PROJECT :
PACKAGE THE PIECEWISE CONSTANT BASELINE HAZARD

INDEX

Abstract

This project had as objective to discover low-level interfacing with Python for scientific computing and learn about likelihood computing with a prior for asymptotic variance estimation by the delta method. The piecewise constant hazard model with accelerated time values or proportional hazard has been chosen for this project because this model is flexible and not implemented in any open-source project. When writing those lines, it seems the only software able to compute this model is SAS. R has a package to fit this model but without regression.

This document will present Python with Cython, the computation of the likelihood of the model and how to use it.

Tools

What is Python?

Python is a high-level and object-oriented programming language. Often considered as slow, the most important thing about Python is it's focus on readability. The rules are defined in The Zen of Python and the first of all is clear: "Beautiful is better than ugly". The clean syntax of Python allows users to focus on code and do complex things if few lines.

The weakness of Python is the GIL or "Global Interpreter Lock". This limitation exists to avoid Python to execute multiple bytecodes at the same time and creating errors in memory management. This limitation forbid any multithreading as in C, C++ or Java and limits us to use parallel Python interpreters and create proxy to share data between processes. This level of parallelism is slow and reacts differently on Windows and Unix systems. Using a proxy increases time reaction delay so accessing shared memory is a bottleneck in every case.

An implementation of a language is an execution environment where this language can be executed. The most common implementation of Python is CPython, translating Python code to C code and executing it. Other are used and deserve few lines of description:

- IronPython: Python in a C# environment
- Jython: Python code on Java virtual machine
- Pypy: a Python interpreter written in Python, very interesting and excellent benchmarks

The Python packages are very useful and is one of the biggest part of the success of Python. Compared to R, Python packages in statistics are not so many but concentrated in Scipy. General purpose packages are sometimes very useful (like strings analysis) and no as well implemented in statistical languages.

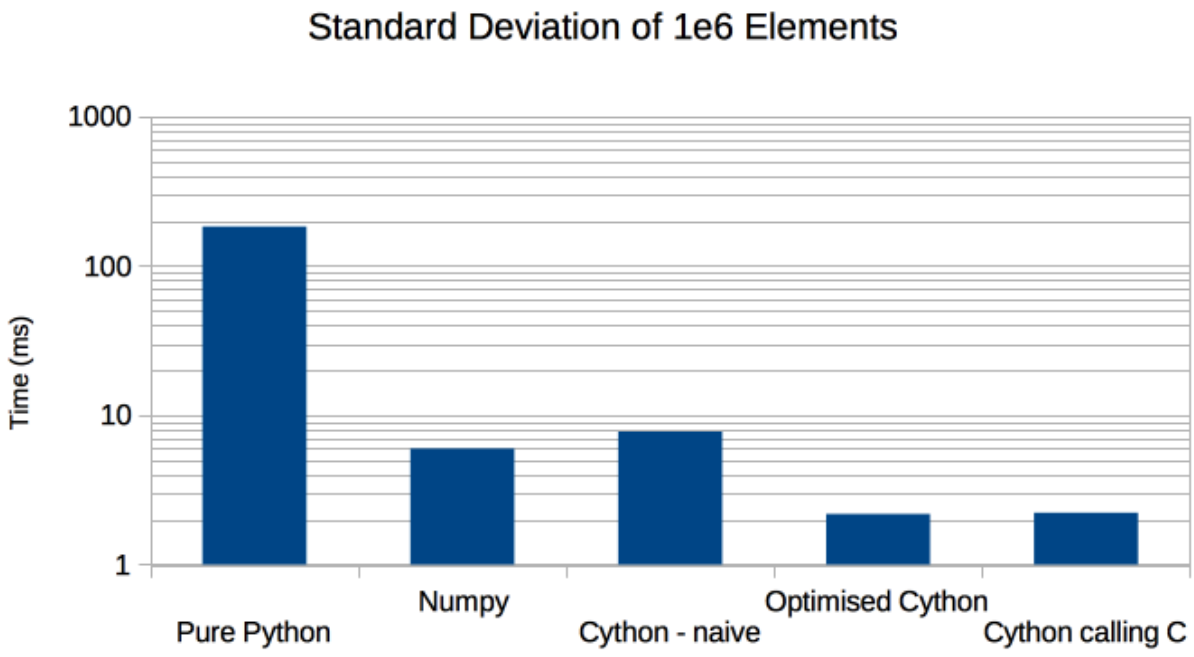
As we said, Python packages for statistics a few because concentrated mainly on three projects: Pandas, Numpy and Scipy. The first one contains a DataFrame object and is a very powerful tool to handle big tables. Those tables are build with Arrays from Numpy, dedicated to n-dimensional arrays management. Scipy contains miscellaneous tools like derivatives, integrations, statistical modeling and more.

Why is it so popular for scientific computing?

A slow language for scientific computing seems to be a non-sense but it is under-estimating the glue power of Python by using Cython. Because CPython is finally C, we can use Cython to:

- Call C and C++ functions (wrapping)
- Meta-programming C by static typing, what we will name "Cython language" even if it is not really a language but a mix between C and Python syntax.

Pandas, Numpy and Scipy (most used libraries for science) are constructed by calling directly C and FORTRAN libraries and Cython compiled sources. Libraries used are the most powerful actually existing : BLAS (Basic Linear Algebra Solver) and LAPACK (Linear Algebra PACKage). Most of matrix computation are automatically distributed on each processor available, what makes linear algebra sometimes thirty times faster than R. Some R distribution like Revolution R use those low-level routines.



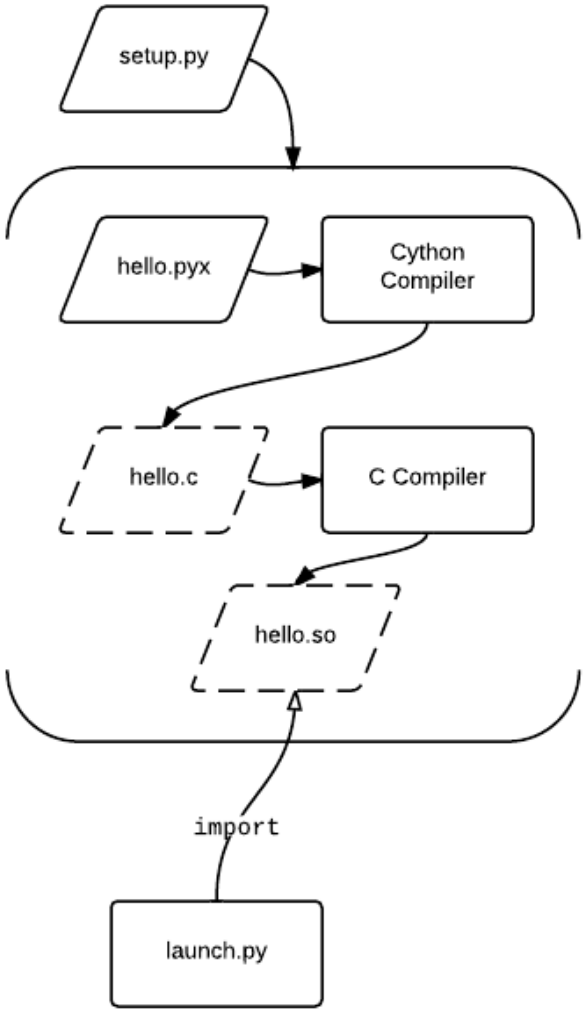
Note : "Naive Cython" means no static typing, "Optimised Cython" means all variables are statically typed. Source : http://notes-on-cython.readthedocs.io

As we can see, even code without changes have a great speed up when used with Cython.

In order to transform Python code to Cython code, we need to:

1. Create a Python script and change the standard extension .py to .pyx.
2. Write a setup.py file containing instructions and specifications for compilation
3. Run the setup.py file in Python by the command line

Then if we compile a script "example.pyx", we get a big text file "example.c" and "example.so". The first contains the C translation of our Python code and the second is a compiled file. By using the import statement, user imports the *.so file in Python and calls functions from this one. Detail of steps in this illustration:



Source : Wikipedia

So Python is very adapted to scientific computing by allowing fast computation by glue C and clean syntax.

Understanding Cython powers and weaknesses

At this point we could think Cython is the universal solution for efficient programming just by adding static typing. It is true for loops and simple functions, but not elsewhere. First, because Cython is limited to C types: it is possible to manipulate in C arrays 32-bits or 64-bit float with ease but not a custom Python object. It is also the same thing for Cython functions: if the returned value or one of the parameters of the function is not C-type transformable, it is impossible to statically type all the function and the gain is not of the same order. To take advantage of Cython, it is also important to use pointers to increase speed, but it is also less user-friendly than Python's concepts.

But if a function is totally statically typed, Cython allows to disable the GIL. If done, we can use very powerful tools to distribute the function on each core of the used computer or server.

