Open in app ↗ Sign up Sign In

Medium Search

Member-only story

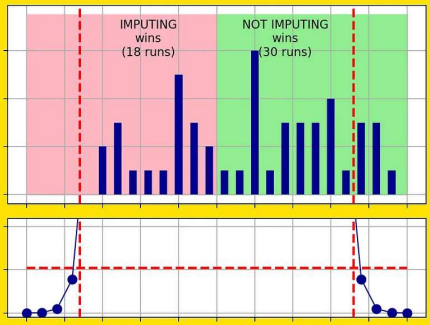
Your Dataset Has Missing Values? Do Nothing!

Models can handle missing values out-of-the-box more effectively than imputation methods. An empirical proof

Samuele Mazzanti · Follow

Published in Towards Data Science
10 min read · Oct 9

Listen Share



[Image by Author]

Missing values are very common in real datasets. Over time, many methods have been proposed to deal with this issue. Usually, they consist either in removing data that contain missing values or in imputing them with some techniques.

In this article, I will test a third alternative:

> Doing nothing.

Indeed, the best models for tabular datasets (namely, XGBoost, LightGBM, and CatBoost) can natively handle missing values. So, the question I will try to answer is:

> Can these models handle missing values effectively, or would we obtain a better result with a preliminary imputation?

Who said we should care about nulls?

There seems to be a **widespread belief that we must do *something* about missing values**. For instance, I asked ChatGPT what should I do if my dataset contain missing values, and it suggested 10 different ways to get rid of them (you can read the full answer here).

But where does this belief come from?

Usually, these kinds of opinions originate from historical models, particularly from linear regression. This is also the case. Let's see why.

Suppose that we have this dataset:

	a	b	c
0	10	100	80
1	NaN	70	30
2	30	30	NaN
3	40	NaN	-60
4	50	10	-90

A dataset with missing values. [Image by Author]

If we tried to train a linear regression on these features, we would get an error. In fact, to be able to make predictions, linear regression needs to multiply each feature by a numeric coefficient. If one or more features are missing, it's impossible to make a prediction for that row.

This is why many imputation methods have been proposed. For instance, one of the simplest possibilities is to replace the nulls with the feature's mean.

	a	b	c
0	10	100	80
1	32.5	70	30
2	30	30	-10
3	40	52.5	-60
4	50	10	-90

Imputation with the feature's mean. [Image by Author]

Another, more sophisticated, approach is to use the relationships between the variables to predict the most likely value to fill that specific entry. This entails training one predictive model for each feature (using the other features as predictors).

	a	b	c
0	10	100	80
1	25.7	70	30
2	30	30	12.1
3	40	43.2	-60
4	50	10	-90

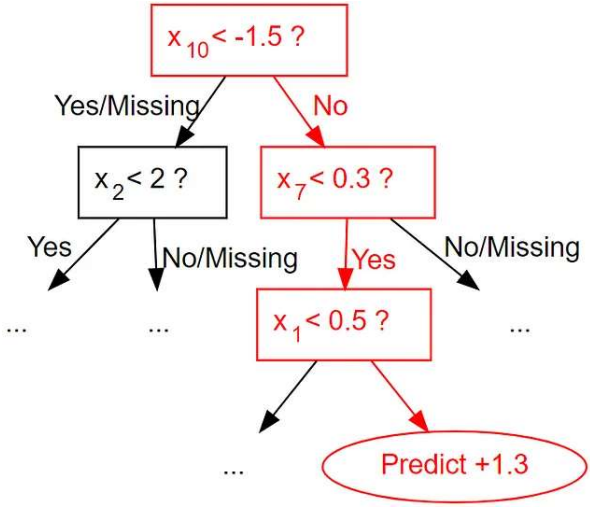
Iterative Imputation: each feature is estimated from all the others. [Image by Author]

However, not all models are like linear regression.

Indeed, as luck would have it, the best-performing models for tabular tasks (namely, tree-based models, such as XGBoost, LightGBM, and CatBoost) can handle missing values natively.

