

Wrangling categorical data in R

Amelia McNamara¹ and Nicholas J Horton²

¹Statistical and Data Sciences Program, Smith College

²Mathematics and Statistics Department, Amherst College

ABSTRACT

Data wrangling is a critical foundation of data science. Wrangling of categorical data is an important component of the data preparation process. Aspects of these operations can sometimes be tricky, particularly for complex transformations that arise in real-world settings. This paper discusses aspects of categorical variable transformations in R. We consider several motivating examples, suggest defensive coding strategies, and outline principles for data wrangling to help ensure data quality and sound analysis.

Keywords: statistical computing; data derivation; data science; data management

INTRODUCTION

Wrangling skills provide an intellectual and practical foundation for data science. Because of the complexity of some transformations, careless data derivation operations can lead to errors or inconsistencies in analysis. The wrangling of categorical data presents particular challenges and is highly relevant because so many variables are categorical (e.g., gender, income bracket, U.S. state).

It's important that statistical and data science tools foster good practice and provide a robust environment for data wrangling and data management. This paper focuses on how R deals with categorical data, and showcases best practices for categorical data manipulation in R to produce reproducible workflows.

In this paper, we consider a number of common idioms related to categorical data that often arise in data cleaning and preparation, propose some guidelines for defensive coding, and discuss some settings where analysts often get tripped up when working with categorical data. For example, data ingested into R from spreadsheets can lead to problems with categorical data because of the different storage methods possible in both R and the spreadsheets themselves. The examples below will help flag when these issues arise or avoid them altogether.

To ground our work, we will compare and contrast how categorical data are treated in **base R** versus the so-called tidyverse (Wickham, 2014). Tools from the tidyverse, discussed in another paper in this special issue (see <https://github.com/dsscollection/tidyflow>), are designed to make analysis purer, more predictable, and pipeable. They help facilitate a reproducible workflow where a new version of the data could be supplied in the code with updated results produced (Broman, 2015). Wrangling of categorical data can make this task even more complex (e.g., if a new level of a categorical variable is added in an updated dataset or inadvertently introduced by a careless error in a spreadsheet to be ingested into R).

CATEGORICAL DATA IN R- FACTORS AND STRINGS

Consider a variable describing gender that includes categories `male`, `female` and `non-conforming`. In R, there are two ways to store this information. One is to use a series of character strings, and the other is to store it as a factor.

Historically, storing categorical data as a factor variable was more efficient than storing the same data as strings, because factor variables only store the factor labels once (Peng, 2015; Lumley, 2015). However, R uses hashed versions of all character strings, so the storage issue is no longer a consideration (Peng, 2015). For historical reasons, many functions store variables by default as factors.

Factors can be very tricky to deal with, since many operations applied to them return different values than when applied to character vectors. As an example, consider a set of decades,

```
x1 <- c(10, 10, 20, 20, 40)
```

```

x1f <- factor(x1)
ds <- data.frame(x1, x1f)
library(dplyr)
ds <- ds %>%
  mutate(x1recover = as.numeric(x1f))
ds

##    x1 x1f x1recover
## 1 10 10         1
## 2 10 10         1
## 3 20 20         2
## 4 20 20         2
## 5 40 40         3

```

43 Instead of creating a new variable with a numeric version of the value of the factor variable `x1f` the
 44 variable is created with a factor number (i.e., 10 is mapped to 1, 20 is mapped to 2, and 40 is mapped to 3).
 45 This result is unexpected because `base::as.numeric()` is intended to recover numeric information
 46 by coercing a character variable. Compare the following:

```

as.numeric(c("hello"))

## [1] NA

as.numeric(factor(c("hello")))

## [1] 1

```

47 The unfortunate behavior of factors in base R has led to an online movement against the default
 48 behavior of many data import functions to take any variable composed as strings and automatically convert
 49 the variable to a factor. The tidyverse is part of this movement, with functions from the **readr** package de-
 50 faulting to leaving strings as-is. (Others have chosen to add options(`stringAsFactors=FALSE`)
 51 into their startup commands.)

52 Although the storage issues have been solved, and there are problems with defaulting strings to factors,
 53 factors are still necessary for some data analytic tasks. The most salient case is in modeling. When
 54 you pass a factor variable into `lm()` or `glm()`, R automatically creates indicator (or more pejoratively
 55 ‘dummy’) variables for each of the levels and picks one as a reference group. This behavior is lost if the
 56 variable is stored as a character vector. Factor variables also allow for the possibility of ordering between
 57 classes. Text strings `low`, `medium`, `high` would not preserve the ordering inherent in the groups.
 58 Again, this can be important for modeling when doing ordinal logistic regression and multinomial logistic
 59 regression.

60 While factors are important, they can often be hard to deal with. Because of the way the group
 61 numbers are stored separately from the factor labels, it can be easy to overwrite data in such a way that
 62 the original data are lost. In this paper, we will suggest best practices for working with factor data.

63 To motivate this process, we will consider data from the General Social Survey (Smith et al., 2015).
 64 The General Social Survey is a product of the National Data Program for the Social Sciences, and the
 65 survey has been conducted since 1972 by NORC at the University of Chicago. It contains data on many
 66 factors of social life, and is widely used by social scientists. (In this paper we consider data from 2014.)