In few words, the power of Cython can be expressed as: "If it can be done with C, it is a good idea to do it with Cython". If the problem is more complex, the best thing to do is to write a C++ extension and to call it from Cython, but user have to keep in mind maintainability and development time are not the same.
It is also more comfortable to code with Cython compared to C due to cleaner syntax, example with a Fibonacci function:

With C:
```
int Fibonacci(int n)
{
   if ( n < 2 )
      return n;
   else
      return ( Fibonacci(n-1) + Fibonacci(n-2) );
}
```

With Cython:
```
cdef int Fibonacci(int n):
    if n < 2:
        return n
    return Fibonacci(n-2) + Fibonacci(n-1)
```

With Python:
```
def Fibonacci(n):
    if n < 2:
        return n
    else :
      return Fibonacci(n-2) + Fibonacci(n-1)
```

As we can see, difference between Cython and C is just a cleaner syntax when typed.

The model

Assumptions

The piecewise constant baseline hazard is a semi parametric model assuming hazard is exponential with different parameter between each intervals previously defined by the user. It make the model very flexible and potentially adaptive to almost any duration modeling. The regression aspect is also interesting because the likelihood is the same with proportional hazard and accelerated times values. Regression holds stronger assumptions because it constraints same "global" shape for all individuals, but we could imagine characteristics with different effects.

For discrete values it is possible to cluster by evaluating one baseline for each unique possibility. As example, we could create a different baseline for each degree category. For continuous variables we could separate by intervals but it is more subjective and less accurate.

The log-likelihood

The log-likelihood of a truncated variable can be given by, with h_T the hazard function and H_T the cumulative hazard function:

$\ell = \sum_{i=1}^{N} d_i \ln h_T(y_i) + \ln S_T(y_i)$ what is equivalent to: $\ell = \sum_{i=1}^{N} d_i \ln h_T(y_i) - H_T(y_i)$

because $H_T(y_i) = -\ln(S_T(y_i))$

The Variance matrix of the likelihood estimator is given by: $\hat{\Sigma}_N = \left(-\frac{\partial^2 \ell}{\partial\theta\,\partial\theta'}(y;\hat{\theta}) \right)^{-1}$.

and the estimator normally distributed. The log likelihood of the piecewise constant hazard is given by:

$\sum_{i=1}^{N} d_i \ln(h_T(y_i)) + H_T(y_i)$ with $d_i = 1$ if the observed duration is full and

$h_T(t) = \sum_{j=1}^{M} \alpha_j h_j$ with $\alpha_j = 1$ if $c_{j-1} > t > c_j$.

Cumulative hazard is $H(t) = \sum_{j=1}^{m-1} (c_j - c_{j-1}) h_{j-1} + (t - c_m) h_m$ for $t < c_{m+1}$. There are M intervals and M-1 hazard values.

Regression implies $S_i(t) = S\left(e^{(X_i' c)} t\right)$

So log-likelihood can be written as follow: $\sum_{i=1}^{N} d_i \ln((h_T(y_i)) + X_i' c) + X_i' c\, H(t)$

Refactoring likelihood for speed gain

We will not use linear algebra to evaluate the likelihood but express it in a simple algorithm easily programmable with only C functions.

The hazard will be:

```
for i in range(M-1) :
     if y > intervals[i+1] :
          break
          return d*h[i]
```

So we iterate over intervals values until we found one higher. Then, we break the loop and return the hazard value.

The cumulative hazard is not more complex:

```
H = 0
for i in range(M-1) :
     if y > intervals[i+1] :
          H += (y - intervals[i]) * h[i]
          break
          return H
     else :
          H += (intervals[i+1] - intervals[i]) * h[i]
```

Here we increment the cumulative hazard until we found an interval higher than our duration. When done, we increment from the last interval to the observed value.

For the model without explicative variables, it was simple to constraints value to avoid negative hazards. With regression, we made the choice of a reparametrization of the model by an exponential transformation of h. The hazard and log hazard become:

```
for i in range(M-1) :
     if y > intervals[i+1] :
          break
          return d*exp(h[i]+Xc)
```

```
for i in range(M-1) :
     if y > intervals[i+1] :
          break
          return d*log(exp(h[i])+Xc)
```

And the cumulative hazard:

```
H = 0
for i in range(M-1) :
     if y > intervals[i+1] :
          H += (y - intervals[i]) * exp(h[i]+Xc)
          break
     else :
          H += (intervals[i+1] - intervals[i]) * exp(h[i]+Xc)
```

Go parallel

At this point, we can easily combine those algorithms in one just by adding the last line of the log hazard before the break statement of the cumulative hazard. We just need to apply this procedure by using cython.parallel prange on the sample. To increase speed, we can use C exponential and logarithm functions and direct memory n-dimensional buffers to access at the maximum speed to the data. Declare a memory buffer is trivial:

```
cdef double[:] = an_array
```

For a one dimensional array of c-type double values. For a higher dimension, user just need to put one more ":" separated by coma from the other. The only problem with this syntax is confusing aliases for c-types and python-types. For example a Python float object is a double in C types. Note strings are not well handled due to C lack with unicode.

Get the C functions is also very simple :

```
from libc.math cimport exp as c_exp
from libc.math cimport log
```

An important point about the time consumed by Python to access data in array to check if the given position is negative. If it is, it take this value as positive and reverse index. For example :

```
an_array[-1]
```

returns the last value of this array. The Python security also check if the element is not higher than the array index size.

We can disable those securities with the decorator @Cython.boundscheck(False). This is necessary also to allow the distributed loop to start at many points at the same time.

Without this option, multi-threading simply doesn't work. If user goes out of bounds, he uses memory he should not and the result is often a blue screen.

Writing the likelihood function with memory buffers and c-static-typing can be done as follow:

```
@cython.boundscheck(False)
cdef double LL_onepiece(double[:] data, double[:] intervals,
 double[:] values, double[:] finished) nogil:
    cdef double LL=0
    cdef int J = data.shape[0]
    cdef int I = intervals.shape[0]
    cdef int i, j
    for j in prange(J, schedule = "static", chunksize = 500) :
        for i in range(I-1) :
            if data[j] <= intervals[i+1] :
                LL += values[i] * (data[j] - intervals[i])
                LL += -log(values[i]) * finished[j]
                break
            else :
                LL += values[i] * (intervals[i+1] - intervals[i])
    return LL
```

With values as h and finished as d. We disabled the GIL at the end of the third line for all the function, what allows compiler to create pure C code. Note we used the negative likelihood because Python packages compute minimization.

Prange allows many way to distribute computations. We used the more efficient, the static. It is adapted because we can think the time to compute each chunk is approximately equal. If not, it is better to use "guided". Using value assignation is not permitted in parallel due to conflict between processes but in our case it is absolutely not a problem: we can increment the likelihood for each individual and for each interval until the break.

Here are 4 arrays used by the function. By introducing explicative variables we need to change the algorithm to accept as data not only durations but explicative variables. We will concatenate observed durations, finished and explicative in one matrix and create a vector containing the product of explicative variables and coefficients for each individual.

So because we will create a matrix, we have to think about it's use: here there will be an iteration on each row. In this case, user is strongly encouraged to use Fortran contiguous array because there are indexed by row and not by columns as C arrays. So by using a different contiguity we are able to use closer memory cells and then have a greet speed up.

If speed up is so important at this point, it is because tests returned a very important number of calls on this function when we use an important number of intervals and regression (more than 800 calls with 40 intervals and 10 explicative variables), so any little gain on each call means mportant gain when accumulated. This step is not made in the likelihood function because we can not use Python object so the simplest way to do it is out in the class we will use after for linking.

