Journal of Statistical Software

February 2024, Volume 108, Issue 4. doi: 10.18637/jss.v108.i04

gcimpute: A Package for Missing Data Imputation

Yuxuan Zhao
Cornell University

Madeleine Udell
Stanford University

Abstract

This article introduces the Python package **gcimpute** for missing data imputation. Package **gcimpute** can impute missing data with many different variable types, including continuous, binary, ordinal, count, and truncated values, by modeling data as samples from a Gaussian copula model. This semiparametric model learns the marginal distribution of each variable to match the empirical distribution, yet describes the interactions between variables with a joint Gaussian that enables fast inference, imputation with confidence intervals, and multiple imputation. The package also provides specialized extensions to handle large datasets (with complexity linear in the number of observations) and streaming datasets (with online imputation). This article describes the underlying methodology and demonstrates how to use the software package.

Keywords: missing data, single imputation, multiple imputation, Gaussian copula, mixed data, imputation uncertainty, Python.

1. Introduction

Missing data is ubiquitous in modern datasets, yet most machine learning algorithms and statistical models require complete data. Thus missing data imputation forms the first critical step of many data analysis pipelines. The difficulty is greatest for mixed datasets, including continuous, binary, ordinal, count, nominal and truncated variables. Mixed datasets may appear either as a single dataset recording different types of attributes or an integrated dataset from multiple sources. For example, social survey datasets are generally mixed since they often contain age (continuous), demographic group variables (nominal), and Likert scales (ordinal) measuring how strongly a respondent agrees with certain stated opinions, such as the five category scale: Strongly disagree, disagree, neither agree nor disagree, agree, strongly agree. The Cancer Genome Atlas Project (https://www.cancer.gov/ccg/research/genome-sequencing/tcga) is an example of integrated mixed dataset: It contains gene expression (microarray, continuous), mutation (binary) and microRNA expression

(RNA-seq count) data. Imputation may be challenging even for datasets with only continuous variables if variables have very different scales and variability.

The Gaussian copula model nicely addresses the challenges of modeling mixed data by separating the multivariate interaction of the variables from their marginal distributions (Liu, Lafferty, and Wasserman 2009; Hoff 2007; Fan, Liu, Ning, and Zou 2017). Specifically, this model posits that each data vector is generated by first drawing a latent Gaussian vector and then transforming it to match the observed marginal distribution of each variable. In this way, ordinals result from thresholding continuous latent variables. A copula correlation matrix fully specifies the multivariate interaction and is invariant to strictly monotonic marginal transformations of the variables.

Zhao and Udell (2020b) propose to impute missing data by learning a Gaussian copula model from incomplete observation and shows empirically the resulting imputation achieves state-of-the-art performance. Following this line of work, Zhao and Udell (2020a) develop a low rank Gaussian copula that scales well to large datasets, and Zhao, Landgrebe, Shekhtman, and Udell (2022) extend the model to online imputation of a streaming dataset using online model updates. This article introduces an additional methodological advance by extending the Gaussian copula model to support truncated variables. Truncated variables are continuous variables that are truncated to an interval (which may be half-open) (see Section 2.1 and Table 2 for precise definition). One example is the zero-inflated variable: A non-negative variable with excess zeros, which often appears when a continuous variable is measured by a machine that cannot distinguish small values from zero.

Reliable decision-making with missing data requires a method to assess the uncertainty introduced by imputation. Typically, imputation software quantifies uncertainty either by providing explicit confidence intervals for imputation, or providing multiple imputations (Rubin 1996). Multiple imputations allow the end user to incorporate imputation uncertainty into subsequent analysis, for example, by conducting the desired analysis on each imputed dataset and combining the results. Zhao and Udell (2020a) derive analytical imputation confidence intervals when all variables are continuous. In this article, we further develop a multiple imputation method for Gaussian copula imputation. Furthermore, we provide confidence intervals based on multiple imputation that are valid for mixed data.

The **gcimpute** package is available at `https://pypi.org/project/gcimpute/` and implements the methodology presented in Zhao and Udell (2020a,b); Zhao *et al.* (2022) and the new advances mentioned above: It supports imputation for continuous, binary, ordinal, count, and truncated data, confidence intervals, multiple imputation, large-scale imputation using the low rank Gaussian copula model, and online imputation. Nominal variables cannot be directly modeled by a Gaussian copula model, but **gcimpute** also accepts nominal variables by one-hot encoding them into binary variables.

We present the technical background in Section 2 and demonstrate how to use **gcimpute** through examples drawn from real datasets in Section 3.

Why not other copulas? There exist many other copula models (see Jaworski, Durante, Hardle, and Rychlik 2010). However, model estimation for other copula models from incomplete data is not well studied, and the conditional distribution required by the imputation task rarely admits a closed form. These two challenges together make accurate imputation using other copula models very difficult.

Method	Large n and small p	Large n and large p
gcimpute	Yes	Yes
Amelia	Yes	No
mice	Yes	No
missForest	Yes	No
missMDA	No	Yes
softImpute	No	Yes
GLRM	No	Yes

Table 1: Suitability of imputation methods for different p (number of variables). For small n (number of samples) scenarios, simple imputation methods such as mean imputation are recommended due to limited information.

1.1. Software for missing data imputation

There are many software implementations available for imputing missing data, with R (R Core Team 2023) offering the greatest variety (Mayer, Sportisse, Josse, Tierney, and Vialaneix 2022). Most imputation packages in Python (Van Rossum *et al.* 2011) re-implement earlier R packages. For instance, **missingpy** (Bhattarai 2018) re-implements **missForest** (Stekhoven 2022) in R; **sklearn.impute.IterativeImputer** in Python (in **scikit-learn**, Pedregosa *et al.* 2011) re-implements **mice** (Van Buuren and Groothuis-Oudshoorn 2011) in R; **sklearn.impute.KNNImputer** in Python re-implements **impute** (Hastie, Tibshirani, Narasimhan, and Chu 2023) in R.

An imputation package will tend to work best on data that matches the distributional assumptions used to develop it. The popular package **Amelia** (Honaker, King, and Blackwell 2011) makes the strong assumption that the input data is jointly normally distributed, which cannot be true for mixed data. Package **missMDA** (Josse and Husson 2016) imputes missing data based on principal component analysis, and handles mixed data by one-hot encoding nominal variables. Package **mice** and **missForest** iteratively train models to predict each variable from all other variables. They handle mixed data by choosing appropriate learning methods based on each data type. Package **missForest** uses random forest models as base learners and performs well provided there are sufficient samples with non-linear relationship. In general, it yields more accurate imputations than **mice**, which uses variants of linear models (Stekhoven and Bühlmann 2012). In the computational experience of the authors, **gcimpute** outperforms **missForest** on binary, ordinal and continuous mixed data (Zhao and Udell 2020b). When the data includes nominal variables, which are poorly modeled by any of the other assumptions (low rank, joint normality, or Gaussian copula), **missForest** generally works best.

The imputation methods mentioned above typically perform better when the sample size n is large, as there is more information available to learn from. When n is small, simple imputation methods like mean imputation are often recommended due to the limited amount of available information. **Amelia**, **mice**, and **missForest** can work well when the number of variables p is small, but run too slowly for large p. Methods with weaker structural assumptions like **gcimpute** and **missForest** yield better imputations, as they are able to learn more complex relationships among the variables. Methods that rely on a low rank assumption scale well to large datasets. They tend to perform well when both n and p are large, as these data tables generally look approximately low rank (Udell and Townsend 2019), but can fail when

either n or p is small. Low rank imputation methods include **missMDA**, **softImpute** (Hastie and Mazumder 2021), **GLRM** (Udell, Horn, Zadeh, and Boyd 2016) and the low rank model from **gcimpute**. Hence, **gcimpute** provides a compelling imputation method for data of all moderately large sizes. We summarize our recommendation in Table 1.

There are also a few copula based imputation packages in R. Package **sbgcop** (Hoff 2018) uses the same model as **gcimpute** but provides a Bayesian implementation using a Markov Chain Monte Carlo (MCMC) algorithm. Package **gcimpute** uses a frequentist approach to achieve the same level of accuracy as **sbgcop** but much more quickly (Zhao and Udell 2020b). Package **mdgc** (Christoffersen 2023) amends the algorithm in Zhao and Udell (2020b) by using a higher quality approximation for certain steps in the computation, improving model accuracy but significantly increasing the runtime when the number of variables is large ($p > 100$). Package **CoImp** (Di Lascio and Giannerini 2019) uses only complete cases to fit the copula model and is unstable when most instances have missing values. In contrast, **gcimpute** can robustly fit the model even when every instance contains missing values. Moreover, **gcimpute** is the first copula package to fit moderately large datasets (large p), by assuming the copula has low rank structure, and the first to fit streaming datasets, using online model estimation.

2. Mathematical background

Package **gcimpute** fits a Gaussian copula model on a data table with missing entries and uses the fitted model to impute missing entries. It can return a single imputed data matrix with imputation confidence intervals, or multiple imputed data matrices. Once a Gaussian copula model is fitted, it can also be used to impute missing entries in new out-of-sample rows.

Let us imagine that we wish to use **gcimpute** on a data table $\mathbf{X}$ with n rows and p columns. We refer to each row $\mathbf{x}$ of $\mathbf{X}$ as a sample, and each column as a variable. Package **gcimpute** is designed for datasets whose variables admit a total order: That is, for any two values of the same variable x_1 and x_2, either $x_1 > x_2$ or $x_1 \leq x_2$. Each variable may have a distinct type: For example, numeric, Boolean, ordinal, count, or truncated. Nominal variables do not have an ordering relationship. By default, **gcimpute** encodes nominal variables as binary variables using a one-hot encoding, although other encodings are possible. However, this encoding is not self-consistent for the copula model. We advise users to model their features directly as ordinal or binary, if possible. Package **gcimpute** learns the univariate distribution of each variable without any distributional assumption, and then estimates the multivariate structure based on the learned univariate distribution.

