Data Prep

Step 1: Loading and Inspecting the Dataset

```
import pandas as pd

# Load the dataset
df = pd.read_csv('/kaggle/input/ugransome-dataset/final(2).csv')

# Initial inspection
print(df.head())
print(df.info())
print(df.describe())
print("Number of missing values:\n", df.isnull().sum())
```

Step 2: Renaming Columns for Clarity

```
df.columns = ['Time', 'Protocol', 'Flag', 'Family', 'Clusters',
              'SeedAddress', 'ExpAddress', 'BTC', 'USD',
              'Netflow_Bytes', 'IPaddress', 'Threats', 'Port', 'Prediction']
```

Step 3: Handling Missing Values

- Fill missing numerical values using mean/median.
- Fill missing categorical values using mode.
- For sequential or time-series data, use interpolation.

```
df['Netflow_Bytes'].fillna(df['Netflow_Bytes'].median(), inplace=True)
df['Threats'].fillna(df['Threats'].mode()[0], inplace=True)
df['Time'] = pd.to_datetime(df['Time']).interpolate(method='linear')
```

Step 4: Removing Duplicates and Incorrect Values

```
# Remove duplicates
df.drop_duplicates(inplace=True)

# Correct misspelled entries
df['Threats'] = df['Threats'].str.replace('Bonet', 'Botnet')
```

Step 5: Handling Outliers

Remove outliers using the Interquartile Range (IQR) method.

```
def remove_outliers(df, column):
    Q1 = df[column].quantile(0.25)
    Q3 = df[column].quantile(0.75)
    IQR = Q3 - Q1
    lower_bound = Q1 - 1.5 * IQR
    upper_bound = Q3 + 1.5 * IQR
    return df[(df[column] >= lower_bound) & (df[column] <= upper_bound)]

df = remove_outliers(df, 'Netflow_Bytes')
df = remove_outliers(df, 'BTC')
df = remove_outliers(df, 'USD')
```

Step 6: Feature Engineering

6.1 Extracting Date and Time Components

```
df['Time'] = pd.to_datetime(df['Time'])
df['Hour'] = df['Time'].dt.hour
df['Day'] = df['Time'].dt.day
df['Month'] = df['Time'].dt.month
```

6.2 Creating New Features

```
df['High_BTC'] = df['BTC'].apply(lambda x: 1 if x > df['BTC'].mean() else 0)
```

Step 7: Data Transformation

7.1 Handling Skewed Data

```
import numpy as np
from scipy import stats

# Log transformation
df['Netflow_Bytes'] = np.log1p(df['Netflow_Bytes'])

# Yeo-Johnson Transformation (handles negative values)
df['BTC'], _ = stats.yeojohnson(df['BTC'])

# Square root transformation
df['USD'] = np.sqrt(df['USD'])
```

7.2 Scaling and Normalization

```
from sklearn.preprocessing import StandardScaler

scaler = StandardScaler()
df[['Netflow_Bytes', 'BTC', 'USD']] =
scaler.fit_transform(df[['Netflow_Bytes', 'BTC', 'USD']])
```

Step 8: Encoding Categorical Variables

8.1 Label Encoding for Ordinal Data

```
from sklearn.preprocessing import LabelEncoder

encoder = LabelEncoder()
df['Protocol'] = encoder.fit_transform(df['Protocol'])
df['Flag'] = encoder.fit_transform(df['Flag'])
df['Family'] = encoder.fit_transform(df['Family'])
```

8.2 One-Hot Encoding for Nominal Data

```
df = pd.get_dummies(df, columns=['Clusters', 'Port'])
```

To transform your **categorical variables** using **Label Encoding** and **One-Hot Encoding** with loops, we'll handle both tasks efficiently. We'll also ensure that the new columns are named clearly to avoid confusion.

Step 1: Label Encoding Using a Loop

Let's start by using a loop to apply `LabelEncoder` to ordinal categorical columns (`Protocol`, `Flag`, `Family`). We'll add a prefix to the new columns to indicate that they have been label-encoded.

