Week 7: Comprehensive Lecture Notes on Pandas (Part 2)

1 Introduction

Pandas is a powerful library for data manipulation and analysis, widely used in data science. In this session, we will explore advanced techniques that can enhance efficiency and performance when working with large datasets.

2 Performance Optimization in Pandas

2.1 Using Efficient Data Types

- Use `astype()` to optimize memory usage. - Convert categorical data using `pd.Categorical()`. - Convert datetime columns to `datetime64` for efficient operations.

```
import pandas as pd
import numpy as np

df = pd.DataFrame({
    'category': ['A', 'B', 'A', 'C'],
    'value': [10, 20, 15, 30]
})

df['category'] = df['category'].astype('category')  # Optimized memory usage
print(df.dtypes)
```

2.2 Vectorized Operations vs. Apply

- Use vectorized operations instead of `apply()` or loops for better performance.

```
df['new_value'] = df['value'] * 2  # Vectorized operation
```

- Avoid `apply()` for row-wise operations unless necessary.

```
df['new_value'] = df.apply(lambda row: row['value'] * 2, axis=1)  # Slower
```

2.3 Using `eval()` and `query()`

- `eval()` and `query()` offer faster alternatives to standard pandas operations.

```
df.eval('new_value = value * 2', inplace=True)  # Faster than apply
df_filtered = df.query('value > 15')  # Efficient filtering
```

3 Advanced Data Manipulation

3.1 MultiIndex for Hierarchical Data

- MultiIndex allows better organization of hierarchical data.

```
arrays = [
    ['A', 'A', 'B', 'B'],
    [1, 2, 1, 2]
]

index = pd.MultiIndex.from_arrays(arrays, names=('Group', 'Num'))
df = pd.DataFrame({'Values': [10, 20, 30, 40]}, index=index)
print(df)
```

- Accessing specific data using `.loc[]`:

```
print(df.loc['A', 1])
```

3.2 Pivot Tables

A pivot table is a powerful tool in pandas that allows you to summarize and reorganize data in a tabular format. It is used to aggregate data in a way that makes it easier to analyze, similar to Excel pivot tables.

- Pivot tables summarize data efficiently.

```
df = pd.DataFrame({
    'Category': ['A', 'A', 'B', 'B'],
    'Type': ['X', 'Y', 'X', 'Y'],
    'Value': [10, 20, 15, 25]
})

pivot_df = df.pivot_table(values='Value', index='Category', columns='Type', aggfunc='sum')
print(pivot_df)
```

3.3 Window Functions (Rolling & Expanding)

- Rolling windows compute statistics over moving time windows.

```
df['rolling_avg'] = df['Value'].rolling(window=2).mean()
print(df)
```

- Expanding applies functions over an expanding window.

```
df['expanding_sum'] = df['Value'].expanding().sum()
print(df)
```

4 Handling Large Datasets

4.1 Chunk Processing

- Use `chunksize` when reading large files.

```
chunks = pd.read_csv('large_file.csv', chunksize=10000)
for chunk in chunks:
    print(chunk.head())
```

4.2 Dask for Parallel Processing

- Dask provides parallel computing for pandas operations.

```
import dask.dataframe as dd
ddf = dd.read_csv('large_file.csv')
print(ddf.head())
```

5 Load Datasets in Pandas

Pandas is a powerful Python library for **data analysis and manipulation**. In this tutorial, we will explore different ways to **load datasets** into Pandas from various file formats such as **CSV, Excel, JSON, SQL, Parquet, and more**.

By the end of this tutorial, you will be able to:

- Read data from different file formats into Pandas.
- Work with large datasets efficiently.
- Load data from online sources and databases.

5.1 Load a CSV File

CSV (Comma-Separated Values) is the most common file format for datasets.

```
import pandas as pd

# Load a CSV file
df = pd.read_csv("dataset.csv")

# Display the first 5 rows
print(df.head())
```

5.1.1 Additional Options for `pd.read_csv()`

- `delimiter=";"` → Use if the file is separated by semicolons.
- `header=None` → Use if the file does not contain column names.
- `index_col=0` → Use if the first column should be used as an index.

