Project Report: Sales & Revenue Analysis for a Small Business

--Sayan Das

Primary Goals

1. **Extract** and **transform** the CSV data (customers, products, and sales) into a consistent, clean format.

2. **Load** the cleaned data into a **MySQL** database, organizing it into fact and dimension tables for easier analysis (star schema approach, but without a separate date dimension).

3. **Analyze** the data using **SQL** queries (top-selling products, low-performing categories, customer segmentation).

4. **Perform Predictive Modeling** using Python to:
   - **Forecast future sales trends** with **Prophet**.
   - **Predict Customer Lifetime Value (CLV)** using **BG/NBD** and **Gamma-Gamma** models from the **lifetimes** package.

Assumptions and Tools

- **Python Environment**: I used Python 3.x with the following libraries:
  - **pandas**, **numpy** for data manipulation
  - **sqlalchemy** and **pymysql** for MySQL connectivity
  - **dateutil** for robust date parsing
  - **matplotlib** and **prophet** for forecasting
  - **lifetimes** for CLV modeling
- **MySQL**: Database named `case5`, user credentials set to `root/12345` on localhost.
- **CSV Encodings**: Detected using the **chardet** library; determined to be `ISO-8859-1`.

1. Loading CSV Data into Pandas

I had four CSV datasets: - **AdventureWorks_Customers.csv**
- **AdventureWorks_Products.csv**
- **AdventureWorks_Sales_2015.csv**, **AdventureWorks_Sales_2016.csv**, **AdventureWorks_Sales_2017.csv**

I used the following Python code to detect file encodings and read the CSVs (excerpt):

```python
import chardet

# Detect file encoding
with open('AdventureWorks_Customers.csv', 'rb') as f:
    raw_data = f.read()
result = chardet.detect(raw_data)
print(result)  # -> {'encoding': 'ISO-8859-1', 'confidence': 0.73, 'language': ''}

# Read CSV files with the detected encoding
customers_df = pd.read_csv("AdventureWorks_Customers.csv", encoding="ISO-8859-1")
products_df = pd.read_csv("AdventureWorks_Products.csv", encoding="ISO-8859-1")
sales_2015_df = pd.read_csv("AW Sales/AdventureWorks_Sales_2015.csv", encoding="ISO-8859-1")
# etc.
```

This ensured I avoided `UnicodeDecodeError` issues.

2. Data Cleaning and Transformation in Python

I performed multiple data cleaning steps:

1. **Parsing BirthDate** (with slashes and dashes) using `dateutil.parser.parse`:

```
def parse_birthdate(date_str):
    try:
        return parser.parse(date_str)
    except:
        return pd.NaT

customers_df['BirthDate'] = customers_df['BirthDate'].apply(parse_birthdate)
```

2. **Cleaning AnnualIncome** by removing symbols ($, commas) and converting to numeric:

```
customers_df['AnnualIncome'] = (
    customers_df['AnnualIncome']
    .replace({r'\$': '', ',': ''}, regex=True)
    .str.strip()
)
customers_df['AnnualIncome'] = pd.to_numeric(customers_df['AnnualIncome'],
errors='coerce')
```

3. **Dropping Rows** with critical missing values:

```
before_drop = len(customers_df)
customers_df.dropna(subset=['CustomerKey', 'BirthDate', 'AnnualIncome'],
inplace=True)
after_drop = len(customers_df)
print(f"Dropped {before_drop - after_drop} rows from customers_df due to missing
fields.")
```

4. **Transforming ProductSize** (S → 44, M → 48, etc.):

```
size_mapping = {'S': 44, 'M': 48, 'L': 52, 'XL': 62}

def transform_size(x):
    if isinstance(x, str):
        x = x.strip()
        if x in size_mapping:
            return size_mapping[x]
        else:
            try:
                return int(x)
            except ValueError:
                return np.nan
    return x

products_df['ProductSize'] = products_df['ProductSize'].apply(transform_size)
products_df['ProductSize'] = pd.to_numeric(products_df['ProductSize'],
errors='coerce')
```

