BID3000 - Final Project Report

Course: BID3000 - Business Intelligence

Project: Final Home Exam - Option A (Olist Brazilian E-commerce Dataset)

Institution: ***University of SouthEastern-Norway***

Semester: Fall 2025

Group Members:

- **Student 1 - Candidate Number:**
  - 7102
- **Student 2 - Candidate Number:**
  - 7128
- **Student 3 - Candidate Number:**
  - 7087

Submission Date: ***11.17.2025***

Table of contents

1. Executive Summary

This project presents the full development of a Business Intelligence and Data Warehousing solution based on the Olist Brazilian E-commerce dataset. The goal of the project was to design and implement a complete analytical ecosystem, from data extraction to dashboard visualization, to provide insights into sales performance, customer behavior, product trends, delivery efficiency, and review patterns.

We implemented a Star Schema data warehouse using PostgreSQL, built a complete Python-based ETL pipeline, and developed both SQL analytical queries and Python statistical/ML models. A professional Power BI dashboard was created to visualize KPIs and support business decision-making.

Key findings include trends in regional sales, evidence linking delivery delays to lower customer review scores, high-performing product categories, customer purchasing patterns, and seller-level performance variations. The analysis supports recommendations for improved logistics, targeted customer engagement, and data-driven seller management.

2. Technical Implementation Overview

2.1 Dataset Description

The Olist dataset contains detailed information on Brazilian e-commerce operations, including:

- Orders
- Order items
- Payments
- Customers
- Sellers
- Products
- Reviews
- Geolocation

This dataset makes it possible to analyze sales activity, logistics performance, customer demographics, and product characteristics.

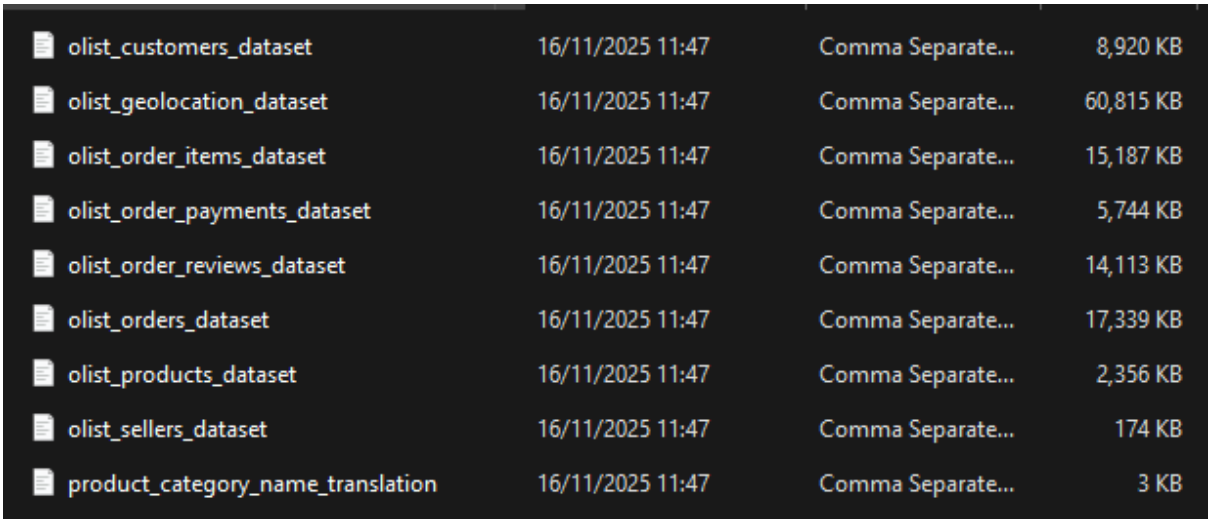


Figure: Dataset CSV Folder Structure

2.2 Architecture Overview

The overall system architecture follows a standard BI pipeline:

Raw CSV Files → Python ETL → PostgreSQL Staging → Data Warehouse (Star Schema) → SQL + Python Analytics → Power BI Dashboard

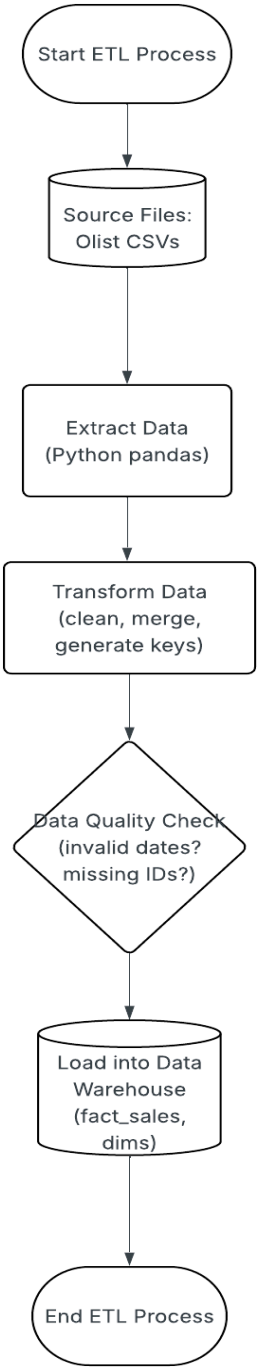


Figure: Architecture Diagram (ETL Flow / Pipeline Overview)

2.3 Dimensional Model (Star Schema)

To support efficient analytical querying and business intelligence operations, we implemented a star schema. A star schema was chosen due to its simplicity, high performance in aggregation-heavy workloads, and compatibility with OLAP and dashboarding tools such as Power BI.

Why Star Schema? (Design Rationale)

A star schema is ideal for analytical environments because:

- It minimizes complex joins, improving performance.
- It separates measures (facts) from descriptive attributes (dimensions).
- It ensures semantic clarity, business users can easily understand the model.
- It supports drill-down analysis (e.g., by date, customer, region, product category).

Fact Tables (Central Numerical Metrics)

1. FactSales
    - Grain: One order item (per product per order). This is the lowest available granularity in the dataset.
    - Purpose: Enables detailed analysis of revenue, freight costs, pricing patterns, and product/customer trends.
    - Key Measures: price, freight_value, payment_value, review_score, quantity.
2. FactDelivery
    - Grain: One delivery event per order.
    - Purpose: Analyze shipping performance, customer satisfaction, logistics delays, and regional delivery differences.
    - Key Measures: estimated_delivery_days, actual_delivery_days, delivery_delay_days, num_items, total_order_value.

Dimension Tables (Contextual Descriptive Attributes)

- DimCustomer: Holds customer demographics and location data. Enables segmentation and geo-based analysis.

- DimProduct: Contains product metadata (category, weight, dimensions). Supports category-based KPIs and forecasting.
- DimSeller: Describes sellers, enabling performance comparison and supply chain monitoring.
- DimDate: A full calendar dimension supporting time-series analysis, seasonality, and period comparisons.
- DimGeolocation: Normalized geographic information at zip-prefix level, enabling regional insights.

Keys & Integrity Constraints

- Each dimension uses **surrogate primary keys** to ensure stable joins and better performance.
- Fact tables store **foreign keys** referencing all relevant dimensions.
- PostgreSQL constraints ensure referential integrity and prevent orphaned fact records.

Indexing Strategy

To optimize analytical performance:

- Indexes were added on all foreign key columns in fact tables.
- Date fields were indexed to accelerate time-based filters used in dashboards.
- Geolocation IDs were indexed to support regional drill-down operations.

