Python Highlights — Data Cleaning & Load to PostgreSQL

Case Study 5 · Customer Shopping Behaviour Analysis · Python (pandas) → PostgreSQL

Profile the raw file, impute missing ratings category-wise, engineer features, prove a column redundant before dropping it, then load to Postgres via SQLAlchemy.

> *Public retail dataset used for portfolio purposes — no client or confidential data. Database credentials are read from environment variables, and environment-specific install logs have been omitted.*

IN [1]

```
!pip install pandas openpyxl
```

IN [2]

```
import pandas as pd

df = pd.read_excel('customer_shopping_behavior_1.xlsx')
df.head()
```

	Customer ID	Age	Gender	Item Purchased	Category	Purchase Amount (USD)	Location	Size	Color	Season	Review Rating	Subscription Status	Shipping Type	Discount Applied	Promo Code Used
0	1	55	Male	Blouse	Clothing	53	Kentucky	L	Gray	Winter	3.1	Yes	Express	Yes	Yes
1	2	19	Male	Sweater	Clothing	64	Maine	L	Maroon	Winter	3.1	Yes	Express	Yes	Yes
2	3	50	Male	Jeans	Clothing	73	Massachusetts	S	Maroon	Spring	3.1	Yes	Free Shipping	Yes	Yes
3	4	21	Male	Sandals	Footwear	90	Rhode Island	M	Maroon	Spring	3.5	Yes	Next Day Air	Yes	Yes
4	5	45	Male	Blouse	Clothing	49	Oregon	M	Turquoise	Spring	2.7	Yes	Free Shipping	Yes	Yes

IN [3]

```
df.info()
```

```
<class 'pandas.DataFrame'>
RangeIndex: 3900 entries, 0 to 3899
Data columns (total 18 columns):
 #   Column                  Non-Null Count  Dtype
---  ------                  --------------  -----
 0   Customer ID             3900 non-null   int64
 1   Age                     3900 non-null   int64
 2   Gender                  3900 non-null   str
 3   Item Purchased          3900 non-null   str
 4   Category                3900 non-null   str
 5   Purchase Amount (USD)   3900 non-null   int64
 6   Location                3900 non-null   str
 7   Size                    3900 non-null   str
 8   Color                   3900 non-null   str
 9   Season                  3900 non-null   str
 10  Review Rating           3863 non-null   float64
 11  Subscription Status     3900 non-null   str
 12  Shipping Type           3900 non-null   str
```

... output truncated

IN [4]

```
# df.describe() # Gives summary statistics for numerical columns only

df.describe(include='all') # Gives summary statistics for all columns
```

	Customer ID	Age	Gender	Item Purchased	Category	Purchase Amount (USD)	Location	Size	Color	Season	Review Rating	Subscription Status	Shipping Type	Discou Applie
count	3900.000000	3900.000000	3900	3900	3900	3900.000000	3900	3900	3900	3900	3863.000000	3900	3900	3900
unique	NaN	NaN	2	25	4	NaN	50	4	25	4	NaN	2	6	2
top	NaN	NaN	Male	Blouse	Clothing	NaN	Montana	M	Olive	Spring	NaN	No	Free Shipping	No
freq	NaN	NaN	2652	171	1737	NaN	96	1755	177	999	NaN	2847	675	2223
mean	1950.500000	44.068462	NaN	NaN	NaN	59.764359	NaN	NaN	NaN	NaN	3.750065	NaN	NaN	NaN

	Customer ID	Age	Gender	Item Purchased	Category	Purchase Amount (USD)	Location	Size	Color	Season	Review Rating	Subscription Status	Shipping Type	Discou Applie
std	1125.977353	15.207589	NaN	NaN	NaN	23.685392	NaN	NaN	NaN	NaN	0.716983	NaN	NaN	NaN
min	1.000000	18.000000	NaN	NaN	NaN	20.000000	NaN	NaN	NaN	NaN	2.500000	NaN	NaN	NaN
25%	975.750000	31.000000	NaN	NaN	NaN	39.000000	NaN	NaN	NaN	NaN	3.100000	NaN	NaN	NaN
50%	1950.500000	44.000000	NaN	NaN	NaN	60.000000	NaN	NaN	NaN	NaN	3.800000	NaN	NaN	NaN
75%	2925.250000	57.000000	NaN	NaN	NaN	81.000000	NaN	NaN	NaN	NaN	4.400000	NaN	NaN	NaN
max	3900.000000	70.000000	NaN	NaN	NaN	100.000000	NaN	NaN	NaN	NaN	5.000000	NaN	NaN	NaN

IN [5]

```
df.isnull().sum() # Checking missing values
```

```
Customer ID               0
Age                       0
Gender                    0
Item Purchased            0
Category                  0
Purchase Amount (USD)     0
Location                  0
Size                      0
Color                     0
Season                    0
Review Rating            37
Subscription Status       0
Shipping Type             0
Discount Applied          0
Promo Code Used           0
Previous Purchases        0
Payment Method            0
Frequency of Purchases    0
```

... output truncated

Use the median instead of the mean because the median is not affected by extreme values, and fill missing values category-wise to avoid bias.

