Understanding Coarsened Factors in `cvam`

Joseph L. Schafer*

October 19, 2021

Abstract

Coarsened data permits values that convey intermediate amounts of information between fully observed and fully missing (e.g., values that are censored, truncated or top-coded). Categorical variables in R, known as factors, provide only one code for missing values, with no convenient way to express other coarsened states. The `cvam` package extends R's factor mechanism to allow categorical variables with arbitary types of coarsening. This document introduces the coarsened factor and describes functions in `cvam` for creating and manipulating them. The package's modeling procedures are described in a separate document *Log-Linear Modeling with Missing and Coarsened Values Using the `cvam` Package*.

*Office of the Associate Director for Research and Methodology, United States Census Bureau, Washington, DC 20233, `joseph.l.schafer@census.gov`. This article is released to inform interested parties of ongoing research and to encourage discussion. The views expressed are those of the author and not necessarily those of the U.S. Census Bureau.

1 Review of categorical variables in R

1.1 Factors and their uses

In the statistical programmming language R (R Core Team, 2018), a categorical variable is called a factor. For example, consider the `ChickWeight` dataset distributed with R as part of its `datasets` package. These data came from a randomized experiment concerning the effects of diet on the growth of newly hatched chicks. The variable `Diet` is a factor with four possible values (levels), which are unceremoniously labeled `"1"`, `"2"`, `"3"`, and `"4"`.

```
> library(datasets)       # attach the library, if needed
> data(ChickWeight)       # load dataset into R workspace
> ChickWeight[1:3,]       # look at first three rows
```

```
  weight Time Chick Diet
1     42    0     1    1
2     51    2     1    1
3     59    4     1    1
```

```
> str(ChickWeight$Diet)   # examine structure of the variable Diet
```

```
 Factor w/ 4 levels "1","2","3","4": 1 1 1 1 1 1 1 1 1 1 ...
```

In exploratory data analyses, factors are used to define classification bins for generating tables and plots. Examples using the `ChickWeight` data are shown below, and the resulting plot is shown in Figure 1.

```
> # compute mean final weight at day 21 by Diet
> aggregate( weight ~ Diet, data = subset(ChickWeight, Time==21),
+     FUN = mean )
```

```
  Diet   weight
1    1 177.7500
2    2 214.7000
3    3 270.3000
4    4 238.5556
```

```
> # side-by-side boxplots of final weight at day 21 by Diet
> plot( weight ~ Diet, data = subset(ChickWeight, Time==21) )
```

Factors are often used as predictors in regression models. When a k-level factor appears on the right-hand side of a model formula, R automatically expresses the factor as a set of $k-1$ variables to contrast the effects of the different levels (Chambers and Hastie, 1992). In the example below, `Diet` is expressed as dummy indicators for levels `"2"`, `"3"`, and `"4"`, so that 1 becomes the reference level.

```
> # regress final weight at day 21 on Diet
> result <- lm( weight ~ Diet, data = subset(ChickWeight, Time==21 ) )
> summary(result)$coef
```

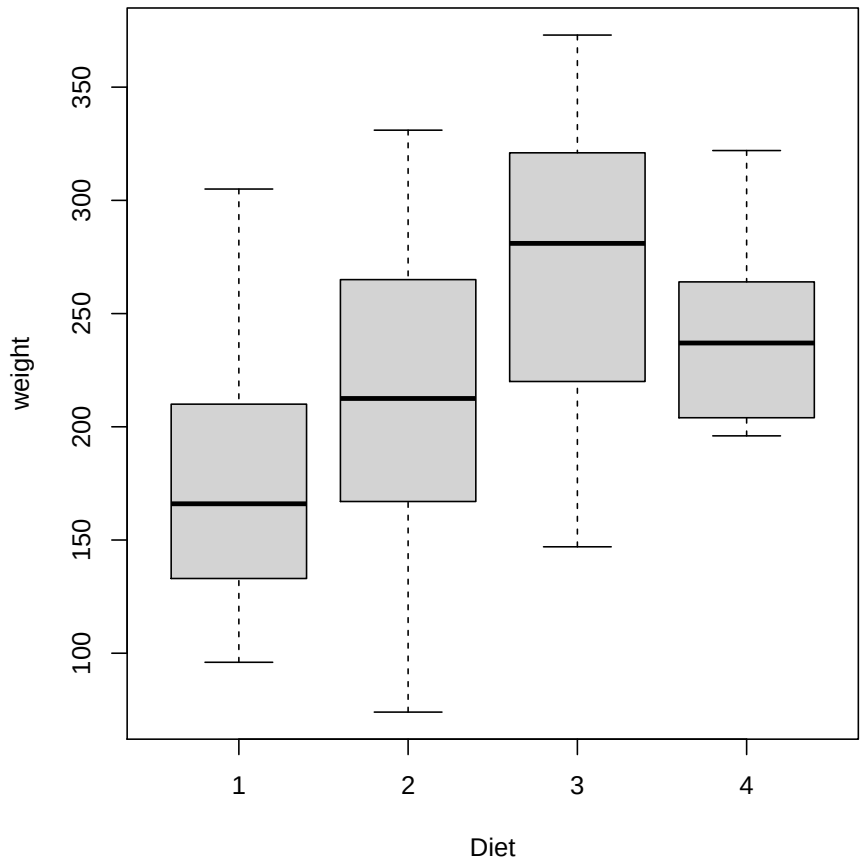

Figure 1: Boxplots of chick final weight, classified by diet.

```
            Estimate Std. Error   t value     Pr(>|t|)
(Intercept) 177.75000   15.99540 11.112571 6.068920e-14
Diet2        36.95000   25.79181  1.432626 1.595459e-01
Diet3        92.55000   25.79181  3.588349 8.796253e-04
Diet4        60.80556   26.65900  2.280864 2.782256e-02
```

A few R modeling functions will accept a factor on the left-hand side of a formula, treating the variable as the outcome in a multinomial regression. For example, the `multinom` function in the package `nnet` fits baseline-category logistic models (Venables and Ripley, 2013).

```
> library(nnet)
> # regress Diet on initial weight to check for balance
> resultA <- multinom(Diet ~ weight,
+    data = subset(ChickWeight, Time==0 ), trace=FALSE )
> # compare fit to that of a null (intercept-only) model
> resultB <- multinom(Diet ~ 1,
```

```
+    data = subset(ChickWeight, Time==0 ), trace=FALSE )
> resultB$deviance - resultA$deviance  # df = 3
[1] 3.56337
```

Factors may serve as identifiers for grouping observations in longitudinal and clustered analyses. In the example below, the `lmer` function from the package `lme4` (Bates et al., 2015) is used to fit a linear mixed-effects growth model with a random interecpt and slope for each chick.

```
> library(lme4)
> # Linear growth model with random intercepts and slopes
> result <- lmer( weight ~ Time + ( Time | Chick ),
+    data = ChickWeight )
```

Data stored as factors are often rearranged into other forms for summarizing and modeling, and many R functions are available for those manipulations. For example, consider the `HairEyeColor` dataset from the `datasets` package, which classifies 592 statistics students by hair color, eye color and sex. The data are stored as a `table`, a three-dimensional array that records the number of students in each cell of the three-way classification.

```
> HairEyeColor
, , Sex = Male

       Eye
Hair    Brown Blue Hazel Green
  Black    32   11    10     3
  Brown    53   50    25    15
  Red      10   10     7     7
  Blond     3   30     5     8

, , Sex = Female

       Eye
Hair    Brown Blue Hazel Green
  Black    36    9     5     2
  Brown    66   34    29    14
  Red      16    7     7     7
  Blond     4   64     5     8
```

At an earlier stage, these data may have existed as a rectangular data frame with 592 rows (one per student) and factor variables named `Hair`, `Eye` and `Sex`. Those factor variables may have been processed into `HairEyeColor`'s present form by the function `table` or `xtabs`.

