Missing value imputation techniques are categorized based on the method used and the nature of the data

1. **Simple Imputation Methods**
   * **Mean Imputation**
   * **Median Imputation**
   * **Mode Imputation**
   * **Constant Imputation**

* **Mean Imputation**

Replace missing values with the mean of the column (numerical data)

**Example**:

import pandas as pd

data = {'Age': [25, 30, None, 35, 40]}

df = pd.DataFrame(data)

df['Age'].fillna(df['Age'].mean(), inplace=True)

print(df)

* **Median Imputation**

Replace missing values with the median.

**Example**:

df['Age'].fillna(df['Age'].median(), inplace=True)

* **Mode Imputation**

Replace missing values with the mode (most frequent value).

df['Age'].fillna(df['Age'].median(), inplace=True)

**Example**:

df['Age'].fillna(df['Age'].mode()[0], inplace=True)

* **Constant Value Imputation**

Replace missing values with a fixed constant.

df['Age'].fillna(0, inplace=True)

**2.Advanced Statistical Methods**

* K-Nearest Neighbors (KNN) Imputation
* Regression Imputation
* Multiple Imputation
* **K-Nearest Neighbors (KNN) Imputation**

Use the values of the nearest neighbors.

Let's assume we have a dataset with missing values in the age and salary columns.

import pandas as pd

import numpy as np

from sklearn.impute import KNNImputer

# Create a sample dataset

data = {

'age': [25, np.nan, 35, 45, np.nan, 55],

'salary': [50000, 60000, np.nan, 80000, 90000, 100000],

'department': ['HR', 'Engineering', 'HR', 'HR', 'Marketing', 'Engineering']

}

df = pd.DataFrame(data)

# Instantiate the KNNImputer

imputer = KNNImputer(n\_neighbors=2)

# Fit and transform the data

df\_imputed = pd.DataFrame(imputer.fit\_transform(df[['age', 'salary']]), columns=['age', 'salary'])

# Combine imputed data with original categorical data

df\_imputed['department'] = df['department']

print(df\_imputed)

* **Create a Sample Dataset**:
* We have missing values in age and salary columns.
* The department column is categorical and remains unchanged in this example.
* **Instantiate the KNNImputer**:

n\_neighbors=2: We use 2 nearest neighbors for imputation.

* **Fit and Transform the Data**:
* imputer.fit\_transform(df[['age', 'salary']]): The KNNImputer fills in the missing values based on the nearest neighbors.
* **Combine Imputed Data with Original Data**:
* The imputed values are combined with the original categorical data to form the final DataFrame.
* **Regression Imputation**

Use a regression model to predict missing values.

**Example**

from sklearn.linear\_model import LinearRegression

import pandas as pd

data = {'Feature1': [1, 2, 3, 4], 'Feature2': [10, 20, None, 40]}

df = pd.DataFrame(data)

train = df.dropna()

model = LinearRegression()

model.fit(train[['Feature1']], train['Feature2'])

df.loc[df['Feature2'].isnull(), 'Feature2'] = model.predict(df[df['Feature2'].isnull()][['Feature1']])

Step-by-Step Process:

Step 1: Import Libraries

Step 2:Create a Sample Dataset

Step 3: Separate the Data

Step 4: Train the Regression Model

Step 5: Predict Missing Values

Step 6: Replace Missing Values

* **Multiple Imputation**

Generate multiple imputations and combine results.

from sklearn.experimental import enable\_iterative\_imputer

from sklearn.impute import IterativeImputer

imputer = IterativeImputer(max\_iter=10, random\_state=0)

data = [[1, 2, None], [3, None, 5], [None, 6, 7]]

print(imputer.fit\_transform(data))

* **Probabilistic Methods**

**Expectation-Maximization (EM) Algorithm**

Iteratively estimates missing values based on probability distributions.

