Appendix

Contents

A Dataset

A.1 Rationale for Dataset Selection

In order to rigorously evaluate the proposed multivariate time-series imputation framework, we deliberately constructed a diverse experimental setting combining synthetic, fully-observed datasets and a real-world clinical dataset. This dual strategy allows us to assess the imputation performance both under controlled conditions with known ground truth and in practical downstream predictive tasks. The five datasets considered in this study are: Lorenz-96, Vector Auto-Regressive (VAR) processes, Fama-French financial factor series, Beijing air quality simulation, and the MIMIC-III clinical database. Each dataset was selected with a specific motivation, described as follows.

The first four datasets—Lorenz-96, VAR, Fama-French finance, and Beijing air quality—are synthetic or complete datasets that provide full ground-truth observations. Their complete nature enables us to introduce missingness in a controlled manner (MCAR, MAR, MNAR) and evaluate the imputation quality quantitatively by directly comparing the reconstructed values with the true underlying signals, typically using metrics such as mean squared error (MSE). These datasets collectively cover a wide range of temporal dynamics:

Lorenz-96 The Lorenz-96 system is a canonical chaotic dynamical model widely used in climate and atmospheric sciences. It is characterized by nonlinear interactions and high sensitivity to initial conditions, which make the resulting time series highly non-stationary and non-trivial to forecast. By using Lorenz-96, we test the ability of the imputation model to handle chaotic and nonlinear multivariate dependencies, where the temporal evolution cannot be easily extrapolated from local patterns.

The model is composed of N coupled ordinary differential equations (ODEs) that describe variables within a periodic one-dimensional spatial domain. For each variable x_i, interactions occur with its nearest and next-nearest neighbors, which are governed by:

$$\frac{dx_i}{dt} = (x_{i+1} - x_{i-2}) \cdot x_{i-1} - x_i + F \tag{1}$$

where:

- $i = 1, 2, \ldots, N$ denotes index of the state variable (typically $N = 8$ or 40),
- x_i is the state variable,
- F is a constant **external forcing parameter** controlling system behavior,
- **Periodic boundary conditions** are enforced:

$$x_{i+N} = x_i, \quad x_{i-N} = x_i. \tag{2}$$

Equation (1) comprises three physical terms:

1. **Advection** $(x_{i+1} - x_{i-2}) \cdot x_{i-1}$: Simulates the nonlinear advection process, reflecting the transfer of energy in space.
2. **Dissipation** $-x_i$: Represents damping or viscosity.
3. **Forcing** $+F$: Constant external energy input to keep the system away from equilibrium.

The system's behavior depends critically on F:

- $F < 1$: The system is in a stable equilibrium state.
- $1 \leq F \leq 8$: The system exhibits a periodic oscillation state.
- $F > 8$: The system exhibits fully developed chaos.

Vector Auto-Regressive (VAR) processes VAR models generate multivariate time series with explicit linear causal relationships among variables. This dataset provides a well-controlled environment for evaluating whether the imputation model can recover temporal dependencies that are more structured and linear in nature. In contrast to the chaotic nature of Lorenz-96, VAR series emphasize longitudinal statistical dependencies and Granger-type causal structures.

The standard VAR(p) model for a k-dimensional vector $\mathbf{Y}_t = (y_{1t}, y_{2t}, \ldots, y_{kt})'$ is:

$$\mathbf{Y}_t = \mathbf{c} + \mathbf{A}_1\mathbf{Y}_{t-1} + \mathbf{A}_2\mathbf{Y}_{t-2} + \cdots + \mathbf{A}_p\mathbf{Y}_{t-p} + \boldsymbol{\varepsilon}_t \tag{3}$$

where:

- $\mathbf{c}$: $k \times 1$ vector of intercepts,
- $\mathbf{A}_j$: $k \times k$ coefficient matrix for lag j ($j = 1, \ldots, p$),
- $\boldsymbol{\varepsilon}_t$: $k \times 1$ vector of white noise errors with $\mathbb{E}(\boldsymbol{\varepsilon}_t) = \mathbf{0}$ and $\text{Cov}(\boldsymbol{\varepsilon}_t) = \boldsymbol{\Omega}$.

Fama-French financial factors Financial time series are known for their structured stochastic volatility, strong correlations, and sudden regime changes. By employing a synthetic version of the Fama-French factor model, we replicate the statistical properties of real-world financial markets (such as factor-driven return series) while retaining a fully-observed ground truth. This dataset challenges the imputation model to capture patterns in heteroscedastic, noisy, and partially correlated data.

This dataset extends the Capital Asset Pricing Model (CAPM) by incorporating size and value factors to explain cross-sectional stock returns more accurately. The dataset is widely used for risk assessment, performance attribution, and asset pricing model validation.

