Help document, Vote

Lukas Arnroth

20 juni 2017

In this help documentation I will provide you with code for how read in one of the data files (code is the same for both with the difference of file name) along with covering some aspects of the factor type variable. Below you find the code for importing the data. The last argument of the read.csv call is specifying how NA has been written down in the data.

```
dat <- read.csv("vote.csv", na.strings = "No answer")
dat_vote <- read.csv("vote_parties.csv", na.strings = "No answer")
str(dat)
```

```
## 'data.frame':    1927 obs. of  24 variables:
##  $ a1 : Factor w/ 9 levels "0,5 hour to 1 hour",..: 7 4 4 5 4 1 4 3 3 5 ...
##  $ a5 : Factor w/ 9 levels "0,5 hour to 1 hour",..: 1 3 4 3 1 1 1 3 3 5 ...
##  $ b1 : Factor w/ 5 levels "Don't know","Hardly interested",..: 2 4 4 4 4 5 4 2 2 5 ...
##  $ b11: Factor w/ 3 levels "No","Not eligible to vote",..: 1 3 3 3 3 3 3 3 2 3 ...
##  $ b12: Factor w/ 12 levels "CENTRE PARTY",..: 9 11 1 12 5 1 3 3 9 12 ...
##  $ c1 : Factor w/ 13 levels "1","2","3","4",..: 7 8 8 9 8 8 9 8 9 7 ...
##  $ c15: Factor w/ 6 levels "Bad","Don't know",..: 3 4 6 3 6 6 4 6 6 1 ...
##  $ c28: Factor w/ 2 levels "No","Yes": 1 2 2 2 2 2 2 2 2 2 ...
##  $ c33: Factor w/ 3 levels "Don't know","No",..: 2 3 3 3 3 3 3 3 3 3 ...
##  $ c35: Factor w/ 3 levels "Don't know","No",..: 2 3 3 3 2 3 3 3 3 3 ...
##  $ c5 : Factor w/ 3 levels "Don't know","No",..: 2 3 2 2 3 2 2 2 2 3 ...
##  $ d4 : Factor w/ 2 levels "No","Yes": 2 2 2 2 1 2 2 1 1 2 ...
##  $ d6 : Factor w/ 2 levels "No","Yes": 1 2 2 2 1 2 2 1 1 2 ...
##  $ d9 : Factor w/ 10 levels "1.000000","12.00000",..: 10 5 3 3 10 4 3 10 10 3 ...
##  $ e4 : Factor w/ 6 levels "Agree","Agree strongly",..: 1 1 1 1 2 1 1 1 1 1 ...
##  $ e5 : Factor w/ 6 levels "Agree","Agree strongly",..: 1 1 1 6 1 1 1 1 1 1 ...
##  $ e7 : Factor w/ 6 levels "Agree","Agree strongly",..: 6 3 1 6 1 1 1 1 1 1 ...
##  $ f1 : num  2 3 2 2 1 3 4 2 6 3 ...
##  $ f2 : Factor w/ 2 levels "Female","Male": 1 2 2 1 2 2 1 1 1 2 ...
##  $ f27: Factor w/ 3 levels "Don't know","No",..: 2 3 2 2 3 2 2 3 2 3 ...
##  $ f3 : num  1980 1949 1936 1949 1984 ...
##  $ f30: Factor w/ 4 levels "Don't know","No",..: 3 4 4 3 2 3 3 2 2 3 ...
##  $ f32: Factor w/ 14 levels "C","D","Don't know",..: 12 4 10 7 8 2 9 2 10 2 ...
##  $ f5 : Factor w/ 6 levels "A big city","Country village",..: 1 6 6 2 2 2 1 4 6 6 ...
```

Note that you will in large work with factors throughout this assignment. Below you find some brief examples of how they work, starting with a made up example and then some of the variables in the actual data.

Working with factors in R

```
var <- c("yes", "no", "maybe", "no", "yes", "maybe") # create a var
typeof(var) # this is a character type
```

```
## [1] "character"
```

Above we have created a character type vector of elements. This variable is typically something you would want to use as a factor, as this is exactly what it is. When converting between different variable types you use the function as.<variable type>() as below.

```
(var <- as.factor(var))
```

```
## [1] yes   no    maybe no    yes   maybe
## Levels: maybe no yes
```

In the ouput you find a second row, apart from the observed values, which is levels. This is the unique categories of your factor. You can look at the levels of a factor using the leves() function

```
levels(var)
```

```
## [1] "maybe" "no"    "yes"
```

