Overview of **durmod**

Simen Gaure
Ragnar Frisch Centre for Economic Research, Oslo, Norway

June 30, 2019

Abstract

This is a walkthrough of an estimation of a generated dataset with the **durmod** package. Also, various tunable parameters and details are provided.

1 A dataset

The **durmod** package fits a mixed proportional hazard model with competing risks to duration data. The model is the one from [Gaure et al., 2007], which was developed in [Heckman and Singer, 1984] based on results in [Lindsay, 1983a, Lindsay, 1983b].

Let's have a look at a generated dataset which simulates an unemployment register with two competing risks. It was generated by `durdata <- datagen(5000,400)`.

```
library(durmod)
data(durdata)
head(durdata,15)
```

```
##     id         x1          x2 alpha       d   duration     state
##  1:  1 -0.3317134 -1.34060625     0     job   0.230954     unemp
##  2:  2 -0.6737615 -0.46594276     0    none  13.327312     unemp
##  3:  2 -1.6495569  1.54143453     0    none  44.964614     unemp
##  4:  2 -2.4295179  0.37094254     0    none 103.924002     unemp
##  5:  2 -1.6927724  1.43136302     0    none  28.231373     unemp
##  6:  2 -1.7235513 -0.15882989     0    none   9.888351     unemp
##  7:  2 -2.7544434  1.51210768     0    none  70.451252     unemp
##  8:  2 -4.3062211  3.40174960     0    none  24.068327     unemp
##  9:  3  0.7160858  0.07710256     0 program   5.759026     unemp
## 10:  3  0.7160858  0.07710256     1    none   2.085667 onprogram
## 11:  3 -0.5252669 -0.05042386     1    none   4.710827 onprogram
## 12:  3 -1.7092856 -0.24886947     1     job   2.305123 onprogram
## 13:  4 -1.4664934 -1.62605988     0     job   1.591296     unemp
## 14:  5 -0.3070180  1.63896593     0    none   0.704823     unemp
## 15:  5 -1.5294205  0.81822933     0    none  24.497315     unemp
```

There is an `id` which identifies an individual. The individuals have been through a process. At the outset they are all unemployed, this is recorded by the factor `state`. As unemployed they face two hazards, i.e. probabilites per time unit. Either they can get a job, or they can enter a labour market programme, like a subsidized wage job or similar.

These transitions are recorded in the `d` factor. In our simulation, individuals who transition to `"job"`, exit the dataset. If a transition to labour market programme occurs, the state variable changes to `"onprogram"`, and the dummy `alpha` changes to 1. It is also possible to do a `"none"` transition, this is typically necessary if a covariate changes, since the model has piecewise constant explanatory covariates. Also, when on a programme, one of the hazards disappear, it is no longer possible to make a transition to a programme, we're already on it.

Each row of the dataset has a `duration`, this is the time until the transition marked in `d` occurs.

The rows of an individual are independent, they can be reordered, but the dataset must be sorted on individuals, i.e. the rows for an individual must be together. I suggest using data tables (from the **data.table** package), then it is easy to sort on individuals: `setorder(data,id)`. If you have factors with hundreds or thousands of levels, it can be time-consuming to compute the Fisher-matrix. However, the algorithm which creates the Fisher matrix is more efficient if individuals with almost the same values for these factors (e.g. county, or company) are close to each other in the dataset.

In our dataset, we have an observation period of 400, and the individuals enter the dataset at a random time in this period. When they reach the end of the observation period they are no longer observed. That means that some individuals do not exit the dataset by doing a transition, but with a `d=="none"`.

2 The mixed proportional hazard competing risk model

There are two covariates, `x1` and `x2`. These are assumed to influence the two hazards. We also assume the `alpha` enters the hazard.

We model the baseline hazard for transition to job as,

$$h^j(\mu^j) = \exp(x_1\beta_1^j + x_2\beta_2^j + \alpha\beta_3^j + \mu^j) \tag{1}$$

The hazard for transition to programme is,

$$h^p(\mu^p) = \exp(x_1\beta_1^p + x_2\beta_2^p + \mu^p) \tag{2}$$

Here we have included an "intercept", a μ, it could equally well have been written as a multiplicative factor $\exp(\mu)$ instead. This $\exp(\mu)$-term is the "proportional hazard".

