SAMSUNG INNOVATION CAMPUS

CODING AND PROGRAMMING

Project Title:

"RETAIL STORE SALES TRENDS ANALYSIS USING TIME-SERIES DATA"

Description:

- The document outlines a **Retail Store Sales Trend Analysis using Time-Series Data** with Pandas.
- It defines the **objective** as analysing daily sales records to identify trends, seasonality, and performance.
- The project is broken into **step-by-step tasks**: loading data, cleaning, time-series manipulation, filtering, grouping, and aggregation.
- It includes creating **derived columns** like cumulative sales and sales categories (High, Medium, Low).
- Finally, results are **exported** into cleaned datasets and summary CSV files for store-wise and weekday-wise sales.

Member Details:

Name:	Swasti M Petkar
USN:	4GW23CI055
Department:	CSE(AI&ML)
Year:	Pre-final Year
Submission date:	02/09/2025
College:	GSSS Institute of Engineering and Technology for Women, Mysuru

Index:

Detailed Explanation on Tech-Stack And their Features :

1. Flask (Python Web Framework)

- **Why it is Used:**
  Flask is a lightweight Python web framework that lets us quickly build web applications with routing, templates, and file handling.
- **Features in your project:**
  - Handles **routes** like / (home page), /upload (CSV upload & analysis), and /download/<filename> (download processed files).
  - Provides integration with **Jinja2 templating engine** to pass processed data into HTML templates (dashboard.html).
  - Easy to scale with extensions (authentication, database integration if needed later).

2. Pandas (Python Data Analysis Library)

- **Why it is Used:**
  Pandas is perfect for **time-series and tabular data analysis** like our retail sales dataset.
- **Features in my project:**
  - **Data Loading & Cleaning**
    - Reads CSV file.
    - Handles missing values.
    - Removes duplicates.
    - Converts dates to datetime.
  - **Time-Series Manipulation**
    - Sets Date as index.
    - Extracts **Weekday** and **Month** columns for grouping.
  - **Filtering & Conditions**
    - Get data for specific StoreID.
    - Filter sales above thresholds.
  - **Aggregations & Grouping**

        - Store-wise total sales.
        - Month-wise average daily sales.
        - Weekday average sales.
    - **Derived Columns**
        - CumulativeSales
        - SalesCategory
    - **Exports CSVs** for cleaned data and summaries (to_csv).

3. HTML + CSS + Bootstrap (Frontend UI)

- **Why Used:**
  To provide a **clean, responsive, and professional dashboard** UI without writing a lot of CSS from scratch.
- **Features in my project:**
    - **index.html**
        - Upload form (CSV file upload).
        - Modern card-based layout.
        - Bootstrap styling (btn, card, form-control).
    - **dashboard.html**
        - Tables showing **store totals**, **weekday averages**, **top 3 stores**.
        - Responsive layout with Bootstrap grid system.
        - Buttons to **download processed CSVs** (cleaned data, summaries).

4. Chart.js (JavaScript Charting Library)

- **Why Used:**
  For **interactive and modern data visualizations** on the frontend.
- **Features in my project:**
    - **Bar Chart** → Store-wise total sales.
    - **Line Chart** → Average sales by weekday.

- **Pie Chart** → Store contribution to total sales.
- Fully integrated with Flask/Jinja → Data dynamically passed from backend (tables dictionary).

5. Jinja2 Templating (Flask's Template Engine)

- **Why Used:**
  Allows embedding Python data directly into HTML.
- **Features in my project:**
  - Loops ({% for row in tables.store_totals %}) to dynamically generate table rows and chart labels.
  - Makes dashboard **dynamic** based on uploaded file.

6. File Handling (OS + Flask Send_file)

- **Why Used:**
  To let users upload their own CSV, process it, and download results.
- **Features in my project:**
  - uploads/ → Raw uploaded CSV files.
  - processed/ → Cleaned & summary CSVs.
  - /download/<filename> → Route to download results.

End-to-End Flow (Features Together)

1. User uploads a CSV file via index.html.
2. Flask (/upload) saves the file → Pandas reads it.
3. Data is cleaned, manipulated, and analysed with Pandas.
4. Results (summaries, derived columns) are saved as CSVs in processed/.
5. Flask renders dashboard.html:
   - Tables show results.
   - Charts visualize insights (store totals, weekday patterns, contributions).
   - Buttons let users download processed data.

6. Everything runs in a clean UI with **Bootstrap styling + Chart.js interactivity**.

Algorithm and its explanation:

Algorithm Steps

Step 1: Start the Application

1. Launch Flask backend server (app.py).
2. Open the web page (frontend) at http://127.0.0.1:5000.

Step 2: Upload Dataset

3. User selects a CSV file (retail_sales.csv) from local system using the upload form on the frontend.
4. Frontend sends the file to the Flask backend using an HTTP POST request to /analyze.

