Overview of simple outlier detection methods with their combination using dplyr and ruler packages.

**Prologue**

During the process of data analysis one of the most crucial steps is to identify and account for [outliers](https://en.wikipedia.org/wiki/Outlier), observations that have essentially different nature than most other observations. Their presence can lead to untrustworthy conclusions. The most complicated part of this task is **to define a notion of “outlier”**. After that, it is straightforward to identify them based on given data.

There are many techniques developed for outlier detection. Majority of them deal with numerical data. This post will describe the most basic ones with their application using dplyr and ruler packages.

After reading this post you will know:

* Most basic outlier detection techniques.
* A way to implement them using dplyr and ruler.
* A way to combine their results in order to obtain a new outlier detection method.
* A way to discover notion of “diamond quality” without prior knowledge of this topic (as a happy consequence of previous point).

**Overview**

We will perform an analysis with the goal to find not typical diamonds listed in [diamonds](http://ggplot2.tidyverse.org/reference/diamonds.html) dataset from ggplot2 package. Here one observation represents one diamond and is stored as a row in data frame.

The way we will do that is **by combining different outlier detection techniques to identify rows which are “strong outliers”**, i.e. which might by considered outliers based on several methods.

Packages required for this analysis:

library(dplyr)

library(tidyr)

library(ggplot2)

library(ruler)

**Outlier detection methods**

To do convenient outlier detection with ruler it is better to define notion of **non-outlier** in form of the rule “Observation **is not** an outlier if …”. This way actual outliers are considered as **rule breakers**, objects of interest of ruler package. **Note** that definition of non-outlier is essentially a definition of outlier because of total two possibilities.

**Z-score**

[Z-score](https://en.wikipedia.org/wiki/Standard_score), also called a standard score, of an observation is [broadly speaking] a distance from the population center measured in number of normalization units. The default choice for center is sample mean and for normalization unit is standard deviation.

⬛ *Observation is not an outlier based on z-score* if its absolute value of default z-score is lower then some threshold (popular choice is 3).

Here is the function for identifying non-outliers based on z-score:

isnt\_out\_z <- function(x, thres = 3, na.rm = TRUE) {

abs(x - mean(x, na.rm = na.rm)) <= thres \* sd(x, na.rm = na.rm)

}

It takes a numeric vector as input and returns logical vector of the same length indicating whether input value is a non-outlier.

**Z-score with MAD**

[Median Absolute Deviation](https://en.wikipedia.org/wiki/Median_absolute_deviation) is a robust normalization unit based on median as a population center. In order to use MAD “as a consistent estimator for the estimation of the standard deviation” one takes its value multiplied by a factor. This way base R function mad is implemented.

⬛ *Observation is not an outlier based on MAD* if its absolute value of z-score with median as center and MAD as normalization unit is lower then some threshold (popular choice is 3).

isnt\_out\_mad <- function(x, thres = 3, na.rm = TRUE) {

abs(x - median(x, na.rm = na.rm)) <= thres \* mad(x, na.rm = na.rm)

}

**Tukey’s fences**

[Tukey’s fences](https://en.wikipedia.org/wiki/Outlier#Tukey&%2339;s_fences) is a technique used in box plots. The non-outlier range is defined with \([Q\_1 – k(Q\_3 – Q\_1),~ Q\_3 + k(Q\_3 – Q\_1)]\), where \(Q\_1\) and \(Q\_3\) are the lower and upper quartiles respectively, \(k\) – some nonnegative constant (popular choice is 1.5).

⬛ *Observation is not an outlier based on Tukey’s fences* if its value lies in non-outlier range.

isnt\_out\_tukey <- function(x, k = 1.5, na.rm = TRUE) {

quar <- quantile(x, probs = c(0.25, 0.75), na.rm = na.rm)

iqr <- diff(quar)

(quar[1] - k \* iqr <= x) & (x <= quar[2] + k \* iqr)

}

**Mahalanobis distance**

All previous approaches were created for univariate numerical data. To detect outliers in multivariate case one can use [Mahalanobis distance](https://en.wikipedia.org/wiki/Mahalanobis_distance) to reduce to univariate case and then apply known techniques.

⬛ *Observation is not an outlier based on Mahalanobis distance* if its distance is not an outlier.

maha\_dist <- . %>% select\_if(is.numeric) %>%

mahalanobis(center = colMeans(.), cov = cov(.))

isnt\_out\_maha <- function(tbl, isnt\_out\_f, ...) {

tbl %>% maha\_dist() %>% isnt\_out\_f(...)

