

# Sales Analysis Application

K. Mohamed Ilham

[Ilhamsk360@gmail.com](mailto:Ilhamsk360@gmail.com)

This report presents the design, development, and implementation of a sales data analysis application developed in Python for Sampath Food City (PVT) Ltd. The goal of the application is to streamline the data analysis process that was previously done manually, by providing automated, real-time insights into key areas of sales performance. The application makes use of SOLID principles, clean coding techniques, and design patterns to ensure that the system is maintainable, scalable, and easy to extend.

The objective of the sales data analysis application is to automate and improve the analysis of sales data for Sampath Food City (PVT) Ltd. It focuses on four key areas: identifying best-selling products, tracking product performance, analyzing customer behavior, and evaluating regional sales.

## Used Dataset

I have used a secondary sales dataset for this sales analysis application. This dataset includes 8,500 rows and 9 columns, including Sales information such as date, customer ID, location, purchase category, etc.

	A	B	C	D	E	F	G	H	I	
1	Date	Customer_ID	Gender	Location	Education Level	Occupation	Age Group	Marital Status	Purchase Category	
2	2020/01/01	Cus1	Female	Jaffna	Undergraduate	Registered Nurse	18-25	Married	Health & Wellness	
3	01/01/2020	Cus2	Male	Monaragala	High School	Call Center Agent	18-25	Single	Tech & Gadgets	
4	01/01/2020	Cus3	Male	Colombo	Undergraduate	Project Coordinator	18-25	Divorced	Crafting & DIY	
5	01/01/2020	Cus4	Male	Vavuniya	Undergraduate	Other	18-25	Married	Miscellaneous	
6	01/01/2020	Cus4	Male	Vavuniya	Undergraduate	Other	18-25	Married	Tech & Gadgets	

Figure 34 dataset overview

## Key Technologies Used

In this Sales Data Analysis Application, I have utilized the following key technologies:

- **Python:** The programming language for the backend logic.
- **Dash:** Framework for creating web applications.
- **Plotly:** A graphing library for rendering interactive visualizations.
- **Pandas:** For data manipulation and analysis.

## Application Source Code

The application's source code is structured into modules, focusing on data ingestion, processing, and visualization. The code follows industry-standard practices such as modularization, abstraction, and the use of design patterns like the Strategy, Template Method, and Factory patterns.

```
Sales_Analysis_App.py > ...
1  # Import required libraries
2  import pandas as pd
3  from abc import ABC, abstractmethod
4  import dash
5  from dash import html, dcc
6  from dash.dependencies import Input, Output
7  import plotly.express as px
8  from itertools import combinations
9  from collections import Counter
10
```

Figure 35 import required libraries

- **Data Ingestion:**

```
11 # Define the abstract interface for data ingestion
12 class DataIngestionStrategy(ABC):
13     @abstractmethod
14     def ingest_data(self, file_path):
15         pass
16
17 # Define the concrete implementation for Excel data ingestion
18 class ExcelDataIngestion(DataIngestionStrategy):
19     def ingest_data(self, file_path):
20         try:
21             data = pd.read_excel(file_path) # Read Excel file
22             # Strip whitespace from column names to avoid issues during processing
23             data.columns = data.columns.str.strip()
24             return data
25         except Exception as e:
26             print(f"Error loading Excel file: {e}") # Log errors for debugging
27             return pd.DataFrame() # Return an empty DataFrame if file cannot be loaded
28
29 # Define a context class to dynamically switch data ingestion strategies
30 class DataIngestionContext:
31     def __init__(self, strategy: DataIngestionStrategy):
32         self._strategy = strategy
33
34     def set_strategy(self, strategy: DataIngestionStrategy):
35         self._strategy = strategy
36
37     def ingest(self, file_path):
38         return self._strategy.ingest_data(file_path)
39
```

Figure 36 data Ingestion Code

The Data Ingestion module is responsible for loading and managing the input data from external sources, ensuring that it is available for further processing. It uses the Strategy Design Pattern to

allow for flexibility in choosing the data ingestion approach. The `DataIngestionStrategy` class defines the interface for different ingestion strategies, while the `ExcelDataIngestion` class implements the ingestion from an Excel file. A `DataIngestionContext` class is used to dynamically select the appropriate strategy. This modular approach ensures that the application can easily switch between different data sources in the future, such as CSV files or databases, without changing the core logic of data processing.

