Imputation of Non-Livestock Commodities

Author: Josh Browning, Michael C. J. Kao, Francesca Rosa

Description:

This module imputes the triplet (input, productivity and output) for a given non-livestock commodity.

Inputs:

- Production domain: domain= `àgriculture`, dataset: `aproduction`
- Complete Key Table: datatable= `fbs_production_comm_codes`
- Livestock Element Mapping Table: datatable= `animal_parent_child_mapping`
- Identity Formula Table: datatable= `item_type_yield_elements`

Flag assignment:

1. Balance by Production Identity: ObsFlag= `<flag aggregation>`; methFlag= `i`
2. Imputation via ensemble approach: ObsFlag= `I`; methFlag= `e`

Data scope

- GeographicAreaM49: All countries specified in the `Complete Key Table`.
- measuredItemCPC: Depends on the session selection. If the selection is "session", then only items selected in the session will be imputed. If the selection is "all", then all the items listed in the `Complete Key Table` excluding the live stock item in the `Livestock Element Mapping Table` will be imputed.
- measuredElement: Depends on the measuredItemCPC, all cooresponding elements in the `Identity Formula Table`.
- timePointYears: All years specified in the `Complete Key Table`.

Initialisation

Load the libraries:

```r
suppressMessages({
    library(faosws)
    library(faoswsUtil)
    library(faoswsFlag)
    library(faoswsImputation)
    library(faoswsProduction)
    library(faoswsProcessing)
    library(faoswsEnsure)
    library(magrittr)
    library(dplyr)
    library(sendmailR)
})
```

Set up for the test environment and parameters:

```
R_SWS_SHARE_PATH = Sys.getenv("R_SWS_SHARE_PATH")

if(CheckDebug()){

    library(faoswsModules)
    #SETTINGS = ReadSettings("modules/impute_non_livestock/sws.yml")
SETTINGS = ReadSettings("sws.yml")
    ## If you're not on the system, your settings will overwrite any others
    R_SWS_SHARE_PATH = SETTINGS[["share"]]

    ## Define where your certificates are stored
    SetClientFiles(SETTINGS[["certdir"]])

    ## Get session information from SWS. Token must be obtained from web interface

    GetTestEnvironment(baseUrl = SETTINGS[["server"]],
                       token = SETTINGS[["token"]])
}
```

Get user specified imputation selection:

```
imputationSelection = swsContext.computationParams$imputation_selection
imputationTimeWindow = swsContext.computationParams$imputation_timeWindow
```

Check the validity of the computational parameter:

1. Get data configuration and session
2. Build processing parameters
3. Get the full imputation Datakey

```
if(!imputationSelection %in% c("session", "all"))
    stop("Incorrect imputation selection specified")


sessionKey = swsContext.datasets[[1]]
datasetConfig = GetDatasetConfig(domainCode = sessionKey@domain,
                                 datasetCode = sessionKey@dataset)


processingParameters =
    productionProcessingParameters(datasetConfig = datasetConfig)


completeImputationKey = getCompleteImputationKey("production")
```

Since the animal/meat are currently imputed by the imputed_slaughtered and synchronise slaughtered module, so they should not be excluded here.

```
liveStockItems =
    getAnimalMeatMapping(R_SWS_SHARE_PATH = R_SWS_SHARE_PATH,
                         onlyMeatChildren = FALSE)
    liveStockItems = liveStockItems[,.(measuredItemParentCPC, measuredItemChildCPC)]
    liveStockItems = unlist(x = liveStockItems, use.names = FALSE)
    liveStockItems = unique(x = liveStockItems )
```

This is the complete list of items that are in the imputation list:

```
nonLivestockImputationItems =
    getQueryKey("measuredItemCPC", completeImputationKey) %>%
    setdiff(., liveStockItems)
```

These are the items selected by the users:

```
sessionItems =
    getQueryKey("measuredItemCPC", sessionKey) %>%
    intersect(., nonLivestockImputationItems)
```

Select the commodities based on the user input parameter:

```
selectedItemCode =
    switch(imputationSelection,
           "session" = sessionItems,
           "all" = nonLivestockImputationItems)
```

Perform Imputation Before starting looping through the commodities to perform the imputation of missing data, we prepare the envirnment:

