In [1]:
```
from IPython.core.display import display, HTML
display(HTML("<style>.container { width:100% !important; }</style>"))
```

Data preprocessing

Data preprocessing refers to the steps taken to prepare and clean raw data before it can be used for analysis or machine learning algorithms. It involves transforming and organizing the data in a way that makes it suitable for further processing and analysis.

In [2]:
```
# importing libraries
import numpy as np
import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt
import statistics as stats

%matplotlib inline
```

Data Cleaning

Data cleaning is an essential step in data preprocessing, aimed at handling missing values, removing duplicates, and addressing errors or outliers in the dataset.

Handling Missing Values

Removing rows or columns with missing values

In [3]:
```
data = pd.DataFrame({
    'Feature1': [1, 2, np.nan, 4, 5, 6, 8, 11, 32, 10],
    'Feature2': [6, 10, 15, 30, 25, 21, 30, 42, 31, 51],
    'Feature3': [6, 7, 8, np.nan, 10, 6, 8, 11, np.nan, 15],
    'Feature4': [11, 12, 13, 14, 15, 7, 8, 11, 10, 30]
})
```

In [4]:
```
print("Original Data :\n",data)

# Remove rows with missing values
row_removed_data = data.dropna(axis=0)
print("Remove rows with missing values :\n",row_removed_data)

# Remove columns with missing values
col_removed_data = data.dropna(axis=1)
print("Remove columns with missing values :\n",col_removed_data)
```

```
Original Data :
    Feature1  Feature2  Feature3  Feature4
0        1.0         6       6.0        11
1        2.0        10       7.0        12
2        NaN        15       8.0        13
3        4.0        30       NaN        14
4        5.0        25      10.0        15
5        6.0        21       6.0         7
6        8.0        30       8.0         8
7       11.0        42      11.0        11
8       32.0        31       NaN        10
9       10.0        51      15.0        30
Remove rows with missing values :
    Feature1  Feature2  Feature3  Feature4
0        1.0         6       6.0        11
1        2.0        10       7.0        12
4        5.0        25      10.0        15
5        6.0        21       6.0         7
6        8.0        30       8.0         8
7       11.0        42      11.0        11
9       10.0        51      15.0        30
Remove columns with missing values :
    Feature2  Feature4
0         6        11
1        10        12
2        15        13
3        30        14
4        25        15
5        21         7
6        30         8
7        42        11
8        31        10
9        51        30
```

Filling missing values with mean, median, or mode

In [5]:
```
print("Original Data :\n",data)

# Fill numerical missing values with mean
filled_mean_data = data.fillna(data.mean())
print("\nFilled numerical missing values with mean :\n",filled_mean_data)

# Fill numerical missing values with median
filled_median_data = data.fillna(data.median())
print("\nFilled numerical missing values with median :\n",filled_median_dat

# Fill categorical missing values with mode
filled_mode_feature1 = data["Feature1"].fillna(stats.mode(data["Feature1"])
filled_mode_feature2 = data["Feature2"].fillna(stats.mode(data["Feature2"])
filled_mode_feature3 = data["Feature3"].fillna(stats.mode(data["Feature4"])
filled_mode_feature4 = data["Feature4"].fillna(stats.mode(data["Feature4"])
filled_mode_data = pd.concat([filled_mode_feature1,filled_mode_feature2,fil
print("\nFilled numerical missing values with mode :\n",filled_mode_data)
```

```
Original Data :
    Feature1  Feature2  Feature3  Feature4
0        1.0         6       6.0        11
1        2.0        10       7.0        12
2        NaN        15       8.0        13
3        4.0        30       NaN        14
4        5.0        25      10.0        15
5        6.0        21       6.0         7
6        8.0        30       8.0         8
7       11.0        42      11.0        11
8       32.0        31       NaN        10
9       10.0        51      15.0        30

Filled numerical missing values with mean :
     Feature1  Feature2  Feature3  Feature4
0   1.000000         6     6.000        11
1   2.000000        10     7.000        12
2   8.777778        15     8.000        13
3   4.000000        30     8.875        14
4   5.000000        25    10.000        15
5   6.000000        21     6.000         7
6   8.000000        30     8.000         8
7  11.000000        42    11.000        11
8  32.000000        31     8.875        10
9  10.000000        51    15.000        30

Filled numerical missing values with median :
    Feature1  Feature2  Feature3  Feature4
0        1.0         6       6.0        11
1        2.0        10       7.0        12
2        6.0        15       8.0        13
3        4.0        30       8.0        14
4        5.0        25      10.0        15
5        6.0        21       6.0         7
6        8.0        30       8.0         8
7       11.0        42      11.0        11
8       32.0        31       8.0        10
9       10.0        51      15.0        30

Filled numerical missing values with mode :
    Feature1  Feature2  Feature3  Feature4
0        1.0         6       6.0        11
1        2.0        10       7.0        12
2        1.0        15       8.0        13
3        4.0        30      11.0        14
4        5.0        25      10.0        15
5        6.0        21       6.0         7
6        8.0        30       8.0         8
7       11.0        42      11.0        11
8       32.0        31      11.0        10
9       10.0        51      15.0        30
```

Forward filling or backward filling missing values

In [6]:
```
# Forward filling missing values
data_ffill = data.ffill()

# Backward filling missing values
data_bfill = data.bfill()

# Print the original dataset
print("Original Dataset :\n",data)


# Print the dataset after forward filling missing values
print("\nForward Filled Dataset :\n",data_ffill)


# Print the dataset after backward filling missing values
print("\nBackward Filled Dataset :\n",data_bfill)
```

```
Original Dataset :
    Feature1  Feature2  Feature3  Feature4
0        1.0         6       6.0        11
1        2.0        10       7.0        12
2        NaN        15       8.0        13
3        4.0        30       NaN        14
4        5.0        25      10.0        15
5        6.0        21       6.0         7
6        8.0        30       8.0         8
7       11.0        42      11.0        11
8       32.0        31       NaN        10
9       10.0        51      15.0        30

Forward Filled Dataset :
    Feature1  Feature2  Feature3  Feature4
0        1.0         6       6.0        11
1        2.0        10       7.0        12
2        2.0        15       8.0        13
3        4.0        30       8.0        14
4        5.0        25      10.0        15
5        6.0        21       6.0         7
6        8.0        30       8.0         8
7       11.0        42      11.0        11
8       32.0        31      11.0        10
9       10.0        51      15.0        30

Backward Filled Dataset :
    Feature1  Feature2  Feature3  Feature4
0        1.0         6       6.0        11
1        2.0        10       7.0        12
2        4.0        15       8.0        13
3        4.0        30      10.0        14
4        5.0        25      10.0        15
5        6.0        21       6.0         7
6        8.0        30       8.0         8
7       11.0        42      11.0        11
8       32.0        31      15.0        10
9       10.0        51      15.0        30
```

Replacing missing values with specific values

In [7]:
```
# Print the original dataset
print("Original Dataset :\n",data)

# Replace missing numerical values with -999
replacing_999 = data.fillna(-999)
print("\nReplace missing numerical values with -999 :\n",replacing_999)

# Replace missing categorical values with 'Unknown'
replacing_unknown = data.fillna('Unknown')
print("\nReplace missing categorical values with 'Unknown' :\n",replacing_u
```

```
Original Dataset :
    Feature1  Feature2  Feature3  Feature4
0        1.0         6       6.0        11
1        2.0        10       7.0        12
2        NaN        15       8.0        13
3        4.0        30       NaN        14
4        5.0        25      10.0        15
5        6.0        21       6.0         7
6        8.0        30       8.0         8
7       11.0        42      11.0        11
8       32.0        31       NaN        10
9       10.0        51      15.0        30

Replace missing numerical values with -999 :
    Feature1  Feature2  Feature3  Feature4
0        1.0         6       6.0        11
1        2.0        10       7.0        12
2     -999.0        15       8.0        13
3        4.0        30    -999.0        14
4        5.0        25      10.0        15
5        6.0        21       6.0         7
6        8.0        30       8.0         8
7       11.0        42      11.0        11
8       32.0        31    -999.0        10
9       10.0        51      15.0        30

Replace missing categorical values with 'Unknown' :
   Feature1  Feature2 Feature3  Feature4
0       1.0         6      6.0        11
1       2.0        10      7.0        12
2   Unknown        15      8.0        13
3       4.0        30  Unknown        14
4       5.0        25     10.0        15
5       6.0        21      6.0         7
6       8.0        30      8.0         8
7      11.0        42     11.0        11
8      32.0        31  Unknown        10
9      10.0        51     15.0        30
```

Using interpolation for filling missing values

In [8]:
```
# Print the original dataset
print("Original Dataset :\n",data)

# Interpolate missing numerical values
data_interpolated = data.interpolate()
print("\nInterpolate missing numerical values :\n",data_interpolated)
```

```
Original Dataset :
    Feature1  Feature2  Feature3  Feature4
0        1.0         6       6.0        11
1        2.0        10       7.0        12
2        NaN        15       8.0        13
3        4.0        30       NaN        14
4        5.0        25      10.0        15
5        6.0        21       6.0         7
6        8.0        30       8.0         8
7       11.0        42      11.0        11
8       32.0        31       NaN        10
9       10.0        51      15.0        30

Interpolate missing numerical values :
    Feature1  Feature2  Feature3  Feature4
0        1.0         6       6.0        11
1        2.0        10       7.0        12
2        3.0        15       8.0        13
3        4.0        30       9.0        14
4        5.0        25      10.0        15
5        6.0        21       6.0         7
6        8.0        30       8.0         8
7       11.0        42      11.0        11
8       32.0        31      13.0        10
9       10.0        51      15.0        30
```

Multiple imputation using fancyimpute library

In [9]:
```
from fancyimpute import IterativeImputer

# Create sample dataset with missing values
data = pd.DataFrame({
    'A': [1, np.nan, 3, np.nan, 5],
    'B': [np.nan, 2, 3, np.nan, 5],
    'C': [1, 2, np.nan, np.nan, 5]
})

# Perform multiple imputation
imputer = IterativeImputer()
data_imputed = imputer.fit_transform(data)

# Convert the imputed array back to a DataFrame
data_imputed = pd.DataFrame(data_imputed, columns=data.columns)

# Print the original dataset
print("Original Dataset:")
print(data)

# Print the imputed dataset
print("\nImputed Dataset:")
print(data_imputed)
```

```
(CVXPY) Jul 17 11:20:36 PM: Encountered unexpected exception importing so
lver CVXOPT:
ImportError('DLL load failed while importing base: The specified module c
ould not be found.')
(CVXPY) Jul 17 11:20:36 PM: Encountered unexpected exception importing so
lver GLPK:
ImportError('DLL load failed while importing base: The specified module c
ould not be found.')
(CVXPY) Jul 17 11:20:36 PM: Encountered unexpected exception importing so
lver GLPK_MI:
ImportError('DLL load failed while importing base: The specified module c
ould not be found.')
Original Dataset:
     A    B    C
0  1.0  NaN  1.0
1  NaN  2.0  2.0
2  3.0  3.0  NaN
3  NaN  NaN  NaN
4  5.0  5.0  5.0

Imputed Dataset:
          A         B         C
0  1.000000  1.000063  1.000000
1  1.999852  2.000000  2.000000
2  3.000000  3.000000  3.000014
3  2.749963  2.750016  2.750004
4  5.000000  5.000000  5.000000
```

