Statistical Data Cleaning for Official Statistics with R

Mark van der Loo, Statistics Netherlands

CBS R&D Methodology

UNECE Meeting on Statistical Data Editing 2020
I’M THE PERFECT ABSTRACTION OF A DUCK!

MAYBE ARE YOU SLIGHTLY TOO ABSTRACT?
Abstraction?

SO MANY LAYERS!

SO CONFUSING!

source: https://thevaluable.dev/abstraction-type-software-example/
Abstraction?

source: https://thevaluable.dev/abstraction-type-software-example/
Abstraction?

source: https://thevaluable.dev/abstraction-type-software-example/
Abstraction?

source: https://thevaluable.dev/abstraction-type-software-example/
Abstraction?

source: https://thevaluable.dev/abstraction-type-software-example/
Abstraction?

source: https://thevaluable.dev/abstraction-type-software-example/
Abstraction?

source: https://thevaluable.dev/abstraction-type-software-example/
Tree by Piet Mondriaan

1909

1911

1912
Example: Data validation

Example: Data validation

Definition 3. A data validation function is a surjective function

\[ v : D^K \rightarrow \{ \text{False, True} \} \]

Data Validation

Mark P.J. van der Loo and Edwin de Jonge

Keywords: data quality, data cleaning

Abstract: Data validation is the activity where one decides whether or not a particular data set is fit for a given purpose. The decision is based on testing observed data against prior expectations that a plausible data set is assumed to satisfy. Examples of prior expectations range widely. They include natural limits on variables (height cannot be negative), restrictions on combinations of multiple variables (a man cannot be pregnant), combinations of multiple opinions (a mother cannot be younger than her child), and combinations of multiple data sources (import value of country A from country B cannot equal the export value of country B from country A). Besides the strict logical constraints mentioned in the examples, there are often softer constraints based on human experience. For example, one may not expect a certain economic sector to grow more than 5% in a quarter. Here, the 5% limit does not represent a physical impossibility but rather a limit based on past experience. Since one must decide in the end whether a data set is usable for its intended purpose, we treat such assessments on equal footing.

The purpose of this article is to formalize the definition of data validation and to demonstrate some of the properties that can be derived from this definition. In particular, it is shown how a formal view of the concept permits a classification of data validation rules (assertions), allowing them to be ordered in increasing levels of "complexity." Here, the term "complexity" refers to the amount of different types of information necessary to evaluate a validation rule. A formal definition also permits development of tools for automated validation and automated reasoning about data validation. Finally, some subtleties arising from combining possibly many such requirements are pointed out.

Example: Validate

Learning to use validate

library(validate)
retailers <- read.csv("supermarkets.csv")

rules <- validator(
  total.rev - total.costs == profit
, mean(profit) >= 10
)
result <- confront(retailers, rules, key="id")

head(as.data.frame(result), 3)
We are extending validate

Increased support for

- long-form (transmission) data formats
- time series data
- grouped checks

Example: checking for gaps in (time) series

```r
library(validate)

is_linear_sequence(c(1,2,3))
## [1] TRUE

is_linear_sequence(c(1,4,5))
## [1] FALSE
```
Example: Validation Report Standard
Example: Validation Report Standard

Design of a generic machine-readable validation report structure

M. van der Loo and O. ten Bosch (2017) *Design of a generic machine-readable report structure* Deliverable of ESSnet ValidatFOSS

Example: Validation Report Standard

Triton (SW)  VTL
Example: Validation Report Standard
Example: imputation
Example: imputation
Example: (s)imputation

An imputation procedure is specified by
1. The variable to impute
2. An imputation model
3. Predictor variables

The simputation interface
impute_<model>(data, <imputed vars> ~ <predictor vars>, [options])
Example: simputation

```r
head(retailers, 3)

##     staff turnover other.rev total.rev
## 1     75      NA        NA       1130
## 2      9     1607      NA       1607
## 3    NA     6886      -33       6919
```
## simputation

```r
retailers %>%
impute_lm(other.rev ~ turnover) %>%
head(3)
```

<table>
<thead>
<tr>
<th></th>
<th>staff</th>
<th>turnover</th>
<th>other.rev</th>
<th>total.rev</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>75</td>
<td>NA</td>
<td>NA</td>
<td>1130</td>
</tr>
<tr>
<td>2</td>
<td>9</td>
<td>1607</td>
<td>5427.113</td>
<td>1607</td>
</tr>
<tr>
<td>3</td>
<td>NA</td>
<td>6886</td>
<td>-33.000</td>
<td>6919</td>
</tr>
</tbody>
</table>
Example: simputation

```r
retailers %>%
impute_lm(other.rev ~ turnover) %>%
impute_lm(other.rev ~ staff) %>%
head(3)
```

