Data Cleaning

Part One:

Saving the Model from the Nightmare of Missing Data

Source of this Post:

Python Data Science Handbook, O'Reilly, Jake VanderPlas

Zahra Amini

What are the stages of Data Cleaning?

Data cleaning is a multi-step process that must be followed sequentially in professional projects:

1. Managing Missing Data
2. Managing Duplicate Data
3. Removing Outliers
4. Handling Inconsistency and Formatting

Now, we will fully dive into the first stage, Missing Data, and explore its challenges with examples and Python code.

Let's begin!😉🤞

What is Missing Data?

The difference between educational datasets (often used in courses) and real-world (business) data is that real-world data is rarely clean and homogeneous. In particular, many interesting datasets will have a significant amount of missing data. Missing data, or NA (Not Available), means that a row (Record) or a column (Feature) in your DataFrame does not have a value.

This problem is the most common challenge in the Data Cleaning process and makes our algorithms unreliable.

How are Missing (Missing) Values Represented?

Missing values are often represented by the following specific terms:

- NaN (Not a Number): This is a special Floating-Point value used to indicate Missing Data in NumPy/Pandas numerical columns.
- Null / None: This is used as a Sentinel value, especially in columns with an Object data type (like text or strings), where it indicates a missing value.
- pd.NA: This is a special value that is used in Pandas Nullable Dtypes (like pd.Int32) and allows for a true Integer value to accommodate a missing value.

What are the Approaches to Managing Missing (NA) Values?

We have two main ways to represent empty slots in the data. When a piece of data is missing, as data professionals, we must indicate its place in the table in a specific way. We have two main strategies for this, each with its own series of trade-offs (pros and cons):

1. Using a Mask (Masking approach)
2. Using a Sentinel Value (Sentinel approach)

Strategy One: Masking

In this approach, we create a separate table that says, "This spot is empty!".
In this method, we either create an entirely separate Boolean Array (Mask) or potentially appropriate one bit in the data representation to locally indicate the null status of a value.
This array (table) tells you: "Is this cell missing data (True) or not (False)?".
The main drawback is that using a separate mask array requires:

- Additional storage space (allocating an extra Boolean array).
- Creating a little bit of computational overhead.

Strategy Two: Sentinel Value

We use a specific value to show that a spot is not data.

- This value might be a data-specific convention, such as using -9999 for missing Integer values.
- Or, it might be a more global convention, such as using NaN (Not a Number), which is used for Floating-Point numbers.

The main drawback of using a sentinel value is that it reduces the range of valid values that can be represented (since you can't use, for example, -9999 as a real value).
It may also require extra (often non-optimized) processing logic because common special values like NaN aren't available for all data types in NumPy.

Why is Pandas a Compromise?

Pandas is forced to use a hybrid approach to manage missing data because it is constrained by its reliance on the NumPy package. NumPy, by default, does not have a built-in notion of NA values for non-floating-point data types (like Integer).

Therefore, Pandas is compelled to use a combination of two "modes" to handle null values:

1. The Default Mode: Using the sentinel-based missing data scheme, with the sentinel values NaN or None depending on the type of data.
2. The Professional (Opt-in) Mode: Using nullable data types (dtypes) (e.g., pd.Int32), which results in the creation of an accompanying Mask array to track missing entries alongside the main data.

How Dangerous is None as a Sentinel (for Object Dtype)?

In general, None is a performance drawback.

- None is a Python object.
- If a NumPy array includes None, the entire array must have dtype=object.
- In this case, all computational operations are no longer performed at the fast NumPy level; instead, they must be done at the Python level, which is much slower and has more overhead.

Additionally, since Python does not support arithmetic operations with None, aggregation functions (like sum) will generally lead to a TypeError.

NaN: Floating–Point Sentinel

What is NaN?NaN (Not a Number) is a special value for Floating-Point numbers that is our savior for speed in numerical calculations.

Key Advantage: When Pandas creates an array with NaN, it can choose a Native data type (like float64) and optimize for speed. This means that, unlike None, operations are performed very quickly in Compiled Code.

A Toxic Note:NaN acts like a data virus; if any number is added, multiplied, or divided by NaN, the result will certainly be another NaN.

