Interacting with the Nature index database and using custom uncertainty distributions

Jens Åström

2019-10-01

Contents

Implementing custom distributions

Until now, uncertainty of an indicator value could only be expressed in the Nature Index by its lower and upper quartiles. Together with the point estimate, these quartiles was then used to find a distribution that that was the best fit. This works well for many cases, and not so well for other cases. Preferably, we should be able to specify the uncertainty distribution attached to an indicator value, if such information exists. There are cases where this would be done as a known distribution, or as an empirical set of values, for example from a simulation or MCMC run.

In terms of the workings of the NIcalc package, the purpose of the uncertaintly of any indicator is to specify a distribution, from which we can draw samples to input into the NI calculations. The step of finding this distribution has been called the "elicitation" in the NI procedure. For the indicators where the user directly specifies the uncertainty distribution, we can sidestep the elicitation and instead sample directly from these given distributions.

To implement this in the `NIcalc` package, we make use of the `distr` package to generate various types of distributions. At this point, we allow for three types of distributions:

1. Named known distributions (Log-Normal and Poisson currently supported.)
2. Empirical distributions (Sample draws, e.g. CODA output from Bayesian modelling)
3. Discrete distributions (Allows for a fixed number of possible values, with individual probabilities specified)

The first option already have sampling functions in R (rnorm, rpois). However, to streamline the process, we handle all cases similarly, by the `distr` package. Currently, we still allow for the traditional specification of uncertainty as well, so the user has the choice of specifying the uncertainty of an indicator value as either lower and upper quartiles, or using one of these custom distributions.

Example of using known distributions

Log-Normal distribution

Let's say we have an indicator with a log point estimate of 0.5. The uncertainty is specified (in this example) as a lognormal distribution with standard error of 0.1. A user case might be if the value of an indicator is simulated or estimated as a lognormally distributed variable with a standard error. We can make a sampling function of this distribution quite easily using the `distr` package.

```
nMean <- 0.5
nSd <- 0.1

N <- distr::Lnorm(mean = nMean, sd = nSd)
```

For convenience, this has been implemented in the function `makeDistribution` in the `NIcalc` package. This function is used to create all types of custom uncertainties.

```
N <- makeDistribution(input = "logNormal", distParams = list(mean = 1,
    sd = 0.2))
```

To sample from the distribution, we use the function `distr::r` under the hood. We wrap this in a `NIcalc` function that we can use for any distribution.

```
sampleDistribution
```

```
## function(dist, nSamples = 10){
##   out <- distr::r(dist)(nSamples)
##   return(out)
## }
## <bytecode: 0xc45f138>
## <environment: namespace:NIcalc>
```

```
sampleDistribution(N, 5)
```

```
## [1] 2.290220 2.613898 2.616603 3.088406 2.246102
```

Poisson distribution

Another probable user case is an indicator value that is expressed as a Poisson variable. Examples might be simulations of a population count. In this case, the distribution can be expressed using only the λ variable.

```
P <- makeDistribution("Poisson", distParams = list(lambda = 5))
```

```
sampleDistribution(P, 10)
```

```
##  [1] 5 8 6 0 3 2 8 1 2 3
```

The distribution object (`N` or `P`) can be stored in the Nature index database together with the indicator value and later be sampled directly from. (More on the database interaction below.)

Using empirical distributions

Empirical and discrete distributions can be handled similarly. Suppose we have output from a Bayesian MCMC model for a parameter, and we wish to use this to sample from. We could of course just do a `sample(codaObject)`, but here we use the `distr` package instead to keep us in a unified framework. The result is the same.

```r
# Simulate an output from a MCMC run.
nSamples <- 1e+05
indicatorPosterior <- rnorm(nSamples, mean = 0.5, sd = 0.1)
EmpiricalPointEstimate <- mean(indicatorPosterior)
EmpiricalPointEstimate  #mean of these particular draws
```

```
## [1] 0.4997299
```

```r
E <- distr::EmpiricalDistribution(indicatorPosterior)

sampleDistribution(E)
```

```
##  [1] 0.4072440 0.3974919 0.5000233 0.4914164 0.4788282
##  [6] 0.6596619 0.7263859 0.5291102 0.3218219 0.5895508
```

The same done with `makeDistribution`, which can accept a vector of empirical samples.

```r
E <- makeDistribution(indicatorPosterior)
sampleDistribution(E)
```

```
##  [1] 0.5652393 0.4064271 0.4174688 0.3908744 0.5233045
##  [6] 0.5108637 0.5724286 0.6229933 0.4419648 0.4195048
```

