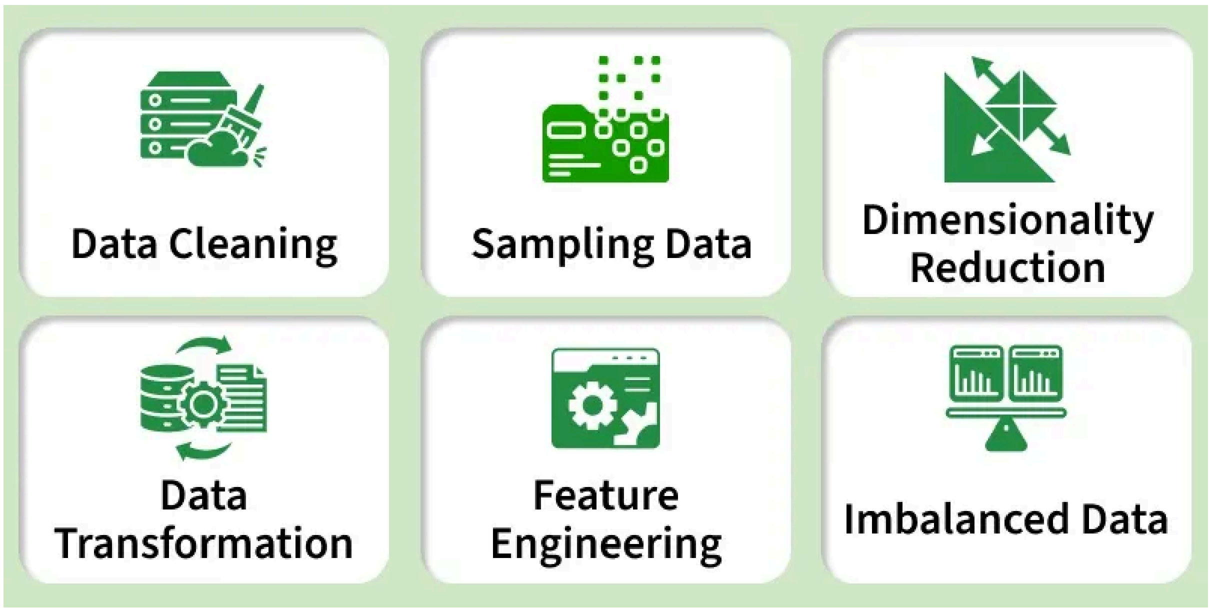
Data Preprocessing in Python

Data preprocessing is the first step in any data analysis or machine learning pipeline. It involves cleaning, transforming and organizing raw data to ensure it is accurate, consistent and ready for modeling. It has a big impact on model building such as:

- Clean and well-structured data allows models to learn meaningful patterns rather than noise.
- Properly processed data prevents misleading inputs, leading to more reliable predictions.
- Organized data makes it simpler to create useful inputs for the model, enhancing model performance.
- Organized data supports better Exploratory Data Analysis (EDA), making patterns and trends more interpretable.



-

Steps-by-Step implementation

Let's implement various preprocessing features,

Step 1: Import Libraries and Load Dataset

We prepare the environment with libraries liike pandas, numpy, scikit learn, matplotlib and seaborn for data manipulation, numerical operations, visualization and scaling. Load the dataset for preprocessing.

The sample dataset can be downloaded from here.

