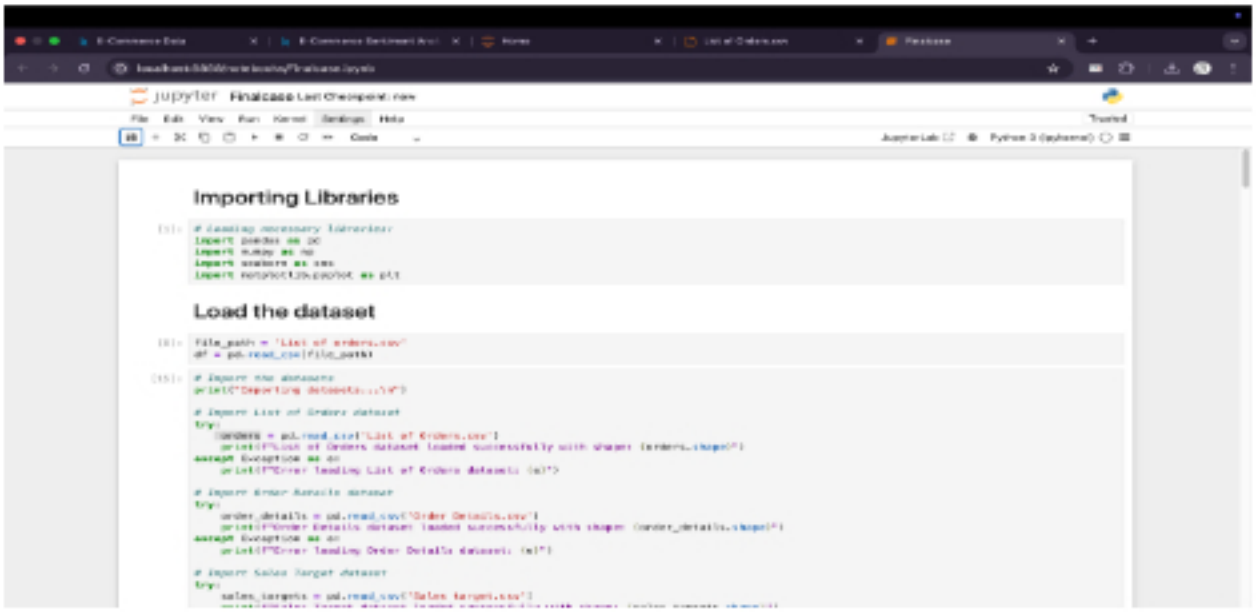
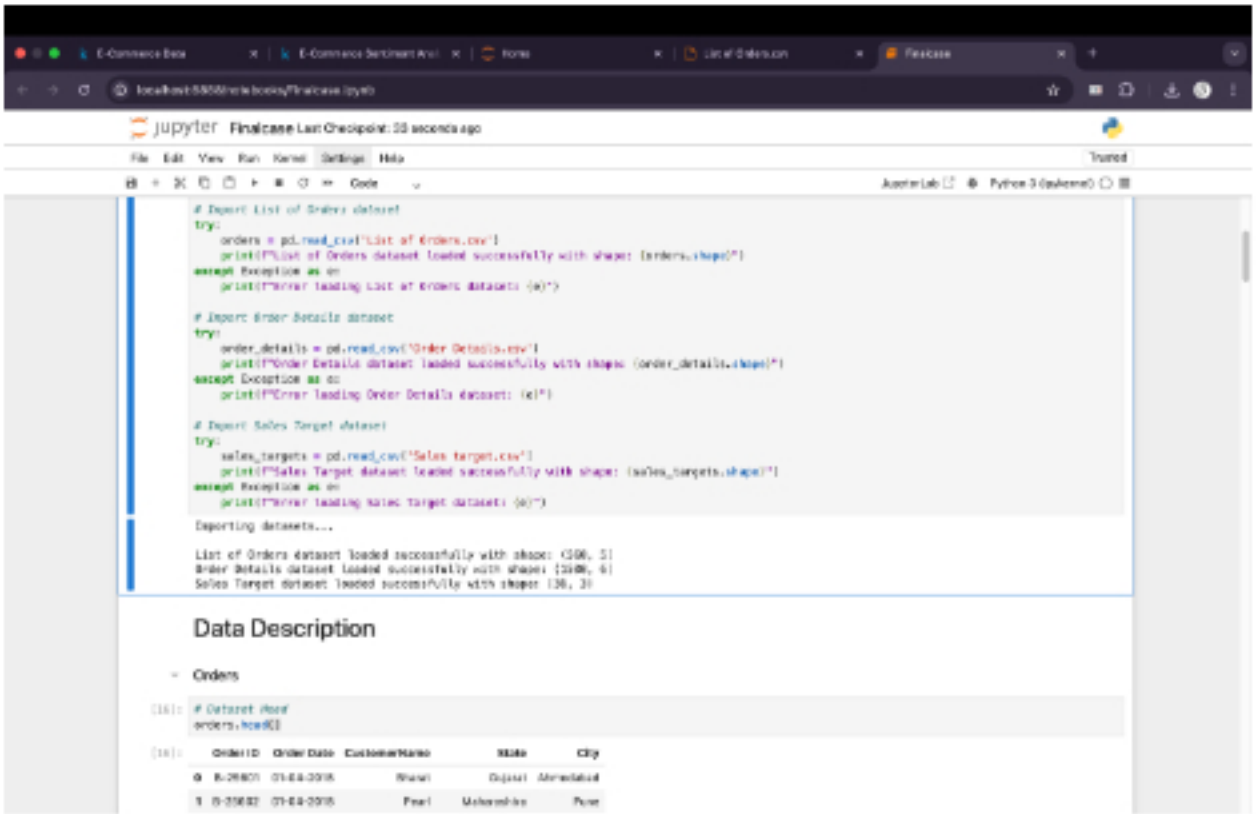
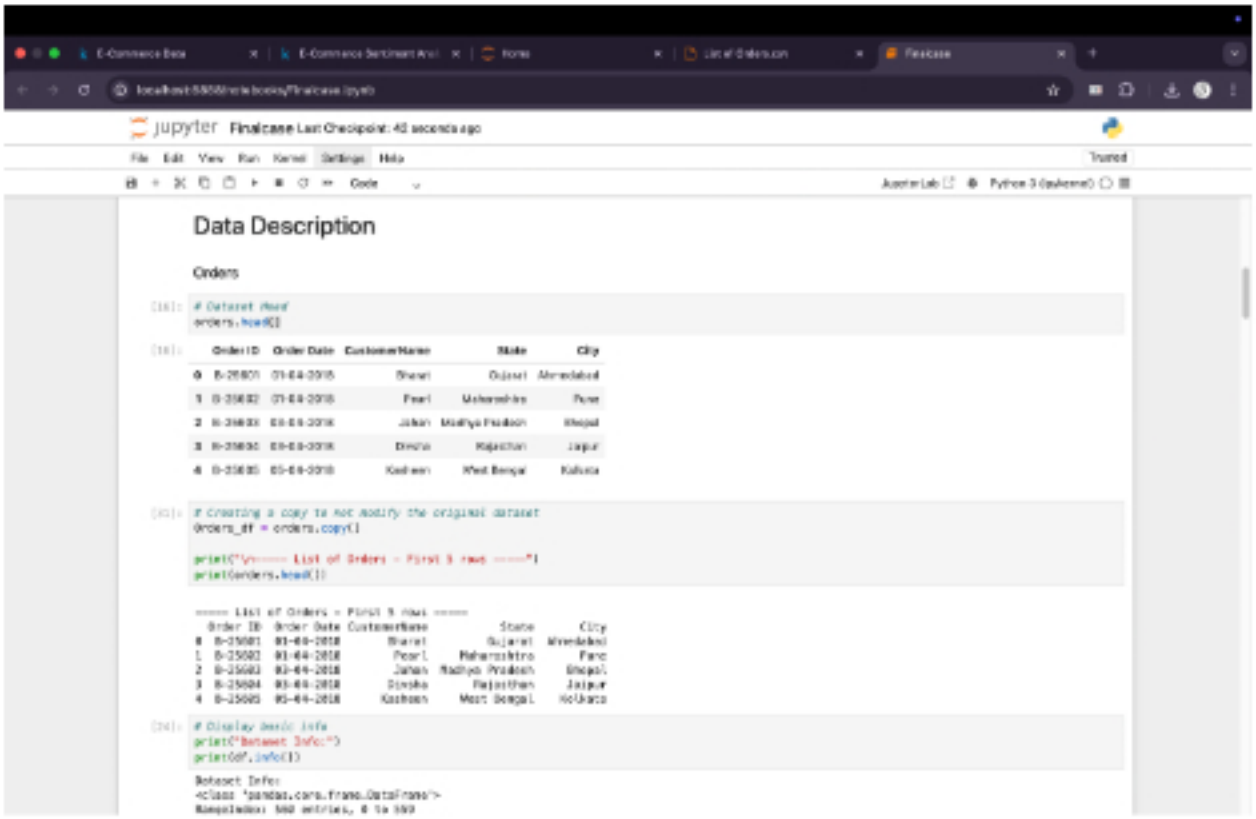
Target: The expected sales amount to be achieved (Float)