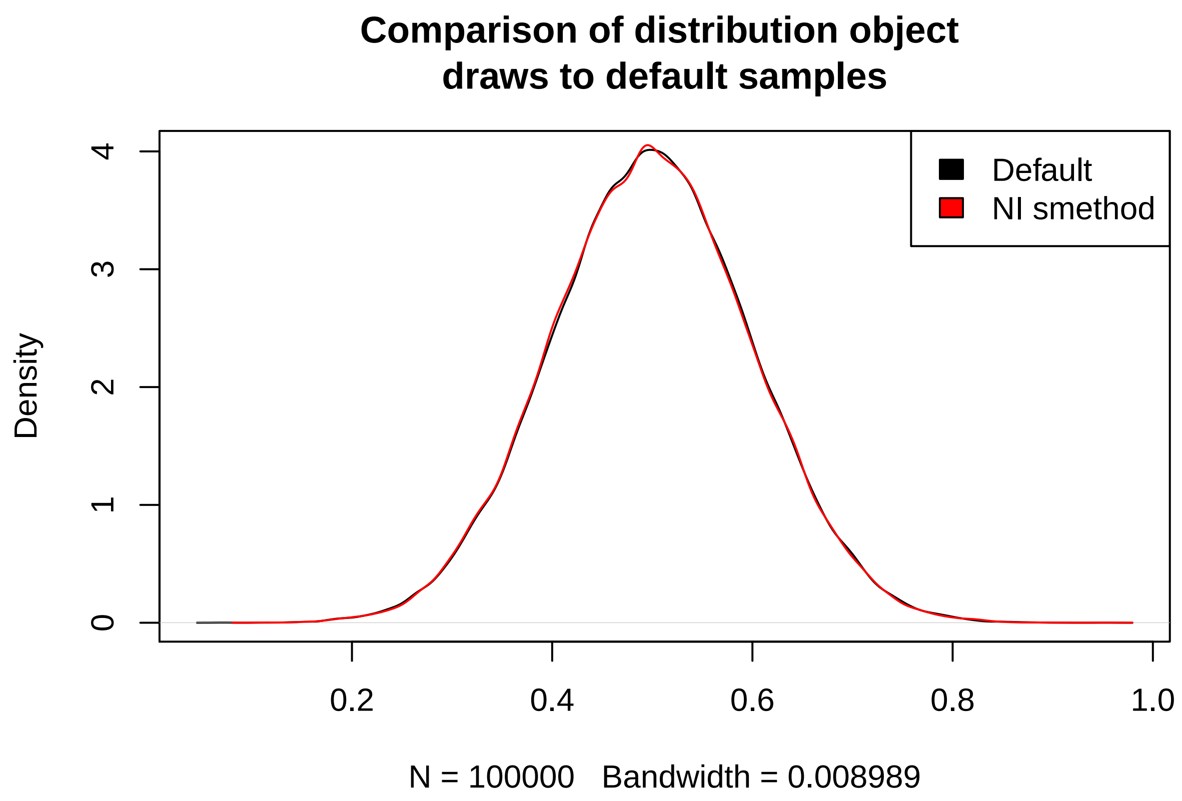
```r
E@q(0.5)  ##mean of the distribution, used internally to set the point estimate
```

```
## [1] 0.4995293
```

And just to confirm that sampling from this recently created object gives the (approximate) same answer as sampling from the original samples.

```r
nSamples <- 1e+05
sampleMethod <- indicatorPosterior[x = sample(1:length(indicatorPosterior),
    size = nSamples, replace = T)]

NImethod <- sampleDistribution(E, nSamples)
plot(density(sampleMethod), col = 1, main = "Comparison of distribution object \ndraws to default sampl
points(density(NImethod), col = 2, type = "l")
legend("topright", legend = c("Default", "NI smethod"), fill = 1:2)
```



In this case, we simply store the E object and retrive the point estimate from this distribution as shown above. The size of such an empirical distribution of 10^5 samples, is 0.68 Mb. This is manageable for the database.