}

This function takes as input a data frame of interest (with possible non-numeric columns which are ignored) and function performing univariate outlier detection. It returns a logical vector of the same length as number of rows in input data frame.

**Using dplyr and ruler**

**Definition of non-outlier row**

Package ruler, based on dplyr grammar of data manipulation, offers tools for validating the following data units: data as a whole, group [of rows] as a whole, column as a whole, row as a whole, cell. Our primary interest is row as a whole. However, using this framework, we can construct several approaches for definition of the non-outlier row:

1. *Row is not an outlier based on some column* if it doesn’t contain outlier (computed based on the target column) on the intersection with that column. In other words, first a univariate outlier detection is performed based solely on data from target column and then all rows containing non-outliers are named non-outlier rows.
2. *Row is not an outlier based on Mahalanobis distance* if its distance (computed based on the selected numeric columns) is not an outlier.
3. *Row is not an outlier based on grouping* if it is a part of a non-outlier group [of rows]. A group [of rows] is not an outlier if its summary value is not an outlier among summary values of other groups.

**Note** that all listed approached depend on the choice of the univariate outlier detection method. We will use all three previously listed univariate techniques.

isnt\_out\_funs <- funs(

z = isnt\_out\_z,

mad = isnt\_out\_mad,

tukey = isnt\_out\_tukey

)

**Implementation**

**Column based non-outlier rows**

For diamonds dataset rules for column based non-outlier rows can be defined based on 7 numeric columns and 3 presented univariate detection methods. There is a convenient way of computing all them at once using scoped variant of dplyr::transmute():

Scope Variant:

The variants suffixed with \_if, \_at or \_all apply an expression (sometimes several) to all variables within a specified subset. This subset can contain all variables (\_all variants), a [vars()](https://dplyr.tidyverse.org/reference/vars.html) selection (\_at variants), or variables selected with a predicate (\_if variants).

The verbs with scoped variants are:

• mutate(), transmute() and summarise(). See summarise\_all().

• filter(). See filter\_all().

• group\_by(). See group\_by\_all().

• rename() and select(). See select\_all().

* arrange(). See arrange\_all()

There are three kinds of scoped variants. They differ in the scope of the variable selection on which operations are applied:

* Verbs suffixed with \_all() apply an operation on all variables.
* Verbs suffixed with \_at() apply an operation on a subset of variables specified with the quoting function [vars()](https://dplyr.tidyverse.org/reference/vars.html). This quoting function accepts tidyselect::vars\_select() helpers like [starts\_with()](https://tidyselect.r-lib.org/reference/starts_with.html). Instead of a [vars()](https://dplyr.tidyverse.org/reference/vars.html) selection, you can also supply an integerish vector of column positions or a character vector of column names.
* Verbs suffixed with \_if() apply an operation on the subset of variables for which a predicate function returns TRUE. Instead of a predicate function, you can also supply a logical vector.

Arguments

|  |  |
| --- | --- |
| **.tbl** | A tbl object. |
| **.funs** | A function fun, a quosure style lambda ~ fun(.) or a list of either form. |
| **.vars** | A list of columns generated by vars(), a character vector of column names, a numeric vector of column positions, or NULL. |
| **.predicate** | A predicate function to be applied to the columns or a logical vector. The variables for which .predicate is or returns TRUE are selected. This argument is passed to rlang::as\_function() and thus supports quosure-style lambda functions and strings representing function names. |
| **...** | Additional arguments for the function calls in .funs. These are evaluated only once, with tidy dots support. |

Grouping variables

Most of these operations also apply on the grouping variables when they are part of the selection. This includes:

* arrange\_all(), arrange\_at(), and arrange\_if()
* distinct\_all(), distinct\_at(), and distinct\_if()
* filter\_all(), filter\_at(), and filter\_if()
* group\_by\_all(), group\_by\_at(), and group\_by\_if()
* select\_all(), select\_at(), and select\_if()

This is not the case for summarising and mutating variants where operations are *not* applied on grouping variables. The behaviour depends on whether the selection is **implicit** (all and if selections) or **explicit** (at selections). Grouping variables covered by explicit selections (with summarise\_at(), mutate\_at(), and transmute\_at()) are always an error. For implicit selections, the grouping variables are always ignored. In this case, the level of verbosity depends on the kind of operation:

