**E-COMMERCE ANALYTICS PLATFORM**

**UIT2312 – PROGRAMMING AND DESIGN PATTERNS PROJECT**

**A PROJECT REPORT**

***Submitted by***

|  |  |
| --- | --- |
| **Devanithimaran E** | **3122 23 5002 029** |
| **Dharanikaran S** | **3122 23 5002 030** |

**SSN COLLEGE OF ENGINEERING, KALAVAKKAM**



# INDEX

|  |  |  |
| --- | --- | --- |
| **S.No** | **Content** | **Page Number** |
| 1 | Aim | 3 |
| 2 | Class Diagram | 3 |
| 3 | Procedure | 4 |
| 4 | Sample Code | 6 |
| 5 | Output | 8 |
| 6 | Result | 10 |

# AIM

To develop a comprehensive e-commerce platform that aggregates, analyzes, and recommends offers and discounts on products in real-time, providing users with tailored insights and an enhanced shopping experience. This project includes page breakdowns for various components, user interface elements, and analytics functionalities for offer recommendations.

# CLASS DIAGRAM

# 

# 

# PROCEDURE

|  |  |  |  |
| --- | --- | --- | --- |
| **S. No.** | **Module Name** | **Design Pattern** | **Justification for Design Pattern Used** |
| 1 | Email Notification | Command pattern | Allows encapsulating the action of sending a thank-you email as an object, making it flexible to queue, log, or execute independently. |
| 2 | Scraper Factory | Factory pattern | Provides a way to create scrapers for various platforms dynamically, promoting flexibility and scalability as new platforms are added. |
| 3 | Database Connection | Singleton pattern | Ensures a single database connection instance, optimizing resource usage and preventing potential conflicts in database access. |

**Singleton Pattern Implementation**

**Step 1: Singleton Pattern Implementation**

**Class Definition and Singleton Instance:**

* The UserRegistration class is defined with a private class variable \_instance set to None.
* The \_\_new\_\_ method is overridden to ensure that only one instance of the UserRegistration class is created.
* If \_instance is None, a new instance is created; otherwise, the existing instance is returned.

**Step 2: Checking Existing User**

**check\_existing\_user Method:**

* This method accepts name, email, and phone as parameters.
* Executes a SQL query to count the number of users with the provided name, email, and phone.
* Returns True if a user exists, otherwise False.

**Step 3: Inserting a New User**

**insert\_user Method:**

* Accepts name, email, phone, and role as parameters.
* Retrieves the maximum user\_id from the Users table and calculates the new user\_id.
* Inserts the new user details into the Users table.
* Commits the transaction and shows a success message.

**Factory Pattern Implementation**

**Step 1: Abstract Product Class**

* Define an abstract Product class with common methods for the products (e.g., get\_product\_name()).
* The Product class serves as the blueprint for all specific products.

**Step 2: Concrete Product Classes**

* Implement concrete classes like AmazonProduct and FlipkartProduct that inherit from the Product class.
* Each concrete class implements the product-specific behavior for methods defined in the Product class (e.g., get\_product\_name()).

**Step 3: Abstract Creator Class**

* Define an abstract Creator class that declares the create\_product() method.
* This method will return an instance of a Product, but the exact product type will be determined by the concrete creator class.

**Step 4: Concrete Creator Classes**

* Implement concrete AmazonProductCreator and FlipkartProductCreator classes that inherit from the Creator class.
* These classes override the create\_product() method to return the specific product type (e.g., AmazonProduct or FlipkartProduct).

**Step 5: Client Code**

* The client code calls the appropriate creator class to obtain a product (e.g., AmazonProductCreator.create\_product()).
* The client does not need to know the exact product type, ensuring the system remains flexible and easy to extend.

**Command Pattern Implementation**

**Step 1: Command Interface**

* Define an abstract Command class with an abstract method execute().
* The Command class represents a contract for all commands, ensuring they implement the execute() method.

**Step 2: Concrete Command Classes**

* Implement concrete classes like SendThankYouEmailCommand that inherit from the Command class.
* Each concrete class implements the execute() method to perform the specific action (e.g., sending a thank-you email).

**Step 3: Invoker Class**

* The Invoker class holds a reference to a Command object.
* It calls the execute() method on the Command object when needed.
* It can execute a single command or multiple commands, depending on the context.

**Step 4: Client Code**

* The client creates an instance of the concrete command class (e.g., SendThankYouEmailCommand).
* It then passes this command to the Invoker to execute the action.

