```
opts_knit$set(concordance = TRUE)
```

Visualization of large multivariate datasets with the `tabplot` package

Martijn Tennekes and Edwin de Jonge

December 7, 2012
(A later version may be available on CRAN)

Abstract

The tableplot is a powerful visualization method to explore and analyse large multivariate datasets. In this vignette, the implementation of tableplots in R is described, and illustrated with the diamonds dataset from the `ggplot2` package.

Contents

1 Introduction

The tableplot is a visualization method that is used to explore and analyse large datasets. Tableplots are used to explore the relationships between the variables, to discover strange data patterns, and to check the occurrence and selectivity of missing values.

A tableplot applied to the diamonds dataset of the `ggplot2` package (where some missing values were added) is illustrated in Figure 1. Each column represents a variable. The whole data set is sorted according to one column (in this case, carat), and then grouped into row bins. Algorithm 1 in Appendix A describes the creation of a tableplot into detail.

Tableplots are aimed to visualize multivariate datasets with several variabels (up tot a dozen) and a large number of records, say at least one thousand. Tableplots can also be generated for datasets with less records, but they may be less useful. The maximum number of rows that can be visualized with the `tabplot` package depends on the R's memory, or, when using the `ff` package, on the limitations of that package.

2 Getting started with the `tableplot` function

The diamonds dataset is very suitable to demonstrate the `tabplot` package. To illustrate the visualization of missing values, we add several `NA`'s.

```
require(ggplot2)
data(diamonds)
## add some NA's
is.na(diamonds$price) <- diamonds$cut == "Ideal"
is.na(diamonds$cut) <- (runif(nrow(diamonds)) > 0.8)
```

A tableplot is simply created by the function `tableplot`. The result is depicted in Figure 1. By default, all variables of the dataset are depicted. With the argument `select`, we can specify which variables are plotted. The dataset is by default sorted according to the values of the first column. With the argument `sortCol`, we can specify on which column(s) the data is sorted.

The resulting tableplot in Figure 2 consists of five columns, where the data is sorted on price. Notice that the missing values that we have added are placed at the bottom and (by default) shown in a bright red color.

Setting an appropriate number of row bins (with the argument `nBins`) is important, like in a histogram. A good number of row bins is a trade of between good polished but meaningless data, and detailed, but noisy data. In practice, we found that the default number of 100 usually is a good starting point.

The percentages near the vertical axis indicate which subset of the data in terms of units (rows) is depicted. The range from 0% to 100% in Figure 2

```
tableplot(diamonds)
```

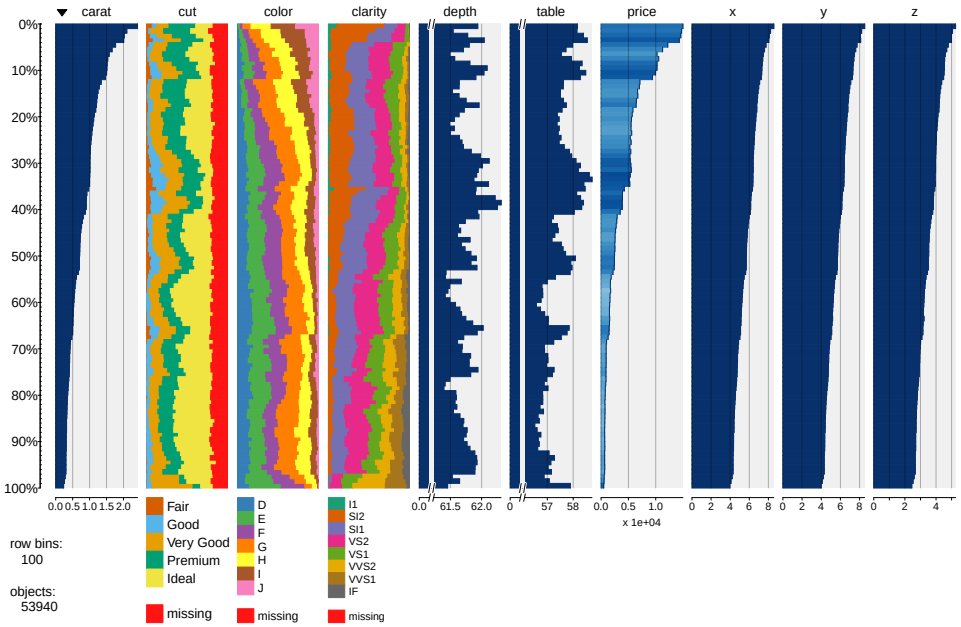

Figure 1: Tableplot of the diamonds dataset

```
tableplot(diamonds, select = c(carat, price,
    cut, color, clarity), sortCol = price)
```