* Summarising operations (summarise\_all() and summarise\_if()) ignore grouping variables silently because it is obvious that operations are not applied on grouping variables.
* On the other hand it isn't as obvious in the case of mutating operations (mutate\_all(), mutate\_if(), transmute\_all(), and transmute\_if()). For this reason, they issue a message indicating which grouping variables are ignored.

diamonds %>% transmute\_if(is.numeric, isnt\_out\_funs)

## # A tibble: 53,940 x 21

## carat\_z depth\_z table\_z price\_z x\_z y\_z z\_z carat\_mad depth\_mad

##

## 1 TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE

## 2 TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE

## 3 TRUE FALSE FALSE TRUE TRUE TRUE TRUE TRUE FALSE

## 4 TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE

## 5 TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE

## # ... with 5.394e+04 more rows, and 12 more variables: table\_mad ,

## # price\_mad , x\_mad , y\_mad , z\_mad ,

## # carat\_tukey , depth\_tukey , table\_tukey ,

## # price\_tukey , x\_tukey , y\_tukey , z\_tukey

The result has outputs for 21 methods. Their names are of the form \_. So the name ‘carat\_z’ is interpreted as result of univariate method with name ‘z’ for column with name ‘carat’.

**Mahalanobis based non-outlier rows**

To define non-outlier rows based on Mahalanobis distance one should apply univariate method for distances computed for some subset of numeric columns. To simplify a little bit, we will one “subset” with all numeric columns and all listed methods:

diamonds %>%

transmute(maha = maha\_dist(.)) %>%

transmute\_at(vars(maha = maha), isnt\_out\_funs)

## # A tibble: 53,940 x 3

## maha\_z maha\_mad maha\_tukey

##

## 1 TRUE TRUE TRUE

## 2 TRUE FALSE FALSE

## 3 TRUE FALSE FALSE

## 4 TRUE TRUE TRUE

## 5 TRUE TRUE TRUE

## # ... with 5.394e+04 more rows

The result has outputs for 3 methods. Their names are considered as method names. **Note** that with this approach outlier rows are not only the ones far from multivariate center, but also the ones that are unnaturally close to it.

**Group based non-outlier rows**

Definition of non-outlier rows based on grouping depends on group summary function and univariate outlier detection method. As grouping column we will choose all non-numeric columns (cut, color and clarity) united into one called group (for later easier imputation of non-outlier rows). As reasonable summary functions we will choose mean value of some numeric column (total of 7 functions):

Unite.R

|  |
| --- |
| unite <- function(data, col, ..., sep = "\_", remove = TRUE, na.rm = FALSE) { |
|  | ellipsis::check\_dots\_unnamed() |
|  | UseMethod("unite") |
|  | } |
|  | #' @export |
|  | unite.data.frame <- function(data, col, ..., sep = "\_", remove = TRUE, na.rm = FALSE) { |
|  | var <- as\_string(ensym2(col)) |
|  |  |
|  | if (dots\_n(...) == 0) { |
|  | from\_vars <- set\_names(seq\_along(data), names(data)) |
|  | } else { |
|  | from\_vars <- tidyselect::eval\_select(expr(c(...)), data) |
|  | } |
|  |  |
|  | out <- data |
|  | if (remove) { |
|  | out <- out[setdiff(names(out), names(from\_vars))] |
|  | } |
|  |  |
|  | if (identical(na.rm, TRUE)) { |
|  | cols <- unname(map(data[from\_vars], as.character)) |
|  | rows <- transpose(cols) |
|  |  |
|  | united <- map\_chr(rows, function(x) paste0(x[!is.na(x)], collapse = sep)) |
|  | } else { |
|  | cols <- unname(as.list(data[from\_vars])) |
|  | united <- exec(paste, !!!cols, sep = sep) |
|  | } |
|  |  |
|  | first\_pos <- which(names(data) %in% names(from\_vars))[1] |
|  | out <- append\_col(out, united, var, after = first\_pos - 1L) |
|  | reconstruct\_tibble(data, out, if (remove) names(from\_vars)) |
|  | } |

data\_tbl <- diamonds %>%

unite(col = "group", cut, color, clarity)

compute\_group\_non\_outliers <- . %>%

# Compute per group mean values of columns

group\_by(group) %>%

summarise\_if(is.numeric, mean) %>%

ungroup() %>%

# Detect outliers among groups

mutate\_if(is.numeric, isnt\_out\_funs) %>%

# Remove unnecessary columns

select\_if(Negate(is.numeric))

data\_tbl %>% compute\_group\_non\_outliers()