Code for Label Encoding with Loop

```
from sklearn.preprocessing import LabelEncoder

# List of columns to label encode
label_columns = ['Protocol', 'Flag', 'Family']

# Apply LabelEncoder to each column using a loop
encoder = LabelEncoder()
for col in label_columns:
    df[f'encoded_{col}'] = encoder.fit_transform(df[col])

# Display the updated DataFrame
print(df.head())
```

Explanation:

- We use a loop to iterate through the columns that need to be label-encoded.
- Each new column is named with a prefix `encoded_` followed by the original column name (e.g., `encoded_Protocol`).
- This helps keep track of which columns have been encoded.

Step 2: One-Hot Encoding Using a Loop

For nominal categorical columns (`Clusters`, `Port`), we'll use `pd.get_dummies()` with a loop to create dummy variables and ensure that the columns are named properly.

Code for One-Hot Encoding with Loop

```python
# List of columns to one-hot encode
one_hot_columns = ['Clusters', 'Port']

# Apply One-Hot Encoding using a loop
for col in one_hot_columns:
    dummies = pd.get_dummies(df[col], prefix=col)
    df = pd.concat([df, dummies], axis=1)

# Drop the original columns after encoding
df.drop(columns=one_hot_columns, inplace=True)

# Display the updated DataFrame
print(df.head())
```

Explanation:

- We loop through the specified columns and generate one-hot encoded variables using `pd.get_dummies()`.
- The `prefix` parameter is used to add a prefix based on the original column name, resulting in columns like `Clusters_0`, `Port_22`, etc.
- The original columns (`Clusters`, `Port`) are then dropped from the DataFrame to avoid redundancy.

Alternative Approach: Combining Both Encodings in a Single Loop

If you want to combine both label encoding and one-hot encoding in one block of code, you can do it like this:

Combined Code

```python
from sklearn.preprocessing import LabelEncoder

# Define which columns need which encoding
label_columns = ['Protocol', 'Flag', 'Family']
one_hot_columns = ['Clusters', 'Port']

# Label Encoding
```

```python
encoder = LabelEncoder()
for col in label_columns:
    df[f'encoded_{col}'] = encoder.fit_transform(df[col])

# One-Hot Encoding
for col in one_hot_columns:
    dummies = pd.get_dummies(df[col], prefix=col)
    df = pd.concat([df, dummies], axis=1)

# Drop original columns
df.drop(columns=one_hot_columns + label_columns, inplace=True)

# Display the final DataFrame
print(df.head())
```

Label Encoding Explained

Label Encoding is a technique used to convert categorical data into numerical values. This method is especially useful for variables that have an inherent ordinal relationship (e.g., low, medium, high). However, it can also be applied to non-ordinal categorical data when the order does not matter.

Label Encoding assigns a unique integer to each category, effectively transforming a categorical column into numerical format.

When to Use Label Encoding?

- **Ordinal Data**: When there is a natural ordering between the categories.
    - Example: "Low", "Medium", "High" → 0, 1, 2.
- **Non-Ordinal Data**: Can also be used for non-ordinal data, but it may introduce unintended biases in some machine learning models (e.g., decision trees are fine, but linear models might assume an ordinal relationship).

Step 2: Apply Label Encoding

```python
from sklearn.preprocessing import LabelEncoder

# Initialize the label encoder
label_encoder = LabelEncoder()
```

```python
# Fit and transform the "Color" column
df['Color_Encoded'] = label_encoder.fit_transform(df['Color'])

print("\nLabel Encoded DataFrame:")
print(df)
```

Output:

```
  Color  Color_Encoded
0   Red              2
1 Green              1
2  Blue              0
3 Green              1
4   Red              2
5  Blue              0
6   Red              2
```

Explanation of the Output:

- The Label Encoder assigns numerical values to each unique category in the column:
  - `Red` → 2
  - `Green` → 1
  - `Blue` → 0

Step 3: Inverse Transform (Optional)

If you want to convert the encoded values back to their original categories:

```python
# Inverse transform the encoded labels back to the original values
df['Color_Decoded'] = label_encoder.inverse_transform(df['Color_Encoded'])

print("\nDecoded DataFrame:")
print(df)
```

Output:

```
   Color  Color_Encoded Color_Decoded
0    Red              2           Red
1  Green              1         Green
2   Blue              0          Blue
3  Green              1         Green
4    Red              2           Red
5   Blue              0          Blue
6    Red              2           Red
```

Notes:

1. **Pros**: Simple and efficient for models that can handle categorical variables.
2. **Cons**: Can introduce unintended ordinal relationships between categories, which might mislead some algorithms (e.g., linear regression).
3. **Alternative**: For non-ordinal data, consider using One-Hot Encoding instead.

