The Journal of Open Source Software

lifelines: survival analysis in Python

Cameron Davidson-Pilon[1]

1 University of Waterloo

DOI: 10.21105/joss.01317

Software

- Review
- Repository
- Archive

Submitted: 06 March 2019
Published: 03 August 2019

Summary

One frustration of data scientists and statisticians is moving between programming languages to complete projects. The most common two are R and Python. For example, a survival analysis model may be fit using R's *survival-package* (Terry M Therneau, 2015) or *flexsurv* (Christopher Jackson, 2016), but needs to be deployed into a Python system. Previously, this may have meant using Python libraries to call out to R (still shuffling between two languages, but now abstracted), or translating the fitted model to Python (likely to introduce bugs). Libraries like Patsy (Nathaniel J. Smith et al., 2018) and Statsmodels (Skipper Seabold et al., 2017) have helped data scientists and statisticians work in solely in Python. *lifelines* extends the toolbox of data scientists so they can perform common survival analysis tasks in Python. Its value comes from its intuitive and well documented API, its flexibility in modeling novel hazard functions, and its easy deployment in production systems & research stations along side other Python libraries. The internals of *lifelines* uses some novel approaches to survival analysis algorithms like automatic differentiation and meta-algorithms. We present high-level descriptions of these novel approaches next.

One goal of *lifelines* is to be pure Python so as to make installation and maintenance simple. This can be at odds with users' desire for high-performance model fitting. Though Python is becoming more and more performant, datasets are getting larger and larger at a faster rate. Internally, *lifelines* uses some interesting tricks to improve performance. These approaches can be applied to other Python libraries. For example, the Cox proportional hazard model with Efron's tie-handling method has a complicated partial-likelihood (Wikipedia contributors", 2019):

$$\ell(\beta) = \sum_j \left(\sum_{i \in H_j} X_i \cdot \beta - \sum_{\ell=0}^{m-1} \log \left(\sum_{i: Y_i \geq t_j} \theta_i - \frac{\ell}{m} \sum_{i \in H_j} \theta_i \right) \right),$$

where $\theta_i = \exp\left(X_i \cdot \beta\right)$, and the Hessian matrix is:

$$\ell''(\beta) = -\sum_j \sum_{\ell=0}^{m-1} \left(\frac{\sum_{i: Y_i \geq t_j} \theta_i X_i X_i' - \frac{\ell}{m} \sum_{i \in H_j} \theta_i X_i X_i'}{\phi_{j,\ell,m}} - \frac{Z_{j,\ell,m} Z_{j,\ell,m}'}{\phi_{j,\ell,m}^2} \right),$$

where

$$\phi_{j,\ell,m} = \sum_{i: Y_i \geq t_j} \theta_i - \frac{\ell}{m} \sum_{i \in H_j} \theta_i$$

$$Z_{j,\ell,m} = \sum_{i: Y_i \geq t_j} \theta_i X_i - \frac{\ell}{m} \sum_{i \in H_j} \theta_i X_i.$$

These could be implemented in Python using, for example, Python's native `for` loops. However, this would be too slow. We would like to use NumPy, which offers vectorized operations. However, even with NumPy, there are still some Python `for` loops. *lifelines* implements these equations using NumPy's `einsum` function, which is used to express tensor products in Einstein summation notation. The result is that the tensor products, which are just `for` loops, are pushed down to as close to the C layer as possible. From internal tests, using `einsum` resulted in a 4x speed improvement.

Another optimization in the *lifelines*' implementation of Cox proportional hazard model is using a meta-algorithm to pick the most performant algorithm at runtime. There are two algorithms that can be used to compute the partial likelihood (and its gradient and Hessian). One algorithm is faster when there is a high cardinality of times in the dataset (low count of ties), and the other is faster when there low cardinality of times (high count of ties). There is not a simple heuristic of when one is faster than the other, at it depends on other factors like the size of the dataset and the average count of data points per time. To overcome this, I generated hundreds of artificial datasets of varying size, varying cardinality of times, and varying average count of data points per time. After running both algorithms against the datasets and recording their durations, I fitted a linear regression model to predict the ratio of duration. This model is accurate (R-squared over 80%), easy to implement, and importantly can predict very fast. Thus, at runtime, we compute summary values for the dataset and let the linear model predict which algorithm will be faster. By choosing the appropriate algorithm, we achieve an up-to 3x performance improvement on some datasets. When other performance improvements are made to either algorithm, we rerun the dataset generation and model training to get a new linear model for prediction.

A non-performance innovation in *lifelines* is the incredible flexibility for user-defined parametric cumulative hazards. This allows the user to specify precise parametric models and perform inference and statistical tests on them. This generalization of parametric models, which previously was confined to well-known probability distributions, is possible due to the automatic differentiation engine, Autograd, and the fact that any probability distribution can be defined by a cumulative hazard function. *lifelines* has implemented the log-likelihood for an arbitrary cumulative hazard, $H(t|\theta)$, including possible censoring and left-truncation:

$$\log l(\theta|t,c,s) = \sum_{i:c_i=1} (\log h(t_i) - H(t_i)) - \sum_{i:c_i=0} H(t_i) + \sum_i H(s_i)$$

where t_i are the observed or censored times of subject i, c_i is 0 if the subject was censored, s_i is time the subject entered the study, and $h(t)$ is the hazard which is the derivative of the cumulative hazard with respect to time. We use automatic differentiation to compute $h(t)$ and the gradient and Hessian with respect to the unknown parameters θ. The user only needs to specify a cumulative hazard (in Python code). *lifelines* will invoke SciPy's `minimize` with the computed derivatives and return the maximum likelihood estimators of the model, along with standard errors, p-values, confidence intervals, etc. An example user defined cumulative hazard is below:

```
from lifelines.fitters import ParametericUnivariateFitter

class ThreeParamHazardFitter(ParametericUnivariateFitter):

    _fitted_parameter_names = ['alpha_', 'beta_', 'gamma_']
    _bounds = [(0, None), (75, None), (0, None)]

    # this is the only function we need to define. It always takes two arguments:
    #   params: an iterable that unpacks the parameters you'll need in the order o
    #   times: a numpy vector of times that will be passed in by the optimizer
    def _cumulative_hazard(self, params, times):
        a, b, c = params
        return a / (b - times) ** c
```

Some more examples of user-defined cumulative hazards are in the main documentation.

Acknowledgments

I'd like to acknowledge all the researchers in survival analysis, specifically Terry Therneau. I'd also like to acknowledge the contributers to the *lifelines* projects.

References

Christopher Jackson. (2016). flexsurv: A platform for parametric survival modeling in R. *Journal of Statistical Software*, *70*(8), 1–33. doi:10.18637/jss.v070.i08

Nathaniel J. Smith et al. (2018, October). Pydata/patsy: V0.5.1. doi:10.5281/zenodo.1472929

Skipper Seabold et al. (2017, February). Statsmodels/statsmodels: Version 0.8.0 release. doi:10.5281/zenodo.275519

Terry M Therneau. (2015). *A package for survival analysis in s*. Retrieved from https://CRAN.R-project.org/package=survival

Wikipedia contributors". (2019). Proportional hazards model — Wikipedia, the free encyclopedia. Retrieved from https://en.wikipedia.org/w/index.php?title=Proportional_hazards_model&oldid=885553852