Here is the algorithm with regression, with k the number of explicative variables:

```
@cython.boundscheck(False)
cdef double LL_reg_onepiece(double[:,:] data, double[:] intervals, double[:]
values, int k, double[:] Xc) nogil:
    """parallel low-level likelihood func of the piecewise constant hazard model
with accelerated time values or proportionnal hazard (same likelihood)"""
    cdef double LL = 0
    cdef int J = data.shape[0]
    cdef int I = intervals.shape[0]
    cdef double[:] h_hat = values[:I-1]
    cdef double[:] c = values[I-1:]
    cdef int i, j, l
    for j in prange(J, schedule = "static", chunksize = 500) :
        for l in range(k) :
            Xc[j] += c[l]*data[j, 2+l]
        for i in range(I-1) :
            if data[j, 0] <= intervals[i+1] :
                LL += (c_exp(h_hat[i]+Xc[j])) * (data[j, 0] - intervals[i])
                LL += -(log(c_exp(h_hat[i]))+Xc[j]) * data[j, 1]
                break
            else :
                LL += c_exp(h_hat[i] + Xc[j])* (intervals[i+1] - intervals[i])
    return LL
```

So the structure of the two dimensional data matrix is as follow:

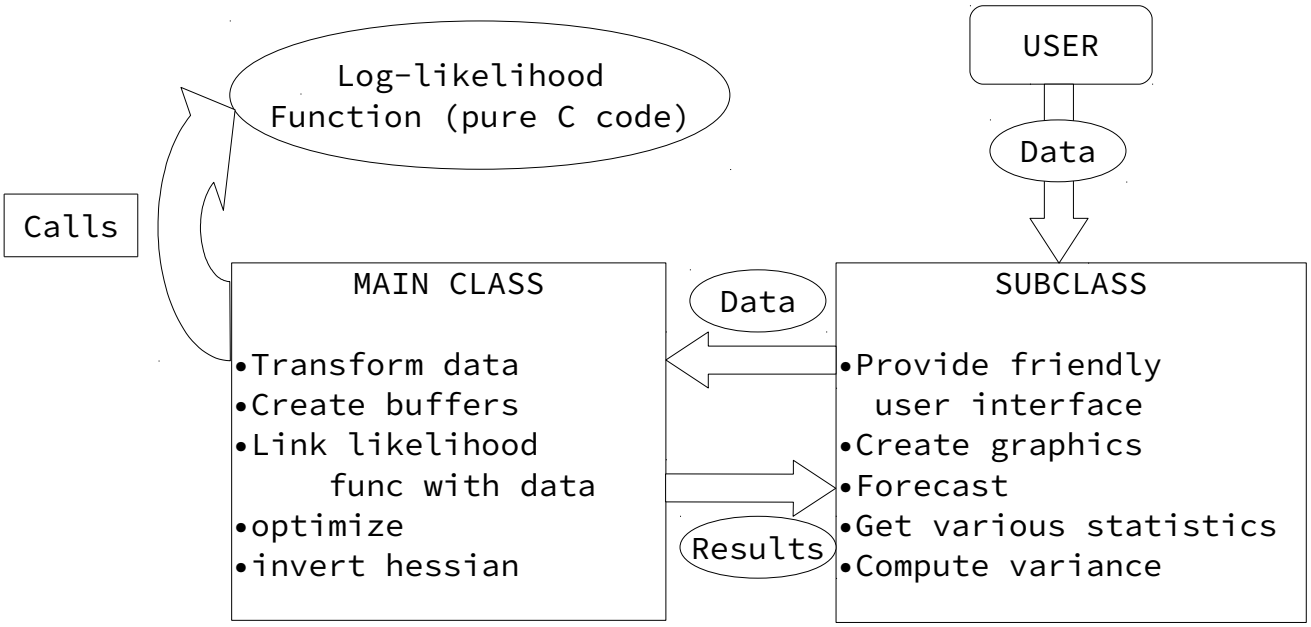
$$\begin{bmatrix} y_1 & d_1 & \widetilde{x_{1,1}} & \dots & \widetilde{x_{1,K}} \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ y_N & d_N & \widetilde{x_{N,1}} & \dots & \widetilde{x_{N,K}} \end{bmatrix}$$

With $\widetilde{x_{n,k}} = x_{n,k} - \overline{x_k}$

Use classes and subclasses to link functions and data

Now we need to connect those functions with data and make it user-friendly. We will use oriented object methods to do it. First, we need a class to transform a user-friendly input into memory buffers used in the likelihood function and optimize it. To use minimization routines, the simplest way is to create a method using only parameters to optimize and link the other to data. We will also compute the invert of the hessian matrix in this class. Note we do not use the negative hessian matrix because we already use the negative function.

Second, we need use a class to inherit of the characteristics from the first. This subclass will contain the results of the optimization and offer to user a clean interface to manipulate the model a get graphics quickly. We will create confidence intervals in this one.



We will not show the code here because it's length and there is no particular statistical interests.

Note we can use the likelihood function because we are in Cython and the file will be compiled. It is impossible to call C function from Python, first we have to create an interface with Cython. Here, we will not make the likelihood function callable from Python because user is not supposed to access it directly.

Estimation of asymptotic VAR-matrix

Because the maximum likelihood estimator asymptotic distribution is normal, we can write:

$$\sqrt{n}\left(h(B)-h(\beta)\right) \xrightarrow{D} N\left(0, \nabla h(\beta)\cdot\Sigma\cdot\nabla h(\beta)^T\right)$$

With $h(B)$ a continuous derivable function and $\nabla h(B)$ the gradient of the function. We can use this form because our gradients will never be equal to zero, otherwise we need to use a higher order of this method.

Because we constrained the likelihood without regression, we do not need to compute the delta method to get the hazard confidence intervals.

Exponential re-parametrization

With regression, we can get the asymptotic variance of each parameter by:

$$h(B)=\left(e^{h_0}, e^{h_1}, \cdots, e^{h_{M-1}}, c_1, \cdots, c_K\right)$$

So $$\nabla h(B)=\begin{bmatrix} e^{h_0} & 0 & \cdots & 0 & 0 & \cdots & 0 \\ 0 & e^{h_1} & \cdots & 0 & 0 & \cdots & 0 \\ 0 & 0 & \cdots & e^{h_{M-1}} & 0 & \cdots & 0 \\ 0 & 0 & \cdots & 0 & 1 & \cdots & 0 \\ 0 & 0 & \cdots & 0 & 0 & \cdots & 1 \end{bmatrix}$$

We just need to apply the formula $\nabla h(B)\bullet\Sigma\bullet\nabla h(B)^T$ and we get the variance of each parameter in diagonal of the result.

Estimation of the survival function asymptotic variance

As we shown previously, the survival function is the exponential of the negative cumulative hazard. It is more interesting to estimate the variance of this function than the survival itself because it will return a log-normal distribution and avoid negative values. The survival gradient is also more complex.

The cumulative hazard is $H(t)=\sum_{j=1}^{m-1}(c_j-c_{j-1})h_{j-1}+(t-c_m)h_m$ for $t<c_{(m+1)}$.

The gradient of the function is given by :

$$\nabla h(B)=\begin{bmatrix} h_0(c_1-c_0) \\ h_1(c_2-c_1) \\ \vdots \\ (t-c_m)h_m \\ 0 \\ \vdots \\ 0 \end{bmatrix} \text{ for } t<c_{(m+1)}$$

For model with explicative variables: $H_T(t)=\sum_{j=1}^{m-1}(c_j-c_{j-1})e^{h_{j-1}+\widetilde{X}_i c}+(t-c_m)e^{h_m+\widetilde{X}_i c}$

$$\nabla h(B)= \begin{matrix} e^{h_0+\widetilde{X_i}'c}(c_1-c_0) \\ e^{h_1+\widetilde{X_i}'c}(c_2-c_1) \\ \vdots \\ (t-c_m)e^{h_m+\widetilde{X_i}'c} \\ 0 \\ \vdots \\ 0 \end{matrix} \begin{matrix} \\ \\ \\ \\ \\ \\ \\ \end{matrix} \begin{matrix} \\ \\ \\ \\ \widetilde{x_{i,1}}\sum_{j=1}^{m-1}(c_j-c_{j-1})e^{h_{j-1}+\widetilde{X_i}'c}+\widetilde{x_{i,1}}(t-c_m)e^{h_m+\widetilde{X_i}'c} \\ \vdots \\ \widetilde{x_{i,K}}\sum_{j=1}^{m-1}(c_j-c_{j-1})e^{h_{j-1}+\widetilde{X_i}'c}+\widetilde{x_{i,K}}(t-c_m)e^{h_m+\widetilde{X_i}'c} \end{matrix}$$

for $t<c_{(m+1)}$. We can see the euclidean distance from the individual increases estimated variance as expected.

So we compute the value of survival confidence intervals by using:

$e^{-\hat{H}_T(t)} \rightarrow e^{N(-H_T(t),\nabla H_T(t)\Sigma\nabla H_T(t)^T)}$ for the two models.

Estimating mean error

We can get the mean by integrating the survival function with the Riemann sum:

$\widehat{M}=\sum_0^\infty (x_{i+1}-x_i)\hat{S}(x_{i+1})$ To build confidence intervals, we can show:

$\widehat{V(M)}=\sum_0^\infty (x_{i+1}-x_i)^2 V(\hat{S}(x_{i+1}))$ because adding normal distributed variables accumulates their variances. So in way to create the confidence intervals for the mean, we can simply integrate the low and high confidence intervals of the survival.