9.1 Removing Low-Variance Features

```python
from sklearn.feature_selection import VarianceThreshold

selector = VarianceThreshold(threshold=0.01)
df = pd.DataFrame(selector.fit_transform(df),
columns=df.columns[selector.get_support()])
```

9.2 Correlation-Based Feature Selection

```python
import seaborn as sns
import matplotlib.pyplot as plt

corr_matrix = df.corr()
plt.figure(figsize=(12, 8))
sns.heatmap(corr_matrix, annot=True, cmap='coolwarm')
plt.title('Correlation Matrix')
plt.show()

# Remove highly correlated features
threshold = 0.9
```

```python
correlated_features = [col for col in corr_matrix if any(corr_matrix[col] >
threshold)]
df.drop(columns=correlated_features, inplace=True)
```

9.3 Feature Importance Using Random Forest

```python
from sklearn.ensemble import RandomForestClassifier

X = df.drop('Prediction', axis=1)
y = df['Prediction']
model = RandomForestClassifier(random_state=42)
model.fit(X, y)
importances = model.feature_importances_

# Plot feature importance
plt.figure(figsize=(10, 6))
plt.barh(X.columns, importances)
plt.title('Feature Importance')
plt.show()
```

Step 10: Dimensionality Reduction

10.1 Principal Component Analysis (PCA)

```python
from sklearn.decomposition import PCA

pca = PCA(n_components=10)
X_pca = pca.fit_transform(X)
```

Explanation:

- **PCA** reduces dimensionality while retaining variance, which speeds up training and reduces overfitting.

Step 11: Handling Class Imbalance

11.1 Using SMOTE for Oversampling

```python
from imblearn.over_sampling import SMOTE

smote = SMOTE(random_state=42)
X_resampled, y_resampled = smote.fit_resample(X, y)
```

Explanation:

- SMOTE generates synthetic samples for the minority class to balance the dataset.

Step 12: Splitting Data into Training and Testing Sets

```python
from sklearn.model_selection import train_test_split

X_train, X_test, y_train, y_test = train_test_split(X_resampled, y_resampled,
train_size=0.8, random_state=42)
```

Step 13: Final Checks and Validation

13.1 Checking Data Distributions

```python
df.hist(bins=20, figsize=(20, 15))
plt.show()
```

13.2 Verifying Class Distribution

```python
df['Prediction'].value_counts().plot(kind='bar', color='skyblue')
plt.title('Class Distribution')
plt.xlabel('Class')
plt.ylabel('Count')
plt.show()
```

Step 14: Saving the Cleaned Dataset

```
df.to_csv('cleaned_ugransome_dataset.csv', index=False)
```

Great! Let's extend the notes to include **encoding techniques**, **data transformations**, and the various strategies for **filling missing values**. This will make your exam prep comprehensive for data preprocessing.

Section 2: Encoding, Transformations, and Handling Missing Values

2.1 Encoding Categorical Variables

Machine learning models require numerical inputs. Therefore, converting categorical variables into numerical formats is essential.

Types of Encoding

1. Label Encoding

- Converts categorical labels into numeric values (0, 1, 2, ...).
- Useful for ordinal variables (where there is a natural order).

Code Example: Label Encoding

```
from sklearn.preprocessing import LabelEncoder

# Initialize the encoder
label_encoder = LabelEncoder()

# Apply to a categorical column
df['category_encoded'] = label_encoder.fit_transform(df['category_column'])

print(df[['category_column', 'category_encoded']].head())
```

2. One-Hot Encoding

- Converts categorical variables into a set of binary (0 or 1) columns.
- Suitable for nominal variables (no intrinsic order).

Code Example: One-Hot Encoding

```
# One-hot encode using pandas
df = pd.get_dummies(df, columns=['category_column'], drop_first=True)

print(df.head())
```

3. Ordinal Encoding

- Assigns numerical values to categories based on their order.
- Useful for ordinal data (e.g., low, medium, high).