1. Create an empty dataset to host those items for which the imputation process fails
2. Define the imputation lastYear
3. In case we are working locally it might be useful to deviate the console output on a txt file that can be browsed and used for validation purposes.

```
imputationResult = data.table()

lastYear=max(as.numeric(completeImputationKey@dimensions$timePointYears@keys))

logConsole1=file("log.txt",open = "w")
sink(file = logConsole1, append = TRUE, type = c( "message"))
```

Loop through the commodities to impute the items individually.

```
for(iter in seq(selectedItemCode)){

        imputationProcess =try({
            set.seed(070416)

            currentItem = selectedItemCode[iter]


            formulaTable =
                getProductionFormula(itemCode = currentItem) %>%
                removeIndigenousBiologicalMeat(formula = .)

            if(nrow(formulaTable) > 1)
                stop("Imputation should only use one formula")
```

Create the formula parameter list: all the routines are generic, this means that we have build some processing parameters, containing for example the column names or the flags to be used for imputed figured, and all the functions point to those generic objects.

```
    formulaParameters =
        with(formulaTable,
             productionFormulaParameters(datasetConfig = datasetConfig,
                                         productionCode = output,
                                         areaHarvestedCode = input,
                                         yieldCode = productivity,
                                         unitConversion = unitConversion)
             )
```

Update the item/element key according to the current commodity:

```
    subKey = completeImputationKey
    subKey@dimensions$measuredItemCPC@keys = currentItem
    subKey@dimensions$measuredElement@keys =
        with(formulaParameters,
             c(productionCode, areaHarvestedCode, yieldCode))
```

Start the imputation:

1. Build imputation parameter (similar to processing paramenters but containes all the `objects` to perfome the ensemble appoach)
2. Extract the data.
3. Skip the imputation if the data contains no entry.

```
    message("Imputation for item: ", currentItem, " (",  iter, " out of ",
            length(selectedItemCode),")")

    imputationParameters =
        with(formulaParameters,
             getImputationParameters(productionCode = productionCode,
                                     areaHarvestedCode = areaHarvestedCode,
                                     yieldCode = yieldCode)
             )


    extractedData =
        GetData(subKey)


    if(nrow(extractedData) == 0){
        message("Item : ", currentItem, " does not contain any data")
        next
    }
```

Data processing

1. `preProcessing`
2. `removeNonProtectedFlag`

Data pulled from the SWS has to be pre-processed in order to be used in the module. In particular we have to replace zero values with a missing flag (`M`) with `NA` values. This is an issue from previous data: some observations will be labeled as missing but given a value of zero instead of a value of NA.

In addition the `preProcessing` function transforms the `timePointYear` colum into a numeric column (instead of `character`).

At this point we have to remove from the data all those figures that have to be overwritten by the new imputations. The first step of the imputation is to remove any previous imputations. Even when using the same methodology and settings, prior imputations will change as more information is received over time. This means that we have to decide which fiures have to be removed and overwritte.

This important choice depends on flags. The list `flagValidTable` contains all the flag combinations (ObsFlag, methFlag) that have been considered realible enough to be kept and to be considered the basis to produce new imputations.

To remove the prior imputations, one will need to specify the column name of the value and corresponding observation and method flags. Further, the character value which represents an imputation and the character value for a flag representing missing values must be provided. The function will convert the previously imputed values to NA and the flags from previous imputations will be set to the missing flag value.

```
   processedData =
       extractedData %>%
       preProcessing(data = .)


 if(imputationTimeWindow=="all"){processedData=removeNonProtectedFlag(processedData)}else{
     processedData=removeNonProtectedFlag(processedData, keepDataUntil = (lastYear-2))}



  processedData =  denormalise(normalisedData = processedData,
                               denormaliseKey = "measuredElement",
                               fillEmptyRecords = TRUE)

  processedData = createTriplet(data = processedData, formula = formulaTable)
```

The following line of code allows to manipulate the `imputationParameter` in order to produce charts to visualize the ensemble approach output.

In particular the graphs contain a lot of information. First, the dots represent observed values, and the crosses represent the imputations. The different colored lines show the different fits, and the thickness of the line is proportional to the weight it received in the ensemble. Of course, if the data point is an observation then no imputation is done, so all lines have the same thickness there. Also, the computed weights will be constant for all imputed values with one exception: models that are not allowed to extrapolate may have positive weights for some imputations and 0 for others. Moreover, if an observation is outside the extrapolation range of a model, then the weight of all other models will need to be rescaled so all values add to 1.