We use the fancyimpute library, which implements EM imputation.

import numpy as np

from fancyimpute import IterativeImputer

# Sample data

data = np.array([

[1, 2, np.nan],

[3, np.nan, 5],

[np.nan, 6, 7],

[8, 10, 12]

])

# Initialize the EM-based imputer

imputer = IterativeImputer(max\_iter=10, random\_state=0)

# Perform imputation

imputed\_data = imputer.fit\_transform(data)

print("Original Data:")

print(data)

print("\nImputed Data:")

print(imputed\_data)

**4. Machine Learning-Based Methods**

Random Forest Imputation

Random Forest Imputation uses a random forest algorithm to predict missing values. It works as follows:

1. For each feature with missing values:
   * Treat it as the target variable.
   * Use the other features (without missing values) as predictors.
2. A random forest model is trained to predict the missing values based on observed data.
3. The predictions are used to fill in the missing values.

import pandas as pd

import numpy as np

from sklearn.ensemble import RandomForestRegressor

from sklearn.impute import SimpleImputer

# Helper function to perform random forest imputation

def random\_forest\_impute(df, target\_column):

# Separate the dataset into features (X) and target (y)

X = df.drop(columns=[target\_column])

y = df[target\_column]

# Separate the rows with and without missing values

X\_missing = X[y.isna()]

X\_complete = X[~y.isna()]

y\_complete = y.dropna()

# Train the Random Forest Regressor on the complete data

rf = RandomForestRegressor(n\_estimators=100, random\_state=42)

rf.fit(X\_complete, y\_complete)

# Predict the missing values

y\_missing = rf.predict(X\_missing)

# Fill in the missing values

df.loc[df[target\_column].isna(), target\_column] = y\_missing

return df

# Create a sample dataset with missing values

data = {

'feature1': [1.2, 2.3, 3.4, np.nan, 5.6, 6.7, 7.8, 8.9],

'feature2': [2.1, np.nan, 4.5, 5.6, 6.7, 7.8, 8.9, 9.0],

'feature3': [3.1, 3.4, 4.6, 5.7, 6.8, 7.9, 8.1, np.nan]

}

df = pd.DataFrame(data)

print("Original Data:")

print(df)

# Perform Random Forest Imputation on feature1

df\_imputed = random\_forest\_impute(df, 'feature1')

print("Imputed Data (feature1):")

print(df\_imputed)

# Perform Random Forest Imputation on feature2

df\_imputed = random\_forest\_impute(df\_imputed, 'feature2')

print("Imputed Data (feature2):")

print(df\_imputed)

# Perform Random Forest Imputation on feature3

df\_imputed = random\_forest\_impute(df\_imputed, 'feature3')

print("Imputed Data (feature3):")

print(df\_imputed)

import pandas as pd

import numpy as np

from sklearn.ensemble import RandomForestRegressor

# Sample data

data = {

'Feature1': [1, 2, np.nan, 4, 5],

'Feature2': [2, 4, 6, 8, 10],

'Feature3': [np.nan, 1, 3, 4, 5]

}

df = pd.DataFrame(data)

print("Original Data:")

print(df)

# Function for random forest imputation

def random\_forest\_impute(df):

for column in df.columns:

if df[column].isnull().sum() > 0:

# Separate rows with and without missing values

train = df[df[column].notnull()]

test = df[df[column].isnull()]

# Use other columns as features to predict the missing column

X\_train = train.drop(column, axis=1)

y\_train = train[column]

X\_test = test.drop(column, axis=1)

# Train random forest regressor

model = RandomForestRegressor(n\_estimators=100, random\_state=42)

model.fit(X\_train, y\_train)

# Predict missing values

df.loc[df[column].isnull(), column] = model.predict(X\_test)

return df

# Perform imputation

imputed\_df = random\_forest\_impute(df)

print("\nImputed Data:")

print(imputed\_df)

**Gradient Boosting Imputation (e.g., XGBoost, LightGBM)**: Train gradient boosting models to predict missing values.

**Neural Network Imputation**: Use deep learning models (e.g., autoencoders) for complex imputation tasks.

import pandas as pd

import numpy as np

from sklearn.ensemble import GradientBoostingRegressor

# Sample dataset with missing values

data = {

'Feature1': [10, 20, np.nan, 40, 50],

'Feature2': [5, np.nan, 15, 20, 25],

'Feature3': [np.nan, 1, 3, 4, 5]