## # A tibble: 276 x 22

## group carat\_z depth\_z table\_z price\_z x\_z y\_z z\_z carat\_mad

##

## 1 Fair\_D\_I1 FALSE TRUE TRUE TRUE TRUE TRUE TRUE FALSE

## 2 Fair\_D\_IF TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE

## 3 Fair\_D\_SI1 TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE

## 4 Fair\_D\_SI2 TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE

## 5 Fair\_D\_VS1 TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE

## # ... with 271 more rows, and 13 more variables: depth\_mad ,

## # table\_mad , price\_mad , x\_mad , y\_mad ,

## # z\_mad , carat\_tukey , depth\_tukey , table\_tukey ,

## # price\_tukey , x\_tukey , y\_tukey , z\_tukey

The result has outputs for 21 methods applied to the 276 groups. Their names are of the form \_. So the name ‘carat\_z’ is interpreted as result of method ‘z’ for summary function equal to mean value of ‘carat’ column. Column group defines names of the groupings.

**Exposure**

Column and Mahalanobis based definition of non-outlier rows can be expressed with row packs and group based – as group packs.

Group Packs

vs\_am\_rules **<-** . **%>%**

**dplyr::**[group\_by](https://dplyr.tidyverse.org/reference/group_by.html)(vs, am) **%>%**

**dplyr::**[summarise](https://dplyr.tidyverse.org/reference/summarise.html)(

nrow\_low = n(.) > 10,

nrow\_up = n(.) < 20,

rowmeans\_low = [rowMeans](https://rdrr.io/r/base/colSums.html)(.) > 19

)

[group\_packs](https://echasnovski.github.io/ruler/reference/rule-packs.html)(vs\_am = vs\_am\_rules, .group\_vars = [c](https://rdrr.io/r/base/c.html)(**"vs"**, **"am"**))

#> $vs\_am

#> A Group rule pack:

#> Functional sequence with the following components:

#>

#> 1. dplyr::group\_by(., vs, am)

#> 2. dplyr::summarise(., nrow\_low = n(.) > 10, nrow\_up = n(.) < 20, rowmeans\_low = rowMeans(.) > 19)

#>

#> Use 'functions' to extract the individual functions.

#>

Row Packs

some\_row\_mean\_rules <- . %>%

dplyr::slice(1:3) %>%

dplyr::mutate(row\_mean = rowMeans(.)) %>%

dplyr::transmute(

row\_mean\_low = row\_mean > 10,

row\_mean\_up = row\_mean < 20

)

all\_row\_sum\_rules <- . %>%

dplyr::mutate(row\_sum = rowSums(.)) %>%

dplyr::transmute(row\_sum\_low = row\_sum > 30)

row\_packs(

some\_row\_mean\_rules,

all\_row\_sum\_rules

)

#> [[1]]

#> A Row rule pack:

#> Functional sequence with the following components:

#>

#> 1. dplyr::slice(., 1:3)

#> 2. dplyr::mutate(., row\_mean = rowMeans(.))

#> 3. dplyr::transmute(., row\_mean\_low = row\_mean > 10, row\_mean\_up = row\_mean < 20)

#>

#> Use 'functions' to extract the individual functions.

#>

#> [[2]]

#> A Row rule pack:

#> Functional sequence with the following components:

#>

#> 1. dplyr::mutate(., row\_sum = rowSums(.))

#> 2. dplyr::transmute(., row\_sum\_low = row\_sum > 30)

#>

#> Use 'functions' to extract the individual functions.

#>

row\_packs\_isnt\_out <- row\_packs(

# Non-outliers based on some column

column = . %>% transmute\_if(is.numeric, isnt\_out\_funs),

# Non-outliers based on Mahalanobis distance

maha = . %>% transmute(maha = maha\_dist(.)) %>%

transmute\_at(vars(maha = maha), isnt\_out\_funs)

)

group\_packs\_isnt\_out <- group\_packs(

# Non-outliers based on grouping

group = compute\_group\_non\_outliers,

.group\_vars = "group"

)

Application of all those packs is called exposing process. The result is an exposure from which we can extract tidy data validation report using get\_report.