When converting to and from factors in R, the levels is the key component to understand. Say that I wanted to store the *var* object as integers, with $0 = no$, $1 = yes$ and $2 = maybe$. What happens if I use $as.integer()$?

```
(var_integer <- as.integer(var))
```

```
## [1] 3 2 1 2 3 1
```

Here we have values but not as how I wanted them to appear. Looking at the levels output we get a hint of what has happened. R has coded the first element of levels as 1 up until the last as 3, meaning that $1 = maybe$, $2 = no$ and $3 = yes$. To get this conversion right, I need to change the levels of the factor. This is done as below. Note that here I use the $factor()$ function rather than $as.factor()$

```
# change order of levels
levels(var) # note how you want to reorder these values
```

```
## [1] "maybe" "no"    "yes"
```

```
# this is how I specified the desired order
levels(var)[c(2,3,1)]
```

```
## [1] "no"    "yes"   "maybe"
```

```
var <- factor(var, levels = levels(var)[c(2,3,1)])
var
```

```
## [1] yes   no    maybe no    yes   maybe
## Levels: no yes maybe
```

Now the order is correct and transforming to integer would almost give exactly the same result. Why not exact? Well I specifie that the values should go from 0 to 2, and not 1 to 3.

```
(var <- as.integer(var))
```

```
## [1] 2 1 3 1 2 3
```

```
# get it correct
(var <- var - 1)
```

```
## [1] 1 0 2 0 1 2
```

In the result we had $1 = no$, $2 = yes$ and $3 = maybe$, so I simply subtract 1 from the integer and we're done.

All of these problems could've been avoided however. If I manually specify the levels when I first created the factor variable we could've skipped all this.

```
var <- c("yes", "no", "maybe", "no", "yes", "maybe") # create a var
var <- factor(var, levels = c("no", "yes", "maybe"))
var
```

```
## [1] yes   no    maybe no    yes   maybe
## Levels: no yes maybe
```

Before wrapping up this brief introduction you should be familiar with how to rename the levels of the factor. This is done calling the levels of the factor object and assigning that a vector of new level names.

```
var <- c("yes", "no", "maybe", "no", "yes", "maybe") # create a var
var <- factor(var, levels = c("no", "yes", "maybe"))
levels(var) <- c("Nah", "Yup", "Uhh")
var
```

```
## [1] Yup Nah Uhh Nah Yup Uhh
## Levels: Nah Yup Uhh
```

This was a quick runthrough of how factors work. It's far from comprehensive. Lets move on to looking at some of the variables in the actual data set, starting with the variable d9. When looking at variables with many observations I recommend you use the head function which by default displays the first 10 observations (sometimes it will be less however depending on the width of your console and the length of elements). Otherwise your console will get cluttered up.

```
# sett n = 10 so that 10 observations get shown
# won't get shown otherwise because of the length
# of "Not applicable"
head(dat$d9, n = 10)
```

```
##  [1] Not applicable 4.000000       2.000000       2.000000
##  [5] Not applicable 3.000000       2.000000       Not applicable
##  [9] Not applicable 2.000000
## 10 Levels: 1.000000 12.00000 2.000000 3.000000 4.000000 ... Not applicable
```

The problem with this variable is not the type, but the levels. I absolutely don't want the .000000 to be printed to plots and other results. It would become very messy. So let's fix this using the brief tutorial above.

```
# levels by hand
levels(dat$d9) <- c("1", "12", "2", "3", "4", "5",
                    "6", "7", "8", "Not applicable")

head(dat$d9, n = 10)
```

```
##  [1] Not applicable 4              2              2
##  [5] Not applicable 3              2              Not applicable
##  [9] Not applicable 2
## Levels: 1 12 2 3 4 5 6 7 8 Not applicable
```

For a factor that you don't plan to transform later, a weird order is something that seldom comes back to haunt you. But It could. So it's always recommended to keep the right order. 1, 12, 2 ... doesn't make much sense so lets reorder the levels.

```
# I want to reorder the levels of d9 by placing
# 12 in the back before No answer.
levels(dat$d9)[c(1,3:9, 2, 10)]
```

```
##  [1] "1"              "2"              "3"              "4"
##  [5] "5"              "6"              "7"              "8"
##  [9] "12"             "Not applicable"
```

```
dat$d9 <- factor(dat$d9, levels = levels(dat$d9)[c(1,3:9, 2, 10)])

head(dat$d9, n = 10)
```

```
##  [1] Not applicable 4              2              2
##  [5] Not applicable 3              2              Not applicable
##  [9] Not applicable 2
```

