

How to Handle Missing Values?

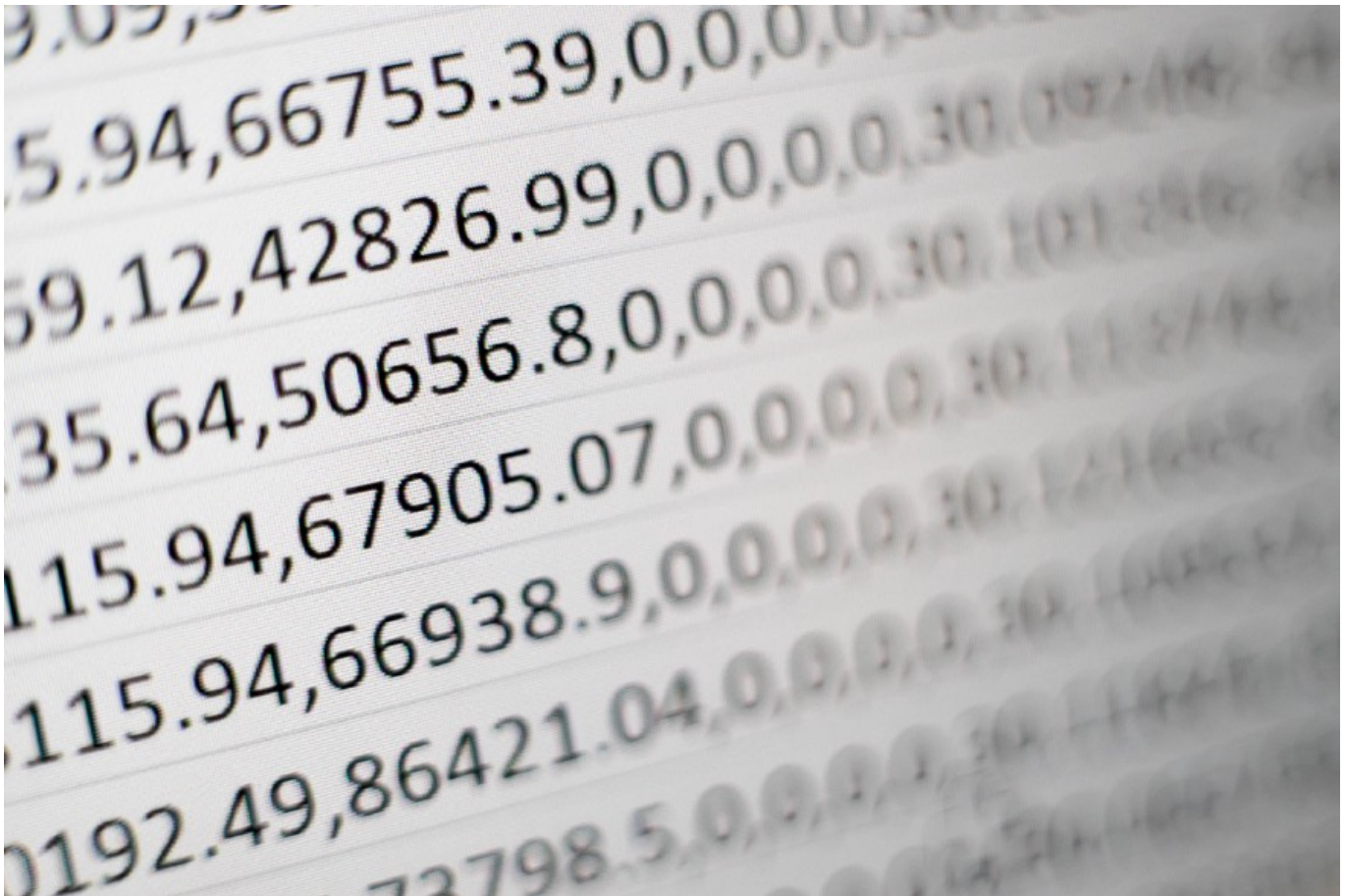


Aryan Chaudhary

Follow

May 29 · 8 min read

A Hands-On Kickstarter Guide.



In previous articles, I promised to discuss how to tackle missing values!

