

IRB Model Validation



Example of an Automated Validation Report

Andrija Djurovic 

Contents

1	Introduction	2
2	Report Configuration	2
3	Data Import	4
4	Descriptive Statistics and Summary Tables	7
5	Data Visualization	9
6	Quantitative Testing	12
7	Alternative Methods and Report Customization	14

1 Introduction

The validation of credit risk models is typically divided into qualitative and quantitative procedures, with the latter often subject to automation. This report demonstrates the use of **R** and **Python** to automate quantitative procedures. While it does not provide a detailed overview of these procedures and tests, it focuses on the technical aspects of integrating **R** and **Python** for validation. The approach presented reflects the author's preferences regarding technical methods, but practitioners are encouraged to explore and adapt it to best suit their specific needs. The following sections are generated entirely in **R**, with specific procedures executed and imported from **Python** using the **reticulate** package. The PDF format is also generated with the **rmarkdown** package, while tables are formatted using the **flextable** package. Graph examples are created with the **ggplot2** package.

Practitioners can find details on specific tests and approaches used in IRB model validation on the following [GitHub page](#).

The rest of the report is organized as follows: The Report Configuration section briefly outlines the **R** session used to generate the report and the **Python** setup for running scripts from **R**. Subsequent sections provide technical details on the most common methods used in validation reports, typically covering data import and processing, descriptive statistics, data visualization, and quantitative testing. Finally, the report raises awareness of alternative approaches and possible improvements for real-world case scenarios.

2 Report Configuration

To run this report, specific packages must be installed for **R** and **Python**. In **R**, these include **rmarkdown**, **knitr**, **flextable**, **ggplot2**, and **reticulate**. The following output provides details of the **R** session used to generate this report:

```
sessionInfo()

## R version 4.3.2 (2023-10-31 ucrt)
## Platform: x86_64-w64-mingw32/x64 (64-bit)
## Running under: Windows 11 x64 (build 22631)
##
## Matrix products: default
##
##
## locale:
## [1] LC_COLLATE=English_United States.utf8
## [2] LC_CTYPE=English_United States.utf8
## [3] LC_MONETARY=English_United States.utf8
## [4] LC_NUMERIC=C
## [5] LC_TIME=English_United States.utf8
##
## time zone: Europe/Budapest
## tzcode source: internal
##
```

```
## attached base packages:
## [1] stats      graphics  grDevices  utils      datasets  methods   base
##
## other attached packages:
## [1] flextable_0.9.5  ggplot2_3.4.4    reticulate_1.34.0 knitr_1.45
## [5] rmarkdown_2.26
##
## loaded via a namespace (and not attached):
##  [1] utf8_1.2.4          generics_0.1.3      fontLiberation_0.1.0
##  [4] xml2_1.3.6          lattice_0.21-9      httpcode_0.3.0
##  [7] digest_0.6.33       magrittr_2.0.3      evaluate_0.23
## [10] grid_4.3.2          fastmap_1.1.1       jsonlite_1.8.8
## [13] Matrix_1.6-1.1      zip_2.3.0           crul_1.4.0
## [16] tinytex_0.49        promises_1.2.1      fansi_1.0.6
## [19] scales_1.3.0        fontBitstreamVera_0.1.1 textshaping_0.3.7
## [22] cli_3.6.1           shiny_1.8.0         rlang_1.1.2
## [25] fontquiver_0.2.1    crayon_1.5.2        ellipsis_0.3.2
## [28] munsell_0.5.0       yaml_2.3.7          withr_2.5.2
## [31] gfonts_0.2.0        gdtools_0.3.7       officer_0.6.5
## [34] tools_4.3.2         uuid_1.2-0          dplyr_1.1.4
## [37] colorspace_2.1-0    httpuv_1.6.13       curl_5.2.0
## [40] vctrs_0.6.5         R6_2.5.1            mime_0.12
## [43] png_0.1-8           lifecycle_1.0.4     ragg_1.2.7
## [46] pkgconfig_2.0.3     pillar_1.9.0        later_1.3.2
## [49] gtable_0.3.4        data.table_1.14.10  glue_1.6.2
## [52] Rcpp_1.0.11         systemfonts_1.0.5   highr_0.10
## [55] xfun_0.41           tibble_3.2.1        tidyselect_1.2.0
## [58] farver_2.1.1        xtable_1.8-4        htmltools_0.5.7
## [61] labeling_0.4.3      compiler_4.3.2      askpass_1.2.0
## [64] openssl_2.1.1
```

