Mastering the Art of Data Cleaning and Preprocessing: Transform Raw Data into Actionable Insights

MUGOT, CHRIS JALLAINE | DS3A

Data cleaning and preprocessing are essential steps in any machine learning project, ensuring that raw data is transformed into a high-quality, usable format for model training. These processes address issues like missing values, inconsistencies, and irrelevant features, which can negatively impact model performance. Preprocessing also involves scaling data, encoding categorical variables, and performing feature engineering to make data interpretable and optimized for algorithms. By improving data quality, these steps enhance model accuracy, efficiency, and speed of convergence.

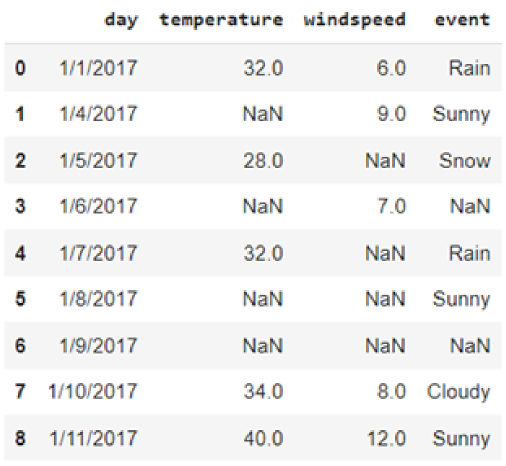
In this report, we will explore why data cleaning and preprocessing are crucial, and we will outline the important steps and processes involved to accomplish this phase effectively.

1. Why do we need to deal with missing values?

Let's keep it real: if we're building a model and feeding it incomplete data, it's like sending someone into a maze without a map. The model will wander around aimlessly, making decisions based on incomplete or unreliable information.

Missing values could be the result of various issues—maybe the data wasn't collected properly, maybe it was just skipped over. But no matter what, those gaps can cause problems. For example, imagine if we are trying to predict someone's salary based on their years of experience, and the "years of experience" field is missing for some people. If you leave that empty, how can the model make an accurate prediction? It's like trying to guess someone's age based only on their hair color.

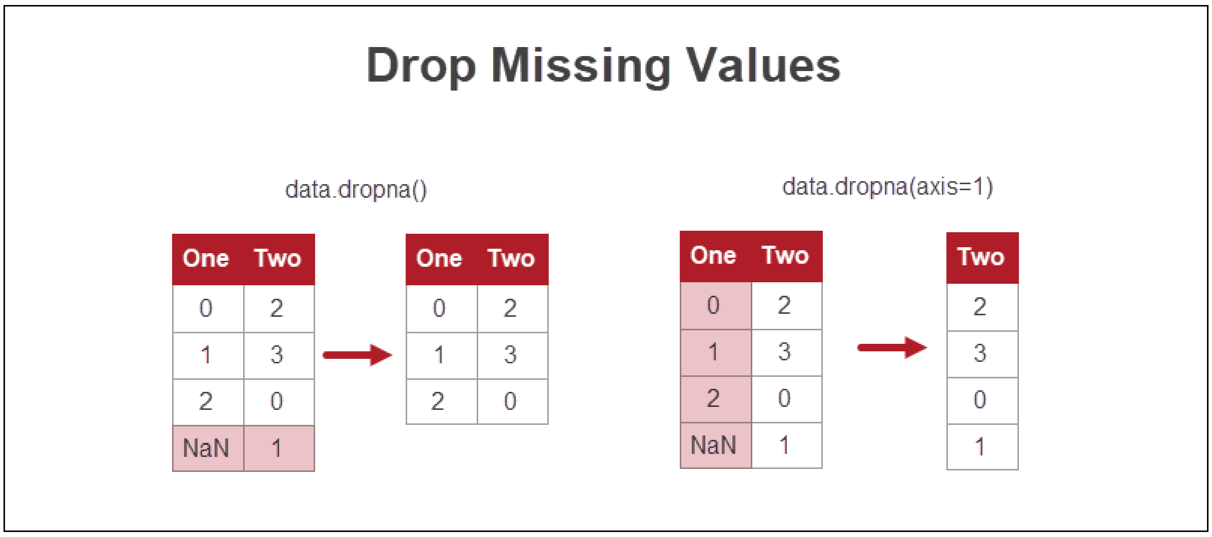
Now, why deal with them? It's not just for neatness—it's about integrity. If we leave gaps, the algorithm might fill them in with random noise, leading to skewed, inaccurate results. Or worse, the algorithm might completely ignore those data points. So, dealing with missing values either by filling them in intelligently or removing them is a necessary step to avoid misleading outcomes. It's about giving the model the best shot at making accurate predictions. See figure 1 for an example of how missing values appear in a dataframe (dataset).