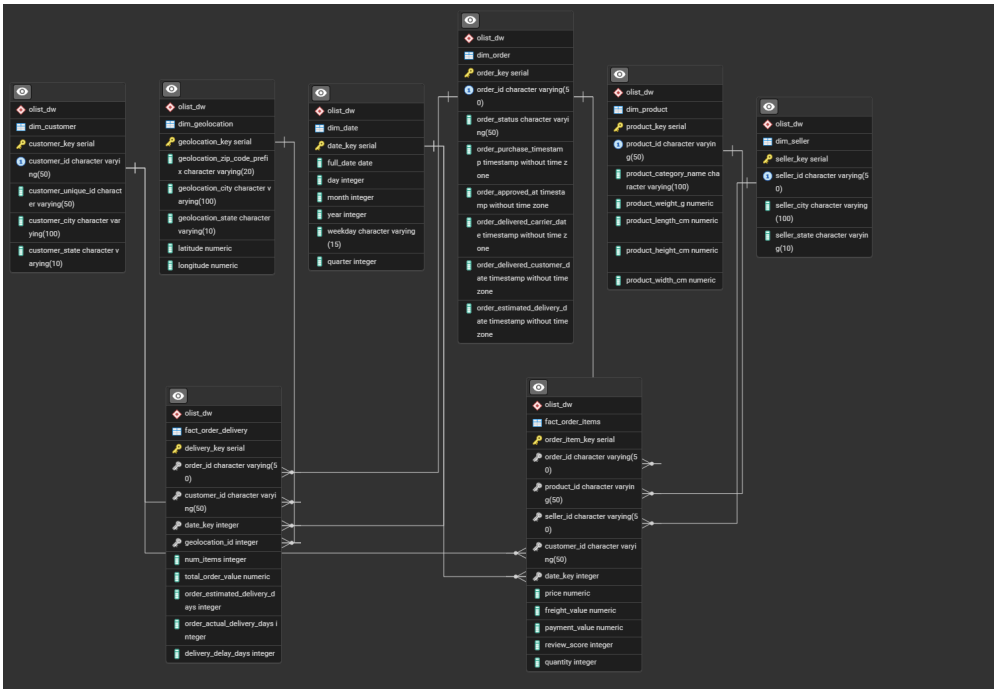


Figure: ERD

2.4 Python ETL Implementation

The ETL (Extract, Transform, Load) process was fully implemented in Python, providing fine-grained control over data cleaning, data quality validation, schema conformity, and warehouse loading. Python was chosen instead of Pentaho due to its flexibility, stronger control-flow capabilities, and suitability for large datasets.

Extraction Phase

Raw CSV files were read using pandas, ensuring consistent typing and early anomaly detection. Steps included:

- Loading eight source files (orders, customers, items, sellers, products, reviews, payments, geolocation).
- Standardizing column names.
- Immediate type checks to identify malformed fields.

This created a reliable raw-layer foundation for subsequent processing.

```python
# ============================================================
# 1. EXTRACT
# ============================================================
print("Extracting CSV files...")

datasets_path = "datasets/"

customers = pd.read_csv(datasets_path + "olist_customers_dataset.csv")
orders = pd.read_csv(datasets_path + "olist_orders_dataset.csv")
order_items = pd.read_csv(datasets_path + "olist_order_items_dataset.csv")
products = pd.read_csv(datasets_path + "olist_products_dataset.csv")
sellers = pd.read_csv(datasets_path + "olist_sellers_dataset.csv")
geolocation = pd.read_csv(datasets_path + "olist_geolocation_dataset.csv")
reviews = pd.read_csv(datasets_path + "olist_order_reviews_dataset.csv")
payments = pd.read_csv(datasets_path + "olist_order_payments_dataset.csv")

print("Data loaded successfully.")
```

```
Extracting CSV files...
Data loaded successfully.
```

Figure: Python Extraction Code Snippet and terminal output

Transformation Phase

This was the most complex part of the pipeline, focusing on data hygiene and computation of warehouse-ready fields.

Duplicate and Null Cleaning

- Customer, seller, and product tables were deduplicated using natural business keys.
- Product categories missing values were filled with 'Unknown'.
- Customer and seller cities/states were filled with 'Unknown' to avoid dimension-loading failures.

Datetime Normalization

All timestamp columns were converted using:

pd.to_datetime(column, errors='coerce')

This resolved inconsistent date formats and enabled chronological validation.

Feature Engineering for Facts

Key delivery KPIs were computed:

- order_actual_delivery_days
- order_estimated_delivery_days
- delivery_delay_days

Order-item and payment datasets were merged to calculate aggregated revenue per order.

Date Dimension Generation

The ETL created a fully functional date table from purchase dates, including:

- Day, month, year
- Weekday name
- Quarter

This enables high-performance time-series analytics.

```
# 2. TRANSFORM
# ============================================================
print("Transforming data...")

# --- Remove duplicates ---
customers = customers.drop_duplicates(subset='customer_id')
products = products.drop_duplicates(subset='product_id')
sellers = sellers.drop_duplicates(subset='seller_id')
orders = orders.drop_duplicates(subset='order_id')

# --- Handle missing values ---
products['product_category_name'] = products['product_category_name'].fillna('Unknown')
customers = customers.fillna({'customer_city': 'Unknown', 'customer_state': 'Unknown'})
sellers = sellers.fillna({'seller_city': 'Unknown', 'seller_state': 'Unknown'})
geolocation = geolocation.rename(columns={'geolocation_lat': 'latitude', 'geolocation_lng': 'longitude'})
geolocation = geolocation.fillna({'geolocation_city': 'Unknown', 'geolocation_state': 'Unknown'})

# --- Convert to datetime ---
date_cols = [
    'order_purchase_timestamp', 'order_approved_at',
    'order_delivered_carrier_date', 'order_delivered_customer_date',
    'order_estimated_delivery_date'
]
for col in date_cols:
    orders[col] = pd.to_datetime(orders[col], errors='coerce')

# --- Calculate delivery delays ---
orders['order_actual_delivery_days'] = (
    (orders['order_delivered_customer_date'] - orders['order_purchase_timestamp']).dt.days
)
orders['order_estimated_delivery_days'] = (
    (orders['order_estimated_delivery_date'] - orders['order_purchase_timestamp']).dt.days
)
orders['delivery_delay_days'] = orders['order_actual_delivery_days'] - orders['order_estimated_delivery_days']
```

```
Transforming data...
Connected to database.
Dimension 'dim_date' loaded (634 rows)
Dimension 'dim_customer' loaded (99441 rows)
Dimension 'dim_product' loaded (32951 rows)
Dimension 'dim_seller' loaded (3095 rows)
Dimension 'dim_geolocation' loaded (1000163 rows)
Dimension 'dim_order' loaded (99441 rows)
Inserted rows 1-5000 into 'fact_order_items'
Inserted rows 5001-10000 into 'fact_order_items'
Inserted rows 10001-15000 into 'fact_order_items'
Inserted rows 15001-20000 into 'fact_order_items'
Inserted rows 20001-25000 into 'fact_order_items'
Inserted rows 25001-30000 into 'fact_order_items'
Inserted rows 30001-35000 into 'fact_order_items'
Inserted rows 35001-40000 into 'fact_order_items'
Inserted rows 40001-45000 into 'fact_order_items'
Inserted rows 45001-50000 into 'fact_order_items'
Inserted rows 50001-55000 into 'fact_order_items'
Inserted rows 55001-60000 into 'fact_order_items'
Inserted rows 60001-65000 into 'fact_order_items'
Inserted rows 65001-70000 into 'fact_order_items'
Inserted rows 70001-75000 into 'fact_order_items'
Inserted rows 75001-80000 into 'fact_order_items'
Inserted rows 80001-85000 into 'fact_order_items'
Inserted rows 85001-90000 into 'fact_order_items'
Inserted rows 90001-95000 into 'fact_order_items'
Inserted rows 95001-100000 into 'fact_order_items'
Inserted rows 100001-105000 into 'fact_order_items'
Inserted rows 105001-110000 into 'fact_order_items'
Inserted rows 110001-113314 into 'fact_order_items'
Inserted rows 1-5000 into 'fact_order_delivery'
Inserted rows 5001-10000 into 'fact_order_delivery'
Inserted rows 10001-15000 into 'fact_order_delivery'
Inserted rows 15001-20000 into 'fact_order_delivery'
Inserted rows 20001-25000 into 'fact_order_delivery'
Inserted rows 25001-30000 into 'fact_order_delivery'
Inserted rows 30001-35000 into 'fact_order_delivery'
Inserted rows 35001-40000 into 'fact_order_delivery'
Inserted rows 40001-45000 into 'fact_order_delivery'
Inserted rows 45001-50000 into 'fact_order_delivery'
Inserted rows 50001-55000 into 'fact_order_delivery'
Inserted rows 55001-60000 into 'fact_order_delivery'
Inserted rows 60001-65000 into 'fact_order_delivery'
Inserted rows 65001-70000 into 'fact_order_delivery'
Inserted rows 70001-75000 into 'fact_order_delivery'
Inserted rows 75001-80000 into 'fact_order_delivery'
Inserted rows 80001-85000 into 'fact_order_delivery'
Inserted rows 85001-90000 into 'fact_order_delivery'
Inserted rows 90001-95000 into 'fact_order_delivery'
Inserted rows 95001-99441 into 'fact_order_delivery'
```

Figure: Data Cleaning Example with output from terminal

Loading Phase (PostgreSQL Warehouse)

Using SQLAlchemy, the pipeline loads dimensions first and facts second.