5. **Combining Sales Data** from 2015, 2016, and 2017:

```
sales_df = pd.concat([sales_2015_df, sales_2016_df, sales_2017_df],
ignore_index=True)
```

6. **Converting OrderDate & StockDate** to datetime and dropping invalid rows:

```
sales_df['OrderDate'] = pd.to_datetime(sales_df['OrderDate'], errors='coerce')
sales_df['StockDate'] = pd.to_datetime(sales_df['StockDate'], errors='coerce')
```

7. **Creating a CompositeKey** = OrderNumber + OrderLineItem for uniqueness:

```
sales_df['CompositeKey'] = (
    sales_df['OrderNumber'].astype(str) + "_" +
    sales_df['OrderLineItem'].astype(str)
)
```

8. **Dropping Missing Values** in sales:

```
before_drop = len(sales_df)
sales_df.dropna(subset=['OrderDate', 'ProductKey', 'CustomerKey', 'OrderQuantity'],
inplace=True)
after_drop = len(sales_df)
print(f"Dropped {before_drop - after_drop} rows from sales_df due to missing
fields.")
```

9. **Calculating Revenue** = OrderQuantity * ProductPrice (after merging ProductPrice from `products_df`).

3. Creating Fact and Dimension Tables

I opted for a star-schema style approach **without a separate date dimension**:

1. **`dim_customer`**: Contains each customer's attributes (e.g., FirstName, LastName, BirthDate, etc.).
2. **`dim_product`**: Contains product attributes (e.g., ProductName, ProductSize, ProductPrice, etc.).
3. **`fact_sales`**: Contains the measures (`OrderQuantity`, `Revenue`) and foreign keys (`CustomerKey`, `ProductKey`), plus the date columns (`OrderDate`, `StockDate`).

Building Dimensions

```
dim_customer = customers_df.drop_duplicates(subset=['CustomerKey']).copy()
dim_product = products_df.drop_duplicates(subset=['ProductKey']).copy()
```

Building the Fact Table

```
fact_sales = sales_df.merge(
    dim_customer[['CustomerKey']], on='CustomerKey', how='left'
).merge(
    dim_product[['ProductKey']], on='ProductKey', how='left'
)

fact_sales = fact_sales[[
    'CompositeKey',
    'CustomerKey',
    'ProductKey',
    'OrderDate',
    'StockDate',
    'OrderQuantity',
    'ProductPrice',
```

```
    'Revenue'
]]
```