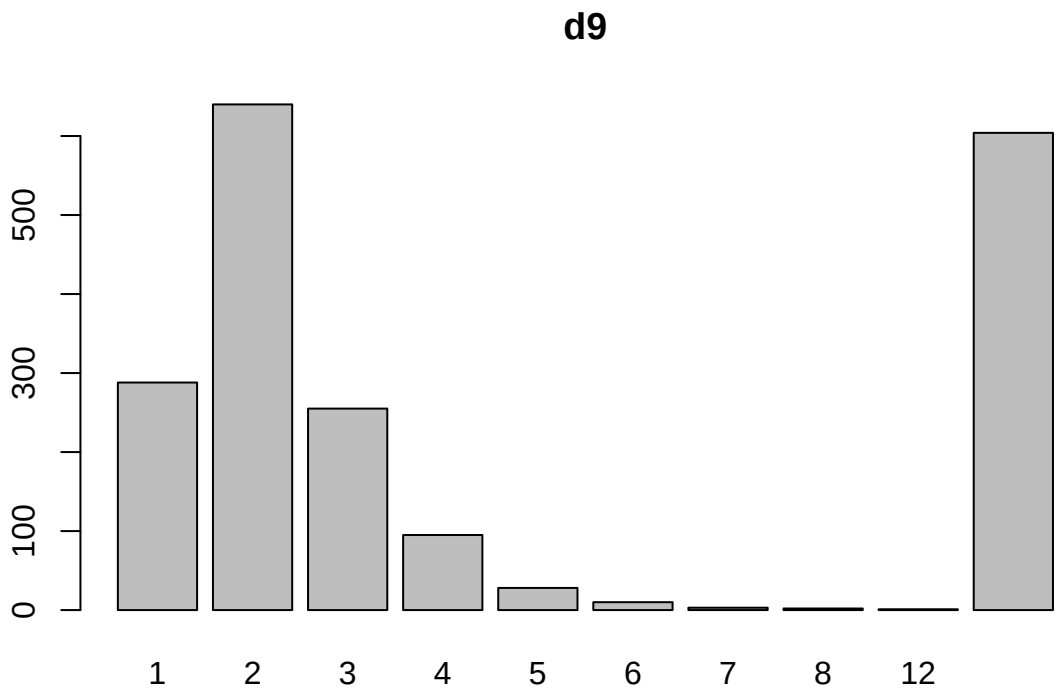
```
## Levels: 1 2 3 4 5 6 7 8 12 Not applicable
```

On a final note, if you have problems reordering the levels of a factor you can always convert it to a character using as.character() first. Then when you convert it back to a factor you can specify the levels. This method is something I sometimes resolve to when things are not working for me!

Factors and plots

Say now that I wan't to display the frequency of each level of d9. I can simply use the plot() function and tell R to plot d9 on the x-axis. The default plot will then be a bar plot. You could plot the variable using a pie chart. Here you have to enclose the vector in table() witch creates a contingency table. However you should almost never choose a pie chart before a bar plot (see ?pie). Fixing the labels so it doesn't become a clutter is quite annoying aswell. Changing the radius and font size as done below does not help either.

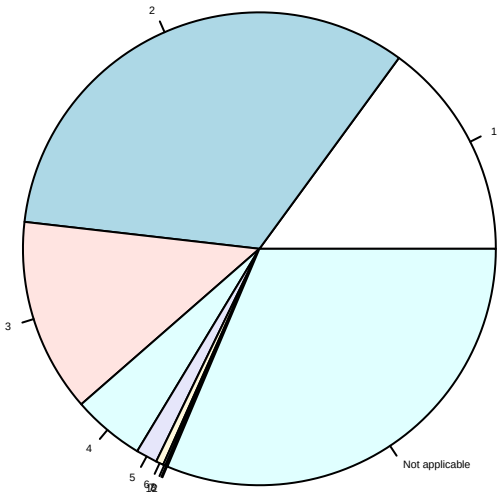
```
# bar plot
plot(dat$d9, main = "d9")
```




```
# pie chart using contigency table
table(dat$d9)
```

```
##
##              1              2              3              4              5
##            288            640            255             95             28
##              6              7              8             12 Not applicable
##             10              3              2              1            604
```

```
pie(table(dat$d9), radius = 1, cex = 0.3)
```