67 There are some import issues inherent to the data which are not particular to categorical data (see
 68 Appendix A for details). We’ll work with the data that has cleaned variable names.

```

library(dplyr)

```

```
GSS <- read.csv("../data/GSScleaned.csv")
glimpse(GSS)

## Observations: 2,540
## Variables: 17
## $ Year <dbl> 2014, 2014, 2014, 2014, 2014,...
## $ Respondent.id.number <dbl> 1, 2, 3, 4, 5, 6, 7, 8, 9, 10...
## $ Labor.force.status <fctr> Working fulltime, Working fu...
## $ Occupational.prestige.score.1970 <dbl> 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,...
## $ Marital.status <fctr> Divorced, Married, Divorced,...
## $ Number.of.children <dbl> 0, 0, 1, 2, 3, 1, 2, 2, 4, 3,...
## $ Age.of.respondent <fctr> 53.000000, 26.000000, 59.000...
## $ Highest.year.of.school.completed <dbl> 16, 16, 13, 16, 17, 17, 12, 1...
## $ Respondents.sex <fctr> Male, Female, Male, Female, ...
## $ Race.of.respondent <fctr> White, White, White, White, ...
## $ Rs.family.income.when.16.yrs.old <fctr> Below average, Average, Belo...
## $ Total.family.income <fctr> $25000 or more, $25000 or mo...
## $ Respondents.income <fctr> $25000 or more, $25000 or mo...
## $ Total.family.income.1 <fctr> Not applicable, Not applicab...
## $ Political.party.affiliation <fctr> Not str republican, Not str ...
## $ Opinion.of.family.income <fctr> Above average, Above average...
## $ Sexual.orientation <fctr> Heterosexual or straight, He...
```

69 The remainder of this paper is organized around case studies (examples) related to particular tasks:

- 70 1. Changing the labels of factor levels,
- 71 2. Reordering factor levels,
- 72 3. Combining several levels into one (both string-like labels and numeric, probably go together), and
- 73 4. Making derived factor variables.

74 Each case study begins with a problem, and then presents several solutions. Typically, a method using
 75 only **base R** functions is contrasted with an approach from the tidyverse with some annotations of the
 76 code as needed. We will argue that the tidyverse solution is more robust.

77 CHANGING THE LABELS OF FACTOR LEVELS

78 In our first example, we will be considering the labor status variable. It has 9 factor levels. Most of the
 79 labels are spelled out fully, but a few are strangely formatted. We want to change this.

80 This is a specific case of the more general problem of changing the text of one (or more) of the factor
 81 labels, so it appears more nicely formatted in a **ggplot2** plot, for example.

82 There are two typical approaches in **base R**. One is more compact, but depends on the levels of the
 83 factor not changing in the data being fed in, and the other is more robust, but extremely verbose. In
 84 contrast, the **dplyr** package offers a method that is much more human readable, while also supporting
 85 reproducibility.

86 Compact but fragile (base R)

```
levels(GSS$Labor.force.status)

## [1] "Keeping house" "No answer" "Other"
## [4] "Retired" "School" "Temp not working"
## [7] "Unempl, laid off" "Working fulltime" "Working parttime"

summary(GSS$Labor.force.status)

## Keeping house No answer Other Retired
## 263 2 76 460
## School Temp not working Unempl, laid off Working fulltime
## 90 40 104 1230
## Working parttime NA's
## 273 2
```

```

levels(GSS$Labor.force.status) <- c(levels(GSS$Labor.force.status)[1:5],
  "Temporarily not working",
  "Unemployed, laid off",
  "Working full time",
  "Working part time")
summary(GSS$Labor.force.status)

##           Keeping house           No answer           Other
##           263             2             76
##           Retired           School Temporarily not working
##           460             90             40
## Unemployed, laid off Working full time Working part time
##           104           1230           273
##           NA's
##           2

```

87 This method is less than ideal, because it depends on the data coming in with the factor levels ordered
 88 in a particular way. By default, R orders factor levels alphabetically. So, “Keeping house” is first not
 89 because it is the most common response, but simply because ‘k’ comes first in the alphabet. If the data
 90 gets changed outside of R, for example so that responses currently label “Working full time” get labeled
 91 “Full time work”, the code will silently fail with invalid results.

92 The workflow will also fail if additional factor levels are added after the fact. In our experience, both
 93 with students and scientific collaborators, spreadsheet data can be easily changed in these ways (Leek,
 94 2016).

95 Robust but verbose (base R)

96 Another (more robust method) to recode this variable in **base R** is to use subsetting to overwrite particular
 97 values in the data.

```

summary(GSS$Political.party.affiliation)

##           Don't know           Ind,near dem           Ind,near rep
##           1             337             249
##           Independent           No answer           Not str democrat
##           502             25             406
## Not str republican           Other party           Strong democrat
##           292             62             419
## Strong republican           NA's
##           245             2

GSS$NewParty <- as.character(GSS$Political.party.affiliation)
GSS$NewParty[GSS$Political.party.affiliation=="Ind,near dem"] <-
  "Independent, near democrat"
GSS$NewParty[GSS$Political.party.affiliation == "Ind,near rep"] <-
  "Independent, near republican"
GSS$NewParty[GSS$Political.party.affiliation == "Not str democrat"] <-
  "Not strong democrat"
GSS$NewParty <- factor(GSS$NewParty)
summary(GSS$NewParty)

##           Don't know           Independent
##           1             502
## Independent, near democrat Independent, near republican
##           337             249
##           No answer           Not str republican
##           25             292
##           Not strong democrat           Other party
##           406             62
##           Strong democrat           Strong republican
##           419             245
##           NA's
##           2

```

98 This approach is much more robust, because if the labels or ordering of levels changes before this
 99 code is run it will not overwrite labels on the incorrect data. However, this approach has a number of

100 limitations in addition to being tedious and error prone. It is possible to miss cases, and misspelling and
101 cut-and-paste errors can mean that pieces of the code do not actually do anything.