1.2 Creating a factor

The most common way to create a factor variable in R is by calling the function `factor`. The primary argument to this function is a vector of data, typically numeric

or character. By default, `factor` will return a factor variable with one level for each distinct value found in the vector, and the levels will be arranged in ascending alphanumeric order.

```
> weather <- c("clear", "rain", "clear", "cloudy", "snow", "clear", "rain")
> weather <- factor(weather)
> table(weather)

weather
 clear cloudy   rain   snow
     3      1      2      1
```

Another commonly used function for creating a factor is `cut`, which bins numeric data into categories according to user-defined break points.

```
> # generate 1,000 U(0,1) random variates, then
> # classify them as low, medium, and high
> uniform <- runif(1000)
> lmh <- cut( uniform, breaks=c(0, .333, .667, 1),
+     labels=c("low", "medium", "high") )
> table(lmh)

lmh
   low medium   high
   331    349    320
```

1.3 Factor levels

If `x` is a factor, then `nlevels(x)` returns its number of levels. Internally, the factor's data values are stored as positive integers `1`, `2`, ..., `nlevels(x)`. For the most part, however, those integers are hidden from the user. Instead, the user typically sees character strings defined by the attribute `levels`, a character vector of length `nlevels(x)`. For example, let's look at `chickwts`, another chick-related dataset from the `datasets` package. This data frame has a factor variable `feed` with six descriptively named levels.

```
> str(chickwts)

'data.frame':        71 obs. of  2 variables:
 $ weight: num  179 160 136 227 217 168 108 124 143 140 ...
 $ feed  : Factor w/ 6 levels "casein","horsebean",..: 2 2 2 2 2 2 2 2 2 2 ...

> levels( chickwts$feed )

[1] "casein"    "horsebean" "linseed"   "meatmeal"  "soybean"   "sunflower"
```

This variable's `storage.mode` is `"integer"`.

```
> storage.mode( chickwts$feed )

[1] "integer"
```

However, it is the character strings in `levels` that are seen when the variable is displayed using the `print` function, and when it is tabulated using `table` or `xtabs`.

```
> chickwts$feed[1:5]   # implicitly calling print

[1] horsebean horsebean horsebean horsebean horsebean
Levels: casein horsebean linseed meatmeal soybean sunflower

> table(chickwts$feed)

   casein horsebean   linseed  meatmeal   soybean sunflower
       12        10        12        11        14        12

> xtabs(~ feed, data=chickwts)

feed
   casein horsebean   linseed  meatmeal   soybean sunflower
       12        10        12        11        14        12
```

Moreover, the relational operators `==` and `!=` compare the strings, not the integers.

```
> sum( chickwts$feed == "meatmeal" )

[1] 11

> chickwts$weight[ chickwts$feed == "horsebean" ]

 [1] 179 160 136 227 217 168 108 124 143 140
```

If you want to work with a factor's integer codes rather than its character-string levels, wrap the factor with `unclass`. This function strips away the object's `class` attribute, so that R no longer calls any of the special methods for factors, but treats the variable as if it were a just a vector of integers.

```
> unclass(chickwts$feed)

 [1] 2 2 2 2 2 2 2 2 2 2 3 3 3 3 3 3 3 3 3 3 3 5 5 5 5 5 5 5 5 5 5 5 5 5 6 6 6 6 6 6 6
[44] 6 6 6 6 6 4 4 4 4 4 4 4 4 4 4 4 4 4 1 1 1 1 1 1 1 1 1 1 1
attr(,"levels")
[1] "casein"    "horsebean" "linseed"   "meatmeal"  "soybean"   "sunflower"
```

1.4 Extracting and replacing portions of a factor

If you extract portions of a factor using the subsetting operator `[`, the result is another factor. By default, the new factor has the same `levels` as the original, even if some of those levels have no obervations in them.

```
> chickwts$feed[1:22]

 [1] horsebean horsebean horsebean horsebean horsebean horsebean horsebean horsebean
 [9] horsebean horsebean linseed   linseed   linseed   linseed   linseed   linseed
[17] linseed   linseed   linseed   linseed   linseed   linseed
Levels: casein horsebean linseed meatmeal soybean sunflower
```

Empty levels can be eliminated by the `droplevels` function, or by supplying the argument `drop=TRUE` when using `[`.

```
> droplevels( chickwts$feed[1:22] )

 [1] horsebean horsebean horsebean horsebean horsebean horsebean horsebean horsebean
 [9] horsebean horsebean linseed   linseed   linseed   linseed   linseed   linseed
[17] linseed   linseed   linseed   linseed   linseed   linseed
Levels: horsebean linseed

> chickwts$feed[1:22, drop=TRUE]   # does the same thing

 [1] horsebean horsebean horsebean horsebean horsebean horsebean horsebean horsebean
 [9] horsebean horsebean linseed   linseed   linseed   linseed   linseed   linseed
[17] linseed   linseed   linseed   linseed   linseed   linseed
Levels: horsebean linseed
```

The replacement version of `[` does not allow you to replace elements of a factor with values that are not already present among its levels. To do that, you would need to first modify the `levels` attribute.

```
> chickwts$feed[2] <- "HotDogs"   # this produces a missing value
> chickwts$feed[1:5]

[1] horsebean <NA>      horsebean horsebean horsebean
Levels: casein horsebean linseed meatmeal soybean sunflower

> levels(chickwts$feed) <- c( levels(chickwts$feed), "HotDogs" )
> chickwts$feed[ 2 ] <- "HotDogs"   # now it works
> chickwts$feed[1:5]

[1] horsebean HotDogs   horsebean horsebean horsebean
Levels: casein horsebean linseed meatmeal soybean sunflower HotDogs
```

1.5 Other factor attributes

To determine whether or not an object is a factor, R examines its `class` attribute. A factor's `class` is either `"factor"` or `c("ordered", "factor")`, depending on whether the variable is assumed to be nominal (whose categories have no intrinsic ordering) or ordinal (having categories that are ordered). An ordered factor may be created by the `ordered` function, or by calling `factor` or `cut` with the argument `ordered=TRUE`.

