PROGRAMMING FOR AI

Lecture 6

Data Handling and Preprocessing

Instructor: Zafar Iqbal

Agenda

- Handling Missing Values
- Scaling and Normalization
- Parsing Dates
- Character Encodings
- Inconsistent Data Entry

1. Handling Missing Values

- **Data Cleaning**
  - *A crucial part of data science but often frustrating.*
  - **Common issues**: *garbled text, missing values, incorrect date formatting, inconsistent data entry.*
- **Goal**
  - *Understand common data cleaning problems and how to fix them.*
  - *Learn to clean data efficiently for faster analysis.*
- Missing values (NaN, None) occur when data is absent in a dataset.
- Can lead to errors in analysis or biased results if not handled properly

Handling Missing Values

- **Common Causes**:
  - Data not recorded (e.g., sensor failure).
  - Data doesn't exist (e.g., no children for a "height of oldest child" field).
- **Load and Inspect Data**:
  - Use **pandas** to read data (e.g., **pd.read_csv()**).
  - Check the first few rows with **head()** to spot missing values.
- **Quantify Missingness**:
  - Count missing values per column: df.isnull().sum()
- **Dropping Missing Values**
  - Drop rows with any missing values: df.dropna().
  - Drop columns with any missing values: df.dropna(axis=1).

Handling Missing Values

- **Filling Missing Values (Imputation)**
  - When retaining data is critical.
  - Replace with a constant (e.g., 0)
  - `df.fillna(0)`

  - **Forward/Backward Fill:** Propagate adjacent values
  - `df.fillna(method='bfill').fillna(0)`   # Backward fill

Before

	A	B
0	1.0	NaN
1	NaN	2.0
2	3.0	NaN
3	NaN	NaN

After

	A	B
0	1.0	2.0
1	3.0	2.0
2	3.0	0.0
3	0.0	0.0

2. Scaling and Normalization

- Scaling
- What it does: Changes the range of the data (e.g., min-max scaling).
- **Example Methods:**
  - Min-Max Scaling (0 to 1 range)
  - Standardization (mean=0, std=1)
- **When to Use:**
  - When features have different units (e.g., kg vs. cm).
  - Needed for distance-based algorithms (e.g., KNN, SVM).

Scaling and Normalization

- **Normalization**
- What it does: Changes the shape of the distribution (e.g., Gaussian).
- Example Methods:
  - Log transformation
  - Box-Cox transformation
- When to Use:
  - When data is skewed (non-normal).
  - Needed for models assuming normality (e.g., linear regression).

Scaling and Normalization

Aspect	Scaling	Normalization
Goal	Adjusts range	Adjusts distribution shape
Example	`MinMaxScaler()`	`Log transformation`
Use Case	Standardizing features	Making data more Gaussian

Simple Rule of Thumb

- **Use Scaling** → For adjusting feature ranges (e.g., preprocessing for ML).
- **Use Normalization** → For fixing skewed data (e.g., statistical modeling).

3. Parsing Dates

- **What is Date Parsing?**
- Date parsing is the process of converting date strings
- (e.g., "3/2/07", "2023-12-25") into a machine-readable datetime format (like Python's datetime64).
- This allows:
  - Proper sorting (2007-03-02 comes before 2007-04-06).
  - Easy extraction of components (day, month, year).
  - Time-based calculations (e.g., time intervals).

Why Parse Dates?

- Correct Data Type
  - Raw dates are often stored as strings (object dtype in pandas).
  - Parsing converts them to datetime64, enabling date operations.
- Avoid Errors
  - Without parsing, functions like .dt.day or .sort_values() won’t work.
- Consistency
  - Fixes varying formats (e.g., MM/DD/YYYY vs. DD-MM-YYYY).

How to Parse Dates in Pandas

1. Basic Parsing

- Use **pd.to_datetime()**

```
df['date_parsed'] = pd.to_datetime(df['date_column'])
```

- Problem: Pandas might misinterpret formats (e.g., "01/04/2023" → January 4th or April 1st?).

