Commercial Revenue Leakage Analysis

A business needed help to understand the reason it was struggling with profit dips, although it made timely turnovers every month, over the last year. A deep dive into how revenue leakages were happening was uncovered by identifying under-billing, pricing errors and improperly applied discounts using Python-based statistical analysis.

Import the libraries

Before cleaning and analysis, download the libraries needed for the analysis.

In [214…

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
```

The business dataset contains 1000 records.

Using Python to do cleaning and normalisation tasks.

In [177…

```
df = pd.read_csv('commercial_revenue_leakage.csv')
df.head()
```

Out[177…

	Transaction_ID	Date	Product_SKU	Quantity_Sold	Standard_Price	Actual_Price_Chargec
0	TXN-1000	2025-04-13	SKU-C	6	500	35(
1	TXN-1001	2025-12-15	SKU-C	1	500	50(
2	TXN-1002	2025-09-28	SKU-A	23	100	10(
3	TXN-1003	2025-04-17	SKU-B	5	250	25(
4	TXN-1004	2025-03-13	SKU-B	27	250	17!

Tansaction_ID has been set as the index, as it is not needed in the analysis.

Step 1: Exploratory Data Analysis (EDA)

In [178…

```
# Check the number of rows, columns of the dataset
df.shape
```

Out[178… (1000, 8)

In [179…
```
# Check if the dataset has null values in any of the columns
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 1000 entries, 0 to 999
Data columns (total 8 columns):
 #   Column                Non-Null Count  Dtype  
---  ------                --------------  -----  
 0   Transaction_ID        1000 non-null   object 
 1   Date                  1000 non-null   object 
 2   Product_SKU           950 non-null    object 
 3   Quantity_Sold         1000 non-null   int64  
 4   Standard_Price        1000 non-null   int64  
 5   Actual_Price_Charged  1000 non-null   int64  
 6   Discount_Applied      1000 non-null   float64
 7   Payment_Status        1000 non-null   object 
dtypes: float64(1), int64(3), object(4)
memory usage: 62.6+ KB
```

In [180…
```
# Check how the null values in the 'Product_SKU' column are represented
df.sample(5)
```

Out[180…

	Transaction_ID	Date	Product_SKU	Quantity_Sold	Standard_Price	Actual_Price_Charg
44	TXN-1044	2025-01-21	SKU-C	10	500	5
876	TXN-1876	2025-04-23	SKU-D	28	1000	1(
690	TXN-1690	2025-06-13	SKU-B	20	250	2
37	TXN-1037	2025-12-30	SKU-A	-3	100	'
700	TXN-1700	2025-08-16	SKU-C	27	500	3

There are 50 rows in the Product_SKU column that have missing values that need to be handled.

Note: since the Product_SKU is an object datatype, mode or Nan, will be used to fill in the missing values instead of deleting those rows.

The date column datatype also needs to be corrected to a datetime datatype.

In [181…
```
# Check the statistical metrics of the dataset
df.describe()
```

Out[181...

	Quantity_Sold	Standard_Price	Actual_Price_Charged	Discount_Applied
count	1000.000000	1000.00000	1000.000000	1000.000000
mean	21.442000	444.35000	423.950000	0.437500
std	15.865319	334.68282	325.348658	0.562957
min	-5.000000	100.00000	70.000000	0.000000
25%	8.000000	100.00000	100.000000	0.050000
50%	21.000000	250.00000	250.000000	0.100000
75%	35.000000	500.00000	500.000000	0.500000
max	49.000000	1000.00000	1000.000000	1.500000

In [182...

```
# Check for the most common Product_SKU code
df['Product_SKU'].mode()
```

Out[182...

```
0    SKU-A
Name: Product_SKU, dtype: object
```

In [ ]:

```
# Replace the Nan's in the Product_SKU column with the mode
df['Product_SKU'].replace(np.nan,'SKU-A', inplace=True)
```

In [184...

```
# Check if the Nan's have been handled
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 1000 entries, 0 to 999
Data columns (total 8 columns):
 #   Column                Non-Null Count  Dtype  
---  ------                --------------  -----  
 0   Transaction_ID        1000 non-null   object 
 1   Date                  1000 non-null   object 
 2   Product_SKU           1000 non-null   object 
 3   Quantity_Sold         1000 non-null   int64  
 4   Standard_Price        1000 non-null   int64  
 5   Actual_Price_Charged  1000 non-null   int64  
 6   Discount_Applied      1000 non-null   float64
 7   Payment_Status        1000 non-null   object 
dtypes: float64(1), int64(3), object(4)
memory usage: 62.6+ KB
```

In [185...