Filling missing values using predictive models

In [10]:
```
from fancyimpute import SoftImpute

# Create sample dataset with missing values
data = pd.DataFrame({
    'A': [1, np.nan, 3, np.nan, 5],
    'B': [np.nan, 2, 3, np.nan, 5],
    'C': [1, 2, np.nan, np.nan, 5]
})

# Print the original dataset
print("Original Dataset :\n",data)
print("")

# Perform matrix completion using SoftImpute
imputer = SoftImpute()
data_imputed = imputer.fit_transform(data)

# Convert the imputed array back to a DataFrame
data_imputed = pd.DataFrame(data_imputed, columns=data.columns)

# Print the imputed dataset
print("\nImputed Dataset:")
print(data_imputed)
```

```
Original Dataset :
      A    B    C
0  1.0  NaN  1.0
1  NaN  2.0  2.0
2  3.0  3.0  NaN
3  NaN  NaN  NaN
4  5.0  5.0  5.0

[SoftImpute] Max Singular Value of X_init = 9.703555
[SoftImpute] Iter 1: observed MAE=0.092302 rank=3
[SoftImpute] Iter 2: observed MAE=0.092414 rank=3
[SoftImpute] Iter 3: observed MAE=0.092508 rank=3
[SoftImpute] Iter 4: observed MAE=0.092584 rank=3
[SoftImpute] Iter 5: observed MAE=0.092639 rank=3
[SoftImpute] Iter 6: observed MAE=0.092673 rank=3
[SoftImpute] Iter 7: observed MAE=0.092684 rank=3
[SoftImpute] Iter 8: observed MAE=0.092669 rank=3
[SoftImpute] Iter 9: observed MAE=0.092627 rank=3
[SoftImpute] Iter 10: observed MAE=0.092554 rank=3
[SoftImpute] Iter 11: observed MAE=0.092447 rank=3
[SoftImpute] Iter 12: observed MAE=0.092301 rank=3
[SoftImpute] Iter 13: observed MAE=0.092111 rank=3
[SoftImpute] Iter 14: observed MAE=0.091870 rank=3
[SoftImpute] Iter 15: observed MAE=0.091569 rank=3
[SoftImpute] Iter 16: observed MAE=0.091195 rank=3
[SoftImpute] Iter 17: observed MAE=0.090387 rank=2
[SoftImpute] Iter 18: observed MAE=0.083420 rank=2
[SoftImpute] Iter 19: observed MAE=0.079818 rank=2
[SoftImpute] Iter 20: observed MAE=0.077962 rank=2
[SoftImpute] Iter 21: observed MAE=0.076999 rank=2
[SoftImpute] Iter 22: observed MAE=0.076488 rank=2
[SoftImpute] Iter 23: observed MAE=0.076205 rank=2
[SoftImpute] Iter 24: observed MAE=0.076038 rank=2
[SoftImpute] Iter 25: observed MAE=0.075932 rank=2
[SoftImpute] Iter 26: observed MAE=0.075862 rank=2
[SoftImpute] Iter 27: observed MAE=0.075822 rank=2
[SoftImpute] Iter 28: observed MAE=0.075458 rank=1
[SoftImpute] Iter 29: observed MAE=0.069518 rank=1
[SoftImpute] Iter 30: observed MAE=0.067249 rank=1
[SoftImpute] Iter 31: observed MAE=0.066347 rank=1
[SoftImpute] Iter 32: observed MAE=0.065973 rank=1
[SoftImpute] Iter 33: observed MAE=0.065812 rank=1
[SoftImpute] Stopped after iteration 33 for lambda=0.194071

Imputed Dataset:
          A         B        C
0  1.000000  0.977734  1.00000
1  1.949875  2.000000  2.00000
2  3.000000  3.000000  2.90091
3 -0.000000 -0.000000 -0.00000
4  5.000000  5.000000  5.00000
```

Filling missing values using K-nearest neighbors (KNN) imputation

In [11]:
```
from fancyimpute import KNN

# Print the original dataset
print("Original Dataset :\n",data)
print("")

# Create an instance of KNN imputer
imputer = KNN()
data_imputed = imputer.fit_transform(data)
print("Filling missing values using K-nearest neighbors (KNN) imputation :\
```

```
Original Dataset :
      A    B    C
0  1.0  NaN  1.0
1  NaN  2.0  2.0
2  3.0  3.0  NaN
3  NaN  NaN  NaN
4  5.0  5.0  5.0

Imputing row 1/5 with 1 missing, elapsed time: 0.001
[KNN] Warning: 3/15 still missing after imputation, replacing with 0
Filling missing values using K-nearest neighbors (KNN) imputation :
 [[1.         2.33333333 1.        ]
 [2.15789468 2.         2.        ]
 [3.         3.         2.33333333]
 [0.         0.         0.        ]
 [5.         5.         5.        ]]
```

Matrix Factorization

In [12]:
```
from fancyimpute import SoftImpute

# Print the original dataset
print("Original Dataset :\n",data)
print("")

# Create an instance of SoftImpute imputer
imputer = SoftImpute()
data_imputed = imputer.fit_transform(data)
print("\nMatrix Factorization Data :\n",data_imputed)
```

```
Original Dataset :
      A    B    C
0  1.0  NaN  1.0
1  NaN  2.0  2.0
2  3.0  3.0  NaN
3  NaN  NaN  NaN
4  5.0  5.0  5.0

[SoftImpute] Max Singular Value of X_init = 9.703555
[SoftImpute] Iter 1: observed MAE=0.092302 rank=3
[SoftImpute] Iter 2: observed MAE=0.092414 rank=3
[SoftImpute] Iter 3: observed MAE=0.092508 rank=3
[SoftImpute] Iter 4: observed MAE=0.092584 rank=3
[SoftImpute] Iter 5: observed MAE=0.092639 rank=3
[SoftImpute] Iter 6: observed MAE=0.092673 rank=3
[SoftImpute] Iter 7: observed MAE=0.092684 rank=3
[SoftImpute] Iter 8: observed MAE=0.092669 rank=3
[SoftImpute] Iter 9: observed MAE=0.092627 rank=3
[SoftImpute] Iter 10: observed MAE=0.092554 rank=3
[SoftImpute] Iter 11: observed MAE=0.092447 rank=3
[SoftImpute] Iter 12: observed MAE=0.092301 rank=3
[SoftImpute] Iter 13: observed MAE=0.092111 rank=3
[SoftImpute] Iter 14: observed MAE=0.091870 rank=3
[SoftImpute] Iter 15: observed MAE=0.091569 rank=3
[SoftImpute] Iter 16: observed MAE=0.091195 rank=3
[SoftImpute] Iter 17: observed MAE=0.090387 rank=2
[SoftImpute] Iter 18: observed MAE=0.083420 rank=2
[SoftImpute] Iter 19: observed MAE=0.079818 rank=2
[SoftImpute] Iter 20: observed MAE=0.077962 rank=2
[SoftImpute] Iter 21: observed MAE=0.076999 rank=2
[SoftImpute] Iter 22: observed MAE=0.076488 rank=2
[SoftImpute] Iter 23: observed MAE=0.076205 rank=2
[SoftImpute] Iter 24: observed MAE=0.076038 rank=2
[SoftImpute] Iter 25: observed MAE=0.075932 rank=2
[SoftImpute] Iter 26: observed MAE=0.075862 rank=2
[SoftImpute] Iter 27: observed MAE=0.075822 rank=2
[SoftImpute] Iter 28: observed MAE=0.075458 rank=1
[SoftImpute] Iter 29: observed MAE=0.069518 rank=1
[SoftImpute] Iter 30: observed MAE=0.067249 rank=1
[SoftImpute] Iter 31: observed MAE=0.066347 rank=1
[SoftImpute] Iter 32: observed MAE=0.065973 rank=1
[SoftImpute] Iter 33: observed MAE=0.065812 rank=1
[SoftImpute] Stopped after iteration 33 for lambda=0.194071

Matrix Factorization Data :
 [[ 1.          0.97773356  1.        ]
 [ 1.9498746   2.          2.        ]
 [ 3.          3.          2.90091019]
 [-0.         -0.         -0.        ]
 [ 5.          5.          5.        ]]
```

Handling Duplicates

Identifying duplicates

In [13]:
```
# Create sample dataset with duplicates
data = pd.DataFrame({
    'ID': [1, 2, 3, 4, 1, 2, 5],
    'Name': ['John', 'Jane', 'Mike', 'Emma', 'John', 'Jane', 'David'],
    'Age': [25, 30, 35, 40, 25, 30, 45]
})

# Identify duplicates based on all columns
duplicates_all = data[data.duplicated()]

# Identify duplicates based on specific columns
columns_to_check = ['ID', 'Name']
duplicates_specific = data[data.duplicated(subset=columns_to_check)]

# Print the original dataset
print("Original Dataset:")
print(data)
print()

# Print duplicates based on all columns
print("Duplicates (All Columns):")
print(duplicates_all)
print()

# Print duplicates based on specific columns
print("Duplicates (Specific Columns):")
print(duplicates_specific)
```

```
Original Dataset:
   ID   Name  Age
0   1   John   25
1   2   Jane   30
2   3   Mike   35
3   4   Emma   40
4   1   John   25
5   2   Jane   30
6   5  David   45

Duplicates (All Columns):
   ID  Name  Age
4   1  John   25
5   2  Jane   30

Duplicates (Specific Columns):
   ID  Name  Age
4   1  John   25
5   2  Jane   30
```

Removing duplicates

In [14]:
```
# Print the original dataset
print("Original Dataset:")
print(data)

# Remove duplicates based on all columns
data_duplicates_removed = data.drop_duplicates()

# Print the dataset after removing duplicates
print("\nDataset after Removing Duplicates:")
print(data_duplicates_removed)
```

```
Original Dataset:
   ID   Name  Age
0   1   John   25
1   2   Jane   30
2   3   Mike   35
3   4   Emma   40
4   1   John   25
5   2   Jane   30
6   5  David   45

Dataset after Removing Duplicates:
   ID   Name  Age
0   1   John   25
1   2   Jane   30
2   3   Mike   35
3   4   Emma   40
6   5  David   45
```

Keeping the first occurrence and removing subsequent duplicates

In [15]:
```
# Print the original dataset
print("Original Dataset:")
print(data)

# Keep the first occurrence and remove subsequent duplicates
data_duplicates_removed = data.drop_duplicates(keep='first')
print("\nKeep the first occurrence and remove subsequent duplicates :\n")
print(data_duplicates_removed)
```

```
Original Dataset:
   ID   Name  Age
0   1   John   25
1   2   Jane   30
2   3   Mike   35
3   4   Emma   40
4   1   John   25
5   2   Jane   30
6   5  David   45

Keep the first occurrence and remove subsequent duplicates :

   ID   Name  Age
0   1   John   25
1   2   Jane   30
2   3   Mike   35
3   4   Emma   40
6   5  David   45
```

Keeping the last occurrence and removing previous duplicates

In [16]:
```
# Print the original dataset
print("Original Dataset:")
print(data)

# Keep the last occurrence and remove previous duplicates
data_duplicates_removed = data.drop_duplicates(keep='last')
print("\nKeep the last occurrence and remove subsequent duplicates :\n")
print(data_duplicates_removed)
```

```
Original Dataset:
   ID   Name  Age
0   1   John   25
1   2   Jane   30
2   3   Mike   35
3   4   Emma   40
4   1   John   25
5   2   Jane   30
6   5  David   45

Keep the last occurrence and remove subsequent duplicates :

   ID   Name  Age
2   3   Mike   35
3   4   Emma   40
4   1   John   25
5   2   Jane   30
6   5  David   45
```

Handling Errors and Outliers

Visual Inspection and Statistical Analysis

Visualize the data using plots (e.g., scatter plots, histograms, box plots) to identify potential errors or outliers. Calculate summary statistics (e.g., mean, median, standard deviation) and identify values that deviate significantly from the expected range.