Estimating forecast error

To estimate the time remaining for an individual knowing it's actual time, we use the Bayesian formula:

$$P(t>T|t)=\frac{P(t>T)}{S(t)}$$ To estimate variance, we will use the same gradient formula used before to estimate cumulative hazard. Because the actual time is known (as t_0 in the next formula), we will make start the gradient at this value. Cumulative hazard become:

$$H(t)=(t_0-c_\alpha)h_\alpha+\sum_{j\ as\ c_j>c_\alpha}^{m-1}(c_j-c_{j-1})h_{j-1}+(t-c_m)h_m \quad \text{for} \quad t<c_{(m+1)} \quad \text{and} \quad t_0>c_\alpha$$

if t_0 and t are or the same interval $[c_{m-1};c_m]$, it becomes $H(t)=(t-t_0)h_m$

So the gradient is:

$$\nabla h(B)=\begin{bmatrix}0\\ \vdots\\ (t_0-c_\alpha)h_\alpha\\ h_{\alpha+1}(c_{\alpha+2}-c_{\alpha+1})\\ \vdots\\ (t-c_m)h_m\\ 0\\ \vdots\\ 0\end{bmatrix}$$ for t_0 and t on different intervals. Else :

$$\nabla h(B)=\begin{bmatrix}0\\ \vdots\\ 0\\ (t_0-t)h_m\\ 0\\ \vdots\\ 0\end{bmatrix}$$ for t_0 and t in $[c_{m-1};c_m]$

For model with regression, the cumulative hazard is :

$$H_T(t)=(t_0-c_\alpha)e^{h_\alpha+\widetilde{X_i}'c}+\sum_{j\,as\,c_j>c_\alpha}^{m-1}(c_j-c_{j-1})e^{h_{j-1}+\widetilde{X_i}'c}+(t-c_m)e^{h_m+\widetilde{X_i}'c}$$

$$\nabla h(B)=\begin{matrix}0\\ \vdots\\ (t_0-c_\alpha)e^{h_\alpha+\widetilde{X_i}'c}\\ \vdots\\ (c_{j+1}-c_j)e^{h_j+\widetilde{X_i}'c}\\ \vdots\\ (t-c_m)e^{h_m+\widetilde{X_i}'c}\\ 0\\ \vdots\\ 0\end{matrix}$$

$$\begin{matrix}\widetilde{x_{i,1}}\left((t_0-c_\alpha)e^{h_\alpha+\widetilde{X_i}'c}+\sum_{j\,as\,c_j>c_\alpha}^{m-1}(c_j-c_{j-1})e^{h_{j-1}+\widetilde{X_i}'c}+(t-c_m)e^{h_m+\widetilde{X_i}'c}\right)\\ \vdots\\ \widetilde{x_{i,K}}\left((t_0-c_\alpha)e^{h_\alpha+\widetilde{X_i}'c}+\sum_{j\,as\,c_j>c_\alpha}^{m-1}(c_j-c_{j-1})e^{h_{j-1}+\widetilde{X_i}'c}+(t-c_m)e^{h_m+\widetilde{X_i}'c}\right)\end{matrix}$$

and if t_0 and t are in $[c_{m-1};c_m]$, the cumulative hazard become:

$H_T(t)=(t-t_0)e^{h_m+\widetilde{X_i}'c}$ then the gradient becomes

$$\nabla h(B) = \begin{bmatrix} 0 \\ \vdots \\ (t-t_0)\, e^{h_m + \widetilde{X_i}' c} \\ \vdots \\ 0 \\ \widetilde{x_{i,1}}(t-t_0)\, e^{h_m + \widetilde{X_i}' c} \\ \vdots \\ \widetilde{x_{i,K}}(t-t_0)\, e^{h_m + \widetilde{X_i}' c} \end{bmatrix}$$

User's guide

This section describes the available API of the package.

Fit the model quickly

There are two objects : PCH_estimator and PCH_regressor. The second one provides regression.

To fit the model, user must pass as arguments

- PCH_estimator : name of the dataset, list or array or intervals, the label of complete observation dummy and observed durations in the dataset
- PCH_estimator : name of the dataset, list or array or intervals, the label of complete observation dummy, explicative variables and observed durations in the dataset

In order to have a simple use, the only difference between those two models is passing explicative variables in the second one.

Here is an example with database relative to the first job duration.

As we can see in this example of few lines :

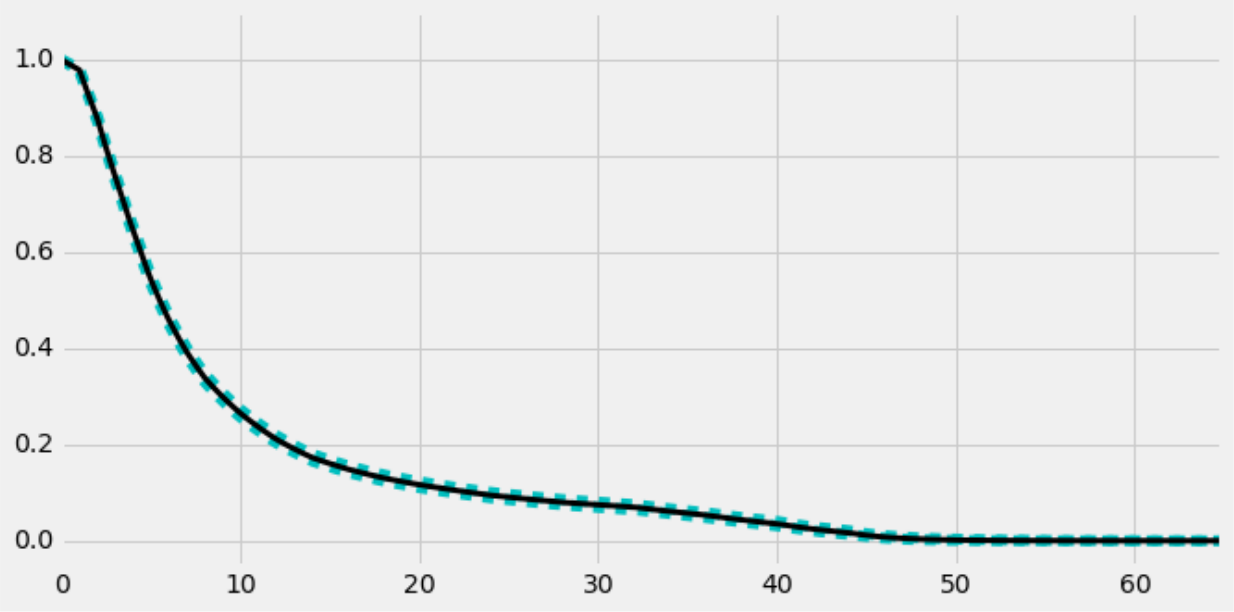
```
IPython 4.2.0 -- An enhanced Interactive Python.
?         -> Introduction and overview of IPython's features.
%quickref -> Quick reference.
help      -> Python's own help system.
object?   -> Details about 'object', use 'object??' for extra details.
%guiref   -> A brief reference about the graphical user interface.

In [1]: import PCH as P
   ...: import pandas as pd
   ...: data = pd.read_sas("/home/jb/data_analysis/base_duree.sas7bdat")
   ...: data["presence"] = 1 - data["censure"]
   ...: data["age_deb2"] = data["age_deb"]**2
   ...: #creating the reg object
   ...: test = P.PCH_regressor(data, [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 12,
   ...:                 14, 16, 18, 20, 24, 28, 32, 36, 40, 44], "presence", ["dip_sup", "dip_sec", "gen_h",
"age_deb", "age_deb2"], "duree_emp")
   ...:
```

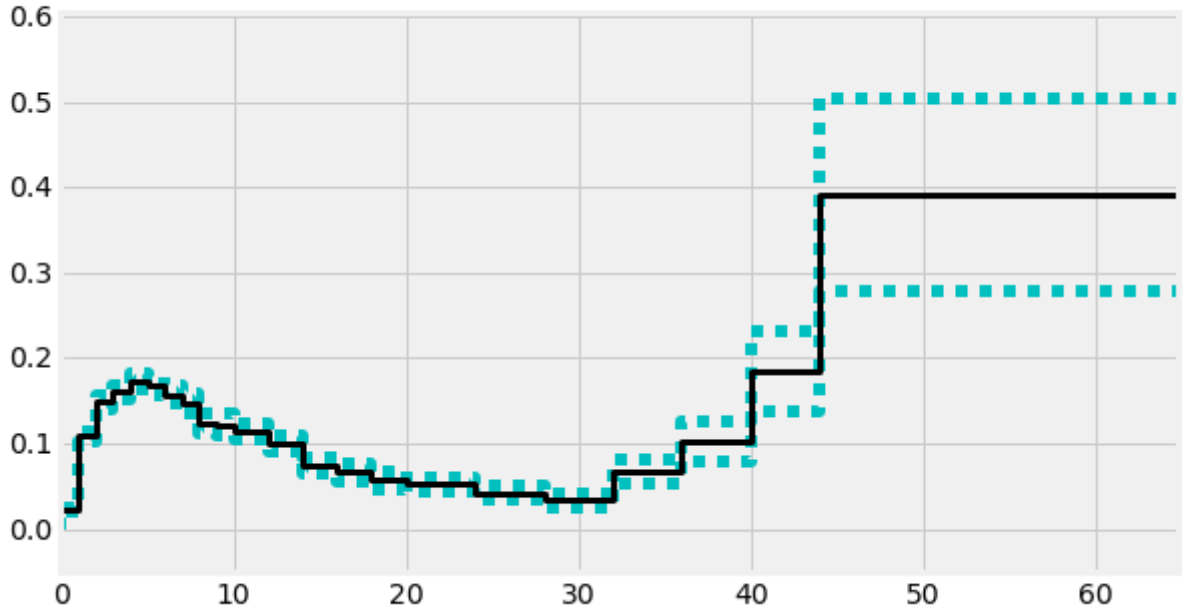
Get graphs, statistics and forecasts

When the object has been created we just need to call it's methods:

```
In [3]: test.graph_common_surv()
```

