Journal of Statistical Software

March 2025, Volume 112, Issue 4. *doi: 10.18637/jss.v112.i04*

Split-Apply-Combine with Dynamic Grouping

Mark P. J. van der Loo
Statistics Netherlands
Leiden University

Abstract

Partitioning a data set by one or more of its attributes and computing an aggregate for each part is one of the most common operations in data analyses. There are use cases where the partitioning is determined dynamically by collapsing smaller subsets into larger ones, to ensure sufficient support for the computed aggregate. These use cases are not supported by software implementing split-apply-combine types of operations. This paper presents the R package **accumulate** that offers convenient interfaces for defining grouped aggregation where the grouping itself is dynamically determined, based on user-defined conditions on subsets, and a user-defined subset collapsing scheme. The formal underlying algorithm is described and analyzed as well.

Keywords: data analysis, estimation, aggregation, R.

1. Introduction

The operation of splitting a data set into non-overlapping groups, computing an aggregate for each group, and combining the results into a new dataset is one of the most common operations in data analyses. Indeed, any software for data analyses includes some functionality for this. For example, the combination of `split`/`lapply`/`unsplit` as well as `aggregate` have been a part of the S (Becker, Chambers, and Wilks 1988) and R (R Core Team 2024) languages for a long time. For R there are several packages that implement functionality for this, including **plyr** (Wickham 2011), it's successor **dplyr** (Wickham, François, Henry, and Müller 2024), it's drop-in replacement **poorman** (Eastwood 2023), and performance-focused R packages **collapse** (Krantz 2024) and **data.table** (Dowle and Srinivasan 2024). In Python (Van Rossum *et al.* 2011) the `pandas` package implements several methods for grouping records and aggregating over one or more columns in data frame objects. The more recent Polars (2024) library for Python and Rust (Matsakis and Klock 2014) also implement such features. Similarly, the **DataFrames** package for julia implements split-apply-combine functionality (Bouchet-Valat and Kamiński 2023).

In all packages mentioned, the calculation for each group uses data available within the group. However, there are valid use cases where a group aggregate is determined using attributes from out-of-group entities. One example where this occurs is in the area of Small Area Estimation (SAE, see e.g., Rao and Molina 2015; Molina and Marhuenda 2015). Here, one wishes to estimate an aggregate for a group, for example a geographical region, or a detailed population subset, where the number of (sampled) observations is so small that the variance of the estimate would be unacceptably large. In small area estimation (SAE) one trades bias for variance by 'borrowing statistical strength' from out-of-group records. The out-of-group records can be obtained, for example by combining the original small group with a group of records that are deemed similar in certain respects. A second area where out-of-group records play a role is in certain hot-deck imputation methods (Andridge and Little 2010). In k-nearest-neighbors imputation for example, one finds a set of k donor records that are preferably in the same group, but this condition may be relaxed if there are not enough records in the group. In the **VIM** package for R (Kowarik and Templ 2016), this is controlled by a combination of the Gower distance and setting conditions on minimal number of donors. In practice, imputation is often performed via a fall-though scenario, where one first tries to estimate a model within a group, but if the group is too small for a reliable estimate of model parameters, the group is enlarged by combining similar groups, similar to the small-area estimation scenario.

SAE as a special case is well supported by R and other free software. Methodology has for example been implemented in the **sae** package of Molina and Marhuenda (2015) and the **hbsae** package of Boonstra (2022). Regarding imputation methodology, the Comprehensive R Archive Network (CRAN) task view on missing data (Josse, Mayer, Tierney, and Vialaneix 2024) currently lists 203 R packages that support some form of estimating missing data. The **simputation** package (Van der Loo 2022) seems to be the only one that allows for some kind of fall-through scenario for selecting methods, but it does not allow for dynamic grouping.

Summarizing, we see that on one hand there are many implementations available for generic aggregation based on fixed groups. On the other hand there are domain-specific implementation for methods where dynamic grouping of a set of records plays a role. This paper presents a generic solution to split-apply-combine aggregation where groups can be collapsed dynamically in the form of R package **accumulate** (Van der Loo 2025). Package **accumulate** is available from CRAN at `https://CRAN.R-project.org/package=accumulate`.

The `accumulate` package serves the use case where a user wishes to compute aggregates for a certain grouping of records. However, if a certain instance of a group does not meet user-defined quality requirements the set of records is expanded by (recursively) collapsing the grouping according to a user-defined scheme. For example, given some financial data on companies, one wishes to compute the average profit to turnover ratio for each combination of economic activity and size class. If for a certain combination of economic activity and size class there are too few records for a reliable estimate, one could drop size class and compute the average ratio over all records within a certain economic activity. Or, one could choose to coarse-grain economic activities by collapsing groups with activities that are deemed similar enough.

The package has been developed with the following design choices in mind. First, the interface should be easy to learn for R users, and thus should resemble existing popular interfaces where possible. Second, users should be free to define any, possibly multi-step, collapsing scheme. Here, we keep in mind that collapsing schemes may be constructed manually based

on domain knowledge and that users may want to experiment with several schemes before deciding on a final solution. This calls for a certain separation of concerns between defining collapsing schemes and applying them to data. The package should also support collapsing schemes that follow naturally from hierarchical classification systems. Third, users should have the flexibility to define any quality requirement on the grouping while common quality requirements are supported out of the box. Common quality requirements include a minimum number of records, or a minimum fraction of records without missing values, or a minimum number of records with non-zero values. Finally, the package should support simple outputs such as counts or averages, but also compound objects such as the output of model estimates.

The rest of this paper is organized as follows. In the next section we visually explain dynamic grouping via a collapsing scheme and introduce the running example that will be used in Section 3 which introduces the **accumulate** package, the main interfaces and helper functions. In Section 3 we also discuss the common case of collapsing via a predefined hierarchical classification scheme. Section 4 demonstrates the package on a realistic synthetic dataset. The case of complex outputs is also demonstrated in this Section. In Section 5 we derive the pseudocode that solves the problem of aggregating with dynamically collapsing groups. We show that it is a precise and straightforward generalization of the standard split-apply-combine problem and analyze its time complexity. A summary and conclusion follows in Section 6.

