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: Hitha Pradeep Y
- USN:4GW23CI017
- Branch: CSE(AI-ML)
- College: GSSS Institute of Engineering And Technology For Women

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 (pd.read_csv).
    - Handles missing values (fillna).
    - Removes duplicates (drop_duplicates).
    - Converts dates to datetime (pd.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 (Sales > 5000, Sales > 4000 on weekends).

- **Aggregations & Grouping**
  - Store-wise total sales.
  - Month-wise average daily sales.
  - Weekday average sales.
  - Top 3 performing stores.
- **Derived Columns**
  - CumulativeSales (running total by store).
  - SalesCategory (High, Medium, Low based on thresholds).
- **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 analyzed 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**.

Procedure:

Step 1 – Start

1. Start the Flask application.
2. Initialize **upload** and **processed** folders.

Step 2 – File Upload

1. User opens the home page (index.html).
2. User selects a CSV file and clicks **Upload & Analyze**.
3. Flask receives the file via /upload route.
4. Save uploaded CSV into the uploads/ folder.

Step 3 – Data Loading

1. Load the CSV file into a **Pandas DataFrame** (pd.read_csv).
2. Display first few rows, column types, and dataset shape (for debugging/logging).

Step 4 – Data Cleaning

1. Fill missing values in **Sales** column with 0.
2. Drop duplicate rows (if any).
3. Convert **Date** column to datetime format.

Step 5 – Time-Series Manipulation

1. Set **Date** column as index.
2. Sort dataset by date.
3. Create new column **Weekday** from Date (e.g., Monday, Tuesday).

4. Create new column **Month** from Date (1–12).

Step 6 – Filtering & Conditional Analysis

1. Extract data for **StoreID = 101**.
2. Identify days where **Sales > 5000**.
3. Identify weekends (**Saturday, Sunday**) where **Sales > 4000**.

Step 7 – Grouping & Aggregations

1. Group data by **StoreID** → calculate **total sales**.
2. Group data by **Month** → calculate **average daily sales**.
3. Group data by **Weekday** → calculate **average sales per day**.
4. Identify **Top 3 stores** by total sales.

Step 8 – Derived Columns

1. Create **CumulativeSales** column using cumulative sum (cumsum) per store.
2. Create **SalesCategory** column:
    - If Sales ≥ 5000 → "High"
    - If 3000 ≤ Sales ≤ 4999 → "Medium"
    - If Sales < 3000 → "Low"

Step 9 – Export Results

1. Save cleaned dataset as **cleaned_retail_sales.csv**.
2. Save store-wise totals as **store_sales_summary.csv**.
3. Save weekday averages as **weekday_sales_summary.csv**.

Step 10 – Dashboard Rendering

1. Pass processed results to dashboard.html using Flask + Jinja2.
2. Display:

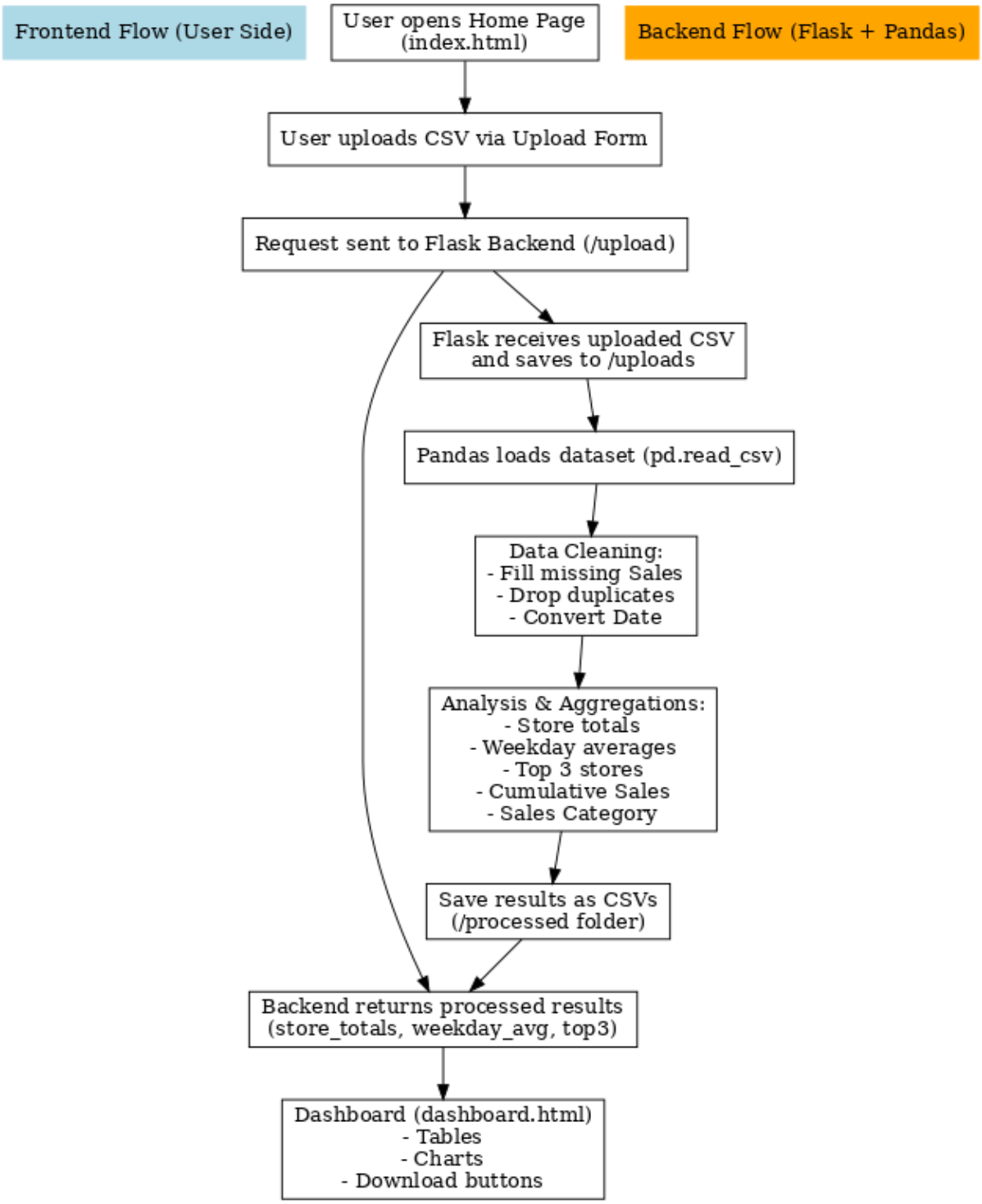
- **Tables** (store totals, weekday averages, top 3 stores).
- **Charts** with Chart.js (Bar, Line, Pie).

