Methodology of Automated Missing-Data Imputation Module

Matin Irajpour M.D.

June 4, 2025

1 Overview

This module provides an end-to-end framework for handling missing data by:

- Preprocessing raw datasets to standardize column types.
- Simulating or injecting missingness according to real or user-specified patterns.
- Performing imputation via multiple engines (KNN, MICE, MissForest, MIDAS).
- Evaluating imputation quality against ground truth (when available) and selecting the best method on a per-column basis.
- Tuning hyperparameters automatically using Optuna.
- Orchestrating the entire workflow through a single pipeline function.
- Packaging as an open-source Python library (todo).

Figure (1 demonstrates a flowchart of how the full pipeline works)

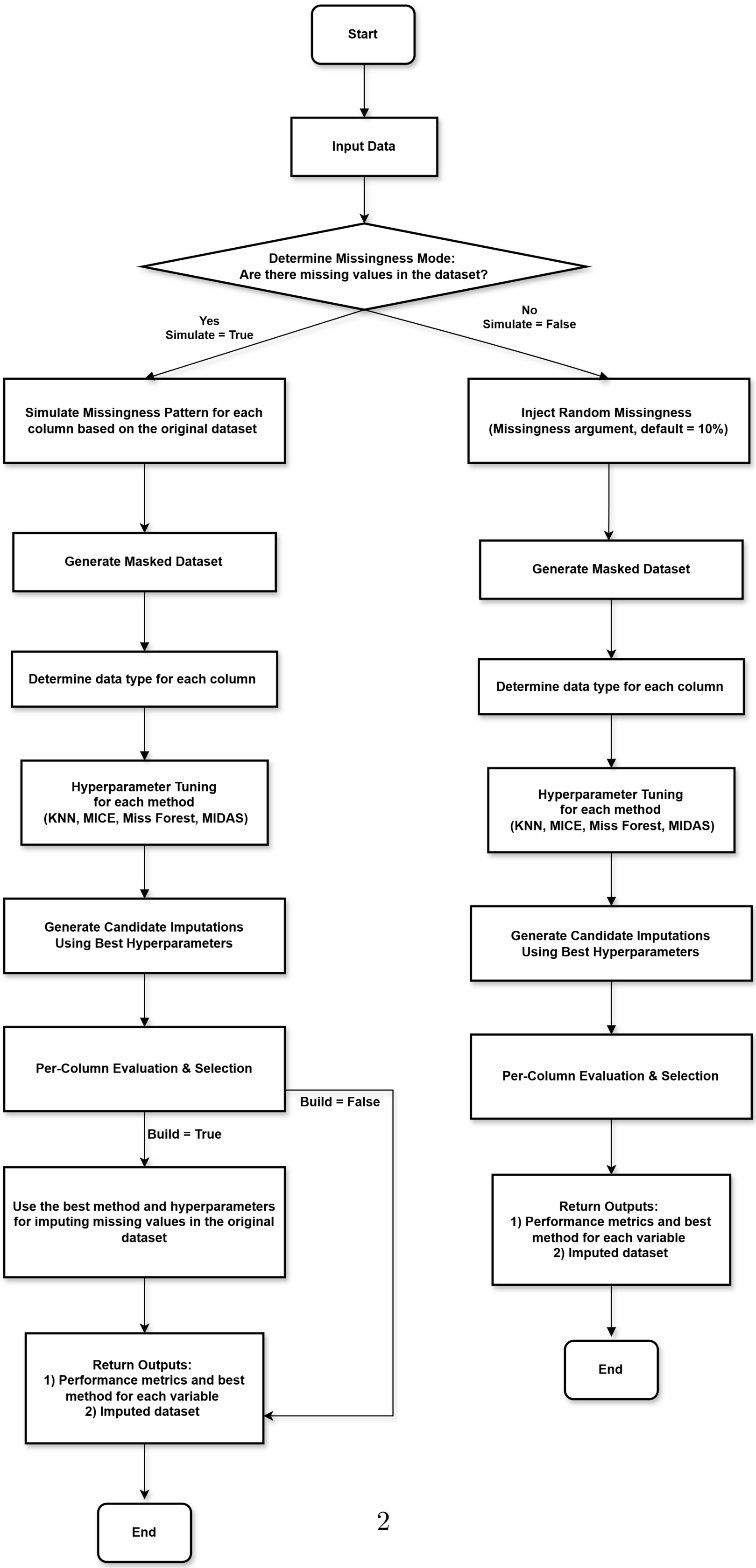

Figure 1: Overview of the full pipeline

2 Data Preprocessing

Purpose. Ensure that all input data are "clean" (no NaNs) and that each column is correctly classified as continuous, discrete numeric, or categorical.

1. *Row filtering.* Drop any rows containing missing values, producing a complete dataset.
2. *Type detection.*
   - Columns of type `object` or with exactly two unique values are marked as categorical.
   - Numeric columns are examined for integer-like values; if more than half are integers, the column is cast to integer (discrete); otherwise, it remains as float (continuous).
3. *Output.* Lists of continuous, discrete, and categorical column names, plus the cleaned DataFrame.