2. Dynamic grouping

In this Section the illustrate the concept of dynamic grouping with a minimal worked example. This example is not very realistic, but it is constructed to be simple enough so the whole procedure can be followed in detail.

We consider a data set with three categorical variables A, B and B_1, and one numerical variable Y. Variable A has levels $\{1, 2, 3\}$ and variable B is a hierarchical classification with levels $\{11, 12, 13, 21, 22\}$. Variable B_1 is a coarsened version of B: For each record the value for B_1 is the first digit of B. Hence, B_1 has levels $\{1, 2\}$.

Our goal is to compute the mean μ_Y over Y, grouped by $A \times B$. We impose the condition that there must be at least three records in each group. If a certain group $(a \in A, b \in B)$ has less than three records, we attempt to compute the value for that group over records in (a, b_1) where b_1 is obtained by taking the first digit of b. If we then still have less than three records, we take records of group a to determine the value for (a, b).

The tables in Figure 2 illustrate the idea. The left table represents the data set to be aggregated by A and B. The table on the right represents the output. Colors indicate which data was used.

The First row in the output represents group $(A = 1, B = 11)$. The collapsing level is zero, which means that no collapsing was necessary. Indeed, in the data table we see that there are three rows with $A = 1$ and $B = 11$ with Y values 1, 2, and 3, resulting in $\mu_Y = (1+2+3)/3 = 2$ for this group.

Next, we try to compute the total for group $(A = 2, B = 12)$ (in green) but find that there are only two such rows. We now define a new group $(A = 2, B_1 = 1)$ and find that there are three records in that group so we get $\mu_Y = (4 + 5 + 6)/3 = 5$. Similarly, there is only one record with $(A = 2, B = 13)$. Collapsing groups to $(A = 2, B_1 = 1)$, yields again $\mu_y = 5$.

Input Data

A	B	B_1	Y
1	11	1	1
1	11	1	2
1	11	1	3
2	12	1	4
2	12	1	5
2	13	1	6
3	21	2	7
3	22	2	8
3	12	1	9

Output Aggregates

$A \times B$	`Level`	μ_Y
1 11	0	2
2 12	1	5
2 13	1	5
3 21	2	8
3 22	2	8
3 12	2	8

Figure 1: Input data (left) with grouping variables A, B and B_1, and means of Y per $A \times B$ after dynamic grouping (right).

Finally, for $(A = 2, B = 21)$ there is only a single record. Collapsing to $(A = 2, B_1 = 2)$ yields only two records, so we need to collapse further to $(A = 2)$ and finally obtain three records. This yields $\mu_Y = (7+8+9)/3 = 8$. Similarly the groups $(A = 3, B = 22)$ and $(A = 3, B = 12)$ are collapsed to $(A = 2)$.

3. R Package accumulate

Grouped aggregation with a fall-though scenario based on a collapsing scheme requires a fair amount of specification by the user. Besides the data to be aggregated, one needs to specify the method(s) of aggregation, the collapsing scheme, and the condition to decide whether a subset is fit for aggregation or a next collapse is necessary. There are two main functions in **accumulate** that offer slightly different interfaces.

```
accumulate(data, collapse, test, fun, ...)
cumulate(data, collapse, test, ...)
```

Here `data` is a data frame holding data to be aggregated; `collapse` represents the collapse sequence (as a '`formula`' or a '`data frame`'), and `test` is a function that accepts a subset of `data` and returns a Boolean that indicates whether a subset is suited for aggregation or not. In `accumulate()`, the parameter `fun` represents an aggregation function that is applied to every column of `data`, and the ellipsis (`...`) is for arguments that are passed as extra argument to `fun`. The interface of `accumulate()` is somewhat similar to that of the `aggregate()` function in R. In `cumulate()`, the ellipsis is a sequence of comma-separated `name = expression` pairs in the style of `summarize()` from the `dplyr` package.

The output of both functions are of the same form. The columns of the output data frame and can schematically be represented as follows.

```
[Grouping Variables, Collapse level, Output aggregates]
```

The first columns represent the variables that define the output grouping, the next column is an integer that indicates the level of collapsing used to compute the aggregate (0 indicating no collapse), and the last set of columns store the aggregates. Output aggregates may be of

a simple data type (`numeric`, `character`, `logical`,...) or of a composed type such as the output of a linear model. The latter case is demonstrated in Section 4.4.

Both functions support two different interfaces for specifying collapsing schemes through the `collapse` parameter. The first and most general is the '`formula`' interface, which requires that the collapsing sequence is represented as variables in the data set to be aggregated. The second is a tabular interface where each row starts with the one of the lowest-level group labels, and subsequent columns contain labels of courser groups.

3.1. The formula interface

We will use the example of Section 2 to illustrate the '`formula`' interface.

```
R> library("accumulate")
R> input <- data.frame(
+    A  = c(1, 1, 1, 2, 2, 2, 3, 3, 3),
+    B  = c(11, 11, 11, 12, 12, 13, 21, 22, 12),
+    B1 = c(1, 1, 1, 1, 1, 1, 2, 2, 1),
+    Y  = 1:9)
R> cumulate(input, collapse = A * B ~ A * B1 + A,
+    test = function(d) nrow(d) >= 3, muY = mean(Y))

  A  B level muY
1 1 11     0   2
4 2 12     1   5
6 2 13     1   5
7 3 21     2   8
8 3 22     2   8
9 3 12     2   8
```

Consider the formula `A * B ~ A * B1 + A` in the call to `cumulate()`. The left-hand-side `A * B` is the target output grouping. The right-hand-side is to be interpreted as the collapsing sequence: If an instance of `A * B` does not pass the test, then collapse to `A * B1`, and if that does not pass the test collapse to `A`. If this final grouping also does not pass the test, the result is `NA`.

