Imputation Methods

- Replacing missing values with estimated values.
- Preserves sample size: Doesn't reduce data points.
- Can introduce bias: Estimated values might not be accurate.

Here are some common imputation methods:

1- Mean, Median, and Mode Imputation:

- Replace missing values with the mean, median, or mode of the relevant variable.
- Simple and efficient: Easy to implement.
- Can be inaccurate: Doesn't consider the relationships between variables.

In this example, we are explaining the imputation techniques for handling missing values in the 'Marks' column of the DataFrame (df). It calculates and fills missing values with the mean, median, and mode of the existing values in that column, and then prints the results for observation.

1. **Mean Imputation:** Calculates the mean of the 'Marks' column in the DataFrame (df).
    - df['Marks'].fillna(...): Fills missing values in the 'Marks' column with the mean value.
    - mean_imputation: The result is stored in the variable mean_imputation.
2. **Median Imputation:** Calculates the median of the 'Marks' column in the DataFrame (df).
    - df['Marks'].fillna(...): Fills missing values in the 'Marks' column with the median value.
    - median_imputation: The result is stored in the variable median_imputation.
3. **Mode Imputation:** Calculates the mode of the 'Marks' column in the DataFrame (df). The result is a Series.
    - .iloc[0]: Accesses the first element of the Series, which represents the mode.
    - df['Marks'].fillna(...): Fills missing values in the 'Marks' column with the mode value.

```
#  Mean, Median, and Mode Imputation

mean_imputation = df['Marks'].fillna(df['Marks'].mean())
```

```
median_imputation = df['Marks'].fillna(df['Marks'].median())

mode_imputation = df['Marks'].fillna(df['Marks'].mode().iloc[0])


print("\nImputation using Mean:")

print(mean_imputation)


print("\nImputation using Median:")

print(median_imputation)


print("\nImputation using Mode:")

print(mode_imputation)
```

Output:

Imputation using Mean:

```
0    85.000000
1    92.000000
2    78.000000
3    89.000000
4    86.714286
5    95.000000
6    80.000000
7    88.000000
Name: Marks, dtype: float64
```

Imputation using Median:

```
0    85.0
1    92.0
2    78.0
3    89.0
4    88.0
5    95.0
6    80.0
7    88.0
```

```
Name: Marks, dtype: float64
```

Imputation using Mode:

```
0    85.0
1    92.0
2    78.0
3    89.0
4    78.0
5    95.0
6    80.0
7    88.0
Name: Marks, dtype: float64
```

2. Forward and Backward Fill

- Replace missing values with the previous or next non-missing value in the same variable.
- Simple and intuitive: Preserves temporal order.
- Can be inaccurate: Assumes missing values are close to observed values

These fill methods are particularly useful when there is a logical sequence or order in the data, and missing values can be reasonably assumed to follow a pattern. The method parameter in fillna() allows to specify the filling strategy, and here, it's set to 'ffill' for forward fill and 'bfill' for backward fill.

1. **Forward Fill (forward_fill)**
    - df['Marks'].fillna(method='ffill'): This method fills missing values in the 'Marks' column of the DataFrame (df) using a forward fill strategy. It replaces missing values with the last observed non-missing value in the column.
    - forward_fill: The result is stored in the variable forward_fill.
2. **Backward Fill (backward_fill)**
    - df['Marks'].fillna(method='bfill'): This method fills missing values in the 'Marks' column using a backward fill strategy. It replaces missing values with the next observed non-missing value in the column.
    - backward_fill: The result is stored in the variable backward_fill.

```
# Forward and Backward Fill

forward_fill = df['Marks'].fillna(method='ffill')
```

```
backward_fill = df['Marks'].fillna(method='bfill')


print("\nForward Fill:")

print(forward_fill)


print("\nBackward Fill:")

print(backward_fill)
```

Output:

Forward Fill:

```
0    85.0
1    92.0
2    78.0
3    89.0
4    89.0
5    95.0
6    80.0
7    88.0
Name: Marks, dtype: float64
```

Backward Fill:

```
0    85.0
1    92.0
2    78.0
3    89.0
4    95.0
5    95.0
6    80.0
7    88.0
Name: Marks, dtype: float64
```

Note

- Forward fill uses the last valid observation to fill missing values.
- Backward fill uses the next valid observation to fill missing values.

3. Interpolation Techniques

- Estimate missing values based on surrounding data points using techniques like linear interpolation or spline interpolation.
- More sophisticated than mean/median imputation: Captures relationships between variables.
- Requires additional libraries and computational resources.

These interpolation techniques are useful when the relationship between data points can be reasonably assumed to follow a linear or quadratic pattern. The method parameter in the interpolate() method allows to specify the interpolation strategy.

1. **Linear Interpolation**
    - df['Marks'].interpolate(method='linear'): This method performs linear interpolation on the 'Marks' column of the DataFrame (df). Linear interpolation estimates missing values by considering a straight line between two adjacent non-missing values.
    - linear_interpolation: The result is stored in the variable linear_interpolation.
2. **Quadratic Interpolation**
    - df['Marks'].interpolate(method='quadratic'): This method performs quadratic interpolation on the 'Marks' column. Quadratic interpolation estimates missing values by considering a quadratic curve that passes through three adjacent non-missing values.
    - quadratic_interpolation: The result is stored in the variable quadratic_interpolation.

```
#  Interpolation Techniques

linear_interpolation = df['Marks'].interpolate(method='linear')

quadratic_interpolation = df['Marks'].interpolate(method='quadratic')


print("\nLinear Interpolation:")

print(linear_interpolation)
print("\nQuadratic Interpolation:")

print(quadratic_interpolation)
```

Output:

```
Linear Interpolation:
0    85.0
1    92.0
2    78.0
3    89.0
4    92.0
5    95.0
6    80.0
7    88.0
Name: Marks, dtype: float64

Quadratic Interpolation:
0    85.00000
1    92.00000
2    78.00000
3    89.00000
4    98.28024
5    95.00000
6    80.00000
7    88.00000
Name: Marks, dtype: float64
```

Note:

- Linear interpolation assumes a straight line between two adjacent non-missing values.
- Quadratic interpolation assumes a quadratic curve that passes through three adjacent non-missing values.

Choosing the right strategy depends on several factors:

- Type of missing data: MCAR, MAR, or MNAR.
- Proportion of missing values.
- Data type and distribution.
- Analytical goals and assumptions.

Impact of Handling Missing Values

Missing values are a common occurrence in real-world data, negatively impacting data analysis and modeling if not addressed properly. Handling missing values effectively is crucial to ensure the accuracy and reliability of your findings.

Here are some key impacts of handling missing values:

1. **Improved data quality:** Addressing missing values enhances the overall quality of the dataset. A cleaner dataset with fewer missing values is more reliable for analysis and model training.
2. **Enhanced model performance:** Machine learning algorithms often struggle with missing data, leading to biased and unreliable results. By appropriately handling missing values, models can be trained on a more complete dataset, leading to improved performance and accuracy.
3. **Preservation of Data Integrity**: Handling missing values helps maintain the integrity of the dataset. Imputing or removing missing values ensures that the dataset remains consistent and suitable for analysis.
4. **Reduced bias:** Ignoring missing values may introduce bias in the analysis or modeling process. Handling missing data allows for a more unbiased representation of the underlying patterns in the data.
5. Descriptive statistics, such as means, medians, and standard deviations, can be more accurate when missing values are appropriately handled. This ensures a more reliable summary of the dataset.
6. **Increased efficiency:** Efficiently handling missing values can save you time and effort during data analysis and modeling.