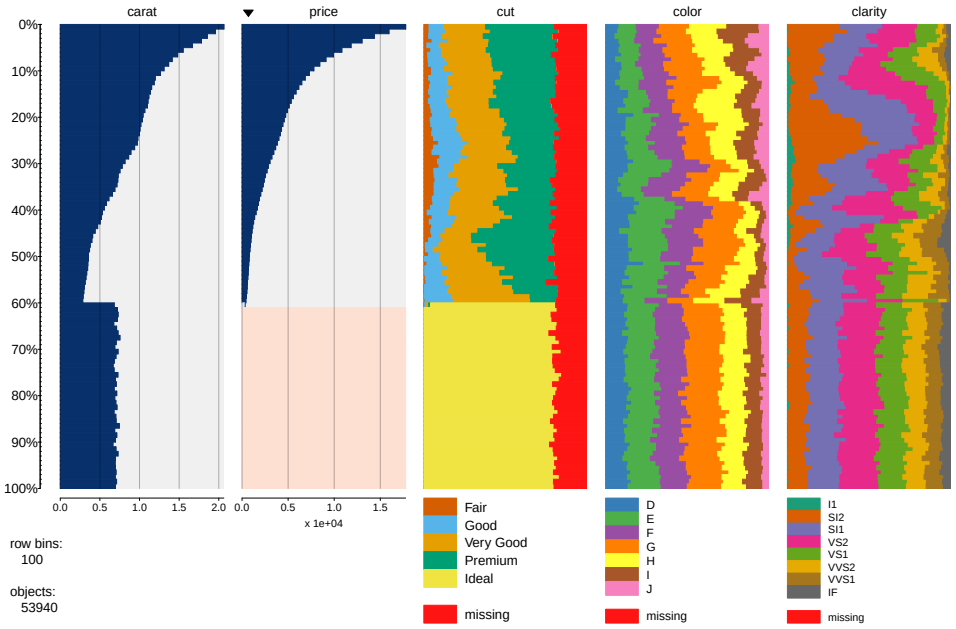

Figure 2: Tableplot sorted by price

means that all units of the data are plotted.

3 Zooming and filtering

3.1 Zooming

We can focus our attention to the 5% most expensive diamonds by setting the `from` argument to 0 and the `to` argument to 5. The resulting tableplot are depicted in Figure 3. Observe that the number of row bins is still 100, so that the number of units per row bin is now 27 instead of 540. Therefore, much more detail can be observed in this tableplot.

```
tableplot(diamonds, select = c(carat, price,
    cut, color, clarity), sortCol = price,
    from = 0, to = 5)
```

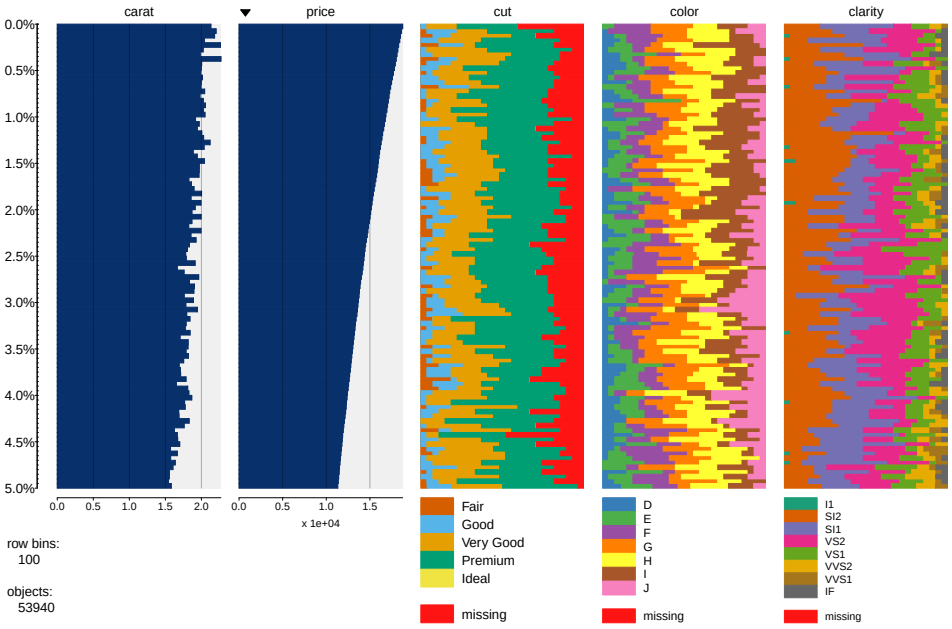

Figure 3: Zooming in

The vertical axis contains two sets of tick marks. The small tick marks correspond with the row bins and the large tick marks correspond with the percentages between `from` and `to`.