- **Data Processing and visualization:**

```
39
40 # Define an abstract interface for data processing
41 class DataProcess(ABC):
42     @abstractmethod
43     def process_data(self, data):
44         pass
45
46 # Define a class for analyzing trends over time
47 class TrendsOverTime(DataProcess):
48     def process_data(self, data):
49         # Validate required columns
50         if 'Date' not in data.columns or 'Customer_ID' not in data.columns:
51             print("Warning: Required columns for Trends Over Time are missing.")
52             return pd.DataFrame(columns=['Date', 'TotalTransactions']) # Return an empty DataFrame if columns are missing
53         data['Date'] = pd.to_datetime(data['Date'], errors='coerce') # Convert Date column to datetime
54         grouped = data.groupby(data['Date'].dt.to_period("M"))['Customer_ID'].count().reset_index() # Group by month
55         grouped.columns = ['Date', 'TotalTransactions'] # Rename columns
56         grouped['Date'] = grouped['Date'].dt.to_timestamp() # Convert period to timestamp
57         return grouped
58
59 # Define a class for analyzing sales distribution by location
60 class LocationDistribution(DataProcess):
61     def process_data(self, data):
62         # Validate required columns
63         if 'Location' not in data.columns or 'Customer_ID' not in data.columns:
64             print("Warning: Required columns for Location Distribution are missing.")
65             return pd.DataFrame(columns=['Location', 'CustomerCount']) # Return an empty DataFrame if columns are missing
66         return data.groupby
67         ('Location')['Customer_ID'].count().reset_index(name='CustomerCount').sort_values(by='CustomerCount', ascending=False)
68
```

```
69 # Define a class for analyzing frequently purchased product pairs
70 class PairProductAnalysis(DataProcess):
71     def process_data(self, data):
72         # Validate required columns
73         if 'Customer_ID' not in data.columns or 'Purchase Category' not in data.columns:
74             print("Warning: Required columns for Pair Product Analysis are missing.")
75             return pd.DataFrame(columns=['Product 1', 'Product 2', 'Frequency']) # Return an empty DataFrame if columns are missing
76         grouped_data = data.groupby('Customer_ID')['Purchase Category'].apply(list) # Group product categories by customer
77         pairs = []
78         for items in grouped_data:
79             pairs.extend(combinations(items, 2)) # Generate all possible pairs of products
80         pair_counts = Counter(pairs) # Count the occurrences of each pair
81         pair_df = pd.DataFrame(pair_counts.items(), columns=['Pair', 'Frequency']) # Convert counts to DataFrame
82         pair_df['Product 1'], pair_df['Product 2'] = zip(*pair_df['Pair']) # Split pairs into separate columns
83         return pair_df.drop(columns=['Pair']).sort_values(by='Frequency', ascending=False)
84
85 # Define a class for analyzing best-selling products
86 class BestSellingProductAnalysis(DataProcess):
87     def process_data(self, data):
88         # Validate required columns
89         if 'Purchase Category' not in data.columns or 'Customer_ID' not in data.columns:
90             print("Warning: Required columns for Best Selling Product Analysis are missing.")
91             return pd.DataFrame(columns=['Purchase Category', 'Frequency']) # Return an empty DataFrame if columns are missing
92         top_categories = data.groupby('Purchase Category')['Customer_ID'].count().reset_index(name='Frequency') # Count transaction
93         return top_categories.sort_values(by='Frequency', ascending=False).head(10) # Return the top 10 categories
94
```



```

95 # Define a class for analyzing product performance over time
96 class ProductPerformanceAnalysis(DataProcess):
97     def process_data(self, data):
98         # Validate required columns
99         if 'Date' not in data.columns or 'Purchase Category' not in data.columns or 'Customer_ID' not in data.columns:
100             print("Warning: Required columns for Product Performance Analysis are missing.")
101             return pd.DataFrame(columns=['Month', 'Product Category', 'Sales']) # Return an empty DataFrame if columns are missing
102         data['Date'] = pd.to_datetime(data['Date'], errors='coerce') # Convert Date column to datetime
103         trends = data.groupby([data['Date'].dt.to_period("M"), 'Purchase Category'])['Customer_ID'].count().reset_index() # Group by month and purchase category
104         trends.columns = ['Month', 'Product Category', 'Sales'] # Rename columns
105         trends['Month'] = trends['Month'].dt.to_timestamp() # Convert period to timestamp
106         return trends
107
108 # File path to the dataset
109 file_path = r"Dataset.xlsx" # Replace with the actual path to the dataset
110
111 # Initialize the data ingestion context and ingest data
112 data_ingestion_context = DataIngestionContext(ExcelDataIngestion()) # Use Excel ingestion strategy
113 sales_data = data_ingestion_context.ingest(file_path) # Load sales data

```

Figure 37 Data Processing and visualization code

The Data Processing module handles the transformation and analysis of the ingested data. It defines an abstract class `DataProcess` with specific subclasses for different types of analysis. Each subclass, such as `TrendsOverTime`, `LocationDistribution`, `PairProductAnalysis`, and others, is responsible for processing the data in a specific way, like calculating transaction trends, customer distribution, or product pair frequencies. By implementing the Strategy Design Pattern, the module allows for easy extension and addition of new processing methods without disrupting the rest of the application. The results are returned as `DataFrames`, ready to be visualized or further analyzed, making this module the backbone of the application's data insights.