Some modeling functions will handle ordered and unordered factors differently. If a k-level unordered factor appears on the right-hand side of a regression formula, then by default R will create a set of $k-1$ dummy indicators that contrast levels 2, 3, ..., k against level 1. If the factor is ordered, then by default R will compute orthogonal contrasts for fitting a polynomial function of degree $k-1$. This behavior is controlled by the factor's attribute `contrasts`, a $k \times (k-1)$ matrix that shows how the regressors are defined.

```
> # For an unordered factor, default contrasts use dummy indicators
> contrasts(chickwts$feed)

          horsebean linseed meatmeal soybean sunflower HotDogs
casein            0       0        0       0         0       0
horsebean         1       0        0       0         0       0
linseed           0       1        0       0         0       0
meatmeal          0       0        1       0         0       0
soybean           0       0        0       1         0       0
sunflower         0       0        0       0         1       0
HotDogs           0       0        0       0         0       1

> # For an ordered factor, the default is orthogonal polynomials;
> # in the example below, they are linear and quadratic
> uniform <- runif(1000)
> lmh <- cut( uniform, breaks=c(0, .333, .667, 1),
+     labels=c("low", "medium", "high"), ordered=TRUE )
> contrasts(lmh)

               .L         .Q
[1,] -7.071068e-01  0.4082483
[2,] -7.850462e-17 -0.8164966
[3,]  7.071068e-01  0.4082483
```

Other types of contrasts are available; see `?contrasts` for details.

1.6 Missing values in factors

1.6.1 The ordinary `NA`

A missing value in a factor variable is displayed as `NA` when the factor is summarized or printed. Depending on the context, however, the `NA` can mean two very different things, and it is crucial to understand the difference.

In the ordinary situation, `NA` is not an element of `levels`. An `NA` in a factor means that the datum belongs to one of the levels, but we do not know which one. This type of missing value is stored as the R constant `NA_integer_` in the vector of integer codes, and its presence is detectable by the function `is.na`.

```
> # create a factor with a missing value
> party <- factor( c("Dem", "Ind", "Rep", NA, "Rep", "Ind", "Dem") )
```

```
> # Note that NA is not one of the levels
> party

[1] Dem  Ind  Rep  <NA> Rep  Ind  Dem
Levels: Dem Ind Rep

> # The missing value appears in the integer codes
> unclass(party)

[1]  1  2  3 NA  3  2  1
attr(,"levels")
[1] "Dem" "Ind" "Rep"

> # is.na returns TRUE if the value is missing, FALSE otherwise
> is.na(party)

[1] FALSE FALSE FALSE  TRUE FALSE FALSE FALSE
```

When NAs are represented in this fashion, most R functions understand them to be missing in the conventional sense, and the system handles them in ways that depend on the function being invoked. For example, with the `table` function, by default NAs will not be reported in the resulting frequency table; to see them, supply the argument `exclude=NULL`.

```
> table(party)

party
Dem Ind Rep
  2   2   2

> table(party, exclude=NULL)

party
 Dem  Ind  Rep <NA>
   2    2    2    1
```

With the modeling functions `lm` and `glm`, if a factor with missing values appears in a regression formula, R may attempt to remove the incomplete cases from the analysis, or the model-fitting procedure may fail. Treatment of missing values in those functions is determined by function arguments or by the global option `na.action`; see `?options`.

As already mentioned, the function `droplevels` will remove empty levels (i.e., levels with no obervations in them) from a factor. The optional argument `exclude` can be used to remove additional levels even if they are non-empty. Any observation within an excluded level becomes a missing value.

```
> party <- droplevels(party, exclude=c("Ind",NA))
> party

[1] Dem  <NA> Rep  <NA> Rep  <NA> Dem
Levels: Dem Rep
```

Notice that `NA` was explicitly included among the values supplied to `exclude`. If it were not, then `droplevels` would have put NAs into a level, as we now describe.

1.6.2 NA as a factor level

As an alternative to the usual way of handling missing values, we can instruct R to classify NAs into a level of their own. This will happen if we call factor with exclude=NULL,

```
> party <- factor( c("Dem", "Ind", "Rep", NA, "Rep", "Ind", "Dem"),
+    exclude=NULL)
> party

[1] Dem  Ind  Rep  <NA> Rep  Ind  Dem
Levels: Dem Ind Rep <NA>
```

or if we pass a factor to the function addNA.

```
> party <- factor( c("Dem", "Ind", "Rep", NA, "Rep", "Ind", "Dem") )
> party

[1] Dem  Ind  Rep  <NA> Rep  Ind  Dem
Levels: Dem Ind Rep

> party <- addNA(party)
> party

[1] Dem  Ind  Rep  <NA> Rep  Ind  Dem
Levels: Dem Ind Rep <NA>
```

The inverse operation to addNA is droplevels with exclude=NA.

```
> party <- droplevels(party, exclude=NA)
> party

[1] Dem  Ind  Rep  <NA> Rep  Ind  Dem
Levels: Dem Ind Rep
```

When NA is a factor level, the factor contains no missing values in the traditional sense. None of the integer codes are NA_integer_, and is.na always returns FALSE.

```
> party <- addNA(party)
> unclass(party)

[1] 1 2 3 4 3 2 1
attr(,"levels")
[1] "Dem" "Ind" "Rep" NA

> is.na(party)

[1] FALSE FALSE FALSE FALSE FALSE FALSE FALSE
```

When a factor with NA as a level appears on the right-hand side of a model formula, the regression functions lm and glm will include the NA cases in the analysis, creating a dummy code or other contrast term to distinguish NA from the other levels. That approach may be sensible if the model is intended only for prediction, but it often leads to unintended or undesirable consequences and should be used only with caution.

1.7 Manipulating factor levels

The replacement version of the function `levels`, known to R as `levels<-`, can be used to change a factor's `levels` attribute. Often it will not affect the underlying integer codes, but sometimes it will. If `x` is a factor, then replacing `levels(x)` with another character vector of the same length simply renames the categories without changing the integer codes.

```
> # draw 25 values of red, green, or blue with equal probabilities
> myFac <- cut( runif(25), breaks=c(0, .333, .667, 1),
+     labels=c("red", "green", "blue") )
> table(myFac)

myFac
  red green  blue
    5    10    10

> # change three colors to Three Stooges
> levels(myFac) <- c("Larry", "Curly", "Moe")
> table(myFac)

myFac
Larry Curly   Moe
    5    10    10
```

It does not matter if the replacement levels happen to be a permutation of the existing ones; the categories are merely renamed.

```
> # replace "Larry" with "Moe", "Curly with "Larry", "Moe" with "Curly"
> levels(myFac) <- c("Moe","Larry", "Curly")
> table(myFac)

myFac
  Moe Larry Curly
    5    10    10
```

Replacing `levels` by a longer vector will introduce empty levels.

```
> # add the mysterious fourth Stooge, creating an empty level
> levels(myFac) <- c("Moe","Larry", "Curly", "Shemp")
> table(myFac)

myFac
  Moe Larry Curly Shemp
    5    10    10     0
```

And replacing a single element of `levels` with another, existing level will change the integer codes, collapsing the two categories into one.

```
> # This will replace every occurrence of "Curly" with "Shemp"...
> levels(myFac)[3] <- "Shemp"
> # ...causing "Curly" to be dropped from the levels
> table(myFac)

myFac
  Moe Larry Shemp
    5    10    10
```

If we try to replace `levels` with a shorter vector, R will report an error, because it has not been told how the existing levels relate to the new ones. When reducing the number of levels, we can specify these relationships via a list. For example, suppose we have a factor with three levels, and we want to combine two levels into one, producing a new factor with two levels.

```
> party <- factor( c("Dem", "Ind", "Rep", "Dem", "Rep", "Ind", "Dem") )
> table(party)

party
Dem Ind Rep
  3   2   2

> # leave "Rep" alone, but combine "Dem" and "Ind" into "notRep"
> levels(party) <- list( Rep = "Rep", notRep = c("Dem", "Ind") )
> table(party)

party
   Rep notRep
     2      5
```

The `names` of the provided list become the `levels` of the new factor.

