AI - Unit 5

Data Processing

Instructor: Saptarshi Jana

Learning Objectives

After studying this chapter, students will be able to:[a]

- Comprehend the fundamentals of data cleaning and its critical importance
- Master various data cleaning techniques using Python Pandas library
- Understand the principles and methods of data transformation

[a] Unit 5: Theoretical and Practical Aspects of Data Processing

January 10, 2026

1 Introduction to Data Cleaning

1.1 Understanding Data Quality

Data cleaning represents the fundamental process of identifying, correcting, or eliminating errors and inconsistencies within a dataset to ensure that information is properly formatted and suitable for analytical purposes. When data fails to undergo this essential cleaning process, it is commonly referred to as *dirty data*.

In real-world scenarios, data is rarely pristine. Organizations collect information from various sources, each potentially introducing different types of errors, formatting inconsistencies, or missing values. The quality of insights derived from data analysis directly depends on the quality of the underlying data itself.

Why Data Cleaning Matters?/Benefits of Clean Data.

Impact on Analysis:

- Poor data quality leads to incorrect conclusions and flawed decision-making
- Clean data enables accurate pattern recognition and predictive modeling
- Data quality directly affects the performance of machine learning algorithms
- Well-cleaned data reduces computational overhead and processing time

1.2 Common Data Quality Issues

Data quality issues appear in many forms and must be identified before effective data cleaning can be performed. The most common problems are outlined below.

1.2.1 Missing Values

Missing values occur when required data is absent from a dataset. This is common in real-world data such as customer records with missing contact details or sensor data with gaps.

Common causes include:

- Data not captured during entry
- System or equipment failures
- Human error or privacy restrictions

Large amounts of missing data can introduce bias and reduce the reliability of analysis.

1.2.2 Outliers

Outliers are data points that differ significantly from the rest of the dataset. They may arise from genuine extreme events or from errors such as incorrect data entry or faulty measurements.

Outliers can:

- Represent valid but rare occurrences
- Distort statistical measures and analysis

Example: A salary of $1,200,000 in a dataset ranging from $30,000 to $120,000 may be valid (CEO salary) or erroneous and should be investigated.

1.2.3 Duplicate Records

Duplicates occur when identical or nearly identical records appear more than once, causing over-representation in analysis.

Duplicates often result from:

- Multiple data sources
- Import or system errors
- Absence of unique identifiers

Records with conflicting values require careful review before removal.

1.2.4 Erroneous Data

Erroneous data consists of incorrect values such as misspellings, outdated information, or invalid identifiers. These errors may appear realistic, making detection difficult.

Common sources include:

- Typographical errors
- Misunderstood field requirements
- Software or data corruption issues

Validation against reliable sources is essential for correction.

1.2.5 Inconsistencies

Inconsistent data refers to mismatches in format, units, labels, or data types across records or sources. These discrepancies can manifest in various ways:

Format Inconsistencies: Different representations of the same type of data. For example:

- Dates: "12/25/2023" vs. "December 25, 2023" vs. "2023-12-25"
- Phone numbers: "(555) 123-4567" vs. "555-123-4567" vs. "5551234567"
- Names: "John Smith" vs. "Smith, John" vs. "JOHN SMITH"

Unit Inconsistencies: Measurements reported in different units:

- Distance: kilometers vs. miles vs. meters
- Temperature: Celsius vs. Fahrenheit vs. Kelvin
- Time: 12-hour vs. 24-hour format

Categorical Inconsistencies: Variations in category labels:

- "NY" vs. "New York" vs. "N.Y."
- "Male" vs. "M" vs. "man"

Type Inconsistencies: Different data types for the same field:

- Age stored as integers vs. strings
- Prices stored as floats vs. integers

When working with multiple data sources, inconsistencies multiply. If data appears incorrect, duplicated, or mislabeled, outcomes and algorithms become unreliable. Data cleaning in data science using Python helps eliminate these issues, ensuring data reliability and analytical validity.

2 The Data Cleaning Cycle

2.1 Essential Steps in Data Cleaning

The data cleaning process follows a systematic cycle comprising six fundamental steps:

1. **Loading and Inspecting Data:** Import the dataset and perform initial examination to understand its structure, size, and basic characteristics.
2. **Identifying Column Issues:** Analyze each column to detect problems such as incorrect data types, unusual values, or formatting problems.
3. **Handling Duplicates:** Identify and appropriately manage duplicate records to prevent overrepresentation.
4. **Managing Missing Values:** Address gaps in the data through imputation, deletion, or other appropriate strategies.
5. **Dealing with Outliers:** Identify extreme values and determine whether to keep, transform, or remove them.
6. **Converting Data Types:** Ensure all columns have appropriate data types for their intended use in analysis.