Winsorization

Winsorization replaces extreme values with less extreme values to reduce the impact of outliers. It involves capping or flooring the extreme values to a specified percentile.

In [17]:
```
from scipy.stats.mstats import winsorize

# Create sample dataset with outliers
data = pd.DataFrame({
    'A': [1, 2, 3, 4, 5, 6, 7, 8, 9, 10],
    'B': [10, 20, 30, 40, 50, 60, 70, 80, 90, 100]
})

# Print the original dataset
print("Original Dataset:")
print(data)

# Apply winsorization on column 'A'
winsorized_A = winsorize(data['A'], limits=(0.1, 0.1))

# Create a new DataFrame with winsorized values
data_winsorized = data.copy()
data_winsorized['A'] = winsorized_A

# Print the dataset after winsorization
print("\nDataset after Winsorization:")
print(data_winsorized)
```

```
Original Dataset:
    A    B
0   1   10
1   2   20
2   3   30
3   4   40
4   5   50
5   6   60
6   7   70
7   8   80
8   9   90
9  10  100

Dataset after Winsorization:
   A    B
0  2   10
1  2   20
2  3   30
3  4   40
4  5   50
5  6   60
6  7   70
7  8   80
8  9   90
9  9  100
```

Trimming

In [18]:
```
# Create sample dataset with outliers
data = pd.DataFrame({
    'A': [1, 2, 3, 4, 5, 6, 7, 8, 9, 10],
    'B': [11, 12, 13, 14, 15, 16, 17, 18, 19, 20]
})

# Add outliers to the dataset
data.loc[5, 'A'] = 100
data.loc[8, 'B'] = 200

# Print the original dataset
print("Original Dataset:")
print(data)

# Trim outliers using clipping
trimmed_data = data.clip(lower=data.quantile(0.05), upper=data.quantile(0.9

# Print the dataset after trimming
print("\nDataset after Trimming:")
print(trimmed_data)
```

```
Original Dataset:
     A    B
0    1   11
1    2   12
2    3   13
3    4   14
4    5   15
5  100   16
6    7   17
7    8   18
8    9  200
9   10   20

Dataset after Trimming:
       A       B
0   1.45   11.45
1   2.00   12.00
2   3.00   13.00
3   4.00   14.00
4   5.00   15.00
5  59.50   16.00
6   7.00   17.00
7   8.00   18.00
8   9.00  119.00
9  10.00   20.00
```

Statistical Methods

Statistical techniques such as z-score or modified z-score can be used to identify and handle outliers based on their deviation from the mean or median.

In [19]:
```
# Create sample dataset with outliers
data = pd.DataFrame({
    'A': [1, 2, 3, 4, 5, 200],
    'B': [10, 20, 30, 40, 50, 1000],
    'C': [100, 200, 300, 400, 500, 100],
})

# Print the original dataset
print("Original Dataset:")
print(data)

# Calculate the mean and standard deviation for each column
means = data.mean()
stds = data.std()

# Define the threshold for outliers (e.g., beyond 3 standard deviations)
threshold = 3

# Identify outliers using the z-score method
outliers = data[(np.abs((data - means) / stds) > threshold).any(axis=1)]

# Print the identified outliers
print("\nIdentified Outliers:")
print(outliers)

# Remove outliers using the z-score method
data_no_outliers = data[(np.abs((data - means) / stds) <= threshold).all(ax

# Print the dataset after removing outliers
print("\nDataset after Removing Outliers:")
print(data_no_outliers)
```

```
Original Dataset:
     A     B    C
0    1    10  100
1    2    20  200
2    3    30  300
3    4    40  400
4    5    50  500
5  200  1000  100

Identified Outliers:
Empty DataFrame
Columns: [A, B, C]
Index: []

Dataset after Removing Outliers:
     A     B    C
0    1    10  100
1    2    20  200
2    3    30  300
3    4    40  400
4    5    50  500
5  200  1000  100
```

Robust Statistical Methods

Robust statistical methods, like the median absolute deviation (MAD), are less sensitive to outliers compared to traditional methods that use mean and standard deviation.

In [20]:
```
# Create sample dataset with errors and outliers
data = pd.DataFrame({
    'A': [1, 2, 3, 4, 5, 6, 7],
    'B': [10, 20, 30, 40, 50, 60, 700],
    'C': [100, 200, 300, 4000, 500, 600, 700]
})

# Print the original dataset
print("Original Dataset:")
print(data)

# Calculate the median absolute deviation (MAD)
median = data.median()
mad = (data - median).abs().median()

# Define the threshold for outliers
threshold = 3.5

# Identify outliers based on the MAD
outliers_mad = (np.abs((data - median) / mad) > threshold).any(axis=1)

# Identify outliers based on the IQR
q1 = data.quantile(0.25)
q3 = data.quantile(0.75)
iqr = q3 - q1
lower_bound = q1 - 1.5 * iqr
upper_bound = q3 + 1.5 * iqr
outliers_iqr = ((data < lower_bound) | (data > upper_bound)).any(axis=1)

# Remove outliers from the dataset
data_cleaned = data[~(outliers_mad | outliers_iqr)]

# Print the dataset after removing outliers
print("\nDataset after Removing Outliers:")
print(data_cleaned)
```

```
Original Dataset:
   A    B     C
0  1   10   100
1  2   20   200
2  3   30   300
3  4   40  4000
4  5   50   500
5  6   60   600
6  7  700   700

Dataset after Removing Outliers:
   A   B    C
0  1  10  100
1  2  20  200
2  3  30  300
4  5  50  500
5  6  60  600
```

Data Transformation

Scaling/Normalization

Scaling or normalization is used to bring numerical features onto a similar scale, preventing any single feature from dominating the analysis due to its larger magnitude. Common scaling techniques include min-max scaling and standardization.

In [21]:
```
from sklearn.preprocessing import MinMaxScaler, StandardScaler

# Create a sample dataset
data = np.array([[10, 3, 5],
                 [20, 6, 9],
                 [5, 1, 2],
                 [15, 4, 7],
                 [8, 2, 4]])

print("Original Data :\n",data)
print("")
# Min-max scaling
scaler = MinMaxScaler()
data_scaled = scaler.fit_transform(data)
print("Scaled Data :\n",data_scaled)

print("")
# Standardization
scaler = StandardScaler()
data_standardized = scaler.fit_transform(data)
print("Standard Data :\n",data_standardized)
```

```
Original Data :
 [[10  3  5]
 [20  6  9]
 [ 5  1  2]
 [15  4  7]
 [ 8  2  4]]

Scaled Data :
 [[0.33333333 0.4        0.42857143]
 [1.         1.         1.        ]
 [0.         0.         0.        ]
 [0.66666667 0.6        0.71428571]
 [0.2        0.2        0.28571429]]

Standard Data :
 [[-0.30108397 -0.11624764 -0.16552118]
 [ 1.58069085  1.62746694  1.4896906 ]
 [-1.24197138 -1.27872403 -1.40693001]
 [ 0.63980344  0.46499055  0.66208471]
 [-0.67743894 -0.69748583 -0.57932412]]
```

Logarithmic Transformation

Logarithmic transformation is used to reduce the impact of large values and handle positively skewed distributions. It compresses the scale of the data and can make it more symmetric.

In [22]:
```
# Create a sample dataset
data = np.array([10, 100, 1000, 10000, 100000])

print("Original Data :\n",data)
# Logarithmic transformation
data_log_transformed = np.log1p(data)  # Log transformation with added 1 to
print("Log Transform Data :\n",data_log_transformed)
```

```
Original Data :
 [    10    100   1000  10000 100000]
Log Transform Data :
 [ 2.39789527  4.61512052  6.90875478  9.21044037 11.51293546]
```

Power Transformation

Power transformation, such as the Box-Cox transformation, is used to stabilize variance and normalize the data. It can be applied to handle data that violates assumptions of normality or homoscedasticity.

In [23]:
```
from scipy.stats import boxcox

# Create a sample dataset
data = np.array([1.0, 2.0, 3.0,4.0, 5.0, 6.0,7.0, 8.0, 9.0])

print("Original Data :\n",data)

# Box-Cox transformation
data_transformed, lambda_val = boxcox(data)
print("Power Transform Data :\n",data_transformed)
```

```
Original Data :
 [1. 2. 3. 4. 5. 6. 7. 8. 9.]
Power Transform Data :
 [0.         0.89887536 1.67448353 2.37952145 3.03633818 3.65711928
 4.2494518  4.81847233 5.36786648]
```

Binning/Discretization

Binning or discretization is used to convert continuous numerical features into categorical features by grouping values into bins or intervals. This can help simplify the analysis and handle non-linear relationships.