3 Missingness Simulation

Two functions allow users either to replicate existing missingness patterns or to inject uniform random missingness.

3.1 Simulating Existing Patterns

- Drop all real missing values to obtain a complete DataFrame.
- Record original per-column missing rates, then re-introduce missing entries at the same rates (randomly, under a reproducible seed).
- Output: the complete DataFrame, the DataFrame with simulated NaNs, and a Boolean mask of simulated missing positions.

3.2 Injecting Random Missingness

- Given a complete DataFrame, generate a random Boolean mask at a user-specified global missingness percentage.
- Apply the mask to produce a partially missing DataFrame.
- Output: the original complete DataFrame, the masked DataFrame, and the mask.

4 Imputation Methods

Each imputation engine accepts an incomplete DataFrame plus the lists of continuous, discrete, and categorical columns, and returns a fully imputed DataFrame with original dtypes restored.

4.1 KNN Imputation

- Label-encode categorical columns.
- (Optionally) scale all features to $[0, 1]$.
- Use a nearest-neighbors regressor to impute missing values.
- Recover discrete columns by rounding to integers and decode categorical codes back to original labels.

4.2 MICE Forest

- Label-encode categorical columns.
- (Optionally) scale numeric features.
- Build a random-forest–based MICE kernel to iteratively predict missing entries.
- Round discrete predictions and decode categorical codes.

4.3 MissForest

- Label-encode categorical features.
- (Optionally) scale numeric features.
- Apply the MissForest algorithm, which iteratively uses random forests to predict each variable with missing data.
- Round discrete outputs and decode categorical columns.

4.4 MIDAS Imputation

- One-hot encode categorical columns to produce dummy matrices and track category groups.
- Scale numeric features.
- Build and train a VAE-based autoencoder that reconstructs missing entries.

- Generate multiple stochastic completion samples, then for each sample:
  - Inverse-scale numeric features.
  - Convert one-hot predictions back to categorical labels by selecting highest-probability dummy.
  - Round discrete columns to integers.
- Return a list of candidate imputations.

5 Imputation Evaluation and Selection

Goal. Given a set of candidate imputations (from one or more engines) and access to ground truth, determine which method performs best on each column:

1. For each column where missingness was simulated:
   - **Numeric (continuous or discrete).** Compute mean absolute error (MAE) over masked positions.
   - **Categorical.** Compute error as $1-$ accuracy over masked positions.
2. For each column, select the method (or MIDAS sample) with minimal MAE or minimal categorical error.
3. Assemble a "best-per-column" DataFrame by replacing each masked column's entries with values from the chosen candidate.
4. Produce a summary table listing, for each column:
   - Column name and data type.
   - Best method identifier.
   - Associated error metric.
   - (For numeric columns) standard deviation of errors, maximum and minimum errors, and proportion within 10% of true values.

6 Hyperparameter Optimization

Approach. For each imputation engine, run an Optuna study that searches key hyperparameters to minimize the aggregate error metric:

- Define an objective function that:
  - Proposes a set of hyperparameters.

    - Runs the corresponding imputation method on a partially missing DataFrame.
    - Evaluates the result versus ground truth via the per-column selection routine, yielding an overall error.
    - Returns this error to Optuna.

- Execute the study under a time limit and minimum number of trials.
- Record the best hyperparameter set and its associated error.

7 End-to-End Pipeline

The central function orchestrates the entire workflow:

1. *Missingness setup.* Depending on whether the user's data already contain real NaNs:
    - If simulating, replicate existing missingness patterns; otherwise, inject uniform missingness at a specified rate.
2. *Preprocessing.* Clean and classify columns into continuous, discrete, and categorical.
3. *Hyperparameter tuning.* For each engine (KNN, MICE, MissForest, MIDAS), run Optuna to find its best settings under simulated conditions.
4. *Candidate generation.* Re-run each engine on the masked data using its tuned hyperparameters; collect all candidate imputations.
5. *Per-column selection.* Evaluate each candidate per column, select the best method/sample, and stitch together a single imputed DataFrame.
6. *Optional final build.* If the original data had real missingness, re-apply each tuned method to the original incomplete DataFrame and perform per-column selection to produce a final imputation.
7. *Outputs.* Return the final imputed DataFrame and a summary table of per-column performance.

8 Key Design Decisions

- **Unified Imputation Interface.** All engine functions accept the same inputs (incomplete DataFrame plus column-type lists) and return a completed DataFrame, facilitating generic evaluation and tuning.

- **Categorical Encoding.** Categorical variables are handled via label encoding (for KNN, MICE, MissForest) or one-hot encoding (for MIDAS), ensuring compatibility with numeric algorithms.
- **Per-Column Selection.** Different variables may favor different imputation strategies; selecting the best per column yields lower overall error than forcing a single method for the entire table.
- **Simulation vs. Build.** Separating the simulation of missingness from the final build step allows for robust evaluation against ground truth before applying imputations to real missing data.
- **Automated Hyperparameter Search.** Integrating Optuna streamlines tuning, preventing manual trial-and-error.