In this article, we will cover different strategies to counter missing values. You will learn to compare the effectiveness of these approaches on any given or most of the dataset.

Introduction

According to Wikipedia, “In statistics, **missing data**, or **missing values**, occur when no data value is stored for the variable in an observation. Missing data are a common occurrence and can have a significant effect on the conclusions that can be drawn from the data.”

For various reasons, many real-world datasets contain missing values, often encoded as **blanks**, **NaNs**, or other **placeholders**.

There are many ways data can incorporate missing values. For example in housing data:

- A 1 bedroom house wouldn't include an answer for *How large is the second bedroom*.
- The house owner may not want to share their earnings.

Python libraries represent missing numbers as **nan** which is short for “**not a number**”. The datasets containing **nan** values, however, are not consistent with **Sklearn** estimators which assume that all values in an array are numerical and that all have and hold meaning so that mathematical operations may be applied to them.

A basic strategy to use datasets containing missing values is to discard entire rows and/or columns consisting of missing values. However, this comes at the cost of losing valuable data which may be important (even though incomplete). A better strategy is to fill in the missing values is to estimate them from the part of the data which is not missing.

You can figure out which cells have missing values and what percentage of values are missing in each column, with the command:

```
def missingcheck(data):  
    total = data.isnull().sum().sort_values(ascending=False)  
    percent_1 = data.isnull().sum()/data.isnull().count()*100  
    percent_2 = (np.round(percent_1, 1)).sort_values(ascending=False)  
    missing_data = pd.concat([total, percent_2], axis=1, keys=['Total',  
    '%'])  
    return missing_data
```

```
1 #just pass the data to the function  
2 missingcheck(train_data)
```

	Total	%
PoolQC	1453	99.5
MiscFeature	1406	96.3
Alley	1369	93.8
Fence	1179	80.8
FireplaceQu	690	47.3
...
CentralAir	0	0.0
SaleCondition	0	0.0
Heating	0	0.0
TotalBsmtSF	0	0.0
Id	0	0.0

All this code is present in Github repository [MLin10Minutes!](#)

Also, segregate Numerical attributes and Categorical attributes

Because it will affect the strategy we will follow for missing value treatment! You can treat missing values separately and later join them together! But to keep this article concise and understandable I will continue only with numerical values.

Function to Evaluate the Accuracy of Each Approach

We define a function `score_dataset()` to compare different approaches to dealing with missing values. This function reports the mean absolute error (MAE) from a random forest model.

```
from sklearn.ensemble import RandomForestRegressor
from sklearn.metrics import mean_absolute_error

# Function for comparing different approaches
def score_dataset(X_train, X_valid, y_train, y_valid):
    RFr = RandomForestRegressor(n_estimators=150, random_state=42)
    RFr.fit(X_train, y_train)
    preds = RFr.predict(X_valid)
    return mean_absolute_error(y_valid, preds)
```

1) A Naive Approach: Drop Columns with Missing Values

The most basic option is to remove the columns with missing values.

Bed	Bath		Bath
1.0	1.0		1.0
2.0	1.0		1.0
3.0	2.0		2.0
NaN	2.0		2.0

Source: Kaggle

Unless most values in the dropped columns are missing, the model loses access to a lot of (maybe useful!) information with this approach. As an extreme example, consider a dataset with 20,000 rows, where one important column is missing only a single entry. This approach would drop the complete column!

```
#get names of columns with missing values
cols_with_missing = [col for col in X_train.columns
                      if X_train[col].isnull().any()]
# drop columns in training and validation data
reduced_X_train = X_train.drop(cols_with_missing, axis=1)
reduced_X_valid = X_valid.drop(cols_with_missing, axis=1)
```

```
1 print("MAE (Drop columns with missing values):")
2 print(score_dataset(reduced_X_train, reduced_X_valid, y_train, y_valid))
```

```
MAE (Drop columns with missing values):
17952.591404109586
```

2) A Superior Alternative: Univariate Imputation

Imputation fills in the missing values with some number. For instance, we can fill in the **mean** value along each **column**.

Bed	Bath		Bed	Bath
1.0	1.0		1.0	1.0
2.0	1.0		2.0	1.0
3.0	2.0		3.0	2.0
NaN	2.0		2.0	2.0

Source: Kaggle

The imputed value won't be exactly right in most cases, but it usually leads to better models than you would get from removing the column entirely.