2. Specify Format

- Use format parameter with strftime directives

```
df['date_parsed'] = pd.to_datetime(df['date_column'], format="%m/%d/%y")
```

- Example Formats:
  - "3/2/07" → format="%m/%d/%y"
  - "17-1-2007" → format="%d-%m-%Y"

How to Parse Dates in Pandas

3. Handle Multiple Formats

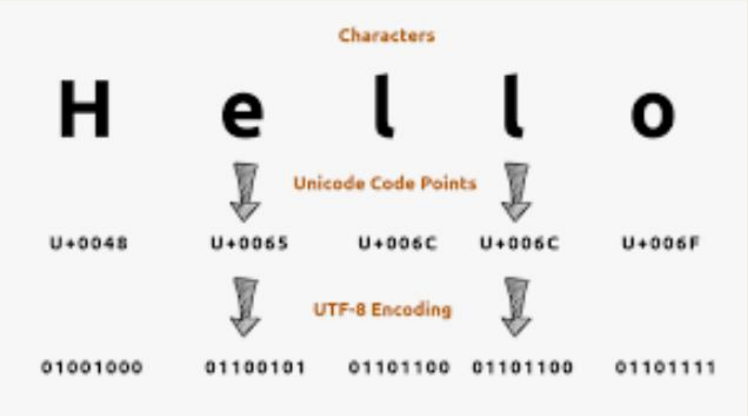
- If dates are mixed (e.g., "3/2/07" and "2007-01-17")
- `df['date_parsed'] = pd.to_datetime(df['date_column'], infer_datetime_format=True)`

Common Use Cases After Parsing

- **Extract Date Components**
- `df['day'] = df['date_parsed'].dt.day   # Day of month (1-31)`
- `df['month'] = df['date_parsed'].dt.month # Month (1-12)`
- `df['year'] = df['date_parsed'].dt.year  # Year (e.g., 2007)`
- **Filter by Date**
- `df_2007 = df[df['date_parsed'].dt.year == 2007]`
- **Plotting Trends Over Time**
- `sns.lineplot(x='date_parsed', y='fatalities', data=df)`

4. Character Encodings

- Character encodings map binary byte sequences (e.g., 01101000 01101001) to human-readable text (e.g., "hi").
- Mismatched encoding can lead to scrambled text (mojibake) or unknown characters □□□□□□.
  - Example: æ–‡å—åŒ–ã??
- UTF-8 is the standard encoding in Python and is recommended for data handling.
- Python 3 simplifies encoding handling compared to Python 2



Encoding and Decoding Strings in Python

- Strings in Python are UTF-8 by default.
- **Bytes data type** represents encoded strings.
- Example:

```
before = "This is the euro symbol: €"
after = before.encode("utf-8", errors="replace")
print(after)  # Output: b'This is the euro symbol: \xe2\x82\xac'
print(after.decode("utf-8"))  # Correct decoding
```

- Incorrect decoding results in errors

```
print(after.decode("ascii"))  # UnicodeDecodeError
```

Handling Encoding Issues in Files

- Reading files with encoding issues might cause **`UnicodeDecodeError`**.
- Use **`charset_normalizer`** to detect encoding

```
import charset_normalizer
with open("file.csv", 'rb') as rawdata:
    result = charset_normalizer.detect(rawdata.read(10000))
print(result)  # Example output: {'encoding': 'Windows-1252', 'confidence': 0.73}
```

- Read file with the detected encoding

```
df = pd.read_csv("file.csv", encoding='Windows-1252')
```

