randomForestSRC: learned test-time imputation vignette

Hemant Ishwaran (https://ishwaran.org/ishwaran.html) Min Lu (ht

www.luminwin.net/)

Udaya B. Kogalur

2026-05-19

Overview

Random forest imputation is typically fit and applied to the same data set, treating imputation as a one-time task rather than a reusable procedure. The new `impute.learn()` function in `randomForestSRC` is built for a different situation, where the imputer is learned once from a training sample and then reused to fill in missing values for new observations at a later date. If the training data are themselves incomplete, the method first imputes them using the existing `impute()` engine. It then fits a single sweep of target-specific random forests on the imputed training data and saves only that sweep for later use. Because nothing from the earlier iterative imputation of the training data needs to be retained, the result is a lean, portable object that can be saved, reloaded, and applied repeatedly without any refitting. The method also works when the training data are already complete, and it lets the user specify which variables will be available at test time. This vignette describes the method, explains the `R` interface, and walks through practical workflows for fitting, saving, loading, and applying the learned imputer.

Introduction

Random forest imputation has become a standard nonparametric tool for mixed-type missing data. It is flexible, capable of capturing nonlinear structure and interactions, and requires little in the way of distributional assumptions. The classical `missForest` algorithm of Stekhoven and Buhlmann (2012) and the random forest missing-data algorithms studied by Tang and Ishwaran (2017) are representative of this approach [1, 2]. In both cases, an incomplete data set is iteratively imputed by fitting forests and using those forests to update currently missing values.

That approach works well when the goal is to impute a data set already in hand. In many practical settings, however, the analyst wants an imputation rule that is estimated once from a training sample and then reused on new observations as they arrive. This need arises naturally in external validation studies and production deployment, where new records must be imputed using only information learned from the training data and the imputer itself must be easy to save and reload without refitting. Recent work on this problem includes `missForestPredict`, which extends `missForest` to impute new observations at prediction time [3].

The function `impute.learn()` is designed for this same setting but takes a different approach. If the training data contain missing values, they are first imputed using `impute()`. The function then fits one forest for each selected target variable and saves only that final sweep of forests, discarding the earlier iterative steps used to impute the training data. This is what sets the method apart from `missForestPredict`, which saves the models produced at every iteration and replays that full sequence when imputing new observations [3]. Saving only the final sweep keeps the learned imputer lean and quick to reload, which is one of the main reasons it remains practical for routine use.

Three additional features are worth noting at the outset. First, the method supports the full range of random forest imputation schemes available in `randomForestSRC`, not just `missForest`-style imputation. Second, `impute.learn()` can be fit even when the training data have no missing values, as long as `target.mode = "all"` is specified, which lets the user build protection against future missingness starting from a complete training sample. Third, the user can restrict which predictors are permitted at test time through `deployment.xvars`, keeping the learned imputer focused on the variables that will actually be available when new data arrive.

Problem setup and notation

Let

$$\mathcal{D}_{\text{tr}} = \{\mathbf{x}_i = (X_{i1}, \ldots, X_{ip})\}_{i=1}^{n}$$

denote the training data after coercion to a tabular form, and write

$$\mathbf{X} = (X_{ij})_{1 \le i \le n,\ 1 \le j \le p}$$

for the associated training matrix. In the current implementation, each variable is either numeric or factor. Rows and columns that are entirely missing are removed before fitting. Write

$$M_{ij} = I\left(X_{ij} \text{ is missing}\right)$$

for the indicator that entry (i, j) is missing in the training data.

The learned imputer saves information from the training data that is needed later, including variable type, factor levels when applicable, and the original column order. Let $\mathcal{T}$ denote the set of target variables that receive saved forests. The argument `target.mode` determines this set through

$$\mathcal{T} = \begin{cases} \{j : \sum_{i=1}^{n} M_{ij} > 0\}, & \text{target.mode = "missing.only"}, \\ \{1, \ldots, p\}, & \text{target.mode = "all"}. \end{cases}$$

Thus the default saves forests only for variables that were missing in the training data, whereas `"all"` saves one forest for every variable. When the training data are complete, `"all"` is required.