Package **gcimpute** offers specialized implementations for large datasets and streaming datasets. Large datasets with many samples or many variables can use an efficient implementation that exploits mini-batch training, parallelism, and low rank structure. For streaming datasets, it can impute missing data immediately upon seeing a new sample and update model parameters without remembering all historical data. This method is more efficient and can offer a better fit for non-stationary data.

2.1. Gaussian copula model

The Gaussian copula (Hoff 2007; Liu *et al.* 2009; Fan *et al.* 2017; Feng and Ning 2019; Zhao and Udell 2020b) models complex multivariate distributions as transformations of latent Gaussian vectors. More specifically, it assumes that the complete data $\mathbf{x} \in \mathbb{R}^p$ is generated

Type		
Continuous	Distribution	x has CDF F.
	$f(z)$	$F^{-1}(\Phi(z))$
	$f^{-1}(x)$	$\Phi^{-1}(F(x))$
Ordinal	Distribution	x has probability mass function $\mathsf{P}(x=i)=p_i$, for $i=1,...,k$.
	$f(z)$	$\max\left\{i:\sum_{l=0}^{i-1}p_l\le\Phi(z)<\sum_{l=0}^{i}p_l\right\}$, with $p_0=0$
	$f^{-1}(x)$	$\left\{z:\sum_{l=0}^{x-1}p_l\le\Phi(z)<\sum_{l=0}^{x}p_l\right\}$, with $p_0=0$
Truncated	Distribution	x is truncated into $[\alpha,\beta]$, with $\mathsf{P}(x=\alpha)=p_\alpha$, $\mathsf{P}(x=\beta)=p_\beta$, and CDF $\tilde{F}$ conditional on $x\in(\alpha,\beta)$, which satisfies $\tilde{F}(\alpha)=0$ and $\tilde{F}(\beta)=1$.
	$f(z)$	$\begin{cases}\alpha, & \Phi(z)\le p_\alpha\\ \tilde{F}^{-1}\left(\frac{\Phi(z)-p_\alpha}{1-p_\alpha-p_\beta}\right), & \Phi(z)\in(p_\alpha,1-p_\beta)\\ \beta, & \Phi(z)\ge 1-p_\beta\end{cases}$
	$f^{-1}(x)$	$\begin{cases}\{z:\Phi(z)\le p_\alpha\}, & x=\alpha\\ \Phi^{-1}\left(p_\alpha+(1-p_\alpha-p_\beta)\tilde{F}(x)\right), & x\in(\alpha,\beta)\\ \{z:\Phi(z)\ge 1-p_\beta\}, & x=\beta\end{cases}$

Table 2: For any random variable x admitting a total order, there exists a unique monotonic transformation f such that $f(z)=x$ for a random standard Gaussian z. For each data type of x, this table includes its distribution specification, the marginal f, and the set inverse $f^{-1}(x)=\{z:f(z)=x\}$ of the marginal. Three different types of truncated variables are summarized together: (1) $\alpha=-\infty$ and $p_\alpha=0$ corresponds to lower truncated x; (2) $\beta=\infty$ and $p_\beta=0$ corresponds to upper truncated x; (3) finite α,β and positive p_α,p_β corresponds to two sided truncated x. $\Phi(\cdot)$ denotes the CDF of a standard normal variable.

as a monotonic transformation of a latent Gaussian vector $\mathbf{z}\in\mathbb{R}^p$:

$$\mathbf{x}=(x_1,\ldots,x_p)=(f_1(z_1),\ldots,f_p(z_p)):=\mathbf{f}(\mathbf{z}),\ \text{for}\ \mathbf{z}\sim\mathcal{N}(\mathbf{0},\Sigma).$$

The *marginal* transformations $f_1,\ldots,f_p:\mathbb{R}\to\mathbb{R}$ match the distribution of the observed variable $\mathbf{x}$ to the transformed Gaussian $\mathbf{f}(\mathbf{z})$ and are uniquely identifiable given the cumulative distribution function (CDF) of each variable x_j. This model separates the multivariate interaction from the marginal distribution, as the monotone $\mathbf{f}$ establishes the mapping from the latent variables to the observed variables while Σ fully specifies the dependence structure. We write $\mathbf{x}\sim\mathrm{GC}(\Sigma,\mathbf{f})$ to denote that $\mathbf{x}$ follows the Gaussian copula model with marginal $\mathbf{f}$ and copula correlation Σ.

Variables and their marginals. When the variable x_j is continuous, f_j is strictly monotonic. When the variable x_j is ordinal (including binary as a special case), f_j is a monotonic step function (Zhao and Udell 2020b). The copula model also supports one or two sided truncated variables. A one sided truncated variable is a continuous variable truncated either below or above. A variable x truncated below α has a CDF F shown below (at realization $x=x^*$):

$$F(x^*)=\mathsf{P}(x=\alpha)\mathbb{1}(x^*\ge\alpha)+(1-\mathsf{P}(x=\alpha))\tilde{F}(x^*),$$

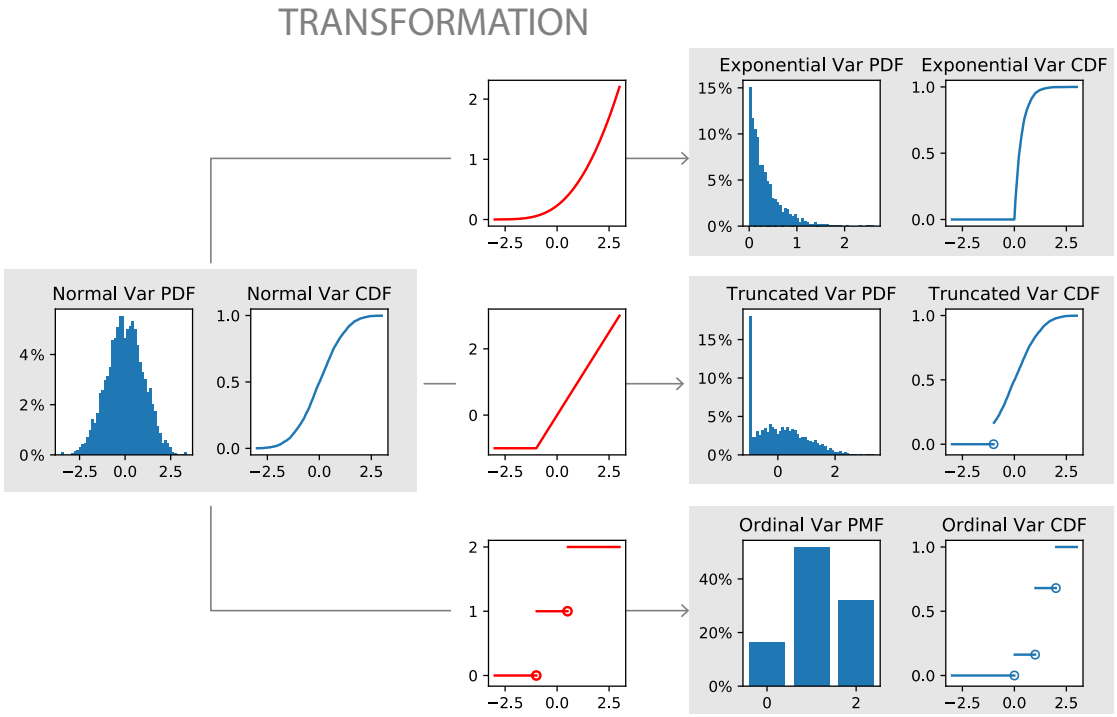

Figure 1: Three monotonic transformations of a Gaussian variable. The third column depicts the transformations that map the data distribution, visualized as both probability distribution function (histogram approximation) and cumulative distribution function (analytical form), in the left two columns to the data distribution in the right two columns.

where $\tilde{F}$ is the CDF of a random variable and satisfies $\tilde{F}(\alpha) = 0$. An upper truncated variable and two sided truncated variable are defined similarly. The CDF of a truncated variable is a strictly monotonic function with a step either on the left (lower truncated) or the right (upper truncated) or both (two sided truncated). The expression of f_j as well as their set inverse $f_j^{-1}(x_j) := \{z_j \mid f_j(z_j) = x_j\}$ are given in Table 2. In short, f_j explains how the data is generated, while f_j^{-1} denotes available information for model inference given the observed data. Figure 1 depicts how a Gaussian variable is transformed into an exponential variable, a lower truncated variable, and an ordinal variable. Figure 2 depicts the dependency structure induced by a Gaussian copula model: It plots randomly drawn samples from 2D Gaussian copula model with the same marginal distributions from Figure 1. It shows that the Gaussian copula model is much more expressive than the multivariate normal distribution.

By default, **gcimpute** categorizes a count variable to one of the above variable types based on its distribution (see Section 3.1 for the rule). Package **gcimpute** also provides a Poisson distribution modeling for count variables. The difference embodies in how to estimate f_j and f_j^{-1}. In short, the estimation of f_j and f_j^{-1} are different depending on if there is a parametric form of the function to be estimated. We defer its detailed discussion to Section 2.3 after introducing marginal transformation estimation.