4. Loading Data to MySQL

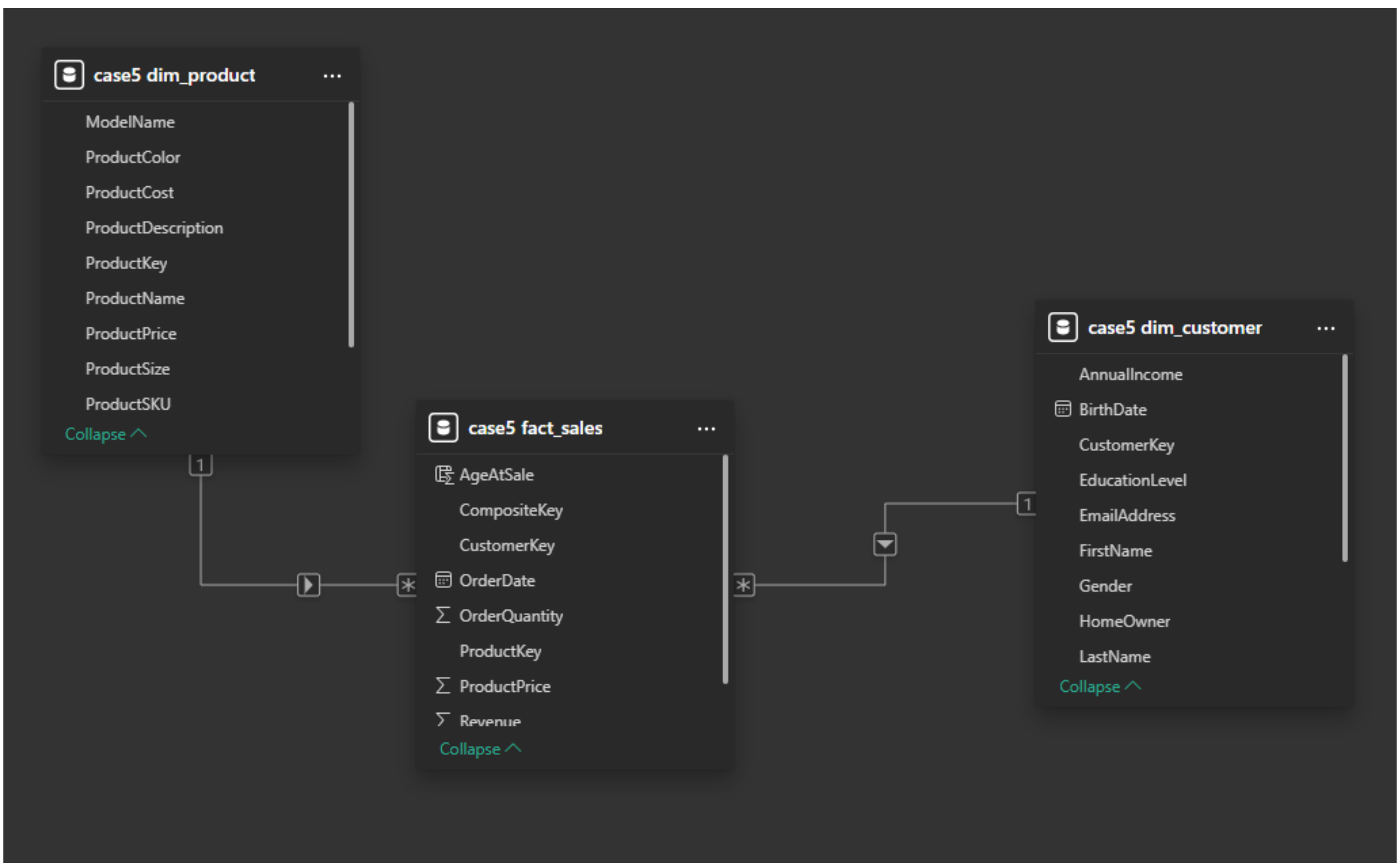
I used **SQLAlchemy** to connect and write the tables to MySQL:

```
username = 'root'
password = '12345'
host = 'localhost'
port = '3306'
database = 'case5'
engine = create_engine(f'mysql+pymysql://{username}:{password}@{host}:{port}/{database}')

dim_customer.to_sql('dim_customer', engine, index=False, if_exists='replace')
dim_product.to_sql('dim_product', engine, index=False, if_exists='replace')
fact_sales.to_sql('fact_sales', engine, index=False, if_exists='replace')

print("Data loaded to MySQL successfully!")
```