This dataset has no missing values

The relatively small size (36 rows) indicates these are likely monthly targets for each major product category







jupyter Finalcase Last Checkpoint: 40 seconds ago

File Edit View Run Kernel Settings Help

Trusted

Code

JupyterLab Python 3 (ipykernel)

Data Description

Orders

```
# Dataset Head
orders.head()
```

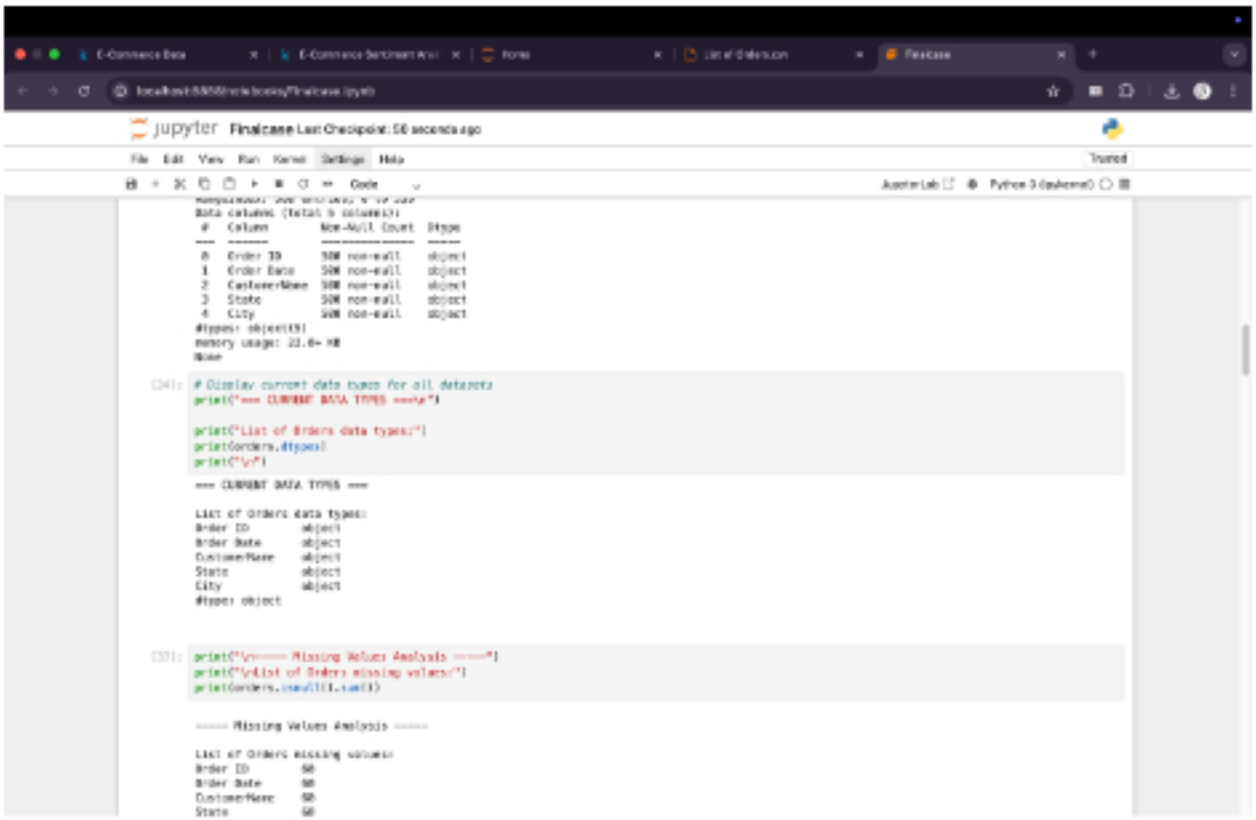
	Order ID	Order Date	CustomerName	State	City
0	B-25601	01-04-2018	Bharat	Gujarat	Ahmedabad
1	B-25602	01-04-2018	Pearl	Maharashtra	Pune
2	B-25603	01-04-2018	Jahan	Madhya Pradesh	Bhopal
3	B-25604	01-04-2018	Divsha	Rajasthan	Jaipur
4	B-25605	01-04-2018	Kashvan	West Bengal	Kolkata

```
# Creating a copy to not modify the original dataset
Orders_df = orders.copy()

print("\n===== List of Orders - First 5 rows =====")
print(orders.head())
```

```
===== List of Orders - First 5 rows =====
  Order ID Order Date CustomerName          State      City
0 B-25601  01-04-2018       Bharat        Gujarat Ahmedabad
1 B-25602  01-04-2018        Pearl    Maharashtra      Pune
2 B-25603  01-04-2018        Jahan  Madhya Pradesh    Bhopal
3 B-25604  01-04-2018       Divsha      Rajasthan    Jaipur
4 B-25605  01-04-2018     Kashvan    West Bengal   Kolkata
```

```
# Display basic info
print("Dataset Info:")
print(df.info())
```

```
Dataset Info:
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 500 entries, 0 to 499
```



jupyter Finalcase Last Checkpoint: 50 seconds ago

File Edit View Run Kernel Settings Help

Trusted

Code

JupyterLab Python 3 (ipykernel)

