', var_name='measure')
dfm1 = pd.melt(dfi[['survival_S', 'event_density_pi']].reset_index(),␣
 ↪id_vars='step', var_name='measure')
f, axs = plt.subplots(1, 2, figsize=(14, 4), sharex=True, sharey=False)
kws = dict(x='step', y='value', hue='measure', lw=3)
ax0 = sns.lineplot(**kws, data=dfm0, ax=axs[0])
ax1 = sns.lineplot(**kws, data=dfm1, ax=axs[1], palette='Set1_r')
_ = f.suptitle('Illustrations of the Fundamental Abstractions\n(constant hazard␣
 ↪(@t>0), assume no Censoring)')
```



6.2.2 2.2.2 Varying hazard λ (here, randomised)

```
steps = 30
lam = lam = np.concatenate((np.zeros(1), RNG.uniform(low=0.1, high=0.7,␣
 ↪size=steps - 1)))
dfi = pd.DataFrame({'hazard_lambda': lam}, index=np.arange(steps))
dfi.index.name = 'step'
dfi['cumulative_hazard_Lambda'] = dfi['hazard_lambda'].cumsum()
dfi['survival_S'] = np.exp(-dfi['cumulative_hazard_Lambda'])
dfi['event_density_pi'] = dfi[['hazard_lambda', 'survival_S']].prod(axis=1)
# eda.display_ht(dfi)
dfm0 = pd.melt(dfi[['hazard_lambda', 'cumulative_hazard_Lambda']].
 ↪reset_index(), id_vars='step', var_name='measure')
dfm1 = pd.melt(dfi[['survival_S', 'event_density_pi']].reset_index(),␣
 ↪id_vars='step', var_name='measure')
f, axs = plt.subplots(1, 2, figsize=(14, 4), sharex=True, sharey=False)
kws = dict(x='step', y='value', hue='measure', lw=3)
ax0 = sns.lineplot(**kws, data=dfm0, ax=axs[0])
ax1 = sns.lineplot(**kws, data=dfm1, ax=axs[1], palette='Set1_r')
_ = f.suptitle('Illustrations of the Fundamental Abstractions\n(variable hazard␣
 ↪(@t>0), assume no Censoring)')
```



6.3 2.3 Estimating $\log \mathcal{L}\pi(t)$ vs actual observed data (durations)

How shall we evidence a survival model against actual observations? We choose to minimise the log-likelihood of the event density function $\pi(t)$ vs the actual observed `duration` aka time-to-event:

$$\begin{aligned} \pi(t) &= \lambda(t) \cdot S(t) \\ &= \lambda(t) \cdot \exp\big(-\Lambda(t)\big) \end{aligned}$$

... so:

$$\log \mathcal{L}\ \pi(t) = \sum_i \log \lambda(t)_i - \Lambda(t)_i$$

This is a general form, and still doesn't affect the actual non/semi/parametric form of the hazard function.

6.4 2.4 Incorporating Censoring

Pragmatically, we can only observe a group of individuals for a study period, so it's possible for individuals to not experience the event of interest. In that case all we know is that their contribution to the estimate of event density π is their survival function:

$$\mathcal{L}\ \pi(t)_i = S(t)_i$$

Referring to 2.1.2 and 2.3 above, we see it's possible to incorporate the event (e.g. death) and the non-event into the same relationship by just using an indicator value $d_i \in \{0, 1\}$ which operates on time step t. Recall this is a general relationship, independent of the non/semi/parametric form of λ

$$
\begin{aligned}
\mathcal{L} &= \prod_i \mathcal{L}_i \\
&= \prod_i \pi(t)_i \\
&= \prod_i \lambda(t)_i^{d_i} S(t)_i \\
&= \prod_i \lambda(t)_i^{d_i} \ \exp\left(-\Lambda(t)_i\right)
\end{aligned}
$$

... so:

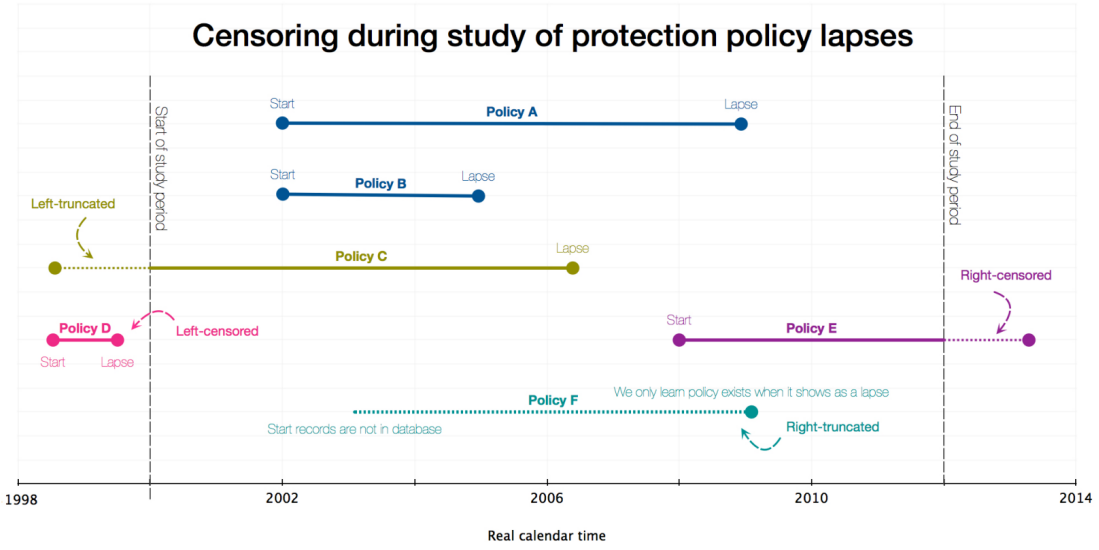
$$
\log \mathcal{L}(\pi(t)) = \sum_i d_i \log \lambda(t)_i - \Lambda(t)_i
$$

See **WWS507 Section 7.2.2 The Likelihood Function for Censored Data** for details and derivations, and Betancourt's article for a thorough set of worked examples w.r.t. interval censoring

6.4.1 2.4.1 Illustration of kinds of Censoring

The following illustration is for policy lapses, but has the same behaviour for whatever the subject. Items under observation have a start event and and end event.

```
f = staticimgio.read(fn='censoring', extension='.jpeg', figsize=(16, 6))
```



7 Notes

```
%load_ext watermark
%watermark -a "jonathan.sedar@oreum.io" -udtmv -iv
```

```
Author: jonathan.sedar@oreum.io

Last updated: 2024-12-04 16:48:49

Python implementation: CPython
Python version       : 3.11.10
IPython version      : 8.29.0

Compiler    : Clang 17.0.6
OS          : Darwin
Release     : 23.6.0
Machine     : arm64
Processor   : arm
CPU cores   : 8
Architecture: 64bit

pyprojroot : 0.3.0
scipy      : 1.14.1
sys        : 3.11.10 | packaged by conda-forge | (main, Oct 16 2024, 01:26:25)
[Clang 17.0.6 ]
pandas     : 2.2.3
matplotlib : 3.9.2
seaborn    : 0.12.2
statsmodels: 0.14.4
numpy      : 1.26.4
oreum_core : 0.9.8
lifelines  : 0.30.0
```
