codeHandleMissingValues

July 30, 2024

1.1: Introduction to Missing Data

Objective: Understand the different types of missing data and their implications for data analysis.

Research and Understand the Different Types of Missing Data:

1. Missing Completely at Random (MCAR): Definition: The likelihood of a value being missing is independent of both observed and unobserved data. Missingness is purely random. Scenario: You have a dataset from an online quiz where some answers are missing randomly because of user errors or technical issues. For example, in a dataset of quiz responses, some answers are missing because users accidentally skipped questions. Example: A survey where some participants accidentally skip a question, and this skipping is unrelated to their answers on other questions. For instance, a survey on customer satisfaction where a few respondents accidentally leave the "satisfaction with customer service" question blank, regardless of their overall satisfaction.

2. Missing at Random (MAR): Definition: The probability of a value being missing is related to other observed variables but not to the missing value itself. Scenario: In a dataset of patient health records, older patients are less likely to report their income. The missing income data is related to the patient's age but not to the income value itself. Example: In a dataset of patient health records, older patients may be less likely to report their income. Here, the missingness of income data is related to the age of the patient but not to the income value itself. For example, a dataset where younger patients report their income more frequently than older patients.

3. Not Missing at Random (NMAR): Definition: The probability of a value being missing is related to the missing value itself. This is the most challenging type to handle. Scenario: A dataset on income levels where high-income individuals are less likely to report their income due to privacy concerns. The missing data is related to the income value itself. Example: In a dataset of income levels, high-income individuals may be less likely to report their income due to privacy concerns. Thus, the missingness is related to the income value itself. For instance, a financial survey where individuals with higher incomes are less likely to disclose their earnings.

Explore Examples:

1. MCAR Example: Scenario: You have a dataset from an online quiz where some answers are missing randomly because of user errors or technical issues. For example, in a dataset of quiz responses, some answers are missing because users accidentally skipped questions. Example Dataset: A dataset from an online educational platform where user responses to certain questions are missing randomly.

2. MAR Example: Scenario: In a dataset of patient health records, older patients are less likely to report their income. The missing income data is related to the patient's age but not to the income value itself. Example Dataset: A dataset from a health survey where missing income data is more common among older patients.

3. NMAR Example: Scenario: A dataset on income levels where high-income individuals are less likely to report their income due to privacy concerns. The missing data is related to the income value itself. Example Dataset: A financial survey dataset where individuals with higher incomes are less likely to disclose their earnings.

Resources: 1. Wikipedia: Missing Data: Provides a general overview of missing data types and strategies for handling them. 2. Introduction to Missing Data by Paul Allison: An informative guide on missing data, including types and handling methods.

1.2: Visualize data

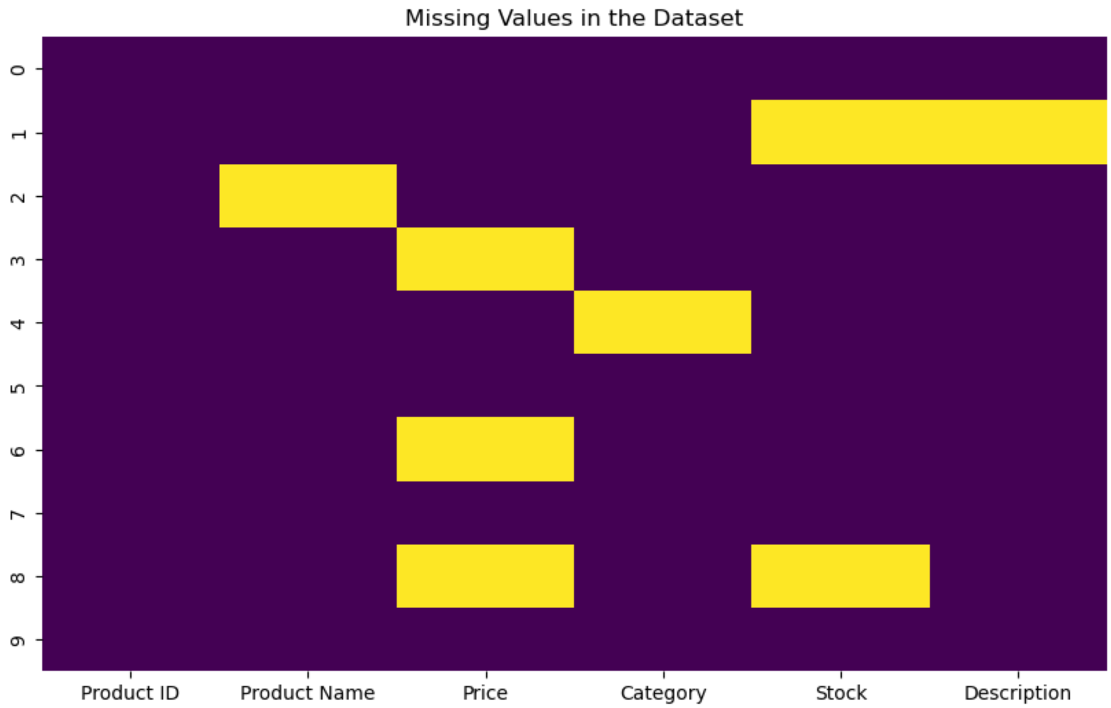
[ ]:
```python
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns

# Load the data from the CSV file
df = pd.read_csv('D:/Projects/Data-cleaning-series/Chapter01 Handling Missing␣
 ↪Values/Products.csv')

# Display the first few rows of the DataFrame
print(df.head())

# Visualize missing values using a heatmap
plt.figure(figsize=(10, 6))
sns.heatmap(df.isnull(), cbar=False, cmap='viridis')
plt.title('Missing Values in the Dataset')
plt.show()
```

```
   Product ID Product Name  Price     Category  Stock             Description
0           1     Widget A  19.99  Electronics  100.0  A high-quality widget
1           2     Widget B  29.99  Electronics    NaN                    NaN
2           3          NaN  15.00   Home Goods   50.0    Durable and stylish
3           4     Widget D    NaN   Home Goods  200.0     A versatile widget
4           5     Widget E   9.99          NaN   10.0  Compact and efficient
```