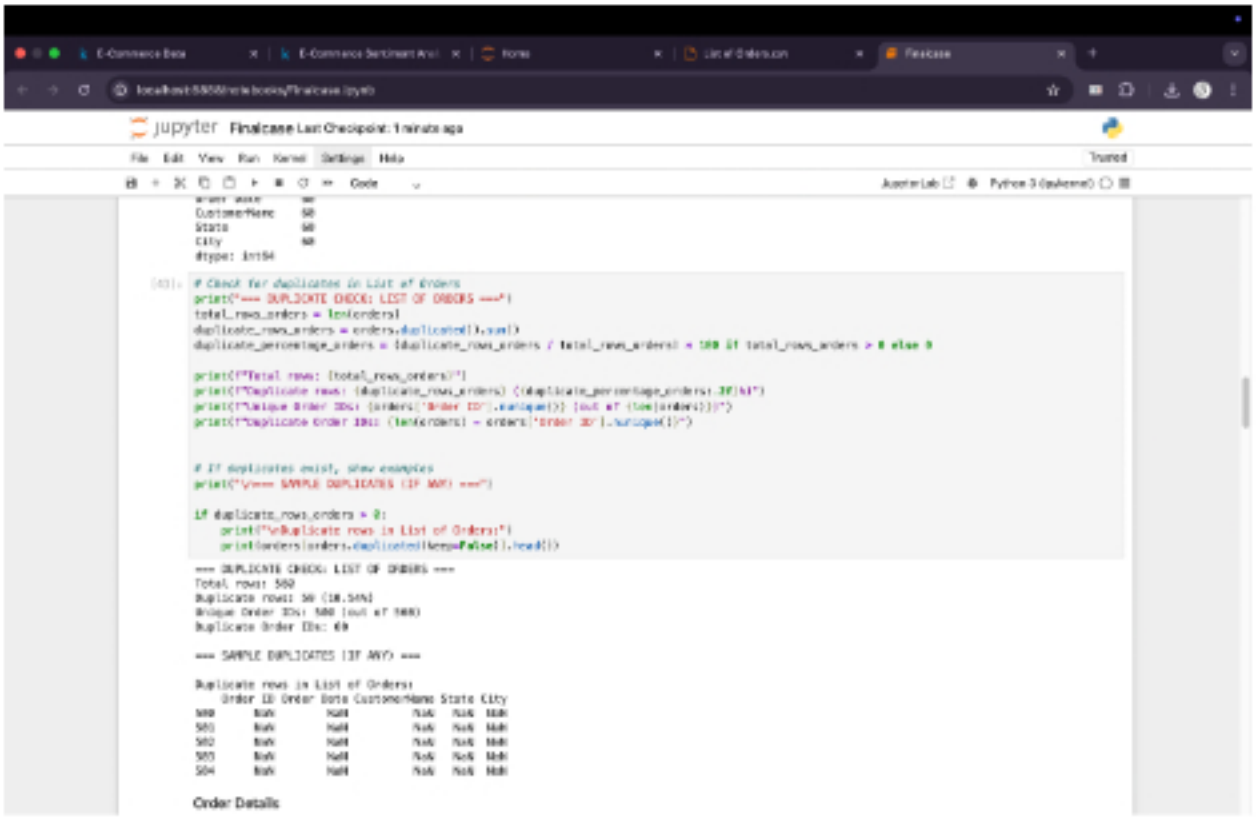
```
Data columns (total 5 columns):
 #   Column        Non-Null Count  Dtype
---  ------        --------------  -----
 0   Order ID      500 non-null    object
 1   Order Date    500 non-null    object
 2   CustomerName  500 non-null    object
 3   State         500 non-null    object
 4   City          500 non-null    object
dtypes: object(5)
memory usage: 23.6+ KB
None
```

```
# Display current data types for all datasets
print("=== CURRENT DATA TYPES ===\n")

print("List of Orders data types:")
print(orders.dtypes)
print("\n")
```

```
=== CURRENT DATA TYPES ===

List of Orders data types:
Order ID        object
Order Date      object
CustomerName    object
State           object
City            object
dtype: object
```

```
print("\n===== Missing Values Analysis =====")
print("\nList of Orders missing values:")
print(orders.isnull().sum())
```

```
===== Missing Values Analysis =====

List of Orders missing values:
Order ID        50
Order Date      50
CustomerName    50
State           50
```

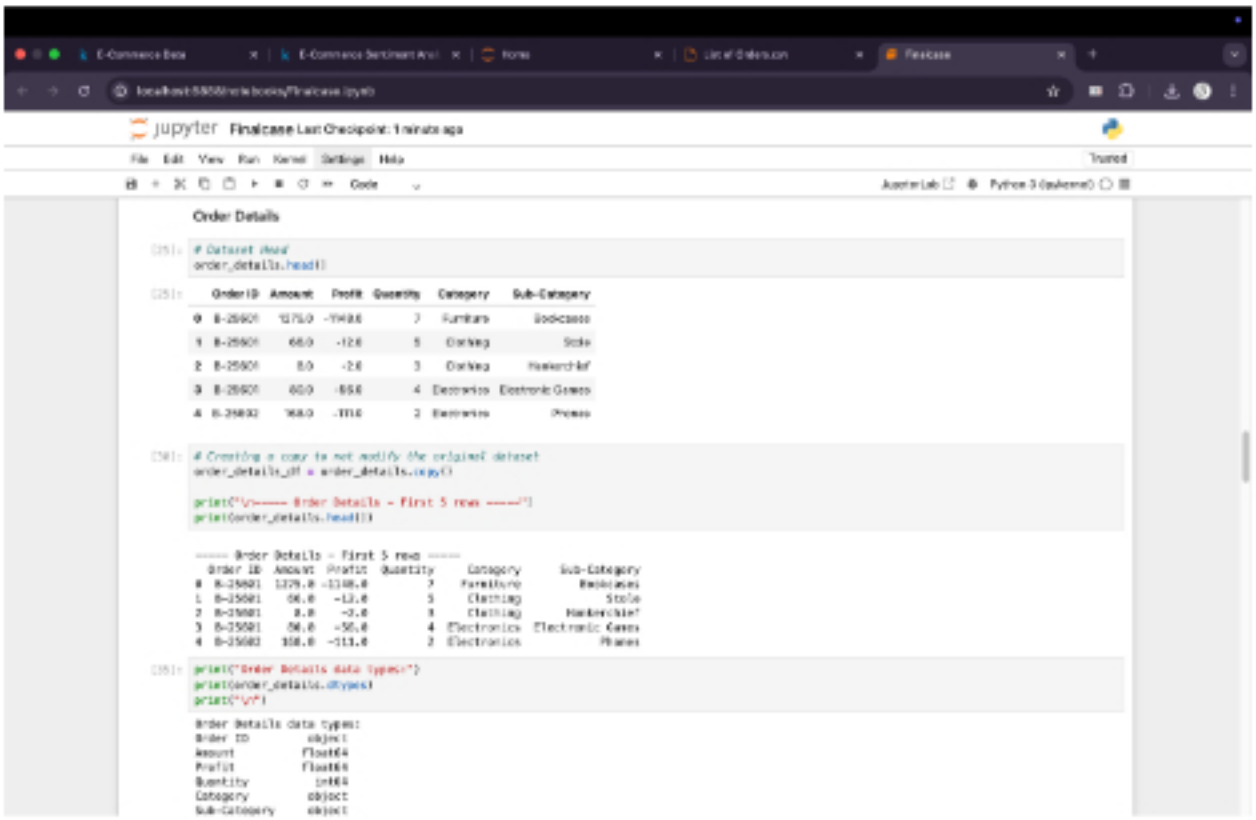


```
CustomerName    60
State           60
City            60
dtype: int64
```

```
# Check for duplicates in List of Orders
print("=== DUPLICATE CHECK: LIST OF ORDERS ===")
total_rows_orders = len(orders)
duplicate_rows_orders = orders.duplicated().sum()
duplicate_percentage_orders = (duplicate_rows_orders / total_rows_orders) * 100 if total_rows_orders > 0 else 0

print(f"Total rows: {total_rows_orders}")
print(f"Duplicate rows: {duplicate_rows_orders} ({duplicate_percentage_orders:.2f}%)")
print(f"Unique Order IDs: {orders['Order ID'].nunique()} (out of {len(orders)})")
print(f"Duplicate Order IDs: {len(orders) - orders['Order ID'].nunique()}")


# If duplicates exist, show examples
print("\n=== SAMPLE DUPLICATES (IF ANY) ===")

if duplicate_rows_orders > 0:
    print("\nDuplicate rows in List of Orders:")
    print(orders[orders.duplicated(keep=False)].head())
```

```
=== DUPLICATE CHECK: LIST OF ORDERS ===
Total rows: 500
Duplicate rows: 30 (10.54%)
Unique Order IDs: 500 (out of 500)
Duplicate Order IDs: 0

=== SAMPLE DUPLICATES (IF ANY) ===

Duplicate rows in List of Orders:
    Order ID Order Date CustomerName State City
500      NaN        NaT          NaN   NaN  NaN
501      NaN        NaT          NaN   NaN  NaN
502      NaN        NaT          NaN   NaN  NaN
503      NaN        NaT          NaN   NaN  NaN
504      NaN        NaT          NaN   NaN  NaN
```

