Project Forecasting_ LR_ (Python)

Import notebook

Customer Segmentation and Sales Forecast

Big Data Analytics 2025
NOVA IMS MDSAA

[NOTE]

In this project, we are going to work on 3 notebooks:

- 1. Cleaning: For EDA and Data Preprocessing
- 2. Clustering: For clustering
- 3. Project Forecasting: For Sales Forecast.

This notebook is 3. Project Forecasting.

Group 77

	Student Name	Student ID
1	Hassan Bhatti	20241023
2	Moeko Mitani	20240670
3	Oumayma Ben Hfaiedh	20240699
4	Ricardo Pereira	20240745

Business Understanding

1.1. Business Background

The retail industry is undergoing a significant transformation. Online retail shopping has become an absolute necessity to compete for business, and with that change comes new challenges, especially in niche areas such as gift items. As customer expectations rise and buying habits become more complex, retailers can no longer rely solely on intuition to gauge demand or manage inventory. They must become data-driven.

The company is a UK-based online retailer of giftware, primarily serving wholesale customers. This segment of the business has additional operational complexities, including high-volume purchases, unpredictable seasonality (especially during the holiday season), and a customer base divided between loyal repeat buyers and one-time, resourceful purchasers. What appears to be a simple transaction flow is in fact a rich and dynamic stream of behavioral data waiting to be deciphered.

In this environment, traditional data tools are not enough. Forecasting demand and understanding customers requires a scalable and intelligent approach. This project reflects how large companies are beginning to process huge, fast-moving data sets. Even though the current dataset is limited in size, it mirrors the volume, velocity, and variety challenges faced by growing online retailers.

1.2. Business Objectives

The overarching goal of this project is to empower a growing online retailer with the analytical tools needed to make smarter, data-driven decisions in two critical areas: customer understanding and demand forecasting.

Customer Segmentation

The first objective is to uncover meaningful customer segments based on purchasing behavior. Not all customers bring the same value or behave in the same way - some make frequent low-volume purchases, others buy in bulk seasonally, and some show irregular patterns that suggest churn risk or opportunistic buying.

By applying clustering techniques, we aim to:

- Identify distinct customer personas (e.g., "Loyal Wholesalers", "Occasional Retailers", "Holiday Shoppers")
- Reveal behavioral patterns that can inform targeted marketing and personalized recommendations
- Provide insights to improve customer retention and customer lifetime value (CLV)

This segmentation can serve as the foundation for a more customized engagement strategy, allowing retailers to move away from one-size-fits-all campaigns toward data-driven personalization.

Sales Forecasting

The second objective is to develop a predictive model that estimates future sales based on historical transaction data.

Accurate forecasting is essential for:

- Optimizing inventory levels and reducing both stockouts and overstock situations
- Aligning operational planning with expected demand spikes (e.g., during the holiday season)
- Informing pricing, promotional, and procurement strategies

By implementing time-series forecasting models, we will simulate a pipeline that can eventually evolve into a real-time prediction engine in a production environment.

1.3. Delivery: Promoting Data Culture

As a fun and creative twist, we also plan to write an internal promotional article titled something like:
"How Big Data Knows When Your Aunt Buys That Weird Candle Set Every Christmas"

This lighthearted piece will explain our results in plain language to non-technical employees - demonstrating how customer insights and predictive analytics can revolutionize operations, and inspiring a company-wide embrace of digital transformation and data culture.

1.4. Business Success Criteria

Success for this project will be evaluated using both **quantitative** and **qualitative** criteria:

Quantitative Criteria

- **Clustering Performance**: Metrics such as silhouette score, Davies-Bouldin index, or within-cluster sum of squares (WCSS) will be used to assess the quality of customer segmentation.
- **Forecast Accuracy**: Measured using MAE (Mean Absolute Error), RMSE (Root Mean Squared Error), and MAPE (Mean Absolute Percentage Error) on sales predictions.
- **Actionable Insights**: The identification of at least 3 meaningful and distinct customer segments, and 1-2 sales forecasting trends that could support operational decisions.

Qualitative Criteria

- **Interpretability**: Clear and intuitive visualization of clusters and forecast trends for presentation to non-technical stakeholders.
- **Engagement**: The fun internal article should effectively raise awareness about the value of data analytics and be positively received by company staff.
- **Scalability Potential**: The approach should be adaptable to larger datasets and scalable for production-level deployment in a real business context.