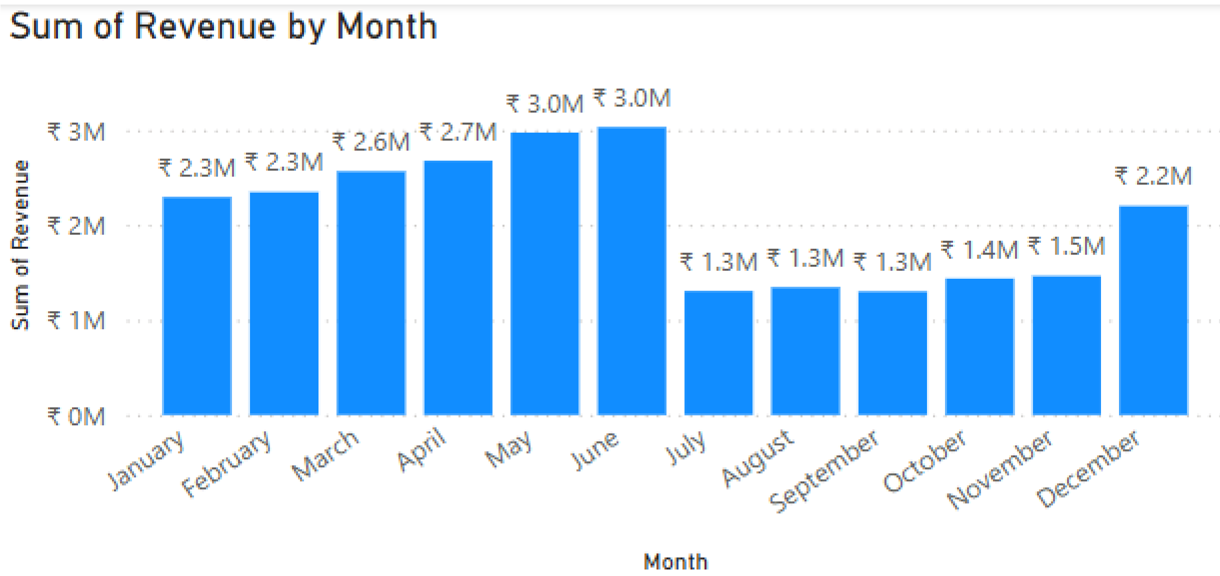
At this point, I had **three tables** in MySQL:
- **`dim_customer`**
- **`dim_product`**
- **`fact_sales`**

5. SQL Queries for Analysis

Schema



Sales trends across months



Best-performing product categories

```sql
-- Top 10 products by total revenue
SELECT
    p.ProductKey,
    p.ProductName,
    SUM(f.Revenue) AS TotalRevenue
FROM fact_sales AS f
JOIN dim_product AS p
    ON f.ProductKey = p.ProductKey
GROUP BY p.ProductKey, p.ProductName
ORDER BY TotalRevenue DESC
LIMIT 10;
```

Result Grid | Filter Rows: | Export:

ProductKey	ProductName	TotalRevenue
362	Mountain-200 Black, 46	1241753.509199993
360	Mountain-200 Black, 42	1233557.1163999934
352	Mountain-200 Silver, 38	1213851.8856000006
356	Mountain-200 Silver, 46	1182780.591600002
358	Mountain-200 Black, 38	1165936.875799997
354	Mountain-200 Silver, 42	1133066.5212000043
377	Road-250 Black, 52	689373.75
371	Road-250 Red, 58	661013.4375
375	Road-250 Black, 48	641379.375
312	Road-150 Red, 48	640510.3300000016

```sql
-- Bottom 5 product subcategories by total revenue
SELECT
    p.ProductSubcategoryKey,
    SUM(f.Revenue) AS SubcategoryRevenue
FROM fact_sales AS f
JOIN dim_product AS p
    ON f.ProductKey = p.ProductKey
```

```
GROUP BY p.ProductSubcategoryKey
ORDER BY SubcategoryRevenue ASC
LIMIT 5;
```

Result Grid | Filter Rows: | Ex

ProductSubcategoryKey	SubcategoryRevenue
23	9556.369999999892
29	13562.699999999897
25	33083.5
19	35882.07420000042
26	36240

