Fraunhofer
IAIS

FRAUNHOFER-INSTITUT FÜR INTELLIGENTE ANALYSE- UND INFORMATIONSSYSTEME IAIS

Code Review PAS

CODE REVIEW PAS

Dr. Alexander Kister

Fraunhofer-Institut für Intelligente Analyse- und Informationssysteme IAIS
in Sankt Augustin

Fraunhofer
IAIS

Inhalt

1
Introduction

This document contains the code review of PwC's Predictive Analytics Suite (PAS). With the PAS, the user can train models to predict future values of a time series. The models are trained on historic values of this and other independent timer series. Currently, two model types are implemented: a regression based model and two VAR models.

In this project all functions regarding the core functionality of PAS were considered. Functions regarding the user interface (R-shiny) are not reviewed.
The functions were reviewed regarding:

- Functional correctness:
  Functions should implement the right mapping between input and output.
- Readability:
  Code should be written such that a reader can easily understand it. Readable code helps the developer, because, particularly in a big project, adding new code usually requires to read old cod again. Choosing proper names for variables, objects and functions is an essential part of readable code. Code should also be well structured. Lines of code that logical belong together should stand close to each other in the code. Lines of code that follow each other should contain commands of a similar level of abstraction. The code should not contain unnecessary commands.
- Maintainability.
  It should be easy to maintain the code. This means that the effort of adding new functions, replacing or improving old functionality should be as small as possible. A necessary condition for maintainability is readability. Code repetition is a big problem for maintainability. Because if the repeated code has to be changed, it is easy to forget doing this for all repetitions.

The most complex functions are in file `lagFunctions.R`, to test their functionality special test functions were written. These test functions are discussed in Section 2. In Sections 3 a detailed discussion of the core functions of PAS is provided. In Section 4 a summary and further recommendations are given.

2
File «lagFunctions.R”

In the following sections the functions defined in `lagFunctions.R` are discussed.

2.1
Function Calculate.data.lags

2.1.1
Functional correctness

The function `calculate.data.lags` is responsible for shifting the dependent and independent variables according to a given lag order. The input is a vector of lag values `lag.vec`, the maximum and minimum of the lag vector `cur.max` and cur.min, the independent variables `reg.daten`, and the dependent variable `dep.temp`. The result is a list of the shifted independent variable and the shifted depended variable.

For a vector with positive lags the test

```
  if((cur.max + abs(cur.min)) < length(reg.daten[[1]]))
```

in line 51 leads to unexpected behaviour for small data sets. The test `test_that("calculate.data.lags has the correct dimension for small data sets"...)` fails because of this.

More importantly, on the right hand sides of lines 93 and 95, the variable dep.daten is used but not defined previously. To fix this, possibly one should replace `dep.daten[[1]]` on the right hand sides of lines 93 and 95 by `dep.temp[[1]]`, but this produces an error: see the last three cases from `list.of.lag.vecs` in `test_that("calculate.data.lags() correctly shifts dependent and independent data, given lag.vec is in list(c(0,0), c(2,3), c(-2,3), c(2,-3), c(-2,-3))'',...).` Even replacing `dep.daten[[1]]` by `dep.temp` still leads to an error for `lag.vec=c(-2,3).`

As the output consists of columns that are all named `temp`, the user has to remember renaming them. In line 197 this is done, but in the function `linear.regression.auto` this is not done for `bigData` set to `FALSE`.

2.1.2
Readability

The major problem, when it comes to readability is that the function has at least two responsibilities, namely shifting the independent variables according to `lag.vec` and also adopting the dependent variable. Choosing a name which makes this clear would help.

Furthermore it is not clear why `cur.max` and `cur.min` are passed to the function, since `cur.max` and `cur.min` are easily obtained from `lag.vec` by calling standard R functions (`cur.max <-`

`max(lag.vec), cur.min<- ...`). If other values than the maxima and minima of `lag.vec` are passed to the function, the output would not be as indented.

To make code that uses `calculate.data.lags` easier to readable, it is recommend to give names to the list elements that are returned: i.e. `list(shifted.independent=daten.grundlage, shifted.dependent=...)`

2.1.3
Maintainability

The maintainability would increase if the function is responsible for returning data frames with proper column names and not temp. Because then the developer does not need to remember to rename the columns of the output of `calculate.data.lags`. Function `linear.regression.auto` is an example where this was forgotten.

Since this is a complex function, test functions should be provided. If the test functions test all assumptions that one makes at another place about the output of the function, this test functions allow to assess whether code changes in `calculate.data.lags` effect the functionality at this other places.

2.2
Function linear.regression.auto

2.2.1
Functional correctness

The function `linear.regression.auto` performs a linear regression. The input is the data frame that contains the independent variables `daten.grundlage`, the data frame of the dependent variable `dep.temp`, an indicator `interaction` whether interactions between the independent variables should be considered, the names `nameDep` of the dependent variables and an indicator `bigData` whether special big data functionality should be used. Not that `bigData` is, throughout the whole program, always set to false. The function returns a linear model `lm.lags`.

Note that casting `daten.grundlage.na.clean` to a `data.frame` leads to an error, see `test_that("linear.regression.auto  works in a noise free context",...)`

Also note that in lines 109 and 110

```
daten.grundlage.na.clean <-
data.frame(daten.grundlage[complete.cases(daten.grundlage), ])
dep.temp.na.clean <- dep.temp[complete.cases(daten.grundlage)]
```

you implicitly make assumptions about the format of `dep.temp`, and `daten.grundlage`: This is why `test_that("linear.regression.auto can deal with  dependent.data passed as data frame",)` fails.