In [24]:
```
from sklearn.preprocessing import KBinsDiscretizer

# Create a sample dataset
data = pd.DataFrame({
    'age': [25, 37, 42, 18, 51, 29, 36, 22, 45, 33],
    'income': [50000, 75000, 100000, 25000, 80000, 60000, 90000, 40000, 950
})

print("Original Data :\n",data)

# Binning into equal-width bins
bin_discretizer = KBinsDiscretizer(n_bins=5, strategy='uniform', encode='or
data_binned = bin_discretizer.fit_transform(data)

print("Binning Data :\n",data_binned)
```

```
Original Data :
    age  income
0   25   50000
1   37   75000
2   42  100000
3   18   25000
4   51   80000
5   29   60000
6   36   90000
7   22   40000
8   45   95000
9   33   70000
Binning Data :
 [[1. 1.]
 [2. 3.]
 [3. 4.]
 [0. 0.]
 [4. 3.]
 [1. 2.]
 [2. 4.]
 [0. 1.]
 [4. 4.]
 [2. 3.]]

C:\Users\RACHIT\anaconda3\lib\site-packages\sklearn\preprocessing\_discre
tization.py:239: FutureWarning: In version 1.5 onwards, subsample=200_000
will be used by default. Set subsample explicitly to silence this warning
in the mean time. Set subsample=None to disable subsampling explicitly.
  warnings.warn(
```

Encoding Categorical Variables

Encoding categorical variables is necessary to represent them in a numerical format suitable for analysis

One-Hot Encoding

One-hot encoding is used to represent categorical variables as binary vectors. It creates new binary columns for each unique category in the variable, with a value of 1 indicating the

In [25]:
```
from sklearn.preprocessing import OneHotEncoder

# Create a sample dataset
data = pd.DataFrame({
    'Color': ['Red', 'Blue', 'Green', 'Red', 'Yellow'],
    'Size': ['Small', 'Large', 'Medium', 'Small', 'Medium'],
    'Shape': ['Circle', 'Rectangle', 'Triangle', 'Circle', 'Square']
})

print("Original Data :\n",data)

# One-hot encoding
encoder = OneHotEncoder()
data_encoded = encoder.fit_transform(data)

print("OneHot Encoder Data :\n",data_encoded)
```

```
Original Data :
     Color    Size      Shape
0     Red   Small     Circle
1    Blue   Large  Rectangle
2   Green  Medium   Triangle
3     Red   Small     Circle
4  Yellow  Medium     Square
OneHot Encoder Data :
   (0, 2)       1.0
  (0, 6)        1.0
  (0, 7)        1.0
  (1, 0)        1.0
  (1, 4)        1.0
  (1, 8)        1.0
  (2, 1)        1.0
  (2, 5)        1.0
  (2, 10)       1.0
  (3, 2)        1.0
  (3, 6)        1.0
  (3, 7)        1.0
  (4, 3)        1.0
  (4, 5)        1.0
  (4, 9)        1.0
```

Label Encoding

Label encoding assigns a numerical label to each unique category in a categorical variable. It is suitable for ordinal variables where the order of the categories matters. Each category is assigned a unique integer value.

In [26]:
```
from sklearn.preprocessing import LabelEncoder

# Create a sample dataset
data = pd.DataFrame({
    'Category': ['Red', 'Blue', 'Green', 'Green', 'Red', 'Blue'],
    'Size': ['Small', 'Large', 'Medium', 'Large', 'Small', 'Medium'],
    'Label': ['A', 'B', 'A', 'C', 'B', 'C']
})

print("Original Data :\n",data)


# LabelEncoder
encoder = LabelEncoder()
data["Encoded label"] = encoder.fit_transform(data['Label'])

print("Label Encoder data :\n",data)
```

```
Original Data :
   Category    Size Label
0       Red   Small     A
1      Blue   Large     B
2     Green  Medium     A
3     Green   Large     C
4       Red   Small     B
5      Blue  Medium     C
Label Encoder data :
   Category    Size Label  Encoded label
0       Red   Small     A              0
1      Blue   Large     B              1
2     Green  Medium     A              0
3     Green   Large     C              2
4       Red   Small     B              1
5      Blue  Medium     C              2
```

Ordinal Encoding

Ordinal encoding is similar to label encoding but is used when the categories have an inherent order or rank. It assigns integer labels to categories based on their order, preserving the ordinal relationship between them

In [27]:
```
from sklearn.preprocessing import OrdinalEncoder

# Create a sample dataset
data = [['Red', 'Small'], ['Blue', 'Large'], ['Green', 'Medium'],
        ['Green', 'Large'], ['Red', 'Small'], ['Blue', 'Medium']]

print("Original Data :\n",data)

# Initialize OrdinalEncoder
ordinal_encoder = OrdinalEncoder()

# Fit and transform the data
encoded_data = ordinal_encoder.fit_transform(data)

# Print the encoded data
print("Encoded Data :\n",encoded_data)
```

```
Original Data :
 [['Red', 'Small'], ['Blue', 'Large'], ['Green', 'Medium'], ['Green', 'La
rge'], ['Red', 'Small'], ['Blue', 'Medium']]
Encoded Data :
 [[2. 2.]
 [0. 0.]
 [1. 1.]
 [1. 0.]
 [2. 2.]
 [0. 1.]]
```

Binary Encoding

Binary encoding represents each unique category with a binary code. It involves converting the categories to binary representation and then creating binary columns for each digit in the binary code.

In [28]:
```
import category_encoders as ce

# Create a sample dataset
data = pd.DataFrame({
    'Category': ['Red', 'Blue', 'Green', 'Green', 'Red', 'Blue'],
    'Size': ['Small', 'Large', 'Medium', 'Large', 'Small', 'Medium'],
    'Label': ['A', 'B', 'A', 'C', 'B', 'C']
})

print("Original Data :\n",data)

# Specify the columns to be binary encoded
columns_to_encode = ['Category', 'Size']

# Initialize BinaryEncoder
binary_encoder = ce.BinaryEncoder(cols=columns_to_encode)

# Fit and transform the data
encoded_data = binary_encoder.fit_transform(data)

# Print the encoded data
print("Encoded Data :\n",encoded_data)
```

```
Original Data :
   Category    Size Label
0       Red   Small     A
1      Blue   Large     B
2     Green  Medium     A
3     Green   Large     C
4       Red   Small     B
5      Blue  Medium     C
Encoded Data :
    Category_0  Category_1  Size_0  Size_1 Label
0           0           1       0       1     A
1           1           0       1       0     B
2           1           1       1       1     A
3           1           1       1       0     C
4           0           1       0       1     B
5           1           0       1       1     C
```

Frequency Encoding

Frequency encoding replaces categories with their frequency of occurrence in the dataset. It assigns a numerical value representing the proportion of observations that belong to each category

In [29]:
```
# Create a sample dataset
data = pd.DataFrame({
    'Category': ['Red', 'Blue', 'Green', 'Green', 'Red', 'Blue'],
    'Size': ['Small', 'Large', 'Medium', 'Large', 'Small', 'Medium'],
    'Label': ['A', 'B', 'A', 'C', 'B', 'C']
})

print("Original Data :\n",data)

# Calculate the frequency of each category
category_freq = data['Category'].value_counts(normalize=True)
size_freq = data['Size'].value_counts(normalize=True)
label_freq = data['Label'].value_counts(normalize=True)

# Create frequency encoding mapping dictionaries
category_mapping = category_freq.to_dict()
size_mapping = size_freq.to_dict()
label_mapping = label_freq.to_dict()

# Perform frequency encoding
data['Category_Frequency_Encoded'] = data['Category'].map(category_mapping)
data['Size_Frequency_Encoded'] = data['Size'].map(size_mapping)

print("Frequency Encoded Data :\n",data)
```

```
Original Data :
   Category    Size Label
0       Red   Small     A
1      Blue   Large     B
2     Green  Medium     A
3     Green   Large     C
4       Red   Small     B
5      Blue  Medium     C
Frequency Encoded Data :
   Category    Size Label  Category_Frequency_Encoded  Size_Frequency_Enc
oded
0       Red   Small     A                    0.333333                0.333
333
1      Blue   Large     B                    0.333333                0.333
333
2     Green  Medium     A                    0.333333                0.333
333
3     Green   Large     C                    0.333333                0.333
333
4       Red   Small     B                    0.333333                0.333
333
5      Blue  Medium     C                    0.333333                0.333
333
```

Feature Selection/Extraction

Univariate Feature Selection

This method evaluates each feature independently based on statistical measures like chi-square, ANOVA F-value, or mutual information with the target variable. It selects the top-k

In [30]:
```
from sklearn.feature_selection import SelectKBest, f_classif
from sklearn.preprocessing import OneHotEncoder

# Create a sample dataset
np.random.seed(0)
data = pd.DataFrame({
    'Feature1': np.random.rand(100),
    'Feature2': np.random.randint(0, 10, 100),
    'Feature3': np.random.normal(0, 1, 100),
    'Feature4': np.random.choice(['A', 'B', 'C'], 100),
    'Target': np.random.randint(0, 2, 100)
})

# Print the original dataset
print("Original Dataset :\n",data.head())

# Perform one-hot encoding for the categorical feature
onehot_encoder = OneHotEncoder(sparse=False, handle_unknown='ignore')
encoded_feature4 = onehot_encoder.fit_transform(data[['Feature4']])

# Create a new DataFrame with the encoded feature
encoded_data = pd.concat([data.drop('Feature4', axis=1), pd.DataFrame(encoc

# Split the dataset into X (features) and y (target variable)
X = encoded_data.drop('Target', axis=1)
y = encoded_data['Target']

# Print the encoded dataset
print("Encoded Dataset :\n",encoded_data.head())

# Assuming you have X as your feature matrix and y as the target variable

# Create an instance of SelectKBest with the desired scoring function
selector = SelectKBest(score_func=f_classif, k=5)  # Select top 5 features

# Apply feature selection
selected_features = selector.fit_transform(X, y)

# Get the selected feature indices
selected_indices = selector.get_support(indices=True)

# Print the selected feature indices and their corresponding names
selected_feature_names = [X.columns[i] for i in selected_indices]
print("Selected Features:")
for feature_name in selected_feature_names:
    print(feature_name)
```

```
Original Dataset :
    Feature1  Feature2  Feature3 Feature4  Target
0  0.548814         2 -0.651715        A       0
1  0.715189         3  0.000038        A       0
2  0.602763         2  0.889754        C       0
3  0.544883         0 -1.156281        B       1
4  0.423655         8  0.397217        C       1
Encoded Dataset :
    Feature1  Feature2  Feature3  Target    A    B    C
0  0.548814         2 -0.651715       0  1.0  0.0  0.0
1  0.715189         3  0.000038       0  1.0  0.0  0.0
2  0.602763         2  0.889754       0  0.0  0.0  1.0
3  0.544883         0 -1.156281       1  0.0  1.0  0.0
4  0.423655         8  0.397217       1  0.0  0.0  1.0
Selected Features:
Feature1
Feature2
A
B
C

C:\Users\RACHIT\anaconda3\lib\site-packages\sklearn\preprocessing\_encode
rs.py:972: FutureWarning: `sparse` was renamed to `sparse_output` in vers
ion 1.2 and will be removed in 1.4. `sparse_output` is ignored unless you
leave `sparse` to its default value.
  warnings.warn(
```

Recursive Feature Elimination (RFE)

RFE recursively eliminates features by fitting a model and discarding the least important feature at each iteration until the desired number of features is reached.