# Don't remove obeyers to compute total number of applied rules

full\_report <- data\_tbl %>%

expose(row\_packs\_isnt\_out, group\_packs\_isnt\_out,

.remove\_obeyers = FALSE) %>%

get\_report()

used\_rules <- full\_report %>%

distinct(pack, rule)

breaker\_report <- full\_report %>%

filter(!(value %in% TRUE))

used\_rules contains data about all definitions of non-outlier rows applied to data. They are encoded with combination of columns pack and rule.

breaker\_report contains data about data units that break certain rules. Packs column and maha has actual row numbers of data\_tbl listed in id column of report (for rows which should be considered as outliers).

Expose Process

Details

expose() applies all supplied rule packs to data, creates an exposure object based on results and stores it to attribute 'exposure'. It is guaranteed that .tbl is not modified in any other way in order to use expose() inside a pipe.

It is a good idea to name all rule packs: explicitly in ... (if they are supplied not inside list) or during creation with respective rule pack function. In case of missing name it is imputed based on possibly existing exposure attribute in .tbl and supplied rule packs. Imputation is similar to one in rules() but applied to every pack type separately.

Default value for .rule\_sep is the regular expression characters .\_. surrounded by non alphanumeric characters. It is picked to be used smoothly with dplyr's scoped verbs and rules() instead of funs(). In most cases it shouldn't be changed but if needed it should align with .prefix in rules().

Guessing

To work properly in some edge cases one should specify pack types with appropriate function. However with .guess equals to TRUE expose will guess the pack type based on its output after applying to .tbl. It uses the following features:

Presence of non-logical columns: if present then the guess is group pack. Grouping columns are guessed as all non-logical. This works incorrectly if some grouping column is logical: it will be guessed as result of applying the rule. Note that on most occasions this edge case will produce error about grouping columns define non-unique levels.

Combination of whether number of rows equals 1 (n\_rows\_one) and presence of .rule\_sep in all column names (all\_contain\_sep). Guesses are:

Data pack if n\_rows\_one == TRUE and all\_contain\_sep == FALSE.

Column pack if n\_rows\_one == TRUE and all\_contain\_sep == TRUE.

Row pack if n\_rows\_one == FALSE and all\_contain\_sep == FALSE. This works incorrectly if output has one row which is checked. In this case it will be guessed as data pack.

Cell pack if n\_rows\_one == FALSE and all\_contain\_sep == TRUE. This works incorrectly if output has one row in which cells are checked. In this case it will be guessed as column pack.

Examples

my\_rule\_pack <- . %>% dplyr::summarise(nrow\_neg = nrow(.) < 0)

my\_data\_packs <- data\_packs(my\_data\_pack\_1 = my\_rule\_pack)

# These pipes give identical results

mtcars %>%

expose(my\_data\_packs) %>%

get\_report()

#> Tidy data validation report:

#> # A tibble: 1 x 5

#> pack rule var id value

#> <chr> <chr> <chr> <int> <lgl>

#> 1 my\_data\_pack\_1 nrow\_neg .all 0 FALSE

mtcars %>%

expose(my\_data\_pack\_1 = my\_rule\_pack) %>%

get\_report()

#> Tidy data validation report:

#> # A tibble: 1 x 5

#> pack rule var id value

#> <chr> <chr> <chr> <int> <lgl>

#> 1 my\_data\_pack\_1 nrow\_neg .all 0 FALSE

# This throws an error because no pack type is specified for my\_rule\_pack

if (FALSE) {

mtcars %>% expose(my\_data\_pack\_1 = my\_rule\_pack, .guess = FALSE)

}

# Edge cases against using 'guess = TRUE' for robust code

group\_rule\_pack <- . %>%

dplyr::mutate(vs\_one = vs == 1) %>%

dplyr::group\_by(vs\_one, am) %>%

dplyr::summarise(n\_low = dplyr::n() > 10)

group\_rule\_pack\_dummy <- . %>%

dplyr::mutate(vs\_one = vs == 1) %>%

dplyr::group\_by(mpg, vs\_one, wt) %>%

dplyr::summarise(n\_low = dplyr::n() > 10)

row\_rule\_pack <- . %>% dplyr::transmute(neg\_row\_sum = rowSums(.) < 0)

cell\_rule\_pack <- . %>% dplyr::transmute\_all(rules(neg\_value = . < 0))

