Getting Started with `hdImpute`

Philip Waggoner

February 28, 2022

Introductory Remarks

This brief vignette walks users through the `hdImpute` package at a high level, covering:

1. The `hdImpute` process in individual stages (correlation matrix, flatten and rank, and impute and join)
2. The `hdImpute` process in a single stage, with minimal arguments to satisfy via the `hdImpute()` function, which does all three steps in a single call.

The first approach offers users a bit more flexibility in preprocessing and in the `hdImpute` process (e.g., storing objects as they are created, setting up timing or benchmarking along the way, etc.). The second approach is slightly more inflexible, but is more intuitive. Users simply pass the raw data object (`data`) and supply the batch size (`batch`) to the `hdImpute()` function. The function takes care of all stages from the first approach in a single function call.

Approach 1: Individual Stages

For the stage-based approach, there are three core functions users must use:

1. `feature_cor()`: creates the correlation matrix. *Note*: Dependent on the size and dimensionality of the data as well as the speed of the machine, this preprocessing step could take some time. For example, in an earlier testing run, a simulated data frame of size 20000×3000 had a runtime of roughly 4.25 hours, while a smaller data frame of size 2519×1558 had a runtime of roughly 1 minute on an AWS EC2 instance with 32 cores.
2. `flatten_mat()`: flattens the correlation matrix from the previous stage, and ranks the features based on absolutely correlations. Thus, the input for `flatten_mat()` should be the stored output from `feature_cor()`.
3. `impute_batches()`: creates batches based on the feature rankings from `flatten_mat()`, and then imputes missing values for each batch, until all batches are completed. Then, joins the batches to give a completed, imputed data set.

Consider a basic example.

First, load the library along with the `tidyverse` library for some additional helpers.

```
library(hdImpute)
library(tidyverse)
```

Next, set up the data and introduce missingness completely at random (MCAR) via the `prodNA()` function from the `missForest` package.

```
d <- data.frame(X1 = c(1:6),
                X2 = c(rep("A", 3),
                       rep("B", 3)),
                X3 = c(3:8),
                X4 = c(5:10),
                X5 = c(rep("A", 3),
                       rep("B", 3)),
                X6 = c(6,3,9,4,4,6))

set.seed(1234)

data <- missForest::prodNA(d, noNA = 0.30) %>%
  as_tibble()
```

Note: This is a tiny sample set, but hopefully the usage is clear enough.

Next, follow each stage mentioned above, starting with building the correlation matrix. Of note, `feature_cor()` as an additional argument `return_cor`, which is logical. The default is `FALSE`, but if `TRUE`, the output is stored as normal, and the correlation matrix is printed in the console. For illustrative purposes, I set `return_cor = TRUE`.

```
all_cor <- feature_cor(data,
                       return_cor = TRUE)
#>    X1        X2        X3        X4        X5        X6
#> X1  1 0.0000000 1.0000000 1.0000000 0.6666667 1.0000000
#> X2  0 1.0000000 0.9707253 0.9707253 1.0000000 0.7559289
#> X3  1 0.9707253 1.0000000 1.0000000 0.9244735 0.3885143
#> X4  1 0.9707253 1.0000000 1.0000000 0.9244735 0.3885143
#> X5  0 1.0000000 0.9244735 0.9244735 1.0000000 0.3885143
#> X6  1 0.7559289 0.4313311 0.4313311 0.6123724 1.0000000
```

Next, flatten the matrix and order features. Similarly, `flatten_mat()` has an optional argument `return_mat`, which by default is set to `FALSE`. If `TRUE`, it prints the ranked features based on the correlation matrix. Here again, I set it to `TRUE`.

```
flat_mat <- flatten_mat(all_cor,
                        return_mat = TRUE)
#> # A tibble: 15 x 3
#>    row   column   cor
#>    <chr> <chr>  <dbl>
#>  1 X1    X2     0
#>  2 X3    X6     0.389
#>  3 X4    X6     0.389
#>  4 X5    X6     0.389
#>  5 X1    X5     0.667
#>  6 X2    X6     0.756
#>  7 X3    X5     0.924
#>  8 X4    X5     0.924
#>  9 X2    X3     0.971
#> 10 X2    X4     0.971
#> 11 X1    X3     1
```

```
#> 12 X1    X4     1
#> 13 X3    X4     1
#> 14 X2    X5     1
#> 15 X1    X6     1
```

Finally, impute on a batch by batch basis, join and return the completed data set. There are several additional argument given the imputation model is chained random forests (built on top of `missRanger`, which is built on top of `missForest`). Of note, the `pmm_k` and `n_trees` arguments allow the user to specify the number of neighbors to search and the number of trees to used in building the random forests, respectively. Inspect the `missRanger` documentation for more on these if desired. The default values in `impute_batches()` are set at `5` and `15`, respectively. Other arguments, e.g., `save`, if set to `TRUE` saves an `.RDS` of the list of imputed batches to the working directory. The default is set to `FALSE`.