**Step 5: (Optional) Command Queues**

* If the application requires, the Invoker can maintain a queue of commands and execute them sequentially.
* This allows for undo/redo functionality and queuing of multiple commands to be executed later.

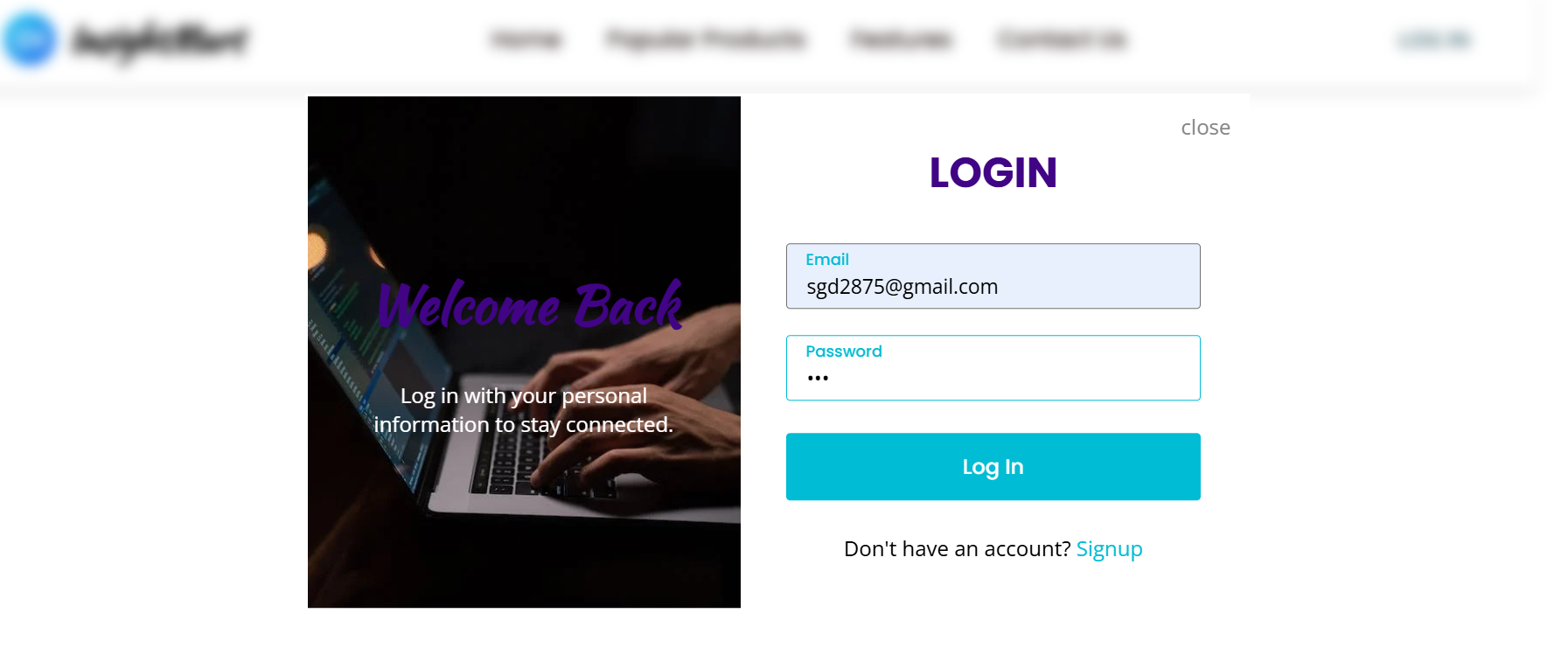
# SAMPLE CODE

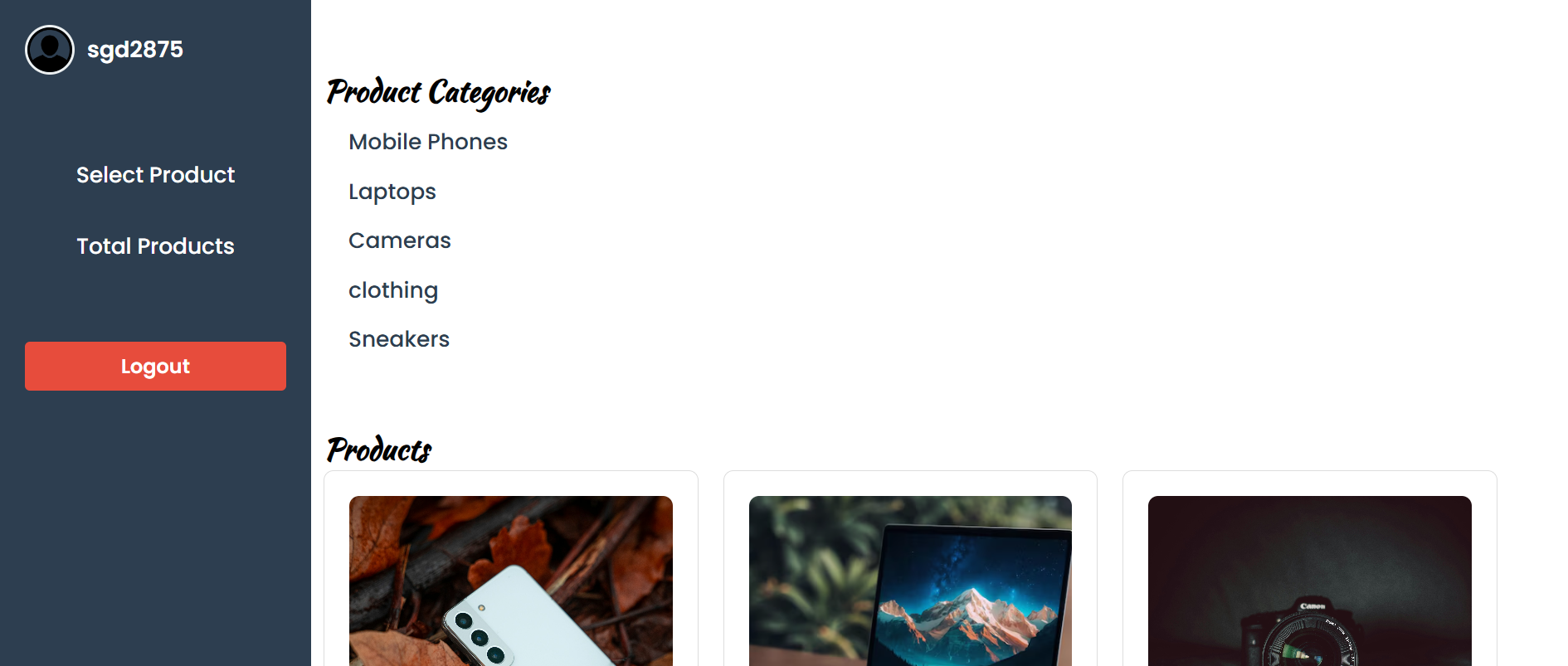