- **Dash Application (User Interface):**

```

115 # Initialize the Dash application
116 app = dash.Dash(__name__)
117
118 # Define the layout of the application
119 app.layout = html.Div(
120     children=[
121         html.H1(
122             children='Customer Sales Analysis', # Title of the application
123             style={'textAlign': 'center', 'color': '#2C3E50', 'padding': '20px'},
124         ),
125         # Dropdowns for Year and Month selection
126         html.Div(
127             children=[
128                 html.Div(
129                     children=[
130                         html.H3('Select Year:', style={'color': '#2C3E50'}),
131                         dcc.Dropdown(
132                             id='year-dropdown',
133                             options=[{'label': str(year),
134                                     'value': year} for year in sorted(sales_data['Date'].dt.year.unique())],
135                             placeholder='Select Year',
136                         ),
137                     ],
138                     style={'width': '48%', 'display': 'inline-block'},
139                 ),
140                 html.Div(
141                     children=[
142                         html.H3('Select Month:', style={'color': '#2C3E50'}),
143                         dcc.Dropdown(id='month-dropdown', placeholder='Select Month'),
144                     ],
145                     style={'width': '48%', 'display': 'inline-block', 'marginLeft': '4%'},
146                 ),
147             ],
148         ),

```

```

149     # Dropdown for selecting chart type
150     html.H3(
151         'Select a Chart to Display:', style={'color': '#2C3E50', 'paddingTop': '20px'})
152     ),
153     dcc.Dropdown(
154         id='chart-selector',
155         options=[
156             {'label': 'Trends Over Time', 'value': 'trends-over-time'},
157             {'label': 'Location Distribution', 'value': 'location-distribution'},
158             {'label': 'Pair Product Analysis', 'value': 'pair-product-analysis'},
159             {'label': 'Best-Selling Products', 'value': 'best-selling-products'},
160             {'label': 'Product Performance', 'value': 'product-performance'}],
161         ],
162         value='trends-over-time',
163     ),
164     # Placeholder for dynamic chart
165     html.Div(id='dynamic-chart'),
166 ],
167 style={'backgroundColor': '#ECF0F1', 'padding': '30px'},
168 )
169
170 # Callback for updating Month dropdown based on selected Year
171 @app.callback(
172     Output('month-dropdown', 'options'),
173     [Input('year-dropdown', 'value')],
174 )
175 def update_month_options(selected_year):
176     if selected_year is None:
177         return []
178     months = sales_data[sales_data['Date'].dt.year == selected_year]['Date'].dt.month.unique()
179     return [{'label': str(month), 'value': month} for month in sorted(months)]
180

```

```

# Callback for generating dynamic visualizations
@app.callback(
    Output('dynamic-chart', 'children'),
    [Input('year-dropdown', 'value'), Input('month-dropdown', 'value'), Input('chart-selector', 'value')],
)
def update_visualizations(selected_year, selected_month, selected_chart):
    filtered_data = sales_data.copy()
    if selected_year:
        filtered_data = filtered_data[filtered_data['Date'].dt.year == selected_year]
    if selected_month:
        filtered_data = filtered_data[filtered_data['Date'].dt.month == selected_month]
    if selected_chart == 'trends-over-time':
        trends_data = TrendsOverTime().process_data(filtered_data)
        fig = px.line(trends_data, x='Date', y='TotalTransactions', title='Trends Over Time')
    elif selected_chart == 'location-distribution':
        location_data = LocationDistribution().process_data(filtered_data)
        fig = px.bar(location_data, x='Location', y='CustomerCount', title='Location Distribution')
    elif selected_chart == 'pair-product-analysis':
        pair_product_data = PairProductAnalysis().process_data(filtered_data)
        fig = px.bar(
            pair_product_data.head(10), x='Product 1', y='Frequency', color='Product 2', title='Top 10 Product Pairs Purchased Together'
        )
    elif selected_chart == 'best-selling-products':
        best_selling_products = BestSellingProductAnalysis().process_data(filtered_data)
        fig = px.bar(best_selling_products, x='Purchase Category', y='Frequency', title='Best-Selling Products')
    elif selected_chart == 'product-performance':
        product_performance = ProductPerformanceAnalysis().process_data(filtered_data)
        fig = px.line(product_performance, x='Month', y='Sales', color='Product Category', title='Product Performance Over Time')
    else:
        fig = None
    return dcc.Graph(figure=fig) if fig else None

```

```

212
213 # Run the application
214 if __name__ == '__main__':
215     app.run_server(debug=True)
216

```

Figure 38 dash User Interface code

The Dash Application (User Interface) module is responsible for creating an interactive web application that allows users to visualize the processed sales data. Built using the Dash framework, the user interface consists of dropdown menus, charts, and dynamic content based on user interactions. The `app.layout` defines the structure of the UI, including options for selecting a year, month, and chart type, along with a dynamic chart area that updates based on user selections. Callbacks link the inputs (e.g., selected year, month, chart type) to the outputs (e.g., updated charts), making the interface interactive and responsive. This module leverages Plotly to generate visually appealing charts, enabling users to explore different aspects of sales data with ease.