3. Provide **download buttons** for processed CSVs.

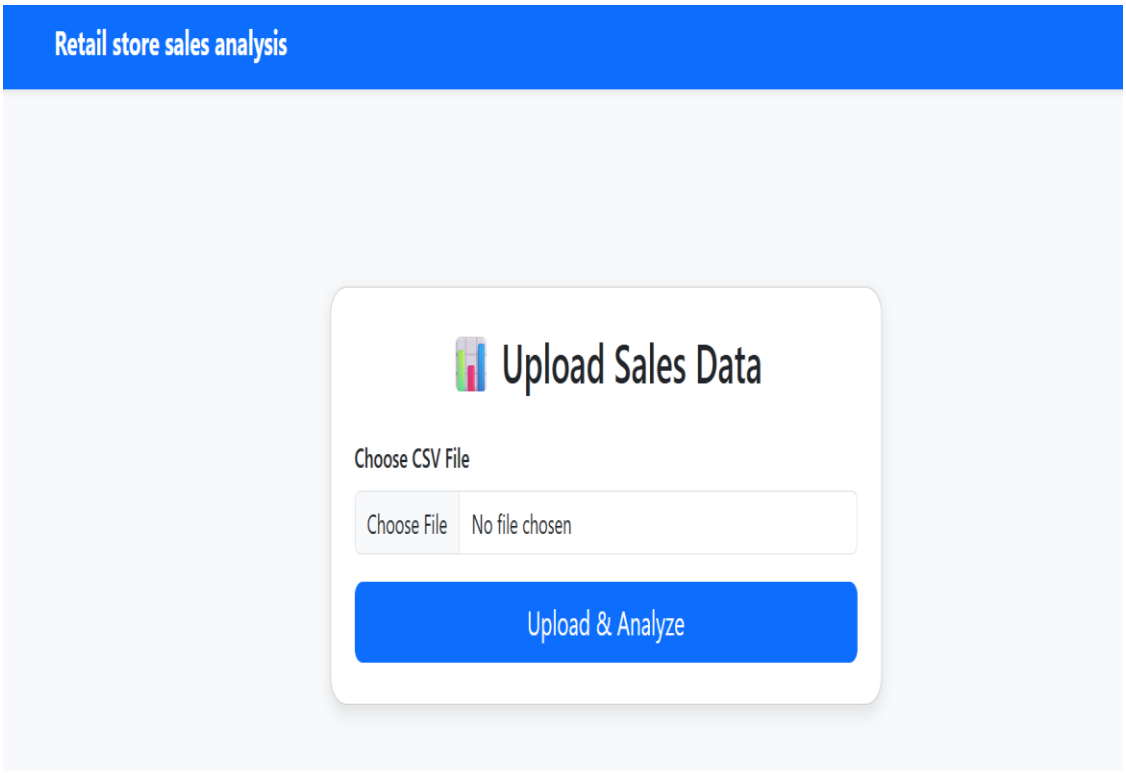
Step 11 – End

1. User views dashboard insights and downloads processed files.
2. End.

Flow-Chart:

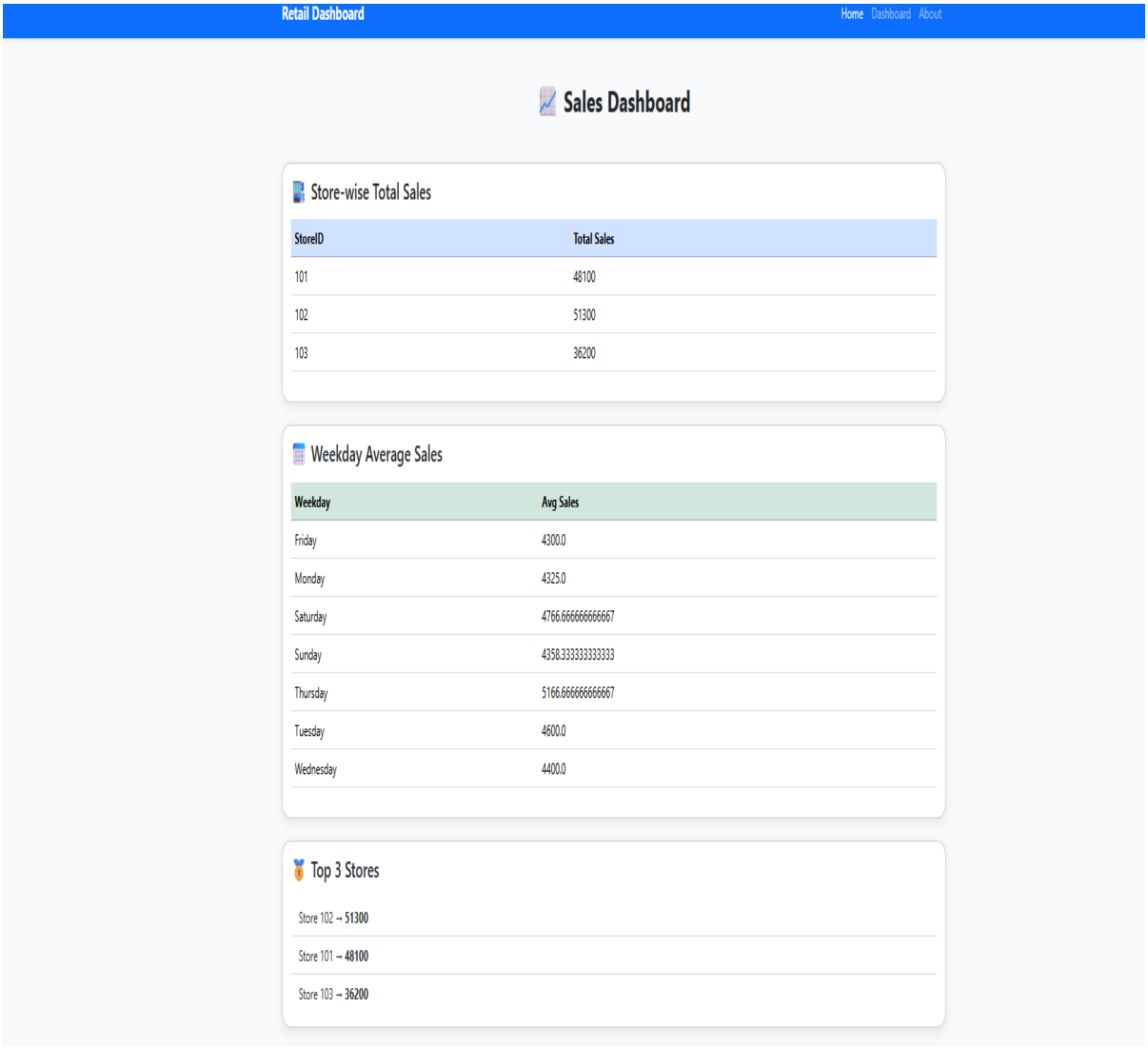

Frontend Flow (User Side)
User opens Home Page (index.html)
Backend Flow (Flask + Pandas)
User uploads CSV via Upload Form
Request sent to Flask Backend (/upload)
Flask receives uploaded CSV and saves to /uploads
Pandas loads dataset (pd.read_csv)
Data Cleaning:
- Fill missing Sales
- Drop duplicates
- Convert Date
Analysis & Aggregations:
- Store totals
- Weekday averages
- Top 3 stores
- Cumulative Sales
- Sales Category
Save results as CSVs (/processed folder)
Backend returns processed results (store_totals, weekday_avg, top3)
Dashboard (dashboard.html)
- Tables
- Charts
- Download buttons

Interface:



Here the user will upload the .csv file of there retail Store Sales and then clicks on Upload & Analyse where they see Retail Dashboard with all the analysis done

Output:

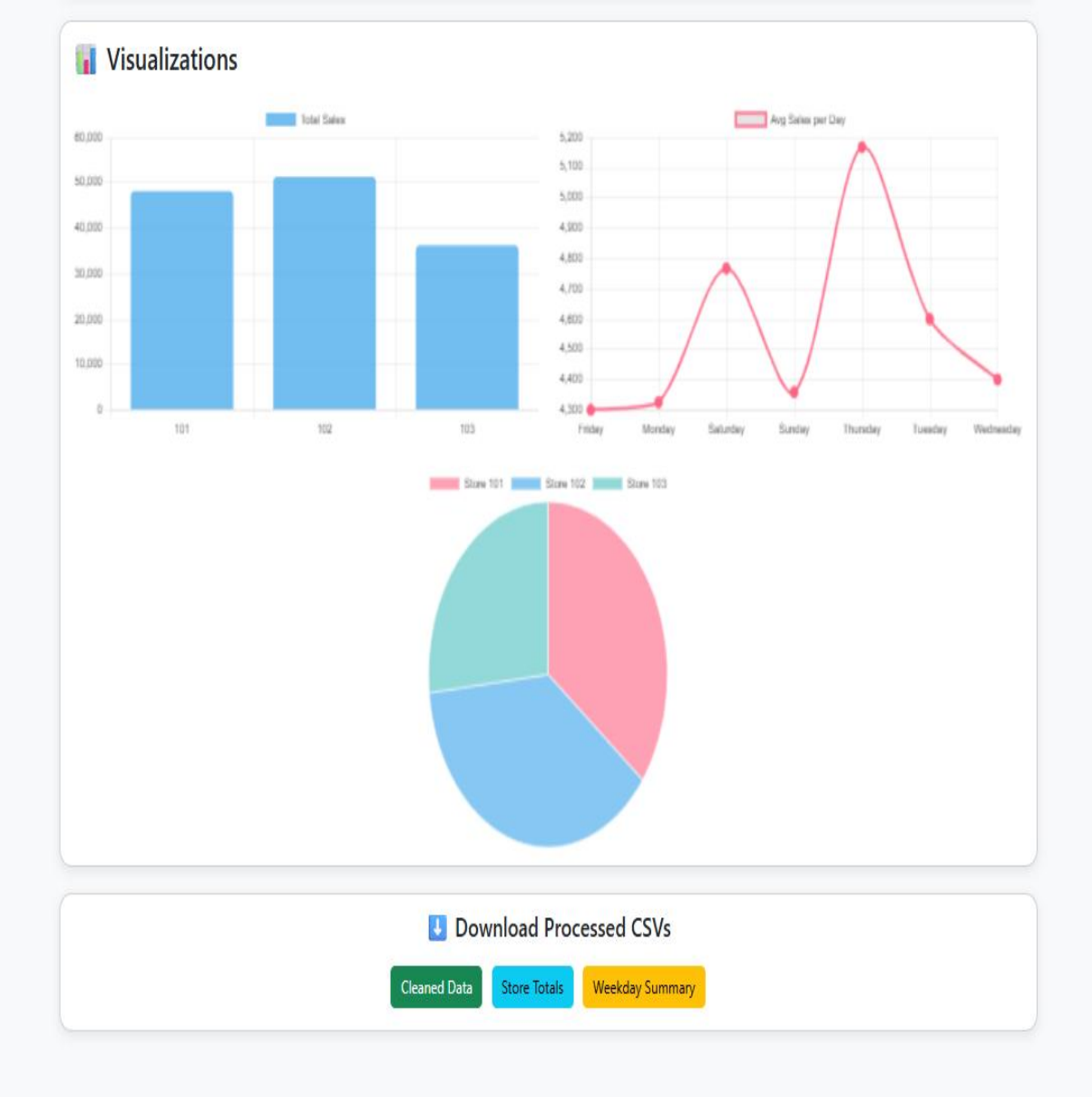


once the .csv file is uploaded the user can seen the analysis of their file which shows:

Total sales per store.

Average sales per weekday.

Top 3 stores by total sales.



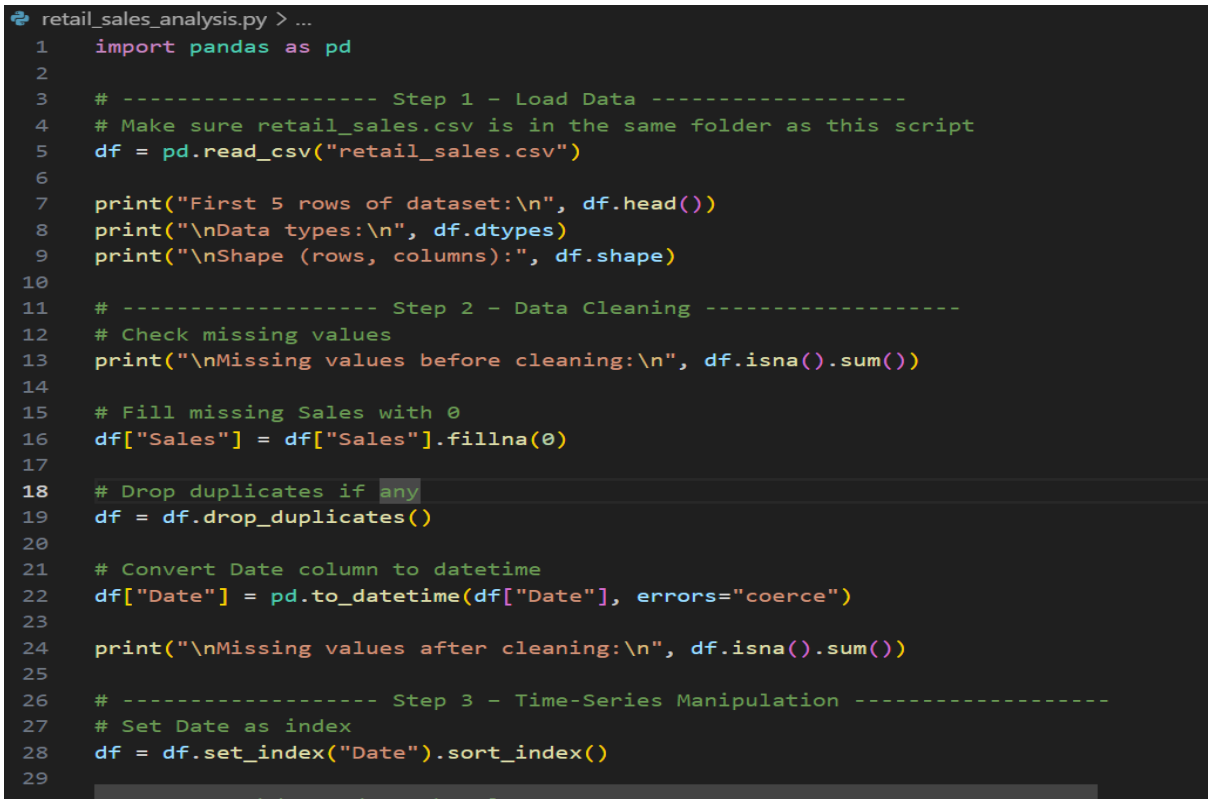
The user can see the visualization so that the analysis will be easy to understand where they can see

- **Bar Chart** → Store-wise total sales.
- **Line Chart** → Average sales by weekday.
- **Pie Chart** → Store contribution to total sales.

and can download the cleaned Data, Store Totals, Weekday Summary

Code Explanation:

Retail_sales_analysis.py



```python
import pandas as pd

# ------------------- Step 1 - Load Data -------------------
# Make sure retail_sales.csv is in the same folder as this script
df = pd.read_csv("retail_sales.csv")

print("First 5 rows of dataset:\n", df.head())
print("\nData types:\n", df.dtypes)
print("\nShape (rows, columns):", df.shape)

# ------------------- Step 2 - Data Cleaning -------------------
# Check missing values
print("\nMissing values before cleaning:\n", df.isna().sum())

# Fill missing Sales with 0
df["Sales"] = df["Sales"].fillna(0)

# Drop duplicates if any
df = df.drop_duplicates()

# Convert Date column to datetime
df["Date"] = pd.to_datetime(df["Date"], errors="coerce")

print("\nMissing values after cleaning:\n", df.isna().sum())

# ------------------- Step 3 - Time-Series Manipulation -------------------
# Set Date as index
df = df.set_index("Date").sort_index()
```

Import pandas

import pandas as pd

- Imports the **pandas library** (commonly aliased as pd).
- Pandas is used for handling tabular data (like Excel sheets or CSVs).