Other useful functions for manipulating factor levels include `relevel` and `reorder`, which may change the integer codes; see `?relevel` and `?reorder` for details.

2 Coarsened categorical variables

2.1 What are coarsened data?

Coarsened data is a general term for quantities that may be fully observed, entirely missing, or somewhere in between. Instead of obtaining a random variable's realized value, we are told that the value lies in a subset of the random variable's support.

Coarsened data are common in survival analysis. Suppose V_i ia a continuously distributed positive outcome (e.g., survival time) for observational unit i. Ideally, the analyst is told the actual value $V_i = v_i$, in which case the datum is fully observed. In lieu of that, the analyst may be told

- $V_i \in (0, a_i)$ for some $a_i > 0$, said to be *left-censored*;
- $V_i \in (b_i, \infty)$ for some $b_i > 0$, said to be *right-censored*;
- $V_i \in (a_i, b_i)$ for $a_i < b_i$, said to be *interval-censored*; or
- $V_i \in (0, \infty)$, which corresponds to a traditional missing value.

Procedures for survival analysis may accept any or all of these types, but special data structures might be needed. A continuous variable with all these types of censoring cannot be stored as a numeric vector with a single missing-value code. To analyze such data, we need to extend the usual objects to hold extra information.

2.2 Theory of coarsened data

A general paradigm for describing and analyzing coarsened data was developed by Heitjan and Rubin (1991) and Heitjan (1994). That framework built upon theory of missing data begun by Rubin (1976), and key concepts from the literature on missing data extend to coarsened data in natural ways.

In the missing-data literature, a *missing-data mechanism* is a process that operates on a sample of complete data to determine which data values will be observed and which ones will be missing. If we assume that the probabilities of missingness do not depend on any missing quantities, the missing values are said to be *missing at random* (MAR), and in those cases, explicit modeling of the missing-data mechanism is (usually) not necessary. With coarsened data, there is a *coarsening mechanism*, a process that operates on the realized data to determine if and how they are being coarsened. The analogue of MAR is *coarsened at random* (CAR), which allows us to forego building a model for the coarsening mechanism. For extended discussion of these topics and more references, see Little and Rubin (2002).

Some areas of applied statistics have developed special terminology for coarsened data, but the concepts are similar to those in the general theory of Heitjan and Rubin (1991) and Heitjan (1994). A prime example is *noninformative censoring* in survival analysis, which essentially means that the censored values are CAR.

2.3 Notation for coarsened categorical variables

Imagine a dataset with J categorical variables. Let V_{ij} denote the jth categorical variable for individual or observational unit i. Denote its set of possible values by

$$\mathbb{V}_j \;=\; \{1, 2, \ldots, \#\mathbb{V}_j\}.$$

(The symbol '#' is the cardinality operator. When applied to a set, it returns the number of elements in the set. We use this symbol to avoid adding unnecessary letters to our notation.) The elements of $\mathbb{V}_j$ are called *base-level codes*; these are all the possible responses that would be seen if there were no nonresponse or coarsening.

Let V_{ij}^* denote the observed, coarsened version of V_{ij}. The possible values of V_{ij}^* lie in the expanded set

$$\mathbb{V}_j^* \;=\; \{1, 2, \ldots, \#\mathbb{V}_j, \ldots, \#\mathbb{V}_j^*\},$$

where $\#\mathbb{V}_j^* > \#\mathbb{V}_j$. The extra codes not found in $\mathbb{V}_j$,

$$\mathbb{V}_j^* \setminus \mathbb{V}_j \;=\; \{\#\mathbb{V} + 1, \ldots, \#\mathbb{V}_j^*\},$$

are called *coarse-level codes*. (The symbol '\' is the set difference operator.)

If V_{ij}^* happens to be one of the base-level codes, then V_{ij} is fully known and is equal to V_{ij}^*,

$$\begin{aligned} V_{ij}^* = 1 \;&\Rightarrow\; V_{ij} = 1, \\ &\vdots \\ V_{ij}^* = \#\mathbb{V}_j \;&\Rightarrow\; V_{ij} = \#\mathbb{V}_j. \end{aligned}$$

However, if V_{ij}^* happens to be one of the coarse-level codes, the exact value of V_{ij} cannot be deduced from it. In that case, V_{ij} is known to lie within a given subset of the base-level codes, a set denoted by $\mathcal{M}_j(V_{ij}^*)$. That is,

$$V_{ij}^* \;=\; v^* \;\Rightarrow\; V_{ij} \;\in\; \mathcal{M}_j(v^*),$$

where $\mathcal{M}_j$ is the *mapping*, a one-to-many relation that maps elements of $\mathbb{V}^*$ onto non-empty subsets of $\mathbb{V}_j$. By convention, we will use the last coarse-level code to denote a traditional missing value,

$$\mathcal{M}_j(v^*) \;=\; \mathbb{V}_j \;\text{ when } v^* \;=\; \#\mathbb{V}_j^*.$$

For example, suppose that V_{ij} denotes a trichotomous political party affiliation with possible values 1=Democrat, 2=Republican, and 3=Independent. If individual i provides her exact affiliation, then V_{ij}^* will be 1, 2, or 3, and V_{ij}^* will coincide with V_{ij}. For those responses, the mappings are one-to-one,

$$\begin{aligned} \mathcal{M}_j(1) &= \{1\}, \\ \mathcal{M}_j(2) &= \{2\}, \\ \mathcal{M}_j(3) &= \{3\}. \end{aligned}$$

Now suppose she indicates that she is not a Democrat, but declines to say whether she is Republican or Independent. If we code that event as $V_{ij}^* = 4$, then the mapping is

$$\mathcal{M}_j(4) \;=\; \{2, 3\}.$$

Similarly, if she only indicates that she is not a Republican, and we code the event as $V_{ij}^* = 5$, then

$$\mathcal{M}_j(5) \;=\; \{1, 3\}.$$

If she indicates she is not an Independent, then $V_{ij}^* = 6$, and

$$\mathcal{M}_j(6) \;=\; \{1, 2\}.$$

Finally, if she declines to provide any information at all, then the coding is $V_{ij}^* = 7$, and the mapping is

$$\mathcal{M}_j(7) \;=\; \{1, 2, 3\},$$

which corresponds to a traditional missing value.

As the number of base-level codes increases, the number of possible coarse-level codes expands rapidly. If we were to include all possible coarsenings, $\#\mathbb{V}_j^*$ would be (two raised to the power of $\#\mathbb{V}_j$, minus one). In practice, we do not need to create a coarse-level code for every possible subset of the base-level codes, but only for groupings that actually happen. Continuing the previous example, suppose that party affiliation is measured by two items on a questionnaire. The first item is, "Do you consider yourself to be Independent?" If the response is "Yes," then the second item is skipped. If the response is "No," then the participant is presented with the second item, "Do you consider yourself to be Democrat or Republican?" Nonresponse to the second item produces a coarsened value of {Democrat, Republican}, and nonresponse to both items gives {Democrat, Republican, Independent}, which is a traditional missing value. The combinations {Democrat, Independent} and {Republican, Independent} do not occur in this study and therefore do not need to be represented in $\mathbb{V}_j^*$.