	day	temperature	windspeed	event
0	1/1/2017	32.0	6.0	Rain
1	1/4/2017	NaN	9.0	Sunny
2	1/5/2017	28.0	NaN	Snow
3	1/6/2017	NaN	7.0	NaN
4	1/7/2017	32.0	NaN	Rain
5	1/8/2017	NaN	NaN	Sunny
6	1/9/2017	NaN	NaN	NaN
7	1/10/2017	34.0	8.0	Cloudy
8	1/11/2017	40.0	12.0	Sunny

FIGURE 1: Missing Values

In practice, missing values are often represented in a dataframe as `NaN` (Not a Number) or `Null`. This avoids the need to completely restructure the dataset while still clearly marking gaps in the data. The challenge here is that these `NaN` or `Null` values signal that some information is absent, and we must take action to prevent the model from misinterpreting these blanks. If left unchecked, the model might either ignore these rows or—worse—treat them as zero or some arbitrary value, both of which can distort the learning process. A simple way to address this is by using functions like `dropna()`,(may refer to Figure 1) which removes rows or columns containing missing values, or by imputing those missing values if they are critical to the analysis. It's important to make these decisions carefully, based on the nature of the missing data and the overall impact on the model's predictions. Handling missing values efficiently is one of the key steps in ensuring that your data remains valid and reliable for machine learning models.



One	Two
0	2
1	3
2	0
NaN	1

One	Two
0	2
1	3
2	0

One	Two
0	2
1	3
2	0
NaN	1

Two
2
3
0
1

FIGURE 2: Dropping Missing Values, Source: https://phoenixnap.com/kb/handling-missing-data-in-python

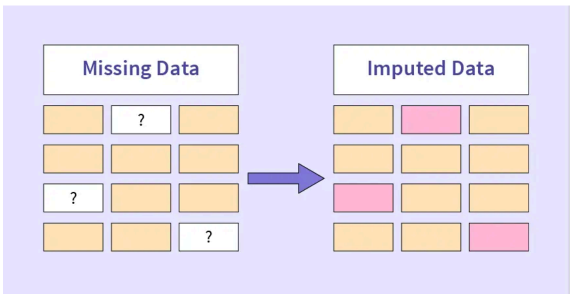
2. Why do we need to deal with missing values?

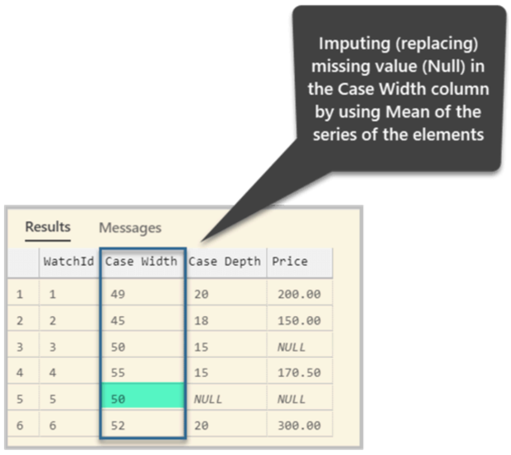
Ah, this is a million-dollar question. When is it *right* to impute missing values? It's not a one-size-fits-all solution. We don't just throw a random number or a placeholder into the gaps and call it a day. Let's look at the context.

First, imputation is most useful when we have a lot of missing values in a non-critical feature. If a feature is critical to our model and 80% of it is missing, then imputation will be like putting a band-aid on a broken leg. But if only a few data points are missing, imputing them with a mean, median, or mode might make sense.

