Cleaning (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 1. Cleaning.

Group 77

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

1. 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.

2. 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.

3. Data Integration

Import Libraries

```
# ──────────────────────────────────────────────
# Spark Core
# ──────────────────────────────────────────────
from pyspark.sql import SparkSession
from pyspark.sql import functions as F
from pyspark.sql.functions import (
    col, lit, to_timestamp, to_date, year, month, dayofmonth,
    count, countDistinct, sum, avg, min, max,
    round, when, datediff, current_date, concat_ws,
    monotonically_increasing_id
)
from pyspark.sql.functions import max as spark_max
from pyspark.sql.functions import date_format, sum as spark_sum

# ──────────────────────────────────────────────
# Spark MLlib
# ──────────────────────────────────────────────
from pyspark.ml.feature import (
    VectorAssembler, PCA, StringIndexer, StandardScaler,
    MinMaxScaler
)
from pyspark.ml.clustering import KMeans
from pyspark.ml.stat import Correlation
from pyspark.ml.evaluation import ClusteringEvaluator
from pyspark.ml import Pipeline

# ──────────────────────────────────────────────
# Python Built-ins and Data Science Libraries
# ──────────────────────────────────────────────
import pandas as pd
import numpy as np
import seaborn as sns
import matplotlib.pyplot as plt
import matplotlib.cm as cm
from matplotlib.colors import ListedColormap
from sklearn.metrics import (
    silhouette_samples, silhouette_score,
    confusion_matrix
)

# ──────────────────────────────────────────────
# Utilities
# ──────────────────────────────────────────────
from itertools import combinations
```

```
# 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)

df.limit(10).display()
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)])
```

4. Data Exploration (EDA)

4.1. Summary Statistics

```
df.describe().show()
```

```
+-------+-----------------+-----------------+--------------------+------------------+------------------+------------------+-----------+
|summary|          Invoice|        StockCode|         Description|          Quantity|             Price|       Customer ID|    Country|
+-------+-----------------+-----------------+--------------------+------------------+------------------+------------------+-----------+
|  count|          1067371|          1067371|             1062989|           1067371|           1067371|            824364|    1067371|
|   mean|537608.1499316233|29011.161534536903|             21848.25|  9.9388984711033| 4.649387727415796| 15324.63850435002|       null|
| stddev|26662.45044690759|  18822.9428661892|   922.9197780233488|172.70579407675396|123.55305872146369|1697.4644503793133|       null|
|    min|           489434|            10002|  DOORMAT UNION J...|            -80995|          -53594.36|           12346.0|  Australia|
|    max|          C581569|                m|   wrongly sold sets|             80995|           38970.0|           18287.0|West Indies|
+-------+-----------------+-----------------+--------------------+------------------+------------------+------------------+-----------+
```

Number of rows in the DataFrame

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

```
Number of rows: 1067371
```

The time range of the DataFrame

```
df.agg(F.min("InvoiceDate"), F.max("InvoiceDate")).show()
```

```
+-------------------+-------------------+
|   min(InvoiceDate)|   max(InvoiceDate)|
+-------------------+-------------------+
|2022-12-01 07:45:00|2024-12-09 12:50:00|
+-------------------+-------------------+
```

4.2. Unique Value Counts

```
    # Get distinct counts (unique value) for all columns
    for col_name in df.columns:
        unique_count = df.select(countDistinct(col_name)).collect()[0][0]
        print(f"Unique values in '{col_name}': {unique_count}")
```

```
Unique values in 'Invoice': 53628
Unique values in 'StockCode': 5305
Unique values in 'Description': 5698
Unique values in 'Quantity': 1057
Unique values in 'InvoiceDate': 47635
Unique values in 'Price': 2807
Unique values in 'Customer ID': 5942
Unique values in 'Country': 43
```

```
    ## Check if there are StockCodes contain any letters
    # Filter rows where StockCode contains any letters
    df.filter(col("StockCode").rlike("[A-Za-z]")) \
      .select("StockCode") \
      .distinct() \
      .show()
```

```
|   84970S|
|   35004B|
|   85132C|
|   84519A|
|   85183B|
|   84507B|
|   84596L|
|   84031A|
|   16161P|
|   84519B|
|   84596F|
|   85014A|
|   84031B|
|   85183A|
|   84032B|
|   48173C|
|   84032A|
|   79323P|
+---------+
only showing top 20 rows
```