Order Details



Order Details

```
# Dataset Head
order_details.head()
```

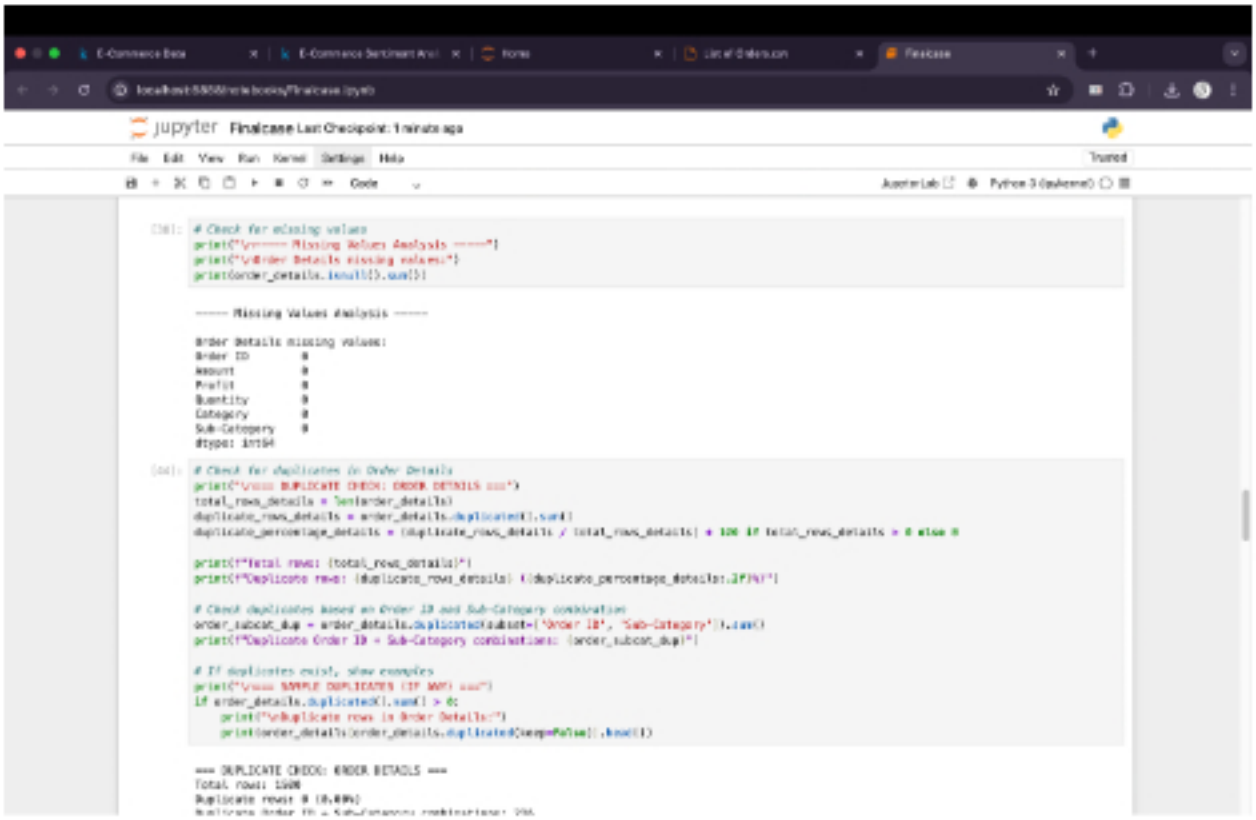
	Order ID	Amount	Profit	Quantity	Category	Sub-Category
0	B-25601	1275.0	-1148.0	7	Furniture	Bookcases
1	B-25601	66.0	-12.0	5	Clothing	Stole
2	B-25601	8.0	-2.0	3	Clothing	Handkerchief
3	B-25601	80.0	-65.0	4	Electronics	Electronic Games
4	B-25602	168.0	-111.0	2	Electronics	Phones

```
# Creating a copy to not modify the original dataset
order_details_df = order_details.copy()

print("\n----- Order Details - First 5 rows -----")
print(order_details.head())
```

```
----- Order Details - First 5 rows -----
  Order ID  Amount  Profit  Quantity     Category      Sub-Category
0  B-25601  1275.0 -1148.0         7    Furniture         Bookcases
1  B-25601    66.0   -12.0         5     Clothing             Stole
2  B-25601     8.0    -2.0         3     Clothing      Handkerchief
3  B-25601    80.0   -65.0         4  Electronics  Electronic Games
4  B-25602   168.0  -111.0         2  Electronics            Phones
```

```
print("Order Details data types:")
print(order_details.dtypes)
print("\n")
```

```
Order Details data types:
Order ID         object
Amount          float64
Profit          float64
Quantity          int64
Category         object
Sub-Category     object
```

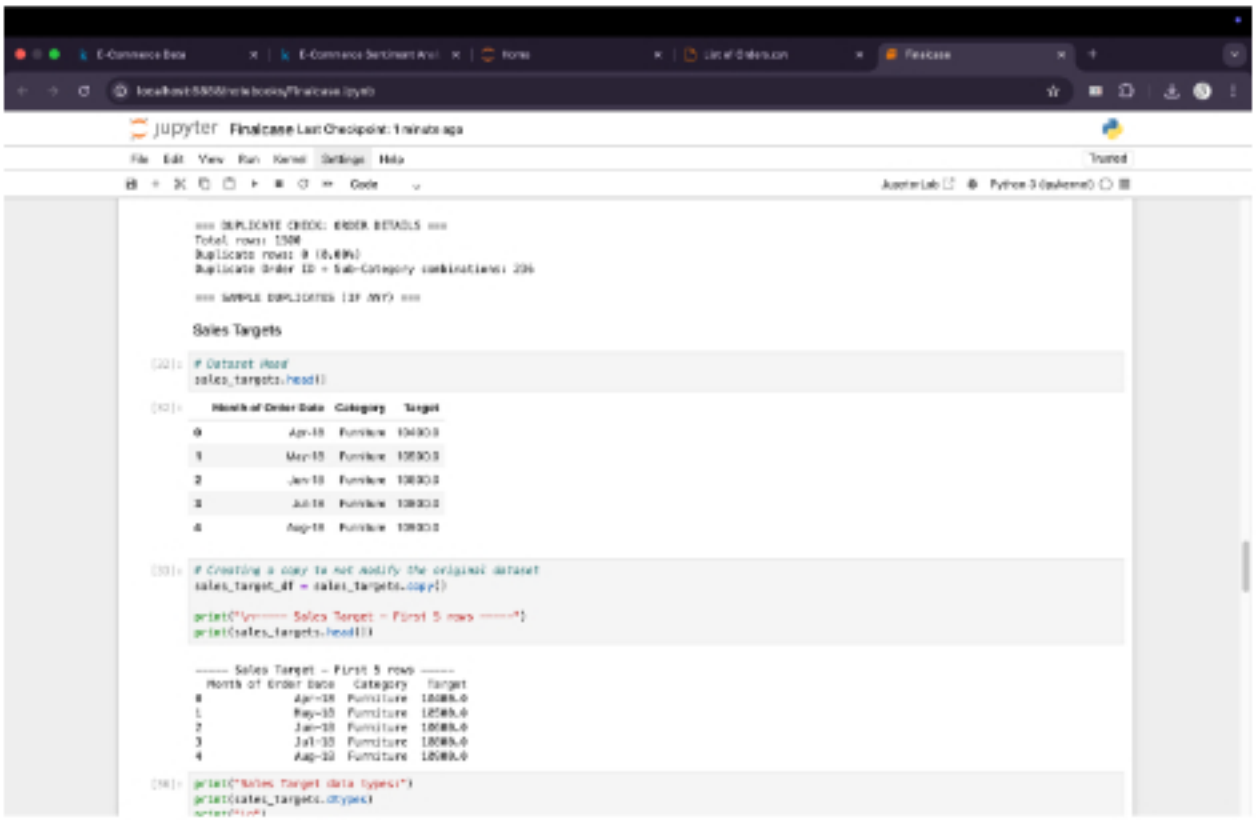


```
# Check for missing values
print("\n------ Missing Values Analysis ------")
print("\nOrder Details missing values:")
print(order_details.isnull().sum())
```

```
------ Missing Values Analysis ------

Order Details missing values:
Order ID        0
Amount          0
Profit          0
Quantity        0
Category        0
Sub-Category    0
dtype: int64
```

```
# Check for duplicates in Order Details
print("\n=== DUPLICATE CHECK: ORDER DETAILS ===")
total_rows_details = len(order_details)
duplicate_rows_details = order_details.duplicated().sum()
duplicate_percentage_details = (duplicate_rows_details / total_rows_details) * 100 if total_rows_details > 0 else 0

print(f"Total rows: {total_rows_details}")
print(f"Duplicate rows: {duplicate_rows_details} ({duplicate_percentage_details:.2f}%)")

# Check duplicates based on Order ID and Sub-Category combination
order_subcat_dup = order_details.duplicated(subset=['Order ID', 'Sub-Category']).sum()
print(f"Duplicate Order ID + Sub-Category combinations: {order_subcat_dup}")

# If duplicates exist, show examples
print("\n=== SAMPLE DUPLICATES (IF ANY) ===")
if order_details.duplicated().sum() > 0:
    print("\nDuplicate rows in Order Details:")
    print(order_details[order_details.duplicated(keep=False)].head())
```

```
=== DUPLICATE CHECK: ORDER DETAILS ===
Total rows: 1500
Duplicate rows: 0 (0.00%)
Duplicate Order ID + Sub-Category combinations: 236

=== SAMPLE DUPLICATES (IF ANY) ===
```