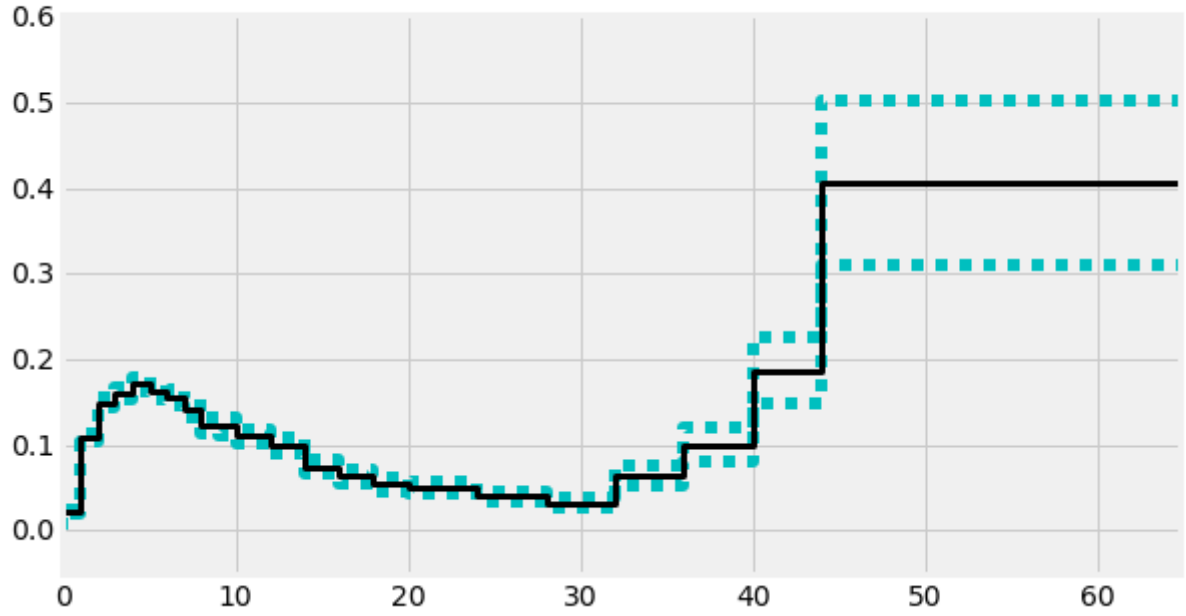


```
In [2]: test.graph_common_hazard()
```



Note hazard and survival are plot at the average point. User can change the confidence intervals alpha value by:

```
In [3]: test.graph_common_hazard(alpha = 10)
```



In some extreme cases, survival high confidence interval can be growing around the distribution tail. To avoid "resurrection" on those curves, we do not allowed any point to be higher than the previous.

For methods get_stats and forecast, if no values are provided for explicative variables, the average point values are used to compute result. User can get any quantile or a list of quantiles by passing multiple values in a list. By default, the median is computed.

```
In [5]: #get statistics of the average indivudual at beginning:
   ...: test.get_stats()
Out[5]:
[{'E': 9.2973137309399885,
  'E_down 2.5%': 9.0272282646813427,
  'E_up 2.5%': 9.5923416926055083},
 {'2.5% down': 5.4036024016010673,
  '2.5% up': 5.6197464976651101,
  '50.0%': 5.4900600400266839}]

In [6]: #get statistics of the average indivudual at beginning with multiple quantiles:
   ...: test.get_stats(stats = [0.1, 0.2, 0.3, 0.4])
Out[6]:
[{'E': 9.2973137309399885,
  'E_down 2.5%': 9.0272282646813427,
  'E_up 2.5%': 9.5923416926055083},
 {'10.0%': 23.602935290193461,
  '2.5% down': 22.478985990660441,
  '2.5% up': 25.029486324216144},
 {'2.5% down': 12.363442294863241,
  '2.5% up': 13.098332221480986,
  '20.0%': 12.70927284856571},
 {'2.5% down': 8.86190793862575,
  '2.5% up': 9.2941961307538357,
  '30.0%': 9.0780520346897919},
 {'2.5% down': 6.78692461641094,
  '2.5% up': 7.0462975316877916,
  '40.0%': 6.9166110740493654}]
```

Passing variables must be done in the same order as used when creating object :

```
In [7]: test.get_stats([1, 0, 1, 25, 25**2])
Out[7]:
[{'E': 9.8884182654721737,
  'E_down 2.5%': 8.9931686709500873,
  'E_up 2.5%': 10.990201367479878},
 {'2.5% down': 5.4900600400266839,
  '2.5% up': 6.2681787858572378,
  '50.0%': 5.8358905937291521}]
```

It is also possible to change the alpha parameter:

```
In [9]: test.get_stats([1, 0, 1, 25, 25**2], alpha = 20)
Out[9]:
[{'E': 9.8884182654721737,
  'E_down 10.0%': 9.2838962124957103,
  'E_up 10.0%': 10.580136053815918},
 {'10.0% down': 5.5765176784523014,
  '10.0% up': 6.1384923282188124,
  '50.0%': 5.8358905937291521}]
```

The forecast method is absolutely equivalent to get_stats, but it takes as parameter the actual duration of the individual in the system. It is a good way to see the power of the model:

```
In [12]: test.forecast(3, [1, 0, 1, 18, 18**2], alpha = 20)
Out[12]:
[{'E': 11.650709246983141,
  'E_down 10.0%': 11.039210253115558,
  'E_up 10.0%': 12.334864883505096},
 {'10.0% down': 6.956782713085234,
  '10.0% up': 7.5997599039615844,
  '50.0%': 7.2535414165666268}]

In [13]: test.forecast(15, [1, 0, 1, 18, 18**2], alpha = 20)
Out[13]:
[{'E': 29.503711917808559,
  'E_down 10.0%': 28.727282170791902,
  'E_up 10.0%': 30.359098267255916},
 {'10.0% down': 27.156062424969988,
  '10.0% up': 30.583673469387755,
  '50.0%': 28.690516206482592}]
```

The first output is for an individual with those characteristics and still in the system after three year. The second is for the same individual but after fifty years. Here we can see the gap between mean and median is absolutely different depending the place on the curve.

Conclusion

This project has been a challenge by combining a non-trivial model and low-level programming. Using technologies like Python and Cython is very stimulating and make user understand a lot of things about computing science in general, like contiguity of memory or static typing power and weaknesses.

By doing this project, I understood an important point about data science in general: the productivity of a data scientist is extremely impacted by the tools he uses. And more the databases are big more the gap grows. Using glue language to link low-level programs to a clean syntax language as Python is extremely efficient. It also allow a division of labor between users and creators.

My personal decision to create the most efficient extensions as possible is to learn C++ now. Using Python helped me to learn algorithmic and Cython low-level languages but there are still limitations like strings management. Memory allocation is most of the time the key between a fast algorithm and another less efficient, but the only way to control it well is to use a low level language. The only language allowing object oriented programming, high level glue (with Python or R) and this kind of control is C++.

