Validatetools

Validatetools: Check and resolve contradictory rule sets

Edwin de Jonge en Mark van der Loo

uRos2018 Tutorial Session, The Hague


CAUTION: BAD DATA
BAD DATA QUALITY MAY RESULT IN FRUSTRATION AND LEAD TO DROP KICKING YOUR COMPUTER

Desirable data cleaning properties:

- Reproducible data checks.
- Automate repetitive data checking (e.g. monthly/quarterly).
- Monitor data improvements / changes.
- **How** do this systematically?

RULES

Data Cleaning philosophy

- **“Explicit is better than implicit”**.
- Data rules are solidified **domain knowledge**.
- Store these as **validation rules** and apply these when necessary.

Advantages:

- Easy checking of rules: data validation.
- Data quality statistics: how often is each rule violated?
- Allows for reasoning on rules: which variables are involved in errors? How do errors affect the resulting statistic?
- Simplifies rule changes and additions.

Refresh: R package validate

With package validate you can formulate explicit rules that data must conform to:

```r
library(validate)
check_that( data.frame(age=160, job = "no", income = 3000),
  age >= 0,
  age < 150,
  job %in% c("yes", "no"),
  if (job == "yes") age >= 16,
  if (income > 0) job == "yes"
)
```

Rules (2)

A lot of datacleaning packages are using validate rules to facilitate their work.

- `validate`: validation **checks** and data **quality stats** on data.
- `errorlocate`: to find **errors** in variables (in stead of records)
- `rspa`: data **correction** under data constraints
- `deductive`: deductive **correction**
- `dcmodify`: deterministic **correction** and **imputation**.

Growing pains

- using explicit rules is great, but when succesful create new and unforeseen issues.

Issues:

- Many variables.
- Many rules, checks or constraints on the data.
- Many sub-domains with specialized rules.
- Many persons working on same rule set.

Assigment 1 (5 min.)

- Collect in small groups:
  - What is maximum columns in a dataset you encountered in your office?
  - Can you give an indication of the maximum number of data validity rules that are checked in production process ?
  - How many persons are involved in checking and maintaining the rules?

Issues:

At CBS:

- Datasets with > 100 columns are common.
- Some systems have 100s of rules.
- Often multiple persons work on rule set.

Most of these issues are not technical, but **organisational** and **cognitive**.

- Does anyone has a clear oversight on a large rule dataset?
- If your co-worker adds a rule, this (may) interfere with the other rules.

Why-o-why `validatetools`?

- We have package `validate`, what is the need?

Because we'd like to...

- clean up rule sets (kind of meta-cleaning...).
- detect and resolve problems with rules:
  - Detect unintended rule **interactions**.
  - Detect **conflicting** rules.
  - Remove **redundant** rules.
  - **Substitute** values and **simplify** rules.
- check the rule set using formal logic (**without any data!**).
- solve these kind of fun problems :-)

Detect rule interactions

- The rules form a consistent system of constraints.
- A combination of rules may *overconstrain* a variable
- One simple option is look at the boundary of allowed values for each variable.

Assignment Check boundaries

1) What are the allowed values for age and `income`?

```
library(validatetools)
rules <- validator( age >= 18
                  , if (job == TRUE) age <= 70
                  , if (income > 0) job == TRUE
                  , income >= 0
                  )
```

2) Check this with `validatetools::detect_boundary_num`.

```r
library(validatetools)
rules <- validator( age >= 18
                  , if (job == TRUE) age <= 70
                  , if (income > 0) job == TRUE
                  , income >= 0
                  )
detect_boundary_num(rules)
```

Rule interactions:

- boundary check is ok, may does not check for forbidden intervals.
- when variable can only have one value, it is fixed.
- extreme case is when allowed range for a variable is empty: infeasibility

Problem: infeasibility

Problem

One or more rules in conflict: all data incorrect, because always one of the rules will be violated! (*and yes that happens when rule sets are large . . .*).

validatetools checks for feasiblity

```r
library(validatetools)
rules <- validator( is_adult = age >=21
                  , is_child = age < 18
                  )
is_infeasible(rules)
```

```
## [1] TRUE
```

KEEP CALM

AND

RESOLVE CONFLICT

Conflict, and now?

```r
rules <- validator( is_adult = age >=21
                  , is_child = age < 18
                  )
# Find out which rule would remove the conflict
detect_infeasible_rules(rules)
```

```
## [1] "is_adult"
```

```r
# And its conflicting rule(s)
is_contradicted_by(rules, "is_adult")
```

```
## [1] "is_child"
```

- One of these rules needs to be removed
- Which one? Depends on human assessment. . .

Assignment Find the conflicting rules

a) Open the file "infeasible_rules.txt" (e.g. `file.edit("infeasible_rules.txt")`). Can you see which records are in conflict?

b) Find which two rule(s) are causing the infeasibility in file "infeasible_rules.txt".

```
rules <- validator(.file = "infeasible_rules.txt")
is_infeasible(rules)
```

```
## [1] TRUE
```

```
# do your thing...
```

Detecting and removing redundant rules

- Often rule set contain redudent rules.
- This may seem not a problem, however:
  - it complicates the rule set
  - it makes automatic checking a lot more problematic.