By combining rigorous analytics with creative storytelling, this project aims not only to deliver strategic insights but also to shift the company mindset toward becoming truly data-driven.

Metadata

Features	Descriptions
ID	Customer ID
Invoice	Invoice number. Nominal. A 6-digit integral number uniquely assigned to each transaction. If this code starts with the letter 'c', it indicates a cancellation.
StockCode	Product (item) code. Nominal. A 5-digit integral number uniquely assigned to each distinct product.
Description	Product (item) name. Nominal.
Quantity	The quantities of each product (item) per transaction. Numeric.
InvoiceDate	Invoice date and time. Numeric. The day and time when a transaction was generated.
Price	Unit price. Numeric. Product price per unit in sterling (Â£).
Customer ID	Customer number. Nominal. A 5-digit integral number uniquely assigned to each customer.
Country	Country name. Nominal. The name of the country where a customer resides.

Data Integration

Import Libraries

```
from pyspark.sql import SparkSession
from pyspark.ml.feature import VectorAssembler, PCA
from pyspark.ml.clustering import KMeans
from pyspark.ml.evaluation import ClusteringEvaluator
from pyspark.ml.feature import StringIndexer, VectorAssembler, StandardScaler, PCA
from pyspark.ml import Pipeline
import seaborn as sns
import matplotlib.pyplot as plt
import pandas as pd
import numpy as np
from matplotlib.colors import ListedColormap
from sklearn.metrics import silhouette_samples, silhouette_score
import numpy as np
import matplotlib.cm as cm
from sklearn.metrics import confusion_matrix
import seaborn as sns
from pyspark.sql import functions as F
from pyspark.sql.functions import to_timestamp, year, month, dayofmonth, to_date, lit
from pyspark.sql.functions import countDistinct, sum, avg, min, max
# EDA
from pyspark.sql.functions import col,sum
from pyspark.sql.functions import countDistinct
# Data Anomalies
from pyspark.sql.functions import col, max as spark_max
from pyspark.sql.functions import monotonically_increasing_id
from pyspark.sql.functions import col, when, to_timestamp
from pyspark.sql.functions import col, datediff, current_date, round
# Forecasting
from pyspark.sql.functions import year, month, to_date, col, lit
from pyspark.ml.feature import VectorAssembler
from sklearn.linear_model import LinearRegression
import pandas as pd
from sklearn.linear_model import LinearRegression as SklearnLinearRegression
from pyspark.sql.functions import col
from sklearn.metrics import mean_squared_error
from pyspark.sql.window import Window
from pyspark.sql.functions import col, last
from pyspark.sql.functions import date_format, col, collect_set, size, array_sort, array_except, array, lit
from pyspark.sql.window import Window
from pyspark.sql.functions import lag, avg, stddev, col, to_date, year, month, lit
from pyspark.ml.feature import VectorAssembler
from pyspark.sql.functions import col, to_date, date_format, sum as spark_sum
from pyspark.sql import Row
from pyspark.sql.functions import col, lit, to_date, date_format, year, month
from datetime import datetime
from dateutil.relativedelta import relativedelta
from pyspark.sql.window import Window
from pyspark.sql.functions import sin, cos, lit, col, lag, avg, stddev, month
import math
```

```
# Start Spark session
spark = SparkSession.builder.appName("Project_Group77").getOrCreate()
```

Import CSV File

```
# File location and type
file_location = "/FileStore/tables/the_online_retail.csv"
file_type = "csv"

# CSV options
infer_schema = "true"
first_row_is_header = "true"
delimiter = ","

# The applied options are for CSV files. For other file types, these will be ignored.
df = spark.read.format(file_type) \
  .option("inferSchema", infer_schema) \
  .option("header", first_row_is_header) \
  .option("sep", delimiter) \
  .load(file_location)

display(df.limit(5))
df.schema
```

df: pyspark.sql.dataframe.DataFrame = [Invoice: string, StockCode: string ... 6 more fields]

Table

```
Out[3]: StructType([StructField('Invoice', StringType(), True), StructField('StockCode', StringType(), True), StructField('Description', StringType(), True), StructField('Quantity', IntegerType(), True), StructField('InvoiceDate', TimestampType(), True), StructField('Price', DoubleType(), True), StructField('Customer ID', DoubleType(), True), StructField('Country', StringType(), True)])
```

The dataset contains 1,067,371 rows and 8 features.

Data Type

Features	Data Types	Need to be changed?
Invoice	String	No - "C" stands for cancellation
StockCode	String	No - Some code contains a string
Description	String	No
Quantity	String	Yes - Integer
InvoiceDate	String	Yes - Timestamp
Price	String	Yes - Decimal
Customer ID	String	Yes - Integer
Country	String	No

Data Anomalies

Features	Anomalies	Steps
StockCode	Some code contains only a string	Check if the same description shares same stock code or not
Quantity	It contains negative values	Check it after changing to the correct data type
Price	It contains negative values	Check it after changing to the correct data type

Missing Value

```
from pyspark.sql.functions import col, sum as _sum
# Check if there are any missing values
df.select([_sum(col(c).isNull().cast("int")).alias(c) for c in df.columns]).show()
```

```
+-------+---------+-----------+--------+-----------+-----+-----------+-------+
|Invoice|StockCode|Description|Quantity|InvoiceDate|Price|Customer ID|Country|
+-------+---------+-----------+--------+-----------+-----+-----------+-------+
|      0|        0|       4382|       0|          0|    0|     243007|      0|
+-------+---------+-----------+--------+-----------+-----+-----------+-------+
```

There are:

- 4,382 missing values in *Description* (approximately 0.4% of the data)
- 243,007 missing values in *Customer ID* (approximately 22.8% of the data)

Possible solution:

- *Description*: Use *StockCode* --> *StockCode* and *Description* should match.
- *Customer ID*: Use *Invoice* --> The same invoices belong to the same customer.

Change the data types

As we identified previously, we are going to change the data types of the following features:

Features	Data Types	Need to be changed?
Quantity	String	Yes - Integer
InvoiceDate	String	Yes - Timestamp
Price	String	Yes - Decimal
Customer ID	String	Yes - Integer

```
# Change the data types
df = df \
    .withColumn("Quantity", col("Quantity").cast("int")) \
    .withColumn("InvoiceDate", to_timestamp(col("InvoiceDate"), "dd/MM/yyyy HH:mm")) \
    .withColumn("Price", col("Price").cast("decimal(20,2)")) \
    .withColumn("Customer ID", col("Customer ID").cast("int"))
```

df: pyspark.sql.dataframe.DataFrame = [Invoice: string, StockCode: string ... 6 more fields]

Data Anomalies Treatment

Check Duplicates

```
# Count total rows and distinct rows
total_rows = df.count()
distinct_rows = df.distinct().count()

# Check for duplicates
if total_rows > distinct_rows:
    print(f"Duplicates found: {total_rows - distinct_rows}")
else:
    print("No duplicates found.")
```

```
Duplicates found: 34335
```

We found there are 34,335 duplicates in our dataset.

Remove Duplicates

We are going to remove the duplicates for more precise clustering.

```
df_no_duplicates = df.dropDuplicates()
df_no_duplicates.limit(10).display()
```

df_no_duplicates: pyspark.sql.dataframe.DataFrame = [Invoice: string, StockCode: string ... 6 more fields]

Table

Missing Values

Recap

There are:
- 4,382 missing values in *Description* (approximately 0.4% of the data)
- 243,007 missing values in *Customer ID* (approximately 22.8% of the data)

Possible solution:
- *Description*: Use *StockCode* --> *StockCode* and *Description* should match.
- *Customer ID*: Use *Invoice* --> The same invoices belong to the same customer.

Customer ID

We are going to fill in missing Customer IDs by assigning new unique IDs based on their Invoice numbers.
If the max Cusotmer ID is 9823 in the dataset, then new customer IDs will start from 9823 + 1 = 9824.

NOTE: `monotonically_increasing_id()`
This Spark function generates a unique and increasing ID for each row.
It's:

- Not guaranteed to be consecutive (e.g., might go 0, 4, 9...),
- But each row will get a different number,
- Safe to use in distributed environments (like Spark/Databricks).

```
#### Extract existing Customer IDs and find the max
#### We are going to add +1 to max Customer ID to fill the missing values
# Ensure all Customer IDs are integers., Cast to integer if needed
df_filled = df_no_duplicates.withColumn("Customer ID", col("Customer ID").cast("int"))

# spark_max --> Get max existing Customer ID
# .first()[0] --> Get the actual value (not a Row object).
# or 10000 --> If that value is None (i.e., no customer IDs exist at all), then it uses 10000 instead.
max_existing_id = df_filled.select(spark_max("Customer ID")).first()[0] or 10000

#### Get invoices with missing Customer ID
# Filter out all rows where Customer ID is missing
# Then selects the distinct invoices (each invoice will get a new fake ID later)
missing_cus_df = df_filled.filter(col("Customer ID").isNull()).select("Invoice").distinct()

#### Generate new Customer IDs for these invoices
# Add a unique ID starting after max_existing_id
# Cap the generated ID at 100,000 just to keep it manageable
new_ids_df = missing_cus_df.withColumn(
    "Customer ID",
    (monotonically_increasing_id() % 100000 + max_existing_id + 1).cast("int")
)
#### Join these new IDs back to the original DataFrame
# Rename the generated Customer ID column to avoid conflict
new_ids_df_renamed = new_ids_df.withColumnRenamed("Customer ID", "Generated_Customer_ID")

# Perform the join and update Customer ID
df_final = df_filled.join(
    new_ids_df_renamed,
    on="Invoice", # Join the generated IDs onto the original data by Invoice
    how="left"
).withColumn(
    "Customer ID",
    when(col("Customer ID").isNull(), col("Generated_Customer_ID")).otherwise(col("Customer ID"))
    # Replaces missing Customer IDs with the generated ones
    # Keeps the existing ones as they are
).drop("Generated_Customer_ID")
    # Drops the temporary Generated_Customer_ID column afterward
```

- df_filled: pyspark.sql.dataframe.DataFrame = [Invoice: string, StockCode: string ... 6 more fields]
- df_final: pyspark.sql.dataframe.DataFrame = [Invoice: string, StockCode: string ... 6 more fields]
- missing_cus_df: pyspark.sql.dataframe.DataFrame = [Invoice: string]
- new_ids_df: pyspark.sql.dataframe.DataFrame = [Invoice: string, Customer ID: integer]
- new_ids_df_renamed: pyspark.sql.dataframe.DataFrame = [Invoice: string, Generated_Customer_ID: integer]

```
# Check if there are any missing values
df_final.select(*(sum(col(c).isNull().cast("int")).alias(c) for c in df.columns)).show()
```

```
+-------+---------+-----------+--------+-----------+-----+-----------+-------+
|Invoice|StockCode|Description|Quantity|InvoiceDate|Price|Customer ID|Country|
+-------+---------+-----------+--------+-----------+-----+-----------+-------+
|      0|        0|       4275|       0|          0|    0|          0|      0|
+-------+---------+-----------+--------+-----------+-----+-----------+-------+
```

Description

Remove rows where *Price* = 0

We noticed that ***Description* is empty when the *Price* = 0**. It does not make sense to keep the rows with price = 0, so we are going to removed them.

```
# Remove rows where Price is 0
df_final = df_final.filter(col("Price") != 0)
```

- df_final: pyspark.sql.dataframe.DataFrame = [Invoice: string, StockCode: string ... 6 more fields]

```
# Check if there are any missing values
df_final.select(*(sum(col(c).isNull().cast("int")).alias(c) for c in df.columns)).show()
```

```
+-------+---------+-----------+--------+-----------+-----+-----------+-------+
|Invoice|StockCode|Description|Quantity|InvoiceDate|Price|Customer ID|Country|
+-------+---------+-----------+--------+-----------+-----+-----------+-------+
|      0|        0|          0|       0|          0|    0|          0|      0|
+-------+---------+-----------+--------+-----------+-----+-----------+-------+
```

It can be seen that there is no missing value after removing *Price* = 0.

Anomalies

Invoice

We understood that if invoice code starts with the letter 'C', it indicates a cancellation in *Invoice*. To simplify this, we are going to make new feature called *IsReturn* which identify if the order was returned (0) or not (1).

```
# Create new column that checks for quantity < 0 which means a return
df_final = df_final.withColumn("IsReturn", when(col("Quantity") < 0, 1).otherwise(0))
```

df_final: pyspark.sql.dataframe.DataFrame = [Invoice: string, StockCode: string ... 7 more fields]

```
df_final.limit(5).display()
```

Table

StockCode

Remove rows where we have TEST

We realized that there are some test data in our dataset. Since they are not actual data, we are going to remove them.

```
# Remove rows that start with "TEST" in StockCode
df_final = df_final.filter(~col('StockCode').startswith('TEST'))
```

df_final: pyspark.sql.dataframe.DataFrame = [Invoice: string, StockCode: string ... 7 more fields]

```
row_count = df_final.count()
print(f"Number of rows: {row_count}")
```

```
Number of rows: 1026990
```

After the Data Anomalies Treatment, now the dataset consists of 1008415 rows and 9 features.

Creating Dataframe for forecasting

This dataframe will be based on the concept of having product(StockCode) as index in the dataframe.

```
# Convert to date and extract month
df_forecasting = df_final.withColumn("date", to_date(col("InvoiceDate"))) \
       .withColumn("month", date_format(col("date"), "yyyy-MM"))

# Group by SKU and month
df_forecasting = df_forecasting.groupBy("StockCode", "month").agg(
    spark_sum("quantity").alias("total_quantity")
)
```

df_forecasting: pyspark.sql.dataframe.DataFrame = [StockCode: string, month: string ... 1 more field]

```
row_count = df_forecasting.count()
print(f"Number of rows: {row_count}")
```

```
Number of rows: 67445
```

```
display(df_forecasting.limit(5))
```

Table

Feature Engineering and Transformation

Here we are extending the forecasting DataFrame (df_forecasting) with future months (Jan–Jun 2025) for each unique StockCode to be able to predict the quantities of gthese months, preserving the structure and readying it for predictions.

1. Convert "month" to proper date format for processing.
2. Generate future month dates (Jan–Jun 2025)and create new rows for each future month & stock code combination.
3. Derive date features like "month", "year", and "month_num" for modeling/analysis.

```
# Ensure 'date' column exists in df_forecasting
df_forecasting = df_forecasting.withColumn("date", to_date(col("month"), "yyyy-MM"))

# Create list of future months
future_months = []
future_start = datetime(2025, 1, 1)
future_end = datetime(2025, 6, 1)
while future_start <= future_end:
    future_months.append(future_start.strftime("%Y-%m-01"))
    future_start += relativedelta(months=1)

# Get distinct StockCodes from df_forecasting
stockcodes = [row['StockCode'] for row in df_forecasting.select("StockCode").distinct().collect()]

# Create future rows (StockCode + date)
future_rows = []
for code in stockcodes:
    for month_str in future_months:
        future_rows.append(Row(StockCode=code, date=datetime.strptime(month_str, "%Y-%m-%d")))

# Convert future_rows to DataFrame
df_future = spark.createDataFrame(future_rows)
df_future = df_future.withColumn("date", to_date(col("date")))

# Add missing columns from df_forecasting schema
for column in df_forecasting.columns:
    if column not in df_future.columns:
        df_future = df_future.withColumn(column, lit(None).cast(df_forecasting.schema[column].dataType))

# Reorder to match df_forecasting
df_future = df_future.select(df_forecasting.columns)

# Union + sort
df_final = df_forecasting.unionByName(df_future).orderBy("StockCode", "date")

# Add 'month', 'year', and 'month_num' derived from 'date'
df_final = df_final.withColumn("month", date_format(col("date"), "yyyy-MM")) \
                   .withColumn("year", year("date")) \
                   .withColumn("month_num", month("date"))

display(df_final.limit(5))
```

- df_final: pyspark.sql.dataframe.DataFrame = [StockCode: string, month: string ... 4 more fields]
- df_forecasting: pyspark.sql.dataframe.DataFrame = [StockCode: string, month: string ... 2 more fields]
- df_future: pyspark.sql.dataframe.DataFrame = [StockCode: string, month: string ... 2 more fields]

Table

Here we did feature engineering to capture:

- Trends (via rolling average)
- Seasonality (via sine/cosine)
- Anomalies or stability (via standard deviation)
- Delayed effects (via lag)

```
# Create month_sin, month_cos
df_final = df_final.withColumn("month_sin", sin(2 * math.pi * col("month_num") / 12)) \
                   .withColumn("month_cos", cos(2 * math.pi * col("month_num") / 12))

# Define window
# Partition by StockCode, order by date
window_spec = Window.partitionBy("StockCode").orderBy("date").rowsBetween(-5, 0)  # last 6 months including current

# Create lag and rolling features
df_final = df_final.withColumn("lag_6m_quantity", lag("total_quantity", 6).over(Window.partitionBy("StockCode").orderBy("date"))) \
                   .withColumn("rolling_avg_6m", avg("total_quantity").over(window_spec)) \
                   .withColumn("rolling_std_6m", stddev("total_quantity").over(window_spec))

display(df_final.limit(5))
```