```
# Format the date 'object' datatype to 'datetime' datatype
df['Date'] = pd.to_datetime(df['Date'], format='%Y-%m-%d')
```

In [186...

```
# Check if the date is now a datetime datatype
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 1000 entries, 0 to 999
Data columns (total 8 columns):
 #   Column                Non-Null Count  Dtype
---  ------                --------------  -----
 0   Transaction_ID        1000 non-null   object
 1   Date                  1000 non-null   datetime64[ns]
 2   Product_SKU           1000 non-null   object
 3   Quantity_Sold         1000 non-null   int64
 4   Standard_Price        1000 non-null   int64
 5   Actual_Price_Charged  1000 non-null   int64
 6   Discount_Applied      1000 non-null   float64
 7   Payment_Status        1000 non-null   object
dtypes: datetime64[ns](1), float64(1), int64(3), object(3)
memory usage: 62.6+ KB
```

In [187…

```
# Check if there are any Nan's remaining in the dataset
(df == np.nan).sum()
```

Out[187…

```
Transaction_ID          0
Date                    0
Product_SKU             0
Quantity_Sold           0
Standard_Price          0
Actual_Price_Charged    0
Discount_Applied        0
Payment_Status          0
dtype: int64
```

In [188…

```
# Change the 'Standard_Price', 'Actual_Price_Charged' columns from int to float

df[['Standard_Price', 'Actual_Price_Charged']] = df[['Standard_Price', 'Actual_Pric
```

In [189…

```
# Check if the datatypes have changed
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 1000 entries, 0 to 999
Data columns (total 8 columns):
 #   Column                Non-Null Count  Dtype
---  ------                --------------  -----
 0   Transaction_ID        1000 non-null   object
 1   Date                  1000 non-null   datetime64[ns]
 2   Product_SKU           1000 non-null   object
 3   Quantity_Sold         1000 non-null   int64
 4   Standard_Price        1000 non-null   float64
 5   Actual_Price_Charged  1000 non-null   float64
 6   Discount_Applied      1000 non-null   float64
 7   Payment_Status        1000 non-null   object
dtypes: datetime64[ns](1), float64(3), int64(1), object(3)
memory usage: 62.6+ KB
```

In [ ]:

```
# Handle the 'paid' in the Payment_Status column to 'Paid'
df['Payment_Status'].replace('paid', 'Paid', inplace=True)
```

In [191…

```
df.head()
```

Out[191…

	Transaction_ID	Date	Product_SKU	Quantity_Sold	Standard_Price	Actual_Price_Chargec
0	TXN-1000	2025-04-13	SKU-C	6	500.0	350.(
1	TXN-1001	2025-12-15	SKU-C	1	500.0	500.(
2	TXN-1002	2025-09-28	SKU-A	23	100.0	100.(
3	TXN-1003	2025-04-17	SKU-B	5	250.0	250.(
4	TXN-1004	2025-03-13	SKU-B	27	250.0	175.(

In [192…

```
# Differentitate the raw dataset and the cleaned dataset
def clean_pipeline(df):
    # Example of cleaning steps
    df = df.dropna()   # removing missing values
    df = df[df['Quantity_Sold'] >= 0]    # handling negative quantities
    df['Product_SKU'] = df['Product_SKU'].replace(['Nan', None], 'SKU-A')
    return df

# Now you can call it
df_raw = df.copy()
df_cleaned = clean_pipeline(df_raw)
df = df_cleaned
df_cleaned.head()
```

Out[192…

	Transaction_ID	Date	Product_SKU	Quantity_Sold	Standard_Price	Actual_Price_Chargec
0	TXN-1000	2025-04-13	SKU-C	6	500.0	350.(
1	TXN-1001	2025-12-15	SKU-C	1	500.0	500.(
2	TXN-1002	2025-09-28	SKU-A	23	100.0	100.(
3	TXN-1003	2025-04-17	SKU-B	5	250.0	250.(
4	TXN-1004	2025-03-13	SKU-B	27	250.0	175.(

Now that the dataset is cleaned, analysis can start

Step 2: Analysis

Pricing Errors

Handle negative values in the 'Quantity_Sold' column. This is one of the primary issues the business is dealing with.

First, detect the negative quantities.

In [194…

```
# Count the number of negative quantities.
df[df['Quantity_Sold'] > 0].value_counts().sample(10)
```

Out[194…