102 Direct and robust (dplyr)

103 The `recode()` function in the **dplyr** package is a vector function, which combines the robustness of the
104 second base R method while also reducing the verbosity. It still suffers from the problem of misspelling
105 and cut-and-paste errors, because it will not throw errors if you try to recode a level that does not exist.

```
GSS <- GSS %>%
  mutate(dplyrParty =
    recode(Political.party.affiliation,
      `Not str republican` = "Not a strong republican",
      `Ind,near dem` = "Independent, near democrat",
      `Ind,near rep` = "Independent, near republican",
      `Not str democrat` = "Not a strong democrat"))
summary(GSS$dplyrParty)

##              Don't know   Independent, near democrat
##                   1                      337
## Independent, near republican      Independent
##                   249                      502
##              No answer      Not a strong democrat
##                   25                      406
##      Not str republican      Other party
##                   292                      62
##      Strong democrat      Strong republican
##                   419                      245
##                   NA's
##                   2
```

106 In the above example, notice the trailing space after “Not str republican” and how the original factor
107 level persists after the recode.

108 Aside – Editing whitespace out of levels

109 Whitespace can be dealt with when data is read, or later using string manipulations. This can be done
110 using the `trimws()` function in **base R**.

```
gender <- factor(c("male ", "male ", "male ", "male "))
levels(gender)

## [1] "male"      "male "    "male "    "male "

gender <- factor(trimws(gender))
levels(gender)

## [1] "male"
```

111 REORDERING FACTOR LEVELS

112 Often, factor levels have a natural ordering to them. However, the default in **base R** is to order levels
113 alphabetically. So, users must have a way to impose order on their factor variables.

114 Again, there is a fragile way to reorder the factor levels in base R, and a more robust method in the
115 tidyverse.

116 Fragile method (base R)

```
summary(GSS$Opinion.of.family.income)
```

```
##      Above average      Average      Below average      Don't know
##      483            1118            666            21
## Far above average Far below average      No answer      NA's
##      65            179            6            2

levels(GSS$Opinion.of.family.income)

## [1] "Above average"      "Average"            "Below average"
## [4] "Don't know"         "Far above average"  "Far below average"
## [7] "No answer"

levels(GSS$Opinion.of.family.income) <-
  levels(GSS$Opinion.of.family.income)[c(5,1:3,6,4,7)]
levels(GSS$Opinion.of.family.income)

## [1] "Far above average" "Above average"      "Average"
## [4] "Below average"    "Far below average"  "Don't know"
## [7] "No answer"
```

117 This is both verbose and depends on the number and order of the levels staying the same. If another
 118 factor level is added to the dataset, the above code will throw an error because the number of levels differs.
 119 This example illustrates why it is sometimes dangerous to replace an old version of a data frame with a
 120 new version.

121 Even worse, if the code gets run more than once, the order will be broken. Particularly when working
 122 dynamically, this is all too easy to do.

```
levels(GSS$Opinion.of.family.income) <-
  levels(GSS$Opinion.of.family.income)[c(5,1:3,6,4,7)]
levels(GSS$Opinion.of.family.income)

## [1] "Far below average" "Far above average" "Above average"
## [4] "Average"          "Don't know"        "Below average"
## [7] "No answer"
```

123 The more times the code is run, the worse it gets.

124 But it gets worse! It is tempting for new analysts to write code such as the following, which completely
 125 ruins the data set.

```
test <- GSS$Opinion.of.family.income
summary(test)

## Far below average Far above average      Above average      Average
##      483            1118            666            21
##      Don't know      Below average      No answer      NA's
##      65            179            6            2

levels(test) <- c("Far above average", "Above average", "Average", "Below Average",
  "Far below average", "Don't know", "No answer")
summary(test)

## Far above average      Above average      Average      Below Average
##      483            1118            666            21
## Far below average      Don't know      No answer      NA's
##      65            179            6            2
```

126 Robust method

127 A new addition to the tidyverse is the package **forcats**, a package for categorical data (and, the name is an
 128 anagram of the word factors!). **forcats** includes a `fct_relevel()` function that does exactly what we
 129 want. It allows us to specify the order of our factor levels (either completely or partially) and is robust to
 130 re-running code in an interactive session.

```
# devtools::install_github("hadley/forcats")
```

```
library(forcats)
summary(GSS$Opinion.of.family.income)

## Far below average Far above average Above average Average
## 483 1118 666 21
## Don't know Below average No answer NA's
## 65 179 6 2

GSS <- GSS %>%
  mutate(Opinion.of.family.income =
    fct_relevel(Opinion.of.family.income,
      "Far above average",
      "Above average",
      "Average",
      "Below average",
      "Far below average"))
summary(GSS$Opinion.of.family.income)

## Far above average Above average Average Below average
## 1118 666 21 179
## Far below average Don't know No answer NA's
## 483 65 6 2
```

131 Notice that the levels we did not mention just end up at the back end of the ordering. Running the
132 code again does not break things.

```
GSS <- GSS %>%
  mutate(Opinion.of.family.income =
    fct_relevel(Opinion.of.family.income,
      "Far above average",
      "Above average",
      "Average",
      "Below average",
      "Far below average"))
summary(GSS$Opinion.of.family.income)

## Far above average Above average Average Below average
## 1118 666 21 179
## Far below average Don't know No answer NA's
## 483 65 6 2
```

133 COMBINING SEVERAL LEVELS INTO ONE

134 Combining discrete levels

135 This is another common task. Maybe you want fewer coefficients in your model, or the process that
136 generated the data makes a finer distinction between categories than your research. For whatever the
137 reason, you want to group together levels that are currently separate.

