GRIM in scrutiny

NOTE — RUN THIS IN THE CONSOLE:

rmarkdown::render("/Users/lukasjung/Documents/R-code/scrutiny/vignettes/grim.Rmd")

```
library(scrutiny)
library(dplyr)
```

Granularity-related inconsistency of means, or GRIM, is a test for the mathematical consistency of reported means or proportions with the corresponding sample size. It applies to summary statistics of all granular (ordinal or interval, i.e., non-continuous) numerical distributions. GRIM answers a simple question: Is it possible that a granular distribution has both the reported mean or percentage and the reported sample size?

This vignette covers scrutiny's implementation of the GRIM test. It has the following sections — to get started, though, you only need the first two:

1. The basic `grim()` function.
2. A specialized mapping function, `grim_map()`.
3. Testing sequences of means or proportions.
4. The `audit()` method for summarizing `grim_map()`'s results.
5. The visualization function `grim_plot()`.
6. Statistical benchmarks, such as granularity and the GRIM ratio.

Basic GRIM testing

To test if a reported mean of 5.27 on a granular scale is GRIM-consistent with a sample size of 43, run this:

```
grim(x = "5.27", n = 43)
#>  5.27
#> FALSE
```

Note that `x`, the reported mean, needs to be a string. The reason is that strings preserve trailing zeros, which can be crucial for GRIM-testing. Numeric values don't, and even converting them to strings won't help. A workaround for larger numbers of such values, `restore_zeros()`, is discussed in the *Data wrangling* vignette.

`grim()` has some further arguments, but all of them can be used from within `grim_map()`. Since `grim_map()` is often the more useful function in practice, the other arguments will be discussed in that context.

Testing multiple cases

Working with `grim_map()`

If you want to test more than a handful of cases, the recommended way is to enter them into a data frame and to run `grim_map()` on the data frame. A useful way to enter such data is to copy them from a PDF file and paste them into `tibble::tribble()`, used here via dplyr:

```
pigs1 <- tribble(
  ~x,
"8.97",
"2.61",
"7.26",
"3.64",
"9.26",
"10.46",
"7.39"
) %>%
  mutate(n = 28)
```

Use RStudio's multiple cursors as described in *Enter numbers as strings* [LINK TO THAT SECTION OF THE DATA WRANGLING VIGNETTE] to draw quotation marks around all the `x` values, and to set commas at the end.

Now, simply run `grim_map()` on that data frame:

```
pigs1 %>%
  grim_map()
#> # A tibble: 7 x 5
#>   x         n items consistency ratio
#>   <chr> <int> <int> <lgl>       <dbl>
#> 1 8.97     28     1 FALSE        0.72
#> 2 2.61     28     1 TRUE         0.72
#> 3 7.26     28     1 FALSE        0.72
#> 4 3.64     28     1 TRUE         0.72
#> 5 9.26     28     1 FALSE        0.72
#> 6 10.46    28     1 TRUE         0.72
#> 7 7.39     28     1 TRUE         0.72
```

The `x` and `n` columns are the same as in the input. By default, the number of `items` composing the mean is assumed to be 1. The main result, `consistency`, is the GRIM consistency of the former three columns. `ratio` is the GRIM ratio (see section *The GRIM ratio*).

Scale items

If the mean is composed of multiple items, set the `items` argument to that number. Below are hypothetical means of a three-items scale. With the single-item default, half of these are wrongly flagged as inconsistent:

```
pigs2 <- tribble(
   ~x,
  "5.90",
  "5.71",
  "3.50",
  "3.82",
  "4.61",
  "5.24",
) %>%
  mutate(n = 40)

pigs2 %>%
  grim_map()  # default is wrong here!
#> # A tibble: 6 x 5
#>   x         n items consistency ratio
#>   <chr> <int> <int> <lgl>       <dbl>
#> 1 5.90     40     1 TRUE          0.6
#> 2 5.71     40     1 FALSE         0.6
#> 3 3.50     40     1 TRUE           0.6
#> 4 3.82     40     1 TRUE           0.6
#> 5 4.61     40     1 FALSE          0.6
#> 6 5.24     40     1 FALSE          0.6
```