Data Cleaning Best Practices

- Always maintain a copy of the original raw data
- Document all cleaning steps and decisions made
- Use consistent naming conventions
- Validate results after each cleaning operation
- Automate repetitive cleaning tasks when possible
- Review cleaned data with domain experts

3 Data Cleaning with Python Pandas

3.1 Introduction to Pandas

Python has emerged as an indispensable skill for Data Scientists. Among its many libraries, **Pandas** stands out as a powerful and efficient tool for data processing, cleaning, manipulation, and analysis. Pandas is built upon Python's Data Analysis Library and consists of classes focused on reading, processing, and writing CSV files.

Numerous data cleaning tools exist in the market, but Pandas provides a fast and efficient approach to managing and exploring data. It accomplishes this through Series and DataFrames, which facilitate efficient data representation and manipulation in various ways.

3.2 Handling Missing Values

Missing value management is a critical aspect of data cleaning. Pandas provides comprehensive methods for both identifying and addressing missing data.

3.2.1 Identifying Missing Values

Pandas offers several functions to detect missing values:

Listing 1: Detecting Missing Values

```
import pandas as pd
import numpy as np

# Load dataset
df = pd.read_csv('data.csv')

# Check for missing values
print(df.isnull())  # Returns boolean DataFrame
print(df.isna())    # Alias for isnull()
print(df.isnull().sum())  # Count missing values per column
```

3.2.2 Handling Missing Values

Pandas provides multiple strategies for dealing with missing data:

1. **Removing Rows/Columns with Missing Values:**

Listing 2: Removing Missing Data

```
# Remove rows with any missing values
df_clean = df.dropna()

# Remove columns with any missing values
df_clean = df.dropna(axis=1)

# Keep rows with at least n non-NA values
df_clean = df.dropna(thresh=n)
```

2. **Filling Missing Values:**

Listing 3: Filling Missing Values

```
# Fill with a specific value
df['column'].fillna(value=0, inplace=True)

# Forward-fill (use previous value)
df['column'].fillna(method='ffill', inplace=True)

# Backward-fill (use next value)
df['column'].fillna(method='bfill', inplace=True)

# Fill with mean
df['column'].fillna(df['column'].mean(), inplace=True)

# Fill with median
df['column'].fillna(df['column'].median(), inplace=True)

# Fill with mode
df['column'].fillna(df['column'].mode()[0], inplace=True)
```

3.3 Managing Duplicate Records

Duplicate records can distort analysis results and must be identified and handled appropriately.

3.3.1 Identifying Duplicates

Listing 4: Detecting Duplicates

```
# Check for duplicate rows
print(df.duplicated())  # Returns boolean Series

# Count duplicate rows
print(df.duplicated().sum())
```

3.3.2 Removing Duplicates

Listing 5: Removing Duplicates

```
# Remove all duplicate rows
df_clean = df.drop_duplicates()

# Remove duplicates based on specific columns
df_clean = df.drop_duplicates(subset=['col1', 'col2'])

# Keep first occurrence (default)
df_clean = df.drop_duplicates(keep='first')

# Keep last occurrence
df_clean = df.drop_duplicates(keep='last')

```

```
# Remove all occurrences of duplicates
df_clean = df.drop_duplicates(keep=False)
```

3.4 Handling Inconsistent Data

Data inconsistencies require careful attention and systematic correction.

3.4.1 Text Data Cleaning

Listing 6: Cleaning Text Data

```
# Convert to lowercase
df['column'] = df['column'].str.lower()

# Convert to uppercase
df['column'] = df['column'].str.upper()

# Remove leading/trailing whitespace
df['column'] = df['column'].str.strip()

# Replace substrings
df['column'] = df['column'].str.replace('old', 'new')
```

3.4.2 Date Data Formatting

Listing 7: Standardizing Dates

```
# Convert to datetime objects
df['date_column'] = pd.to_datetime(df['date_column'])

# Format datetime strings
df['date_column'] = df['date_column'].dt.strftime('%Y-%m-%d')
```