5.2 Load an Excel File

Pandas also supports reading data from Excel files (`.xlsx`, `.xls`).

```
df = pd.read_excel("dataset.xlsx", sheet_name="Sheet1")  # Specify sheet name if needed
print(df.head())
```

5.2.1 Install Required Library

Excel reading requires `openpyxl` or `xlrd`. If not installed, run:

```
pip install openpyxl
```

5.3 Load a JSON File

JSON (JavaScript Object Notation) is commonly used for **web APIs and structured data**.

```
df = pd.read_json("dataset.json")
print(df.head())
```

5.3.1 Read JSON from an API

```
import requests

# Fetch JSON from an API
url = "https://api.example.com/data"
response = requests.get(url)

# Convert JSON to DataFrame
df = pd.DataFrame(response.json())
print(df.head())
```

5.4 Load Data from a SQL Database

If your dataset is stored in a **SQL database**, you can directly import it using Pandas.

```
import sqlite3

# Connect to a database
conn = sqlite3.connect("database.db")

# Run a SQL query
df = pd.read_sql_query("SELECT * FROM my_table", conn)
print(df.head())
```

5.4.1 Read from MySQL or PostgreSQL

Install `SQLAlchemy` if using MySQL/PostgreSQL:

```
pip install sqlalchemy
```

5.5 Load a Parquet File (Efficient for Large Data)

Parquet is a high-performance file format often used in **big data applications**.

```
df = pd.read_parquet("dataset.parquet")
print(df.head())
```

5.5.1 Install Parquet Support

If you get an error, install `pyarrow`:

```
pip install pyarrow
```

5.6 Load a Dataset from a URL

If your dataset is hosted online, you can load it directly using a URL.

```
url = "https://example.com/dataset.csv"
df = pd.read_csv(url)
print(df.head())
```

Works for: CSV, JSON, and other file formats.

5.7 Load Data from a ZIP File

If your dataset is compressed inside a ZIP file, Pandas can read it directly.

```
df = pd.read_csv("dataset.zip", compression="zip")
print(df.head())
```

5.8 Load a Dataset from Google Sheets

Google Sheets can be **converted into a CSV link** and loaded into Pandas.

```
sheet_url = "https://docs.google.com/spreadsheets/d/example/export?format=csv"
df = pd.read_csv(sheet_url)
print(df.head())
```

Exercises: Pandas Problems and Solutions

Problem 1: Optimize Memory Usage with Categorical Data

You have the following dataset:

```
import pandas as pd

df = pd.DataFrame({
    'Category': ['Apple', 'Banana', 'Apple', 'Orange', 'Banana', 'Apple'],
    'Value': [10, 20, 30, 40, 50, 60]
})

print(df.dtypes)
```

Task: Convert the "Category" column to an optimized data type.
Solution:

```
df['Category'] = df['Category'].astype('category')
print(df.dtypes)
```

Problem 2: Use Vectorized Operations Instead of `apply()`

Given the dataset:

```
df = pd.DataFrame({'Value': [10, 20, 30, 40, 50]})
```

Task: Double the values in the "Value" column using a vectorized operation.
Solution:

```
df['Double_Value'] = df['Value'] * 2
print(df)
```

Problem 3: Use `eval()` for Efficient Computation

Compute a new column named "Triple_Value" using `eval()`.
Solution:

```
df.eval('Triple_Value = Value * 3', inplace=True)
print(df)
```

Problem 4: Filter Data Using `query()`

Given:

```
df = pd.DataFrame({
    'Name': ['Alice', 'Bob', 'Charlie', 'David'],
    'Age': [25, 30, 35, 40]
})
```

Task: Filter out rows where Age is greater than 30.
Solution:

```
filtered_df = df.query('Age > 30')
print(filtered_df)
```

Problem 5: Use MultiIndex for Hierarchical Data

Given:

```
arrays = [
    ['USA', 'USA', 'Canada', 'Canada'],
    ['East', 'West', 'East', 'West']
]

index = pd.MultiIndex.from_arrays(arrays, names=('Country', 'Region'))

df = pd.DataFrame({'Sales': [100, 150, 200, 250]}, index=index)
```

Task: Retrieve sales for Canada in the East region.
Solution:

```
print(df.loc['Canada', 'East'])
```

Problem 6: Create a Pivot Table

Given:

```
df = pd.DataFrame({
    'Department': ['HR', 'HR', 'IT', 'IT'],
    'Gender': ['Male', 'Female', 'Male', 'Female'],
    'Salary': [50000, 52000, 70000, 68000]
})
```