Summarizing, the '`formula`' interface is always of the following form.

```
Target grouping ~ Alternative1 + Alternative2 + ... + AlternativeN
```

It is possible to get the same result with `accumulate()`. This will cause summation over all variables that are not used in the formula object. In the below example we also introduce the helper function `min_records()`.

```
R> input$Y2 <- 11:19
R> accumulate(input, collapse = A * B ~ A * B1 + A,
+    test = min_records(3), fun = mean)

  A  B level Y Y2
1 1 11     0 2 12
```

```
4 2 12     1 5 15
6 2 13     1 5 15
7 3 21     2 8 18
8 3 22     2 8 18
9 3 12     2 8 18
```

This means that for experimentation, users must be careful to exclude categorical variables that are not used from the input data set. For example, if `B1` is not used, we get the following.

```
R> accumulate(input[-3], collapse = A * B ~ A,
+    test = min_records(3), fun = mean)

  A  B level Y Y2
1 1 11     0 2 12
4 2 12     1 5 15
6 2 13     1 5 15
7 3 21     1 8 18
8 3 22     1 8 18
9 3 12     1 8 18
```

The '`formula`' interface allows users to quickly experiment with different collapsing schemes. It does require that all categorical variables have been added to the data. As an alternative, one can use the data frame specification, which allows for a further separation between defining the collapsing scheme and the actual data processing.

3.2. The data frame interface

The data frame interface is somewhat limited because it only allows for a single grouping variable. The advantage however setting up a collapsing scheme in the form of a table closely connects to domain knowledge and allows fine-grained control on how groups are collapsed.

In order to use the data frame interface, the input dataset must include the most fine-grained grouping variable. In the running example this is $A \times B$, so we need to combine that into a single variable, and remove the other ones.

```
R> input1 <- input
R> input1$AB <- paste(input$A, input$B, sep = "-")
R> input1 <- input1[-(1:3)]
R> input1

  Y Y2   AB
1 1 11 1-11
2 2 12 1-11
3 3 13 1-11
4 4 14 2-12
5 5 15 2-12
6 6 16 2-13
7 7 17 3-21
```

```
8 8 18 3-22
9 9 19 3-12
```

We now define the collapsing scheme as follows.

```
R> csh <- data.frame(
+    AB  = c("1-11", "2-12", "2-13", "3-21", "3-22", "3-12"),
+    AB1 = c("1-1", "2-1", "2-1", "3-2", "3-2", "3-1"),
+    A   = c("1", "2", "2", "3", "3", "3"))
R> csh

    AB AB1 A
1 1-11 1-1 1
2 2-12 2-1 2
3 2-13 2-1 2
4 3-21 3-2 3
5 3-22 3-2 3
6 3-12 3-1 3
```

In this data frame, two consecutive columns should be read as a child-parent relation. For example, in the first collapsing step the groups defined by `AB == "2-12"` and `AB == "2-13"` both collapse to `AB1 == "2-1"`. In this artificial example the codes do not mean anything, but in realistic cases where codes represent a (hierarchical) classification, domain experts usually have a good grasp of which codes can be combined.

The calls to `cumulate()` and `accumulate()` now look as follows.

```
R> accumulate(input1, collapse = csh, test = min_records(3), mean)

    AB level Y Y2
1 1-11     0 2 12
4 2-12     1 5 15
6 2-13     1 5 15
7 3-21     2 8 18
8 3-22     2 8 18
9 3-12     2 8 18
```

```
R> cumulate(input1, collapse = csh, test = min_records(3),
+    muY = mean(Y), muY2 = mean(Y2))

    AB level muY muY2
1 1-11     0   2   12
4 2-12     1   5   15
6 2-13     1   5   15
7 3-21     2   8   18
8 3-22     2   8   18
9 3-12     2   8   18
```

3.3. Specifying tests

The `test` parameter of `cumulate()` and `accumulate()` accepts a function that takes a subset of the data and returns a Boolean. For common test conditions, including requiring a minimal number of records, or a minimal number or fraction of complete records there are helper functions available.

`min_records(n)`	At least n records.
`min_complete(n, vars)`	At least n records complete for variables `vars`.
`frac_complete(r, vars)`	At least $100r\%$ complete records for variables `vars`.
`from_validator(v, ...)`	Construct a testing function from a `validator` object of R package **validate**.

Second, for multiple, possibly complex requirements on variables users can express conditions with the **validate** package Van der Loo and De Jonge (2021). The **validate** packages offers a domain-specific language for expressing, manipulating, and investigating conditions on datasets. It's core concept is a list of 'data validation rules' stored as a '`validator`' object. A '`validator`' object is constructed with the eponymous function `validator()`. For example, to demand that there are at least 3 rows in a group, and that there are at least three records where $Y \geq 2$ we create the following rule set.

```
R> library("validate")
R> rules <- validator(nrow(.) >= 3, sum(Y >= 2) >= 3)
R> rules

Object of class 'validator' with 2 elements:
 V1: nrow(.) >= 3
 V2: sum(Y >= 2) >= 3
```

Here the `.` refers to the dataset as a whole, while rules that can be evaluated within the dataset can be written as Boolean R expressions.

We will apply these conditions to the `input` dataset that was constructed in Section 3.1. As a reminder we print the first few records.

```
R> head(input, 4)

  A  B B1 Y Y2
1 1 11  1 1 11
2 1 11  1 2 12
3 1 11  1 3 13
4 2 12  1 4 14
```

We use `A * B ~ A * B1 + B1` as collapsing scheme. The function `from_validator` passes the requirements as a test function to `accumulate()` (or `cumulate()`).

```
R> accumulate(input, collapse = A * B ~ A * B1 + B1,
+    test = from_validator(rules), fun = mean)
```

```
  A  B level        Y       Y2
1 1 11     2 4.285714 14.28571
4 2 12     1 5.000000 15.00000
6 2 13     1 5.000000 15.00000
7 3 21    NA       NA       NA
8 3 22    NA       NA       NA
9 3 12     2 4.285714 14.28571
```

Note that for target groups $(A = 3, B = 21)$ and $(A = 3, B = 22)$ none of the available collapsing levels lead to a group that satisfied all conditions. Therefore the collapsing level and output variables are all missing (`NA`).

