Journal of Statistical Software

MMMMMM YYYY, Volume VV, Issue II. *http://www.jstatsoft.org/*

Efficient Multiple Imputation for Diverse Data in Python and R: MIDASpy and rMIDAS

Ranjit Lall
University of Oxford

Thomas Robinson
London School of Economics
and Political Science

Abstract

This paper introduces software packages for efficiently imputing missing data using deep learning methods in Python (**MIDASpy**) and R (**rMIDAS**). The packages implement a recently developed approach to multiple imputation known as MIDAS, which involves introducing additional missing values into the dataset, attempting to reconstruct these values with a type of unsupervised neural network known as a denoising autoencoder, and using the resulting model to draw imputations of originally missing data. These steps are executed by a fast and flexible algorithm that expands both the quantity and the range of data that can be analyzed with multiple imputation. To help users optimize the algorithm for their particular application, **MIDASpy** and **rMIDAS** offer a host of user-friendly tools for calibrating and validating the imputation model. We provide a detailed guide to these functionalities and demonstrate their usage on a large real dataset.

Keywords: missing data, multiple imputation, machine learning, Python, R.

1. Introduction

Approaches to analyzing data with missing values, one of the most common methodological challenges facing empirical researchers, have become increasingly sophisticated in recent years. Conscious of the biases and inefficiencies of traditional ad-hoc methods, such as deleting or guessing missing data, analysts across a wide range of disciplines have been turning to the general-purpose framework of *multiple imputation*. Recommended by a well-developed body of statistical theory (e.g., Rubin 1987, 1996; Little and Rubin 2002), multiple imputation involves replacing each missing element with several values that preserve relationships within the observed data while representing uncertainty about the correct imputation model.

This paper introduces a pair of software packages that deliver a high-performance implemen-

tation of multiple imputation in Python (**MIDASpy**) and R (**rMIDAS**).[1] The packages leverage a type of unsupervised neural network known as a denoising autoencoder, which is designed to efficiently learn latent representations of data for the purpose of dimensionality reduction (or feature selection and extraction in machine learning terminology) (Vincent, Larochelle, Bengio, and Manzagol 2008; Vincent, Larochelle, Lajoie, Bengio, and Manzagol 2010). Denoising autoencoders corrupt a subset of observed values via the injection of stochastic noise and attempt to reconstruct them through a series of nested nonlinear transformations. The packages repurpose denoising autoencoders for multiple imputation by treating missing values as an additional portion of corrupted data and drawing imputations from a model trained to minimize the reconstruction error on the originally observed portion. They thus employ an effectively nonparametric imputation model that imposes constraints only on the distribution of *functions* that are consistent with the data, giving it the flexibility to capture relationships of widely varying complexity. This method, which was proposed recently by Lall and Robinson (2022), is known as MIDAS (Multiple Imputation with Denoising Autoencoders), from which the packages derive their names.

MIDASpy and **rMIDAS** offer a suite of easy-to-use computational tools for implementing MIDAS — to our knowledge, the first full-featured, open-source software for performing multiple imputation with neural network technology. The software's principal advantages are its accuracy, efficiency, end-to-end capabilities, and availability in two major programming languages. As summarized in Table 1, multiple imputation software has traditionally been based on parametric imputation methods, employed expectation-maximization (EM) or Markov chain Monte Carlo (MCMC) algorithms, and been limited to R. Popular examples include **norm** (Schafer and Olsen 1998), **mice** (van Buuren and Groothuis-Oudshoorn 2011), **mi** (Su, Gelman, Hill, and Yajima 2011), and **Amelia** (Honaker, King, and Blackwell 2011). Although suitable for many applications, EM and MCMC imputation algorithms can suffer from convergence problems and suboptimal imputation accuracy when applied to large datasets with complex features, such as high dimensionality, severe nonlinearities, and unconventional functional forms — features that are becoming common in the emerging era of "Big Data" (Honaker and King 2010; Lall and Robinson 2022).

In recent years, software based on more flexible nonparametric imputation methods, most notably random forests, has become available in Python as well as R, providing accuracy gains in complex applications. Among the most widely used are **MissForest** (Stekhoven 2013), **miceRanger** (Wilson 2020), **miceforest** Wilson (2021), and **sklearn.impute**, **scikit-learn**'s imputation library (Pedregosa, Varoquaux, Gramfort, Michel, Thirion, Grisel, Blondel, Prettenhofer, Weiss, Dubourg, Vanderplas, Passos, Cournapeau, Brucher, Perrot, and Duchesnay 2011).[2] However, these packages can also exhibit lengthy runtimes with large (in particular wide) datasets because, similarly to MCMC-based approaches, they iteratively impute missing values in one incomplete variable at a time. In addition, they rarely cover all stages of the typical multiple imputation workflow.[3] For instance, they typically lack functionalities for combining the results of statistical models estimated post-imputation.

[1] The software was developed by the authors and Alex Stenlake. For more information, see `https://github.com/MIDASverse`. As discussed below, each package is registered in its programming language's official software repository.

[2] **miceRanger** and **miceforest** implement the same method in R and Python, respectively. **sklearn.impute** is designed primarily for single imputation; multiple imputation is implemented by enabling the experimental `IterativeImputer` class and setting `sample_posterior = True`.

[3] A detailed comparison of the stages covered by the packages listed in Table 1 is provided in Section 4.3.

Software package	Imputation model	Algorithm	Implementation	
			R	Python
Amelia	Multivariate normal	EM	✓	
mi	Conditional	MCMC (chained equations)	✓	
mice	Conditional	MCMC (chained equations)	✓	
norm	Multivariate normal	MCMC (imputation-posterior)	✓	
MissForest	Nonparametric	Iterative random forests	✓	
miceRanger/miceforest	Nonparametric	Iterative random forests	✓	✓
sklearn.impute	Variable	MCMC (chained equations)		✓
MIDASpy/rMIDAS	Nonparametric	Autoencoder neural network	✓	✓

Table 1: Comparison of **MIDASpy/rMIDAS** with popular multiple imputation software in R and Python

This paper provides an overview and demonstration of **MIDASpy** and **rMIDAS**'s key functionalities: instantiating, building, and training the imputation model; calibrating model hyperparameters and validating outputs; and storing, exporting, and analyzing completed datasets.[4] Details on the statistical theory underlying MIDAS, as well as systematic evidence of **MIDASpy** and **rMIDAS**'s accuracy and efficiency across diverse data types relative to existing multiple imputation software — including several of the packages listed in Table 1 — are presented in Lall and Robinson (2022).

The core code base for both packages is written in Python using the efficient and flexible architecture of the **TensorFlow** library, whose capacity to operate at scale and in heterogeneous environments enables the software to expand both the quantity and the range of data that can be analyzed with multiple imputation. **TensorFlow** supports high degrees of parallelization as well as graphics processing unit (GPU) computation, making it possible for users to further accelerate the imputation process.

rMIDAS wraps the key functionalities of **MIDASpy** into a package tailored specifically for R users. **rMIDAS** directly interfaces with **MIDASpy** without disrupting users' existing workflows or compromising the speed and scalability of the MIDAS algorithm. To the contrary, the package takes advantage of R-specific tools for efficiently processing and structuring data to further accelerate computation.

In the next section, we provide an overview of the MIDAS approach and the algorithm we have developed for implementing it. Section 3 describes how to install **MIDASpy** in Python and its main functions and arguments. Section 4 presents an analogous description of **rMIDAS**, including the features it inherits from **MIDASpy** as well as the unique tools it offers for enhancing the efficiency of the imputation process. Section 5 presents an applied demonstration of the software, using **MIDASpy** and **rMIDAS** to impute missing values in a large-scale electoral dataset. Section 6 discusses three of the software's diagnostic tools for model calibration and validation, illustrating them on the same dataset.[5] Section 7 concludes and outlines future plans for the software's development.

[4]Hyperparameters are features of neural networks that are manually specified by the analyst rather than learned during training.

[5]To comply with the *Journal of Statistical Software*'s requirement to provide a single replication script, we generate all figures using **rMIDAS** and R, providing the code for producing equivalent results with **MIDASpy** in the text.

2. The MIDAS approach

Multiple imputation always involves (1) filling in missing values with M independently drawn imputations that preserve relationships within the observed data, (2) estimating parameters of interest with the M completed datasets, and (3) combining the M separate parameter estimates using a set of simple rules (described in Section 3.3) that exploit variation across the datasets to capture uncertainty about the correct imputation model. Approaches to performing the first step, however, vary widely in their assumptions about the structure and distribution of the data (Lall 2016). Traditionally, the most popular approaches have been modeling the data as a sample from a multivariate normal distribution (e.g., **Amelia**, **norm**) and estimating each variable's distribution conditionally on all others (e.g., **mi**, **mice**).

This section describes the neural network-based approach to multiple imputation that underlies **MIDASpy** and **rMIDAS**, before outlining the computational procedure we have developed to implement this approach. Our exposition follows the standard linear algebraic notation used in the neural network literature: italicized upper-case symbols denote random vectors (e.g., X); bold lower-case symbols ($\mathbf{x}$) denote ordinary column vectors, i.e., realizations of random vectors; bold upper-case symbols (e.g., $\mathbf{W}$) denote matrices; and superscripts in parentheses index hidden layers of a neural network. We define $\mathbf{D} = \{\mathbf{D}_{obs}, \mathbf{D}_{mis}\}$ as an input dataset (including all variables to feature in subsequent analyses) in which $\mathbf{D}_{obs}$ is observed and $\mathbf{D}_{mis}$ is missing.

2.1. Adapting neural networks for multiple imputation

The MIDAS approach makes use of denoising autoencoders, a type of unsupervised neural network developed to learn informative lower-dimensional representations of data. Denoising autoencoders are an extension of traditional autoencoders, which are composed of two sequential parts. First, an *encoder* deterministically maps an input vector $\mathbf{x}$ to a lower-dimensional representation $\mathbf{y}$ by passing it through a series of shrinking hidden layers ending with a "bottleneck" layer (indexed by B):

$$\mathbf{y} = f_\theta(\mathbf{x}) = \sigma(\mathbf{W}^{(B)}[\ldots[\sigma(\mathbf{W}^{(2)}[\sigma(\mathbf{W}^{(1)}\mathbf{x} + \mathbf{b}^{(1)})] + \mathbf{b}^{(2)})]\ldots] + \mathbf{b}^{(B)}) \tag{1}$$

where $\mathbf{W}$ is a matrix of weights connecting the nodes between hidden layers, $\mathbf{b}$ is a vector of biases for each layer, and σ is a nonlinear activation function. Second, a *decoder* maps $\mathbf{y}$ back to a reconstruction $\mathbf{z}$ with the same probability distribution and dimensions as $\mathbf{x}$ by passing it through a parallel series of expanding hidden layers finishing with the output layer:

$$\mathbf{z} = g_{\theta'}(\mathbf{y}) = \Phi(\mathbf{W}'^{(H)}[\ldots[\sigma(\mathbf{W}'^{(B+2)}[\sigma(\mathbf{W}'^{(B+1)}\mathbf{y} + \mathbf{b}'^{(B+1)})] + \mathbf{b}'^{(B+2)})]\ldots] + \mathbf{b}'^{(H)}) \tag{2}$$

where Φ is a final activation function that converts outputs to their appropriate distribution.[6] Weights are adjusted by the method of backpropagation (Rumelhart, Hinton, and Williams 1986b,a) to minimize a loss function $L(\mathbf{x}, \mathbf{z})$ measuring the distance between $\mathbf{x}$ and $\mathbf{z}$, that is, the average reconstruction error. $\mathbf{z}$ can thus be interpreted as the parameters of a distribution that generates $\mathbf{x}$ with a high probability.

Traditional autoencoders with multiple hidden layers run the risk of simply learning the identity function and hence becoming redundant. Denoising autoencoders seek to avoid this

[6] The prime symbols in Equation 2 distinguish decoder parameters from encoder parameters.

possibility — while enabling the extraction of even more robust features — by partially corrupting inputs via the injection of stochastic noise: $\mathbf{x} \rightarrow \tilde{\mathbf{x}} \sim q_D(\mathbf{x}|\tilde{\mathbf{x}})$. As in a traditional autoencoder, the corrupted input is then mapped to a hidden representation $\mathbf{y} = f_\theta(\tilde{\mathbf{x}})$, from which a "repaired" version $\mathbf{z} = g_{\theta'}(\mathbf{y})$ is constructed. Unlike before, however, $\mathbf{z}$ is now a deterministic function of $\tilde{\mathbf{x}}$ rather than of $\mathbf{x}$. Since the corruption process typically involves forcing a random subset of inputs to 0, denoising autoencoders effectively perform a form of imputation: predicting corrupted (missing) elements based on relationships among uncorrupted (observed) elements.[7] In other words, missing values can be seen as a special case of corrupted or noisy input data.

Building on this insight, MIDAS adapts denoising autoencoders for the task of multiple imputation through two key modifications. First, it sets all missing values to 0 and trains the network to predict corrupted elements that were both originally missing ($\tilde{\mathbf{x}}_{\text{mis}}$) and originally observed ($\tilde{\mathbf{x}}_{\text{obs}}$) using a loss function that only measures the reconstruction error on the latter. Second, as a further bulwark against overfitting, MIDAS regularizes the encoder with the complementary technique of dropout, which extends the corruption process to its hidden layers (Hinton, Srivastava, Krizhevsky, Sutskever, and Salakhutdinov 2012; Srivastava, Hinton, Krizhevsky, Sutskever, and Salakhutdinov 2014). This is implemented by multiplying outputs from these layers by a Bernoulli vector $\mathbf{v}$ that takes a value of 1 with probability p: $\tilde{\mathbf{y}}^{(h)} = \mathbf{v}^{(h)}\mathbf{y}^{(h)}$, $\mathbf{v}^{(h)} \sim \text{Bernoulli}(p)$. At test time, *multiple* imputations are generated by sampling M "thinned" networks.