The likelihood for a single observation k consists of two parts. Let $H(\mu) = h^j(\mu^j) + h^p(\mu^p)$ be the sum of the hazards, where μ is the vector (μ^j, μ^p).

For an observation k there is a survival probability/density up until the transition:

$$s_k(\mu) = \exp(-t_k H(\mu)), \tag{3}$$

where t_k is the duration of the period.

If there is a transition, $s(\mu)$ is multiplied by the transition hazard, $h^d(\mu)$, where d is either p or j. If there is no transition, $h^d(\mu)$ is taken to be 1. Taken together, all the observations for an individual i yields an individual likelihood. We call it $\ell_i(\mu)$.

$$\ell_i(\mu) = \prod_{k\in K_i} h^{d_k}(\mu)s_k(\mu), \tag{4}$$

where K_i is the set of observations for individual i.

However, there is also a mixture part, designed to account for unobserved individual heterogeneity. The μ-vector is stochastic with a discrete distribution. That is, there is an n, a set of probabilites p_j, and vectors μ_j, for $j = 1..n$. Of course, we have $\sum_{j=1}^n p_j = 1$.

The mixture likelihood for an individual i is $L_i = \sum_j p_j\ell_i(\mu_j)$.

The log-likelihood for the dataset is thus, $L = \sum_i \log(L_i)$.

The L must be maximized with respect to the five βs, the n, the probabilites p_j, and the vectors μ_j for $j = 1..n$.

3 Estimation

The estimation proceeds as follows. We start with $n = 1$, estimate the βs and the two μs. Then we increase n to 2, let $p_2 = 0$, and find a vector μ_2 such that the derivative of the log-likelihood in the direction of positive p_2 is positive. This is used as starting point for a new likelihood maximization. Then n is increased to 3, and we proceed in this fashion, adding masspoints to the distribution until we are no longer able to increase the likelihood. Note that even if we are not able to find a point with positive derivative (as in [Lindsay, 1983a, Theorem 4.1]), we anyway use the closest

we have found, and do another round optimizing the full model. The reason is that the theorem assumes the coefficients for the covariates have been found, but in practice they may still change, creating an opportunity for another point.

In **durmod** we use the `mphcrm` function for this purpose. Here is an example. First we create a "riskset", a specification of which hazards are experienced in various states:

```
risksets <- list(unemp=c('job','program'), onprogram='job')
```

Note that the names of the list `risksets` are the same as the levels in the factor `state`. And that the entries in the list are levels of the factor `d`, i.e. possible transitions.

Then we create a set of control parameters. Since this vignette is to be created by the busy CRAN repository, we limit ourselves to 4 iterations, i.e. no more than 4 masspoints in the distribution. For the same reason we also limit to 1 cpu, or threads, in the computation. The default is to use all the available cpus/cores.

```
ctrl <- mphcrm.control(iters=4, threads=1)
```

Then we are ready to estimate. There are a couple of special terms in the formula we use:

```
set.seed(42) # for reproducibility
opt <- mphcrm(d ~ x1 + x2 +
                C(job, alpha) + ID(id) + D(duration) + S(state),
              data=durdata, risksets=risksets, control=ctrl)

## mphcrm 09:44:47 i:1 p:1 L:-23397.2902 g:1.67e-05 mp:1 rc:0.024 e:-0.0000 t:0.6s
## mphcrm 09:44:48 i:2 p:2 L:-22568.1711 g:6.05e-06 mp:0.33279 rc:0.01 e:0.6361 t:1.1s
## mphcrm 09:44:52 i:3 p:3 L:-22483.5537 g:1.08e-05 mp:0.074596 rc:0.0014 e:0.9029 t:4.0s
## mphcrm 09:44:56 i:4 p:4 L:-22469.0497 g:2.62e-05 mp:0.031328 rc:0.00022 e:1.1030 t:4.6s
```

The left hand side of the formula, `d`, is the outcome, the transition that is taken. The `C(job, alpha)` term is a list of conditional covariates, the `alpha` should only explain the `"job"` transition. The `ID(id)` specifies that the covariate `id` identifies individuals. The `D(duration)` specifies that the covariate `duration` contains the durations of the observations. Finally, the `S(state)` term specifies that the covariate `state` is a factor which indexes into the `risksets` argument. In this application, we could as well have replaced `C(job,alpha)` with `C(job,state)` in the formula, since these two covariates are essentially the same.

