A Batch Process for High Dimensional Imputation

Philip D. Waggoner*
Columbia University & YouGov America

Forthcoming, *Computational Statistics*

Abstract

This paper describes a correlation-based batch process for addressing high dimensional imputation problems. There are relatively few algorithms designed to efficiently handle imputation of missing data in high dimensional contexts. Fewer still are flexible enough to natively handle mixed-type data, often requiring lengthy pre-processing to get the data into proper shape, and then post-processing to return the data to usable form. Such decisions as well as assumptions made by many methods (e.g., data generating process) limit their performance, flexibility, and usability. Built on a set of complementary algorithms for nonparametric imputation via chained random forests, I introduce a batching process to ease computational costs associated with high dimensional imputation by subsetting data based on ranked cross-feature absolute correlations. The algorithm then imputes each batch separately, and joins imputed subsets in the final step. The process, hdImpute, is fast and accurate. As a result, high dimensional imputation is more accessible, and researchers are not forced to decide between speed or accuracy. Complementary software is available in the form of an R package, and is openly developed on Github under the MIT public license. In the spirit of open science, collaboration and engagement with the actively developing software are encouraged.

Keywords: Imputation, High dimensional data, Chained random forests, Missing data

*pdw2119@columbia.edu. I am grateful to Doug Rivers, Delia Bailey, and other colleagues at YouGov for many helpful conversations and support throughout, as well as the Institute for Social and Economic Research and Policy (ISERP) at Columbia. Any mistakes remain my own.

1 Introduction

Complete data are increasingly rare in the modern (big) data landscape. Data sets across virtually all disciplines and domains include some degree of missingness. The general task of imputing missing data is to learn from natural patterns in data and then imputing plausible values for the missing cases. Though intuitively straightforward, imputation in practice remains a complex task. Many algorithms, models, and processes have been offered to address the problem of missing data (Lavanya, Reddy and Eswara Reddy, 2019; Shah et al., 2017; Van Buuren, 2018; Zhao and Long, 2016). But solutions are often appropriate for only a handful of domains and not one-size-fits-all, because the cause of missing data, the dimensionality of the data space, the class(es) of features that comprise the data, approaches to pre- and post-processing, and the substantive interpretation of imputed values vary widely across domains. Further, the implementation and accessibility of more nuanced solutions to domain-flavored versions of high dimensional missing data problems are often limited by lack of availability of open software. As such, while the problem of missing data is ubiquitous and the need for imputation is clear, the *solutions* are typically more nuanced, context-specific, and aimed at addressing special cases of missingness. One of these special cases making solutions increasingly complex is high dimensional data. This is especially the case as bigger data become more commonplace. That is, the size (rows) and dimensionality (columns) of data are growing across most fields as the production and expectations of data are simultaneously and rapidly growing.

As a result, I introduce a new approach to ease the complexity of imputation in a high dimensional context, applicable in virtually any domain. To widen access to the approach, I also offer complementary open software to implement the process described in this paper. The process, which is built on top of two well-established algorithms for imputation and described more below, results in low runtimes and accurate solutions with no pre-processing of input features required.

High dimensional data are notoriously difficult to work with as odd phenomena can occur in high dimensional space. For example, dubbed the "curse of dimensionality," distances between observations in a low dimensional setting become increasingly obscured and potentially meaningless as the dimensionality of the data space grows (De Marchi, 2005; Rhys, 2020). This curse shows up in many disciplines, from engineering (Bessa et al., 2017; Daum and Huang, 2003) and machine learning (Bach, 2017; Verleysen and François, 2005), to the social sciences (De Marchi, 2005; Salas and Yepes, 2019) and medical research (Berisha et al., 2021; Chattopadhyay and Lu, 2019). Beyond the curse, a variety of other tasks such as detecting patterns (Waggoner, 2021) or solving partial differential equations (Han, Jentzen and Weinan, 2018) are complicated by the dimensionality of a data space. Indeed, these and other issues are present across virtually every domain, and increasingly so in the age of big data (Bollier, Firestone et al., 2010)

Therefore, if working with data in general to complete various tasks is complicated by the size and dimensionality of the data space, *learning* from the space to effectively impute missing data is all the more complicated. When data are missing across hundreds or even thousands of columns in a data matrix, it becomes less clear how best to first, think about the missingness, and then second, how best to go about imputing plausible values for the missing cases. Assuming a mechanism of missingness has been detected (e.g., MCAR, MAR, MNAR; Little and Rubin (2019)), it is a difficult task to determine how best to go about imputation. This includes decisions regarding whether the model should be parametric or not; how best to pre-process input features; which classes the algorithm can (and cannot) handle; the assumed underlying data generating process and accompanying distributional assumptions to be made; and many others.

Another complication in high dimensional imputation is the tradeoff between speed and accuracy. Some of the better (more accurate) solutions may not converge if the computational costs are too demanding and runtimes too long. Whereas, alternatively, the results from very fast algorithms may be useless if the imputed values are implausible (e.g., modal imputation in many cases). Navigating this tradeoff is a central task of *any* imputation effort, and especially in a high dimensional setting where calculations simply take more time as there are more data combinations to consider. In sum, the problems and complications associated with all data spaces and in all imputations tasks are often magnified in high dimensional data settings.

Given these complexities and considerations, I offer a new way of addressing high dimensional data imputation flowing mostly from the tradeoff between speed and accuracy. At the heart of the present approach called hdImpute, is simplifying the imputation task by first reducing the dimensionality the data space through creation of correlation-based batches of columns in a data matrix. That is, cross-feature correlations calculated across all columns, and then ranked by absolute correlation magnitude. Then, batches of a given size (specified by the researcher) are built and result in smaller subsets of data to work with, given that an entire data space may not be *simultaneously* required for generating plausible imputed values. In short, both speed and accuracy are possible with hdImpute through a hyperparameter that controls the batch size.[1] The smaller the batch size, the faster the algorithm. But the corollary as it relates to *accuracy* is not always the case with hdImpute. That is, accuracy is not necessarily dependent on the batch size as one might expect, i.e., learning from more or less of the data across different fits of the algorithm. Through several benchmarking experiments using real and simulated data with varying amounts of missingness detailed below, the *amount of missing data* is shown to play a prominent role in the accuracy of the solution, in addition to the batch size. When more or less data are missing, the algorithm learns differently across different batch sizes. A modest amount of tuning will often yield highly accurate results, relative to several other comparable algorithms as detailed later in the paper. Importantly, as hdImpute is built on top of chained random forests as the imputation engine, regardless of the batch sizes or learning patterns, hdImpute remains extremely fast, requires no pre-processing of input features (unless desired by the researcher), natively handles mixed-type data, and bypasses data generating process assumptions through nonparametric imputation.

The paper proceeds accordingly. Next, the hdImpute process is introduced. Then, a series of benchmarking experiments using real data are presented, including a detailing of these results in the form of tables, figures, and discussion. This is followed by a similar set of experiments but using simulated data in four contexts. Then, an additional section details results from a hyperparameter grid search to offer guidance on batch size selection. The paper then overviews the complementary open source software used to run hdImpute (the `hdImpute` R package) along with several syntactic details and pointers to keep in mind for optimal use. Concluding remarks are offered at the end.

2 hdImpute: A Batch Process for High Dimensional Imputation

The hdImpute algorithm eases the computational burden associated with imputation, which is particularly pronounced in high dimensional data settings. To do so, hdImpute subsets across columns in a data matrix based on column-wise correlations, and then imputes each subset/batch individually. The batching process follows a series of steps to create batches, and then impute missing values for each batch using a nonparametric chained random forest engine. The imputed batches are joined in a final stage, and the completed data set is returned. This section walks

[1] The hyperparameter is an argument called `batch` in the hdImpute software detailed later in the paper.

through each step of the hdImpute sequence.