Code Example: Ordinal Encoding

```
from sklearn.preprocessing import OrdinalEncoder

# Define the order
categories = [['low', 'medium', 'high']]
ordinal_encoder = OrdinalEncoder(categories=categories)

df['priority_encoded'] = ordinal_encoder.fit_transform(df[['priority']])

print(df[['priority', 'priority_encoded']].head())
```

4. Binary Encoding

- Converts categories into binary digits, then encodes them as binary numbers.
- Useful for high-cardinality categorical features.

Code Example: Binary Encoding

```
!pip install category_encoders
import category_encoders as ce

# Initialize the binary encoder
binary_encoder = ce.BinaryEncoder(cols=['category_column'])
df = binary_encoder.fit_transform(df)

print(df.head())
```

5. Frequency Encoding

- Replaces each category with its frequency in the dataset.

- Useful for high-cardinality features where preserving the frequency information is beneficial.

Code Example: Frequency Encoding

```
df['category_freq'] =
df['category_column'].map(df['category_column'].value_counts())
print(df[['category_column', 'category_freq']].head())
```

2.2 Data Transformations

Transformations are used to modify data distributions, handle skewness, and make variables more suitable for machine learning algorithms.

Types of Transformations

1. Log Transformation

- Reduces skewness for data that has a long right tail (positive skew).
- Useful for features like income, prices, or any positively skewed data.

Code Example: Log Transformation

```
df['log_transformed'] = np.log1p(df['skewed_feature'])
```

2. Square Root Transformation

- Used to reduce right skew, especially when the data has zeros.

Code Example: Square Root Transformation

```
df['sqrt_transformed'] = np.sqrt(df['feature'])
```

3. Box-Cox Transformation

- Transforms data to approximate normal distribution.
- Requires all values to be positive.

Code Example: Box-Cox Transformation

```
from scipy.stats import boxcox

df['boxcox_transformed'], _ = boxcox(df['positive_feature'] + 1)
```

4. Yeo-Johnson Transformation

- Similar to Box-Cox but can handle zero and negative values.

Code Example: Yeo-Johnson Transformation

```
from sklearn.preprocessing import PowerTransformer

power_transformer = PowerTransformer(method='yeo-johnson')
df['yeojohnson_transformed'] =
power_transformer.fit_transform(df[['feature']])
```

5. Exponential Transformation

- Used to handle left-skewed data.

Code Example: Exponential Transformation

```
df['exp_transformed'] = np.exp(df['feature'])
```

2.3 Handling Missing Values

Missing values are common in real-world data, and there are different strategies for handling them based on the type of data and the extent of missingness.

Techniques to Fill Missing Values

1. Dropping Missing Values

- Use this when the dataset is large and missing values are negligible.

Code Example: Dropping Missing Values

```
df.dropna(inplace=True)
```

2. Imputation Strategies

- **Mean Imputation**: For numerical columns with normally distributed data.

  ```
  df['column'] = df['column'].fillna(df['column'].mean())
  ```

- **Median Imputation**: For numerical columns with skewed data.

  ```
  df['column'] = df['column'].fillna(df['column'].median())
  ```

- **Mode Imputation**: For categorical columns.

  ```
  df['category'] = df['category'].fillna(df['category'].mode()[0])
  ```

3. Forward Fill/Backward Fill

- Useful for time-series data where trends are preserved.

Code Example: Forward/Backward Fill

```
df.fillna(method='ffill', inplace=True)  # Forward fill
df.fillna(method='bfill', inplace=True)  # Backward fill
```

4. K-Nearest Neighbors Imputation

- Imputes missing values based on the similarity of features.

Code Example: KNN Imputer

```
from sklearn.impute import KNNImputer

imputer = KNNImputer(n_neighbors=5)
df_imputed = pd.DataFrame(imputer.fit_transform(df), columns=df.columns)
```

5. Interpolation

- Useful for filling missing values in continuous data (e.g., time series).