3.2 Filtering

The argument `subset` serves as a data filter. The tableplot in Figure 4 shows that data of premium cut diamonds that cost less than 5000$.

```
tableplot(diamonds, subset = price < 5000 & cut == "Premium")
```

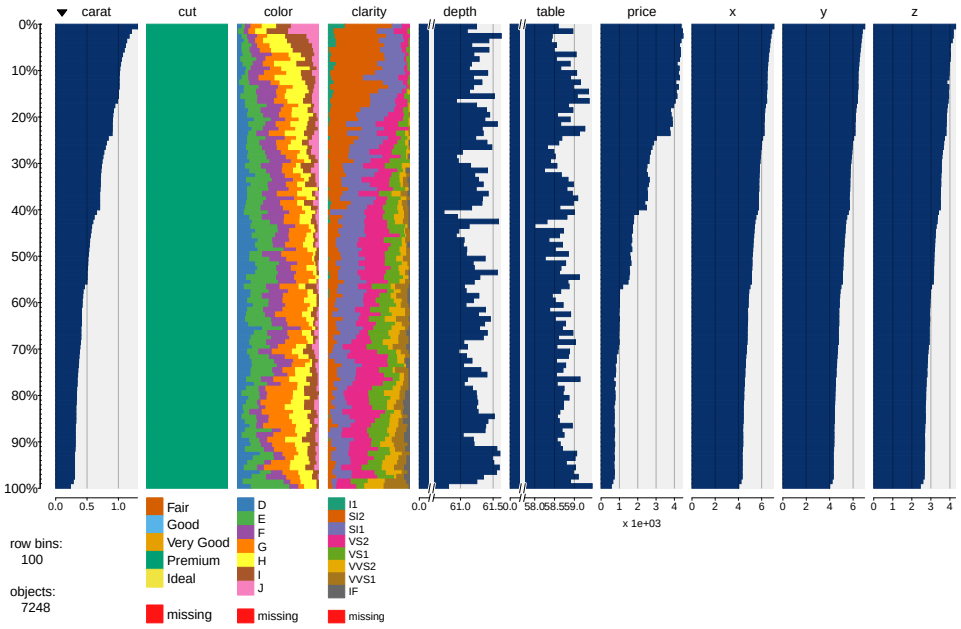

Figure 4: Tableplot of filtered diamonds data

It is also possible to create a tableplot for each category of a categorical variable in one call. For instance, by setting `subset=color` we create a tableplot for each color class.

4 Continuous variables

4.1 Scaling

For each bin of a continuous variable, the mean value is calculated (see Algorithm 1). When the distribution of these mean values is exponential, it is useful to apply a logarithmic transformation. The argument `scales` can be set to linear mode `"lin"`, logarithmic mode `"log"`, or the default value `"auto"`, which automatically determines which of the former two modes is used.

4.2 Used colors

The colors of the bins indicate the fraction of missing values. By default, a sequential color palette of blues is used. If a bin does not contain any missing values, the corresponding bar is depicted in dark blue. The more missing values, the brighter the color. (Alternatively, other quantitative palettes can be used by setting the argument `numPals`; see Figure 5.) Bars of which all values are missing are depicted in light red.

```
tablePalettes()
```