Task: Create a pivot table showing the average salary per department and gender.
Solution:

```
pivot_df = df.pivot_table(values='Salary', index='Department', columns='Gender', aggfunc='mean')
print(pivot_df)
```

Problem 7: Apply Rolling Window Mean

Given:

```
df = pd.DataFrame({'Values': [10, 20, 30, 40, 50]})
```

Task: Calculate the rolling average with a window size of 2.
Solution:

```
df['Rolling_Mean'] = df['Values'].rolling(window=2).mean()
print(df)
```

Problem 8: Process Large CSV Files in Chunks

Suppose you have a large CSV file called `large_file.csv`.
Task: Read and process the file in chunks of 5000 rows.
Solution:

```
chunks = pd.read_csv('large_file.csv', chunksize=5000)

for chunk in chunks:
    print(chunk.head())  # Process each chunk separately
```

Problem 9: Merge Two DataFrames

Given:

```
df1 = pd.DataFrame({'ID': [1, 2, 3], 'Value': [100, 200, 300]})
df2 = pd.DataFrame({'ID': [2, 3, 4], 'Score': [90, 80, 70]})
```

Task: Merge the DataFrames on "ID" using an inner join.
Solution:

```
merged_df = pd.merge(df1, df2, on='ID', how='inner')
print(merged_df)
```

Problem 10: Resample Time Series Data

Given:

```
df = pd.DataFrame({
    'Date': pd.date_range(start='2024-01-01', periods=10, freq='D'),
    'Sales': [100, 120, 130, 90, 150, 160, 200, 180, 210, 190]
})
df.set_index('Date', inplace=True)
```

Task: Resample the data to compute the mean sales per week.
Solution:

```
df_weekly = df.resample('W').mean()
print(df_weekly)
```

Problems to be Solved

Problem 1: Memory Optimization

You are given the following DataFrame:

```
import pandas as pd

df = pd.DataFrame({
    'Product': ['Laptop', 'Tablet', 'Laptop', 'Smartphone'],
    'Price': [1200, 600, 1100, 900],
    'Stock': [30, 50, 20, 80]
})
```

Optimize this DataFrame for memory usage without losing any information.

Solution:

```
df['Product'] = df['Product'].astype('category')
df['Price'] = df['Price'].astype('int16')
df['Stock'] = df['Stock'].astype('int16')
```

Problem 2: MultiIndexing

Create a MultiIndex DataFrame where products belong to different categories and cities. Then, retrieve all the products available in "New York."

Solution:

```
index = pd.MultiIndex.from_tuples([
    ('Electronics', 'New York', 'Laptop'),
    ('Electronics', 'New York', 'Smartphone'),
    ('Electronics', 'Chicago', 'Tablet')
], names=['Category', 'City', 'Product'])

df = pd.DataFrame({'Stock': [20, 30, 40]}, index=index)
df.xs('New York', level='City')
```

Problem 3: Merging and Joining

You have two DataFrames:

```
df1 = pd.DataFrame({'ID': [1, 2, 3], 'Name': ['Alice', 'Bob', 'Charlie']})
df2 = pd.DataFrame({'ID': [2, 3, 4], 'Salary': [50000, 60000, 70000]})
```

Perform an outer join on the "ID" column.

Solution:

```
df_merged = pd.merge(df1, df2, on='ID', how='outer')
```

Problem 4: Pivot Tables

Create a pivot table from the following DataFrame to show total sales per product per region.

```
df = pd.DataFrame({
    'Region': ['North', 'South', 'North', 'South'],
    'Product': ['A', 'B', 'A', 'B'],
    'Sales': [100, 200, 150, 250]
})
```

Solution:

```
df.pivot_table(values='Sales', index='Region', columns='Product', aggfunc='sum')
```

Problem 5: Custom Aggregations

Calculate the range (max - min) of sales for each region.

```
df = pd.DataFrame({
    'Region': ['East', 'East', 'West', 'West'],
    'Sales': [100, 150, 200, 250]
})
```

Solution:

```
df.groupby('Region')['Sales'].agg(lambda x: x.max() - x.min())
```

Problem 6: Loading and Working with Datasets

- Try loading a dataset using different formats.
- Explore **data cleaning and transformation** with Pandas.
- Practice with **real-world datasets** (Kaggle, UCI, etc.).