Code Example: Interpolation

```
df['feature'] = df['feature'].interpolate(method='linear')
```

2.4 Putting It All Together

Let's consolidate all the steps into a streamlined workflow:

Comprehensive Data Preprocessing Pipeline

```python
import pandas as pd
import numpy as np
from sklearn.preprocessing import StandardScaler, MinMaxScaler,
PowerTransformer, LabelEncoder
from sklearn.impute import KNNImputer
import category_encoders as ce

# Step 1: Load the Data
df = pd.read_csv('data.csv')

# Step 2: Handle Duplicates
df.drop_duplicates(inplace=True)

# Step 3: Handle Missing Values
df.dropna(subset=['important_column'], inplace=True)  # Drop rows where
crucial data is missing
df['numeric_column'] =
df['numeric_column'].fillna(df['numeric_column'].mean())
df['category_column'] =
df['category_column'].fillna(df['category_column'].mode()[0])

# Step 4: Encoding Categorical Variables
label_encoder = LabelEncoder()
df['label_encoded'] = label_encoder.fit_transform(df['category_column'])
df = pd.get_dummies(df, columns=['one_hot_column'], drop_first=True)

# Step 5: Normalize/Standardize Features
scaler = StandardScaler()
df[['col1', 'col2']] = scaler.fit_transform(df[['col1', 'col2']])

# Step 6: Data Transformation
df['log_transformed'] = np.log1p(df['skewed_feature'])
power_transformer = PowerTransformer(method='yeo-johnson')
df['yeo_transformed'] = power_transformer.fit_transform(df[['feature']])

# Step 7: Final Check
print(df.head())
print(df.info())
```

To **fully prepare data for machine learning**, it's essential to go beyond just cleaning, encoding, and normalizing. The data preparation phase is one of the most crucial steps to ensure a model performs optimally. Below, I've outlined a **comprehensive list of techniques and steps** to help prepare your data for machine learning, with code examples and explanations.

Step 1: Data Loading and Initial Exploration

```
import pandas as pd

# Load your dataset
df = pd.read_csv("your_dataset.csv")

# Display first few rows
print(df.head())

# Summary statistics
print(df.describe())

# Check for missing values
print(df.isnull().sum())
```

Step 2: Data Cleaning

2.1 Handling Missing Values

- **Strategy 1: Remove rows with missing values**

  ```
  df.dropna(inplace=True)
  ```

- **Strategy 2: Fill missing values with mean/median/mode**

  ```
  df['Age'].fillna(df['Age'].mean(), inplace=True)  # For numerical columns
  df['City'].fillna(df['City'].mode()[0], inplace=True)  # For categorical columns
  ```

- **Strategy 3: Interpolation for time-series data**

  ```
  df['Temperature'] = df['Temperature'].interpolate(method='linear')
  ```

2.2 Removing Duplicates

```
df.drop_duplicates(inplace=True)
```

2.3 Correcting Data Types

```
# Convert data types if needed
df['Date'] = pd.to_datetime(df['Date'])
df['Price'] = df['Price'].astype(float)
```

2.4 Handling Outliers

```
import numpy as np

# Remove outliers using the IQR method
Q1 = df['Age'].quantile(0.25)
Q3 = df['Age'].quantile(0.75)
IQR = Q3 - Q1
df = df[~((df['Age'] < (Q1 - 1.5 * IQR)) | (df['Age'] > (Q3 + 1.5 * IQR)))]
```

Step 3: Feature Engineering

3.1 Encoding Categorical Variables

- **One-Hot Encoding**

  ```
  df = pd.get_dummies(df, columns=['Category'], drop_first=True)
  ```

- **Label Encoding**

  ```
  from sklearn.preprocessing import LabelEncoder
  le = LabelEncoder()
  ```

```
df['Category'] = le.fit_transform(df['Category'])
```

- **Ordinal Encoding (for ordinal data)**

```
df['Education_Level'] = df['Education_Level'].map({'High School': 0,
'Bachelor': 1, 'Master': 2, 'PhD': 3})
```

3.2 Feature Scaling

- **Normalization (Min-Max Scaling)**

```
from sklearn.preprocessing import MinMaxScaler
scaler = MinMaxScaler()
df[['Age', 'Income']] = scaler.fit_transform(df[['Age', 'Income']])
```