Solving the Problem of Summing NaNs (np.nansum)

The problem: Standard NumPy functions don't work well with NaN! The main issue is that if we have an array containing NaN, standard aggregation functions like sum(), min(), or max() will give us unhelpful outputs, typically NaN.

What is the solution? NumPy provides NaN-aware versions of these aggregation functions. These versions ignore the missing values and only operate on the valid data points.

```
np.nansum(vals2), np.nanmin(vals2), np.nanmax(vals2)
```

Data Type Promotion(Upcasting)

NaN is specifically for Float; if you have another data type, it forces a conversion!

The main problem is that NaN is only equivalent to a Floating-Point value and there is no equivalent for other data types like Integer or String.

If you put NaN (or None) into an array that primarily contains Integers, Pandas automatically typecasts (Casts to) the entire array to a Floating-Point type (float64) so that it can store NaN.

Example: You have a Series of Integers. If you set one cell to np.nan or None, the whole Series is converted (upcast) to dtype: float64.

Output Example: A Series of [1, np.nan, 2, None] is automatically converted to dtype: float64 to accommodate the missing values.

```
pd.Series([1, np.nan, 2, None])
```

```
Out[11]:
0    1.0
1    NaN
2    2.0
3    NaN
dtype: float64
```

Summary of Forced Conversions (Upcasting)

See this table which Pandas performs in the background:

Typeclass	Conversion when storing NAs	NA sentinel value
floating	No change	np.nan
object	No change	None or np.nan
integer	Cast to float64	np.nan
boolean	Cast to object	None or np.nan

Note: These conversions might feel a bit confusing (or "hackish"), but in practice, the Pandas sentinel/casting approach works quite well and simplifies your work!

The Professional Solution: Nullable Dtypes (Integer with Null!)

Using Int32 allows you to accept empty slots without forcing a conversion! Previously, it was impossible to have a true Integer array that also contained missing data (because we were forced to upcast to float64).

The solution is that Pandas added Nullable Dtypes.

Note: In these data types, missing values are shown with the special value pd.NA.

```
pd.Series([1, np.nan, 2, None, pd.NA], dtype='Int32')
```

```
Out[14]:
0       1
1    <NA>
2       2
3    <NA>
4    <NA>
dtype: Int32
```

Now, with this approach, you have true Integer values, and at the same time, empty spots are filled with <NA>, without having to change the data type to float.

Core Pandas Tools for Nulls Management

Now let's look at the four main methods that Pandas provides for managing Null values (None, NaN, NA) in your data:

- isnull(): Generates a Boolean mask indicating the missing (null) values.
- notnull(): The opposite of isnull(), which shows the non-missing (valid) values.
- dropna(): Filters the data and removes the missing values (rows or columns).
- fillna(): Returns a copy of the data where the empty slots have been filled or imputed.

Detecting Nulls with data.isnull()

The isnull() method gives us a True/False map!

The data.isnull() command creates a Boolean Mask over the data:

- Empty slots (Null) ---> True
- Valid values ---> False

This method is the practical starting point for any cleaning process.

```
In [15]:

 data = pd.Series([1, np.nan, 'hello', None])

In [16]:

 data.isnull()

Out[16]:
0    False
1     True
2    False
3     True
dtype: bool
```

Using notnull() for Filtering

You don't need to manually delete rows!
You can directly apply the Boolean mask created by notnull() as an index on a Series or DataFrame to keep only the valid rows.

The code below displays all rows that have a valid value:

```
data[data.notnull()]
```

```
Out[17]:
0        1
2    hello
dtype: object
```

Removing Nulls with dropna()

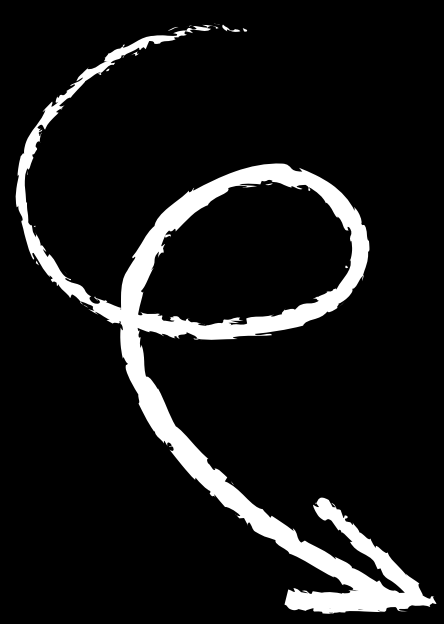
The quickest way to get rid of Nulls is with dropna()!