How is that possible?

Because in tree-like structures, missing values can be treated just like other values, i.e. the model can assign them to a branch of the tree. For example, this image comes from XGBoost documentation:



How tree-based models handle missing values. [Image from XGBoost documentation]

As you can see, for each split XGBoost chooses a default branch on which missing values (if any) are routed.

So, are we saying that just because these models can handle missing values, we should avoid imputation? I never said that.

As data scientists, we usually want to know well **in practice**. So, the focus of the next paragraphs will be to compare the model performance with and without imputation and see which approach performs better.

Experimenting with real datasets

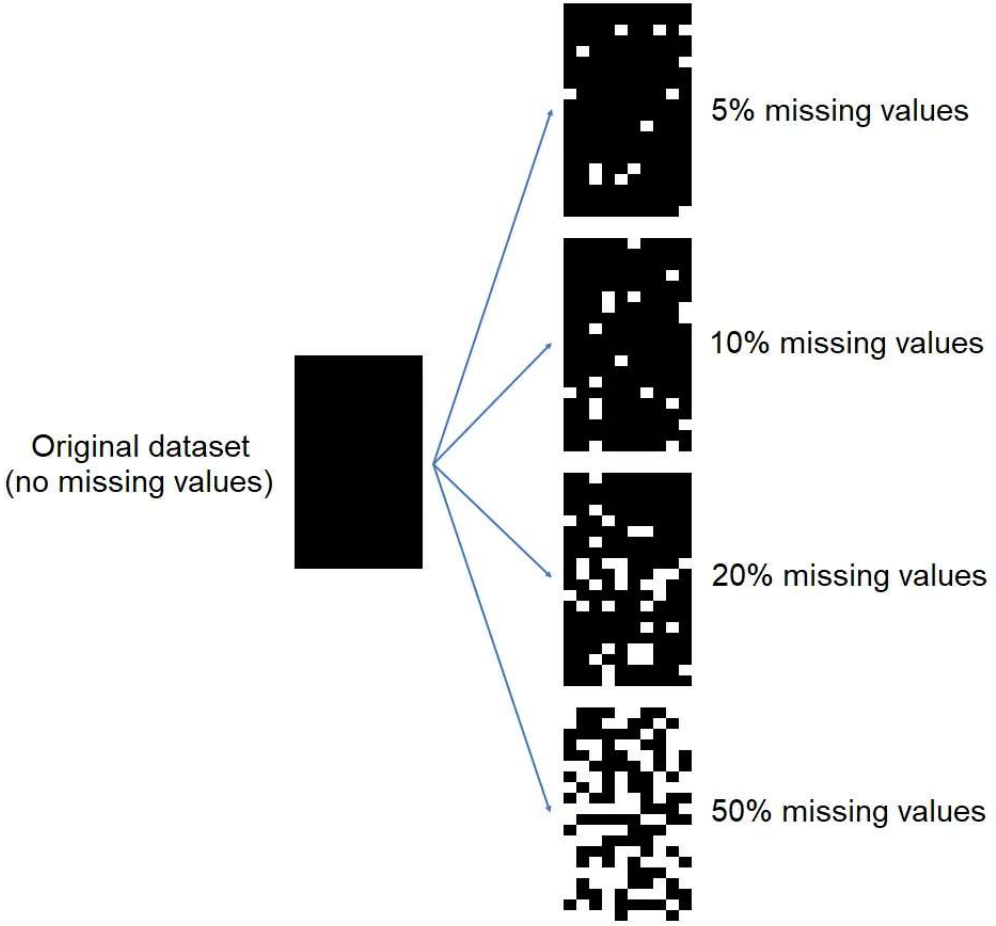
Our objective is to compare two approaches:

1. Train and test the model on the dataset **with missing values**.
2. Train and test the model on the dataset with no missing values (after they have been **imputed through some imputation method**).

I will use 14 real datasets that are available in Pycaret (a Python library under MIT license).