The Fama-French three-factor model (FFTFM) suggests that stock returns are influenced by three systematic risk factors:

1. **Market Risk**: Excess return of the market portfolio over the risk-free rate ($R_m - R_f$).
2. **Size Factor (SMB)**: Return differential between small-cap and large-cap stocks.
3. **Value Factor (HML)**: Return differential between high and low book-to-market (B/M) ratio stocks.

These factors address CAPM's empirical shortcomings, such as the outperformance of small-cap stocks (size effect) and value stocks (value effect).

The expected excess return of asset i is modeled as:

$$E[R_i] - R_f = \alpha_i + \beta_i(E[R_m] - R_f) + \gamma_i \cdot \text{SMB} + \delta_i \cdot \text{HML} + \epsilon_i \tag{4}$$

where:

- $E[R_i] - R_f$: Excess return of asset i,
- α_i: Abnormal return,
- $\beta_i, \gamma_i, \delta_i$: Sensitivities to market, size, and value factors,
- ϵ_i: Idiosyncratic error term with $\mathbb{E}[\epsilon_i] = 0$.

The dataset constructs SMB and HML using portfolio sorting:

- **SMB (Small Minus Big)**:

$$\text{SMB} = \frac{1}{3}\left(R_{\text{Small, Value}} + R_{\text{Small, Neutral}} + R_{\text{Small, Growth}}\right) - \frac{1}{3}\left(R_{\text{Big, Value}} + R_{\text{Big, Neutral}} + R_{\text{Big, Growth}}\right) \tag{5}$$

- **HML (High Minus Low)**:

$$\text{HML} = \frac{1}{2}\left(R_{\text{High B/M}} + R_{\text{Small, High B/M}}\right) - \frac{1}{2}\left(R_{\text{Low B/M}} + R_{\text{Small, Low B/M}}\right) \tag{6}$$

The dataset includes:

1. **Time Series**: Monthly and annual returns for key factors (SMB, HML, $R_m - R_f$) across global markets (e.g., North America, Europe, Asia-Pacific).
2. **Portfolio Returns**: Returns for 25 portfolios sorted by size and book-to-market ratios.
3. **Risk-Free Rate**: Generally represented by 1-month U.S. Treasury bill yields.
4. **Detailed Breakdowns**: Sector-specific data (e.g., energy, technology) for in-depth analysis.

Beijing air quality simulation The synthetic Beijing air quality dataset incorporates seasonality, trend, and periodicity typical of environmental monitoring systems. Unlike purely stochastic signals, these series are characterized by long-range dependencies as well as short-term bursts caused by simulated external conditions. This dataset thus assesses the imputation algorithm's ability to handle periodic components, temporal irregularities, and multiscale dependencies.

The dataset includes the following variables:

- **No**: Row number.
- **Year**: Year of data in this row.
- **Month**: Month of data in this row.
- **Day**: Day of data in this row.
- **Hour**: Hour of data in this row.
- **PM2.5**: PM2.5 concentration (unit: $\mu g/m^3$).
- **PM10**: PM10 concentration (unit: $\mu g/m^3$).
- **SO2**: SO2 concentration (unit: $\mu g/m^3$).
- **NO2**: NO2 concentration (unit: $\mu g/m^3$).
- **CO**: CO concentration (unit: $\mu g/100m^3$).
- **O3**: O3 concentration (unit: $\mu g/m^3$).
- **TEMP**: Temperature (unit: degree Celsius).
- **PRES**: Pressure (unit: hPa).
- **DEWP**: Dew point temperature (unit: degree Celsius).
- **RAIN**: Precipitation (unit: mm).
- **wd**: Wind direction.
- **WSPM**: Wind speed (unit: m/100s).
- **station**: Name of the air-quality monitoring site.

MIMIC-III The MIMIC-III (Medical Information Mart for Intensive Care) dataset represents a high-dimensional, irregularly-sampled, and partially observed real-world clinical dataset, collected from intensive care units. Unlike the synthetic datasets, MIMIC-III is inherently incomplete, and the true unobserved values cannot be recovered. In this context, the primary goal of imputation is not to reconstruct a known signal but to improve the quality of data for subsequent predictive tasks.

We use MIMIC-III specifically for evaluating the practical utility of the proposed imputation methods in downstream clinical prediction tasks, such as in-hospital mortality prediction, sepsis onset forecasting, and early detection of acute kidney injury. The dataset presents challenges that go beyond controlled conditions, including asynchronous sampling, varying measurement frequencies, correlated missingness patterns, and the presence of outliers. Evaluating imputation on MIMIC-III allows us to demonstrate whether the improvements seen in controlled synthetic experiments translate to real-world benefits in high-stakes medical decision-making.