This procedure, Gal and Ghahramani (2016) show, is equivalent to Bayesian variational approximation of a Gaussian process, a well-known distribution over possible functions.[8] Rather than assuming a joint distribution of the data, therefore, MIDAS only places a prior on the distribution of *functions* that can characterize the data. As there is no limit to the potential number of parameters in a Gaussian process model, MIDAS employs an effectively nonparametric imputation model that (with appropriate specification) can estimate any continuous function arbitrary well.

The encoding portion of a MIDAS network can thus be described as:

$$\tilde{\mathbf{y}} = f_\theta(\tilde{\mathbf{x}}) = \sigma(\mathbf{W}^{(B)}\mathbf{v}^{(B)}[\ldots[\sigma(\mathbf{W}^{(2)}\mathbf{v}^{(2)}[\sigma(\mathbf{W}^{(1)}\tilde{\mathbf{x}} + \mathbf{b}^{(1)})] + \mathbf{b}^{(2)})]\ldots] + \mathbf{b}^{(B)}). \tag{3}$$

The decoder takes the form:

$$\mathbf{z} = g_{\theta'}(\tilde{\mathbf{y}}) = \Phi(\mathbf{W}^{(H)\prime}[\ldots[\sigma(\mathbf{W}^{(B+2)\prime}[\sigma(\mathbf{W}^{(B+1)\prime}\tilde{\mathbf{y}} + \mathbf{b}^{(B+1)\prime})] + \mathbf{b}^{(B+2)\prime})]\ldots] + \mathbf{b}^{(H)\prime}) \tag{4}$$

where $\mathbf{z}$ now represents a fully observed vector containing predictions of $\tilde{\mathbf{x}}_{\text{obs}}$ and $\tilde{\mathbf{x}}_{\text{mis}}$. To produce a completed dataset, predictions of $\tilde{\mathbf{x}}_{\text{mis}}$ are substituted for $\mathbf{x}_{\text{mis}}$ in the input dataset $\mathbf{D}$.

By default, MIDAS employs exponential linear unit (ELU) activation functions, which are known to facilitate fast and accurate training in deep neural networks:

$$\sigma(\alpha, \mathbf{m}^{(h)}) = \begin{cases} \alpha(e^{\mathbf{m}^{(h)}} - 1) & \text{for } \mathbf{m}^{(h)} \leq 0 \\ \mathbf{m}^{(h)} & \text{for } \mathbf{m}^{(h)} > 0 \end{cases} \tag{5}$$

[7] The choice of 0 for corrupted data points is not substantively important (any value would work); it is a popular choice mainly because it is often close to the "true" value being estimated, minimizing the adjustment to network parameters in training and hence accelerating model convergence.

[8] GPs are a generalization of multivariate normal distributions over finite dimensional vectors to infinite dimensionality. Specifically, a finite vector $\mathbf{x}$ is a GP if the vector of function values $\mathbf{f} = f(x^1), f(x^2), \ldots, f(x^n)$ follows a multivariate normal distribution.

where $\mathbf{m}^{(h)}$ represents the output from layer $h-1$ and α denotes a positive constant initialized as 1. The final activation function is chosen according to the distribution of the input data. Following standard practice, MIDAS assigns identity, logistic, and softmax (i.e., normalized exponential) functions to continuous, binary, and categorical variables, respectively:

$$\Phi(\mathbf{m}_j^{(H)}) = \begin{cases} \mathbf{m}^{(H)_j} & \text{if } \mathbf{x} \text{ is continuous} \\ \frac{1}{1+e^{-\mathbf{m}_j^{(H)}}} & \text{if } \mathbf{x} \text{ is binary} \\ \frac{e^{\mathbf{m}_j^{(H)}}}{\sum_{k=1}^{K} e^{\mathbf{m}_k^{(H)}}} & \text{if } \mathbf{x} \text{ is categorical} \end{cases} \tag{6}$$

where j indexes realized components of $\mathbf{m}^{(H)}$.

As in a regular denoising autoencoder, the loss function penalizes greater distances between $\mathbf{x}$ and $\mathbf{z}$: $L(\mathbf{x}, \mathbf{z})$. Since we are only interested in the reconstruction error on predictions of originally *observed* values, however, it is multiplied by a missingness indicator vector $\mathbf{r}$, i.e., a vector the same length as $\mathbf{x}$ whose elements are a 1 if the corresponding entry in $\mathbf{x}$ is missing and a 0 if it is observed. Specifically, we assign root mean squared error (RMSE) and cross-entropy loss functions for continuous and binary or categorical variables, respectively:[9]

$$L(\mathbf{x}, \mathbf{z}, \mathbf{r}) = \begin{cases} [\frac{1}{J} \sum_{j=1}^{J} \mathbf{r}_j (\mathbf{x}_j - \mathbf{z}_j)^2]^{\frac{1}{2}} & \text{if } \mathbf{x} \text{ is continuous} \\ -\frac{1}{J} \sum_{j=1}^{J} \mathbf{r}_j [\mathbf{x}_j \log \mathbf{z}_j + (1 - \mathbf{x}_j) \log(1 - \mathbf{z}_j)] & \text{if } \mathbf{x} \text{ is binary/categorical.} \end{cases} \tag{7}$$

2.2. Algorithm

The algorithm underlying **MIDASpy** and **rMIDAS** takes an incomplete input dataset $\mathbf{D}$ and uses it to initialize, build, and train an imputation model based on the MIDAS approach, from which it draws imputations to generate M completed versions of this dataset. The algorithm, which is summarized in pseudocode form in Table 2, has three principal stages: data preparation, model training, and imputation.

In the first stage, $\mathbf{D}$ is formatted for training. A missingness indicator matrix $\mathbf{R}$ (i.e., a matrix equivalent of $\mathbf{r}$) is constructed for $\mathbf{D}$, enabling the algorithm to later distinguish between $\mathbf{x}_{\text{mis}}$ and $\mathbf{x}_{\text{obs}}$. All elements of $\mathbf{D}_{\text{mis}}$ are set to 0. A MIDAS network is then initialized according to the dimensions of $\mathbf{D}$. To prevent backpropagated gradients from vanishing or exploding, the network is parameterized using a variant of Xavier Initialization (Glorot and Bengio 2010), which draws weights from a truncated normal distribution:

$$\mathbf{W}^{(h)} \sim (0, \frac{1}{\sqrt{n^{(h)} + n^{(h+1)}}}). \tag{8}$$

where $n^{(h)}$ is the size of layer $h-1$ (i.e., the number of columns of $\mathbf{W}^{(h)}$).

In the training stage, five steps are repeated (see Figure 1 for a visual schematic):

1. $\mathbf{D}$ and $\mathbf{R}$ are shuffled and divided row-wise into paired mini-batches $\mathcal{B}_1, \mathcal{B}_2, \ldots, \mathcal{B}_n$. This step reduces training time — storing all training data in memory and calculating

[9] While mean squared error (MSE) is more commonly used for continuous variables, we have found RMSE to yield more stable and robust training. A small constant is added to the cross-entropy loss function to prevent log(0) values.

Data : Incomplete dataset $\mathbf{D}$
Result : M completed datasets
Parameters : Network weights $\mathbf{W}$
Hyperparameters: Network structure, no. training epochs t, corruption proportion p, mini-batch size s, learning rate γ, weight decay rate λ

begin
  Generate missingness indicator matrix $\mathbf{R}$;
  Set missing values in $\mathbf{D}$ to 0: $\mathbf{D} \rightarrow \mathbf{D}[\mathbf{R} = 0] = 0$;
  Initialize DA based on dimensions of $\mathbf{D}$ (with Xavier Initialization);
  while *epoch* $< t$ **do**
    Shuffle $\mathbf{D}$ and $\mathbf{R}$ in same order;
    Slice $\mathbf{D}$ row-wise into n mini-batches $\mathcal{B}_1, \mathcal{B}_2, ..., \mathcal{B}_n$ of size s;
    Partially corrupt inputs: $\tilde{\mathbf{x}} = [v^{(0,1)}\mathcal{B}_1, ..., v^{(0,n)}\mathcal{B}_n]$, where $\mathbf{v}^{(0)} \sim \text{Bernoulli}(p = 0.8)$;
    Partially corrupt outputs of hidden nodes (dropout): $\tilde{\mathbf{y}}^{(h)} = \mathbf{y}^{(h)}\mathbf{v}^{(h)}$, where $\mathbf{v}^{(h)} \sim \text{Bernoulli}(p = 0.5)$;
    Perform forward pass through entire network;
    Calculate reconstruction error against $\tilde{\mathbf{x}}_{\mathbf{obs}}$: $E = L(\mathbf{x}, \mathbf{z}, \mathbf{r}) + \lambda||\mathbb{E}[\mathbf{W}]||^2$, where $\mathbf{r}$ is missingness indicator vector;
    Backpropagate loss through network to find error gradients: $\frac{\partial E}{\partial \mathbf{W}^{(h)}}$;
    Update weights for next epoch: $\Delta\mathbf{W}^{(h)} = -\gamma\frac{\partial E}{\partial \mathbf{W}^{(h)}}$;
  end
  repeat
    Pass $\mathbf{D}$ into trained DA;
    Construct completed dataset using predictions of $\tilde{\mathbf{x}}_{\text{mis}}$;
  until *M completed datasets generated*;
end

Table 2: MIDAS algorithm pseudocode

loss for the whole sample are memory-intensive — and increases the frequency of model updates, ensuring more robust convergence.

2. Mini-batch inputs are partially corrupted via multiplication by a vector $\mathbf{v}^{(0)} \sim \text{Bernoulli}(p = 0.8)$ (resulting in a corruption rate of 20%).

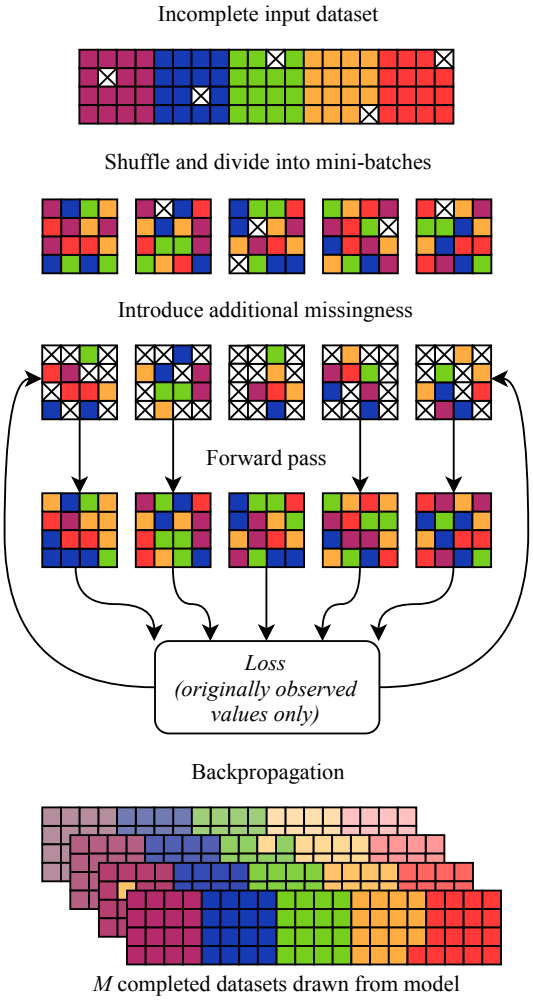
3. To implement dropout, outputs from half of the nodes in hidden layers of the network are multiplied by another vector $\mathbf{v}^{(h)} \sim \text{Bernoulli}(p = 0.5)$.

4. Data are stochastically passed forward through the "thinned" network to produce the input reconstruction $\mathbf{z}$, and error is calculated using the loss functions defined in Equation 7.[10]

5. Loss values are aggregated into a single term and backpropagated through the network,

[10] As an additional check against overfitting, we include in these functions a weight decay regularization term λ that penalizes the squared sum of weights.

Figure 1: Schematic of the MIDAS training process



with the resulting error gradients used to adjust weights for the next epoch. The size of this adjustment is controlled by a learning rate hyperparameter γ, by which loss function derivatives are multiplied. Mini-batch stochastic gradient descent is implemented with the widely used Adam algorithm (Kingma and Ba 2015).

Finally, a stochastic forward pass is conducted through the whole of $\mathbf{D}$, and corrupted values are reconstructed according to the data manifold learned during training. A completed dataset is formed by replacing $\mathbf{D}_{\text{mis}}$ with predictions of the corresponding elements from the MIDAS network's output. This last stage is repeated M times.

2.3. Memory management

Analyzing large incomplete datasets almost inevitably involves memory-intensive computation, which can result in slower task execution and more frequent crashes. The MIDAS algorithm cannot, of course, eliminate the need for sufficient memory to handle such datasets in the first place, and users may run up against the limits of available processing power in

sizable applications.

Even so, the algorithm can meaningfully reduce random-access memory (RAM) usage and hence the likelihood of computational problems. The nodes and layers of a MIDAS network are fixed and require relatively little memory to perform their functions. For example, prior to imputation (but after one-hot encoding), the electoral dataset that we analyze in Section 5 uses 925MB of RAM. The MIDAS model we train on the data, in contrast, consumes approximately 30MB of RAM. Importantly, since the algorithm scales linearly, users can easily — and affordably — "offshore" model training to a virtual server with greater memory capacity.[11]

Furthermore, our software for implementing the algorithm is designed to be as efficient as possible and includes tools for further minimizing memory intensity. For instance, **MIDASpy** offers a function for accessing completed datasets individually rather than concurrently (detailed in Section 3.2.5 and demonstrated in Section 5.1.2), while **rMIDAS** leverages a variety of fast data-formatting and -shaping capabilities now available in R (described in Section 4.2.1). As mentioned earlier, the software's speed and scalability have been shown to compare favorably with popular alternatives (Lall and Robinson 2022).