## Assessment of Effectiveness

The developed sales analysis application for Sampath Food City effectively automates the sales data analysis process, addressing issues such as human bias, errors, and inefficiencies in the previous manual approach. This assessment evaluates how the application adheres to SOLID principles, employs clean coding techniques, and integrates design patterns to ensure modularity, scalability, and maintainability.

### • Adherence to SOLID Principles

#### 1. Single Responsibility Principle (SRP):

Each class in the application is designed to handle a single responsibility:

- **ExcelDataIngestion:** Focuses on reading data from Excel files and cleaning column names.
- **Data Processing Classes:** For example, `TrendsOverTime` calculates transaction trends, and `PairProductAnalysis` identifies frequently purchased product pairs.
- **Visualization Layer:** Handles user interaction and chart rendering using Dash.

This separation of concerns ensures that each module is easy to maintain and extend without affecting unrelated functionalities.

#### 2. Open/Closed Principle (OCP):

The application is designed to be open for extension but closed for modification. For instance: ○ New data ingestion methods, such as handling CSV or database inputs, can be added by creating new classes implementing the `DataIngestionStrategy` interface, without altering existing

code. ○ Similarly, new types of analyses can be implemented by adding new subclasses of `DataProcess`.

### 3. Liskov Substitution Principle (LSP):

Subclasses like `ExcelDataIngestion` can seamlessly replace their parent `DataIngestionStrategy` without affecting the application's behavior. This ensures that any strategy implementing the interface can be used interchangeably in the `DataIngestionContext`.

### 4. Interface Segregation Principle (ISP):

The use of narrowly focused interfaces such as `DataIngestionStrategy` and `DataProcess` ensures that classes are not burdened with unnecessary methods. Each interface enforces the implementation of methods specific to their functionality.

### 5. Dependency Inversion Principle (DIP):

High-level modules like `DataIngestionContext` depend on abstractions (`DataIngestionStrategy`), not concrete implementations (`ExcelDataIngestion`). This decoupling makes the system flexible and promotes reusability.

## • Implementation of Clean Coding Techniques

### 1. Descriptive Naming:

The code uses clear and meaningful names for classes, methods, and variables:

- **Example:** `BestSellingProductAnalysis` explicitly describes its purpose of analyzing topselling products.
- Functions like `ingest_data` and `process_data` are self-explanatory, making the code intuitive for developers.

### 2. Separation of Concerns:

The application separates logic into distinct layers:

- **Data Ingestion:** Handles file loading and initial validation (e.g., Excel Data Ingestion).
- **Data Processing:** Each analysis type (e.g., Trends Over Time, Region Analysis) is managed by a dedicated class.
- **Visualization:** Dash manages user interaction and rendering charts dynamically.

This modular design improves readability and maintainability.

### 3. Error Handling:

The application includes comprehensive error handling:

- Excel Data Ingestion gracefully handles missing files or invalid data by returning an empty Data Frame and logging appropriate warnings.
- Data processing classes validate input data for required columns and provide warnings when inputs are incomplete.

#### **4. Readability and Maintainability:**

- Code blocks are structured logically, with functions performing a single task.
- Comments and consistent formatting make the code easy to understand and maintain.

### **• Use of Design Patterns**

#### **1. Strategy Pattern:**

- The DataIngestionStrategy interface defines a common structure for all data ingestion methods.
- The ExcelDataIngestion class implements this interface, and the DataIngestionContext dynamically selects the appropriate strategy.
- Future support for additional data formats (e.g., CSV, JSON) can be added by implementing new strategies.

#### **2. Template Method Pattern:**

- The DataProcess interface standardizes the structure for all data processing classes, defining a process\_data method to be implemented by subclasses.
- Subclasses like TrendsOverTime and PairProductAnalysis provide specific implementations while following the same framework.

#### **3. Factory Pattern:**

- A factory pattern could be applied to streamline the creation of chart objects in the visualization module, further decoupling the chart rendering logic.

#### **4. Observer Pattern (Proposed Enhancement):**

- An observer pattern could be introduced to dynamically update visualizations when the dataset or filters change, improving real-time interactivity.