B Classifier Architecture

For the downstream prediction tasks, we implement two recurrent neural classifiers, *LSTMClassifier* and *GRUClassifier*, which share the same overall design but differ in their recurrent backbone (LSTM vs. GRU). The complete architecture is as follows:

- **Input Normalization.** Each input sequence $\mathbf{X} \in \mathbb{R}^{B\times T\times F}$ (batch size B, sequence length T, feature dimension F) can be optionally normalized along the feature dimension. Depending on configuration, either feature-wise BatchNorm1d or LayerNorm is applied. Batch normalization is performed by first reshaping to $(B \cdot T, F)$ and reshaping back.

[1] https://archive.ics.uci.edu/dataset/501/beijing+multi+site+air+quality+data

- **Recurrent Encoder.**
  - **LSTM-based model:** A bidirectional LSTM with `hidden_dim`=128, `num_layers`=2, dropout rate of 0.3 (between layers only), and batch-first input layout.
  - **GRU-based model:** Identical to the LSTM variant except that the backbone is a bidirectional GRU.

  The recurrent output has dimension $T \times (2 \cdot 128)$ because of bidirectionality.

- **Normalization after Recurrence.** The output of the recurrent encoder is optionally passed through BatchNorm1d or LayerNorm across the feature dimension.

- **Multi-head Self-Attention.** The normalized recurrent features are processed by a Multi-Head Attention module:

$$\mathrm{Attn}(\mathbf{H}) = \mathrm{MHA}(\mathbf{H}, \mathbf{H}, \mathbf{H}), \tag{7}$$

  where $\mathbf{H} \in \mathbb{R}^{B \times T \times 256}$. The attention module uses 8 heads with an embedding dimension of 256 (again followed by optional normalization). This step refines temporal dependencies and highlights salient time steps.

- **Temporal Pooling.** The attention-enhanced sequence $\mathbf{Z}$ is aggregated along the temporal axis by simple mean pooling:

$$\mathbf{z}_{\text{pool}} = \frac{1}{T} \sum_{t=1}^{T} \mathbf{Z}_{:,t,:}. \tag{8}$$

- **Classifier Head.** The pooled representation is fed into a three-layer feed-forward classifier:
  - **Layer 1:** $\mathbb{R}^{256} \to \mathbb{R}^{128}$, followed by normalization, ReLU activation, and dropout;
  - **Layer 2:** $\mathbb{R}^{128} \to \mathbb{R}^{32}$, again followed by normalization, ReLU, and dropout;
  - **Layer 3:** $\mathbb{R}^{32} \to \mathbb{R}^{1}$, producing a single logit for binary classification.

  Both intermediate layers apply BatchNorm1d (or LayerNorm) depending on the configuration.

The forward computation can be summarized as:

$$\mathbf{x} \xrightarrow{(\mathrm{Norm})} \mathrm{RNN} \xrightarrow{(\mathrm{Norm})} \mathrm{MHA} \xrightarrow{(\mathrm{Norm})} \mathrm{MeanPool} \xrightarrow{\mathrm{FFN}} \hat{y}. \tag{9}$$

The only difference between *SimpleLSTMClassifier* and *SimpleGRUClassifier* lies in whether the recurrent encoder uses an LSTM or a GRU cell.

C Reproductivity Configuration

C.1 Seed for Missing Value Imputation

To ensure strict reproducibility of Missing Value Imputation task, this implementation adopts a comprehensive random seed setting mechanism that explicitly controls all sources of randomness. The function `set_seed_all(seed)` synchronizes the random number generators of Python's `random` module, NumPy, and PyTorch (including CUDA backends), and fixes the hash seed of Python via the `PYTHONHASHSEED` environment variable. Additionally, deterministic modes in PyTorch are enabled by setting `torch.backends.cudnn.deterministic = True` and `torch.backends.cudnn.benchmark = False`, which prevents non-deterministic behavior in convolution-related operations.

Throughout the pipeline, this seed-setting routine is invoked at all stages that involve stochastic behavior, including:

- **Missingness simulation:** Before applying MAR, MCAR, or MNAR masking functions, the seed is fixed to ensure that the same missing entries are generated for each experimental run.
- **Model initialization and training:** Prior to model instantiation and optimizer initialization, seeds are reset to ensure identical weight initialization and training dynamics across different runs.
- **Parallel processing:** In multi-GPU and multi-process environments, the base seed is offset by the task index (`seed + idx`) so that each worker operates deterministically while remaining independent from others.
- **Baseline methods:** Every baseline imputation method is executed with the same seed, ensuring that all methods operate on an identical missing-data pattern and initialization state.

C.2 Seed for Downstream Prediction Task

To ensure that the downstream evaluation results are strictly reproducible, this implementation adopts a unified seed–setting strategy that synchronizes all sources of randomness across the data-loading, model training, and evaluation stages.