3. Python interface: The MIDASpy package

3.1. Installing MIDASpy

MIDASpy is available on the Python Package Index (PyPI) (`https://pypi.org/project/MIDASpy`) and can thus be installed via the command line (in Python 3.5 or higher).

```
>>> pip install MIDASpy
```

MIDASpy requires the following Python libraries:

- **Matplotlib** (Hunter 2007).
- **NumPy** (>= 1.5) (Oliphant 2006).
- **pandas** (>= 0.19) (McKinney 2010).
- **TensorFlow** (>= 1.10, >= 2.2) (Abadi, Agarwal, Barham, Brevdo, Chen, Citro, Corrado, Davis, Dean, Devin, Ghemawat, Goodfellow, Harp, Irving, Isard, Jia, Jozefowicz, Kaiser, Kudlur, Levenberg, Mane, Monga, Moore, Murray, Olah, Schuster, Shlens, Steiner, Sutskever, Talwar, Tucker, Vanhoucke, Vasudevan, Viegas, Vinyals, Warden, Wattenberg, Wicke, Yu, and Zheng 2016).[12]

3.2. Main functions and arguments

The typical workflow for users of **MIDASpy** is the following:

[11] For example, this paper's final results were generated using Amazon Web Services (AWS) servers. Guidance on setting up a server instance is provided on **MIDASpy**'s webpage and as a vignette in **rMIDAS**.

[12] When using version 2.2 and above of **TensorFlow**, **MIDASpy** also requires **TensorFlow Addons** (>=0.11).

1. Preprocess the input dataset.
2. Initialize the neural network.
3. Build the imputation model.
4. Train the imputation model.
5. Generate the completed datasets.

In the rest of this section, we describe the functions and associated arguments required in each stage. Unless otherwise specified, these functions must be called in the order they appear below; failure to do so will raise an error. For the sake of brevity, we exclude nonessential functions, documentation on which can be found on the software's homepage. We postpone the discussion of diagnostic tools, which are optional but strongly recommended, to Section 6.

Preprocess the input dataset

To ensure that their incomplete input dataset is in the appropriate format for **MIDASpy**, users must begin by undertaking two preprocessing steps. First, binary variables must be converted to dummy (i.e., 0/1) form without altering their missingness pattern. For **pandas** DataFrames, this can be done using **MIDASpy**'s `.binary_conv()` function, which is applied on a variable-by-variable basis (generating a variable of equal length containing 1s and 0s).

- `x`: *pd.Series array.* An indexable array containing only two unique values.

Second, categorical variables must be one-hot encoded (i.e., divided into separate dummy variables for each unique class) while again preserving their missingness pattern. One-hot encoding functions are available in several Python libraries, including **pandas**, **scikit-learn**, and **dask** (Dask Development Team 2016). As these functions typically re-code missing values as 0, we have included in **MIDASpy** the `.cat_conv()` function, which wraps **pandas**'s `get_dummies()` function but copies the location of such values and reinstates them after one-hot encoding. In addition, `.cat_conv()` stores the names of the one-hot encoded variables, generating a list of lists that can be subsequently passed to the **MIDASpy** imputation model.

- `cat_data`: *pd.DataFrame.* A dataframe containing only the categorical columns to be one-hot encoded.

Finally, we also recommend rescaling continuous variables between 0 and 1, which tends to improve the MIDAS algorithm's convergence. This can be done using, for instance, **scikit-learn**'s `MinMaxScaler()` function.

Initialize the neural network

The MIDAS network is initialized with the `Midas()` function. The separation between the initialization and model-building stages of the imputation process allows for the usage of out-of-memory datasets, a useful feature for Big Data applications that stretch in-memory capacity. The main network hyperparameters are specified in this function.

- `layer_structure`: *List of integers.* The number of nodes in each layer of the network (default = [256, 256, 256], denoting a three-layer network with 256 nodes per layer). Larger networks can learn more complex data structures but require longer training and are more prone to overfitting. We discuss this tradeoff in greater detail in Section 6.2.

- `learn_rate`: *Float.* The learning rate γ (default = 0.0001), which controls the size of the adjustment to weights and biases in each training epoch. In general, higher values reduce training time at the expense of less accurate results (see Section 6.2

- `input_drop`: *Float between 0 and 1.* The probability of corruption for input columns in training mini-batches (default = 0.8). Higher values increase training time but reduce the risk of overfitting. In our experience, values between 0.7 and 0.95 deliver the best balance between speed and accuracy.

- `train_batch`: *Integer.* The number of observations in training mini-batches (default = 16). Common choices are 8, 16, 32, 64, and 128; powers of 2 tend to enhance memory efficiency. In general, smaller batches lead to faster convergence at the cost of greater noise and thus less accurate estimates of the error gradient. Where memory management is a concern, they should be favored.

- `savepath`: *String.* The location to which the trained model will be saved.

- `seed`: *Integer.* The value to which Python's pseudo-random number generator is initialized. This enables users to reproduce data shuffling, weight and bias initialization, and missingness indicator vectors. As discussed in Section 3.4, however, it is not possible to perfectly replicate completed datasets.

- `loss_scale`: *Float.* A constant by which the RMSE loss functions are multiplied (default = 1). This hyperparameter performs a similar function to the learning rate. If loss during training is very large, increasing its value can help to prevent overtraining.

- `init_scale`: *Float.* The numerator of the variance component of Equation 8 (default = 1). In very deep networks, higher values may help to prevent extreme gradients (though this problem is less common with ELU activation functions).

- `softmax_adj`: *Float.* A constant by which the cross-entropy loss functions are multiplied (default = 1). This hyperparameter is the equivalent of `loss_scale` for categorical variables.

- `vae_layer`: *Boolean.* Specifies whether to include a variational autoencoder layer in the network (default = `False`), one of the key diagnostic tools in **MIDASpy**. If set to `true`, variational autoencoder hyperparameters must be specified via a number of additional arguments. This functionality is discussed in detail in Section 6.3.

Build the imputation model

The function `.build_model()` constructs the imputation model. The main arguments declare the incomplete input dataset and the list of binary and categorical variables (to ensure correct assignment of loss functions).

- `imputation_target`: *DataFrame.* The input dataset. Upon being read in, the dataset will be appropriately formatted and stored for training.
- `binary_columns`: *List of names.* A list of all binary variables.
- `softmax_columns`: *List of lists.* The outer list should include all (non-binary) categorical variables; each inner list should contain the set of mutually exclusive classes in a given variable.
- `unsorted`: *Boolean.* Specifies whether the input dataset is unordered in terms of variable type (default = `True`, denoting no ordering). If `False`, `binary_columns` and `softmax_columns` should be a list of integers denoting shape attributes for each category.
- `additional_data`: *DataFrame.* Data that should be included in the imputation model but are not required for later analysis. Such data will not be formatted, rearranged, or included in the loss functions, reducing training time.
- `verbose`: *Boolean.* Specifies whether to print messages to the terminal (default = `True`).

Train the imputation model

Once the imputation model has been constructed, network parameters are optimized with the `.train_model()` function. This function automatically saves the model after training.

- `training_epochs`: *Integer.* The number of complete passes through the network during training (default = 100).
- `verbose`: *Boolean.* Specifies whether to print messages to the terminal during training, including loss values (default = `True`).
- `verbosity_ival`: *Integer.* The number of training epochs between messages (default = 1).
- `excessive`: *Boolean.* Specifies whether to print loss for each mini-batch to the terminal (default = `False`), which can help with troubleshooting.

Generate the completed datasets

Finally, the M completed datasets are generated with the `.generate_samples()` function. These datasets are stored in `.output_list`, from which they can be accessed in any order. If a model has been pre-trained, `.generate_samples()` can be called in the absence of `.build_model()`.

- `m`: *Integer.* The number of completed datasets to produce (default = 50).
- `verbose`: *Boolean.* Specifies whether to print messages to the terminal (default = `True`).

When working with particularly large datasets (or limited RAM), users may not wish to hold all M datasets in memory at the same time. **MIDASpy** allows users to access one dataset at a time with the function `.yield_samples()`, which takes the same arguments as `.generate_samples()`. Instead of a list of datasets, however, it returns a Python generator from which each dataset can be called sequentially. We illustrate this feature in Section 5.1.2.

3.3. Analyze the completed datasets

As noted earlier, multiple imputation typically serves as a data-processing step for a subsequent statistical analysis using full-data methods. To obtain valid results, this analysis must be run on the M completed datasets separately, with the M sets of estimates then aggregated using Rubin's (1987) combination rules. If β denotes a parameter of interest (say, a regression coefficient), these rules state that the overall point estimate $\hat{\beta}$ is equal to the average estimate across the completed datasets:

$$\hat{\beta}_M = \frac{1}{M} \sum_{m=1}^{M} \hat{\beta}_m \tag{9}$$

The variance of $\hat{\beta}$ is a weighted sum of the estimated variance within (U) and between (B) the M completed datasets:

$$var(\hat{\beta}_M) = U_M + (1 + \frac{1}{M})B_M \tag{10}$$

where $U_M = \frac{1}{M} \sum_{m=1}^{M} var(\hat{\beta}_m)$ and $B_M = \frac{1}{M-1} \sum_{m=1}^{M} (\hat{\beta}_m - \hat{\beta}_M)^2$.

Due to the relative paucity of multiple imputation software in Python, **MIDASpy** includes a function for estimating generalized linear regression models and combining the results according to the Rubin rules: `combine()`. This function is independent from the `Midas` class, allowing users to apply it to completed datasets generated by any software.[13] `combine()` wraps the **statsmodels** package's `sm.GLM()` function (Seabold and Perktold 2010), accepting its keyword arguments. As a result, `combine()` enables users to perform multiple imputation regression analysis using a variety of generalized linear model (GLM) families, including Gaussian (i.e., ordinary least squares), Binomial, and Poisson.

- `df_list`: *List of pd.DataFrames.* A list of the M completed datasets to be analyzed.
- `y_var`: *String.* The outcome variable.
- `X_vars`: *List of strings.* A list of predictor variables.
- `dof_adjust`: *Boolean.* Indicates whether to apply the Barnard and Rubin (1999) degrees-of-freedom adjustment for small samples.
- `incl_constant`: *Boolean.* Indicates whether to include an intercept in the null model, the default in GLM software packages.

[13] Conversely, completed datasets produced by **MIDASpy** can be written to a general-purpose file format (such as CSV with the `to_csv()` function in **pandas**) and then analyzed in another programming language.

- `**glm_args`: Further arguments to be passed to `statsmodels.GLM()`, e.g., to specify model family, offsets, and variance and frequency weights (see the **statsmodels** documentation for details). If `None`, a Gaussian model will be estimated.

3.4. Note on replicability

Due to randomness introduced by the **TensorFlow** parallelization process, training parameters will vary minutely across runs of the same neural network in **MIDASpy**.[14] As a result, perfect replication of completed datasets produced by the software is not possible, even with seed settings. Nevertheless, variation between runs tends to be so small that it is unlikely to make a substantive difference to users. In seeded tests based on the data we analyze in Section 5, for instance, we were able to replicate training loss values to several decimal places. Hence, seed specification can still ensure essentially identical output from a given network. Note also that the trained imputation model and completed datasets can be saved to disk, allowing exact reproduction of analysis results.

4. R interface: The rMIDAS package

rMIDAS interoperates with **MIDASpy** via the **reticulate** package (Allaire, Ushey, Tang, and Eddelbuettel 2017) while maintaining a simple and intuitive workflow for users.

4.1. Installation and setup

As **rMIDAS** is available on the Comprehensive R Archive Network (`https://cran.r-project.org/web/packages/rMIDAS`), it can be installed and loaded by calling its name in R's `install.packages()` and `library()` functions.

```
R> install.packages("rMIDAS")
R> library("rMIDAS")
```

rMIDAS requires the following R packages:

- **data.table** (Dowle and Srinivasan 2020).
- **mltools** (Gorman 2018).
- **reticulate** (Allaire *et al.* 2017).

Users must also ensure that they have Python 3 installed on their system. When **rMIDAS** is first loaded, it will check whether the Python dependencies listed in Section 3.1 are present; if not, it will prompt users to install them from the R console.

rMIDAS supports the usage of custom Python binary paths as well as `conda` and virtual environments, which can be set using the following arguments in the `set_python_env()` function.

[14] Tiny differences in processor speed cause the order in which loss values are passed to mathematical operators to be randomized outside the **MIDASpy** code base. This affects imputations because operators involving floating-point numbers (i.e., non-integer real numbers) are non-commutative (Goldberg 1991).

```
# Point to a Python binary
R> set_python_env(python = "path/to/python/binary")

# Or point to a virtual environment binary
R> set_python_env(python = "virtual_env", type = "virtualenv")

# Or point to a conda environment (also specifying an executable)
R> set_python_env(python = "conda_env", type = "condaenv", conda = "auto")
```

4.2. Main functions and arguments

rMIDAS has four main functions, which in turn (1) preprocess the input dataset, (2) build and train the imputation model, (3) generate the M completed datasets, and (4) analyze these datasets using multiple imputation. Since **rMIDAS** directly calls **MIDASpy**, the arguments passed to these functions are identical to those of equivalent functions described in Section 3.2. We thus do not present the full list of **rMIDAS** arguments here (which can be found in the CRAN documentation), focusing instead on those that are unique to the package.

Preprocessing data

Leveraging efficient tools for manipulating data in the **data.table** (Dowle and Srinivasan 2020) and **mltools** (Gorman 2018) packages, the function `convert()` consolidates and automates the three data preprocessing steps discussed in Section 3.2.1.[15] In addition, it stores the parameters needed to reverse these steps post-imputation.

- `data`: `data.frame`, `data.table`, *or a path to a regular, delimited file.* The input dataset.
- `bin_cols`: *Vector of names.* A vector of all binary variables.
- `cat_cols`: *Vector of names.* A vector of all categorical variables.
- `minmax_scale`: *Boolean.* Indicates whether to scale all continuous variables between 0 and 1 (to improve algorithmic convergence) (default = `TRUE`).

Building and training the MIDAS network

The `train()` function combines the instantiation, building, and training of the MIDAS model into a single call. It accepts all arguments that can be passed to **MIDASpy**'s `Midas()`, `.build_model()`, and `.train_model()` functions (see Section 3.2).[16]

Generating completed datasets

[15] Users who are familiar with the syntax of **data.table** can pass data.tables directly to **rMIDAS**.