For a Series, the dropna() method works very simply and just removes the Null values.

Important Note: Be careful that this method returns a filtered copy of the data and does not change the original data, unless you use inplace=True.

```
data.dropna()
```

```
Out[18]:
0        1
2    hello
dtype: object
```



df.dropna(inplace=True)

What's the Problem with dropna() in a DataFrame?

We cannot just remove a single value from a DataFrame!

You cannot remove only one Null value in a DataFrame; you must remove either the entire row or the entire column.

Therefore, depending on the use case, you might want to remove a row or a column. This is why the dropna() method has many parameters to control the operation.

dropna() in a DataFrame with Default Mode (Any)

By default, if there is even one Null value in a row, the entire row is dropped!

By default, df.dropna() uses the parameter how='any'.

Note: The removal of all rows that have at least one Null value often results in dropping a significant volume of data, and only the perfectly clean rows remain.

	0	1	2
0	1.0	NaN	2
1	2.0	3.0	5
2	NaN	4.0	6

```
df.dropna()
```

```
Out[20]:
```

	0	1	2
1	2.0	3.0	5

Removing Columns with Nulls (Controlling Axis)

We want to remove the "bad" columns.

By using axis='columns' (or axis=1), you can tell dropna to perform the removal across the columns.

In this case, all columns that have at least one Null value are removed!

	0	1	2
0	1.0	NaN	2
1	2.0	3.0	5
2	NaN	4.0	6

```
df.dropna(axis='columns')
```

Out[21]:

	2
0	2
1	5
2	6

dropna() in a DataFrame with how='all'

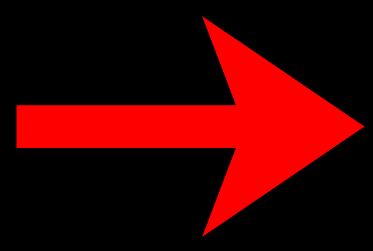
Only remove rows/columns that contain NO data.

If you specify how='all', dropna will only drop rows or columns where all their values are Null.

Example: Assume column 3 (df[3]) is entirely np.nan.

This code will only remove column 3 and keep the rest.

	0	1	2	3
0	1.0	NaN	2	NaN
1	2.0	3.0	5	NaN
2	NaN	4.0	6	NaN



```
df.dropna(axis='columns', how='all')
```

```
Out[23]:
```

	0	1	2
0	1.0	NaN	2
1	2.0	3.0	5
2	NaN	4.0	6

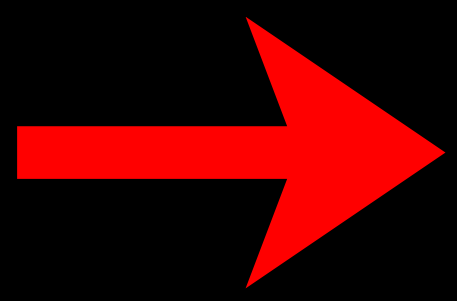
Filling Nulls with a Constant Value using fillna()

The simplest imputation is to fill the missing values with a constant number.

Using the command data.fillna(value), you can replace all Null values with a constant replacement value (e.g., zero).

Warning: Replacing with zero (0) might mislead the model, as zero usually has a special real-world meaning!

```
a       1
b    <NA>
c       2
d    <NA>
e       3
dtype: Int32
```



```
data.fillna(0)
```

```
Out[26]:
a    1
b    0
c    2
d    0
e    3
dtype: Int32
```

Filling Nulls with Mean or Median (Imputation)

Instead of using zero, we fill our data with the mean (or median).

Imputation: Filling missing values with a logical measure (such as the mean, median, or mode).

Note: Using the Median for numerical data that contains Outliers is better than using the mean.

```
df['Column name'].fillna(df['Column name'].mean(), inplace=True)
```

Sequential Replacement with ffill (Forward Fill)

Take the previous value and carry it forward!

