Distributed Calibration Belt

Stefano Finazzi

March 8, 2019

Brief documentation about code for computing the calibration belt in a distributed way. All the examples can be run using the dataset in `sample.Rdata` using `cbScript.r`.

1 Server side

`distributedCalibrationBelt()`

`distributedCalibrationBelt` implements the computations necessary to plot the calibration belt. Call `plot` on the object returned by `distributedCalibrationBelt` to make the plot. See documentation of `givitiR` package fro more info about the calibration belt functions.

Parameters

`o` A string with the name of the variable containing the binary outcomes. The elements must assume only the values 0 or 1. The predictions in variable `e` must represent the probability of the event coded as 1.

`e` A string with the name of the variable containing the predictions of the model under evaluation. The elements must be numeric and between 0 and 1.

`devel` A character string specifying if the model has been fit on the same dataset under evaluation (`internal`) or if the model has been developed on an external sample (`external`). See also the 'Details' section.

`subset` An optional string specifying the subset of observations to be considered in terms of variables present in the data frame (e.g., "age ¿ 70", "sex=='female'", etc...).

`server` the name(s) of the client(s) with the database on which the distributed analysis must run

`port` the port(s) on which the client(s) is/are listening

`confLevels` A numeric vector containing the confidence levels of the calibration belt. The default values are set to .80 and .95.

`thres` A numeric scalar between 0 and 1 representing 1 - the significance level adopted in the forward selection. By default is set to 0.95.

`maxDeg` The maximum degree considered in the forward selection. By default is set to 4.

`nPoints` A numeric scalar indicating the number of points to be considered to plot the calibration belt. The default value is 200.

Return An object of class `givitiCalibrationBelt`. After computing the calibration belt with the present function, the `plot` method can be used to plot the calibration belt. The object returned is a list that contains the following components:

`n` The size of the sample evaluated in the analysis, after discarding missing values from the vectors `o` and `e`.

`resultCheck` Result of the check on the data. If the data are compatible with the construction of the calibration belt, the value is the boolean `TRUE`. Otherwise, the element contain a character string describing the problem found.

`m` The degree of the polynomial at the end of the forward selection.

`statistic` The value of the test's statistic.

p.valueThe p-value of the test.

`seqP` The vector of the probabilities where the points of the calibration belt has been evaluated.

`minMax` A list with two elements named `min` and `max` representing the minimum and maximum probabilities in the model under evaluation

`confLevels` The vector containing the confidence levels of the calibration belt.

`intersByConfLevel` A list whose elements report the intervals where the calibration belt is significantly over/under the bisector for each confidence level in `confLevels`.

Details The calibration belt and the associated test can be used both to evaluate the calibration of the model in external samples or in the development dataset. However, the two cases have different requirements. When a model is evaluated on independent samples, the calibration belt and the related test can be applied whatever is the method used to fit the model. Conversely, they can be used on the development set only if the model is fitted with logistic regression.