`mphcrm` writes diagnostic output, one line per iteration. It contains potentially useful information. There is a time stamp, the iteration number, the number of masspoints, the resulting log likelihood, the 2-norm of the gradient, the smallest probability in the masspoint distribution, the reciprocal condition number of the Fisher matrix, the entropy of the masspoint distribution, and the time used in the iteration.

`mphcrm` returns a list with one entry for each iteration, it has a special print method which sums up the estimation in reverse order:

```
print(opt)

## iter4: estimate with 4 points, log-likelihood: -22469.0497
##
##     job.x1     job.x2  job.alpha program.x1 program.x2
##  0.9964333 -0.9752032 -0.0402971  1.0106482  0.3441593
##
## Proportional hazard distribution
##                 prob        job    program
## point  1 0.51260712 0.11689178 0.07037033
## point  2 0.29869519 0.03138931 0.03095598
## point  3 0.15736986 0.41290599 0.24906807
## point  4 0.03132783 0.00926154 0.00557658
```

```
##
## iter3: estimate with 3 points, log-likelihood: -22483.5537
## iter2: estimate with 2 points, log-likelihood: -22568.1711
## iter1: estimate with 1 points, log-likelihood: -23397.2902
## nullmodel: estimate with 1 points, log-likelihood: -28974.1282
```

Unless something has gone wrong, you will normally be interested in the first entry, the one with the largest likelihood. We can look at a summary:

```
best <- opt[[1]]
summary(best)

## $loglik
## [1] -22469.05
##
## $coefs
##                 value         se           t     Pr(>|t|)
## job.x1      0.9964333 0.01836658  54.2525201 0.000000e+00
## job.x2     -0.9752032 0.02086101 -46.7476405 0.000000e+00
## job.alpha  -0.0402971 0.04567200  -0.8823152 3.776256e-01
## program.x1  1.0106482 0.02322823  43.5094774 0.000000e+00
## program.x2  0.3441593 0.02532524  13.5895787 1.001321e-41
##
## $moments
##               mean    variance         sd
## job     0.13456450 0.016023689 0.12658471
## program 0.08468925 0.005415783 0.07359201
```

It has three entries, `"loglik"`, which is simply the log likelihood, `"coefs"` which is the values and standard errors of the estimated coefficients. And `"moments"`, which is the first and second moments of the proportional hazard distribution.

We can see how the `alpha` changes with more points:

```
t(sapply(opt, function(o) summary(o)$coefs["job.alpha",]))

##                 value         se          t    Pr(>|t|)
## iter4     -0.04029710 0.04567200 -0.8823152 0.377625621
## iter3     -0.04614334 0.04348011 -1.0612515 0.288598886
## iter2     -0.03352000 0.03936906 -0.8514298 0.394549132
## iter1     -0.08024691 0.02459507 -3.2627236 0.001106833
## nullmodel  0.00000000         NA         NA          NA
```

Here is a pre-made fit:

```
summary(fit[[1]])

## $loglik
## [1] -22444.41
##
## $coefs
##                 value         se          t      Pr(>|t|)
## job.x1      1.0008735 0.01953029  51.247230  0.000000e+00
## job.x2     -1.0325680 0.02423803 -42.601146  0.000000e+00
## job.alpha   0.2639588 0.07260225   3.635683  2.785089e-04
## program.x1  1.0277278 0.02595800  39.591941 1.256903e-320
## program.x2  0.4615064 0.03425654  13.472069  4.824799e-41
##
```

```
## $moments
##                mean    variance         sd
## job     0.13617702 0.028997653 0.17028697
## program 0.08489246 0.009307573 0.09647576
```

The full estimation can be rerun with the commands:

```
library(durmod)
data(durdata)
newfit <- eval(attr(fit,'call'))
```