The method='ffill' (Forward Fill) finds the previous valid value and uses it to fill the current Null value.

This method is excellent for Time Series data, where today's data is likely close to yesterday's data.

```
a      1
b    <NA>
c      2
d    <NA>
e      3
dtype: Int32
```

```
# forward fill
data.fillna(method='ffill')
```

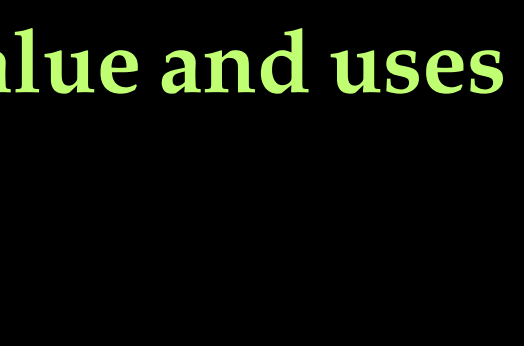
```
Out[27]:
a    1
b    1
c    2
d    2
e    3
dtype: Int32
```

Sequential Replacement with bfill（Backward Fill）

Take the next value and bring it backward!

The method='bfill' (Backward Fill) finds the next valid value and uses it to fill the current Null value.

```
a       1
b    <NA>
c       2
d    <NA>
e       3
dtype: Int32
```



```
# back fill
data.fillna(method='bfill')
```

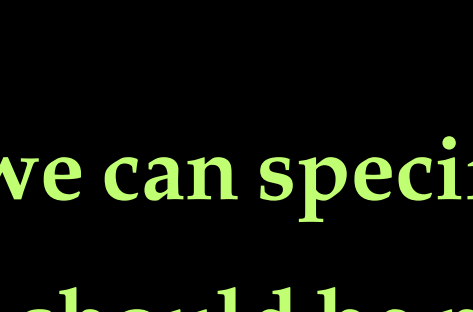
```
Out[28]:
a    1
b    2
c    2
d    3
e    3
dtype: Int32
```

Sequential Replacement in a DataFrame using Axis

You can specify whether to fill row-wise or column-wise!

In a DataFrame, we can specify whether the ffill or bfill operation should be performed along the rows (axis=1) or along the columns (axis=0).

	0	1	2	3
0	1.0	NaN	2	NaN
1	2.0	3.0	5	NaN
2	NaN	4.0	6	NaN



```
df.fillna(method='ffill', axis=1)
```

```
Out[30]:
```

	0	1	2	3
0	1.0	1.0	2.0	2.0
1	2.0	3.0	5.0	5.0
2	NaN	4.0	6.0	6.0

Professional Imputation with Group Imputation

The best guess is to fill in the missing values with the mean of the relevant group.

Why is this method accurate?

Instead of using the mean of the entire column (which has low accuracy), we group the data based on other relevant columns (like Age and Lifestyle), and then fill the Null values with the mean of that specific group.

Example: Assume a patient has a Null value for the LDL (bad cholesterol) level.

- **If a young, 20-year-old patient Smokes (Smoke: Yes), their mean LDL will be higher than the global mean.**
- **If a young, 20-year-old patient is a Non-Smoker (Smoke: No), their mean LDL will be lower than the overall smoker's mean.**

Professional Imputation with Group Imputation (Continuation)

We group our dataset based on the columns AgeGroups (Age Group) and Smoke (Smoking Status). Then, we fill the missing LDL values with the mean LDL calculated only within that "Age + Smoking Status" group.

We start by using the groupby() function and then the transform('mean') method (grouping by two columns):

```
df['LDL'].fillna(df.groupby(['AgeGroups', 'Smoke'])
    ['LDL'].transform('mean'), inplace=True)
```

Key Point: This method has a lower error compared to using the global mean and better preserves the actual structure of the data.

Clean Data is Your First AI Advantage!

Ultimately, managing Missing Data is the foundation of every successful project. We learned how to detect missing data with isnull(), remove it with dropna(), and intelligently replace it with advanced techniques like fillna() (e.g., ffill) and Group Imputation.

What is your preference?

Do you use ffill for Time Series or KNN Imputer for higher accuracy?

Share your experience in the comments! 👇

Zahra Amini