[16] At present, **rMIDAS** (version 0.3.0) keeps the following arguments at their default values: `train_batch`, `output_layers`, `loss_scale`, `init_scale`, `individual_outputs`, `manual_outputs`, `output_structure`, `weight_decay`, `act`, `noise_type`, `kld_min`. We plan to allow manual adjustment of these parameters in future releases.

In addition to producing completed datasets, the `complete()` function allows users to automatically save these datasets to file. Usage of **datatable**'s `fwrite()` function enables users to quickly write large datasets to CSV files, a significant bottleneck in imputation workflows.

- `mid_obj`: *Object of class* `midas`. The output from `rMIDAS::train()`.
- `m`: *Integer.* The number of completed datasets to be produced, i.e., M (default = 5).
- `unscale`: *Boolean.* Indicates whether to restore continuous variables that were previously scaled between 0 and 1 (default = `TRUE`).
- `bin_label`: *Boolean.* Indicates whether to restore binary variable names (default = `TRUE`).
- `cat_coalesce`: *Boolean.* Indicates whether to decode one-hot encoded categorical variables (default = `TRUE`).
- `fast`: *Boolean.* For categorical variables, indicates whether to impute the category with highest predicted probability (`TRUE`) or to return a weighted sample of categories based on their predicted probabilities (`FALSE`; default).
- `file`: *String.* Path for saving completed datasets. If `NULL`, these datasets are stored in memory.
- `file_root`: *String.* If `file` is specified, the root in file names of saved completed datasets. If `NULL`, these datasets will be saved as "file/midas_impute_yymmdd_hhmmss_m.csv."

Multiple imputation analysis

The `combine()` function runs generalized linear regression models on completed datasets and pools the results using Rubin's rules (see Section 3.3).

- `formula`: *Formula or character string.* The formula of the regression model to be estimated.
- `df_list`: *List of names.* A list of completed datasets (or objects coercible to data.frames) to be analyzed in the regression model.
- `dof_adjust`: *Boolean.* Indicates whether to apply the Barnard and Rubin (1999) degrees-of-freedom adjustment for small samples (default = TRUE).
- `...` Further arguments passed to `glm()`, R's inbuilt function for GLM estimation.

4.3. Summary and comparison

To summarize, **rMIDAS** and **MIDASpy** implement the same underlying algorithm but vary in their front-end programming language and in the specific steps they require users to undertake. Figure 2 provides a comparison of these steps and their associated functions in **rMIDAS** and

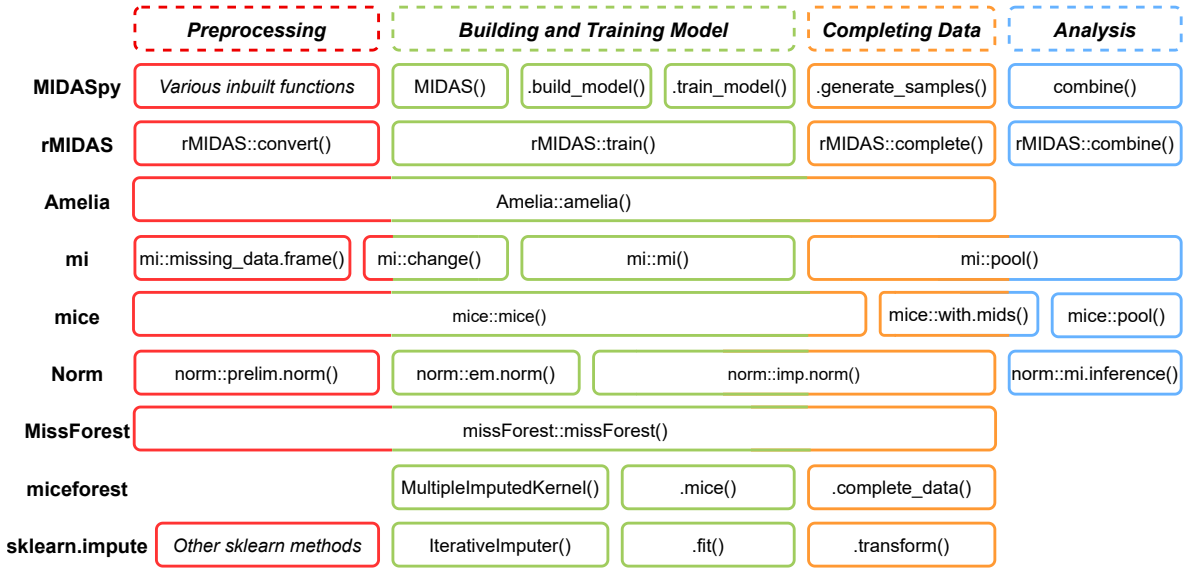


Figure 2: Comparison of workflows across multiple imputation software packages

MIDASpy as well as the multiple imputation packages discussed in Section 1 (and listed in Table 1).

Both packages include functions for building and training a MIDAS network, generating completed datasets, and analyzing these datasets using full-data regression methods. In **MIDASpy**, however, network initialization, construction, and training are divided into three separate functions, while in **rMIDAS** they are merged into one. Accordingly, **MIDASpy** permits slightly greater customizability of the MIDAS algorithm than **rMIDAS**. For instance, users of **MIDASpy** can specify the size of the mini-batches into which the input dataset is sliced during training to optimize memory allocation, an option not available in **rMIDAS**.

A second difference between the packages is that **rMIDAS** contains one function for converting input data to the specific format accepted by the MIDAS algorithm, whereas **MIDASpy** includes separate functions for binary and categorical variables (and no inbuilt function for the optional step of scaling continuous variables). This difference stems from the availability both of fast data-shaping functions in R (see Section 4.2.1) and of a wider range of approaches to handling and formatting data in Python (e.g., **pandas**, **NumPy**, **dask**, or a Structured Query Language strategy). Nonetheless, since none of the required preprocessing steps are computationally complex, we expect this stage to be relatively fast in both packages.

Most of the R packages reviewed in Section 1 also cover all stages of the standard imputation workflow. They do not, however, include separate functions for each stage; thus, they offer less fine-grained control over the workflow than **rMIDAS**. Each package combines at least two stages into a single call, and none contains a standalone function for generating completed datasets. This prevents users from specifying the number of draws from the trained imputation model and hence from extracting additional completed datasets after training. A further benefit of the separation between completion and analysis in **rMIDAS** is that users can pass output from `rMIDAS::complete()` to analysis functions in other packages, such as **mi**'s `mi::pool()`.

In contrast, none of the Python packages offer complete coverage of the workflow. Tools

for analyzing completed datasets are absent from all three packages, and **miceforest** also lacks any preprocessing functionalities.[17] As **MIDASpy**'s `combine()` function accepts a list of dataframes (see Section 3.3), it can be applied to the output from any of these packages. It thus fills a notable gap in existing multiple imputation software in Python.

5. Illustration with large-scale electoral data

In this section, we demonstrate the software's core functionalities by using it to impute missing responses to the 2018 Cooperative Congressional Election Study (CCES), an electoral survey conducted in the United States whose size and complexity poses computational difficulties for many existing multiple imputation algorithms.[18] We describe the **MIDASpy** and **rMIDAS** workflows in turn.

5.1. MIDASpy demonstration

The full CCES has 525 columns and 60,000 rows, the latter corresponding to individual survey respondents. After removing variables that either require extensive preprocessing or are unhelpful for imputation purposes — open-ended string variables, time indices, and ZIP code variables — the dataset contains 349 columns. The vast majority of these variables are categorical and must therefore be one-hot encoded for most multiple imputation software packages — that is, each $1 \times 60,000$ categorical variable with K unique classes must be expanded into a $K \times 60,000$ matrix of 1s and 0s — increasing their number to 1,914.

Loading and preprocessing the data

We begin by loading **MIDASpy**, its dependencies, and additional packages called in the workflow. We then read in the formatted CCES data and sort variables into continuous, binary, and categorical types.

```
>>> import numpy as np
>>> import pandas as pd
>>> import tensorflow as tf
>>> from sklearn.preprocessing import MinMaxScaler
>>> import sys
>>> import MIDASpy as md

>>> data_in = pd.read_csv("data/cces_jss_format.csv")
```

[17] We have identified only one multiple imputation package in Python that contains an analysis function: **autoimpute** (Kearney, Barkat, and Bose 2021). However, this function only supports a narrow range of regression models and requires users to specify this model within the imputation object itself, which is likely to be incompatible with some workflows. **autoimpute** is still at an early stage of development, and we were unable to successfully apply to it to several of our own datasets.

[18] Lall and Robinson (2022) show that **MIDASpy** imputes missing values in increasingly wide and long samples of the CCES more efficiently than **mi**, **Amelia**, and **norm**, exhibiting a speed advantage that increases linearly with length and exponentially with width. These samples were specially constructed and formatted to enable the latter packages to complete the exercise without crashing. We were not able to successfully apply any of them to the larger version of the dataset that we analyze in this section.

```
>>> cont_vars = ["citylength_1","numchildren","birthyr"]
>>> vals = data_in.nunique()

>>> cat_vars = list(
...  data_in.columns[(vals.values > 2) & ~(data_in.columns.isin(cont_vars))])

>>> bin_vars = list(data_in.columns[vals.values == 2])
```

Next, we apply the `.binary_conv()` function to the list of binary variables (which are not in dummy form), before appending them and the continuous variables to a `constructor_list` object, the basis for our final preprocessed dataset.

```
>>> data_bin = data_in[bin_vars].apply(md.binary_conv)

>>> constructor_list = [data_in[cont_vars], data_bin]
```

To one-hot encode categorical variables, we apply the `.cat_conv()` function to a dataframe containing them. We concatenate the resulting matrix to the existing `constructor_list` object.

```
>>> data_cat = data_in[cat_vars]

>>> data_oh, cat_col_list = md.cat_conv(data_cat)

>>> constructor_list.append(data_oh)

>>> data_0 = pd.concat(constructor_list, axis=1)
```

The final preprocessing step, which is nonessential, is to scale all variables between 0 and 1 to aid model convergence. We use **scikit-learn**'s `MinMaxScaler()` function for this step.

```
>>> scaler = MinMaxScaler()

>>> data_scaled = scaler.fit_transform(data_0)
>>> data_scaled = pd.DataFrame(data_scaled, columns = data_0.columns)

>>> na_loc = data_scaled.isnull()
>>> data_scaled[na_loc] = np.nan
```

Imputation

Once the data are preprocessed, training a MIDAS network with **MIDASpy** is straightforward. We declare an instance of the `Midas` class, pass our data to this object (including the sorted variable names) with the `.build_model()` function, and train the network for 10 epochs with the `.train_model()` function. For the purposes of this illustration, we maintain most of **MIDASpy**'s default hyperparameter settings.

```
>>> imputer = md.Midas(layer_structure= [256,256],
...  vae_layer = False,
...  seed= 89,
...  input_drop = 0.75)

>>> imputer.build_model(data_scaled,
...  binary_columns = bin_vars,
...  softmax_columns =  cat_col_list)

>>> imputer.train_model(training_epochs = 10)
```

Once the model is trained, we draw a list of 10 completed datasets. When datasets are very large, as in this case, we recommend accessing each one separately rather than simultaneously holding all of them in memory. We thus construct a dataset generator using the `.yield_samples()` function.

```
>>> imputations = imputer.yield_samples(m=10).output_list
```

From instantiation to completion, the imputation process took 14.7 minutes on a medium-performance computer.[19]

Analysis of completed datasets

We analyze the 10 completed datasets using **MIDASpy**'s inbuilt `combine()` function. We estimate a simple linear probability model in which `"CC18_415a"`, a respondent's degree of support for giving the United States Environmental Protection Agency power to regulate carbon dioxide emissions, is regressed on `"age"` (2018− `"birthyr"`), a respondent's age.[20] As we scaled the input dataset prior to imputation with the `MinMaxScaler()` function, for each completed dataset we first invert this transformation via **scikit-learn**'s `.inverse_transform()` function and also convert predicted probabilities for `CC18_415a` into binary categories using a threshold of 0.5. To save memory, we append the relevant subset of the data, for analysis, to a list.

```
>>> analysis_dfs = []

>>> for df in imputations:
...  df_unscaled = scaler.inverse_transform(df)
...  df_unscaled = pd.DataFrame(df_unscaled, columns = data_scaled.columns)
...  df['age'] = 2018 - df_unscaled['birthyr']
...  df['CC18_415a'] = np.where(df_unscaled['CC18_415a'] >= 0.5,1,0)
...  analysis_dfs.append(df.loc[:,["age","CC18_415a"]])
```

[19] An 8-core AMD 2700X processor with 16GB RAM running Ubuntu 22.04). In our replication materials, we show that it takes 29.8 minutes on a dual-core 2017 MacBook Pro computer with 8GB RAM.

[20] As noted in Section 3.4, users can ensure exact reproducibility of analytical results by saving completed datasets to disk. The trained MIDAS model itself is also saved by default (in **MIDASpy** to the location specified in the `savepath` argument of `Midas()`, in **rMIDAS** to the R session directory).

```
>>> model = md.combine(y_var = "CC18_415a",
...  X_vars = ["age"],
...  df_list = analysis_dfs)


>>> model

    term  estimate  std.error   statistic           df  p.value
0  const  0.934493   0.005515  169.459700  3056.421238      0.0
1    age -0.005259   0.000107  -49.160665  4565.125518      0.0
```

As noted in Section 3.4, users can ensure exact reproducibility of analytical results by saving completed datasets to disk.[21]

5.2. rMIDAS demonstration

Loading and preprocessing the data

Similarly to before, we start by loading **rMIDAS** and reading in the CCES sample.

```
R> library("rMIDAS")
R> set.seed(89)

R> data_0 <- fread("data/cces_jss_format.csv")
```

We then preprocess the data into the format required by the MIDAS algorithm using the `rMIDAS::convert()` function, which only requires vectors of binary and categorical variables. We set `minmax_scale = TRUE` to scale continuous variables between 0 and 1.

```
R> vals <- apply(data_0,2, function (x) length(unique(x)[!is.na(unique(x))]))
R> cont_vars <- c("citylength_1","numchildren","birthyr")
R> cat_vars <- names(vals)[vals > 2 & !(names(vals) %in% cont_vars)]
R> bin_vars <- names(vals)[vals == 2]

R> data_conv <- convert(data_0,
+    bin_cols = bin_vars,
+    cat_cols = cat_vars,
+    minmax_scale = TRUE)
```

Imputation

To train the MIDAS network, we pass our preprocessed data to the `rMIDAS::train()` function and specify network hyperparameters. Unlike in **MIDASpy**, we do not need to additionally declare categorical variables and their classes with the `softmax_columns` argument.