There are also some functions for extracting the proportional hazard distribution:

```
round(mphdist(fit[[1]]),6)

##               prob      job  program
## point  1 0.295516 0.181172 0.049529
## point  2 0.175513 0.029290 0.019901
## point  3 0.143208 0.067335 0.103531
## point  4 0.132564 0.065707 0.027457
## point  5 0.073408 0.023677 0.115193
## point  6 0.063346 0.155159 0.336856
## point  7 0.037025 0.515057 0.386386
## point  8 0.027400 0.590028 0.114730
## point  9 0.024802 0.006909 0.036665
## point 10 0.017977 0.010285 0.002475
## point 11 0.009242 1.296720 0.010676

# and the moments,
mphmoments(fit[[1]])

##                mean    variance         sd
## job     0.13617702 0.028997653 0.17028697
## program 0.08489246 0.009307573 0.09647576

# and covariance matrix
mphcov(fit[[1]])

##                 job     program
## job     0.028997653 0.005300171
## program 0.005300171 0.009307573

# and some pseudo R2
pseudoR2(fit)

##         mcfadden adjmcfadden  coxsnell       ncu
## iter11 0.2253638   0.2240868 0.9266042 0.9266127
## iter10 0.2253580   0.2241846 0.9265992 0.9266078
## iter9  0.2253543   0.2242844 0.9265961 0.9266047
## iter8  0.2253491   0.2243827 0.9265916 0.9266002
## iter7  0.2253142   0.2244514 0.9265620 0.9265706
## iter6  0.2251987   0.2244394 0.9264636 0.9264721
## iter5  0.2247808   0.2241250 0.9261065 0.9261151
## iter4  0.2244687   0.2239165 0.9258388 0.9258474
## iter3  0.2240128   0.2235641 0.9254459 0.9254544
## iter2  0.2210923   0.2207472 0.9228792 0.9228878
## iter1  0.1924765   0.1922349 0.8925506 0.8925589
```

The true values used to generate the dataset was `job.x1=1`, `job.x2=-1`, `job.alpha=0.2`, `program.x1=1`, and `program.x2=0.5`. The true proportional hazard moments are for convenience stored as attributes in the dataset

```
attributes(durdata)[c('means','cov')]

## $means
##        job    program
## 0.13819404 0.08270112
##
## $cov
##                  job     program
## job     0.030584053 0.005931508
## program 0.005931508 0.010850049
```

In this case the estimated mixture has moments fairly close to the true ones, but beware that the estimation process sometimes finds a couple of points with very low probability and very high hazard. If these very low probabilities are imprecisely estimated, so that they should really have been an order of magnitude smaller (10^{-6} instead of 10^{-5}), the moments of the mixture distribution can be way off. It is good practice to inspect the mixture distribution for such extreme points, and go back to a previous iteration (which has slightly worse likelihood) without such extreme points.

3.1 Overparameterization?

We did note in [Gaure et al., 2007] that picking the number of points which yields the lowest AIC yields satisfactory results. We did not, however, have any theoretical justification for doing this, and still don't. It is easy to look at the estimates with the lowest AIC:

```
summary(fit[[which.min(sapply(fit,AIC))]])

## $loglik
## [1] -22445.84
##
## $coefs
##                 value         se          t     Pr(>|t|)
## job.x1      1.0001819 0.01910274  52.358024 0.000000e+00
## job.x2     -1.0263325 0.02356950 -43.544939 0.000000e+00
## job.alpha   0.2405894 0.06952164   3.460641 5.409340e-04
## program.x1  1.0261267 0.02570300  39.922443 0.000000e+00
## program.x2  0.4512076 0.03273131  13.785200 7.098126e-43
##
## $moments
##                mean    variance         sd
## job     0.13471732 0.023595274 0.15360753
## program 0.08443194 0.008140664 0.09022563
```

AIC is generally used to pick a model which is parsimonious, but still explains the data well, i.e. to avoid overparameterization. AIC has an interpretation as the distance between the model and reality, see e.g. [Burnham and Anderson, 2002]. However, since the points are not found in a canonical order, the AIC is really not well defined in these models. Another estimation of the same data may find the points in a different order, with different log likelihoods along the way, resulting in another set of points having the lowest AIC in the new estimation.

Also, keep in mind that the method of using a discrete distribution in this way is *not* an "approximation". When the likelihood can't be improved by adding more points, we have actually reached the likelihood which would result if we were to integrate with the true distribution, be it discrete or continuous. This is the content of [Lindsay, 1983a, Theorems 3.1 and 4.1]. Thus, there is really no theoretical risk of adding "too many" points in the distribution, save for those which may result from numerical problems.