2.4 Where do coarsened categorical variables come from?

In a trivial sense, every dataset with missing values has coarsened data, because traditional missing values are a particular type of coarsening. As shown by in the previous discussion, coarsened values can arise when variables are created from multiple items on a questionnaire, if participants respond to some questions but not others. Coarsening may also result from attempts to harmonize data from multiple sources, when those sources attempt to measure similar constructs but with different levels of granularity.

Apart from certain areas of statistics (e.g., survival analysis), however, methods for coarsened variables are not widely used, lending the impression that coarsened values ought to be eliminated by editing them out of a dataset or recoding them as missing. As techniques and software become available, coarsened values can be incorporated into analyses in more principled manner, leading to more efficient results, because partial information is better than none.

3 Working with coarsened factors

3.1 How to create a coarsened factor

In the `cvam` package, coarsened factors are created by the function `coarsened`. Let us begin with a trivial example.

```
> myFac <- factor( c("red", "green", NA, "yellow",
+    "notRed", "green", "notGreen") )
> table(myFac, exclude=NULL)

myFac
   green notGreen   notRed      red   yellow     <NA>
       2        1        1        1        1        1
```

This factor, which R does not yet understand to be a coarsened factor, has five levels.

```
> levels(myFac)

[1] "green"    "notGreen" "notRed"   "red"      "yellow"
```

Based on their names, it appears to us that

- `"green"`, `"red"` and `"yellow"` are base levels,
- `"notGreen"` is a coarse level that maps to `c("red", "yellow")`, and
- `"notRed"` is a coarse level that maps to `c("green", "yellow")`.

Moreover, the missing value `NA` is a coarse level that maps to `c("green", "red", "yellow")`.

To turn this factor into a coarsened factor, we load the `cvam` package and call the `coarsened` function.

```
> library(cvam)
> myCoarsenedFac <- coarsened( myFac, levelsList =
+    list( notGreen = c("red", "yellow"), notRed = c("green", "yellow") ) )
```

The result is a factor,

```
> is.factor(myCoarsenedFac)
[1] TRUE
```

with all the usual factor properties,

```
> storage.mode(myCoarsenedFac)
[1] "integer"
> nlevels(myCoarsenedFac)
[1] 6
> levels(myCoarsenedFac)
[1] "green"    "red"      "yellow"   "notGreen" "notRed"   NA
```

plus some new properties which are displayed by the `print` function.

```
> myCoarsenedFac
[1] red      green    <NA>     yellow   notRed   green    notGreen
  Base levels: green red yellow
Coarse levels: notGreen notRed <NA>
Mapping:
         green red yellow
notGreen     0   1      1
notRed       1   0      1
<NA>         1   1      1
```

The `levelsList` argument that we supplied to `coarsened` instructed the function to

- interpret `"notGreen"` as a combination of `"red"` and `"yellow"`, and
- interpret `"notRed"` as a combination of `"green"` and `"yellow"`.

Notice that we did not explicitly tell `coarsened` that `"green"`, `"red"` and `"yellow"` were base levels. The function discerned the base levels by looking at `levels(myFac)` and eliminating everything in `names(levelsList)`. Notice also that we did not explicitly say that `NA` was a combination of `"green"`, `"red"` and `"yellow"`. Once the function identified the base levels, it automatically interpreted `NA` as a combination of all of them.

The `coarsened` function has only three arguments.

```
coarsened(obj, levelsList = list(), warnIfCoarsened = TRUE)
```

`obj`: a factor to be turned into a coarsened factor. This factor may have missing values, but it should not have `NA` as a level.

`levelsList`: a list that identifies each coarse level (except `NA`) and its mapping to the base levels.

`warnIfCoarsened`: if `TRUE`, a warning will be provided if `obj` is already a coarsened factor

The default value of `levelsList` is an empty list, which tells `coarsened` to treat every level in `levels(obj)` as a base level, and to create `NA` as the only coarse level.

3.2 Attributes of a coarsened factor

The coarsened factor that we created has the following attributes.

```
> attributes( myCoarsenedFac )

$levels
[1] "green"    "red"      "yellow"   "notGreen" "notRed"   NA

$class
[1] "coarsened" "factor"

$mapping
         green red yellow
notGreen     0   1      1
notRed       1   0      1
<NA>         1   1      1

$baseLevels
[1] "green"  "red"    "yellow"

$coarseLevels
[1] "notGreen" "notRed"   NA

$nBaseLevels
[1] 3

$nCoarseLevels
[1] 3

$baseLevelCodes
[1] 1 2 3

$coarseLevelCodes
[1] 4 5 6

$latent
[1] FALSE

$contrasts
```

```
         [,1] [,2]
green       1    0
red         0    1
yellow     -1   -1
notGreen    0    0
notRed      0    0
<NA>        0    0
```

- The `class` of coarsened factor is either `c("coarsened", "factor")` or `c("coarsened", "ordered", "factor")`, depending on whether the main argument to `coarsened` was ordered.
- The `levels` attribute includes the base levels and the coarse levels. The base levels are listed first, and `NA` always comes last.
- The `mapping` attribute is an integer matrix with elements `0` and `1`, showing the combination of base levels for each coarse level.
- The `contrasts` attribute is designed to facilitate log-linear modeling, as explained in the document *Log-Linear Modeling with Missing and Coarsened Values Using the* ***cvam*** *Package*. It is not intended for use by functions outside of the `cvam` package, e.g., regression analyses with `lm` or `glm`. Using a coarsened factor on the right-hand side of a model formula with those functions can produce nonsensical results.

Some attributes can be retrieved by functions of the same name. For example,

```
> baseLevels( myCoarsenedFac )

[1] "green"  "red"    "yellow"
```

is a convenient shorthand for `attr(myCoarsenedFac, "baseLevels")`.

Please note that, with very few exceptions, the attributes of a coarsened factor should only be set by the `coarsened` function and should not be directly changed by the user.

3.3 Example: Race and Hispanic origin

Over the last half century, it has become standard practice in the United States for census and survey questionnaires to include separate items for race and Hispanic origin. In the year 2000, the General Social Survey (GSS) (Smith et al., 2019) included an item based on the race question from the U.S. Census. Participants could choose from over a dozen race categories, or they could select "Some other race" and provide