One type of imputation algorithm given in sklearn is **univariate**,

“which imputes values in the i -th feature dimension using only non-missing values in that feature dimension” (e.g. `impute.SimpleImputer`).

The **SimpleImputer** from the sklearn package provides basic strategies for imputing missing values. Missing values can be imputed with a provided **constant value**, or using the **statistics** (mean, median, or most frequent) of each column in which the missing values are located.

The following snippet demonstrates how to replace missing values:

```
from sklearn.impute import SimpleImputer
# imputation
simple_imputer = SimpleImputer()

imputed_X_train= pd.DataFrame(simple_imputer.fit_transform(X_train))

imputed_X_valid = pd.DataFrame(simple_imputer.transform(X_valid))

# imputation removed column names; put them back
imputed_X_train.columns = X_train.columns
imputed_X_valid.columns = X_valid.columns
```

```
1 print("MAE (Imputation):")
2 print(score_dataset(imputed_X_train, imputed_X_valid, y_train, y_valid))
```

```
MAE (Imputation):
18250.608013698627
```

Remember, I told you that our strategy for a numerical and categorical attribute will be different. Observe yourself, if the column has categorical data like “high”, “medium”, “low” one can not use mean.

The SimpleImputer class also supports categorical data represented as string values or *pandas categoricals* when using the ‘*most_frequent*’ or ‘*constant*’ strategy.

3)Another Option: Multivariate Imputation

As defined in *Sklearn* documentation, “Multivariate imputer that estimates each feature from all the others. A strategy for imputing missing values by modelling each feature with missing values as a function of other features in a round-robin fashion.”

This more advanced approach is implemented in the **IterativeImputer** class, which models each feature with missing values as a function of other features, and uses that estimate for imputation. It does so in an iterated round-robin fashion: at each step, a feature column is designated as output y and the other feature columns are treated as inputs X . A **regressor** is fit on (X, y) for known y . Then, the regressor is used to **predict** the **missing** values of y . This is iteratively done for each feature and then is repeated for `max_iteration` imputation rounds. The results of the final imputation round are returned.

Here is the code snippet:

```
br_imputer = IterativeImputer(BayesianRidge())
imputed_X_train = pd.DataFrame(br_imputer.fit_transform(X_train))

imputed_X_valid = pd.DataFrame(br_imputer.transform(X_valid))

# imputation removed column names; put them back
imputed_X_train.columns = X_train.columns
imputed_X_valid.columns = X_valid.columns
```

```
1 print("MAE (Imputation):")
2 print(score_dataset(imputed_X_train, imputed_X_valid, y_train, y_valid))
```

```
MAE (Imputation):
17976.146438356165
```

Some of the estimators which we can use in **IterativeImputer**:

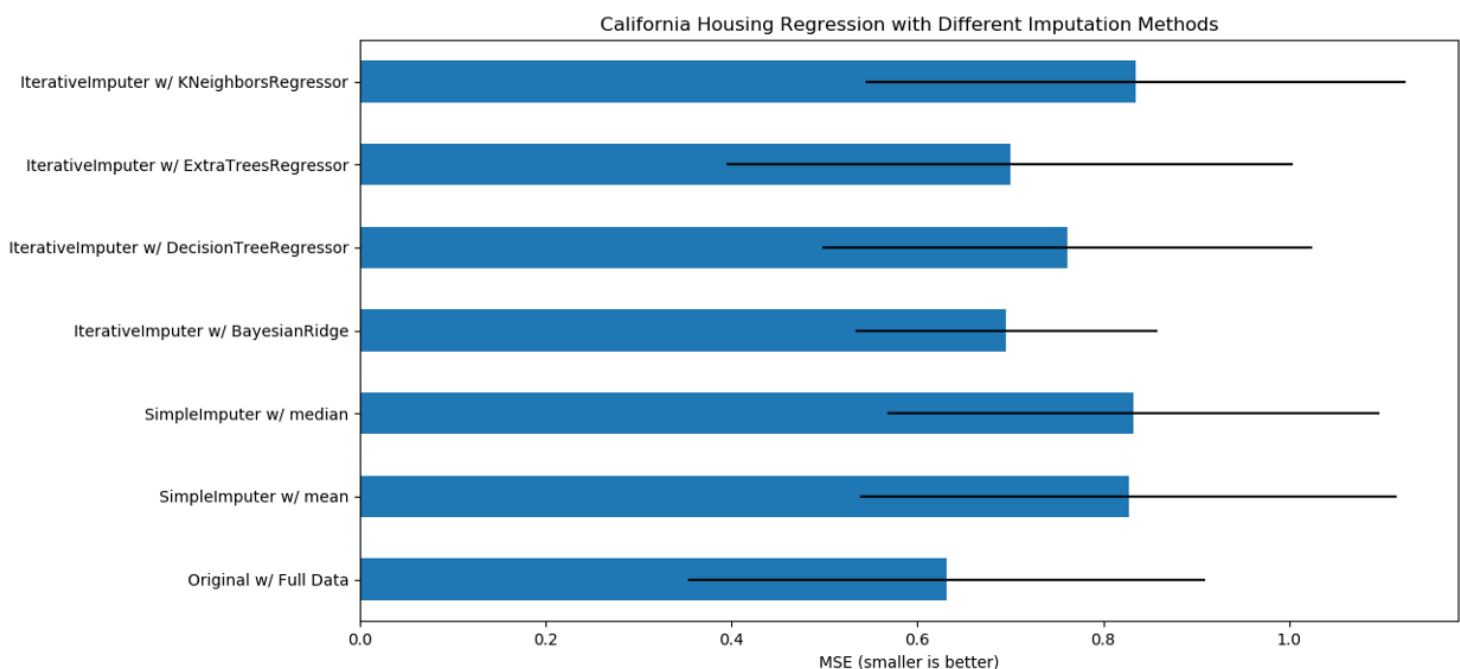
- BayesianRidge : Regularized Linear Regression
- DecisionTreeRegressor : Non-linear Regression
- ExtraTreesRegressor : Similar to `missForest` in R

- **KNeighborsRegressor** : Comparable to other KNN imputation approaches

There are many well-established imputation packages in the **R data science ecosystem**: **Amelia**, **mi**, **mice**, **missForest**, etc.

There can be a lot of variations of sequential imputation algorithms that can all be implemented with **IterativeImputer** by-passing in **different regressors** to be used for predicting missing feature values.

In the case of the popular missForest method, this regressor is a Random Forest.



Comparison of Various Imputation Strategies on California House dataset. Source: Scikit-learn

4) Nearest Neighbors Imputation

The *k nearest neighbors algorithm* can be used for imputing missing data by finding the *k closest neighbors* to the observation with missing data and then imputing them based on the non-missing values in the neighbors.

There are several possible strategies for this. One can use **1NN**, where we search the *most similar neighbor* and then use its value to replace the missing data.

Alternatively, we can use **kNN**, with *k neighbors* and calculate the **mean** of the neighbors, or **weighted mean**, where the distances to neighbors are used as weights, so the

closer the neighbour is, the more weight it has while calculating the mean. Most commonly weighted mean technique is used.

The **KNNImputer** class provides imputation for filling in missing values using the **k-Nearest Neighbors** approach. It uses a Euclidean distance metric that has support for missing values. It is known as *nan_euclidean_distance* and is used to find the nearest neighbours.

Note according to the official documentation: “*sklearn.neighbors.KNeighborsRegressor is different from KNN imputation, which learns from samples with missing values by using a distance metric that accounts for missing values, rather than imputing them.*”

Here is a code snippet :

```
knn_imputer = KNNImputer(n_neighbors=2, weights="uniform")

imputed_X_train = pd.DataFrame(knn_imputer.fit_transform(X_train))

imputed_X_valid = pd.DataFrame(knn_imputer.transform(X_valid))

# imputation removed column names; put them back
imputed_X_train.columns = X_train.columns
imputed_X_valid.columns = X_valid.columns
```


```
1 print("MAE (Imputation):")
2 print(score_dataset(imputed_X_train, imputed_X_valid, y_train, y_valid))
```

```
MAE (Imputation):
18095.66794520548
```

5) Beyond Imputation

Imputation is a well-accepted approach, and it generally works well. However, imputed values may be arranged below or above their actual values (which weren't collected in the dataset). Or missing row values may be different in some other way. In that case, your model would perform better by considering which values were originally missing.