2.2.2
Readability

For the reader it is irritating to encounter a »`else if(bigData == FALSE)`" branch (lines 128 till 137) within a "`if(bigData == TRUE)`" branch (lines 112 till 138). Since this branch has no effect on the program it should be deleted.

2.2.3
Maintainability

To produce maintainable codes it is advisable to be consistent. It is not consistent to encode `interaction` by 0/1 but `bigData` by `TRUE/FALSE`. It is recommended to encode all Boolean variables by `TRUE/FALSE`.

2.3
Function regression.auto

2.3.1
Functional correctness

The function `regression.auto`
is responsible for creating a linear regression model, it automatically picks a lag structure that minimises the AIC. As for the function `linear.regression` the input is the data frame that contains the independent variables `datengrundlage`, the data frame of the dependent variable `dep.daten.temp`, an indicator `interaction` whether interactions between the independent variables should be considered the names `nameDep` of the dependent variables and an indicator `bigData` whether special big data functionality should be used. Additionally the variable `range` is passed. This variable defines the space in which the optimal lag structure has to be found. Furthermore a variable `dep.daten` is passed. This variable is not used in the function `regression.auto`. The output is a list, containing the optimal linear model and additional information about this model.

Therefore all possible combinations of lags are generated and stored in the variable `lag.vec.frame`. In the loop in lines 181 till 210 one regression model for each lag combination is generated and its AIC is saved in the variable `erg.eval`. This variable is used to determine the model with the smallest AIC. The loop over the lag structures is implemented with the foreach package. This package allow to perform operations in parallel. On the machine where this was tested this functionality leads to an error.

Note that `length(datengrundlage)` does not return the number of variables, but the total number of entries. A possible fix is to use `ncol(datengrundlage)`, but clearly this might fail for vectors.

A reader only gradually understands what the purpose of this function is. Judging from the function name, the main job seems to be to calculate `lm.lags`. Only later, it turns out that the main job of this function is to select the best (according to AIC) combination of lags.
A look to the return values is also not helpful since there are five of them. Furthermore the return value `lags` easily follows from `lag.vec.erg`. Using calculate.data.lags it is even possible to obtain two more return values: `daten.grundlage` and `dep.temp`.

It remains unclear why and for what the variable `dep.daten` is passed to `regression.auto()`. It is not used as input in any of the commands within the function `regression.auto()`.

For the reader it is also confusing that the following variables are defined but never used: `dap.date`, `erg.eval.par`, `erg`, `lag.vec.frame`, `dep.temp.erg`

Also note that after `reg.daten <- datengrundlage` (in line 162), none of the variables `reg.daten` or `datengrundlage` is changed, so `reg.daten` and `datengrundlage` are two names for exactly the same object. The two names are also not a signal for two different meanings of the data.

2.3.2
Maintainability

To increase the maintainability it would help to be more consistent in the definitions of the functions. In this file there are two functions `linear.regression.auto` and `regression.auto` that on a logical level have similar inputs but the order in which they have to be provided to the functions is different. A developer who does not know (or has forgotten) the internal definitions of the functions might accidently provide the arguments in the wrong order.

Furthermore reassigning the names to the columns colnames`(daten.grundlage) <- attributes(reg.daten)$names` (line 197) should be the responsibility of `calculate.data.lags,` since `calculate.data.lags` causes the loss of these names.

The code in lines 183 till 200 is up to one parameter repeated in the lines 220 till 235. Avoiding this repetition by defining a function would make this file easier to maintain.

2.4
cor.lag

The function `cor.lag` is responsible for providing the correlation table. Its input consist of the depended variable `abVar`, the independent variable `unabVar` and the `Lags`. The output is the correlation table `korrelations.data.frame`. This function is functionally correct. To make it more readable and maintainable, it is recommended to rename the input variables such that they are consistent with the names used in the other functions of lagFunction.R.

2.5
verteilungsBetrachtung

The function `verteilungsBetrachtung` is responsible for fitting a given histogram to a list of distribution. The input is the histogram `punktwerte.` The output is a list of distributions `erg.verteilungsfunktionen` fitted to the histogram `punktwerte and` a list of statistics `temp.anpassungstests` of how good each fit is. To provide the functionality, the developers used the R package MASS.

Additional comments would help the reader. The name `verteilungsBetrachtung` is not very specific, since the function mainly fits the histogram to the distribution this should be reflected in the name.

2.6
webplot

This function allows to draw spider web charts. It is not used in the app. The code is published on http://www.statisticstoproveanything.com/2013/11/spider-web-plots-in-r.html.

3 Discussion of all files

3.1 Files «0.0.server_home.R” and «0.1.server_password.R”

3.1.1 Functional correctness

In this files functions for given the user initial information about the PAS and for the Login are defined. The implementation of the Login is in an early state. Currently password and user name are the empty string.

3.1.2 Readability

As the files are short the readability is good.

3.1.3 Maintainability

Implementing, a more sophisticated Login process is easier in other languages than R.

3.2 Files «1.0.server_dataimport.R” and «1.1.server_mainsettings.R”

3.2.1 Functional correctness

In this files the functionality for loading a data set is defined. The `reactiveValues` defined here have a role similar to global variables.

1.0.server_dataimport.R: The most essential `reactiveValues` object is `daten.under,` it contains the variable `daten.under$base.` This variable is the basis data set for all calculations. In addition it contains variables that are used once the dataset is rescaled or spitted and variables that help to organise the output in the "Data Analysis" tap. With the objects observeEvent`(input$showdataset,` ...), and `observeEvent(input$stahlpreiscsv,...)` the user can load a data set.