Sales Targets

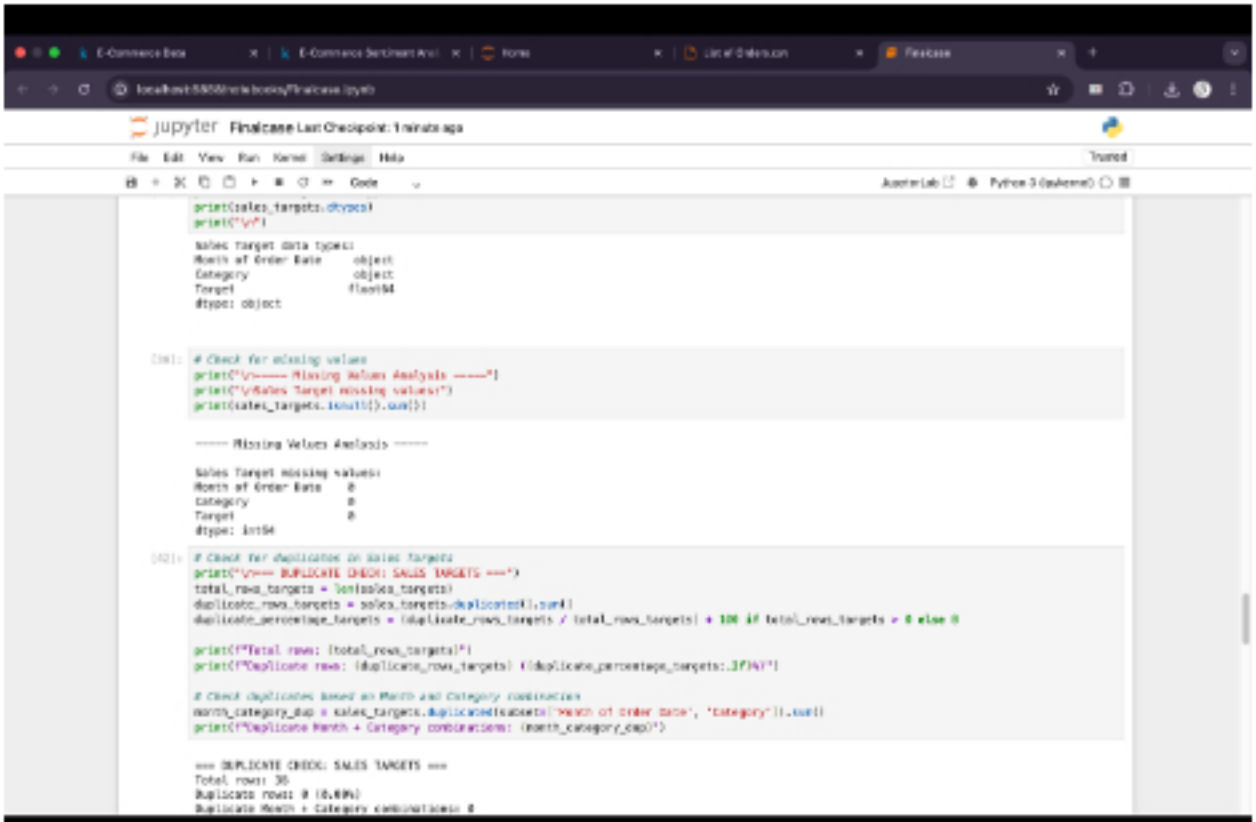
```
# Dataset Head
sales_targets.head()
```

	Month of Order Date	Category	Target
0	Apr-18	Furniture	104303
1	May-18	Furniture	109303
2	Jun-18	Furniture	108303
3	Jul-18	Furniture	108303
4	Aug-18	Furniture	108303

```
# Creating a copy to not modify the original dataset
sales_target_df = sales_targets.copy()

print("\n------ Sales Target - First 5 rows ------")
print(sales_targets.head())
```

```
------ Sales Target - First 5 rows ------
  Month of Order Date   Category   Target
0              Apr-18  Furniture  18000.0
1              May-18  Furniture  18000.0
2              Jun-18  Furniture  18000.0
3              Jul-18  Furniture  18000.0
4              Aug-18  Furniture  18000.0
```

```
print("Sales Target data types:")
print(sales_targets.dtypes)
```