<table>
<thead>
<tr>
<th>staff</th>
<th>turnover</th>
<th>other.rev</th>
<th>total.rev</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>75</td>
<td>NA</td>
<td>4114.065</td>
</tr>
<tr>
<td>2</td>
<td>9</td>
<td>1607</td>
<td>5427.113</td>
</tr>
<tr>
<td>3</td>
<td>NA</td>
<td>6886</td>
<td>-33.000</td>
</tr>
</tbody>
</table>
Example: simputation

code:
```
retailers %>%
impute_rlm(other.rev ~ turnover) %>%
impute_rlm(other.rev ~ staff) %>%
head(3)
```

```
## staff turnover other.rev total.rev
## 1 75 NA 64.88174 1130
## 2 9 1607 17.25247 1607
## 3 NA 6886 -33.00000 6919
```
Modularity and composability

We call a system **modular** when it is composed of various parts that can be linked to each other.

We call a system **composable** when it shows no emergent behaviour.
A hard composability problem

parameters

data → process → data'
A hard composability problem
A hard composability problem

Flow of data

Input data

Rules, parameters

Log

Step 1

Step 2

Step 3

Clean data

Flow of metadata
Example: cleaning SBS data

```r
# read data, rules
supermarkets <- read.csv("data/input.csv")
rules <- validator(.file="data/rules.yaml")

# error localization, imputation, adjusting
supermarkets <- replace_errors(supermarkets, rules)
A <- is.na(supermarkets)
supermarkets <- impute_mf(supermarkets, . ~ .)
supermarkets <- match_restrictions(supermarkets, rules, adjust=A)

# write output
write.csv(supermarkets, "data/clean_data.csv", row.names=FALSE)
```
Example: cleaning SBS data

Before

| V18  | abs(profit - (total.rev - total.costs)) < 0.01 |
| V04  | abs(turnover + other.rev - total.rev) < 0.01 |
| V11  | (profit - 0.6 * turnover) <= 0.01          |
| V17  | (total.costs - staff.costs) >= -0.01       |
| V16  | (total.rev - 2 * vat) <= 0.01              |
| V15  | (turnover - 2 * vat) <= 0.01               |
| V09  | (profit - 250 * staff) <= 0.01             |
| V03  | (other.rev - 0) >= -0.01                   |
| V14  | (other.rev - 0.5 * vat) <= 0.01            |
| V13  | (staff.costs - 100 * staff) <= 0.01        |
| V12  | (profit - 0.6 * turnover) <= 0.01          |
| V08  | (other.rev - 1000 * staff) <= 0.01         |
| V07  | (turnover - 1000 * staff) <= 0.01          |
| V06  | (other.rev - 1000 * staff) <= 0.01         |
| V05  | (staff.costs - 0) >= -0.01                 |
| V02  | (turnover - 0) >= -0.01                    |
| V01  | (staff - 0) >= -0.01                       |

After

| V18  | abs(profit - (total.rev - total.costs)) < 0.01 |
| V17  | (total.costs - staff.costs) >= -0.01           |
| V16  | (total.rev - 2 * vat) <= 0.01                  |
| V15  | (turnover - 2 * vat) <= 0.01                   |
| V14  | (other.rev - 0.5 * vat) <= 0.01                |
| V13  | (staff.costs - 100 * staff) <= 0.01            |
| V12  | (profit - 0.6 * turnover) <= 0.01              |
| V11  | (profit - 0.6 * turnover) <= 0.01              |
| V10  | (profit - 250 * staff) <= 0.01                 |
| V09  | (profit - 250 * staff) <= 0.01                 |
| V08  | (total.rev - 1000 * staff) <= 0.01             |
| V07  | (turnover - 1000 * staff) <= 0.01              |
| V06  | (other.rev - 1000 * staff) <= 0.01             |
| V05  | (staff.costs - 0) >= -0.01                    |
| V04  | abs(turnover + other.rev - total.rev) < 0.01   |
| V03  | (other.rev - 0) >= -0.01                      |
| V02  | (turnover - 0) >= -0.01                       |
| V01  | (staff - 0) >= -0.01                          |
A method for deriving information from running R code

By Mark P.J. van der Loo

Abstract

This paper describes a method to derive information from running R code. The method can be used to monitor the execution of R code, such as in a production environment, to detect errors or to gather metrics on the performance of a code. The method is based on the idea of instrumenting R code with additional code that captures execution events and annotates them with metadata, such as timing information. The captured events are then processed to derive useful information, such as error rates or execution times. This method can be particularly useful for understanding the performance of large, complex R code bases in production environments.