3.4.3 Numeric Data Cleaning

Listing 8: Cleaning Numeric Data

```
# Convert to integer
df['column'] = df['column'].astype(int)

# Convert to float
df['column'] = df['column'].astype(float)
```

3.5 Handling Categorical Inconsistencies

Listing 9: Standardizing Categories

```
# Replace inconsistent category labels
category_mapping = {'old_value': 'new_value'}
```

```
df['column'].replace(category_mapping, inplace=True)

# Identify unique values
print(df['column'].unique())
```

3.6 Detecting and Handling Outliers

3.6.1 Identifying Outliers

Outliers can be detected through various visualization and statistical methods:

Visualization Methods:

- Box plots (box-and-whisker plots)
- Scatter plots
- Histograms

Statistical Methods:

- **Z-score method:** This identifies outliers by measuring how far a data value lies from the mean in terms of standard deviation. It shows how many standard deviations a value is above or below the mean. A data point is typically considered an outlier if $|z| > 3$.
- **Interquartile Range (IQR) method:** Here a statistical technique used to identify outliers by measuring the spread of the middle 50% of a dataset. It is based on quartiles and is not affected by extreme values. Data points that lie far below or above the typical range of values are considered outliers using this method.

Listing 10: Detecting Outliers

```
# Using Z-score
from scipy import stats
import numpy as np

z_scores = np.abs(stats.zscore(df['column']))
outliers = df[z_scores > 3]

# Using IQR
Q1 = df['column'].quantile(0.25) # Q1 (25th percentile)
Q3 = df['column'].quantile(0.75) # Q3 (75th percentile)
IQR = Q3 - Q1

lower_bound = Q1 - 1.5 * IQR
upper_bound = Q3 + 1.5 * IQR

outliers = df[(df['column'] < lower_bound) | (df['column'] >
   upper_bound)]
```

3.6.2 Handling Outliers

Listing 11: Managing Outliers

```
# Remove outliers
df_clean = df[(df['column'] >= lower_bound) & (df['column'] <=
    upper_bound)]

# Cap outliers (winsorization)
df['column'] = df['column'].clip(lower=lower_bound, upper=
    upper_bound)

# Transform data (log transformation)
df['column'] = np.log(df['column'])
```

4 Available Datasets for Practice (Exploring Kaggle Dataset)

The availability of diverse, well-curated datasets is essential for learning and advancing artificial intelligence. Practice datasets allow learners to apply machine learning concepts, experiment with algorithms, and understand real-world data challenges. While several libraries and frameworks provide built-in datasets for guided learning, platforms like **Kaggle** play a central role in practical, hands-on experience.

Libraries such as **Scikit-learn**, **Keras/TensorFlow**, and **PyTorch** offer small, structured datasets (e.g., Iris, MNIST, CIFAR-10) that are especially useful for beginners to understand core algorithms in a controlled environment. These datasets are ideal for learning basic classification, regression, and image recognition tasks.

Hugging Face Datasets further extend practice opportunities by providing large-scale and diverse datasets, particularly for natural language processing, enabling experimentation with modern transformer-based models.

However, **Kaggle** stands out as the most comprehensive platform for practical learning. It hosts thousands of real-world datasets across domains such as finance, healthcare, social sciences, and computer vision. Kaggle also offers interactive notebooks, public code examples, and competitions, allowing learners to explore data quality issues, feature engineering, model evaluation, and performance optimization in realistic scenarios.

In summary, while built-in datasets help build foundational understanding, Kaggle provides a complete ecosystem for applied learning, making it an indispensable resource for developing practical data science and machine learning skills.

5 Practical Example: Iris Dataset

The Iris Dataset is one of the most famous and widely used datasets in machine learning and statistics, especially for classification problems.

What is the Iris Dataset?

The Iris Dataset contains measurements of iris flowers belonging to three different species. It is commonly used to demonstrate and test classification algorithms because of its simplicity and clarity.

5.1 Loading and Exploring Data

Step 1: Download the 'iris.csv' file from Kaggle platform. Now, let's work through a complete example using the famous Iris dataset:

Listing 12: Loading the Iris Dataset

```
import pandas as pd
import numpy as np

# Define column names
column_names = ['sepal_length', 'sepal_width',
                'petal_length', 'petal_width', 'species']

# Load dataset
iris_data = pd.read_csv('iris.csv', names=column_names, header=0)

# Display first few rows
print(iris_data.head())
```

Output example:

	sepal_length	sepal_width	petal_length	petal_width	species
0	5.1	3.5	1.4	0.2	Iris-setosa
1	4.9	3.0	1.4	0.2	Iris-setosa
2	4.7	3.2	1.3	0.2	Iris-setosa
3	4.6	3.1	1.5	0.2	Iris-setosa
4	5.0	3.6	1.4	0.2	Iris-setosa

The `header=0` parameter indicates that the first row of the CSV file contains column names (header).