Step 3: Save and Process Data

5. Flask receives the file and saves it in the backend/ folder.
6. Flask calls analyze_sales(filepath) function from analysis.py.
7. Inside analyze_sales():
    - Load CSV into Pandas DataFrame.
    - Clean the data:
        - Fill missing Sales values with 0.
        - Remove duplicates.
        - Convert Date column to datetime.
    - Enhance the data:
        - Set Date as index.
        - Add new columns:
            - Weekday → Day name (Monday, Tuesday, etc.).
            - Month → Extract month number.
    - Perform Analysis:

        - Group by StoreID → Calculate total sales per store.
        - Group by Weekday → Calculate average sales per weekday.
    - Add Derived Columns:
        - CumulativeSales → Running total per store.
        - SalesCategory → Categorize sales as High, Medium, or Low.
    - Export Results:
        - cleaned_retail_sales.csv
        - store_sales_summary.csv
        - weekday_sales_summary.csv

Step 4: Send Response to Frontend

8. After processing:
    - Flask returns a JSON response with download links for all three output CSV files.

Step 5: Display Results

9. Frontend (JavaScript):
    - Reads the JSON response.
    - Dynamically shows Download Buttons for:
        - Cleaned Data
        - Store-wise Sales Summary
        - Weekday Sales Summary

Step 6: User Downloads Results

10. User clicks on buttons → Flask sends the requested CSV file for download.

Algorithm in Pseudocode

START

Display upload form on frontend

WAIT for user to upload CSV

ON file upload:

    SEND file to Flask /analyze route

Flask:

    Save file

    Call analyze_sales(file_path)

        Read CSV

        Clean and transform data

        Perform analysis (aggregations)

        Add new columns (Weekday, Month, CumulativeSales, Category)

        Save results to CSV files

    RETURN JSON with file download links

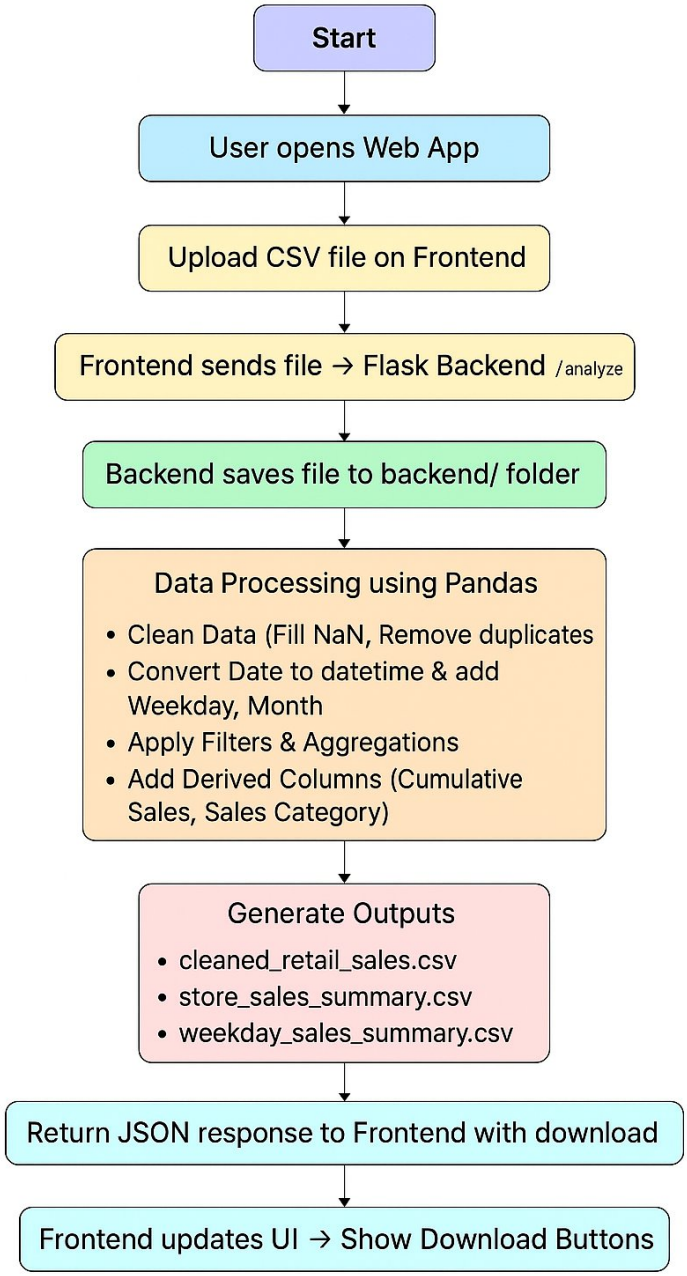
Frontend:

    Show download buttons

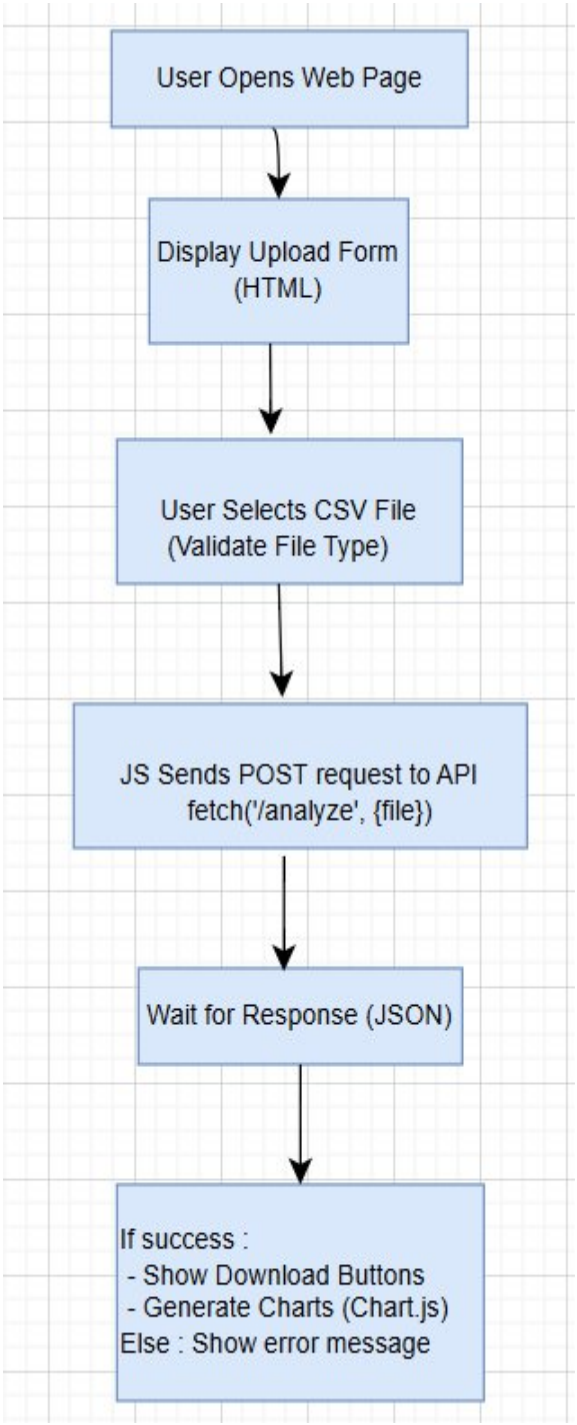
    (Optional: Render visual charts)