For each target $j \in \mathcal{J}$, let $\mathcal{P}(j) \subseteq \{1, \ldots, p\} \setminus \{j\}$ denote the set of predictors allowed for that target. By default, all other variables are eligible. The argument `deployment.xvars` allows the user to restrict these predictors to the variables that will truly be available at test time.

Let

$$\mathcal{D}_{\text{te}} = \{\mathbf{x}_i^{\text{te}} = (X_{i1}^{\text{te}}, \ldots, X_{ip}^{\text{te}})\}_{i=1}^{m}$$

denote later data to be imputed. The central task is to estimate the imputation rule from $\mathcal{D}_{\text{tr}}$ once and then apply that same rule to $\mathcal{D}_{\text{te}}$ without refitting any forest.

Learned test-time imputation

Learning the imputer from the training data

If the training data contain missing values, the first step uses `impute()` to produce an imputed training matrix

$$\widetilde{\mathbf{X}} = \left(\widetilde{X}_{ij}\right)_{1 \le i \le n,\ 1 \le j \le p}.$$

The arguments `mf.q`, `max.iter`, and `formula` retain the same meaning they have in `impute()`, and `formula` affects only this first stage. If the training data are already complete and `target.mode = "all"`, this stage is skipped and $\widetilde{\mathbf{X}}$ is simply the training matrix itself.

Once $\widetilde{\mathbf{X}}$ is in hand, the method builds one supervised forest for each target variable $j \in \mathcal{J}$. The forest for target j is trained only on rows where that variable was actually observed in the original training data,

$$\mathcal{O}_j = \{i : M_{ij} = 0\},$$

A forest for target j is then fit on rows $i \in \mathcal{O}_j$, using the completed predictors $\widetilde{X}_{i,\mathcal{P}(j)}$ and the observed target values X_{ij}:

$$\widehat{f}_j = \mathrm{RF}_j(\{(\widetilde{X}_{i,\mathcal{P}(j)}, X_{ij}) : i \in \mathcal{O}_j\}), \qquad j \in \mathcal{J}.$$

Only one forest is produced and saved per target. This single-sweep design is a defining feature of the method. Because nothing from the earlier iterative imputation of the training data needs to be retained, the saved object stays lean and is straightforward to reload and reuse.

Matching later data to the training variables

Before any predictions are made, the new data are matched to the training variables in four steps.

1. Missing columns are added and filled with `NA`.
2. Extra columns not seen in training are dropped.
3. Column order is restored to match the training order.
4. Factor levels not seen in training are converted to `NA`.

The last step is important. A factor label that did not appear in the training data is treated as missing rather than carried forward as an unrecognized level.

Once the columns are matched, missing entries are initialized with training means or modes. Integer-valued variables are restored as integers at the end. In this step, only cells that are missing in the new data are touched; observed values supplied by the user are left exactly as they are.

Iterative test-time sweep

Let $\mathcal{M}_j^{\text{te}}$ denote the set of rows in the new data for which target $j \in \mathcal{J}$ is missing after the column-matching step. Starting from the initialized matrix $\mathbf{X}^{(0)}$, the method sweeps repeatedly through the saved targets in their stored order. In each pass, for each target $j \in \mathcal{J}$, the saved forest $\widehat{f}_j$ fills in only the rows that are still missing,

$$X_{ij}^{(t)} \leftarrow \widehat{f}_j(X_{i,\mathcal{P}(j)}^{(t,\text{curr})}), \qquad i \in \mathcal{M}_j^{\text{te}}, \quad j \in \mathcal{J}.$$

Here $X^{(t,\text{curr})}$ denotes the current working matrix within pass t, so updates made earlier in the pass are immediately available to targets visited later in the same pass.

When `target.mode = "missing.only"`, only variables that had missing values in the training data receive saved forests. A variable that was fully observed in training but turns up missing in new data will therefore keep its starting value throughout, with no further adjustment. When future missingness could affect any variable, `target.mode = "all"` is generally the safer choice.