138 *Fragile method (base R)*

139 This method overwrites the labels of factor levels with repeated labels in order to group levels together.

```
levels(GSS$Labor.force.status) <- c("Not employed", "No answer",
  "Other", "Not employed",
  "Not employed", "Not employed",
  "Not employed", "Employed", "Employed")
summary(GSS$Labor.force.status)

## Not employed No answer Other Employed NA's
## 957 2 76 1503 2
```

140 As before, this is fragile because it depends on the order of the factor levels not changing, and on a
141 human accurately counting the indices of all the levels they wish to change.

142 **Robust method**

143 The `recode()` does what we want.

```
levels(GSS$Race.of.respondent)

## [1] "Black" "Other" "White"

GSS <- GSS %>%
  mutate(Race.of.respondent = recode(Race.of.respondent,
    `Black` = "Nonwhite",
    `Other` = "Nonwhite"))
levels(GSS$Race.of.respondent)

## [1] "Nonwhite" "White"
```

144 **Combining numeric-type levels**

145 Combining numeric-type levels is a problem that often arises even when `stringsasfactors=FALSE`.
146 Often variables like age or income are right-censored, so there is a final category containing the lumped
147 remainder of people. This means the data is necessarily at least a character string if not a factor. However,
148 it may be more natural to work with numeric expressions when recoding this data.

149 In this data, age is provided as an integer for respondents 18-88, but then also includes the possible
150 answer “89 or older” as well as a possible “No answer” and NA values.

151 We might want to turn this into a factor variable with two levels: 18-65, and over 65. In this case, it
152 would be much easier to deal with a conditional statement about the numeric values, rather than writing
153 out each of the numbers as a character vector.

154 **Fragile method (base R)**

155 In order to break this data apart as simply as possible, we need to make it numeric. To start, we recode the
156 label for “89 or older” to read “89”. Already, we are doing something fragile.

```
GSS$BaseAge <- GSS$Age.of.respondent
levels(GSS$BaseAge)

## [1] "18.000000" "19.000000" "20.000000" "21.000000" "22.000000"
## [6] "23.000000" "24.000000" "25.000000" "26.000000" "27.000000"
## [11] "28.000000" "29.000000" "30.000000" "31.000000" "32.000000"
## [16] "33.000000" "34.000000" "35.000000" "36.000000" "37.000000"
## [21] "38.000000" "39.000000" "40.000000" "41.000000" "42.000000"
## [26] "43.000000" "44.000000" "45.000000" "46.000000" "47.000000"
## [31] "48.000000" "49.000000" "50.000000" "51.000000" "52.000000"
## [36] "53.000000" "54.000000" "55.000000" "56.000000" "57.000000"
## [41] "58.000000" "59.000000" "60.000000" "61.000000" "62.000000"
## [46] "63.000000" "64.000000" "65.000000" "66.000000" "67.000000"
## [51] "68.000000" "69.000000" "70.000000" "71.000000" "72.000000"
## [56] "73.000000" "74.000000" "75.000000" "76.000000" "77.000000"
## [61] "78.000000" "79.000000" "80.000000" "81.000000" "82.000000"
## [66] "83.000000" "84.000000" "85.000000" "86.000000" "87.000000"
## [71] "88.000000" "89 or older" "No answer"

levels(GSS$BaseAge) <- c(levels(GSS$BaseAge)[1:71], "89", "No answer")
```

157 When we look at the levels, we can see that the first 71 levels correspond to the ages 18-88, and are
158 in the order we would expect, so we are leaving those as-is. Then we are overwriting the data where
159 `BaseAge == "89 or older"` with simply 89. Once that is done, we can convert the factor to a
160 character vector and then to a numeric one.

```
GSS$BaseAge <- as.numeric(as.character(GSS$BaseAge))
summary(GSS$BaseAge)

##      Min. 1st Qu.  Median    Mean 3rd Qu.    Max.    NA's
##      18.00   34.00   49.00   49.01   62.00   89.00     11
```

161 We’re avoiding the pitfall from the introduction here by not just using `as.numeric()` on the factor
162 variables (this would convert 18 to 1, 19 to 2, etc.). And of course, we’re cheating a little bit here– if we

163 were going to use this as a numeric variable in an analysis, we wouldn't necessarily want to turn all the
 164 "89 or older" cases into the number "89". But, we're just on our way to a two-category factor, so those
 165 cases would have gone to the "65 and up" category one way or the other.
 166 Now, we can write some conditional logic

```
splitf <- function(x){
  return(ifelse(x<65, "18-64", "65 and up"))
}
summary(GSS$BaseAge)

##      Min. 1st Qu.  Median    Mean 3rd Qu.    Max.     NA's
##    18.00   34.00   49.00   49.01  62.00   89.00      11

GSS$BaseAge <- sapply(GSS$BaseAge, splitf)
GSS$BaseAge <- factor(GSS$BaseAge)
summary(GSS$BaseAge)

##      18-64 65 and up      NA's
##      2011      518        11
```