The first step is to build a correlation matrix across all input features, from which ranked batches are created. Of note, if a correlation matrix is already obtained or exists, then, this step is not necessary.[2] Once the correlation matrix is obtained, the matrix is flattened to give the ranking of all features on the basis of absolute correlations, from 0 to 1. This step informs the next step of batch creation. Small batches are created, without regard for the data type/column class, from the correlation-ranked list of features by proceeding element-wise down the list up to the specified batch size, which is set by the user. The ability to consider mixed-type data simultaneously during batch creation is extremely useful as it prevents the need for lengthy pre-processing of input features to coerce all features into a common data type as required by many imputation algorithms such as `softImpute`, which requires quantitative data as input (Hastie, Mazumder and Hastie, 2013). As the size of each batch is controlled by a tuning parameter, if, for example, the batch size is set at 10 and there are 100 features/columns in the data matrix, then upon ranking the features based on correlations, there would be 10 batches each consisting of 10 features.

Then, with the batches created, hdImpute leverages nonparametric chained random forests to impute missing values on a batch-by-batch basis. As the goal of this paper is practical rather than theoretical, I present only a brief overview of chained random forests, and point interested readers to the paper introducing the `missForest` algorithm (Stekhoven and Bühlmann, 2012) for details on the method. The first step of chained random forests is to make a guess at a plausible value for missing cases typically using a simple nonparametric approach like mean imputation, reminiscent of the first step in several matrix completion approaches to imputation. "Then, sort the variables...according to the amount of missing values starting with the lowest amount. For each variable [with missingness], the missing values are imputed by first fitting an RF [(random forest) using the observed values]; then, predicting the missing values...by applying the trained RF to [variables with missing values]. The imputation procedure is repeated until a stopping criterion is met" (Stekhoven and Bühlmann, 2012: 113). The repetition of the process is the chaining part, which is similar in intuition to the MICE algorithm (Van Buuren and Oudshoorn, 1999; Van Buuren, 2018).

The `missForest` algorithm, which implements chained random forests for imputation, was selected as the imputation engine for several reasons. First, it is nonparametric, and thus avoids any assumptions about the data generating process or distributions. Second, `missForest` easily handles mixed-class data types. That is, a data matrix comprised of a few or many classes of data, with types ranging from factor, to string, to floats and everything in between, may simply be passed to the algorithm, which natively handles these classes and imputes accordingly. This prevents the need for any pre-processing of input features. This is an extremely valuable feature of the algorithm given the prevalence of mixed-type data in so many domains, e.g., in survey research with many different response categories and options, or in medical research with many classes of patient-level features (e.g., medical history, characteristics, illness types, past treatments, etc.). And third, a recent expansion of `missForest`, called `missRanger` (Mayer, 2019), implemented `missForest` using the very fast `ranger` R package for fitting random forests on top of a C++ engine (Wright and Ziegler, 2015). The `missRanger` package allows for extremely fast computation, combined with all of the benefits of `missForest`.

Despite `missRanger`'s speed, hdImpute's process of imputation on small batches of features

[2]Creation of the correlation matrix is not a formal step in the hdImpute algorithm, but it can easily be built using the `hdImpute` package as described in Section 6.

offers a remarkable increase in speed over many other approaches to high dimensional imputation, while avoiding a loss in accuracy. In fact, as shown in the experiments below, hdImpute's accuracy is better than other widely used approaches such as matrix completion via `softImpute` (Hastie, Mazumder and Hastie, 2013) in nearly all conditions, and even faster than `missRanger` in all conditions.

The key to balancing between speed and accuracy with hdImpute is the batch size, which is tuned by the user. The effective balancing of speed and accuracy is seen below in a key finding, which is that both learning from data patterns as well as the amount of missingness to impute plausible values can be optimized by examining subsets of the data at a time, instead of all data simultaneously. While ultimately *all* data is still used by hdImpute across the batches, the experimental results demonstrate that reasonable and accurate imputed values require only subsets of the full data to be imputed at a time, which has the added benefit of significantly speeding up the imputation process. When data are extremely sparse (i.e., 99% missingness in the third experimental condition later) similar to contexts like the "Netflix Problem" (Bennett, Lanning et al., 2007), very small batches of 10 features offer the most accurate *and* fastest approach to imputation compared to all other models, including other versions of hdImpute with larger batch sizes, as well as the `missRanger` and `softImpute` algorithms. A key takeaway from this is that if all of the data are not *simultaneously* required to learn a good imputation solution, then all of the data should not be fed a priori to the algorithm in an effort to avoid slowing down computation.

Of note, while there are many benefits to using chained random forests via `missForest` as the imputation engine as addressed to this point, there are also some drawbacks to both the method (chained random forests) and the algorithm (`missForest`) to consider. Most notably, regarding the method, while adding trees to the forest, and also deepening trees often leads to improved accuracy, these can quickly become computationally expensive choices. Gaining ground in accuracy in this way must be weighed against the cost in speed and efficiency when tuning *any* random forest. The same is certainly a drawback for use of random forests in imputation. Further, regarding the algorithm, recent work demonstrated that in highly skewed nonlinear contexts, `missForest` can produce biased regression coefficient estimates (Hong and Lynn, 2020). Effectively, any predictive model can be used in a chaining process. For example, some prefer a boosting algorithm to impute missing data. The miceforest algorithm is one such algorithm that uses LightGBM on the back end, but chains like another other chained algorithm to generate imputations (Wilson, 2022). Hong and Lynn (2020) recommend a calibrated version of random forests, `CALIBERrfimpute`, to be used in highly skewed nonlinear scenarios. Ultimately, the drawbacks of random forests are a threat at some level and in some form for *all* predictive models that make assumptions and choices in computation as well as how the data are handled to generate plausible imputations. As the limitations are ostensibly universal and often at the margins in many use cases, the alternative modeling options did not outweigh the many benefits of the version of chained random forests via `missForest` used in hdImpute.

2.1 A Simple Example

Consider a basic example of the hdImpute sequence to clarify movement across stages: correlation matrix, flatten and rank the features, impute batches based on batches built from ranking.

Suppose we start with a simple raw data frame.

With these data, begin with calculating the cross-feature matrix of absolute correlations.

feat_1	feat_2	feat_3
...	...	...
...	...	...
...	...	...

$$\begin{bmatrix} |\rho(\text{feat}_1, \text{feat}_1)| & \dots & \dots \\ |\rho(\text{feat}_2, \text{feat}_1)| & |\rho(\text{feat}_2, \text{feat}_2)| & \dots \\ |\rho(\text{feat}_3, \text{feat}_1)| & |\rho(\text{feat}_3, \text{feat}_2)| & |\rho(\text{feat}_3, \text{feat}_3)| \end{bmatrix},$$

giving the following correlations

$$\begin{bmatrix} |\rho|_{11} = 1.00 & \dots & \dots \\ |\rho|_{21} = 0.11 & |\rho|_{22} = 1.00 & \dots \\ |\rho|_{31} = 0.75 & |\rho|_{32} = 0.37 & |\rho|_{33} = 1.00 \end{bmatrix}.$$

With the correlation matrix produced, progress to flatten the matrix, and rank the features based on absolute correlations. To do so, first order by absolute correlation accordingly.

Row	Column	$\lvert\rho\rvert$
feat_1	feat_3	0.75
feat_2	feat_3	0.37
feat_2	feat_1	0.11

Next, rank and return a vector of the feature rankings.

Rank	Feature
1	feat_3
2	feat_1
3	feat_2

Importantly, the ranking is based on the summed correlation coefficients, s.t., feature 1 = 0.86 total ($|\rho|_{12} + |\rho|_{13}$), feature 2 = 0.48 total ($|\rho|_{12} + |\rho|_{23}$), and feature 3 = 1.12 total ($|\rho|_{13} + |\rho|_{23}$).