C.3 Global seed setting.

The function `set_seed(seed)` is used throughout the code to fix the random state for all major libraries. Specifically, it sets the seed for the Python `random` module, NumPy, and PyTorch (both CPU and CUDA backends) and configures `torch.backends.cudnn.de` `= True` and `torch.backends.cudnn.benchmark = False` to eliminate non-deterministic behavior in convolutional kernels. The environment variable `PYTHONHASHSEED` is also fixed to ensure deterministic hashing.

C.4 Application in the pipeline.

This routine is invoked consistently at critical points:

- **Cross-validation splits:** Before performing K-fold splits and shuffling, the base seed is set so that the same folds are generated across runs.
- **Data loading:** For each worker process in the `DataLoader`, an offset of `seed + fold + worker_id` ensures deterministic but distinct shuffling across workers.
- **Model initialization and training:** Prior to creating model instances and optimizers, the same seed guarantees identical weight initialization and training dynamics.
- **Parallel folds:** In the multi-GPU setting, each fold receives a deterministic offset `seed + fold` so that parallel processes are independent yet reproducible.

C.5 Seed Usage in the Main Execution Pipeline

In the main entry point of the program, the predefined seed mechanism is applied to ensure reproducible results throughout the entire workflow. At the beginning of the `__main__` block, the multiprocessing start method is explicitly set to `spawn` to guarantee deterministic process initialization across different platforms. Subsequently, a global seed value (`SEED = 42`) is passed to the `set_seed()` function, which synchronizes the random states of Python, NumPy, and PyTorch (including CUDA).

This global seed initialization influences every subsequent experimental component:

- **Data preparation:** When `Prepare_data` loads time series datasets, the fixed seed ensures that any shuffling or ordering of samples remains consistent between runs.
- **Causal discovery:** The causal graph construction via `causal_discovery` becomes deterministic since the seed controls both the preprocessing randomness and the clustering procedure.
- **Evaluation of imputation quality:** The function `parallel_mse_evaluate` uses the same global seed to control all aspects of missingness generation, model initialization, and multi-process task distribution, thereby ensuring identical evaluation outcomes even in parallel execution.
- **Downstream predictive evaluation:** When the imputed datasets are fed into `evaluate_downstream(data_arr1, label_arr1, k, epochs, lr, seed)`, the seed ensures deterministic K-fold partitioning, consistent mini-batch sampling, and stable model initialization in downstream classification models (e.g., LSTM/GRU classifiers). This guarantees that downstream task results such as AUROC, F1, accuracy, and precision are reproducible.

By enforcing this seed setting at the start of the main script, the pipeline guarantees that the same inputs, models, and evaluation metrics can be exactly reproduced across different executions and computational environments.

D Hyperparameters Configuration

D.1 Training Hyperparameters

Parameter	Location	Typical Range	Value Used	Description

epochs	Multiple	50–500	100	Training epochs
lr	Multiple	0.001–0.1	0.02	Learning rate
patience	impute()	5–30	15	Early stopping patience
max_norm	Gradient clipping	0.5–5.0	1.0	Gradient clipping norm
eta_min	LR scheduler	lr×0.001 – lr×0.1	lr×0.01	Minimum learning rate for cosine annealing

D.2 Data Preprocessing Hyperparameters

Parameter	Location	Typical Range	Value Used	Description
threshold	FirstProcess	0.5–0.95	0.8	Threshold for imputing columns with dominant values
perturbation_prob	SecondProcess	0.0–0.5	0.3	Probability of perturbing imputed values
perturbation_scale	SecondProcess	0.1–0.5	0.3	Perturbation scale during preprocessing

D.3 Loss Function Weights

Parameter	Location	Typical Range	Value Used	Description
loss_1_weight	Loss computation	0.4–0.8	0.6	Weight for reconstruction loss on observed entries
loss_2_weight	Loss computation	0.2–0.6	0.4	Weight for distributional regularization terms

D.4 Synthetic Missingness Generation Hyperparameters

Parameter	Location	Typical Range	Value Used	Description
obs_rate	mar_logistic	0.1–0.9	0.1	Proportion of initially observed variables (MAR)
missing_rate	mar_logistic	0.1–0.8	0.6	Rate of MAR-based missingness
offset	mnar_x	0.3–0.9	0.6	Offset controlling self-masking severity (MNAR)
p	mcar	0.1–0.7	0.5	Probability for MCAR random masking

D.5 Clustering and Causal Discovery Hyperparameters

Parameter	Location	Typical Range	Value Used	Description
n_cluster	causal_discovery	3–20	5	Number of clusters for feature grouping
significance	Causal discovery	0.5–3.0	1.2	Significance threshold for causal edges
n_init	KMeans	5–20	10	KMeans initializations