- **Standardization (Z-Score Scaling)**

```
from sklearn.preprocessing import StandardScaler
scaler = StandardScaler()
df[['Age', 'Income']] = scaler.fit_transform(df[['Age', 'Income']])
```

3.3 Data Transformation

- **Log Transformation (to handle skewed data)**

```
df['Income'] = np.log(df['Income'] + 1)
```

- **Box-Cox Transformation (for normalizing distributions)**

```
from scipy.stats import boxcox
df['Income'], _ = boxcox(df['Income'] + 1)
```

- **Square Root Transformation**

```
df['Income'] = np.sqrt(df['Income'])
```

3.4 Handling Missing Values in Categorical Data

- **Using `fillna` with a new category**

```
df['City'] = df['City'].fillna('Unknown')
```

- **Imputation with mode**

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

Step 4: Feature Selection

- **Correlation Matrix**

```
import seaborn as sns
import matplotlib.pyplot as plt

corr_matrix = df.corr()
sns.heatmap(corr_matrix, annot=True, cmap='coolwarm')
plt.show()
```

- **Removing Features with Low Variance**

```
from sklearn.feature_selection import VarianceThreshold
selector = VarianceThreshold(threshold=0.1)
df = selector.fit_transform(df)
```

- **Feature Importance using Random Forest**

```
from sklearn.ensemble import RandomForestClassifier
model = RandomForestClassifier()
model.fit(X, y)
importances = model.feature_importances_
```

Step 5: Train-Test Split

```python
from sklearn.model_selection import train_test_split

X = df.drop('target', axis=1)
y = df['target']
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
random_state=42)
```

Step 6: Handling Imbalanced Datasets

- **Using SMOTE (Synthetic Minority Over-sampling Technique)**

  ```python
  from imblearn.over_sampling import SMOTE
  smote = SMOTE()
  X_resampled, y_resampled = smote.fit_resample(X_train, y_train)
  ```

- **Using Class Weights in Algorithms**

  ```python
  from sklearn.ensemble import RandomForestClassifier
  model = RandomForestClassifier(class_weight='balanced')
  ```

Let's dive into **data cleaning and preprocessing**. These tasks are essential before you perform any analysis or build models. Below, we'll cover:

1. **Isolating Data Types**
2. **Checking for Null Values**
3. **Handling Missing Values**
4. **Standardizing Text** (e.g., converting to lowercase)
5. **Renaming Columns**

---

1. Isolating Data Types

1.1 Isolate Numerical Columns

```
# Select only numerical columns
numerical_df = df.select_dtypes(include=['number'])
print(numerical_df.head())
```

1.2 Isolate Categorical Columns

```
# Select only categorical columns
categorical_df = df.select_dtypes(include=['object'])
print(categorical_df.head())
```

1.3 Isolate Boolean Columns

```
# Select only boolean columns
boolean_df = df.select_dtypes(include=['bool'])
print(boolean_df.head())
```

1.4 Isolate Date/Time Columns

```
# Select only datetime columns
datetime_df = df.select_dtypes(include=['datetime'])
print(datetime_df.head())
```

1.5 Get All Data Types in the DataFrame

```
# Display data types of all columns
print(df.dtypes)
```

---

2. Checking for Null Values

2.1 Checking for Missing Values in the Entire DataFrame

```
# Check for missing values in each column
print(df.isnull().sum())
```

2.2 Checking for Missing Values Percentage

```
# Calculate the percentage of missing values for each column
missing_percentage = df.isnull().mean() * 100
print(missing_percentage)
```

2.3 Visualizing Missing Data with a Heatmap

```
import seaborn as sns
import matplotlib.pyplot as plt

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

---

3. Handling Missing Values

3.1 Dropping Rows with Missing Values

```
# Drop rows with any missing values
df.dropna(inplace=True)
```

3.2 Dropping Columns with Missing Values

```
# Drop columns where more than 50% of the values are missing
df.dropna(thresh=int(0.5 * len(df)), axis=1, inplace=True)
```

3.3 Filling Missing Values with a Specific Value

```
# Fill missing values in 'Score' column with 0
df['Score'].fillna(0, inplace=True)
```

3.4 Filling Missing Values with the Mean/Median