## User Interface (UI)

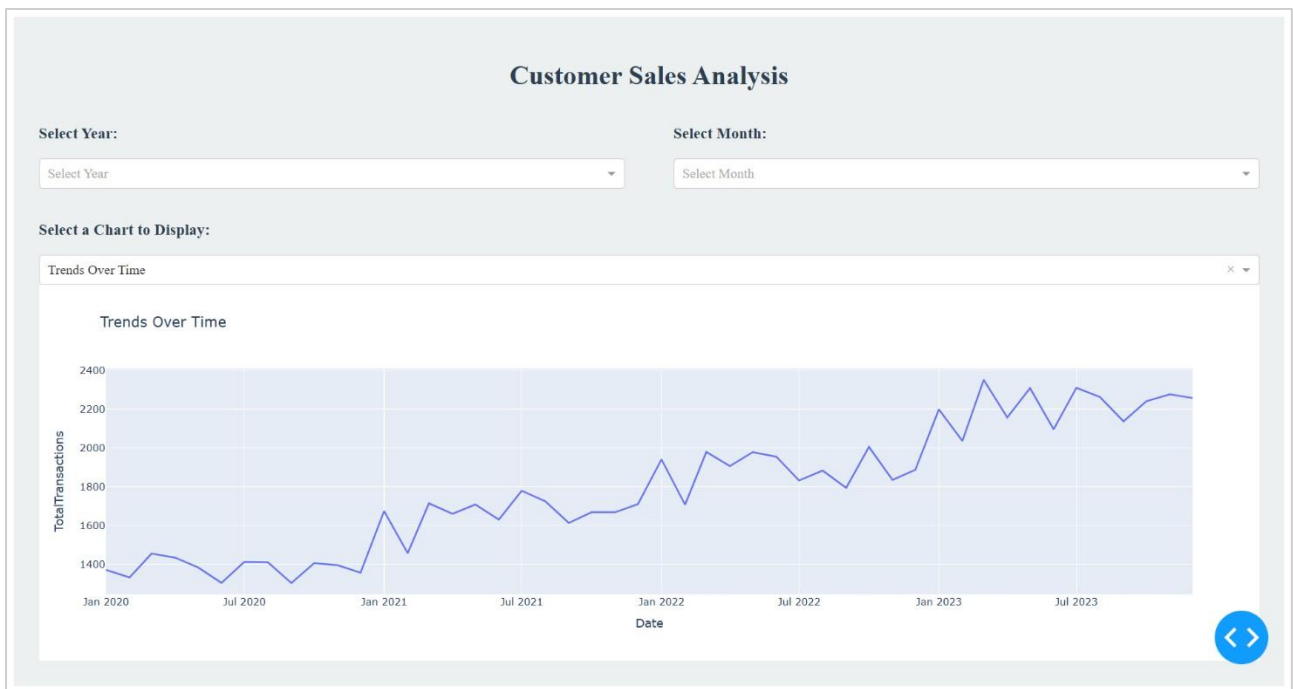


Figure 39 Overview of Sales Analysis Application

The Sales Analysis Application provides an interactive dashboard to analyze customer sales data. The design ensures that users can explore different aspects of sales data without requiring technical expertise. It includes filters for selecting the desired year, month, and analysis type. The header of the application displays the name "Customer Sales Analysis", clearly defining the purpose of the tool as a comprehensive sales data analysis platform.

### • Header with Filter Dropdown Option:

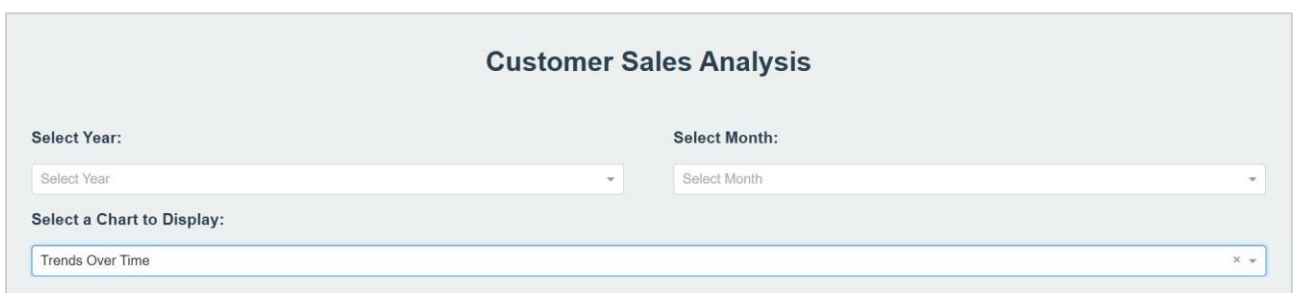


Figure 40 UI\_Header with Filter Dropdown Option

The header of the application is titled "**Customer Sales Analysis**" and is styled prominently to emphasize its purpose. It provides dropdown filters for year, month, and chart selection, allowing users to customize their analysis dynamically. These filters ensure users can explore specific aspects of the dataset with ease.