167 **Robust method**

168 The **dplyr** method follows similar logic. However, instead of explicitly overwriting 89 or older
 169 with the number 89, we use the **readr** `parse_numeric()` function to remove the numbers from the
 170 factor labels. This works for the labels that already look numeric, like "18.000000" as well as for
 171 "89 or older". Then, we can include the conditional logic for splitting the variable within a mutate
 172 command.

```
library(readr)
GSS <- GSS %>%
  mutate(dplyrAge = parse_numeric(Age.of.respondent)) %>%
  mutate(dplyrAge = if_else(dplyrAge < 65, "18-65", "65 and up"),
         dplyrAge = factor(dplyrAge))
summary(GSS$dplyrAge)

##      18-65 65 and up      NA's
##      2011      518        11
```

173 **CREATING DERIVED CATEGORICAL VARIABLE**

174 Challenges often arise when data scientists need to create derived categorical variables. As an exam-
 175 ple, consider an indicator of moderate drinking status. The National Institutes of Alcohol Abuse and
 176 Alcoholism have published guidelines for moderate drinking National Institute of Alcohol Abuse and
 177 Alcoholism (2016). These guidelines state that women, or men aged 65 or older should drink no more
 178 than one drink per day on average and no more than three drinks on any single day or at a sitting. Men
 179 under age 65 should drink no more than two drinks per day on average and no more than four drinks
 180 on any single day. The `HELPMiss` dataset from the **mosaicData** package includes baseline data from a
 181 randomized clinical trial (Health Evaluation and Linkage to Primary Care). These subjects were recruited
 182 from a detoxification center, hence those that are alcohol-involved have extremely high rates of drinking.

variable	description
sex	gender of subject female or male
i1	average number of drinks per day (in last 30 days)
i2	maximum number of drinks per day (in past 30 days)
age	age (in years)

184 These guidelines can be used to create a new variable called `abstinent` for those that reported no
 185 drinking based on the value of their `i1` variable and `moderate` for those that do not exceed the NIAAA
 186 guidelines, with all other non-missing values coded as `highrisk`.

```
library(dplyr)
library(mosaic)
library(readr)
```

187 Because missing values can become especially problematic in more complex derivations, we will
188 make one value missing so we can ensure our data wrangling accounts for the missing value.

```
data(HELPmiss)
HELPsmall <- HELPmiss %>%
  mutate(i1 = ifelse(id==1, NA, i1)) %>% # make one value missing
  select(sex, i1, i2, age)
head(HELPsmall, 2)

##      sex i1 i2 age
## 1 male NA 26 37
## 2 male 56 62 37
```

189 Fragile method (base R)

```
# create empty repository for new variable
drinkstat <- character(length(HELPsmall$i1))
# create abstinent group
drinkstat[HELPsmall$i1==0] = "abstinent"
# create moderate group
drinkstat[(HELPsmall$i1>0 & HELPsmall$i1<=1 &
  HELPsmall$i2<=3 & HELPsmall$sex=="female") |
  (HELPsmall$i1>0 & HELPsmall$i1<=2 &
  HELPsmall$i2<=4 & HELPsmall$sex=="male")] = "moderate"
# create highrisk group
drinkstat[((HELPsmall$i1>1 | HELPsmall$i2>3) & HELPsmall$sex=="female") |
  ((HELPsmall$i1>2 | HELPsmall$i2>4) & HELPsmall$sex=="male")] = "highrisk"
# account for missing values
is.na(drinkstat) <- is.na(HELPsmall$i1) | is.na(HELPsmall$i2) |
  is.na(HELPsmall$sex)
drinkstat <- factor(drinkstat)
tally(~ drinkstat, exclude=NULL)

## drinkstat
## abstinent highrisk moderate <NA>
##          69       372       28        1
```

190 While this approach works, it is hard to follow or to debug. The logical conditions are all correctly
191 coded, but the missing value was not handled by default (without the `is.na()` call the missing value
192 would default to be "highrisk" because of their extreme value for `i2`).

193 Robust method (dplyr)

```
HELPsmall <- with(HELPsmall, # this won't work with current dplyr
  # unless HELPsmall is made accessible to mutate() through with()
  # Hadley is aware of this issue with case_when()
  mutate(HELPsmall,
    drink_stat = case_when(
      i1 == 0 ~ "abstinent",
      i1 <= 1 & i2 <= 3 & sex=="female" ~ "moderate",
      i1 <= 1 & i2 <= 3 & sex=="male" & age >= 65 ~ "moderate",
      i1 <= 2 & i2 <= 4 & sex=="male" ~ "moderate",
      is.na(i1) ~ "missing", # this can't be NA
      TRUE ~ "highrisk"
    ))
  tally(~ drink_stat, exclude=NULL, data = HELPsmall)

## drink_stat
## abstinent highrisk missing moderate <NA>
##          69       372        1       28        0
```

194 The same logic is used, but the conditions are much clearer and comprehensible. Instead of one big
195 Boolean condition for moderate, three separate lines can be used to match the different options.

DEFENSIVE CODING

It is always good practice to practice defensive coding. For the setting we are considering, this might include adding conditional testing statements into code that creates or modified factors. These testing statements can help ensure that data has not changed from one session to another, or as the result of changes to the raw data.