D.6 System Resource Hyperparameters

Parameter	Location	Typical Range	Value Used	Description

simultaneous_per_gpu	Parallel processing	1–4	2–3	Number of tasks per GPU
batch_size	Preprocessing	50–200	100	Mini-batch size
threshold_mb	GPU waiting	200–1000	500	GPU idle memory threshold (MB)
sleep_time	GPU waiting	5–30	10	Polling interval (s) for GPU availability

D.7 Imputation baseline Hyperparameters

Parameter	Location	Typical Range	Value Used	Description
		KNN Imputation		
n_neighbors	knn_impu()	1–20	5	Number of neighbors for KNN imputation
MICE Imputation				
max_iter	mice_impu()	5–20	5	Maximum iterations for MICE
SAITS Imputation				
epochs	saits_impu()	50–200	20/50/100	Training epochs (adaptive)
d_model	saits_impu()	32–512	64/128	Model dimension
n_layers	saits_impu()	1–6	1/2	Transformer layers
n_heads	saits_impu()	2–16	min(4, d_model/32)	Attention heads
d_k	saits_impu()	16–128	d_model/8	Key dimension
d_v	saits_impu()	16–128	d_model/8	Value dimension
d_ffn	saits_impu()	64–2048	d_model	FFN dimension
dropout	saits_impu()	0.1–0.5	0.1	Dropout rate
patience	saits_impu()	5–20	10	Early stopping patience
batch_size	saits_impu()	8–128	32	Batch size
TimeMixer++ Imputation				
epochs	timemixerpp_impu()	50–200	100	Training epochs
n_layers	timemixerpp_impu()	1–8	1	Model layers
d_model	timemixerpp_impu()	32–512	64	Model dimension
d_ffn	timemixerpp_impu()	64–2048	128	FFN dimension
top_k	timemixerpp_impu()	1–T/2	T/2	Top-K selection
n_heads	timemixerpp_impu()	1–16	2	Attention heads
n_kernels	timemixerpp_impu()	3–12	6	Number of convolution kernels
dropout	timemixerpp_impu()	0.1–0.5	0.1	Dropout rate
patience	timemixerpp_impu()	3–15	3	Early stopping patience
downsampling_layers	timemixerpp_impu()	0–3	1	Downsampling layers
downsampling_window	timemixerpp_impu()	2–8	2	Downsampling window size
TEFN Imputation				
epochs	tefn_impu()	50–300	100	Training epochs
n_fod	tefn_impu()	1–5	2	Fractional derivative order
patience	tefn_impu()	5–20	5	Early stopping patience
ORT_weight	tefn_impu()	0.1–10.0	1.0	Orthogonal regularization weight
MIT_weight	tefn_impu()	0.1–10.0	1.0	Mutual information weight
TimesNet Imputation				
epochs	timesnet_impu()	50–200	100	Training epochs
n_layers	timesnet_impu()	1–6	2	Model layers
top_k	timesnet_impu()	1–5	1	Top-K selection
d_model	timesnet_impu()	32–512	64	Model dimension
d_ffn	timesnet_impu()	64–2048	256	FFN dimension
n_kernels	timesnet_impu()	3–12	6	Number of convolution kernels
dropout	timesnet_impu()	0.1–0.5	0.1	Dropout rate
patience	timesnet_impu()	5–20	5	Early stopping patience
TSDE Imputation				

n_samples	tsde_impu()	10–100	40	Diffusion sampling steps
GRIN Imputation				
window_size	grin_impu()	5–50	10/15/20	Window size (adaptive)
hidden_dim	grin_impu()	8–128	8/16/32	Hidden dimension (adaptive)
epochs	grin_impu()	50–200	80/100/120	Training epochs (adaptive)
lr	grin_impu()	0.001–0.1	0.01	Learning rate
MIRACLE Imputation				
n_hidden	miracle_impu()	8–128	min(32, max(8, n_feats/2))	Hidden size
lr	miracle_impu()	0.001–0.05	0.008	Learning rate
max_steps	miracle_impu()	20–200	50	Maximum training steps
window	miracle_impu()	3–10	5	Sliding window size

Remark. These configurations were empirically selected to balance efficiency, stability, and reproducibility. Values fall within standard practice while being tailored to the complexity of the datasets and the scale of multi-GPU experiments.

E Imputation performance

This paper also visualizes the imputation performance for each dataset (see Figures 1, 2, 3, 4, and 5), with each dataset displaying the imputation results of three features across all methods.