These datasets do not contain missing values, so we will need to fabricate them. I will call this procedure null-sowing because it involves disseminating null values on the original dataset (i.e. "canceling" some of the original values).

How many values are we going to cancel? To ensure that the experiment is representative, we will try with different percentages of nulls: 5%, 10%, 20% and 50%.



Null-sowing with different percentages. [Image by Author]

I use two different strategies to create null values:

- **Random**: a random value between 0 and 1 is attributed to each entry of the dataset and, if smaller than the threshold, the entry is canceled.
- **Non-random**: for each feature, I sort the values either in ascending or descending order, and attribute a proportional probability of being canceled based on the position of the value in the sorted sequence.

Moreover, for each combination, I will try 25 different random train/test splits. The purpose of doing this is to ensure that the results we observe are consistent over many repetitions and not simply due to chance.

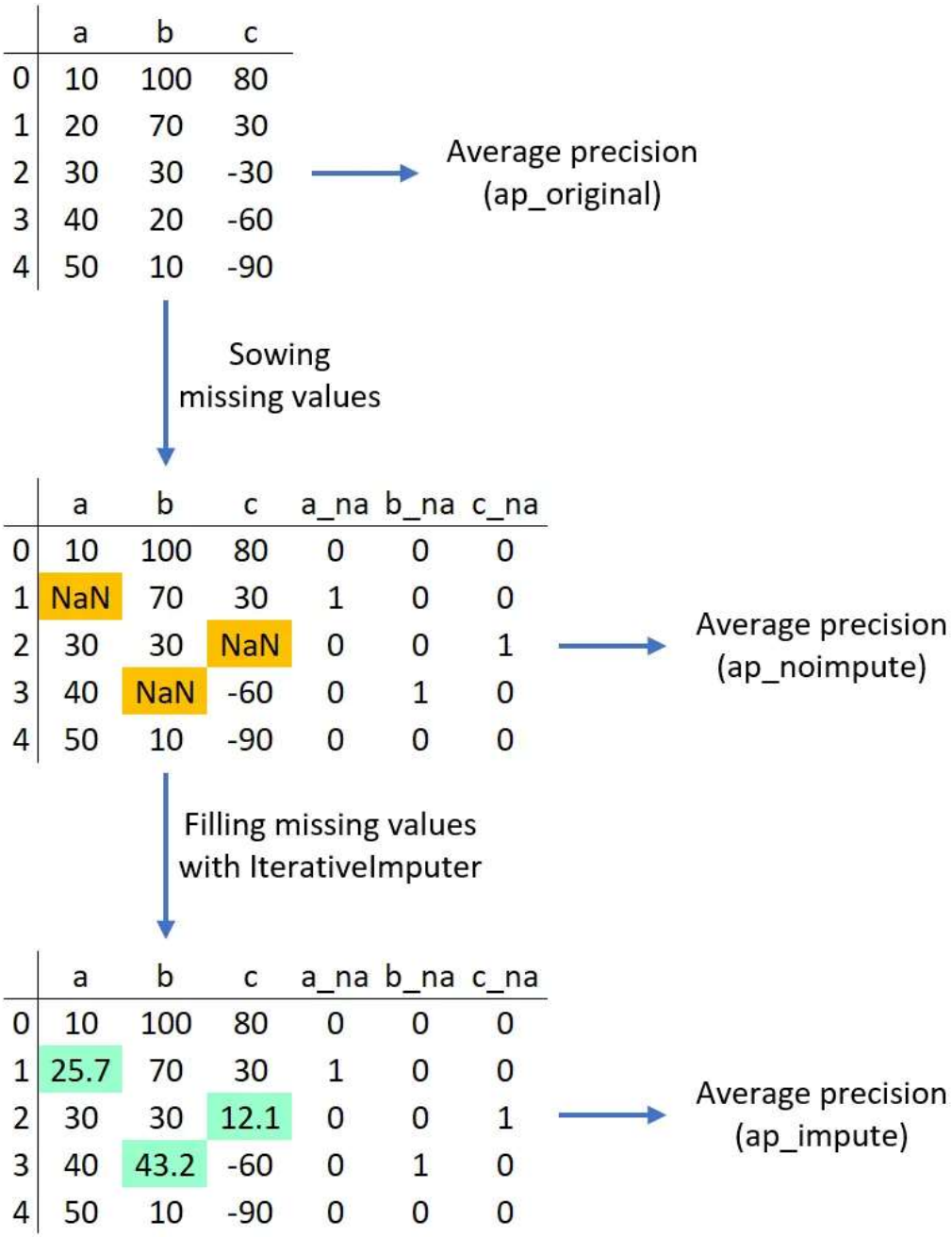
To sum up, these are all the combinations that I will try.

