## 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):

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)

1. **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')

1. **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.")

1. **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')

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

* sales\_df = pd.concat([sales\_2015\_df, sales\_2016\_df, sales\_2017\_df], ignore\_index=True)

1. **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')

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

* sales\_df['CompositeKey'] = (  
   sales\_df['OrderNumber'].astype(str) + "\_" +  
   sales\_df['OrderLineItem'].astype(str)  
  )

1. **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.")

1. **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

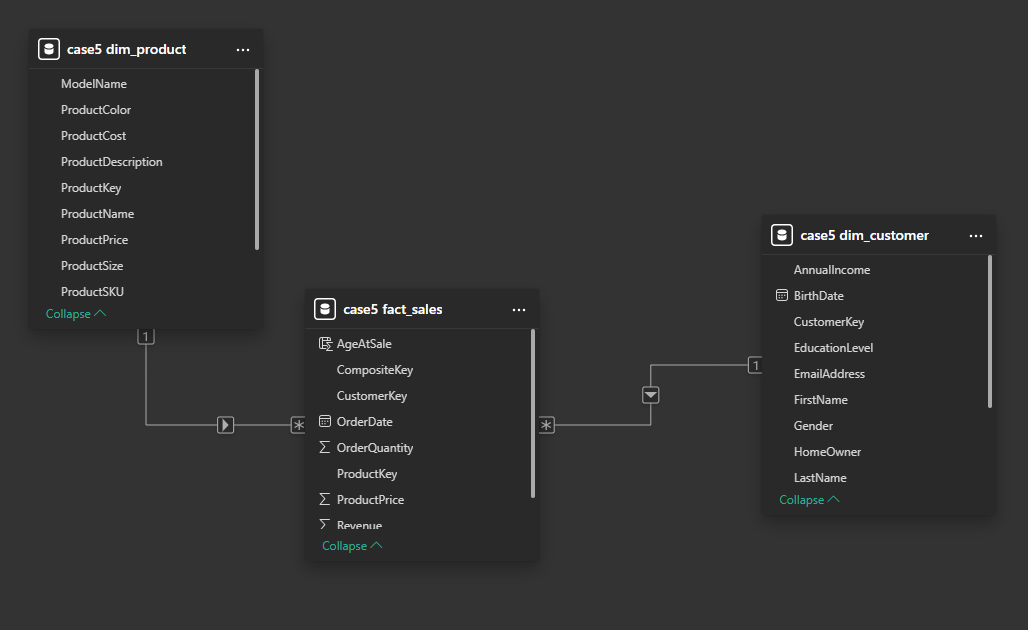
I used **SQLAlchemy** to connect and write the tables to MySQL:

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!")

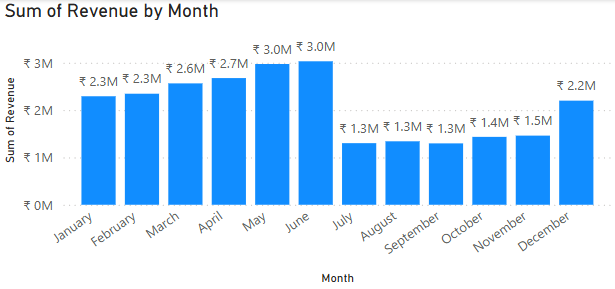
At this point, I had **three tables** in MySQL:  
- **dim\_customer**  
- **dim\_product**  
- **fact\_sales**

### 5. SQL Queries for Analysis

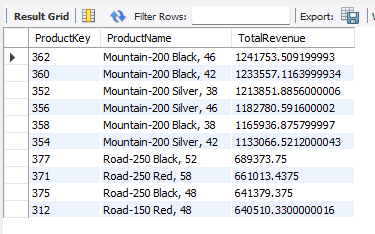
**Schema**

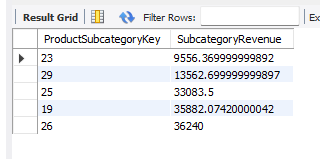


**Sales trends across months**



**Best-performing product categories**  
-- 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;

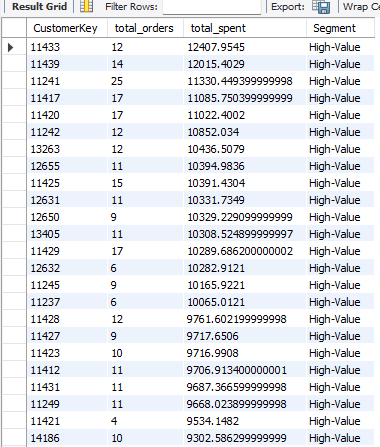
  
  
-- Bottom 5 product subcategories by total revenue  
SELECT   
 p.ProductSubcategoryKey,  
 SUM(f.Revenue) AS SubcategoryRevenue  
FROM fact\_sales AS f  
JOIN dim\_product AS p   
 ON f.ProductKey = p.ProductKey  
GROUP BY p.ProductSubcategoryKey  
ORDER BY SubcategoryRevenue ASC  
LIMIT 5;