Examples distibuted calibration belt with three sample clients all running on localhost an listening on ports 7000, 8000, and 9000.

```
cb <- distributedCalibrationBelt(
        o = 'hospOutcomeLatest_RIC10',
        e = 'probGiViTI_2017_Complessiva',
        devel = 'external',
        subset = c( 'centreCode=="a"',
                    'centreCode=="b"',
                    'centreCode=="c"' ),
        server = c( 'localohost' )
        port = c( 7000, 8000, 9000 ),
        nPoints = 20 )
cb

$n
[1] 722

$m
[1] 1

$statistic
    Stat
2.670887
```

```
$p.value
[1] 0.2630415

$seqP
 [1] 0.001854899 0.001854899 0.007438237 0.029333790 0.108628287
    0.112548303 0.223241707 0.329506743 0.333935111 0.444628515
[11] 0.555321919 0.664629986 0.666015323 0.776708726 0.887402130
   0.888786858 0.969904494 0.992364141 0.998095534 0.998095534

$minMax
$minMax$min
[1] 0.001854899

$minMax$max
[1] 0.9980955


$confLevels
[1] 0.95 0.80

$cbBoundByConfLevel
$cbBoundByConfLevel[[1]]
               L           U
1  0.0001906998 0.002764402
2  0.0001906998 0.002764402
3  0.0013326634 0.010160797
4  0.0091492011 0.036795652
5  0.0588534124 0.128438312
6  0.0619072775 0.132960059
7  0.1606245650 0.263679814
8  0.2668774938 0.396775661
9  0.2716899765 0.402458791
10 0.3844393441 0.544736685
11 0.4977109753 0.679921057
12 0.6109571488 0.797201886
13 0.6124095733 0.798554071
14 0.7306714952 0.894142464
15 0.8555347877 0.963057980
16 0.8571649298 0.963737328
17 0.9571380172 0.994618300
18 0.9880078070 0.999231892
19 0.9967036804 0.999890716
20 0.9967036804 0.999890716

$cbBoundByConfLevel[[2]]
               L           U
1  0.0002856916 0.002021885
2  0.0002856916 0.002021885
3  0.0018131627 0.008019063
4  0.0113226708 0.031506072
5  0.0663501607 0.117495167
6  0.0695987200 0.121781385
```

```
7  0.1728734714 0.248715004
8  0.2829895643 0.378505568
9  0.2878661505 0.383848539
10 0.4047989242 0.522597591
11 0.5220343303 0.656222095
12 0.6379229335 0.775237649
13 0.6393944997 0.776635094
14 0.7573796549 0.877590100
15 0.8764810776 0.954540854
16 0.8779835673 0.955330704
17 0.9664624067 0.992666141
18 0.9913364180 0.998844247
19 0.9978053120 0.999818704
20 0.9978053120 0.999818704


$intersByConfLevel
$intersByConfLevel[[1]]
$intersByConfLevel[[1]]$overBisector
list()

$intersByConfLevel[[1]]$underBisector
list()


$intersByConfLevel[[2]]
$intersByConfLevel[[2]]$overBisector
list()

$intersByConfLevel[[2]]$underBisector
list()



attr(,"class")
[1] "givitiCalibrationBelt"
```

`getFunctionPartialResults()`

`getFunctionPartialResults()` implements the interface between central server and clients on the server side. It sends `GET` requests to a single and return the `R` object computed by the client.

```
getFunctionPartialResults <- function( server, port, fun, ... ){
  urlString <- url_compose(
    list(
      scheme = 'http',
      domain = server,
      port = port,
      path = fun,
      parameter = NA,
      fragment = NA )
  )

  functionParameters <- list( ... )

  for( par in names( functionParameters ) ){
    value <- functionParameters[[ par ]]
    if( is.numeric( value ) ){
      value <- toJSON( value, digits = 15 )
    }
    urlString <- param_set(
      urlString,
      par,
      url_encode( value ) )
  }

  httrObj <- GET( urlString, timeout( 60 ) )

  return(
    content(
      httrObj,
      encoding = 'utf8',
      as= 'parsed',
      simplifyVector = TRUE
    )
  )
}
```

Parameters

`server` the name of the client with the database on which the distributed analysis must run.

`port` the port on which the client is listening

`fun` the name of the method that must be run on the client.

`...` any parameter that must be passed to `fun` on the client.

Return An `R` object containing the result of the computation on the client. The result is converted in `JSON` on the client and the parsed back to an `R` object before returning. This object typically contains a value (a number, a vector, or a matrix, depending on the type of function `fun`) and, possibly, a `dataInfo` object which contain additional information about the computation performed by the client.

`n` number of records used in the computation

`minE` the minimum value of the expected probability (e.g., the minimum value of the variable `e`).

`minE` the maximum value of the expected probability.

`o` the name of the variable containing the binary outcome.

`e` the name of the variable containing the outcome probability.

`subset` the string with record selection.