# 

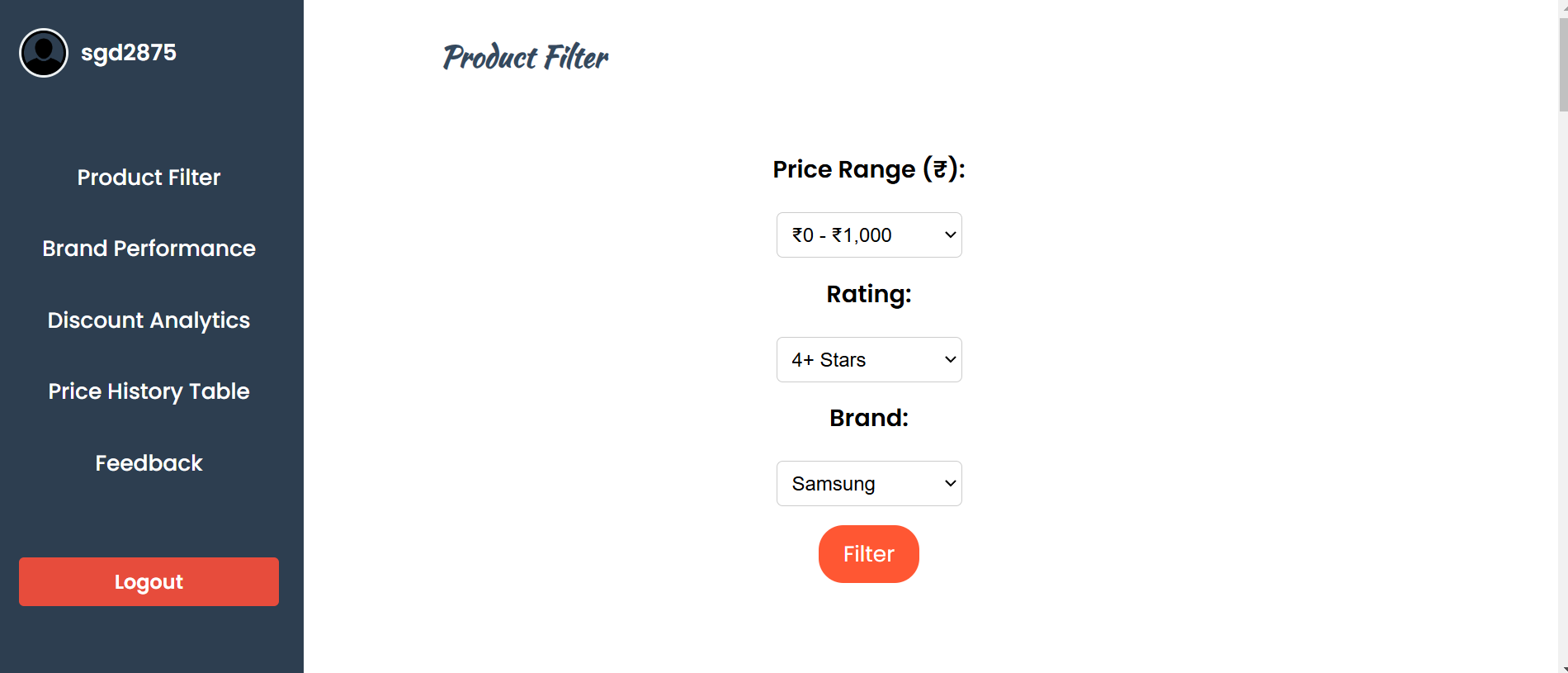
# 

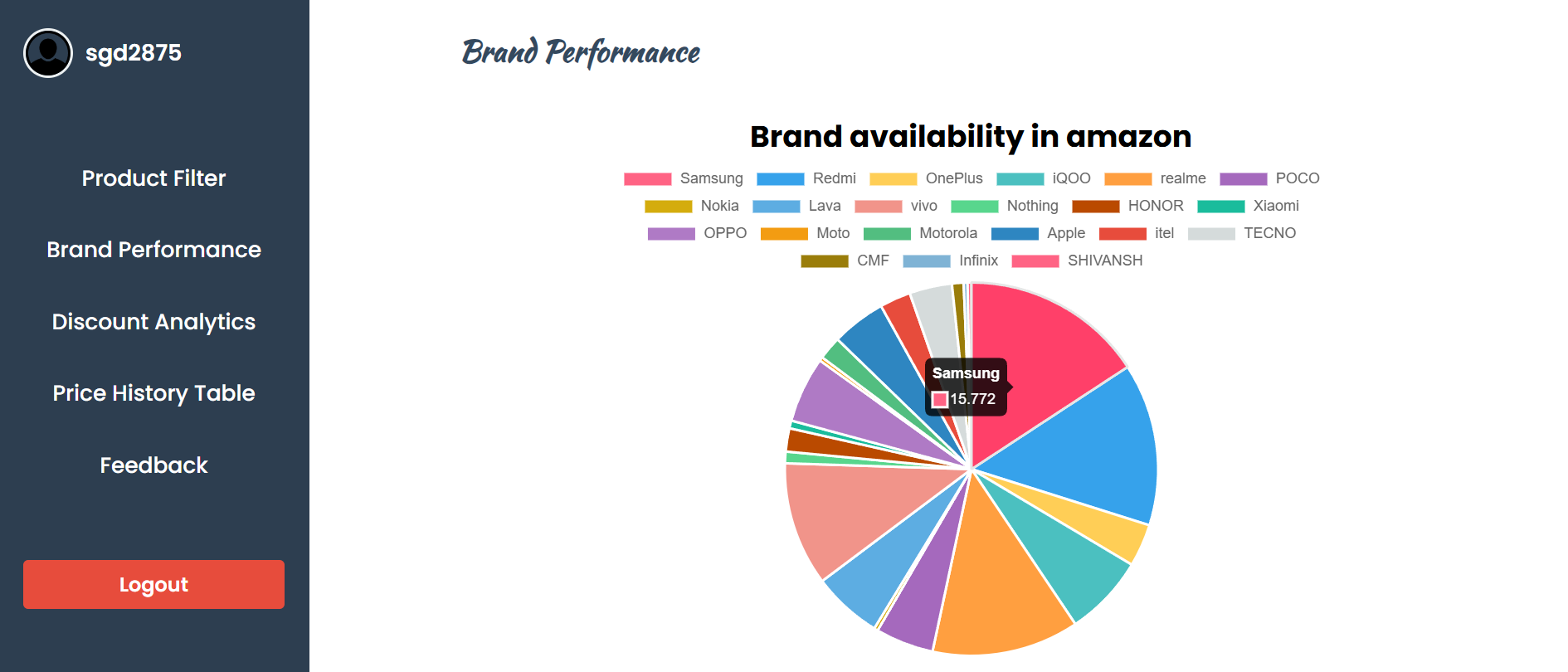
****

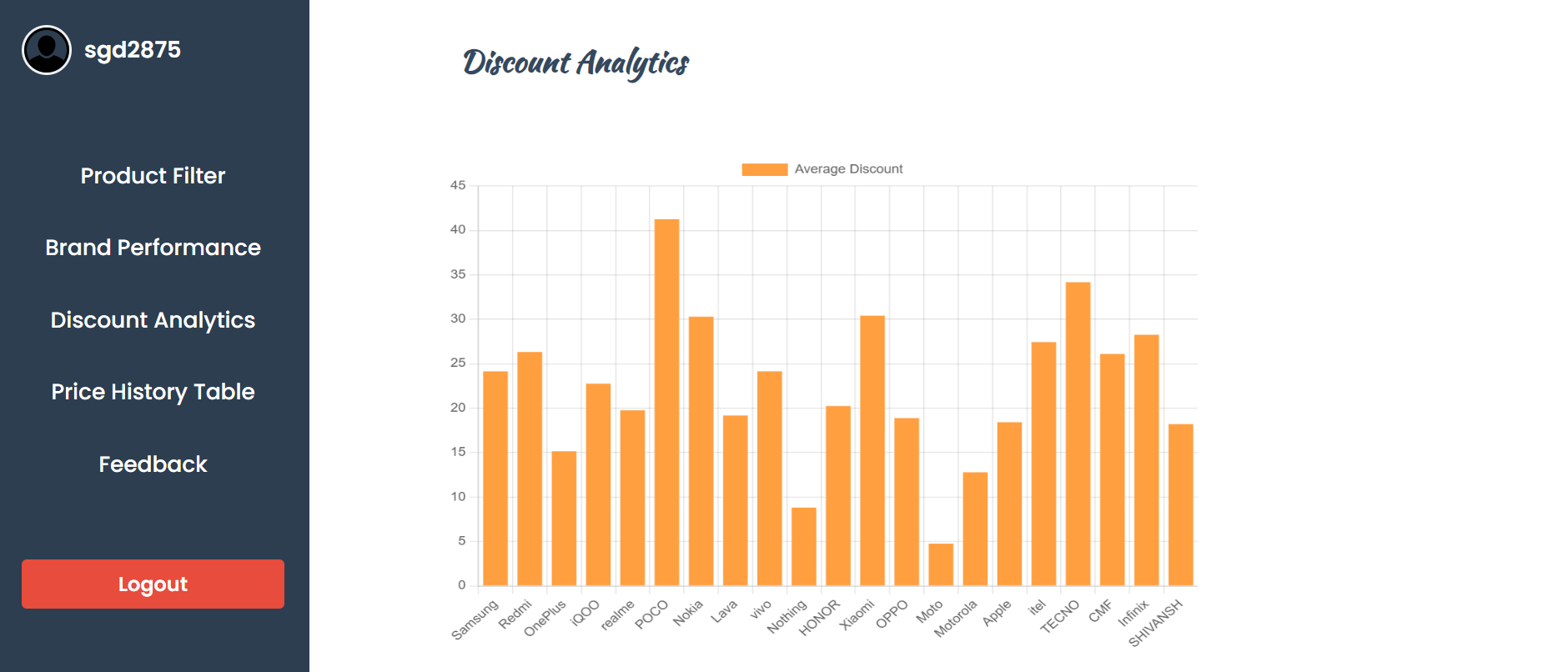
1. **OUTPUT**

****

****

****

****

****

1. **RESULT**

Thus, we have successfully developed and built a ecommerce analytics platform which given statistical datas and updates of products from popular ecommerce website like amazon