Converting Data to UTF-8

- Convert text to UTF-8 as soon as possible.
- Avoid lossy encoding conversions

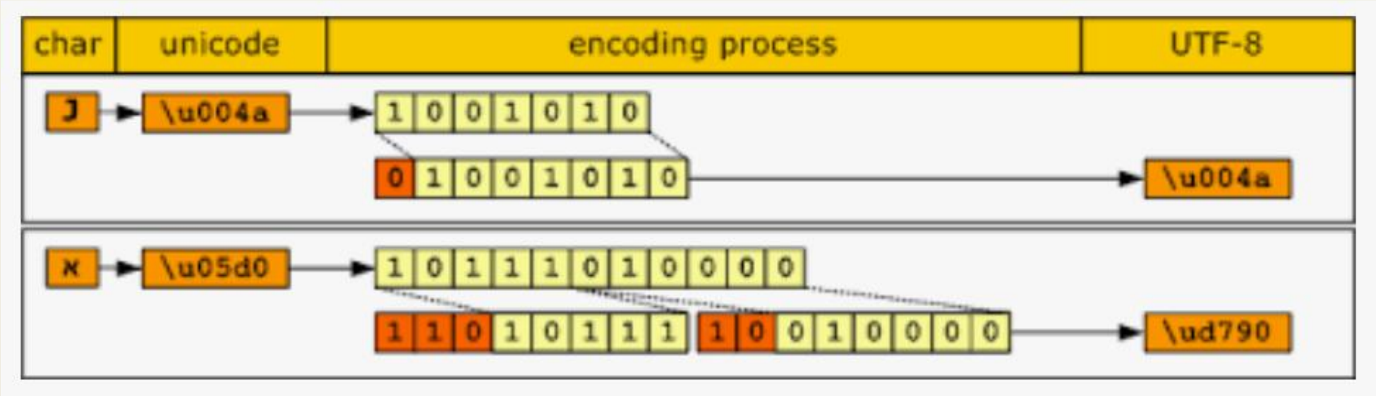
```
before = "This is the euro symbol: €"
after = before.encode("ascii", errors="replace")
print(after.decode("ascii"))  # Output: "This is the euro symbol: ?" (lost characte
```

- Save files with UTF-8 encoding

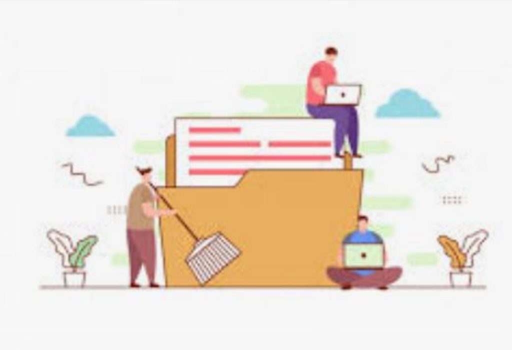
```
df.to_csv("file_utf8.csv", encoding='utf-8', index=False)
```

Character Encodings - Suggestions

- Always prefer UTF-8 encoding.
- Detect encoding issues early with **`charset_normalizer`**.
- Handle encoding conversions carefully to prevent data loss.
- Save files explicitly in UTF-8 to maintain consistency.



5. Inconsistent Data Entry



Need of Data Cleaning

- Ensures data consistency and accuracy.
- Reduces errors and improves analysis.
- Essential for handling large datasets efficiently.

Common Issues in Text Data

- Inconsistent capitalizations (e.g., "Germany" vs. "germany").
- Extra white spaces (e.g., " Germany" vs. "Germany").
- Misspellings and variations (e.g., "South Korea" vs. "SouthKorea").
- Abbreviations and alternative spellings (e.g., "USA" vs. "USofA").

Initial Data Inspection

- Before cleaning, always inspect the dataset to understand inconsistencies.
- Use `head()` and `unique()` to identify variations in text-based data.

```
# Load dataset
import pandas as pd
import numpy as np

professors = pd.read_csv("../input/pakistan-intellectual-capital/pakistan_intellectual_

# Inspect first few rows
print(professors.head())

# Check unique values in a column (e.g., Country)
countries = professors['Country'].unique()
countries.sort()
print(countries)
```