[21]The trained MIDAS model itself is also saved by default (in **MIDASpy** to the location specified in the `savepath` argument of `Midas()`, in **rMIDAS** to the R session directory).

```
R> data_train <- train(data_conv,
+    layer_structure= c(256,256),
+    vae_layer= FALSE,
+    seed= 89,
+    input_drop = 0.75,
+    training_epochs = 10)
```

We then generate 10 completed datasets using the `rMIDAS::complete()` function, saving them in memory. The function returns scaled continuous variables and one-hot encoded categorical variables to their original form using the parameters saved in the preprocessing step. Unlike in the **MIDASpy** demonstration above, by default this function converts imputed probabilities for binary variables into binary categories by taking draws from a binomial distribution with $P(x = 1) = p$. Categorical labels are similarly assigned by taking a weighted random draw using the vector of predicted probabilities from the imputed data.[22]. Note this difference slightly impacts the regression estimates compared to the Python results in the previous section.

```
R> imputations <- complete(data_train, m=10)
```

Overall, the imputation process took 32.6 minutes on the same medium-performance computer. This is longer than **MIDASpy**'s runtime because it also includes rescaling and collapsing one-hot encoded categorical variables after imputation (the time taken to interface with Python is negligible).[23]

Analysis of completed datasets

We estimate the same linear probability model as in the previous section by means of the `rMIDAS::combine()` function.

```
R> for (d in 1:10) {
+    imputations[[d]]$age <- 2018 - imputations[[d]]$birthyr
+    imputations[[d]]$CC18_415a <- ifelse(imputations[[d]]$CC18_415a == 1, 1, 0)
+    }

R> combine("CC18_415a ~ age", imputations)

## No model family specified -- assuming gaussian model.

##          term     estimate    std.error statistic       df      p.value
## 1 (Intercept)  0.860882795 0.0059241063 145.31859 457.8736  0.00000e+00
## 2         age -0.004134505 0.0001132612 -36.50416 809.2863 3.26543e-173
```

rMIDAS *output*

[22]This behavior can be disabled by setting `fast = TRUE` when calling `rMIDAS::complete()`, in which case binary variables are assigned 1 if $p \geq 0.5$, and the label with the highest predicted probability is chosen for categorical variables

[23]Users could accelerate runtime by not decoding categorical variables (`cat_coalesce = FALSE`) and instead including them as dummy variables in subsequent analyses.

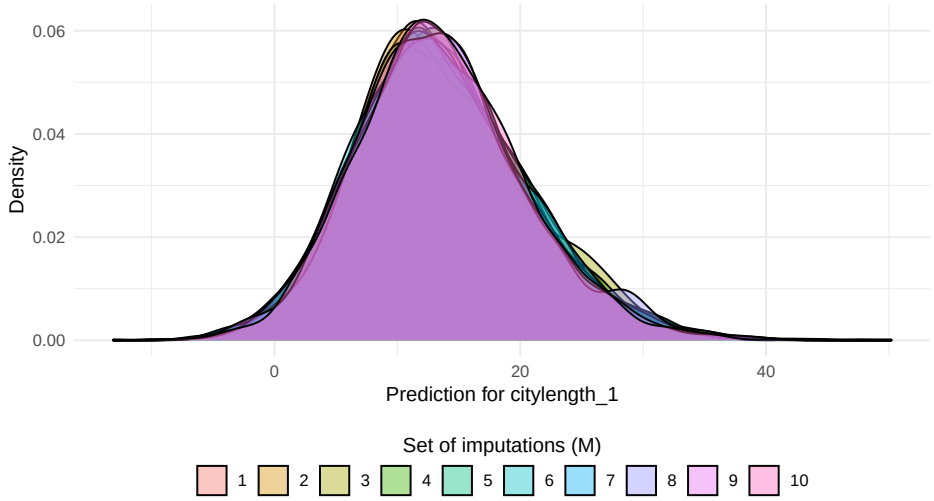


Figure 3: Imputed values of CCES variable `"citylength_1"` (generated with **rMIDAS**)

Figures 3 and 4 illustrate the results of the **rMIDAS** imputation process. The former plots the kernel densities of the 10 sets of imputed values of `"citylength_1"`, a continuous variable measuring how long a respondent has lived in their current city of residence. The distributions have an approximately normal shape centered on the same mean, with the small differences reflecting random variation in the **rMIDAS** algorithm's draws from the missing-data posterior (which, in turn, reflects uncertainty about the correct imputation model).

Figure 4 displays the densities of original and imputed values of six CCES variables, of which two are categorical (`"CC18_401"`, `"CC18_4123"`), two are binary (`"CC18_415a"`, `"CC18_417_a"`), and two are numeric (`"citylength_1"`, `"numchildren"`). Imputations are averaged across completed datasets, with categorical variables rounded to the nearest integer. Only minor differences between the two distributions are apparent.[24]

6. Model calibration and validation

MIDASpy and **rMIDAS** offer a variety of functionalities for calibrating and assessing the fit of the imputation model. This section describes the three main tools: (1) the technique of "overimputation"; (2) the adjustment of MIDAS network hyperparameters; and (3) the inclusion of a variational autoencoder layer in the network.

6.1. Overimputation

Overimputation, which was developed by Honaker *et al.* (2011) and Blackwell, Honaker, and King (2017), involves removing random observed values from the dataset, generating multiple imputations for each value, and checking the accuracy of these imputations.[25] This method is useful both for the evaluating the overall performance of the imputation model and, in

[24]Perhaps the clearest is that the original distribution of `CC18_401`, which measures a respondent's reported voting behavior in the 2018 Midterm Election, has slightly fewer values of 2 (thought about voting), 3 (usually votes but not this time) and 4 (attempted to vote) than the imputed distribution.

[25]It thus has a similar logic to MIDAS itself. The method of imputing the removed values (and the proportion

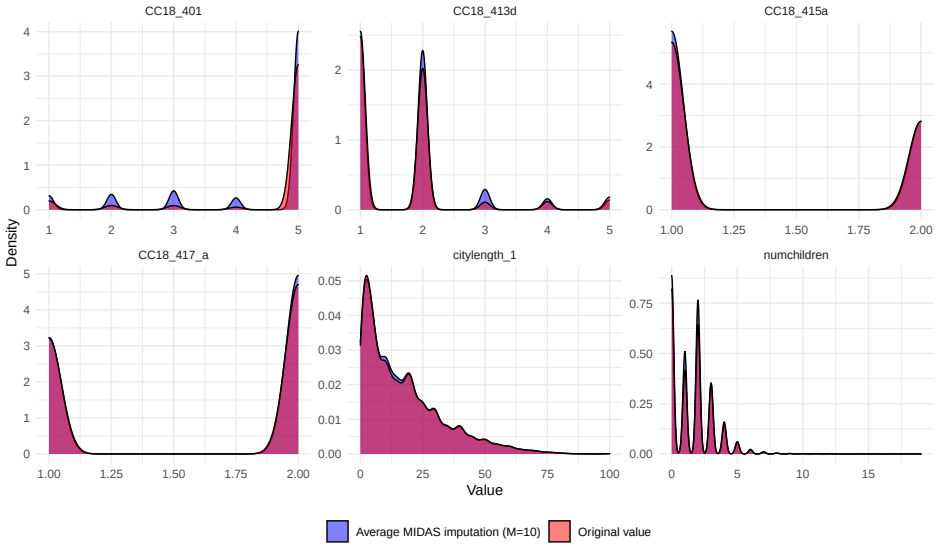


Figure 4: Density plots of original and imputed values of six CCES variables (based on **rMIDAS** output)

particular, for determining the optimal length of the training process, which is difficult to know *a priori*. Initially, additional training epochs should improve the model's predictive performance as it learns more accurate approximations of the missing-data posterior. However, longer training yields diminishing returns and increases the likelihood of overfitting. Overimputation allows users to track model performance over the course of training, helping them to establish an appropriate point at which to halt this process in the final imputation run.

The `overimpute()` function in **MIDASpy** and **rMIDAS** spikes additional missingness into the input data and reports imputation accuracy at training intervals specified by the user. Accuracy is measured as the RMSE of imputed values versus actual values for continuous variables and classification error for binary and categorical variables (i.e., the fraction of correctly predicted classes subtracted from 1). Metrics are reported in two forms: (1) their summed value over all Monte Carlo samples from the estimated missing-data posterior ("Aggregated RMSE", "Aggregated binary error", and "Aggregated softmax error"); and (2) their summed value divided by the number of such samples ("Individual RMSE", "Individual binary error", and "Individual softmax error").

In the final model, we recommend selecting the number of training epochs that minimizes the average value of these metrics — weighted by the proportion (or relative substantive importance) of continuous and categorical variables — in the overimputation exercise. This "early stopping" rule reduces the risk of overtraining and thus effectively serves as an extra layer of regularization in the network.

We favor overimputation over customary train/test split approaches to model calibration and validation for two reasons. First, the latter have been found to systematically underestimate error in autoencoders and other unsupervised methods of nonlinear dimensionality reduction

of such values) are, of course, very different.

(Christiansen 2005; Scholz 2012). Second, they prevent us from training the imputation model on the full dataset, which compromises performance (particularly when there are high levels of missingness).

Arguments

The arguments of `overimpute()` are similar to those of **rMIDAS**'s `train()` function but allow users to specify the frequency with which loss values are calculated.

- `spikein`: *Float (between 0 and 1).* The probability of corruption for observed values in the input dataset (default = 0.1).
- `training_epochs`: *Integer.* The number of overimputation training epochs (default = 100). Selecting a low value increases the risk that trends in the loss metrics have not stabilized by the end of training, in which case the exercise is less informative.
- `plot_vars`: *Boolean.* Specifies whether to plot the distribution of original versus overimputed values (default = `True`). A density plot is generated for continuous variables and a barplot for categorical variables (showing proportions of each class).
- `plot_main`: *Boolean.* Specifies whether to display the main graphical output, a plot of error trends during overimputation training, at every reporting interval (set by `report_ival`) (default = `True`). If set to `False`, this output will only appear at the end of the training process. Error values are still shown at each interval.
- `skip_plot`: *Boolean.* Specifies whether to suppress the main graphical output (default = `False`). This may be desirable when users are conducting a series of overimputation exercises and are primarily interested in the console output.
- `save_figs`: *Boolean.* Specifies whether to save generated figures instead of displaying them in the console (default = `False` in **MIDASpy**; the argument is always `TRUE` in **rMIDAS** to prevent compatibility issues).
- `save_path`: *String.* Indicates path to save overimputation figures (default = the working directory). Users should include a trailing "/" at the end of the path (i.e., `save_path = "path/to/figures/"`).
- `spike_seed`: *Integer.* The value to which the Python or R pseudo-random number generator is initialized for the missingness spike-in. This is separate to the `seed` specified in the `Midas()` or `rMIDAS::train()` call.
- `excessive`: *Boolean.* Specifies whether to print aggregate mini-batch loss to the terminal (default = `False`).[26] This allows users to check for unusual imputations, which may be helpful if loss is not declining during overimputation training.

Demonstration

We demonstrate the `overimpute()` function on the CCES data analyzed in the previous section, specifying the imputation model in the same way. We set the function to run for 100

[26]The `excessive` argument in **MIDASpy**'s `.train_model()` function, in contrast, prints individual mini-batch loss.

training epochs, removing 30% of observed values and reporting error every 25 epochs. In **MIDASpy**, we set `plot_main = False` to generate a single graphical plot at the end of the run rather than separate plots after each reporting interval.

```
>>> imputer.overimpute(spikein = 0.3,
...  training_epochs = 100,
...  report_ival = 25,
...  plot_vars = False,
...  plot_main = False,
...  spike_seed = 89)
```

In **rMIDAS**, we save generated figures directly to disk via the `save_path` argument, which prevents the R console from waiting for each plot to be manually closed before proceeding.

```
R> overimpute(data_conv,
+    layer_structure= c(256,256),
+    vae_layer= FALSE,
+    seed= 89,
+    spikein = 0.3,
+    training_epochs = 100,
+    report_ival = 25,
+    plot_vars = TRUE,
+    spike_seed = 89,
+    save_path = "figures/")
```

Figure 5 displays the `rMIDAS::overimpute()` function's main graphical output, with red crosses indicating the lowest value of each error metric. The two RMSE metrics decline over the first 25 training epochs, after which they increase slightly, which is typically a sign of overfitting. In such instances, users should cap the number of epochs in the real imputation process before the start of this increase. The binary error metrics are also relatively stable over training, though record their minimum values later in the process (at 75 epochs for the individual metric and 100 epochs for the aggregate metric). In contrast, the classification error metrics fall sharply during the first 25 epochs and more slowly thereafter, reaching their lowest points at the end of the training.

The length of the training process can therefore have uneven effects on MIDAS's imputation accuracy for different types of variables, and users will often have make tradeoffs according to the type of error they wish to minimize. In this particular case, all error metrics are close to their minimum values after 25 epochs, so users may decide that additional training is not worth the extra computation time (and risk of overfitting).

In many applications, it is also useful to inspect overimputed values of individual variables and to compare their distribution to that of the original data. When `plot_vars = True`, `overimpute()` plots these two distributions for each variable at every reporting interval. Figures 6 and 7 provide an example of this output in **rMIDAS** for a continuous variable (`"numchildren"`) and a categorical variable with many levels (`"industryclass"`), respectively. We recommend setting `plot_vars = FALSE` for very wide datasets to speed up training and prevent a crowded console.