Stopping rule

After each pass, the procedure compares the newly imputed values against those from the previous pass. For a numeric variable, the comparison uses a scaled root mean squared difference computed over rows that were missing at test time. For a factor variable, it counts the fraction of labels that changed. These $\Delta_j^{(t)}$ comparison scores are then averaged across targets,

$$\Delta^{(t)} = \frac{1}{|\mathcal{J}|} \sum_{j \in \mathcal{J}} \Delta_j^{(t)},$$

where targets with no missing rows in the new data are left out of the average. The passes stop early when $\Delta^{(t)} < \varepsilon$, where ε is the user-supplied value of `eps`, and otherwise continue until `max.predict.iter` passes have been completed.

Saving, loading, and caching

The learned imputer is returned as an object with three parts.

1. A small record of variable types, factor levels, targets, predictor sets, sweep order, starting values, and timing information.
2. Optionally, the saved forests held in memory.
3. Optionally, an on-disk form consisting of `manifest.rds` plus one saved forest file for each target.

At least one storage mode must be used. The forests can remain in memory, be written to disk, or both. The on-disk form relies on the package wrappers `fast.save()` and `fast.load()`, so the `fst` package is required when disk storage is used. Keeping the object small and fast to reload is a deliberate design goal, since the whole point is that the learned imputer can be picked up and reused with minimal overhead.

User interface and implementation in `randomForestSRC`

The learned test-time imputer is used through four principal functions.

```
fit  <- impute.learn(...)
newdata.imp  <- predict(fit, newdata = ...)
save.impute.learn(fit, path = ...)
load.fit  <- load.impute.learn(path = ...)
```

The fitting and prediction interfaces follow the familiar `impute()` workflow, but several arguments play a special role for test-time use.

Fitting the learned imputer

A typical fitting call has the form

```
fit <- impute.learn(
  data = train,
  mf.q = 1,
  max.iter = 5,
  full.sweep.options = list(ntree = 100, nsplit = 10),
  target.mode = "all",
  deployment.xvars = NULL,
  anonymous = TRUE
)
```

The main fitting arguments are the following.

1. `formula` controls only the initial imputation of the training data when the training data are incomplete.
2. `deployment.xvars` specifies which variables may be used as predictors at test time.
3. `target.mode` determines which variables receive saved forests. Use `"all"` whenever later missingness may occur in any variable, and always when the training data are complete.
4. `full.sweep.options` sets the forest tuning values for the saved test-time forests and not for the initial call to `impute()`.
5. `anonymous`, `keep.models`, `out.dir`, and `save.on.fit` control object size and where the saved forests are stored.

The distinction between `formula` and `deployment.xvars` is worth spelling out. The formula can draw on any variables that help impute the training data. The deployment argument then states which variables are permitted to enter the saved test-time forests. This makes it possible to use a rich formula during training while still restricting prediction to the variables that will actually be observed when new data arrive.

Test-time imputation

Once a learned imputer has been fit or reloaded, imputing new data is as simple as

```
newdata.imp <- predict(
  fit,
  newdata = test,
  max.predict.iter = 3,
  eps = 1e-3,
)
```

The returned object is the imputed data frame itself. Additional information is attached as an attribute containing record pass history, columns added or removed when the new data are matched to the training variables, unseen factor levels, forest load counts, and any per-target problems that arose during prediction.

Saving, loading, and loading only selected targets

When `out.dir` is supplied during fitting, the learned imputer is written directly to disk. A fitted object held in memory can also be saved at any later point.

```
save.impute.learn(fit, path = bundle.dir)
load.fit <- load.impute.learn(bundle.dir, lazy = TRUE)
```

The load function can also read only selected targets, which is useful when later work needs only a subset of variables to be updated.

Worked examples

Incomplete training data and held out rows

The `airquality` data provides a small example with both numeric variables and missing values. We keep five variables, convert `Month` to a factor, split the rows into training and held out sets, and learn the imputer from the training rows. Because the training data are incomplete, `impute.learn()` first calls `impute()` to impute the training sample and then fits the saved test-time forests. Setting `target.mode = "all"` asks the function to save a forest for every variable, including variables that happen to be fully observed in this particular training split.

```
library(randomForestSRC)

set.seed(101)
aq <- airquality[, c("Ozone", "Solar.R", "Wind", "Temp", "Month")]
aq$Month <- factor(aq$Month)

id <- sample(seq_len(nrow(aq)), 100)
train <- aq[id, ]
test  <- aq[-id, ]

fit <- impute.learn(
  data = train,
  mf.q = 1,
  max.iter = 5,
  full.sweep.options = list(ntree = 25, nsplit = 5),
  target.mode = "all"
)

test.imp <- predict(fit, test, max.predict.iter = 2, verbose = FALSE)
info <- attr(test.imp, "impute.learn.info")

head(test.imp)

   Ozone Solar.R Wind Temp Month
1     41     190  7.4   67     5
5     17     158 14.3   56     5
11     7      96  6.9   74     5
13    11     290  9.2   66     5
16    14     334 11.5   64     5
19    30     322 11.5   68     5

info$pass.diff

[1] 0.6191309 0.1669537
```