import pandas as pd

import numpy as np

from sklearn.preprocessing import MinMaxScaler, StandardScaler

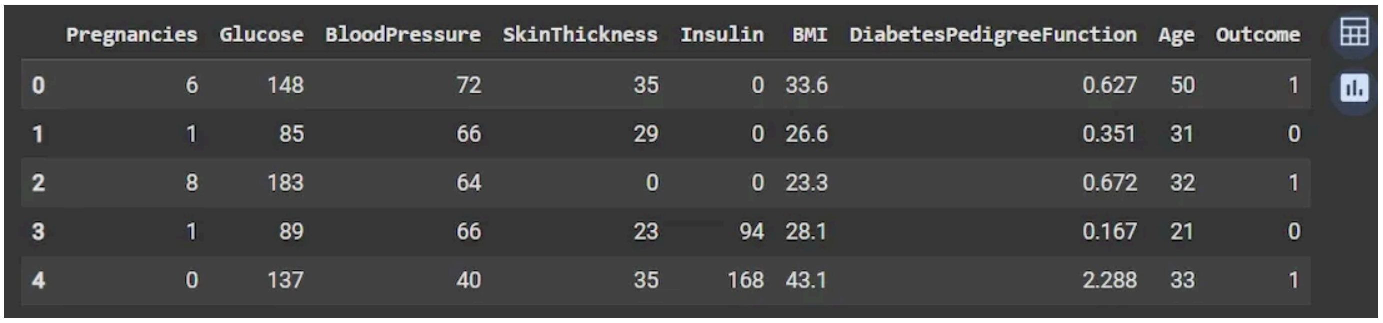
import seaborn as sns

import matplotlib.pyplot as plt

df = pd.read_csv('Geeksforgeeks/Data/diabetes.csv')

df.head()

Output:



	Pregnancies	Glucose	BloodPressure	SkinThickness	Insulin	BMI	DiabetesPedigreeFunction	Age	Outcome
0	6	148	72	35	0	33.6	0.627	50	1
1	1	85	66	29	0	26.6	0.351	31	0
2	8	183	64	0	0	23.3	0.672	32	1
3	1	89	66	23	94	28.1	0.167	21	0
4	0	137	40	35	168	43.1	2.288	33	1

Step 2: Inspect Data Structure and Check Missing Values

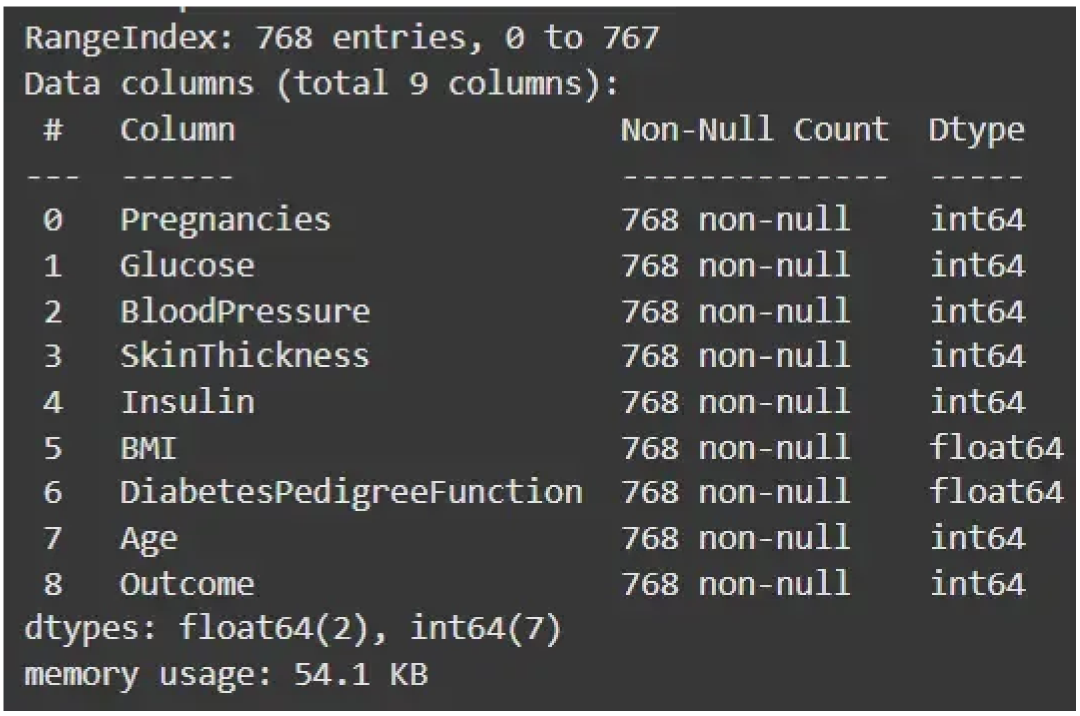
We understand dataset size, data types and identify any incomplete (missing) data that needs handling.

