Package: despack (via r-universe)

September 10, 2024

Title Helpers for CKMR design

Version 1.0.11

Author Mark Bravington

Description Help to compute expected Hessian given a design (ie sample sizes) and your CKMR model

Imports utils, mvbutils, offarray

Suggests TMBO

Maintainer Mark Bravington <markb2@summerinsouth.net>

License GPL-3

Repository https://markbravington.r-universe.dev

RemoteUrl https://github.com/markbravington/despack

RemoteRef HEAD

RemoteSha 22dd6ab1a24b7700506183d108a0da90040f3745

Contents

`despack-package` *How to use the despack package*

Description

This package provides helper functions for CKMR design. To use it fully, you need:

- working code for a CKMR model, either in R or TMB(O), that can calculate all the kinship probabilities;
- guessestimates of true parameter values;
- explicit "designs" ie proposals for sample sizes, disaggregated to whatever level would be used in the "real data" (eg by year and age and sex)
- a clear definition of some quantity whose precision you're interested in (eg a total abundance in some specified year; a total biomass; the natural mortality rate; replacement yield in some year; ...). This should be a scalar function that takes all the pop dyn parameters as its argument.

Note that you do *not* need any datasets, whether real or simulated!

There are currently 4 functions in the package, but *don't* start by looking at their helpfiles! Look at the WORKFLOW below, then have a go at the script/vignette which is *not* currently part of the package. The main "documentation" is really a script, with accompanying functions and data, that doesn't currently live inside this package (I'm not sure what the best arrangement is: maybe a "demo package" that goes alongside this one, or maybe "just" as a vignette, but it's complicated). Of necessity, CK examples tend to be quite application-specific, whereas the functions in this package are deliberately generic.

Workflow

To calculate the likely SE of your quantity-of-interest, do this (assuming your model is in TMB(O) rather than R; see later for all-R comments):

- `TMB(O)::MakeADFun` to set up data ranges and constants in the model (but no data required).
- `CK_logalike_funs` to make convenient wrappers for calling your TMB(O) function, and for returning REPORTed quantities, such as CK probability arrays and population dynamics numbers;
- `get_Hbits` to compute all the expected Fisher Information matrices ("negative Hessians") from doing one single comparison between two samples with specified covariates. Results are stored in compressed form.
- `finfo_onetype` for each *type* of CK comparison in turn (eg POPs, XMHSPs, POPs where one sample only has approximate age, ...). Here you will need to supply a crucial new argument, derived from the design itself: an array holding the relevant *number* of comparisons disaggregated by covariates (eg an `offarray` called `n_comp_MOP_BYA`). Result is the expected Fisher Info for all comparisons of that *type*.
- Add up all the `finfo_onetype` results (there might only be one type, if your application is very simple) to get the overall Fisher Info.
- Invert that to get the parameter covariance matrix
- Double-dot that with the numerical derivative of your quantity-of-interest function (ie apply "the delta method"), to get the expected approximate variance of your thing.

A simple example is given in the vignette (which I haven't yet included in the package... there's a separate script).

There is also a schematic example (ie it shows plausible R code, but doesn't actually work) in the help for `get_Hbits`. Also, there is individual documentation for each of `CK_logalike_funs`, `get_Hbits`, and `finfo_onetype`, plus for the helper function `mvbutils::numvbderiv` and *especially* its parallel version `mvbutils::numvbderiv_parallel`, but for goodness sake don't *start* by reading their helpfiles. Follow the vignette/script instead, until and unless you get stuck.

R only design: If your CK model is entirely in R— presumably using `offarray` otherwise you will go mad with index confusion— then you obviously don't need the `MakeADFun` step, and the second step can be "hand-written" easily; again, the vignette/script is the place to look, for the function `reportees_Ronly`. Following steps are identical. Because there's no need to fit the model to actual data, an all-R version is quite feasible even for pretty complex designs— though if you ever turn the design into reality, you might need to write a TMB(O) version anyway to be able to actually fit it— though package **RTMB** may now obviate C-TMB code altogether. But you'll have several years before you need to do that ;)

Efficiently investigating different designs

Once you have gotten all the steps working "manually" for your application, you will want to consider various possible designs and see how they influence Standard Errors / CVs. Since there are a huge number of individual sample sizes that potentially *could* be controlled, you can't really do that manually. For serious use, you will instead want to write your own "driver" function that groups together several of these steps; `varcalcs_delfi_A` (which accompanies the script) is an example. The key part is a "design generator", which:

- takes a small number of control parameters (total sample sizes by category, trends in sample size by year, how to apply selectivity, ...; it all depends on your application);
- generates disaggregated sample sizes, ie a complete design specification;

Once you have generated the detailed design, the rest of your driver function needs to:

- runs the design thru `finfo_onetype` (also using results of one previous call to `get_Hbits`), once for each type of CK comparison (eg MOPs, XSHPs);
- sums those Hessians, and inverts the result to get the parameter covariance matrix;
- applies the "delta method" to that covariance matrix, using numerical derivatives of your quantity-of-interest WRTO the parameters, to calculate the variance of your Thing.