Another reason to impute is when the missing data could be predictive. For example, in a customer dataset, if some people's income is missing but the rest of their demographic info (age, location, etc.) is available, imputing income based on those other features can be more sensible than just discarding the record.

Bottom line: Impute only when it *makes sense* and when you're reasonably confident that the imputed data won't distort the real patterns you're trying to detect. You wouldn't try to repair a glass window with duct tape—same principle.




Imputing (replacing) missing value (Null) in the Case Width column by using Mean of the series of the elements

	WatchId	Case Width	Case Depth	Price
1	1	49	20	200.00
2	2	45	18	150.00
3	3	50	15	*NULL*
4	4	55	15	170.50
5	5	50	*NULL*	*NULL*
6	6	52	20	300.00

3. What is one-hot encoding? Explain.

One-hot encoding is a classic. Imagine that we are at a party with friends, and someone asks, "What kind of pizza do you want?" We could say "Cheese, Pepperoni, Veggie, or Margherita." Now, one-hot encoding is basically turning those four pizza options into binary flags:

- Cheese = [1, 0, 0, 0]
- Pepperoni = [0, 1, 0, 0]
- Veggie = [0, 0, 1, 0]
- Margherita = [0, 0, 0, 1]

Each option gets its own hotspot (a 1), and everything else gets a 0. We might ask why? The simple explanation is that most machine learning algorithms need to understand categories numerically, but with one-hot encoding, we avoid treating these pizza types as a scale (e.g., "Cheese" isn't "higher" than "Pepperoni"). Instead, they are all distinct and equal, keeping the relationships clear.

In short, it's like breaking down a pizza into slices—each slice gets its own label, but they're all part of the same pizza party. It works great for non-ordinal categories (where the categories don't have an inherent order).

Refer to table 1 for example.

Table 1: One Hot Encoding Example

Flavor	ONE HOT ENCODED			
	Cheese	Pepperoni	Veggie	Margherita
Cheese	1	0	0	0
Pepperoni	0	1	0	0
Veggie	0	0	1	0
Margherita	0	0	0	1

In machine learning, **one-hot encoding** is a technique used to convert categorical data into a numerical format that algorithms can understand. It transforms each category into a binary vector, where each category corresponds to a unique column in the data.

For example, again if we have a categorical feature with four possible values (Cheese, Pepperoni, Veggie, and Margherita), one-hot encoding creates four new binary columns. Each row will have a `1` in the column that corresponds to its category and `0`s in the others. This ensures that the machine learning algorithm doesn't mistakenly interpret any relationship between the categories (e.g., "Cheese" is not greater than "Pepperoni").

It's particularly useful for **nominal (non-ordinal)** categories, where the values have no intrinsic order. By using one-hot encoding, you allow the model to treat each category as a distinct entity, without imposing any order or hierarchy between them.

4. Why do you need to encode ordinal features and class labels?

Ordinal features, like education level (e.g., High School, Bachelor's, Master's, PhD), have an inherent rank or order that is critical for the model to capture. Without encoding these features numerically, we might risk misrepresenting the relationships between categories. Machine learning algorithms, particularly those based on mathematical models, can't interpret text or labels directly. If we were to leave these as raw strings, the model would have no understanding of their relative importance or hierarchy, rendering it essentially blind to the order of these categories. To encode ordinal features, you need to assign a numerical value that preserves the order. For instance, assigning:

- High School = 1,
- Bachelor's = 2,
- Master's = 3,
- PhD = 4.

This mapping ensures that the algorithm understands that there's a progression from one category to the next, meaning it can learn the relative relationships between them. This encoding method is essential for algorithms like decision trees, logistic regression, or SVMs, which rely on understanding the distance or ranking between data points.

For class labels (e.g., 'spam' vs. 'not spam'), they do not have an intrinsic order but are still categorical. To use these labels in machine learning, they must also be numerically encoded. For binary classification tasks, this can often be as simple as encoding 'spam' as 1 and 'not spam' as 0. For multi-class classification, techniques like label encoding or one-hot encoding are typically used to map each class to a unique integer or binary vector.