The third and most flexible way for users to express tests is to write a custom testing function. The requirements are that it must work on any subset of a data frame, including a dataset with zero rows. The previous example can thus also be expressed as follows.

```
R> my_test <- function(d) nrow(d) >= 3 && sum(d$Y >= 2) >= 3
R> accumulate(input, collapse = A * B ~ A * B1 + B1,
+    test = my_test, fun = mean)
```

```
  A  B level        Y       Y2
1 1 11     2 4.285714 14.28571
4 2 12     1 5.000000 15.00000
6 2 13     1 5.000000 15.00000
7 3 21    NA       NA       NA
8 3 22    NA       NA       NA
9 3 12     2 4.285714 14.28571
```

It is easy to overlook some edge cases when specifying test functions. Recall that a test function is required to return `TRUE` or `FALSE`, regardless of the data circumstances. The only thing that a test function can assume is that the received data set is a subset of records from the dataset to be aggregated. As a service to the user, **accumulate** exports a function that checks a test function against common edge cases, including the occurrence of missing values, a dataset with zero rows, and the full dataset. The function `smoke_test()` checks whether the output is `TRUE` or `FALSE` under all circumstances and also reports errors, warnings, and messages. It accepts a (realistic) dataset and a testing function and prints test results to the console.

```
R> smoke_test(input, my_test)
```

```
Test with full dataset and Y is NA for all records raised issues.

   NA detected in output (must be TRUE or FALSE)
```

By default only failing tests are printed. In this case our test function is not robust against missing values for Y. This can be remedied by passing `na.rm = TRUE` as parameter to `sum()` in the test function.

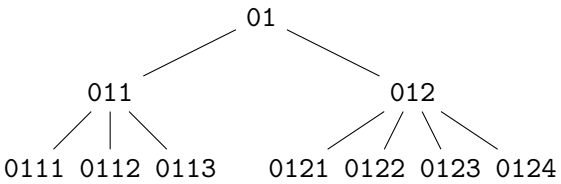
```
R> my_test1 <- function(d) nrow(d) >= 3 && sum(d$Y >= 2, na.rm = TRUE) >= 3
R> smoke_test(input, my_test1)
```

The smoke test is aimed at preventing complicated stack traces when errors occur in a call to `accumulate()` or `cumulate()`. Users should be aware that it does not guarantee correctness of the results, only robustness against certain edge cases.

3.4. Balanced and unbalanced hierarchical classifications

Hierarchical classifications are abundant in (official) statistics. They represent a classification of entities into non-overlapping, nested groupings. Examples include the international standard industrial classification of economic activities (ISIC, United Nations Statistical Division 2008), the related statistical classification of economic activities in Europe (NACE, Council of European Union 2006) and the European skills, competences, qualifications and occupations classification (ESCO, European Commision 2022).

Hierarchical classifications offer a natural mechanism for collapsing fine-grained groupings into larger groups because of the parent-child relationships. As an example consider a small piece of the NACE classification.



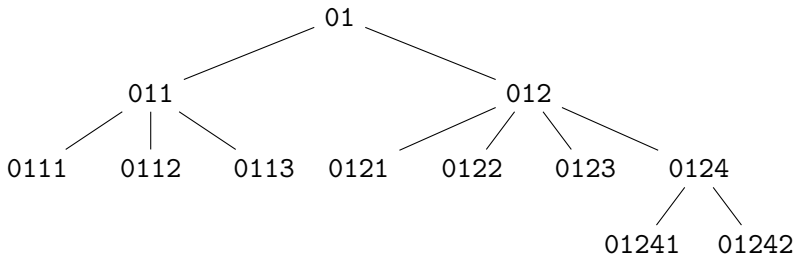
Here, the hierarchy suggests to collapse $\{0111, 0112, 0113\}$ into $\{011\}$ when needed, and similarly for the right branch. The second level of collapsing would combine $\{012\}$ with $\{011\}$ into $\{01\}$. The **accumulate** package comes with a helper function that creates the collapsing scheme from the lowest-level digits.

```
R> nace <- c("0111", "0112", "0113", "0121", "0121", "0122", "0123", "0124")
R> csh_from_digits(nace, levels = 2)
```

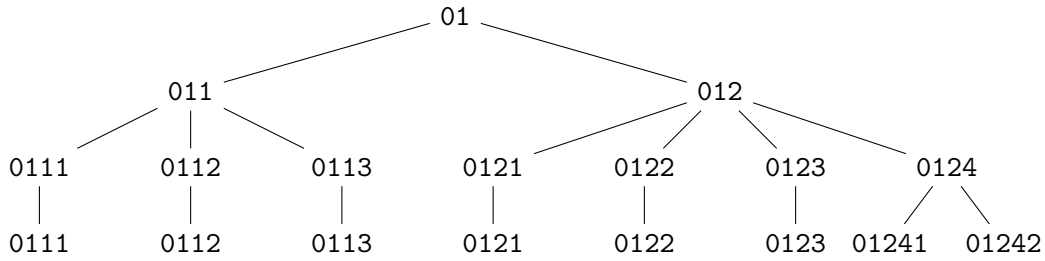
```
    A0  A1 A2
1 0111 011 01
2 0112 011 01
3 0113 011 01
4 0121 012 01
5 0121 012 01
6 0122 012 01
7 0123 012 01
8 0124 012 01
```

Here, the parameter `levels` determines how many collapsing steps will be computed. Since all codes are prepended with zero, there is no need to collapse 01 any further. The output can be used as argument to the `collapse` parameter of the `accumulate()` or `cumulate()` functions.