1.3 Techniques to handle the missing values Techniques

1.3.1 Removal Techniques: 1. Listwise Deletion: Description: Remove rows where any value is missing. This technique is straightforward but can result in a significant loss of data if many rows have missing values. Example: df.dropna()

2. Pairwise Deletion: Description: Use available data for each pair of variables in analysis, ignoring missing values for that pair. This method is useful in correlation or covariance calculations. Example: df.corr(method='pearson', min_periods=1)

1.3.2 Imputation Techniques: 1. Mean/Median/Mode Imputation: Description: Replace missing values with the mean (for numerical data), median (for numerical data with outliers), or mode (for categorical data) of the observed values. Example: df.fillna(df.mean()) (Mean), df.fillna(df.median()) (Median)

2. K-Nearest Neighbors (KNN) Imputation: Description: Use the K-nearest neighbors algorithm to estimate missing values based on the values of the nearest neighbors. Example: from sklearn.impute import KNNImputer (Python)

3. Multiple Imputation by Chained Equations (MICE): Description: Use multiple imputations to handle missing values by modeling each feature with missing data conditional on other features. Combines multiple imputation models to account for uncertainty. Example: from miceforest import ImputationKernel (Python)

4. Predictive Modeling: Description: Use regression or other predictive models to estimate missing values based on other available data. Example: from sklearn.linear_model import LinearRegression (Python)

1.3.3 Advanced Techniques: 1. Interpolation: Description: Estimate missing values based on the values of neighboring data points. Useful for time series data where values are missing at specific time points. Example: df.interpolate(method='linear')

2. Data Augmentation: Description: Generate additional data to fill in missing values. This technique can be used in conjunction with machine learning models. Example: Using generative models or synthetic data methods.

1.3.4 Handling Categorical Data: 1. Mode Imputation: Description: Replace missing values in categorical variables with the most frequent category. Example: df['category'].fillna(df['category'].mode()[0])

2. Categorical Encoding: Description: Replace missing categories with a special placeholder or encode missing values as a separate category. Example: df['category'].fillna('Missing')

1.3.1 Removal Techniques

1. Listwise Deletion

[ ]:
```
# Listwise Deletion: Remove rows with any missing data
listwise_deleted_df = df.dropna()

print("\nDataFrame after Listwise Deletion:")
print(listwise_deleted_df.to_string(index=False))
```

```
DataFrame after Listwise Deletion:
 Product ID Product Name  Price    Category  Stock               Description
          1     Widget A  19.99 Electronics  100.0    A high-quality widget
          6     Widget F  25.00 Electronics    0.0 Latest technology widget
          8     Widget H  39.99     Kitchen   75.0          Premium quality
         10     Widget J  49.99 Electronics   60.0            Best in class
```

2. Pairwise Deletion

[ ]:
```
# Pairwise Deletion: Remove rows with missing data in specific columns, e.g.,␣
↪'Price' and 'Stock'
pairwise_deleted_df = df.dropna(subset=['Price', 'Stock'])

print("\nDataFrame after Pairwise Deletion (on 'Price' and 'Stock'):")
print(pairwise_deleted_df.to_string(index=False))
```

```
DataFrame after Pairwise Deletion (on 'Price' and 'Stock'):
 Product ID Product Name  Price    Category  Stock               Description
          1     Widget A  19.99 Electronics  100.0    A high-quality widget
          3          NaN  15.00  Home Goods   50.0       Durable and stylish
          5     Widget E   9.99         NaN   10.0    Compact and efficient
          6     Widget F  25.00 Electronics    0.0 Latest technology widget
          8     Widget H  39.99     Kitchen   75.0          Premium quality
         10     Widget J  49.99 Electronics   60.0            Best in class
```