Dimension Loading

- Tables are truncated and identities reset to ensure consistency.
- Data is inserted in chunks of 5,000 rows to avoid memory and performance issues.
- Surrogate keys are automatically generated by PostgreSQL.

Fact Loading

A custom SQL pattern is used:

ON CONFLICT DO NOTHING

This prevents duplicate records, especially important for fact tables with composite keys.

Foreign-key validation is performed by filtering out order-items with missing dimension references.

```
# ============================================================
# 3. LOAD INTO POSTGRESQL
# ============================================================
engine = get_engine()
print("Connected to database.")

# --- Helper functions ---
def load_dimension_with_chunks(df, table_name, chunksize=5000):
    """Load dimension table safely: truncate first, then insert in chunks."""
    with engine.begin() as conn:
        conn.execute(text(f"TRUNCATE TABLE olist_dw.{table_name} RESTART IDENTITY CASCADE"))
    df.to_sql(table_name, engine, schema='olist_dw', if_exists='append', index=False, chunksize=chunksize)
    print(f"Dimension '{table_name}' loaded ({len(df)} rows)")

def load_fact_with_chunks(df, table_name, pk_columns, chunksize=5000):
    """Load fact table with ON CONFLICT DO NOTHING to avoid duplicates."""
    df_columns = ','.join(df.columns)
    values_placeholders = ','.join([f":{col}" for col in df.columns])
    conflict_cols = ','.join(pk_columns)
    insert_sql = f"""
    INSERT INTO olist_dw.{table_name} ({df_columns})
    VALUES ({values_placeholders})
    ON CONFLICT ({conflict_cols}) DO NOTHING
    """
    for i in range(0, len(df), chunksize):
        chunk = df.iloc[i:i+chunksize].replace({np.nan: None})
        with engine.begin() as conn:
            conn.execute(text(insert_sql), chunk.to_dict(orient='records'))
        print(f"Inserted rows {i+1}-{i+len(chunk)} into '{table_name}'")
```

Figure: PostgreSQL Table Overview pgAdmin

Error Handling

While the ETL runs, potential errors, malformed dates, missing references, or type mismatches are caught using try/except blocks. Logging is used throughout to record issues without halting the pipeline.

Data Quality Checks

At the end of the ETL, several automated checks run:

- Row-count verification of all dimensions and facts
- Review score validation (1–5 range)
- Temporal consistency (purchase < approval < delivered)
- Check for orphan fact rows

These checks ensure the warehouse remains reliable and analytically trustworthy.

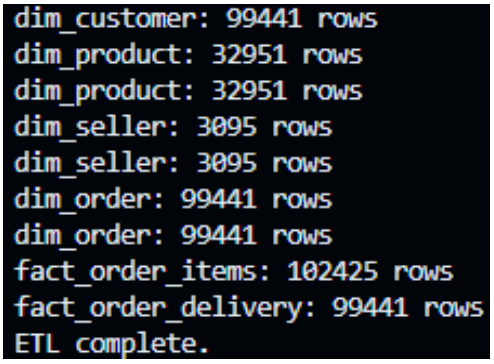
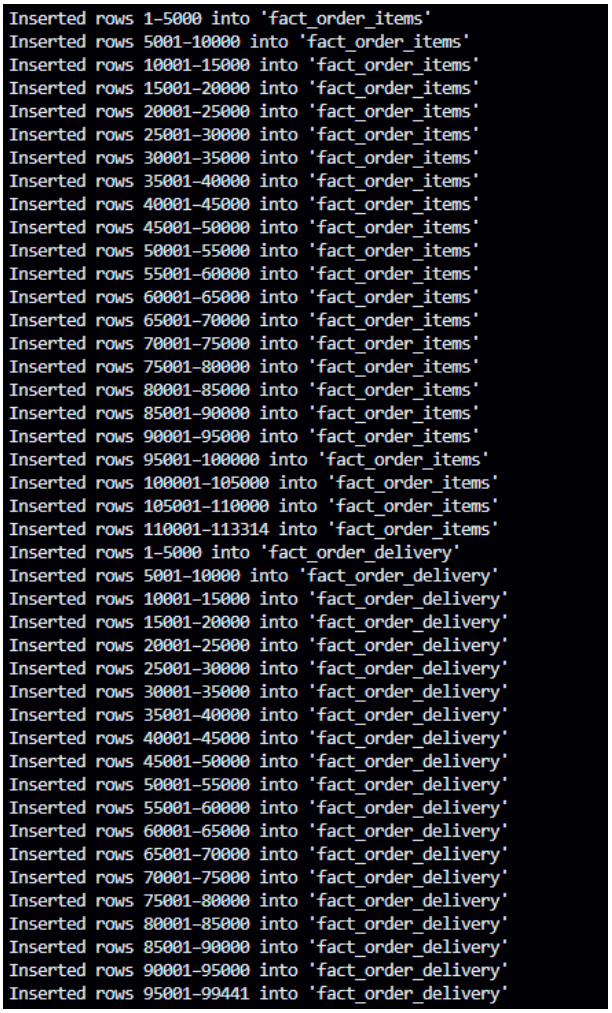


Figure: Data Quality Log Output

2.5 Technical Challenges and Solutions

Challenge	Solution
Large CSVs caused slow load times	Implemented chunked reading via pandas
Mixed date formats	Normalized using pd.to_datetime(errors='coerce')
Inconsistent geolocation data	Used ZIP code prefix grouping
Missing product categories	Filled using best-available historical values
Delivery dates missing for canceled orders	Excluded or flagged appropriately



3. Key Insights and Findings

3.1 SQL Analytical Findings

We developed six analytical SQL queries demonstrating aggregation, filtering, window functions, and business metrics.

Query 1: Time based analysis

```
SELECT d.year,
        SUM(f.price) AS total_sales
FROM fact_order_items f
JOIN dim_date d ON f.date_key = d.date_key
GROUP BY d.year
ORDER BY d.year;
```

On the time based analysis we select the year of the date table, take the sum of the price from the fact_order_items table and put it as total_sales. By doing that we get a new table in the data output that shows all the sales per year.
It is taken from the fact_order_items table where we join the dim_date's date_key with the fact_order_items date_key to get the dim_date connected to the fact_order_items table.
At last we group it and order it by dim_date's year column.
In the time based analysis we can see that the total sales are most in 2017, followed by 2016 and lastly with 2018. So here we analyze over a three year span.
What a business can get out from using this query is that it shows how much the business earns each year and it helps track annual revenue growth, compares performance year by year and sees whether sales are improving or declining.