Load Data

df = pd.read_csv("retail_sales.csv")

- Reads the CSV file retail_sales.csv into a **DataFrame** called df.
- df now holds all rows and columns from your dataset.

print("First 5 rows of dataset:\n", df.head())

```
print("\nData types:\n", df.dtypes)
print("\nShape (rows, columns):", df.shape)
```

- df.head() → previews the first **5 rows** (to check if data loaded correctly).
- df.dtypes → shows each column's **data type** (e.g., int64, float64, object).
- df.shape → returns (rows, columns) → helps know dataset size.

Data Cleaning

```
print("\nMissing values before cleaning:\n", df.isna().sum())
```

- df.isna().sum() → counts missing values (**NaN**) in each column before cleaning.

```
df["Sales"] = df["Sales"].fillna(0)
```

- If any value in the **Sales** column is missing, replace it with 0.

```
df = df.drop_duplicates()
```

- Removes duplicate rows (if the same record appears multiple times).

```
df["Date"] = pd.to_datetime(df["Date"], errors="coerce")
```

- Converts the **Date** column into **datetime objects**.
- If a value isn't a valid date, errors="coerce" replaces it with NaT (*Not a Time*).

```
print("\nMissing values after cleaning:\n", df.isna().sum())
```

- Re-checks missing values after cleaning, to confirm fixes.

Time-Series Manipulation

```
df = df.set_index("Date").sort_index()
```

- Makes the **Date column the index** of the DataFrame.
- Sorting ensures rows are arranged in chronological order.
- This step is crucial for **time-series analysis** (like trends, monthly averages, etc.).

This section mainly contain

1. **Load data** → pd.read_csv()
2. **Inspect** → preview, check dtypes, and shape
3. **Clean** → fix missing values, remove duplicates, parse dates

4. **Prepare for analysis** → set Date as index and sort

```python
# Create Weekday and Month columns
df["Weekday"] = df.index.day_name()
df["Month"] = df.index.month

# ------------------- Step 4 - Filtering & Conditional Operations -------------------
# Filter StoreID = 101
store_101 = df[df["StoreID"] == 101]
print("\nSales for StoreID 101:\n", store_101.head())

# Sales > 5000
sales_gt_5000 = df[df["Sales"] > 5000]
print("\nDays with Sales > 5000:\n", sales_gt_5000.head())

# Weekends with Sales > 4000
weekends_high = df[df.index.dayofweek.isin([5, 6]) & (df["Sales"] > 4000)]
print("\nWeekends with Sales > 4000:\n", weekends_high.head())

# ------------------- Step 5 - Grouping & Aggregations -------------------
# Group by StoreID → Total sales
store_totals = df.groupby("StoreID", as_index=False)["Sales"].sum().rename(columns={"Sales": "TotalSales"})
print("\nTotal sales by Store:\n", store_totals)

# Group by Month → Average daily sales
month_avg = df.groupby("Month", as_index=False)["Sales"].mean().rename(columns={"Sales": "AvgDailySales"})
print("\nAverage daily sales by Month:\n", month_avg)

# Group by Weekday → Average sales
weekday_avg = df.groupby("Weekday", as_index=False)["Sales"].mean().rename(columns={"Sales": "AvgSalesPerDay"})
print("\nAverage sales by Weekday:\n", weekday_avg)
```

Filtering & Conditional Operations

1. Add Weekday and Month columns

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

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

- df.index.day_name() → extracts the day of the week (e.g., Monday, Tuesday) from the **Date index**.
- df.index.month → extracts the numeric month (1 = January, 12 = December).
- Both become **new columns** in the DataFrame.

---

2. Filter data for StoreID = 101

store_101 = df[df["StoreID"] == 101]

print("\nSales for StoreID 101:\n", store_101.head())

- Keeps only the rows where StoreID == 101.
- store_101.head() → shows the first 5 rows for that store.

---

3. Filter data where Sales > 5000

sales_gt_5000 = df[df["Sales"] > 5000]

```
print("\nDays with Sales > 5000:\n", sales_gt_5000.head())
```

- Selects rows where Sales is greater than 5000.
- This is useful to identify high-performing days.

---

4. Filter Weekends with Sales > 4000

```
weekends_high = df[df.index.dayofweek.isin([5, 6]) & (df["Sales"] > 4000)]

print("\nWeekends with Sales > 4000:\n", weekends_high.head())
```

- df.index.dayofweek → gives a number for each day (0 = Monday, ..., 6 = Sunday).
- .isin([5, 6]) → selects **Saturday (5) and Sunday (6)**.
- Combined with (df["Sales"] > 4000) using &.
- Result: rows where **it's the weekend AND sales > 4000**.

Grouping & Aggregations

1. Group by StoreID → Total Sales

```
store_totals = df.groupby("StoreID",
as_index=False)["Sales"].sum().rename(columns={"Sales": "TotalSales"})

print("\nTotal sales by Store:\n", store_totals)
```

- Groups rows by **StoreID**.
- Sums the Sales values for each store.
- Renames the column to TotalSales.
- Result: each store and its **total sales**.

---

2. Group by Month → Average Daily Sales

```
month_avg = df.groupby("Month",
as_index=False)["Sales"].mean().rename(columns={"Sales": "AvgDailySales"})

print("\nAverage daily sales by Month:\n", month_avg)
```

- Groups rows by **Month**.
- Calculates the **mean (average)** sales per day for each month.
- Renames the column to AvgDailySales.

- Helps track **monthly sales trends**.

---

3. Group by Weekday → Average Sales

```
weekday_avg = df.groupby("Weekday",
as_index=False)["Sales"].mean().rename(columns={"Sales": "AvgSalesPerDay"})

print("\nAverage sales by Weekday:\n", weekday_avg)
```

- Groups rows by **Weekday**.
- Finds the **average sales** for each day of the week.
- Renames the column to AvgSalesPerDay.
- Useful to see if weekends or weekdays perform better.