1.3.2 Imputation Techniques

1. Mean/Median/Mode

```
[ ]: # Mean Imputation: Replace missing numerical values with the mean of the column
df_mean_imputed = df.copy()
df_mean_imputed['Price'] = df_mean_imputed['Price'].fillna(df['Price'].mean())
df_mean_imputed['Stock'] = df_mean_imputed['Stock'].fillna(df['Stock'].mean())

print("\nDataFrame after Mean Imputation:")
print(df_mean_imputed.to_string(index=False))
```

```
DataFrame after Mean Imputation:
 Product ID Product Name     Price    Category   Stock               Description
          1     Widget A 19.990000 Electronics 100.000    A high-quality widget
          2     Widget B 29.990000 Electronics  80.625                      NaN
          3          NaN 15.000000  Home Goods  50.000      Durable and stylish
          4     Widget D 27.135714  Home Goods 200.000       A versatile widget
          5     Widget E  9.990000         NaN  10.000    Compact and efficient
          6     Widget F 25.000000 Electronics   0.000 Latest technology widget
          7     Widget G 27.135714     Kitchen 150.000     Multi-purpose widget
          8     Widget H 39.990000     Kitchen  75.000          Premium quality
          9     Widget I 27.135714 Electronics  80.625        Advanced features
         10     Widget J 49.990000 Electronics  60.000            Best in class
```

```
[ ]: # Median Imputation: Replace missing numerical values with the median of the␣
  ↪column
df_median_imputed = df.copy()
df_median_imputed['Price'] = df_median_imputed['Price'].fillna(df['Price'].
  ↪median())
df_median_imputed['Stock'] = df_median_imputed['Stock'].fillna(df['Stock'].
  ↪median())

print("\nDataFrame after Median Imputation:")
print(df_median_imputed.to_string(index=False))
```

```
DataFrame after Median Imputation:
 Product ID Product Name  Price    Category  Stock              Description
          1     Widget A  19.99 Electronics  100.0    A high-quality widget
          2     Widget B  29.99 Electronics   67.5                      NaN
          3          NaN  15.00  Home Goods   50.0      Durable and stylish
          4     Widget D  25.00  Home Goods  200.0       A versatile widget
          5     Widget E   9.99         NaN   10.0    Compact and efficient
          6     Widget F  25.00 Electronics    0.0 Latest technology widget
          7     Widget G  25.00     Kitchen  150.0     Multi-purpose widget
          8     Widget H  39.99     Kitchen   75.0          Premium quality
          9     Widget I  25.00 Electronics   67.5        Advanced features
```

```
        10      Widget J  49.99 Electronics   60.0            Best in class
```

[ ]:
```
# Mode Imputation: Replace missing categorical values with the mode of the␣
↪column
df_mode_imputed = df.copy()
df_mode_imputed['Product Name'] = df_mode_imputed['Product Name'].
↪fillna(df['Product Name'].mode()[0])
df_mode_imputed['Category'] = df_mode_imputed['Category'].fillna(df['Category'].
↪mode()[0])
df_mode_imputed['Description'] = df_mode_imputed['Description'].
↪fillna(df['Description'].mode()[0])

print("\nDataFrame after Mode Imputation:")
print(df_mode_imputed.to_string(index=False))
```

```
DataFrame after Mode Imputation:
 Product ID Product Name  Price    Category  Stock               Description
          1      Widget A  19.99 Electronics  100.0     A high-quality widget
          2      Widget B  29.99 Electronics    NaN     A high-quality widget
          3      Widget A  15.00  Home Goods   50.0      Durable and stylish
          4      Widget D    NaN  Home Goods  200.0        A versatile widget
          5      Widget E   9.99 Electronics   10.0     Compact and efficient
          6      Widget F  25.00 Electronics    0.0 Latest technology widget
          7      Widget G    NaN     Kitchen  150.0     Multi-purpose widget
          8      Widget H  39.99     Kitchen   75.0           Premium quality
          9      Widget I    NaN Electronics    NaN        Advanced features
         10      Widget J  49.99 Electronics   60.0             Best in class
```