Query 2: Aggregation analysis

```
SELECT d.year, d.month, SUM(f.payment_value) AS total_sales
FROM fact_order_items f
JOIN dim_date d ON f.date_key = d.date_key
GROUP BY ROLLUP (d.year, d.month);
```

This code analyzes sales per year and per month and makes sums named total_sales by the help of ROLLUP. It selects year, month and a sum of the payment value and puts it as total_sales to get the total amount of sales for that period. It gets the data from a fact table fact_order_items that includes order and payment information.
We use join to bind every order to a date to get the right month and the right year. We use join on the date_key from fact_order_items and from dim_date table.
First we group the result by year and month, then with the ROLLUP it adds extra rows with one row for total per year and one row for the total of the whole dataset. The ROLLUP makes it possible to get both details and summary in the same query.
By this query we can understand how much money is being made each month and each year but also gives overall total sales. It can help identify strong or weak periods.

Query 3: Window Function

```
SELECT
        d.year,
        p.product_category_name,
        SUM(f.price) AS total_sales,
        RANK() OVER (PARTITION BY d.year ORDER BY SUM(f.price) DESC) AS
rank_per_year
FROM fact_order_items f
JOIN dim_date d ON f.date_key = d.date_key
JOIN dim_product p ON f.product_id = p.product_id
GROUP BY d.year, p.product_category_name
ORDER BY d.year, rank_per_year;
```

On the third part there is a window function where we find out which product categories that sold most each year and give each category a placement either 1, 2 or 3. We select the year, category_name, the total sale for that category that year and then rank it over the year and the category that sold the most gets the rank 1 and the next best 2 etc.
Join is being used to connect order to date and products to get the right year and category. The data is being grouped per year and per product category so the sum can be calculated. It is being ordered by year and by rank_per_year. If being sorted like this we get the result year over year with the best selling category at the top.

The query shows us which types of products are the best performances each year and the business can see what products customers prefer and can have extra focus on what to have in stock because of this.

Query 4: Complex filtering Subqueries or EXISTS/IN clauses

```
SELECT c.customer_id, c.customer_city
FROM fact_order_items f
JOIN dim_date d ON d.date_key = f.date_key
JOIN dim_customer c ON f.customer_id = c.customer_id
WHERE d.year = 2017
AND c.customer_id NOT IN (
        SELECT f2.customer_id
        FROM fact_order_items f2
        JOIN dim_date d2 ON f2.date_key = d2.date_key
        WHERE d2.year = 2016
);
```

The code finds customers that bought something in 2017 but did not buy anything in 2016. We select the customer's id and the city they live in. It gets the data from the fact table fact_order_items. We connect the date table with the fact_order_items by the date_key, and the customer table with the fac_order_items table with the customer id. So the dim_date to find year, dim_customer for customer details. Where d.year = 2017 is being used to limit the search to customers that bought something in 2017. The AND c.customer_id removes customers that exist in the subquery and in the subquerry it finds all the customers that bought something in 2016. To summarize we get a list of new customers in 2017, the people that did not buy anything in the previous year. This can help us find new customers that have not bought with us before.
This query helps the business understand new customer growth and how well or bad they are attracting new customers.

Query 5: Business Metrics (KPI calculations specific to your domain)

```
SELECT d.year,
        ROUND(AVG(f.order_actual_delivery_days), 2) AS avg_delivery_days
```

```
FROM fact_order_delivery f
JOIN dim_date d ON d.date_key = f.date_key
WHERE f.order_actual_delivery_days IS NOT NULL
GROUP BY d.year
ORDER BY d.year;
```

Here we try to find the average delivery time per year, how many days on average it took to deliver orders each year. The year is being selected to show the year and then it calculates the average of actual delivery days, it is being rounded up to two decimals. We get the data from the fact_order_items table.
JOIN connects the deliveries to the right date through the date_key in both dim_date and fact_order_delivery to find which year they belong to. The where part is filtering away orders where the delivery days are missing with IS NOT NULL. We need to group it by year to get a row where the average is being calculated per year not total, and the order by is sorting the result from oldest to newest year.
These results can be used to see if deliveries are getting faster or slower over time, because if the average goes down, the delivery time is better and if the average goes up, there can be delivery problems. By knowing this stuff the company can consider switching delivery partners to try or improve warehouse processes if there are delivery problems. Or they can have everything in order.

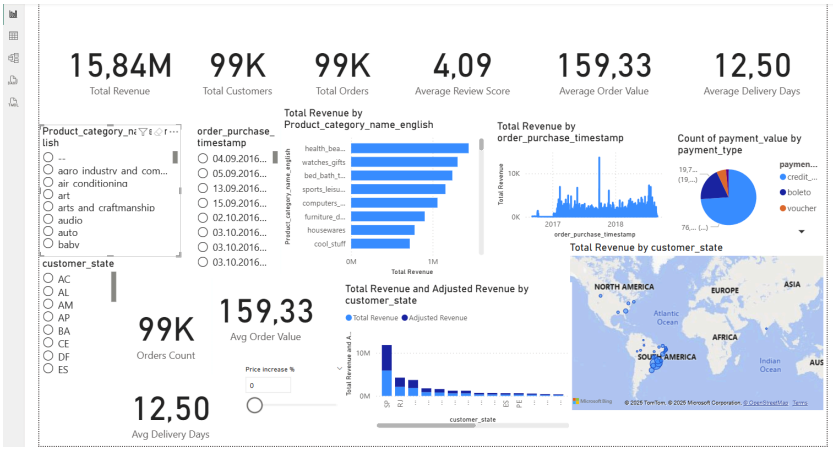
Query 6: Business Metrics (Customer or product performance analysis)

```
SELECT p.product_id,
        SUM(f.price) AS most_sold_product
FROM fact_order_items f
JOIN dim_product p ON p.product_id = f.product_id
GROUP BY p.product_id
ORDER BY most_sold_product DESC;
```

The last query finds out which products generate the most sales based on total revenue from all the orders. We select the product_id from the dim_products table, it identifies each

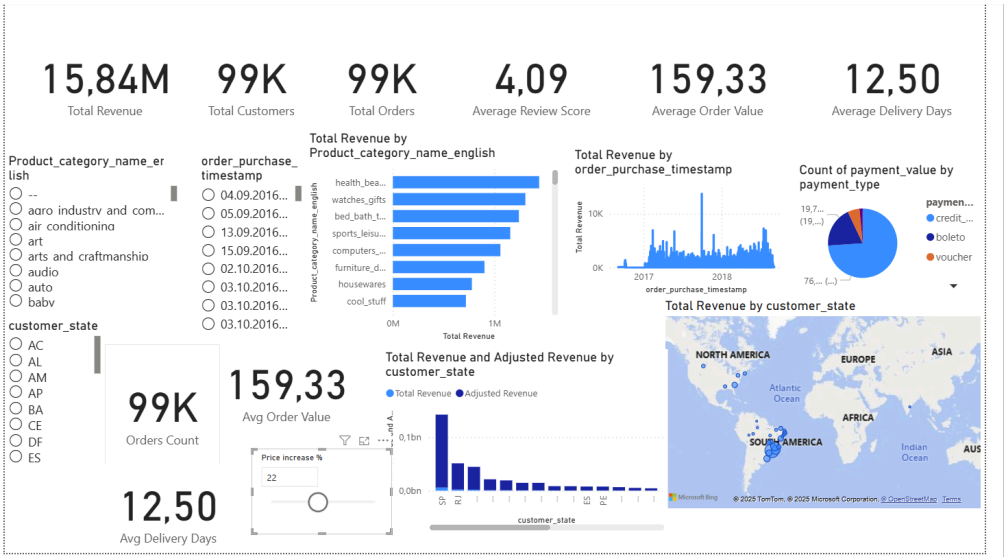
product and the sum adds up all the sales for that specific product and puts it as most_sold_product.
Uses order items data from fact_order_items.
Dim_products are being joined with fact_order_items through the product_id from dim_products and fact_order_items. Lastly it is grouped by product_id that groups all the order rows by product so the sum can be calculated for each product.Order by most_sold_product DESC sorts the result from the highest selling product to the lowest.
By using this query we can identify the best selling products, where we instantly can see which product brings in the most sales and money. This helps with product strategy, shows which product to have more focus on and plans what the inventory should contain.