their own. This item was given to a random half-sample, so it is missing for about 50% of participants. A separate question on Hispanic origin was given to the full sample. These two items are provided in the data frame `abortion2000` distributed with the `cvam` package. A cross-tabulation for these two items is shown below.

```
> str(abortion2000)

'data.frame':       2817 obs. of  19 variables:
 $ Age     : Ord.factor w/ 4 levels "18-29"<"30-49"<..: 1 2 4 2 1 1 2 2 2 2 ...
 $ Sex     : Factor w/ 2 levels "Female","Male": 2 1 1 1 1 1 2 1 2 2 ...
 $ Race    : Factor w/ 3 levels "White","Black",..: 1 1 1 1 1 1 1 1 1 1 ...
 $ CenRace : Factor w/ 4 levels "White","Black",..: NA NA NA 1 1 1 1 1 NA 1 ...
 $ Hisp    : Factor w/ 2 levels "nonHisp","Hisp": 1 2 1 1 1 1 1 1 1 1 ...
 $ Degree  : Ord.factor w/ 5 levels "<HS"<"HS"<"JunCol"<..: 4 2 2 2 3 2 2 3 5 4 ...
 $ Relig   : Factor w/ 5 levels "Prot","Cath",..: 1 1 1 5 4 1 5 1 1 1 ...
 $ Party   : Factor w/ 3 levels "Dem","Rep","Ind/Oth": 2 2 3 2 1 1 2 1 1 2 ...
 $ PolViews: Ord.factor w/ 3 levels "Con"<"Mod"<"Lib": 1 1 1 1 3 3 2 3 3 1 ...
 $ AbDefect: Factor w/ 3 levels "Yes","No","DK": 1 1 NA NA NA 1 1 1 NA NA ...
 $ AbNoMore: Factor w/ 3 levels "Yes","No","DK": 2 2 NA NA NA 1 2 2 NA NA ...
 $ AbHealth: Factor w/ 3 levels "Yes","No","DK": 1 2 NA NA NA 1 1 1 NA NA ...
 $ AbPoor  : Factor w/ 3 levels "Yes","No","DK": 2 2 NA NA NA 1 2 2 NA NA ...
 $ AbRape  : Factor w/ 3 levels "Yes","No","DK": 1 2 NA NA NA 1 1 1 NA NA ...
 $ AbSingle: Factor w/ 3 levels "Yes","No","DK": 2 2 NA NA NA 1 2 2 NA NA ...
 $ AbAny   : Factor w/ 3 levels "Yes","No","DK": 2 2 NA NA NA 1 2 2 NA NA ...
 $ WTSSALL : num  1.099 0.549 0.549 0.549 0.549 ...
 $ VSTRAT  : int  1687 1687 1687 1687 1687 1687 1687 1687 1687 1687 ...
 $ VPSU    : int  1 1 1 1 1 1 1 1 2 2 ...

> CenRace <- abortion2000$CenRace
> Hisp <- abortion2000$Hisp
> table(CenRace, Hisp, exclude=NULL)

       Hisp
CenRace nonHisp Hisp <NA>
  White    1042   50    1
  Black     198    3    0
  Hisp        0   41    0
  Other      44   19    0
  <NA>     1320   99    0
```

Notice that 41 persons (about 3% of the half-sample) have a value of `"Hisp"` for `CenRace`. Hispanic ancestry is viewed by some to be both an ethnicity and a race. These persons selected "Some other race" and described themselves as Hispanic, Latina, Latino, or something similar.

Data analysts often combine race and Hispanic origin into a single variable. Consider a classification into four levels,

$$
\begin{aligned}
1 &= \text{non-Hispanic White,} \\
2 &= \text{non-Hispanic Black,} \\
3 &= \text{non-Hispanic Other,} \\
4 &= \text{Hispanic.}
\end{aligned}
$$

In R, the colon operator ':' combines two factors into a single factor with a level for every possible combination of the operands' levels. Observe what happens if we apply this operator to `CenRace` and `Hisp`, both of which have missing values.

```
> RH <- Hisp:CenRace
> table(RH, exclude=NULL)

RH
nonHisp:White nonHisp:Black  nonHisp:Hisp nonHisp:Other    Hisp:White    Hisp:Black
         1042           198             0            44            50             3
    Hisp:Hisp    Hisp:Other          <NA>
           41            19          1420
```

Every case with a missing value for either of the two variables received a missing value in the result, and a large amount of useful information has been needlessly discarded. Notice that 99 missing values came from Hispanic persons with missing race; we may assume that they are Hispanic and manually assign them to level 4. But 1,320 missing values came from non-Hispanic persons with missing race; these are more problematic, because each of them could belong to any of the levels 1, 2, or 3. An ordinary factor in R cannot handle that partial information, but a coarsened factor can.

To create our coarsened factor, we first apply `addNA` to each factor, combine them with ':', and drop the empty levels.

```
> CenRace <- addNA(CenRace)
> Hisp <- addNA(Hisp)
> RH <- Hisp:CenRace
> table(RH)

RH
nonHisp:White nonHisp:Black  nonHisp:Hisp nonHisp:Other    nonHisp:NA    Hisp:White
         1042           198             0            44          1320            50
   Hisp:Black     Hisp:Hisp    Hisp:Other       Hisp:NA      NA:White      NA:Black
            3            41            19            99             1             0
      NA:Hisp      NA:Other         NA:NA
            0             0             0

> RH <- droplevels(RH)
> table(RH)

RH
nonHisp:White nonHisp:Black nonHisp:Other    nonHisp:NA    Hisp:White    Hisp:Black
         1042           198            44          1320            50             3
    Hisp:Hisp    Hisp:Other       Hisp:NA      NA:White
           41            19            99             1
```

In this example, there happen to be no observations with missing values for both `CenRace` and `Hisp`. If there were, they would belong to a level named `"NA:NA"`, and at this point we would want to set them to `NA` and drop the empty `"NA:NA"` level, like this:

```
> RH[ RH == "NA:NA" ] <- NA
> RH <- droplevels(RH)
```

Before applying the `coarsened` function, we reorder and combine levels using the `levels<-` function with a list, as described in Section 1.7.

```
> levels(RH) <- list(
+    nonHispWhite = "nonHisp:White",
+    nonHispBlack = "nonHisp:Black",
+    nonHispOther = "nonHisp:Other",
+    Hisp = c("Hisp:White", "Hisp:Black", "Hisp:Hisp", "Hisp:Other", "Hisp:NA"),
+    nonHispNA = "nonHisp:NA",
+    NAWhite = "NA:White" )
> table(RH)

RH
nonHispWhite nonHispBlack nonHispOther         Hisp    nonHispNA      NAWhite
        1042          198           44          212         1320            1
```

The factor now has six levels. The first four will become base levels, and the last two will become coarse levels. We are ready to create the coarsened factor.

```
> RH  <- coarsened( RH, levelsList = list(
+    nonHispNA = c("nonHispWhite", "nonHispBlack", "nonHispOther"),
+    NAWhite = c("nonHispWhite", "Hisp" ) ) )
> table(RH)

RH
nonHispWhite nonHispBlack nonHispOther         Hisp    nonHispNA      NAWhite
        1042          198           44          212         1320            1
        <NA>
           0
```

It's a good idea to examine the `mapping` matrix to make sure everything looks correct.

```
> mapping(RH)

          nonHispWhite nonHispBlack nonHispOther Hisp
nonHispNA            1            1           1    0
NAWhite              1            0           0    1
<NA>                 1            1           1    1
```

Notice that `coarsened` automatically added an extra coarse level called `NA`, which in this example happens to be empty.

Because `RH` has the same length as the other variables in `abortion2000`, it may be put into the data frame.

```
> abortion2000 <- data.frame(abortion2000, RH)
> abortion2000$RH <- RH    # does the same thing
```

When a coarsened factor is put into a data frame, all of its attributes are preserved.

```
> identical( attributes(abortion2000$RH), attributes(RH) )
```

```
[1] TRUE
```

These attributes are needed by `cvam`'s modeling functions, which are described in the companion document *Log-Linear Modeling with Missing and Coarsened Values Using the `cvam` Package*.