- **df.info():** Prints concise summary including count of non-null entries and data type of each column.
- **df.isnull().sum():** Returns the number of missing values per column.

df.info()

print(df.isnull().sum())

Output:



```
RangeIndex: 768 entries, 0 to 767
Data columns (total 9 columns):
 #   Column                    Non-Null Count  Dtype  
---  ------                    --------------  -----  
 0   Pregnancies               768 non-null    int64  
 1   Glucose                   768 non-null    int64  
 2   BloodPressure             768 non-null    int64  
 3   SkinThickness             768 non-null    int64  
 4   Insulin                   768 non-null    int64  
 5   BMI                       768 non-null    float64
 6   DiabetesPedigreeFunction  768 non-null    float64
 7   Age                       768 non-null    int64  
 8   Outcome                   768 non-null    int64  
dtypes: float64(2), int64(7)
memory usage: 54.1 KB
```

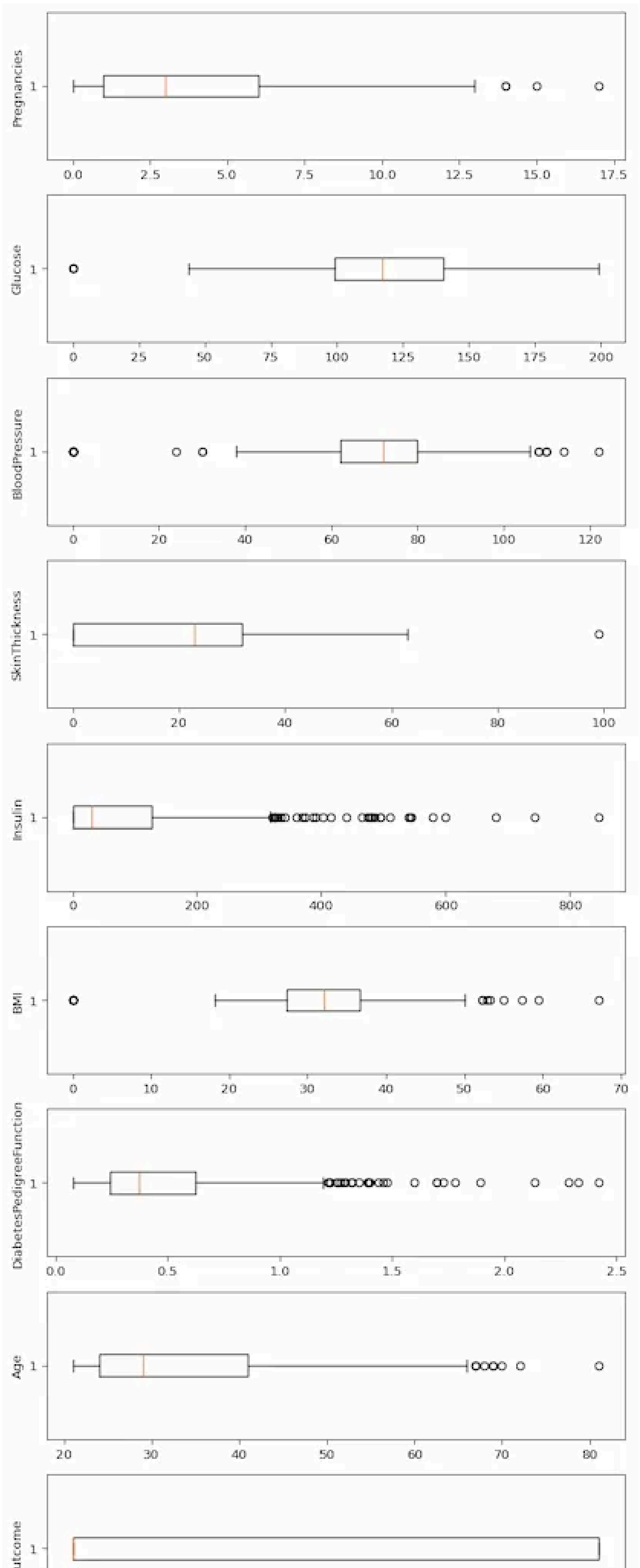
Step 3: Statistical Summary and Visualizing Outliers

Get numeric summaries like mean, median, min/max and detect unusual points (outliers). Outliers can skew models if not handled.