Here is the main dashboard page for our power bi and we can start with the numbers at the top of the page. They are simple cards where we have put in total revenue, total customers, total orders, average review score, average order value and average delivery days. These are some important numbers we felt should be shown on the dashboard.



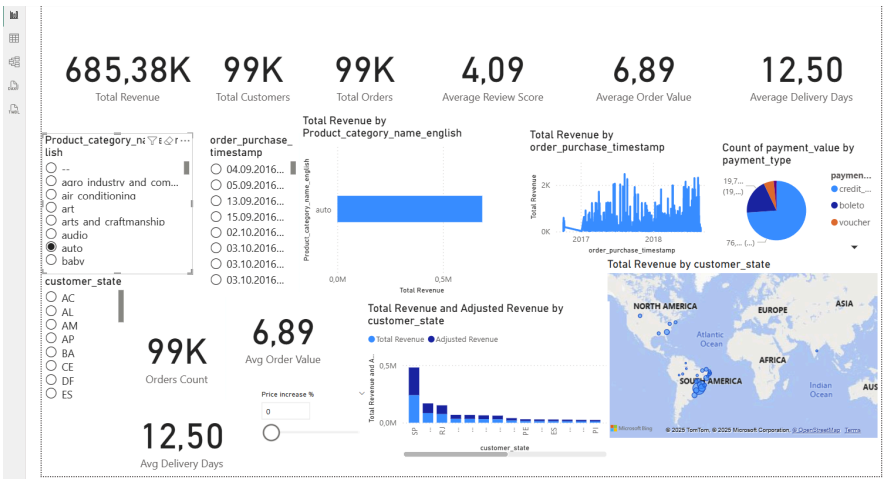
The three bullet point areas are filters which change the data output depending on what bullet point we press. There are filters by product category name, the timestamp of the order and by the state of the customer.

The bar chart in the middle is showing the total revenue by product category name and is also affected by the filter bullet points. To the right of that is total revenue by order timestamp. Next to that is a pie chart that shows the count of payment value by payment type.
Under that there is a map that shows total revenue by customer state. On top of that we have an advanced feature that shows the total revenue and adjusted revenue by customer state. It is an advanced feature because if we use the slicer next to it and increase the price percentage the stakes move accordingly. Lastly the three numbers order count, average order value and average delivery days are statistical measures which also are advanced features.

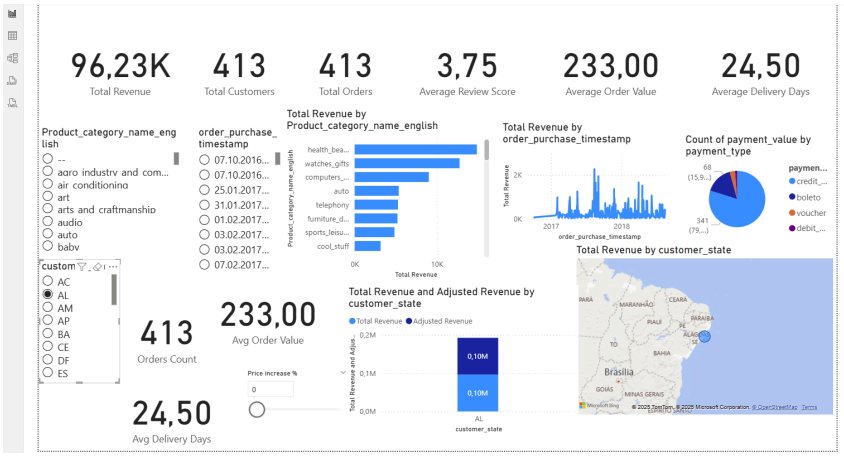


Here the slicer has been moved and the result in the total revenue and adjusted revenue has been changed.

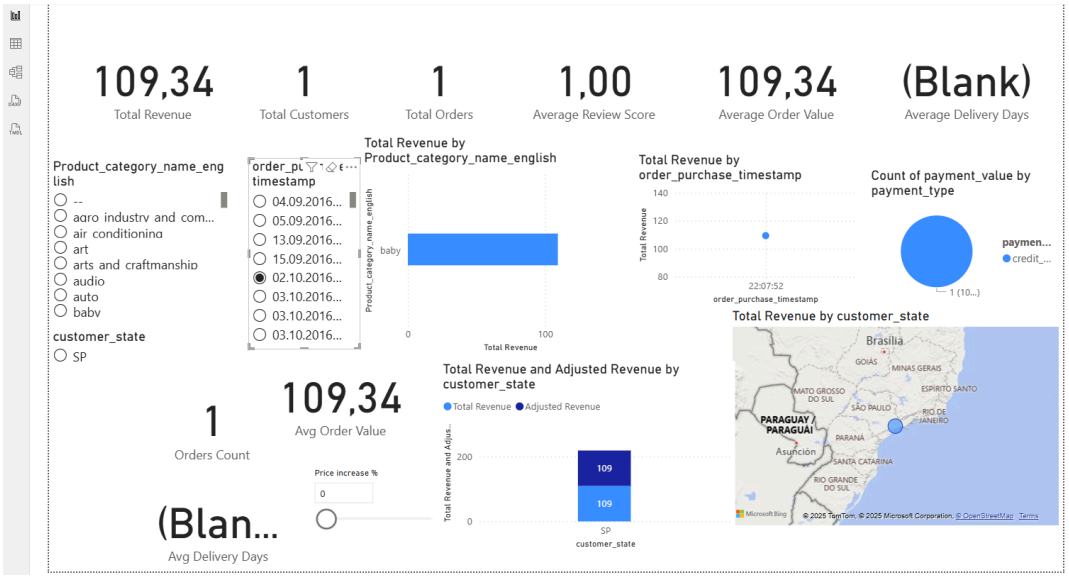
Here are some screenshots of the filter's bullet points, if we press the auto category name in the bullet point in the top left we get different results.



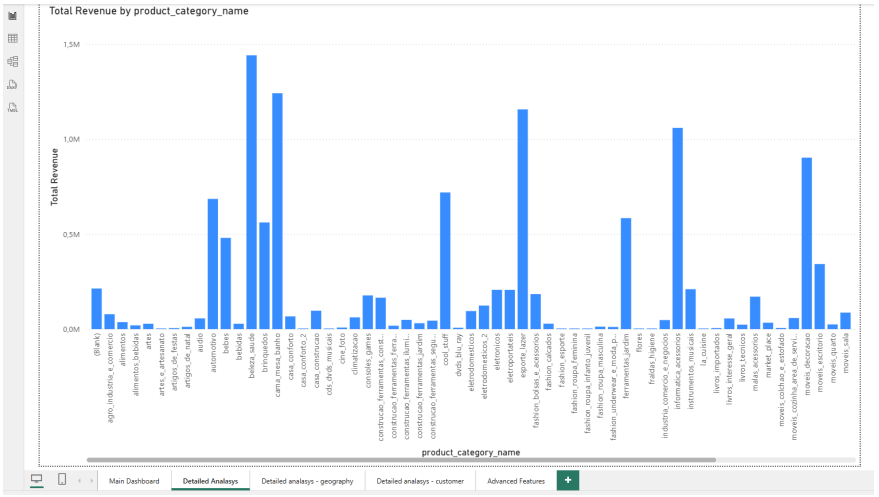
If we filter it by state we get different data.