Bibliography

"Cython" from Kurt W. Smith, O'Reilly, 2015 (very useful and clear)

"Apprenez à programmer en Python" from Vincent Le Goff, Openclassrooms, 2014

"A practiotioner's guide to robust covariance matrix estimation" from Wouter J. den Haan, Andew Levin (no date)

"On the so-called ‹‹Huber Sandwich Estimator›› and ‹‹Robust standard Error››" from David A. Freedman 2006

"Statistical Inference" from Casella, G. and Berger, R. L. 2002

"Mathematical Methods of Statistics" from Cramér, H. 1946

Annexes

Code:

```
# -*- coding: utf-8 -*-
cimport cython
from libc.math cimport log
from libc.math cimport exp as c_exp
from cython.parallel import prange
import scipy.optimize
from scipy.misc import derivative
from scipy.stats import norm
import numpy as np
import numdifftools as n
import matplotlib.pyplot as pp
from math import sqrt, exp
import pandas as pd

def converter(array) :
    """function returns buffer for fast 1-d array access"""
    cdef double[:] view
    if (type(array) != np.ndarray) & (array.dtype != "float64") :
        raise TypeError("only numpy arrays")
    else :
        view = array
        return view[:]

cdef void _forbid_res(double[:] X) nogil:
    cdef int i, I
    I = X.shape[0]
    for i in range(I-1) :
        if X[i] < X[i+1] :
            X[i+1] = X[i]

def converter_2d(matrix) :
    """function returns buffer for fast 2d array access"""
    cdef double[:, :] view
    if (type(matrix) != np.ndarray) & (matrix.dtype != "float64") :
        raise TypeError("only numpy arrays")
    else :
        view = matrix
    return matrix[:, :]

def create_cum_hazard_error(t, intervals, sigma_hat):
    grad = np.zeros(shape=(1, intervals.shape[0] - 1))
    i = 0
    while t > intervals[i+1] :
        grad[0, i] += intervals[i+1] - intervals[i]
        i+=1
    grad[0, i] += t - intervals[i]
```

```
    e = grad.dot(sigma_hat).dot(grad.T)
    return e

def create_cum_bay_hazard_error(t, actual, intervals, sigma_hat):
    grad = np.zeros(shape=(1, intervals.shape[0] - 1))
    i = 0
    while intervals[i+1] <= actual :
        i += 1
    if t < intervals[i+1] :
            grad[0, i] += (t-actual)
    else :
        grad[0, i] += (intervals[i+1]-actual)
        while t > intervals[i+1] :
            grad[0, i] += intervals[i+1] - intervals[i]
            i+=1
        grad[0, i] += t - intervals[i]
    e = grad.dot(sigma_hat).dot(grad.T)
    return e

def create_cum_prop_hazard_error(t, intervals, h, c_hat, explicatives,
sigma_hat) :
    grad = np.zeros(shape = (1, sigma_hat.shape[0]))
    if type(explicatives) != np.array :
        explicatives = np.array(explicatives)
    i = 0
    while t > intervals[i+1] :
        grad[0, i] -= exp(h[i]+(explicatives*c_hat).sum()) * (intervals[i+1] -
intervals[i])
        i+=1
    grad[0, i] -= exp(h[i]+(explicatives*c_hat).sum()) * (t - intervals[i])
    temp = grad.sum()
    for j in range(c_hat.shape[0]) :
        grad[0, -j] -= explicatives[-j]*temp
    e = grad.dot(sigma_hat).dot(grad.T)
    return e

def create_cum_prop_bay_hazard_error(t, actual, intervals, h, c_hat,
explicatives, sigma_hat) :
    grad = np.zeros(shape = (1, sigma_hat.shape[0]))
    i = 0
    while intervals[i+1] <= actual :
        i += 1
    if t < intervals[i+1] :
        grad[0, i] += (t-actual) * exp(h[i]+(explicatives*c_hat).sum())
    else :
        grad[0, i] -= exp(h[i]+(explicatives*c_hat).sum()) * (intervals[i+1]-
actual)
        while t > intervals[i+1] :
            grad[0, i] -= exp(h[i]+(explicatives*c_hat).sum()) * (intervals[i+1]
- intervals[i])
            i+=1
        grad[0, i] -= exp(h[i]+(explicatives*c_hat).sum()) * (t - intervals[i])
    temp = grad.sum()
    for j in range(c_hat.shape[0]) :
        grad[0, -j] -= explicatives[-j]*temp
```

```
    e = grad.dot(sigma_hat).dot(grad.T)
    return e

@cython.boundscheck(False)
cdef double LL_onepiece(double[:] data, double[:] intervals,
                        double[:] values, double[:] finished) nogil:
    """parallel low-level likelihood func of the piecewise constant hazard
model"""
    cdef double LL=0
    cdef int J = data.shape[0]
    cdef int I = intervals.shape[0]
    cdef int i, j
    for j in prange(J, schedule = "static", chunksize = 500) :
        for i in range(I-1) :
            if data[j] <= intervals[i+1] :
                LL += values[i] * (data[j] - intervals[i])
                LL += -log(values[i]) * finished[j]
                break
            else :
                LL += values[i] * (intervals[i+1] - intervals[i])
    return LL

@cython.boundscheck(False)
cdef double LL_reg_onepiece(double[:,:] data, double[:] intervals, double[:]
values, int k,
                            double[:] Xc) nogil:
    """parallel low-level likelihood func of the piecewise constant hazard model
     with accelerated time values or proportionnal hazard (same likelihood)"""
    cdef double LL = 0
    cdef int J = data.shape[0]
    cdef int I = intervals.shape[0]
    cdef double[:] h_hat = values[:I-1]
    cdef double[:] c = values[I-1:]
    cdef int i, j, l
    for j in prange(J, schedule = "static", chunksize = 500) :
        for l in range(k) :
            Xc[j] += c[l]*data[j, 2+l]
        for i in range(I-1) :
            if data[j, 0] <= intervals[i+1] :
                LL += (c_exp(h_hat[i]+Xc[j])) * (data[j, 0] - intervals[i])
                LL += -(log(c_exp(h_hat[i]))+Xc[j]) * data[j, 1]
                break
            else :
                LL += c_exp(h_hat[i] + Xc[j])* (intervals[i+1] - intervals[i])
    return LL

def cumulative_hazard(t, intervals, values) :
    """survival func for the piecewise constant hazard model"""
    H = 0
    j = 0
    while t > intervals[j+1]:
        H += values[j] * (intervals[j+1] - intervals[j])
        j += 1
    H += values[j] * (t - intervals[j])
    return H
```

```
class _PCH_estimation :
    """subclass of user's tool PCH_estimator"""
    def __init__(self, data, intervals, finished, time = "") :
        if (type(data) == pd.DataFrame) & (type(finished) == str) & (time != "")
:
            self._dta = data[[time]+[finished]]
            self._dta = self._dta[~pd.isnull(self._dta)]
            self.finshed = self._dta[finished].astype(float).values
            self.dta = self._dta[time].astype(float).values
        else : raise TypeError("incorrect input")
        self.intrvals = np.array(np.hstack((intervals, 1e100)), dtype =
np.float)
        self.data = converter(self.dta)
        self.intervals = converter(self.intrvals)
        self.finished = converter(self.finshed)
    def _for_estimation(self, values) :
        values_ = np.array(values, dtype = np.float)
        values_view = converter(values_)
        return LL_onepiece(self.data, self.intervals, values_view,
self.finished)
    def _fit(self):
        return scipy.optimize.minimize(self._for_estimation, np.zeros(shape =
(self.intervals.shape[0]-1))+0.5,
                 bounds = [[1e-9, 1e9]]*(self.intervals.shape[0]-1), method =
"L-BFGS-B")
    def get_results(self) :
        estimations = self._fit()
        self.h_hat = estimations["x"]
        self.neg_hessian = n.Hessian(self._for_estimation, step = 0.0001)
(self.h_hat)
        self.sigma_hat = np.linalg.inv(self.neg_hessian)
        return {"h_hat" : self.h_hat, "sigma_hat" : self.sigma_hat,
                "std_h_hat" : np.sqrt(np.diag(self.sigma_hat))}

class PCH_estimator :
    """python interface for low-level likelihood func optimisation"""
    def __init__(self, data, intervalle, finished, time = "") :
        self.results = _PCH_estimation(data, intervalle, finished,
time).get_results()
        self._dta_max = data[time].max()
        self.intrvals = np.hstack((intervalle, 1e100))
    def graph_surv(self, dim = (10, 5), alpha = 5, nb_dots = 2500):
        """plot estimated survival with confidence intervals"""
        dataplot_x = np.linspace(0, np.max(self._dta_max)*1.2, nb_dots)[1:]
        err = [sqrt(create_cum_hazard_error(x,
                    self.intrvals, self.results["sigma_hat"])) for x in
dataplot_x]
        Y = [cumulative_hazard(x, self.intrvals, self.results["h_hat"]) for x in
dataplot_x]
        CI_down = [norm.ppf(1-alpha/200, loc = y, scale = e) for y, e in zip(Y,
err)]
        CI_down = np.maximum(CI_down, 0).astype(float)
        CI_up = [norm.ppf(alpha/200, loc = y, scale = e) for y, e in zip(Y,