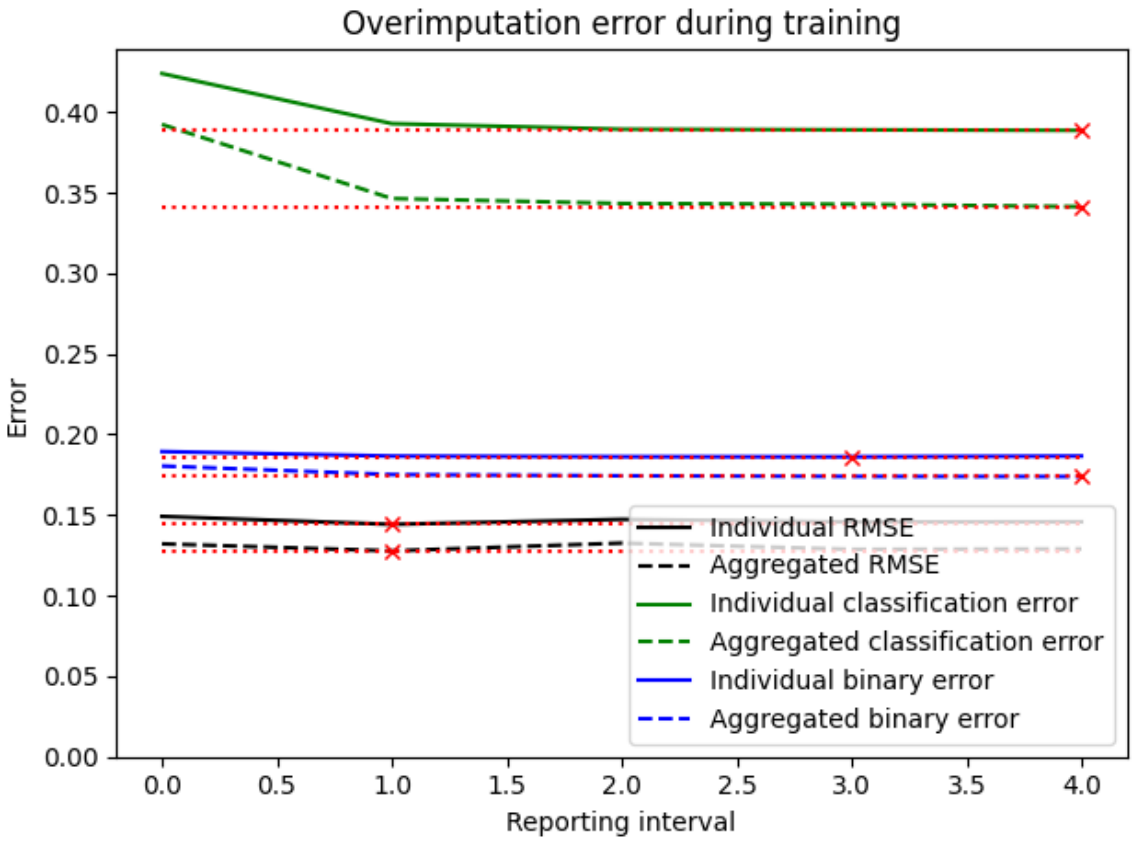


Figure 5: Graphical output of the `rMIDAS::overimpute()` function applied to the CCES. The reporting interval (x-axis) is 25 training epochs. The red crosses indicate the lowest value of each error metric.

6.2. Assessing sensitivity to additional hyperparameters

The performance of neural networks can also be sensitive to hyperparameters other than the number of training epochs, such as the structure of the network, the learning rate, and activation functions. To assess such sensitivity and optimize model specification, users of **MIDASpy** and **rMIDAS** should ideally compare results under alternative values of a selection of hyperparameters. The default values discussed in Section 3.2 will often provide an appropriate baseline for comparison, though may be far from optimal in some applications. In this section, we compare the software's performance across varying values of three of the most consequential hyperparameters in deep neural networks: network depth, network width, and the learning rate.

Network depth and breadth

The optimal depth and breadth of a MIDAS network naturally depends on the structure of the input data. In general, when data have complex characteristics (such as a large number of dimensions and extreme nonlinearities) the network's performance will be more sensitive to depth and width, which are key determinants of its capacity to learn the underlying data

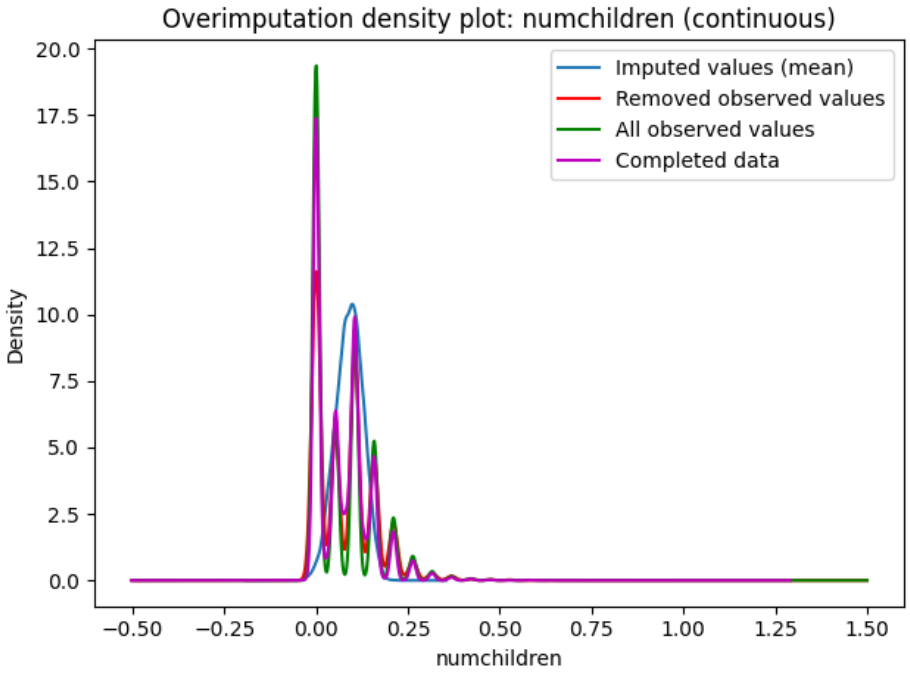


Figure 6: Example of the `rMIDAS::overimpute()` function's `plot_vars` output for a continuous variable (`"numchildren"`)

manifold. Other things equal, wider networks are better able to learn complex interactions between variables, while deeper networks are better able to capture complex nonlinearities. Importantly, however, networks that are too deep or wide relative to the complexity of the data are more susceptible to overfitting — in addition to wasting valuable computation time.

To illustrate this point, we vary the number of layers and nodes per layer in the MIDAS network applied to the CCES in Section 5. We instantiate 12 separate versions of this network with 2, 3, or 4 hidden layers and 64, 128, 256, or 512 nodes per layer. To gauge performance, we use the `.overimpute()` function to spike 30% additional missingness into the dataset and calculate loss after 100 training epochs.[27] In **MIDASpy**, we save the console output to a text file, from which we scrape the relevant loss values.

```
>>> nodes = [64, 128,256,512]
>>> layers = [2,3,4]
>>> node_layers = []

>>> for n in nodes:
...  for l in layers:
...    node_layers.append([n]*l)

# Record output
>>> text_path = "data/hyperparameter_python_output.txt"
```

[27] We set `skip_plot = True` to suppress the function's main graphical output.

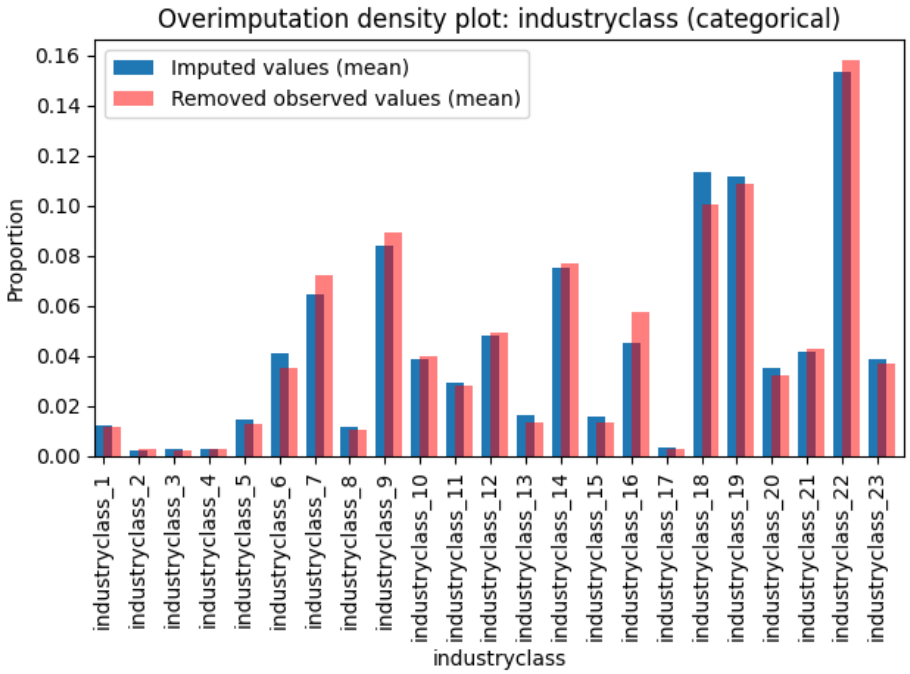


Figure 7: Example of the `rMIDAS::overimpute()` function's `plot_vars` output for a categorical variable (`"industryclass"`)

```
>>> txt_file = open(text_path, 'w')
>>> sys.stdout = txt_file

>>> for nl in node_layers:
...  print("Layer structure: ",nl)
...  imputer = Midas(layer_structure=nl, vae_layer= False,
...    seed= 89, input_drop = 0.75)
...  imputer.build_model(data_scaled,
...    binary_columns = bin_vars,
...    softmax_columns =  cat_col_list)
...  imputer.overimpute(spikein = 0.3, training_epochs = 101, report_ival = 50,
...    spike_seed = 89, plot_vars = False, plot_main = False,
...    skip_plot = True)
...  print("\n\n")

# Scrape loss values
>>> txt_input = open(text_path, 'r')

>>> structures = []
>>> softmax_agg = []

>>> for line in txt_input:
...  if "Layer structure" in line:
```

```
...     structures.append(line[18:len(line)-1])
...   elif "Aggregated error on softmax spike-in" in line:
...     softmax_agg.append(line[38:len(line)-1])

>>> softmax_agg = list(map(float, softmax_agg))

>>> epochs = [0,50,100]*len(structures)

>>> structure_list = >>> np.repeat(structures,3)
>>> layers = list(np.repeat([2,3,4], 3))*4
>>> nodes = list(np.repeat([64,128,256,512], 9))

>>> data = pd.DataFrame({"structure":structure_list, "layers":layers,
...   "nodes":nodes, "epoch": epochs,
...   "softmax_agg": softmax_agg})
```

In **rMIDAS**, we extract the loss values from a character vector saved from the R console output.

```
R> nodes <- c(64,128,256,512)
R> layers <- c(2,3,4)
R> node_layers <- list()

R> for (n in nodes) {
+    for (l in layers) {
+      node_layers[[length(node_layers) + 1]] <- rep(n, l)
+      }
+    }

R> hyper_res <- py_capture_output(
+    for (nl in node_layers) {
+      print("Node structure: ")
+      print(nl)
+      overimpute(data_conv,
+        layer_structure= nl,
+        vae_layer= FALSE,
+        seed= 89,
+        input_drop = 0.75,
+        spikein = 0.3,
+        training_epochs = 100,
+        report_ival = 100,
+        plot_vars = FALSE,
+        skip_plot = TRUE,
+        spike_seed = 89)
+      },
+    type = c("stdout"))
```

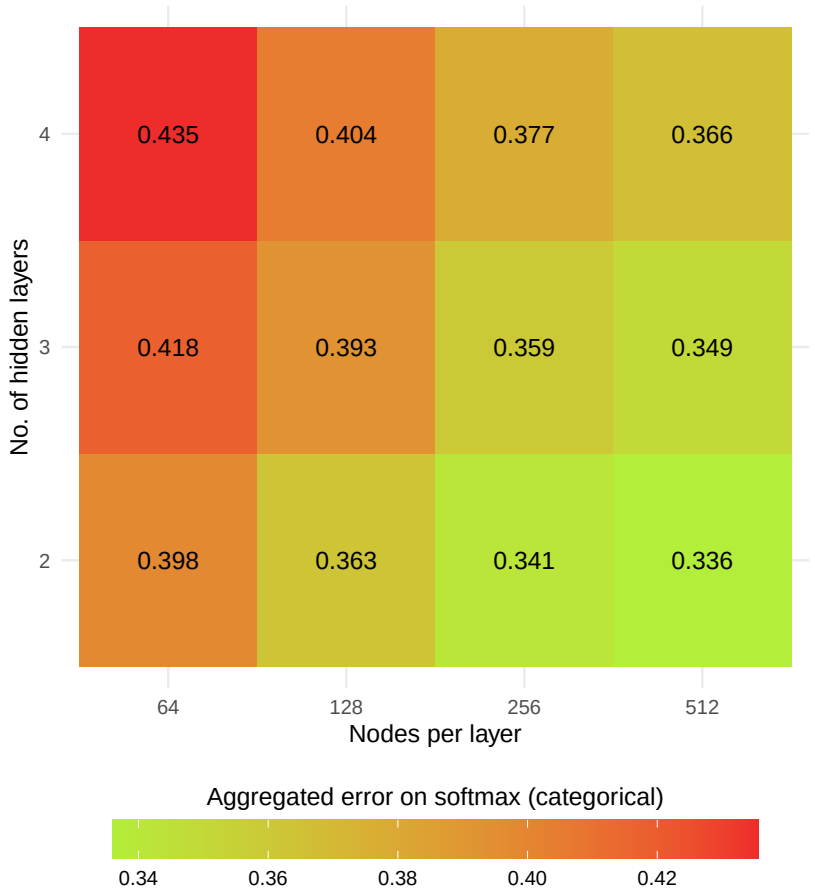


Figure 8: Aggregated softmax error of 12 MIDAS networks of varying depth and width after 10 epochs of training on the CCES (based on **rMIDAS** output).

```
# Scrape loss values
R> hyper_vec <- strsplit(hyper_res, "\n")[[1]]

R> softmax_agg = c()

R> for (line in hyper_vec) {
+    if (grepl("Aggregated error on softmax spike-in", line)) {
+      softmax_agg <- append(softmax_agg, substr(line,38,nchar(line)))
+      }
+    }

R> hyperp_data <- data.frame(
+    epochs = rep(c(0,100),length(node_layers)),
+    layers = rep(layers,each=2),
+    nodes = rep(nodes, each = 2*length(layers)),
+    softmax_agg = as.numeric(softmax_agg))
```

Figure 8 reports the aggregate softmax error for the 12 models based on the **rMIDAS** output (as noted in Section 5, the preponderance of CCES variables are categorical).[28] Error is minimized with a network comprising two hidden layers of 512 nodes and maximized with a network comprising four hidden layers of 64 nodes. These results indicate that the kind of complexity that characterizes the CCES is better captured by wider and shallower networks, perhaps because it is primarily manifested in interactions between variables rather than in nonlinearities.