2. KNN Imputation

[ ]:
```
#K-Nearest Neighbors (KNN) Imputation
from sklearn.impute import KNNImputer

# KNN Imputation
# We need to convert categorical data to numeric for KNNImputer to work
# Encoding categorical variables
df_encoded = pd.get_dummies(df[['Price', 'Stock']], drop_first=True)

# Applying KNN Imputer
imputer = KNNImputer(n_neighbors=3)
df_imputed = df_encoded.copy()
df_imputed[:] = imputer.fit_transform(df_encoded)

# Mapping back to original DataFrame
df[['Price', 'Stock']] = df_imputed

print("\nDataFrame after KNN Imputation:")
```

```python
print(df.to_string(index=False))
```

```
DataFrame after KNN Imputation:
 Product ID Product Name     Price    Category      Stock
Description
          1     Widget A 19.990000 Electronics 100.000000    A high-quality
widget
          2     Widget B 29.990000 Electronics  58.333333
NaN
          3          NaN 15.000000  Home Goods  50.000000      Durable and
stylish
          4     Widget D 36.656667  Home Goods 200.000000       A versatile
widget
          5     Widget E  9.990000         NaN  10.000000    Compact and
efficient
          6     Widget F 25.000000 Electronics   0.000000 Latest technology
widget
          7     Widget G 36.656667     Kitchen 150.000000     Multi-purpose
widget
          8     Widget H 39.990000     Kitchen  75.000000          Premium
quality
          9     Widget I 27.135714 Electronics  80.625000        Advanced
features
         10     Widget J 49.990000 Electronics  60.000000           Best in
class
```

Explanation Encoding Categorical Variables:

Before applying KNN imputation, categorical data needs to be encoded into numeric values. Here, we only encode the numerical columns ('Price' and 'Stock') for simplicity. Applying KNN Imputer:

We use KNNImputer from sklearn.impute to perform imputation based on the nearest neighbors. The n_neighbors parameter specifies the number of neighbors to use for imputation. Mapping Back:

After imputation, we update the original DataFrame with the imputed values. Output The code will output the DataFrame with missing values in 'Price' and 'Stock' columns filled based on the values of their nearest neighbors.

This technique is useful when you believe that missing data can be inferred from similar instances in the dataset. However, it requires all features to be numeric, so categorical variables need to be encoded or handled separately.

3. Mice Imputation

Multiple Imputation by Chained Equations (MICE) is a sophisticated technique for handling missing data. It involves creating multiple imputations (complete datasets) for missing values and then combining the results to account for the uncertainty associated with missing data.

Multiple Imputation by Chained Equations (MICE) Description: MICE performs multiple imputations by iteratively filling in missing values using chained equations. Each variable with missing values is modeled conditional on other variables, and this process is repeated multiple times to generate several imputed datasets. The results from these multiple datasets are then combined to provide a more robust estimate of the missing values.

Here's how to apply MICE in Python using the mice package:

```
[ ]: import pandas as pd
     from fancyimpute import IterativeImputer

     # Preprocessing: Encoding categorical variables
     df_encoded = pd.get_dummies(df[['Price', 'Stock']], drop_first=True)

     # Define the MICE imputer
     mice_imputer = IterativeImputer()

     # Perform MICE Imputation
     df_imputed = df_encoded.copy()
     df_imputed[:] = mice_imputer.fit_transform(df_encoded)

     # Mapping back to original DataFrame
     df[['Price', 'Stock']] = df_imputed

     print("\nDataFrame after MICE Imputation:")
     print(df.to_string(index=False))
```

```
DataFrame after MICE Imputation:
 Product ID Product Name     Price    Category      Stock
Description
          1     Widget A 19.990000 Electronics 100.000000    A high-quality
widget
          2     Widget B 29.990000 Electronics  58.333333
NaN
          3          NaN 15.000000  Home Goods  50.000000      Durable and
stylish
          4     Widget D 36.656667  Home Goods 200.000000       A versatile
widget
          5     Widget E  9.990000         NaN  10.000000    Compact and
efficient
          6     Widget F 25.000000 Electronics   0.000000 Latest technology
widget
          7     Widget G 36.656667     Kitchen 150.000000     Multi-purpose
widget
          8     Widget H 39.990000     Kitchen  75.000000          Premium
quality
          9     Widget I 27.135714 Electronics  80.625000         Advanced
features
         10     Widget J 49.990000 Electronics  60.000000            Best in
class
```

Explanation Encoding Categorical Variables: Convert categorical columns to numerical format. This step is crucial because MICE requires all features to be numerical.

Define the MICE Imputer: IterativeImputer from fancyimpute performs MICE.

Perform MICE Imputation: The fit_transform() method imputes the missing values based on the specified MICE method.

Update Original DataFrame: Replace the missing values in the original DataFrame with the imputed values.

Output The DataFrame will have missing values in the 'Price' and 'Stock' columns filled using the MICE method from the fancyimpute library. This method provides a robust approach to handling missing data and is effective for datasets where the relationships between features are complex.

4. Predictive Modelling

Predictive Modeling for Missing Data Imputation involves using statistical or machine learning models to predict missing values based on the observed data. This approach can be more accurate than simple imputation methods because it takes into account the relationships between variables.

Here's how to implement Predictive Modeling for missing data imputation in Python:

Predictive Modeling Imputation Description: This technique uses models to predict the missing values based on the other observed data in the dataset. It can be implemented using various machine learning algorithms, such as linear regression, decision trees, or even more complex models like random forests or gradient boosting.