```
Transaction_ID  Date        Product_SKU  Quantity_Sold  Standard_Price  Actual_Pri
ce_Charged  Discount_Applied  Payment_Status
TXN-1233        2025-04-07  SKU-C        22             500.0           500.0
0.5               Pending           1
TXN-1736        2025-06-26  SKU-B        17             250.0           250.0
0.0               Cancelled         1
TXN-1914        2025-01-25  SKU-C        3              500.0           500.0
0.1               Pending           1
TXN-1135        2025-12-12  SKU-B        10             250.0           250.0
0.5               Cancelled         1
TXN-1136        2025-02-11  SKU-A        48             100.0           100.0
1.5               Paid              1
TXN-1451        2025-04-28  SKU-A        35             100.0           100.0
0.5               pd                1
TXN-1435        2025-04-02  SKU-B        43             250.0           250.0
0.5               Cancelled         1
TXN-1835        2025-06-04  SKU-D        10             1000.0          1000.0
0.5               Paid              1
TXN-1213        2025-05-06  SKU-A        40             100.0           100.0
0.5               Paid              1
TXN-1117        2025-08-04  SKU-C        26             500.0           500.0
0.1               Cancelled         1
Name: count, dtype: int64
```

Problem solving

The negative values look like legitimate transactions.

I'll now use SQL to analyse pricing errors, under-billing and incorrectly applied discounts.

I'll also use MySQL Workbench to connect to the both the raw dataset and cleaned dataset for analysis.

Connecting to SQL & Analysis

Downloading and connecting to SQL, including connecting to MySQL Workbench through the commercial_revenue_leakage_db database.

In [ ]:
```
pip install mysql-connector-python sqlalchemy
```

In [195…
```
from sqlalchemy import create_engine

# Replace with your Workbench connection details
engine = create_engine("mysql+mysqlconnector://username:password@localhost:port /co
```

In [196…
```
# This is the cleaned dataset
df_cleaned.to_sql("commercial_revenue_leakage_data", con=engine, if_exists="replace
```

Out[196…
```
-1
```

In [197…
```
# This is the raw, messy dataset
df.to_sql("c_r_l_data", con=engine, if_exists="replace", index=False)
```

Out[197…
```
-1
```

In [198…
```
DATABASE_URL="mysql+mysqlconnector://username:password@localhost:port/commercial_re
```

In [199…
```
%sql mysql+mysqlconnector://username:password@localhost:port/commercial_revenue_lea
```

In [200…
```
%sql SELECT * FROM revenue_query1 LIMIT 5;
```

```
 * mysql+mysqlconnector://root:***@localhost:3306/commercial_revenue_leakage_db
4 rows affected.
```

Out[200…

Product_SKU	revenuetotalss
SKU-C	1318075.0
SKU-A	468710.0
SKU-B	491600.0
SKU-D	1853250.0

Confirm that the dataset has no missing values.

In [201…
```
%%sql
SELECT
    *
FROM
    commercial_revenue_leakage_db.c_r_l_data
WHERE Product_SKU IS NULL
LIMIT 10;
```

```
 * mysql+mysqlconnector://root:***@localhost:3306/commercial_revenue_leakage_db
0 rows affected.
```

Out[201…

Transaction_ID	Date	Product_SKU	Quantity_Sold	Standard_Price	Actual_Price_Charged	D

Find the difference between the raw df and cleaned df pinpointing where pricing errors.

revenuetotalss -- represents the cleaned dataset revenue.

totalrevenue -- represents the raw dataset revenue.

In [202…

```
%%sql
SELECT
    q1.Product_SKU,
    q1.revenuetotalss,
    q2.totalrevenue,
    (q1.revenuetotalss - q2.totalrevenue) AS difference
FROM revenue_query1 AS q1
LEFT JOIN revenue_query2 AS q2
    ON q1.Product_SKU = q2.Product_SKU
ORDER BY q1.Product_SKU;
```

```
 * mysql+mysqlconnector://root:***@localhost:3306/commercial_revenue_leakage_db
4 rows affected.
```

Out[202…

Product_SKU	revenuetotalss	totalrevenue	difference
SKU-A	468710.0	450365.0	18345.0
SKU-B	491600.0	463681.25	27918.75
SKU-C	1318075.0	1258412.5	59662.5
SKU-D	1853250.0	1761630.0	91620.0

In [203…

```
%%sql
SELECT
    SUM(q1.revenuetotalss - q2.totalrevenue) AS Pricingerror_totals
FROM revenue_query1 AS q1
LEFT JOIN revenue_query2 AS q2
    ON q1.Product_SKU = q2.Product_SKU
ORDER BY q1.Product_SKU;
```