Examples

1. Computation of partial loglikelihood on localhost, port 7000, calling client method `partiallogLikelihood` with parameters `o`, `r`, `subset`, and `beta`

```
results <- getFunctionPartialResults(
        server = 'localhost',
        port = 7000,
        fun = 'partiallogLikelihood',
        o = 'hospOutcomeLatest_RIC10',
        e = 'probGiViTI_2017_Complessiva',
        subset = 'centreCode=="a"',
        beta = c( 0, 1 )
)

results

$logLik
[1] -83.45772

$dataInfo
$dataInfo$n
[1] 296

$dataInfo$minE
[1] 0.00209055

$dataInfo$maxE
[1] 0.993224

$dataInfo$o
[1] "hospOutcomeLatest_RIC10"

$dataInfo$e
[1] "probGiViTI_2017_Complessiva"

$dataInfo$subset
[1] "centreCode==\"a\""
```

2. Computation of partial jacobian of loglikelihood on localhost, port 7000, calling client method `partialjacLogLikelihood` with parameters `o`, `r`, `subset`, and `beta`

```
results <- getFunctionPartialResults(
        server = 'localhost',
        port = 7000,
```

```
        fun = 'partialjacLogLikelihood',
        o = 'hospOutcomeLatest_RIC10',
        e = 'probGiViTI_2017_Complessiva',
        subset = 'centreCode=="a"',
        beta = c( 0, 1 )
)

results

          [,1]     [,2]
[1,] -4.135052 17.6103
```

3. Computation of partial hessian matrix of loglikelihood on localhost, port 7000, calling client method `partialjacLogLikelihood` with parameters `o`, `r`, `subset`, and `beta`

```
results <- getFunctionPartialResults(
        server = 'localhost',
        port = 7000,
        fun = 'partialhessianLogLikelihood',
        o = 'hospOutcomeLatest_RIC10',
        e = 'probGiViTI_2017_Complessiva',
        subset = 'centreCode=="a"',
        beta = c( 0, 1 )
)

results

          [,1]      [,2]
[1,]  31.52494 -26.66524
[2,] -26.66524  82.00049
```

`getFunctionResults()`

`getFunctionResults` calls a method on several clients using `getFunctionPartialResults` and aggregate the results using `joinDataInfo`

```
getFunctionResults <- function( server, port, fun, ... ){
  parameters <- list( ... )
  if( 'subset' %in% names( parameters ) ){
    subset <- parameters$subset
  }else{
    subset <- TRUE
  }
  connectionParameters <- c( 'server', 'port', 'subset' )
  nconnectionParameters <- sapply( connectionParameters, function(x)
     length( get(x) ) )
  ncall <- max( nconnectionParameters )
  if( ! all( nconnectionParameters %in% c( 1, ncall ) ) ){
    stop( 'incompatible number of connection parameters' )
  }else{
    for( var in connectionParameters ){
      assign( var, rep( get( var), length.out = ncall ) )
    }
  }

  results <- lapply(
    1:ncall,
    function(i)
      do.call(
        getFunctionPartialResults,
        c(
          list(
            server = server[i], port = port[i],
            fun = fun,
            subset = subset[i]
          ),
          parameters[ setdiff( names( parameters ), 'subset' ) ]
        )
      )
  )

  if( all( sapply( results, is.numeric ) ) ){
    result <- Reduce( '+', results )
  }

  if( all( sapply( results, function(x) 'dataInfo' %in% names( x ) ) )
      ){
    result <- results[[1]]
    if( ncall > 1 ){
      for( res in results[-1] ){
        if( !identical( names( result ), names( res ) ) ){
          stop( 'incompatible results from servers' )
        }else{
          result$dataInfo <- joinDataInfo( result$dataInfo, res$
             dataInfo )
          for( var in setdiff( names( result ), 'dataInfo' ) ){
```

```
            result[[ var ]] <- result[[ var ]] + res[[ var ]]
          }
        }
      }
    }
  }

  return( result )
}
```

Parameters

`server` the name(s) of the client(s) with the database(s) on which the distributed analysis must run.

`port` the port(s) on which the client(s) is/are listening

`fun` the name of the method that must be run on the client.