- **Date (Year and Month) Filter:**

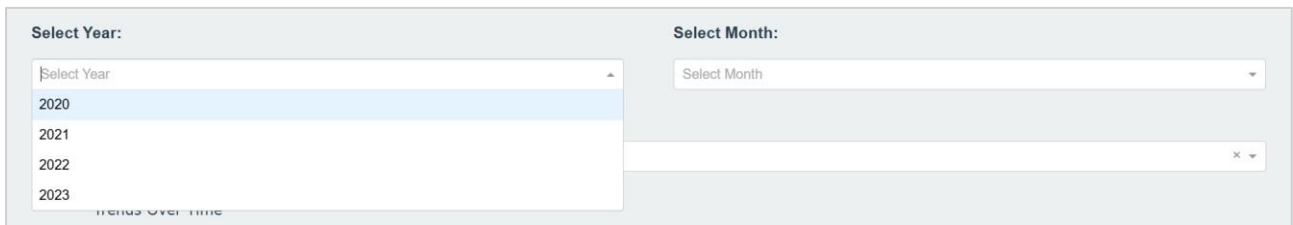


Figure 41 date vice filter

The application includes dropdown filters for **year** and **month**. These dropdowns dynamically populate available options by fetching unique years and months from the dataset (sales\_data), specifically from the **Date** column. Users can select a specific year and month to focus their analysis on a particular time period.

- **Chart Selector Dropdown:**

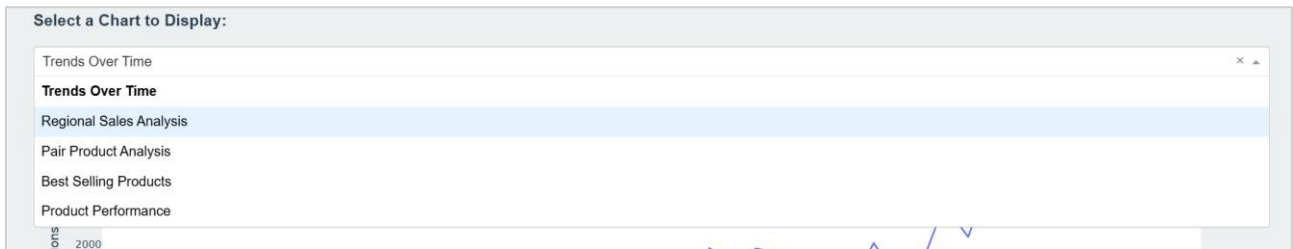


Figure 42 UI\_Chart Selector

The chart selector dropdown provides the user with the ability to choose from five chart types, each offering a unique perspective on the sales data:

1. **Trends Over Time:** Displays a line chart showing the number of customer transactions over time.
2. **Product Performance:** Displays a line chart showing the sales trends of product categories over time.
3. **Best-Selling Products:** Displays a bar chart highlighting the top-selling product categories.
4. **Regional Sales Analysis:** Displays a bar chart showing customer distribution across different locations.
5. **Pair Product Analysis:** Displays a bar chart showing the frequency of product category pairs purchased together.

Once a chart type is selected, the corresponding chart is displayed in the dynamic section of the dashboard.

- **Analysis Charts:**

## 1. Trends Over Time:

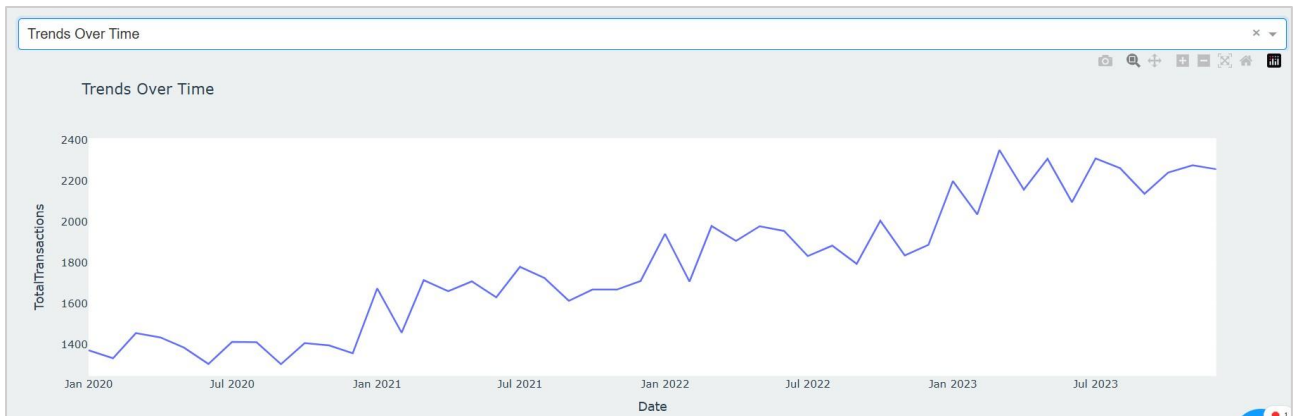


Figure 43 Trends over time analysis

The "Trends Over Time" chart visualizes customer transactions over time using a line chart.