This section does two main things:

1. **Filtering**: Extracts specific subsets of data (like sales above a threshold, weekend performance, or specific stores).
2. **Grouping & Aggregation**: Summarizes data by categories (StoreID, Month, Weekday) to get totals and averages.

```
59
60  # Top 3 stores by sales
61  top3_stores = store_totals.sort_values("TotalSales", ascending=False).head(3)
62  print("\nTop 3 Stores by Total Sales:\n", top3_stores)
63
64  # ------------------ Step 6 - Derived Columns ------------------
65  # Cumulative sales per store
66  df["CumulativeSales"] = df.groupby("StoreID")["Sales"].cumsum()
67
68  # Sales category
69  def categorize_sales(s):
70      if s >= 5000:
71          return "High"
72      elif 3000 <= s <= 4999:
73          return "Medium"
74      else:
75          return "Low"
76
77  df["SalesCategory"] = df["Sales"].apply(categorize_sales)
78
79  print("\nData with Derived Columns:\n", df.head())
80
81  # ------------------ Step 7 - Export Results ------------------
82  df.reset_index().to_csv("cleaned_retail_sales.csv", index=False)
83  store_totals.to_csv("store_sales_summary.csv", index=False)
84  weekday_avg.to_csv("weekday_sales_summary.csv", index=False)
85
86  print("\n✅ Exported cleaned_retail_sales.csv, store_sales_summary.csv, weekday_sales_summary.csv")
```

Top 3 stores by sales

top3_stores = store_totals.sort_values("TotalSales", ascending=False).head(3)

print("\nTop 3 Stores by Total Sales:\n", top3_stores)

- store_totals (from earlier) has each store's total sales.
- sort_values("TotalSales", ascending=False) → sorts stores from **highest to lowest sales**.
- .head(3) → takes the **top 3 stores**.
- print → displays the result.

Derived Columns

Cumulative Sales per Store

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

- Groups data by **StoreID**.
- cumsum() → calculates the **cumulative sum** of sales for each store.
- This creates a running total of sales over time.
- Stored in a new column **CumulativeSales**.

Sales Category

```
def categorize_sales(s):
  if s >= 5000:
    return "High"
  elif 3000 <= s <= 4999:
    return "Medium"
  else:
    return "Low"
```

- Defines a function categorize_sales that classifies each sales value:
  - ≥ 5000 → **High**
  - 3000–4999 → **Medium**
  - < 3000 → **Low**

```
df["SalesCategory"] = df["Sales"].apply(categorize_sales)
```

- Applies the function to the **Sales column**.
- Creates a new column **SalesCategory** with labels "High", "Medium", or "Low".

```
print("\nData with Derived Columns:\n", df.head())
```

- Prints the first few rows of the updated DataFrame (to check the new columns).

🔹 **Export Results**

```
df.reset_index().to_csv("cleaned_retail_sales.csv", index=False)
store_totals.to_csv("store_sales_summary.csv", index=False)
weekday_avg.to_csv("weekday_sales_summary.csv", index=False)
```

- df.reset_index() → moves the **Date index** back into a normal column.
- Saves:
  - cleaned_retail_sales.csv → full cleaned dataset (with derived columns).
  - store_sales_summary.csv → total sales per store.
  - weekday_sales_summary.csv → average sales per weekday.

print("\n Exported cleaned_retail_sales.csv, store_sales_summary.csv, weekday_sales_summary.csv")

- Prints confirmation message with ✅ to indicate success.

Summary of this section

- **Finds top 3 stores** by total sales.
- Adds **Cumulative Sales** and **Sales Category** as new columns.
- Exports processed datasets to CSV files.

app.py:

What the app.py does :

1. Starts a Flask server.
2. Ensures two folders exist: uploads/ (raw files) and processed/ (outputs).
3. Serves a home page (index.html) with an upload form.
4. Accepts a CSV upload at /upload, cleans and analyzes it with pandas:
    - fills missing sales, drops duplicates
    - parses dates, adds Weekday & Month
    - computes CumulativeSales per store & a SalesCategory
    - builds summaries: totals per store, average per weekday, top 3 stores
5. Saves 3 CSVs under processed/.
6. Renders dashboard.html with tables.
7. Serves processed files back via /download/<filename>.

app.py > ...

```python
from flask import Flask, render_template, request, redirect, url_for, send_file, jsonify
import pandas as pd
import os

app = Flask(__name__)
UPLOAD_FOLDER = "uploads"
PROCESSED_FOLDER = "processed"

os.makedirs(UPLOAD_FOLDER, exist_ok=True)
os.makedirs(PROCESSED_FOLDER, exist_ok=True)

# ---------- Home Page ----------
@app.route('/')
def index():
    return render_template('index.html')

# ---------- Handle File Upload ----------
@app.route('/upload', methods=['POST'])
def upload_file():
    file = request.files['file']
    if file.filename == '':
        return "No file selected"
    filepath = os.path.join(UPLOAD_FOLDER, file.filename)
    file.save(filepath)

    # Process file
    df = pd.read_csv(filepath)
```

Imports

```
from flask import Flask, render_template, request, redirect, url_for, send_file, jsonify
import pandas as pd
import os
```

- Flask → main class to create the web application.
- render_template → loads HTML files from the templates/ folder.
- request → handles form data and file uploads from the client.
- redirect, url_for → used for navigation (though not used yet in this part).
- send_file → sends files back to the client (for downloads).
- jsonify → converts Python dicts to JSON (used for API-like responses).
- pandas as pd → used for CSV processing and data cleaning.
- os → used for file and folder handling.