In [31]:
```
from sklearn.feature_selection import RFE
from sklearn.linear_model import LogisticRegression
from sklearn.preprocessing import OneHotEncoder

# Create a sample dataset
np.random.seed(0)
data = pd.DataFrame({
    'Feature1': np.random.rand(100),
    'Feature2': np.random.randint(0, 10, 100),
    'Feature3': np.random.normal(0, 1, 100),
    'Feature4': np.random.choice(['A', 'B', 'C'], 100),
    'Target': np.random.randint(0, 2, 100)
})

# Perform one-hot encoding for the categorical feature
onehot_encoder = OneHotEncoder(sparse=False, handle_unknown='ignore')
encoded_feature4 = onehot_encoder.fit_transform(data[['Feature4']])

# Create a new DataFrame with the encoded feature
encoded_data = pd.concat([data.drop('Feature4', axis=1), pd.DataFrame(encoc

# Split the dataset into X (features) and y (target variable)
X = encoded_data.drop('Target', axis=1)
y = encoded_data['Target']

# Initialize the estimator (model) for feature importance estimation
estimator = LogisticRegression()

# Initialize the Recursive Feature Elimination (RFE) object
rfe = RFE(estimator, n_features_to_select=3)  # Select the top 2 features

# Perform RFE feature selection
selected_features = rfe.fit_transform(X, y)

# Get the selected feature indices
selected_indices = rfe.get_support(indices=True)

# Print the selected feature indices and their corresponding names
selected_feature_names = [X.columns[i] for i in selected_indices]
print("Selected Features:")
for feature_name in selected_feature_names:
    print(feature_name)
```

```
Selected Features:
Feature1
B
C

C:\Users\RACHIT\anaconda3\lib\site-packages\sklearn\preprocessing\_encode
rs.py:972: FutureWarning: `sparse` was renamed to `sparse_output` in vers
ion 1.2 and will be removed in 1.4. `sparse_output` is ignored unless you
leave `sparse` to its default value.
  warnings.warn(
```

L1 Regularization (Lasso)

L1 regularization can be used to induce sparsity in the coefficients of a linear model, thereby automatically selecting a subset of features.

In [32]:
```
from sklearn.linear_model import Lasso

# Create a sample dataset
np.random.seed(0)
data = pd.DataFrame({
    'Feature1': np.random.rand(100),
    'Feature2': np.random.randint(0, 10, 100),
    'Feature3': np.random.normal(0, 1, 100),
    'Feature4': np.random.choice(['A', 'B', 'C'], 100),
    'Target': np.random.randint(0, 2, 100)
})

# Encode categorical feature
data_encoded = pd.get_dummies(data, columns=['Feature4'], drop_first=True)

# Split the dataset into X (features) and y (target variable)
X = data_encoded.drop('Target', axis=1)
y = data_encoded['Target']

# Initialize the Lasso estimator
lasso = Lasso(alpha=0.1)  # Set the regularization strength (alpha)

# Fit the Lasso model to the data
lasso.fit(X, y)

# Get the feature importance scores
feature_importances = lasso.coef_

# Get the selected feature indices (non-zero coefficients)
selected_indices = np.nonzero(feature_importances)[0]

# Print the selected feature indices and their corresponding names
selected_feature_names = X.columns[selected_indices]
print("Selected Features:")
for feature_name in selected_feature_names:
    print(feature_name)
```

```
Selected Features:
Feature2
```

Recursive Feature Addition (RFA):

RFA is the opposite of RFE. It starts with an empty set of features and iteratively adds one feature at a time based on a selected evaluation metric until a stopping criterion is met.

In [33]:
```
from sklearn.feature_selection import RFECV
from sklearn.linear_model import LogisticRegression
from sklearn.preprocessing import OneHotEncoder

# Create a sample dataset
np.random.seed(0)
data = pd.DataFrame({
    'Feature1': np.random.rand(100),
    'Feature2': np.random.randint(0, 10, 100),
    'Feature3': np.random.normal(0, 1, 100),
    'Feature4': np.random.choice(['A', 'B', 'C'], 100),
    'Target': np.random.randint(0, 2, 100)
})

# Perform one-hot encoding for the categorical feature
onehot_encoder = OneHotEncoder(sparse=False, handle_unknown='ignore')
encoded_feature4 = onehot_encoder.fit_transform(data[['Feature4']])

# Create a new DataFrame with the encoded feature
encoded_data = pd.concat([data.drop('Feature4', axis=1), pd.DataFrame(encoc

# Split the dataset into X (features) and y (target variable)
X = encoded_data.drop('Target', axis=1)
y = encoded_data['Target']

# Initialize the estimator (model) for feature importance estimation
estimator = LogisticRegression()

# Initialize the Recursive Feature Addition (RFA) object
rfa = RFECV(estimator, cv=5)  # Use 5-fold cross-validation

# Perform RFA feature selection
selected_features = rfa.fit_transform(X, y)

# Get the selected feature indices
selected_indices = rfa.get_support(indices=True)

# Print the selected feature indices and their corresponding names
selected_feature_names = [X.columns[i] for i in selected_indices]
print("Selected Features:")
for feature_name in selected_feature_names:
    print(feature_name)
```

```
C:\Users\RACHIT\anaconda3\lib\site-packages\sklearn\preprocessing\_encode
rs.py:972: FutureWarning: `sparse` was renamed to `sparse_output` in vers
ion 1.2 and will be removed in 1.4. `sparse_output` is ignored unless you
leave `sparse` to its default value.
  warnings.warn(

Selected Features:
C
```

SelectFromModel:

SelectFromModel is a meta-transformer that selects features based on the importance scores provided by a base model. It allows you to specify a threshold to control the number of selected features.

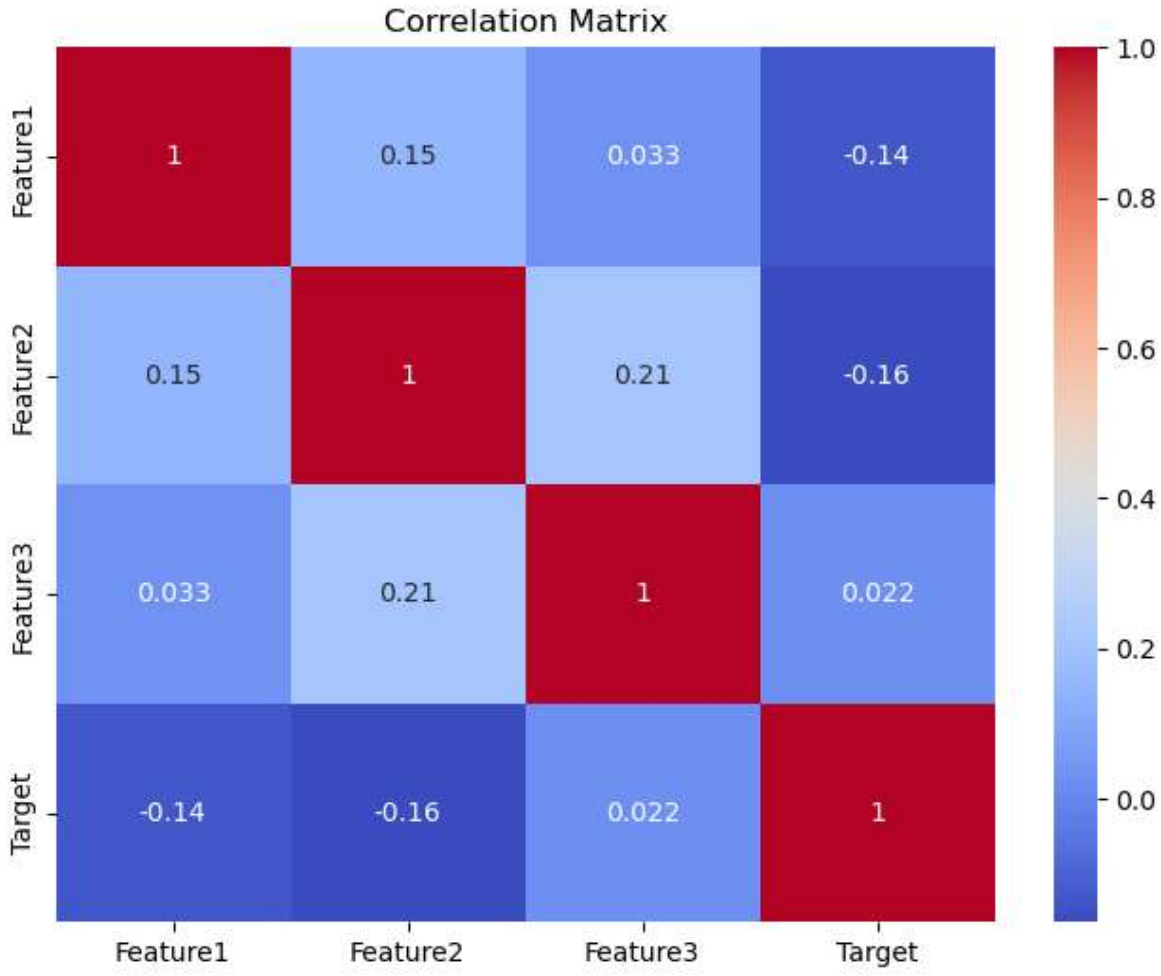
In [34]:
```
from sklearn.feature_selection import SelectFromModel
from sklearn.ensemble import RandomForestClassifier

# Create a sample dataset
np.random.seed(0)
data = pd.DataFrame({
    'Feature1': np.random.rand(100),
    'Feature2': np.random.randint(0, 10, 100),
    'Feature3': np.random.normal(0, 1, 100),
    'Target': np.random.randint(0, 2, 100)
})

# Split the dataset into X (features) and y (target variable)
X = data.drop('Target', axis=1)
y = data['Target']

# Initialize the estimator (model) for feature importance estimation
estimator = RandomForestClassifier()

# Initialize the SelectFromModel object
selector = SelectFromModel(estimator)

# Perform feature selection
selected_features = selector.fit_transform(X, y)

# Get the selected feature indices
selected_indices = selector.get_support(indices=True)

# Print the selected feature indices and their corresponding names
selected_feature_names = [X.columns[i] for i in selected_indices]
print("Selected Features:")
for feature_name in selected_feature_names:
    print(feature_name)
```

```
Selected Features:
Feature1
Feature3
```

Correlation Matrix

The correlation matrix can be used to identify highly correlated features and select only one representative feature from each correlated group.