Standardizing Text Data

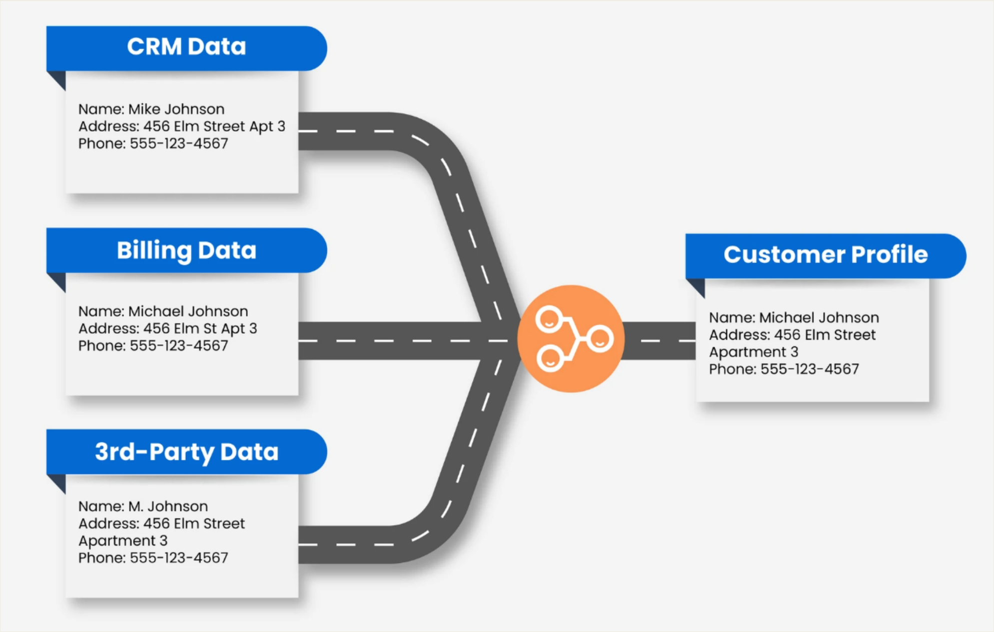
- Convert text to lowercase to maintain uniformity.
- Remove leading and trailing whitespace.

```
# Convert to lowercase
professors['Country'] = professors['Country'].str.lower()

# Remove leading and trailing whitespaces
professors['Country'] = professors['Country'].str.strip()
```

Using Fuzzy Matching for Data Cleaning

- Fuzzy matching helps detect and correct variations in text entries.
- The fuzzywuzzy package can be used to find similar text strings.

```python
from fuzzywuzzy import process, fuzz

# Get top 10 closest matches to 'south korea'
matches = process.extract("south korea", countries, limit=10, scorer=fuzz.token_sort_ratio)
print(matches)
```



Automating Inconsistent Data Correction

- A function can be written to replace similar text entries automatically.

```python
# Function to replace similar text values
def replace_matches_in_column(df, column, string_to_match, min_ratio=47):
    # Get unique values
    strings = df[column].unique()

    # Get closest matches
    matches = process.extract(string_to_match, strings, limit=10, scorer=fuzz.token_sort_ratio)

    # Filter matches above similarity threshold
    close_matches = [match[0] for match in matches if match[1] >= min_ratio]

    # Replace values in DataFrame
    df.loc[df[column].isin(close_matches), column] = string_to_match

    print("All done!")

# Apply function to clean 'Country' column
replace_matches_in_column(professors, 'Country', "south korea")
```

Verifying Data Cleaning

- After making changes, always recheck unique values to confirm corrections.

```
# Check updated unique values
countries = professors['Country'].unique()
countries.sort()
print(countries)
```

Task: Cleaning Inconsistent University Names in a Dataset

Problem Statement
A dataset contains a column "University Currently Teaching", but due to inconsistent data entry, some universities have multiple variations.
For example:

Original Entry
myu University
Muslim University
MY University
Maslim university
Stanford Univ.
Stanford University

Your task is to standardize all entries to **“MY University"** & Stanford University using preprocessing techniques.

Code

```
import pandas as pd
from fuzzywuzzy import process, fuzz
# Sample dataset
data = {'University Currently Teaching': [
    'myu University', 'Muslim University', 'MY University', 'Maslim university',
    'Stanford Univ.', 'Stanford University',
]}
df = pd.DataFrame(data)
# Convert text to lowercase and remove extra spaces
df['University Currently Teaching'] = df['University Currently Teaching'].str.lower().str.strip()
# Function to standardize university names
def replace_matches_in_column(df, column, string_to_match, min_ratio=80):
    unique_values = df[column].unique()
 # Find close matches
    matches = process.extract(string_to_match, unique_values, limit=10,
scorer=fuzz.token_sort_ratio)
```

Code

```
# Get matches above similarity threshold
    close_matches = [match[0] for match in matches if match[1] >= min_ratio]
    # Replace matches in the dataset
    df.loc[df[column].isin(close_matches), column] = string_to_match
    print(f"Replaced {close_matches} with '{string_to_match}'")
# Standardize MY University entries
replace_matches_in_column(df, 'University Currently Teaching', "my university")
# Standardize Stanford University entries
replace_matches_in_column(df, 'University Currently Teaching', "stanford university")
# View cleaned dataset
print(df)
```