The `reticulate` package configures the environment to run Python from R. First, a virtual environment is created and initiated using the following code:

```
#create virtual environment
reticulate::virtualenv_create(envname = my_envname,
                             python = path_to_the_python_exe_file)
#initiate virtual environment
reticulate::use_virtualenv(virtualenv = my_envname)
```

After successfully setting up the environment, the `pandas` and `scipy` packages are installed using the following code:

```
reticulate::virtualenv_install(envname = my_envname,
                              packages = c("pandas", "scipy"))
```

To check the versions of installed Python packages, practitioners can run the following command:

```
import importlib.metadata
```

```

#list of packages to check
packages = ["pandas", "scipy", "numpy"]

#print installed versions
for pkg in packages:
    try:
        version = importlib.metadata.version(pkg)
        print(f"{pkg}: {version}")
    except importlib.metadata.PackageNotFoundError:
        print(f"{pkg} is not installed.")

## pandas: 2.2.3
## scipy: 1.11.4
## numpy: 1.26.2

```

3 Data Import

The first step in any data analysis is usually data import and processing. In credit risk model validation, practitioners typically access various datasets stored in centralized systems or imported from different sources. The most common approach combines data from specific risk marts (often stored on different SQL servers) with other IT systems/applications that collect relevant data.

For simplicity, this section focuses on creating data directly within the session and demonstrating how data can be exchanged between Python and R. Let's start by generating a dummy data frame in R.

```

#rating scale data frame
rating <- paste0("R0", 1:7)
no <- c(170, 118, 274, 100, 91, 196, 51)
nb <- c(3, 10, 47, 31, 43, 122, 44)
rs <- data.frame(rating, no, nb, odr = nb/no)

```

The created data frame can be printed directly in the report or formatted using a package such as `flextable`. The following code prints the data frame as is and then formats it using the `flextable` function from the same package.

Print rs data frame:

```

rs

##   rating  no  nb      odr
## 1    R01 170   3 0.01764706
## 2    R02 118  10 0.08474576
## 3    R03 274  47 0.17153285
## 4    R04 100  31 0.31000000
## 5    R05  91  43 0.47252747
## 6    R06 196 122 0.62244898
## 7    R07  51  44 0.86274510

```

Print rs as flextable:

```
flextable(data = rs)
```

rating	no	nb	odr
R01	170	3	0.01764706
R02	118	10	0.08474576
R03	274	47	0.17153285
R04	100	31	0.31000000
R05	91	43	0.47252747
R06	196	122	0.62244898
R07	51	44	0.86274510

Once the `rs` data frame is created in `R`, it can be accessed and processed from `Python`. Let's confirm that `rs` is accessible and print it from `Python` session.

```
import pandas as pd
```

```
#access rs by running r.rs
```

```
rs_py = r.rs
```

```
#print rs_py
```

```
rs_py
```

```
##   rating    no    nb    odr
## 0    R01  170.0    3.0 0.017647
## 1    R02  118.0   10.0 0.084746
## 2    R03  274.0   47.0 0.171533
## 3    R04  100.0   31.0 0.310000
## 4    R05   91.0   43.0 0.472527
## 5    R06  196.0  122.0 0.622449
## 6    R07   51.0   44.0 0.862745
```

Let's add a column to `rs_py` in `Python`:

```
rs_py["segment"] = "Retail"
```

Now print the data frame as a `flextable` in `R`:

```
flextable(data = py$rs_py)
```

rating	no	nb	odr	segment
R01	170	3	0.01764706	Retail
R02	118	10	0.08474576	Retail
R03	274	47	0.17153285	Retail
R04	100	31	0.31000000	Retail

rating	no	nb	odr segment
R05	91	43	0.47252747 Retail
R06	196	122	0.62244898 Retail
R07	51	44	0.86274510 Retail

Besides the demonstrated ability to pass objects between R and Python, **reticulate** also provides an option to run Python scripts and functions within R. This is especially useful when part of the data import and processing is prepared in Python. The following code demonstrates the execution of a Python command within R:

```
#run python command
py_run_string("x = 10")
#print object x in r
py$x
```

```
## [1] 10
```

To demonstrate how practitioners can execute or source a Python script from R, let's first write a Python file containing a function and a list.

Python code:

```
#define file name
file_name = "python_script.py"
#define file content
content = """def hello_world():
    print("Hello, World!")
import numpy as np
np_list = np.array([1, 2, 3, 4, 5])
"""
#write file
with open(file_name, "w") as file:
    _ = file.write(content)
```

Once the Python script is ready, we can source or execute it within R as follows.