You can also incorporate Hessian components from (some types of) non-CKMR data, such as age-at-length compositional samples, and priors on random effects and parameters— probably not CPUE though :) I've done that myself in several earlier examples. It's really up to you...

Not for now

"Optimal" design subject to arbitrary constraints, using a "driver" function as per last section. (It actually works, but will take me some time, ie at least months, to port over into this package.)

CK_logalike_funs *Make wrappers for TMB(O) functions*

Description

Takes a TMB(O) object as returned by `MakeADFun`, and Returns a list containing three functions: `lglk`, `Dlglk`, and `reportees`. `lglk` can be used exactly as in the all-R example, and returns a **positive** log-likelihood! `Dlglk` returns the gradient. `reportees` returns all the things from TMB REPORT() statements (as a list).

These wrappers are just convenient things that facilitate subsequent use in `despack` functions. You could easily enough write functions yourself to call `TMBOBJ$fn` and `TMBOBJ$report` and so on, and reorganize the output. But you don't have to, because I have...

Before running this, you have to manually call something like `boring_data_prep_<blah>()` to set up an environment containing all the data (which will be `env` here), and then `MakeAdFun` to create the "TMB object" (which will be `TMBOBJ` here).

Calling `lglk(<pars>,report.=TRUE) will create variables in` environment(lglk)' that can be accessed afterwards, just like what happens in the R version.

Calling `reportees` will by default just return variables whose name starts with "Pr_". You can make it also return other things, such as pop dyn quantities or expected kin-pairs, via the parameter `want="all"`.

Usage

```
CK_logalike_funs(env, TMBOBJ, suffix='')
```

Arguments

`env`	environment containing all necessary data
`TMBOBJ`	TMB object (an R list) returned by `MakeADFun`.
`suffix`	string to append to the names of each function— so `suffix="_D"` would lead to a list with functions called `lglk_D`, `Dlglk_D`, and `reportees_D`.

Value

A list of three functions.

See Also

`get_Hbits`

finfo_onetype — *Expected Fisher Info from all CK comparisons of given type*

Description

This function computes the expected Fisher Information (ie negative of the Hessian) from a set of CK pairwise comparisons all of the same "type", but with many combinations of individual covariates. The "of the same type" means, for example, that this could be applied to cross-cohort half-sib comparisons between place A and place B where both samples have exact age measurements. A different "type" would be needed for, say, parent-offspring comparisons, or for half-sib comparisons where one animal has an inexact age, or...

The two ingredients are:

- derivs of sqrts of the CK probs for all covariate combinations, with respect to parameters;
- expected number of comparisons, for all the same covariate combinations.

The former comes from calling `get_Hbits` (qv). The number of comparisons is something you (partly) have to set up yourself, though there might be functions around to help.

Number of comparisons: If the indices of `ncomp` are `(j1,k1,...1,j2,k2,...2)` where `1` pertains to the first sample and `2` to the second, then in general

```
ncomp(j1,k1,...,j2,k2,...2) =
  nsamp(j1,k1,...1) * nsamp(j2,k2,...2) * const(j1,......)
```

where `const()` is usually 1 (include all such comparisons) or 0 (omit these even if they could be done). Sometimes it is 0.5, to avoid double-counting when `j1==j2 & k1==k2 & ...`, etc.

Note: `finfo_onetype` only deals with *one* type of CK comparison at a time (see its doco) whereas `get_Hbits` deals with all of them. Thus, if you have more than one type of CK comparison (say XHSPs as well as POPs), then you'll need multiple calls to `finfo_onetype` to get a Hessian from each type, which can then be added up. You can also add in Hessians stuff from (certain types of) non-CK data, such as age-composition samples and priors on parameters/latent variables.

Usage

```
finfo_onetype( dsp, ncomp)
```

Arguments

`dsp`	An (off)array with indices `(i,j,k,...)` where `i` refers to the i-th parameter, and `j,k,...` to all the covariates. `dsp[i,j,k,...]` is `D(sqrt(Pr_this_kintype|j,k,...))/D(params(i)` evaluated at (some guess at) the true parameter values.
`ncomp`	An (off)array with indices `(j,k,...)`, as for `dsp`.

Value

A square matrix with dimension `(n_params,n_params)`, where `n_params=length( dim( dsp)[1]))`.

See Also

get_Hbits

Examples

```
# See 'get_Hbits'
```

---

get_Hbits *Preparation for CK design Hessian calculation*

Description

`get_Hbits` prepares the ingredients needed to compute the expected Fisher Information ("negative Hessian") from any kinship comparison between two samples, depending on the covariate values of the samples and the type of kinship being considered. To calculate the overall Hessian, the results then need to be combined with proposed sample size information and added up, which is done (partly) by `finfo_onetype` (qv).