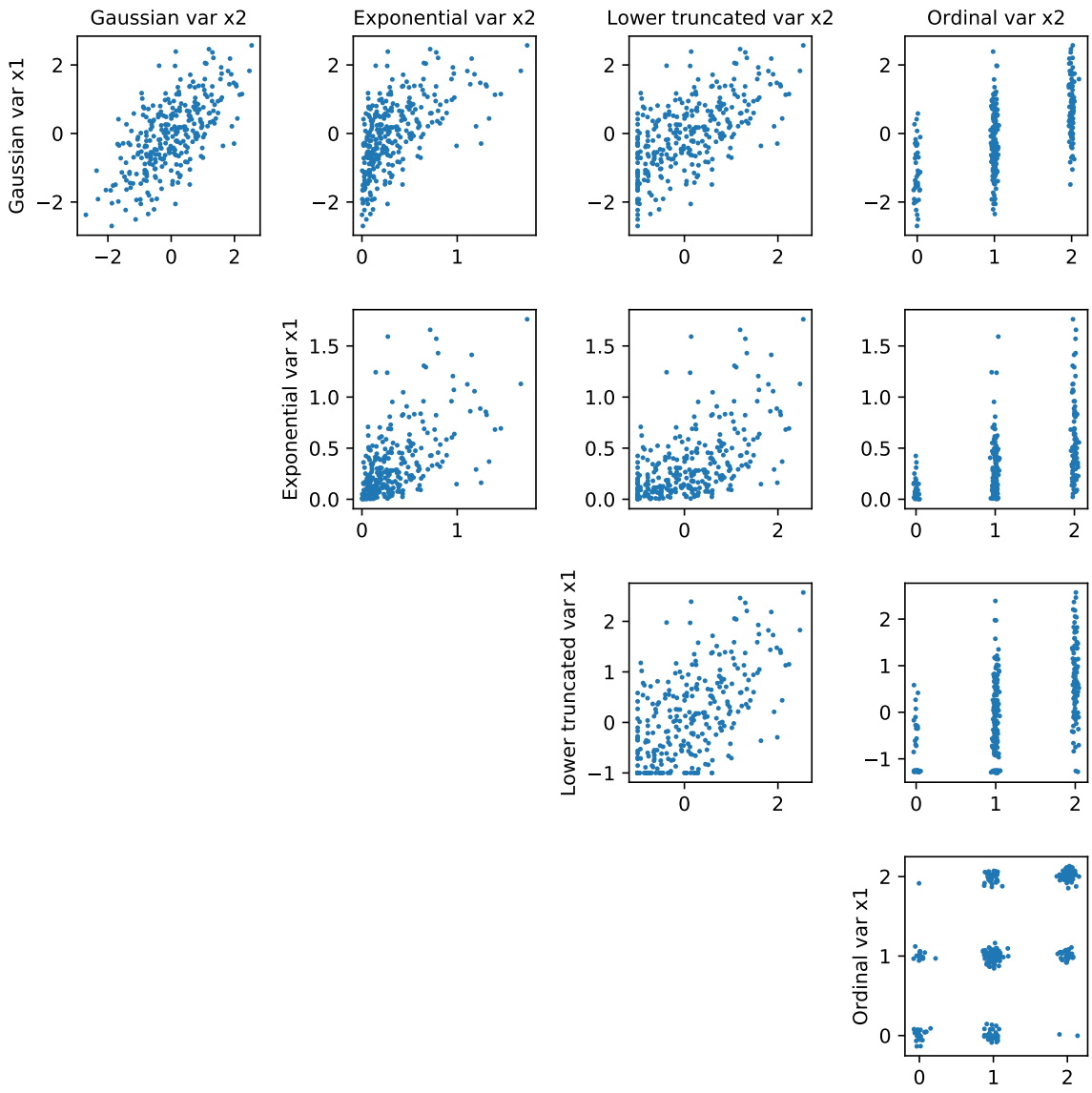

Figure 2: Scatter plot of samples from several 2D Gaussian copula models with different marginals. The data is generated by sampling (z_1, z_2) from a 2D Gaussian distribution with zero mean, unit variance and 0.65 correlation and computing $x_1 = f_1(z_1)$ and $x_2 = f_2(z_2)$, where f_1 and f_2 denote the transformations corresponding to the marginals for each model. For Gaussian marginals (1st row and 1st column), the transformation is the identity. For other marginals, the corresponding transformations are plotted as the third column of Figure 1.

2.2. Missing data imputation

Package **gcimpute** uses the observed entries, along with the estimated dependence structure of the variables, to impute the missing entries. In this section, let us suppose we have estimates of the model parameters $\mathbf{f}$ and Σ and see how to impute the missing entries. We discuss how to estimate the model parameters in the next section.

Every sample is independent conditional on $\mathbf{f}$ and Σ, so we may independently consider how to impute missing data in each sample. For a sample $\mathbf{x} \sim \mathrm{GC}(\Sigma, \mathbf{f})$, let us denote the observed variables as $\mathcal{O}$ and the missing variables as $\mathcal{M}$, so $\mathbf{x}_\mathcal{O}$ is a vector of length $\mid \mathcal{O} \mid$ that collects the observed entries. Since $\mathbf{x} \sim \mathrm{GC}(\Sigma, \mathbf{f})$, there exists $\mathbf{z} \sim \mathcal{N}(\mathbf{0}, \Sigma)$ such that $\mathbf{x} = \mathbf{f}(\mathbf{z})$. Our first task is to learn the distribution of the missing entries in the latent space. Denote $\Sigma_{I,J}$ as the sub-matrix of Σ with rows in I and columns in J. Now, $\mathbf{z}$ is multivariate normal, so

$$\mathbf{z}_\mathcal{M} \mid \mathbf{z}_\mathcal{O} \sim \mathcal{N}(\Sigma_{\mathcal{M},\mathcal{O}}\Sigma_{\mathcal{O},\mathcal{O}}^{-1}\mathbf{z}_\mathcal{O}, \Sigma_{\mathcal{M},\mathcal{M}} - \Sigma_{\mathcal{M},\mathcal{O}}\Sigma_{\mathcal{O},\mathcal{O}}^{-1}\Sigma_{\mathcal{O},\mathcal{M}}).$$

We can map this distribution to a distribution on $\mathbf{x}_\mathcal{M}$ using the marginal transformation $\mathbf{f}$.

Observations and their latent consequences. To estimate the distribution of $\mathbf{z}_\mathcal{M}$, we must model the distribution of $\mathbf{z}_\mathcal{O}$ using the observed values $\mathbf{x}_\mathcal{O}$. For an observed continuous variable value x_j, the corresponding z_j takes value $f_j^{-1}(x_j)$ with probability 1. For an observed ordinal variable value x_j, $f_j^{-1}(x_j)$ is an interval, since f_j is a monotonic step function. Hence the distribution of z_j conditional on x_j is a truncated normal in the interval $f_j^{-1}(x_j)$. For an observed truncated variable x_j, z_j takes value $f_j^{-1}(x_j)$ with probability 1 if x_j is not the truncated value, otherwise is a truncated normal in the interval $f_j^{-1}(x_j)$.

If only a single imputation is used, **gcimpute** will first compute the conditional mean of $\mathbf{z}_\mathcal{M}$ given $\mathbf{x}_\mathcal{O}$, and then return the imputation by applying the transformation $\mathbf{f}$. If multiple imputations are used, **gcimpute** will instead sample from the conditional distribution of $\mathbf{z}_\mathcal{M}$ given $\mathbf{x}_\mathcal{O}$, and then transform the sampled values by applying the transformation $\mathbf{f}$.

Package **gcimpute** can return confidence intervals for any single imputation. If all observed variables $\mathbf{x}_\mathcal{O}$ are continuous, $\mathbf{z}_\mathcal{O}$ has all probability mass at a single point and thus $\mathbf{z}_\mathcal{M}$ has a multivariate normal distribution. In this scenario, **gcimpute** first computes the normal confidence interval and then transforms it through $\mathbf{f}$ to produce a confidence interval for the imputation. In other cases, **gcimpute** computes an approximate confidence interval by assuming that $\mathbf{z}_\mathcal{O}$ has all probability mass at its conditional mean given $\mathbf{x}_\mathcal{O}$ and then computes the normal confidence interval of $\mathbf{z}_\mathcal{M}$ as it does for all continuous variables. The approximated confidence intervals are still reasonably well calibrated if there are not too many ordinal variables. Otherwise, **gcimpute** provides a safer approach to build confidence intervals by performing multiple imputation and taking a confidence interval on the empirical percentiles of imputed values.

2.3. Algorithm

Inference for the Gaussian copula model estimates the marginal transformations $f_1, \ldots, f_p$, as well as their inverses, and the copula correlation matrix Σ. The estimate of the marginal distribution and its inverse relies on the empirical distribution of each observed variable. The marginals may be consistently estimated under a missing completely at random (MCAR) mechanism. See Little and Rubin (2019, Chapter 1.3) for precise definition of missing mechanisms. Otherwise, these estimates are generally biased: For example, if larger values are missing with higher probability, the empirical distribution is not a consistent estimate of the true distribution.

Estimating the copula correlation Σ is a maximum likelihood estimate (MLE) problem. Estimates for the correlation are consistent under the missing at random (MAR) mechanism, provided the marginals are known or consistently estimated (Little and Rubin 2019).

Marginal transformation estimation. As shown in Table 2, both f_j and f_j^{-1} only depend on the distribution of the observed variable x_j. Thus to estimate the transformation, we must estimate the distribution of x_j, for example, by estimating the CDF and quantile function (for continuous and truncated) or the probability of discrete values with positive probability mass (for ordinal and truncated). Package **gcimpute** uses the empirical CDF, quantile function, or discrete probability as estimates.

All imputed values are obtained through estimated f_j, and thus the empirical quantile estimate of F_j^{-1}. For continuous variables, a linear interpolated f_j is used so that the imputation is a weighted average of the observed values. For ordinal variables, the imputation is the most likely observed ordinal level.

Suppose you know a parametric form of f_j for the observed data. Can you use this information? Should you use this information? (1) Yes, you can use this information. We include a capacity to do this in our package. (2) No, you probably should not. We have never seen a case in which using the parametric form helps in our simulation study. For example, for Poisson (count) data with a small mean, most likely values are observed, so treating the data as ordinal works well. For Poisson data with a large mean, the empirical distribution misses certain values, so certain values will never appear as imputations. Yet we find that fitting a parametric form instead barely outperforms. We believe that the dangers of model misspecification generally outweigh the advantage of a correctly specified parametric model. Parametric and nonparametric models differ most in their predictions of tail events. Alas, these predictions are never very reliable: It is difficult to correctly extrapolate the tail of a distribution from the bulk (Clauset, Shalizi, and Newman 2009).