R script to source a Python script:

```
#source python script
source_python("python_script.py")
#print np_list object
py$np_list
```

```
## [1] 1 2 3 4 5
```

```
#run python function
py_run_string("hello_world()")
```

```
## Hello, World!
```

R script to execute a Python script:

```
#execute python script
py_run_file("python_script.py")
#print np_list object
py$np_list
```

```
## [1] 1 2 3 4 5
```

```
#run python function
py_run_string("hello_world()")
```

```
## Hello, World!
```

4 Descriptive Statistics and Summary Tables

After successfully importing the data, practitioners typically summarize specific metrics and data subject to analysis and validation. This step often involves printing and presenting tables in different formats, emphasizing specific figures or points that require attention.

The following examples demonstrate how to print and format tables primarily using the **flextable** package. For this purpose, we will use the previously defined data frame **rs** from the last section. R script:

```
#print rs
rs
```

```
##   rating no  nb      odr
## 1    R01 170   3 0.01764706
## 2    R02 118  10 0.08474576
## 3    R03 274  47 0.17153285
## 4    R04 100  31 0.31000000
## 5    R05  91  43 0.47252747
## 6    R06 196 122 0.62244898
## 7    R07  51  44 0.86274510
```

```
#print rs as flextable
flextable(data = rs)
```

rating	no	nb	odr
R01	170	3	0.01764706
R02	118	10	0.08474576
R03	274	47	0.17153285
R04	100	31	0.31000000
R05	91	43	0.47252747
R06	196	122	0.62244898
R07	51	44	0.86274510

```
#format odr column to 2 decimals
rs.f <- flextable(rs) |>
  colformat_double(j = 4,
                  digits = 2)
rs.f
```







rating	no	nb	odr
R01	170	3	0.02
R02	118	10	0.08
R03	274	47	0.17
R04	100	31	0.31
R05	91	43	0.47
R06	196	122	0.62
R07	51	44	0.86

Sometimes data aggregation consolidates figures from the different countries. The `flextable` package provides easy to use ways of incorporating images to the table. The following code showcase how country flags can be added to the table.

```
#random seed
set.seed(1234)
#simulate exposure values
Average_Exposure <- runif(n = 6,
                        min = 100,
                        max = 1000)
Average_Exposure <- round(x = Average_Exposure)
#create country level data
cd <- data.frame(Country = "",
                Average_Exposure = Average_Exposure)
#path to the country flags
flags <- c("me.png",
          "mk.png",
          "si.png",
          "rs.png",
          "ba.png",
          "hr.png")
#flextable
cd.f <- flextable(cd) |>
  compose(j = 1,
         value = as_paragraph(as_image(src = flags,
                                       width = 0.75,
                                       height = 0.75)))) |>
  autofit()
```



```
#print flextable
cd.f
```

Country	Average_Exposure
	202
	660
	648
	661
	875
	676

5 Data Visualization

Data visualization is an essential step in any data analysis, including the validation of IRB models. The way practitioners visualize data largely depends on the type of data and the context of validation. Both R and Python have well-developed graphical engines.

The following examples demonstrate two commonly used methods for presenting data. The first method involves inline plots, where graphs are embedded directly into tables to provide additional insights for practitioners. The second method uses standard visualizations, such as histograms and line charts, to display data separately in a more traditional format.

Let's start with inline plots embedded in flextable.

R script for inline plots:

```
#dummy data frame at the country level
cd <- data.frame(Country = rep(x = "", times = 6),
                 Distribution = "")
#simulated data per country
sim.data <- lapply(X = 1:nrow(cd),
                  FUN = function(x) rnorm(n = 100))
#flags to be added to the table
flags <- c("me.png",
           "mk.png",
           "si.png",
           "rs.png",
           "ba.png",
           "hr.png")
```

```

#function to generate histograms as images
generate.hist <- function(data, filename) {
  p <- ggplot(data.frame(x = data), aes(x)) +
    geom_histogram(bins = 10, fill = "gray", color = "black") +
    theme_void()
  ggsave(filename, plot = p, width = 2, height = 1.5, dpi = 100)
}

#generate histogram images
hist.files <- paste0("hist_", 1:length(sim.data), ".png")
mapply(generate.hist, sim.data, hist.files)