```
 * mysql+mysqlconnector://root:***@localhost:3306/commercial_revenue_leakage_db
1 rows affected.
```

Out[203…

Pricingerror_totals
197546.25

The total amount of revenue loss caused by pricing errors is $197,546.25 (revenue in dollars).

Mis-application of Discounts

Dicounts were capped at 1.5% for the year. Any amount more than 1.5% is a mis-application that will be stored as a 'view' for future referrence.

In [204…

```
%%sql
SELECT
    Transaction_ID,
    Product_SKU,
    Quantity_Sold,
```

```
    Standard_Price,
    Actual_Price_Charged,
    (Quantity_Sold * Standard_Price) as expectedrevenue,
    (Quantity_Sold * Actual_Price_Charged) as actualrevenue,
    ((Quantity_Sold * Standard_Price) - (Quantity_Sold * Actual_Price_Charged))/100
FROM
    commercial_revenue_leakage_db.commercial_revenue_leakage_data
LIMIT 10;
```

```
 * mysql+mysqlconnector://root:***@localhost:3306/commercial_revenue_leakage_db
10 rows affected.
```

Out[204…

Transaction_ID	Product_SKU	Quantity_Sold	Standard_Price	Actual_Price_Charged	expectec
TXN-1000	SKU-C	6	500.0	350.0	
TXN-1001	SKU-C	1	500.0	500.0	
TXN-1002	SKU-A	23	100.0	100.0	
TXN-1003	SKU-B	5	250.0	250.0	
TXN-1004	SKU-B	27	250.0	175.0	
TXN-1005	SKU-D	0	1000.0	1000.0	
TXN-1006	SKU-B	41	250.0	250.0	
TXN-1007	SKU-A	9	100.0	100.0	
TXN-1008	SKU-C	26	500.0	350.0	
TXN-1009	SKU-B	45	250.0	175.0	

In [ ]:

```
%%sql
USE commercial_revenue_leakage_db;

CREATE VIEW Discount_Audit_View AS
SELECT
    Transaction_ID,
    Product_SKU,
    Quantity_Sold,
    Standard_Price,
    Actual_Price_Charged,
    (Quantity_Sold * Standard_Price) as expectedrevenue,
    (Quantity_Sold * Actual_Price_Charged) as actualrevenue,
    ((Quantity_Sold * Standard_Price) - (Quantity_Sold * Actual_Price_Charged))/100
FROM
    commercial_revenue_leakage_db.commercial_revenue_leakage_data
HAVING discountapplied > 1.5;
```

In [206…

```
%%sql
SELECT
    COUNT(*)
FROM
    Discount_Audit_View
WHERE discountapplied > 1.5
```

```
ORDER BY discountapplied DESC
LIMIT 10;
```

```
 * mysql+mysqlconnector://root:***@localhost:3306/commercial_revenue_leakage_db
1 rows affected.
```

Out[206...

COUNT(*)
126

In [233...

```
%%sql
SELECT
    *
FROM
    Discount_Audit_View
WHERE discountapplied > 1.5
ORDER BY discountapplied DESC
LIMIT 10;
```

```
 * mysql+mysqlconnector://root:***@localhost:3306/commercial_revenue_leakage_db
10 rows affected.
```

Out[233...

Transaction_ID	Product_SKU	Quantity_Sold	Standard_Price	Actual_Price_Charged	expectec
TXN-1423	SKU-D	49	1000.0	700.0	
TXN-1167	SKU-D	49	1000.0	700.0	
TXN-1487	SKU-A	48	1000.0	700.0	
TXN-1987	SKU-D	46	1000.0	700.0	
TXN-1766	SKU-D	45	1000.0	700.0	
TXN-1286	SKU-D	42	1000.0	700.0	
TXN-1162	SKU-D	40	1000.0	700.0	
TXN-1763	SKU-D	39	1000.0	700.0	
TXN-1269	SKU-D	33	1000.0	700.0	
TXN-1681	SKU-D	33	1000.0	700.0	

From the analysis it's clear to see there are 126 records that exceed the expected 1.5% discount. This signals actual revenue charged as significantly lower than the revenue baseline. It also ties in to the pricing errors earlier identified.

The discounts applied are above the allowable percentage, which could suggest gaps in pricing controls. This shows there were revenue leakages, though timely turnovers were reached, causing reduced profitability.

These records are to be flagged for audit to determine if they were authorised discounts, system errors or data entry mistakes.