`subset` if present, a string or a vector of strings containing a selection of records to be used on each client for the calculation. The selection of patient can be different for each client.

`...` any other parameter that must be passed to `fun` on the clients.

Return An `R` object containing the aggregated result of the computation on the clients. This object typically contains a value (a number, a vector, or a matrix, depending on the type of function `fun`), which is the sum of the values returned by each client and, possibly, a `dataInfo` object which contain additional information about the computation performed by the clients, which is aggregated as follows

`n` sum of numbers of records used in the computation by each client

`minE` the minimum value of the minimum expected probabilities (e.g., the minimum value of the variable `e` returned by the clients

`minE` the maximum value of the maximum expected probabilities returned by the clients.

`o` the name of the variable containing the binary outcome (it must be identical on all clients).

`e` the name of the variable containing the outcome probability (it must be identical on all client).

`subset` a vector containing the strings with the rules used to select records on each client.

Examples

1. Computation of loglikelihood on three clients listening on localhost on ports 7000, 8000, 9000 calling client method `partiallogLikelihood` with different patient selection `subset` for each client and parameters `o`, `r`, and `beta`

```
results <- getFunctionResults(
        server = 'localhost',
        port = c( 7000, 8000, 9000 ),
        fun = 'partiallogLikelihood',
        o = 'hospOutcomeLatest_RIC10',
        e = 'probGiViTI_2017_Complessiva',
        subset = c( 'centreCode=="a"', 'centreCode=="b"', '
           centreCode=="c"' ),
        beta = c( 0, 1 )
)

results
```

```
$logLik
[1] -224.9555

$dataInfo
$dataInfo$n
[1] 722

$dataInfo$minE
[1] 0.001854899

$dataInfo$maxE
[1] 0.9980955

$dataInfo$o
[1] "hospOutcomeLatest_RIC10"

$dataInfo$e
[1] "probGiViTI_2017_Complessiva"

$dataInfo$subset
[1] "centreCode==\"a\"" "centreCode==\"b\"" "centreCode==\"c\""
```

2. Computation of partial jacobian of loglikelihood on three clients listening on localhost on ports 7000, 8000, 9000 calling client method `partialjacLogLikelihood` with different patient selection `subset` for each client and parameters `o`, `r`, and `beta`

```
results <- getFunctionResults(
        server = 'localhost',
        port = c( 7000, 8000, 9000 ),
        fun = 'partialjacLogLikelihood',
        o = 'hospOutcomeLatest_RIC10',
        e = 'probGiViTI_2017_Complessiva',
        subset = c( 'centreCode=="a"', 'centreCode=="b"', '
           centreCode=="c"' ),
        beta = c( 0, 1 )
)

results

          [,1]     [,2]
[1,] -1.35116 19.56837
```

3. Computation of partial hessian matrix of loglikelihood on three clients listening on localhost on ports 7000, 8000, 9000 calling client method `partialjacLogLikelihood` with different patient selection `subset` for each client and parameters `o`, `r`, and `beta`

```
results <- getFunctionResults(
        server = 'localhost',
        port = c( 7000, 8000, 9000 ),
        fun = 'partialhessianLogLikelihood',
        o = 'hospOutcomeLatest_RIC10',
        e = 'probGiViTI_2017_Complessiva',
        subset = c( 'centreCode=="a"', 'centreCode=="b"', '
           centreCode=="c"' ),
        beta = c( 0, 1 )
```

```
)

results

          [,1]      [,2]
[1,]  76.3858 -65.7878
[2,] -65.7878 196.8707
```

2 Client side

`partiallogLikelihood`, `partialhacLogLikelihood`, `partialhessianLikelihood` implement on the client the `GET` method running the functions required for the distributed computation of the calibration belt. These methods accept parameters as described in the above section (see `getFunctionPartialResults()`) and return the results of `logLikelihood`, `jacLogLikelihood`, `hessianLogLikelihood`, respectively in a `JSON` object.