err)]
```

```
        dataplot_y = [exp(-y) for y in Y]
        dataplot_CI_down = np.array([exp(-up) for up in CI_up])
        dataplot_CI_up = np.array([exp(-down) for down in CI_down])
        _forbid_res(converter(dataplot_CI_down))
        _forbid_res(converter(dataplot_CI_up))
        pp.figure(figsize = dim)
        pp.style.use("fivethirtyeight")
        pp.xlim([0, self._dta_max*1.2])
        pp.ylim([-0.05, 1])
        pp.plot(dataplot_x, dataplot_CI_up, linewidth = 6, color = "c",
linestyle = "--")
        pp.plot(dataplot_x, dataplot_CI_down, linewidth = 6, color = "c",
linestyle = "--")
        pp.plot(dataplot_x, dataplot_y, linewidth = 3, color = "k")
    def graph_hazard(self, dim = (10, 5), alpha = 5, y_max = None) :
        """plot estimated hasard with confidence intervals"""
        dataplot_x = list(self.intrvals[:-1])*2 ; dataplot_x.sort() ;
dataplot_x.append(1e100)
        CI_down = []
        CI_up = []
        for i in range(len(self.results["h_hat"])) :
            for j in range(2) :
                CI_down.append(norm.ppf(alpha/200, loc = self.results["h_hat"]
[i],
                            scale = sqrt(self.results["sigma_hat"][i, i])))
                CI_up.append(norm.ppf(1-alpha/200, loc = self.results["h_hat"]
[i],
                            scale = sqrt(self.results["sigma_hat"][i, i])))
        dataplot_y = []
        for i in self.results["h_hat"] :
            for j in range(2) :
                dataplot_y.append(i)
        dataplot_y =[0] + dataplot_y
        CI_down = [0] + CI_down
        CI_up = [0] + CI_up
        CI_down = np.maximum(CI_down, 0)
        if y_max == None :
            y_max = np.max(CI_up)*1.2
        pp.figure(figsize = dim)
        pp.style.use("fivethirtyeight")
        pp.xlim([0, self._dta_max])
        pp.ylim([0, y_max])
        pp.plot(dataplot_x, CI_up, linewidth = 3, color = "c", linestyle =
"--" )
        pp.plot(dataplot_x, CI_down, linewidth = 3, color = "c", linestyle =
"--")
        pp.plot(dataplot_x, dataplot_y, linewidth = 3, color = "k")
    def get_stats(self, alpha = 5, stats=[0.5], nb_dots = 2500):
        """returns mean, median and other stats with custom precision"""
        X = np.linspace(0, self._dta_max*1.2, nb_dots)[1:]
        err = [sqrt(create_cum_hazard_error(x,
                    self.intrvals, self.results["sigma_hat"])) for x in X]
        Y = [cumulative_hazard(x, self.intrvals, self.results["h_hat"]) for x in
X]
        Y = np.array(Y)
```

```
        CI_down = [norm.ppf(1-alpha/200, loc = y, scale = e) for y, e in zip(Y,
err)]
        CI_up = [norm.ppf(alpha/200, loc = y, scale = e) for y, e in zip(Y,
err)]
        CI_up = np.maximum(CI_up,0)
        y_up = np.exp(-np.array(CI_up))
        _forbid_res(converter(y_up))
        y_down = np.exp(-np.array(CI_down))
        _forbid_res(converter(y_down))
        E = sum([(X[i+1]-X[i])*exp(-Y[i+1]) for i in range(nb_dots-2)])
        E_down = sum([(X[i+1]-X[i])*y_down[i+1] for i in range(nb_dots-2)])
        E_up = sum([(X[i+1]-X[i])*y_up[i+1] for i in range(nb_dots-2)])
        results = [{"E" : E, "E_up "+str(alpha/2)+"%" : E_up,
                    "E_down "+str(alpha/2)+"%" : E_down}]
        for stat in stats :
            i = np.argmin((np.exp(-Y)-stat)**2)
            j = np.argmin((y_up-stat)**2)
            k = np.argmin((y_down-stat)**2)
            results.append({str(stat*100)+"%" : X[i],
                            str(alpha/2)+"% up" : X[j],
                             str(alpha/2)+"% down" : X[k]})
        return results
    def forecast(self, duration, alpha = 5,  stats=[0.5], nb_dots = 2500) :
        """returns P(t>T|t) by the bayesian formula, may fail if p(t > T) is
around zero"""
        X = np.linspace(duration, self._dta_max*1.2, nb_dots)[1:]
        p_t = exp(-cumulative_hazard(duration, self.intrvals,
self.results["h_hat"]))
        K = 2
        while exp(-cumulative_hazard(X[-1], self.intrvals,
self.results["h_hat"]))/p_t > 0.02 :
            X = np.linspace(duration, self._dta_max*1.2*K, nb_dots)[1:]
            K += 1
        err = [sqrt(create_cum_bay_hazard_error(x, duration,
                    self.intrvals, self.results["sigma_hat"])) for x in X]
        Y = np.exp([-cumulative_hazard(x, self.intrvals, self.results["h_hat"])
for x in X])/p_t
        _Y = [cumulative_hazard(x, self.intrvals, self.results["h_hat"]) for x
in X]
        CI_down = [norm.ppf(1-alpha/200, loc = y, scale = e) for y, e in zip(_Y,
err)]
        CI_up = [norm.ppf(alpha/200, loc = y, scale = e) for y, e in zip(_Y,
err)]
        CI_up = np.maximum(CI_up, 0)
        CI_down = np.array(CI_down)
        y_up = np.exp(-np.array(CI_up))/p_t
        _forbid_res(converter(y_up))
        y_down = np.exp(-np.array(CI_down))/p_t
        _forbid_res(converter(y_down))
        E = sum([(X[i+1]-X[i])*Y[i+1] for i in range(nb_dots-2)])
        E_down = sum([(X[i+1]-X[i])* y_down[i+1] for i in range(nb_dots-2)])
        E_up = sum([(X[i+1]-X[i])*y_up[i+1] for i in range(nb_dots-2)])
        results = [{"E" : E+duration , "E_up "+str(alpha/2)+"%" : E_up+duration,
                    "E_down "+str(alpha/2)+"%" : E_down+duration}]
        for stat in stats :
```

```
            i = np.argmin((Y-stat)**2)
            j = np.argmin((y_up-stat)**2)
            k = np.argmin((y_down-stat)**2)
            results.append({str(stat*100)+"%" : X[i],
                            str(alpha/2)+"% up" : X[j],
                             str(alpha/2)+"% down" : X[k]})
        return results
class _PCH_regression :
    """python interface for low-level likelihood optimisation including
regression."""
    def __init__(self, data, intervals, finished, explicatives, time = "") :
        if (type(data) == pd.DataFrame) & (type(finished) == str) & \
            ((type(explicatives) == str)|(type(explicatives) == list)) & (time !
= "") :
            if type(explicatives) == str :
                self.k = 1
                temp = data[[time, finished, explicatives]].dropna(axis = 0)
            else :
                self.k = len(explicatives)
                temp = data[explicatives+[time]+[finished]].dropna(axis = 0)
            self.dta = temp[time].astype(float).values
            self.finshed = temp[finished].astype(float).values
            self.explictives = temp[explicatives].astype(float).values
            if self.k == 1 :
                self.means = self.explictives.mean()
                self.explictives -= self.explictives.mean()
            else :
                for i in range(self.k) :
                    self.explictives[:, i] -= self.explictives[:, i].mean()
        else : raise TypeError("invalid input")
        self.intrvals = np.array(np.hstack((intervals, 1e100)), dtype =
np.float)
        if self.k == 1 :
            self.total = np.vstack((self.dta, self.finshed, self.explictives)).T
        else :
            self.total = np.hstack((np.vstack((self.dta, self.finshed)).T,
self.explictives))
        self.total = np.array(self.total, order = "F")
        self.total_view = converter_2d(self.total)
        self.intervals_view = converter(self.intrvals)
    def _for_estimation(self, values) :
        values_ = np.array(values, dtype = np.float)
        values_view = converter(values_)
        reg_container = np.zeros(shape = self.total_view.shape[0], dtype =
float)
        reg_container_view = converter(reg_container)
        return LL_reg_onepiece(self.total_view, self.intervals_view ,
                               values_view, self.k, reg_container_view)
    def _fit(self):
        return scipy.optimize.minimize(self._for_estimation,
                                       np.zeros(shape =
(self.intervals_view.shape[0]+self.k-1),
                                       dtype = float),  method = "L-BFGS-B")
    def get_results(self) :
        estimations = self._fit()
```

```
        h_hat = np.exp(estimations["x"][:-self.k])
        c_hat = estimations["x"][-self.k:]
        neg_hessian = n.Hessian(self._for_estimation, step = 0.001)
(estimations["x"])
        sigma_hat = np.linalg.inv(neg_hessian)
        return {"h_hat" : h_hat, "c_hat" : c_hat, "sigma_hat" : sigma_hat,
                "std_h_hat" : np.sqrt(np.diag(sigma_hat)[:-self.k]*h_hat**2),
                "theta_hat" : estimations["x"][:-self.k],