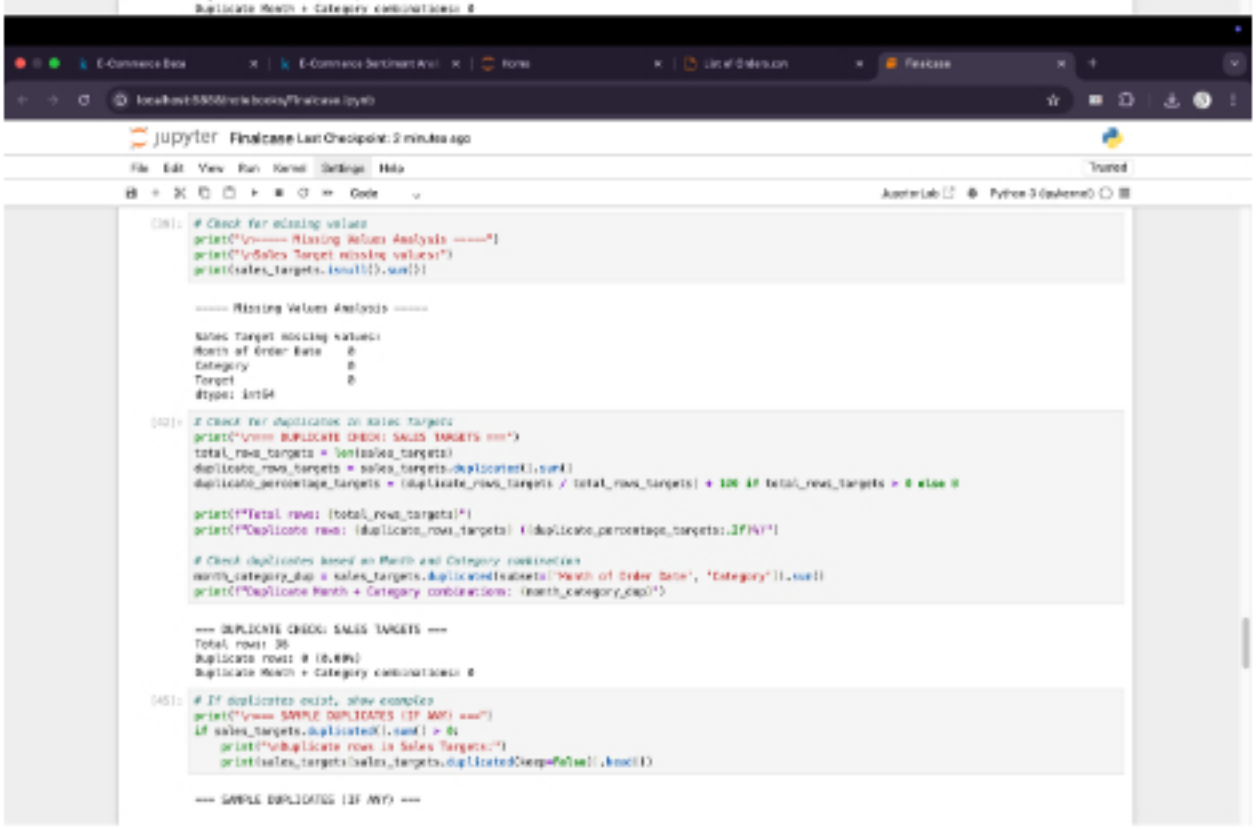
```
print(sales_targets.dtypes)
print("\n")

Sales Target data types:
Month of Order Date     object
Category                object
Target                 float64
dtype: object
```

```
# Check for missing values
print("\n===== Missing Values Analysis =====")
print("\nSales Target missing values:")
print(sales_targets.isnull().sum())
```

```
===== Missing Values Analysis =====

Sales Target missing values:
Month of Order Date    0
Category               0
Target                 0
dtype: int64
```

```
# Check for duplicates in Sales Targets
print("\n=== DUPLICATE CHECK: SALES TARGETS ===")
total_rows_targets = len(sales_targets)
duplicate_rows_targets = sales_targets.duplicated().sum()
duplicate_percentage_targets = (duplicate_rows_targets / total_rows_targets) * 100 if total_rows_targets > 0 else 0

print(f"Total rows: {total_rows_targets}")
print(f"Duplicate rows: {duplicate_rows_targets} ({duplicate_percentage_targets:.2f}%)")

# Check duplicates based on Month and Category combinations
month_category_dup = sales_targets.duplicated(subset=['Month of Order Date', 'Category']).sum()
print(f"Duplicate Month + Category combinations: {month_category_dup}")
```

```
=== DUPLICATE CHECK: SALES TARGETS ===
Total rows: 36
Duplicate rows: 0 (0.00%)
Duplicate Month + Category combinations: 0
```



```
# Check for missing values
print("\n===== Missing Values Analysis =====")
print("\nSales Target missing values:")
print(sales_targets.isnull().sum())
```

```
===== Missing Values Analysis =====

Sales Target missing values:
Month of Order Date    0
Category               0
Target                 0
dtype: int64
```

```
# Check for duplicates in Sales Targets
print("\n=== DUPLICATE CHECK: SALES TARGETS ===")
total_rows_targets = len(sales_targets)
duplicate_rows_targets = sales_targets.duplicated().sum()
duplicate_percentage_targets = (duplicate_rows_targets / total_rows_targets) * 100 if total_rows_targets > 0 else 0

print(f"Total rows: {total_rows_targets}")
print(f"Duplicate rows: {duplicate_rows_targets} ({duplicate_percentage_targets:.2f}%)")

# Check duplicates based on Month and Category combinations
month_category_dup = sales_targets.duplicated(subset=['Month of Order Date', 'Category']).sum()
print(f"Duplicate Month + Category combinations: {month_category_dup}")
```

```
=== DUPLICATE CHECK: SALES TARGETS ===
Total rows: 36
Duplicate rows: 0 (0.00%)
Duplicate Month + Category combinations: 0
```

```
# If duplicates exist, show examples
print("\n=== SAMPLE DUPLICATES (IF ANY) ===")
if sales_targets.duplicated().sum() > 0:
    print("\nDuplicate rows in Sales Targets:")
    print(sales_targets[sales_targets.duplicated(keep=False)].head())
```

```
=== SAMPLE DUPLICATES (IF ANY) ===
```

Data Pre-Processing:

1. Handling Null Values

Why is this important?

Missing or null values could have a wide impact on data validity and reliability in analysis.

- Effect on Statistical Analysis: Null values can play a role to affect calculations such as averages, summation, or other aggregates.
- Data Integrity: Unresolved nulls can lead to biased estimates and misinterpretation, especially if they are systematic instead of random in nature.
- Model Performance in Machine Learning: Nulls can taint training algorithms, decimating the predictive capability of models. Most machine learning algorithms do not inherently deal with missing data in an explicit way.
- Decision-Making: Erroneous data caused by unsolved nulls can lead to erroneous business decisions based on erroneous insight.

How to handle Null values?

- The percentage of null values in a particular column was calculated. If it is less than 50 %, then we can proceed with handling, otherwise, those rows are deleted.
- The result showed varying degrees of missing data in columns:
    - Percentage of missing values in each column:
        Order ID: 10.71%
        Order Date: 10.71%
        Customer Name : 10.71%
        State: 10.71%
        City: 10.71%

Pre Processing and Cleaning

Handling Null Data

Finding the null % of the values. If the % exceeds 60, then it's better to drop the rows.

[58]:
```
# Function to check and display missing values
def check_missing_values(df, dataset_name):
    print(f"\n------ Missing Values Analysis for {dataset_name} ------")

    # Count missing values
    missing_values = df.isnull().sum()

    # Calculate percentage of missing values
    null_percentages = (df.isnull().sum() / len(df) * 100).sort_values(ascending=False)

    # Create a summary DataFrame
    missing_summary = pd.DataFrame({
        'Missing Values': missing_values,
        'Percentage (%)': null_percentages
    })

    # Display only columns with missing values
    missing_cols = missing_summary[missing_summary['Missing Values'] > 0]

    if len(missing_cols) > 0:
        print(f"\n{dataset_name} missing values:")
        for col, row in missing_cols.iterrows():
            print(f"{col:<15} {int(row['Missing Values'])}")
        print(f"Dtypes {df.dtypes.iloc[0]}")
    else:
        print(f"\n{dataset_name} has no missing values.")

# Check missing values in all datasets
check_missing_values(orders, "List of Orders")
check_missing_values(order_details, "Order Details")
```

```
check_missing_values(order_details, "Order Details")
check_missing_values(sales_targets, "Sales Target")

# Additional overall summary
print("\n----- Overall Missing Values Summary -----")
datasets = {
    "List of Orders": orders,
    "Order Details": order_details,
    "Sales Target": sales_targets
}

for name, df in datasets.items():
    total_missing = df.isnull().sum().sum()
    total_elements = df.size
    pct_missing = (total_missing / total_elements) * 100
    print(f"{name}: {total_missing} missing values out of {total_elements} elements ({pct_missing:.2f}%)")
```

```
----- Missing Values Analysis for List of Orders -----

List of Orders missing values:
Order ID        60
Order Date      60
CustomerName    60
State           60
City            60
dtype: object

----- Missing Values Analysis for Order Details -----

Order Details has no missing values.

----- Missing Values Analysis for Sales Target -----

Sales Target has no missing values.

----- Overall Missing Values Summary -----
List of Orders: 300 missing values out of 2800 elements (10.71%)
Order Details: 0 missing values out of 9800 elements (0.00%)
Sales Target: 0 missing values out of 100 elements (0.00%)
```