- **df.describe():** Computes count, mean, std deviation, min/max and quartiles for numerical columns.
- **Boxplots:** Visualize spread and detect outliers using matplotlib's boxplot().

```
df.describe()

fig, axs = plt.subplots(len(df.columns), 1, figsize=(7, 18), dpi=95)
for i, col in enumerate(df.columns):
    axs[i].boxplot(df[col], vert=False)
    axs[i].set_ylabel(col)
plt.tight_layout()
plt.show()
```

Output:


Pregnancies
1
0.0 2.5 5.0 7.5 10.0 12.5 15.0 17.5
Glucose
1
0 25 50 75 100 125 150 175 200
BloodPressure
1
0 20 40 60 80 100 120
SkinThickness
1
0 20 40 60 80 100
Insulin
1
0 200 400 600 800
BMI
1
0 10 20 30 40 50 60 70
DiabetesPedigreeFunction
1
0.0 0.5 1.0 1.5 2.0 2.5
Age
1
20 30 40 50 60 70 80
utcome
1

Boxplot

Step 4: Remove Outliers Using the Interquartile Range (IQR) Method

Remove extreme values beyond a reasonable range to improve model robustness.

- IQR = Q3 (75th percentile) – Q1 (25th percentile).
- Values below Q1 - 1.5IQR or above Q3 + 1.5IQR are outliers.
- Calculate lower and upper bounds for each column separately.
- Filter data points to keep only those within bounds.

```
q1, q3 = np.percentile(df['Insulin'], [25, 75])

iqr = q3 - q1

lower = q1 - 1.5 * iqr

upper = q3 + 1.5 * iqr

clean_df = df[(df['Insulin'] >= lower) & (df['Insulin'] <= upper)]
```