                "std_c_hat" : np.sqrt(np.diag(sigma_hat)[-self.k:])}



class PCH_regressor :
    def __init__(self, data, intervalle, finished, explicatives, time = "") :
        self.results = _PCH_regression(data, intervalle, finished, explicatives,
time).get_results()
        if (type(data) == pd.DataFrame) & (time != "") :
            self._dta_max = np.max(data[time])
            if type(explicatives) == str :
                self.k = 1
                self.means = data[explicatives].dropna(axis = 0).mean()
            else :
                self.k = len(explicatives)
                self.means = data[explicatives].dropna(axis = 0).mean().values
        else :
            raise TypeError("Invalid input")
        self.intrvals = np.hstack((intervalle, 1e100))
    def graph_common_surv(self, dim = (10, 5), alpha = 5, nb_dots = 2500):
        explicatives_ = np.zeros(shape = self.k)
        Xc = exp(sum(explicatives_ * self.results["c_hat"]))
        X = np.linspace(0, self._dta_max*1.2, nb_dots)[1:]
        nb_dots = X.shape[0]
        err = [sqrt(create_cum_prop_hazard_error(x,
                    self.intrvals, self.results["theta_hat"],
self.results["c_hat"],
                    explicatives_,
                    self.results["sigma_hat"])) for x in X]
        Y = [cumulative_hazard(x, self.intrvals, self.results["h_hat"]) for x in
X]
        CI_down = [norm.ppf(1-alpha/200, loc = y, scale = e) for y, e in zip(Y,
err)]
        CI_up = [norm.ppf(alpha/200, loc = y, scale = e) for y, e in zip(Y,
err)]
        CI_up = np.maximum(CI_up, 0)
        CI_down = np.array(CI_down)
        dataplot_y = np.exp(-np.array(Y))
        dataplot_ci_down = np.exp(-CI_down)
        _forbid_res(converter(dataplot_ci_down))
        dataplot_ci_up = np.exp(-CI_up)
        _forbid_res(converter(dataplot_ci_up))
        pp.figure(figsize = dim)
        pp.style.use("fivethirtyeight")
        pp.xlim([0, self._dta_max*1.2])
        pp.ylim([-0.05, 1.1])
```

```
        pp.plot(X, dataplot_ci_down, linewidth = 6, color = "c", linestyle =
"--")
        pp.plot(X, dataplot_ci_up, linewidth = 6, color = "c", linestyle = "--")
        pp.plot(X, dataplot_y, linewidth = 3, color = "k")
    def graph_common_hazard(self, dim = (10, 5), alpha = 5, y_max = None) :
        dataplot_x = list(self.intrvals[:-1])*2 ; dataplot_x.sort() ;
dataplot_x.append(1e100)
        CI_down = []
        CI_up = []
        for i in range(len(self.results["h_hat"])) :
            for j in range(2) :
                CI_down.append(norm.ppf(alpha/200, loc = self.results["h_hat"]
[i],
                            scale = self.results["std_h_hat"][i]))
                CI_up.append(norm.ppf(1-alpha/200, loc = self.results["h_hat"]
[i],
                            scale = self.results["std_h_hat"][i]))

        dataplot_y = []
        for i in self.results["h_hat"] :
            for j in range(2) :
                dataplot_y.append(i)
        dataplot_y =[0] + dataplot_y
        CI_down = [0] + CI_down
        CI_up = [0] + CI_up
        CI_down = np.maximum(CI_down, 0)
        if y_max == None :
            y_max = np.max(CI_up)*1.2
        pp.figure(figsize = dim)
        pp.style.use("fivethirtyeight")
        pp.xlim([0, self._dta_max*1.2])
        pp.ylim([-0.05, y_max])
        pp.plot(dataplot_x, CI_up, linewidth = 6, color = "c", linestyle =
"--" )
        pp.plot(dataplot_x, CI_down, linewidth = 6, color = "c", linestyle =
"--")
        pp.plot(dataplot_x, dataplot_y, linewidth = 3, color = "k")
    def get_stats(self, explicatives = [], alpha = 5, stats=[0.5], nb_dots =
1500):
        """returns mean, median and other stats with custom precision"""
        if (type(explicatives) == list) & (len(explicatives) == 0) :
            explicatives_ = np.zeros(shape = self.k)
        else :
            explicatives_ = np.array(explicatives) - self.means
        Xc = exp(sum(explicatives_ * self.results["c_hat"]))
        X = np.linspace(0, self._dta_max*1.2, nb_dots)[1:]
        ytest = exp(-cumulative_hazard(X[-1]*Xc, self.intrvals,
self.results["h_hat"]))
        K=2
        while ytest > 0.02 :
            X = np.linspace(0, self._dta_max*1.2*K, nb_dots)[1:]
            ytest = exp(-cumulative_hazard(X[-1]*Xc, self.intrvals,
self.results["h_hat"]))
            K += 1
        err = [sqrt(create_cum_prop_hazard_error(x,
```

```
                    self.intrvals, self.results["theta_hat"],
self.results["c_hat"],
                    explicatives_,
                    self.results["sigma_hat"])) for x in X]
        Y = [cumulative_hazard(x*Xc, self.intrvals, self.results["h_hat"]) for x
in X]
        Y = np.array(Y)
        CI_down = [norm.ppf(1-alpha/200, loc = y, scale = e) for y, e in zip(Y,
err)]
        CI_down = np.array(CI_down, dtype = float)
        CI_up = [norm.ppf(alpha/200, loc = y, scale = e) for y, e in zip(Y,
err)]
        CI_up = np.maximum(CI_up, 0).astype(float)
        y_up = np.exp(-CI_up)
        _forbid_res(converter(y_up))
        y_down = np.exp(-CI_down)
        _forbid_res(converter(y_down))
        E = sum([(X[i+1]-X[i])*exp(-Y[i+1]) for i in range(nb_dots-2)])
        E_down = sum([(X[i+1]-X[i])*y_down[i+1] for i in range(nb_dots-2)])
        E_up = sum([(X[i+1]-X[i])*y_up[i+1] for i in range(nb_dots-2)])
        results = [{"E" : E, "E_up "+str(alpha/2)+"%" : E_up,
                    "E_down "+str(alpha/2)+"%" : E_down}]
        for stat in stats :
            i = np.argmin((np.exp(-Y)-stat)**2)
            j = np.argmin((y_up-stat)**2)
            if j == 0 :
                j = nb_dots-2
            k = np.argmin((y_down-stat)**2)
            results.append({str(stat*100)+"%" : X[i],
                            str(alpha/2)+"% up" : X[j],
                             str(alpha/2)+"% down" : X[k]})
        return results
    def forecast(self, duration, explicatives = [], alpha = 5,  stats=[0.5],
nb_dots = 2500) :
        """returns P(t>T|t) by the bayesian formula"""
        if (type(explicatives) == list) & (len(explicatives) == 0) :
            explicatives_ = np.zeros(shape = self.k)
        else :
            explicatives_ = np.array(explicatives) - self.means
        Xc = exp(sum(explicatives_ * self.results["c_hat"]))
        X = np.linspace(duration, self._dta_max*1.2, nb_dots)[1:]
        p_t = exp(-cumulative_hazard(duration*Xc, self.intrvals,
self.results["h_hat"]))
        ytest = exp(-cumulative_hazard(X[-1]*Xc, self.intrvals,
self.results["h_hat"]))/p_t
        K=2
        while ytest > 0.02 :
            X = np.linspace(duration, self._dta_max*1.2*K, nb_dots)[1:]
            ytest = exp(-cumulative_hazard(X[-1]*Xc, self.intrvals,
self.results["h_hat"]))/p_t
            K += 1
        err = [sqrt(create_cum_prop_bay_hazard_error(x, duration,
                    self.intrvals, self.results["theta_hat"],
self.results["c_hat"],
                    explicatives_,
```

```
                    self.results["sigma_hat"])) for x in X]
        Y = np.exp([-cumulative_hazard(x*Xc, self.intrvals,
self.results["h_hat"]) for x in X])/p_t
        _Y = [cumulative_hazard(x*Xc, self.intrvals, self.results["h_hat"]) for
x in X]
        CI_down = [norm.ppf(1-alpha/200, loc = y, scale = e) for y, e in zip(_Y,
err)]
        CI_down = np.array(CI_down, dtype = float)
        CI_up = [norm.ppf(alpha/200, loc = y, scale = e) for y, e in zip(_Y,
err)]
        CI_up = np.maximum(CI_up, 0)
        y_up = np.exp(-CI_up)/p_t
        _forbid_res(converter(y_up))
        y_down = np.exp(-CI_down)/p_t
        _forbid_res(converter(y_down))
        E = sum([(X[i+1]-X[i])*Y[i+1] for i in range(nb_dots-2)])
        E_down = sum([(X[i+1]-X[i])*y_down[i+1] for i in range(nb_dots-2)])
        E_up = sum([(X[i+1]-X[i])*y_up[i+1] for i in range(nb_dots-2)])
        results = [{"E" : E+duration, "E_up "+str(alpha/2)+"%" : E_up+duration,
                    "E_down "+str(alpha/2)+"%" : E_down+duration}]
        for stat in stats :
            i = np.argmin((Y-stat)**2)
            j = np.argmin((y_up-stat)**2)
            if j == 0 :
                j = nb_dots-2
            k = np.argmin((y_down-stat)**2)
            results.append({str(stat*100)+"%" : X[i],
                            str(alpha/2)+"% up" : X[j],
                             str(alpha/2)+"% down" : X[k]})
        return results
```