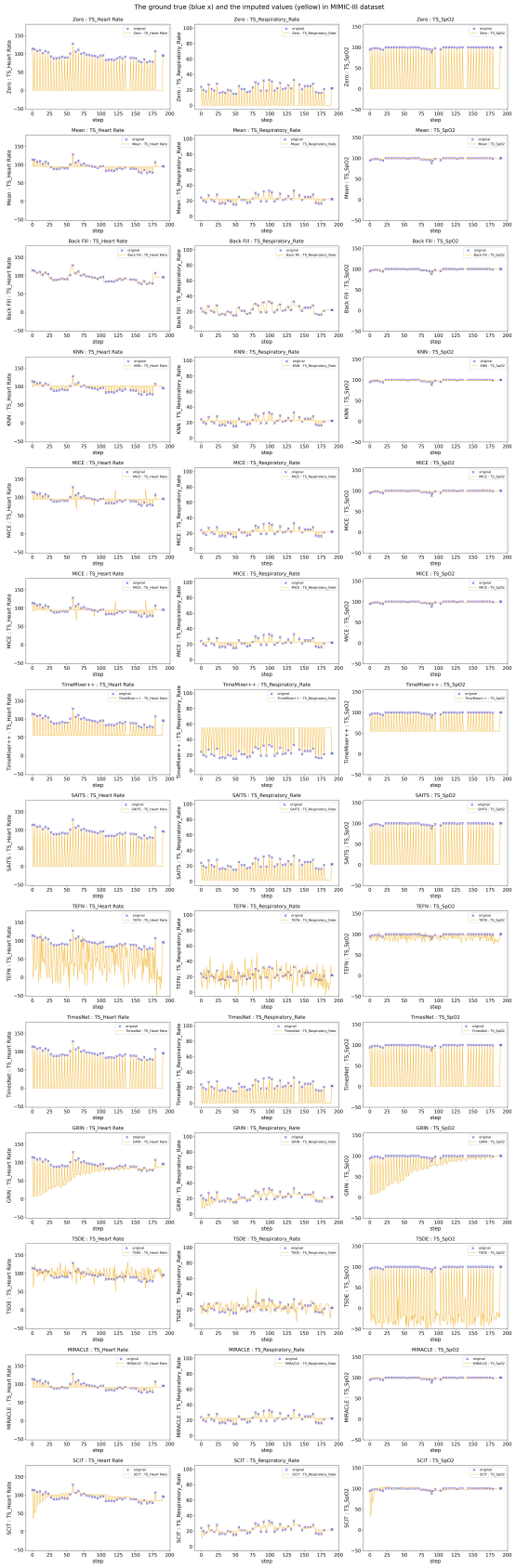

Figure 1: The comparison of various imputation methods on the MIMIC-III dataset for clinical time series data with the 'ground true' values (blue x) against the 'imputed values' (yellow line) for Heart Rate, Respiratory Rate, and SpO2.

