# Data Leakage

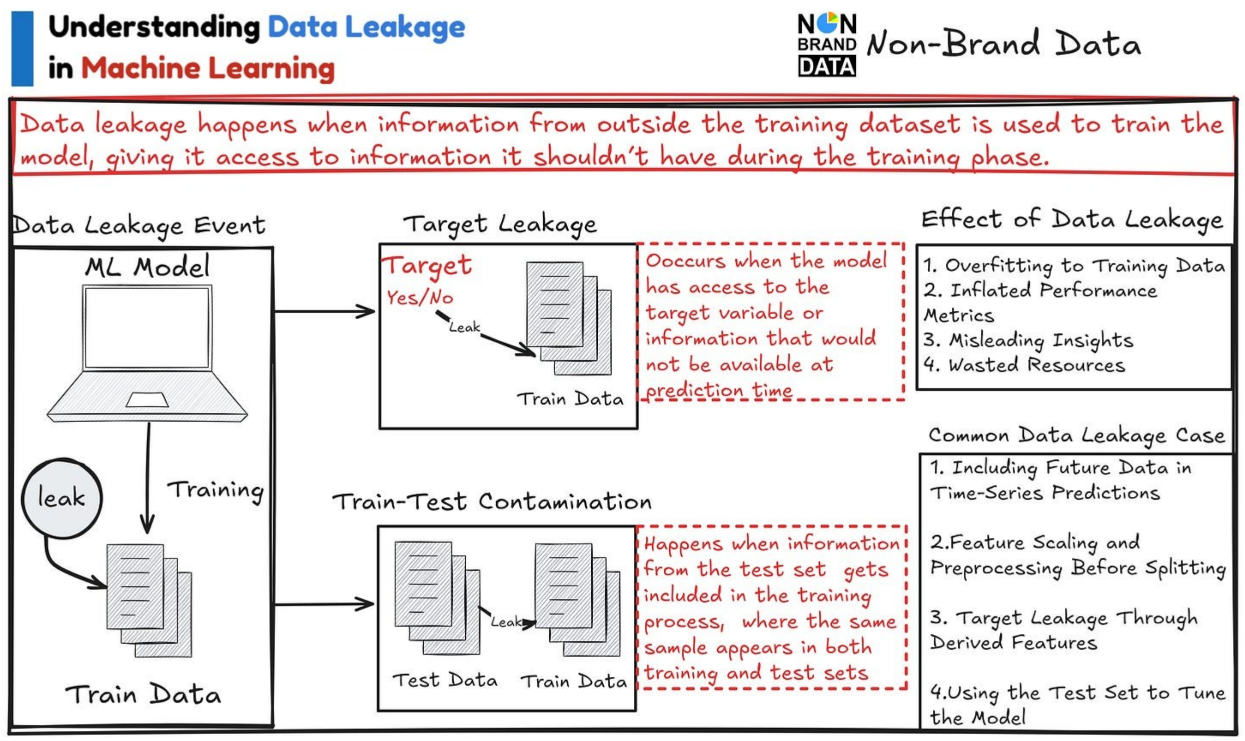
Have you ever had to train your model and achieve perfect performance?

Like 100% accuracy, precision, recall, etc., in the test set?

Would you be happy if that happened? Well, I would certainly dumbfounded as I can't believe it.

There is a saying that "***All models are wrong; some are useful."***

It means that models can't be perfect; if it happens, something might be wrong.


Understanding Data Leakage in Machine Learning
Non-Brand Data
Data leakage happens when information from outside the training dataset is used to train the model, giving it access to information it shouldn't have during the training phase.
Data Leakage Event
ML Model
leak
Training
Train Data
Target Leakage
Target
Yes/No
Leak
Train Data
Ooccurs when the model has access to the target variable or information that would not be available at prediction time
Train-Test Contamination
Leak
Test Data
Train Data
Happens when information from the test set gets included in the training process, where the same sample appears in both training and test sets
Effect of Data Leakage
1. Overfitting to Training Data
2. Inflated Performance Metrics
3. Misleading Insights
4. Wasted Resources
Common Data Leakage Case
1. Including Future Data in Time-Series Predictions
2.Feature Scaling and Preprocessing Before Splitting
3. Target Leakage Through Derived Features
4.Using the Test Set to Tune the Model

**Data Leakage**

Data Leakage is an event where the training dataset we have contains information from outside that shouldn't be.