Yet, all of them are consistent if the correct number of items is stated:

```
pigs2 %>%
  grim_map(items = 3)
#> # A tibble: 6 x 5
#>   x         n items consistency ratio
#>   <chr> <int> <int> <lgl>       <dbl>
#> 1 5.90     40     3 TRUE         -0.2
#> 2 5.71     40     3 TRUE         -0.2
#> 3 3.50     40     3 TRUE         -0.2
#> 4 3.82     40     3 TRUE         -0.2
#> 5 4.61     40     3 TRUE         -0.2
#> 6 5.24     40     3 TRUE         -0.2
```

It is also possible to include an `items` column in the data frame instead:

```
pigs3 <- tribble(
   ~x,    ~items,
  "6.92",  1,
  "3.48",  1,
  "1.59",  2,
  "2.61",  2,
  "4.04",  3,
  "4.50",  3,
) %>%
  mutate(n = 30)

pigs3 %>%
  grim_map()
#> # A tibble: 6 x 5
#>   x         n items consistency ratio
#>   <chr> <int> <dbl> <lgl>       <dbl>
#> 1 6.92     30     1 FALSE         0.7
#> 2 3.48     30     1 FALSE         0.7
#> 3 1.59     30     2 FALSE         0.4
#> 4 2.61     30     2 FALSE         0.4
#> 5 4.04     30     3 TRUE          0.1
#> 6 4.50     30     3 TRUE          0.1
```

Percentage conversion

An underappreciated strength of GRIM is testing percentages. Since these are actually decimal numbers inflated by a factor of 100, percentages come with two "free" decimal places. However, percentages are often reported with decimal places beyond those two, which increases the probability of GRIM-inconsistencies unless true values were correctly reported.

Both `grim()` and `grim_map()` have a `percent` argument which, if set to `TRUE`, divides the `x` values by 100 and increases the decimal count by two, so that percentages can be tested just like means:

```
pigs4 <- tribble(
  ~x,     ~n,
  "32.5",  438,
  "35.6",  455,
  "21.7",  501,
  "39.3",  516,
)

pigs4 %>%
  grim_map(percent = TRUE)
#> # A tibble: 4 x 5
#>   x         n items consistency ratio
#>   <chr> <int> <int> <lgl>       <dbl>
#> 1 32.5    438     1 FALSE       0.562
#> 2 35.6    455     1 TRUE        0.545
#> 3 21.7    501     1 FALSE       0.499
#> 4 39.3    516     1 TRUE        0.484
```

Rounding

The scrutiny package provides infrastructure for reconstructing rounded numbers. All of that can be commanded from within `grim()` and `grim_map()`. Several arguments allow for stating the precise way in which the original numbers have supposedly been rounded.

First and foremost is the `rounding` argument. It takes a string with the rounding procedure's name, which leads to the number being rounded in either of these ways:

1. Rounded `"up"` or `"down"` from 5. Note that SAS, SPSS, Stata, Matlab, and Excel round `"up"` from 5, whereas Python rounds `"down"` from 5.
2. Rounded to `"even"` using base R's own `round()`.
3. Rounded `"up_from"` or `"down_from"` some number, which then needs to be specified via the `threshold` argument.
4. Given a `"ceiling"` or `"floor"` at the respective decimal place.
5. Rounded towards zero with `"trunc"` or away from zero with `"anti_trunc"`.

The default, `"up_or_down"`, allows for numbers rounded either `"up"` or `"down"` from 5 when GRIM-testing; and likewise for `"up_from_or_down_from"` and `"ceiling_or_floor"`. For more about these procedures, see documentation for `round()`, `round_up()`, and `round_ceiling()`. These include all of the above ways of rounding.

Points 3 to 5 above list some quite obscure options that were only included to cover a wide spectrum of possible rounding procedures. The same is true for the `threshold` and `symmetric` arguments, so these aren't discussed here any further.

By default, `grim()` and `grim_map()` accept values rounded either up or down from 5. If you have reason to impose stricter assumptions on the way `x` was rounded, write code like this:

```
pigs1 %>%
  grim_map(rounding = "up")
#> # A tibble: 7 x 5
#>   x         n items consistency ratio
#>   <chr> <int> <int> <lgl>       <dbl>
#> 1 8.97     28     1 FALSE        0.72
#> 2 2.61     28     1 TRUE         0.72
#> 3 7.26     28     1 FALSE        0.72
#> 4 3.64     28     1 TRUE         0.72
#> 5 9.26     28     1 FALSE        0.72
#> 6 10.46    28     1 TRUE         0.72
#> 7 7.39     28      1 TRUE          0.72

pigs1 %>%
  grim_map(rounding = "even")
#> # A tibble: 7 x 5
#>   x         n items consistency ratio
#>   <chr> <int> <int> <lgl>       <dbl>
#> 1 8.97     28      1 FALSE         0.72
#> 2 2.61     28      1 TRUE          0.72
#> 3 7.26     28      1 FALSE         0.72
#> 4 3.64     28      1 TRUE          0.72
#> 5 9.26     28      1 FALSE         0.72
#> 6 10.46    28      1 TRUE          0.72
#> 7 7.39     28      1 TRUE          0.72
```