You can also use it, at the same time or later, to conveniently calculate other numerical derivatives of your CK model, eg of a log-likelihood (if you have real or simulated data) or of "quantities of interest" (for use in Delta-method later on). If your CK stuff is at all complicated, then it is generally cheaper to do a bunch of numerical derivs all at once. However, that's not compulsory. My most general pattern is:

- 1. call `get_Hbits` once just to get the prob derivs;
- 2. pick a sample-size scenario, and call `finfo_onetype` (as many times as required, once per different prob array) to get the overall Hessian for that scenario;
- 3. Some numderiv stuff for derivs of quantities-of-interest (for which I might again use `get_Hbits`);
- 4. Delta-method to combine #3 with #2.

I like the flexibility of being able to deal with quantities-of-interest after-the-fact; they do not affect the parameter Hessian. Also, they can often be calculated very quickly, without needing to compute all CK prob arrays. However, if you are sure you know what your QofIs are, then you *can* merge steps 1 & 3 into a single initial call to `get_Hbits`, which will save some time if your model is big and/or has lots of parameters.

In more detail: your CK code should compute at least one probability array, more if there are two types of kin or if some animals have qualitatively different covariates from others. `get_Hbits` automatically computes the derivatives of the square-roots of those probabilities. If your code also returns other (must be numerical) quantities, `get_Hbits` calculates the derivs of those *untransformed* things (i.e. no square root).

Usage

```
get_Hbits(
  PARS_FOR_H,
  all_probs_fun, Pargs= list(),
  numderiv_fun= mvbutils::numvbderiv_parallel,
  Dargs= list( eps=1e-6)
)
```

Arguments

`PARS_FOR_H`	"true" parameter values
`all_probs_fun`	function taking parameter vector as first argument, and returning a list of all CK prob arrays (or a list of any numeric things, actually; non-numeric elements are ignored). Prob arrays must have names starting "Pr". If you wrote your CK stuff in TMB(O), then `all_probs_fun` might just be `CK_logalike_from_TMBO_obj(...)$reportees`. Can optionally have other args...
`Pargs`	... which you set via this argument, eg `list(want=TRUE)` for functions obtained from `CK_logalike_from_TMB_obj`.
`numderiv_fun`	function that computes numerical derivatives of its first arg WRTO its second arg (a vector of parameters). The parameter dimension should be the last one in the result. Default should be OK, but don't blame me if it isn't; check its helpfile. See **Details**.
`Dargs`	optional list of extra args for `numderiv_fun`; for the default `numderiv_fun`, this might be `list( eps=1e-7)` to change the step, or `list( PARALLEL=FALSE)` if you have not set up a parallel cluster (which you should; `get_Hbits` can be slow).

Details

In this situation, where the dimension of the output (the nubmer of covariate-combinations in the CK probabilities) is large, numerical differentiation is almost as fast as Automatic Differentiation, and is completely general. The nice-sounding idea of using forward-mode AD apparently can't even be done in TMB! So, relax and just use numeric derivs.

The numerical derivatives don't have to be particularly accurate for this application. Currently, `get_Hbits` uses my `numvbderiv` which is very simple— and fast, if you use the parallel version, which is the default. However, it not as accurate or robust as if you used a special-purpose R package. Note that it will be called via `numderiv_fun( fun_to_diff, param_vals, <Dargs>)` and `fun_to_diff` must be a *function* taking a parameter vector as its first argument. Thus, if you wanted to use `stats::numericDeriv` (which I do not recommend), you'd have to write a wrapper for it, because it expects an `expression` argument not a function. Yawn. Boring!

Value

A list with components `DSP`, `Dnonprob`, `PARS_FOR_H`, and `Prkin`. The latter is the actual CK probabilities at `PARS_FOR_H`; it's a list, because there might be more than one type of CK probability. `DSP` is also a list, for the same reason; it's the derivs of the square-roots of the probabilities. `Dnonprob` is also a list (possibly empty) holdings the derivs of any non-probability returnees from all_probs_fun, such as an actual log-likelihood or some popdyn quantities; square-roots are *not* applied to such things. You can't really use the key return value, DSP, directly; it only makes sense in future calls to finfo_onetype.

See Also

finfo_onetype, CK_logalike_from_TMB_obj

Examples

```
## Not run:
if( require( TMBO)){ # offsets like offarray; better debugging
  compile( 'myCKex')
  dyn.load( dynlib( 'myCKex'))
  tmbob <- with( env, # so it knows about all the variables
    MakeADFun(
      data= returnList(
          Amat,
          n_MOP_BYA= array(NA),
          n_comp_MOP_BYA= array( NA)
        ),
      parameters= list( log_Nfad_ystart= starto[1], RoI= starto[2]),
      ranges= TMBO_ranges( years, Yad_range, Aad_range, Bju_range),
      DLL="lglk_POP_ideal_mammal",
      silent=TRUE
    ))
  tmbeq <- CK_logalike_funs( tmbob)
  Hbits <- get_Hbits( trupars, tmbeq$reportees)
  design_n_comp_MOP_BYA <- "something or other"
  H_MOPonly <- finfo_onetype(
      Hbits$DSP$DSP_MOP_BYA,
      design_n_comp_MOP_BYA)
  Vpar_MOPonly <- solve( H_MOPonly) # expected parameter covariances
}

## End(Not run)
```

Index