Combinations that form our experiment. [Image by Author]

For each of these 2,800 combinations, I will compute the average precision (on the test set) of LightGBM with three different approaches:

1. Using the original dataset (with **no missing values**). I will call this average precision `ap_original`.
2. Using the dataset **containing the missing values I have sown.** I will call this average precision `ap_noimpute`.
3. Using the dataset where the **missing values have been filled** using Scikit-Learn's IterativeImputer. I will call this average precision `ap_impute`.

Let's clarify the difference between these three metrics with the aid of a figure:



3 versions of the dataset. A different LightGBM has been trained on each version. [Image by Author]

Results

I tried all the 2,800 combinations listed above and tracked `ap_original`, `ap_noimpute`, and `ap_impute` for each of them. I stored the results in a table like this:

dataset	null_sowing	null_pct	ap_original	ap_noimpute	ap_impute
bank	random	0.05	[0.32559236856708396, 0.35085791563155616, 0.3...	[0.31181540890920756, 0.33432555774817807, 0.3...	[0.3078448513757593, 0.32717735654577806, 0.33...
bank	random	0.10	[0.33431684339419165, 0.3879766612965307, 0.38...	[0.3170154224422668, 0.33589370405082153, 0.34...	[0.3233840467577366, 0.34199048254862546, 0.34...
bank	random	0.20	[0.3498213392062083, 0.3797030963901035, 0.355...	[0.32301874229652255, 0.3234312707177882, 0.32...	[0.3104301196642582, 0.31751866017527053, 0.31...
bank	random	0.50	[0.3364635704265141, 0.35434117142164856, 0.37...	[0.19997544115134763, 0.22174176523398276, 0.2...	[0.18150614747802896, 0.2042229068854265, 0.23...
blood	random	0.05	[0.4829764065335753, 0.3247368421052632, 0.273...	[0.4056140350877193, 0.3058569500674764, 0.258...	[0.41077040427154843, 0.30526315789473685, 0.2...
...	...	...	...	...	...

Results of the experiment. [Image by Author]

This table has 112 rows (i.e. 14 datasets x 2 null-sowing strategies x 4 null frequencies). The first three columns indicate which dataset, null-sowing strategy, and null frequency identify that row. Then, there are three columns that store the average precisions reported by that approach on the 25 bootstrap iterations.

To make it even clearer, let's take the fourth column of the first row. This is an array of 25 elements: each of them is the average precision that LightGBM realized — for that particular train/test split (i.e. bootstrap iteration) — on the original dataset (that's

why it's called `ap_original`) for dataset "bank" with 5% of missing values randomly sown.

Of course, we are not interested in each single bootstrap iteration, so we need to aggregate the arrays in some way. Since we want to compare the three approaches with each other, the most simple aggregation is counting the number of times that one approach is superior to another (meaning it has a higher average precision).

Thus, for each row of the table, I calculated the percentage of times that:

- `ap_original` > `ap_noimpute`, which means that the model on the original dataset is better than the model on the dataset containing missing values.
- `ap_noimpute` > `ap_impute`, which means that the model on the dataset containing missing values is better than the model on the dataset filled with IterativeImputer.

This is the result:

dataset	null_sowing	null_pct	original_>_noimpute	noimpute_>_impute
bank	random	0.05	1.00	0.60
bank	random	0.10	1.00	0.48
bank	random	0.20	1.00	0.72
bank	random	0.50	1.00	0.84
blood	random	0.05	0.64	0.64
...	...	...	...	...

