Lecture Notes - Data Cleaning and Pre-processing Session 01

Introduction

In this lecture, we'll explore the process of data cleaning and preprocessing for a time series dataset on air quality. We'll use Python with libraries such as Pandas, Matplotlib, and Google Colab to handle, analyze, and visualize our data.

1. Data Loading and Initial Exploration

1.1 Connecting to Google Drive

We start by mounting Google Drive to access our dataset:

```python
from google.colab import drive
drive.mount('/content/drive')
```

This step is necessary when working in Google Colab to access files stored in Google Drive.

1.2 Loading the Dataset

We use Pandas to read our CSV file:

```python
import pandas as pd
dataframe = pd.read_csv(r"/content/drive/MyDrive/air_pollution_project/air_quality_hourly_data_Colombo_METDept.csv")
```

The `pd.read_csv()` function reads the CSV file and creates a DataFrame, which is a 2-dimensional labeled data structure in Pandas.

1.3 Initial Data Exploration

We use various Pandas methods to explore our data:

```python
dataframe.head()  # Display first few rows
print(f"Dataset Shape: {dataframe.shape}")  # Show number of rows and columns
print("\nMissing Values Count:")
print(dataframe.isnull().sum())  # Check for missing values
print("\nData Types:")
print(dataframe.dtypes)  # Display data types of each column
print("\nUnique Values per Column:")
print(dataframe.nunique())  # Show number of unique values in each column
```

These commands help us understand the structure of our data, identify missing values, and get an overview of the data types and unique values in each column.

2. Data Cleaning

2.1 Creating a Copy of the DataFrame

Before modifying our data, we create a copy to preserve the original:

```python
dataframe_cleaned = dataframe.copy()
```

This is a good practice to avoid unintended modifications to the original data.

2.2 Removing Unnecessary Columns

We remove the 'Unnamed: 0' column as it's not needed:

```python
dataframe.drop(['Unnamed: 0'], axis=1, inplace=True)
```

The `drop()` method removes specified columns or rows. Here, `axis=1` indicates we're dropping a column, and `inplace=True` means the change is applied directly to the DataFrame.

2.3 Handling Missing Values

We remove rows with missing values:

```python
dataframe_cleaned.dropna(inplace=True)
```

The `dropna()` method removes any row that contains at least one missing value.

2.4 Converting Data Types

We convert the 'CO2' column to numeric type and 'Time' column to datetime:

```python
dataframe_cleaned['CO2'] = pd.to_numeric(dataframe_cleaned['CO2'], errors='coerce')
dataframe_cleaned['Time'] = pd.to_datetime(dataframe_cleaned['Time'], format='mixed', errors='coerce')
```

`pd.to_numeric()` converts the 'CO2' column to numeric type, while `pd.to_datetime()` converts the 'Time' column to datetime type. The `errors='coerce'` parameter tells Pandas to set any values that can't be converted to NaN (Not a Number).

2.5 Removing Newly Created NaN Values

After conversion, we remove any new NaN values:

```python
dataframe_cleaned.dropna(inplace=True)
```

3. Data Visualization

3.1 Initial Visualization

We create a simple line plot of CO2 concentration over time:

```python
import matplotlib.pyplot as plt

fig, axs = plt.subplots(1, 1, figsize=(10, 6))
axs.plot(dataframe_cleaned['Time'], dataframe_cleaned['CO2'], color='blue', marker='o')
axs.set_title('CO2 Concentration')
axs.set_ylabel('CO2 (ppm)')
axs.grid(True)

plt.tight_layout()
plt.show()
```

This creates a line plot showing CO2 concentration over time.

3.2 Identifying Continuous Data

We need to identify the continuous data clumps that can be used to train a machine learning model.

```python
dataframe_cleaned['TimeDelta'] = dataframe_cleaned['Time'].diff()
continuous_rows = dataframe_cleaned[dataframe_cleaned['TimeDelta'] == pd.Timedelta(hours=1)]
continuous_rows['index_diff'] = continuous_rows.index.diff()
continuous_rows['group'] = (continuous_rows['index_diff'] != 1).cumsum()
```

This code identifies continuous sections of hourly data and groups them.

3.3 Selecting the Largest Continuous Section

We select the largest continuous section of data (Remember we want to get the largest amount of data for training - more data the better !!)

```python
largest_group = continuous_rows['group'].value_counts().idxmax()
largest_continuous_data = continuous_rows[continuous_rows['group'] == largest_group]
largest_continuous_data = largest_continuous_data.drop(['index_diff', 'group'], axis=1)
```

Let's break down this code section line by line:

1. `dataframe_cleaned['TimeDelta'] = dataframe_cleaned['Time'].diff()`

   This line creates a new column called 'TimeDelta' in the dataframe_cleaned DataFrame. The `diff()` function calculates the difference between consecutive values in the 'Time' column. Since 'Time' is a datetime column, this difference represents the time interval between consecutive rows.

   Logic: This step helps identify gaps or irregularities in the time series data.