```

```
[1] "hist_1.png" "hist_2.png" "hist_3.png" "hist_4.png" "hist_5.png" [6] "hist_6.png"
```

```

#prepare flextable
ft <- flextable(cd) |>
  compose(j = 1, value = as_paragraph(as_image(src = flags,
                                                width = 0.75,
                                                height = 0.75))) |>
  compose(j = 2, value = as_paragraph(as_image(src = hist.files,
                                                width = 0.75,
                                                height = 0.75))) |>
  autofit()
#print flextable
ft

```

Country	Distribution
	
	
	
	
	
	

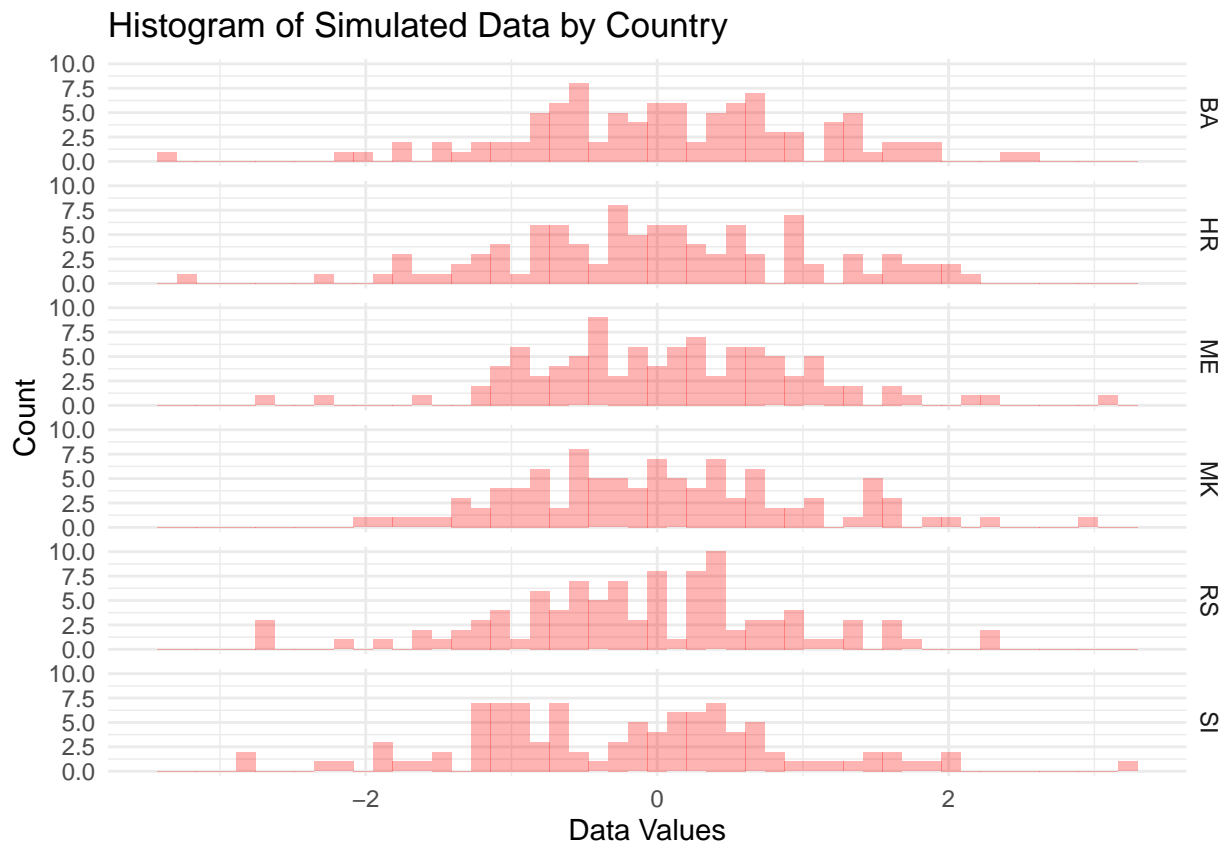
R script for ggplot2 graph:

```
library(ggplot2)

#iso country code
iso <- c("ME", "MK", "SI", "RS", "BA", "HR")
#convert the named list into a dataframe
sim.data <- unname(c(sim.data, recursive = TRUE))
db <- data.frame(Country = rep(x = iso,
                              time = 100),
                Data = sim.data)

#distribution of Data variable by country
gg <- ggplot(db, aes(x = Data)) +
  geom_histogram(alpha = 0.30, bins = 50, fill = "red") +
  facet_grid(Country ~ .) +
  theme_minimal() +
  labs(title = "Histogram of Simulated Data by Country",
       x = "Data Values",
       y = "Count")

gg
```



6 Quantitative Testing

One of the most critical aspects of IRB model validation is quantitative testing. For this task, practitioners typically perform a predefined set of statistical tests to examine and validate the model from different perspectives. Key aspects assessed in quantitative testing include model structure, model calibration, and the Margin of Conservatism. More details on specific tests and approaches for Probability of Default (PD) modeling can be found [here](#).