As an example, we might want to check that there are exactly three levels for the drinking status variable in the HELP dataset. If there were fewer or more than three levels, something would have gone wrong with our code. We can use the **assertthat** package to help with this.

```
library(assertthat)
levels(drinkstat)

## [1] "abstinent" "highrisk" "moderate"

assert_that(length(levels(drinkstat))==3)

## [1] TRUE
```

We also might want to ensure that the factor labels are exactly what we were expecting. Perhaps we want to make sure our Race variable has been collapsed into two categories, with particular levels. We can use `expect_equivalent()` and `expect_equal()` from the **testthat** package to make this check.

```
library(testthat)
str(levels(GSS$Race.of.respondent))

## chr [1:2] "Nonwhite" "White"

str(c("White", "Nonwhite"))

## chr [1:2] "White" "Nonwhite"

str(sort(c("White", "Nonwhite")))

## chr [1:2] "Nonwhite" "White"

#expect_equivalent(levels(GSS$Race.of.respondent),
# c("White", "Nonwhite")) # This doesn't work, but we wish it did.
expect_equivalent(levels(GSS$Race.of.respondent), c("Nonwhite", "White")) # Does work
expect_equal(levels(GSS$Race.of.respondent), c("Nonwhite", "White")) # Does work
expect_equivalent(levels(GSS$Race.of.respondent), sort(c("Nonwhite", "White"))) # Does work
```

CONCLUSION

Categorical variables arise commonly in surveys and observational data. Aspects of data wrangling involving categorical variables can be problematic and error-prone. In this paper we have outlined some example case studies where analytic tasks can be simplified and made more robust through use of new tools available in R. We believe that further work is needed to make it easier to undertake analyses that requires data wrangling and that new tools and an increased emphasis on defensive coding may help improve the quality of data science moving forward.

ACKNOWLEDGEMENTS

Thanks to Hadley Wickham, who read an early version of this paper and helped solve several issues with the **forcats** package.

QUERIES FOR REVIEWERS

1. Is it useful to demonstrate two ways to do each thing (as long as one isn't totally stupid)
2. Do we clarify why each of the tasks are hard?
3. Do we clarify why each of the standard approaches are error-prone?

- 221 4. Should we focus more on missing values? less?
- 222 5. Add appendices or online resources for other examples? Move closing exercise to be online only?
- 223 6. Add other references?

224 APPENDIX A: LOADING THE DATA

225 Since this is a reproducible special issue, we want to make sure our data ingest process is as reproducible
226 as possible. We are using the General Social Survey (GSS) data, which includes many years of data
227 (1972-2014) and many possible variables (150-800 variables, depending on the year) (Smith et al., 2015).
228 However, the GSS data has some idiosyncrasies. So, we are attempting good-enough practices for data
229 ingest (Wilson et al., 2016).

230 The most major issue related to reproducibility is that the data is not available through an API. For
231 SPSS and Stata users, yearly data are available for direct download on the website. For more format
232 possibilities, users must go through an online wizard to select variables and years for the data they wish
233 to download (NORC at the University of Chicago, 2016). For this paper, we selected a subset of the
234 demographic variables and the year 2014. The possible output options from the wizard are Excel (either
235 data and metadata or metadata only), SPSS, SAS, Stata, DDI, or R script. We selected both the Excel and
236 R formats to look at the differences.

237 The R format provided by the GSS is actually a Stata file and custom R script that uses the **foreign**
238 package to do the translation for you. Here is the result of that process.

```
source('../data/GSS.r')
glimpse(GSS)

## Observations: 2,538
## Variables: 17
## $ YEAR      <int> 2014, 2014, 2014, 2014, 2014, 2014, 2014, 2014, 2014, ...
## $ ID_       <int> 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16...
## $ WRKSTAT    <int> 1, 1, 4, 2, 5, 1, 9, 1, 8, 1, 7, 8, 5, 1, 6, 2, 2, 1, ...
## $ PRESTIGE   <int> 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, ...
## $ MARITAL    <int> 3, 1, 3, 1, 1, 1, 1, 1, 5, 1, 1, 5, 3, 1, 5, 1, 3, 5, ...
## $ CHILDS     <int> 0, 0, 1, 2, 3, 1, 2, 2, 2, 4, 3, 2, 0, 5, 2, 0, 3, 3, 0, ...
## $ AGE        <int> 53, 26, 59, 56, 74, 56, 63, 34, 37, 30, 43, 56, 69, 4...
## $ EDUC       <int> 16, 16, 13, 16, 17, 17, 12, 17, 10, 15, 5, 11, 8, 11, ...
## $ SEX        <int> 1, 2, 1, 2, 2, 2, 1, 1, 2, 2, 2, 1, 1, 2, 2, 1, 2, 1, ...
## $ RACE       <int> 1, 1, 1, 1, 1, 1, 1, 1, 1, 3, 3, 1, 1, 1, 1, 1, 3, 1, ...
## $ INCOM16    <int> 2, 3, 2, 2, 4, 4, 2, 3, 3, 1, 1, 2, 2, 2, 2, 3, 2, 3, ...
## $ INCOME     <int> 12, 12, 12, 12, 13, 12, 13, 12, 10, 12, 9, 9, 10, 11, ...
## $ RINCOME    <int> 12, 12, 0, 9, 0, 12, 13, 12, 0, 12, 0, 0, 0, 11, 12, ...
## $ INCOME72   <int> 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, ...
## $ PARTYID    <int> 5, 5, 6, 5, 3, 6, 6, 8, 3, 3, 3, 3, 3, 1, 3, 6, 1, 3, ...
## $ FINRELA    <int> 4, 4, 2, 4, 3, 4, 9, 3, 2, 3, 8, 5, 1, 1, 3, 3, 2, 3, ...
## $ SEXORNT    <int> 3, 3, 3, 3, 3, 9, 0, 0, 3, 3, 3, 3, 3, 0, 3, 3, 0, 0, ...
```