Thus, users should not assume that increasing the width and depth of the network will always improve imputation accuracy; the nature of this impact rests on subtle characteristics of the input data. The sizable differences in error shown in Figure 8 show that assessing sensitivity to network structure is a simple but effective strategy for improving the fit of the imputation model as well as for making the most efficient use of computational resources.

Adjusting the learning rate

As noted in Section 3.2, the learning rate determines the scale of the adjustment made to the MIDAS network's weights and biases after each training epoch. A higher learning rate results in larger but less precise updates to these parameters, which reduces overall training time but tends to increase error by limiting the extent to which the network can "fine-tune" its parameters. Consequently, there is often a tradeoff between training time and imputation accuracy when specifying the learning rate.

To illustrate this tradeoff, we train a series of MIDAS networks with learning rates varying from a minimum of $\gamma = 0.00001$ to a maximum of $\gamma = 0.1$, holding constant other hyperparameters at their default settings.

In **MIDASpy**, we follow a similar strategy to the network structure example, recording the Python console output to a text file.

```
>>> learn_rates = [0.00001, 0.0001, 0.001, 0.01,0.1]

# Record output
>>> lr_path = "data/learn_rate_results.txt"
>>> lr_txt = open(lr_path, 'w')
>>> sys.stdout = lr_txt

>>> for lr in learn_rates:
...  imputer = md.Midas(layer_structure= [128,128],
...    vae_layer = False,
...    seed= 89,
...    input_drop = 0.75,
...    learn_rate = lr)
...  imputer.build_model(data_scaled,
...    binary_columns = bin_vars,
...    softmax_columns = cat_col_list)
...  imputer.train_model(training_epochs = 20)
```

[28] The aggregated RMSE results are essentially the same.

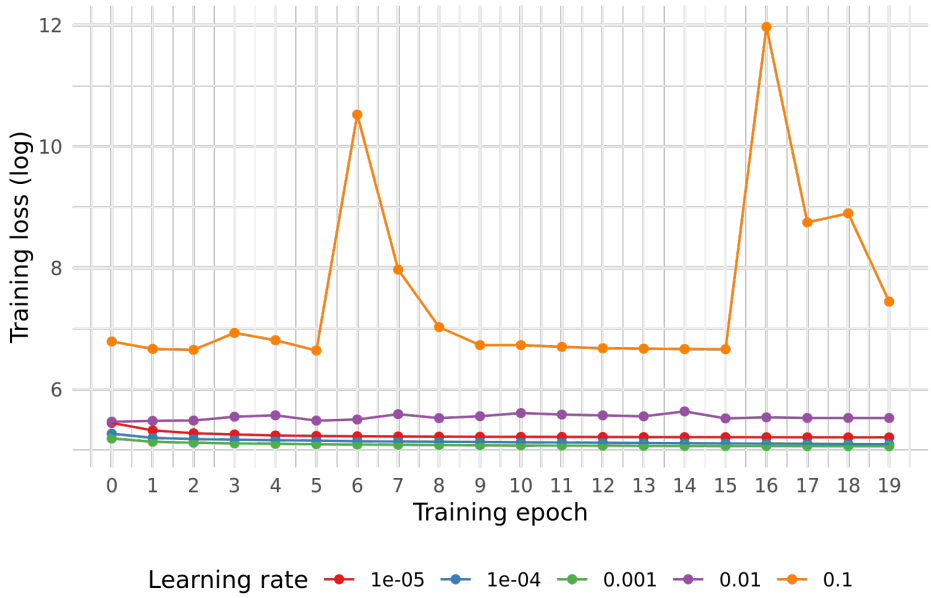


Figure 9: Training loss of MIDAS networks with different learning rates (based on **rMIDAS** output)

In **rMIDAS**, we record the R console's output at each training epoch and scrape the loss value.

```
R> learn_rates <- c(0.00001, 0.0001, 0.001, 0.01,0.1)

R> lr_out <- py_capture_output(
+    {
+    for (lr in learn_rates) {
+      train(data_conv,
+        layer_structure= c(128,128),
+        vae_layer= FALSE,
+        seed= 89,
+        input_drop = 0.75,
+        training_epochs = 20,
+        learn_rate = lr)
+      }
+    })
```

Figure 9 plots the **rMIDAS** training loss for each learning rate in each epoch. The optimal learning rate ($\gamma = 0.001$) is higher than the default setting ($\gamma = 0.0001$), though the two networks perform comparably. The lowest rate ($\gamma = 0.00001$) also yields larger (but declining) loss, as might be expected, given the smaller parameter adjustments it permits. The two highest rates ($\gamma = 0.01$ and $\gamma = 0.1$) deliver the worst performance and generate erratic loss, indicating that parameter adjustments are so large that the model is incapable of learning better approximations of the input data.

In sum, the learning rate can also influence MIDAS's accuracy and speed in important ways, and there is no guarantee that its default setting in **MIDASpy** and **rMIDAS** will deliver the

best performance. Testing different rates can therefore help users to optimize the software for their application, particularly where loss values exhibit an irregular or increasing trend during training (suggesting that a lower rate could improve accuracy) or convergence is very slow (suggesting that a higher rate could improve speed). As with other tunable hyperparameters, the optimal setting will depend on the size and structure of the input data.

6.3. Variational autoencoder component

A third diagnostic tool provided by the software is the generation of an alternative set of imputations using a variational autoencoder component, which changes the process by which input data are encoded into latent representations. These imputations can then be compared against the input data and the regular imputations as a form of model validation.

Variational autoencoders encode inputs not to a fixed vector $\mathbf{y}$ but to a chosen *distribution* over the latent space of the data. The typical choice is a multivariate normal distribution parameterized by a vector of means μ and corresponding variances $\mathbf{\Sigma}$. The loss function minimized during training includes a Kullback-Leibler divergence term (in addition to the usual reconstruction term) that serves as a regularizer, reducing the risk of an uneven latent space in which similar data points can become very different after decoding. Decoded samples from the latent distribution therefore tend to closely approximate the form of the input data, rendering the method well suited to the task of generative modeling.[29]

Figure 10 provides a visual summary of how a standard MIDAS network (panel a) is altered by the inclusion of a variational autoencoder component (panel b). In both networks, missing values are set to 0, nodes are stochastically dropped from hidden layers, and the encoder compresses the corrupted input $\tilde{\mathbf{x}}$ through a series of shrinking hidden layers. In the variational network, however, the bottleneck layer maps the output of the previous layer to a latent multivariate normal distribution: $\mathbf{y} \sim \mathcal{N}(\mu|\mathbf{\Sigma})$. A vector of values is then sampled from this distribution and decompressed through the decoder. The output $\mathbf{z}$ is identical in form to that of the standard network, and imputations can be drawn from the estimated posterior of the model.

Imputations produced by the variational MIDAS network are thus based on an alternative — more stringent but not necessarily less plausible — set of assumptions about the distribution of the latent space. If these values are similar to both the observed input data and the standard imputations, therefore, the MIDAS network is likely to have learned a robust latent representation of such data.

Associated arguments

As documented in Section 3.2, the variational autoencoder component can be activated by setting `vae_layer = True` in **MIDASpy**'s `Midas()` function and in **rMIDAS**'s `train()` function. This component has separate hyperparameters that are specified in three additional arguments.

- `latent_space_size`: *Integer.* The number of normal dimensions used to parameterize the latent space (default = 4).

[29] Variational autoencoders have been used in applications as diverse as generating fake photographic portraits and composing purely synthetic music.

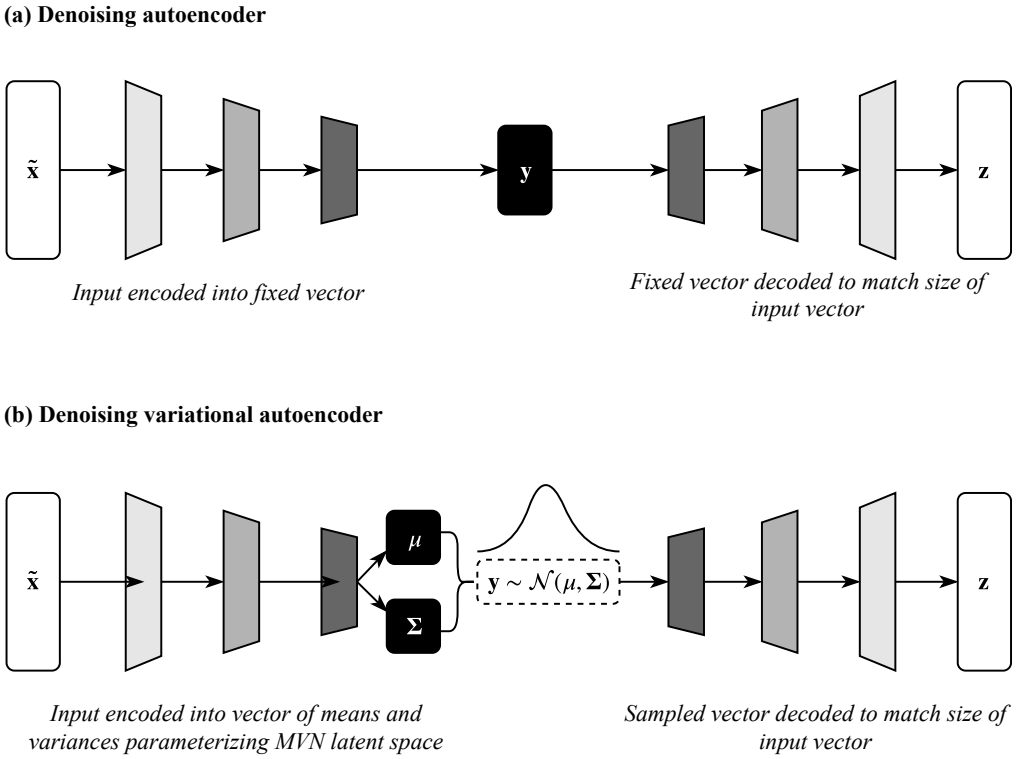


Figure 10: The architecture of a MIDAS network with and without a variational autoencoder layer

- `vae_sample_var`: *Float.* The sampling variance of the normal distributions used to parameterize the latent space (default = 1).
- `vae_alpha`: *Float.* The strength of the prior imposed on the Kullback-Leibler divergence term in the variational autoencoder loss functions (default = 1).

Demonstration

We demonstrate the variational autoencoder functionality by comparing the results from four MIDAS networks in which (1) this option is either enabled or disabled and (2) there two layers of either 128 or 512 nodes. We train each network on a reduced subset of the CCES for 10 epochs and draw 10 sets of imputed values, again rounding categorical variables to integers.[30] The **MIDASpy** code for the variational network with two 512-node layers is as follows.

```
>>> imputer = Midas(layer_structure= [512,512],
...  seed= 89,
```

[30]We limit the number of variables to prevent memory shortages caused by holding four separate imputation models — and therefore 40 completed datasets — in memory. The subset contains a mixture of binary, categorical, and continuous variables. See the replication material for further details.