- X-Axis: Represents dates.
- Y-Axis: Represents the total number of transactions.

This chart highlights trends and patterns in customer activity, helping to identify peak periods of engagement.

## 2. Product Performance Analysis:

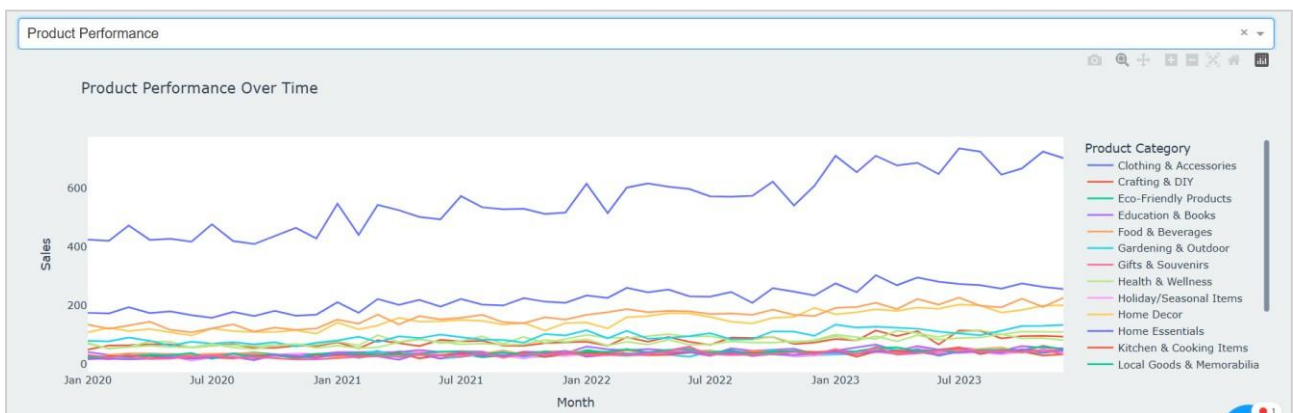


Figure 44 Product Performance analysis

The "Product Performance Over Time" chart is a line chart that illustrates the sales trends of different product categories.

- X-Axis: Represents months.
- Y-Axis: Represents the number of sales for each product category.

Each line represents a specific product category, enabling a detailed comparison of sales patterns.

## 3. Regional Sales Analysis:

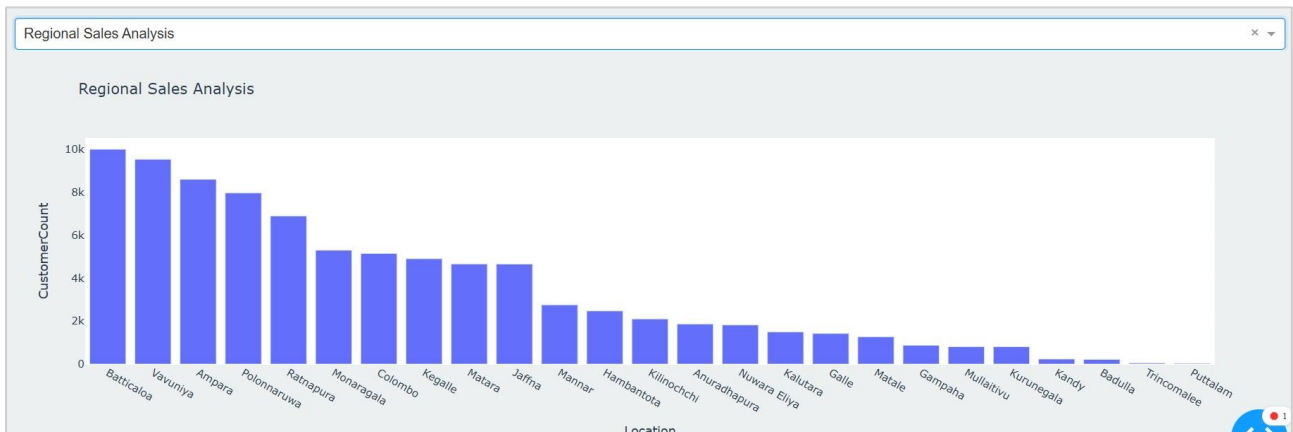


Figure 45 Regional Sales Analysis

The "Regional Sales Analysis" chart is a bar chart that shows customer distribution across different regions.

- X-Axis: Represents locations (e.g., Colombo, Batticaloa).
- Y-Axis: Represents the total number of customers.

This chart offers insights into the geographic spread of customer activity.