In [35]:
```
# Create a sample dataset
np.random.seed(0)
data = pd.DataFrame({
    'Feature1': np.random.rand(100),
    'Feature2': np.random.randint(0, 10, 100),
    'Feature3': np.random.normal(0, 1, 100),
    'Feature4': np.random.choice(['A', 'B', 'C'], 100),
    'Target': np.random.randint(0, 2, 100)
})

# Calculate the correlation matrix
corr_matrix = data.corr()

# Plot the correlation matrix as a heatmap
plt.figure(figsize=(8, 6))
sns.heatmap(corr_matrix, annot=True, cmap='coolwarm')
plt.title('Correlation Matrix')
plt.show()

# Set the threshold for correlation value
threshold = 0.3

# Get the absolute correlation values above the threshold
high_corr_features = corr_matrix[abs(corr_matrix) > threshold].stack().rese

# Filter out the self-correlations (diagonal elements) and duplicate correl
high_corr_features = high_corr_features[high_corr_features['level_0'] != hi

# Get the unique pairs of highly correlated features
unique_high_corr_features = high_corr_features.groupby('level_0')['level_1'
unique_high_corr_features.columns = ['Feature', 'Highly Correlated Features

# Print the highly correlated feature pairs
print("Highly Correlated Features:")
print(unique_high_corr_features)
```




```
Highly Correlated Features:
Empty DataFrame
Columns: [Feature, Highly Correlated Features]
Index: []
```

Principal Component Analysis (PCA)

PCA is a popular technique that transforms the original features into a new set of uncorrelated variables called principal components. It aims to capture the maximum variance in the data.

In [36]:
```
from sklearn.decomposition import PCA
from sklearn.preprocessing import StandardScaler

# Create a sample dataset
np.random.seed(0)
data = pd.DataFrame({
    'Feature1': np.random.rand(100),
    'Feature2': np.random.randint(0, 10, 100),
    'Feature3': np.random.normal(0, 1, 100),
    'Target': np.random.randint(0, 2, 100)
})

# Split the dataset into X (features) and y (target variable)
X = data.drop('Target', axis=1)
y = data['Target']

# Perform feature scaling using StandardScaler
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)

# Initialize PCA object
pca = PCA(n_components=2)  # Set the number of principal components to keep

# Perform PCA on the scaled features
X_pca = pca.fit_transform(X_scaled)

# Create a new DataFrame with the principal components
principal_components = pd.DataFrame(data=X_pca, columns=['PC1', 'PC2'])

# Print the explained variance ratio
print("Explained Variance Ratio:")
print(pca.explained_variance_ratio_)

# Print the principal components dataframe
print("Principal Components:")
print(principal_components.head())
```

```
Explained Variance Ratio:
[0.42444987 0.32299404]
Principal Components:
        PC1       PC2
0 -0.941287  0.641507
1 -0.078539  0.726065
2  0.014491 -0.083057
3 -1.706534  0.946253
4  0.879801 -0.394987
```

Random Forest Importance

Random Forest Importance assigns an importance score to each feature based on how much the feature contributes to the accuracy of a random forest model.

In [37]:
```
from sklearn.ensemble import RandomForestClassifier

# Create a sample dataset
np.random.seed(0)
data = pd.DataFrame({
    'Feature1': np.random.rand(100),
    'Feature2': np.random.randint(0, 10, 100),
    'Feature3': np.random.normal(0, 1, 100),
    'Feature4': np.random.choice(['A', 'B', 'C'], 100),
    'Target': np.random.randint(0, 2, 100)
})

# Perform one-hot encoding for the categorical feature
encoded_data = pd.get_dummies(data, columns=['Feature4'])

# Split the dataset into X (features) and y (target variable)
X = encoded_data.drop('Target', axis=1)
y = encoded_data['Target']

# Initialize the Random Forest Classifier
rf_classifier = RandomForestClassifier(n_estimators=100, random_state=0)

# Fit the classifier to the data
rf_classifier.fit(X, y)

# Get feature importances from the classifier
importances = rf_classifier.feature_importances_

# Create a DataFrame of feature importances
feature_importances = pd.DataFrame({'Feature': X.columns, 'Importance': imp
feature_importances = feature_importances.sort_values(by='Importance', asce

# Print the feature importances
print("Feature Importances:")
print(feature_importances)
```

```
Feature Importances:
      Feature  Importance
0    Feature1    0.347744
2    Feature3    0.312635
1    Feature2    0.235444
5  Feature4_C    0.042192
4  Feature4_B    0.038426
3  Feature4_A    0.023559
```

Independent Component Analysis (ICA)

ICA is used to separate a multivariate signal into additive subcomponents by assuming the independence of the components. It can be used for feature extraction in scenarios where the underlying sources are statistically independent

In [38]:
```
from sklearn.decomposition import FastICA

# Create a sample dataset
np.random.seed(0)
data = pd.DataFrame({
    'Feature1': np.random.rand(100),
    'Feature2': np.random.randint(0, 10, 100),
    'Feature3': np.random.normal(0, 1, 100),
    'Feature4': np.random.choice(['A', 'B', 'C'], 100),
    'Target': np.random.randint(0, 2, 100)
})

# Split the dataset into X (features) and y (target variable)
X = data.drop('Target', axis=1)
y = data['Target']

# Perform one-hot encoding for the categorical feature
X_encoded = pd.get_dummies(X, columns=['Feature4'])

# Initialize the Independent Component Analysis (ICA) object
ica = FastICA(n_components=2)  # Select the top 2 independent components

# Perform ICA for feature extraction
X_ica = ica.fit_transform(X_encoded)

# Create a DataFrame with the extracted features
ica_data = pd.DataFrame(X_ica, columns=['ICA1', 'ICA2'])

# Print the extracted features
print("ICA Features:")
print(ica_data.head())
```

```
ICA Features:
       ICA1      ICA2
0 -0.505701  1.004970
1  0.049168  0.628112
2  1.198251  0.925680
3 -0.748373  1.720097
4  0.178875 -1.093848
```

Non-Negative Matrix Factorization (NMF)

NMF is a technique that factorizes the data matrix into non-negative matrices, which can be interpreted as parts-based representation of the original data. It can be used for feature extraction when the features are non-negative

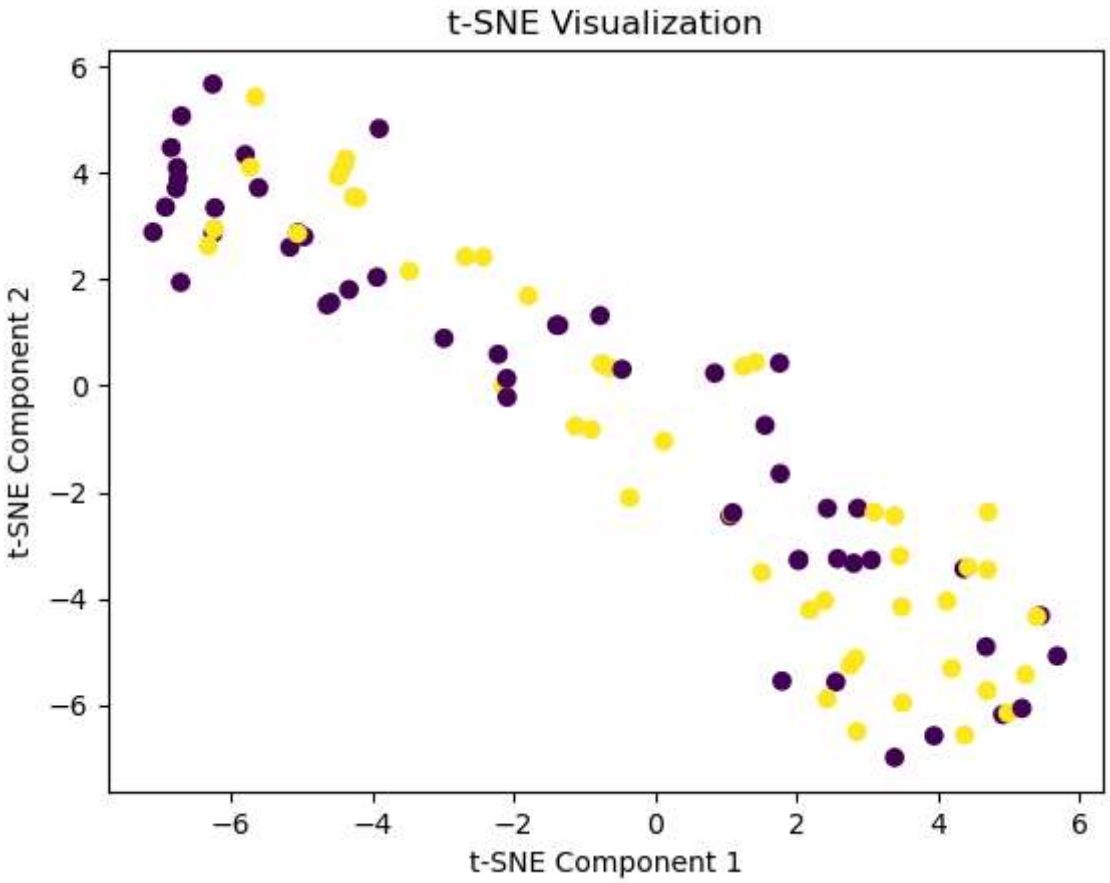
In [39]:
```
from sklearn.decomposition import NMF
from sklearn.preprocessing import MinMaxScaler

# Create a sample dataset
np.random.seed(0)
data = pd.DataFrame({
    'Feature1': np.random.randn(100),
    'Feature2': np.random.randint(0, 10, 100),
    'Feature3': np.random.normal(0, 1, 100),
    'Feature4': np.random.choice(['A', 'B', 'C'], 100),
    'Target': np.random.randint(0, 2, 100)
})

# Perform one-hot encoding for the categorical feature
encoded_data = pd.get_dummies(data, columns=['Feature4'])

# Apply log transformation and shift to make the values non-negative
scaler = MinMaxScaler()
X = np.log1p(scaler.fit_transform(encoded_data.drop('Target', axis=1)) + 1e

# Split the dataset into X (features) and y (target variable)
y = encoded_data['Target']

# Initialize the NMF object
nmf = NMF(n_components=2)  # Select the top 2 components

# Perform NMF feature selection
selected_features = nmf.fit_transform(X)

# Get the selected feature indices
selected_indices = np.argsort(nmf.components_)[-2:]

# Print the selected feature indices and their corresponding names
selected_feature_names = encoded_data.columns[selected_indices]
print("Selected Features:")
for feature_name in selected_feature_names:
    print(feature_name)
```

```
Selected Features:
['Feature4_B' 'Target' 'Feature1' 'Feature3' 'Feature4_A' 'Feature2']
['Feature4_A' 'Target' 'Feature2' 'Feature1' 'Feature3' 'Feature4_B']

C:\Users\RACHIT\AppData\Local\Temp\ipykernel_1616\2732190983.py:34: Futur
eWarning: Support for multi-dimensional indexing (e.g. `obj[:, None]`) is
deprecated and will be removed in a future version.  Convert to a numpy a
rray before indexing instead.
  selected_feature_names = encoded_data.columns[selected_indices]
```

t-distributed Stochastic Neighbor Embedding (t-SNE)

t-SNE is a technique commonly used for data visualization and exploratory analysis. It maps high-dimensional data to a lower-dimensional space while preserving local relationships and capturing non-linear structures. t-SNE is particularly useful for visualizing clusters or identifying patterns in the data.