The object `test.imp` is the imputed held out data. The vector `info\$pass.diff` reports the change from one test-time pass to the next and gives a direct check that the sweep is settling. The example shows the intended use of the method in its simplest form: the imputer is learned once from the training rows (here using `missForest` since `mf.q=1`) and then applied to new rows as they arrive.

Complete training data with missing values introduced later

The next example illustrates a feature that is easy to overlook. The training data do not need to contain missing values. Here the complete `iris` data are used to learn one forest for every variable, and missing values are introduced only in the new data. Because the training data are complete, `target.mode = "all"` is required.

```
library(randomForestSRC)

train <- iris
test <- iris[1:12, ]

set.seed(7)
test$Sepal.Length[sample(seq_len(nrow(test)), 3)] <- NA
test$Species[sample(seq_len(nrow(test)), 2)] <- NA

fit <- impute.learn(
  data = train,
  target.mode = "all",
  full.sweep.options = list(ntree = 50, nsplit = 5)
)

test.imp <- predict(fit, test, max.predict.iter = 2, verbose = FALSE)
test.imp

   Sepal.Length Sepal.Width Petal.Length Petal.Width Species
1      5.100000         3.5          1.4         0.2  setosa
2      4.900000         3.0          1.4         0.2  setosa
3      4.656523         3.2          1.3         0.2  setosa
4      4.600000         3.1          1.5         0.2  setosa
5      5.000000         3.6          1.4         0.2  setosa
6      5.400000         3.9          1.7         0.4  setosa
7      4.950964         3.4          1.4         0.3  setosa
8      5.000000         3.4          1.5         0.2  setosa
9      4.400000         2.9          1.4         0.2  setosa
10     4.802320         3.1          1.5         0.1  setosa
11     5.400000         3.7          1.5         0.2  setosa
12     4.800000         3.4          1.6         0.2  setosa
```

In this example the initial training imputation step is skipped entirely and the function fits the saved forests directly from the complete training data. This example are for users who want to build protection against missingness that may only arise after the model has been trained.

Save and reload with an unseen factor level

The last example illustrates a more detailed workflow. The `pbc` data are converted to a mainly factor representation, the learned imputer is fit once, the object is written to disk, and it is later reloaded with lazy loading. We also introduce a new level of `stage` in the new data to show how unseen factor levels are handled. The example further illustrates the distinction between `formula` and `deployment.xvars` : the survival formula is used while imputing the training data (supervised imputation), but the saved test-time forests are restricted to a chosen predictor set.

```
library(randomForestSRC)

data(pbc, package = "randomForestSRC")
dta <- data.frame(lapply(pbc, factor))
dta$days <- pbc$days

set.seed(1)
idx <- sample(seq_len(nrow(dta)), round(nrow(dta) * .25))
train <- dta[idx, ]
test  <- dta[-idx, ]

xvars <- setdiff(names(train), c("days", "status"))

levels(test$stage) <- c(levels(test$stage), "fake")
test$stage[sample(seq_len(nrow(test)), 10)] <- "fake"

fit <- suppressWarnings(
  impute.learn(
    Surv(days, status) ~ .,
    train,
    deployment.xvars = xvars,
    keep.models = TRUE,
    target.mode = "all",
    anonymous = TRUE
  )
)

bundle.dir <- file.path(tempdir(), "pbc.imputer")
save.impute.learn(fit, bundle.dir, verbose = FALSE)
load.fit <- load.impute.learn(bundle.dir, lazy = TRUE, verbose = FALSE)

test.imp <- predict(
  load.fit,
  test,
  max.predict.iter = 2,
  verbose = FALSE
)

attr(test.imp, "impute.learn.info")$unseen.levels$stage
unlink(bundle.dir, recursive = TRUE)
```

The inserted level `"fake"` is converted to `NA` when the new data are matched to the training factor levels, after which it is imputed like any other missing value. Passing `lazy = TRUE` to `load.impute.learn()` delays each file read until the corresponding forest is actually needed during prediction.