END

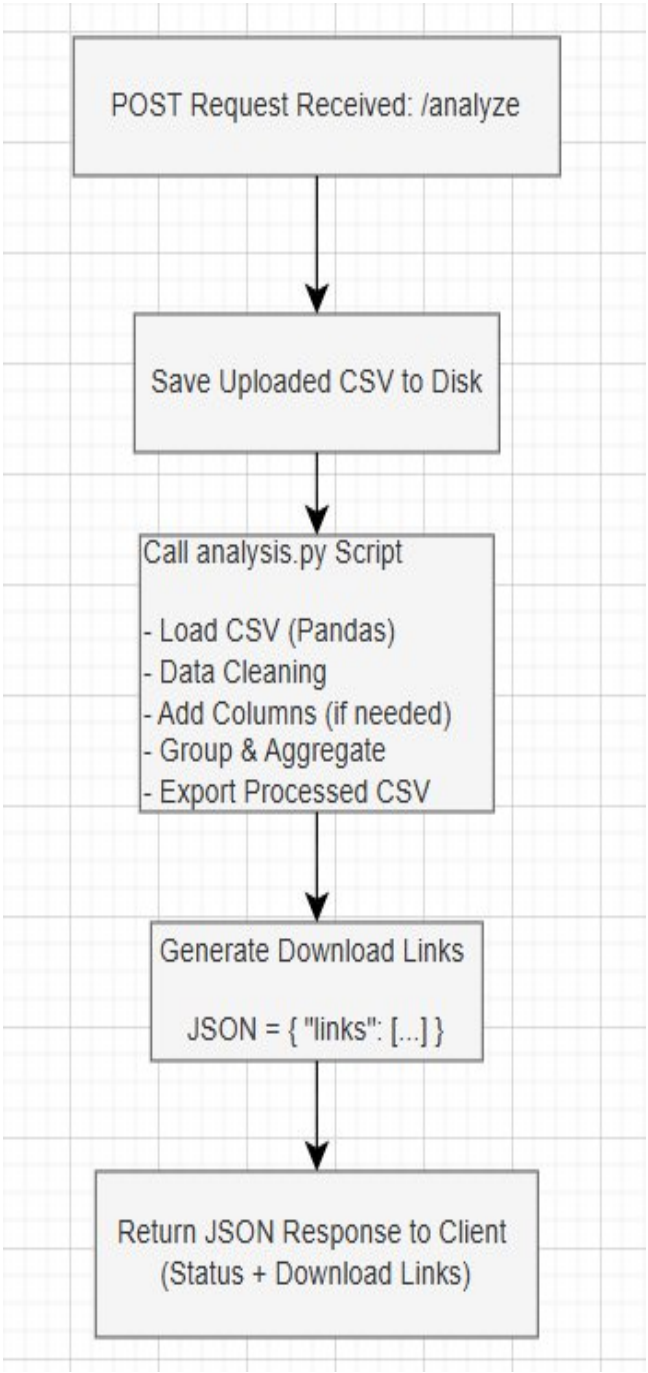
UML Diagram of the working of entire project :



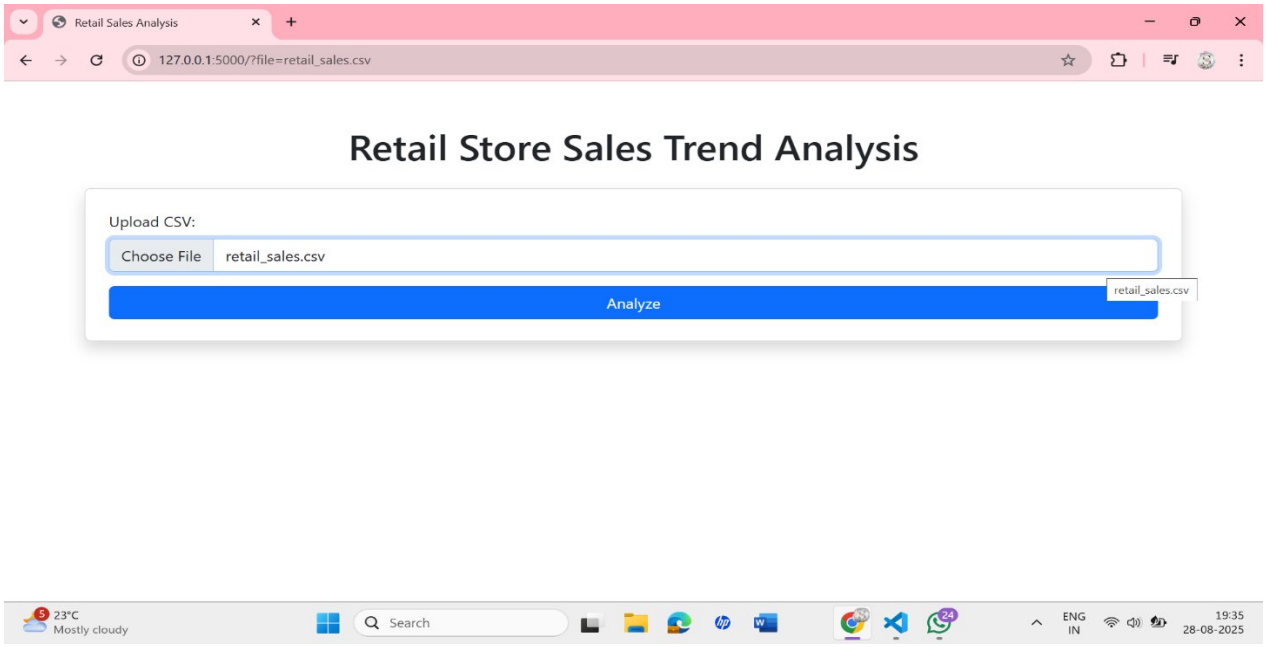
UML diagram of Frontend:

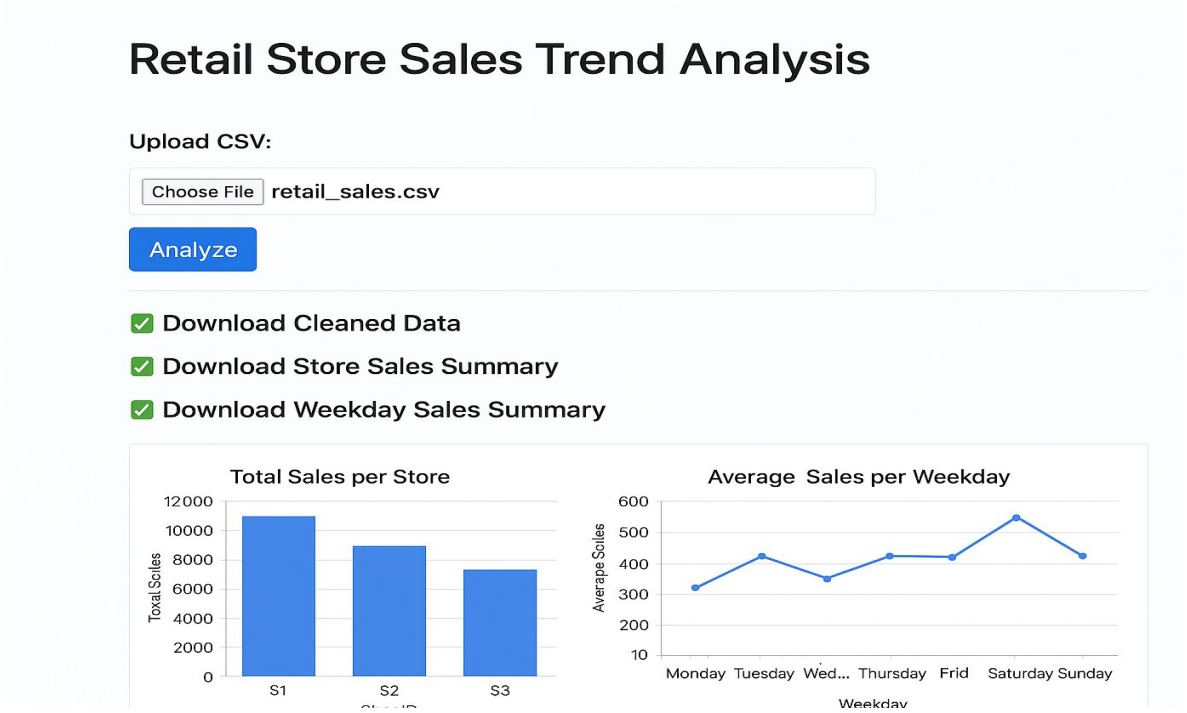


UML diagram of Backend:



Interface:





Code Explanation:

app.py

```python
backend > app.py > home
from flask import Flask, request, jsonify, send_file, render_template
import os
from analysis import analyze_sales

app = Flask(__name__, static_folder="../frontend", template_folder="../frontend")

UPLOAD_FOLDER = 'backend/'
os.makedirs(UPLOAD_FOLDER, exist_ok=True)

@app.route('/')
def home():
    return render_template('index.html')

@app.route('/analyze', methods=['POST'])
def analyze():
    file = request.files['file']
    filepath = os.path.join(UPLOAD_FOLDER, file.filename)
    file.save(filepath)

    # Call analysis function
    results = analyze_sales(filepath)

    return jsonify({
        "message": "Analysis completed",
        "cleaned_file": f"/download/{os.path.basename(results['cleaned'])}",
        "store_summary": f"/download/{os.path.basename(results['store_summary'])}",
        "weekday_summary": f"/download/{os.path.basename(results['weekday_summary'])}"
    })
```

```python
@app.route('/download/<filename>')
def download(filename):
    return send_file(os.path.join('backend/outputs/', filename), as_attachment=True)

if __name__ == '__main__':
    app.run(debug=True)
```

Working of app.py :

from flask import Flask, request, jsonify, send_file, render_template

- **Flask** - creates the app.
- **request** - lets you read incoming HTTP data (like uploaded files).
- **jsonify** - returns JSON responses.
- **send_file** - sends a file back to the browser (for download).
- **render_template** - renders HTML templates (e.g., index.html).

import os

from analysis import analyze_sales

- **os** - is for file paths and creating folders.
- **analyze_sales** - is your function (in analysis.py) that processes the uploaded file and returns paths to output files.