Why should we be concerned about data leakage? There are a few things that could happen, including:

1. Overfitting to Training Data
2. Inflated Performance Metrics
3. Misleading Insights

4. Wasted Resources

We don’t want data leakage during the model training process.

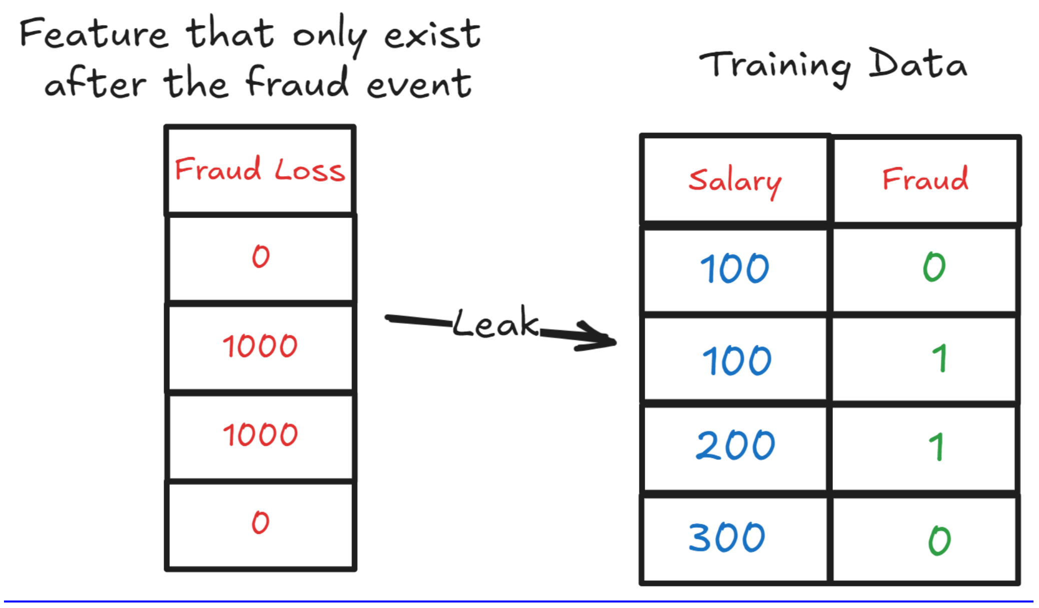
There are two types of Data Leakage, they are:

---

**1. Target Leakage**

Target Leakage occurs when the model is trained on training data that contains target or feature information that should not be available at the prediction time.

For example, let’s take a look at the table below.



We have training data that wants to predict fraud occurrence. However, there is a leak with a feature called Fraud Loss, which exists only after the fraud event.

The presence of a Fraud Loss feature means there is a Target Leakage that would cause the **model overfitting**.

That’s why we need to avoid any information that directly affects the prediction but should not be available during the prediction time.

