




Vazul: An R Package for Analysis Blinding

Tamás Nagy 
ELTE Eotvos Lorend University

Alexandra Sarafoglou 
University of Amsterdam

Abstract

The abstract of the article.

Keywords: keywords, not capitalized, Java.

1. Functionality of the vazul package

The package provides functions for blinding data to facilitate unbiased analysis. The package covers two main approaches to data blinding:

1. **Masking:** Replaces original values with anonymous labels, hiding the original information.
2. **Scrambling:** Randomizes the order of existing values while preserving all original data content.

Each approach is available at three levels:

- **Vector level:** `mask_labels()` and `scramble_values()` - operate on single vectors
- **Data frame level:** `mask_variables()` and `scramble_variables()` - operate on columns in a data frame
- **Row-wise level:** `mask_variables_rowwise()` and `scramble_variables_rowwise()` - operate within rows across columns

```
R> library(vazul)
```

```
R> library(dplyr)
```

1.1. Masking Functions

Masking functions replace categorical values with anonymous labels. This is useful when you want to completely hide the original information, such as treatment conditions or group assignments. Masking is best suited for categorical variables with a limited number of unique values. Therefore, the function requires character or factor vectors as input.

The `mask_labels()` function takes a character or factor vector and replaces each unique value with a randomly assigned masked label.

```
R> # Create a simple treatment vector
R> treatment <- c("control", "treatment", "control", "treatment", "control")
R>
R> # Mask the labels
R> set.seed(123)
R> masked_treatment <- mask_labels(treatment)
R> masked_treatment
```

```
      control      treatment      control      treatment
"masked_group_01" "masked_group_02" "masked_group_01" "masked_group_02"
      control
"masked_group_01"
```

Notice that each unique value receives a unique masked label, the same original value always maps to the same masked label, and assignment of masked labels to original values is randomized. The function preserves factor structure when the input is a factor.

You can customize the prefix used for masked labels:

```
R> set.seed(456)
R> mask_labels(treatment, prefix = "group_")
```

```
      control      treatment      control      treatment      control
"group_01" "group_02" "group_01" "group_02" "group_01"
```

The functionality of `mask_labels()` is extended to data frames in the `mask_variables()` function, which applies masking to multiple columns in a data frame simultaneously. By default, each column gets its own set of masked labels with the column name as prefix.

```
R> df <- data.frame(
+   treatment = c("control", "intervention", "control", "intervention"),
+   outcome = c("success", "failure", "success", "failure"),
+   score = c(85, 92, 78, 88)
+ )
R>
R> set.seed(123)
R> result <- mask_variables(df, c("treatment", "outcome"))
R> result
```

	treatment	outcome	score
1	treatment_group_01	outcome_group_01	85
2	treatment_group_02	outcome_group_02	92
3	treatment_group_01	outcome_group_01	78
4	treatment_group_02	outcome_group_02	88

When `across_variables = TRUE`, all selected columns share the same mapping:

```
R> df2 <- data.frame(
+   pre_condition = c("A", "B", "C", "A"),
+   post_condition = c("B", "A", "A", "C"),
+   score = c(1, 2, 3, 4)
+ )
R>
R> set.seed(456)
R> result_shared <- mask_variables(df2, c("pre_condition", "post_condition"),
+                                   across_variables = TRUE)
R> result_shared
```

	pre_condition	post_condition	score
1	masked_group_01	masked_group_03	1
2	masked_group_03	masked_group_01	2
3	masked_group_02	masked_group_01	3
4	masked_group_01	masked_group_02	4

You can use `tidyselect` helpers, such as `starts_with()`, `any_of()`, etc. to select columns.

```
R> set.seed(789)
R> mask_variables(df, where(is.character))
```

	treatment	outcome	score
1	treatment_group_01	outcome_group_02	85
2	treatment_group_02	outcome_group_01	92
3	treatment_group_01	outcome_group_02	78
4	treatment_group_02	outcome_group_01	88

The `mask_variables_rowwise()` function applies consistent masking within each row across multiple columns. This is useful when you have multiple independent variables that require concealment.

```
R> df <- data.frame(
+   treat_1 = c("control", "treatment", "placebo"),
+   treat_2 = c("treatment", "placebo", "control"),
+   treat_3 = c("placebo", "control", "treatment"),
+   id = 1:3
+ )
```

```
R>
R> set.seed(123)
R> result <- mask_variables_rowwise(df, starts_with("treat_"))
R> result
```

```
      treat_1      treat_2      treat_3 id
1 masked_group_03 masked_group_01 masked_group_02 1
2 masked_group_01 masked_group_02 masked_group_03 2
3 masked_group_02 masked_group_03 masked_group_01 3
```

Within each row, the original values are consistently mapped to masked labels, but the mapping is independent across rows.