From here, we build batches of subsets of columns, based on the rankings. So, in this simple case, if the batch size were set to 2 (i.e., `batch = 2`), then batch 1 would include features 3 and 1, and then batch 2 would include only feature 2. Of note, the batch sizes will include the number of features *up to* the value passed to the `batch` hyperparameter. So in this simple case, as the batch size is `2`, there are a maximum of two features per batch. But given that there is an odd number of features in the data frame, the last batch will comprise the remaining *unbatched* features, which in this case includes a single feature (feat_2). Deciding which features belong to which batch is a key task of the algorithm. Once features are ranked based on absolute correlations, the batches are defined simply by dividing the remaining features in batches of size `batch`. Using the default implementation of the algorithm, then, the user cannot move features between or to different batches, as batch creation happens automatically. However, this type of flexible batch creation is certainly possible, if the user proceeds in steps and defines their own batches once the cross-correlations have been

calculated and the features have been ranked. More on the iterative, step-by-step approach (and flexibility therein) is addressed below in the section reviewing the complementary software.

With the batches defined, we progress to the final stage, which is to impute batches individually using chained random forests. Then, upon imputing each batch, we join the columns together at the end to return a complete, imputed data set.

At an intuitive level, the idea behind imputing individual batches at a time is similar to other dimension reduction techniques such as the LASSO, subset selection, or even principal components analysis, where estimation or pattern detection is eased by reducing the dimensionality of the data space by making use of relationships and patterns that naturally exist across the full data space. hdImpute is very similar in this regard. Yet, where hdImpute differs from these and other dimension reduction techniques is in the amount of data fed to the algorithm at a given time. In other words, for hdImpute, all data is used to impute, just not at one time. The reason this is valuable is based in the generic task of imputation, which is based on many calculations across various combinations across all rows and columns in a data matrix, which are increasingly expensive, time consuming, and complex to make as the dimensionality of the data space increases. As such, the motivation underlying hdImpute is to simplify these complications, which are magnified in high dimensional settings, while still making use of *all* of the data as well as exploiting natural relationships that exist across the features of the data, seen via the starting place of a correlation matrix.

3 Benchmarking Experiments: Real Data

There are three approaches to experimentally exploring hdImpute: experiments with real data, experiments with simulated data, and a grid search experiment to find the optimal batch size.

This first section presents results from several benchmarking experiments using real data. Three conditions are defined by varying amounts of missingness imposed on real, complete data: 30% missingness, 80% missingness, and 99% missingness. True values are masked completely at random. Error is calculated by comparing the imputed values to the true values. The models compared across the three conditions are: six versions of hdImpute with varying batch sizes (10, 50, 195, 390, 520, and 779 features), `missRanger` (which is equivalent to hdImpute with a batch size of p, i.e., all features in the full data), and finally matrix completion with alternating ridge regression via the `softImpute` algorithm. All models are fit using default values to maximize comparability. In general, fitting models consisted of passing the masked/missing versions of the full data frames to the algorithm, making adjustments as needed. For example, in the hdImpute context, the batch size was set. And for softImpute, the data needed to first be transformed to a matrix of all quantitative inputs. Each run of each model was timed (seconds) and recorded in order to compare each against the other in the second part of the results reporting, after inspecting and comparing the error across each model.

The data used in the first set of experiments are high dimensional data on wafer manufacturing from Machine Hack, and are archived at Kaggle.[3] The data are complete with no missingness, and are comprised of 2519 rows and 1558 columns. The original set was comprised of individual training and testing sets. For present purposes, these were combined to give a full, single set of data. Though these data are not extremely high dimensional as is common in several fields, e.g., genomics, they are big enough to demonstrate the value of hdImpute relative to other widely used algorithms for high dimensional imputation, in that they allow for varying sized batches to be created and corresponding versions of the hdImpute algorithm to be compared.

[3] `https://www.kaggle.com/subham07/detecting-anomalies-in-water-manufacturing`

3.1 Results: Comparing Error Across Models

The main results comparing error across all models and all conditions of missingness (30, 80, 99%) are displayed in Table 1. The bold cell entries highlight the lowest error, signifying the most accurate model compared to all others. Cell entries displaying error are normalized root mean squared error (NRMSE), given the need to compare error across varying classes of input features and across multiple models (Oba et al., 2003).

Table 1: Comparing Error from Models Fit on Real Data

Model	30% Missingness	80% Missingness	99% Missingness
hdImpute (batch=10)	0.724	0.735	**0.722**
hdImpute (batch=50)	0.719	0.742	0.729
hdImpute (batch=195)	0.617	0.766	1.356
hdImpute (batch=390)	0.582	0.697	1.141
hdImpute (batch=520)	**0.473**	0.746	0.900
hdImpute (batch=779)	0.473	0.699	1.165
softImpute	1.011	1.010	1.004
missRanger	0.497	**0.669**	0.781

From Table 1, hdImpute is the best (most accurate) in two of the three missingness conditions (30% and 99%), and `missRanger` is best in the 80% condition. In the 30% condition, the version of hdImpute with larger sized batches (520 features per batch) was best, whereas in the extreme missingness condition with 99% missing data, the version of hdImpute with much smaller sized batches (10 features per batch) was best. See the optimal models visually in Figure 1.

The dashed line represents NRMSE > 1.0, which is akin to random imputation. Thus, realistic models should have an NRMSE < 1.0 to suggest imputation is useful, regardless of the algorithm.

Interestingly, though `missRanger` was best in the 80% condition, two versions of hdImpute were relatively close in error: `missRanger` = 0.669, hd390 = 0.697, hd779 = 0.699. We see a similar result for other instances when hdImpute was best, with `missRanger` performing similarly to hdImpute. And of note, all versions of `softImpute` performed extremely poorly (worse than guessing) in all conditions. Practically, this means softImpute generated odd values not on the underlying scale, which are not only useless for assessing the quality of the fit, but may also be dangerous for inferences made using imputed values. Consider the results visually in Figures 2 and 3.

Beyond high and low values of error, in Figure 2 we can also see that as the sparsity of the data increases, so too does the error, meaning more mistakes made in imputation for all models *except* hd10 and hd50. This suggests that versions of hdImpute with *smaller* batch sizes learn better from data than other algorithms, including versions of hdImpute with larger batch sizes. This point is a recurring theme to this point and throughout, which is that speed and accuracy are not opposite extremes on a single pole. But rather are separate dimensions entirely, which with a modest amount of tuning (i.e., a single hyperparameter for `hdImpute`), can both be optimized as seen with the hdImpute fit with the smallest batch size ("hd10"), which had the lowest error *and*

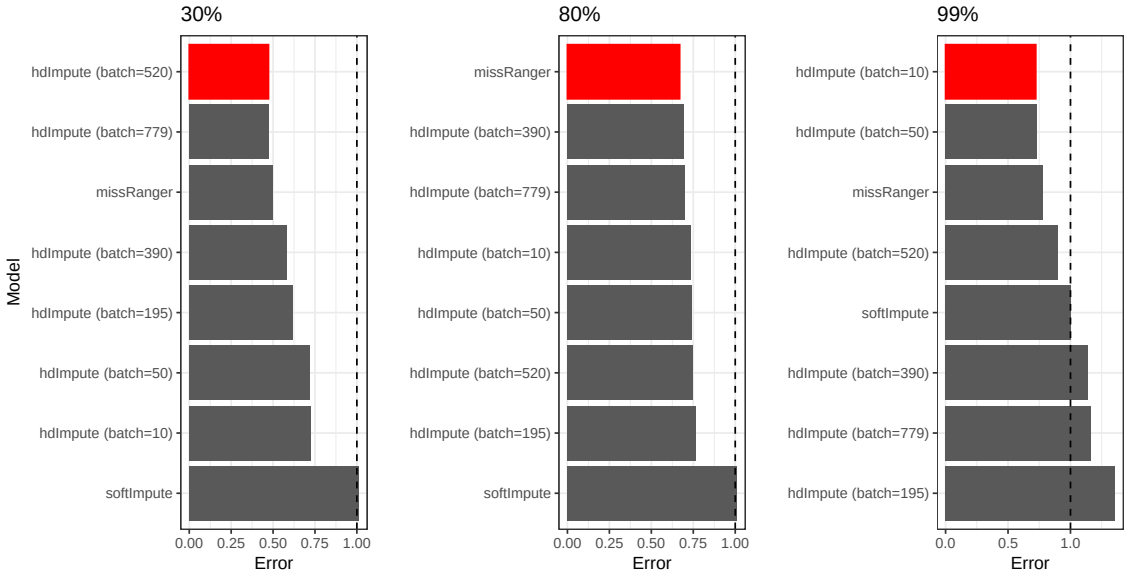

Figure 1: Optimal Models Based on `min`(Error)

the fastest runtime among the realistic models (i.e., models better than guessing with NRMSE $<$ 1.0) as we will see below in Table 2 and Figures 4 and 5.