In both cases, encoding allows algorithms to interpret and process categorical data correctly, facilitating accurate learning and predictions. It also ensures that models don't incorrectly assume any relationships where there are none (as with nominal data), or misinterpret rank and progression (as with ordinal data)

5. Why do we need to bring features onto the same scale?

When we're working with machine learning algorithms, especially those like gradient descent-based models (e.g., linear regression, logistic regression, neural networks), feature scaling becomes crucial. This is because these models optimize using gradient-based optimization, which depends on the cost function's gradient. If one feature has a vastly larger scale than another (e.g., income ranging from 1,000 to 1,000,000 versus age ranging from 0 to 100), the optimization algorithm will prioritize minimizing the larger magnitude feature, essentially ignoring the smaller-scale feature.

Consider the effect of gradient descent: It works by adjusting weights in the model based on the gradient (or slope) of the error function. If one feature (like income) has a significantly larger scale than others (like age), the gradient steps for income will dominate, causing the model to focus excessively on income while completely downplaying age. This biases the model's learning process, leading to suboptimal performance.

Facts Supporting the Need for Feature Scaling:

1. SVM (Support Vector Machines): SVM uses the Euclidean distance between points to calculate the margin of separation. If features have different scales, the distance calculation will be heavily skewed by the features with larger values, leading to misclassification. For example, if you're classifying income (1,000–1,000,000) versus age (0–100), the algorithm will consider income far more important than age unless both are scaled to similar ranges.

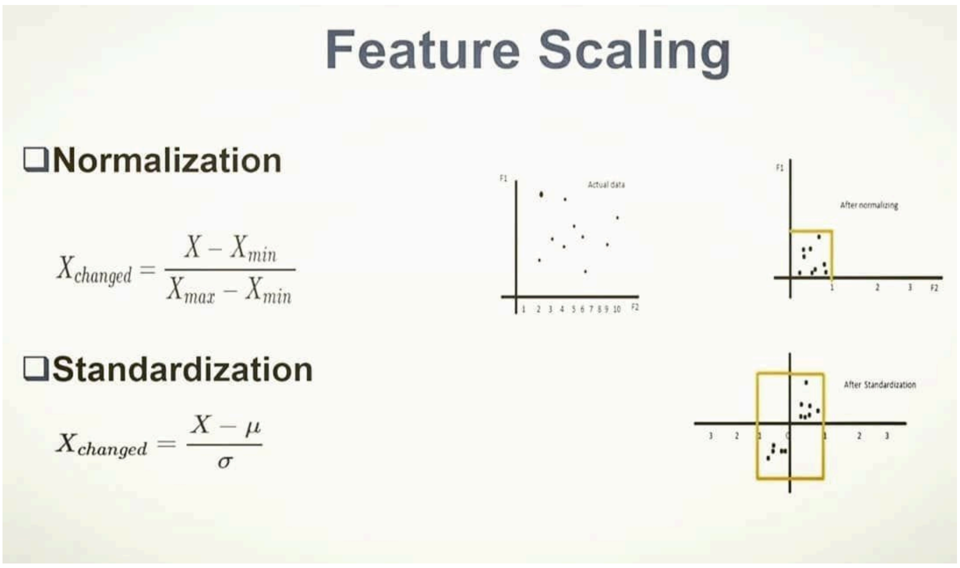
2. K-Nearest Neighbors (KNN): KNN relies on distance metrics (e.g., Euclidean distance) to find the nearest neighbors to a point. If features are not scaled, the algorithm will prioritize features with larger scales (e.g., income) and ignore smaller-scale features (e.g., age). This is problematic because KNN essentially makes decisions based on distances, which are distorted when features differ in magnitude.

3. Neural Networks: For neural networks, backpropagation uses gradients to adjust weights across the network. If one feature has a much larger range than others, the weight update will disproportionately affect that feature. This can lead to an imbalance in learning and slower convergence, making it harder for the model to find an optimal solution.

To ensure each feature contributes equally to the model, it's common practice to use scaling techniques like:

- Min-Max Scaling: This scales the data to a fixed range, usually [0, 1], ensuring each feature is treated with equal importance.
- Standardization (Z-Score Normalization): This rescales features to have a mean of 0 and a standard deviation of 1. It's especially useful when the data contains outliers and for algorithms that assume normally distributed data (e.g., linear regression).

