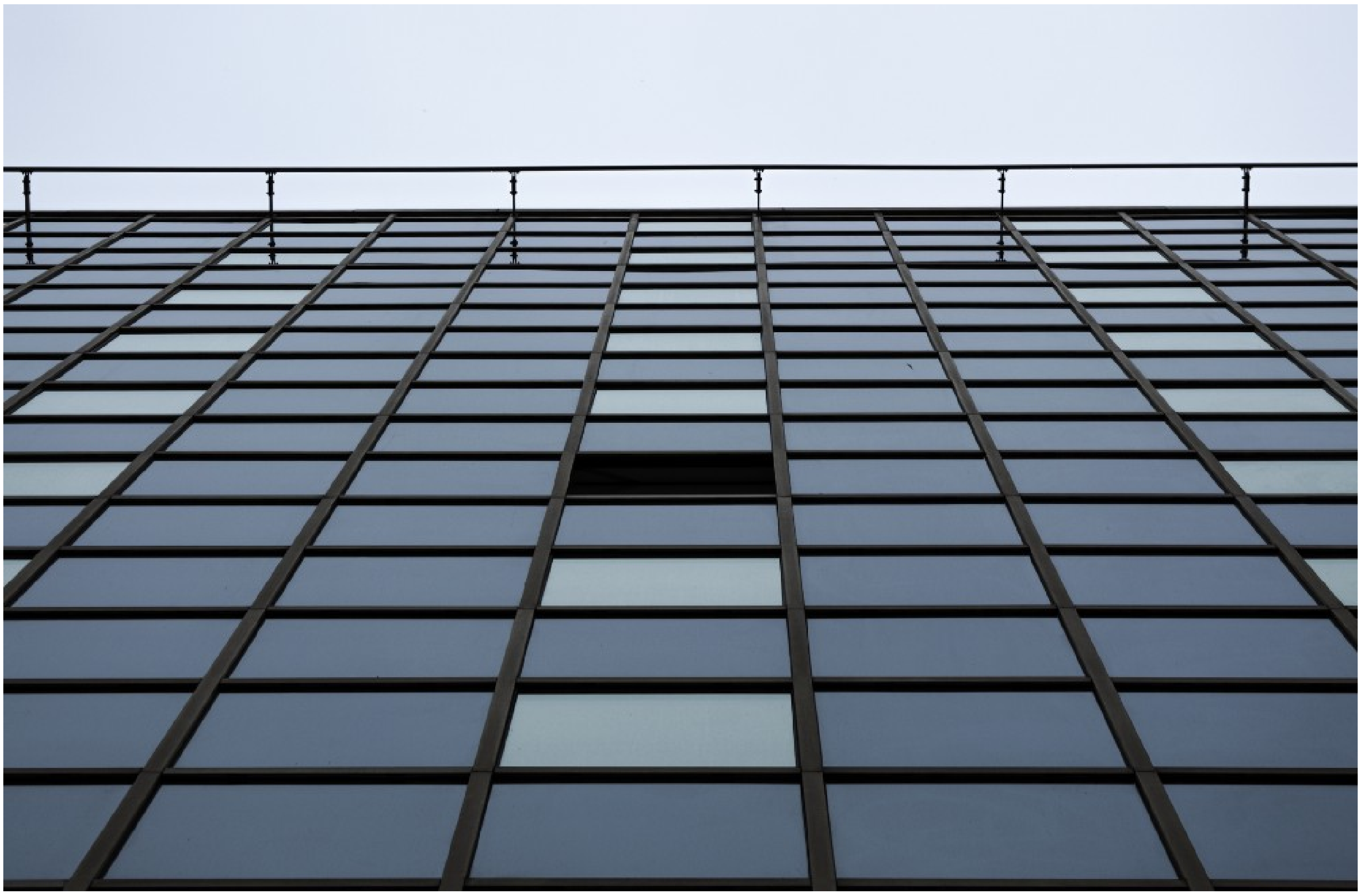


Photo by Vilmos Heim on Unsplash

6 Different Ways to Compensate for Missing Values In a Dataset (Data Imputation with examples)

Popular strategies to statistically impute missing values in a dataset.