In [40]:
```
from sklearn.manifold import TSNE
import matplotlib.pyplot as plt

# Create a sample dataset
np.random.seed(0)
data = pd.DataFrame({
    'Feature1': np.random.rand(100),
    'Feature2': np.random.randint(0, 10, 100),
    'Feature3': np.random.normal(0, 1, 100),
    'Feature4': np.random.choice(['A', 'B', 'C'], 100),
    'Target': np.random.randint(0, 2, 100)
})

# Perform one-hot encoding for the categorical feature
encoded_data = pd.get_dummies(data, columns=['Feature4'])

# Split the dataset into X (features) and y (target variable)
X = encoded_data.drop('Target', axis=1)
y = encoded_data['Target']

# Initialize the t-SNE object
tsne = TSNE(n_components=2)

# Perform t-SNE transformation
X_tsne = tsne.fit_transform(X)

# Visualize the t-SNE plot
plt.scatter(X_tsne[:, 0], X_tsne[:, 1], c=y)
plt.xlabel('t-SNE Component 1')
plt.ylabel('t-SNE Component 2')
plt.title('t-SNE Visualization')
plt.show()
```



Linear Discriminant Analysis (LDA)

LDA is a supervised dimensionality reduction technique that aims to find a linear combination of features that maximizes class separability. It considers class labels in addition to the data itself, making it useful for classification tasks. LDA seeks to find discriminative features that maximize between-class variance while minimizing within-class variance.

In [41]:
```
from sklearn.discriminant_analysis import LinearDiscriminantAnalysis

# Create a sample dataset
np.random.seed(0)
data = pd.DataFrame({
    'Feature1': np.random.rand(100),
    'Feature2': np.random.randint(0, 10, 100),
    'Feature3': np.random.normal(0, 1, 100),
    'Feature4': np.random.choice(['A', 'B', 'C'], 100),
    'Target': np.random.randint(0, 3, 100)  # Increase the number of classe
})

# Perform one-hot encoding for the categorical feature
encoded_data = pd.get_dummies(data, columns=['Feature4'])

# Split the dataset into X (features) and y (target variable)
X = encoded_data.drop('Target', axis=1)
y = encoded_data['Target']

# Initialize the LDA object
lda = LinearDiscriminantAnalysis(n_components=2)  # Set the n_components va

# Perform LDA feature selection
selected_features = lda.fit_transform(X, y)

# Print the selected feature shape
print("Selected Features Shape:", selected_features.shape)
```

```
Selected Features Shape: (100, 2)
```

Manifold Learning (e.g., Isomap, Locally Linear Embedding)

Manifold Learning methods aim to capture the underlying non-linear structure of the data by mapping it to a lower-dimensional space. Techniques like Isomap and Locally Linear Embedding (LLE) preserve the local relationships of the data points and can reveal intricate structures in the data that may not be captured by linear methods like PCA.

In [42]:
```
from sklearn.manifold import Isomap, LocallyLinearEmbedding

# Create a sample dataset
np.random.seed(0)
data = pd.DataFrame({
    'Feature1': np.random.rand(100),
    'Feature2': np.random.randint(0, 10, 100),
    'Feature3': np.random.normal(0, 1, 100),
    'Target': np.random.randint(0, 2, 100)
})

# Split the dataset into X (features) and y (target variable)
X = data.drop('Target', axis=1)
y = data['Target']

# Initialize the Isomap object
isomap = Isomap(n_components=2)  # Set the number of components to 2

# Perform Isomap dimensionality reduction
isomap_features = isomap.fit_transform(X)

# Initialize the LLE object
lle = LocallyLinearEmbedding(n_components=2)  # Set the number of component

# Perform LLE dimensionality reduction
lle_features = lle.fit_transform(X)

# Print the shape of the transformed features
print("Isomap Transformed Features Shape:", isomap_features.shape)
print("LLE Transformed Features Shape:", lle_features.shape)
```

```
Isomap Transformed Features Shape: (100, 2)
LLE Transformed Features Shape: (100, 2)
```

Handling Imbalanced Data

Resampling Techniques

Oversampling: Oversampling increases the number of instances in the minority class by replicating existing instances or generating synthetic samples. Common oversampling methods include Random Oversampling, SMOTE (Synthetic Minority Over-sampling Technique), and ADASYN (Adaptive Synthetic Sampling).

In [43]:
```
from sklearn.datasets import make_classification
from sklearn.utils import resample

# Create a sample imbalanced dataset
np.random.seed(0)
X, y = make_classification(
    n_samples=1000,
    n_features=5,
    n_informative=3,
    n_redundant=2,
    n_classes=2,
    weights=[0.9, 0.1],  # Imbalanced class distribution
)

# Convert the dataset into a DataFrame
data = pd.DataFrame(X, columns=['Feature1', 'Feature2', 'Feature3', 'Featur
data['Target'] = y

# Check the class distribution
print("Class Distribution:")
print(data['Target'].value_counts())

# Perform random oversampling
oversampled_data = resample(data[data['Target'] == 1], n_samples=data['Targ
oversampled_data = pd.concat([data[data['Target'] == 0], oversampled_data])

# Perform random undersampling
undersampled_data = resample(data[data['Target'] == 0], n_samples=data['Tar
undersampled_data = pd.concat([data[data['Target'] == 1], undersampled_data

# Check the class distribution after resampling
print("\nClass Distribution after Random Oversampling:")
print(oversampled_data['Target'].value_counts())

print("\nClass Distribution after Random Undersampling:")
print(undersampled_data['Target'].value_counts())
```

```
Class Distribution:
0    898
1    102
Name: Target, dtype: int64

Class Distribution after Random Oversampling:
0    898
1    898
Name: Target, dtype: int64

Class Distribution after Random Undersampling:
1    102
0    102
Name: Target, dtype: int64
```

In [44]:
```
from sklearn.datasets import make_classification
from imblearn.over_sampling import RandomOverSampler, SMOTE, ADASYN

# Create a sample imbalanced dataset
np.random.seed(0)
X, y = make_classification(
    n_samples=1000,
    n_features=5,
    n_informative=3,
    n_redundant=2,
    n_classes=2,
    weights=[0.9, 0.1],  # Imbalanced class distribution
)

# Convert the dataset into a DataFrame
data = pd.DataFrame(X, columns=['Feature1', 'Feature2', 'Feature3', 'Featur
data['Target'] = y

# Check the class distribution
print("Class Distribution:")
print(data['Target'].value_counts())

# Random Oversampling
ros = RandomOverSampler(random_state=0)
X_ros, y_ros = ros.fit_resample(data.drop('Target', axis=1), data['Target']

# SMOTE
smote = SMOTE(random_state=0)
X_smote, y_smote = smote.fit_resample(data.drop('Target', axis=1), data['Ta

# ADASYN
adasyn = ADASYN(random_state=0)
X_adasyn, y_adasyn = adasyn.fit_resample(data.drop('Target', axis=1), data[

# Check the class distribution after resampling
print("\nClass Distribution after Random Oversampling:")
print(pd.Series(y_ros).value_counts())

print("\nClass Distribution after SMOTE:")
print(pd.Series(y_smote).value_counts())

print("\nClass Distribution after ADASYN:")
print(pd.Series(y_adasyn).value_counts())
```

```
Class Distribution:
0    898
1    102
Name: Target, dtype: int64

Class Distribution after Random Oversampling:
1    898
0    898
Name: Target, dtype: int64

Class Distribution after SMOTE:
1    898
0    898
Name: Target, dtype: int64

Class Distribution after ADASYN:
1    900
0    898
Name: Target, dtype: int64
```

Undersampling

Undersampling reduces the number of instances in the majority class by randomly removing samples. It can be effective when the dataset is large, and the majority class has many redundant instances. Common undersampling methods include Random Undersampling, NearMiss, and Tomek Links.

In [45]:
```
from sklearn.datasets import make_classification
from imblearn.under_sampling import RandomUnderSampler, NearMiss
from imblearn.under_sampling import TomekLinks

# Create a sample imbalanced dataset
np.random.seed(0)
X, y = make_classification(
    n_samples=1000,
    n_features=5,
    n_informative=3,
    n_redundant=2,
    n_classes=2,
    weights=[0.9, 0.1],  # Imbalanced class distribution
)

# Convert the dataset into a DataFrame
data = pd.DataFrame(X, columns=['Feature1', 'Feature2', 'Feature3', 'Featur
data['Target'] = y

# Check the class distribution
print("Class Distribution:")
print(data['Target'].value_counts())

# Random Undersampling
rus = RandomUnderSampler(random_state=0)
X_rus, y_rus = rus.fit_resample(data.drop('Target', axis=1), data['Target']

# NearMiss
nm = NearMiss(version=1)
X_nm, y_nm = nm.fit_resample(data.drop('Target', axis=1), data['Target'])

# TomekLinks
tl = TomekLinks()
X_tl, y_tl = tl.fit_resample(data.drop('Target', axis=1), data['Target'])

# Check the class distribution after undersampling
print("\nClass Distribution after Random Undersampling:")
print(pd.Series(y_rus).value_counts())

print("\nClass Distribution after NearMiss:")
print(pd.Series(y_nm).value_counts())

print("\nClass Distribution after TomekLinks:")
print(pd.Series(y_tl).value_counts())
```

```
Class Distribution:
0    898
1    102
Name: Target, dtype: int64

Class Distribution after Random Undersampling:
0    102
1    102
Name: Target, dtype: int64

Class Distribution after NearMiss:
0    102
1    102
Name: Target, dtype: int64

Class Distribution after TomekLinks:
0    886
1    102
Name: Target, dtype: int64
```

Class Weighting

Assigning class weights can be used to give more importance to the minority class during model training. This approach is applicable to algorithms that accept class weights as a parameter, such as decision trees or support vector machines.

In [46]:
```
from sklearn.datasets import make_classification
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import classification_report

# Create a sample imbalanced dataset
np.random.seed(0)
X, y = make_classification(
    n_samples=1000,
    n_features=5,
    n_informative=3,
    n_redundant=2,
    n_classes=2,
    weights=[0.9, 0.1],  # Imbalanced class distribution
)

# Convert the dataset into a DataFrame
data = pd.DataFrame(X, columns=['Feature1', 'Feature2', 'Feature3', 'Featur
data['Target'] = y

# Check the class distribution
print("Class Distribution:")
print(data['Target'].value_counts())

# Split the dataset into train and test sets
X_train, X_test, y_train, y_test = train_test_split(data.drop('Target', axi

# Calculate class weights
class_weights = dict(1 / data['Target'].value_counts())

# Create and train the model with class weighting
model = LogisticRegression(class_weight=class_weights)
model.fit(X_train, y_train)

# Make predictions on the test set
y_pred = model.predict(X_test)

# Evaluate the model
print("\nClassification Report:")
print(classification_report(y_test, y_pred))
```

```
Class Distribution:
0    898
1    102
Name: Target, dtype: int64

Classification Report:
              precision    recall  f1-score   support

           0       0.90      0.73      0.81       173
           1       0.22      0.48      0.30        27

    accuracy                           0.69       200
   macro avg       0.56      0.60      0.55       200
weighted avg       0.81      0.69      0.74       200
```

Ensemble Techniques

Ensemble methods combine multiple models to improve predictive performance. They can be effective for imbalanced data by giving more importance to the minority class. Examples include Balanced Random Forest and EasyEnsemble.

