Nadhif Rif'at Rasendriya

udemy

Data Preprocessing

With Python

Data Preprocessing in Machine Learning

Data preprocessing in Machine Learning (ML) is the crucial step of preparing raw data before feeding it into a model. It involves transforming, cleaning, and organizing the data to improve model accuracy and efficiency.

Steps in Data Preprocessing:

- Cleaning the data → Fix missing values, remove duplicates, and correct errors.
- Transforming the data → Convert data into a format the model understands (e.g., changing words into numbers).
- Scaling the data → Make sure all numbers have a similar range to avoid big differences affecting the model.
- Reducing unnecessary data → Remove unimportant details to make training faster and more efficient.

Why is Data Preprocessing Important?

Data preprocessing is important because raw data is often messy and inconsistent. If we use unprocessed data in machine learning, the model might learn incorrect patterns and give poor results.

Imagine making a smoothie. If you put in unwashed, unpeeled fruits with seeds and dirt, the smoothie won't taste good. But if you wash, peel, and cut the fruits properly, you get a smooth and delicious drink. Similarly, preprocessing data ensures the machine learning model gets the best "ingredients" to learn from!

Fixes Missing or Incorrect Data

Missing values can mislead the model. Filling them with averages or removing them helps maintain accuracy.

Ensures Fair Comparisons Between Data Points

Different units (e.g., cm and kg) can create imbalances. Scaling data (e.g., between 0 and 1) helps the model learn fairly.

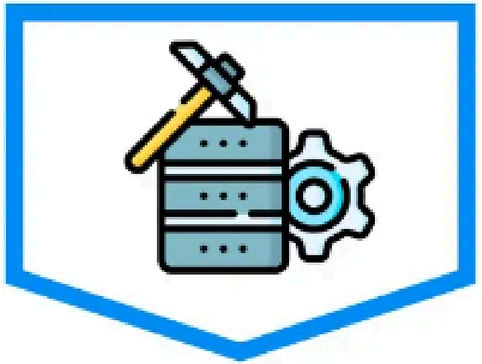
Makes Data Understandable for the Model

Machine learning models work with numbers, so we need to convert text data (e.g., "Male"/"Female") into numerical values (e.g., 0 and 1).

Removes Unnecessary or Duplicate Data

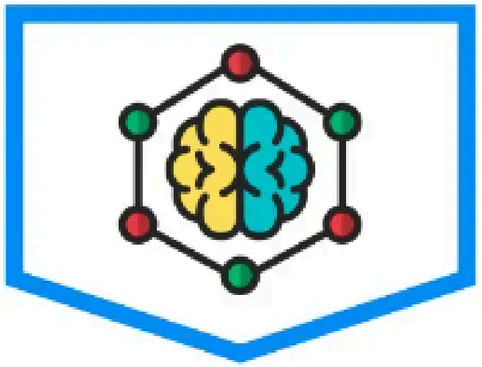
Extra or repeated data can slow down the model and reduce accuracy. Cleaning it up makes training faster and more efficient.

The Machine Learning Process



Data Pre-Processing

- Import the data
- Clean the data
- Split into training & test sets
- Feature Scaling



Modelling

- Build the model
- Train the model
- Make predictions

Evaluation

- Calculate performance metrics
- Make a verdict

DATASET

https://drive.google.com/drive/folders/1rUf_K9L3Xdizpi Wr-b9USRDoro7Cp12-?usp=sharing

Taking Care of Missing Data

Imputer

In this code, we are using **SimpleImputer** from the sklearn.impute module, which is part of the Scikit-Learn library.

Here's a breakdown of the code:

1. Library & Tools Used
   - We import SimpleImputer from sklearn.impute, which is a tool used for handling missing values in datasets.
   - We also use np.nan (assuming NumPy is imported as np) to specify missing values.
2. Method Used
   - SimpleImputer is instantiated with the strategy='mean', meaning it will replace missing values with the mean of the respective column.
3. Applying Imputation
   - The method .fit(X[:, 1:3]) calculates the mean of columns 1 and 2 (indexing starts from 0) in the dataset X.
   - The method .transform(X[:, 1:3]) replaces missing values in these columns with the computed mean.
   - The result is assigned back to X[:, 1:3], updating the dataset.