App setup and folders:

app = Flask(__name__, static_folder="../frontend", template_folder="../frontend")

- Creates a Flask app.
- Tells Flask that your **static files** (JS/CSS/images) and **templates** (HTML) live in ../frontend relative to where this script runs.
  That means your folder structure probably looks like:

```
UPLOAD_FOLDER = 'backend/'
os.makedirs(UPLOAD_FOLDER, exist_ok=True)
```

- Sets where uploaded files will be saved.
- Creates the folder if it doesn't exist.

Routes

1) Home page

```
@app.route('/')
def home():
    return render_template('index.html')
```

- When the browser hits /, Flask renders frontend/index.html.
- Inside index.html, you'll have a form like:

```
<form action="/analyze" method="POST" enctype="multipart/form-data">
        <input type="file" name="file" />
        <button type="submit">Analyze</button>
</form>
```

The important part is name="file"—that's how Flask finds the file in request.files['file'].

---

2) Analyse (file upload + run analysis)

```
@app.route('/analyze', methods=['POST'])
def analyze():
    file = request.files['file']
```

```
    filepath = os.path.join(UPLOAD_FOLDER, file.filename)
    file.save(filepath)

    results = analyze_sales(filepath)    # Call analysis function

    return jsonify({
        "message": "Analysis completed",
        "cleaned_file": f"/download/{os.path.basename(results['cleaned'])}",
        "store_summary": f"/download/{os.path.basename(results['store_summary'])}",
        "weekday_summary":
f"/download/{os.path.basename(results['weekday_summary'])}"
    })
```

What happens here:

1. request.files['file'] pulls the uploaded file object. (This fails if the form didn't include a file field.)
2. filepath decides where to save it (e.g., backend/sales.csv).
3. file.save(...) writes the file to disk.
4. analyze_sales(filepath) runs your analysis logic. **You** implement this function in analysis.py. It should:
    - Read the uploaded file.
    - Produce three result files (e.g., cleaned CSV, store summary, weekday summary).
    - Return a dict like:

```
{
  "cleaned": "backend/outputs/cleaned_sales.csv",
  "store_summary": "backend/outputs/store_summary.csv",
  "weekday_summary": "backend/outputs/weekday_summary.csv"
}
```

5. The response is JSON with human-readable "message" and three **download URLs** (e.g., /download/cleaned_sales.csv).

---

3) Download files

```
@app.route('/download/<filename>')
def download(filename):
    return send_file(os.path.join('backend/outputs/', filename), as_attachment=True)
```

- When you open /download/cleaned_sales.csv, Flask finds the file in backend/outputs/cleaned_sales.csv and sends it as a download (as_attachment=True).

---

4) Dev server

```
if __name__ == '__main__':
    app.run(debug=True)
```

- Runs the Flask dev server at http://127.0.0.1:5000/.
- debug=True auto-reloads on save and shows helpful errors. **Don’t** use debug in production.

Analysis.py

```
index.html | style.css | launch.json | analysis.py | script.js
backend > analysis.py > ...
import pandas as pd
import os

OUTPUT_FOLDER = 'backend/outputs/'
os.makedirs(OUTPUT_FOLDER, exist_ok=True)

def analyze_sales(filepath):
    # Load dataset
    df = pd.read_csv(filepath)

    # Data Cleaning
    df['Sales'].fillna(0, inplace=True)
    df.drop_duplicates(inplace=True)
    df['Date'] = pd.to_datetime(df['Date'])
    df.set_index('Date', inplace=True)
    df['Weekday'] = df.index.day_name()
    df['Month'] = df.index.month

    # Grouping & Aggregations
    store_sales = df.groupby('StoreID')['Sales'].sum()
    avg_weekday_sales = df.groupby('Weekday')['Sales'].mean()

    # Derived Columns
    df['CumulativeSales'] = df.groupby('StoreID')['Sales'].cumsum()
    df['SalesCategory'] = df['Sales'].apply(lambda x: 'High' if x>=5000 else 'Medium' if x>=3000 else 'Low')
```

```
    # Export results
    cleaned_path = os.path.join(OUTPUT_FOLDER, 'cleaned_retail_sales.csv')
    store_sales_path = os.path.join(OUTPUT_FOLDER, 'store_sales_summary.csv')
    weekday_sales_path = os.path.join(OUTPUT_FOLDER, 'weekday_sales_summary.csv')

    df.to_csv(cleaned_path)
    store_sales.to_csv(store_sales_path)
    avg_weekday_sales.to_csv(weekday_sales_path)

    return {
        "cleaned": cleaned_path,
        "store_summary": store_sales_path,
        "weekday_summary": weekday_sales_path
    }
```

Overview of working of analysis.py :

- Reads a **CSV file** (retail sales data).
- **Cleans and transforms the data** (fills missing values, removes duplicates, converts dates).
- Adds **new columns** (weekday, month, cumulative sales, sales category).
- Calculates **summaries**:
  - Total sales by store.
  - Average sales by weekday.
- Saves **three CSV files**: cleaned data, store summary, weekday summary.
- Returns the file paths for further use in the Flask app.

Imports

```
import pandas as pd
import os
```

- **pandas**: A powerful Python library for data analysis.
  - pd.read_csv(), groupby(), to_csv() are common operations.
- **os**: For interacting with the file system (creating folders, joining paths).