The drawback with the method is that when the likelihood can't be improved by adding more points, the purely technical reason is that something degenerates, i.e. something breaks down, at least the Fisher matrix. That is, we will necessarily run into numerical problems at the end of the estimation, and this may result in irrelevant points with very low probabilities being added, because they seem to increase the likelihood by some small amount due to numerical inaccuracies. This may be the case if the algorithm has run several iterations at the end which barely moved the likelihood.

4 More options

4.1 Interval timing

The example above had exactly recorded time. For some applications we do have that, while in other applications we only have a time interval when the transition is known to have taken place. The data above is actually a prime example, perhaps we only have labour data on a monthly basis. When a transition takes place, it is only registered at the end of the month, and there is no record of the day. In this case, the `duration` would be 1 for every observation, and one should use the `timing="interval"` argument in `mphcrm`. The observation likelihood is replaced by,

$$\frac{h^{d_k}(\mu)}{H(\mu)}(1-\exp(-t_k H(\mu)). \tag{5}$$

If the hazards are small and we use unit intervals, the difference between the interval and exact model is quite small, so one may opt for using the exact model instead.

4.2 No timing

In some applications there isn't any time. A transition occurs, or not. In this case the specification `timing="none"` can be used. It will use a logit model for the transition probabilities.

4.3 Factors

`mphcrm` treats factors specially. There is, I hope, nothing special to see, but internally `mphcrm` does *not* create a large model matrix filled with dummy variables. This means that factors with many levels are quite fast to estimate, there is no wasteful multiplication by zero.

`mphcrm` removes factor levels as explanatory for a transition if they are not observed in any state with a risk for this transition. There can, however, be problems with what the contrasts/references should be if you interact a factor in one transition, but keeps it uninteracted in another transition. For the purpose of automatic removal of reference levels in factors, it is assumed that all terms in all transitions are present everywhere. If this problem occurs, you can create a copy with a different name, so it looks like different factors in different transitions. This may be fixed in a later version, if I find a sensible way to do it.

5 Control parameters

There are many control parameters. Rather than scattering more or less arbitrary constants around the program, I have collected them here with their defaults. Some of them you may want to tinker with.

- `threads=getOption("durmod.threads").` An integer. The number of parallel threads used by `mphcrm`. The default is taken from `getOption("durmod.threads")`, which is initialized from the environment variable `DURMOD_THREADS`, `OMP_NUM_THREADS`, `OMP_THREAD_LIMIT`, `NUMBER_OF_PROCESSORS`, or else from `parallel::detectCores()`.

  It is not always true that the estimation runs twice as fast on twice as many cpus. This depends on the cpu- and memory architecture of your computer, as well as on the implementation of OpenMP in the compiler used to compile the C++ parts of **durmod**. Besides, not

all parts of **durmod** run in parallel, so by Amdahl's law you may not expect linear speedup when the number of cpus tends to infinity.

Also, if you intend to use your computer for something else while `mphcrm` runs, you should not give it all your cpus, but save one or two for your other work. If one of the 16 threads in `mphcrm` shares a cpu with your mail program trying to sort your inbox, the speed may be halved.

For creation of the Fisher matrix, `mphcrm` calls into the BLAS from a single thread. For large datasets with many coefficients to estimate, there can be some benefit from linking R with a highly optimized and parallel BLAS, like mkl from Intel. See the R documentation for how to do this.

In case you run on a cluster (see the `cluster` control parameter), the `threads` parameter is the number of threads on each node in the cluster. In this case, `threads` can be a vector of integers, if you have different number of cpus on each node.

- `iters=50`. An integer. The number of iterations to perform. The estimation may stop earlier, if neither the log likelihood *nor* the entropy improves.
- `ll.improve=0.001`. A numeric. The amount the log-likelihood must increase with to be considered an improvement.
- `e.improve=0.001`. A numeric. The amount the entropy of the hazard distribution must change with to be considered an improvement.
- `gdiff=TRUE`. A logical. When searching for a new point, should we try one with zero probability, and positive derivative in the direction of positive probability? This usually works, but in case of problems with finding new points, it can be set to `FALSE`, and a search for a location with a small probability is done instead. See `newprob`.
- `startprob`=10^{-4}. A numeric. When a location point has been found with zero probability, we can not use a zero probability for further optimization due to our parametrization of probabilities which do not allow it. We must set it to some small value `startprob`. Setting it too small will make it difficult to move away from it.
- `overshoot=0.001`. A numeric. When searching for a positive directional derivative, this is the threshold we must be above. It is positive to avoid marginally positive derivatives due to numerical inaccuracies.
- `newprob`=10^{-4}. A numeric. When searching for a new masspoint with `gdiff=FALSE`, a new support point is added to the distribution with a small probability, then a search is done for a location for this probability which increases the likelihood. `newprob` is this small probability. If the search fails, the new probability is set to zero, and a search for a positive directional derivative of the likelihood (in the direction of positive probability) is done.