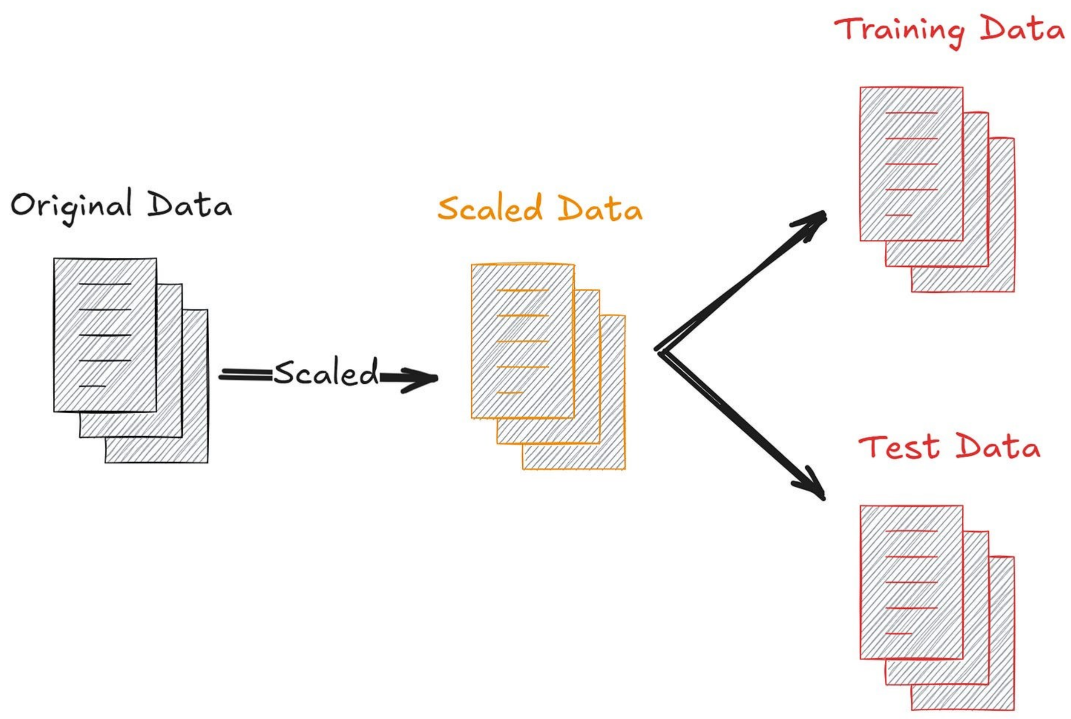
---

**2. Train-Test Contamination**

The Train-Test contamination is an event where the test data “leaks” into the training data.

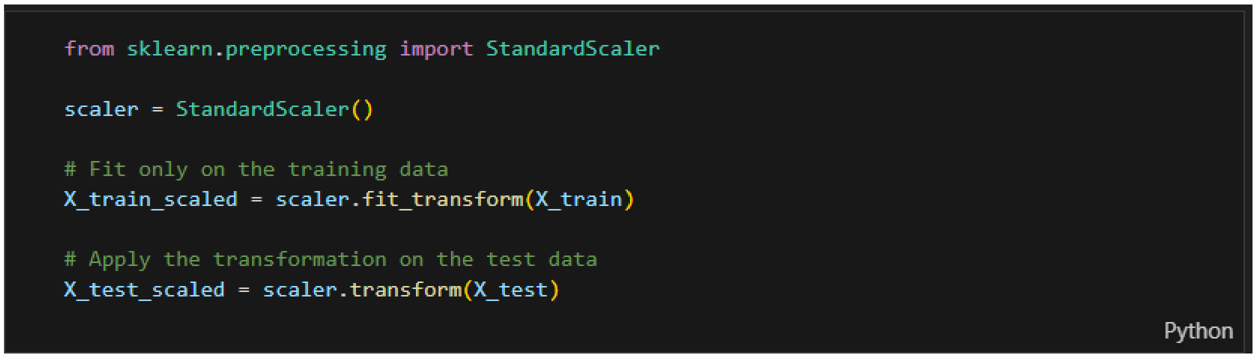
There are many situations where it could happen, including:

- **Data preprocessing steps for transformation (e.g., scaling or encoding) are applied before splitting the dataset.**



Data transformation, such as normalization, requires parameters from the data applied to the whole dataset. Test data should not contain any information from the training data. Thus, data leakage happens if data transformation is done before splitting the dataset.

- **Improper time-series handling, where future information is used to predict past events.**

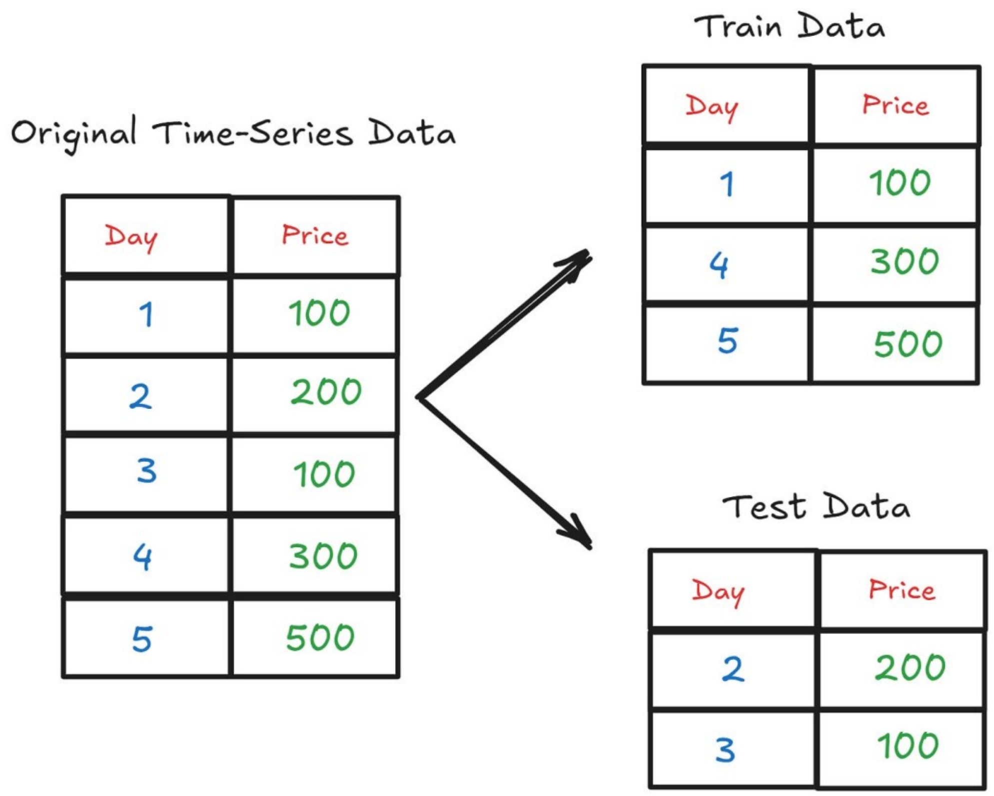


```python
from sklearn.preprocessing import StandardScaler

scaler = StandardScaler()

# Fit only on the training data
X_train_scaled = scaler.fit_transform(X_train)

# Apply the transformation on the test data
X_test_scaled = scaler.transform(X_test)
```