Discrete distributions

A discrete distribution is handled much the same way. This is just a set of possible values, with associated probabilities for each value (For the empirical distribution, each individual value have the same probability of being drawn.)

```
allowedIndicatorValues <- c(0.1, 0.2, 0.3, 0.4)
indicatorValueProbabilities <- c(0.1, 0.4, 0.4, 0.1)

discretePointEstimate <- mean(allowedIndicatorValues)

D <- distr::DiscreteDistribution(allowedIndicatorValues, indicatorValueProbabilities)

sampleDistribution(D)
```

```
##  [1] 0.3 0.3 0.1 0.2 0.2 0.2 0.2 0.4 0.3 0.2
```

The same can be done via the `makeDistribution`.

```
allowedIndicatorValues <- c(0.1, 0.2, 0.3, 0.4)
indicatorValueProbabilities <- c(0.1, 0.4, 0.4, 0.1)

myProbs <- cbind(allowedIndicatorValues, indicatorValueProbabilities)
```

```r
D <- makeDistribution(myProbs)

sampleDistribution(D, 10)
```

```
##  [1] 0.3 0.1 0.3 0.3 0.2 0.3 0.2 0.3 0.2 0.4
```

Like before, we then store the `D` object together with the point estimate.

Storing the values and the `indicatorData`-class.

R objects cannot be stored directly in the database as they are. They can, however, be transformed into a raw data string and then stored. An R object cannot be stored as an element in an R data frame either. This means we have to handle these custom distribution objects a little differently than when we use only point estimates and lower and upper quartiles.

We use the `indicatorData` class to store the information in R of the indicator values. This object type is used to retrieve and send indicator values to the NI database. It is simply a list in two parts. The first is a dataframe mimicking the information in a database table containing the individual indicator values. The last column in this dataframe contains a unique identifier, refering to a custom distribution, for the indicator values where this is used. The second part of the list is another list containing the custom distributions related to the data frame in the first element. A user should interact with these types of objects mainly through dedicated functions in the package. This is exemplified below.

```r
# display an example data set here Doesn't go well with
# Rmarkdown!
str(indicatorData)
```

Working with the Nature Index database

We communicate with the Nature index database through a web based API. R has nice functionality for this through the `httr` and `jsonlite`-packages. The `NIcalc` package contains functions for communicating with the database where the heavy lifting is done by these packages.

We currently support the main task of updating/setting indicator values via a series of steps:

1. Connect to the database (using `getToken`)
2. Identify which indicators the user can alter (using `getIndicators`)
3. Retrieve current indicator values for a given indicator (using `getIndicatorValues`)
4. Setting new indicator values (using `setIndicatorValues`), in the same step, the `makeDistribution` function is called.
5. Writing new indicator values to the database (using `writeIndicatorValues`)

The procedure is designed to work with one indicator at a time. Although it is not recommended, it should also be possible to alter several indicators in the same function calls.

Example of updating values

We connect to the database using a "token" which is issued individually and temporarily for a connection. This is retrieved by providing a username and password to the database. Use the same credentials as for the web-page http://naturindeks.nina.no. See further information on that page regarding the username and passwords. The `getToken` function retrieves a token and assigns it the object name "niToken" which is used in further functions. Typically, the user only runs `getToken` in the beginning of a session and need not specify the token later on.

```r
exists("niToken")
```

```
## [1] "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJodHRwOi8vc2NoZW1hcy54bWxzb2FwLm9yZy93cy8yMDA1LzA1L2lkZW
```

```r
getToken("your.username", "secretPassword")

niToken
```

The user can the retrive a list of the indicator values he/she is responsible for, or is allowed to alter. This is done by the `getIndicators` function. The `id` variable is later used to identify the indicator we wish to work with.

```r
myIndicators <- getIndicators()
myIndicators
```

```
##    id                   name
## 1 351 Dagsommerfugler i skog
## 2 359           Humler i skog
```

The next typical step is to retrieve the current values for a given indicator. This can be done for all historical values in the database by simply:

```r
indicatorData <- getIndicatorValues(indicatorID = 351)
```