239 Obviously, the result is less than ideal. All of the factor variables are encoded as integers, but their
240 level labels have been lost. We have to look at a codebook to determine if `SEX == 1` indicates male or
241 female. We would rather preserve the integrated level labels. In order to do this, our best option is to use
242 the Excel file and use the **readxl** package to load it.

```
library(readxl)
```

```
GSS <- read_excel("../data/GSS.xls")
glimpse(GSS)

## Observations: 2,540
## Variables: 17
## $ Gss year for this respondent      <dbl> 2014, 2014...
## $ Respondent id number              <dbl> 1, 2, 3, 4...
## $ Labor force status                 <chr> "Working f...
## $ Rs occupational prestige score   (1970) <dbl> 0, 0, 0, 0...
## $ Marital status                    <chr> "Divorced"...
## $ Number of children                <dbl> 0, 0, 1, 2...
## $ Age of respondent                 <chr> "53.000000...
## $ Highest year of school completed <dbl> 16, 16, 13...
## $ Respondents sex                   <chr> "Male", "F...
## $ Race of respondent                <chr> "White", "...
## $ Rs family income when 16 yrs old <chr> "Below ave...
## $ Total family income               <chr> "$25000 or...
## $ Respondents income                <chr> "$25000 or...
## $ Total family income               <chr> "Not appli...
## $ Political party affiliation        <chr> "Not str r...
## $ Opinion of family income          <chr> "Above ave...
## $ Sexual orientation                <chr> "Heterosex..."
```

243 That's a little better. Now we have preserved the character strings. But, the data is not yet usable in an
 244 analysis. One problem is that some of the variable names include spaces, so they are hard to use. Also,
 245 one variable name is repeated, perhaps because of an error in the data wizard. To fix these issues, we need
 246 to rename the variables so all variables have unique names without spaces.

```
names(GSS) <- make.names(names(GSS), unique=TRUE)
names(GSS)

## [1] "Gss.year.for.this.respondent....."
## [2] "Respondent.id.number"
## [3] "Labor.force.status"
## [4] "Rs.occupational.prestige.score...1970."
## [5] "Marital.status"
## [6] "Number.of.children"
## [7] "Age.of.respondent"
## [8] "Highest.year.of.school.completed"
## [9] "Respondents.sex"
## [10] "Race.of.respondent"
## [11] "Rs.family.income.when.16.yrs.old"
## [12] "Total.family.income"
## [13] "Respondents.income"
## [14] "Total.family.income.1"
## [15] "Political.party.affiliation"
## [16] "Opinion.of.family.income"
## [17] "Sexual.orientation"
```

247 These names are an improvement, but now some are full of periods. We'd like to rename the most
 248 extreme cases to make the names more human readable. As with all the tasks in this paper, there is
 249 a fragile way to do this in **base R**, but we'll use the more robust `rename()` function from the **dplyr**
 250 package. `rename()`

```
library(dplyr)
```

```
GSS <- GSS %>%
  rename(Year = Gss.year.for.this.respondent.....,
         Occupational.prestige.score.1970 = Rs.occupational.prestige.score...1970.)
names(GSS)

## [1] "Year" "Respondent.id.number"
## [3] "Labor.force.status" "Occupational.prestige.score.1970"
## [5] "Marital.status" "Number.of.children"
## [7] "Age.of.respondent" "Highest.year.of.school.completed"
## [9] "Respondents.sex" "Race.of.respondent"
## [11] "Rs.family.income.when.16.yrs.old" "Total.family.income"
## [13] "Respondents.income" "Total.family.income.1"
## [15] "Political.party.affiliation" "Opinion.of.family.income"
## [17] "Sexual.orientation"
```

Now that we have the data loaded and the names adjusted, we can write the data to a new file for use in the body of the paper.

```
write_csv(GSS, path="../data/GSScleaned.csv")
```

APPENDIX B: CLOSING EXERCISE

We have included the following as a possible closing exercise.

Subjects in the HELP study were also categorized into categories of drug and alcohol involvement, as displayed in the following table.

```
HELPbase <- HELFull %>%
  filter(TIME==0)
tally( ~ PRIM_SUB + SECD_SUB, data=HELPbase)

##          SECD_SUB
## PRIM_SUB  0  1  2  3  4  5  6  7  8
##      1 99  0 57 13  1  3 11  0  1
##      2 51 84  0  6  0  0 15  0  0
##      3 57 28 29  0  0  6  5  1  2
##      6  0  1  0  0  0  0  0  0  0
```

The following coding of substance use involvement was used in the study.

value	description
0	None
1	Alcohol
2	Cocaine
3	Heroin
4	Barbituates
5	Benzos
6	Marijuana
7	Methadone
8	Opiates

Create a new variable called 'primsub' that combines the primary and secondary substances into a categorical variable with values corresponding to primary and secondary substances of the form: alcohol only, cocaine only, 'heroin only', 'alcohol-cocaine', 'cocaine-alcohol', or 'other'. Code any group with fewer than 5 entries as 'alcohol-other', 'cocaine-other', or 'heroin-other'. If 'PRIM.SUB==6' make the 'primsub' variable missing.