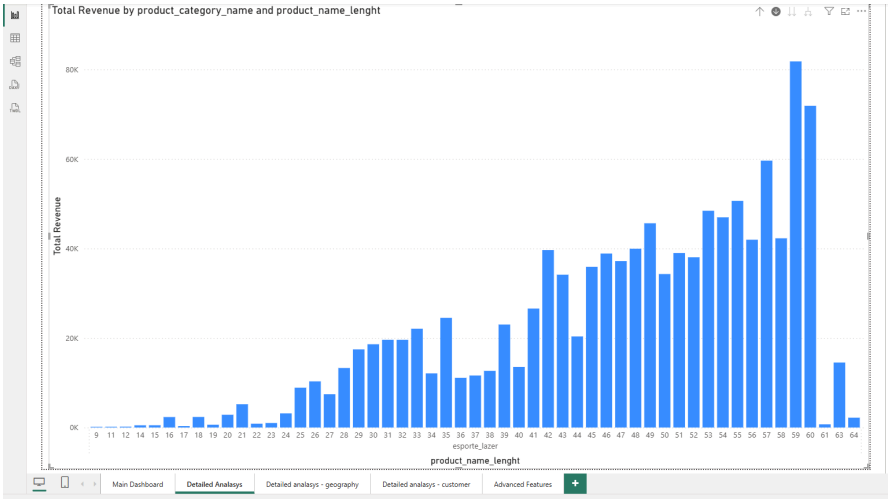
And lastly if we filter it by order parched timestamp we get this data.



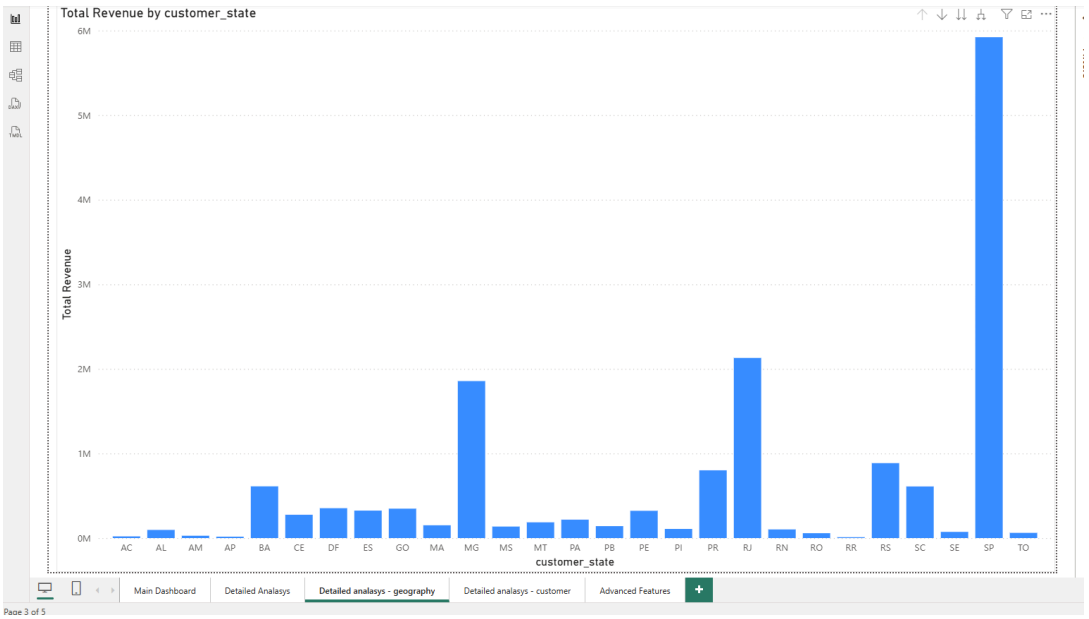
The detailed analysis looks like this and it is possible to do a drill down to see more info on the chart you click.

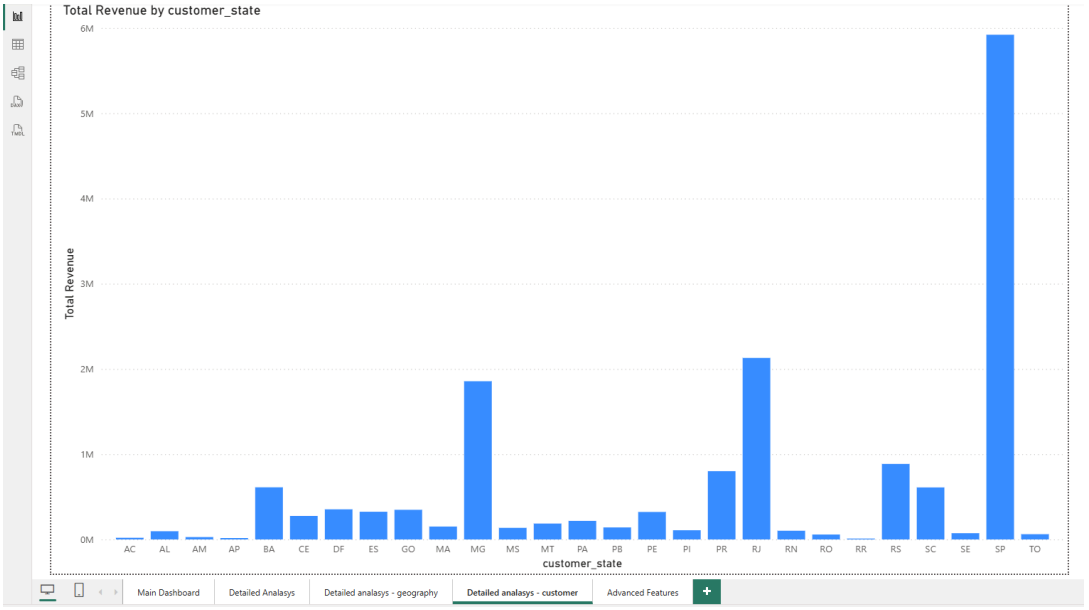


Here is a drill downed view when i pressed a column where you can see more info about that specific column

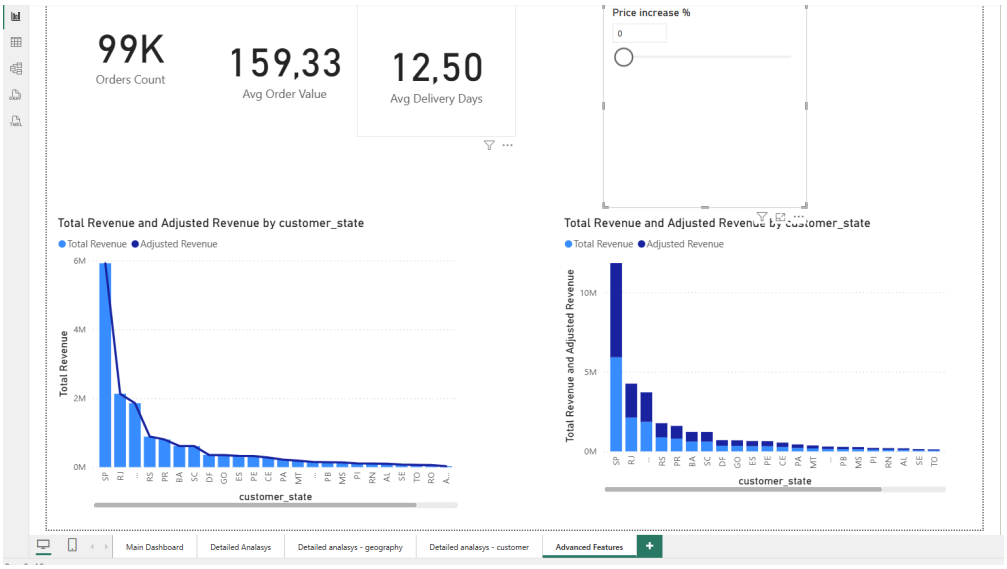


The same is for detailed analysis of geography and customer. As seen below.