Given the previous note that an NRMSE value $>=$ 1.0 can be thought of as random imputation for present purposes, Figure 3 zooms in on all models with an NRMSE value $<$ 1.0 to give a tighter look at the error across the useful models.

We get a cleaner look at the optimal models, which show two versions of hdImpute with larger batch sizes for the 30% missingness condition, and then two different versions of hdImpute with smaller batch sizes in the sparse 99% missingness condition. Note also that `missRanger` is hovering around the lowest NRMSE across all conditions as well, though only optimal in the 80% missingness condition.

In the case of least amount of missing data (30%), hdImpute with larger batch sizes were best with the lowest error, compared to scenarios of extremely high missingness (99%), which saw optimal hdImpute fits with the smallest batch sizes. Specifically, when 30% missingness optimal models were hd520 and hd779, with 0.47283 and 0.47347, respectively. Yet, in the 99% missing data condition, the hd10 and hd50 models had the lowest, nearly identical in error at 0.72224 and 0.72947, respectively. And in this extreme sparsity condition, hdImpute models with the largest batch sizes performed the worst. To reiterate, then, this suggests that in data contexts with relatively low amounts of missing data, hdImpute requires more data (larger batch sizes) to impute realistic, better values. This is compared to scenarios with a large amount of missing data, where better imputed values are learned based on smaller subsets of the data.

The latter point is important in high dimensional data contexts, because the deeper reality is that whether the data are extremely sparse or not, the algorithm is flexible enough to accommodate all scenarios of dimensionality and missingness, giving more or less accuracy based on patterns in the data. This is learning from data in a very real sense.

A question at this point, though, might fairly be, *with `missRanger` and select hdImpute versions*

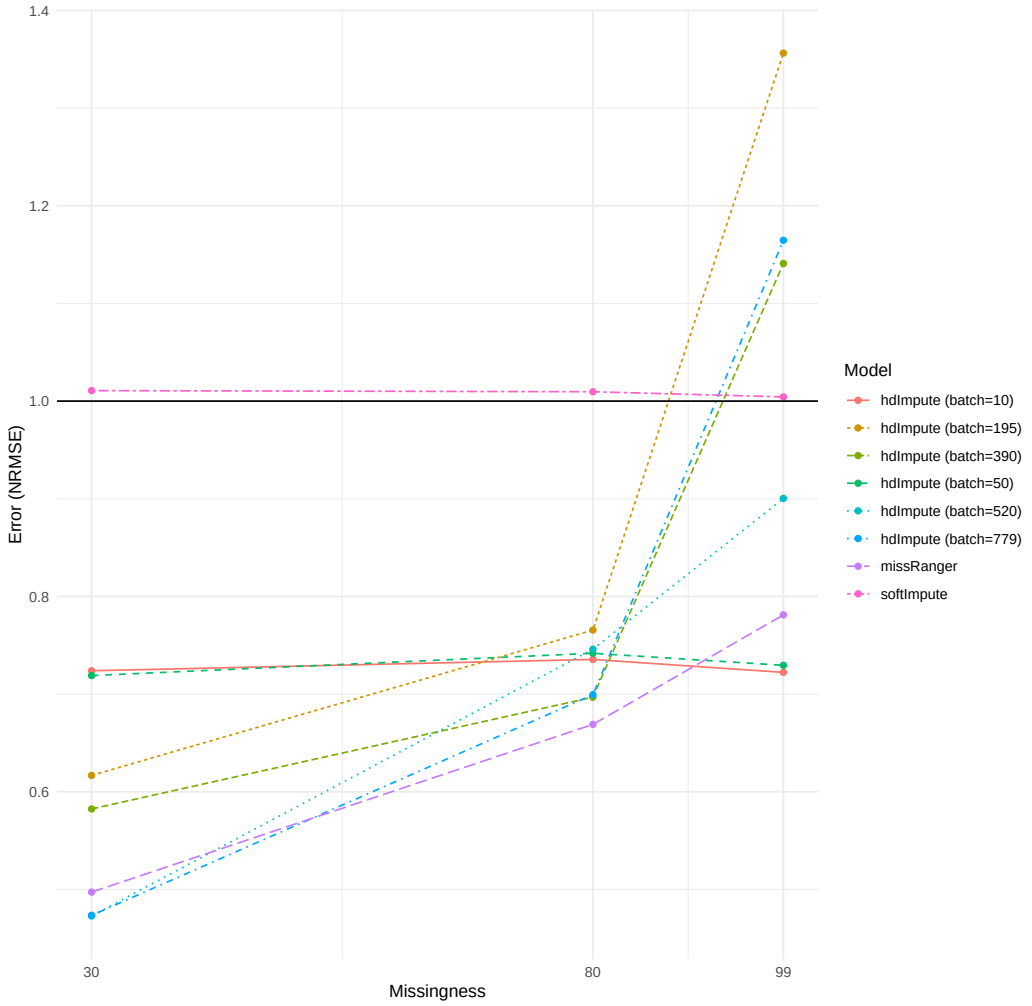

Figure 2: Comparing Error (NRMSE) across All Models

performing so similarly across all missingness conditions, why not stick with `missRanger` *and avoid complicating the imputation process?* Though fair, the *error* of the imputation procedure is not the only consideration, especially in high dimensional contexts. The *speed* at which we can arrive at an optimal imputation result is also a very important consideration, especially in business contexts and production pipelines where delivery speed is a very real component of the value of the product. This value of speed is even more pronounced when the differences between error rates are extremely small as they were in several cases mentioned above. Thus, with often similarly low error rates, users may be willing to sacrifice a small amount of error for a significant gain in speed. We will explicitly explore this question and tradeoff in the following section comparing runtimes across the same models.

3.2 Results: Comparing Runtime Across Models

To begin with the benchmarking results for runtimes, see the runtime comparisons across all models in Table 2. Cell entries are in seconds. Similar to Table 1, bold entries depict the fastest model.