Saving, loading, and loading only selected targets

When `out.dir` is supplied during fitting, the learned imputer can be written directly to disk. Alternatively, a fitted object kept in memory can be saved later.

```
save.impute.learn(fit, path = bundle.dir)
load.fit <- load.impute.learn(bundle.dir, lazy = TRUE)
```

The load function can also read only selected targets. This is useful when later work needs only a subset of variables to be updated. Combined with `cache.learners` , this gives fine control over memory use and file reads.

Practical guidance and caveats

The learned imputer is straightforward to use, but several practical points are worth keeping in mind.

1. With the default `"missing.only"` , only variables that were missing in the training data are saved as targets. Those variables can be updated at test time. A variable that was complete in the training data is not a saved target under `"missing.only"` . If it is missing in new data, it is filled with its stored starting value and is not updated further. Complete training data therefore require `target.mode = "all"` .
2. . This determines which variables may be used as predictors when updating a saved target, but it does not determine which variables are themselves saved targets. A variable omitted from `deployment.xvars` can still be updated at test time if it is a saved target. Omitting a variable from `deployment.xvars` only means that it is not used as a predictor for other targets. A variable that is not a saved target receives only its stored starting value when it is missing in new data, whether or not it appears in `deployment.xvars` . It may still be used as a predictor for other targets if `deployment.xvars` allows it.
3. Related to point 2, by default if `deployment.xvars` is left unspecified, every variable (excluding the target) will be used to impute each target. This can be too permissive when the training data contain outcomes, information measured after baseline, identifiers, or other fields that will not be available when new data arrive. On the other hand, if `deployment.xvars` is chosen too narrowly, fitting stops whenever a saved target is left with no predictors.
4. . They are removed from `newdata` at prediction time. Suppose, for example, that a single new observation is missing in every variable retained by the fitted imputer. That observation will still be imputed. The method first fills every retained variable with its stored starting value and then applies the saved forests to the saved targets in the usual sweep order. Under `target.mode = "all"` , every retained variable is eligible for update. Under `"missing.only"` , only variables that were missing in the training data are updated, and the remaining variables stay at their starting values.
5. If the training data have no missing values and `target.mode = "all"` is set, the method skips the initial call to `impute()` and fits the saved forests directly from the training data.
6. This is the main reason the object stays lean. The iterative steps used to impute the training data are not stored and do not need to be.
7. A level not present in training is converted to `NA` before the first test-time pass.
8. The saved object consists of a small `manifest.rds` file plus one forest file for each target. Saving and loading in this format requires the package.
9. Only cells that are missing after the new data are matched to the training variables are ever updated during prediction.

Cite this vignette as

H. Ishwaran, M. Lu, and U. B. Kogalur. 2026. “randomForestSRC: learned test-time imputation vignette.” http://randomforestsrc.org/articles/testimput.html (http://randomforestsrc.org/articles/testimput.html).

```
@misc{HemantTESTIM,
    author = "Hemant Ishwaran and Min Lu and Udaya B. Kogalur",
    title = {{randomForestSRC}: learned test-time imputation vignette},
    year = {2026},
    url = {http://randomforestsrc.org/articles/testimput.html},
    howpublished = "\url{http://randomforestsrc.org/articles/testimput.html}",
    note = "[accessed date]"
}
```

References

1. Stekhoven DJ, Bühlmann P. MissForest—non-parametric missing value imputation for mixed-type data. Bioinformatics. 2012;28:112–8.
2. Tang F, Ishwaran H. Random forest missing data algorithms. Statistical Analysis and Data Mining. 2017;10:363–77.
3. Albu E, Gao S, Wynants L, Van Calster B. missForestPredict: Missing data imputation for prediction settings. PLOS ONE. 2025;20:e0334125.