```
# Fill missing values with the mean
df['Score'] = df['Score'].fillna(df['Score'].mean())

# Fill missing values with the median
df['Score'] = df['Score'].fillna(df['Score'].median())
```

3.5 Forward Fill & Backward Fill

```
# Forward fill (use previous value to fill NaN)
df.fillna(method='ffill', inplace=True)

# Backward fill (use next value to fill NaN)
df.fillna(method='bfill', inplace=True)
```

---

4. Standardizing Text Data

4.1 Converting All Text to Lowercase

```
# Convert all text in the 'Name' column to lowercase
df['Name'] = df['Name'].str.lower()
```

4.2 Removing Leading/Trailing Whitespaces

```
# Remove whitespaces from the 'Name' column
df['Name'] = df['Name'].str.strip()
```

4.3 Replacing Characters in Text

```
# Replace underscores with spaces in the 'Course' column
df['Course'] = df['Course'].str.replace('_', ' ')
```

4.4 Removing Special Characters

```
# Remove special characters using regex
df['Name'] = df['Name'].str.replace('[^a-zA-Z0-9 ]', '', regex=True)
```

---

5. Renaming Columns

5.1 Renaming Columns Using a Dictionary

```
# Rename specific columns using a dictionary
df.rename(columns={'old_column_name': 'new_column_name', 'Name': 'Full Name'},
inplace=True)
```

5.2 Converting All Column Names to Lowercase

```
# Convert all column names to lowercase
df.columns = df.columns.str.lower()
```

5.3 Replacing Spaces in Column Names with Underscores

```
# Replace spaces with underscores in column names
df.columns = df.columns.str.replace(' ', '_')
```

---

6. Checking for Duplicate Rows

6.1 Identifying Duplicates

```
# Check for duplicates
print(df.duplicated().sum())
```

6.2 Removing Duplicates

```
# Remove duplicate rows
df.drop_duplicates(inplace=True)
```

---

7. Changing Data Types

7.1 Converting Columns to Numeric

```
# Convert 'Score' column to numeric type
df['Score'] = pd.to_numeric(df['Score'], errors='coerce')
```

7.2 Converting Columns to Date/Time

```
# Convert 'Date' column to datetime type
df['Date'] = pd.to_datetime(df['Date'], errors='coerce')
```

7.3 Converting Boolean Columns

```
# Convert 'flag' column to boolean
df['flag'] = df['flag'].astype(bool)
```

These notes cover a wide range of essential data cleaning and preprocessing tasks in Python using Pandas. Let me know if you need more details or specific examples!

Comprehensive Notes on Filtering Data Types and Selecting Data in Pandas

In data analysis, especially when working with **large datasets**, it is often crucial to **filter columns** based on their data types or other properties. This can help in **data cleaning, transformation**, and **feature engineering**. Below, we provide a comprehensive guide on how to filter columns by their data type and other useful techniques using the Pandas library in Python.

1. Importing the Required Libraries

Before starting, make sure you have imported Pandas:

```
import pandas as pd
import numpy as np
```

Section 1: Filtering by Data Type

1.1 Filtering Columns Based on Data Type

You can filter columns of a DataFrame by specifying the **data type** using the `select_dtypes()` method.

Example: Selecting Only Numerical Columns

```python
# Selecting only numerical columns
numerical_columns = df.select_dtypes(include=['number'])
print(numerical_columns)
```

- `include=['number']` selects all columns with numerical data types (`int` , `float`).

Example: Selecting Only Object (String) Columns

```python
# Selecting only object (string) columns
object_columns = df.select_dtypes(include=['object'])
print(object_columns)
```

- `include=['object']` selects all columns with string data types.

Example: Selecting Only Boolean Columns

```python
# Selecting only boolean columns
boolean_columns = df.select_dtypes(include=['bool'])
print(boolean_columns)
```

- `include=['bool']` selects all columns with boolean values (`True` or `False`).

1.2 Excluding Specific Data Types

If you want to **exclude** specific data types, you can do so using the `exclude` parameter.

Example: Excluding Object Columns

```python
# Excluding object (string) columns
non_object_columns = df.select_dtypes(exclude=['object'])
print(non_object_columns)
```

Example: Excluding Numerical Columns