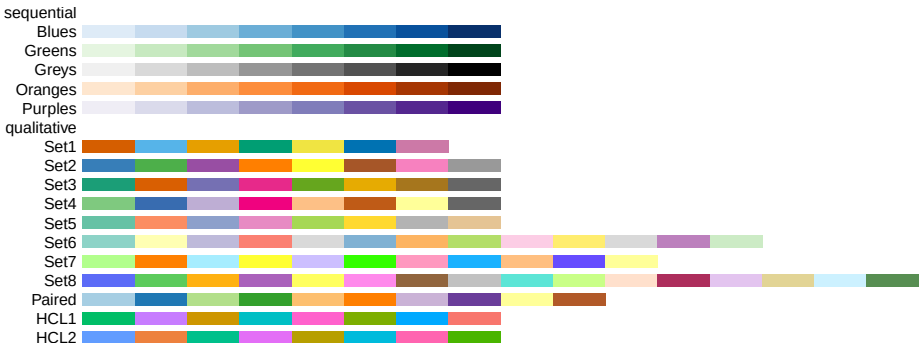

Figure 5: Color palettes

4.3 X-axes

The x-axes a plotted as compact as possible. This is illustrated in the x-axis for the variable price.

Observe that the x-axes of the variables depth and table in Figure 1 are broken. In this way the bars are easier to differentiate. The argument `bias_brokenX` can be set to determine when a broken x-axis is applied. See Appendix B for details.

For each numerical variable, the limits of the x-axes can be determined manually with the argument `limitsX`.

5 Categorical variables

5.1 Color palettes

The implemented palettes are depicted in Figure 5. These palettes, as well as own palettes, can be assigned to the categorical variables with the argument `pals`.

Suppose we want a to use the default palette for the variable `cut`, but starting with the sixth color, blue. Further we want the fifth palette for the variable `color`, and a custom palette, say a rainbow palette, for the variable `clarity`. The resulting tableplot is depicted in Figure 6.

```
tableplot(diamonds, pals = list(cut = "Set1(6)",
    color = "Set5", clarity = rainbow(8)))
```

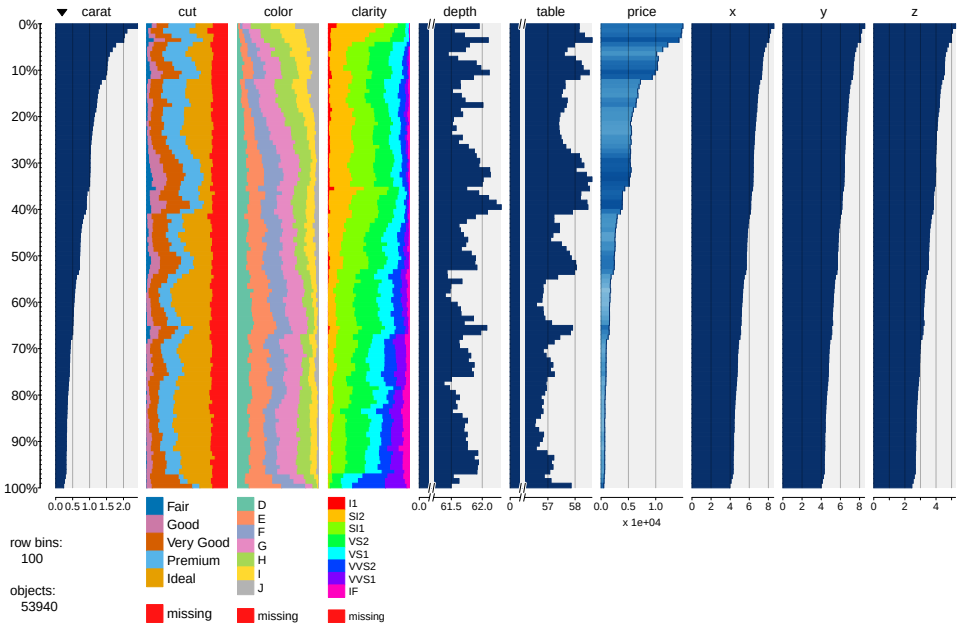

Figure 6: Tableplot with other color palettes

Also quantitative palettes can be used (for instance by setting `clarity="Greens"`. Missing values are by default depicted in red. This can be changed with the argument `colorNA`.