Since this exercise aims to showcase aspects of report automation, the following code snippets will demonstrate how testing procedures can be executed, formatted, and presented within tables.

Let's start with the exact binomial test, commonly used for testing PD models.

R script:

```
#set seed
set.seed(2211)
#add calibrated pds to the rating scale rs
rs$pd <- rs$odr - runif(n = nrow(rs), min = -0.10, max = 0.10)
#print rs
rs
```

```
##   rating no  nb      odr      pd
## 1   R01 170   3 0.01764706 0.0004717041
## 2   R02 118  10 0.08474576 0.1356595537
## 3   R03 274  47 0.17153285 0.2334632658
## 4   R04 100  31 0.31000000 0.2276874342
## 5   R05  91  43 0.47252747 0.3840618209
## 6   R06 196 122 0.62244898 0.6803554137
## 7   R07  51  44 0.86274510 0.8417771827
```

```
#run exact binomial test
rs$binom.test <- pbinom(q = rs$nb - 1,
                      size = rs$no,
                      prob = rs$pd,
                      lower.tail = FALSE)
#print the p-values
rs$binom.test
```

```
## [1] 0.00007959578 0.96682999912 0.99495191372 0.03590130777 0.05299616355
## [6] 0.96379413142 0.43041998679
```

Once the test results are available, practitioners typically compare the observed p-value to a predefined threshold. Let's assume the threshold is set at 5% for this exercise. The following code will compare the observed p-value against this threshold and format the results accordingly.

```
#define threshold
threshold <- 0.05
#define color
col <- ifelse(rs$binom.test < threshold, "red", "green")
#prepare the table with results
rs.ft <- flextable(rs) |>
```

```

color(j = 6,
      color = col) |>
colformat_double(j = 6,
                 digits = 2) |>
autofit()
#print the results
rs.ft

```

rating	no	nb	odr	pd	binom.test
R01	170	3	0.01764706	0.0004717041	0.00
R02	118	10	0.08474576	0.1356595537	0.97
R03	274	47	0.17153285	0.2334632658	0.99
R04	100	31	0.31000000	0.2276874342	0.04
R05	91	43	0.47252747	0.3840618209	0.05
R06	196	122	0.62244898	0.6803554137	0.96
R07	51	44	0.86274510	0.8417771827	0.43

Similarly to the examples from the previous sections, quantitative procedures can be executed in Python and printed and formatted in R. The following code demonstrates the execution of the same statistical test in Python.

```

from scipy.stats import binom

#access r object
rs_py = r.rs

#run exact binomial test
rs_py["binom_test"] = 1 - binom.cdf(rs_py["nb"] - 1,
                                   rs_py["no"],
                                   rs_py["pd"])

#print rs_py
rs_py

```

```

## rating no nb odr pd binom.test binom_test
## 0 R01 170.0 3.0 0.017647 0.000472 0.000080 0.000080
## 1 R02 118.0 10.0 0.084746 0.135660 0.966830 0.966830
## 2 R03 274.0 47.0 0.171533 0.233463 0.994952 0.994952
## 3 R04 100.0 31.0 0.310000 0.227687 0.035901 0.035901
## 4 R05 91.0 43.0 0.472527 0.384062 0.052996 0.052996
## 5 R06 196.0 122.0 0.622449 0.680355 0.963794 0.963794
## 6 R07 51.0 44.0 0.862745 0.841777 0.430420 0.430420

```

7 Alternative Methods and Report Customization

Although the presented examples demonstrate only a small part of the technical details of report automation, they provide a solid basis for extension to real-world cases. Practitioners can opt for formats other than the presented PDF format, such as MS Word, MS PowerPoint, or HTML. When evaluating different formats, practitioners should consider their pros and cons, as not all formatting options are available.

Other packages besides the R packages used for tables and graphs are available. Among the most popular are `tinytable` and `gt`. For practitioners who prefer Python over R, the same tasks can be accomplished in Python.

Finally, successful report automation depends not solely on the choice of statistical programs or packages but on the overall process design. Practitioners should prioritize process design as a key factor in ensuring a flexible and maintainable reporting system.