Original Time-Series Data

| Day | Price |
|---|---|
| 1 | 100 |
| 2 | 200 |
| 3 | 100 |
| 4 | 300 |
| 5 | 500 |

Train Data

| Day | Price |
|---|---|
| 1 | 100 |
| 4 | 300 |
| 5 | 500 |

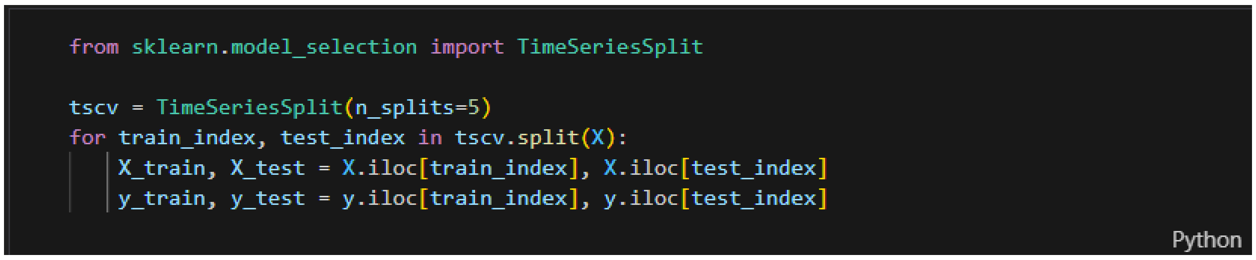
Test Data

| Day | Price |
|---|---|
| 2 | 200 |
| 3 | 100 |

We can't split time-series data in the same way as our normal tabular data. Time-series data is special in that the data is ordered, and each data point is related in some way.

If we split them, there would usually be leakage as the data from the future is used to predict the past—which should not happen.

Here is a simple Python implementation for splitting the time series.



```
from sklearn.model_selection import TimeSeriesSplit

tscv = TimeSeriesSplit(n_splits=5)
for train_index, test_index in tscv.split(X):
    X_train, X_test = X.iloc[train_index], X.iloc[test_index]
    y_train, y_test = y.iloc[train_index], y.iloc[test_index]
```

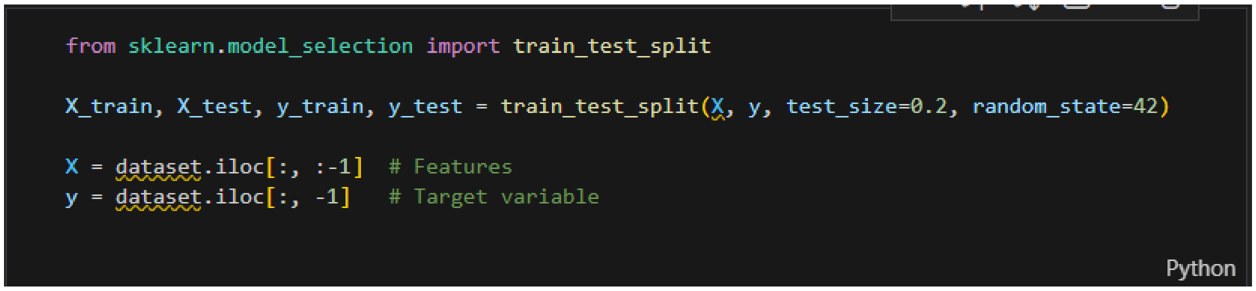
## Normalize data before or after split of training and testing data?