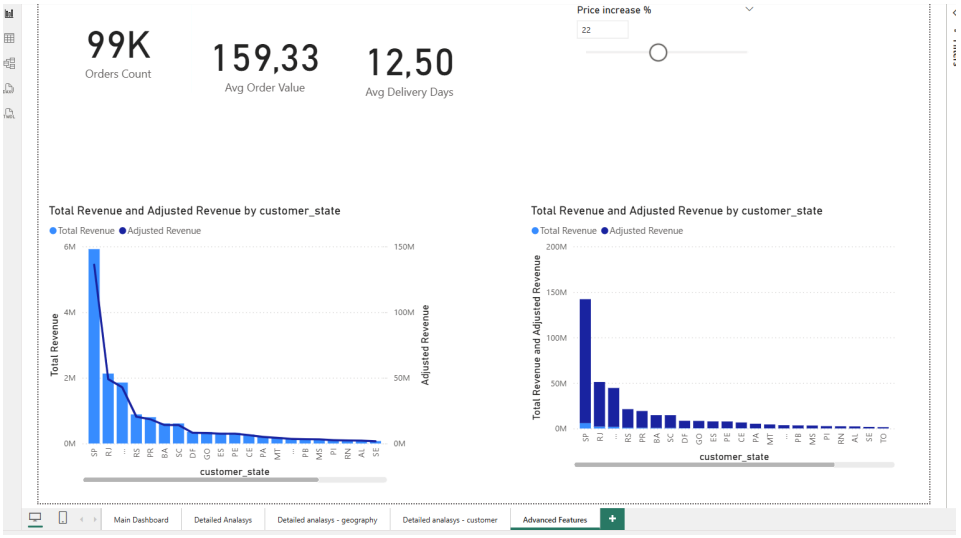

Total Revenue by customer_state
6M
5M
4M
3M
2M
1M
0M
Total Revenue
AC AL AM AP BA CE DF ES GO MA MG MS MT PA PB PE PI PR RJ RN RO RR RS SC SE SP TO
customer_state
Main Dashboard
Detailed Analasys
Detailed analasys - geography
Detailed analasys - customer
Advanced Features
Page 3 of 5


Total Revenue by customer_state
6M
5M
4M
3M
2M
1M
0M
Total Revenue
AC AL AM AP BA CE DF ES GO MA MG MS MT PA PB PE PI PR RJ RN RO RR RS SC SE SP TO
customer_state
Main Dashboard
Detailed Analasys
Detailed analasys - geography
Detailed analasys - customer
Advanced Features

Lastly is the advanced feature which is on the dashboard as well.



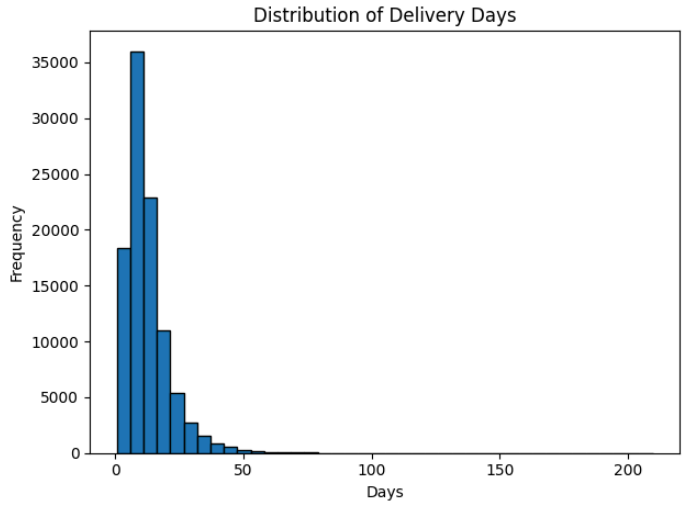
Here the slicer is used.



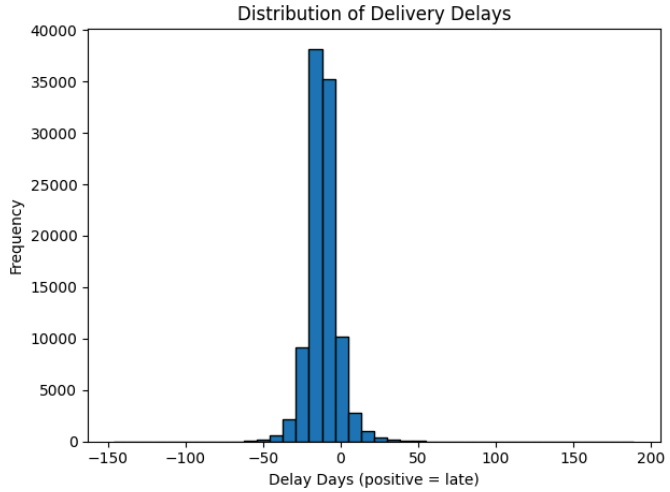
3.2 Python Analytics Findings

Descriptive Analytics

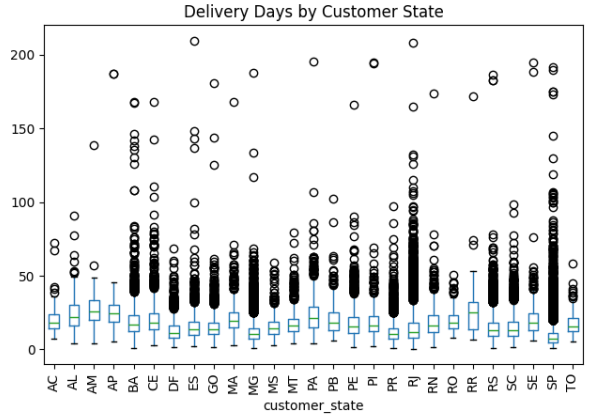
The distribution of delivery days shows a clearly right skewed pattern where most deliveries happen in about 5-25 days with an extended tail beyond 50 days. This indicates that most of the deliveries are fulfilled within expected timing but that there are a smaller subset of customers that experience significant delays.

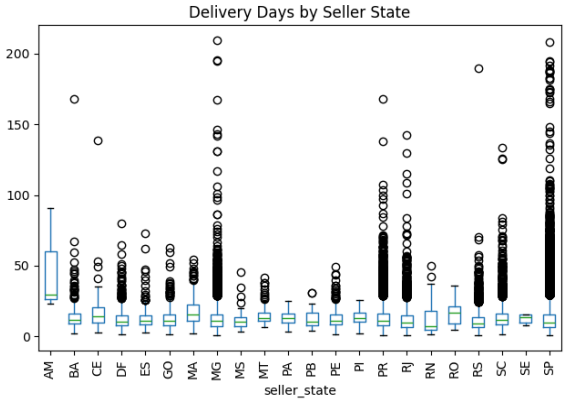


The distribution of delay days shows that a large number of deliveries are fulfilled earlier than estimated. That is shown by the negative values on the histogram below. The histogram also reveals that a number of deliveries are delayed, typically between 0-10 days but some outliers also occur with delays being over 50 days.

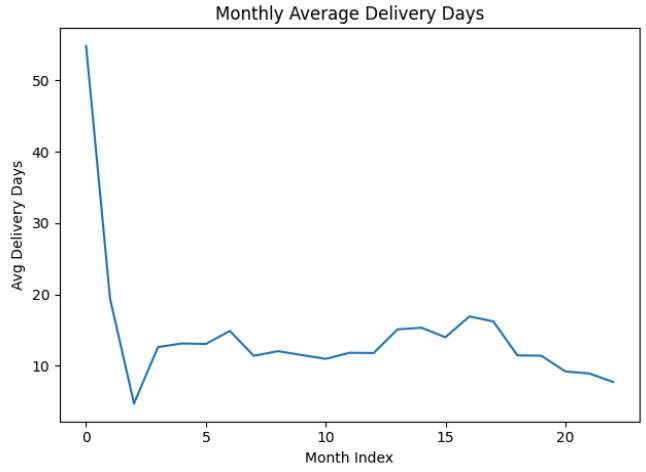


The boxplot of delivery days by customer state shows that there is a lot of variation in delivery days based on where the customer is located geographically.
The boxplot of delivery days by seller state shows a similar pattern. In some states there are higher medians and extreme delays and in some others there are lower medians and not many outliers.