5.2 High cardinality data

To illustrate how tableplots deal with high cardinality data, we extend the diamonds dataset:

```
diamonds$carat_class <- cut(diamonds$carat,
    breaks = pretty(diamonds$carat, n = 20))
diamonds$price_class <- cut(diamonds$price,
    breaks = pretty(diamonds$price, n = 100))
```

For variables with over `change_palette_type_at` (by default 20) categories, color palettes are constructed by using interpolated colors. This creates a rainbow effect (see Figure 7). If the number of categories is than `change_palette_type_at`, the assigned palette is recycled in order to obtain the number of categories.

```
tableplot(diamonds, select = c(carat, price,
    carat_class, price_class))
```

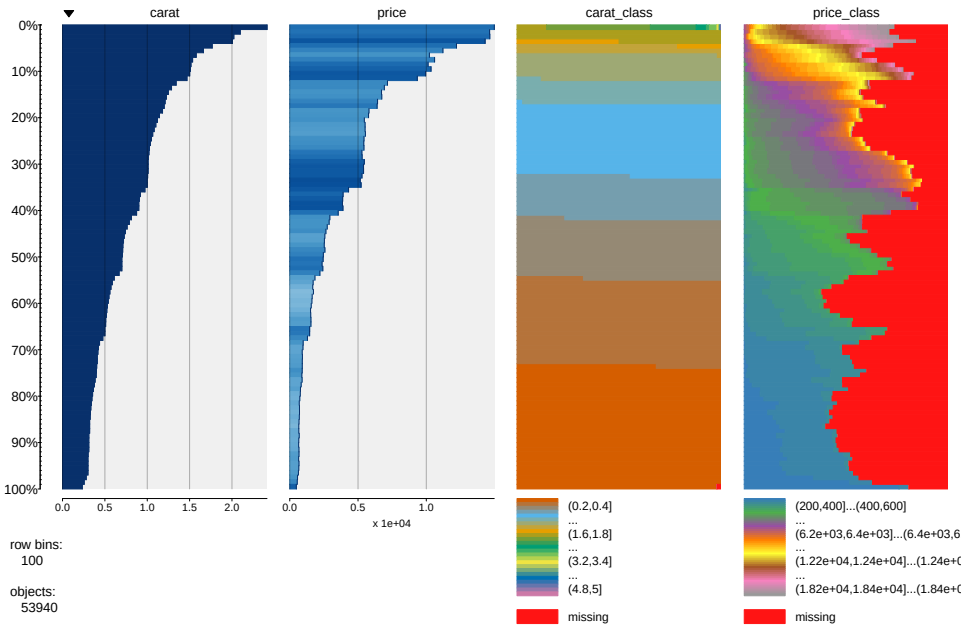

Figure 7: Tableplot with other color palettes

If the number of categories exceeds `max_level` (by default 50), the categories are rebinned into `max_level` category groups. This is illustrated by the variable price_class in Figure 7.

6 Layout options

There are several arguments that determine the layout of the plot: `fontsize`, `legend.lines`, `title`, `showTitle`, and `fontsize.title`. An example of the use of these arguments is given in Figure 8.

```
tableplot(diamonds, select = 1:7, fontsize = 14,
    legend.lines = 8, title = "Shine on you crazy Diamond",
    fontsize.title = 18)
```

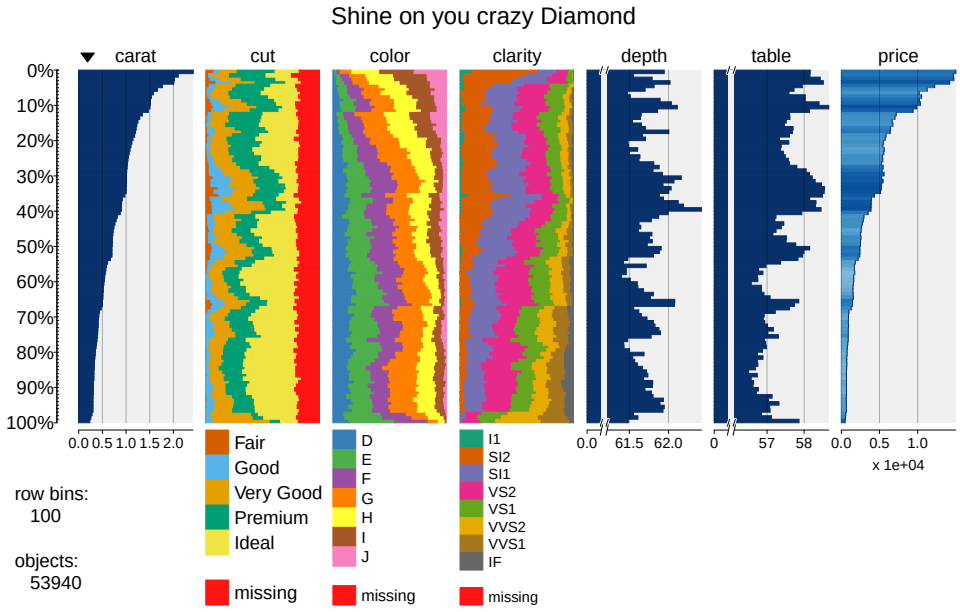

Figure 8: Tableplot with other color palettes

These layout arguments are not handled by the `tableplot` function directly, but they are on to `plot.tabplot`. This function plots a `tabplot` object, which is explained in Section 8.1. The layout arguments will be especially important when saving a tableplot (see Section 8.4).