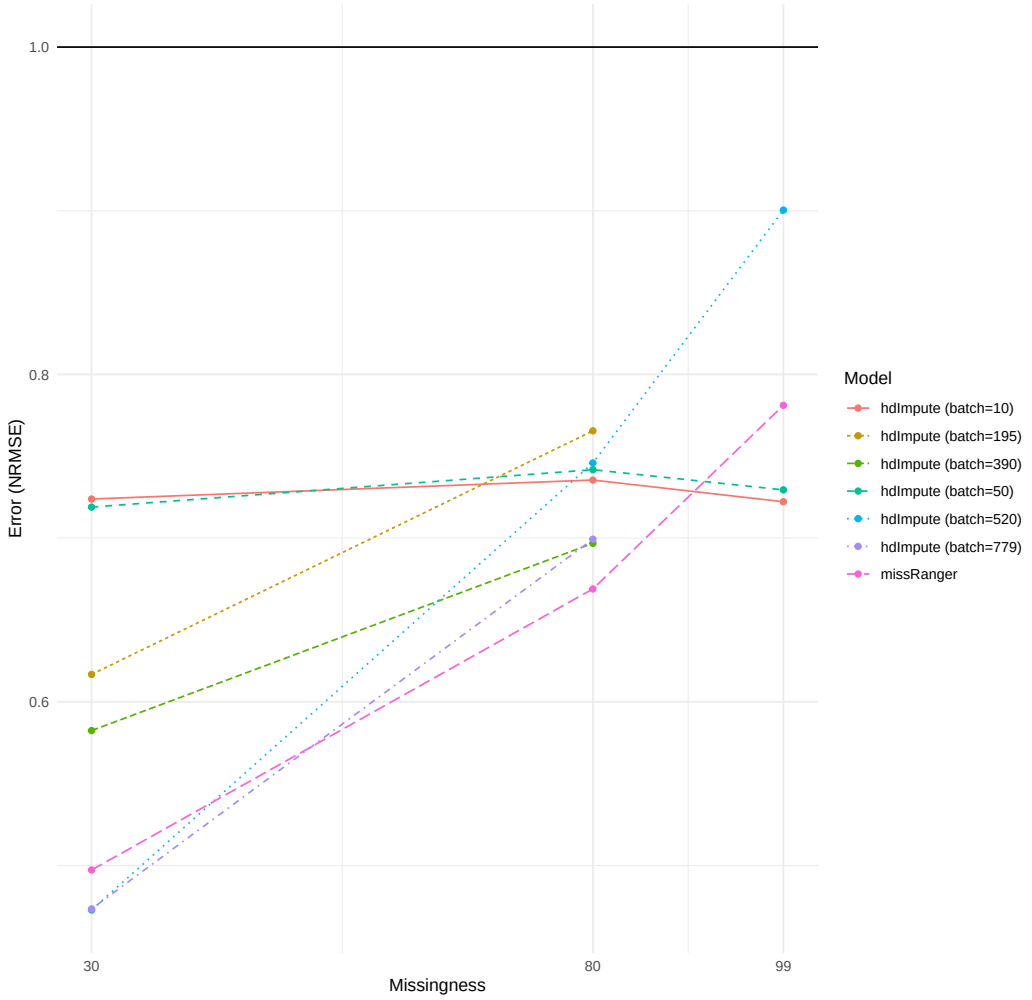

Figure 3: Comparing Error (NRMSE < 1.0)

It is clear from Table 2 that the `softImpute` algorithm is extremely fast by a large factor compared to all other models, with fits taking only 1-4 seconds across all missingness conditions. However, taken with the previous discussion and display of error in Table 1, the `softImpute` models all performed very poorly, and worse than *random* imputation as all NRMSE values were greater than 1.0.[4] As such, considering only the models with reasonable accuracy (`missRanger` and most versions of hdImpute), the dominant trend is a significant benefit of speed from all hdImpute runs compared to all runs of `missRanger`. Consider a visual look at these results in Figures 4 and 5.

While speed certainly slows for hdImpute versions with increasingly large batch sizes, even the slowest version of hdImpute (hd779 in the 99% missingness condition) with a runtime of 2604 seconds, is more than 2 times faster than the fastest version of `missRanger` (5362 seconds in the 30%

[4]Of note, all algorithms were run using the default settings. It is certainly conceivable that some tuning could result in a more accurate (and likely slower) fit of `softImpute`. However, the same could also be said for `missRanger` and all iterations of hdImpute. Thus, in the absence of precise model tuning given the poor comparability across models in such a case, default values were used.

Table 2: Comparing Runtime (Seconds) from Models Fit on Real Data

Model	30% Missingness	80% Missingness	99% Missingness
hdImpute (batch=10)	170	116	179
hdImpute (batch=50)	286	188	272
hdImpute (batch=195)	604	449	704
hdImpute (batch=390)	1082	979	1066
hdImpute (batch=520)	1612	1012	1409
hdImpute (batch=779)	2276	2455	2604
softImpute	1	2	4
missRanger	5362	8175	8841

missingness condition), and 3.4 times faster than the slowest version of `missRanger` (8841 seconds in the 99% missingness condition). Comparing more directly with error, in the 99% missingness condition, where the best result was from hdImpute with a batch size of 10, the runtime for this model was only 179 seconds. This is compared to the `missRanger` model in the same condition (99%), which was 49.4 times slower to run at 8841 seconds, and still resulting in a worse NRMSE of 0.781 compared to the hd10 model with an NRMSE of 0.722. In such a case of extreme missingness, where only 1% of the data is complete, the hdImpute algorithm with the smallest batch size resulted in the most accurate imputed values for the data compared to all other algorithms *and* it took the least amount of time to get there, excluding the `softImpute` outlier due to extremely high and unrealistic error as previously mentioned.