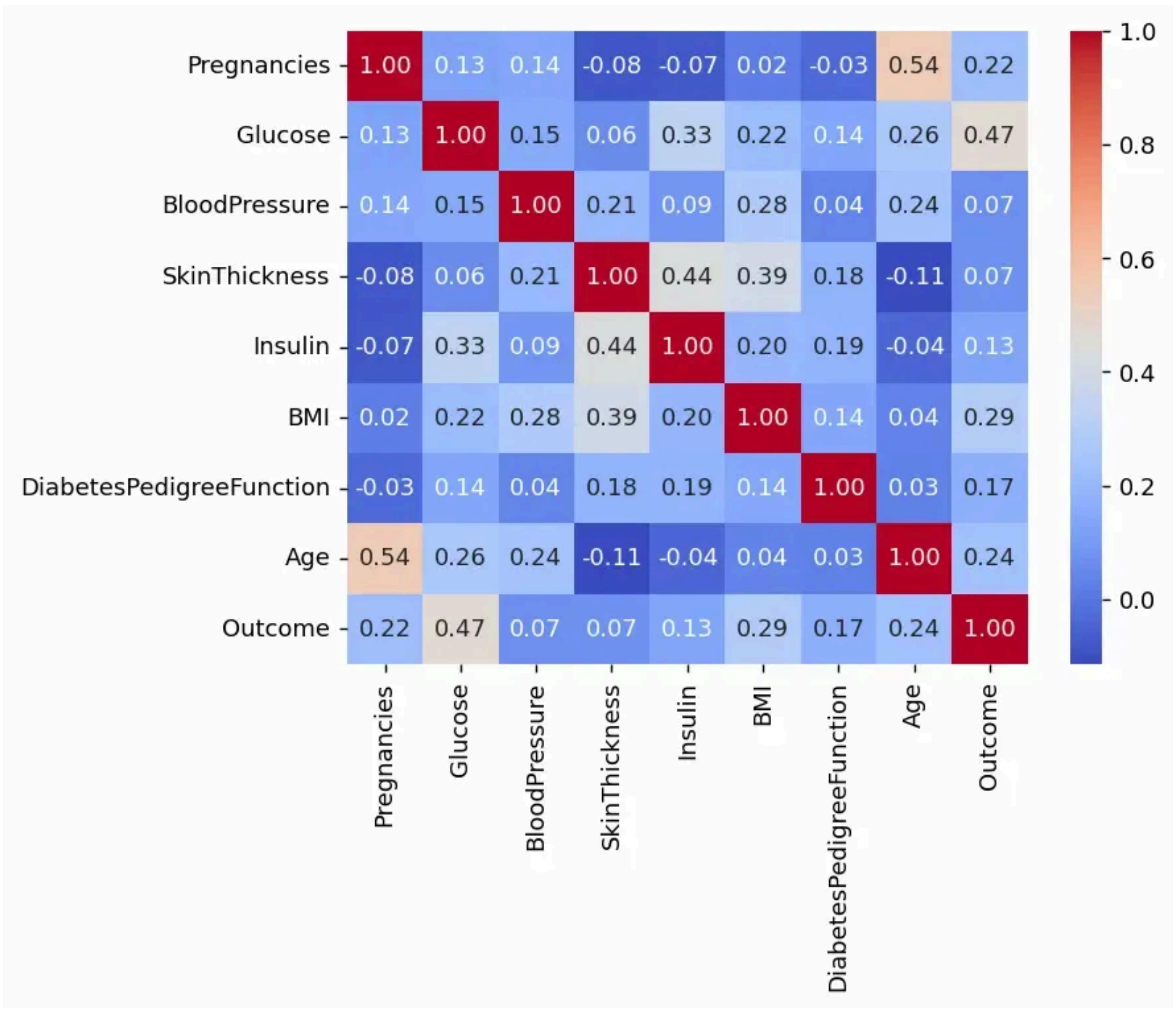
Step 5: Correlation Analysis

Understand relationships between features and the target variable (Outcome). Correlation helps gauge feature importance.

- **df.corr():** Computes pairwise correlation coefficients between columns.
- Heatmap via seaborn visualizes correlation matrix clearly.
- Sorting correlations with corr['Outcome'].sort_values() highlights features most correlated with the target.

```
corr = df.corr()

plt.figure(dpi=130)

sns.heatmap(corr, annot=True, fmt='.2f', cmap='coolwarm')

plt.show()


print(corr['Outcome'].sort_values(ascending=False))
```

Output:



	Pregnancies	Glucose	BloodPressure	SkinThickness	Insulin	BMI	DiabetesPedigreeFunction	Age	Outcome
Pregnancies	1.00	0.13	0.14	-0.08	-0.07	0.02	-0.03	0.54	0.22
Glucose	0.13	1.00	0.15	0.06	0.33	0.22	0.14	0.26	0.47
BloodPressure	0.14	0.15	1.00	0.21	0.09	0.28	0.04	0.24	0.07
SkinThickness	-0.08	0.06	0.21	1.00	0.44	0.39	0.18	-0.11	0.07
Insulin	-0.07	0.33	0.09	0.44	1.00	0.20	0.19	-0.04	0.13
BMI	0.02	0.22	0.28	0.39	0.20	1.00	0.14	0.04	0.29
DiabetesPedigreeFunction	-0.03	0.14	0.04	0.18	0.19	0.14	1.00	0.03	0.17
Age	0.54	0.26	0.24	-0.11	-0.04	0.04	0.03	1.00	0.24
Outcome	0.22	0.47	0.07	0.07	0.13	0.29	0.17	0.24	1.00

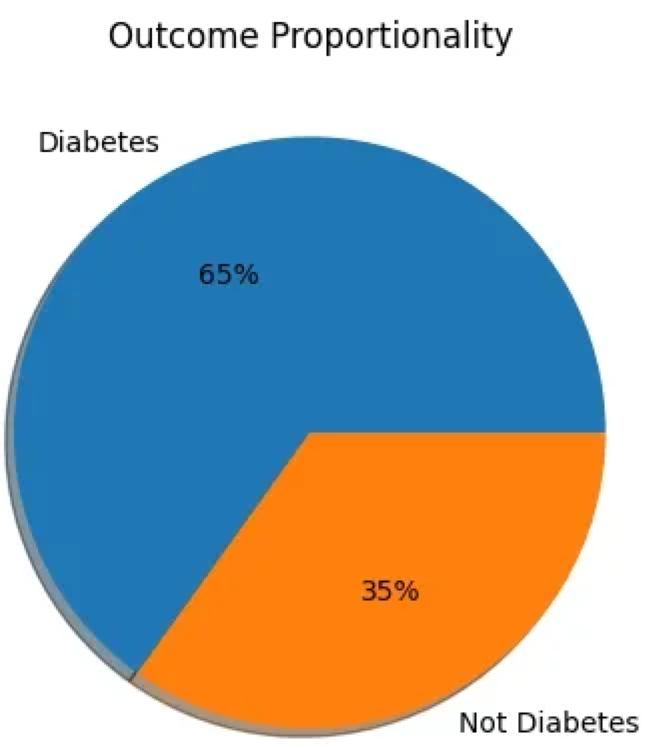
Step 6: Visualize Target Variable Distribution