# Only column 'am' is guessed as grouping which defines non-unique levels.

if (FALSE) {

mtcars %>%

expose(group\_rule\_pack, .remove\_obeyers = FALSE, .guess = TRUE) %>%

get\_report()

}

# Values in `var` should contain combination of three grouping columns but

# column 'vs\_one' is guessed as rule. No error is thrown because the guessed

# grouping column define unique levels.

mtcars %>%

expose(group\_rule\_pack\_dummy, .remove\_obeyers = FALSE, .guess = TRUE) %>%

get\_report()

#> Tidy data validation report:

#> # A tibble: 64 x 5

#> pack rule var id value

#> <chr> <chr> <chr> <int> <lgl>

#> 1 group\_pack\_\_1 vs\_one 10.4.5.25 0 FALSE

#> 2 group\_pack\_\_1 vs\_one 10.4.5.424 0 FALSE

#> 3 group\_pack\_\_1 vs\_one 13.3.3.84 0 FALSE

#> 4 group\_pack\_\_1 vs\_one 14.3.3.57 0 FALSE

#> 5 group\_pack\_\_1 vs\_one 14.7.5.345 0 FALSE

#> 6 group\_pack\_\_1 vs\_one 15.3.57 0 FALSE

#> 7 group\_pack\_\_1 vs\_one 15.2.3.435 0 FALSE

#> 8 group\_pack\_\_1 vs\_one 15.2.3.78 0 FALSE

#> 9 group\_pack\_\_1 vs\_one 15.5.3.52 0 FALSE

#> 10 group\_pack\_\_1 vs\_one 15.8.3.17 0 FALSE

#> # … with 54 more rows

# Results should have in column 'id' value 1 and not 0.

mtcars %>%

dplyr::slice(1) %>%

expose(row\_rule\_pack) %>%

get\_report()

#> Tidy data validation report:

#> # A tibble: 1 x 5

#> pack rule var id value

#> <chr> <chr> <chr> <int> <lgl>

#> 1 data\_pack\_\_1 neg\_row\_sum .all 0 FALSE

mtcars %>%

dplyr::slice(1) %>%

expose(cell\_rule\_pack) %>%

get\_report()

#> Tidy data validation report:

#> # A tibble: 11 x 5

#> pack rule var id value

#> <chr> <chr> <chr> <int> <lgl>

#> 1 col\_pack\_\_1 neg\_value mpg 0 FALSE

#> 2 col\_pack\_\_1 neg\_value cyl 0 FALSE

#> 3 col\_pack\_\_1 neg\_value disp 0 FALSE

#> 4 col\_pack\_\_1 neg\_value hp 0 FALSE

#> 5 col\_pack\_\_1 neg\_value drat 0 FALSE

#> 6 col\_pack\_\_1 neg\_value wt 0 FALSE

#> 7 col\_pack\_\_1 neg\_value qsec 0 FALSE

#> 8 col\_pack\_\_1 neg\_value vs 0 FALSE

#> 9 col\_pack\_\_1 neg\_value am 0 FALSE

#> 10 col\_pack\_\_1 neg\_value gear 0 FALSE

#> 11 col\_pack\_\_1 neg\_value carb 0 FALSE

On the other hand, pack group defines group pack and is represented in breaker\_report with id 0. To obtain row outliers based on grouping we need to expand those rows with information about rows in the data that belong to those groups. This can be done using dplyr::left\_join():

group\_breakers <- breaker\_report %>%

# Filter group packs

filter(pack == "group") %>%

# Expand rows by matching group with its rows

select(-id) %>%

left\_join(

y = data\_tbl %>% transmute(var = group, id = 1:n()),

by = "var"

) %>%

select(pack, rule, var, id, value)

outliers <- bind\_rows(

breaker\_report %>% filter(pack != "group"),

group\_breakers

) %>%

select(pack, rule, id)

# Not all group based definitions resulted with outliers

outliers %>%

count(pack, rule) %>%

filter(pack == "group") %>%

print(n = Inf)

## # A tibble: 13 x 3

## pack rule n

##

## 1 group carat\_mad 37

## 2 group carat\_tukey 37

## 3 group carat\_z 29

## 4 group depth\_mad 1093

## 5 group depth\_tukey 1016

## 6 group depth\_z 156

## 7 group price\_mad 209

## 8 group price\_tukey 1146

## 9 group price\_z 44

## 10 group table\_mad 920

## 11 group table\_tukey 8

## 12 group table\_z 7

## 13 group z\_z 23

Tibble outliers contains data about outlier rows. Combination of columns pack and rule defines non-outlier/outlier definition approach and column id defines row number of input data frame that should be considered an outlier based on the definition.