We report an example to interpret the graph.

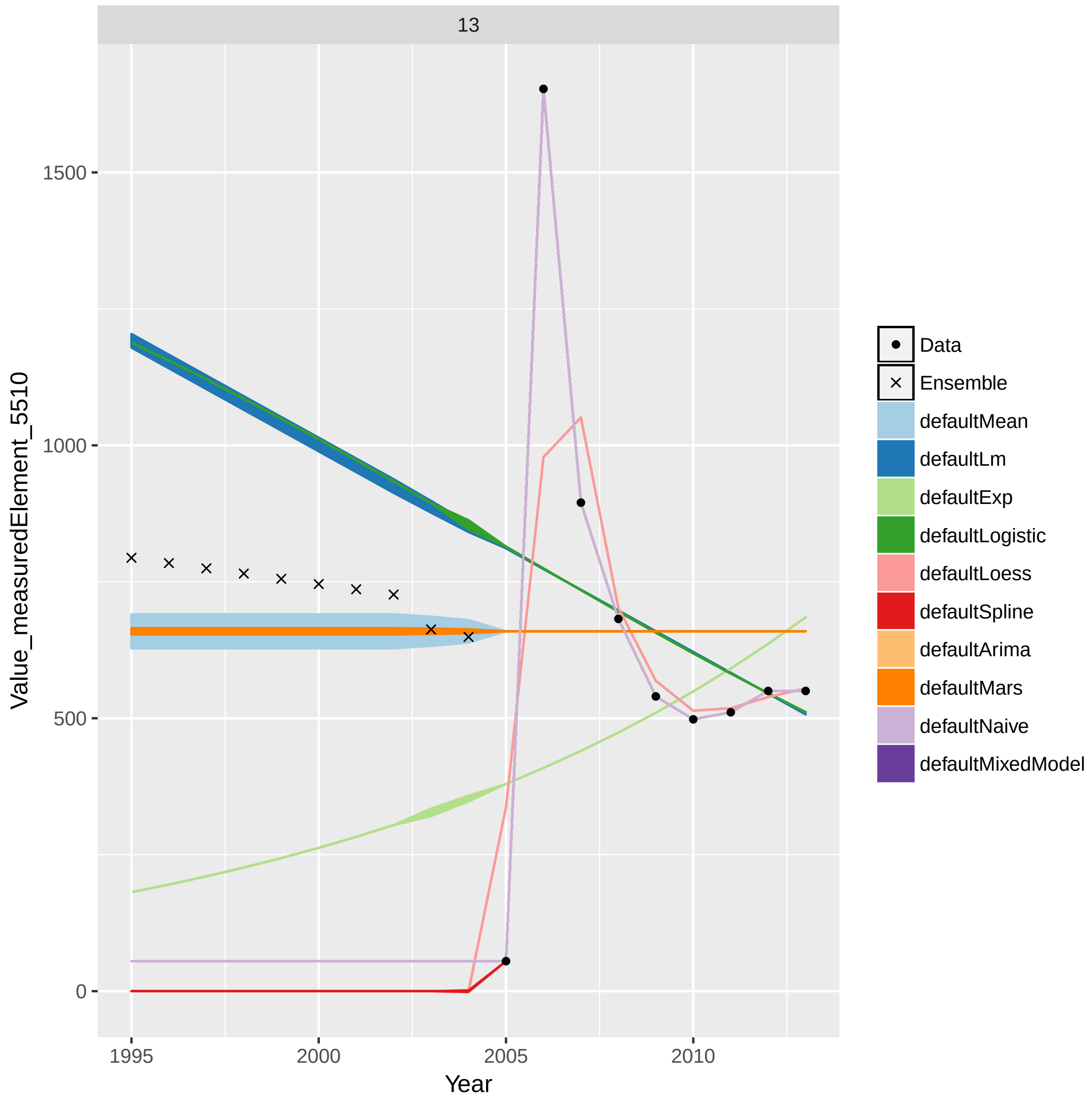

Figure 1: We see that the purple line (the model corresponding to our naive model which always estimates the value 10) rarely gets any weight. This makes sense, as it's not a very good model. However, in some particular cases (i.e. geographicAreaM49 = 13), no models do very well. The mean model thus gets most of the weight, but our naive model also gets a little weight. You can also see how it only has weight up to 5 observations outside of the range of the data; this is because we gave the model an extrapolation range of 5.

```
imputationParameters$productionParams$plotImputation="prompt"
```