Bed	Bath		Bed	Bath	Bed_was_missing
1.0	1.0		1.0	1.0	FALSE
2.0	1.0		2.0	1.0	FALSE



2.0	1.0	FALSE
3.0	2.0	FALSE
NaN	2.0	TRUE

In this approach, just as before, we impute the missing values. And, then, for each column with missing entries in the original dataset, we add a new column that marks the position of the imputed entries.

```
imputed_X_train_plus = X_train.copy()
imputed_X_test_plus = X_valid.copy()

cols_with_missing = (col for col in X_train.columns
                      if X_train[col].isnull().any())
for col in cols_with_missing:
    imputed_X_train_plus[col + '_was_missing'] =
    imputed_X_train_plus[col].isnull()
    imputed_X_test_plus[col + '_was_missing'] =
    imputed_X_test_plus[col].isnull()
#see what happend to the dataset
imputed_X_train_plus.head()

# And now we Imputation
my_imputer = SimpleImputer()
imputed_X_train_plus = my_imputer.fit_transform(imputed_X_train_plus)
imputed_X_test_plus = my_imputer.transform(imputed_X_test_plus)
```

```
1 print("Mean Absolute Error from Imputation while Track What Was Imputed:")
2 print(score_dataset(imputed_X_train_plus, imputed_X_test_plus, y_train, y_valid))
```

```
Mean Absolute Error from Imputation while Track What Was Imputed:
18253.31479452055
```

In some cases, this will meaningfully improve results. In other cases, it doesn't help at all.

6) May Not Be A Missing Value!

Wait, What?

Yup, it's possible that what you think is missing value may not be a missing value. It may have happened due to NA (not available) encoding confusion or we are interpreting it wrongly. For eg:

PoolArea	PoolQC	Fence	MiscFeature	MiscVal
0	NaN	NaN	NaN	0
0	NaN	NaN	NaN	0
0	NaN	NaN	NaN	0
0	NaN	NaN	NaN	0
0	NaN	NaN	NaN	0

PoolQC: data description says **NA** means “**No Pool**”. That makes sense, given the huge ratio of missing value (+99%) and the majority of houses have no Pool at all in general.

The same goes for **Fence** and **MiscFeature**, as these houses may not have any Fences or Miscellaneous Features.

What's the catch then?

Data Science people believe it is on the researcher's critical “style” how to deal with it. There is no right or wrong “generally” even if someone wants to do a savage mean/median/mode replacement or delete them. It's okay.

One can compare MAE to choose the best result. Multiple Imputations can be done and each variation result should be analyzed. For example, Out of mean, median, or mode what one should choose. In iterative imputer what estimator should one use, or even in KNNimputer what should be the value of k?

One's approach to imputing is “**do no harm**” and let your algorithm learn, which the simple imputation strategies are all about. But, again, every situation is different, so you have to try different approaches and, if you have enough data, verify them through a test set and then a validation set.

All this depends upon the *Dataset and domain of the problem*. It's merely your *intuition* that will guide you to choose the *best-suited method*!

If you enjoyed this post, go ahead and smash that clap button. Interested in more posts to come? Make sure to follow this series **MLin10Minutes**.

Feel free to drop comments to ask questions! You can also **DM** your doubts.

Acknowledgement:

- https://en.wikipedia.org/wiki/Missing_data
- <https://leandeep.com/datalab-kaggle/handling-missing-values.html>
- <https://scikit-learn.org/stable/modules/impute.html>
- <https://stats.stackexchange.com/questions/327074/k-nearest-neighbor-imputation-explanation>



Sign up for Analytics Vidhya News Bytes

By Analytics Vidhya

Latest news from Analytics Vidhya on our Hackathons and some of our best articles! [Take a look](#)

Your email

Get this newsletter

By signing up, you will create a Medium account if you don't already have one. Review our [Privacy Policy](#) for more information about our privacy practices.

Handling Missing Values

Data Imputation

Sklearn

Missing Values

Machine Learning

[About](#) [Help](#) [Legal](#)

Get the Medium app