Definitions with most outliers are as follows:

outliers %>%

count(pack, rule, sort = TRUE)

## # A tibble: 37 x 3

## pack rule n

##

## 1 maha maha\_mad 6329

## 2 maha maha\_tukey 5511

## 3 column price\_mad 5386

## 4 column price\_tukey 3540

## 5 column table\_mad 2560

## # ... with 32 more rows

Two out of three Mahalanobis based definition yielded the most row outliers.

**Combination**

Given outliers data frame, one can do whatever he/she wants to identify outliers. Here we will use the basic combination approach based on average score.

*Combined outlier detection score* for certain row can be defined as **share of applied methods that tagged it as outlier**. Alternatively one can define it just as number of those methods as it will only change absolute value of the result and not the order.

outlier\_score <- outliers %>%

group\_by(id) %>%

# nrow(used\_rules) equals total number of applied methods

summarise(score = n() / nrow(used\_rules))

# Top 10 outliers

outlier\_score %>% arrange(desc(score)) %>% slice(1:10)

## # A tibble: 10 x 2

## id score

##

## 1 26432 0.5777778

## 2 27416 0.5777778

## 3 27631 0.5777778

## 4 27131 0.4666667

## 5 23645 0.4222222

## 6 26445 0.4222222

## 7 26745 0.4000000

## 8 27430 0.4000000

## 9 15952 0.3777778

## 10 17197 0.3777778

Finally we will tag those rows as **strong outliers** which has score more than 0.2 (subjective threshold which should be researched more).

diam\_tbl <- diamonds %>%

mutate(id = 1:n()) %>%

left\_join(y = outlier\_score, by = "id") %>%

mutate(

score = coalesce(score, 0),

is\_out = if\_else(score > 0.2, "Outlier", "Not outlier")

)

# Total number of outliers

sum(diam\_tbl$score > 0.2)

## [1] 161

Tibble diam\_tbl is basically the diamonds but with three more columns: id for row number, score for combined outlier score and is\_out for non-outlier/outlier tag.

Plots illustrating strong outliers:

theme\_set(theme\_bw())

plot\_outliers <- function(tbl, x, y, facet\_var) {

tbl %>%

arrange(is\_out) %>%

ggplot(aes\_string(x, y, colour = "is\_out")) +

geom\_point() +

facet\_wrap(facets = facet\_var) +

scale\_colour\_manual(values = c("#AAAAAA", "#004080")) +

guides(colour = guide\_legend(title = NULL,

override.aes = list(size = 4))) +

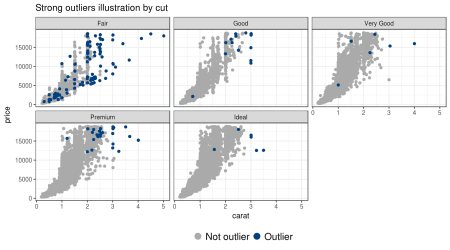
labs(title = paste0("Strong outliers illustration by ", facet\_var)) +

theme(legend.position = "bottom",

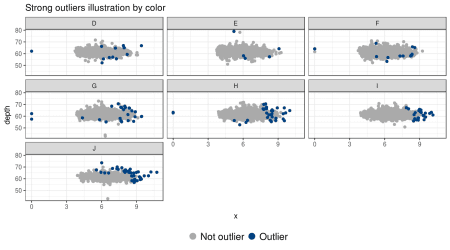
legend.text = element\_text(size = 14))

}

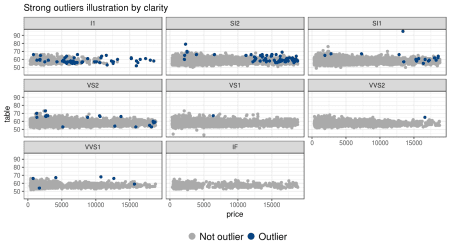
diam\_tbl %>% plot\_outliers("carat", "price", facet\_var = "cut")



diam\_tbl %>% plot\_outliers("x", "depth", facet\_var = "color")



diam\_tbl %>% plot\_outliers("price", "table", facet\_var = "clarity")



Based on those plots we see the complicated nature of “strong outliers”. They are not necessary located “on the edge” of two-dimensional scatter plots, but most extreme cases are tagged as outliers.