The situation becomes a little more involved when hierarchical classifications form a tree such that the distance from leave to trunk is not the same for all leaves (unbalanced tree). This occurs in practice, for example when local organizations create an extra level of detail for some, but not all leaves. Below is an example of such a situation.



In this case, not all leaves can be collapsed with the same number of collapsing levels. This provides an issue for specifying the collapsing sequence as it now depends on the leaf where you start how many collapsing levels are possible. It also complicates interpretability of the result as the collapsing level reported in the output, now means different things for different target groups. The solution chosen in **accumulate** is to extend the tree by making copies of the leaves that are not on the lowest level as follows.



The trade off is that although there may be some extra calculations in the case where a leaf is collapsed to itself. The gain is that the specification of the calculation as well as the interpretation of the results are now uniform across all hierarchical classifications. Again, using `csh_from_digits()` deriving the collapsing scheme can be automated.

```
R> nace <- c("0111", "0112", "0113", "0121", "0122", "0123", "01241",
+    "01242")
R> csh_from_digits(nace, levels = 3)

     A0   A1  A2 A3
1  0111 0111 011 01
2  0112 0112 011 01
3  0113 0113 011 01
4  0121 0121 012 01
5  0122 0122 012 01
6  0123 0123 012 01
7 01241 0124 012 01
8 01242 0124 012 01
```

4. Extensive example: Economic data

In this Section we discuss three practical examples using a synthetic dataset included with the package.

```
R> data("producers")
R> head(producers)
```

```
   sbi size industrial trade other other_income  total
1 3410    8     151722  2135     0        -1775 152082
2 2840    7      50816    NA   158          949  59876
3 2752    5       4336    NA     0           36   4959
4 3120    6      18508    NA     0           80  20682
5 2524    7      21071     0     0          442  21513
6 3410    6      24220  1069     0          239  25528
```

This `producers` dataset contains synthetic data records of various income sources for 1734 industrial produces. The records are classified into a local version of the NACE classification called `sbi` and into `size` classes with values in $\{5, 6, 7, 8, 9\}$.

4.1. Small area estimation

Small area estimation (SAE) is a collection of methods that are aimed at estimating sub-population parameters in cases where there the number of observations in a sub-population is so mall that direct estimation leads to unacceptable estimation variance. Instead one may resort for example to indirect estimation, meaning that one estimates parameters for a larger sub-population which are then used in the estimate for the target sub-population. Here, we shall be interested in estimating the average turnover from industrial activities (`industrial`) by SBI and size class.

In the simplest case, where no auxiliary information is available or used, one replaces the estimator of the mean over a sub-population with the estimator of the mean over a larger sub-population that includes the target sub-population (Rao and Molina 2015, Section 3.2.1). If we assume that the dataset is obtained by simple random sampling from the population, the mean can be estimated with the sample mean. In this example we will demand that there are at least ten records for which turnover has been measured. The collapsing scheme is given by `sbi * size ~ sbi + sbi2 + sbi1` where `sbi2` and `sbi1` are classification by respectively the first two SBI digits and the first SBI digit. We first add those variables to the dataset.

```
R> producers$sbi2 <- substr(producers$sbi, 1, 2)
R> producers$sbi1 <- substr(producers$sbi, 1, 1)
R> head(producers, 3)
```

```
   sbi size industrial trade other other_income  total sbi2 sbi1
1 3410    8     151722  2135     0        -1775 152082   34    3
2 2840    7      50816    NA   158          949  59876   28    2
3 2752    5       4336    NA     0           36   4959   27    2
```

Using `cumulate()` we obtain the means.

```
R> a <- cumulate(producers, collapse = sbi * size ~ sbi + sbi2 + sbi1,
+    test = min_complete(n = 10, vars = "industrial"),
+    mean_industrial = mean(industrial, na.rm = TRUE))
R> head(a,3)
```

```
   sbi size level mean_industrial
1 3410    8     2        96157.67
2 2840    7     0        23160.25
3 2752    5     2        56966.33
```

In terms of SAE, the collapsing scheme expresses the assumption that estimates on the level of respectively `sbi`, `sbi2` and `sbi1` introduce an acceptable bias.

4.2. Imputing missing values using SAE and ratio imputation

Our goal in this example is to impute missing values for the `industrial` variable based on ratio imputation with `total` as predictor. Ratio imputation is a method where the imputed value $\hat{y}_i$ for variable Y of record i is estimated as $\hat{y}_i = \hat{R}_d x_i$, where $\hat{R}_d$ is an estimate of the ratio between Y and an auxiliary variable X in sub-population d. An unbiased estimate for R_d is given by $\hat{\bar{Y}}_d / \hat{\bar{X}}_d$ where $\hat{\bar{Y}}_d$ and $\hat{\bar{X}}_d$ are estimated sub-population means.

We use SAE to estimate the sub-population ratios, and then use the `simputation` package (Van der Loo 2022) to impute the missing values.

```
R> r <- cumulate(producers, collapse = sbi * size ~ sbi + sbi2 + sbi1,
+    test = min_complete(n = 10, vars = "industrial"),
+    R = mean(industrial, na.rm = TRUE)/mean(total, na.rm = TRUE))
R> head(r,3)
```

```
   sbi size level         R
1 3410    8     2 0.7450223
2 2840    7     0 0.8972639
3 2752    5     2 0.8725726
```

To impute the values, using `impute_proxy()` we need to merge the ratios with the producers dataset (which automatically happens by SBI and size class).

```
R> library("simputation")
R> dat <- merge(producers, r)
R> dat <- impute_proxy(dat, industrial ~ R * total)
```

We can inspect the imputed values as follows.

```
R> iNA <- is.na(producers$industrial)
R> head(dat[iNA, c("sbi", "size", "level", "R", "industrial", "total")])
```

```
       sbi size level         R industrial total
274   1581    5     0 0.9653030        830  1005
348   1581    6     0 0.9214068       5600  5600
392   1582    6     0 0.9521066       7969  8028
664  21122    7     2 0.9076063      45282 46149
1333  2840    6     0 0.9384657      17797 18186
1364  2851    6     0 0.9843794       7958  8315
```

From this, we get all the information to interpret the imputed values. For example, we see that in record 664, the ratio was estimated after collapsing the group (`sbi`, `size`) = (21122, 7) to `sbi2` = 21 since the collapse `level` equals 2.