  If `mphcrm` after optimization seems to find a series of points with probability equal to `newprob`, it has perhaps got stuck for numerical reasons. Try to increase it to find higher probability points first.
- `minprob`=10^{-20}. A numeric. Masspoints with probability below this are removed.
- `eqtol=0.001`. A numeric. It sometimes happens that two location points in the mixed proportional hazard distribution turns out to be equal. One of them can then be removed. `eqtol` is the threshold below which the distance between two location points is so small that the two points are thought to be equal. The "distance" is the maximum of pointwise relative differences, i.e. with two vectors $\{a_i\}_i$ and $\{b_i\}_i$, the distance is $\max_i |a_i - b_i|/(|a_i| + |b_i|)$.
- `newpoint.maxtime=120`. A numeric. When searching for a new location point, a global search algorithm from package **nloptr** is used. `newpoint.maxtime` is its time limit in seconds. Should be increased if `mphcrm` repeatedly complains about not being able to find a new point. However, when there are no new points to be found at the end of the estimation, this will necessarily happen. See also `lowint`.

- `addmultiple=`∞. A positive numeric. The algorithm tries to add more points to the distribution without optimizing the structural parameters as long as the improvement from the last point is larger than `addmultiple`. If the improvement is less, all parameters are optimized before more points are attempted. Setting `addmultiple=1` or smaller may save some time if there are many structural parameters, but setting it too small risks ending up with too many points which later may be discarded, and possible convergence problems.

- `callback=mphcrm.callback`. A function. If the one-line diagnostic from `mphcrm` is insufficient, it is possible to write your own. It will replace the default callback (which you can call from your function). In this way you can e.g. get diagnostics on particular coefficients, save intermediate results to file, or other partakings. See `mphcrm.callback`.

- `jobname="mphcrm"`. A character string. The initial portion of the one-line diagnostic. Useful if you e.g. use `parallel::mclapply` to run several estimations in parallel. They can have individual names so you can see the progress.

- `tspec="%T"`. A character string. The format of the time stamp in the one-line diagnostic, as described in `help(format.POSIXct)`. Use `"%c"` to get both date and time.

- `trap.interrupt=interactive()`. A logical. If you decide to interrupt an interactive estimation before it has terminated, either because you don't want to wait, or because it seems to have run astray, the default behaviour for `mphcrm` is to catch the interrupt, and return gracefully with the result of the estimation so far. This behaviour can be switched off with `trap.interrupt=FALSE`.

- `tol=`10^{-4}. A numeric. The (absolute) tolerance for the log-likelihood maximization.

- `itfac=20`. An integer. The maximum number of iterations in the BFGS-method is `itfac*K`, where `K` is the number of parameters to estimate.

- `fishblock=128`. An integer. The Fisher matrix is created by calling the BLAS `dsyrk` with blocks of individual gradients. This is the size of the blocks. The optimal size depends on the BLAS version and details of your computing contraption. If memory use is an issue with a large number of coefficients to estimate, it can be lowered, typically to some other power of two.

- `lowint=4`. A numeric, possibly a vector of length the number of transitions. When searching for the location of a new point, an interval centered at the mean of the old points is used. `lowint` is how far from the mean (in log units) the interval goes to the left. `mphcrm` makes no effort at analyzing where the location points may lie, as described in [Lindsay, 1983b], but does a brute force search in a (hyper)rectangle. Setting this parameter too high, increases the risk of finding numerically unfavourable points, in particular large ones (if `highint` is too large), but if `mphrcm` repeatedly complains about not being able to find new points, it can be increased. See also `newpoint.maxtime`.