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

4.3. 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

4.4. 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

4.5. Missing Value

```
    # 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.

4.6. Change The Data Types

Before moving to create visualizations, we are going to change the data type. 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]

4.7. Visual EDA

4.7.1. Numerical Features' Visualization

```
display(df)
```

Table

4.7.2. Timestamp Feature's Visualization

```
# Creates a new column called 'InvoiceDateOnly' with just the date part from InvoiceDate (drop the time)
# Group all rows by the date --> Count how many invoices happened for each date and name the column 'NumInvoices'
df_daily_invoices = df.withColumn("InvoiceDateOnly", to_date("InvoiceDate"))\
                      .groupBy("InvoiceDateOnly")\
                      .agg(count("Invoice").alias("NumInvoices"))\
                      .orderBy("InvoiceDateOnly")  # Sort the results by the date in ascending order
```

▸ df_daily_invoices: pyspark.sql.dataframe.DataFrame = [InvoiceDateOnly: date, NumInvoices: long]

4.7.3. Categorical Features' Visualization

```
display(df)
```

Table

5. Data Anomalies Treatment

5.1. Duplicates 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.

```
# Drop Dulpicates
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

5.2. Missing Values Treatment

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).

```
#### Step 1: 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

#### Step 2: 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()

#### Step 3: 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")
)
#### Step 4:  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
# For each column, checks if the value is null
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 the rows where Price is equal to 0
df_final = df_final.filter(col("Price") != 0)
```

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

```
# Check again if there are any missing values in the dataset
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.

5.3. Anomalies Treatment

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(10).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]
```

```
df_final.display()
```

Table

```
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.

6. Creating Dataframe for Clustering

6.1. Feature Engineering

For better cluster identification, we engineered the following features:

New Feature	Description	Equation
Total Price (*TotalPrice*)	Total price of each transaction line	`Quantity * Price`
Number of Products (*num_products*)	Number of unique products purchased by the customer	`countDistinct(StockCode)`
Total Quantity (*total_quantity*)	Total number of items purchased by the customer	`sum(Quantity)`
Total Price (*total_price*)	Total amount spent by the customer	`sum(TotalPrice)`

New Feature	Description	Equation
Average Unit Price (*avg_unit_price*)	Average price per unit item purchased	`avg(Price)`
First Purchase Date (*first_purchase_date*)	Date of customer's first recorded purchase	`min(InvoiceDateOnly)`
Last Purchase Date (*last_purchase_date*)	Date of customer's most recent purchase	`max(InvoiceDateOnly)`
Purchase Span (*purchase_span_days*)	Time between first and last purchase	`datediff(last_purchase_date, first_purchase_date)`
Average Quantity per Invoice (*avg_quantity_per_invoice*)	Average number of items per invoice	`total_quantity / num_invoices`
Recency (*recency_days*)	Days since the customer's last purchase (as of 09/12/2024)	`datediff(09/12/2024, last_purchase_date)`
Monthly Purchase Counts	Count of purchases per month (pivoted)	`pivot(year_month).count(Invoice)`

These engineered features aim to capture customer behavior such as spending habits, purchasing frequency, and recency. This enriched dataset forms the basis for meaningful clustering and segmentation analysis.

```
# Make a copy
df_cl = df_final
```

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

```
# Create a 'TotalPrice' feature
df_cl = df_cl.withColumn("TotalPrice", col("Quantity") * col("Price"))
```

▸ df_cl: pyspark.sql.dataframe.DataFrame = [Invoice: string, StockCode: string ... 8 more fields]

```
# Extract year, month, and day from InvoiceDate and create features
df_cl = df_cl.withColumn("InvoiceYear", year("InvoiceDate")) \
       .withColumn("InvoiceMonth", month("InvoiceDate")) \
       .withColumn("InvoiceDay", dayofmonth("InvoiceDate"))

# Create a feature called InvoiceDateOnly that contain only the date of invoice
df_cl = df_cl.withColumn("InvoiceDateOnly", to_date("InvoiceDate"))
```

▸ df_cl: pyspark.sql.dataframe.DataFrame = [Invoice: string, StockCode: string ... 12 more fields]