7 Preprocessing of big data

8 Advanced tableplot features

8.1 The tabplot object

The function `tableplot` returns a tabplot-object, that can be used to make minor changes to the tableplot, for instance the order of columns or the color palettes. Of course, these changes can also be made by generating a new tableplot. However, if it takes considerable time to generate a tableplot, then it is practical to make minor changes immediately.

The output of the `tableplot` function can be assigned to a variable. The graphical output can be omitted by setting the argument `plot` to `FALSE`.

```
tab <- tableplot(diamonds, plot = FALSE)
```

The tabplot-object is a list that contains all information to depict a tableplot. The generic functions `summary` and `plot` can be applied to the `tabplot` object.

```
summary(tab)
```

```
##       general                variable1             variable2
##  dataset  :diamonds   name       :carat     name      :cut
##  variables:12         type       :numeric   type      :categorical
##  objects  :53940      sort       :TRUE      sort      :NA
##  bins     :100        scale_init :auto      categories:6
##  from     :0%         scale_final:lin
##  to       :100%
##       variable3                variable4                 variable5
##  name      :color         name      :clarity       name       :depth
##  type      :categorical   type      :categorical   type       :numeric
##  sort      :NA            sort      :NA            sort       :NA
##  categories:8             categories:9             scale_init :auto
##                                                    scale_final:lin
##
##       variable6              variable7             variable8
##  name       :table     name       :price     name       :x
##  type       :numeric   type       :numeric   type       :numeric
##  sort       :NA        sort       :NA        sort       :NA
##  scale_init :auto      scale_init :auto      scale_init :auto
##  scale_final:lin       scale_final:lin       scale_final:lin
##
##       variable9              variable10           variable11
##  name       :y         name       :z         name      :carat_class
##  type       :numeric   type       :numeric   type      :categorical
##  sort       :NA        sort       :NA        sort      :NA
##  scale_init :auto      scale_init :auto      categories:26
##  scale_final:lin       scale_final:lin
##
##       variable12
##  name      :price_class
##  type      :categorical
##  sort      :NA
##  categories:51
##
##
```

```
plot(tab)
```

8.2 Multiple tableplots

When a dataset contains more variables than can be plotted, multiple tableplots can be generated with the argument `nCols`. This argument determines the maximum number of columns per tableplot. When the number of selected columns is larger than `nCols`, multiple tableplots are generated. In each of them, the sorted columns are plotted on the lefthand side.

When multiple tableplots are created, the (silent) output is a list of `tabplot` objects. This is also the case when the dataset is filtered by a categorical variable, e. g. `subset = color` (see Section 3.2).

8.3 Minor changes

The function `tableChange` is used to make minor changes to a tabplot-object. Suppose we want the columns in the order of 1, and we want to change all color palettes to default starting with the second color. The code and the resulting tableplot are given in Figure 9.

```
tab2 <- tableChange(tab, select_string = c("carat",
    "price", "cut", "color", "clarity"),
    pals = list("Set1(2)"))
plot(tab2)
```