1.1.server_mainsettings.R: While most of the objects defined here are used to organise the output, one object (namely `observeEvent( input$restart1dataset, ...)`) is used to rest all `reactiveValues` objects from file 1.0.server_dataimport.R.

In the implementation of this functionality no errors were found.

3.2.2
Readability

While the comments are good, not all variable names help the reader to understand the code. First of all the variable names should be derived from the same language, i.e. only English and not English and German. So objects names like `output$dependent_variable_eins` (File 1.1.server_mainsettings.R line 163) should be avoided. More severely names of frequently reoccurring variables should be easy to interpreter. Especially since the comment (lines 102 till 104 in file 1.0.server_dataimport.R) introduces `daten.under` as a central object of the programm, it should not contain variables with the following names of unclear meaning: `data.temp.1, data.temp.2, default2` (examples are taken from lines 114,132 and 136). Especially `default2` needs more explanation, since after it is defined in file 2.1.server_datearearrrange.R in line 146 it is not used any more.

3.2.3
Maintainability

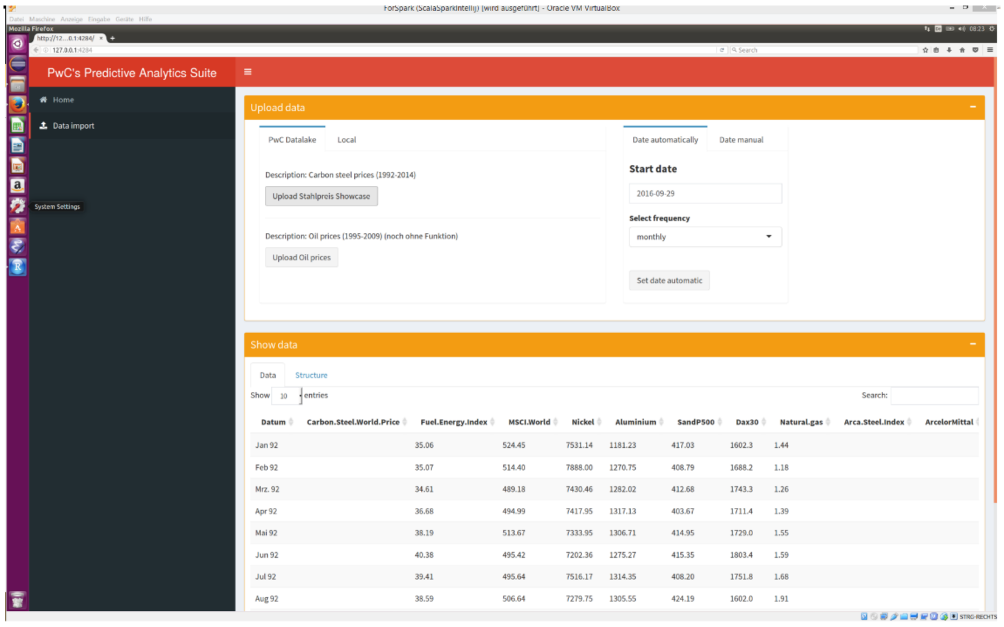
To make this file more maintainable it is advisable to split `daten.under` (Line 105 in File 1.0.server_dataimport.R) such that the parts have a higher cohesion. A possible split is to create four separate `reactiveValues` objects: one for the basis data, one for splitting, one for scaling and one for the graphical analysis. Splitting the variable according to their purpose makes it easier to check whether they are updated correctly.

Due to their similar purpose, the functions `observeEvent(input$showdataset, ...)`, `observeEvent(input$stahlpreiscsv,...)` have 26 lines (namely lines 180 till 206 and 231 till 256 in File 1.0.server_dataimport.R:) in common. It is recommended to define a separate function that implements the functionality of this 26 shared lines.

The object `observeEvent(input$restart1dataset, ...)` defined in File 1.1.server_mainsettings.R has a functionality that is different from the one of all other objects in this file. This `observeEvent` object is responsible for resetting all `reactiveValues` objects from file 1.0.server_dataimport.R. It is hidden between objects that are used to organise the output. If the definition of one `reactiveValues` object in file 1.0.server_dataimport.R changes, it is likely that the developer forgets making the appropriate changes in the code of `observeEvent(input$restart1dataset, ...)` in File 1.1.server_mainsettings.R.

3.2.4
UI

For the user the purpose of the "set date automatically" button is not clear. Instead of automatically using the starting date from the input file, it use the current date (since this is the default value of `input$reg.date.start`). In the screenshot below the user expects that by pressing «set date automatically", the start date is still the January of 1992 and not the current date.


ForSpark (ScalaSparkIntellij) [wird ausgeführt] - Oracle VM VirtualBox

PwC's Predictive Analytics Suite

Upload data

Description: Carbon steel prices (1992-2014)

Description: Oil prices (1995-2009) (noch ohne Funktion)

Start date: 2016-09-29

Select frequency: monthly

Show data

Show 10 entries

Datum	Carbon.Steel.World.Price	Fuel.Energy.Index	MSCI.World	Nickel	Aluminium	SandP500	Dax30	Natural.gas	Arca.Steel.Index	ArcelorMittal
Jan 92	35.06	524.45	7531.14	1181.23	417.03	1602.3	1.44			
Feb 92	35.07	514.40	7888.00	1270.75	408.79	1688.2	1.18			
Mrz. 92	34.61	489.18	7430.46	1282.02	412.68	1743.3	1.26			
Apr 92	36.68	494.99	7417.95	1317.13	403.67	1711.4	1.39			
Mai 92	38.19	513.67	7333.95	1306.71	414.95	1729.0	1.55			
Jun 92	40.38	495.42	7202.36	1275.27	415.35	1803.4	1.59			
Jul 92	39.41	495.64	7516.17	1314.35	408.20	1751.8	1.68			
Aug 92	38.59	506.64	7279.75	1305.55	424.19	1602.0	1.91			

3.3
Files «2.1.server_datarearrange.R" and «2.2.server_datasplit.R"

3.3.1
Functional correctness

The purpose of the objects in this files is to rearrange (i.e. scale and tweak) the initial data set and to split the initial data set into a train and test set. The objects in this files manipulate the `reactiveValues` object `daten.under`, which was defined in file 1.0.server_dataimport.R.