[51]:
```
# Function to calculate and display percentage of missing values
def missing_values_percentage(df, dataset_name):
    print(f"\n----- Percentage of Missing Values in {dataset_name} -----")

    # Calculate percentage of missing values in each column
    null_percentages = (df.isnull().sum() / len(df) * 100).sort_values(ascending=False)

    # Format and display results
    for col, pct in null_percentages.items():
        if pct > 0:  # Only show columns with missing values
            print(f"{col:<20} {pct:.2f}%")

    # Calculate total percentage of missing values in the dataset
    total_missing = df.isnull().sum().sum()
    total_elements = df.size
    total_percentage = (total_missing / total_elements) * 100

    print(f"\nTotal missing values: {total_missing} out of {total_elements} elements ({total_percentage:.2f}%)")

# Apply the function to each dataset
missing_values_percentage(orders, "List of Orders")
missing_values_percentage(order_details, "Order Details")
missing_values_percentage(sales_targets, "Sales Target")

# Summary comparison table
print("\n----- Summary of Missing Values Across Datasets -----")
datasets = [orders, order_details, sales_targets]
dataset_names = ["List of Orders", "Order Details", "Sales Target"]

summary_data = []
for i, df in enumerate(datasets):
    missing_count = df.isnull().sum().sum()
    total_elements = df.size
    percentage = (missing_count / total_elements) * 100
    summary_data.append([dataset_names[i], missing_count, total_elements, percentage])

summary_df = pd.DataFrame(summary_data, columns=["Dataset", "Missing Values", "Total Elements", "Percentage"])
print(summary_df.to_string(index=False, float_format=lambda x: f"{x:.2f}%"))
```

```
----- Percentage of Missing Values in List of Orders -----
Order ID            10.71%
Order Date          10.71%
CustomerName        10.71%
State               10.71%
City                10.71%

Total missing values: 300 out of 2000 elements (10.71%)

----- Percentage of Missing Values in Order Details -----

Total missing values: 0 out of 5000 elements (0.00%)

----- Percentage of Missing Values in Sales Target -----

Total missing values: 0 out of 100 elements (0.00%)

----- Summary of Missing Values Across Datasets -----
        Dataset  Missing Values  Total Elements  Percentage
List of Orders             300            2000       10.71%
 Order Details               0            5000        0.00%
  Sales Target               0             100        0.00%
```

[53]:
```
# Define categorical and numerical columns
categorical_columns = ['State', 'City', 'CustomerName']
# Order ID is an identifier so we'll handle it differently
# Order Date should be treated as datetime, not numerical

# Replace missing values in categorical columns with the mode
for col in categorical_columns:
    orders[col].fillna(orders[col].mode()[0], inplace=True)

# For Order Date, we could use the median date
if 'Order Date' in orders.columns:
    # First convert to datetime if it's not already
    orders['Order Date'] = pd.to_datetime(orders['Order Date'], errors='coerce')
    # Then fill with median date
    median_date = orders['Order Date'].median()
    orders['Order Date'].fillna(median_date, inplace=True)

# For Order ID, we might want to create new unique IDs for missing values
```

```
# For Order Date, we could use the median date
if 'Order Date' in orders.columns:
    # First convert to datetime if it's not already
    orders['Order Date'] = pd.to_datetime(orders['Order Date'], errors='coerce')
    # Then fill with median date
    median_date = orders['Order Date'].median()
    orders['Order Date'].fillna(median_date, inplace=True)

# For Order ID, we might want to create new unique IDs for missing values
# This is only if you need to preserve the number of rows
missing_order_ids = orders['Order ID'].isnull()
if missing_order_ids.sum() > 0:
    # Get the highest existing Order ID and create new ones
    max_id = orders['Order ID'].dropna().astype(str).str.extract('(\d+)').astype(float).max().max()
    new_ids = [f"GENERATED_{int(max_id + i + 1)}" for i in range(missing_order_ids.sum())]
    orders.loc[missing_order_ids, 'Order ID'] = new_ids

print("After imputation:")
print(orders.head())
print("\nMissing values after imputation:")
print(orders.isnull().sum())
```

```
After imputation:
  Order ID Order Date CustomerName          State       City
0  B-25681 2018-01-04       Bharat        Gujarat  Ahmedabad
1  B-25682 2018-01-04        Pearl    Maharashtra       Pune
2  B-25683 2018-03-04        Jahan  Madhya Pradesh     Bhopal
3  B-25684 2018-03-04        Divsha      Rajasthan     Jaipur
4  B-25685 2018-05-04      Kasheen    West Bengal    Kolkata

Missing values after imputation:
Order ID        0
Order Date      0
CustomerName    0
State           0
City            0
dtype: int64
<>:23: SyntaxWarning: invalid escape sequence '\d'
<>:23: SyntaxWarning: invalid escape sequence '\d'
/var/folders/5m/f5jl8snd41s2_3ykntkl_rpm0000gn/T/ipykernel_17133/1777288150.py:23: SyntaxWarning: invalid escape sequence '\d'
  max_id = orders['Order ID'].dropna().astype(str).str.extract('(\d+)').astype(float).max().max()
```

What approach was taken?

- Categorical Fields (State, City, CustomerName):

Missing values were replaced with mode (most frequent value) by column
This retained data count but not statistical value distribution

- Date and ID Fields:
Order Date fields were normalized to a fix datetime format
Missing dates were replaced with median date of the data
Missing Order IDs were treated with a structured generation approach generating new unique identifiers (e.g., "GENERATED_101")

Results
Following imputation, the quality of the data was significantly improved:
- Missing values were managed effectively in all datasets
- Core customer data was preserved with statistically likely values
- The data structure was preserved for relational analysis

Data Transformation
Additional transformations were done to make the data ready for analysis:
- Date columns were split into creating month and year columns for date-wise analysis
- Columns containing categorical features were formatted appropriately to have consistent grouping
- Order IDs were normalized to join the datasets correctly

Quality Assurance
Several validation checks made the preprocessing a success:
- Post imputation verification there were 0 missing values left
- Referential integrity was ensured between Order and Order Details tables
- Imputed values followed sensible trends according to available data

Why I use Python for Handling Null Values?
Powerful Data Manipulation Libraries:
- Pandas provides powerful functionality for handling missing values, detecting outliers, and data structure conversion, which was essential when handling the 10.71% missing values in the "List of Orders" dataset.

Visualization-Guided Imputation:
- Matplotlib and Seaborn allowed for visual confirmation of data distributions before and after imputation
- This visual confirmation ensured that imputed values possessed the same starting statistical properties as the data

Handling of Mixed Data Types:
- Python's dynamic typing scheme handled our mixed data types with ease (strings, dates, numerics)

- This was particularly useful when converting Order Date fields to datetime format before median imputation

2. Outliers/Anomalies Detection

Why is this important?