Copula correlation estimation. Package **gcimpute** uses an expectation maximization (EM) algorithm to estimate the copula correlation matrix Σ. Suppose $\mathbf{x}^1, \ldots, \mathbf{x}^n$ are n i.i.d. samples from a Gaussian copula model, with observed parts $\{\mathbf{x}^i_{\mathcal{O}_i}\}_{i=1,\ldots,n}$. Denote their corresponding latent Gaussian variables as $\mathbf{z}^1, \ldots, \mathbf{z}^n$. At each E-step, the EM method computes the expected covariance matrix of the latent variables $\mathbf{z}^i$ given the observed entries $\mathbf{x}^i_{\mathcal{O}_i}$, i.e., $\frac{1}{n}\sum_{i=1}^n \mathsf{E}[\mathbf{z}^i(\mathbf{z}^i)^\top \mid \mathbf{x}^i_{\mathcal{O}_i}]$ and $\frac{1}{n}\sum_{i=1}^n \mathsf{E}[\mathbf{z}^i \mid \mathbf{x}^i_{\mathcal{O}_i}]$. The M-step finds the MLE for the correlation matrix of $\mathbf{z}^1, \ldots, \mathbf{z}^n$: It updates the model parameter Σ as the correlation matrix associated with the expected covariance matrix computed in the E-step. Each EM step has computational complexity $O(np^3)$ (for dense data).

2.4. Acceleration for large datasets

Package **gcimpute** runs quickly on large datasets by exploiting parallelism, mini-batch training and low rank structure to speed up inference. Our EM algorithm parallelizes easily: The most expensive computation, the E-step, is computed as a sum over samples and thus can be easily distributed over multiple cores.

When the number of samples n is large, users can invoke mini-batch training to accelerate inference (Zhao *et al.* 2022), since a small batch of samples already gives an accurate estimate of the full covariance. This method shuffles the samples, divides them into mini-batches, and uses an online learning algorithm. Concretely, for the t-th mini-batch, **gcimpute** computes the copula correlation estimate, $\hat{\Sigma}$, using only this batch and then updates the model estimate as

$$\Sigma^t = (1 - \eta_t)\Sigma^{t-1} + \eta_t \hat{\Sigma}, \tag{1}$$

where Σ^t denotes the correlation estimate and $\eta_t \in (0, 1)$ denotes the step size at iteration t. To guarantee convergence, the step size $\{\eta_t\}$ must be monotonically decreasing and satisfy $\sum_{t=0}^{\infty} \eta_t^2 < \sum_{t=0}^{\infty} \eta_t = \infty$. This online EM algorithm converges much faster as the model is updated more frequently. Zhao *et al.* (2022) report the mini-batch algorithm can reduce training time by up to 85%.

When the number of variables p is large, users can invoke a low rank assumption on the covariance to speed up training. This low rank Gaussian copula (LRGC, Zhao and Udell 2020a) assumes a factor model for the latent Gaussian variables:

$$\mathbf{z} = W\mathbf{t} + \boldsymbol{\epsilon}, \text{ where } W \in \mathbb{R}^{p\times k}, \mathbf{t} \sim \mathcal{N}(0, \mathbf{I}_k), \boldsymbol{\epsilon} \sim \mathcal{N}(0, \sigma^2\mathbf{I}_p) \text{ with } \sigma^2 > 0,$$

for some rank $k \ll p$. The k-dimensional $\mathbf{t}$ denotes the data generating factors and $\boldsymbol{\epsilon}$ denotes random noise. Consequently, the copula correlation matrix has a low rank plus diagonal structure: $\Sigma = WW^\top + \sigma^2\mathbf{I}_p$. This factorization decreases the number of parameters from $O(p^2)$ to $O(pk)$ and decreases the per-iteration complexity from $O(np^3)$ to $O(npk^2)$ for dense data. For sparse data, the computation required is linear in the number of observations. Thus, **gcimpute** can easily fit datasets with thousands of variables (Zhao and Udell 2020a).

2.5. Imputation for streaming datasets

Package **gcimpute** provides an online method to handle imputation in the streaming setting: As new samples arrive, it imputes the missing data immediately and then updates the model parameters. The model update is similar to offline mini-batch training as presented in Equation 1, with $\hat{\Sigma}$ estimated from the new samples. Online imputation methods can outperform offline imputation methods for non-stationary data by quickly adapting to a changing distribution, while offline methods are restricted to a single, static model.

Package **gcimpute** responds to the changing distribution by updating its estimate of parameters $\mathbf{f}$ and Σ after each sample is observed. The marginal estimate only uses the m most recent data points, so the model forgets stale data and the empirical distribution requires constant memory. The hyperparameter m should be chosen to reflect how quickly the distribution changes. A longer window works better when the data distribution is mostly stable but has a few abrupt changes. On the other hand, if the data distribution changes rapidly, a shorter window is needed. The correlation Σ is updated according to the online EM update after observing each new mini-batch, using a constant step size $\eta_t \in (0, 1)$. A constant step size ensures the model keeps learning from new data and forgets stale data.

Streaming datasets may have high autocorrelation, which can improve online imputation. By default, **gcimpute** imputes missing entries by empirical quantiles of the most recent stored observations. However, it also supports allocating different weights to different stored observations and imputing missing entries by empirical weighted quantiles. Package **gcimpute** provides an implementation using decaying weights for the m stored observations: d^t with $d \in (0, 1]$ for each time lag $t = 1, ..., m$. The decay rate d should be tuned for best performance. This approach interpolates between imputing the last observed value (as $d \to 0$) and the standard Gaussian copula imputation (when $d = 1$). The user may also supply their own choice of weights.

3. Software usage

This article demonstrates how to use the Python implementation of **gcimpute** (an R implementation, **gcimputeR**, is available at https://github.com/udellgroup/gcimputeR), i.e., two model classes `GaussianCopula` and `LowRankGaussianCopula`, whose core methods are displayed in Table 3. Our examples rely on some basic Python modules for data manipulation and plotting:

```
>>> import numpy as np
>>> import pandas as pd
>>> import time
>>> import matplotlib.pyplot as plt
>>> import seaborn as sns
>>> from tabulate import tabulate
```

3.1. Basic usage

To demonstrate the basic usage of **gcimpute**, we use demographic data from the 2014 General Social Survey (GSS) data (https://gss.norc.org/): We consider the variables age (`AGE`), highest degree (`DEGREE`), income (`RINCOME`), subjective class identification (`CLASS`), satisfaction with the work (`SATJOB`), weeks worked last year (`WEEKSWRK`), general happiness (`HAPPY`), and condition of health (`HEALTH`). All variables are ordinal variables encoded as integers, with varying number of ordinal categories. The integers could represent numbers, such as $0, 1, \ldots, 52$ for `WEEKSWRK`, or ordered categories, such as 1 ("Very happy"), 2 ("Pretty happy"), 3 ("Not too happy") for the question "How would you say things are these days?" (`HAPPY`). Many missing entries appear due to answers like "Don't know", "No answer", "Not applicable", etc. Variable histograms are plotted in Figure 3 using the following code:

```
>>> from gcimpute.helper_data import load_GSS
>>> data_gss = load_GSS()
>>> fig, axes = plt.subplots(2, 4, figsize = (12, 6))
```

Method	Description
`fit`	Fit a Gaussian copula from (incomplete) data
`transform`	Impute incomplete data using a Gaussian copula
`fit_transform`	Impute incomplete data using the Gaussian copula `fitted` from itself
`fit_transform_evaluate`	Conduct an evaluation on imputed data returned at each iteration during model fitting
`sample_evaluation`	Sample multiple imputed data using a Gaussian copula
`get_params`	Get parameters of the fitted Gaussian copula
`get_vartypes`	Get the specified variable types used in model fitting
`get_confidence_interval`	Get the confidence intervals for the imputed missing entries

Table 3: Overview of the core methods for both the '`GaussianCopula`' class and the '`LowRankGaussianCopula`' class.

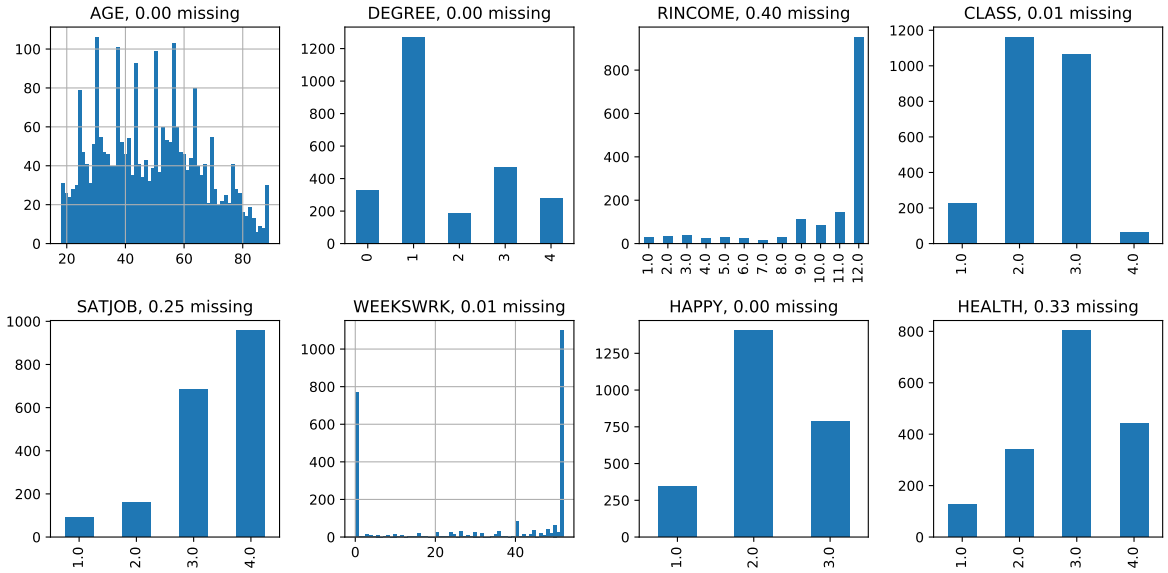

Figure 3: Histogram plots for GSS variables. There are 2538 samples in total.

```
>>> for i, col in enumerate(data_gss):
...     ax_index = np.unravel_index(i, (2, 4))
...     if col in ["AGE", "WEEKSWRK"]:
...         data_gss[col].dropna().hist(ax = axes[ax_index], bins = 60)
...     else:
...         to_plot = data_gss[col].dropna().value_counts().sort_index()
...         to_plot.plot(kind = "bar", ax = axes[ax_index]
...     _title = f"{col}, {data_gss[col].isna().mean():.2f} missing"
...     axes[ax_index].set_title(_title)
>>> plt.tight_layout()
```