2.1.server_datarearrange.R: The object `observeEvent(input$scale.data, ...)` is responsible for scaling the numeric column the user has selected in `output$scale.var.`: It scales the column in `daten.under$data.temp.1`, `daten.under$data.temp.2` and in `daten.under$base` and manages the list `daten.under$scale.values`. The list `daten.under$scale.values` contains for each scaled variable the name and the original mean and standard deviation.

On the other hand the object `observeEvent(input$rescale.data,…)` is responsible for inverting the scaling process. It hence effects the variables `daten.under$data.temp.1`, `daten.under$data.temp.2` and `daten.under$base` and manages the list `daten.under$scale.values`.

The object `observeEvent(input$tweak,...)` allows the user to tweak values in a numeric column. It effects the variables `daten.under$data.temp.1`, `daten.under$data.temp.2`, `daten.under$base` and additionally `daten.under$default2`. Currently `daten.under$default2` is not used as input to any other function.

2.2.server_datasplit.R: The object `observeEvent(input$scale.data, ...)` is responsible for splitting the data set into a training and testing set. Via the UI the user sets the number `input$numbersplit`. The object `observeEvent(input$scale.data, ...)` uses `input$numbersplit` to update the variables `daten.under$data.train` and `daten.under$data.test` appropriately and stores the parameter of the split to the variable `splitindex`.

On the other hand the object `observeEvent(input$unsplit,…)` is responsible for inverting the splitting process. Therefore it has an effect on `daten.under$base`, daten.under$data.test, daten.under$data.train and on the variable `splitindex`. Note that for this split the variable `daten.under$default` is used.

No functional errors where found.

3.3.2
Readability

While the names for the events (i.e `input$tweak`) are easy to interpret, the names `output$texthelp2`, `output$texthelp3`, `output$texthelp4`, `output$texthelp5` and `output$texthelp6` (which are defined in lines 184 till of 2.1.server_datarearrange.R) only tell the reader that their purpose is to help. For example a better name for `output$texthelp6` would be `output$texthelpResetButton`.

3.3.3
Maintainability

The code contains many lines (for example line 45, 46, 47, 51, 52 in 2.1.server_datarearrange.R) where only certain parameters are changed. But as these lines are easy to read and it is very unlikely that their functionality has to be changed this is a less sever point.

3.4
File «3.1.server_data_analysis.R"

3.4.1
Functional correctness

The functions in this file are responsible for producing scatter plots, scatter matrices, visual inspection of the time series, distribution analysis and boxplots. The plots (their underlying data) can be stored in the reactive value `report` and are then available for producing a report. The object responsible for organising the variable `report` are `observeEvent(input$acceptscattereinsreport,…)`, `observeEvent(input$acceptscatterzweireport,…)`, `observeEvent(input$acceptscattermatrixreport,…)`, `observeEvent(input$acceptplotAllCompreport,…)`, `observeEvent(input$addreportcorrelation,…)`, `observeEvent(input$addreportcollinearity,…)`.

Note that the function `output$lag.highest.all` fails if the data set is split. This happens because `corresp.var` does not have the right dimension after the split.

The purpose of `hhh<<-xts(dat.vis.relevant, daten.under$base[[1]])`(line 1024) is not clear. The variable is not used after it is defined. It is recommended to check whether this line could be deleted.

3.4.2
Readability

Variable names should be easy to read. For names consisting of several words, the start and the end of each word should be easy to recognize. For the name `acceptscattereinsreport` (line 750) this is not possible. A way to improve this is to use capital letters when a new word starts.
The idea to define the function `histoDist()` is good, but expressions like `histoDist()[[1]]` are not very readable. To avoid this it is recommended to give names to the list members.
Furthermore in the last halve of the file adding comments would be a great help for the reader.

3.4.3
Maintainability

To make the code easier to maintain it is recommended to remove all repetition. For example the code for the 1. and 2. Scatterplot differ only in some parameters, this effects the objects
`observeEvent(input$scatter1_click,…)`, `observeEvent(input$scatter2_click,…)`, `observeEvent(input$exclude_toggle.1…)`, `observeEvent(input$exclude_toggle.2,…)`, `output$scatt.1 and output$scatt.2`
Similar but less severe is the relation between the four objects:
`output$dependent_variable_scatter1`, `output$dependent_variable_scatter2`, `output$independent_variable_scatter1`, `output$independent_variable_scatter2`.
Also the code appearing in `boxplotsgraph` is repeated in `boxplotsgraphscaled`.

3.4.4
UI

Please note that there is a spelling error in the menu point «Collinearity analysis": «Hightest collinearities". Additionally the content of `output$texthelp7` is not very helpful.

3.5
File «4.1.server_createRegressionmodel.R"

3.5.1
Functional correctness

The functions in this file allow the user to define two kind of regression models: an automatically generated and a user defined model.

The `reactiveValues` object `regression.modelle` contains the models that are created by the other objects in this file.