In [47]:
```
from sklearn.datasets import make_classification
from sklearn.model_selection import train_test_split
from sklearn.metrics import classification_report
from imblearn.ensemble import BalancedRandomForestClassifier, EasyEnsembleC

# Create a sample imbalanced dataset
np.random.seed(0)
X, y = make_classification(
    n_samples=1000,
    n_features=5,
    n_informative=3,
    n_redundant=2,
    n_classes=2,
    weights=[0.9, 0.1],  # Imbalanced class distribution
)

# Convert the dataset into a DataFrame
data = pd.DataFrame(X, columns=['Feature1', 'Feature2', 'Feature3', 'Featur
data['Target'] = y

# Check the class distribution
print("Class Distribution:")
print(data['Target'].value_counts())

# Split the dataset into train and test sets
X_train, X_test, y_train, y_test = train_test_split(data.drop('Target', axi

# Balanced Random Forest Classifier
brf = BalancedRandomForestClassifier(random_state=0)
brf.fit(X_train, y_train)
y_pred_brf = brf.predict(X_test)

# Easy Ensemble Classifier
eec = EasyEnsembleClassifier(random_state=0)
eec.fit(X_train, y_train)
y_pred_eec = eec.predict(X_test)

# Evaluate the models
print("\nClassification Report - Balanced Random Forest Classifier:")
print(classification_report(y_test, y_pred_brf))

print("\nClassification Report - Easy Ensemble Classifier:")
print(classification_report(y_test, y_pred_eec))
```

```
Class Distribution:
0    898
1    102
Name: Target, dtype: int64
```

```
C:\Users\RACHIT\anaconda3\lib\site-packages\imblearn\ensemble\_forest.py:
546: FutureWarning: The default of `sampling_strategy` will change from
``'auto'`` to ``'all'`` in version 0.13. This change will follow the implemen
tation proposed in the original paper. Set to ``'all'`` to silence this war
ning and adopt the future behaviour.
  warn(
C:\Users\RACHIT\anaconda3\lib\site-packages\imblearn\ensemble\_forest.py:
558: FutureWarning: The default of `replacement` will change from `False`
to `True` in version 0.13. This change will follow the implementation pro
posed in the original paper. Set to `True` to silence this warning and ad
opt the future behaviour.
  warn(


Classification Report - Balanced Random Forest Classifier:
              precision    recall  f1-score   support

           0       0.96      0.92      0.94       173
           1       0.62      0.78      0.69        27

    accuracy                           0.91       200
   macro avg       0.79      0.85      0.82       200
weighted avg       0.92      0.91      0.91       200


Classification Report - Easy Ensemble Classifier:
              precision    recall  f1-score   support

           0       0.97      0.84      0.90       173
           1       0.44      0.81      0.57        27

    accuracy                           0.83       200
   macro avg       0.70      0.83      0.73       200
weighted avg       0.90      0.83      0.85       200
```

Evaluation Metrics

Instead of using standard accuracy, consider using evaluation metrics that are more appropriate for imbalanced data, such as precision, recall, F1-score, or area under the ROC curve (AUC-ROC).

In [48]:
```
from sklearn.metrics import roc_auc_score
```

In [49]:
```
# Predict probabilities of class 1 (positive class)
y_pred_proba = brf.predict_proba(X_test)[:, 1]

# Calculate the ROC AUC score with out predict_proba() method
roc_auc = roc_auc_score(y_test, y_pred)
print("\nROC AUC Score:", roc_auc)

# Calculate the ROC AUC score with predict_proba() method
roc_auc = roc_auc_score(y_test, y_pred_proba)
print("\nROC AUC Score:", roc_auc)
```

```
ROC AUC Score: 0.6049025904517233

ROC AUC Score: 0.9173624491543566
```

Data Integration

Concatenation

Concatenation is the simplest technique that involves merging datasets vertically or horizontally based on a common key or index.

In [50]:
```
# Create sample datasets
dataset1 = pd.DataFrame({
    'ID': [1, 2, 3, 4],
    'Feature1': [0.5, 0.8, 0.2, 0.4],
    'Feature2': ['A', 'B', 'C', 'D']
})

dataset2 = pd.DataFrame({
    'ID': [5, 6, 7, 8],
    'Feature1': [0.9, 0.7, 0.3, 0.6],
    'Feature2': ['E', 'F', 'G', 'H']
})

# Concatenate the datasets along rows
concatenated_data = pd.concat([dataset1, dataset2], axis=0)

# Print the concatenated dataset
print("Concatenated Data:")
print(concatenated_data)
```

```
Concatenated Data:
   ID  Feature1 Feature2
0   1       0.5        A
1   2       0.8        B
2   3       0.2        C
3   4       0.4        D
0   5       0.9        E
1   6       0.7        F
2   7       0.3        G
3   8       0.6        H
```

Joins

Joins are used to combine datasets based on common columns or keys. Common types of joins include inner join, left join, right join, and full outer join.

In [51]:
```
# Create sample datasets
data1 = pd.DataFrame({
    'ID': [1, 2, 3, 4],
    'Feature1': ['A', 'B', 'C', 'D'],
    'Feature2': [10, 20, 30, 40]
})

data2 = pd.DataFrame({
    'ID': [3, 4, 5, 6],
    'Feature3': ['X', 'Y', 'Z', 'W'],
    'Feature4': [50, 60, 70, 80]
})

# Perform inner join
inner_join = pd.merge(data1, data2, on='ID', how='inner')
print("Inner Join:")
print(inner_join)

# Perform left join
left_join = pd.merge(data1, data2, on='ID', how='left')
print("\nLeft Join:")
print(left_join)

# Perform right join
right_join = pd.merge(data1, data2, on='ID', how='right')
print("\nRight Join:")
print(right_join)

# Perform outer join
outer_join = pd.merge(data1, data2, on='ID', how='outer')
print("\nOuter Join:")
print(outer_join)
```

```
Inner Join:
   ID Feature1  Feature2 Feature3  Feature4
0   3        C        30        X        50
1   4        D        40        Y        60

Left Join:
   ID Feature1  Feature2 Feature3  Feature4
0   1        A        10      NaN       NaN
1   2        B        20      NaN       NaN
2   3        C        30        X      50.0
3   4        D        40        Y      60.0

Right Join:
   ID Feature1  Feature2 Feature3  Feature4
0   3        C      30.0        X        50
1   4        D      40.0        Y        60
2   5      NaN       NaN        Z        70
3   6      NaN       NaN        W        80

Outer Join:
   ID Feature1  Feature2 Feature3  Feature4
0   1        A      10.0      NaN       NaN
1   2        B      20.0      NaN       NaN
2   3        C      30.0        X      50.0
3   4        D      40.0        Y      60.0
4   5      NaN       NaN        Z      70.0
5   6      NaN       NaN        W      80.0
```

Union

Union combines rows from multiple datasets into a single dataset, and it is commonly used when datasets have the same structure and columns.

In [52]:
```
# Create sample datasets
data1 = pd.DataFrame({'A': [1, 2, 3], 'B': ['a', 'b', 'c']})
data2 = pd.DataFrame({'A': [4, 5, 6], 'B': ['d', 'e', 'f']})

# Perform union operation
union_data = pd.concat([data1, data2], ignore_index=True)

# Print the union dataset
print("Union Dataset:")
print(union_data)
```

```
Union Dataset:
   A  B
0  1  a
1  2  b
2  3  c
3  4  d
4  5  e
5  6  f
```

Data Matching

Data matching involves identifying and linking similar or matching records across datasets based on common attributes. Techniques like fuzzy matching or record linkage can be used for data matching.

In [53]:
```
from fuzzywuzzy import fuzz, process

# Create sample dataset
data = pd.DataFrame({'Name': ['John Smith', 'Jane Doe', 'Mark Johnson', 'Da
                     'Phone': ['123-456-7890', '555-123-4567', '987-654-321

# Target string for fuzzy matching
target_string = 'John Smtih'

# Perform fuzzy matching using process.extractOne()
best_match = process.extractOne(target_string, data['Name'])
best_match_index = best_match[2]

# Get the best matching record
matched_record = data.loc[best_match_index]

# Print the best matching record
print("Best Matching Record:")
print(matched_record)
```

```
Best Matching Record:
Name       John Smith
Phone    123-456-7890
Name: 0, dtype: object
```

```
C:\Users\RACHIT\anaconda3\lib\site-packages\fuzzywuzzy\fuzz.py:11: UserWa
rning: Using slow pure-python SequenceMatcher. Install python-Levenshtein
to remove this warning
  warnings.warn('Using slow pure-python SequenceMatcher. Install python-L
evenshtein to remove this warning')
```

Entity Resolution

Entity Resolution, also known as Record Linkage or Deduplication, is the process of identifying and merging records from different datasets that refer to the same real-world entity. This technique is particularly useful when dealing with data from multiple sources that contain duplicate or overlapping information.

In [54]:
```
import recordlinkage

# Create sample datasets
data1 = pd.DataFrame({'ID': [1, 2, 3],
                      'Name': ['John Smith', 'Jane Doe', 'Mark Johnson'],
                      'Phone': ['123-456-7890', '555-123-4567', '987-654-32

data2 = pd.DataFrame({'ID': [4, 5, 6],
                      'Name': ['John Smith', 'Jane Doe', 'David Williams'],
                      'Phone': ['123-456-7890', '555-123-4567', '123-123-45

# Create indexer
indexer = recordlinkage.Index()
indexer.full()

# Generate pairs
pairs = indexer.index(data1, data2)

# Create comparison algorithm
compare = recordlinkage.Compare()

# Compare fields for similarity
compare.string('Name', 'Name', method='jarowinkler', threshold=0.85)
compare.string('Phone', 'Phone', method='jarowinkler', threshold=0.85)

# Compute similarity scores
scores = compare.compute(pairs, data1, data2)

# Select matches above threshold
matches = scores[scores.sum(axis=1) >= 1]

# Print matches
print("Matches:")
print(matches)
```

```
WARNING:recordlinkage:indexing - performance warning - A full index can r
esult in large number of record pairs.
Matches:
       0    1
0 0  1.0  1.0
  2  0.0  1.0
1 1  1.0  1.0
```

In [ ]: