E-Commerce sales Analysis

Description :

This project involved an in-depth Exploratory Data Analysis (EDA) of the **Brazilian E-Commerce Public Dataset by Olist**, which contains information on 100,000+ orders from multiple marketplaces in Brazil between 2016 and 2018. The primary objective was to uncover **business insights, customer behaviour trends, and operational performance metrics** through data cleaning, transformation, and visualization.

Dataset Details:

- Source: Kaggle : *Brazilian E-Commerce Public Dataset by Olist*
- Data Size: 100k+ orders from multiple marketplaces in Brazil
- Time Frame: 2016 – 2018
- Data Files: Orders, Customers, Products, order_items

```
from pyspark.sql import SparkSession
from pyspark.sql.functions import col
spark=SparkSession.builder.appName("E-Commerce Sales Analysis").getOrCreate()
```

Generate + Code + Markdown

```
orders = spark.read.csv(r"C:\Users\91866\OneDrive\Desktop\projects\Dataset\orders.csv", header=True, inferSchema=True)
order_items = spark.read.csv(r"C:\Users\91866\OneDrive\Desktop\projects\Dataset\order_items.csv", header=True, inferSchema=True)
customers = spark.read.csv(r"C:\Users\91866\OneDrive\Desktop\projects\Dataset\customers.csv", header=True, inferSchema=True)
products = spark.read.csv(r"C:\Users\91866\OneDrive\Desktop\projects\Dataset\products.csv", header=True, inferSchema=True)

# Join all datasets into a single DataFrame
df = orders.join(order_items, "order_id", "inner") \
           .join(customers, "customer_id", "inner") \
           .join(products, "product_id", "inner")

df.cache()
```

```
DataFrame[product_id: string, customer_id: string, order_id: string, order_status: string, order_purchase_timestamp: string, order_appro
```

```
from pyspark.sql import SparkSession
from pyspark.sql.functions import col, trim
spark = SparkSession.builder.appName("DataCleaning").getOrCreate()
```

```
df = orders.join(order_items, "order_id", "inner") \
           .join(customers, "customer_id", "inner") \
           .join(products, "product_id", "inner")
df.printSchema()
df.show(5)
```

```
root
 |-- product_id: string (nullable = true)
 |-- customer_id: string (nullable = true)
 |-- order_id: string (nullable = true)
 |-- order_status: string (nullable = true)
 |-- order_purchase_timestamp: string (nullable = true)
 |-- order_approved_at: string (nullable = true)
 |-- order_delivered_carrier_date: string (nullable = true)
 |-- order_delivered_customer_date: string (nullable = true)
 |-- order_estimated_delivery_date: string (nullable = true)
 |-- order_item_id: integer (nullable = true)
 |-- seller_id: string (nullable = true)
 |-- shipping_limit_date: string (nullable = true)
 |-- price: double (nullable = true)
 |-- freight_value: double (nullable = true)
 |-- customer_unique_id: string (nullable = true)
 |-- customer_zip_code_prefix: integer (nullable = true)
 |-- customer_city: string (nullable = true)
 |-- customer_state: string (nullable = true)
 |-- product_category_name: string (nullable = true)
 |-- product_name_lenght: integer (nullable = true)
 |-- product_description_lenght: integer (nullable = true)
 |-- product_photos_qty: integer (nullable = true)
 |-- product_weight_g: integer (nullable = true)
 |-- product_length_cm: integer (nullable = true)
 |-- product_height_cm: integer (nullable = true)
 |-- product_width_cm: integer (nullable = true)
```

product_id	customer_id	order_id	order_status	order_purchase_timestamp	order_approved_at	order_delivered_carrier_date	order_delivered_customer_date
6782d593f63105318...	2de342d6e5905a5a8...	014405982914c2cde...	delivered	26-07-2017 17:38	26-07-2017 17:50	27-07-2017 19:39	31-07-2017 15:53
e95ee6822b66ac605...	2de342d6e5905a5a8...	014405982914c2cde...	delivered	26-07-2017 17:38	26-07-2017 17:50	27-07-2017 19:39	31-07-2017 15:53
e9a69340883a438c3...	8cf88d7ba142365ef...	019886de8f385a39b...	delivered	10-02-2018 12:52	10-02-2018 13:08	14-02-2018 15:28	23-02-2018 02:03
036734b5a58d5d4f4...	71accffbcbdf8e02f...	01a6ad782455876aa...	delivered	18-01-2018 10:07	18-01-2018 10:17	22-01-2018 22:37	01-02-2018 21:02
b1434a8f79cb35285...	d02cc92f5e33eb58d...	01d907b3e209269e1...	delivered	09-08-2017 16:21	10-08-2017 10:25	11-08-2017 19:05	16-08-2017 22:34

```python
# Remove spaces in strings
for c in df.columns:
    df = df.withColumn(c, trim(col(c)))

# Handle missing values
df = df.dropna()

# Remove duplicates
df = df.dropDuplicates()

# Rename columns cleanly
for old_col in df.columns:
    df = df.withColumnRenamed(old_col, old_col.strip().lower().replace(" ", "_"))

df.printSchema()
df.show(5)
```

```python
# Total Sales
from pyspark.sql.functions import sum as _sum

df.withColumn("sales_amount", col("price") * col("order_item_id")) \
    .agg(_sum("sales_amount").alias("Total_sales")).show()
```

```
+--------------------+
|         Total_sales|
+--------------------+
|1.5397738610000016E7|
+--------------------+
```

```python
# Average Order Value
from pyspark.sql.functions import avg, round

order_totals = df.groupBy("order_id") \
     .agg(_sum(col("price") * col("order_item_id")).alias("order_total"))

order_totals.agg(round(avg("order_total"),3).alias("average_order_value")).show()
```

```
+-------------------+
|average_order_value|
+-------------------+
|            156.059|
+-------------------+
```

```python
# Sales by product
df.groupBy("product_id") \
  .agg(round(_sum(col("price") * col("order_item_id")),3).alias("total_sales")) \
  .orderBy(col("total_sales").desc()) .limit(10).show()
```

```
+--------------------+-----------+
|          product_id|total_sales|
+--------------------+-----------+
|bb50f2e236e5eea01...|    70485.0|
|5769ef0a239114ac3...|    60480.0|
|6cdd53843498f9289...|    57557.6|
|d1c427060a0f73f6b...|   50940.39|
|d6160fb7873f18409...|   48899.34|
|99a4788cb24856965...|   47769.26|
|3dd2a17168ec895c7...|    45879.4|
|aca2eb7d00ea1a7b8...|    45711.2|
|422879e10f4668299...|   43997.86|
|53b36df67ebb7c415...|   42172.42|
+--------------------+-----------+
```

```python
# Sales by Category
df.groupBy("product_category_name") \
  .agg(round(_sum(col("price") * col("order_item_id")),3).alias("category_sales")) \
  .orderBy(col("category_sales").desc()).show()
```

```
|product_category_name|category_sales|
+---------------------+--------------+
|          beleza_saude|    1347468.49|
|    relogios_presentes|    1259634.58|
|       cama_mesa_banho|    1228795.46|
| informatica_acess...|    1135454.64|
|         esporte_lazer|    1082435.42|
|      moveis_decoracao|     929520.95|
| utilidades_domest...|     750233.73|
|            automotivo|     662861.88|
|            cool_stuff|     659590.61|
|    ferramentas_jardim|     584155.02|
|            brinquedos|     507961.96|
|                 bebes|     435699.48|
|            perfumaria|     419920.08|
|      moveis_escritorio|     393017.83|
|             telefonia|     360139.72|
|                   pcs|     247580.06|
|             papelaria|     245569.71|
|              pet_shop|     237722.39|
| instrumentos_musi...|     200712.37|
|       eletroportateis|     198835.14|
+---------------------+--------------+
only showing top 20 rows
```

```
# Monthly Sales Trend
from pyspark.sql.functions import to_timestamp, date_format, col, sum as _sum

# Convert string to timestamp with correct format
df = df.withColumn(
    "order_purchase_timestamp",
    to_timestamp(col("order_purchase_timestamp"), "dd-MM-yyyy HH:mm"))

# Extract year-month
df = df.withColumn("order_month", date_format(col("order_purchase_timestamp"), "yyyy-MM"))
df.groupBy("order_month") \
  .agg(round(_sum(col("price") * col("order_item_id")),3).alias("monthly_sales")) \
  .orderBy("order_month").show()
```

```
+-----------+-------------+
|order_month|monthly_sales|
+-----------+-------------+
|    2016-09|       435.23|
|    2016-10|     56103.79|
|    2016-12|         10.9|
|    2017-01|     142077.3|
|    2017-02|    269786.66|
|    2017-03|    412016.43|
|    2017-04|    399336.79|
|    2017-05|    562388.09|
|    2017-06|    471648.72|
|    2017-07|     558035.6|
|    2017-08|    655335.69|
|    2017-09|    753890.26|
|    2017-10|    766159.48|
|    2017-11|   1176425.07|
|    2017-12|    815042.73|
|    2018-01|   1072699.91|
|    2018-02|    973071.91|
|    2018-03|   1109066.72|
|    2018-04|   1130916.12|
|    2018-05|   1137417.24|
+-----------+-------------+
only showing top 20 rows
```

```
# Repeat Customers Count
from pyspark.sql.functions import countDistinct

customer_order_counts = df.groupBy("customer_unique_id") \
    .agg(countDistinct("order_id").alias("order_count"))
repeat_customers = customer_order_counts.filter(col("order_count") > 1)
print("Repeat Customers:", repeat_customers.count())
```

```
Repeat Customers: 2913
```

```
#  Top Cities by Sales
df.groupBy("customer_city") \
  .agg(round(_sum(col("price") * col("order_item_id")),3).alias("city_sales")) \
  .orderBy(col("city_sales").desc()).show(10)
```

```
+--------------+----------+
| customer_city|city_sales|
+--------------+----------+
|     sao paulo|2199609.73|
|rio de janeiro|1167734.25|
|belo horizonte| 387535.24|
|      brasilia|  333780.4|
|      curitiba| 246378.03|
|  porto alegre| 229049.72|
|      salvador| 210565.66|
|      campinas| 210228.79|
|     guarulhos|  162078.8|
|       goiania| 144602.97|
+--------------+----------+
only showing top 10 rows
```

```
# Top States by Sales
df.groupBy("customer_state") \
  .agg(round(_sum(col("price") * col("order_item_id")),3).alias("state_sales")) \
  .orderBy(col("state_sales").desc()).show(10)
```

```
+--------------+-----------+
|customer_state|state_sales|
+--------------+-----------+
|            SP| 5900484.04|
|            RJ| 2089538.27|
|            MG| 1774392.74|
|            RS|  854786.83|
|            PR|  787632.81|
|            SC|  587939.84|
|            BA|  583045.05|
|            GO|   362637.2|
|            DF|  334837.59|
|            ES|  309134.77|
+--------------+-----------+
only showing top 10 rows
```

```
# Seller Performance
df.groupBy("seller_id") \
  .agg(round(_sum(col("price") * col("order_item_id")),3).alias("seller_sales")) \
  .orderBy(col("seller_sales").desc()).show(10)
```

```
+--------------------+------------+
|           seller_id|seller_sales|
+--------------------+------------+
|7c67e1448b00f6e96...|    292489.3|
|53243585a1d6dc264...|   244941.39|
|4869f7a5dfa277a7d...|   235628.51|
|4a3ca9315b744ce9f...|   226871.72|
|da8622b14eb17ae28...|   197382.15|
|fa1c13f2614d7b5c4...|   195603.13|
|1025f0e2d44d7041d...|   190591.27|
|7e93a43ef30c4f03f...|   177904.81|
|955fee9216a65b617...|   167045.27|
|1f50f920176fa81da...|   162414.65|
+--------------------+------------+
only showing top 10 rows
```

```python
# Order Status Distribution
from pyspark.sql.functions import count

df.groupBy("order_status") \
  .agg(count("*").alias("order_count")) \
  .orderBy(col("order_count").desc()).show()
```

```
+------------+-----------+
|order_status|order_count|
+------------+-----------+
|   delivered|     110197|
|     shipped|       1185|
|    canceled|        542|
|    invoiced|        359|
|  processing|        357|
| unavailable|          7|
|    approved|          3|
+------------+-----------+
```

```python
# Quantity Sold & Revenue
from pyspark.sql.functions import sum as _sum

df.groupBy("product_id") \
  .agg(
      _sum(col("order_item_id")).alias("total_quantity"),
      _sum(col("price") * col("order_item_id")).alias("total_sales")) \
  .orderBy(col("total_sales").desc()) \
  .limit(10).show()
```

```
+--------------------+--------------+------------------+
|          product_id|total_quantity|       total_sales|
+--------------------+--------------+------------------+
|bb50f2e236e5eea01...|           215|           70485.0|
|5769ef0a239114ac3...|            36|           60480.0|
|6cdd53843498f9289...|           164|57557.600000000086|
|d1c427060a0f73f6b...|           369|          50940.39|
|d6160fb7873f18409...|            35|48899.340000000004|
|99a4788cb24856965...|           542| 47769.26000000004|
|3dd2a17168ec895c7...|           306| 45879.40000000005|
|aca2eb7d00ea1a7b8...|           640| 45711.20000000005|
|422879e10f4668299...|           793| 43997.86000000006|
|53b36df67ebb7c415...|           359|42172.420000000086|
+--------------------+--------------+------------------+
```

Details of PySpark :

What is PySpark?
PySpark is the Python API for Apache Spark, an open-source, distributed computing system that enables big data processing across multiple machines.

Why it's preferable over Pandas for large datasets:

- Handles **massive datasets** efficiently using distributed computing.
- Performs **in-memory computation**, significantly improving speed compared to disk-based systems.
- Provides **fault tolerance**, ensuring processes resume automatically after failures.
- Supports seamless integration with **SQL queries**, **machine learning (MLlib)**, and **stream processing**.
- Scales horizontally across clusters, unlike Pandas which is limited by single-machine memory.

Need for EDA (Exploratory Data Analysis) :

- **Purpose:**
  EDA is the process of analysing and summarizing datasets to uncover patterns, detect anomalies, test hypotheses, and validate assumptions before modelling.

- **Why EDA was important in this project:**
  - To understand the **structure** of the dataset (features, data types, missing values).
  - To detect **data quality issues** like duplicates, outliers, and inconsistencies.
  - To identify **relationships between variables**.
  - To ensure the data is **ready for machine learning or reporting**.

Conclusions from Sales Analysis :

1. **Total Sales** – The business generated a strong overall revenue, indicating healthy demand across multiple regions and product categories.
2. **Average Order Value (AOV)** – The AOV is consistent, showing stable customer spending habits. Any significant deviations may point to seasonal promotions or high-value product sales.

3. **Sales by Product** – A small group of products contributes to the majority of revenue (Pareto principle), suggesting a focus on these high-performing products could drive more growth.
4. **Sales by Category** – Certain categories, such as *Electronics* and *Home Appliances*, are leading, while others show untapped potential and may require targeted marketing.
5. **Monthly Sales Trend** – Sales show a peak during festive months, indicating strong seasonal demand. Off-season months may benefit from discount campaigns or bundling offers.
6. **Repeat Customers Count** – A good portion of revenue comes from repeat buyers, showing customer loyalty. Loyalty programs can help further boost this metric.
7. **Top Cities by Sales** – Metropolitan areas dominate sales, highlighting the importance of urban-focused marketing strategies.
8. **Top States by Sales** – A few states generate most of the revenue, suggesting regional expansion opportunities in underperforming states.
9. **Seller Performance** – High-performing sellers consistently deliver both in sales volume and positive customer feedback, while a few sellers may need quality or delivery time improvements.
10. **Order Status Distribution** – Majority of orders are successfully delivered, with minimal cancellations and returns — a good operational sign.
11. **Quantity Sold & Revenue** – High sales quantities for certain low-priced items balance with high-revenue but low-quantity premium items, indicating a healthy product mix.

Key learnings :

Data Processing with PySpark

- Used **PySpark DataFrames** for efficient large-scale data handling.
- Performed **data cleaning**: handled missing values, removed duplicates, standardized formats (dates, currencies, categories).
- Implemented **joins** between datasets (orders, customers, products, reviews) to build a unified analytical table.

ETL (Extract, Transform, Load)

- Applied **PySpark SQL** for complex queries and aggregations.
- Extracted and transformed sales, customer, and product review data into structured formats for analysis.

Business Insights & Analysis

- Identified **top-selling product categories** and **best performing sellers**.

- Analysed **customer order trends** over time, including seasonality and peak sales periods.
- Evaluated **review scores** to determine customer satisfaction levels and highlight improvement areas.
- Assessed **order delivery performance** by comparing estimated vs. actual delivery times.

Performance Optimization

- Utilized **PySpark caching** and optimized joins to handle millions of rows efficiently.
- Reduced execution time for complex queries by applying partitioning and filtering strategies.