The object `regression` implements the manual regression model. Outside of the object, by assigning values to the variables `input$columns_manual` and `input$lagAll`, the user can define the independent variables and the lags for each independent variable. The branch for `bigData == TRUE` (lines 214 till 243) is never used since in line 108 the assignment bigData `= FALSE` is made. This object effects the variable `regression.modelle$manuelles.modell`.

The object `reg.auto` creates the automatic regression model. Outside of the object, by assigning values to the variables `input$columns, input$range, and input$interaction.1`, the user can define the independent variables, the range of the lags and whether the interaction between the independent variables should be considered. The result is stored in the variable `regression.modelle$automatisches.modell`.

The objects `observeEvent(input$storeReg,…)` and `observeEvent(input$storeRegAuto,…)` allow to store the models in the variable `regression.modelle$gespeichertes.modell`.

To make the program more efficient it is recommended to call the function `reg.auto()` less frequent. The function `reg.auto()` is called in lines 352 and 364. But the result of `reg.auto()` is already stored in `regression.modelle$automatisches.modell`. For lines 352 and 364 it is recommended to use the stored result.

3.5.2
Readability

The repetition makes this file hard to read. The lines 399 till 408 are particularly puzzling: The first tree lines are nearly identical with the last three lines. As the first three lines are redundant, it is recommended to delete them.
As mentioned in an earlier section, it is recommended to use more informative names: a name like `output$texthelpPurposeOfIndependentVars` is easier to understand than output`$texthelp14`. Using a more informative name could prevent the developer from displaying the wrong helping text just because he mixed up the numbers. The same is true for output`$texthelp15`, output`$texthelp16` and output`$texthelp17`.

3.5.3
Maintainability

To avoid the repetition, it is highly recommended to use the functions from `lagFunctions.R`. For example the lines 153 till 188 are identical to the core of function `calculate.data.lags` in file «lagFunctions.R" and the lines 208 till 249 are identical to the core of function `linear.regression.auto` in file «lagFunctions.R".

As the block for reading the lag structure (first appearance in lines 527 till 530) appears 12 times, creating a function that has `input$lagAll` as input and `diff` as output is recommended.

Less sever but still important to notice is that the codes of `output$choose_columns` and `output$choose_columns_manual` differ in only one line. Furthermore note that the steps to generate `reg.daten` appear twice: once in `regression` and once in `reg.auto`. It is recommended to check whether outsourcing the repeated code into functions is possible.

3.6
Files «4.2.server_impactAnalysis.R" and «4.3.server_modelValidation.R"

3.6.1
Functional correctness

The functions in this files allow the user to investigate the previously created models. They are currently not used. The functions in file «4.2.server_impactAnalysis.R" allow the user to investigate how, according to the previously estimated model, changes in the independent variables effect the predicted dependent variable. The functions in file
«4.3.server_modelValidation.R" implement a cross validation and allow to assess the results in graphical and numerical form.

No functional errors were found.

3.6.2
Readability

It is recommended to name the members of the list that is returned by the function `table.beta`. Because that way expressions like `table.beta()[[2]]` (Line 115 in 4.2.server_impactAnalysis.R) could be avoided. Names like `table.beta()[[2]]` could lead to errors, since it is possible to mix up the number, while it is easier to notice that something is wrong if a meaningful name appears in a context where it does not fit.

3.6.3
Maintainability

Lines 125 till 133 in «4.2.server_impactAnalysis.R" and the lines 52 till 60 in
«4.3.server_modelValidation.R" are identical. Encapsulating this lines in a new function would increase the maintainability: Currently if a fourth model (beside the manual, automatic and saved model) should be included the code has to be changed in at least two files, by defining a new function only one function has to be changed.

3.7
File "5.1.server_forecastArima.R", «5.2.server_forecastTrend.R", «5.3.server_forecastManual.R" and «5.4.server_forecastSavedModels.R"

3.7.1
Functional correctness:

The functions in this files provide preliminary tools for the prediction of the dependent variable. The actual prediction is implemented in the files starting with 6. For this prediction method the user has to define a way how the history of the independent variables is elongate into the future. The files we consider here contain the functions for defining this elongations.
In each of the first three files a list `(daten.under$arima.modells, trend.values$trends, submitted.manual$submitted.data)` for storing the elongations is managed. Possible elongation methods are to use an ARIMA Model, set a linear trend, or make manual input. In the last file («5.4.server_forecastSavedModels.R") the output of the saved prediction model is organised.

The objects `output$chooseIndVar <- renderUI(…)`, `output$trendForcast <- renderUI(…)`,`output$chooseVariablemanual <- renderUI(…)` ask the user of the UI to pick the independent variable for which he likes to define an elongation via respectively an ARIMA model, a trend or manual input. To organise this the `renderUI` objects use the variable `arima.values$tabelle.variable`. This variable keeps (among other things) track of the independent variables for which an elongation already exists.

5.1.server_forecastArima.R: The central function in this file is `observeEvent(input$storeArima,…)`. This function is responsible for creating and storing a list of ARIMA models `daten.under$arima.modells`. An ARIMA model is either automatically or manually defined (Therefore the objects `arima.model.auto` and `arima.model.manual` are used). The ARIMA models are added one by one. Which creation method is used for building a particular ARIMA model depends on the variable `input$arimaSelected`. Before the model is added to the list `daten.under$arima.modells` it is stored in the variable `arima.model`.

5.2.server_forecastTrend.R: In this file the object `observeEvent(input$subTrend,…)` is responsible for adding an elongation `trend.values$trends.temp[[index.name]]` to the list `trend.values$trends`. The elongation itselves is created in `observeEvent(input$prevTrend,…)` (the central line is 328).