Very simple explanation 👇

- **Mean** can change a lot if there are extreme values (outliers).
- **Median** is more stable and not affected much by outliers.

So, when filling missing ratings, we prefer **median**.

But instead of using one overall median for all products, we:

👉 Calculate the **median rating within each category** (like Clothing, Footwear, etc.)

👉 Fill missing values using that category's median

Why?

Because different categories have different rating patterns.

Using one overall median could mix them up and introduce bias.

In short:

Use category-wise median to fill missing values so the data stays fair and accurate.

IN [6]

```
df['Review Rating'] = df.groupby('Category')['Review Rating'].transform(lambda x: x.fillna(x.median()))
```

This code fills missing ratings using the median rating within each category.

Very briefly 👇

- `df` → your data table
- `groupby('Category')` → split data by category
- `['Review Rating']` → select rating column
- `transform(...)` → apply function but keep original shape
- `lambda x: x.fillna(x.median())` → replace missing values with that category's median
- `fillna()` in Python (Pandas) **replaces missing values (NaN) in a DataFrame or Series with a specified value or method** (e.g., 0, mean, forward fill).

IN [7]

```
df.isnull().sum()
```

```
Customer ID               0
Age                       0
Gender                    0
Item Purchased            0
Category                  0
Purchase Amount (USD)     0
Location                  0
Size                      0
Color                     0
Season                    0
Review Rating             0
Subscription Status       0
Shipping Type             0
Discount Applied          0
Promo Code Used           0
Previous Purchases        0
Payment Method            0
Frequency of Purchases    0
```

... output truncated

IN [8]

```
# Column names re-written with Snake Casing: Lower cased with underscores

df.columns = df.columns.str.lower()
df.columns = df.columns.str.replace(' ','_')
df = df.rename(columns={'purchase_amount_(usd)':'purchase_amount'})
```

IN [9]

```
df.columns
```

```
Index(['customer_id', 'age', 'gender', 'item_purchased', 'category',
       'purchase_amount', 'location', 'size', 'color', 'season',
       'review_rating', 'subscription_status', 'shipping_type',
       'discount_applied', 'promo_code_used', 'previous_purchases',
       'payment_method', 'frequency_of_purchases'],
      dtype='str')
```

IN [10]

```
# Create a column age_group

labels = ['Young Adult', 'Adult', 'Middle-aged', 'Senior']
df['age_group'] = pd.qcut(df['age'], q=4, labels=labels)

# qcut splits the ages into four equal-sized groups based on the data distribution and assigns the labels we
defined.
```

IN [11]

```
df[['age','age_group']].head(10)
```

	age	age_group
0	55	Middle-aged
1	19	Young Adult
2	50	Middle-aged
3	21	Young Adult
4	45	Middle-aged
5	46	Middle-aged
6	63	Senior
7	27	Young Adult
8	26	Young Adult
9	57	Middle-aged

IN [12]

```
# Create column purchase_frequency_days

frequency_mapping = {
    'Fortnightly': 14,
    'Weekly': 7,
```

```
    'Monthly': 30,
    'Quarterly': 90,
    'Bi-weekly': 14,
    'Annually': 365,
    'Every 3 months': 90
}

df['purchase_frequency_days'] = df['frequency_of_purchases'].map(frequency_mapping)

# map() function is used to replace the text in the column with the corresponding number of days.
```

IN [13]

```
df[['purchase_frequency_days','frequency_of_purchases']].head(10)
```

	purchase_frequency_days	frequency_of_purchases
0	14.0	Fortnightly
1	14.0	Fortnightly
2	7.0	Weekly
3	7.0	Weekly
4	365.0	Annually
5	7.0	Weekly
6	90.0	Quarterly
7	7.0	Weekly
8	365.0	Annually
9	90.0	Quarterly

IN [14]

```
df[['discount_applied','promo_code_used']].head(10)
```

	discount_applied	promo_code_used
0	Yes	Yes
1	Yes	Yes
2	Yes	Yes
3	Yes	Yes
4	Yes	Yes
5	Yes	Yes
6	Yes	Yes
7	Yes	Yes
8	Yes	Yes
9	Yes	Yes