Taking Care of Missing Data

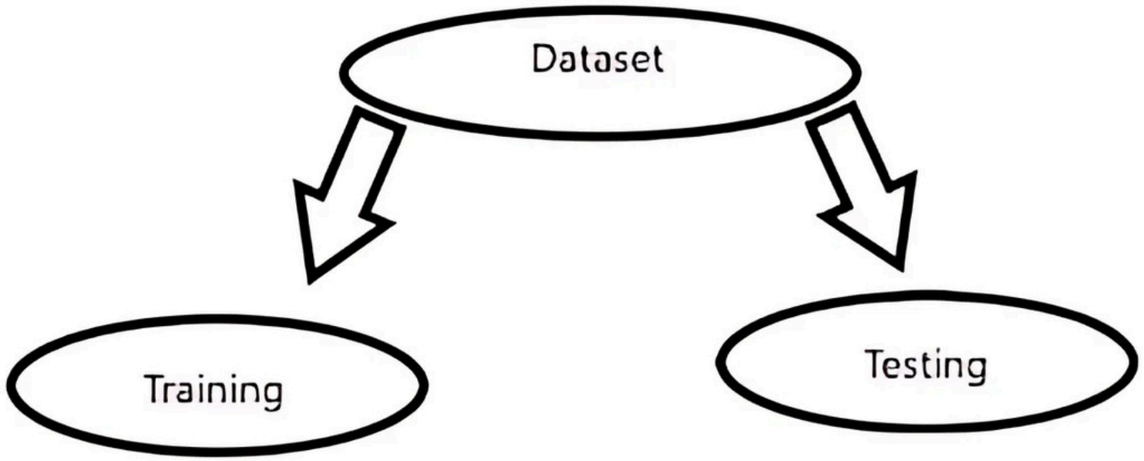
Replace the missing value of the data by the average of all the values in the coloumn.

```
[6] from sklearn.impute import SimpleImputer
    imputer = SimpleImputer(missing_values=np.nan, strategy='mean')
    imputer.fit(X[:, 1:3])
    X[:, 1:3] = imputer.transform(X[:, 1:3])
```

```
[7] print(X)
```

```
[['France' 44.0 72000.0]
 ['Spain' 27.0 48000.0]
 ['Germany' 30.0 54000.0]
 ['Spain' 38.0 61000.0]
 ['Germany' 40.0 63777.77777777778]
 ['France' 35.0 58000.0]
 ['Spain' 38.77777777777778 52000.0]
 ['France' 48.0 79000.0]
 ['Germany' 50.0 83000.0]
 ['France' 37.0 67000.0]]
```

Splitting the dataset into the Training set and Test set

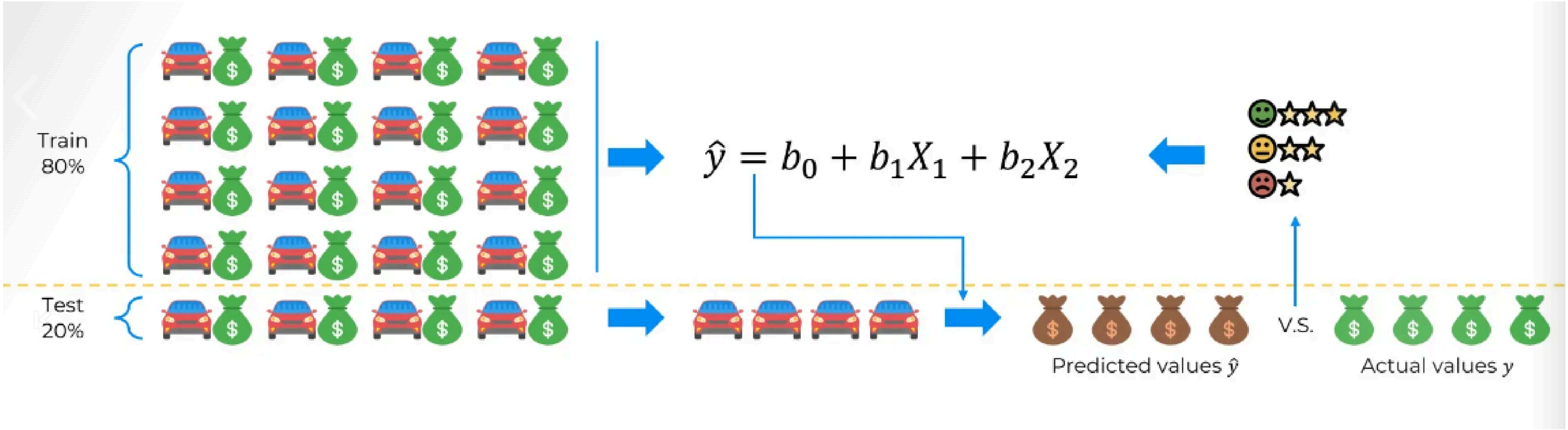


Why in machine learning, we have to do dataset splitting before feature scaling? Why not after?

In machine learning, data splitting must be done before feature scaling to prevent data leakage and ensure a realistic model evaluation. If scaling is performed before splitting, statistics like mean and standard deviation are computed from the entire dataset, allowing information from the test set to influence the training process. This can lead to overly optimistic performance estimates that do not reflect real-world scenarios.

By splitting first, the scaler learns only from the training data, ensuring that the test set remains independent and mimics unseen real-world data. This approach helps avoid bias, maintain model generalization, and produce more reliable results when the model is applied to new data.