2. `continuous_rows = dataframe_cleaned[dataframe_cleaned['TimeDelta'] == pd.Timedelta(hours=1)]`

   This line creates a new DataFrame called continuous_rows. It selects only the rows from dataframe_cleaned where the 'TimeDelta' is exactly one hour (pd.Timedelta(hours=1)).

   Logic: This filters the data to include only consecutive hourly measurements, removing any rows where there's a gap larger than one hour.

3. `continuous_rows['index_diff'] = continuous_rows.index.diff()`

   This creates a new column 'index_diff' in the continuous_rows DataFrame. It calculates the difference between the index values of consecutive rows.

Logic: In a perfectly continuous dataset, this difference should always be 1. Any value greater than 1 indicates a gap in the data, even if the time difference is still one hour (which can happen if there are missing rows in the original data).

4. `continuous_rows['group'] = (continuous_rows['index_diff'] != 1).cumsum()`

   This line is quite complex, so let's break it down further:

   a. `continuous_rows['index_diff'] != 1`: This creates a boolean series. It's True where the index difference is not 1 (indicating a gap in the data), and False otherwise.

   b. `(...).cumsum()`: This applies a cumulative sum to the boolean series.

   Logic: This clever technique assigns a group number to each continuous chunk of data. Here's how it works:

   - The cumulative sum increases by 1 every time it encounters a True value (i.e., a gap in the data).
   - All rows between gaps get the same group number.
   - When a gap is encountered, the group number increases, starting a new group.

   For example, if we have data like this:

```
index_diff: [1, 1, 1, 2, 1, 1, 3, 1, 1]
!= 1:       [F, F, F, T, F, F, T, F, F]
cumsum:     [0, 0, 0, 1, 1, 1, 2, 2, 2]
```

This would create three groups: one before the first gap, one between the gaps, and one after the second gap.

The overall logic of this code section is to identify and group continuous chunks of hourly data. This is particularly useful for time series analysis, where you often want to work with uninterrupted sequences of data. By grouping the data this way, you can later select the largest continuous chunk for analysis, ensuring that you're working with a consistent, uninterrupted dataset.

3.4 Final Visualization

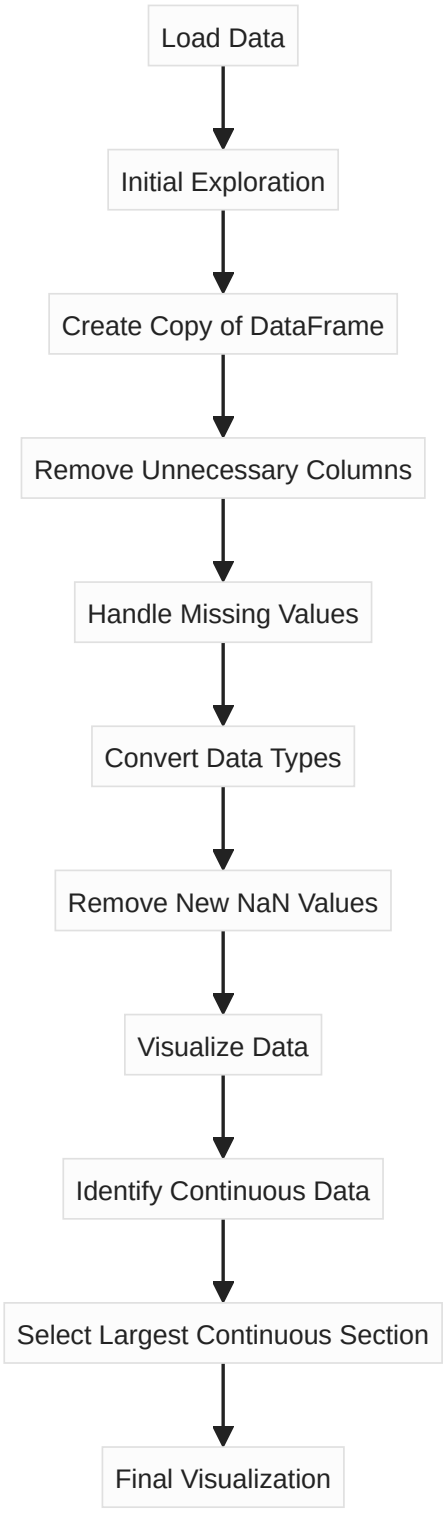
We create a new plot with the largest continuous section of data:

```python
fig, axs = plt.subplots(1, 1, figsize=(10, 6))
axs.plot(largest_continuous_data['Time'], largest_continuous_data['CO2'], color='green', marker='o')
axs.set_title('CO2 Concentration')
axs.set_ylabel('CO2 (ppm)')
axs.grid(True)

plt.tight_layout()
plt.show()
```

This final plot gives a clearer view of the CO2 concentration trends over a continuous time period.

Summary of Lecture 01


Load Data
Initial Exploration
Create Copy of DataFrame
Remove Unnecessary Columns
Handle Missing Values
Convert Data Types
Remove New NaN Values
Visualize Data
Identify Continuous Data
Select Largest Continuous Section
Final Visualization