# 15 - working with CCSV files

03 November 2024 19:48

# Working with CSV Files in Machine Learning

## Introduction

- **Welcome Message**: Hello guys, welcome to my YouTube channel dedicated to machine learning.
- **Progress Update**: We are gradually advancing through the machine learning process.
- **Previous Video Recap**: In the last video, we learned how to frame a machine learning problem.
- **Current Focus**: In the next three to four videos, we will concentrate on data gathering and preprocessing.

## Importance of Data in Machine Learning

- **Data Quality Impact**: The performance of a machine learning model is heavily dependent on the quality and quantity of data.
  - **Good Data**: Leads to better model performance.
  - **Poor Data**: Results in subpar model performance.
- **Goal**: Learn various methods to acquire and handle data effectively.

## Data Acquisition Strategies

- **Total Videos**: We will cover this topic over four videos.
- **Objective**: Step-by-step guide on how to obtain data when given a problem statement.
- **Key Learning**: Understand different sources and techniques to gather the required data.

## Video Breakdown

1. **Working with CSV Files**
   - **Focus**: Reading and handling CSV files using pandas.
   - **Reason**: CSV is the most commonly used data format, especially for beginners.
2. **Working with JSON Files**
   - **Focus**: Reading and handling JSON data.
   - **Use Case**: JSON is widely used in web APIs and data interchange between applications.
3. **Fetching Data from APIs**
   - **Focus**: Retrieving data from web APIs (e.g., IPL data).
   - **Benefit**: Learn how to modify and use data according to requirements.
4. **Web Scraping**
   - **Focus**: Extracting data from websites when APIs are not available.
   - **Tools**: Use Python libraries to parse HTML and extract useful information.

## Why Start with CSV Files?

- **Prevalence**: Approximately 90% of datasets you encounter initially will be in CSV format.
- **Ease of Use**: CSV files are straightforward to handle compared to other formats.
- **Foundation**: Building proficiency with CSV files prepares you for more complex data sources.

## Setting Up the Environment

- **Tools Used**:
  - **Jupyter Notebook**: For interactive coding and visualization.
  - **Pandas Library**: For data manipulation and analysis.
- **Dataset**: Various CSV files uploaded into the working directory.

## Understanding CSV Files

- **Definition**: CSV stands for Comma-Separated Values.
- **Structure**:
  - Each row represents a record.
  - Columns are separated by commas.
- **Variations**:
  - **TSV Files**: Tab-Separated Values, where columns are separated by tabs.
  - **Handling Different Delimiters**: Learn to specify custom delimiters in pandas.

# Reading CSV Files with Pandas

## Basic Usage

- **Function**: pd.read_csv()
- **Purpose**: Load CSV data into a pandas DataFrame.
- **Syntax**:
  python
  Import pandas as pd
  df = pd.read_csv('filename.csv')

## Reading Local Files

- **Method**:
  - Place the CSV file in the same directory as your notebook.
  - Use the file name directly in read_csv().
- **Example**:

  python
  df = pd.read_csv('aug_train.csv')

## Reading Files from URLs

- **Scenario**: When the CSV file is hosted online.
- **Steps**:
  - Use the requests library to fetch the content.
  - Read the content into pandas.
- **Code Snippet**:

  python
  Import pandas as pd
  Import requests
  From io import StringIO
  url = 'http://example.com/data.csv'content = requests.get(url).content
  df = pd.read_csv(StringIO(content.decode('utf-8')))

# Important Parameters in pd.read_csv()

## 1. sep (Separator)

- **Purpose**: Specify the delimiter used in the file.
- **Default**: ',' (comma)
- **Usage**:
  - For tab-separated files: sep='\t'
- **Example**:

  python
  df = pd.read_csv('file.tsv', sep='\t')

## 2. names

- **Purpose**: Define custom column names.

- **Usage**:
  - When the file does not contain header rows.
- **Example**:

  python
  column_names = ['Serial No', 'Movie Name', 'Release Year', 'Rating']
  df = pd.read_csv('movies.csv', names=column_names)

## 3. index_col

- **Purpose**: Set a specific column as the index of the DataFrame.
- **Usage**:
  - Provide the column name or index.
- **Example**:

  python

  df = pd.read_csv('data.csv', index_col='ID')

## 4. header

- **Purpose**: Specify the row number to use as column names.
- **Usage**:
  - Use header=None if there is no header row in the file.
- **Example**:

  python
  df = pd.read_csv('data.csv', header=None, names=column_names)

## 5. usecols

- **Purpose**: Read only specific columns.
- **Usage**:
  - Provide a list of column names or indices.
- **Example**:

  python
  df = pd.read_csv('data.csv', usecols=['ID', 'Gender', 'Education_Level'])

## 6. squeeze

- **Purpose**: Return a Series instead of a DataFrame when reading a single column.
- **Usage**:
  - Set squeeze=True when reading one column.
- **Example**:

  python

  df = pd.read_csv('data.csv', usecols=['ID'], squeeze=True)

## 7. skiprows

- **Purpose**: Skip specified rows while reading the file.
- **Usage**:
  - Provide a list of row indices to skip.
- **Example**:

python

df = pd.read_csv('data.csv', skiprows=[0, 2])

## 8. nrows

- **Purpose**: Read a limited number of rows.
- **Usage**:
  - Useful for loading large files partially.
- **Example**:

  python

  df = pd.read_csv('data.csv', nrows=100)