> ### Discount Applied vs Promo Code Used
>
> We have two columns:
>
> - `Discount Applied`
> - `Promo Code Used`
>
> At first glance, these columns may seem identical.
>
> If a promo code is used, then a discount is obviously applied.
>
> However, businesses sometimes apply discounts **without** promo codes.
>
> For example:
>
> - Automatic seasonal sales
> - Exclusive member discounts
> - Limited-time promotional offers
>
> This raises an important question:
>
> - Do we actually need both columns?
> - Do both columns contain the same values?
> - Is one of them redundant?

> Before removing any column, we should carefully compare them to understand whether they provide unique information.

IN [15]

```
(df['discount_applied'] == df['promo_code_used']).all()
```

```
np.True_
```

> **Checking for Redundant Columns**
>
> The comparison result came out to be **True**.
>
> This means both columns contain exactly the same information.
>
> Since `Discount Applied` and `Promo Code Used` carry identical values, keeping both is unnecessary.
>
> To avoid redundancy and keep the dataset clean, we can safely remove the `Promo Code Used` column.

IN [16]

```
df = df.drop('promo_code_used', axis=1)
```

IN [17]

```
df.columns
```

```
Index(['customer_id', 'age', 'gender', 'item_purchased', 'category',
       'purchase_amount', 'location', 'size', 'color', 'season',
       'review_rating', 'subscription_status', 'shipping_type',
       'discount_applied', 'previous_purchases', 'payment_method',
       'frequency_of_purchases', 'age_group', 'purchase_frequency_days'],
      dtype='str')
```

IN [18]

```
pip install psycopg2-binary sqlalchemy
```

IN [19]

```
from sqlalchemy import create_engine

# Step 1: Connect to PostgreSQL
# Replace placeholders with your actual details
username = os.getenv("PGUSER", "postgres")
password = os.getenv("PGPASSWORD")   # redacted
host = "localhost"
port = "5432"
database = "customer_behavior_analysis"

engine = create_engine(f"postgresql+psycopg2://{username}:{password}@{host}:{port}/{database}")

# Step 2: Load DataFrame into PostgreSQL
table_name = "customer"
df.to_sql(table_name, engine, if_exists="replace", index=False)

print(f"Data successfuly loaded into table '{table_name}' in database '{database}'.")
```

```
Data successfuly loaded into table 'customer' in database 'customer_behavior_analysis'.
```

> **`from sqlalchemy import create_engine`**
>
> Imports the **create_engine function** from the SQLAlchemy library, which is used to **create a connection between Python and a database**.
>
> **`engine = create_engine(f"postgresql+psycopg2://{username}:{password}@{host}:{port}/{database}")`**
>
> Creates a **database connection object (engine)** using SQLAlchemy.
>
> This connection string contains:
>
> - `postgresql+psycopg2` → database type and driver
> - `username:password` → login credentials
> - `host:port` → database location

- `database` → the database name

So this line **connects Python to PostgreSQL**.

`df.to_sql(table_name, engine, if_exists="replace", index=False)`

Writes the **Pandas DataFrame `df` into the PostgreSQL table**.

Parameters:

- `table_name` → name of the SQL table
- `engine` → database connection
- `if_exists="replace"` → if the table already exists, **delete and recreate it**
- `index=False` → **do not store the DataFrame index as a column**

IN [20]

```
from sqlalchemy import text

with engine.connect() as conn:
    print("Current DB:", conn.execute(text("SELECT current_database();")).scalar())
    print("Current user:", conn.execute(text("SELECT current_user;")).scalar())
    print("Current schema:", conn.execute(text("SELECT current_schema();")).scalar())

    tables = conn.execute(text("""
        SELECT table_schema, table_name
        FROM information_schema.tables
        WHERE table_type='BASE TABLE'
          AND table_schema NOT IN ('pg_catalog','information_schema')
        ORDER BY table_schema, table_name;
    """)).fetchall()

tables
```

```
Current DB: customer_behavior_analysis
Current user: postgres
Current schema: public
```

```
[('public', 'customer')]
```

IN [21]

```
df.to_sql("customer", engine, schema="public", if_exists="replace", index=False)
```

```
900
```

IN [22]

```
from sqlalchemy import text
with engine.connect() as conn:
    print(conn.execute(text("SHOW port;")).scalar())
    print(conn.execute(text("SELECT inet_server_addr(), inet_server_port();")).fetchone())
```

```
5432
('::1', 5432)
```