We mask 10% of the observed entries uniformly at random as a test set to evaluate our imputations.

```
>>> from gcimpute.helper_mask import mask_MCAR
>>> gss_masked = mask_MCAR(X = data_gss, mask_fraction = 0.1)
```

The Python package has an API consistent with the `sklearn.impute` module (Buitinck *et al.* 2013). To impute the missing entries in an incomplete dataset, we simply create a model and call `fit_transform()`. The default choice uses `training_mode = "standard"`, corresponding to the algorithm in Zhao and Udell (2020b).

```
>>> from gcimpute.gaussian_copula import GaussianCopula
>>> model = GaussianCopula()
>>> Ximp = model.fit_transform(X = gss_masked)
```

To compare imputation performance across variables with different scales, we use scaled mean absolute error (SMAE) for each variable: The MAE of imputations scaled by the imputation

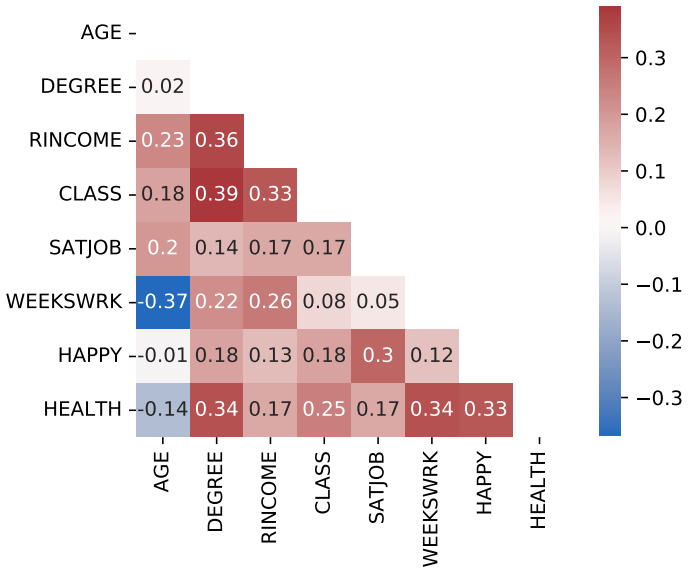

Figure 4: The estimated latent copula correlation among GSS variables.

MAE of median imputation. As shown below, the Gaussian copula imputation improves over median imputation by 10.9% on average.

```
>>> from gcimpute.helper_evaluation import get_smae
>>> smae = get_smae(Ximp, x_true = data_gss, x_obs = gss_masked)
>>> print(f"SMAE average over all variables: {smae.mean():.3f}")

SMAE average over all variables: 0.866
```

We can also extract the copula correlation estimates to see which variables are correlated, as in Figure 4. Interestingly, `DEGREE` and `CLASS` have the largest positive correlation 0.39, while `WEEKSWRK` and `AGE` have the largest negative correlation −0.37.

```
>>> corr_est = model.get_params()["copula_corr"].round(2)
>>> mask = np.zeros_like(corr_est)
>>> mask[np.triu_indices_from(mask)] = True
>>> names = data_gss.columns
>>> sns.heatmap(
...     corr_est, xticklabels = names, yticklabels = names,
...     annot = True, mask = mask, square = True, cmap = "vlag"
... )
```

Determining the variable types

The choice of variable type can have a strong effect on inference and imputation. Package **gcimpute** defines five variable types: `"continuous"`, `"ordinal"`, `"lower_truncated"`, `"upper_truncated"` and `"twosided_truncated"`. Package **gcimpute** provides good default

guesses of data types, which we used in the previous call. After fitting the model, we can query the model to ask which variable type was chosen as shown below. Only `AGE` is treated as continuous; all other variables are treated as ordinal. No variable is treated as truncated.

```
>>> for k, v in model.get_vartypes(feature_names = names).items():
>>>     print(f'{k}: {", ".join(v)}')

continuous: AGE
ordinal: DEGREE, RINCOME, CLASS, SATJOB, WEEKSWRK, HAPPY, HEALTH
lower_truncated:
upper_truncated:
twosided_truncated:
```

We can specify the type of each variable in `model.fit_transform()` directly. Otherwise, the default setting works well. It guesses the variable type based on the observed value frequency. A variable is treated as continuous if its mode's frequency is less than 0.1. A variable is treated as lower/upper/two sided truncated if its minimum's/maximum's/minimum's and maximum's frequency is more than 0.1 and the distribution, excluding these values, is continuous by the previous rule. All other variables are ordinal. The default threshold value 0.1 works well in general, but can be changed using the parameter `min_ord_ratio` in the model call `GaussianCopula()`. For example, let us look at the frequency of the min, max, and mode for each GSS variable.

```
>>> def key_freq(col):
...     freq = col.value_counts(normalize = True)
...     _min, _max = col.min(), col.max()
...     freqmid = freq.drop(index = [_min, _max])
...     key_freq = {
...         "mode": freq.max(), "min": freq[_min], "max": freq[_max],
...         "mode_freq_nominmax": freqmid.max()/freqmid.sum()
...     }
...     return pd.Series(key_freq).round(2)
>>> table = data_gss.apply(lambda x : key_freq(x.dropna())).T
>>> print(tabulate(table, headers = "keys", tablefmt = "psql"))

+----------+--------+-------+-------+-----------------+
|          |   mode |   min |   max |   mode_nominmax |
|----------+--------+-------+-------+-----------------|
| AGE      |   0.02 |  0    |  0.01 |            0.02 |
| DEGREE   |   0.5  |  0.13 |  0.11 |            0.66 |
| RINCOME  |   0.62 |  0.02 |  0.62 |            0.26 |
| CLASS    |   0.46 |  0.09 |  0.03 |            0.52 |
| SATJOB   |   0.5  |  0.5  |  0.05 |            0.81 |
| WEEKSWRK |   0.44 |  0.31 |  0.44 |            0.13 |
| HAPPY    |   0.55 |  0.31 |  0.13 |            1    |
| HEALTH   |   0.47 |  0.26 |  0.07 |            0.7  |
+----------+--------+-------+-------+-----------------+
```

Only `AGE` has mode frequency below 0.1 and thus is treated as continuous. All other variables have strong concentration on a single value, even after removing the min and max, so these are treated as ordinal. `WEEKSWRK` is an interesting example. It has 53 levels, yet 75% of the population works either 0 or 52 weeks per year: Thus it is not treated as a continuous variable. Interestingly, if we insist that `WEEKWRK` be treated as continuous, the algorithm diverges! We discuss this phenomenon in an online vignette (https://github.com/udellgroup/gcimpute/blob/master/Examples/Trouble_shooting.ipynb).

Monitoring the algorithm fitting

Package **gcimpute** considers the model to have converged when the model parameters no longer change rapidly: It terminates when $\|\Sigma^{t+1} - \Sigma^t\|_F / \|\Sigma^t\|_F$ falls below the specified `tol`, where Σ^t is the model parameter estimate at the t-th iteration and $\|\cdot\|_F$ denotes the Frobenius norm. In practice, the default value `tol = 0.01` works well and the algorithm converges in less than 30 iterations in most cases.

Tracking the objective value may also be useful. The objective value is the marginal likelihood at the observed locations, averaged over all instances. When all variables are continuous, **gcimpute** computes the exact likelihood. In other cases, **gcimpute** computes an approximation to the likelihood. The approximation behaves well in most cases including those with all ordinal variables: It monotonically increases during the fitting process and finally converges. Setting `verbose = 1` allows to monitor the parameter updates and the objective during fitting.

```
>>> model = GaussianCopula(verbose = 1)
>>> Ximp = model.fit_transform(X = gss_masked)

Iter 1: copula parameter change 0.0847, likelihood -9.6374
Iter 2: copula parameter change 0.0484, likelihood -9.5680
Iter 3: copula parameter change 0.0284, likelihood -9.5239
Iter 4: copula parameter change 0.0176, likelihood -9.4982
Iter 5: copula parameter change 0.0115, likelihood -9.4828
Iter 6: copula parameter change 0.0078, likelihood -9.4732
Convergence achieved at iteration 6
```

Using a tolerance `tol` that is too small can require many more iterations and can cause overfitting. Hence users may wish to tune `tol` for a specific dataset for best performance using `fit_transform_evaluate()`. This function runs the EM algorithm for specified `n_iter` iterations and evaluates the imputed dataset using the provided `eval_func` at each iteration. The function `eval_func` should take an imputed dataset as input and output the desired evaluation results. We can design `eval_func` to evaluate the imputation accuracy or the prediction accuracy of a supervised learning pipeline with the imputed dataset as feature matrix. For example, to evaluate the mean SMAE of the GSS dataset for up to 15 iterations, we can run the following code:

```
>>> m = GaussianCopula(verbose = 1)
>>> def err(x):
...     return get_smae(x, x_true = data_gss, x_obs = gss_masked).mean()
>>> r = m.fit_transform_evaluate(gss_masked, eval_func = err, num_iter = 15)
```