1.2. Scrambling Functions

Scrambling functions randomize the order of values while preserving all original data content. This approach maintains the data distribution while breaking the connection between observations and their original values. The `scramble_values()` function randomly reorders the elements of a vector. It can be used to any type of data, including numeric, character, and factor, with any number of unique values. The function preserves the data type.

```
R> # Numeric data
R> set.seed(123)
R> numbers <- 1:10
R> scramble_values(numbers)
```

```
[1] 3 10 2 8 6 9 1 7 5 4
```

Extending the scrambling functionality to data frames, `scramble_variables()` scrambles the values of specified columns. By default, each column is scrambled independently. The function can also use tidyselect helpers to select columns.

```
R> df <- data.frame(
+   x = 1:6,
+   y = letters[1:6],
+   group = c("A", "A", "A", "B", "B", "B")
+ )
R>
R> set.seed(123)
R> scramble_variables(df, c("x", "y"))
```

```
  x y group
1 3 e     A
2 6 d     A
3 2 b     A
4 4 f     B
5 5 a     B
6 1 c     B
```

When `together = TRUE`, the selected columns are scrambled as a unit, preserving row-level relationships.

```
R> set.seed(456)
R> scramble_variables(df, c("x", "y"), together = TRUE)
```

	x	y	group
1	5	e	A
2	6	f	A
3	3	c	A
4	2	b	B
5	1	a	B
6	4	d	B

Use the `.groups` parameter to scramble within groups:

```
R> data(williams)
R>
R> # Scramble age and ecology within gender groups
R> set.seed(42)
R> williams_scrambled <- williams |>
+   scramble_variables(c("age", "ecology"), .groups = "gender")
```

The `scramble_variables_rowwise()` function scrambles values within each row across specified columns. This is useful for scrambling repeated measures or item responses. Within each row, the values are shuffled among the item columns.

```
R> df <- data.frame(
+   item1 = c(1, 4, 7),
+   item2 = c(2, 5, 8),
+   item3 = c(3, 6, 9),
+   id = 1:3
+ )
R>
R> set.seed(123)
R> result <- scramble_variables_rowwise(df, c("item1", "item2", "item3"))
R> result
```

	item1	item2	item3	id
1	3	1	2	1
2	5	4	6	2
3	8	9	7	3

You can scramble multiple sets of columns independently:

```

R> df2 <- data.frame(
+   day_1 = c(1, 4, 7),
+   day_2 = c(2, 5, 8),
+   day_3 = c(3, 6, 9),
+   score_a = c(10, 40, 70),
+   score_b = c(20, 50, 80),
+   id = 1:3
+ )
R>
R> set.seed(456)
R> result2 <- scramble_variables_rowwise(
+   df2,
+   starts_with("day_"),
+   c("score_a", "score_b")
+ )
R> result2

```

```

  day_1 day_2 day_3 score_a score_b id
1     1     3     2     10     20  1
2     5     4     6     40     50  2
3     9     7     8     80     70  3

```

1.3. Choosing Between Masking and Scrambling

Aspect	Masking	Scrambling
Original values	Hidden (replaced)	Preserved (reordered)
Distribution	Same proportions, but new labels	Unchanged
Best for	Categorical variables	Numeric or categorical
Use case	Hide treatment conditions	Break individual links
Reversibility	Requires mapping key	Irreversible

When to Use Masking

- When you need to hide categorical labels (e.g., treatment conditions, group names)
- When analysts should not know the meaning of categories
- When you want different prefixes for different variables

When to Use Scrambling

- When you want to preserve the original data distribution
- When you need to break the link between observations and values
- When working with numeric data or many unique categorical levels in a variable

1.4. Working with Included Datasets

The `vazul` package includes two research datasets for demonstration and practice. The `marp` dataset comes from the Many Analysts Religion Project (MARP) dataset contains 10,535 participants from 24 countries, and includes variables about religious beliefs, well-being, and demographics. The `williams` dataset comes from a stereotyping study with 112 participants, including measures of impulsivity, opportunity, and investment in education and children.

1.5. Functions

The `vazul` package provides a comprehensive toolkit for data blinding:

Function	Level	Purpose
<code>mask_labels()</code>	Vector	Replace categorical values with anonymous labels
<code>mask_variables()</code>	Data frame	Mask multiple columns
<code>mask_variables_rowwise()</code>	Row-wise	Consistent masking within rows
<code>scramble_values()</code>	Vector	Randomize value order
<code>scramble_variables()</code>	Data frame	Scramble multiple columns
<code>scramble_variables_rowwise()</code>	Row-wise	Scramble values within rows
<code>marp</code>	Dataset	Many Analysts Religion Project data
<code>williams</code>	Dataset	Williams stereotyping study data

Affiliation:

Tamás Nagy
 ELTE Eotvos Lorend University
 Institute of Psychology,
 ELTE Eotvos Lorend University,
 Budapest, Hungary
 E-mail: nagy.tamas@ppk.elte.hu

Alexandra Sarafoglou
University of Amsterdam
Department of Psychology,
University of Amsterdam,
Amsterdam, The Netherlands
E-mail: a.s.g.sarafoglou@uva.nl