```
df_cl.limit(10).display()
```

Table

```
# Aggregate features per customer (Customer ID) and change their names
customer_df_final = df_cl.groupBy("Customer ID").agg(
    countDistinct("Invoice").alias("num_invoices"),
    countDistinct("StockCode").alias("num_products"),
    sum("Quantity").alias("total_quantity"),
    sum("TotalPrice").alias("total_price"),
    avg("Price").alias("avg_unit_price"),
    min("InvoiceDateOnly").alias("first_purchase_date"),
    max("InvoiceDateOnly").alias("last_purchase_date")
)
```

▸ customer_df_final: pyspark.sql.dataframe.DataFrame = [Customer ID: integer, num_invoices: long ... 6 more fields]

Here we have created 7 features:

- *num_invoices*: How many unique invoices a customer made
- *num_products*: How many unique products they bought
- *total_quantity*: Sum of all quantities purchased
- *total_price*: Total revenue from this customer
- *avg_unit_price*: Average unit price per item
- *first_purchase_date*: When they made their first purchase
- *last_purchase_date*: Their most recent purchase

```
customer_df_final.limit(10).display()
```

Table

```
### Create new time-based features
# Define and convert last date of dataset (December 9, 2024) to calculate the recency
last_date_of_dataset = to_date(lit("09/12/2024"), "dd/MM/yyyy")

# Create new features: purchase_span_days, avg_quantity_per_invoice, and recency_days
customer_df_final = customer_df_final.withColumn(
    "purchase_span_days", datediff("last_purchase_date", "first_purchase_date")
).withColumn(
    "avg_quantity_per_invoice", col("total_quantity") / col("num_invoices")
).withColumn(
    "recency_days", datediff(last_date_of_dataset, col("last_purchase_date"))
)

# Rounding the new price features to 2 decimal places for better readability and consistency
customer_df_final = customer_df_final.withColumn(
    "total_price", round(col("total_price"), 2)
).withColumn(
    "avg_unit_price", round(col("avg_unit_price"), 2)
).withColumn(
    "avg_quantity_per_invoice", round(col("avg_quantity_per_invoice"), 2)
)
```

▸ customer_df_final: pyspark.sql.dataframe.DataFrame = [Customer ID: integer, num_invoices: long ... 9 more fields]

Here we have created

- *purchase_span_days*: Number of days between first and last purchase (customer lifetime)
- *avg_quantity_per_invoice*: Quantity of items per invoice
- *recency_days*: How many days since the customer last purchased

We are going to add dummy features that will count how many times the client bought during each month.

```
# Step 1: Extract year-month and join them into a string like "2024-5", then create new feature called year_month
df_monthly = df_c1.withColumn("year_month", concat_ws("-", year("InvoiceDateOnly"), month("InvoiceDateOnly")))

# Step 2: Group by Customer ID and year_month, count purchases
df_monthly_count = df_monthly.groupBy("Customer ID", "year_month").agg(count("Invoice").alias("monthly_purchases"))

# Step 3: Pivot year_month to wide format
# Pivot "turns" rows into columns
df_monthly_pivot = df_monthly_count.groupBy("Customer ID").pivot("year_month").sum("monthly_purchases")

# Step 4: Fill nulls with 0 (meaning no purchases that month)
df_monthly_pivot = df_monthly_pivot.fillna(0)

# Step 5: Join with customer_df_final
df_clustering = customer_df_final.join(df_monthly_pivot, on="Customer ID", how="left")
```

▸ df_clustering: pyspark.sql.dataframe.DataFrame = [Customer ID: integer, num_invoices: long ... 34 more fields]
▸ df_monthly: pyspark.sql.dataframe.DataFrame = [Invoice: string, StockCode: string ... 13 more fields]
▸ df_monthly_count: pyspark.sql.dataframe.DataFrame = [Customer ID: integer, year_month: string ... 1 more field]
▸ df_monthly_pivot: pyspark.sql.dataframe.DataFrame = [Customer ID: integer, 2022-12: long ... 24 more fields]

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

```
Number of rows: 9443
```

After processing, we have 9,443 rows in our DataFrame for clustering.

```
df_clustering.limit(10).display()
```

Table

6.2. Exporting CSV File of Clustering DataFrame

```
df_pandas = df_clustering.toPandas()
```

```
/databricks/spark/python/pyspark/sql/pandas/utils.py:124: UserWarning: The conversion of DecimalType columns is inefficient and may take a long time. Column names: [total_price, avg_unit_price] If those columns are not necessary, you may consider dropping them or converting to primitive types before the conversion.
  warnings.warn(
```

```
df_pandas.to_csv("/tmp/df_clustering.csv", index=False)
```

```
Out[38]: True
```