How many subjects are there in the 'alcohol-none' group? How many subjects are there in the 'alcohol-other' group? What are the three most common groups?

SOLUTION:

```
HELPbase <- with(HELPbase,
```

```
mutate(HELPhbase,
  primary= recode(PRIM_SUB,
    `1`="alcohol",
    `2`="cocaine",
    `3`="heroin",
    `4`="barbituates",
    `5`="benzos",
    `6`="marijuana",
    `7`="methadone",
    `8`="opiates"),
  second=recode(SECD_SUB,
    `0`="none",
    `1`="alcohol",
    `2`="cocaine",
    `3`="heroin",
    `4`="barbituates",
    `5`="benzos",
    `6`="marijuana",
    `7`="methadone",
    `8`="opiates"),
  title=paste0(primary, "-", second)
))
```

```
tally(~ primary, data=HELPhbase)

## primary
##   alcohol   cocaine   heroin marijuana
##      185      156      128         1

tally(~ second, data=HELPhbase)

## second
##   alcohol barbituates   benzos   cocaine   heroin   marijuana
##      113         1         9       86      19        31
##   methadone      none   opiates
##         1      207         3

counts <- HELPhbase %>%
  group_by(primary, second) %>%
  summarise(observed=n())

merged <- left_join(HELPhbase, counts, by=c("primary", "second"))
```

```
merged <- with(merged,
```

```
mutate(merged,
  title =
    case_when(
      observed < 5 & primary=="alcohol" ~ "alcohol-other",
      observed < 5 & primary=="cocaine" ~ "cocaine-other",
      observed < 5 & primary=="heroin" ~ "heroin-other",
      TRUE ~ title),
  title = ifelse(primary=="marijuana", NA, title)))
tally(~ title + observed, data=merged)

##              observed
## title
## alcohol-cocaine    1  2  3  5  6 11 13 15 28 29 51 57 84 99
## alcohol-heroin     0  0  0  0  0  0 13  0  0  0  0  0  0  0
## alcohol-marijuana  0  0  0  0  0 11  0  0  0  0  0  0  0  0
## alcohol-none       0  0  0  0  0  0  0  0  0  0  0  0  0 99
## alcohol-other      2  0  3  0  0  0  0  0  0  0  0  0  0  0
## cocaine-alcohol    0  0  0  0  0  0  0  0  0  0  0  0 84  0
## cocaine-heroin     0  0  0  0  6  0  0  0  0  0  0  0  0  0
## cocaine-marijuana  0  0  0  0  0  0  0 15  0  0  0  0  0  0
## cocaine-none       0  0  0  0  0  0  0  0  0  0 51  0  0  0
## heroin-alcohol      0  0  0  0  0  0  0  0 28  0  0  0  0  0
## heroin-benzos       0  0  0  0  6  0  0  0  0  0  0  0  0  0
## heroin-cocaine      0  0  0  0  0  0  0  0  0 29  0  0  0  0
## heroin-marijuana    0  0  0  5  0  0  0  0  0  0  0  0  0  0
## heroin-none         0  0  0  0  0  0  0  0  0  0  0 57  0  0
## heroin-other       1  2  0  0  0  0  0  0  0  0  0  0  0  0
## <NA>               1  0  0  0  0  0  0  0  0  0  0  0  0  0
```

```
tally(~ title=="alcohol-none", data=merged)

## title == "alcohol-none"
## TRUE FALSE <NA>
##    99   370     1

tally(~ title=="alcohol-other", data=merged)

## title == "alcohol-other"
## TRUE FALSE <NA>
##     5   464     1

sort(tally(~ title, data=merged), decreasing=TRUE)[1:3]

## title
## alcohol-none cocaine-alcohol alcohol-cocaine
##           99             84             57
```

REFERENCES

- Broman, K. (2015). Initial steps toward reproducible research. <http://kbroman.org/steps2rr/>.
- Leek, J. (2016). How to share data with a statistician. <https://github.com/jtleek/datasharing>.
- Lumley, T. (2015). stringsAsFactors = `jsigh`. <http://notstatschat.tumblr.com/post/124987394001/stringsasfactors-sigh>.
- National Institute of Alcohol Abuse and Alcoholism (2016). Rethinking drinking: What's 'low-risk' drinking? <http://rethinkingdrinking.niaaa.nih.gov/How-much-is-too-much/Is-your-drinking-pattern-risky/Whats-Low-Risk-Drinking.aspx>.
- NORC at the University of Chicago (2016). GSS data explorer. <https://gssdataexplorer.norc.org/>.
- Peng, R. D. (2015). stringsAsFactors: An unauthorized biography. <http://simplystatistics.org/2015/07/24/stringsasfactors-an-unauthorized-biography/>.
- Smith, T. W., Mardsen, P., Hout, M., and Kim, J. (2015). General social surveys, 1972-2014 [machine-readable data file].

282 Wickham, H. (2014). Tidy data. *Journal of Statistical Software*, 59(10).
283 Wilson, G., Bryan, J., Cranston, K., Kitzes, J., Nederbragt, L., and Teal, T. K. (2016).
284 Good enough practices for scientific computing. [https://swcarpentry.github.io/](https://swcarpentry.github.io/good-enough-practices-in-scientific-computing/)
285 [good-enough-practices-in-scientific-computing/](https://swcarpentry.github.io/good-enough-practices-in-scientific-computing/).