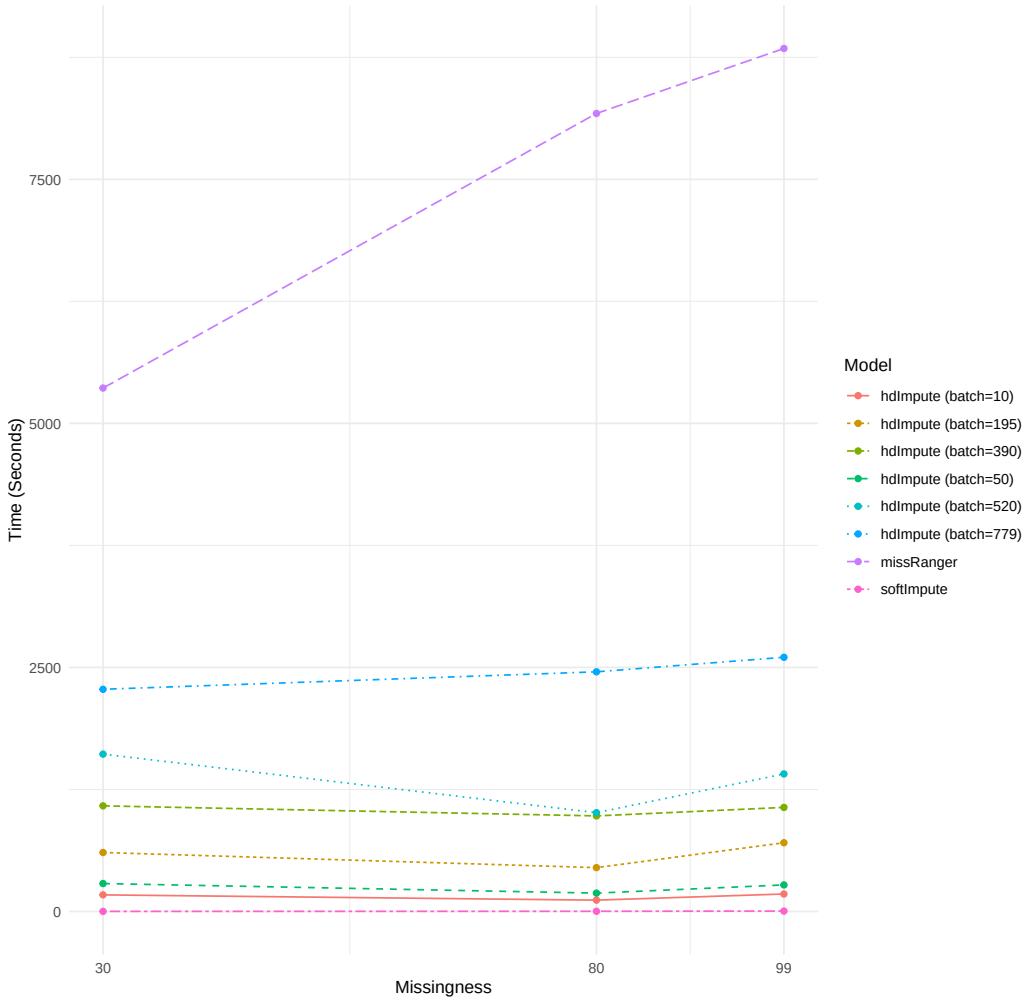

Figure 4: Comparing Runtime

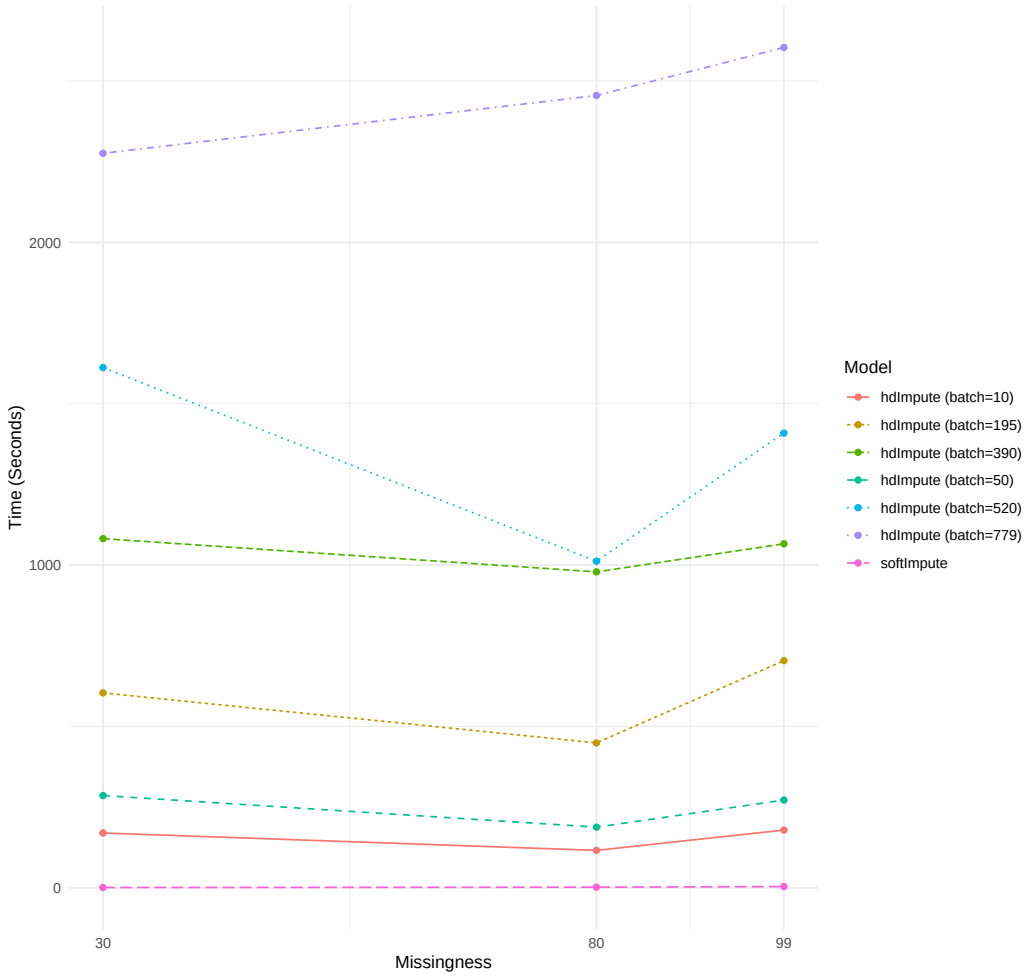

Figure 5: Comparing Runtime (Dropping `missRanger`)

4 Benchmarking Experiments: Simulated Data

In this section, we will work with a new set of experimental results. Building on the findings detailed in the previous sections with real data, we will explore only hdImpute's performance on a series of *simulated* scenarios. The purpose behind the section is to both focus on hdImpute for the remainder of the paper, and also to consider additional frequently occurring contexts in which the method may be used. To this end, the following scenarios are simulated, varying both data structure and rates of missingness:

- N (2,000 rows) > P (1,000 columns) with 30% missingness
- N (2,000 rows) > P (1,000 columns) with 70% missingness
- N (1,000 rows) < P (2,000 columns) with 30% missingness
- N (1,000 rows) < P (2,000 columns) with 70% missingness

The experimental set up began with simulating the two data scenarios, both with a compound symmetry correlation structure of the variance-covariance matrix (Goldfeld and Wujciak-Jens, 2020). This choice was made in order to emulate data from a realistic sample or study, with some similarity among units in a common data space but not uniformity, while avoiding appending randomly simulated values together in a data frame. With each data scenario created, either 30% or 70% of the data were amputed giving a total of four simulated scenarios that are similar to data that may be collected from a real study or sample.

Once the four scenario data frames were constructed, I passed each to the hdImpute algorithm with a common batch size set at 25. A constant batch size was set in order to directly compare error and timing across these scenarios. Yet, varying the batch size in an effort to find the optimal batch size by some definition (e.g., a batch size associated with minimal error) is address and explored in the following section with a grid search. Upon fitting each, the error and runtimes were recorded. These are presented and discussed in Tables 3 and 4.

4.1 Results: Comparing Error Across Scenarios

See the results reporting the error from the fit of hdImpute (batch size of 25) to each scenario from the simulated data in Table 3. Recall, the cell entries reporting error are reporting the normalized root mean squared error (NRMSE), given the need to compare error across multiple models with different features.

As error values > 1.0 are functionally useless, the first notable point is that all error values are much less than 1.0, meaning we have useful, non-random and plausible imputed values from this simulation exercise in all four scenarios.

Also, it is interesting to note that, while the error is lowest in both missingness conditions in the N > P scenario, the error across both scenarios in *30% missingness* condition was nearly identical. This is compared to similar, but more starkly different errors in the 70% condition by comparison. This suggests that, as expected at an intuitive level, the results are more accurate in cases with less missingness. This makes sense given the construction of the algorithm begins with building a correlation matrix. Yet, at a more substantive level, this also suggests that we are getting realistic performance from hdImpute, in that it does better when there is more observed data to learn from, and though still performing well, it does worse by comparison when there is less data to learn from (i.e., more missingness, as in the 70% condition).

Table 3: Comparing Error from Models Fit on Simulated Data

Scenario	30% Missingness	70% Missingness
N (2,000) > P (1,000)	**0.408**	**0.510**
N (1,000) < P (2,000)	0.409	0.550

4.2 Results: Comparing Runtime Across Scenarios

Next, consider the results from the runtimes from the four scenarios from simulated data presented in Table 4. Again, recall that the cell entries reporting the runtimes are in seconds. Also, these runtimes include the building and flattening of the correlation matrix, the former of which is the slowest part of the process.

Of note, the fastest version of the model in each missingness condition is, perhaps unsurprisingly, in the context with fewer columns, as there are fewer batches to sort through, impute, and join. That is, the N > P scenario saw the fastest runtimes in both missingness conditions. While the difference is not overly large in both, it is strikingly close in the 70% missingness condition, with runtimes at 128 and 132 seconds respectively, in N > P, and N < P scenarios.

Still, these differences seem to be mostly at the margins, as the greatest span of difference across all four scenarios and conditions is 74 second (from the slowest of 202 to the fastest of 128).

Table 4: Comparing Runtime (Seconds) from Models Fit on Simulated Data

Scenario	30% Missingness	70% Missingness
N (2,000) > P (1,000)	**169**	**128**
N (1,000) < P (2,000)	202	132

In sum, across all experimental contexts, the most useful take away is that the hdImpute algorithm is performing stably and well, giving plausible imputed values for missing cases, across varying rates of missingness and data structures. This is encouraging for use in general.

Further, as hinted at in the real data experiments above, and reinforced in these findings, the batch size is an important and largely data-dependent factor conditioning performance and quality of the imputations as well as the speed of the algorithm. *But how should the batch size be selected?* This question is taken up in the following section.

5 Grid Search to Find Optimal Batch Size

In this section, we consider the question of how a user might select the batch size, when it's effect on the results is largely dependent on the data and missingness of those data, which of course is variable and project-dependent. Given the inability to establish for a formal definition of optimality in light of these constraints, we instead start with a simple heuristic that is widely recognized as a reliable way to evaluate any predictive model's performance: error. We have implicitly relied on assessing and comparing error across all versions of all models in this paper. Therefore, we begin by suggesting the optimal version of hdImpute, and thus the batch size, is the batch size that results in the smallest amount of error in the imputations.