#### 4. Pair Product Analysis:

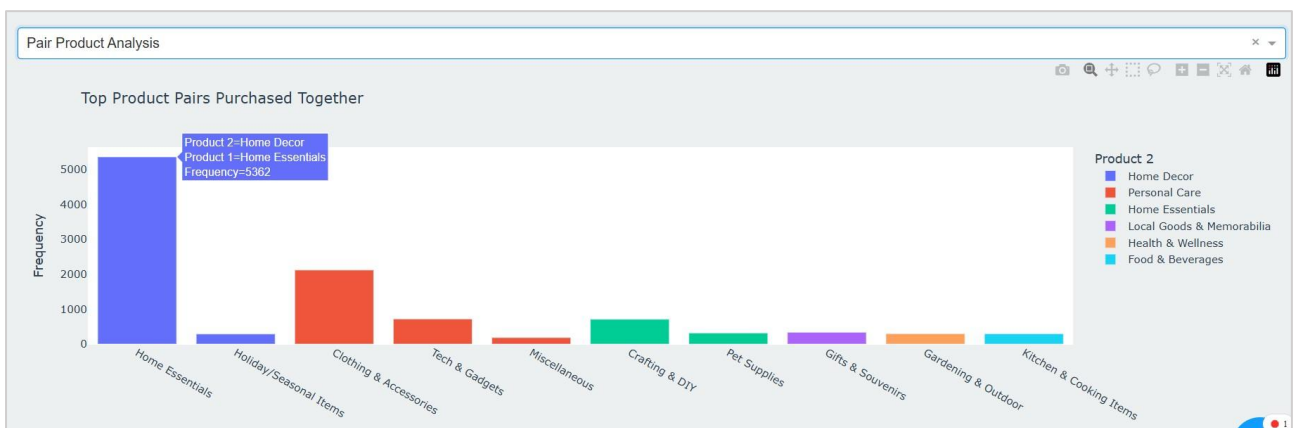


Figure 46 Pair Product Analysis

The "Pair Product Analysis" chart is a bar chart showing frequently purchased product category pairs.

- X-Axis: Represents the first product in the pair.
- Y-Axis: Represents the frequency of the pair being purchased together.

The bar color represents the second product in the pair, with a legend for easy identification. For example, pairs like "Home Essentials" and "Home Decor" are highlighted.

#### 5. Best-Selling Product Analysis:

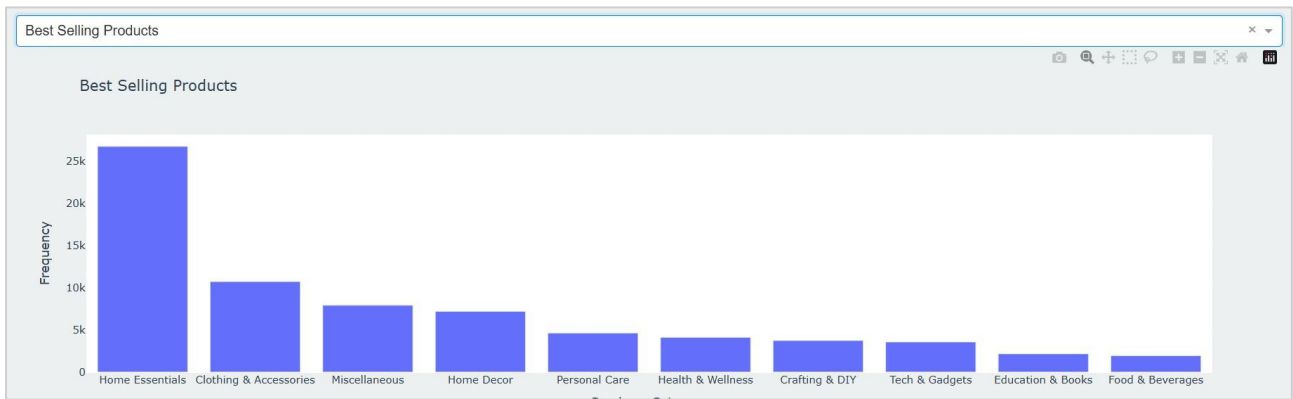


Figure 47 Best-Selling Product analysis

The "Best-Selling Products" chart is a bar chart highlighting the top-performing product categories.

- X-Axis: Represents product categories (e.g., "Home Essentials").
- Y-Axis: Represents the frequency of sales for each category.

The chart provides a clear view of the most popular product categories, helping management optimize inventory and marketing efforts.

The proposed sales data analysis application for Sampath Food City (PVT) Ltd is an effective solution for automating and enhancing the data analysis process. By using Python and incorporating SOLID principles, clean coding techniques, and design patterns, the application is both flexible and maintainable.

The application provides valuable insights to management in real-time, helping in decision-making processes related to product sales, customer behavior, and regional performance.

The implementation of SOLID principles and clean coding techniques has significantly improved the structure and maintainability of the codebase, making it easier to extend and adapt as business needs evolve.