4.3. Random nearest neighbors imputation with collapsing groups

Nearest neighbor (NN) imputation is a donor imputation method where the imputation value is copied from a record that is (randomly) chosen from a donor pool (Andridge and Little 2010). In this example we use the grouping variables in `producers` to define the donor pools. To prevent the same donor from being used too often, it is not uncommon to demand a minimum number of records in the donor pool. A collapsing scheme is one way of guaranteeing this, and below we demonstrate how this problem can be expressed in `accumulate`.

We wish to impute the variable `trade` in the `producers` dataset using donor imputation, where donors come from the same (`sbi`, `size`) combination. We wish to sample donor values from a group of at least 5 donors. If this is not possible, we use the same fallback scenario as in the previous Section.

We first define an 'aggregation' function that takes a vector of donors and returns a non-empty sample.

```
R> random_element <- function(x) sample(x[!is.na(x)], 1)
```

We will use `cumulate()` to ensure that there are at least five non-empty values in `x` when `random_element()` is called. To make sure we obtain a donor for each record, we add an identifying column `id` to use as grouping variable.

```
R> producers <- cbind(id = sprintf("ID%03d", seq_len(nrow(producers))),
+    producers)
R> set.seed(111)
R> imputations <- cumulate(producers,
+    collapse = id ~ sbi * size + sbi + sbi2 + sbi1,
+    test = min_complete(5,"trade"), donor_trade = random_element(trade))
R> head(imputations, 3)

     id level donor_trade
1 ID001     3        2755
2 ID002     2        3293
3 ID003     3           0
```

To use the donor imputations, we merge the imputation candidates with the original dataset and use `impute_proxy()` of the **simputation** package for imputation.

```
R> imputed <- merge(producers, imputations) |>
+    impute_proxy(trade ~ donor_trade)
R> cols <- c(1:3, 9:10, 5)
R> head(producers[cols], 3)

     id  sbi size sbi2 sbi1 trade
1 ID001 3410    8   34    3  2135
2 ID002 2840    7   28    2    NA
3 ID003 2752    5   27    2    NA

R> head(imputed[c(cols, 11)], 3)

     id  sbi size sbi2 sbi1 trade level
1 ID001 3410    8   34    3  2135     3
2 ID002 2840    7   28    2  3293     2
3 ID003 2752    5   27    2     0     3
```

The merge operation automatically merges on the `id` column, which also adds the `level` column to the output. The function `impute_proxy` copies values from `donor_trade` into trade where `trade` is missing. In the last expressions we only print the columns of interest.

4.4. Computing complex aggregates

Until now, the aggregates have been simple (scalar) values. With the `cumulate()` function it is also possible to specify complex aggregates that go beyond simple aggregates. Below, we estimate the following linear model

$$\texttt{total} = \beta_0 + \beta_i \texttt{industrial} + \varepsilon,$$

demanding that there are at least 10 records where both predictor and predicted variable are available.

```
R> r <- cumulate(producers, collapse = sbi * size ~ sbi + sbi2 + sbi1,
+    test = min_complete(n = 10, vars = c("total","industrial")),
+    model = lm(total ~ industrial))
R> head(r,3)

   sbi size level model
1 3410    8     2  <lm>
2 2840    7     0  <lm>
3 2752    5     2  <lm>
```

Here, the last column is a list of class `object_list`, where each element is either an object of class `lm` or `NA`. Variables of class `object_list` only differ from from a standard R list by their print method.

5. Formal description and algorithms

In this Section we give a formal description of the algorithm for aggregation with dynamic grouping. We start by giving an algorithm for ordinary split-apply-combine to demonstrate how the algorithm must be generalized to allow for a collapsing scheme.

5.1. Split-apply-combine

To analyze a data set group by group we need to specify a data set, a way to split it into groups, and a function that takes a subset of data and returns an aggregate. Let us introduce some notation for that.

Denote with U a finite set, and let $\phi : 2^U \to X$ be a function that accepts a subset of U and returns a value in some domain X. Here, U represents a data set, 2^U its power set, and ϕ an aggregating function. We split U into groups using the following notation. Let A be finite set that has no more elements than U, and let $f : U \twoheadrightarrow A$ be a surjective function that takes an element of U and returns a value in A. We can think of A as a set of group labels, and f as the function that assigns a label to each element of U. This way, f divides U into non-overlapping subsets. We say that $f : U \twoheadrightarrow A$ is a *partition* of U. We also introduce the *pullback along* f, $f^* : 2^A \to 2^U$ defined as

$$f^*(S) = \{u \in U \mid f(u) \in S\},$$

where S is a subset of A (see e.g., Fong and Spivak 2019, Section 1.4).

In this notation, any split-apply-combine operation can be computed as shown in Algorithm 1.

In this algorithm the output is collected in a set R containing pairs from $A \times X$: One pair for each element of A. (Incidentally, the algorithm can be summarized even shorter in this notation as $R = \cup_{a \in A}\{(a, (\phi \circ f^*)(\{a\}))\}$, where $\circ$ denotes function composition).

It is interesting to see how the elements U, f, and ϕ are implemented in practice. Consider the signature of the R's `aggregate()` function (we skip arguments that are not important for the discussion).

```
aggregate(x, by, FUN)
```

Here, `x` is a data frame where each row represents an element of U. The parameter `by` is a list of vectors of group labels, where each vector has a length that equals the number of rows in `x`. So the function $f : U \twoheadrightarrow A$ is implemented by asking the user to make sure that the

Algorithm 1: Split-Apply-Combine: SAC(U, ϕ, f)

Input : A finite set U, an aggregator $\phi : 2^U \to X$, and a partition $f : U \twoheadrightarrow A$.
Output: R: The value of ϕ for every part of U as a set of pairs $(a, x) \in A \times X$.