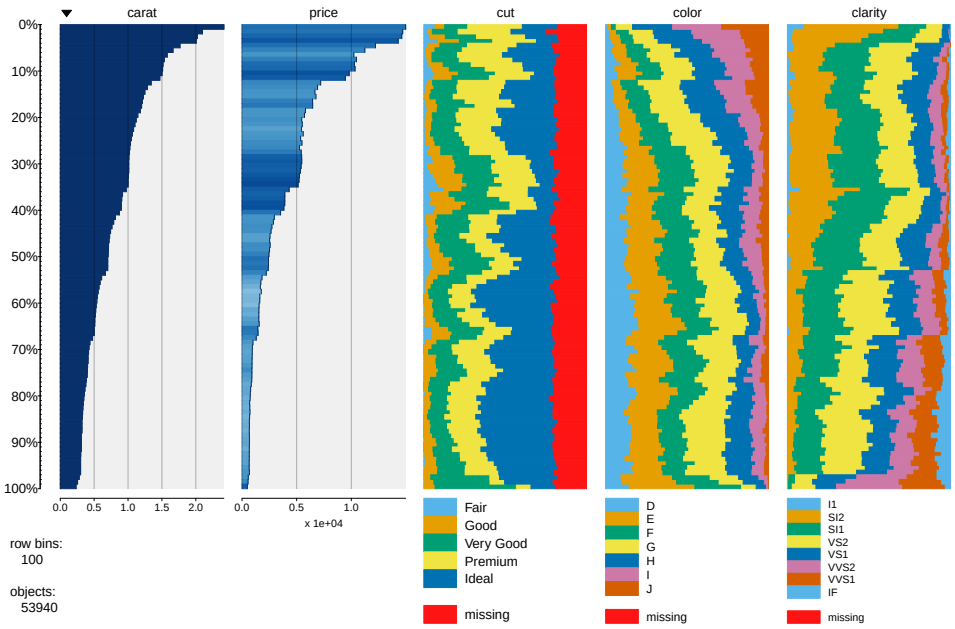

Figure 9: Plot of a `tabplot` object

8.4 Save tableplots

With the function `tableSave`, tableplots can be saved to a desired grahical output format: pdf, eps, svg, wmf, png, jpg, bmp, or tiff.

```
tableSave(tab, filename = "diamonds.png",
    width = 5, height = 3, fontsize = 6,
    legend.lines = 6)
```

Resources

- Summary of the package: `help(package=tabplot)`
- The main help page: `?tabplot`
- Project site: http://code.google.com/p/tableplot/
- References:
  - Tennekes, M., Jonge, E. de, Daas, P.J.H. (2011) Visual profiling of large statistical datasets. Paper presented at the 2011 New Techniques and Technologies for Statistics conference, Brussels, Belgium. (paper, presentation)

A Tableplot creation algorithm

A tabplot is basically created by Algorithm 1.

Algorithm 1 Create tableplot

Input: Tabular dataset t, column i_s of which the distribution is of interest[a], number of row bins n.

$t' \leftarrow$ sort t according to the values of column i_s.
Divide t' into n equally sized row bins according to the order of t'.
for each column i **do**
    if i is numeric **then**
        $m_{ib} \leftarrow$ mean value per bin b
        $c_{ib} \leftarrow$ fraction of missing values per bin b
    end if
    if i is categorical **then**
        $f_{ijb} \leftarrow$ frequency of each category j (including missing values) per bin b
    end if
end for
for each column i **do**
    if i is numeric **then**
        Plot a bar chart of the mean values $\{m_{i1}, m_{i2}, \ldots, m_{in}\}$, optionally with a logarithmic scale. The fraction of missing values $\{c_{i1}, c_{i2}, \ldots, c_{in}\}$ determines the lightness of the bar colour. The lighter the colour, the more missing values occur in bin b. If all values are missing, a light red bar of full length is drawn.
    end if
    if i is categorical **then**
        Plot a stacked bar chart according to the frequencies $\{f_{i1b}, f_{i2b}, \ldots\}$ for each bin b. Each category is shown is a distinct colour. If there are missing values, they are depicted by a red colour.
    end if
end for

Output: Tableplot

[a] The dataset t can also be sorted according to multiple columns.

B Broken x-axes

The x-axis of a variable i is broken if either

$$0 < max(m_{i1}, m_{i2}, \ldots, m_{in}) \qquad \text{AND}$$
$$\texttt{bias\_brokenX} \cdot max(m_{i1}, m_{i2}, \ldots, m_{in}) < min(m_{i1}, m_{i2}, \ldots, m_{in})$$

OR

$$0 > min(m_{i1}, m_{i2}, \ldots, m_{in}) \qquad \text{AND}$$
$$\texttt{bias\_brokenX} \cdot min(m_{i1}, m_{i2}, \ldots, m_{in}) > max(m_{i1}, m_{i2}, \ldots, m_{in}),$$

where `bias_brokenX` is a bias parameter that should be a number between 0 and 1. If `bias_brokenX = 1` then the above conditions are always false, which implies that the x-axes are never broken. On the other hand, if `bias_brokenX = 0` then the x-axes are always broken. By default, `bias_brokenX = 0.8`, which mean that an x-axis is broken if (in case of a variable with positive values) the minimum value is at least 0.8 times the maximum value. In Figure 1, this applies to the variables depth and table.