- `highint=2`. A numeric. Like `lowint`, but to the right.

- `method="BFGS"`. A character string. The default is the most robust method. In case of convergence problems, or for fun, one may try one of the local gradient based NLOPT-methods: `"TNEWTON_PRECOND", "TNEWTON", "SLSQP", "MMA", "TNEWTON_RESTART", "TNEWTON_PRECOND_RESTART", "VAR1", "VAR2"`. The Newton-methods can achieve a smaller gradient than BFGS. For these, it could be a good idea to increase `itfac`.

- `cluster=NULL`. Cluster specification from package **parallel** or **snow**. In addition to utilizing all the cores/cpus on a computer, `mphcrm` may also spread across several computers. It supports running on a cluster from package `parallel` or `snow`. The dataset will be split among the cluster nodes, with approximately equally many observations on each. The nodes will then do their share of the likelihood computations. If using a cluster, the `threads` parameter will be the number of cpus used on each cluster node. It can also be a vector, different number of threads on each node, in which case the dataset will be split unevenly to approximately match the number of threads.

If running on a professional cluster with a resource manager such as SLURM, TORQUE or similar, it is usually possible to pick up information about the cluster nodes from the resource manager, typically from environment variables. With SLURM you can e.g. obtain the number of allocated cpus on each node of your job with something like `threads <- clusterEvalQ(cluster, as.integer(Sys.getenv("SLURM_CPUS_ON_NODE")))`. In general, you will have to check local documentation for how this can be done, and how to best set up a cluster with package parallel. (There can be some catches, in particular with MPI which spin-waits to lower latency, and thus takes up a CPU for the "dormant" master process.)

In general, when using parallelization, one should make sure that the cpus are not overbooked and that the nodes you are running on are approximately equally fast. There is some overhead when distributing the computation among many nodes, depending on how the nodes are interconnected, and due to somewhat sub-optimal implementations in package **parallel** which e.g. does not utilize collective MPI-calls. `mphcrm` will usually run faster if using `threads` on a single node, or few nodes, both because the thread algorithm uses shared memory and because the division of data between the threads are dynamic.

- `nodeshares=NULL`. A numeric vector. When running on a cluster, the default is to share the data equally among the nodes, or in the case `threads` is a vector of node-specific threads, equally among the threads on each node. If this is unsatisfactory, e.g. because the nodes have different computing speeds, you can specify a vector of node specific shares of the data set.

  Say you have 4 nodes in the cluster, but the last two are half as fast as the first two, just specify `nodeshares=c(2,2,1,1)`, and the last two nodes will get half as much data as the first two. If in addition, the first two have 8 cpus, and the last two have four, specify: `threads=c(8, 8, 4, 4)`, and `nodeshares=c(2*8, 2*8, 4, 4)`. The entries in the `nodeshares` vector will be divided by their sum, so it is not necessary that they sum to 1, it is their relative sizes which matter.

- `fisher=TRUE`. A logical. Should the Fisher matrix be computed? You will normally want this, it is used to compute the standard errors. However, it may take some time if you have a large number of covariates. If you do not need standard errors, e.g. if you do a bootstrap or Monte Carlo simulation or similar, you may switch off the computation of the Fisher matrix and save some time.

- `gradient=TRUE`. A logical. As `fisher`, but for the gradient. I doubt much time can be saved by switching this off.

References

[Burnham and Anderson, 2002] Burnham, K. P. and Anderson, D. R. (2002). *Model selection and multimodel inference*. Springer, New York, 2nd edition.

[Gaure et al., 2007] Gaure, S., Røed, K., and Zhang, T. (2007). Time and causality: A Monte Carlo assessment of the timing-of-events approach. *Journal of Econometrics*, 141(2):1159 – 1195.

[Heckman and Singer, 1984] Heckman, J. and Singer, B. (1984). A method for minimizing the impact of distributional assumptions in econometric models for duration data. *Econometrica*, 52(2):271–320.

[Lindsay, 1983a] Lindsay, B. G. (1983a). The geometry of mixture likelihoods: A general theory. *The Annals of Statistics*, 11(1):86–94.

[Lindsay, 1983b] Lindsay, B. G. (1983b). The geometry of mixture likelihoods, part II: The exponential family. *The Annals of Statistics*, 11(3):783–792.