Figure 5: The imputation error among GSS variables is plotted w.r.t. the number of iterations run in **gcimpute**. Satisfying results emerge after four iterations.

```
>>> plt.plot(list(range(1, 16, 1)), r["evaluation"])
>>> plt.title("Imputation error versus run iterations")
>>> plt.xlabel("Run iterations")
>>> plt.ylabel("SMAE")
```

```
Iter 1: copula parameter change 0.0847, likelihood -9.6374
Iter 2: copula parameter change 0.0484, likelihood -9.5680
Iter 3: copula parameter change 0.0284, likelihood -9.5239
Iter 4: copula parameter change 0.0176, likelihood -9.4982
Iter 5: copula parameter change 0.0115, likelihood -9.4828
Iter 6: copula parameter change 0.0078, likelihood -9.4732
Iter 7: copula parameter change 0.0055, likelihood -9.4669
Iter 8: copula parameter change 0.0040, likelihood -9.4628
Iter 9: copula parameter change 0.0029, likelihood -9.4599
Iter 10: copula parameter change 0.0022, likelihood -9.4580
Iter 11: copula parameter change 0.0016, likelihood -9.4566
Iter 12: copula parameter change 0.0012, likelihood -9.4556
Iter 13: copula parameter change 0.0009, likelihood -9.4549
Iter 14: copula parameter change 0.0007, likelihood -9.4544
Iter 15: copula parameter change 0.0006, likelihood -9.4541
```

Shown in Figure 5, the imputation error fluctuates in a small range from 0.865 to 0.868 after four iterations. The default parameter setting of `tol = 0.01` would have stopped at iteration 6.

3.2. Acceleration for large datasets

In this section, we will see how to speed up **gcimpute** with the acceleration tools described in Section 2.4. To use parallelism with m cores, we call `GaussianCopula(n_jobs = m)`. To use mini-batching training, we set `training_mode` as `"minibatch-offline"` also in the model call `GaussianCopula()`. The low rank Gaussian copula is invoked using a different model

call `LowRankGaussianCopula(rank = k)` with desired rank k. Mini-batch training for the low rank Gaussian copula is more challenging and remains for future work, as the low rank update is nonlinear. Nevertheless, for large n and large p, the parallel low rank Gaussian copula already converges quite rapidly.

Accelerating datasets with many samples: Mini-batch training

Mini-batch training requires choosing a decaying step size $\{\eta_t\}$ in Equation 1, a batch size and a maximum number of iterations. The default setting can be simply invoked by calling `GaussianCopula(training_mode = "minibatch-offline")` or explicitly as below:

```
model_minibatch = GaussianCopula(
    training_mode = "minibatch-offline",
    stepsize_func = lambda t, c = 5: c / (c + t),
    batch_size = 100,
    num_pass = 2
)
```

The step size sequence η_t must satisfy $\eta_t \in (0, 1)$ for all t and $\sum_{t=1}^{\infty} \eta_t^2 < \sum_{t=1}^{\infty} \eta_t = \infty$. By default, we recommend using $\eta_t = c/(c + t)$ with $c > 0$. We find it generally suffices to tune c in the range $(0, 10)$. The default setting $c = 5$ works well in many of our experiments.

Mini-batch training requires a batch size $s \geq p$ to avoid inverting a singular matrix (Zhao *et al.* 2022). In practice, it is easy to select $s \geq p$, since problems with large p should use `LowRankGaussianCopula()` instead.

The maximum number of iterations matters more for mini-batch methods, because the stochastic fluctuation over mini-batches makes it hard to decide convergence based on the parameter update. Instead of specifying an exact maximum number of iterations, it may be more convenient to select a desired number of complete passes through the data (epochs), i.e., `max_iter`$= \lceil \frac{n}{s} \rceil \times$`num_pass` with s as the mini-batch size. Often using `num_pass`$= 2$ (the default setting) or 3 gives satisfying results.

We now run mini-batch training with the defaults on the GSS dataset:

```
>>> t1 = time.time()
>>> model_minibatch = GaussianCopula(training_mode = "minibatch-offline")
>>> Ximp_batch = model_minibatch.fit_transform(X = gss_masked)
>>> t2 = time.time()
>>> print(f"Runtime: {t2 - t1:.2f} seconds")
>>> smae_batch = get_smae(Ximp_batch, x_true = data_gss, x_obs = gss_masked)
>>> print(f"Imputation error: {smae_batch.mean():.3f}")

Runtime: 2.35 seconds
Imputation error: 0.871
```

Let us also re-run and record the runtime of the standard training mode:

```
>>> t1 = time.time()
>>> _ = GaussianCopula().fit_transform(X = gss_masked)
>>> t2 = time.time()
>>> print(f"Runtime: {t2 - t1:.2f} seconds")
```

```
Runtime: 7.17 seconds
```

Mini-batch training reduces runtime by 68% and achieves very similar imputation accuracy compared to standard training (imputation error 0.866).

Accelerating datasets with many variables: Low rank structure

The low rank Gaussian copula (LRGC) model accelerates convergence by decreasing the number of model parameters. Here we showcase its performance on a subset of the MovieLens1M dataset (Harper and Konstan 2015): The 400 movies with the most ratings and users who rated at least 150 of these movies in the scale of $\{1, 2, 3, 4, 5\}$. That yields a dataset consisting of 914 users and 400 movies with 53.3% of ratings observed. We further mask 10% entries for evaluation.

```
>>> gcimpute.helper_data import load_movielens1m
>>> data_movie = load_movielens1m(num = 400, min_obs = 150)
>>> movie_masked = mask_MCAR(X = data_movie, mask_fraction = 0.1)
```

We run `GaussianCopula()` as well as `LowRankGaussianCopula(rank = 10)`. Our goal is not to choose the optimal rank, but rather show the runtime comparison between two models.

```
>>> from gcimpute.low_rank_gaussian_copula import LowRankGaussianCopula
>>> a = time.time()
>>> model_movie_lrgc = LowRankGaussianCopula(rank = 10)
>>> m_imp_lrgc = model_movie_lrgc.fit_transform(X = movie_masked)
>>> print(f"LRGC runtime {time.time() - a:.2f} seconds.")
>>> a = time.time()
>>> model_movie_gc = GaussianCopula()
>>> m_imp_gc = model_movie_gc.fit_transform(X = movie_masked)
>>> print(f"GC runtime {time.time() - a:.2f} seconds.")

LRGC runtime 10.09 seconds.
GC runtime 56.75 seconds.
```

Here we already see that LRGC already reduces the runtime by 82% compared to the standard Gaussian copula, although the number of variables $p = 400$ is not particularly large. When the number of variables is much larger, the acceleration is also more important. Moreover, LRGC improves the imputation error from 0.613 to 0.577, as shown below.

```
>>> from gcimpute.helper_evaluation import get_mae
>>> mae_gc = get_mae(imp_gc, x_true = data_movie, x_obs = movie_masked)
>>> mae_lrgc = get_mae(imp_lrgc, x_true = data_movie, x_obs = movie_masked)
>>> print(f"LRGC imputation MAE: {mae_lrgc:.3f}")
>>> print(f"GC imputation MAE: {mae_gc:.3f}")

LRGC imputation MAE: 0.577
GC imputation MAE: 0.613
```

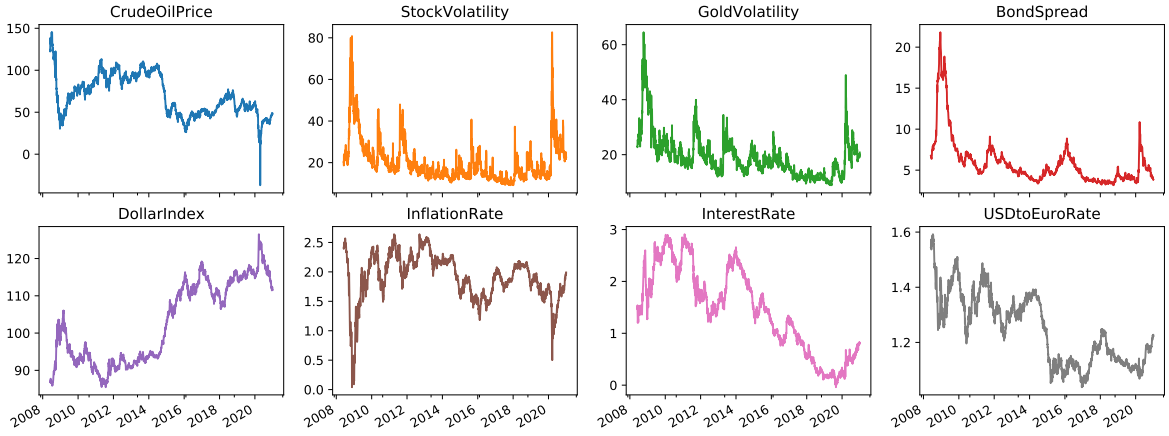

Figure 6: Values of eight selected FRED economic variables from 2008-06-03 to 2020-12-31 are plotted. For CrudeOilPrice, StockVolatility and GoldVolatility, the units are US dollars. For BondSprea, the unit is percentage. For other variables, the units are the ratios to a baseline (a historical value).