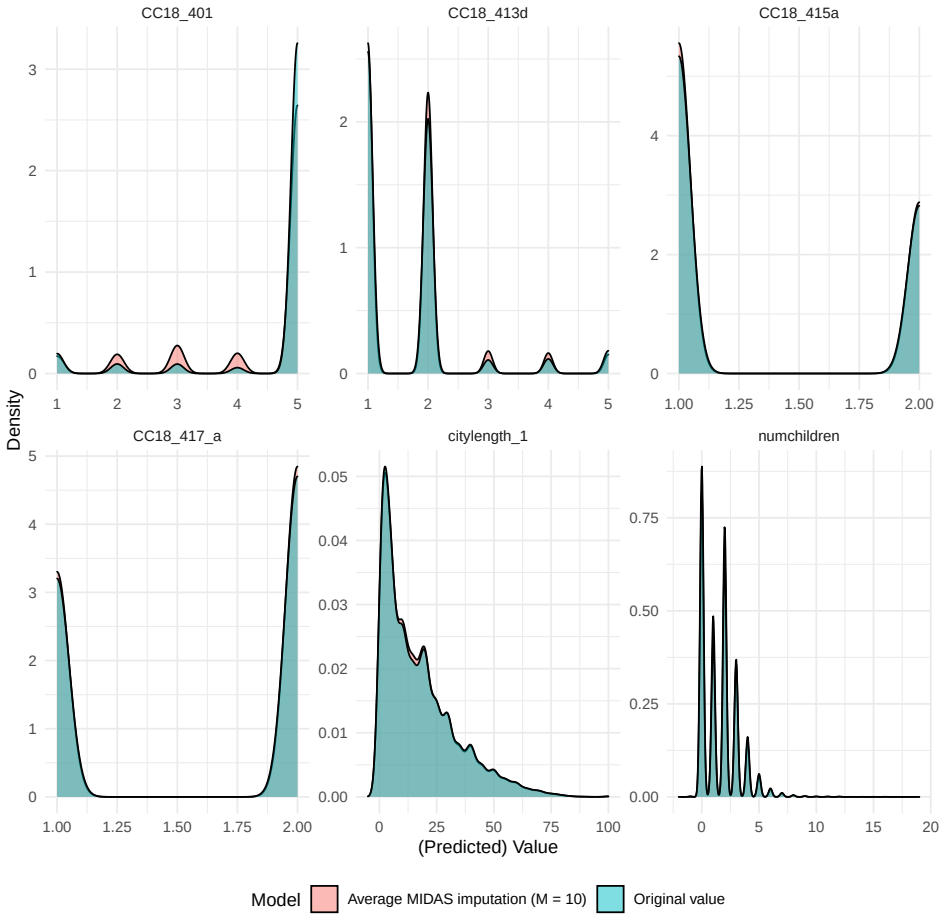


Figure 11: Density plots of original values and variational imputed values for six CCES variables (based on **rMIDAS** output)

```
...   input_drop = 0.75,
...   vae_layer= True)

>>> imputer.build_model(data_scaled, softmax_columns= columns_list)

>>> imputer.train_model(training_epochs = 10)
```

In **rMIDAS**, only the `rMIDAS::train()` function is required.

```
R> train(data_short,
+     layer_structure= c(512,512),
+     vae_layer= TRUE,
+     seed= 89,
+     input_drop = 0.75,
+     training_epochs = 10)
```

From **rMIDAS**'s output, Figure 11 plots the distribution of original values and average imputed values produced by the two-layer, 512-node variational network for the six CCES vari-

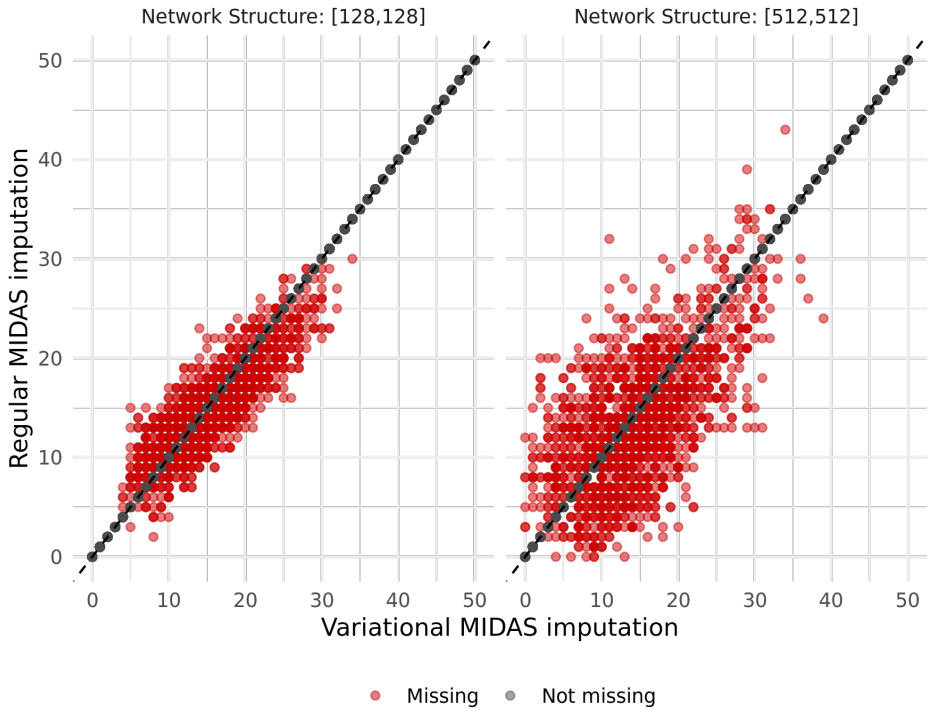


Figure 12: Imputed values of `"citylength_1"` generated by a MIDAS network without and without a variational autoencoder component (based on **rMIDAS** output)

ables inspected in Section 5. The close alignment between the two densities in each graph indicates that the variational autoencoder has learned a good approximation of the latent space. The four-layer, 128-node network yields comparable distributions (not reported). In line with the results of Section 6.2, however, it performs slightly worse on predictive accuracy in an overimputation exercise.

Figure 12 compares imputed values of `"citylength_1"` generated by both versions of the variational and standard networks. The variational imputations have a very similar distribution to the regular ones, with both exhibiting greater variance in the case of the 512-node network. As more varied samples generally indicate a better mapping of the latent space, this constitutes further evidence that a relatively wide but shallow network is most effective at capturing the type of complexity expressed by the CCES. Taken together, these comparisons suggest that the standard MIDAS imputation model is a good fit to the input data, giving us increased confidence in its output.

7. Concluding remarks

MIDASpy and **rMIDAS** enable users of Python and R, respectively, to efficiently multiply impute missing values in datasets of widely varying size and complexity. The packages implement a recently developed approach to multiple imputation — MIDAS — that harnesses the capacity of deep neutral networks to efficiently learn informative and robust latent representations of observed data. Leveraging the efficiency of the **TensorFlow** platform for machine

learning programming, they provide a set of user-friendly and flexible tools for constructing and calibrating MIDAS imputation models, validating model outputs, and extracting and analyzing completed datasets.

We intend to continually refine and incorporate additional features into the software to enhance its performance. In the near term, we plan to improve the scope of its input pipeline for massive datasets and to develop a consolidated and efficient preprocessing function for **MIDASpy** (similar to that in **rMIDAS**) to further streamline users' workflows. Looking further ahead, we are exploring the possibility of adding explicit functionalities for handling temporal dependence, such as recurrent neural network cells and locally weighted smoothing, implementing categorical embeddings, and of allowing users to incorporate observation- and cell-level prior information about specific missing values into the imputation model.

References

Abadi M, Agarwal A, Barham P, Brevdo E, Chen Z, Citro C, Corrado GS, Davis A, Dean J, Devin M, Ghemawat S, Goodfellow I, Harp A, Irving G, Isard M, Jia Y, Jozefowicz R, Kaiser L, Kudlur M, Levenberg J, Mane D, Monga R, Moore S, Murray D, Olah C, Schuster M, Shlens J, Steiner B, Sutskever I, Talwar K, Tucker P, Vanhoucke V, Vasudevan V, Viegas F, Vinyals O, Warden P, Wattenberg M, Wicke M, Yu Y, Zheng X (2016). "**TensorFlow**: Large-Scale Machine Learning on Heterogeneous Distributed Systems." arXiv:1603.04467, URL https://arxiv.org/abs/1603.04467.

Allaire J, Ushey K, Tang Y, Eddelbuettel D (2017). ***reticulate**: R Interface to Python*. URL https://github.com/rstudio/reticulate.

Barnard J, Rubin DB (1999). "Small-Sample Degrees of Freedom with Multiple Imputation." *Biometrika*, **86**(4), 948–955. ISSN 00063444. URL http://www.jstor.org/stable/2673599.

Blackwell M, Honaker J, King G (2017). "A Unified Approach to Measurement Error and Missing Data: Overview and Applications." *Sociological Methods & Research*, **46 (3)**, 303–341.

Christiansen B (2005). "The Shortcomings of Nonlinear Principal Component Analysis in Identifying Circulation Regimes." *Journal of Climate*, **18**(22), 4814–4823.

Dask Development Team (2016). ***Dask**: Library for Dynamic Task Scheduling*. URL https://dask.org.

Dowle M, Srinivasan A (2020). ***data.table**: Extension of **data.frame***. URL https://github.com/Rdatatable/data.table.

Gal Y, Ghahramani Z (2016). "Dropout as a Bayesian Approximation: Representing Model Uncertainty in Deep Learning." In *Proceedings of the 33rd International Conference on Machine Learning*, pp. 1050–1059. ACM.

Glorot X, Bengio Y (2010). "Understanding the Difficulty of Training Deep Feedforward Neural Networks." In *Proceedings of the 13th International Conference on Artificial Intelligence and Statistics*, pp. 249–256. ICAIS.

Goldberg D (1991). "What Every Computer Scientist Should Know About Floating-Point Arithmetic." *ACM Computing Surveys (CSUR)*, **23**(1), 5–48.

Gorman B (2018). ***mltools**: Machine Learning Tools*. URL https://github.com/ben519/mltools.

Hinton GE, Srivastava N, Krizhevsky A, Sutskever I, Salakhutdinov RR (2012). "Improving Neural Networks by Preventing Co-Adaptation of Feature Detectors." *Neural Networks*, **2**, 1–18.

Honaker J, King G (2010). "What to Do about Missing Values in Time-Series Cross-Section Data." *American Journal of Political Science*, **54**(2), 561–581.

Honaker J, King G, Blackwell M (2011). "**Amelia II**: A Program for Missing Data." *Journal of Statistical Software*, **45 (7)**, 1–47.

Hunter JD (2007). "**Matplotlib**: A 2D Graphics Environment." *Computing in Science & Engineering*, **9**(3), 90–95.

Kearney J, Barkat S, Bose A (2021). ***autoimpute**, Version 0.12.2*. URL https://github.com/kearnz/autoimpute.

Kingma DP, Ba J (2015). "Adam: A Method for Stochastic Optimization." In *3rd International Conference on Learning Representations (ICLR), Conference Track Proceedings*. ICLR.

Lall R (2016). "How Multiple Imputation Makes a Difference." *Political Analysis*, **24**(4), 414–433.

Lall R, Robinson T (2022). "The MIDAS Touch: Accurate and Scalable Missing-Data Imputation with Deep Learning." *Political Analysis*, **30**(2), 179–196.

Little RJ, Rubin D (2002). *Statistical Analysis with Missing Data (Second Edition)*. John Wiley & Sons, New York.

McKinney W (2010). "Data Structures for Statistical Computing in Python." In *Proceedings of the 9th Python in Science Conference*, volume 445, pp. 51–56. Austin, TX.

Oliphant TE (2006). *A Guide to **NumPy***. Trelgol Publishing. URL https://web.mit.edu/dvp/Public/numpybook.pdf.

Pedregosa F, Varoquaux G, Gramfort A, Michel V, Thirion B, Grisel O, Blondel M, Prettenhofer P, Weiss R, Dubourg V, Vanderplas J, Passos A, Cournapeau D, Brucher M, Perrot M, Duchesnay E (2011). "**scikit-learn**: Machine Learning in Python." *Journal of Machine Learning Research*, **12**, 2825–2830.

Rubin D (1996). "Multiple Imputation After 18+ Years." *Journal of the American Statistical Association*, **91 (434)**, 473–489.

Rubin DB (1987). *Multiple Imputation for Nonresponse in Surveys*. John Wiley & Sons, New York.

Rumelhart DE, Hinton GE, Williams RJ (1986a). "Learning Internal Representations by Error Propagation." In DE Rumelhart, JL McClelland (eds.), *Parallel Distributed Processing (Volume 1)*, pp. 318–362. MIT Press, Cambridge.

Rumelhart DE, Hinton GE, Williams RJ (1986b). "Learning Representations by Back-Propagating Errors." *Nature*, **323 (6088)**, 533–536.

Schafer JL, Olsen MK (1998). "Multiple Imputation for Multivariate Missing-Data Problems: A Data Analyst's Perspective." *Multivariate Behavioral Research*, **33 (4)**, 545–571.

Scholz M (2012). "Validation of Nonlinear PCA." *Neural Processing Letters*, **36 (1)**, 21–30.

Seabold S, Perktold J (2010). "**statsmodels**: Econometric and Statistical Modeling with Python." In *9th Python in Science Conference*.

Srivastava N, Hinton G, Krizhevsky A, Sutskever I, Salakhutdinov R (2014). "Dropout: A Simple Way to Prevent Neural Networks from Overfitting." *The Journal of Machine Learning Research*, **15 (1)**, 1929–1958.

Stekhoven DJ (2013). ***missForest**: Nonparametric Missing Value Imputation using Random Forest*. R package version 1.4.

Su YS, Gelman AE, Hill J, Yajima M (2011). "Multiple Imputation with Diagnostics (**mi**) in R: Opening Windows Into the Black Box." *Journal of Statistical Software*, **45 (2)**, 1–31.

van Buuren S, Groothuis-Oudshoorn K (2011). "**mice**: Multivariate Imputation by Chained Equations in R." *Journal of Statistical Software*, **45 (3)**, 1–68.

Vincent P, Larochelle H, Bengio Y, Manzagol PA (2008). "Extracting and Composing Robust Features with Denoising Autoencoders." In *Proceedings of the 25th International Conference on Machine Learning*, pp. 1096–1103. ACM.

Vincent P, Larochelle H, Lajoie I, Bengio Y, Manzagol PA (2010). "Stacked Denoising Autoencoders: Learning Useful Representations in a Deep Network with a Local Denoising Criterion." *Journal of Machine Learning Research*, **11**, 3371–3408.

Wilson S (2020). ***miceRanger**: Multiple Imputation by Chained Equations with Random Forests*. R package version 1.3.5, URL https://CRAN.R-project.org/package=miceRanger.

Wilson S (2021). ***miceforest**, Version 2.0.4*. URL https://github.com/AnotherSamWilson/miceforest.

Affiliation:

Ranjit Lall
Department of Politics and International Relations
Manor Road, Oxford, OX1 3UQ
United Kingdom
E-mail: ranjit.lall@politics.ox.ac.uk
URL: https://ranjitlall.github.io/

Thomas Robinson
Department of Methodology
London School of Economics and Political Science
Columbia House, Aldwych, WC2A 2AE
United Kingdom
E-mail: t.robinson7@lse.ac.uk
URL: https://ts-robinson.com/

Journal of Statistical Software http://www.jstatsoft.org/
published by the American Statistical Association http://www.amstat.org/
Volume VV, Issue II *Submitted:* yyyy-mm-dd
MMMMMM YYYY *Accepted:* yyyy-mm-dd