We have to remove (M,-) from yield: since yield is usually computed ad identity, it results inusual that it exists a last available protected value different from NA and when we perform the function expandYear we risk to block the whole time series. We replace all the (M,-) in yield with (M,u) the triplet will be sychronized by the imputeProductionTriplet function.

```
processedData[get(formulaParameters$yieldObservationFlag)==
                  processingParameters$missingValueObservationFlag,
              ":="(c(formulaParameters$yieldMethodFlag),
                   list(processingParameters$missingValueMethodFlag)) ]



 processedData= normalise(processedData)
processedData=expandYear(data = processedData,
           areaVar = processingParameters$areaVar,
           elementVar = processingParameters$elementVar,
           itemVar = processingParameters$itemVar,
           valueVar = processingParameters$valueVar,
           newYears= lastYear)
processedData= denormalise(processedData, denormaliseKey = "measuredElement" )
```

Impute Production Triplet

Now we are ready to perform the imputation. The function `imputeProductionTriplet` is a wrapper containing the imputation process af the three elements composing the triplet. The triplet is imputed starting from `yield` and at each imputation step the other two variables are computed as identity (where possible). The function imputeVariable allows the user to perform imputation on the dataset, and it accepts a list of imputation parameters which control how the imputation is done. To run the imputation, we need to construct a list with the default imputation parameters (similar to the list with the processing parameters) and adjust them as necessary for our specific use case. The documentation page for defaultImputationParameters() provides some detail on what each of the different elements of this list are.

One very important part of this list is the ensembleModels element. This element specifies all of the models which should be used to form the final ensemble. By default, eleven models are used:

1. "defaultMean"
2. "defaultLogistic"
3. "defaultArima"
4. "defaultMixedModel"
5. "defaultLm"
6. "defaultLoess"
7. "defaultMars"
8. "defaultExp"
9. "defaultSpline"
10. "defaultNaive"
11. "defaultMa"

You can also manually create your own model for use. See the documentation page for "?ensembleModel for more details, and below for an example":

```
newModel = ensembleModel(
    model = function(data){
        rep(10, length(data))
    },
    extrapolationRange = 5,
    level = "local")
is(newModel)
imputationParams$ensembleModels = c(imputationParams$ensembleModels,
                                   newModel = newModel)
names(imputationParams$ensembleModels)
```

This new model returns a constant prediction of 10. It's not a good model, but it's a simple example of how to create a new model. The extrapolation range specifies that the model can be used in an ensemble up to 5 observations outside the range of the data, but no more. The level argument specifies that the model should operate on data for all countries for a fixed commodity (as opposed to one model for each unique country-commodity pair).

```
            imputed =
                imputeProductionTriplet(
                    data = processedData,
                    processingParameters = processingParameters,
                    formulaParameters = formulaParameters,
                    imputationParameters = imputationParameters)
```

For example after the imputation of yield, we proceed to impute the production. The function "imputeVariable" is used again, but we first need to impute by "balancing," i.e. updating missing values of production when yield and area harvested both exist. This is because we have the relationship:

$$Y = P/A$$

where Y is yield, P is production, and A is the area harvested. If no value for area harvested is available, then the function proceeds to impute the remaining production values with ensemble learning. This balancing is handled in the faoswsProduction package (which depends on this imputation package). To avoid dealing with strange dependency issues, we'll simply ignore this relationship here.

Note: imputations that are interpolations are always present, but some extrapolations are not imputed. The reason for this is that some models are not reasonable to extrapolate with (such as LOESS, Splines, etc.). For these models, an "extrapolation range" is defined, and this value dictates how far outside the range of the data a particular model is allowed to extrapolate. In some cases, no models are valid at a certain range and thus no imputation is performed. To avoid these kinds of issues, we recommend including a simple model that will rarely fail and that can be used to extrapolate (for example, defaultMean or defaultLm). Such models will ensure that most values are imputed.

Thus, the Logistic, Arima, and Mars models are the only models that are allowed to extrapolate more than one observation away from the data. For most of the examples provided here, those three models all failed to fit to the data, and so imputations were not available.