3.3. Imputation for streaming datasets

Package **gcimpute**'s `"minibatch-online"` training mode performs streaming imputation: As new samples arrive, it imputes the missing data immediately and then updates the model parameters. We showcase its performance on eight daily recorded economic time series variables from the Federal Reserve Bank of St. Louis (FRED, https://fred.stlouisfed.org/), consisting of 3109 days from 2008-06-03 to 2020-12-31. The selected eight variables are diverse and among the most popular economic variables in FRED: Gold volatility index, stock volatility index, bond spread, dollar index, inflation rate, interest rate, crude oil price, and US dollar to Euro rate, shown in Figure 6.

```
>>> from gcimpute.helper_data import load_FRED
>>> fred = load_FRED()
>>> fred.plot(
...     subplots = True, layout = (2, 4), figsize = (16, 6),
...     legend = False, title = fred.columns.to_list()
... )
```

Here we consider a scenario in which some variables are observed as soon as they are generated, while others are observed after a lag of one day. The goal is to predict the unobserved variables each day. We use stock `StockVolatility` and `CrudeOilPrice` as two unobserved variables. Each day, using a fitted Gaussian copula model, we predict their values based on both their historical values (through the marginal) and the six other observed variables at that day (through the copula correlation). After we make our prediction, the actual values are revealed and used to update the Gaussian copula model. Package **gcimpute** conveniently supports this task. Let us first create a Gaussian copula model to impute streaming datasets (`training_mode = "minibatch-online"`), shown as below.

```
>>> model = GaussianCopula(
...     training_mode = "minibatch-online",
```

```
...     window_size = 10,
...     const_stepsize = 0.1,
...     batch_size = 10,
...     decay = 0.01
... )
```

Three hyperparameters control the learning rate of the model: `window_size` controls the number of recent observations used for marginal estimation; `const_stepsize` controls the size of the copula correlation update; and `batch_size` is the frequency of the copula correlation update. In contrast, `decay` only controls the imputation and does not influence the model update (decay rate d in Section 2.5). Smaller values of `decay` put less weight on old observations, i.e., forget stale data faster. In economic time series, yesterday's observation often predicts today's value well. We use a small value `decay = 0.01`, so that the imputation depends most strongly on yesterday's observation, but interpolates all values in the window. These parameters can be tuned for best performance.

Next, to conduct the experiment described above, we prepare two data matrices with one row for each temporal observation: `X` for imputing missing entries and `X_true` for updating the model. We use first 25 rows to initialize the model.

```
>>> Xmasked = fred.assign(StockVolatility = np.nan, CrudeOilPrice = np.nan)
>>> Ximp = model.fit_transform(X = Xmasked, X_true = fred, n_train = 25)
```

More concretely, a Gaussian copula model receives the t-th row of `X`, imputes its missing entries, and then is asked to update parameters of the model using the t-th row of `X_true`. `X_true` must agree with `X` at all observed entries in `X`, but may reveal additional entries that are missing in `X`. By default, `X_true = None`, indicating no additional entries beyond `X` are available. In this example, two columns of `Xmasked` are missing: `StockVolatility` and `CrudeOilPrice`. All other columns are fully observed. All columns in `fred` are fully observed.

We now evaluate the imputation performance and compare against a simple but powerful alternative, yesterday's observation. The predicted series of both methods are almost visually indistinguishable from the true values in Figure 6, but the Gaussian copula predictions perform better on average, with lower mean squared error (MSE).

```
>>> n_train = 25
>>> for i, col in enumerate(["CrudeOilPrice", "StockVolatility"]):
...     _true = fred[col][n_train:].to_numpy()
...     _err_yes = fred[col][n_train - 1:-1].to_numpy() - _true
...     _err_GC = Ximp[n_train:, i] - _true
...     print(f"MSE of {col}:")
...     print(f"Gaussian Copula Pred: {np.power(_err_GC, 2).mean():.3f}")
...     print(f"Yesterday Value Pred: {np.power(_err_yes, 2).mean():.3f}")

MSE of CrudeOilPrice:
Gaussian Copula Pred: 3.672
Yesterday Value Pred: 4.313
MSE of StockVolatility:
Gaussian Copula Pred: 3.998
Yesterday Value Pred: 4.368
```

3.4. Imputation uncertainty

So far we have seen several methods to impute missing data. Package **gcimpute** also provides functionality to quantify the uncertainty of the imputations: Multiple imputation, confidence interval for a single imputation, and relative reliability for a single imputation. We present the first two notions here, since they are widely used. The third, relative reliability, ranks the imputation quality among all imputed entries, which is well suited for the top-k recommendation task in collaborative filtering (Zhao and Udell 2020a).

Multiple imputation

Multiple imputation creates several imputed copies of the original dataset, each having potentially different imputed values. The uncertainty due to imputations can be propagated into subsequent analyses by analyzing each imputed dataset. See Little and Rubin (2019, Chapter 5.4) for a more detailed introduction to multiple imputation. One common use case for multiple imputation is supervised learning with missing entries: A researcher creates multiple imputed feature datasets, then trains a model with each imputed training feature dataset and predicts with each imputed test feature vector. Finally, they pool all predictions into a single prediction, for example, using the mean or majority vote. An ensemble model like this often outperforms a single model trained from a single imputation.

We show how to use multiple imputation in **gcimpute** on a regression task from UCI datasets, the white wine quality dataset (Cortez, Cerdeira, Almeida, Matos, and Reis 2009). This dataset has 11 continuous features and a rating target for 4898 samples. The (transposed) header of the dataset is shown below.

```
>>> gcimpute.helper_data import load_whitewine
>>> wine = load_whitewine()
>>> print(tabulate(wine.head().T, headers = "keys", tablefmt = "psql"))
```

```
+----------------------+---------+---------+---------+----------+----------+
|                      |       0 |       1 |       2 |        3 |        4 |
|----------------------+---------+---------+---------+----------+----------|
| fixed acidity        |   7     |   6.3   |  8.1    |   7.2    |   7.2    |
| volatile acidity     |   0.27  |   0.3   |  0.28   |   0.23   |   0.23   |
| citric acid          |   0.36  |   0.34  |  0.4    |   0.32   |   0.32   |
| residual sugar       |  20.7   |   1.6   |  6.9    |   8.5    |   8.5    |
| chlorides            |   0.045 |   0.049 |  0.05   |   0.058  |   0.058  |
| free sulfur dioxide  |  45     |  14     | 30      |  47      |  47      |
| total sulfur dioxide | 170     | 132     | 97      | 186      | 186      |
| density              |   1.001 |   0.994 |  0.9951 |   0.9956 |   0.9956 |
| pH                   |   3     |   3.3   |  3.26   |   3.19   |   3.19   |
| sulphates            |   0.45  |   0.49  |  0.44   |   0.4    |   0.4    |
| alcohol              |   8.8   |   9.5   | 10.1    |   9.9    |   9.9    |
| quality              |   6     |   6     |  6      |   6      |   6      |
+----------------------+---------+---------+---------+----------+----------+
```

We now randomly mask 30% of entries and fit a Gaussian copula model to the masked dataset.

```
>>> X = data_wine.to_numpy()[:, :-1]
>>> Xmasked = mask_MCAR(X, mask_fraction = 0.3)
>>> model_wine = GaussianCopula()
>>> Ximputed = model_wine.fit_transform(X = Xmasked)
```

Now we use the first 4000 instances as a training dataset and the remaining 898 instances as test dataset. Since the goal is to show how to use multiple imputation, we use a simple linear model as the prediction model. Now, let us first examine the MSE of the linear model fitted on the complete feature dataset.

```
>>> from sklearn.metrics import mean_squared_error as MSE
>>> from sklearn.linear_model import LinearRegression as LR
>>> Xtrain, Xtest = X[:4000], X[4000:]
>>> y = wine["quality"]
>>> ytrain, ytest = y[:4000], y[4000:]
>>> ypred = LR().fit(Xtrain, ytrain).predict(Xtest)
>>> np.round(MSE(ytest, ypred), 4)
```

```
0.5121
```

Now let us examine the MSE of the linear model fitted on the single imputed dataset.

```
>>> Xtrain_imp, Xtest_imp = Ximputed[:4000], Ximputed[4000:]
>>> ypred_imp = LR().fit(Xtrain_imp, ytrain).predict(Xtest_imp)
>>> np.round(MSE(ytest, ypred_imp), 4)
```

```
 0.5295
```

Not surprisingly, replacing 30% feature values with the corresponding imputation hurts the prediction accuracy. Now let us draw 5 imputed datasets, train a linear model and get prediction for each imputed dataset, and derive the final prediction as the average across 5 different predictions. As shown below, the mean-pooled prediction improves upon the results from single imputation and has performance results very close to those of using the complete dataset.

```
>>> Ximputed_mul = model_wine.sample_imputation(Xmasked, num = 5)
>>> ypred_mul_imputed = []
>>> for i in range(5):
...     Ximputed = Ximputed_mul[..., i]
...     _Xtrain_imp, _Xtest_imp = Ximputed[:4000], Ximputed[4000:]
...     _ypred = LR().fit(_Xtrain_imp, ytrain).predict(_Xtest_imp)
...     ypred_mul_imputed.append(_ypred)
>>> ypred_mul_imputed = np.array(ypred_mul_imputed).mean(axis = 0)
>>> np.round(MSE(ytest, ypred_mul_imputed), 4)
```

```
 0.5152
```

The preceding example learns a single imputation model from both the training set and the test set, rather than learning an imputation model on the training set and then applying it to the test set. The reason for this choice is that more samples improve imputation accuracy and we assume the training and test sets have the same feature distribution. This assumption is plausible in many scenarios, for example, when the train set and test set are randomly split. However, when this assumption fails, we can use a safer alternative: Train imputation models and impute missing values on the training set and the test set separately.