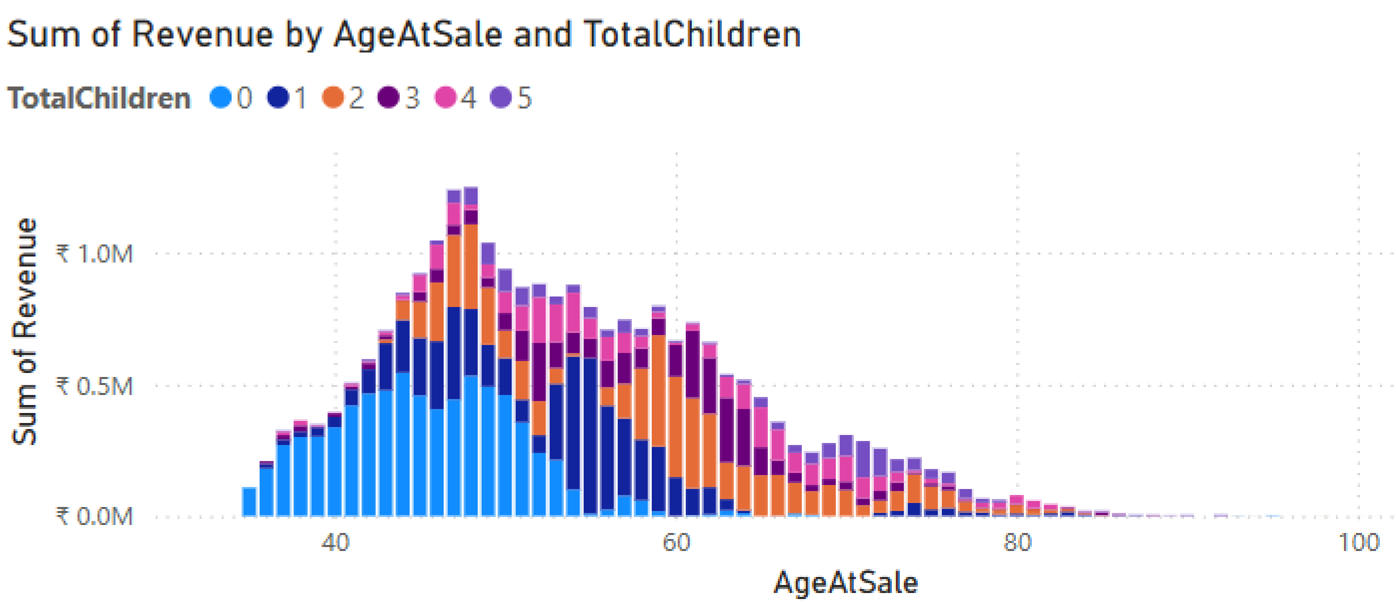
Revenue impact of discounts and promotions:

Cannot be generated as no discount or promotion data is given

Customer purchase patterns and segmentation

```
-- Customer Segmentation (High-Value, Frequent, Occasional)
WITH customer_summary AS (
    SELECT
        c.CustomerKey,
        COUNT(DISTINCT f.CompositeKey) AS total_orders,
        SUM(f.Revenue) AS total_spent
    FROM fact_sales AS f
    JOIN dim_customer AS c
        ON f.CustomerKey = c.CustomerKey
    GROUP BY c.CustomerKey
)
SELECT
    CustomerKey,
    total_orders,
    total_spent,
    CASE
        WHEN total_spent >= 1000 THEN 'High-Value'
        WHEN total_orders >= 10 THEN 'Frequent Buyer'
        WHEN total_orders BETWEEN 2 AND 9 THEN 'Occasional Buyer'
        ELSE 'Rare Buyer'
    END AS Segment
FROM customer_summary
ORDER BY total_spent DESC;
```

CustomerKey	total_orders	total_spent	Segment
11433	12	12407.9545	High-Value
11439	14	12015.4029	High-Value
11241	25	11330.449399999998	High-Value
11417	17	11085.750399999999	High-Value
11420	17	11022.4002	High-Value
11242	12	10852.034	High-Value
13263	12	10436.5079	High-Value
12655	11	10394.9836	High-Value
11425	15	10391.4304	High-Value
12631	11	10331.7349	High-Value
12650	9	10329.229099999999	High-Value
13405	11	10308.524899999997	High-Value
11429	17	10289.686200000002	High-Value
12632	6	10282.9121	High-Value
11245	9	10165.9221	High-Value
11237	6	10065.0121	High-Value
11428	12	9761.602199999998	High-Value
11427	9	9717.6506	High-Value
11423	10	9716.9908	High-Value
11412	11	9706.913400000001	High-Value
11431	11	9687.366599999998	High-Value
11249	11	9668.023899999998	High-Value
11421	4	9534.1482	High-Value
14186	10	9302.586299999999	High-Value