3.4 Tabulating coarsened factors

Because a coarsened factor inherits from class `"factor"`, it can be passed to any R function that accepts factors. If that function is not part of the `cvam` package, it will treat coarse levels no differently from base levels. For example, the `table` function, which is called by `summary`, displays frequencies for all base levels and all coarse levels, including `NA`.

```
> summary(RH)    # essentially the same as table(RH)
```

```
nonHispWhite nonHispBlack nonHispOther         Hisp    nonHispNA      NAWhite
        1042          198           44          212         1320            1
        <NA>
           0
```

When applied to ordinary factors, however, the `table` function omits ordinary `NA`s by default. So if a coarsened and ordinary factor are cross-tabulated, the default behavior is to treat `NA` as a level for the coarsened factor but omit `NA`s from the ordinary factor.

```
> #  from abortion2000, a three-level factor
> PolViews <- abortion2000$PolViews
> # there are some missing values
> table( is.na(PolViews) )
```

```
FALSE  TRUE
 2644   173
```

```
> # but the NAs don't show up in a table
> table(RH, PolViews)
```

```
             PolViews
RH            Con Mod Lib
  nonHispWhite 337 373 274
  nonHispBlack  44  90  45
```

```
  nonHispOther   9  21  11
  Hisp          59  74  63
  nonHispNA    440 496 307
  NAWhite        1   0   0
  <NA>           0   0   0
```

To display NAs for the ordinary factor, you can

- call table with the argument exclude=NULL,
- explicitly turn NA into a level of the ordinary factor by calling addNA, or
- turn the ordinary factor into a coarsened factor, which does essentially the same thing as addNA.

```
> table(RH, PolViews, exclude=NULL)

             PolViews
RH             Con Mod Lib <NA>
  nonHispWhite 337 373 274   58
  nonHispBlack  44  90  45   19
  nonHispOther   9  21  11    3
  Hisp          59  74  63   16
  nonHispNA    440 496 307   77
  NAWhite        1   0   0    0
  <NA>           0   0   0    0

> table(RH, PolViews = addNA(PolViews) )

             PolViews
RH             Con Mod Lib <NA>
  nonHispWhite 337 373 274   58
  nonHispBlack  44  90  45   19
  nonHispOther   9  21  11    3
  Hisp          59  74  63   16
  nonHispNA    440 496 307   77
  NAWhite        1   0   0    0
  <NA>           0   0   0    0

> table(RH, PolViews = coarsened(PolViews) )

             PolViews
RH             Con Mod Lib <NA>
  nonHispWhite 337 373 274   58
  nonHispBlack  44  90  45   19
  nonHispOther   9  21  11    3
  Hisp          59  74  63   16
  nonHispNA    440 496 307   77
  NAWhite        1   0   0    0
  <NA>           0   0   0    0
```

The xtabs function is similar to table, but the variables to be tabulated are specified in a formula. To instruct xtabs to display NAs in an ordinary factor, use the argument addNA=TRUE,

```
> xtabs( ~ RH + PolViews, addNA=TRUE)
```

```
             PolViews
RH            Con Mod Lib <NA>
  nonHispWhite 337 373 274   58
  nonHispBlack  44  90  45   19
  nonHispOther   9  21  11    3
  Hisp          59  74  63   16
  nonHispNA    440 496 307   77
  NAWhite        1   0   0    0
  <NA>           0   0   0    0
```

or wrap the ordinary factor with `addNA` or `coarsened`. Coarse levels are also displayed in flat tables, which are two-dimensional displays of multiway frequency tables created by the function `ftable`. To display `NA`s in an ordinary factor, wrap the factor with `addNA` and call `ftable` with `exclude=NULL`.

```
> # display a flat version of a three-way table, with Sex:RH as
> # the row and PolViews as the column, showing the NAs in PolViews
> Sex <- abortion2000$Sex
> ftable( addNA(PolViews) ~ Sex + RH, exclude=NULL )
```

```
                   addNA(PolViews) Con Mod Lib  NA
Sex    RH
Female nonHispWhite                163 214 154  41
       nonHispBlack                 34  52  26  10
       nonHispOther                  4  15   6   1
       Hisp                         29  41  45   9
       nonHispNA                   214 299 177  53
       NAWhite                       1   0   0   0
       NA                            0   0   0   0
Male   nonHispWhite                174 159 120  17
       nonHispBlack                 10  38  19   9
       nonHispOther                  5   6   5   2
       Hisp                         30  33  18   7
       nonHispNA                   226 197 130  24
       NAWhite                       0   0   0   0
       NA                            0   0   0   0
```

To tabulate a coarsened factor without displaying its coarse levels, use the `cvam` function `dropCoarseLevels`. This function removes the coarse levels from a coarsened factor, sets the coarsened values to `NA`, and returns an ordinary factor as its result.

```
> table( RH=dropCoarseLevels(RH), PolViews)
```

```
             PolViews
RH            Con Mod Lib
  nonHispWhite 337 373 274
  nonHispBlack  44  90  45
  nonHispOther   9  21  11
  Hisp          59  74  63
```

If the only coarse level is `NA`, then no information is lost when `dropCoarseLevels` is applied. If other non-empty coarse levels are present, however, the partial information carried by those observations is effectively discarded.

3.5 Creating coarsened factors from tabulated or grouped data

In Section 3.3, we created `RH` from a data frame with one row per individual in the survey. Datasets with rows for individual units are called microdata. For the most part, any procedure for creating coarsened factors from microdata can also be applied to tabulated or grouped data, if those data exist in a data frame.

To illustrate, let's create a grouped dataset from the demographic variables `Age`, `Sex`, `CenRace`, and `Hisp`.

```
> groupedData = as.data.frame( xtabs( ~ Age + Sex + CenRace + Hisp,
+    data=abortion2000, addNA=TRUE) )
> dim(groupedData)
```

```
[1] 150   5
```

```
> head(groupedData)
```

```
    Age    Sex CenRace    Hisp Freq
1 18-29 Female   White nonHisp   96
2 30-49 Female   White nonHisp  244
3 50-64 Female   White nonHisp  114
4   65+ Female   White nonHisp  115
5  <NA> Female   White nonHisp    3
6 18-29   Male   White nonHisp   91
```

```
> # eliminate rows with Freq == 0
> groupedData <- subset( groupedData, Freq > 0 )
> dim(groupedData)
```

```
[1] 69  5
```

The `xtabs` function created a four-dimensional array of frequencies, and the option `addNA=TRUE` ensured that missing values in the factors were retained. The number of cells in that four-dimensional array is $5 \times 2 \times 5 \times 3 = 150$. Wrapping `xtabs` with `as.data.frame` reshaped the array into a data frame with 150 rows and five variables: one factor for each of the four dimensions, plus an integer-valued variable `Freq` containing the cell counts. Many cells in the four-dimensional table were empty, and removing rows of the data frame with frequencies of zero reduced its size to 69 by 5.