When working with machine learning models, it is important to preprocess the data before training the model. One common preprocessing technique is data normalization, which involves scaling the features of the dataset to a standard range. However, a crucial question arises: Should we normalize the data before or after splitting it into training and testing sets? In this explanation, we'll explore this question, provide the reasoning behind the recommended approach, and demonstrate its implementation in Python.

The recommended approach is to normalize the data after splitting it into training and testing sets. The rationale behind this recommendation is to prevent any information leakage from the testing set into the training set, which can lead to over-optimistic results and unrealistic performance evaluations. Here's the step-by-step process:

1.Data Splitting: First, we split the dataset into a training set and a testing set. This can be achieved using various techniques, such as random sampling or time-based splitting. It's important to ensure that the data points in the testing set are representative of real-world, unseen data.

The training set is used to train the model, while the testing set is used to evaluate its performance. The train_test_split function from scikit-learn is commonly used for this purpose. The test_size parameter determines the proportion of the dataset allocated for testing (e.g., 0.2 for a 20% testing set). The random_state parameter ensures reproducibility by fixing the random seed for the splitting process.



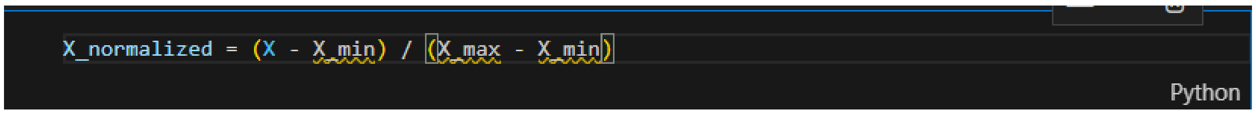
```
from sklearn.model_selection import train_test_split

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

X = dataset.iloc[:, :-1]  # Features
y = dataset.iloc[:, -1]   # Target variable
```

Python

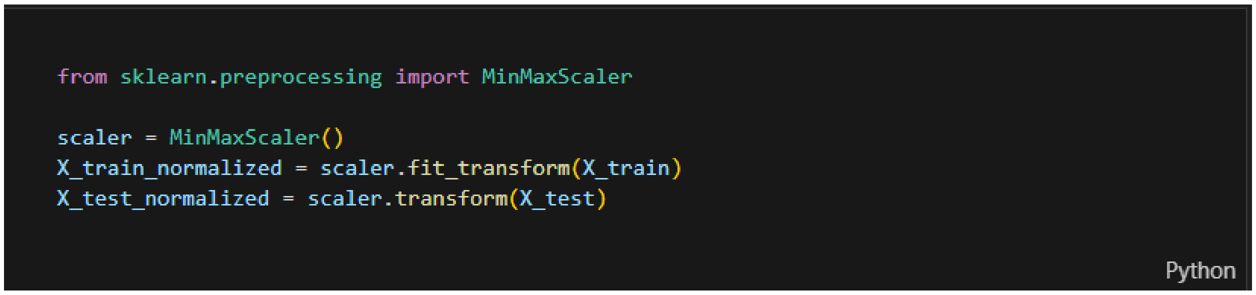
2. Normalization: Normalization involves scaling the features of the dataset to a standard range. One popular technique is Min-Max scaling, which scales the data to a specific range, often between 0 and 1, using the minimum and maximum values of the feature.



```
X_normalized = (X - X_min) / (X_max - X_min)
```