Will Badr Follow
Jan 5 · 5 min read ★

Many real-world datasets may contain missing values for various reasons. They are often encoded as NaNs, blanks or any other placeholders. Training a model with a dataset that has a lot of missing values can drastically impact the machine learning model's quality. Some algorithms such as *scikit-learn estimators* assume that all values are numerical and have and hold meaningful value.

One way to handle this problem is to get rid of the observations that have missing data. However, you will risk losing data points with valuable information. A better strategy would be to impute the missing values. In other words, we need to infer those missing values from the existing part of the data. There are three main types of missing data:

- Missing completely at random (MCAR)
- Missing at random (MAR)
- Not missing at random (NMAR)

However, in this article, I will focus on 6 popular ways for data imputation for cross-sectional datasets (Time-series dataset is a different story).

1- Do Nothing:

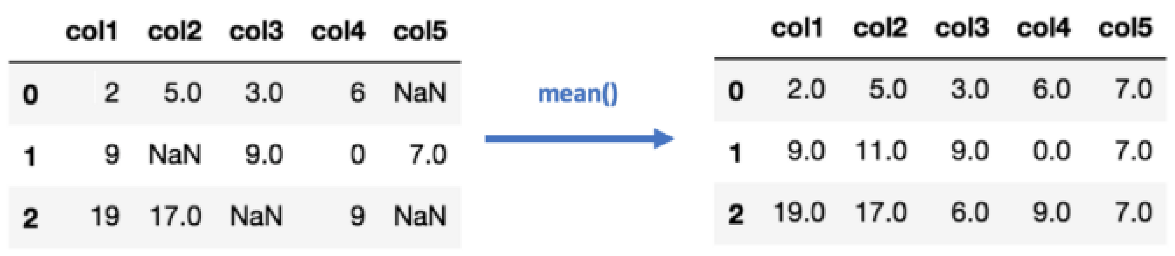
That's an easy one. You just let the algorithm handle the missing data. Some algorithms can factor in the missing values and learn the best imputation values for the missing data based on the training loss reduction (ie. XGBoost). Some others have the option to just ignore them (ie. LightGBM — *use_missing=false*). However, other algorithms will panic and throw an error complaining about the missing values (ie. Scikit learn — LinearRegression). In that case, you will need to handle the missing data and clean it before feeding it to the algorithm.

Let's see some other ways to impute the missing values before training:

> ***Note: All the examples below use the California Housing Dataset from Scikit-learn.***

2- Imputation Using (Mean/Median) Values:

This works by calculating the mean/median of the non-missing values in a column and then replacing the missing values within each column separately and independently from the others. It can only be used with numeric data.



	col1	col2	col3	col4	col5
0	2	5.0	3.0	6	NaN
1	9	NaN	9.0	0	7.0
2	19	17.0	NaN	9	NaN

mean()

	col1	col2	col3	col4	col5
0	2.0	5.0	3.0	6.0	7.0
1	9.0	11.0	9.0	0.0	7.0
2	19.0	17.0	6.0	9.0	7.0

Mean Imputation

Pros:

- Easy and fast.
- Works well with small numerical datasets.

Cons:

- Doesn’t factor the correlations between features. It only works on the column level.
- Will give poor results on encoded categorical features (do NOT use it on categorical features).
- Not very accurate.
- Doesn’t account for the uncertainty in the imputations.

```python
from sklearn.datasets import fetch_california_housing
from sklearn.linear_model import LinearRegression
from sklearn.model_selection import StratifiedKFold
from sklearn.metrics import mean_squared_error
from math import sqrt
import random
import numpy as np
random.seed(0)

#Fetching the dataset
import pandas as pd
dataset = fetch_california_housing()
train, target = pd.DataFrame(dataset.data), pd.DataFrame(dataset.target)
train.columns = ['0','1','2','3','4','5','6','7']
train.insert(loc=len(train.columns), column='target', value=target)

#Randomly replace 40% of the first column with NaN values
column = train['0']
print(column.size)
missing_pct = int(column.size * 0.4)
i = [random.choice(range(column.shape[0])) for _ in range(missing_pct)]
column[i] = np.NaN
print(column.shape[0])

#Impute the values using scikit-learn SimpleImpute Class
from sklearn.impute import SimpleImputer
imp_mean = SimpleImputer( strategy='mean') #for median imputation replace 'mean' with 'median'
imp_mean.fit(train)
imputed_train_df = imp_mean.transform(train)
```

mean-impute.py hosted with ♡ by GitHub view raw

Mean/Median Imputation

3- Imputation Using (Most Frequent) or (Zero/Constant) Values:

Most Frequent is another statistical strategy to impute missing values and YES!! It works with categorical features (strings or numerical representations) by replacing missing data with the most frequent values within each column.