Results of the experiment. [Image by Author]

For example, if we take the last column of the first row, it means that `ap_noimpute` is greater than `ap_impute` in 60% out of the 25 iterations.

Let's look at the column called `original_>_noimpute`. Apparently, it's 100% most of the time. And this makes sense: it's reasonable that the model trained on the original dataset does better than the model trained on the dataset in which we have canceled some entries.

But the most important information for us is in the last column. In fact:

- when `noimpute_>_impute` is greater than 50%, it means that the model trained on the null dataset outperformed most of the times the model trained on the imputed dataset.
- when `noimpute_>_impute` is less than 50%, the opposite.

So we could be tempted to simply take the mean of this column and decide which approach works better based on whether the global mean is above or below 50%.

But if we do that, we could be misled by the effect of chance. Indeed, if a value is close to 50%, like 48% or 52%, this could be easily due to randomness. To account for that, we need to frame this as a statistical test.

Setting up a statistical test

To avoid being fooled by chance, I will take a couple of precautions.

First, I will keep only the cases in which the average precision on the original dataset is greater than the average precision on the missing dataset more than 90% of the time (in other words, I will keep only the rows where `original_>_noimpute` > .90).

I will do this because I want to keep only the cases where missing values have a clear negative impact on the performance of the model. After doing that, out of the initial 112 rows of the table, only 48 remain.

Secondly, I will need to compute a "significance threshold" that helps us to understand whether a value is significant or not.

We already said that when a value is very close to 50%, like 48% or 52%, then it's probably just due to chance. But how close is "very close"? To answer this, we need some statistics.

The hypothesis we want to test (a.k.a. the null hypothesis) is that **imputing or not imputing makes no difference**. This is like saying that the probability of `ap_impute` being greater than `ap_noimpute` (or vice versa) is 50%. Since we have 25 independent iterations, we can compute the probability of obtaining a specific result through the binomial distribution.

For example, let's say that, out of 25 iterations, `ap_impute` has been better than `ap_noimpute` in 10 iterations. How compatible is this result with our hypothesis?

```
from scipy.stats import binom

binom.cdf(k=10, n=25, p=.50) * 2

# result: 0.42435622
```

So, the probability of obtaining a result as extreme as 10, given our hypothesis, is 42%.

Note that I multiplied the cumulative distribution function of the binomial distribution by 2 because we are interested in the two-tailed p-value. We can double-check this by looking at the p-value associated with any possible outcome:

0	0.00000006	25	0.00000006
1	0.00000155	24	0.00000155
2	0.00001943	23	0.00001943
3	0.00015652	22	0.00015652
4	0.00091052	21	0.00091052
5	0.00407732	20	0.00407732
6	0.01463330	19	0.01463330
7	0.04328525	18	0.04328525
8	0.10775214	17	0.10775214
9	0.22952294	16	0.22952294
10	0.42435622	15	0.42435622
11	0.69003797	14	0.69003797
12	1.00000000	13	1.00000000

P-values associated with any possible outcome (out of 25 iterations). [Image by Author]

The p-value associated with 12 and 13 is exactly 100%. And this makes sense: the probability of obtaining a result at least as extreme as 12 or 13 is necessarily 100% since they are the least extreme results, given our hypothesis.

So, what is our significance level? As per convention, I will take a significance level of 1%. However, we must consider that we