Output folder setup

```
OUTPUT_FOLDER = 'backend/outputs/'
os.makedirs(OUTPUT_FOLDER, exist_ok=True)
```

- OUTPUT_FOLDER is the directory where all output CSV files will be stored.
- os.makedirs(..., exist_ok=True):
  - Creates the folder if it does not exist.
  - exist_ok=True prevents an error if the folder already exists.

Function definition

```
def analyze_sales(filepath):
```

- This function takes **filepath** (path to the uploaded CSV) as input.
- Returns a **dictionary with paths** to the output files.

Step 1: Load dataset

```
df = pd.read_csv(filepath)
```

- Reads the CSV file into a **DataFrame** (think of it as an Excel table in memory).
- df now holds your entire dataset (columns like Date, StoreID, Sales).

Step 2: Data Cleaning

```
df['Sales'].fillna(0, inplace=True)
```

- Fills any missing values in the **Sales** column with 0.
- inplace=True means changes happen directly in df (no need to assign back).

df.drop_duplicates(inplace=True)

- Removes duplicate rows from the dataset.

df['Date'] = pd.to_datetime(df['Date'])

- Converts the **Date** column from text (string) to actual **date objects**.

  Because date operations (like extracting weekday or month) require proper datetime type.

df.set_index('Date', inplace=True)

- Makes **Date** the index of the DataFrame instead of the default integer index (0,1,2…).
- This helps when grouping by time.

df['Weekday'] = df.index.day_name()

df['Month'] = df.index.month

- Adds two new columns:
  - Weekday → Monday, Tuesday, etc.
  - Month → numeric month (1–12).

Step 3: Grouping & Aggregations

store_sales = df.groupby('StoreID')['Sales'].sum()

- Groups data by StoreID and **sums Sales** for each store.
- Example:

  StoreID | Sales

  101 | 12000

  102 | 8500

avg_weekday_sales = df.groupby('Weekday')['Sales'].mean()

- Groups data by weekday (Mon–Sun) and **calculates average sales**.
- Example:

  Weekday | Avg Sales

  Monday | 2500

  Tuesday | 3000

Step 4: Derived Columns

df['CumulativeSales'] = df.groupby('StoreID')['Sales'].cumsum()

- Adds a running total of sales for each store (cumulative sum).
- For Store 101:

  Sales | CumulativeSales

  1000 | 1000

  2000 | 3000

  1500 | 4500

df['SalesCategory'] = df['Sales'].apply(

  lambda x: 'High' if x >= 5000 else 'Medium' if x >= 3000 else 'Low'

)

- Classifies each sale as:
  - **High** if ≥ 5000
  - **Medium** if ≥ 3000
  - **Low** otherwise
- This uses a **lambda function** (anonymous function) applied to each Sales value.

Step 5: Export results

cleaned_path = os.path.join(OUTPUT_FOLDER, 'cleaned_retail_sales.csv')

store_sales_path = os.path.join(OUTPUT_FOLDER, 'store_sales_summary.csv')

weekday_sales_path = os.path.join(OUTPUT_FOLDER, 'weekday_sales_summary.csv')

- Prepares **file paths** for saving results.

df.to_csv(cleaned_path)

store_sales.to_csv(store_sales_path)

avg_weekday_sales.to_csv(weekday_sales_path)

- Saves it as :

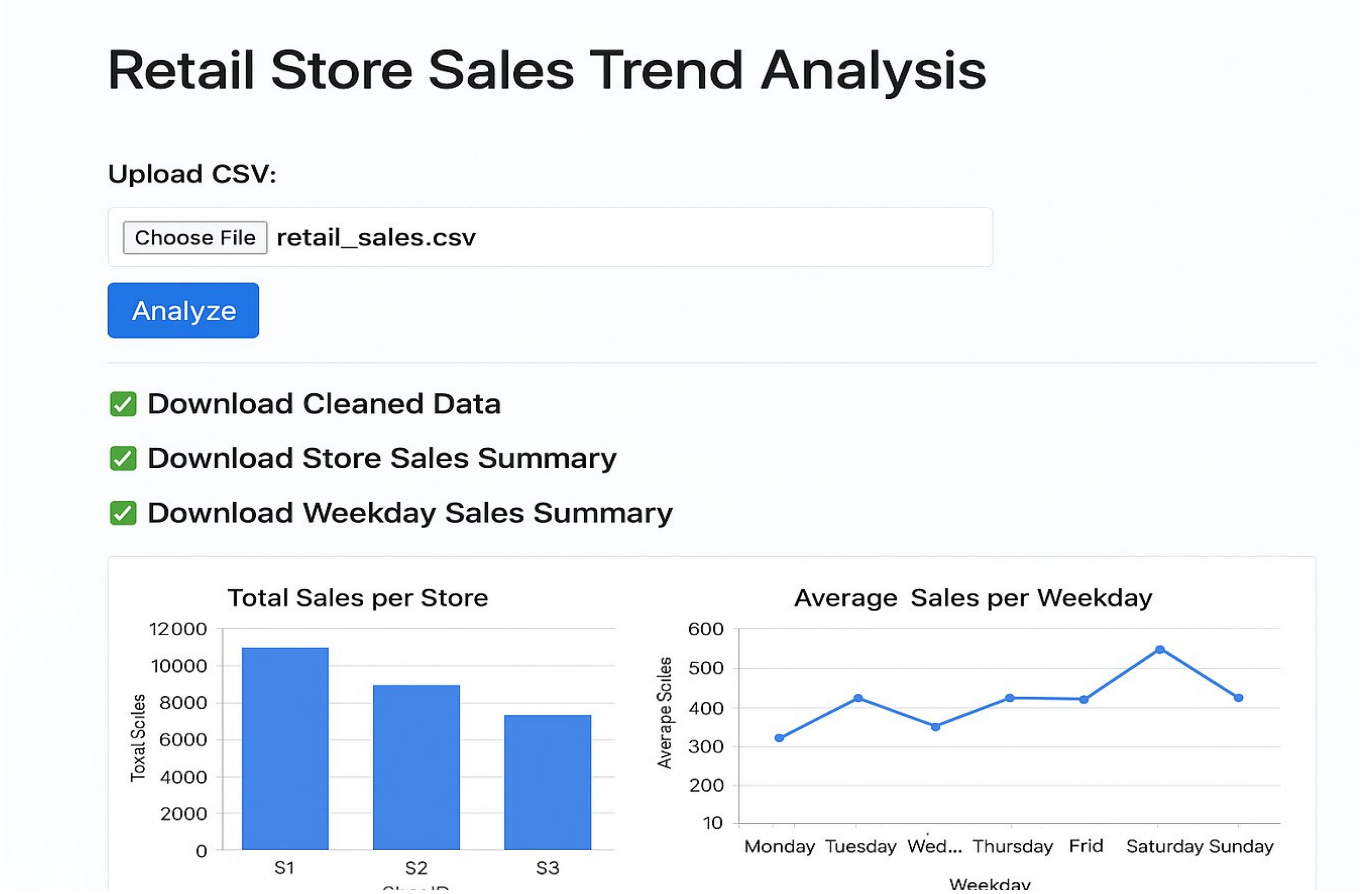
- The **cleaned full dataset** with new columns (cleaned_retail_sales.csv).
- The **store-wise total sales summary** (store_sales_summary.csv).
- The **weekday-wise average sales summary** (weekday_sales_summary.csv).