Although changing the rounding procedure didn't make any difference in this case, it is important to account for the different ways in which numbers might be rounded, if only to demonstrate that some given results are robust to those variable decisions. To err on the side of caution, the default for `rounding` is the permissive `"up_or_down"`.

Test numeric sequences

Analysts might be interested in a mean or percentage value's numeric neighborhood. Suppose you found a reported mean of 5.30 to be GRIM-inconsistent with a sample size of 32. You might hypothesize that it was swapped with the nearby correct value.

There are two ways to go about this: (1) testing sequence vectors with `seq_distance()` and `grim()`, or (2) testing sequence data frames with `seq_distance_df()` and `grim_map()`. I think the second way is more useful, but see for yourself.

With `seq_distance()` and `grim()`

Create a sequence starting from 5.30 with `seq_distance()` and pipe the result into `grim()`, where you specify `n` as 32. Although the starting point is allowed to be numeric, we need to enter 5.30 as a string to preserve the trailing zero:

```
seq_distance(from = "5.30") %>%
  grim(n = 32)
#>  5.30  5.31  5.32  5.33  5.34  5.35  5.36  5.37  5.38
#> FALSE  TRUE FALSE FALSE  TRUE FALSE FALSE  TRUE  TRUE
#>  5.39
#> FALSE
```

With `seq_distance_df()` and `grim_map()`

First, create a data frame with a sequence column starting from 5.30 with `seq_distance_df()`, adding an `n` column that's constant at 32:

```
seq_distance_df(.from = "5.30", n = 32)
#> # A tibble: 10 x 2
#>    x         n
#>    <chr> <dbl>
#>  1 5.30     32
#>  2 5.31     32
#>  3 5.32     32
#>  4 5.33     32
#>  5 5.34     32
#>  6 5.35     32
#>  7 5.36     32
#>  8 5.37     32
#>  9 5.38     32
#> 10 5.39     32
```

This data frame can readily be piped into `grim_map()`, which allows for an immediate assessment:

```
seq_distance_df(.from = "5.30", n = 32) %>%
  grim_map()
#> # A tibble: 10 x 5
#>    x         n items consistency ratio
#>    <chr> <int> <int> <lgl>       <dbl>
#>  1 5.30     32     1 FALSE        0.68
#>  2 5.31     32     1 TRUE         0.68
#>  3 5.32     32     1 FALSE        0.68
#>  4 5.33     32     1 FALSE        0.68
#>  5 5.34     32     1 TRUE         0.68
#>  6 5.35     32     1 FALSE        0.68
#>  7 5.36     32     1 FALSE        0.68
#>  8 5.37     32     1 TRUE         0.68
#>  9 5.38     32     1 TRUE         0.68
#> 10 5.39     32     1 FALSE        0.68
```

These results, in turn, can be analyzed with `seq_test_ranking()`:

```
seq_distance_df(.from = "5.30", n = 32) %>%
  grim_map() %>%
  seq_test_ranking()
#> Explanation:
#> i There are 4 consistent value sets, starting with row
#>   number 2 in the data frame created by `grim_map()`.
#> i All other value sets are inconsistent.
#> i The consistent sets lag the inconsistent ones by numbers
#>   of places from 1 to 3 in the `grim_map()` data frame.
#>
#> # A tibble: 6 x 3
#>   consistent inconsistent lead_lag
#>        <int>        <int>    <int>
#> 1          2            1       -1
#> 2          5            3       -2
#> 3          8            4       -4
#> 4          9            6       -3
#> 5         NA            7       NA
#> 6         NA           10       NA
```

Summarize results with `audit()`

Following up on a call to `grim_map()`, the generic function `audit()` summarizes test results:

```
pigs1 %>%
  grim_map() %>%
  audit()
```

```
#> # A tibble: 1 x 7
#>   incons_cases all_cases incons_rate mean_ratio
#>          <int>     <int>       <dbl>      <dbl>
#> 1            3         7       0.429       0.72
#> # ... with 3 more variables: incons_base_ratio <dbl>,
#> #   testable_cases <int>, testable_rate <dbl>
```