🔹 **Flask app initialization**

```
app = Flask(__name__)
UPLOAD_FOLDER = "uploads"
PROCESSED_FOLDER = "processed"
```

- Creates the Flask app instance.
- Defines two folders:
  - uploads/ → where uploaded CSV files will be stored.
  - processed/ → where cleaned/processed results will be saved.

🔹 **Ensure folders exist**

```
os.makedirs(UPLOAD_FOLDER, exist_ok=True)
os.makedirs(PROCESSED_FOLDER, exist_ok=True)
```

- os.makedirs(path, exist_ok=True) → creates the folder if it doesn't already exist.
- Prevents errors if you run the app for the first time.

🔹 **Home page route**

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

- Defines the **home page (/)** route.
- When a user visits the root URL, Flask will render and return templates/index.html.
- This HTML file likely contains a **file upload form**.

🔹 **Handle file uploads**

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

- Defines the /upload route, which accepts only **POST requests** (for form submissions).
- request.files['file'] → gets the uploaded file from the form (where the input field must have name="file").

```
  if file.filename == '':
    return "No file selected"
```

- Checks if the user submitted the form without selecting a file.

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

- Builds a path under the **uploads folder**.
- Saves the uploaded file there.

🔹 **Process the file**

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

- Reads the uploaded CSV file into a pandas **DataFrame**.
- From here, later parts of the script will **clean, analyze, and generate outputs**.

Summary of this section

1. Import Flask, pandas, and OS utilities.
2. Create Flask app instance.
3. Prepare folders for uploads and processed results.
4. Define a **home page** that renders index.html.
5. Define an **upload route** to accept CSV files and load them into pandas for processing.

```
    # Cleaning
    df["Sales"] = df["Sales"].fillna(0)
    df = df.drop_duplicates()
    df["Date"] = pd.to_datetime(df["Date"], errors="coerce")
    df = df.set_index("Date").sort_index()
    df["Weekday"] = df.index.day_name()
    df["Month"] = df.index.month

    # Derived columns
    df["CumulativeSales"] = df.groupby("StoreID")["Sales"].cumsum()
    df["SalesCategory"] = df["Sales"].apply(
        lambda s: "High" if s >= 5000 else "Medium" if s >= 3000 else "Low"
    )

    # Summaries
    store_totals = df.groupby("StoreID")["Sales"].sum().reset_index().rename(columns={"Sales": "TotalSales"})
    weekday_avg = df.groupby("Weekday")["Sales"].mean().reset_index().rename(columns={"Sales": "AvgSalesPerDay"})
    top3_stores = store_totals.sort_values("TotalSales", ascending=False).head(3)

    # Save results
    cleaned_path = os.path.join(PROCESSED_FOLDER, "cleaned_retail_sales.csv")
    store_summary_path = os.path.join(PROCESSED_FOLDER, "store_sales_summary.csv")
    weekday_summary_path = os.path.join(PROCESSED_FOLDER, "weekday_sales_summary.csv")
```

Cleaning

df["Sales"] = df["Sales"].fillna(0)

df = df.drop_duplicates()

df["Date"] = pd.to_datetime(df["Date"], errors="coerce")

df = df.set_index("Date").sort_index()

```
df["Weekday"] = df.index.day_name()
df["Month"] = df.index.month
```

- **Fill missing sales** → replaces NaN in Sales with 0.
- **Drop duplicates** → removes repeated rows.
- **Convert Date column** → ensures values are proper datetime objects; invalid ones become NaT.
- **Set Date as index** → makes Date the DataFrame index and sorts chronologically.
- **Add Weekday column** → extracts day name (e.g., Monday, Friday).
- **Add Month column** → extracts numeric month (1–12).

After this step, your dataset is **clean, time-indexed, and enriched with date-related columns**.

🔹 Derived Columns

```
df["CumulativeSales"] = df.groupby("StoreID")["Sales"].cumsum()
df["SalesCategory"] = df["Sales"].apply(
  lambda s: "High" if s >= 5000 else "Medium" if s >= 3000 else "Low"
)
```

- **CumulativeSales** → running total of sales per store (groupby("StoreID")).
- **SalesCategory** → classifies each row into:
  - "High" if Sales ≥ 5000
  - "Medium" if 3000 ≤ Sales < 5000
  - "Low" if Sales < 3000

These help you **analyze performance trends** and **segment sales levels**.

🔹 Summaries

```
store_totals = df.groupby("StoreID")["Sales"].sum().reset_index().rename(columns={"Sales": "TotalSales"})
```

```
weekday_avg = df.groupby("Weekday")["Sales"].mean().reset_index().rename(columns={"Sales": "AvgSalesPerDay"})

top3_stores = store_totals.sort_values("TotalSales", ascending=False).head(3)
```

- **Total sales per store** → sums sales grouped by StoreID.
- **Average sales per weekday** → calculates mean sales grouped by day of the week.
- **Top 3 stores** → sorts stores by TotalSales (highest first) and picks top 3.

These summaries are used later in your dashboard to display **insights**.

🔹 **Save Results**

```
cleaned_path = os.path.join(PROCESSED_FOLDER, "cleaned_retail_sales.csv")

store_summary_path = os.path.join(PROCESSED_FOLDER, "store_sales_summary.csv")

weekday_summary_path = os.path.join(PROCESSED_FOLDER, "weekday_sales_summary.csv")
```

- Prepares file paths under the **processed folder** for saving:
    - cleaned_retail_sales.csv → full cleaned dataset (with derived columns).
    - store_sales_summary.csv → total sales per store.
    - weekday_sales_summary.csv → average sales by weekday.

Later in the script, these DataFrames are exported to CSV so users can **download the results**.