Step 6: Return file paths

```
return {
    "cleaned": cleaned_path,
    "store_summary": store_sales_path,
    "weekday_summary": weekday_sales_path
}
```

- Returns a dictionary of paths so the Flask app can give these as download links.

Screenshots of output with explanation:



Overall Working of This Output

1. The user uploaded a CSV file.
2. The system **cleaned the data**, generated **summaries**, and **visualized trends**.
3. The dashboard provides:
    - Downloadable processed data files.
    - Key performance charts for **store comparison** and **weekday analysis**.
4. This helps businesses:
    - Identify **top-performing stores**.
    - Plan **promotions for low-sales days**.
    - Optimize **inventory for peak days (weekends)**.

Upload CSV Section

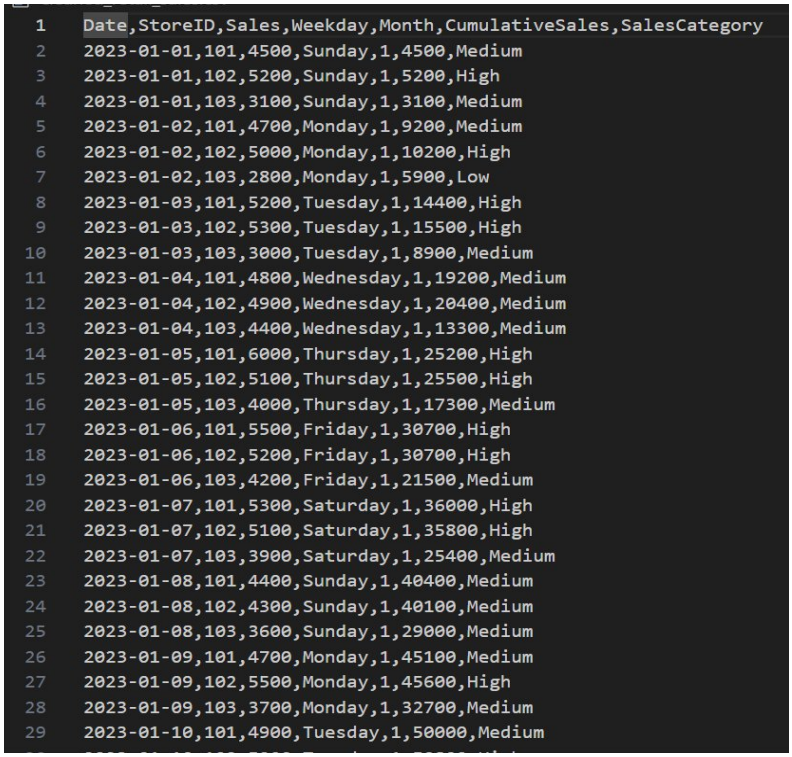
- **Upload CSV: retail_sales.csv**
    - The user has uploaded a CSV file named retail_sales.csv. This file contains raw retail sales data, such as:
        - **Store ID**

        - **Date**
        - **Sales amount**
        - **Other details like product category, quantity, etc.**
- **Analyse Button** - Clicking this button sends the uploaded file to the **Flask backend**, where the data is processed and analyzed using **Pandas**.

3. Download Links (Green Checkmarks ✔)

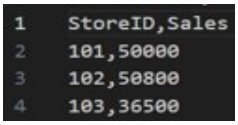
After analysis, three downloadable files are generated:

- **Download Cleaned Data**
    - This file contains the original data after **cleaning** (handling missing values, removing duplicates, formatting dates, etc.).