Usually, the user is only interested in the latest values and do not need to change values for earlier years. We can then specify the current year for which we want to set values (at the time of writing, 2018). In case the indicator ID's change, we can refer to the names of the indicators this way.

```r
indicatorData <- getIndicatorValues(indicatorID = myIndicators$id[myIndicators$name ==
    "Dagsommerfugler i skog"], year = 2018)
indicatorData
```

```
## $indicatorValues
##   indicatorId          indicatorName areaId
## 1         351 Dagsommerfugler i skog   7040
## 2         351 Dagsommerfugler i skog   7041
## 3         351 Dagsommerfugler i skog   7042
##                                   areaName yearId yearName
## 1    Dagsommerfugler skog Østfold Vestfold      9     2018
## 2 Dagsommerfugler skog Vest-Agder Rogaland      9     2018
## 3           Dagsommerfugler skog Trøndelag      9     2018
##   verdi nedre_Kvartil ovre_Kvartil datatypeId
## 1   0.9           0.7            1          2
## 2   0.3            NA           NA         NA
## 3   0.4            NA           NA         NA
##      datatypeName unitOfMeasurement
## 1 Overvåkingsdata         Enhetsløs
## 2            <NA>         Enhetsløs
## 3            <NA>         Enhetsløs
##                 customDistributionUUID distributionName
## 1                                 <NA>               NA
## 2 6ea2679c-b4bd-4f7a-8e5e-9f137261f257               NA
## 3 4c37b474-d3f9-4309-baed-6a7009feb327               NA
##   distributionId distParam1 distParam2
## 1             NA         NA         NA
## 2             NA         NA         NA
## 3             NA         NA         NA
##
## $customDistributions
## $customDistributions$`6ea2679c-b4bd-4f7a-8e5e-9f137261f257`
## Distribution Object of Class: DiscreteDistribution
##
## $customDistributions$`4c37b474-d3f9-4309-baed-6a7009feb327`
## Distribution Object of Class: DiscreteDistribution
##
##
## attr(,"class")
## [1] "indicatorData" "list"
```

We can set an indicator value using the `setIndivatorValue` function. Here, we use the standard upper and lower quartiles.

```
# setIndicatorValues(indicatorData, areaId = 7040, year =
# 2018, est = 0.9, lower = 0.7, upper = 1)

updatedIndicatorData <- setIndicatorValues(indicatorData, areaId = 7040,
    year = 2018, est = 0.9, lower = 0.7, upper = 1)

updatedIndicatorData
```

```
## $indicatorValues
##   indicatorId          indicatorName areaId
## 1         351 Dagsommerfugler i skog   7040
## 2         351 Dagsommerfugler i skog   7041
## 3         351 Dagsommerfugler i skog   7042
##                                 areaName yearId yearName
## 1    Dagsommerfugler skog Østfold Vestfold      9     2018
## 2 Dagsommerfugler skog Vest-Agder Rogaland      9     2018
## 3           Dagsommerfugler skog Trøndelag      9     2018
##   verdi nedre_Kvartil ovre_Kvartil datatypeId
## 1   0.9           0.7            1          2
## 2   0.3            NA           NA         NA
## 3   0.4            NA           NA         NA
##      datatypeName unitOfMeasurement
## 1 Overvåkingsdata         Enhetsløs
## 2            <NA>         Enhetsløs
## 3            <NA>         Enhetsløs
##                  customDistributionUUID distributionName
## 1                                  <NA>               NA
## 2 6ea2679c-b4bd-4f7a-8e5e-9f137261f257               NA
## 3 4c37b474-d3f9-4309-baed-6a7009feb327               NA
##   distributionId distParam1 distParam2
## 1             NA         NA         NA
## 2             NA         NA         NA
## 3             NA         NA         NA
##
## $customDistributions
## $customDistributions$`6ea2679c-b4bd-4f7a-8e5e-9f137261f257`
## Distribution Object of Class: DiscreteDistribution
##
## $customDistributions$`4c37b474-d3f9-4309-baed-6a7009feb327`
## Distribution Object of Class: DiscreteDistribution
##
##
## attr(,"class")
## [1] "indicatorData" "list"
```