$R = \{\}$;
for $a \in A$ **do**
  $d = f^*(\{a\})$; `// get subset of` U
  $R = R \cup \{(a, \phi(d))\}$; `// aggregate and add to result`
end

position of each label in `by` corresponds to the correct row number in the data frame `x`. The argument `FUN` (of class '`function`') represents the function ϕ that aggregates each subset of `x`. When the records in `x` contain more than one variable, the aggregator is applied to each one of them. Here is an example of how a user might call this function from the R prompt.

```
R> aggregate(iris[1:2], by = iris["Species"], FUN = mean)

     Species Sepal.Length Sepal.Width
1     setosa        5.006       3.428
2 versicolor        5.936       2.770
3  virginica        6.588       2.974
```

Note that the correspondence in position of the `Species` label and the record position is implemented by taking them from the same data frame. The output also reveals in the first column the set A: Each row corresponds to a unique value in `Species` column.

5.2. Split-apply-combine with collapsing groups

The goal of the algorithm is to compute a value for each part of a dataset, possibly using values external to the part —conditional to restrictions placed on each part. The input of the algorithm consists again of a finite set U and an aggregation function ϕ that takes a subset of U and returns a value in some domain X. Compared to Algorithm 1 two other inputs are needed. First, a function must be defined that that checks whether a given subset d of U is suitable for computing $\phi(d)$. We will denote this function $\beta : 2^U \to \mathbb{B}$, where $\mathbb{B} = \{\texttt{True}, \texttt{False}\}$. Typical tests are checking whether there are sufficient records available, or whether certain variables have a low enough fraction of missing values. Second, we need a *collapsing scheme* C, defined as sequence of $n + 1$ mappings

$$C \equiv U \xrightarrow{f} \!\!\!\!\rightarrow\, A \xrightarrow{f_1} \!\!\!\!\rightarrow\, A_1 \xrightarrow{f_2} \!\!\!\!\rightarrow \cdots \xrightarrow{f_n} \!\!\!\!\rightarrow\, A_n. \tag{1}$$

A collapsing scheme is a sequence of partitions where each f_i partitions its domain in $|A_i|$ groups while f partitions U in $|A|$ groups.

Denote with $F_k : A \to A_k$, the function that accepts a label in A and returns the corresponding label in A_k. In other words, F_k is the composition $f_k \circ f_{k-1} \circ \cdots \circ f_1$. Similarly we define the pullback along F_k as $F_k^* = f_1^* \circ f_2^* \circ \cdots \circ f_k^*$. This function accepts a set of labels in A_k and returns all the labels in A that are mapped to those labels via the collapsing sequence of Equation 1. With this notation we can define the Algorithm 2 for aggregation with dynamic grouping.

In this algorithm the collapsing level i is increased until the test is passed or the maximum collapsing level n is reached (Lines 5–8) collapsing is determined dynamically by data algorithm also reports the collapsing level i used. The condition in Line 9 ensures that if no suitable dataset is found after the whole collapsing sequence has been executed, then no answer is returned. This means that in contrast with Algorithm 1 there is no guarantee that a value for each member of A will be found. For each member of A where an aggregate is computed, there is a triple $(a, i, \phi(d))$, where $a \in A$ is the label to which the value $\phi(d)$ pertains, and i is the number of collapses applied to reach a suitable dataset.

Algorithm 2: Split-Apply-Combine with Collapsing Groups: $\mathrm{SACCG}(U, \phi, \beta, C)$

Input : A finite set U, an aggregator $\phi : 2^U \to X$, a test function $\beta : 2^U \to \mathbb{B}$, and a collapsing sequence $C \equiv U \overset{f}{\twoheadrightarrow} A \overset{f_1}{\twoheadrightarrow} A_1 \overset{f_2}{\twoheadrightarrow} \cdots \overset{f_n}{\twoheadrightarrow} A_n$.

Output: R: The value of ϕ for every part of U, for which a suitable collapsing group can be found, as a set of triples $(a, k, x) \in A \times \underline{n} \times X$ where $\underline{n} = \{0, 1, \ldots, n\}$.

```
1  R = {};
2  for a ∈ A do
3  |  i = 0 ;                                  // Initiate collapse level
4  |  d = f*({a}) ;                            // Get subset of U
5  |  while i < n ∧ ¬β(d) do
6  |  |  i = i + 1 ;                           // Increase collapse level
7  |  |  d = (f* ∘ F_i* ∘ F_i)(a) ;            // Collapse and get subset
8  |  end
9  |  if i < n ∨ β(d) then
10 |  |  R = R ∪ {(a, i, φ(d))};
11 |  end
12 end
```

Comparing Algorithms 1 and 2, we see that the standard split-apply-combine algorithm has worst-case run time complexity $O(|A|)$, as determined by counting applications of f^*. This is equal to the algorithm's best case $\Omega(|A|)$. The split-apply-combine with collapsing groups algorithm also has best case $\Omega(|A|)$ but has worst case $O(n|A|)$ (with n the number of collapsing steps). In fact, the best case for Algorithm 2 is achieved by setting $\beta : d \mapsto \texttt{True}$. In this case Algorithm 2 reduces to Algorithm 1. In other words, we have

$$\mathrm{SACCG}(U, \phi, \texttt{True}, C) = \mathrm{SAC}(U, \phi, f).$$

To see this, observe that in Algorithm 2 the condition in line 5 is always `False` and the condition in line 10 is always `True` in this case.

The worst case is achieved by setting $\beta : d \mapsto \texttt{False}$. In that case the while loop in Line 5 is iterated $n - 1$ times (yielding total n executions of f^*) while Line 10 is never executed.