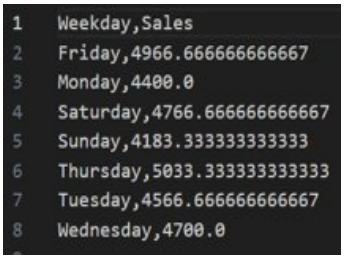
```
Date,StoreID,Sales,Weekday,Month,CumulativeSales,SalesCategory
2023-01-01,101,4500,Sunday,1,4500,Medium
2023-01-01,102,5200,Sunday,1,5200,High
2023-01-01,103,3100,Sunday,1,3100,Medium
2023-01-02,101,4700,Monday,1,9200,Medium
2023-01-02,102,5000,Monday,1,10200,High
2023-01-02,103,2800,Monday,1,5900,Low
2023-01-03,101,5200,Tuesday,1,14400,High
2023-01-03,102,5300,Tuesday,1,15500,High
2023-01-03,103,3000,Tuesday,1,8900,Medium
2023-01-04,101,4800,Wednesday,1,19200,Medium
2023-01-04,102,4900,Wednesday,1,20400,Medium
2023-01-04,103,4400,Wednesday,1,13300,Medium
2023-01-05,101,6000,Thursday,1,25200,High
2023-01-05,102,5100,Thursday,1,25500,High
2023-01-05,103,4000,Thursday,1,17300,Medium
2023-01-06,101,5500,Friday,1,30700,High
2023-01-06,102,5200,Friday,1,30700,High
2023-01-06,103,4200,Friday,1,21500,Medium
2023-01-07,101,5300,Saturday,1,36000,High
2023-01-07,102,5100,Saturday,1,35800,High
2023-01-07,103,3900,Saturday,1,25400,Medium
2023-01-08,101,4400,Sunday,1,40400,Medium
2023-01-08,102,4300,Sunday,1,40100,Medium
2023-01-08,103,3600,Sunday,1,29000,Medium
2023-01-09,101,4700,Monday,1,45100,Medium
2023-01-09,102,5500,Monday,1,45600,High
2023-01-09,103,3700,Monday,1,32700,Medium
2023-01-10,101,4900,Tuesday,1,50000,Medium
```

- **Download Store Sales Summary -** Summarized data showing **total sales per store**.



```
StoreID,Sales
101,50000
102,50800
103,36500
```

- **Download Weekday Sales Summary**
    - Summarized data showing **sales trends across weekdays**.



```
Weekday,Sales
Friday,4966.666666666667
Monday,4400.0
Saturday,4766.666666666667
Sunday,4183.333333333333
Thursday,5033.333333333333
Tuesday,4566.666666666667
Wednesday,4700.0
```

4. Graphs / Charts

The dashboard displays two important visualizations:

a) Total Sales per Store (Bar Chart)

- **Description**:
  - Each bar represents a store (S1, S2, S3).
  - The height of the bar indicates the **total sales amount** for that store.
- **Insights from Chart**:
  - **Store S1** has the highest sales.
  - **Store S3** has the lowest sales.
  - Useful for identifying which store is performing the best.

b) Average Sales per Weekday (Line Chart)

- **Description**:
  - X-axis: Days of the week (Monday to Sunday).
  - Y-axis: Average sales amount.
  - The line shows how sales fluctuate during the week.
- **Insights from Chart**:
  - Sales are **lower at the start of the week** (Monday).
  - **Sales peak on Saturday**, indicating high weekend demand.
  - Slight drop on Sunday, but still higher than weekdays.

Practical Use Cases of Retail Sales Analysis Application

1. Upload Retail Sales Dataset :

A store manager at a retail chain wants to analyse sales performance for the past year. They export sales data from their POS system (as a CSV) and upload it to this application.

2. Data Preprocessing :

The uploaded data might have:

Missing sales values for some dates (e.g., store closed).

Incorrect date formats.

Duplicate rows.

The backend cleans this automatically, just like a data analyst would do before analysis.

3. Trend Analysis & Visualization :

The system generates:

A line chart showing daily sales trends.

A bar chart showing monthly totals.

The manager uses this to see which months performed best and identify seasonal trends (e.g., higher sales during Diwali or Christmas).

4. Forecast Future Sales :

The manager wants to prepare inventory for the next month.

The system predicts that sales will increase by 20% next month (due to festive season).

The manager orders more stock in advance.

5. Generate Downloadable Reports :

After analysis:

The manager downloads a summary CSV and charts.

Shares them in a business meeting with the regional head to explain sales performance and forecast.

6. Interactive Dashboard (Frontend) :

A regional manager wants quick insights without digging into raw data:

They log in, select date range = last 6 months.

See charts for store-wise performance and weekday sales patterns.

Decide which stores need more marketing campaigns.

7. Error Handling & Notifications :

If the manager uploads a wrong file (e.g., customer data instead of sales data),

the system immediately shows:

“Invalid file format. Please upload a CSV with columns: Date, StoreID, Sales.”

This avoids confusion and wasted time.

Bibliography :

1. Hyndman, R. J., & Athanasopoulos, G. (2018). Forecasting: Principles and Practice. OTexts. - https://otexts.com/fpp3/

2. Box, G. E., Jenkins, G. M., & Reinsel, G. C. (2015). Time Series Analysis: Forecasting and Control. Wiley.

3. Pandas Development Team. (2025). Pandas Documentation. https://pandas.pydata.org/docs/

4. Flask Documentation. (2025). Flask Web Framework. https://flask.palletsprojects.com/

5. Plotly Technologies Inc. (2025). Plotly Python Graphing Library. https://plotly.com/python/

6. Facebook Prophet. (2025). Forecasting at Scale. https://facebook.github.io/prophet/

7. Kaggle Datasets. (2025). Retail Sales Data for Time Series Analysis. https://www.kaggle.com/

8. Investopedia. (2025). Seasonality in Retail Sales. https://www.investopedia.com/