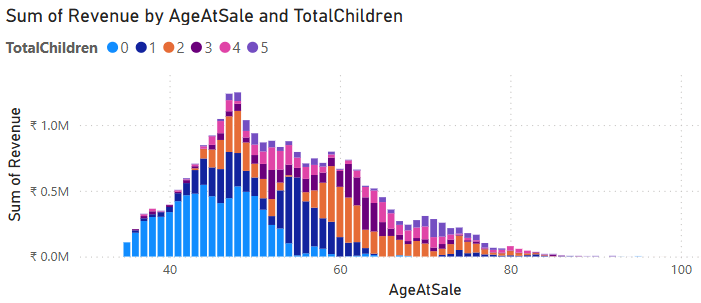
  
  
**Revenue impact of discounts and promotions:**

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;





### 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()  
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()

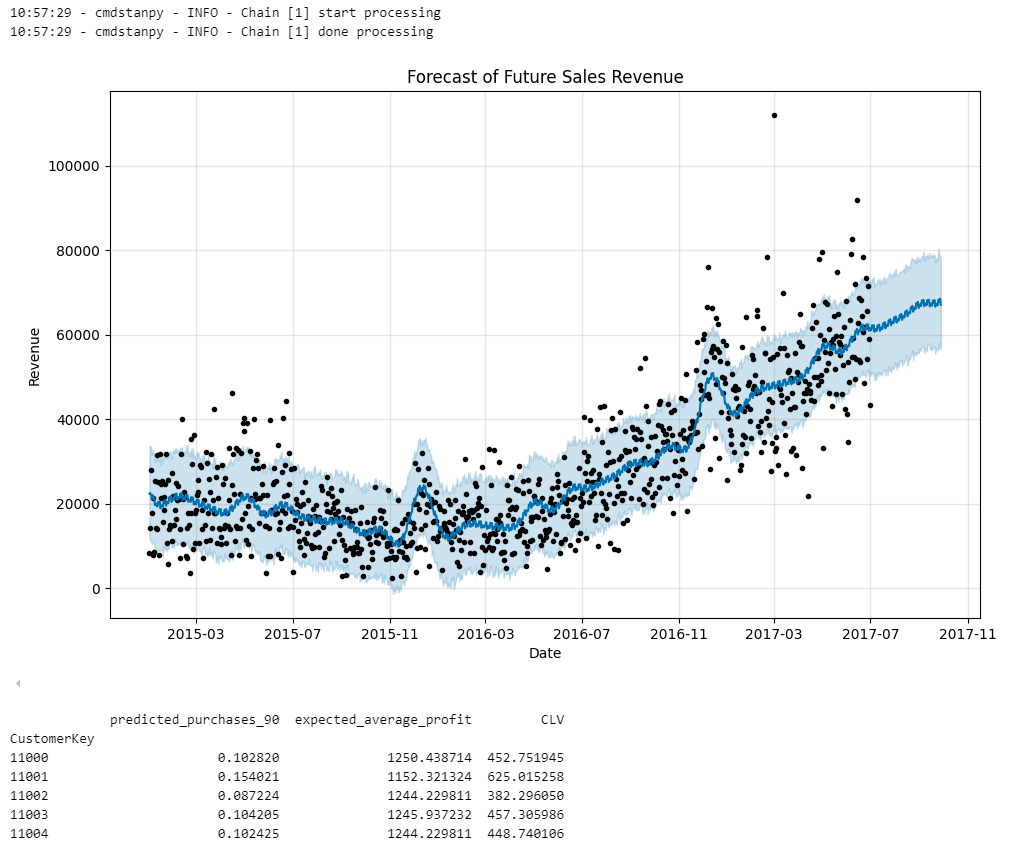
Prophet produced a forecast line (blue) and confidence intervals (shaded region) for the next 90 days. This helps me anticipate **future revenue trends** and manage inventory.

#### 6.2 Predict Customer Lifetime Value (CLV)

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

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:**



### 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