}

df = pd.DataFrame(data)

print("Original Data:")

print(df)

# Function for gradient boosting imputation

def gradient\_boosting\_impute(df):

for column in df.columns:

if df[column].isnull().sum() > 0:

# Separate rows with and without missing values

train = df[df[column].notnull()]

test = df[df[column].isnull()]

# Use other columns as predictors

X\_train = train.drop(column, axis=1)

y\_train = train[column]

X\_test = test.drop(column, axis=1)

# Train Gradient Boosting Regressor

model = GradientBoostingRegressor(n\_estimators=100, random\_state=42)

model.fit(X\_train, y\_train)

# Predict missing values

df.loc[df[column].isnull(), column] = model.predict(X\_test)

return df

# Perform imputation

imputed\_df = gradient\_boosting\_impute(df)

print("\nImputed Data:")

print(imputed\_df)

**Gradient Boosting Imputation Logic**:

* For each column with missing values:
  + Rows with missing values are used as the **test set**.
  + Remaining rows are used as the **training set**.
* A GradientBoostingRegressor model is trained to predict the missing values using the other columns.

**Model Training**:

* Gradient boosting uses an ensemble of decision trees to model complex relationships.

**Neural Network Imptutaion**

import numpy as npimport pandas as pd

from sklearn.model\_selection import train\_test\_split

from tensorflow.keras.models import Model

from tensorflow.keras.layers import Input, Dense

# Sample dataset with missing values

data = {

'Feature1': [1, 2, np.nan, 4, 5],

'Feature2': [np.nan, 2, 3, 4, 5],

'Feature3': [5, 4, 3, np.nan, 1]

}

df = pd.DataFrame(data)

print("Original Data:")

print(df)

# Preprocessing: Replace missing values with the column mean (to initialize)

df\_filled = df.fillna(df.mean())

# Normalize data (optional, helps neural networks perform better)

df\_normalized = (df\_filled - df\_filled.mean()) / df\_filled.std()

# Train-test split

X\_train, X\_test = train\_test\_split(df\_normalized.values, test\_size=0.2, random\_state=42)

# Define autoencoder

input\_dim = X\_train.shape[1]

input\_layer = Input(shape=(input\_dim,))

encoded = Dense(64, activation='relu')(input\_layer)

encoded = Dense(32, activation='relu')(encoded)

decoded = Dense(64, activation='relu')(encoded)

decoded = Dense(input\_dim, activation='linear')(decoded)

autoencoder = Model(inputs=input\_layer, outputs=decoded)

autoencoder.compile(optimizer='adam', loss='mean\_squared\_error')

# Train the autoencoder

autoencoder.fit(X\_train, X\_train, epochs=100, batch\_size=8, shuffle=True, validation\_data=(X\_test, X\_test), verbose=0)

# Impute missing values

predicted\_data = autoencoder.predict(df\_normalized.values)

imputed\_data = pd.DataFrame(predicted\_data, columns=df.columns)

# Denormalize data

imputed\_data = (imputed\_data \* df\_filled.std()) + df\_filled.mean()

print("\nImputed Data:")

print(imputed\_data)

**MISSING VALUES**

In order to get the count of missing values in each column,we use the in-built function.isnull().sum()

* obtain the total missing values for each variable
* 'isnull().sum()' returns the number of missing values in each variable
* sort the variables on the basis of total null values in the variable using sort\_values()
* 'ascending = False' sorts values in the descending order
* the variable with highest number of missing values will appear first

Total = df\_insurance.isnull().sum().sort\_values(ascending=False)

* 'isnull().sum()' returns the number of missing values in each variable
* 'isnull().count()' returns the count of the data, i.e. count of outcomes 'True' and 'False' of isnull()
* sort the variables on the basis of total null values in the variable using sort\_values()
* 'ascending = False' sorts values in the descending order
* the variable with highest percentage of missing values will appear first

Percent =(df\_insurance.isnull().sum()\*100/df\_insurance.isnull().count()).sort\_values(ascending=False)

missing\_data = pd.concat([Total, Percent], axis = 1, keys = ['Total', 'Percentage of Missing Values'])

kkkkkkk