Summary of this section:

1. **Cleans data** → fixes missing values, removes duplicates, ensures proper dates.
2. **Adds derived columns** → cumulative sales & sales category.
3. **Generates summaries** → per store, per weekday, and top 3 stores.
4. **Prepares export paths** → for saving processed results.

```
52
53      df.reset_index().to_csv(cleaned_path, index=False)
54      store_totals.to_csv(store_summary_path, index=False)
55      weekday_avg.to_csv(weekday_summary_path, index=False)
56
57      # Render dashboard with data
58      return render_template(
59          'dashboard.html',
60          tables={
61              "store_totals": store_totals.to_dict(orient="records"),
62              "weekday_avg": weekday_avg.to_dict(orient="records"),
63              "top3": top3_stores.to_dict(orient="records")
64          }
65      )
66
67  # ---------- Download Processed Files ----------
68  @app.route('/download/<filename>')
69  def download_file(filename):
70      filepath = os.path.join(PROCESSED_FOLDER, filename)
71      return send_file(filepath, as_attachment=True)
72
73  if __name__ == '__main__':
74      app.run(debug=True)
75
```

Save Processed Data

df.reset_index().to_csv(cleaned_path, index=False)

store_totals.to_csv(store_summary_path, index=False)

weekday_avg.to_csv(weekday_summary_path, index=False)

- **df.reset_index()** → moves the Date index back into a regular column.
- Saves three CSVs into the **processed folder**:
    - cleaned_retail_sales.csv → cleaned dataset with derived columns.
    - store_sales_summary.csv → total sales per store.
    - weekday_sales_summary.csv → average sales by weekday.

These files can later be downloaded by the user.

🔹 **Render Dashboard with Data**

return render_template(

```
    'dashboard.html',
    tables={
        "store_totals": store_totals.to_dict(orient="records"),
        "weekday_avg": weekday_avg.to_dict(orient="records"),
        "top3": top3_stores.to_dict(orient="records")
    }
)
```

- Renders dashboard.html (a template inside the templates/ folder).
- Passes three datasets as **dictionaries**:
    - store_totals → total sales per store.
    - weekday_avg → average sales by weekday.
    - top3 → top 3 stores by total sales.
- .to_dict(orient="records") → converts each DataFrame into a list of dictionaries ([{"col1":val1,"col2":val2}, ...]).
- This format is easy to loop through in Jinja2 inside the HTML file.

This step is what **connects your Python data processing with your HTML dashboard**.

🔹 **Download Processed Files**

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

- Defines a new route /download/<filename>.
- filename is dynamic → e.g., /download/cleaned_retail_sales.csv.
- Joins the filename with the processed folder.
- Uses send_file to send the CSV back to the browser as a **download**.

This gives the user a way to **download the cleaned dataset or summaries**.

◆ **Run Flask App**

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

- Runs the Flask app only if this script is executed directly (not imported).
- debug=True → enables:
  - **Auto-reload** on code changes.
  - **Debugging information** in case of errors.
  - (Should be False in production).

Summary of this part of script:

1. **Saves processed CSVs** to the processed/ folder.
2. **Passes analysis results to the dashboard** (dashboard.html).
3. **Provides a download route** so users can get the processed files.
4. **Runs the Flask server** with debugging enabled.

Index.html

```
<form action="/upload" method="post" enctype="multipart/form-data">
 <div class="mb-3">
  <label class="form-label fw-semibold">Choose CSV File</label>
  <input type="file" name="file" class="form-control" required>
 </div>
 <div class="d-grid">
  <button type="submit" class="btn btn-primary btn-lg">Upload & Analyze</button>
 </div>
</form>
```

<form ...>

- action="/upload" → sends the file to your Flask /upload route (where you process it with pandas).

- method="post" → uses POST request (required for file uploads).
- enctype="multipart/form-data" → ensures the file is actually sent, not just the filename.

File input

<input type="file" name="file" class="form-control" required>

- type="file" → lets the user pick a file from their computer.
- name="file" → very important! It matches request.files['file'] in your Flask code.
- required → prevents empty submissions.

Submit button

<button type="submit" class="btn btn-primary btn-lg">Upload & Analyze</button>

- Submits the form and triggers Flask's /upload route.
- Styled with Bootstrap (btn btn-primary).

dashboard.html:

```
{% for row in tables.store_totals %}
 <tr><td>{{ row.StoreID }}</td><td>{{ row.TotalSales }}</td></tr>
{% endfor %}
```

{% for row in tables.store_totals %} ... {% endfor %}

- This is a **Jinja2 loop**.
- It takes the store_totals list (passed from Flask in app.py) and iterates through each row.

{{ row.StoreID }} and {{ row.TotalSales }}

- These insert actual values into the HTML table cells.
- For each store, it shows the **Store ID** and its **total sales**.

Practical Use Case of Retail Sales Analysis Application

1. Upload Retail Sales Dataset

A store manager at a retail chain wants to analyze 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.

Reference:

[1] M. Grinberg, *Flask Web Development: Developing Web Applications with Python*, 2nd ed. Sebastopol, CA, USA: O'Reilly Media, 2018.

[2] Flask Documentation, "Welcome to Flask Documentation." [Online]. Available: https://flask.palletsprojects.com/. [Accessed: Sept. 2, 2025].

[3] W. McKinney, *Python for Data Analysis: Data Wrangling with pandas, NumPy, and IPython*, 2nd ed. Sebastopol, CA, USA: O'Reilly Media, 2017.

[4] pandas Development Team, "pandas Documentation." [Online]. Available: https://pandas.pydata.org/docs/. [Accessed: Sept. 2, 2025].

[5] The Jinja Team, "Jinja2 Documentation." [Online]. Available: https://jinja.palletsprojects.com/. [Accessed: Sept. 2, 2025].

[6] Python Software Foundation, "Python 3 Documentation." [Online]. Available: https://docs.python.org/3/. [Accessed: Sept. 2, 2025].

[7] Retail Dataset, *retail_sales.csv* [CSV dataset], 2025. Provided as project data source.