6. Predictive Modeling

6.1 Forecast Future Sales with Prophet

I aggregated daily revenue from `fact_sales` and fit a **Prophet** model:

```
import matplotlib.pyplot as plt
from prophet import Prophet

sales_trends = fact_sales.groupby('OrderDate')['Revenue'].sum().reset_index()
```

```python
sales_trends.columns = ['ds', 'y']  # Prophet requires ds, y

model = Prophet()
model.fit(sales_trends)

future = model.make_future_dataframe(periods=90)
forecast = model.predict(future)

model.plot(forecast)
plt.title("Forecast of Future Sales Revenue")
plt.xlabel("Date")
plt.ylabel("Revenue")
plt.show()
```

Prophet produced a forecast line (blue) and confidence intervals (shaded region) for the next 90 days. This helps me anticipate **future revenue trends** and manage inventory.

6.2 Predict Customer Lifetime Value (CLV)

Using the **lifetimes** library, I applied the **BG/NBD** and **Gamma-Gamma** models:

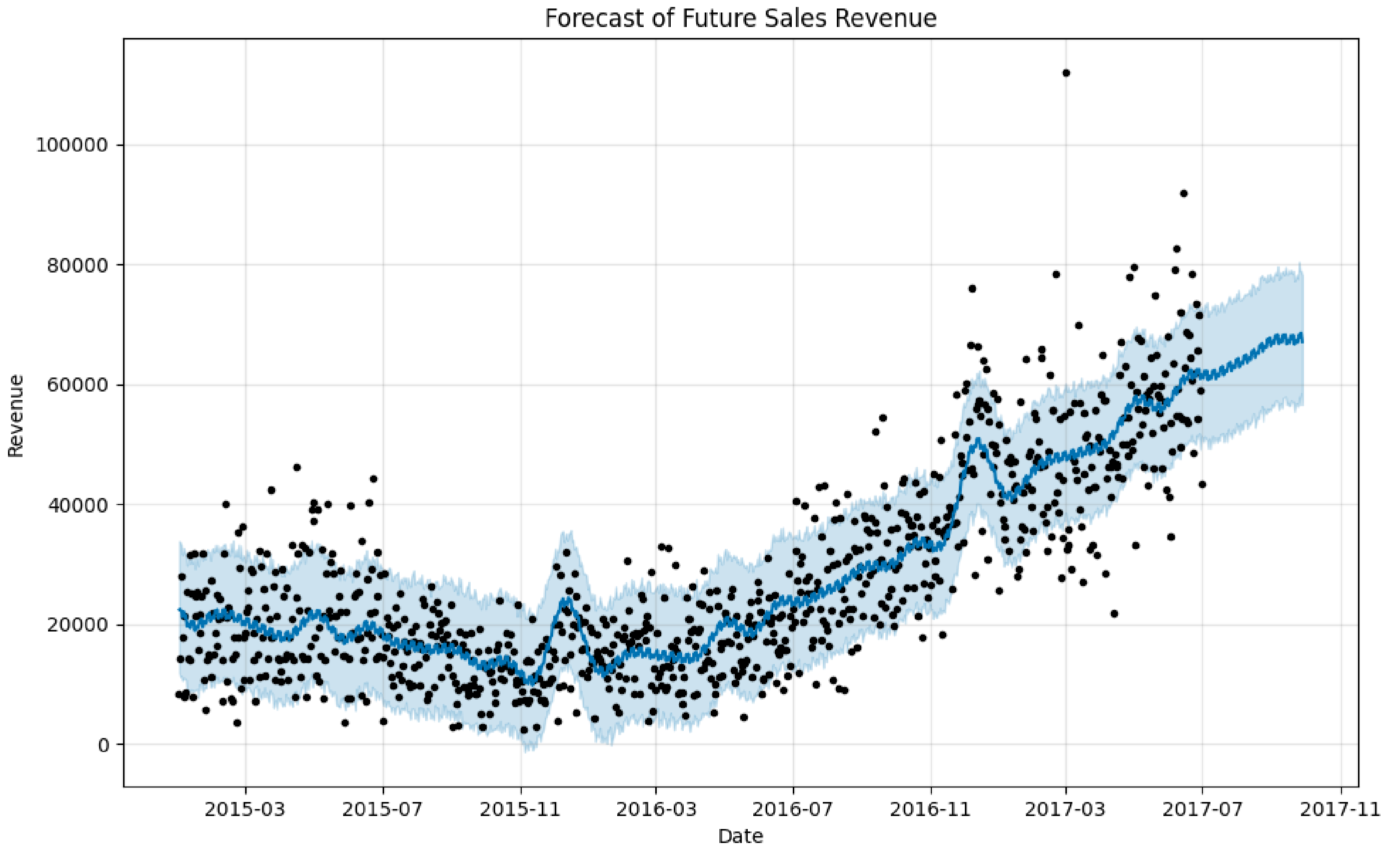
```python
from lifetimes.utils import summary_data_from_transaction_data
from lifetimes import BetaGeoFitter, GammaGammaFitter

summary = summary_data_from_transaction_data(
    transactions=fact_sales,
    customer_id_col='CustomerKey',
    datetime_col='OrderDate',
    monetary_value_col='Revenue',