Imputation confidence intervals

Confidence intervals (CI) are another important measure of uncertainty. Package **gcimpute** can return a CI for each imputed value: For example, a 95% CI should contain the true missing data with probability 95%. In general, these CI are not symmetric around the imputed value due to the nonlinear transformation $\mathbf{f}$. We will continue to use the white wine dataset for illustration. After fitting the Gaussian copula model, we can obtain the imputation CI as shown below.

```
>>> ct = model_wine.get_confidence_interval()
>>> upper, lower = ct["upper"], ct["lower"]
```

By default, the method `get_confidence_interval()` extracts the imputation CI of the data used to fit the Gaussian copula model, with significance level `alpha = 0.05`. The empirical coverage of the returned CI is 0.943, as shown below. Hence we see the constructed CI are well calibrated on this dataset.

```
>>> missing = np.isnan(Xmasked)
>>> Xmissing = X[missing]
>>> cover = (lower[missing] < Xmissing) & (upper[missing] > Xmissing)
>>> np.round(cover.mean(), 3)
```

```
0.943
```

The default setting uses an analytic expression to obtain the CI. As in Section 2.2, when some variables are not continuous, a safer approach builds CI using empirical quantiles computed from multiple imputed values. Let us now construct the quantile CI and compare them with the analytical counterparts. Here we follow the default setting to use 200 samples for computing quantiles. As shown below, the quantile CI has almost the same empirical coverage rate as the analytical CI, validating that the CI are well calibrated.

```
>>> ct_q = model_wine.get_confidence_interval(type = "quantile")
>>> upper_q, lower_q = ct_q["upper"], ct_q["lower"]
>>> cover_q = (lower_q[missing] < X_missing) & (upper_q[missing] > X_missing)
>>> np.round(cover_q.mean(), 3)
```

```
0.942
```

4. Concluding remarks

Package **gcimpute** supports a variety of missing data imputation tasks including single imputation, multiple imputation, imputation confidence intervals, as well as imputation for large datasets and streaming datasets. As a complement to this article, we provide usage vignettes (https://github.com/udellgroup/gcimpute/blob/master/Examples) detailing more specific topics such as trouble shooting, relative reliability for a single imputation, etc.

Although this article focuses on missing data imputation, **gcimpute** can also be used to fit a Gaussian copula model to complete mixed datasets. The resulting latent correlations may be useful to understand multi-view data collected on the same subjects from different sources. As far as we know, no other software supports Gaussian copula estimation for mixed continuous, binary, ordinal and truncated variables. Fan *et al.* (2017) only support continuous and binary mixed data; Feng and Ning (2019) support continuous, binary and ordinal mixed data; Yoon, Carroll, and Gaynanova (2020) support continuous, binary and zero-inflated (a special case of truncated) mixed data.

Package **gcimpute** estimates the model provably well when data is missing uniformly at random (MCAR), and can estimate the copula provably well given the marginals if the data is missing at random (MAR). Adapting the theory to handle data missing not at random (MNAR) is challenging. However, we find empirically that **gcimpute** still performs reasonably well in this setting. Indeed, many different missing patterns may be called MNAR, and imputation methods designed for one MNAR mechanism do not necessarily outperform on other MNAR data due to this heterogeneity. We advise users to make the choice by evaluating on a validation dataset.

References

Bhattarai A (2018). ***missingpy**: Missing Data Imputation for Python*. Python package version 0.2.0, URL https://pypi.org/project/missingpy.

Buitinck L, Louppe G, Blondel M, Pedregosa F, Mueller A, Grisel O, Niculae V, Prettenhofer P, Gramfort A, Grobler J, Layton R, VanderPlas J, Joly A, Holt B, Varoquaux G (2013). "API Design for Machine Learning Software: Experiences from the **scikit-learn** Project." In *ECML PKDD Workshop: Languages for Data Mining and Machine Learning*, pp. 108–122.

Christoffersen B (2023). ***mdgc**: Missing Data Imputation Using Gaussian Copulas*. R package version 0.1.7, URL https://CRAN.R-project.org/package=mdgc.

Clauset A, Shalizi CR, Newman MEJ (2009). "Power-Law Distributions in Empirical Data." *SIAM Review*, **51**(4), 661–703. doi:10.1137/070710111.

Cortez P, Cerdeira A, Almeida F, Matos T, Reis J (2009). "Modeling Wine Preferences by Data Mining from Physicochemical Properties." *Decision Support Systems*, **47**(4), 547–553. doi:10.1016/j.dss.2009.05.016.

Di Lascio FML, Giannerini S (2019). ***CoImp**: Copula Based Imputation Method*. R package version 1.0, URL https://CRAN.R-project.org/package=CoImp.

Fan J, Liu H, Ning Y, Zou H (2017). "High Dimensional Semiparametric Latent Graphical Model for Mixed Data." *Journal of the Royal Statistical Society B*, **79**(2), 405–421. doi:10.1111/rssb.12168.

Feng H, Ning Y (2019). "High-Dimensional Mixed Graphical Model with Ordinal Data: Parameter Estimation and Statistical Inference." In *The 22nd International Conference on Artificial Intelligence and Statistics*, pp. 654–663.

Harper FM, Konstan JA (2015). "The Movielens Datasets: History and Context." *ACM Transactions on Interactive Intelligent Systems*, **5**(4), 1–19. doi:10.1145/2827872.

Hastie T, Mazumder R (2021). ***softImpute**: Matrix Completion via Iterative Soft-Thresholded SVD*. R package version 1.4-1, URL https://CRAN.R-project.org/package=softImpute.

Hastie T, Tibshirani R, Narasimhan B, Chu G (2023). ***impute**: Imputation for Microarray Data*. doi:10.18129/B9.bioc.impute. R package version 1.76.0.

Hoff PD (2007). "Extending the Rank Likelihood for Semiparametric Copula Estimation." *The Annals of Applied Statistics*, **1**(1), 265–283. doi:10.1214/07-aoas107.

Hoff PD (2018). ***sbgcop**: Semiparametric Bayesian Gaussian Copula Estimation and Imputation*. R package version 0.980, URL https://CRAN.R-project.org/package=sbgcop.

Honaker J, King G, Blackwell M (2011). "**Amelia II**: A Program for Missing Data." *Journal of Statistical Software*, **45**(7), 1–47. doi:10.18637/jss.v045.i07.

Jaworski P, Durante F, Hardle WK, Rychlik T (2010). *Copula Theory and Its Applications*. Springer-Verlag. doi:10.1007/978-3-642-12465-5.

Josse J, Husson F (2016). "**missMDA**: A Package for Handling Missing Values in Multivariate Data Analysis." *Journal of Statistical Software*, **70**(1), 1–31. doi:10.18637/jss.v070.i01.

Little RJA, Rubin DB (2019). *Statistical Analysis with Missing Data*, volume 793. John Wiley & Sons.

Liu H, Lafferty J, Wasserman L (2009). "The Nonparanormal: Semiparametric Estimation of High Dimensional Undirected Graphs." *Journal of Machine Learning Research*, **10**(10).

Mayer I, Sportisse A, Josse J, Tierney N, Vialaneix N (2022). "R-Miss-Tastic: A Unified Platform for Missing Values Methods and Workflows." *The R Journal*, **14**, 244–266. doi:10.32614/rj-2022-040.

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.

R Core Team (2023). *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria. URL https://www.R-project.org/.

Rubin DB (1996). "Multiple Imputation after 18+ Years." *Journal of the American Statistical Association*, **91**(434), 473–489. doi:10.1080/01621459.1996.10476908.

Stekhoven DJ (2022). ***missForest***: *Nonparametric Missing Value Imputation Using Random Forest*. R package version 1.5, URL https://CRAN.R-project.org/package=missForest.

Stekhoven DJ, Bühlmann P (2012). "MissForest – Non-Parametric Missing Value Imputation for Mixed-Type Data." *Bioinformatics*, **28**(1), 112–118. doi:10.1093/bioinformatics/btr597.

Udell M, Horn C, Zadeh R, Boyd S (2016). "Generalized Low Rank Models." *Foundations and Trends® in Machine Learning*, **9**(1), 1–118. doi:10.1561/2200000055.

Udell M, Townsend A (2019). "Why Are Big Data Matrices Approximately Low Rank?" *SIAM Journal on Mathematics of Data Science*, **1**(1), 144–160. doi:10.1137/18m1183480.

Van Buuren S, Groothuis-Oudshoorn K (2011). "**mice**: Multivariate Imputation by Chained Equations in R." *Journal of Statistical Software*, **45**(3), 1–67. doi:10.18637/jss.v045.i03.

Van Rossum G, *et al.* (2011). *Python Programming Language*. URL https://www.python.org/.

Yoon G, Carroll RJ, Gaynanova I (2020). "Sparse Semiparametric Canonical Correlation Analysis for Data of Mixed Types." *Biometrika*, **107**(3), 609–625. doi:10.1093/biomet/asaa007.

Zhao Y, Landgrebe E, Shekhtman E, Udell M (2022). "Online Missing Value Imputation and Change Point Detection with the Gaussian Copula." In *Proceedings of the AAAI Conference on Artificial Intelligence*.

Zhao Y, Udell M (2020a). "Matrix Completion with Quantified Uncertainty through Low Rank Gaussian Copula." In *Advances in Neural Information Processing Systems*, volume 33.

Zhao Y, Udell M (2020b). "Missing Value Imputation for Mixed Data via Gaussian Copula." In *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pp. 636–646.

Affiliation:

Yuxuan Zhao
Department of Statistics and Data Science
Cornell University
Ithaca, NY 14850, United States of America
E-mail: yz2295@cornell.edu
URL: https://sites.coecis.cornell.edu/yuxuanzhao/

Madeleine Udell
Management Science and Engineering
Stanford University
Stanford, CA 94305, United States of America
E-mail: udell@stanford.edu
URL: https://web.stanford.edu/~udell/

Journal of Statistical Software https://www.jstatsoft.org/
published by the Foundation for Open Access Statistics https://www.foastat.org/
February 2024, Volume 108, Issue 4 *Submitted:* 2022-08-22
doi:10.18637/jss.v108.i04 *Accepted:* 2023-06-30