These columns are —

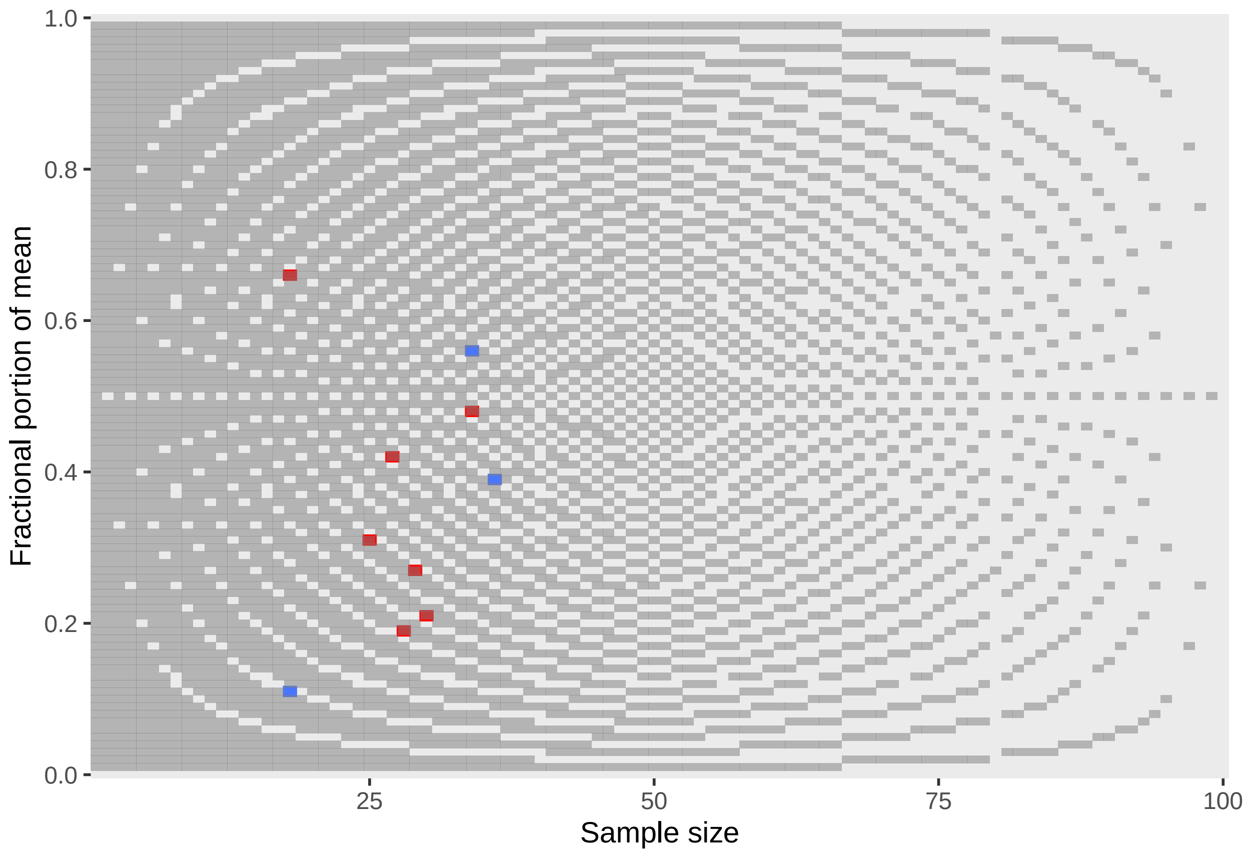
1. `incons_cases`: number of GRIM-inconsistent value sets.
2. `all_cases`: total number of value sets.
3. `incons_rate`: proportion of GRIM-inconsistent value sets.
4. `mean_ratio`: average of GRIM ratios.
5. `incons_base_ratio`: ratio of `incons_rate` to `mean_ratio`.
6. `testable_cases`: number of GRIM-testable value sets (i.e., those with a positive ratio).
7. `testable_rate`: proportion of GRIM-testable value sets.