5.2 Exploring Dataset Structure

Listing 13: Dataset Information

```
# Get dataset info
print(iris_data.info())

# Get statistical summary
print(iris_data.describe())
```

Sample Output:

```
RangeIndex: 150 entries, 0 to 149
Data columns (total 5 columns):
 #   Column        Non-Null Count  Dtype
---  ------        --------------  -----
 0   sepal_length  150 non-null    float64
 1   sepal_width   150 non-null    float64
 2   petal_length  150 non-null    float64
 3   petal_width   150 non-null    float64
 4   species       150 non-null    object
```

```
dtypes: float64(4), object(1)
memory usage: 6.0+ KB
```

The `info()` function provides valuable insights into the overall structure, including the number of non-null values in each column and memory usage.

5.3 Checking Class Distribution

Understanding class distribution is crucial for classification tasks:

Listing 14: Class Distribution Analysis

```
# Check species distribution
print(iris_data['species'].value_counts())
```

Output:

```
Iris-setosa        50
Iris-versicolor    50
Iris-virginica     50
Name: species, dtype: int64
```

The results show balanced representation with equal numbers across all three species classes, which establishes a fair foundation for evaluation and comparison across the classes.

5.4 Handling Missing Values

Although the Iris dataset typically doesn't have missing values, let's demonstrate the process:

Listing 15: Managing Missing Values

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

# If missing values exist, handle them
# Fill with mean (for numeric columns)
iris_data['sepal_length'].fillna(
    iris_data['sepal_length'].mean(),
    inplace=True
)
```

5.5 Removing Duplicates

Listing 16: Duplicate Management

```
# Check for duplicates
duplicate_rows = iris_data.duplicated()
print(f"Number␣of␣duplicate␣rows:␣{duplicate_rows.sum()}")

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

```
print(f"Dataset␣shape␣after␣removing␣duplicates:␣{iris_data.shape}
    ")
```

We verify that no duplicates exist in this dataset. However, if duplicates were present, the `drop_duplicates()` function would remove them effectively.

5.6 Normalizing Data

Normalization scales numerical features to have a mean of 0 and standard deviation of 1, ensuring consistent scaling across features:

Listing 17: Data Normalization

```
from sklearn.preprocessing import StandardScaler

# Initialize scaler
scaler = StandardScaler()

# Select columns to normalize
cols_to_normalize = ['sepal_length', 'sepal_width',
                     'petal_length', 'petal_width']

# Fit and transform
iris_data[cols_to_normalize] = scaler.fit_transform(
    iris_data[cols_to_normalize]
)

print(iris_data.head())
```

5.7 Short Answer Questions

1. Why is it important to check for duplicate records in a dataset? What problems can duplicates cause?
2. Describe three methods for handling missing values and explain when each method would be appropriate.
3. List and explain the six fundamental steps in the data cleaning cycle.
4. What is dirty data? Provide three examples of how data becomes "dirty."
5. Explain the IQR method for detecting outliers.
6. Why is data cleaning considered the most important task in data science?
7. What is the purpose of the `describe()` function in Pandas?
8. How can you identify inconsistencies in categorical data?

5.8 Practical Exercises

1. Load a CSV file containing student grades, check for missing values, and fill them with the mean of each column.

2. Create a Python script that identifies and removes duplicate rows from a DataFrame based on specific columns.

3. Describe 'the Z-score' and 'IQR' methods for detecting outliers. Write code to detect outliers in a numeric column using both methods, then compare the results.

4. Clean a dataset with inconsistent date formats by converting all dates to a standard format (YYYY-MM-DD).

5. Create a function that takes a DataFrame and performs a complete cleaning process: handling missing values, removing duplicates, and detecting outliers.

6. Load the Iris dataset, perform exploratory data analysis, check class distribution, and normalize the features.

7. Write code to handle categorical inconsistencies by mapping similar values to a standard format (e.g., "NY", "N.Y.", "New York" $\rightarrow$ "New York").