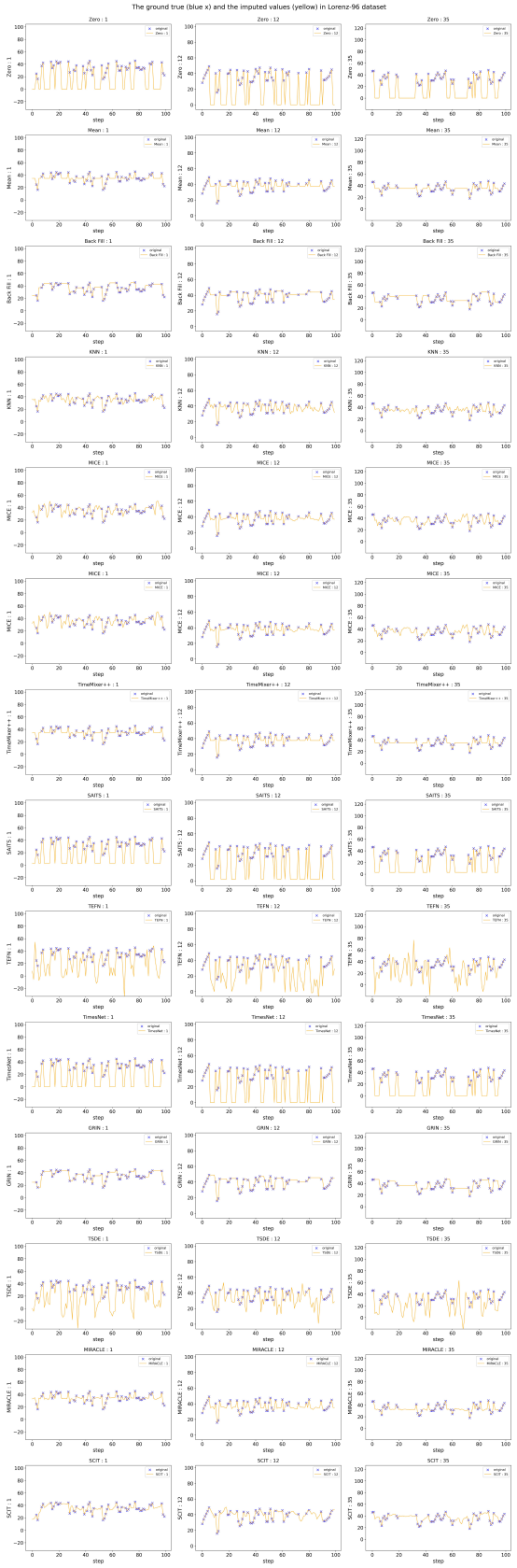

Figure 2: The comparison of various imputation methods on the Lorenz-96 dataset for clinical time series data with the 'ground true' values (blue x) against the 'imputed values' (yellow line) for three features.

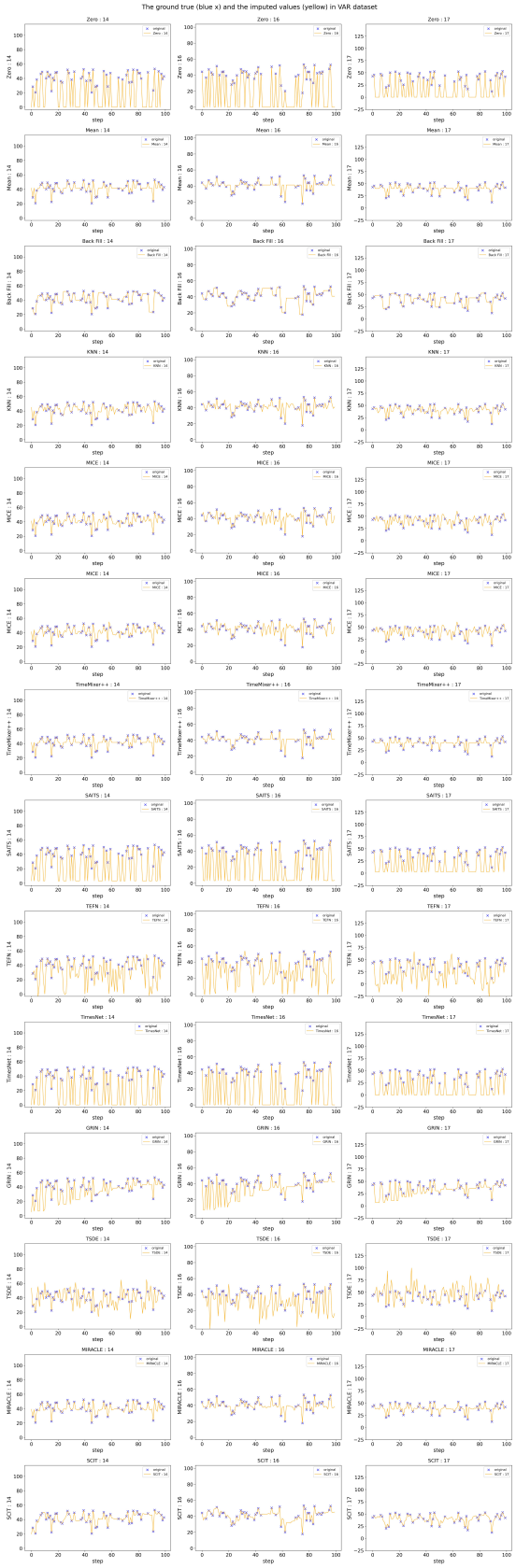

Figure 3: The comparison of various imputation methods on the VAR dataset for clinical time series data with the 'ground true' values (blue x) against the 'imputed values' (yellow line) for three features.