Here's an example using linear regression for numerical data imputation:

[ ]:
```
import pandas as pd
from sklearn.linear_model import LinearRegression
from sklearn.model_selection import train_test_split
from sklearn.impute import SimpleImputer

# Read the data from the specified location
df = pd.read_csv('D:/Projects/Data-cleaning-series/Chapter01 Handling Missing␣
 ↪Values/Products.csv')

# Display the initial DataFrame
#print("Initial DataFrame:")
#print(df.to_string(index=False))

# Encoding categorical columns to numerical (if necessary)
df['Category'] = df['Category'].astype('category').cat.codes

# Splitting data into feature matrix X and target variable y
# For simplicity, let's predict missing values in 'Price'
X = df.drop(columns=['Price', 'Product Name', 'Description'])
y = df['Price']

# Identifying rows with missing 'Price'
```

```python
missing_mask = y.isnull()

# Splitting the data into training (non-missing) and testing (missing) sets
X_train = X[~missing_mask]
y_train = y[~missing_mask]
X_test = X[missing_mask]

# Handling missing values in the features
imputer = SimpleImputer(strategy='mean')
X_train = imputer.fit_transform(X_train)
X_test = imputer.transform(X_test)

# Training the linear regression model
model = LinearRegression()
model.fit(X_train, y_train)

# Predicting the missing values
y_pred = model.predict(X_test)

# Updating the original DataFrame with the predicted values
df.loc[missing_mask, 'Price'] = y_pred

# Display the DataFrame after handling missing values
print("\nDataFrame after handling missing values with Linear Regression:")
print(df.to_string(index=False))
```

```
DataFrame after handling missing values with Linear Regression:
 Product ID Product Name      Price  Category  Stock              Description
          1     Widget A 19.990000         0  100.0    A high-quality widget
          2     Widget B 29.990000         0    NaN                      NaN
          3          NaN 15.000000         1   50.0      Durable and stylish
          4     Widget D 52.751192         1  200.0       A versatile widget
          5     Widget E  9.990000        -1   10.0    Compact and efficient
          6     Widget F 25.000000         0    0.0 Latest technology widget
          7     Widget G 53.242217         2  150.0     Multi-purpose widget
          8     Widget H 39.990000         2   75.0          Premium quality
          9     Widget I 40.868404         0    NaN        Advanced features
         10     Widget J 49.990000         0   60.0            Best in class
```

Explanation

Read the Data: Load the dataset from the specified location using pd.read_csv().

Encoding Categorical Variables: Convert the ‘Category’ column to numerical values for use in the model.

Split Data into Features and Target: Separate the feature matrix X (all columns except ‘Price’) and the target variable y (‘Price’).

Identify Missing Data: Create a mask (missing_mask) to identify rows with missing 'Price' values.

Split Data into Training and Testing Sets: Use rows with non-missing 'Price' values for training and those with missing 'Price' values for testing.

Handle Missing Values in Features: Use SimpleImputer to fill missing values in the features with the mean value.

Train Linear Regression Model: Train the model using the non-missing data.

Predict Missing Values: Use the trained model to predict the missing 'Price' values.

Update Original DataFrame: Replace missing 'Price' values in the original DataFrame with the predicted values.

Output the Final DataFrame: Display the DataFrame after handling missing values.

This approach is useful when you believe that the missing data can be predicted based on other available data. However, it's important to validate the model's performance and ensure that the assumptions of linear regression are reasonably met.

1.3.3 Advance Technique

1. Interpolation

Interpolation is a technique used to estimate missing values based on the known values of neighboring data points. It's particularly useful for time series data, where missing values can be inferred by assuming a linear or other defined relationship between the data points.

Description: Interpolation estimates missing values by assuming a certain pattern or trend in the data. For instance, in time series data, it can use linear, polynomial, or spline interpolation to fill in the missing values. Linear interpolation assumes that the change between data points is linear, making it a simple and commonly used method.

Here's the full code to handle missing values using interpolation:

[ ]:
```
import pandas as pd

# Read the data from the specified location
df = pd.read_csv('D:/Projects/Data-cleaning-series/Chapter01 Handling Missing␣
↪Values/Products.csv')

# Display the initial DataFrame
#print("Initial DataFrame:")
#print(df.to_string(index=False))

# Handling missing values using linear interpolation
df_interpolated = df.copy()
df_interpolated['Price'] = df_interpolated['Price'].interpolate(method='linear')
df_interpolated['Stock'] = df_interpolated['Stock'].interpolate(method='linear')

# Display the DataFrame after handling missing values with Interpolation
print("\nDataFrame after handling missing values with Interpolation:")
```

```python
print(df_interpolated.to_string(index=False))
```

```
DataFrame after handling missing values with Interpolation:
 Product ID Product Name  Price    Category  Stock              Description
          1     Widget A 19.990 Electronics  100.0    A high-quality widget
          2     Widget B 29.990 Electronics   75.0                      NaN
          3          NaN 15.000  Home Goods   50.0      Durable and stylish
          4     Widget D 12.495  Home Goods  200.0       A versatile widget
          5     Widget E  9.990         NaN   10.0    Compact and efficient
          6     Widget F 25.000 Electronics    0.0 Latest technology widget
          7     Widget G 32.495     Kitchen  150.0     Multi-purpose widget
          8     Widget H 39.990     Kitchen   75.0          Premium quality
          9     Widget I 44.990 Electronics   67.5        Advanced features
         10     Widget J 49.990 Electronics   60.0            Best in class
```