5.3.server_forecastManual.R: In this file the object `observeEvent(input$saveforecast,…)` is responsible for adding an elongation `submitted.manual$submitted.temp[[submitted.manual$current.name]]]]` to the list `submitted.manual$submitted.data`. The elongation itselves is created with `observeEvent(input$acceptforecast,…)` (the central line is 187). The object `observeEvent(input$acceptforecast,…)` is called repeatetly, once for each date in the forcast horizont.

5.4.server_forecastSavedModels.R: The central object in this file is `observeEvent(input$deleteModel,...)`. It is responsible for deleting an elongation from one of the lists `daten.under$arima.modells`, `trend.values$trends`, `submitted.manual$submitted.data`.

3.7.2
Readability

A reader who knows which buttons are defined in the user interface, can understand the code, but as the code should be understandable without knowing how the user interface looks, it is recommended to let the reader know at the binning what the purpose of the temp variables in the global variables `reactiveValues` Object `(store.graph, trend.values, submitted.manual)` is (for instance: the temp values allow the user of the app to experiment with different kind of elongations before saving them).

For the reader it is important that the variables are defined before they are used. The variable `arima.values$tabelle.variable` is defined in file 6.1.server_forecastFrame.R in lines 1 till 9 but it is already used in file 5.1.server_forecastArima.R in line 48. It is recommended to make the definition before using it.
Reading is easier if objects that have a similar purpose have a similar name. So the similarity in the purpose of the functions `observeEvent(input$subTrend,…)` and

`observeEvent(input$saveforecast,…)` should be reflect by for example renaming `input$subTrend` into `input$saveTrend`.

As mentioned earlier variable names like `output$texthelp18, output$texthelp19 …, output$texthelp23` are not readable.

3.7.3
Maintainability

The file «5.4.server_forecastSavedModels.R" contains an object `observeEvent(input$deleteModel,…)` that is responsible for deleting elongations. This is misleading since all other functions in the file deal with the graphical presentation of models. A developer who likes to improve the deleting functionality but who does not know or forgot the name of the object has to invest time to search for this function.

3.8
Files"6.1.server_forecastAlgo.R" and «6.1.server_forecastFrame.R"

3.8.1
Functional correctness

The functions in 6.1.server_forecastAlgo.R implement the forecasting procedure for the regression model. The forecast algorithm adds a normal distributed error to the elongation of the independent variables. These elongations are defined in the files discussed in the previous section. The core of this algorithm are Lines 197 and 199. The functions in 6.1.server_forecastFrame.R are responsible for the output.

6.1.server_forecastAlgo.R: The object `data.preparation` in file 6.1.server_forecastFrame.R contains four blocks that work together to implement the forecast algorithm. The results of this process is a vector of predictions which are based on the elongations with added noise.

In the first block, in lines 14 – 40, the regression model (including the lag structure) and the elongations (from the files starting with 5) are loaded.

In the second block, lines 41 till 57, the variable `daten.lag` is defined. To understand the definition of `daten.lag`, assume the regression model uses an independent variable that is lagged by 1. Recall that which value of the independent variable enters into the prediction, depends on the lag this variable has in the regression model. At the first prediction step, for an independent variable that is lagged by 1 the historic value has to be used for the prediction; only for an independent variable that is not lagged the value has to be taken from the elongation. To provide the historic values `daten.lag` contains for each independent variable a list of historic values with a length corresponding to its lag.

The next block (lines 58 till 178), which itselve consist of three sub blocks, extracts the required information from the elongations. In other words this block provides information about the values that enter a prediction, but that do not appear in the history `daten.lag`. The result of this block is stored in the list `norm.parameter`. Each entry of this list corresponds to an independent variable. And an entry is a data frame consisting of two columns: the mean and standard deviation. For each independent variable, the length of the entry is chosen such that together with the values from `daten.lag` either a lagged value or a mean and a standard deviation is available for every point in the prediction horizon. The first sub block (lines 61 till 101) is responsible for using the Arima models, which are created in file 5.1.server_forecastArima.R and stored in variable `daten.under$arima.modells.` The second sub block (lines 102 till 142) is responsible for using the manual elongations, which are created in file 5.3.server_forecastManual.R and stored in variable `submitted.manual$submitted.data.` The third sub block is responsible for using the elongations via trends, which are created in file 5.2.server_forecastTrend.Rand stored in variable `trend.values$trends`.

In the fourth block (lines 180 till 227) the normally distributed error term is added to the elongation of the independent variables. The output is a vector of simulated predictions, `sim.values`. The number of simulation rounds is read from the variable `input$sim` (line 188). The loop starting in 188 loops over this rounds. The loop inside this loop starting at line 192 loops over all independent variables. The result of the second loop is `daten.lag.temp`, a list of lists. Each list in `daten.lag.temp` has the length of the prediction horizon, as far as possible this list consists of the lagged values and the remaining places are filled with simulated data. Using the elongation with added error terms `daten.lag.temp` as input for the regression model `regression.modell[[1]]` a prediction value `sim.values.temp` is generated and added to the vector `sim.values`.

In the fifth block (lines 228 till 261) the mean of the simulated predictions is stored in the variable `confidence.sim.mean.1`. The logic for doing this is the same as in the fourth block.

6.1.server_forecastFrame.R: The objects in this file organise the UI. Of special importance is the eventhandler in lines 21 till 146. It guaranties that all needed elongations are actually defined.

The file 6.1.server_forecastAlgo.R contains a line of code that does not have any effect on the program, namely line. It seems that this line was used for testing. Testing this function is a good idea, but it should be done by writing test functions.