Check if target classes (Diabetes vs Not Diabetes) are balanced, affecting model training and evaluation.

- **plt.pie():** Pie chart to display proportion of each class in the target variable 'Outcome'.

```
plt.pie(df['Outcome'].value_counts(), labels=[

    'Diabetes', 'Not Diabetes'], autopct='%.f%%', shadow=True)

plt.title('Outcome Proportionality')

plt.show()
```

Output:



Result

Step 7: Separate Features and Target Variable

Prepare independent variables (features) and dependent variable (target) separately for modeling.

- **df.drop(columns=[...]):** Drops the target column from features.
- Direct column selection df['Outcome'] selects target column.

```
X = df.drop(columns=['Outcome'])
```

```
y = df['Outcome']
```

Step 8: Feature Scaling: Normalization and Standardization

Scale features to a common range or distribution, important for many ML algorithms sensitive to feature magnitudes.

1. Normalization (Min-Max Scaling): Rescales features between 0 and 1. Good for algorithms like k-NN and neural networks.

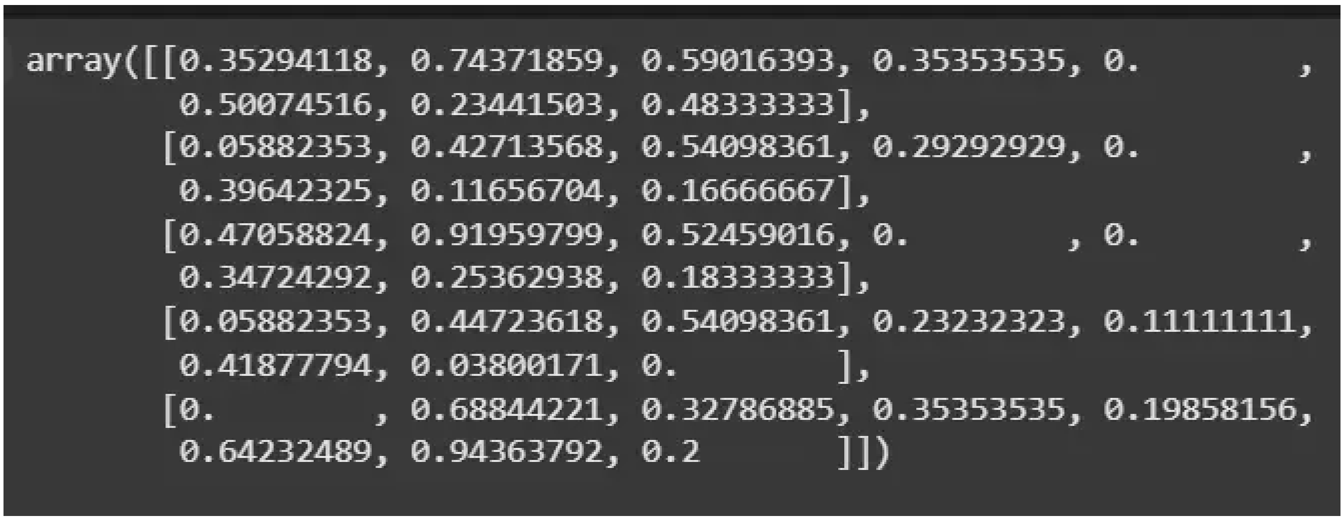
- **Class:** MinMaxScaler from sklearn.
- **.fit_transform():** Learns min/max from data and applies scaling.

scaler = MinMaxScaler()

X_normalized = scaler.fit_transform(X)

print(X_normalized[:5])