Explanation

Read the Data: Load the dataset from the specified location using pd.read_csv().

Initial Display: Display the initial DataFrame to see the missing values before applying interpolation.

Applying Linear Interpolation: Use the interpolate() function with the method='linear' argument to fill in missing values in the 'Price' and 'Stock' columns. This method assumes a linear change between the neighboring points.

Final Display: Display the DataFrame after interpolation to show the filled values.

Additional Information Interpolation Methods: Besides 'linear', other methods such as 'polynomial', 'spline', or 'pad' (propagate the last valid observation forward) can be used depending on the nature of the data. Time Series Data: Interpolation is particularly useful for time series data, where values are missing at specific time points, and the assumption of continuity can reasonably be made.

This method is simple and effective for datasets where the relationship between data points is expected to be continuous and can be linearly approximated.

2. Data Augmentation

Data Augmentation is a technique used to increase the amount of data by adding slightly modified copies of existing data or newly created synthetic data. In the context of handling missing values, data augmentation can involve generating additional data points to fill in gaps, which can be particularly useful when working with machine learning models that require a complete dataset.

Advanced Techniques: Data Augmentation Description: Data augmentation involves generating new data points that can either be slightly altered versions of existing data or completely synthetic data points created using statistical or machine learning methods. This technique helps in filling missing values by increasing the dataset's size, diversity, and richness, providing more information for training models.

Example: Using generative models like Variational Autoencoders (VAEs) or Generative Adversarial Networks (GANs), or simpler methods like synthetic data generation based on existing data distributions.

Here's a basic example using Python to demonstrate synthetic data generation for handling missing values. We'll use the existing dataset and generate additional rows to simulate data augmentation.

```
[ ]: import pandas as pd
import numpy as np

# Read the data from the specified location
df = pd.read_csv('D:/Projects/Data-cleaning-series/Chapter01 Handling Missing␣
 ↪Values/Products.csv')

# Display the initial DataFrame
print("Initial DataFrame:")
print(df.to_string(index=False))

# Function to generate synthetic data based on existing data distributions
def generate_synthetic_data(df, num_samples):
    synthetic_data = pd.DataFrame()
    for col in df.columns:
        if df[col].dtype == 'object':
            # For categorical data, randomly sample from existing categories
            synthetic_data[col] = np.random.choice(df[col].dropna().unique(),␣
 ↪num_samples, replace=True)
        else:
            # For numerical data, sample from a normal distribution based on␣
 ↪existing data
            mean, std = df[col].mean(), df[col].std()
            synthetic_data[col] = np.random.normal(mean, std, num_samples)
    return synthetic_data

# Generate synthetic data with the same length as the original DataFrame
num_synthetic_samples = len(df)
synthetic_df = generate_synthetic_data(df, num_synthetic_samples)

# Append synthetic data to the original DataFrame
df_augmented = pd.concat([df, synthetic_df], ignore_index=True)

# Display the DataFrame after data augmentation
print("\nDataFrame after Data Augmentation with Synthetic Data:")
print(df_augmented.to_string(index=False))
```

```
Initial DataFrame:
 Product ID Product Name  Price    Category  Stock            Description
          1     Widget A  19.99 Electronics  100.0    A high-quality widget
          2     Widget B  29.99 Electronics    NaN                      NaN
          3          NaN  15.00  Home Goods   50.0      Durable and stylish
          4     Widget D    NaN  Home Goods  200.0       A versatile widget
```

```
         5     Widget E   9.99         NaN   10.0    Compact and efficient
         6     Widget F  25.00 Electronics    0.0 Latest technology widget
         7     Widget G    NaN     Kitchen  150.0     Multi-purpose widget
         8     Widget H  39.99     Kitchen   75.0          Premium quality
         9     Widget I    NaN Electronics    NaN        Advanced features
        10     Widget J  49.99 Electronics   60.0            Best in class

DataFrame after Data Augmentation with Synthetic Data:
 Product ID Product Name     Price    Category      Stock
Description
   1.000000     Widget A 19.990000 Electronics 100.000000    A high-quality
widget
   2.000000     Widget B 29.990000 Electronics        NaN
NaN
   3.000000          NaN 15.000000  Home Goods  50.000000      Durable and
stylish
   4.000000     Widget D       NaN  Home Goods 200.000000       A versatile
widget
   5.000000     Widget E  9.990000         NaN  10.000000    Compact and
efficient
   6.000000     Widget F 25.000000 Electronics   0.000000 Latest technology
widget
   7.000000     Widget G       NaN     Kitchen 150.000000     Multi-purpose
widget
   8.000000     Widget H 39.990000     Kitchen  75.000000          Premium
quality
   9.000000     Widget I       NaN Electronics        NaN        Advanced
features
  10.000000     Widget J 49.990000 Electronics  60.000000           Best in
class
   6.529963     Widget B 23.342296     Kitchen 133.797916 Latest technology
widget
   3.580865     Widget F 40.737700     Kitchen  49.072782     Multi-purpose
widget
   0.694007     Widget D 27.300388  Home Goods  58.869285          Premium
quality
   5.662413     Widget I 46.985558  Home Goods 132.278255      Durable and
stylish
   5.799225     Widget I 56.583819  Home Goods 112.983776 Latest technology
widget
   3.921844     Widget H 30.392279 Electronics  86.609957     Multi-purpose
widget
   6.949042     Widget D 23.992961     Kitchen 129.306928      Durable and
stylish
  -1.279711     Widget H 19.773278     Kitchen  43.129315        Advanced
features
   5.917893     Widget I  3.646258  Home Goods  41.905164          Premium
quality
```

```
   1.849101     Widget I  2.095535     Kitchen  90.742687        Advanced
features
```