In lines 79 and 80 in file 6.1.server_forecastAlgo.R the term `(pred.data$mean - pred.data$mean),` which sums to `0,` is added to the right hand side.

3.8.2 Readability

Even though the comments structure the function `data.preparation` on a higher level, it is still recommended to actually split the function into separate functions. The goal should be that each function that is called in `data.preparation` has the same level of abstraction. It is possible to split this function into at least four functions corresponding to the four blocks mentioned in the discussion of the functionality. Smaller functions are easier to read since they make it more transparent what input is used to generate which output.

For the reader telling names are easier to understand than numbers that's why Expressions like `trend.data[[i]][[2]]` (in line 162) are not recommended.

To save the reader time, code should not contain unnecessary code. As the block from line 77 till 83 in file 6.1.server_forecastAlgo.R appears unnecessary (since it implements the same functionality as in the lines 84-94) it should be deleted. Also the «if branch" in lines 219-220 in file 6.1.server_forecastAlgo is empty and hence has no effect on the program.

3.8.3 Maintainability.

Having one big function that does many things is problematic when it comes to maintainability. A developer who likes to update one of the things the function might be forced to read the whole function to understand possible side effects of his change. Furthermore it is easier to write test functions for small functions. Test function contribute to the maintainability. So it is recommended to split the function `data.preparation` into four functions.

3.9 File «7.server_forecastVar.R”

3.9.1 Functional correctness

The objects in this file allow the user to generate VAR and restricted VAR models. These models are stored in the `eventReactive` object `var_data`.

The model `VAR$model` is generated in both of the following objects: `observeEvent(input$EST,...)`, `bserveEvent(input$FCST,...)`. The difference is that the second object additionally sets the flag `dygraph$indexforecast` to true. The second object is called in parallel with `varDygraphforecast<-eventReactive(input$FCST,…)`, a function that is responsible for using `VAR$model` to generate a forecast `VAR$fcst` (see line 336). Once `observeEvent(input$saveEST, …)` is called, the model is saved in `var.modelle$gespeicherte.expl.modelle.var`.

The restricted model is generated in an analogue way. It is first stored in `VAR$modelres` and once `observeEvent(input$saveresEST,…)` is called it is saved to `var.modelle$gespeicherte.expl.modelle.restvar.`

In this file, it is necessary to think about error handling since in the menu item "VAR forcasting/Prepossessing" the following error is produced when Arca.Stell.Index is picked as one of the independent variables:

```
Error in <-: 'dimnames' applied to non-array
```

3.9.2 Readability

This file is good to read since it consist of short functions. The repetition appearing in this file has a bigger effect on maintainability (see the section below) than on readability.

3.9.3 Maintainability

The repetition appearing in this file reduces its maintainability. Note that except for one line the codes for the buttons «Estimate VAR-Model” and «Forecast VAR-Model” are the same. It is recommended to avoid this repetition by defining a function. Furthermore the repetition in `observeEvent(input$saveEST,...)`, `observeEvent(input$saveresEST,...)` should be avoided as well.

3.10
Files”8.1.server_backtesting.R” and «8.2.server_backtestingonestepahead.R”

3.10.1
Functional correctness

8.1.server_backtesting.R : The objects in this file perform back tests. This functionality is mainly implemented in `output$checkboxgroupbacktest<- renderUI( …).` To organise the UI the object `output$checkboxgroupbacktest<- renderUI( …)` consists of 31 «if blocks”. For each «if block” one or a combination of up to four of the following four models is back tested: the stored models, a Random Walk model or an ARIMA model. To implement the Random Walk and the ARIMA model, methods from the R package «forcast” are used.

The main input to the object `output$checkboxgroupbacktest<- renderUI( …)` is the variable `input$selectHorizont`. In all «if blocks” the back tests are performed on the dataset `daten.under$data.test`. To do so the dataset `daten.under$data.test` is split into forecasting-periods of consecutive observations. Each forecasting-period has the length `input$selectHorizont` (as the length of `daten.under$data.test` is not necessarily a multiple of `input$selectHorizont`, it is possible that some data points are not part of a forecasting-period, these data points are not used). The first time this split is done is in lines 158 till 160 of file 8.1.server_backtesting.R.
For the models of a given «if block”, forecasts are made for all forecasting-periods. For a given forecasting period, the complete histories up to this forecasting period are used for the independent variables in the prediction. The first time this history is defined is in lines 172.
While the regression model is trained only once, namely with the values in `daten.under$data.train,` the Random Walk model, the ARIMA model and the VAR models are train on the entire history (but for the VAR models the lag order is saved in `var.modelle$gespeicherte.expl.modelle.var[[5]]` or `var.modelle$gespeicherte.expl.modelle.restvar[[5]]`, this lag order is fixed using only the data available in `daten.under$data.test`).

Now all «if blocks” are discussed:

The output of the first fife «if blocks” is the time series `RMSE` of the absolute value of the difference between the actual realisation and the prediction of one of the stored models, or a Random Walk model or an ARIMA model.

The «If blocks” 1 and 2 do not take into account independent variables. They measure the performance of a Random Walk and an auto ARIMA model respectively.

The «If block” 3 tests the regression model `regression.modelle$gespeichertes.modell` on the test data set. If the horizon `input$selectHorizont` reaches so far into the future that the lagged values cannot be extracted from the history (where the present represents the time when the forecast is made), these values are generated by ARIMA models for the independent data. This filling up procedure is analogue to the one in file 6.1.server_forecastAlgo.R lines 43 till 56. As there, the result of the filling up is stored in the variable `daten.lag`.