With a goal in mind for the batch size and model performance, a relatively simple approach to finding this value is a grid search. That is, set up a sequence of models, varying the batch size. And after each fit, record the error. Then, after all versions are fit and all error recorded, plot the error against the batch size, and select the batch size associated with the lowest error. This is the process we follow in this section.

Before presenting and discussing results, though, it is important to highlight that this is a very informal approach to determining optimality. Indeed, we are not calculating optimality relative to a formally defined benchmark. As this is difficult to define given the above-mentioned constraints, instead the goal at present is to offer one (of many) way(s) to think about searching for an optimal (or *acceptable*) batch size, relative to a widely acknowledged and defensible heuristic, which is low error. And this heuristic-style approach is applicable to all applications of hdImpute. As such, users should treat this section more as advice, rather than a formal defense of an approach or method.

With that, the first step in setting up the grid search is to define a range of batch sizes to search over: For the grid search, we use the one of the simulated scenarios (N (2,000) > P (1,000)) with 30% missingness. We then fit 20 versions of the model, each with a different batch size. The batch size, which defined the grid over which we searched, was from 5 to 1,000, increasing by increments of 50, giving the following batch sizes: 5, 55, 105, 155, 205, 255, 305, 355, 405, 455, 505, 555, 605, 655, 705, 755, 805, 855, 905, and 955. The grid search took 5.3 hours to run on a computer with 8 cores. See the results in Figure 6.

From the figure, a couple of point stand out. First, and foremost, the optimal batch size according to our heuristic of lowest error, is size 55. This is highlighted in red and labeled in Figure 6. Also, and perhaps more informatively, we now have a better sense of the distribution of error over a range of batch sizes. For these data, the error is highest at batch size of 5 and drastically drops at the next model, which is also the lowest at batch size 55. Yet, it slowly starts to increase. Yet, this increase is in a relatively flat way, spanning an NRMSE error range from about 0.38 at the lowest to 0.42 at the biggest batch size of 955. The anomalous case is a batch size of 5, with an NRMSE at nearly 0.55, suggesting the algorithm is unable to sufficiently learn from such small subsets of the full data. More data is helpful to a point, but slowly becomes inefficient as the batch size is at it's maximum.

Regardless of the specific values and the optimal batch size, this exercise is also meant to offer a way to think about and select an acceptable (and defensible) batch size for any application of hdImpute. Though the range of error and the range of batch sizes will vary and be dependent on the data and application in question, this section has introduced a simple way to think about searching for and selecting a batch size, while also visualizing the search results in an intuitive way.

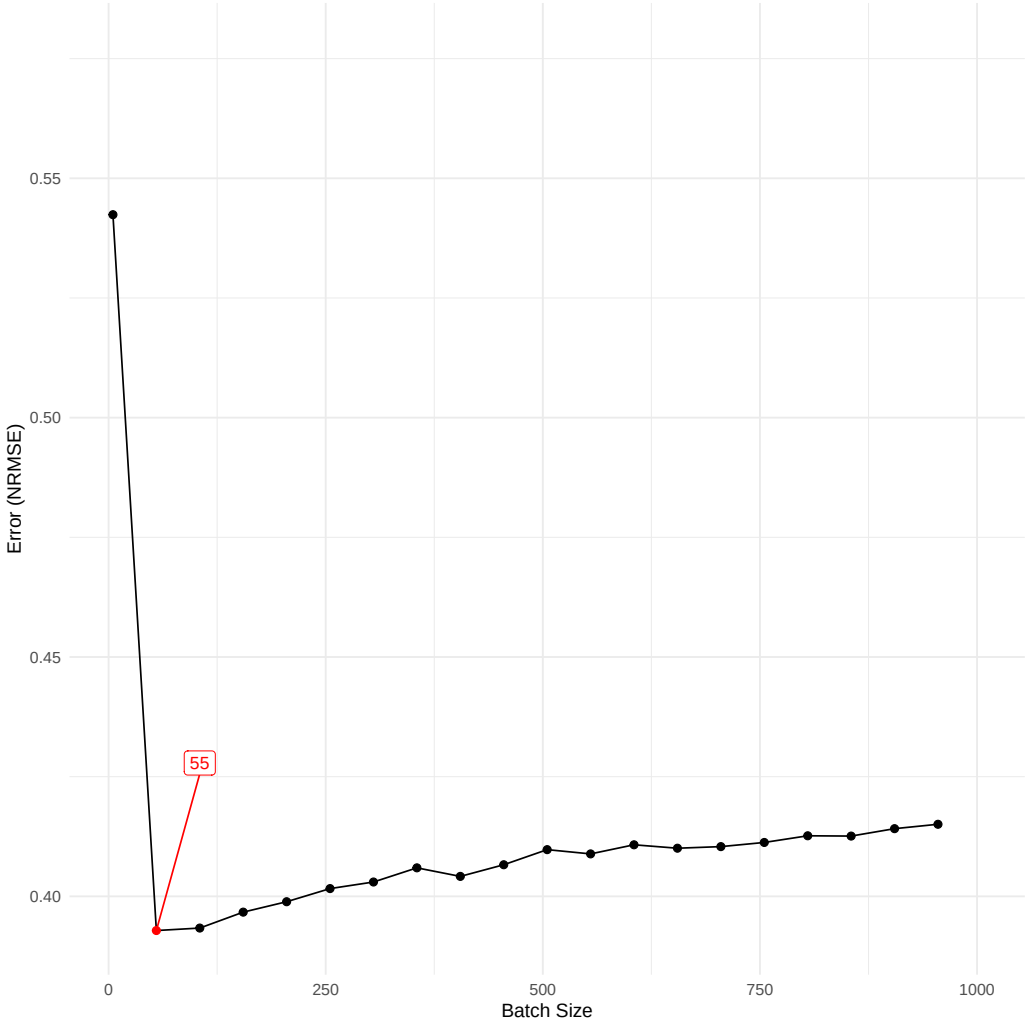

Figure 6: Grid Search Results on Simulated Data to Find Optimal Batch Size

6 A Brief Tour of the `hdImpute` Software

This section offers a brief overview of the `hdImpute` R package, which was recently released. The package is openly developed on Github under the MIT public license. Full details of the software, including documentation, unit testing, an issue tracker for bug reports and feature enhancements, as well as a contributor code of conduct are included at the Github repository, `hdImpute`. The following demonstration of the package is adapted from the package vignette, also available at the Github repository.

The package was built with non-programmers in mind in an effort to ease implementation of the hdImpute process introduced in this paper, as well as minimize the barrier to access and encourage flexible usage of the method. As such, there are two approaches to use the method: first, in individuals stages, and second, from a single function call with minimal required arguments. The overview covers each of these in turn.

6.1 Individual Stages

To support building out the `hdImpute` process in stages, there are three core functions to use:

1. `feature_cor()`: Builds the correlation matrix based on the full data matrix as input. Returns a correlation matrix.

2. `flatten_mat()`: Flattens the correlation matrix and ranks the features with the correlation matrix as input. Returns a vector of ranked features based on absolute correlation magnitude.

3. `impute_batches()`: Builds batches of size `batch` based on the ranking of features returned from `flatten_mat()` and the original data matrix as input. Returns a completed data set. Users may specify the batch size by passing a non-negative integer to the `batch` argument. Also users may specify other preferences for fitting the random forests such as number of trees (`n_trees`), number of neighbors to consider in predictive mean matching (`pmm_k`), and so on.

6.2 Single Call

To avoid the individual steps in the previous approach, users may simply pass the raw data object and specify the batch size by passing a non-negative integer to the argument (hyperparameter) `batch` in the function `hdImpute()`. Users have additional optional arguments available including, for example, the number of trees used to fit the random forests (`n_trees`). The returned object is a completed (imputed) data set.

As noted, the package is under active development with additional features in process. Wide collaboration is encouraged in any form, from submitting an issue with a bug report or requesting a new feature, to suggesting changes directly via pull request.