All these methods call the function `computeLikelihoodFunction`, specifying the name of the function that must be run on the client as follows

computeLikelihoodFunction()

This function has been modified to take as argument also a dataset `data` which is no longer loaded here through `loadDataIntoGlovalEnvir`. The following two lines have been removed

```
loadDataIntoGlovalEnvir(
    variables = c( o, e ), subset = subset, objectName = 'dataTable' )

data <- dataTable
```

```
computeLikelihoodFunction <- function(
  funName, o, e, subset, beta,
  addDataInfo = FALSE, resultName = 'value' ){

  print( paste( 'Requested', funName, 'with␣o:', o, 'e:', e, 'beta:',
     beta, 'subset:', subset ) )

  if( is.character( beta ) ){
    beta <- fromJSON( beta )
  }
  m <- length( beta ) - 1

  loadDataIntoGlovalEnvir(
    variables = c( o, e ), subset = subset, objectName = 'dataTable' )

  data <- dataTable
  x <- outer( logit( data[ , e ] ), 0:m, '^' )

  result <- get( funName )( beta = beta, x = x, o = data[ , o ] )

  if( addDataInfo ){
    dataInfo <- list(
      n = nrow( data ),
      minE = min( data[ , e ] ),
      maxE = max( data[ , e ] ),
      o = o,
      e = e,
      subset = subset
    )

    result <- list(
      result,
      dataInfo = dataInfo
    )
```

```
    names( result )[1] <- resultName
  }

  return( toJSON( result, digits = 20 ) )
}
```

Parameters

`funName` the name of the function that must be run on the client.

`o` A string with the name of the variable containing the binary outcomes. The elements must assume only the values 0 or 1. The predictions in variable `e` must represent the probability of the event coded as 1.

`e` A string with the name of the variable containing the predictions of the model under evaluation. The elements must be numeric and between 0 and 1.

`subset` An optional string specifying the subset of observations to be considered in terms of variables present in the data frame (e.g., "age ¿ 70", "sex=='female'", etc...).

`beta` a vector (can be in `JSON` from) with the parameters to be passed to the client function

`addDataInfo` a flag indicating if a structure of the form `dataInfo` (see previous section) must be appended to the output

`resultName` the name of the value that is returned by the function.

Return A `JSON` object containing the result of the computation of the function `funName`. It may contain a `dataInfo` object if `addDataInfo` is set to `TRUE`.

Examples

1. Computation of loglikelihood with flag `addDataInfo`

```
JSONresults <- computeLikelihoodFunction(
        funName = 'logLikelihood',
        o = 'hospOutcomeLatest_RIC10',
        e = 'probGiViTI_2017_Complessiva',
        subset = 'centreCode==\"a\"',
        beta = '[0,1]',
        addDataInfo = TRUE,
        resultName = 'logLik'
)

JSONresults

{"logLik":[-83.4577217358995],"dataInfo":{"n":[296],"minE"
   :[0.00209054974152624],"maxE":[0.993224047123833],"o":["
   hospOutcomeLatest_RIC10"],"e":["probGiViTI_2017_Complessiva"],"
   subset":["centreCode==\"a\""]}}
```

2. Computation of partial jacobian of loglikelihood

```
JSONresults <- computeLikelihoodFunction(
        funName = 'jacLogLikelihood',
        o = 'hospOutcomeLatest_RIC10',
        e = 'probGiViTI_2017_Complessiva',
```

```
        subset = 'centreCode==\"a\"',
        beta = '[0,1]'
)

JSONresults

[[-4.13505231542442,17.6102991538352]]
```

3. Computation of partial hessian matrix of loglikelihood

```
JSONresults <- computeLikelihoodFunction(
        funName = 'hessianLogLikelihood',
        o = 'hospOutcomeLatest_RIC10',
        e = 'probGiViTI_2017_Complessiva',
        subset = 'centreCode==\"a\"',
        beta = '[0,1]'
)

JSONresults

[[31.524944152993,-26.6652442881833],[-26.6652442881833,82.0004925171426]]
```