Scaling our features removes bias from the model, allowing it to learn effectively from all features—big and small—without giving undue weight to any one dimension based purely on its numerical range.



Feature Scaling

❑Normalization

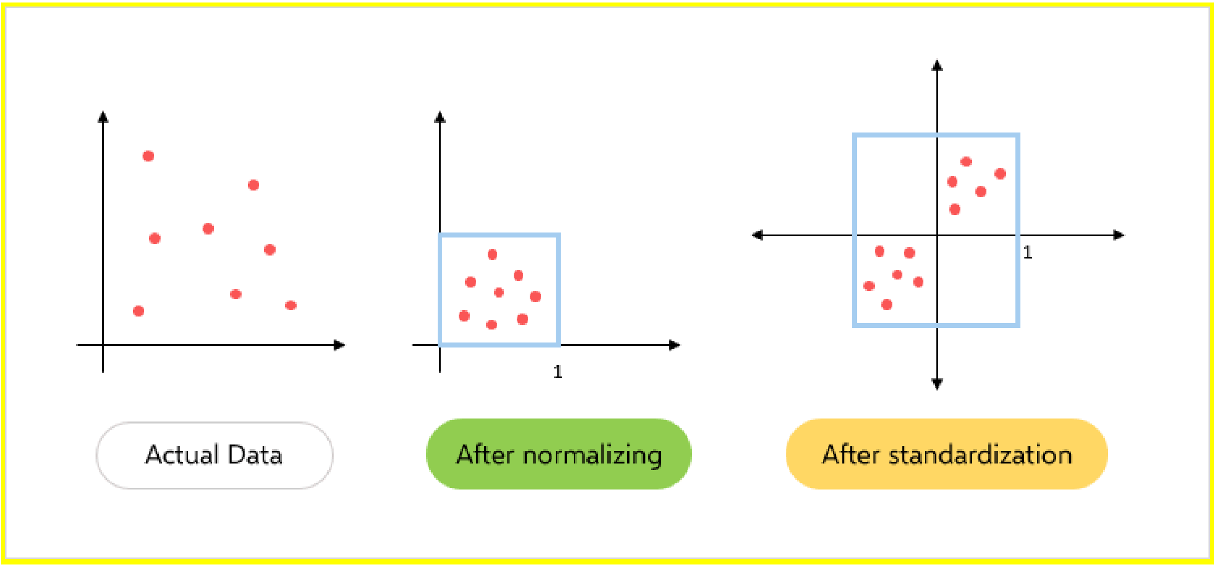
$$X_{changed} = \frac{X - X_{min}}{X_{max} - X_{min}}$$

❑Standardization

$$X_{changed} = \frac{X - \mu}{\sigma}$$

Actual data
After normalizing
After Standardization

In an illustration, it could look like this:



6. What is MinMax scaling? Explain.

MinMax scaling in simple terms is like saying, "I'll take the range of my data and squash it between 0 and 1, no exceptions." It's a simple yet effective technique that re-scales all your features so they fit into a set range, typically [0, 1].

Here's the deal: say we have a feature with values between 10 and 1,000. MinMax scaling will map 10 to 0 and 1,000 to 1. Any value in between is proportionally transformed to fit between 0 and 1.

This is great for algorithms like KNN (K-nearest neighbors) that are sensitive to the scale of the input features. Think of it as a way to fit your data into a neat, standardized grid. However, a big caveat—if there's an outlier (a value way outside your normal range), it can totally throw off the scaling because the transformation is based on the minimum and maximum values. So, while it's useful, you've got to make sure those outliers aren't playing dirty.

Take this example implementation in scikit-learn:

```python
from sklearn.preprocessing import MinMaxScaler

# Initialize the scaler
scaler = MinMaxScaler()

# Fit the scaler to the data and transform the data
scaled_data = scaler.fit_transform(data)
```

Considerations:

- **Sensitivity to Outliers:** Min-Max Scaling is sensitive to outliers. An outlier can significantly affect the minimum and maximum values, leading to a compressed range for the majority of the data. In such cases, alternative scaling methods like Robust Scaling, which uses the interquartile range, might be more appropriate.