Introduction

R is a language designed for statistical computing and graphics. The language is widely used in research and industry for data analysis and visualization. However, the traditional ways of monitoring R code execution, such as through console logs or output files, can be inadequate for understanding the performance of large code bases in production environments. This paper proposes a method to instrument R code and derive performance metrics from the execution events.

In the last instrument every value that can be instrumented is replaced with a load symbol, and read, writing values are replaced with the number of the load symbol.

1. Introduction

1.1. The basis of the method

The basis of the method is to instrument the code with additional code that captures execution events and annotates them with metadata, such as timing information. The captured events are then processed to derive useful information, such as error rates or execution times. This method can be particularly useful for understanding the performance of large, complex R code bases in production environments.

The R Journal Vol. 13/2, 2020, pages 453–464
ISSN 2073-3202

MARK P.J. VON DER LOO
Statistiek Werkwijze

Monitoring data in R with the lumberjack package

Mark P.J. van der Loo
Statistiek Werkwijze

Abstract

Monitoring data while it is processed and transformed is a crucial step in the management of data and the control of processes. The lumberjack package is a powerful tool for monitoring R code execution. It allows developers to monitor R code execution efficiently and effectively, providing real-time feedback on the performance of R code. This is particularly useful in situations where the performance of R code is critical, such as in high-frequency trading or in scientific computations.

Keywords: Data Quality, Process Monitoring, Debugging, Logging

3. Introduction

3.1. The basic idea

The basic idea behind the lumberjack package is to instrument the code with additional code that captures execution events and annotates them with metadata, such as timing information. The captured events are then processed to derive useful information, such as error rates or execution times. This method can be particularly useful for understanding the performance of large, complex R code bases in production environments.

The R Journal Vol. 13/2, 2020, pages 453–464
ISSN 2073-3202

MARK P.J. VON DER LOO
Statistiek Werkwijze

Monitoring data in R with the lumberjack package

Mark P.J. van der Loo
Statistiek Werkwijze

Abstract

Monitoring data while it is processed and transformed is a crucial step in the management of data and the control of processes. The lumberjack package is a powerful tool for monitoring R code execution. It allows developers to monitor R code execution efficiently and effectively, providing real-time feedback on the performance of R code. This is particularly useful in situations where the performance of R code is critical, such as in high-frequency trading or in scientific computations.

Keywords: Data Quality, Process Monitoring, Debugging, Logging

3. Introduction

3.1. The basic idea

The basic idea behind the lumberjack package is to instrument the code with additional code that captures execution events and annotates them with metadata, such as timing information. The captured events are then processed to derive useful information, such as error rates or execution times. This method can be particularly useful for understanding the performance of large, complex R code bases in production environments.
Example: cleaning SBS data

```r
# read data, rules
supermarkets <- read.csv("data/input.csv")
rules <- validator(.file="data/rules.yaml")

start_log(supermarkets, logger = cellwise$new()) # <- ADD ONE LINE

# error localization, imputation, adjusting
supermarkets <- replace_errors(supermarkets, rules)
A <- is.na(supermarkets)
supermarkets <- impute_mf(supermarkets, . ~ .)
supermarkets <- match_restrictions(supermarkets, rules, adjust=A)

# write output
write.csv(supermarkets, "data/clean_data.csv", row.names=FALSE)
```
Example: cleaning SBS data

lumberjack::run_file("cleanup.R")

## Dumped a log at supermarkets_cellwise.csv
read.csv("supermarkets_cellwise.csv")[100:102, ]

## step time srcref
## 100 4 2020-08-31 11:29:01 CEST cleanup.R#20-20
## 101 4 2020-08-31 11:29:01 CEST cleanup.R#20-20
## 102 4 2020-08-31 11:29:01 CEST cleanup.R#20-20

## expression key variable old
## 100 supermarkets <- impute_mf(supermarkets, . ~ .) 70799197 total.rev NA
## 101 supermarkets <- impute_mf(supermarkets, . ~ .) 70799197 turnover NA
## 102 supermarkets <- impute_mf(supermarkets, . ~ .) 71774143 other.rev NA

## new
## 100 626.8672
## 101 598.3333
## 102 50.7900
Conclusions
Take-home messages

Abstraction ≠ hiding
Don’t hide the details, get rid of them!

Modularity is not enough
Composability is key!
References, tutorials

Tutorials:
- uRos2019
- useR2019
- EESW 2020
- ISM 2020

data-cleaning.org