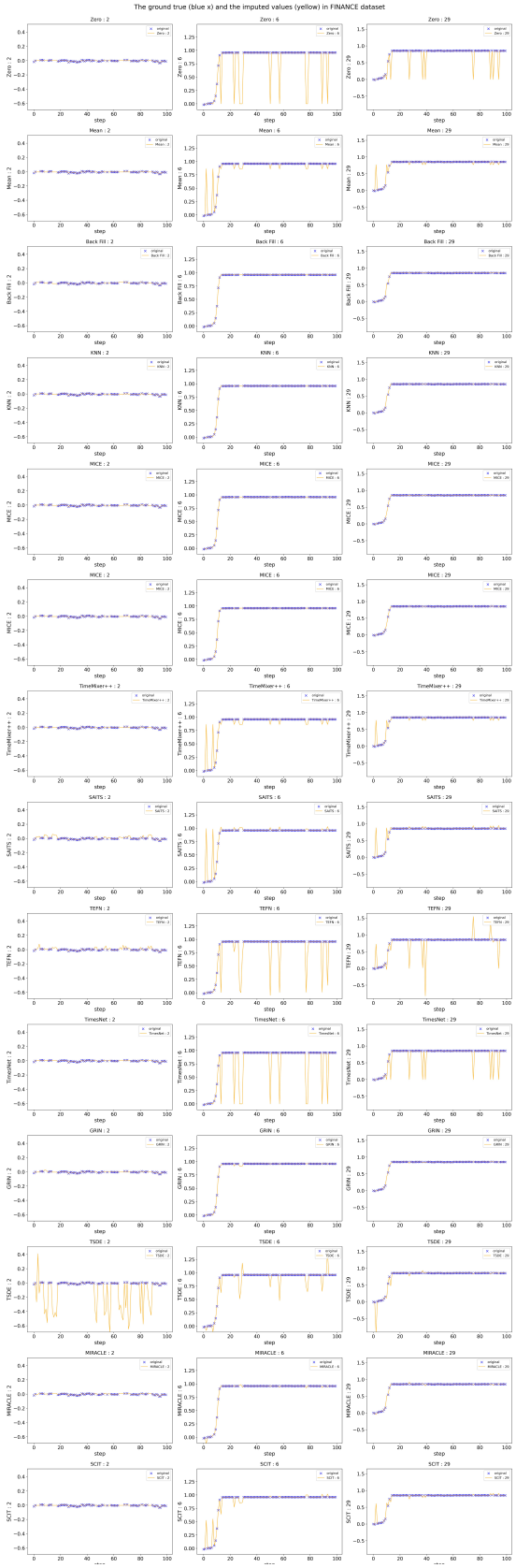

Figure 4: The comparison of various imputation methods on the FINANCE dataset for clinical time series data with the 'ground true' values (blue x) against the 'imputed values' (yellow line) for three features.

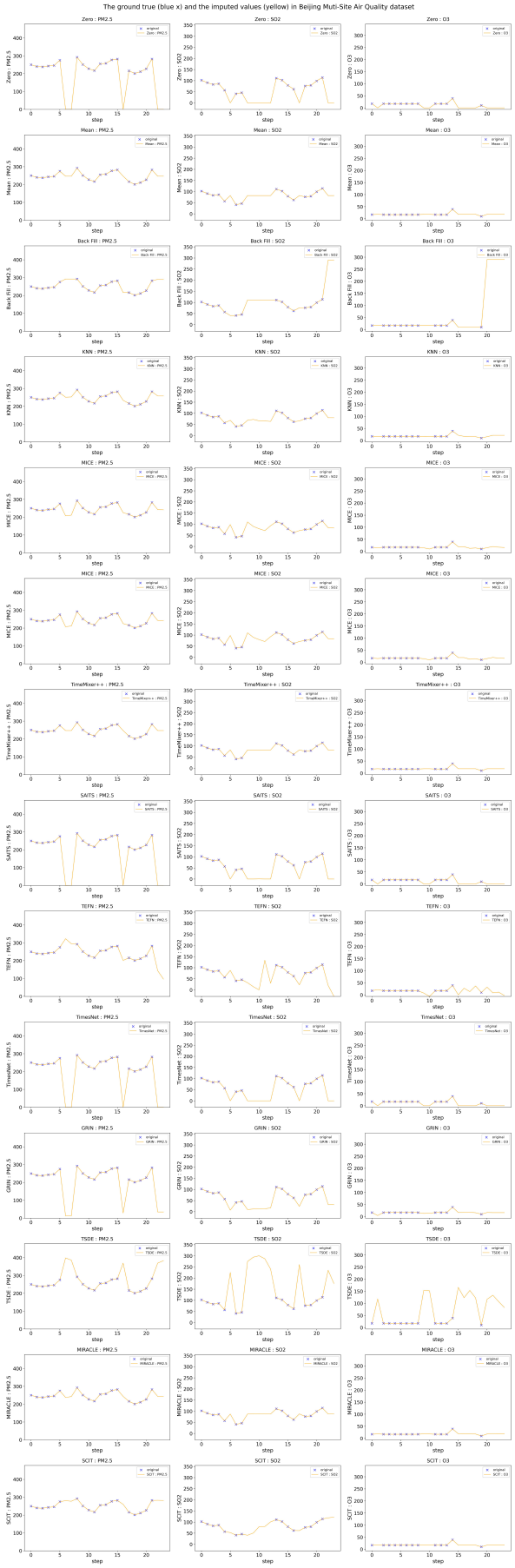

Figure 5: The comparison of various imputation methods on the Beijing Multi-Site Air-Quality dataset for clinical time series data with the 'ground true' values (blue x) against the 'imputed values' (yellow line) for PM2.5, SO_2 Rate, and O_3.