From this grouped dataset, we may now form the coarsened factor `RH` using exactly the same procedure that we used with microdata.

```
> CenRace <- addNA(groupedData$CenRace)
> Hisp <- addNA(groupedData$Hisp)
> RH <- Hisp:CenRace
> RH <- droplevels(RH)
> levels(RH) <- list(
+    nonHispWhite = "nonHisp:White",
+    nonHispBlack = "nonHisp:Black",
```

```
+    nonHispOther = "nonHisp:Other",
+    Hisp = c("Hisp:White", "Hisp:Black", "Hisp:Hisp", "Hisp:Other", "Hisp:NA"),
+    nonHispNA = "nonHisp:NA",
+    NAWhite = "NA:White" )
> RH  <- coarsened( RH, levelsList = list(
+    nonHispNA = c("nonHispWhite", "nonHispBlack", "nonHispOther"),
+    NAWhite = c("nonHispWhite", "Hisp" ) ) )
> # copy the coarsened factor into the grouped data frame
> groupedData$RH <- RH
```

To produce a one-way classification by `RH` from this grouped dataset, we sum the variable `Freq` within levels of `RH` using `aggregate`.

```
> aggregate( Freq ~ RH, FUN=sum, data=groupedData)

           RH Freq
1 nonHispWhite 1042
2 nonHispBlack  198
3 nonHispOther   44
4         Hisp  212
5    nonHispNA 1320
6      NAWhite    1
```

3.6 Retaining coarsened factor attributes

Standard R functions for manipulating and reshaping data were not designed for coarsened factors. The `cvam` package provides versions of the extraction functions `[` and `[[`, and versions of the replacement functions `[<-` and `[[<-`, to preserve the special attributes of coarsened factors through subsetting and replacement. For example, consider what happens when we extract rows from a data frame using `[` or `subset`.

```
> # list the attributes of our coarsened factor RH
> names( attributes( abortion2000$RH ) )

 [1] "levels"           "class"            "mapping"          "baseLevels"
 [5] "coarseLevels"     "nBaseLevels"      "nCoarseLevels"    "baseLevelCodes"
 [9] "coarseLevelCodes" "latent"           "contrasts"

> # extract females using `[` and list the attributes
> femOnly <- abortion2000[ abortion2000$Sex == "Female", ]
> names( attributes( femOnly$RH ) )

 [1] "levels"           "class"            "mapping"          "baseLevels"
 [5] "coarseLevels"     "nBaseLevels"      "nCoarseLevels"    "baseLevelCodes"
 [9] "coarseLevelCodes" "latent"           "contrasts"

> # do the same thing with subset
> femOnly <- subset( abortion2000, Sex == "Female" )
> names( attributes( femOnly$RH ) )
```

```
[1] "levels"           "class"            "mapping"          "baseLevels"
[5] "coarseLevels"     "nBaseLevels"      "nCoarseLevels"    "baseLevelCodes"
[9] "coarseLevelCodes" "latent"           "contrasts"
```

Unfortunately, when coarsened factors are subjected to other manipulations, their special attributes are sometimes lost. For example, none of the special attributes persist through an application of `xtabs` and `as.data.frame`:

```
> newGrouped <- as.data.frame( xtabs( ~ Age + Sex + RH, data=abortion2000,
+    addNA = TRUE ) )
> newGrouped <- subset( newGrouped, Freq > 0 )
> names( attributes( newGrouped$RH ) )
```

```
[1] "levels" "class"
```

In this case, the attributes can be restored manually:

```
> attributes( newGrouped$RH ) <- attributes( abortion2000$RH )
```

An experimental R package named `sticky` (Brown, 2017) was created for this purpose. If we apply the `sticky` function to a coarsened factor, its `class` is modified to `c("sticky", "coarsened", "factor")`, and the `sticky` package works silently behind the scenes to help retain the extra attributes. This package does not solve every problem, however, and in certain cases you may still need to restore the attributes yourself.

4 Looking ahead

At this point, we have introduced coarsened factors and explained how to create and manipulate them, but readers may still be wondering why anyone should bother with these new objects. Handling `NA`s is difficult enough, and coarsened values are yet another inconvenience that analysts would rather avoid. In typical applications, the base levels of variables are important, and observations at the coarse levels are worth paying attention to only if they improve our understanding what is happening at the base levels. That is precisely why `cvam` was created. This package allows us to fit models that describe the base levels using the information in coarsened values.

Returning to the notation of Section 2.3, our goal is to describe the categorical variables $(V_{i1}, \ldots, V_{iJ})$ and the relationships among them, but the available data are coarsened versions $(V^*_{i1}, \ldots, V^*_{iJ})$. The `cvam` package allows a user to model the joint distribution of $(V_{i1}, \ldots, V_{iJ})$ from observations of $(V^*_{i1}, \ldots, V^*_{iJ})$. To compute proper answers, special procedures are needed; we cannot simply discard the coarsened values, even in the univariate $(J = 1)$ case. The modeling functions in `cvam` provides those answers an efficient and hassle-free manner.

To see why this matters, suppose we try to estimate proportions within the categories of race and Hispanic origin defined in Section 3.3,

$$\begin{aligned} 1 &= \text{non-Hispanic White}, \\ 2 &= \text{non-Hispanic Black}, \\ 3 &= \text{non-Hispanic Other}, \\ 4 &= \text{Hispanic}, \end{aligned}$$

from the frequencies in our coarsened factor `RH`. Dropping the coarsened values, we obtain these sample proportions.

```
> dropRH <- dropCoarseLevels( abortion2000$RH )
> round( table(dropRH) / sum( table(dropRH) ), 4 )

dropRH
nonHispWhite nonHispBlack nonHispOther         Hisp
      0.6965       0.1324       0.0294       0.1417
```

However, the maximum-likelihood (ML) estimates based on the full data are starkly different:

```
nonHispWhite nonHispBlack nonHispOther         Hisp
      0.7506       0.1425       0.0317       0.0753
```

Using ML reduces the estimated proportion of Hispanics by nearly one half. We explain how to obtain these results in the companion vignette *Log-Linear Modeling with Missing and Coarsened Values Using the* `cvam` *Package*.

References

Bates, D., Mächler, M., Bolker, B., and Walker, S. (2015). Fitting linear mixed-effects models using lme4. *Journal of Statistical Software*, 67(1):1–48.

Brown, C. (2017). *sticky: Persist Attributes Across Data Operations*. R package version 0.5.2.

Chambers, J. M. and Hastie, T. J., editors (1992). *Statistical Models in S*, volume 251. Wadsworth & Brooks/Cole Advanced Books & Software Pacific Grove, CA.

Heitjan, D. F. (1994). Ignorability in general incomplete-data models. *Biometrika*, 81(4):701–708.

Heitjan, D. F. and Rubin, D. B. (1991). Ignorability and coarse data. *The Annals of Statistics*, 19(4):2244–2253.

Little, R. J. and Rubin, D. B. (2002). *Statistical Analysis with Missing Data, Second Edition*. John Wiley & Sons, New York.

R Core Team (2018). *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria.

Rubin, D. B. (1976). Inference and missing data. *Biometrika*, 63(3):581–592.

Smith, T. W., Davern, M., Freese, J., and Morgan, S. L. (2019). *General Social Surveys, 1972–2018*. National Data Program for the Social Sciences, No. 25. NORC, Chicago. 1 data file (64,814 logical records) + 1 codebook (3,758 pp.).

Venables, W. N. and Ripley, B. D. (2013). *Modern Applied Statistics with S*. Springer Science & Business Media, New York, fourth edition.