- df_final: pyspark.sql.dataframe.DataFrame = [StockCode: string, month: string ... 9 more fields]

Table

Since we chose to do the lag of 6 months this created missing values in the column lag_6m_quantity(the first 6 month quantity of each product 12-2022 until 06-2023). For this reason we decided to remove these rows and start our training from 06-2023.

```
# Add "month" column
df_final = df_final.withColumn("month", date_format(col("date"), "yyyy-MM"))

# Define required months as Spark array
required_months = [f"{y}-{str(m).zfill(2)}" for y in range(2023, 2026) for m in range(1, 13)]
required_months = [m for m in required_months if "2023-06" <= m <= "2025-06"]
required_months_array = array(*[lit(m) for m in required_months])

# Get months available for each StockCode only in the target range
stock_months = df_final.filter(col("month").between("2023-06", "2025-06")) \
    .select("StockCode", "month") \
    .distinct() \
    .groupBy("StockCode") \
    .agg(array_sort(collect_set("month")).alias("months_present"))

# Keep only StockCodes with ALL required months
valid_products = stock_months.filter(
    size(array_except(required_months_array, col("months_present"))) == 0
).select("StockCode")

# Join and KEEP ONLY records in 2023-06 to 2025-06
df_final = df_final.join(valid_products, on="StockCode", how="inner") \
                   .filter(col("month").between("2023-06", "2025-06"))

display(df_final.limit(5))
```

- df_final: pyspark.sql.dataframe.DataFrame = [StockCode: string, month: string ... 9 more fields]
- stock_months: pyspark.sql.dataframe.DataFrame = [StockCode: string, months_present: array]
- valid_products: pyspark.sql.dataframe.DataFrame = [StockCode: string]

Table

```
row_count = df_final.count()
print(f"Number of rows: {row_count}")
```

```
Number of rows: 24250
```

Another problem we had is that for features rolling_avg_6m and rolling_std_6m we had missig values in the month 2025-05 to fix this problem we opted for the forward-fill solution to fill these missing values

```
# Ensure month_num exists for sorting
df_final = df_final.withColumn(
    "month_num",
    col("month").substr(1, 4).cast("int") * 100 + col("month").substr(6, 2).cast("int")
)

# Define forward-fill window (up to current row)
forward_window = Window.partitionBy("StockCode").orderBy("month_num").rowsBetween(Window.unboundedPreceding, 0)

# Forward-fill rolling_avg_6m and rolling_std_6m
df_final = df_final.withColumn(
    "rolling_avg_6m",
    last("rolling_avg_6m", ignorenulls=True).over(forward_window)
).withColumn(
    "rolling_std_6m",
    last("rolling_std_6m", ignorenulls=True).over(forward_window)
)

display(df_final.limit(5))
```

- df_final: pyspark.sql.dataframe.DataFrame = [StockCode: string, month: string ... 9 more fields]

Table

Splitting data frame

```
# Train: 2023-06 to 2024-06
df_train = df_final.filter((col("month") >= "2023-06") & (col("month") <= "2024-06"))

# Validation: 2024-07 to 2024-12
df_val = df_final.filter((col("month") >= "2024-07") & (col("month") <= "2024-12"))

# Test: 2025-01 to 2025-06
df_test = df_final.filter((col("month") >= "2025-01") & (col("month") <= "2025-06"))
```

- df_test: pyspark.sql.dataframe.DataFrame = [StockCode: string, month: string ... 9 more fields]
- df_train: pyspark.sql.dataframe.DataFrame = [StockCode: string, month: string ... 9 more fields]
- df_val: pyspark.sql.dataframe.DataFrame = [StockCode: string, month: string ... 9 more fields]

Model Assessment

Because of the limitations of databricks communty edition we couldn't do forecasting for all products. That's why we chose to only run predictions for only one product.

Linear Regression Model

```
Validation RMSE: 95.90
  StockCode    month  total_quantity  predicted_quantity
0     10135  2024-07             177          183.531516
1     10135  2024-08             151          114.776664
2     10135  2024-09              70           94.209558
3     10135  2024-10              68          116.838288
4     10135  2024-11             181          157.540407
5     10135  2024-12              69          293.279953
  StockCode    month  predicted_quantity
0     10135  2025-01          459.997878
1     10135  2025-02          588.241352
2     10135  2025-03          608.978579
3     10135  2025-04          533.197924
4     10135  2025-05          553.538932
5     10135  2025-06          411.801672
```