    observation_period_end=fact_sales['OrderDate'].max()
)

# Filter out customers with frequency 0
summary_filtered = summary[summary['frequency'] > 0]

# BG/NBD model
bgf = BetaGeoFitter(penalizer_coef=0.0)
bgf.fit(summary_filtered['frequency'], summary_filtered['recency'],
summary_filtered['T'])

summary_filtered['predicted_purchases_90'] = bgf.\
    conditional_expected_number_of_purchases_up_to_time(
        90,
        summary_filtered['frequency'],
        summary_filtered['recency'],
        summary_filtered['T']
    )

# Gamma-Gamma for monetary value
ggf = GammaGammaFitter(penalizer_coef=0.0)
ggf.fit(summary_filtered['frequency'], summary_filtered['monetary_value'])

summary_filtered['expected_average_profit'] = ggf.\
    conditional_expected_average_profit(
        summary_filtered['frequency'],
```

```
        summary_filtered['monetary_value']
    )

# CLV over 12 months
summary_filtered['CLV'] = ggf.customer_lifetime_value(
    bgf,
    summary_filtered['frequency'],
    summary_filtered['recency'],
    summary_filtered['T'],
    summary_filtered['monetary_value'],
    time=12,
    freq='D',
    discount_rate=0.01
)

print(summary_filtered[['predicted_purchases_90', 'expected_average_profit',
'CLV']].head())
```