Output:



```
array([[0.35294118, 0.74371859, 0.59016393, 0.35353535, 0.        ,
        0.50074516, 0.23441503, 0.48333333],
       [0.05882353, 0.42713568, 0.54098361, 0.29292929, 0.        ,
        0.39642325, 0.11656704, 0.16666667],
       [0.47058824, 0.91959799, 0.52459016, 0.        , 0.        ,
        0.34724292, 0.25362938, 0.18333333],
       [0.05882353, 0.44723618, 0.54098361, 0.23232323, 0.11111111,
        0.41877794, 0.03800171, 0.        ],
       [0.        , 0.68844221, 0.32786885, 0.35353535, 0.19858156,
        0.64232489, 0.94363792, 0.2       ]])
```

Normalization

2. Standardization: Transforms features to have mean = 0 and standard deviation = 1, useful for normally distributed features.

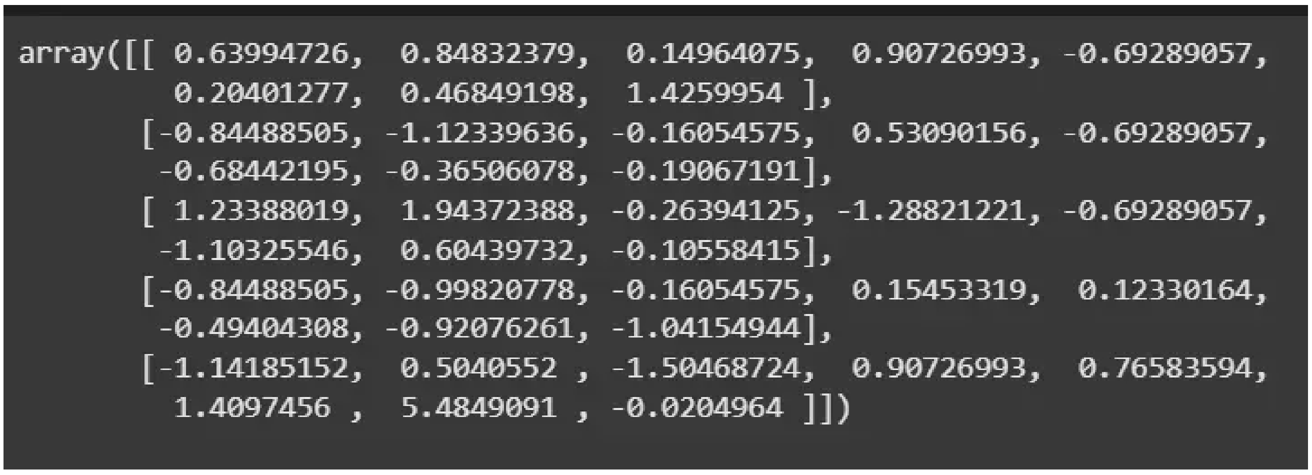
- **Class:** StandardScaler from sklearn.

scaler = StandardScaler()

X_standardized = scaler.fit_transform(X)

print(X_standardized[:5])

Output:



```
array([[ 0.63994726,  0.84832379,  0.14964075,  0.90726993, -0.69289057,
         0.20401277,  0.46849198,  1.4259954 ],
       [-0.84488505, -1.12339636, -0.16054575,  0.53090156, -0.69289057,
        -0.68442195, -0.36506078, -0.19067191],
       [ 1.23388019,  1.94372388, -0.26394125, -1.28821221, -0.69289057,
        -1.10325546,  0.60439732, -0.10558415],
       [-0.84488505, -0.99820778, -0.16054575,  0.15453319,  0.12330164,
        -0.49404308, -0.92076261, -1.04154944],
       [-1.14185152,  0.5040552 , -1.50468724,  0.90726993,  0.76583594,
         1.4097456 ,  5.4849091 , -0.0204964 ]])
```

Standardization

Advantages

Let's see the advantages of data preprocessing,

- **Improves Data Quality:** Cleans and organizes raw data for better analysis.
- **Enhances Model Accuracy:** Removes noise and irrelevant data, leading to more precise predictions.
- **Reduces Overfitting:** Handles outliers and redundant features, improving model generalization.
- **Speeds Up Training:** Efficiently scaled data reduces computation time.
- **Ensures Algorithm Compatibility:** Converts data into formats suitable for machine learning models.