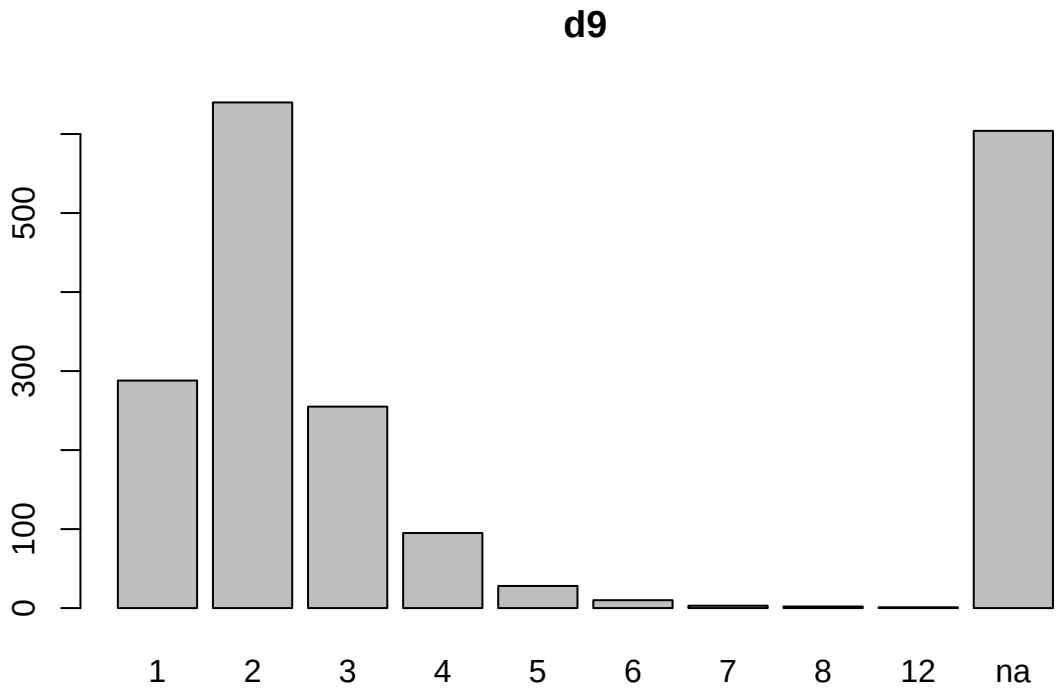


When using the plot() function R decides the most appropriate plot. If you want to specifically tell R to use a bar plot you use the barplot() function and supply it with a contigency table. Also we can use an abbrivation of "Not applicable" so it also will be included in the plot. You would however have to be very clear about what "na" means.

```
# bar plot
# want to change the names aswell on the x-axis aswell so
# "Not applicable" gets left out.
c(levels(dat$d9)[c(1:9)], "na") #1:9 is all levels but not applicable
```

```
##  [1] "1"  "2"  "3"  "4"  "5"  "6"  "7"  "8"  "12" "na"
```

```
barplot(table(dat$d9), main = "d9", names.arg = c(levels(dat$d9)[c(1:9)], "na"))
```



Estimating a logit model in R

Let's say I want to estiamte a logit model for probability of voting on the greens. We will use the glm() function for this, glm standing for generalized linear model. I'll just use variables f2 and f3 as predictors aswell. Lets start by looking at the binary variable and how it was red into R.

```
head(dat_vote$green)
```

```
## [1] 0 0 0 0 1 0
```

```
typeof(dat_vote$green)
```

```
## [1] "integer"
```

The variable has been read as an integer. When estimating the logit model it won't matter whether this is stored as integer or factor.

```
head(dat_vote$green)
```

```
## [1] 0 0 0 0 1 0
```

```
typeof(dat_vote$green)
```

```
## [1] "integer"
```

```
m <- glm(green ~ f3 + f2, data = dat_vote, family = binomial())
summary(m)
```

```
##
```

```
## Call:
## glm(formula = green ~ f3 + f2, family = binomial(), data = dat_vote)
##
## Deviance Residuals:
##     Min       1Q   Median       3Q      Max
## -0.4991  -0.3827  -0.3269  -0.2734   2.6657
##
## Coefficients:
##               Estimate Std. Error z value Pr(>|z|)
## (Intercept) -44.850717  10.949160  -4.096  4.2e-05 ***
## f3            0.021512   0.005573   3.860 0.000113 ***
## f2Male       -0.342028   0.198808  -1.720 0.085360 .
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## (Dispersion parameter for binomial family taken to be 1)
##
##     Null deviance: 848.99  on 1925  degrees of freedom
## Residual deviance: 830.91  on 1923  degrees of freedom
##   (1 observation deleted due to missingness)
## AIC: 836.91
##
## Number of Fisher Scoring iterations: 6
```