Explanation

Read the Data: Load the dataset from the specified location using pd.read_csv().

Function to Generate Synthetic Data: The generate_synthetic_data function generates synthetic samples based on existing data distributions. For categorical columns, it randomly samples existing categories. For numerical columns, it samples from a normal distribution using the mean and standard deviation of the existing data.

Generating Synthetic Data: Generate a specified number of synthetic samples (num_synthetic_samples).

Appending Synthetic Data: The synthetic data is appended to the original DataFrame, augmenting the dataset.

Final Display: Display the augmented DataFrame to show the additional synthetic rows.

Additional Information Generative Models: For more sophisticated applications, generative models like GANs or VAEs can be used to create realistic synthetic data. These models learn the underlying data distribution and can generate new data points that are similar to the original data.

Applications: Data augmentation is particularly useful in machine learning when dealing with small datasets or when there is a need to fill in missing data creatively. It enhances the model's ability to generalize by providing a richer dataset for training.

This example demonstrates a basic approach to data augmentation. More advanced techniques can include domain-specific knowledge or using machine learning models to generate more realistic synthetic data.

1.3.4 Handling Categorical Data

1. Mode Imputation

Mode Imputation is a technique used to handle missing values in categorical variables by replacing them with the most frequent category (mode). This method is simple and effective for categorical data where the most common value can reasonably substitute for missing entries.

Description: Mode Imputation involves filling missing values in categorical variables with the most frequently occurring value (mode). This approach assumes that the most common category is a reasonable guess for the missing data.

Example: Use df['category'].fillna(df['category'].mode()[0]) to replace missing values in the 'category' column with the mode of that column.

Here's how you can apply mode imputation to a dataset:

[ ]:
```
import pandas as pd

# Read the data from the specified location
df = pd.read_csv('D:/Projects/Data-cleaning-series/Chapter01 Handling Missing␣
 ↪Values/Products.csv')
```

```python
# Display the initial DataFrame
print("Initial DataFrame:")
print(df.to_string(index=False))

# Mode Imputation for categorical data
df['Category'] = df['Category'].fillna(df['Category'].mode()[0])

# Display the DataFrame after mode imputation
print("\nDataFrame after Mode Imputation:")
print(df.to_string(index=False))
```

```
Initial DataFrame:
 Product ID Product Name  Price    Category  Stock              Description
          1     Widget A  19.99 Electronics  100.0    A high-quality widget
          2     Widget B  29.99 Electronics    NaN                      NaN
          3          NaN  15.00  Home Goods   50.0      Durable and stylish
          4     Widget D    NaN  Home Goods  200.0       A versatile widget
          5     Widget E   9.99         NaN   10.0    Compact and efficient
          6     Widget F  25.00 Electronics    0.0 Latest technology widget
          7     Widget G    NaN     Kitchen  150.0     Multi-purpose widget
          8     Widget H  39.99     Kitchen   75.0          Premium quality
          9     Widget I    NaN Electronics    NaN        Advanced features
         10     Widget J  49.99 Electronics   60.0            Best in class

DataFrame after Mode Imputation:
 Product ID Product Name  Price    Category  Stock              Description
          1     Widget A  19.99 Electronics  100.0    A high-quality widget
          2     Widget B  29.99 Electronics    NaN                      NaN
          3          NaN  15.00  Home Goods   50.0      Durable and stylish
          4     Widget D    NaN  Home Goods  200.0       A versatile widget
          5     Widget E   9.99 Electronics   10.0    Compact and efficient
          6     Widget F  25.00 Electronics    0.0 Latest technology widget
          7     Widget G    NaN     Kitchen  150.0     Multi-purpose widget
          8     Widget H  39.99     Kitchen   75.0          Premium quality
          9     Widget I    NaN Electronics    NaN        Advanced features
         10     Widget J  49.99 Electronics   60.0            Best in class
```

Explanation

Read the Data: Load the dataset from the specified location using pd.read_csv().