Pros:

- Works well with categorical features.

Cons:

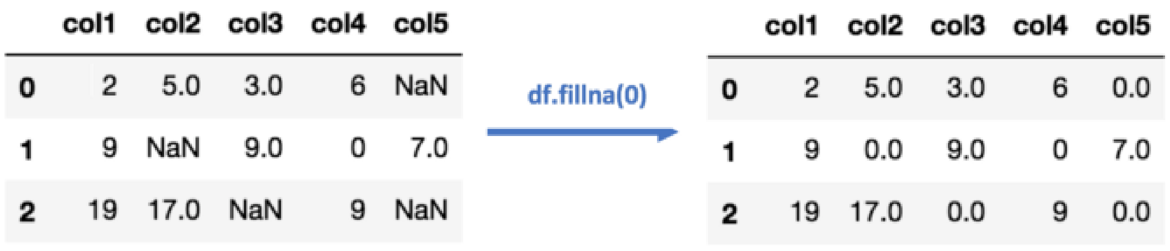
- It also doesn't factor the correlations between features.
- It can introduce bias in the data.

```python
#Impute the values using scikit-learn SimpleImpute Class

from sklearn.impute import SimpleImputer
imp_mean = SimpleImputer( strategy='most_frequent')
imp_mean.fit(train)
imputed_train_df = imp_mean.transform(train)
```

impute_most_frequent.py hosted with ♡ by GitHub view raw

Most Frequent Imputation

Zero or Constant imputation — as the name suggests — it replaces the missing values with either zero or any constant value you specify



	col1	col2	col3	col4	col5
0	2	5.0	3.0	6	NaN
1	9	NaN	9.0	0	7.0
2	19	17.0	NaN	9	NaN

	col1	col2	col3	col4	col5
0	2	5.0	3.0	6	0.0
1	9	0.0	9.0	0	7.0
2	19	17.0	0.0	9	0.0

4- Imputation Using k-NN:

The *k* nearest neighbours is an algorithm that is used for simple classification. The algorithm uses '**feature similarity**' to predict the values of any new data points. This means that the new point is assigned a value based on how closely it resembles the points in the training set. This can be very useful in making predictions about the missing values by finding the *k's* closest neighbours to the observation with missing data and then imputing them based on the non-missing values in the neighbourhood. Let's see some example code using `Impyute` library which provides a simple and easy way to use KNN for imputation:

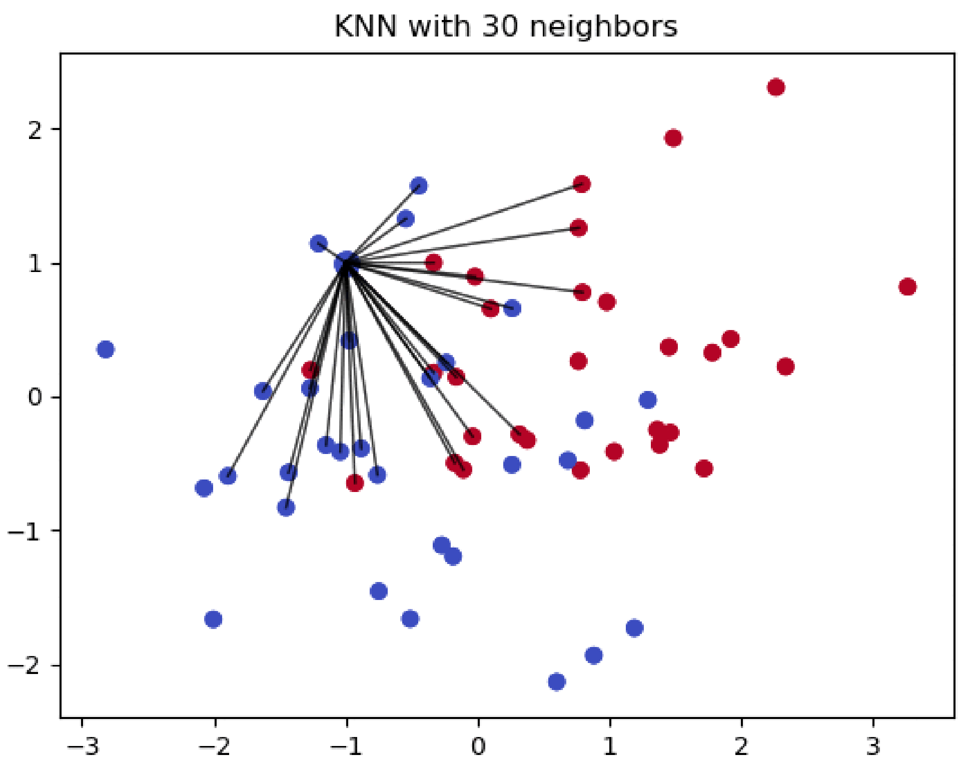
```python
import sys
from impyute.imputation.cs import fast_knn
sys.setrecursionlimit(100000) #Increase the recursion limit of the OS

# start the KNN training
imputed_training=fast_knn(train.values, k=30)
```

impute_knn.py hosted with ♡ by GitHub view raw

KNN Imputation for California Housing Dataset

How does it work?

It creates a basic mean impute then uses the resulting complete list to construct a KDTree. Then, it uses the resulting KDTree to compute nearest neighbours (NN). After it finds the k-NNs, it takes the weighted average of them.



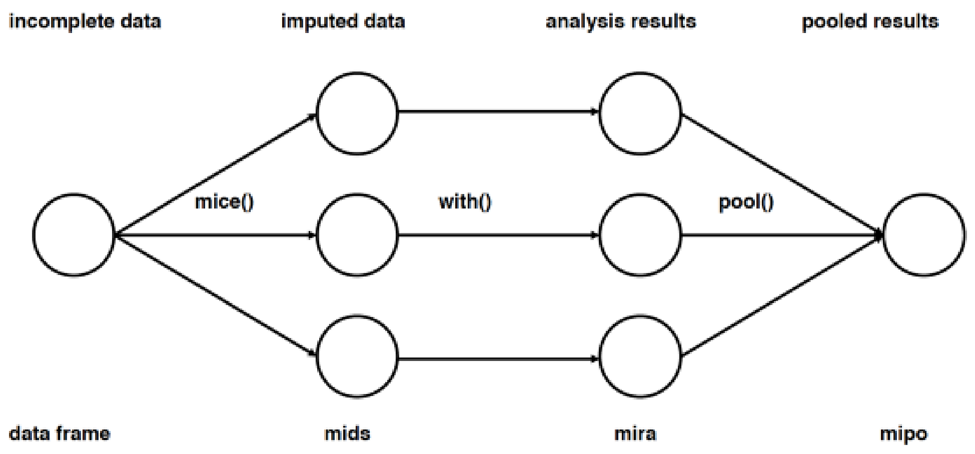
Pros:

- Can be much more accurate than the mean, median or most frequent imputation methods (It depends on the dataset).

Cons:

- Computationally expensive. KNN works by storing the whole training dataset in memory.
- K-NN is quite sensitive to outliers in the data (**unlike SVM**)

5- Imputation Using Multivariate Imputation by Chained Equation (MICE)