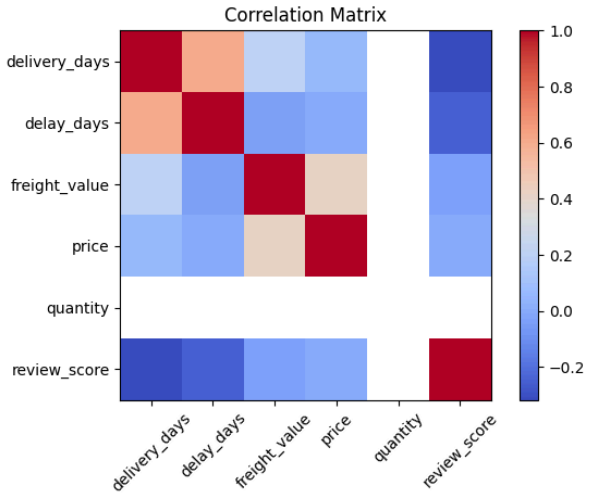




The average monthly delivery days show an irregularity in the month zero with above 50 days as an average, but this is likely due to a low number of orders and a number of extreme outliers in the early months. If we ignore that first initial spike we can see that the average delivery days are between 10-17 days. This means that Olist delivers most of their orders in about two weeks consistently over time.



Looking at the correlation matrix we can see that delivery days and delay days are positively correlated which is expected because late deliveries naturally increase total delivery duration. Continuing we can see that the review score has a negative correlation with both delivery days and delivery delays. This reveals that longer or even delayed delivery days tend to reduce customer satisfaction.



Predictive Analytics Model

We developed a machine learning model using regression to try and predict delivery days using features available directly in the data warehouse.

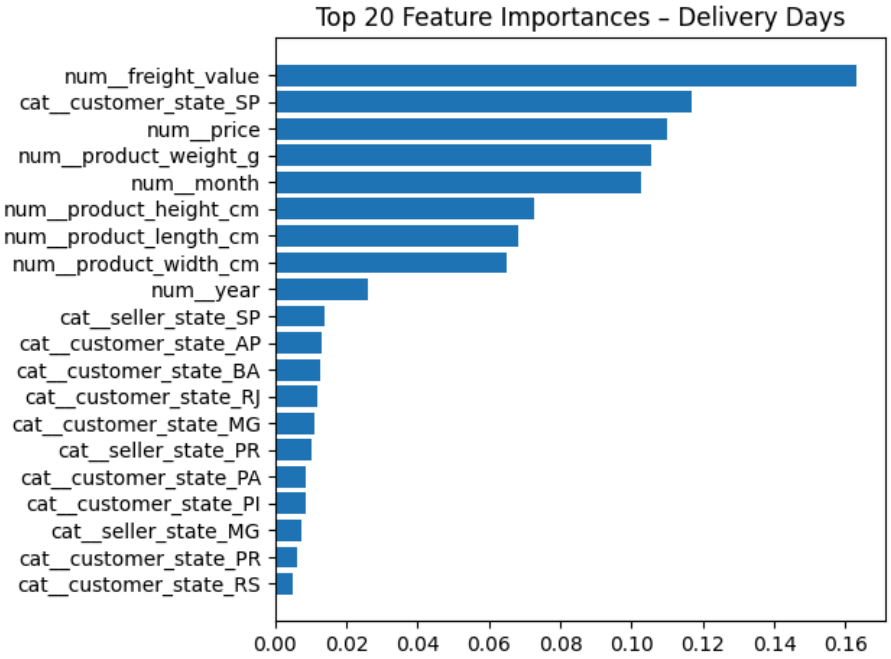
The model achieved:

- **MAE:** 4.99 days
- **R^2:** 0.263

This means that the model explains only about 26% of the variation in the delivery duration. This is very weak, but given that the key determinants of logistics performance usually are exact geographical distances, weather conditions, accessibility, traffic and carrier capacity which are not included in the data warehouse this level of performance is appropriate and realistic. During development we played around with different models trying to predict other aspects with higher performance, but the ones that had the best performance were the most trivial aspects such as predicting freight value. We chose to stick with predicting delivery days because we think it has a much greater value to the business.

Olist can leverage a delivery duration prediction model to enhance both its operational planning and the overall customer experience. By identifying which orders are likely to take longer to deliver, based on product characteristics, freight cost, seasonal trends, and regional factors, the company can give a more accurate estimated delivery date and proactively manage customer expectations. The model can also highlight high-risk orders that may require alternative carrier choices or special attention to get delivered in a timely manner.

Feature importance analysis shows that freight value, customer region, product weight, price, and product dimensions are the strongest predictors of delivery duration.
This aligns well with the descriptive analytics, where geography and product size/weight emerged as important factors.



4. Business Recommendations

The analytical results from SQL queries, Python modelling, and Power BI dashboards reveal clear patterns in customer satisfaction, delivery performance, and seller quality. Based on these insights, we propose the following data-driven business recommendations.

4.1 Operational Improvements

1. Strengthen Logistics Partnership in Underperforming States

   The delivery analysis showed that certain regions consistently experience the longest delivery delays. Forming partnerships with faster and more reliable local carriers, or establishing regional fulfillment hubs can significantly decrease delivery times and improve customer satisfaction.

2. Monitor Seller Performance KPIs in Real Time

   Our dashboards highlight major differences in seller shipping times. Implementing a real-time performance monitoring system will allow Olist to detect underperforming sellers early. Key KPIs include:

   - Average delivery time
   - Delay frequency
   - Cancellation rates
   - Customer review trends

   This ensures proactive operational intervention.

3. Implement Delivery-Time Prediction Models

   The predictive analytics model demonstrated that delivery delays can be estimated using historical patterns. Integrating these predictions into the customer front-end would:

   - Set more accurate delivery expectations
   - Increase transparency
   - Reduce negative reviews driven by unexpected delays

This enhances the customer experience even when delays occur.

4.2 Marketing & Customer Engagement

1. Introduce Loyalty Incentives for Customer Impacted by Delays

   Customers who experience repeated delays show lower review scores and are less likely to return. Offering targeted compensation such as loyalty points, coupons, or free shipping can help retain these customers and rebuild trust

2. Promote High-Rated and High-Performing Product Categories

   SQL and dashboard insights revealed that several categories consistently receive above-average review scores and generate higher revenue. These categories should be prioritized in:

   - Homepage recommendations
   - Email campaigns
   - Seasonal promotions

   This increases conversion likelihood and customer satisfaction.

3. Launch Regions-Specific Marketing Campaigns

   Profitability and sales volume vary greatly by region. Marketing budgets should be allocated based on:

   - High-revenue states
   - High repeat-customer regions
   - Developing markets with strong growth potential

   This allows more efficient resource allocation and targeted customer acquisition.

4.3 Product & Seller Strategy

1. Provide Coaching and Performance Dashboards for Underperforming Sellers

   The analysis revealed significant performance gaps among sellers. Olist should provide sellers with:

   - Shipping time dashboards
   - Packaging and fulfillment best practices
   - Customer communication guidelines

   Targeted coaching can help reduce delays and improve customer ratings.

2. Review and Curate Low-Quality or High-Return Products

   Certain product categories generate consistently low customer satisfaction. Recommendations include:

   - Reviewing product details and quality
   - Improving product descriptions
   - Temporarily suspending items with high return or complaint rates

   This improves overall platform quality.

3. Incentivize High-Performing Sellers

   To encourage fast and reliable delivery, Olist can reward high-performing sellers through:

   - Reduced commission fees
   - Featured placement in search results
   - Priority promotions

   This fosters a competitive seller ecosystem and improves customer experience.

6. References

IBM. (n.d.). *What is ETL (extract, transform, load)?* IBM.

https://www.ibm.com/think/topics/etl

Kaggle. (n.d.). *Olist Brazilian e-commerce dataset* [Data set]. Kaggle.
https://www.kaggle.com/datasets/olistbr/brazilian-ecommerce

Villacorta, E. (2025). *BID3000-Exam* [Repository]. GitHub.
https://github.com/estebanvillaco/BID3000-Exam