Also one interesting observation: most outliers are concentrated in the combination of “Fair” cut, “J” colour and “I1” clarity which are worst options among their features. The reason of this effect is group-based definitions of non-outliers which tagged certain groups more than others:

breaker\_report %>%

filter(pack == "group") %>%

count(var, sort = TRUE) %>%

print(n = 10)

## # A tibble: 47 x 2

## var n

##

## 1 Fair\_D\_I1 7

## 2 Fair\_J\_I1 7

## 3 Fair\_H\_VVS1 6

## 4 Ideal\_J\_I1 6

## 5 Fair\_J\_VVS1 5

## 6 Fair\_G\_VVS1 4

## 7 Fair\_D\_VVS1 3

## 8 Fair\_E\_I1 3

## 9 Fair\_F\_I1 3

## 10 Fair\_H\_I1 3

## # ... with 37 more rows

Here we see that “Fair” cut is among majority of top breaker groups. There are also some interesting combinations: Fair\_D\_I1 (“worst”-“best”-“worst”), Fair\_J\_I1 (“worst”-“worst”-“worst”), Ideal\_J\_I1 (“best”-“worst”-“worst”).

This fact might be interpreted as **suggested combined outlier detection approach discovered notion of diamond quality without prior knowledge about it**.

**Conclusions**

* Using only basic outlier detection methods one can achieve insightful results by combining them. Observations which are tagged as outlier by more than some threshold number of methods might be named as “strong outliers”. Those should be considered as outliers based on the whole data rather then on separate features.
* With ruler combining results of several outlier detection methods is straightforward due to the format of tidy data validation report.
* Suggested “strong outlier” observations in diamonds dataset are not only those with extreme numerical values but also ones based on quality of diamonds. This is achieved without prior knowledge of “diamond quality” notion.

sessionInfo()

## R version 3.4.3 (2017-11-30)

## Platform: x86\_64-pc-linux-gnu (64-bit)

## Running under: Ubuntu 16.04.3 LTS

##

## Matrix products: default

## BLAS: /usr/lib/openblas-base/libblas.so.3

## LAPACK: /usr/lib/libopenblasp-r0.2.18.so

##

## locale:

## [1] LC\_CTYPE=ru\_UA.UTF-8 LC\_NUMERIC=C

## [3] LC\_TIME=ru\_UA.UTF-8 LC\_COLLATE=ru\_UA.UTF-8

## [5] LC\_MONETARY=ru\_UA.UTF-8 LC\_MESSAGES=ru\_UA.UTF-8

## [7] LC\_PAPER=ru\_UA.UTF-8 LC\_NAME=C

## [9] LC\_ADDRESS=C LC\_TELEPHONE=C

## [11] LC\_MEASUREMENT=ru\_UA.UTF-8 LC\_IDENTIFICATION=C

##

## attached base packages:

## [1] methods stats graphics grDevices utils datasets base

##

## other attached packages:

## [1] bindrcpp\_0.2 ruler\_0.1.0 ggplot2\_2.2.1 tidyr\_0.7.2 dplyr\_0.7.4

##

## loaded via a namespace (and not attached):

## [1] Rcpp\_0.12.14 knitr\_1.17 bindr\_0.1 magrittr\_1.5

## [5] tidyselect\_0.2.3 munsell\_0.4.3 colorspace\_1.3-2 R6\_2.2.2

## [9] rlang\_0.1.4 plyr\_1.8.4 stringr\_1.2.0 tools\_3.4.3

## [13] grid\_3.4.3 gtable\_0.2.0 htmltools\_0.3.6 lazyeval\_0.2.1

## [17] yaml\_2.1.16 rprojroot\_1.2 digest\_0.6.13 assertthat\_0.2.0

## [21] tibble\_1.3.4 bookdown\_0.5 purrr\_0.2.4 glue\_1.2.0

## [25] evaluate\_0.10.1 rmarkdown\_1.8 blogdown\_0.4 labeling\_0.3

## [29] stringi\_1.1.6 keyholder\_0.1.1 compiler\_3.4.3 scales\_0.5.0

## [33] backports\_1.1.2 pkgconfig\_2.0.1