```python
# Excluding numerical columns
non_numerical_columns = df.select_dtypes(exclude=['number'])
```

```
print(non_numerical_columns)
```

1.3 Selecting Multiple Data Types at Once

You can specify a list of data types to include or exclude:

```
# Selecting both numerical and boolean columns
selected_columns = df.select_dtypes(include=['number', 'bool'])
print(selected_columns)
```

Section 2: Filtering Columns Based on Other Criteria

2.1 Filtering Columns Based on Column Names

You can filter columns based on their names using list comprehensions.

Example: Selecting Columns Containing the Word 'Time'

```
# Selecting columns that contain the word 'Time'
time_columns = df[[col for col in df.columns if 'Time' in col]]
print(time_columns)
```

2.2 Filtering Columns Based on the Presence of Missing Values

You can filter columns with missing values using `isna()` or `notna()`.

Example: Selecting Columns with Missing Values

```
# Columns with missing values
missing_value_columns = df.columns[df.isna().any()]
print("Columns with missing values:", missing_value_columns)
```

Example: Selecting Columns Without Missing Values

```
# Columns without missing values
no_missing_value_columns = df.columns[df.notna().all()]
print("Columns without missing values:", no_missing_value_columns)
```

2.3 Filtering Rows Based on Specific Column Values

You can filter rows based on specific conditions in a column.

Example: Filtering Rows Where 'Protocol' Is 'TCP'

```
# Filtering rows where 'Protocol' column has value 'TCP'
tcp_rows = df[df['Protocol'] == 'TCP']
print(tcp_rows)
```

Example: Filtering Rows Where 'BTC' > 10

```
# Filtering rows where the value in 'BTC' column is greater than 10
high_btc = df[df['BTC'] > 10]
print(high_btc)
```

Section 3: Advanced Filtering Techniques

3.1 Filtering Using Regular Expressions

You can filter columns using regular expressions with the `filter()` method.

Example: Selecting Columns Starting with 'Netflow'

```
# Selecting columns that start with 'Netflow'
netflow_columns = df.filter(regex='^Netflow')
print(netflow_columns)
```

Example: Selecting Columns Ending with 'Bytes'

```
# Selecting columns that end with 'Bytes'
bytes_columns = df.filter(regex='Bytes$')
print(bytes_columns)
```

3.2 Filtering Based on DataFrame Index

You can filter rows based on index values.

Example: Selecting Rows by Index Range

```python
# Selecting rows between index 100 and 200
filtered_rows = df.loc[100:200]
print(filtered_rows)
```

Example: Selecting Rows Based on Index Values

```python
# Selecting rows with a specific index value
specific_row = df.loc[df.index == '2023-11-13']
print(specific_row)
```

3.3 Filtering Using Multiple Conditions

You can combine multiple conditions using logical operators.

Example: Filtering Rows with Multiple Conditions

```python
# Filtering rows where 'Protocol' is 'TCP' and 'BTC' > 10
filtered_data = df[(df['Protocol'] == 'TCP') & (df['BTC'] > 10)]
print(filtered_data)
```

Section 4: Practical Example with the UGRansome Dataset

Let's apply these techniques to the **UGRansome dataset** for practical understanding.

Step 1: Load the Dataset

```python
import pandas as pd
df = pd.read_csv('UGRansome_Dataset.csv')
df.columns = ["Time", "Protocol", "Flag", "Family", "Clusters", "SeedAddress",
"ExpAddress", "BTC", "USD", "Netflow_Bytes", "IPaddress", "Threats", "Port",
"Prediction"]
```

Step 2: Select Only Numerical Columns for Correlation Analysis

```python
numerical_df = df.select_dtypes(include=['number'])
print("Numerical columns:\n", numerical_df.head())
```

```python
# Calculate Spearman correlation
spearman_corr = numerical_df.corr(method='spearman')
print("Spearman Correlation:\n", spearman_corr)
```

Step 3: Filter Rows Based on Specific Conditions

```python
# Filter rows where 'Prediction' is 'A' (anomaly)
anomaly_rows = df[df['Prediction'] == 'A']
print("Anomalies:\n", anomaly_rows)
```