Interpretation: - **`predicted_purchases_90`**: The expected number of purchases in the next 90 days.
- **`expected_average_profit`**: The average revenue (profit) per transaction.
- **`CLV`**: The total present value of expected future revenue from each customer over a 12-month horizon.

Output:

```
10:57:29 - cmdstanpy - INFO - Chain [1] start processing
10:57:29 - cmdstanpy - INFO - Chain [1] done processing
```




```
             predicted_purchases_90  expected_average_profit         CLV
CustomerKey
11000                      0.102820              1250.438714  452.751945
11001                      0.154021              1152.321324  625.015258
11002                      0.087224              1244.229811  382.296050
11003                      0.104205              1245.937232  457.305986
11004                      0.102425              1244.229811  448.740106
```

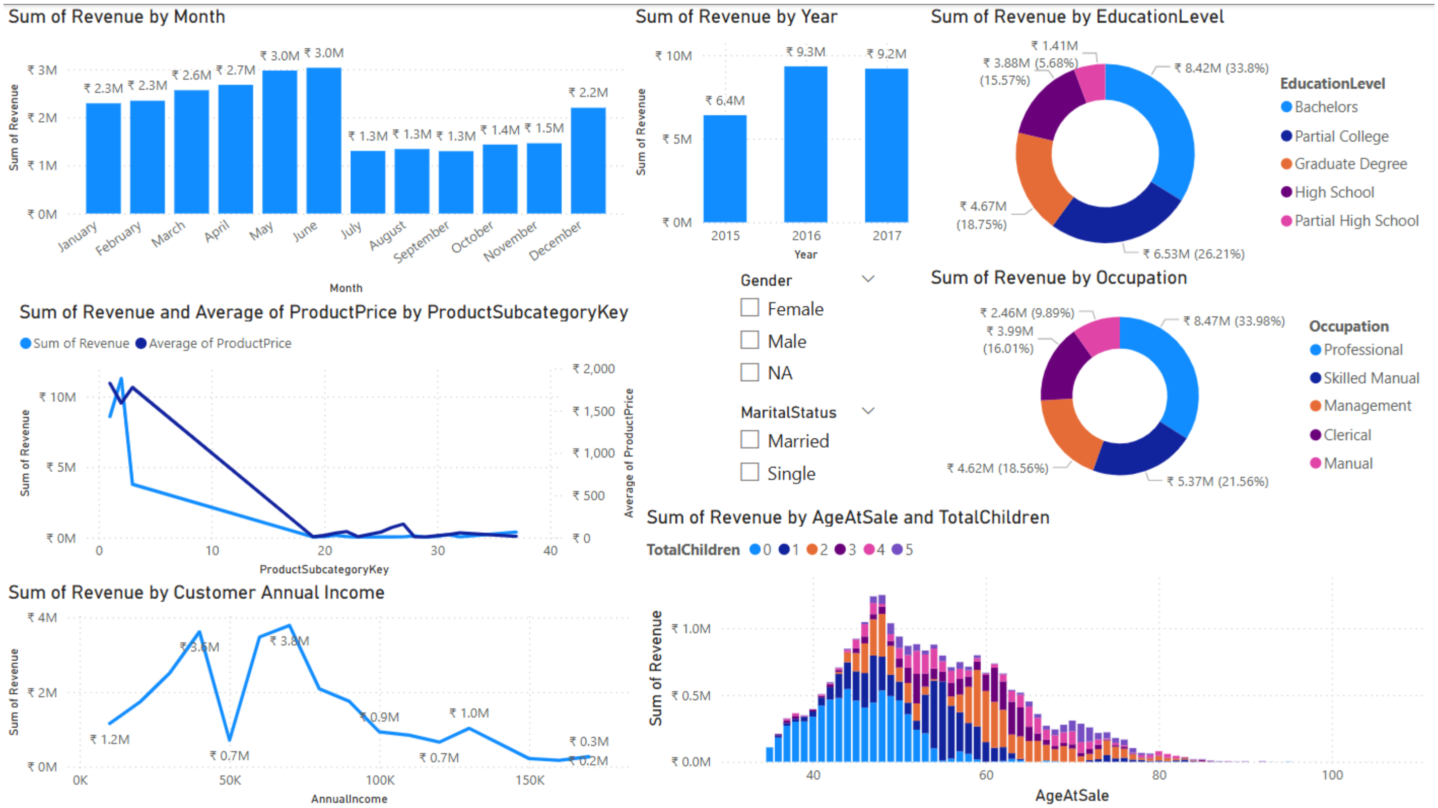
7. PowerBi Report

DAX –

```
AgeAtSale =
DATEDIFF(
    RELATED('case5 dim_customer'[BirthDate]),
    'case5 fact_sales'[OrderDate],
    YEAR
)
```

Renames:

M -> Male, F -> Female, S->Single, M-> Married, etc.



Conclusion

Education Level: Customers with higher education (Bachelor's or Graduate Degree) tend to generate more revenue than other groups.

Occupation: Occupations like "Professional" rank at the top of total revenue.

AgeAtSale & TotalChildren: A substantial portion of revenue comes from customers in their **40s–50s** who have **0 children**.

High-Revenue Subcategories: One or two product subcategories drive the majority of revenue, often with a balanced price–demand ratio.

Peak Months: Revenue increase steadity till June and then drops abruptly from June to July.

Year-over-Year Growth: Revenue increased from 2015 to 2016 but stayed almost same from 2016 - 2017