Detecting and removing redundant rules

Rule r_1 may imply r_2, so r_2 can be removed.

```
rules <- validator( r1 = age >= 18
                  , r2 = age >= 12
                  )
detect_redundancy(rules)
```

```
##    r1    r2
## FALSE  TRUE
```

```
remove_redundancy(rules)
```

```
## Object of class 'validator' with 1 elements:
##  r1: age >= 18
```

Value substitution

In complex statistics, many rules are specific for sub domains/sub groups

- This can be mitigated by splitting the rule sets in different pieces
- But can also be handled by simplifying the rule set for each subdomain:
- Fill in a value into a variable (making it a constant) and simplify the remaining rules.

Value substitution

```
rules <- validator( r1 = if (gender == "male") weight > 50
                  , r2 = gender %in% c("male", "female")
                  )

substitute_values(rules, gender = "male")
```

```
## Object of class 'validator' with 2 elements:
##  r1           : weight > 50
##  .const_gender: gender == "male"
```

Assignment: Can you simplify this one?

a)

```r
validator( if (income > 0) age >= 16
         , age < 12
         )
```

```
## Object of class 'validator' with 2 elements:
##  V1: !(income > 0) | (age >= 16)
##  V2: age < 12
```

b) Use `simplify_conditional` to let validatetools do it.

A bit more complex reasoning, but still classical logic:

```r
rules <- validator( r1 = if (income > 0) age >= 16
                  , r2 = age < 12
                  )
# age > 16 is always FALSE so r1 can be simplified
simplify_conditional(rules)
```

```
## Object of class 'validator' with 2 elements:
##  r1: income <= 0
##  r2: age < 12
```

Assigment: all together now!

`simplify_rules` applies all simplification methods to the rule set.

a) If we know that job must be “yes”, can you see how this rule set can be simplified?

```r
rules <- validator( r1 = job %in% c("yes", "no")
                  , r2 = if (job == "yes") income > 0
                  , r3 = if (age < 16) income == 0
                  )
```

b) Apply `simplify_rules(rules, job = "yes")`
c) Can you do the same using the other simplifying functions?

```r
rules <- validator( r1 = job %in% c("yes", "no")
                  , r2 = if (job == "yes") income > 0
                  , r3 = if (age < 16) income == 0
                  )
simplify_rules(rules, job = "yes")
```

```
## Object of class 'validator' with 3 elements:
##  r2        : income > 0
##  r3        : age >= 16
##  .const_job: job == "yes"
```

How does it work?

`validatetools`:

- reformulates rules into formal logic form.
- translates them into a mixed integer program for each of the problems.

Rule types

- *linear* restrictions
- *categorical* restrictions
- *if* statements with linear and categorical restrictions

If statement is Modus ponens:

$$\begin{aligned} & \text{if } P \text{ then } Q \\ \Leftrightarrow \quad & P \implies Q \\ \Leftrightarrow \quad & \neg P \vee Q \end{aligned}$$

Example

```r
rules <- validator(
  example = if (job == "yes") income > 0
)
```

$$r_{\text{example}}(x) = \text{job} \not\in \text{"yes"} \lor \text{income} > 0$$

```r
print(rules)
```

```
## Object of class 'validator' with 1 elements:
##  example: !(job == "yes") | (income > 0)
```

Addendum

Formal logic

Rule set S

A validation rule set S is a conjunction of rules r_i, which applied on record $\boldsymbol{x}$ returns `TRUE` (valid) or `FALSE` (invalid)

$$S(\boldsymbol{x}) = r_1(\boldsymbol{x}) \wedge \cdots \wedge r_n(\boldsymbol{x})$$

Note

- a record has to comply to each rule r_i.
- it is thinkable that two or more r_i are in conflict, making each record invalid.

Formal logic (2)

Rule $r_i(x)$

A rule a disjunction of atomic clauses:

$$r_i(x) = \bigvee_j C_i^j(x)$$

with:

$$C_i^j(\boldsymbol{x}) = \begin{cases} \boldsymbol{a}^T\boldsymbol{x} \leq b \\ \boldsymbol{a}^T\boldsymbol{x} = b \\ x_j \in F_{ij} \text{with } F_{ij} \subseteq D_j \\ x_j \notin F_{ij} \text{with } F_{ij} \subseteq D_j \end{cases}$$

Mixed Integer Programming

Each rule set problem can be translated into a mip problem, which can be readily solved using a mip solver.

`validatetools` uses `lpSolveApi`.

$$\begin{aligned} \text{Minimize } f(\mathbf{x}) &= 0; \\ \text{s.t. } \mathbf{R}\mathbf{x} &\leq \mathbf{d} \end{aligned}$$

with $\boldsymbol{R}$ and $\boldsymbol{d}$ the rule definitions and $f(\boldsymbol{x})$ is the specific problem that is solved.