We can also specify the values as inputs to the `makeDistribution` function. Here, the point estimate is automatically retrieved from the created distribution. In this example, we make a discrete distribution of three possible values.

```
updatedIndicatorData <- setIndicatorValues(updatedIndicatorData,
    areaId = 7041, year = 2018, distribution = cbind(c(0.1, 0.3,
        0.4), c(0.33, 0.33, 0.34)))

updatedIndicatorData <- setIndicatorValues(updatedIndicatorData,
    areaId = 7042, year = 2018, distribution = cbind(c(0.2, 0.4,
        0.5), c(0.33, 0.33, 0.34)))


updatedIndicatorData
```

```
## $indicatorValues
##   indicatorId          indicatorName areaId
## 1         351 Dagsommerfugler i skog   7040
## 2         351 Dagsommerfugler i skog   7041
## 3         351 Dagsommerfugler i skog   7042
##                                 areaName yearId yearName
## 1    Dagsommerfugler skog Østfold Vestfold      9     2018
## 2 Dagsommerfugler skog Vest-Agder Rogaland      9     2018
## 3           Dagsommerfugler skog Trøndelag      9     2018
##   verdi nedre_Kvartil ovre_Kvartil datatypeId
## 1   0.9           0.7            1          2
## 2   0.3            NA           NA         NA
## 3   0.4            NA           NA         NA
##      datatypeName unitOfMeasurement
## 1 Overvåkingsdata         Enhetsløs
## 2            <NA>         Enhetsløs
## 3            <NA>         Enhetsløs
##                  customDistributionUUID distributionName
## 1                                  <NA>               NA
## 2 8a4d308f-eccf-46f7-9255-9b12bc92d70a               NA
## 3 e1dac76c-0d38-4eec-9fb4-920e8b075f57               NA
##   distributionId distParam1 distParam2
## 1             NA         NA         NA
## 2             NA         NA         NA
## 3             NA         NA         NA
##
## $customDistributions
## $customDistributions$`6ea2679c-b4bd-4f7a-8e5e-9f137261f257`
## Distribution Object of Class: DiscreteDistribution
##
## $customDistributions$`4c37b474-d3f9-4309-baed-6a7009feb327`
## Distribution Object of Class: DiscreteDistribution
##
## $customDistributions$`8a4d308f-eccf-46f7-9255-9b12bc92d70a`
## Distribution Object of Class: DiscreteDistribution
##
## $customDistributions$`e1dac76c-0d38-4eec-9fb4-920e8b075f57`
## Distribution Object of Class: DiscreteDistribution
##
##
## attr(,"class")
## [1] "indicatorData" "list"
```

If we where to specify a Poisson distribution, the call would look something like this:

```
updatedIndicatorData <- setIndicatorValues(updatedIndicatorData,
    areaId = 7041, year = 2018, distribution = "logNormal", distParams = list(mean = 0.7,
        sd = 0.15))
```

If we would like to use an empirical distribution from for example a mcmc modelling run, we could do:

```
myCodasamples <- rnorm(1000)  ## toy example coda results

updatedIndicatorData <- setIndicatorValues(updatedIndicatorData,
    areaId = 7041, year = 2018, distribution = myCodasamples)
```

When you are satisfied with the new data, the next step is to write the new data to the database. For this, we use the `writeIndicatorValues` function.

```
writeIndicatorValues(updatedIndicatorData)
```

```
## Year 9 is closed for update
```

We can double check that the new values stuck by downloading them again. As we can se, we now get the updated values. Currently, potential old and now outdated distribution objects are not deleted. This may change in the future.

```
newValues <- getIndicatorValues(351, year = 2018)
newValues
```

```
## $indicatorValues
##   indicatorId          indicatorName areaId
## 1         351 Dagsommerfugler i skog   7040
## 2         351 Dagsommerfugler i skog   7041
## 3         351 Dagsommerfugler i skog   7042
##                                  areaName yearId yearName
## 1    Dagsommerfugler skog Østfold Vestfold      9     2018
## 2 Dagsommerfugler skog Vest-Agder Rogaland      9     2018
## 3           Dagsommerfugler skog Trøndelag      9     2018
##   verdi nedre_Kvartil ovre_Kvartil datatypeId
## 1   0.9           0.7            1          2
## 2   0.3            NA           NA         NA
## 3   0.4            NA           NA         NA
##      datatypeName unitOfMeasurement
## 1 Overvåkingsdata         Enhetsløs
## 2            <NA>         Enhetsløs
## 3            <NA>         Enhetsløs
##                  customDistributionUUID distributionName
## 1                                  <NA>               NA
## 2 6ea2679c-b4bd-4f7a-8e5e-9f137261f257               NA
## 3 4c37b474-d3f9-4309-baed-6a7009feb327               NA
##   distributionId distParam1 distParam2
## 1             NA         NA         NA
## 2             NA         NA         NA
## 3             NA         NA         NA
##
## $customDistributions
## $customDistributions$`6ea2679c-b4bd-4f7a-8e5e-9f137261f257`
## Distribution Object of Class: DiscreteDistribution
##
## $customDistributions$`4c37b474-d3f9-4309-baed-6a7009feb327`
## Distribution Object of Class: DiscreteDistribution
##
##
## attr(,"class")
## [1] "indicatorData" "list"
```