Initial Display: Display the DataFrame to see the missing values before applying mode imputation.

Mode Imputation: Use fillna() with the mode value to replace missing values in the ‘Category’ column. df[‘Category’].mode()[0] retrieves the most frequent category in the ‘Category’ column.

Final Display: Display the DataFrame after mode imputation to show the filled values.

Additional Information Mode Calculation: df[‘Category’].mode() returns a Series of modes. [0] selects the first mode if there are multiple. Applicability: Mode imputation is appropriate for categorical data but not suitable for numerical data, where other imputation methods like mean or median may be more appropriate.

This method provides a straightforward way to handle missing values in categorical data, ensuring that the most common category is used as a substitute for missing entries.

2. Categorical Encoding

Categorical Encoding is a technique used to handle missing values in categorical data by replacing them with a special placeholder or encoding them as a separate category. This method is useful when you want to maintain the integrity of categorical data and ensure that missing values are explicitly recognized in the analysis or modeling.

Handling Categorical Data: Categorical Encoding Description: Categorical Encoding involves replacing missing values in categorical variables with a predefined placeholder or encoding missing values as a separate, distinct category. This approach helps preserve the categorical nature of the data and allows the missing values to be handled explicitly during analysis or modeling.

Example: Use df['category'].fillna('Missing') to replace missing values in the 'category' column with the string 'Missing'.

[ ]:
```
import pandas as pd

# Read the data from the specified location
df = pd.read_csv('D:/Projects/Data-cleaning-series/Chapter01 Handling Missing␣
↪Values/Products.csv')

# Display the initial DataFrame
print("Initial DataFrame:")
print(df.to_string(index=False))

# Categorical Encoding for missing values
df['Category'] = df['Category'].fillna('Missing')

# Display the DataFrame after categorical encoding
print("\nDataFrame after Categorical Encoding:")
print(df.to_string(index=False))
```

```
Initial DataFrame:
 Product ID Product Name  Price    Category  Stock              Description
          1     Widget A  19.99 Electronics  100.0    A high-quality widget
          2     Widget B  29.99 Electronics    NaN                      NaN
          3          NaN  15.00  Home Goods   50.0       Durable and stylish
          4     Widget D    NaN  Home Goods  200.0      A versatile widget
          5     Widget E   9.99         NaN   10.0    Compact and efficient
          6     Widget F  25.00 Electronics    0.0 Latest technology widget
          7     Widget G    NaN     Kitchen  150.0     Multi-purpose widget
          8     Widget H  39.99     Kitchen   75.0          Premium quality
          9     Widget I    NaN Electronics    NaN        Advanced features
         10     Widget J  49.99 Electronics   60.0            Best in class
```

```
DataFrame after Categorical Encoding:
 Product ID Product Name  Price    Category  Stock              Description
          1     Widget A  19.99 Electronics  100.0    A high-quality widget
          2     Widget B  29.99 Electronics    NaN                      NaN
          3          NaN  15.00  Home Goods   50.0      Durable and stylish
          4     Widget D    NaN  Home Goods  200.0      A versatile widget
          5     Widget E   9.99     Missing   10.0    Compact and efficient
          6     Widget F  25.00 Electronics    0.0 Latest technology widget
          7     Widget G    NaN     Kitchen  150.0     Multi-purpose widget
          8     Widget H  39.99     Kitchen   75.0          Premium quality
          9     Widget I    NaN Electronics    NaN        Advanced features
         10     Widget J  49.99 Electronics   60.0            Best in class
```

Explanation

Read the Data: Load the dataset from the specified location using pd.read_csv().

Initial Display: Display the DataFrame to see the missing values before applying categorical encoding.

Categorical Encoding: Use fillna() with the placeholder string 'Missing' to replace missing values in the 'Category' column.

Final Display: Display the DataFrame after categorical encoding to show the filled values.

Additional Information Placeholder Choice: The placeholder 'Missing' can be any string that clearly indicates the absence of data. You can choose a placeholder that best fits your analysis needs. Encoding as a Separate Category: This method is often useful when you want to treat missing values as a distinct category in machine learning models. It allows the model to learn patterns associated with missing values if they carry any significance.

This method helps to ensure that missing values in categorical data are explicitly represented and can be handled appropriately during data analysis or modeling.