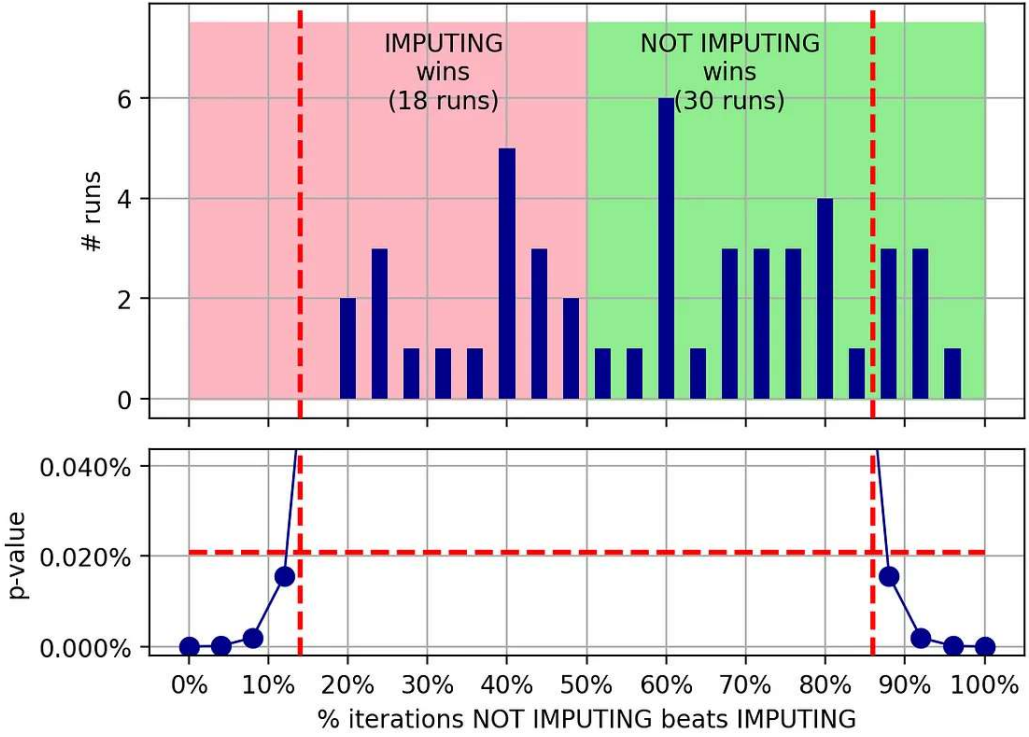
have many runs, not just one, so we must adjust our significance level accordingly. Since we have 48 runs, I will use the Bonferroni correction and simply divide 1% by 48, obtaining a final significance level of 0.0002.

Comparing this number with the p-values listed above, this means we will consider a run significant only if it has less than 3 or more than 22 (both inclusive) successes.

Now that we have taken measures to avoid being fooled by chance, we are ready to look at the results.

For simplicity, let's take as metric the percentage of times that `ap_noimpute` is greater than `ap_impute`. For example, if the model based on the dataset containing missing values has a better average precision than the model on the imputed dataset for 10 out of the 25 iterations, this metric will be 40%.

Since we have 48 runs, we will have 48 values. So let's create a bar plot to see all of them.



Results of the experiment. [Image by Author]

The red dashed lines identify the significance threshold: any value that is smaller than 12% (inclusive) or greater than 88% (inclusive) is significant (12% and 88% correspond respectively to 3/25 and 22/25).

From the bar plot, we can see that **when we don't impute the missing values we obtain a better result in 30 out of 48 runs** (63%). In 7 of these 30 cases, the result is so extreme that it's also statistically significant according to a 1% p-value with Bonferroni adjustment. Instead, in the 18 cases in which imputing wins, the result is never significantly different from pure chance.

Based on these results, we can say that **either the difference between imputing and not imputing is not significant or it is significant in favor of not imputing.**

In short, there is no reason to impute missing values.

Conclusions

In this article, we empirically proved that the difference between imputing and not imputing is either not significant or significant in favor of not imputing. If you also consider that not imputing missing values will keep your pipeline cleaner and faster, leaving nulls in your dataset should be the standard, when you can do that.

Of course, this is not always possible. Some models that cannot handle missing values: take for instance Linear Regression or K-Means.

However, the good news is that when you use the most common models for tabular tasks (i.e. tree-based models), not imputing the missing values and just letting the model handle them is the most effective and efficient approach.

. . .

You can reproduce all the code used for this article with this notebook.