Visualize results with `grim_plot()`

There is a specialized visualization function for GRIM test results, `grim_plot()`. Only run it on `grim_map()`'s output, and only if the `adjust_x` argument to `grim_map()` was set to `1`. It will fail otherwise.

```
pigs5 <- tribble(
  ~x,        ~n,
  "7.19",    28,
  "4.56",    34,
  "0.42",    27,
  "1.31",    25,
  "3.48",    34,
  "4.27",    29,
  "6.21",    30,
  "3.11",    18,
  "5.39",    36,
  "5.66",    18,
)


pigs5 %>%
  grim_map(adjust_x = 1) %>%
  grim_plot()
```



The sheer optics of this plot will probably not fit everyone's taste. However, it is based on the laws governing GRIM.[1] The background grid shows all consistent (light) and inconsistent (dark) value pairs for two decimal places. The empirical values are shown in blue if consistent and red if inconsistent. Color settings and other ggplot2-typical options are available via arguments. For more, see `grim_plot()`'s documentation.

GRIM statistics

The GRIM ratio

Formula

The final column in a tibble returned by `grim_map()` is what I call the "GRIM ratio", i.e.:

$$\frac{10^D - NL}{10^D}$$

where D is the number of decimal places in `x` (the mean or proportion), N is the sample size, and L is the number of scale items.

Function

`grim_ratio()` takes the arguments `x`, `n`, `items`, and `percent` as in `grim()` and `grim_map()`:

[1]The plot shows some leniency at $N = 40$ and $N = 80$, which is due to permissive rounding: Fewer cases are flagged as inconsistent than might be warranted. This pattern generalizes to $N = 4(10^{D-1})$ and $N = 8(10^{D-1})$, where D is the number of decimal places in the fractional portions. Leniency is in line with scrutiny's philosophy of erring on the side of caution. However, `grim_plot()` only reveals this feature or bug rather than causing it.

```
grim_ratio(x = 1.42, n = 72)
#> [1] 0.28

grim_ratio(x = 5.93, n = 80, items = 3)
#> [1] -1.4

grim_ratio(x = "84.20", n = 40, percent = TRUE)  # Enter `x` as string to preserve trailing zero
#> [1] 0.996
```

In addition, `grim_total()` takes the same arguments but returns only the numerator of the above formula:

```
grim_total(x = 1.42, n = 72)
#> [1] 28

grim_total(x = 5.93, n = 80, items = 3)
#> [1] -140

grim_total(x = "84.20", n = 40, percent = TRUE)  # Enter `x` as string to preserve trailing zero
#> [1] 9960
```

Interpretation

If the GRIM ratio is between 0 and 1 (inclusive), it can be interpreted as the proportion of inconsistent value sets corresponding to a given set of parameters. If the ratio is negative, the proportion is zero; if it is greater than 1, the proportion is 1.

Similarly, if the `grim_total()` value is non-negative, it can be interpreted as the absolute total number of GRIM inconsistencies corresponding to a given set of parameters. If it is negative, that number is zero.

Non-originality

Although the term "GRIM ratio" is new, the formula is arguably implicit in Brown and Heathers' (2017) paper on GRIM. The numerator is a transformation of the formula presented on p. 364, and the authors discuss a common special case of the ratio on p. 367:

> With reporting to two decimal places, for a sample size $N < 100$ [and a single item], a random mean value will be consistent in approximately N% of cases.

Thus, all I did here was to make the formula explicit and give it a name. Researchers may judge for themselves how useful it is for further analyses.

Granularity and items

A distribution's granularity is the minimal amount by which two means or proportions of a non-continuous distribution can differ. It is derived from the sample size and the number of scale items. Conversely, the number of items naturally follows from the distribution's sample size and granularity.

Formulas

The granularity (G) formula is

$$G = \frac{1}{NL}$$

where N is the sample size and L is the number of items.

The scale items formula is the converse:

$$L = \frac{1}{NG}$$

Functions

Suppose you have an ordinal distribution with 80 observations and five items. To get its granularity, run this:

```
grim_granularity(n = 80, items = 4)
#> [1] 0.003125
```

Now, imagine a distribution with 50 observations and a granularity of 0.01. To get the number of its items, use this code:

```
grim_items(n = 50, gran = 0.01)
#> [1] 2
```

As the number of items itself has a granularity of 1, a call to `grim_items()` that doesn't return whole numbers indicates a problem in earlier computations. A warning to that effect will be displayed:

```
grim_items(n = c(65, 93), gran = 0.02)
#> Warning: 2 items aren't whole numbers
#> > This concerns 0.7692308 and 0.5376344.
#> ! Item numbers have a granularity of 1, so they should be
#>   whole numbers. If they aren't, there must be some
#>   problem.
#> [1] 0.7692308 0.5376344
```