Python

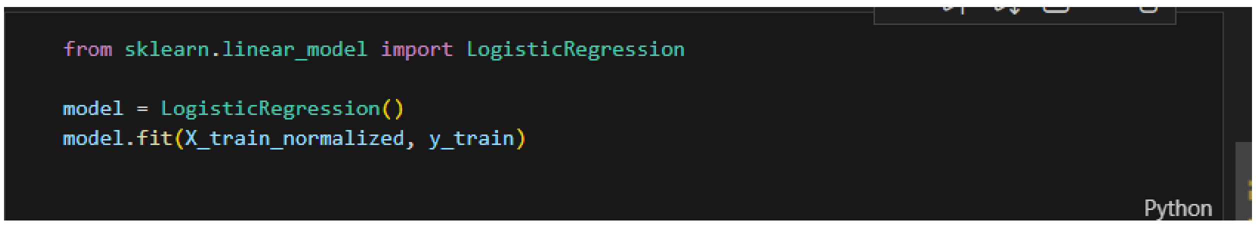
For each feature X, we subtract the minimum value (X_min) and divide by the range (X_max - X_min). This ensures that the minimum value of the feature becomes 0, and the maximum value becomes 1. Values in between are linearly scaled accordingly.



```python
from sklearn.preprocessing import MinMaxScaler

scaler = MinMaxScaler()
X_train_normalized = scaler.fit_transform(X_train)
X_test_normalized = scaler.transform(X_test)
```

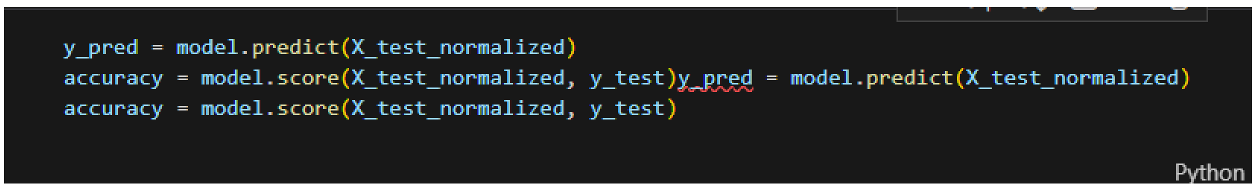
3. Model Training: Now, we can train our machine learning model using the normalized training data. The model learns from the scaled features and aims to capture patterns and relationships in the data.



```python
from sklearn.linear_model import LogisticRegression

model = LogisticRegression()
model.fit(X_train_normalized, y_train)
```

4. Model Evaluation: Finally, we evaluate the trained model's performance using the normalized testing data. By applying the same normalization technique used on the training set to the testing set, we ensure that the model's predictions are consistent with real-world scenarios.



```python
y_pred = model.predict(X_test_normalized)
accuracy = model.score(X_test_normalized, y_test)y_pred = model.predict(X_test_normalized)
accuracy = model.score(X_test_normalized, y_test)
```

By normalizing the data after splitting, we maintain the integrity of the testing set as an unbiased evaluation metric. This approach simulates the scenario where the model encounters unseen data during deployment.

Note that the normalization process involves calculating statistical parameters (e.g., minimum, maximum, mean, standard deviation) on the training set and then applying those parameters to normalize the testing set. This ensures that the testing set remains entirely separate from the training process.

Remember to apply the same normalization technique used during training to any new, unseen data that your model encounters during deployment to ensure consistency.

Overall, normalizing the data after splitting the training and testing sets is the recommended approach to obtain reliable performance evaluations for machine learning models.

---

**What do fit and transform mean?**

- **fit**: means **calculating statistics** from the data.
    - Example: If we use MinMaxScaler, it will calculate X_min and X_max for each feature.
- **transform**: means **applying the transformation** to the data using the calculated statistics.
- **fit_transform** = calculate statistics + apply the transformation at once.

**Why do we use fit_transform only on the training set?**

Because the training set is **the data the model learns from**.

- We want the model to **learn only from the training data**.
- If we use the test data during fit, the model would "see" test values before training. This is called **Data Leakage**, and it makes the model evaluation unrealistic.

**Why do we use transform only on the test set?**

- Once we calculate X_min and X_max from the training set, we want to **apply the same transformation to the test set**.
- We do not recalculate X_min and X_max from the test set because the test set **should remain unseen by the model**.
- This way, the model evaluation reflects performance on truly new, unseen data.

**Small illustrative example:**

Imagine we have one feature in the training and test sets:

| **Training Data** | **Test Data** |
|---|---|
| 2 | 10 |
| 4 | 12 |
| 6 | 14 |

- **fit_transform on training**:
    - min = 2, max = 6
    - after normalization: [0, 0.5, 1]
- **transform on test**:
    - we use the same min=2 and max=6
    - after normalization: [(10-2)/(6-2)=2], [(12-2)/4=2.5] → outside the 0-1 range, which is normal because test data is different.

If we had also used fit_transform on the test set, **the model would see test values before training**, giving misleading results.

In short:

fit_transform on training = learn from training and apply
transform on test = apply what we learned without seeing the test data beforehand