## 9. encoding

- **Purpose**: Specify the character encoding of the file.
- **Default**: 'utf-8'
- **Usage**:
  - When dealing with files in different encodings (e.g., 'ISO-8859-1').
- **Example**:

  python

  df = pd.read_csv('data.csv', encoding='ISO-8859-1')

## 10. error_bad_lines

- **Purpose**: Skip lines with too many fields (irregular data).
- **Usage**:
  - Set error_bad_lines=False to skip problematic lines.
- **Example**:

  python

  df = pd.read_csv('data.csv', error_bad_lines=False)

## 11. dtype

- **Purpose**: Force specific data types for columns.
- **Usage**:
  - Provide a dictionary mapping column names to data types.
- **Example**:

  python

  df = pd.read_csv('data.csv', dtype={'Age': int, 'Salary': float})

## 12. parse_dates

- **Purpose**: Parse columns as datetime objects.
- **Usage**:
  - Provide a list of columns to parse.
- **Example**:

python

```
df = pd.read_csv('data.csv', parse_dates=['Date'])
```

## 13. converters

- **Purpose**: Apply custom functions to transform column data.
- **Usage**:
  - Provide a dictionary mapping column names to functions.
- **Example**:

python

```
def shorten_team_name(name):
  return name[:3].upper()
df = pd.read_csv('ipl.csv', converters={'Team': shorten_team_name})
```

## 14. na_values

- **Purpose**: Specify additional strings to recognize as NaN.
- **Usage**:
  - Provide a list of strings to consider as missing values.
- **Example**:

python

```
df = pd.read_csv('data.csv', na_values=['?', 'N/A', 'null'])
```

## 15. chunksize

- **Purpose**: Read the file in chunks (useful for large datasets).
- **Usage**:
  - Specify the number of rows per chunk.
- **Example**:

python

```
chunks = pd.read_csv('large_data.csv', chunksize=5000)
For chunk in chunks:
  # Process each chunk process(chunk)
```

# Practical Examples

## Handling Different Delimiters

- **Tab-Separated Files**:
  - Use sep='\t' for TSV files.
- **Example**:

python

```
df = pd.read_csv('data.tsv', sep='\t')
```

## Dealing with Missing Headers

- **Scenario**: First row contains data, not column names.
- **Solution**:
  - Use header=None and provide names.
- **Example**:

```
python
column_names = ['ID', 'Name', 'Age']
df = pd.read_csv('data.csv', header=None, names=column_names)
```

## Selecting Specific Columns

- **Reading Only Necessary Data**:
  - Reduces memory usage and processing time.
- **Example**:

```
python

df = pd.read_csv('data.csv', usecols=['ID', 'Score'])
```

## Skipping Rows

- **Ignoring Unnecessary Data**:
  - Skip rows that are not needed for analysis.
- **Example**:

```
python

df = pd.read_csv('data.csv', skiprows=range(1, 1000))
```

## Reading Limited Rows

- **Previewing Data**:
  - Read the first few rows to understand the data structure.
- **Example**:

```
python

df = pd.read_csv('data.csv', nrows=5)
```

## Handling Encoding Issues

- **Common Encodings**:
  - UTF-8 (default), ISO-8859-1, Windows-1252.
- **Detecting Encoding**:
  - Use a text editor or the chardet library to find the encoding.
- **Example**:

```
python

df = pd.read_csv('data.csv', encoding='ISO-8859-1')
```

## Dealing with Irregular Data

- **Problem**:
  - Lines with incorrect number of fields cause parsing errors.
- **Solution**:
  - Use error_bad_lines=False to skip problematic lines.
- **Example**:

```
python
```

```
df = pd.read_csv('data.csv', error_bad_lines=False)
```

## Parsing Dates Correctly

- **Multiple Date Columns**:
  - Combine multiple columns into a single datetime column.
- **Example**:

  python

  ```
  df = pd.read_csv('data.csv', parse_dates=[['Year', 'Month', 'Day']])
  ```

## Using Converters

- **Custom Data Transformation**:
  - Apply functions to clean or modify data during import.
- **Example**:

  python

  ```
  def clean_name(name):
    return name.strip().upper()
  df = pd.read_csv('data.csv', converters={'Name': clean_name})
  ```

## Handling Missing Values

- **Custom Missing Indicators**:
  - Recognize custom strings as NaN. If there is something that is missed
- **Example**:

  python
  ```
  df = pd.read_csv('data.csv', na_values=['--', '??'])
  ```

## Chunking Large Files

- **Processing in Batches**:
  - Use chunksize to read and process data in parts.
- **Example**:

  python

  ```
  chunks = pd.read_csv('large_data.csv', chunksize=10000)
  For chunk in chunks:
    # Perform operations on each chunk result = chunk.groupby('Category').sum()
  ```

# Tips and Best Practices

- **Always Check Data Types**:
  - Use df.info() to inspect data types after loading.
- **Handle Missing Data Early**:
  - Identify and address missing values during the import process.
- **Optimize Memory Usage**:
  - Specify data types using dtype to reduce memory footprint.
- **Be Aware of Encoding**:
  - Incorrect encoding can lead to misread data or errors.

# Conclusion

- **Mastering pd.read_csv()**:
  - Understanding the parameters allows you to handle various data import scenarios.
- **Future Learning**:
  - In the next video, we will explore working with JSON files.
- **Practice**:
  - Try loading different datasets and experiment with the parameters.