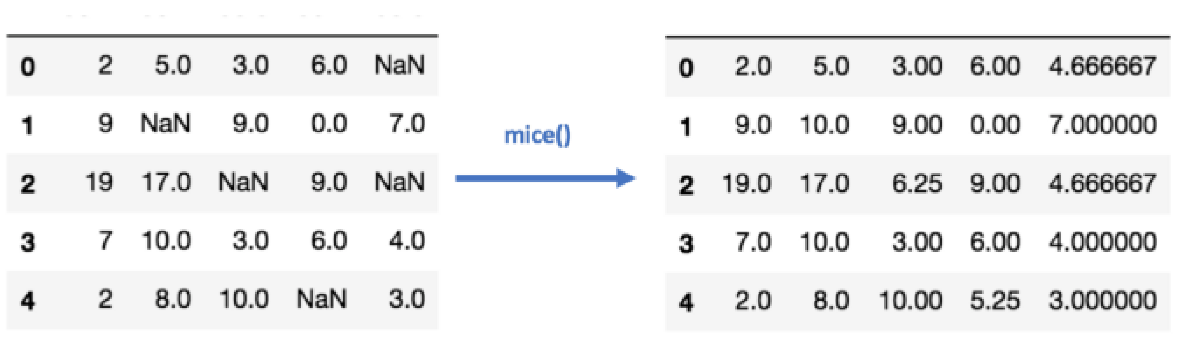


Main steps used in multiple imputations [1]

This type of imputation works by filling the missing data multiple times. Multiple Imputations (MIs) are much better than a single imputation as it measures the uncertainty of the missing values in a better way. The chained equations approach is also very flexible and can handle different variables of different data types (ie., continuous or binary) as well as complexities such as bounds or survey skip patterns. For more information on the algorithm mechanics, you can refer to the Research Paper

```python
from impyute.imputation.cs import mice

# start the MICE training
imputed_training=mice(train.values)
```

impute_mice.py hosted with ♡ by GitHub view raw



0	2	5.0	3.0	6.0	NaN
1	9	NaN	9.0	0.0	7.0
2	19	17.0	NaN	9.0	NaN
3	7	10.0	3.0	6.0	4.0
4	2	8.0	10.0	NaN	3.0

0	2.0	5.0	3.00	6.00	4.666667
1	9.0	10.0	9.00	0.00	7.000000
2	19.0	17.0	6.25	9.00	4.666667
3	7.0	10.0	3.00	6.00	4.000000
4	2.0	8.0	10.00	5.25	3.000000

MICE imputation using impyute

6- Imputation Using Deep Learning (Datawig):

This method works very well with categorical and non-numerical features. It is a library that learns Machine Learning models using Deep Neural Networks to impute missing values in a dataframe. It also supports both CPU and GPU for training.

```python
import datawig

df_train, df_test = datawig.utils.random_split(train)
```

```

#Initialize a SimpleImputer model
imputer = datawig.SimpleImputer(
    input_columns=['1','2','3','4','5','6','7', 'target'], # column(s) containing information ab
    output_column= '0', # the column we'd like to impute values for
    output_path = 'imputer_model' # stores model data and metrics
    )

#Fit an imputer model on the train data
imputer.fit(train_df=df_train, num_epochs=50)

#Impute missing values and return original dataframe with predictions
imputed = imputer.predict(df_test)
```

impute-datawig.py hosted with by GitHub view raw

Imputation using Datawig

Pros:

- Quite accurate compared to other methods.
- It has some functions that can handle categorical data (Feature Encoder).
- It supports CPUs and GPUs.

Cons:

- Single Column imputation.
- Can be quite slow with large datasets.
- You have to specify the columns that contain information about the target column that will be imputed.

Other Imputation Methods:

Stochastic regression imputation:

It is quite similar to regression imputation which tries to predict the missing values by regressing it from other related variables in the same dataset plus some random residual value.

Extrapolation and Interpolation:

It tries to estimate values from other observations within the range of a discrete set of known data points.

Hot-Deck imputation:

Works by randomly choosing the missing value from a set of related and similar variables.

In conclusion, there is no perfect way to compensate for the missing values in a dataset. Each strategy can perform better for certain datasets and missing data types but may perform much worse on other types of datasets. There are some set rules to decide which strategy to use for particular types of missing values, but beyond that, you should experiment and check which model works best for your dataset.

References:

- [1] Buuren, S. V., & Groothuis-Oudshoorn, K. (2011). Mice: Multivariate Imputation by Chained Equations in R. Journal of Statistical Software
- https://impyute.readthedocs.io/en/master/index.html

Machine Learning   Data Science   Data   Python   Analytics