The «If block” 4 test the VAR model with lag order `var.modelle$gespeicherte.expl.modelle.var[[5]]`.

The «If block" 5 test the VAR model with lag order `var.modelle$gespeicherte.expl.modelle.restvar[[5]].`

The «If blocks" 6 till 15 allow the user to compare two of the previous 5 methods. The output of this blocks is the data frame `RMSE,` it contains the two time series, one for each method. The time series are the absolute value of the difference between the prediction of the method and the actual realisation.

The «If blocks" 16 till 22 are responsible for comparing any three models.

The «If blocks" 23 till 30 are responsible for comparing any four models.

The «If blocks" 31 is responsible for comparing any fife model.

8.2.server_backtestingonestepahead.R: In this file the central functionality is implemented in `output$checkboxgroupbacktestOnestep<- renderUI(…).` The idea of the test here is to predict the values for the time points of the entire set `daten.under$data.test.` The structure is the same as in 8.1.server_backtesting.R.

Functional correctness: The variable `windowZweiTs` is not defined when it is used in line 431. Guessing from the context, this is a spelling error and it should be `windowTs.`

Note that in the one step ahead forecast the prediction horizon is not the complete test set, but only the next `input$steps` observations. This is because for making this prediction `RMSEonestepahead <- eventReactive(input$startbacktestingonestepahead,…)` uses the values saved in `regression.modelle$gespeicherte.forecasts[[2]]` (for the first time this is done see line 179 in file 8.1.server_backtesting.R). The same is true for the VAR models. If this behaviour is not intended the code has to be changed appropriately.

3.10.2 Readability

The amount of repetition makes the file 8.1server_backtesting.R hard to read: The last 11 lines of each if block differ only in few parameters. The block for the random walk forecast in lines 188 till 191 appears also at line 482, 586, 639, 699, 1278, 1350, 1416, 1518, 1628, 1689, 2024, 2089, 2213, 2327, 2729. Also the blocks for the ARIMA forecast (first appearance in lines 237 till 241), for the regression model (first appearance in lines 291 till 330), for the VAR forecast (first appearance in lines 379 till 382), for the restrict VAR forecast (first appearance in lines 431 till 434) appear multiple times. As the two files staring with 8 have a similar high level structure, the file 8.2.server_backtestingonestepahead.R also contains many blocks that appear multiple times.

3.10.3 Maintainability.

This files are very hard to maintain since they contains much repetition. Defining functions to avoid the repetition would allow to test the code better and would avoid to perform updates only partially (by accidently not performing the update to all repeated code segments).

3.11
File «9.server_reporting.R”

3.11.1
Functional correctness

The objects in this file organize the report which is shown to the app user at the end. The user can download this report.   The object `output$downloadReport` implements the functionality for downloading this report. To create the report the object uses the library `rmarkdown`. This file contains an object that is still in development: `output$headervisual`.

To get the download functionality running on the system which was used to perform this code review, it was necessary to replace `'Reports\\report.Rmd'` by the actual complete path to the Rmd file.

3.11.2
Readability

For a reader the unfished object is irritating. Also using the assignment operator <<- instead of <- in line 531 (where the variable is defined, while this variable is never used again) cause unnecessary efforts for the reader.

3.11.3
Maintainability

The repetition appearing in this file does not effect the maintainability (or readability) too much. But for making adding additional scatter matrices easier it would help to store the text in lines 317 till 320 in a variable.

4
Summary and Recommendations

4.1
Functionality

Overall the implementation of the models is good. But there are errors, as documented in Sections 2 and 3. Regarding the functionality, it is recommended to write more test functions, to guarantee that the functions work as intended and to notice probable side effects of future changes to the code.

4.2
Readability

The stylistic issues are as important as the functional issues. Even through some issues might be subjective, a cleaner code is easier to maintain and extend. Particularly clean code is easier to test. Unclear code can lead to problems, most of the time unforeseen problems, for example when the code is updated, the functionality is changed, the amount of input is increased, et cetera. A guideline for writing clean code is to divide the code such that the parts have high cohesion and small coupling. The parts have high cohesion if each part fulfils one and only one purpose, if within a part each sub module is necessary for the purpose of the part and if all sub modules of one part have the same level of abstraction. The parts have small coupling if changes in one part only lead to a small number of changes in other parts.

Robert C. Martin's book, [Martin 2008], is a very helpful guide to write code that is easy to maintain and extend. A short summary of this book is given at the link: Clean-Code-summary.pdf. Most vitally, Martin recommends:

- to use meaningful names.
- be consistent.
- to write functions that do only one thing.
- to write functions with a small number of input variables.
- to avoid duplication.
- automated tests should cover every detail of our code's functionality, should accompany the code in the archive, and be easy to execute.

Generally, it is recommended to go through your code again and to try to remove all repetitions and to find more meaningful names for the variables and functions.

Fraunhofer
IAIS

4.3 Extensibility

For extensibility it is worthwhile to invest into the quality of the code. Software technology offers a wide range of tools and processes to support the development of high quality code, i.e. unit testing, type-safety and memory management. A good support for unit testing is essential for writing functional correct code. Type-safe languages prevent type errors. For larger project memory management is also important. Such tools are usually available in object oriented languages as Java. As R does not offer such tools, it is usually used for smaller projects.

For the current scope of the tool the programming language R is enough. But if the sizes of the data sets grow, it is advisable to think about switching to a language which scales better.

References

[Martin 2008] **Robert C. Martin, Clean Code: A Handbook of Agile Software Craftsmanship, Prentice Hall PTR, Upper Saddle River, NJ, USA, 2008.**