By now we did not have touched those triplt-configuration in which production or areaHarvested is ZERO: We may have some yield different from zero even if production or areaHarvested or the both are ZERO.

We check at the end of the process that all those triplet where at least one element is ZERO, all the others are computed accordingly.

```
zeroProd=imputed[,get(formulaParameters$productionValue)==0 ]
zeroreHArv=imputed[,get(formulaParameters$areaHarvestedValue)==0]

nonZeroYield=imputed[,   (get(formulaParameters$yieldValue)!=0)]
```

```
filter=(zeroProd|zeroreHArv) & nonZeroYield

imputed[filter, ":="(c(formulaParameters$yieldValue),list(0))]

imputed[filter, ":="(c(formulaParameters$yieldObservationFlag),
aggregateObservationFlag(get(formulaParameters$productionObservationFlag),
                         get(formulaParameters$areaHarvestedObservationFlag)))]
                         imputed[filter, ":="(c(formulaParameters$yieldMethodFlag),
                         list(processingParameters$balanceMethodFlag))]
```

Note that flags associated to a "new imputation" computed as identity starting from other two figured require a procedure to be aggregated.

The implementation also require a table to map the hierachical relation of the observation flags. It provides a rule for "flag aggregation" (the process of assigning a new observation flag to an observation which is computed from other observations). An example of the table is given below. For more details on flags and how to create/interpret such tables, please see the vignette of the faoswsFlag. package.

```
library(faoswsFlag)
imputationParams
```

Save the imputation back to the database.

```
imputed= removeInvalidDates(data = imputed, context = sessionKey)%>%
         ensureProductionOutputs(data = .,
                                 processingParameters = processingParameters,
                                 formulaParameters = formulaParameters,
                                 normalised = FALSE)%>%
         normalise(.)
```

Only data with method flag `i` for balanced, or flag combination `(I, e)` for imputed are
saved back to the database.

Also the new (M,-) data have to be sent back, series must be blocked!!!

```
 imputed= imputed[(flagMethod == "i" |
                      (flagObservationStatus == "I" &
                           flagMethod == "e"))|
                     (flagObservationStatus == "M" &
                          flagMethod == "-"),]
```

I should send to the data.base also the `(M,-)` value added in the last year in order to highlight that the series is closed.

```
                if(imputationTimeWindow=="lastThree")
                {
                    imputed=imputed[get(processingParameters$yearVar) %in% c(lastYear,
                                                                              lastYear-1,
                                                                              lastYear-2)]
                    imputed= postProcessing(data =  imputed)
                    SaveData(domain = sessionKey@domain,
                             dataset = sessionKey@dataset,
                             data =  imputed)
                }else{


                imputed= postProcessing(data =  imputed)
                SaveData(domain = sessionKey@domain,
                         dataset = sessionKey@dataset,
                         data =  imputed)

                }

        })
```

Capture the items that failed:

```
    if(inherits(imputationProcess, "try-error"))
        imputationResult =
            rbind(imputationResult,
                  data.table(item = currentItem,
                             error = imputationProcess[iter]))

}
```

Return Message

```
if(nrow(imputationResult) > 0){
    ## Initiate email
    from = "sws@fao.org"
    to = swsContext.userEmail
    subject = "Imputation Result"
    body = paste0("The following items failed, please inform the maintainer "
                  , "of the module")

    errorAttachmentName = "non_livestock_imputation_result.csv"
    errorAttachmentPath =
        paste0(R_SWS_SHARE_PATH, "/kao/", errorAttachmentName)
    write.csv(imputationResult, file = errorAttachmentPath,
              row.names = FALSE)
    errorAttachmentObject = mime_part(x = errorAttachmentPath,
                                      name = errorAttachmentName)

    bodyWithAttachment = list(body, errorAttachmentObject)

    sendmail(from = from, to = to, subject = subject, msg = bodyWithAttachment)
    stop("Production imputation incomplete, check following email to see where ",
         " it failed")
}

    msg = "Imputation Completed Successfully"
    message(msg)
    msg
```

```r
if(!CheckDebug()){
## Initiate email
from = "sws@fao.org"
to = swsContext.userEmail
subject = "Crop-production imputation plugin has correctly run"
body = paste0("The plug-in has saved the Production imputation in your session.
              Session number: ",  sessionKey@sessionId)

sendmail(from = from, to = to, subject = subject, msg = body)
}


print(msg)
```