Ultimately, the goal of the `hdImpute` package is to maximize usability of the `hdImpute` process for high dimensional imputation for all researchers and scholars engaged in this space, without worrying about understanding complex computer code or architectures. For those interested in greater complexity, though, the package's functions are flexible enough to tie into existing research pipelines, or to customize to meet unique challenges. Even still, as the software is being actively developed, the current menu of options is limited to an efficient fitting of `hdImpute` only. Package functionality will be iteratively expanded.

7 Concluding Remarks

Perhaps the most valuable feature of hdImpute is the ability to tune the batch size. That is, there were several cases in the real data experiments where specific batch sizes resulted in very poor imputation. Yet, in those cases, tuning the batch size in either direction led to significantly better imputation, resulting in a version of hdImpute being the most accurate model in 2 of 3 conditions (30% and 99%), and a close second in the remaining condition (80%). In addition to the remarkable error rates by comparison, the speed of hdImpute in all versions and across all missingness conditions far surpassed its closest competitor (`missRanger`, again excluding `softImpute` given the error > 1.0 in all conditions). Results in the simulations focusing only on hdImpute corroborated the findings from the real data experiments.

Taken together, hdImpute offers a range of benefits over many existing algorithms for high dimensional imputation (native handling of mixed-type data, nonparametric imputation, etc.), *in addition to* offering a significant reduction in runtime compared to the closest competitor. As the dimensionality of the data space grows even larger, these benefits and their value to the research pipeline becoming increasingly valuable. That is, hdImpute offers a notably sped up and usable approach to imputation, while not only avoiding a loss in accuracy in most cases, but rather a *gain* in accuracy more often than the alternative as demonstrated throughout.

The hdImpute algorithm offers an efficient approach to high dimensional imputation by tackling a series of more manageable, smaller-sized problems, and then aggregating imputed batches to return a full, imputed dataset. Indeed, it turns out we can learn a great deal from bite-sized chunks of a full data space when considering individual correlated batches of input features at a time. Such a process also offers better use of computational resources for such an expensive task (imputation in a high dimensional setting). Of note, the benefit of hdImpute is impossible without the extant `missForest` (Stekhoven and Bühlmann, 2012) and `missRanger` (Mayer, 2019) algorithms. The process of imputation via chained random forests remains the same. Rather, the advance reported in this paper is not a new method in its own rite, but rather an extension of an existing approach to imputation through a more efficient, flexible approach to tackling the complexities of high dimensional data.

Future work can readily build on these findings. For example, others might consider alternative approaches to ranking the flattened matrix that is used to construct batches. Currently the approach is based on correlations. Yet, batches might be considered using, for example, a spatial metric or some domain-relevant weighting factor. Further, future work might include a different underlying imputation engine instead of chained random forests. For example, a deep learning approach such as the MIDAS algorithm could be used to impute plausible missing values (Lall and Robinson, 2021). Such an extension, building on the batch process introduced in this paper, would provide a useful step in refining high dimensional imputation strategies, both in terms of accuracy and runtime.

References

Bach, Francis. 2017. "Breaking the curse of dimensionality with convex neural networks." *The Journal of Machine Learning Research* 18(1):629–681.

Bennett, James, Stan Lanning et al. 2007. The netflix prize. In *Proceedings of KDD cup and workshop*. Vol. 2007 New York, NY, USA. p. 35.

Berisha, Visar, Chelsea Krantsevich, P Richard Hahn, Shira Hahn, Gautam Dasarathy, Pavan Turaga and Julie Liss. 2021. "Digital medicine and the curse of dimensionality." *NPJ digital medicine* 4(1):1–8.

Bessa, Miguel A, R Bostanabad, Zeliang Liu, A Hu, Daniel W Apley, C Brinson, Wei Chen and Wing Kam Liu. 2017. "A framework for data-driven analysis of materials under uncertainty: Countering the curse of dimensionality." *Computer Methods in Applied Mechanics and Engineering* 320:633–667.

Bollier, David, Charles M Firestone et al. 2010. *The promise and peril of big data*. Aspen Institute, Communications and Society Program Washington, DC.

Chattopadhyay, Amrita and Tzu-Pin Lu. 2019. "Gene-gene interaction: the curse of dimensionality." *Annals of translational medicine* 7(24).

Daum, Fred and Jim Huang. 2003. Curse of dimensionality and particle filters. In *2003 IEEE Aerospace Conference Proceedings (Cat. No. 03TH8652)*. Vol. 4 IEEE pp. 4_1979–4_1993.

De Marchi, Scott. 2005. *Computational and mathematical modeling in the social sciences*. Cambridge University Press.

Goldfeld, Keith and Jacob Wujciak-Jens. 2020. "simstudy: Illuminating research methods through data generation." *Journal of Open Source Software* 5(54):2763.
URL: *https://doi.org/10.21105/joss.02763*

Han, Jiequn, Arnulf Jentzen and E Weinan. 2018. "Solving high-dimensional partial differential equations using deep learning." *Proceedings of the National Academy of Sciences* 115(34):8505–8510.

Hastie, Trevor, Rahul Mazumder and Maintainer Trevor Hastie. 2013. "Package 'softImpute'.".

Hong, Shangzhi and Henry S Lynn. 2020. "Accuracy of random-forest-based imputation of missing data in the presence of non-normality, non-linearity, and interaction." *BMC medical research methodology* 20(1):1–12.

Lall, Ranjit and Thomas Robinson. 2021. "The MIDAS touch: accurate and scalable missing-data imputation with deep learning." *Political Analysis* pp. 1–18.

Lavanya, K, LSS Reddy and B Eswara Reddy. 2019. A study of high-dimensional data imputation using additive LASSO regression model. In *Computational Intelligence in Data Mining*. Springer pp. 19–30.

Little, Roderick JA and Donald B Rubin. 2019. *Statistical analysis with missing data*. Vol. 793 John Wiley & Sons.

Mayer, Michael. 2019. "Package 'missRanger'." *R Package* .

Oba, Shigeyuki, Masa-aki Sato, Ichiro Takemasa, Morito Monden, Ken-ichi Matsubara and Shin Ishii. 2003. "A Bayesian missing value estimation method for gene expression profile data." *Bioinformatics* 19(16):2088–2096.

Rhys, Hefin. 2020. *Machine Learning with R, the tidyverse, and mlr*. Simon and Schuster.

Salas, Jorge and Víctor Yepes. 2019. "VisualUVAM: A decision support system addressing the curse of dimensionality for the multi-scale assessment of urban vulnerability in Spain." *Sustainability* 11(8):2191.

Shah, Jasmit S, Shesh N Rai, Andrew P DeFilippis, Bradford G Hill, Aruni Bhatnagar and Guy N Brock. 2017. "Distribution based nearest neighbor imputation for truncated high dimensional data with applications to pre-clinical and clinical metabolomics studies." *BMC bioinformatics* 18(1):1–13.

Stekhoven, Daniel J and Peter Bühlmann. 2012. "MissForest—non-parametric missing value imputation for mixed-type data." *Bioinformatics* 28(1):112–118.

Van Buuren, Stef. 2018. *Flexible imputation of missing data*. CRC press.

Van Buuren, Stef and Karin Oudshoorn. 1999. *Flexible multivariate imputation by MICE*. Leiden: TNO.

Verleysen, Michel and Damien François. 2005. The curse of dimensionality in data mining and time series prediction. In *International work-conference on artificial neural networks*. Springer pp. 758–770.

Waggoner, Philip D. 2021. *Modern dimension reduction*. Cambridge University Press.

Wilson, Samuel. 2022. "Package 'miceforest' v5.6.2." *Python package. PyPi* .
URL: *https://pypi.org/project/miceforest/*

Wright, Marvin N and Andreas Ziegler. 2015. "ranger: A fast implementation of random forests for high dimensional data in C++ and R." *arXiv preprint arXiv:1508.04409* .

Zhao, Yize and Qi Long. 2016. "Multiple imputation in the presence of high-dimensional data." *Statistical Methods in Medical Research* 25(5):2021–2035.