The analyses above leaves open the question of how the run time depends on application of the pullbacks in both algorithms. First note that there needs to be no difference in applying f^* or $f^* \circ F_i^* \circ F_i$: In practice a collapsing scheme can be represented in tabular form just like f. The pullback then comes down to a lookup of records based on matching one or more attributes with a set of attribute values. The time complexity of such operations are typically reduced by proper preparation of the dataset. For example, databases can be prepared to speed up certain often-used lookup operations. In the implementation of the `accumulate` package the pullback is implemented using standard join operations as implemented by R's standard `merge()` function.

6. Summary and conclusion

The R package `accumulate` introduced in this paper offers convenient interfaces for computing grouped aggregates where the grouping is dynamically determined based on user-defined

conditions and a user-defined group collapsing scheme. We demonstrated that these interfaces can be used in several situations, including those where multiple grouping variables are used, where complex collapsing schemes are applied, as well as situations where multiple variables need to be aggregated over. Conditions on groups of records are principally represented by a function, but the package includes several convenience functions for defining conditions on record groups. It also interfaces with the **validate** R package to facilitate cases where multiple test conditions must be met. Moreover, the package supports the collection of complex aggregates, such as model outputs and offers several service functions that aim to facilitate the definition of the collapsing conditions.

The pseudocode underlying the package's main functions has been formally analyzed and it is shown that aggregation with collapsing groups can be interpreted as a precise generalization of standard grouped aggregation. We hope that this stimulates broader implementation of this algorithm in software offering grouped aggregation.

References

Andridge RR, Little RJA (2010). "A Review of Hot Deck Imputation for Survey Non-Response." *International Statistical Review*, **78**(1), 40–64. doi:10.1111/j.1751-5823.2010.00103.x.

Becker RA, Chambers JM, Wilks AR (1988). *The New S Language*. Chapman & Hall/CRC, New York. doi:10.1201/9781351074988.

Boonstra HJ (2022). ***hbsae**: Hierarchical Bayesian Small Area Estimation*. R package version 1.2, URL https://CRAN.R-project.org/package=hbsae.

Bouchet-Valat M, Kamiński B (2023). "**DataFrames.jl**: Flexible and Fast Tabular Data in Julia." *Journal of Statistical Software*, **107**(4), 1–32. doi:10.18637/jss.v107.i04.

Council of European Union (2006). "Council Regulation (EU) No 1893/2006." URL https://eur-lex.europa.eu/legal-content/EN/TXT/?uri=CELEX:32006R1893.

Dowle M, Srinivasan A (2024). ***data.table**: Extension of 'data.frame'*. R package version 1.16.0, URL https://CRAN.R-project.org/package=data.table.

Eastwood N (2023). ***poorman**: A Poor Man's Dependency Free Recreation of **dplyr***. R package version 0.2.7, URL https://CRAN.R-project.org/package=poorman.

European Commision (2022). "European Skills, Competences, Qualifications, and Occupations." URL https://esco.ec.europa.eu/.

Fong B, Spivak DI (2019). *An Invitation to Applied Category Theory: Seven Sketches in Compositionality*. Cambridge University Press.

Josse J, Mayer I, Tierney N, Vialaneix N (2024). *CRAN Task View: Missing Data*. Version 2024-06-20, URL https://CRAN.R-project.org/view=MissingData.

Kowarik A, Templ M (2016). "Imputation with the R Package **VIM**." *Journal of Statistical Software*, **74**(7), 1–16. doi:10.18637/jss.v074.i07.

Krantz S (2024). ***collapse**: Advanced and Fast Data Transformation*. R package version 2.0.16, URL https://CRAN.R-project.org/package=collapse.

Matsakis ND, Klock FS (2014). "The Rust Language." In *HILT '14: Proceedings of the 2014 ACM SIGAda Annual Conference on High Integrity Language Technology*, pp. 103–104. Association for Computing Machinery, New York. doi:10.1145/2663171.2663188.

Molina I, Marhuenda Y (2015). "sae: An R Package for Small Area Estimation." *The R Journal*, **7**(1), 81–98. doi:10.32614/rj-2015-007.

Polars (2024). *Polars User Guide*. URL https://docs.pola.rs/.

R Core Team (2024). *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria. URL https://www.R-project.org/.

Rao JNK, Molina I (2015). *Small Area Estimation*. John Wiley & Sons.

United Nations Statistical Division (2008). "International Standard Industrial Classification of Economic Activities (ISIC)." URL https://unstats.un.org/unsd/classifications/Econ/ISIC.cshtml.

Van der Loo M (2022). ***simputation**: Simple Imputation*. R package version 0.2.8, URL https://github.com/markvanderloo/simputation.

Van der Loo M (2025). ***accumulate**: Split-Apply-Combine with Collapsing Groups*. R package version 1.0.0, URL https://CRAN.R-project.org/package=accumulate.

Van der Loo MPJ, De Jonge E (2021). "Data Validation Infrastructure for R." *Journal of Statistical Software*, **97**(10), 1–31. doi:10.18637/jss.v097.i10.

Van Rossum G, *et al.* (2011). *Python Programming Language*. URL http://www.python.org/.

Wickham H (2011). "The Split-Apply-Combine Strategy for Data Analysis." *Journal of Statistical Software*, **40**, 1–29. doi:10.18637/jss.v040.i01.

Wickham H, François R, Henry L, Müller K (2024). ***dplyr**: A Grammar of Data Manipulation*. R package version 1.1.4, URL https://CRAN.R-project.org/package=dplyr.

Affiliation:

Mark P. J. van der Loo
Research and Development
Statistics Netherlands
Henri Faasdreef 312
2492JP Den Haag, The Netherlands
E-mail: mpj.vanderloo@cbs.nl
URL: https://www.markvanderloo.eu/

and
Leiden Institute of Advanced Computer Science (LIACS)
Leiden University
P.O. Box 9512, 2300 RA Leiden, The Netherlands

Journal of Statistical Software https://www.jstatsoft.org/
published by the Foundation for Open Access Statistics https://www.foastat.org/
March 2025, Volume 112, Issue 4 *Submitted:* 2023-07-24
doi:10.18637/jss.v112.i04 *Accepted:* 2024-05-17