Training Set & Test Set



Training Set & Test Set 80:20

This split is commonly used when you have a moderate to large dataset and do not need a separate validation set. The model is trained on 80% of the data and evaluated on the remaining 20% to estimate its performance on unseen data. This method is suitable when hyperparameter tuning is minimal or done using techniques like cross-validation, ensuring a reliable performance estimate without sacrificing too much training data.

Training Set & Test Set & Validation Set 80:10:10

This split is ideal when hyperparameter tuning is required, especially in deep learning or complex models. The training set (80%) is used to train the model, the validation set (10%) helps in tuning hyperparameters and preventing overfitting, and the test set (10%) provides a final unbiased evaluation. This method is crucial when you cannot use cross-validation or when optimizing model performance is a priority.

The Result

Splitting the dataset into the Training set and Test set

```
[24] 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 = 1)
```

```
[25] print(X_train)
```

```
[[0.0 0.0 1.0 38.77777777777778 52000.0]
 [0.0 1.0 0.0 40.0 63777.77777777778]
 [1.0 0.0 0.0 44.0 72000.0]
 [0.0 0.0 1.0 38.0 61000.0]
 [0.0 0.0 1.0 27.0 48000.0]
 [1.0 0.0 0.0 48.0 79000.0]
 [0.0 1.0 0.0 50.0 83000.0]
 [1.0 0.0 0.0 35.0 58000.0]]
```

```
[26] print(X_test)
```

```
[[0.0 1.0 0.0 30.0 54000.0]
 [1.0 0.0 0.0 37.0 67000.0]]
```

```
[27] print(y_train)
```

```
[0 1 0 0 1 1 0 1]
```

```
[28] print(y_test)
```

```
[0 1]
```

Feature Scaling

Feature scaling is a technique used in data preprocessing for machine learning to standardize or normalize the range of independent variables (features) in a dataset. It ensures that all features contribute equally to the model and prevents certain features from dominating others due to differences in scale.

Why is Feature Scaling Important? *(although it is not always implemented)*

1. Improves Model Performance – Many machine learning algorithms, such as gradient descent-based models (e.g., linear regression, logistic regression, neural networks), perform better when features are on a similar scale. This helps in faster convergence and better optimization.
2. Enhances Distance-Based Models – Algorithms like k-NN (k-Nearest Neighbors), SVM (Support Vector Machine), and K-Means clustering rely on distance measurements (e.g., Euclidean distance). If features have different scales, the larger values will dominate the distance calculation, leading to biased results.
3. Prevents Numerical Instability – In some models, especially those using regularization (like Lasso and Ridge regression), unscaled features can lead to instability in the computations and poor performance.
4. Ensures Fair Weight Distribution – In algorithms that assign weights to features (e.g., logistic regression, neural networks), scaling ensures that all features contribute proportionally, preventing one feature from disproportionately influencing the model.

Feature Scaling

Normalization

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

[0 ; 1]

Standardization

$$X' = \frac{X - \mu}{\sigma}$$

[-3 ; +3]

Implementation

Feature Scaling

```
[72] # Standarisation
     from sklearn.preprocessing import StandardScaler

     sc = StandardScaler()
     X_train[:,3:] = sc.fit_transform(X_train[:,3:])
     X_test[:,3:] = sc.transform(X_test[:,3:])
```

```
[73] print(X_train)
```

```
[[0.0 0.0 1.0 -0.19159184384578545 -1.0781259408412425]
 [0.0 1.0 0.0 -0.014117293757057777 -0.07013167641635372]
 [1.0 0.0 0.0 0.566708506533324 0.633562432710455]
 [0.0 0.0 1.0 -0.30453019390224867 -0.30786617274297867]
 [0.0 0.0 1.0 -1.9018011447007988 -1.420463615551582]
 [1.0 0.0 0.0 1.1475343068237058 1.232653363453549]
 [0.0 1.0 0.0 1.4379472069688968 1.5749910381638885]
 [1.0 0.0 0.0 -0.7401495441200351 -0.5646194287757332]]
```

```
[74] print(X_test)
```

```
[[0.0 1.0 0.0 -1.4661817944830124 -0.9069571034860727]
 [1.0 0.0 0.0 -0.44973664397484414 0.2056403393225306]]
```

We do not apply standardization to the dummy variables, because it will cause a loss of interpretation.
Apart from that, our dummy data is only 0 and 1, where these numbers are already in the range of -3 and +3.

Note:

- **fit_transform() is used on the training data (X_train) because we need to calculate the mean and standard deviation first.**
- **transform() is used on the testing data (X_test) to ensure the scale remains consistent with the training data.**

Source Code & Datasets

github.com/nadhif-royal/DataPreprocessing

Thank You

github.com/nadhif-royal

linkedin.com/in/royalnadhif50/

instagram.com/royal_nadhif/