Ultimately, users need only pass the original/raw data object (`data`), the ranked features (`features`) from `flatten_mat()`, and the batch size (`batch`) to the `impute_batches()` function. The output is the completed, imputed data set.

```
imputed1 <- impute_batches(data = data,
                           features = flat_mat,
                           batch = 2)
#>
#> Missing value imputation by random forests
#>
#>   Variables to impute:       X1
#>   Variables used to impute:  X1
#> iter 1:  .
#>
#> Missing value imputation by random forests
#>
#>   Variables to impute:       X3, X4
#>   Variables used to impute:  X3, X4
#> iter 1:  ..
#>
#> Missing value imputation by random forests
#>
#>   Variables to impute:       X5, X2
#>   Variables used to impute:  X5, X2
#> iter 1:  ..
#> iter 2:  ..
#> iter 3:  ..
#> iter 4:  ..
#>
#> Missing value imputation by random forests
#>
#>   Variables to impute:       X6
#>   Variables used to impute:  X6
#> iter 1:  .
```

```
imputed1
#> # A tibble: 6 x 6
#>      X1 X2       X3    X4 X5       X6
#>   <int> <chr> <int> <int> <chr> <dbl>
#> 1     1 A         3     5 A         6
#> 2     1 A         4     6 A         3
```

```
#> 3     3 B         5     7 A         9
#> 4     1 B         5     5 B         4
#> 5     1 B         7     9 B         9
#> 6     3 B         8    10 B         6
```

Compare to our synthetic `data` object:

```
data
#> # A tibble: 6 x 6
#>      X1 X2       X3    X4 X5       X6
#>   <int> <chr> <int> <int> <chr> <dbl>
#> 1     1 <NA>      3     5 A         6
#> 2    NA A         4     6 A         3
#> 3     3 <NA>      5     7 A         9
#> 4    NA B        NA    NA <NA>      4
#> 5    NA B         7     9 B        NA
#> 6    NA B         8    10 B         6
```

Approach 2: Single call

The alternative to the individual stages approach, which is slightly less flexible but also simpler, is to make a single call to a single function, `hdImpute()`. The function does everything for you. To call this function, users need only pass the raw data object (`data`, which must have at least one missing value) along with specifying the batch size (`batch`) to `hdImpute()`. The returned output is the same from calling `impute_batches()`: a complete, imputed data set that you would get from the individual stages approach previously covered. Users can of course update default argument values (e.g., `pmm_k`, `n_trees`, etc.) if so desired. But there is no need to do so for the function to work properly.

```
imputed2 <- hdImpute(data = data,
                     batch = 2)
#>
#> Missing value imputation by random forests
#>
#>   Variables to impute:       X1
#>   Variables used to impute:  X1
#> iter 1:  .
#>
#> Missing value imputation by random forests
#>
#>   Variables to impute:       X3, X4
#>   Variables used to impute:  X3, X4
#> iter 1:  ..
#>
#> Missing value imputation by random forests
#>
#>   Variables to impute:       X5, X2
#>   Variables used to impute:  X5, X2
#> iter 1:  ..
#> iter 2:  ..
#> iter 3:  ..
#> iter 4:  ..
#>
```

```
#> Missing value imputation by random forests
#>
#>   Variables to impute:       X6
#>   Variables used to impute:  X6
#> iter 1:  .
```

```
imputed2
#> # A tibble: 6 x 6
#>      X1 X2       X3    X4 X5       X6
#>   <int> <chr> <int> <int> <chr> <dbl>
#> 1     1 A         3     5 A         6
#> 2     1 A         4     6 A         3
#> 3     3 B         5     7 A         9
#> 4     1 B         5     5 B         4
#> 5     1 B         7     9 B         9
#> 6     3 B         8    10 B         6
```

Concluding Remarks

This software is in its infancy. As such, wide engagement with it and collaboration is welcomed! Feel free to submit an issue reporting a bug, requesting a feature enhancement, etc. Or consider suggesting changes directly via a pull request. Or feel free to reach out to me directly with ideas if you're uneasy with public interaction. Thanks for reading and using the tool! I hope its useful.