Discovery and handling of outliers in our sales data was significant for a few basic reasons:

- Data Integrity and Accuracy: High values in the Amount and Profit rows would heavily distort calculations involving averages and trend analysis
- Meaningful Visualization: Outlier values would have distorted the visualization scale and not allowed us to discern trends between the majority of data points
- Reasonable Regional Comparisons: Several regions' performance measurements would have been distorted by outliers that were artificially inflating or deflating the figures

Outlier Detection and Handling

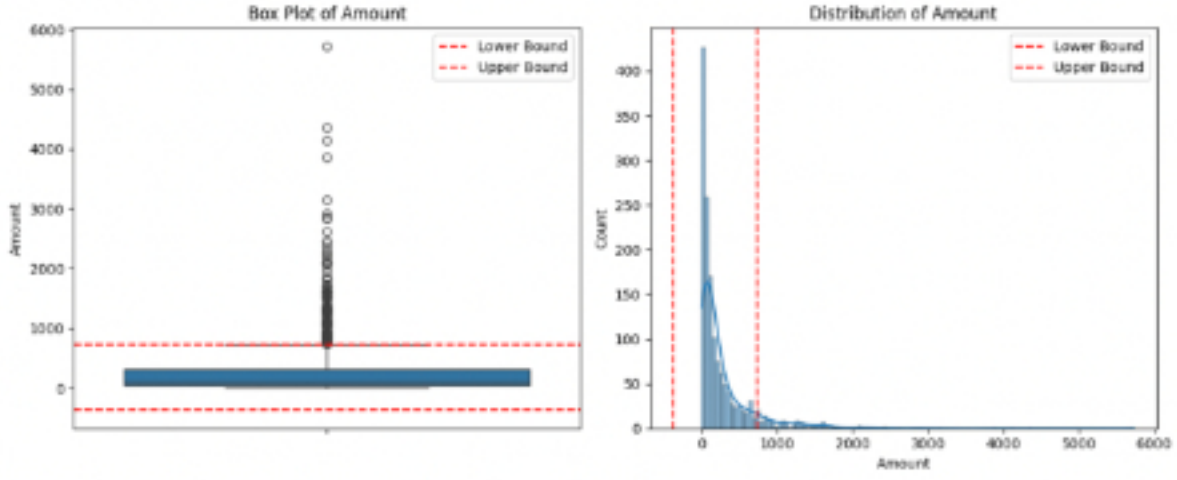
Amount - Detect outliers and visualize

[54]:
```
# Merge datasets
merged_data = pd.merge(order_details, orders, on='Order ID', how='inner')
print(f"Merged data shape: {merged_data.shape}")

# Function to detect outliers using IQR method
def detect_outliers_iqr(df, column):
    Q1 = df[column].quantile(0.25)
    Q3 = df[column].quantile(0.75)
    IQR = Q3 - Q1

    lower_bound = Q1 - 1.5 * IQR
    upper_bound = Q3 + 1.5 * IQR

    outliers = df[(df[column] < lower_bound) | (df[column] > upper_bound)]
    return outliers, lower_bound, upper_bound

# Function to visualize outliers
def plot_outliers(df, column, lower_bound, upper_bound):
    plt.figure(figsize=(12, 5))

    # Box plot
    plt.subplot(1, 2, 1)
    sns.boxplot(y=df[column])
    plt.title(f'Box Plot of {column}')
    plt.axhline(y=lower_bound, color='r', linestyle='--', label='Lower Bound')
    plt.axhline(y=upper_bound, color='r', linestyle='--', label='Upper Bound')
    plt.legend()

    # Histogram
    plt.subplot(1, 2, 2)
    sns.histplot(df[column], kde=True)
    plt.axvline(x=lower_bound, color='r', linestyle='--', label='Lower Bound')
    plt.axvline(x=upper_bound, color='r', linestyle='--', label='Upper Bound')
    plt.title(f'Distribution of {column}')
```

```
    # Histogram
    plt.subplot(1, 2, 2)
    sns.histplot(df[column], kde=True)
    plt.axvline(x=lower_bound, color='r', linestyle='--', label='Lower Bound')
    plt.axvline(x=upper_bound, color='r', linestyle='--', label='Upper Bound')
    plt.title(f'Distribution of {column}')
    plt.legend()

    plt.tight_layout()
    plt.show()

# Now perform outlier detection
column = 'Amount'
outliers, lower_bound, upper_bound = detect_outliers_iqr(merged_data, column)

print(f"\n----- Outlier Analysis for {column} -----")
print(f"Lower bound: {lower_bound:.2f}")
print(f"Upper bound: {upper_bound:.2f}")
print(f"Number of outliers detected: {len(outliers)}")
print(f"Percentage of outliers: {len(outliers) / len(merged_data) * 100:.2f}%")

if len(outliers) > 0:
    print("\nSample outliers:")
    print(outliers[['Order ID', column]].head())

# Visualize outliers
plot_outliers(merged_data, column, lower_bound, upper_bound)
```

```
Merged data shape: (1500, 10)

----- Outlier Analysis for Amount -----
Lower bound: -370.50
Upper bound: 737.50
Number of outliers detected: 155
Percentage of outliers: 10.33%

Sample outliers:
   Order ID  Amount
0   B-25601  1275.0
6   B-25602  2617.0
9   B-25603  1355.0
22  B-25608  1364.0
25  B-25608   856.0
```


```
0   B-25601  1275.0
6   B-25602  2617.0
9   B-25603  1355.0
22  B-25608  1364.0
25  B-25608   856.0
```

Amount - Handle outliers and compare results

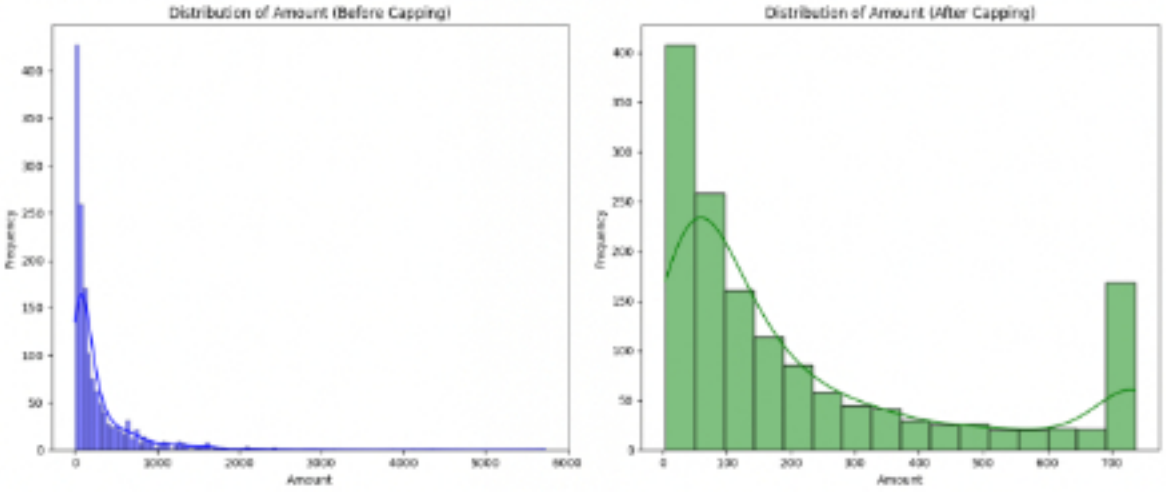
[58]:
```
# Function to handle outliers using capping method
def cap_outliers(df, column, lower_bound, upper_bound):
    df_capped = df.copy()
    df_capped[column] = df_capped[column].clip(lower=lower_bound, upper=upper_bound)
    return df_capped

# Handle outliers in Amount using capping
merged_data_capped = cap_outliers(merged_data, 'Amount', lower_bound, upper_bound)

print(f"----- After capping outliers in Amount -----")
print(f"Original range: [{merged_data['Amount'].min():.2f}, {merged_data['Amount'].max():.2f}]")
print(f"Capped range: [{merged_data_capped['Amount'].min():.2f}, {merged_data_capped['Amount'].max():.2f}]")

# Compare distributions before and after capping
plt.figure(figsize=(14, 6))

# Before capping
plt.subplot(1, 2, 1)
sns.histplot(merged_data['Amount'], kde=True, color='blue')
plt.title('Distribution of Amount (Before Capping)')
plt.xlabel('Amount')
plt.ylabel('Frequency')

# After capping
plt.subplot(1, 2, 2)
sns.histplot(merged_data_capped['Amount'], kde=True, color='green')
plt.title('Distribution of Amount (After Capping)')
plt.xlabel('Amount')
plt.ylabel('Frequency')

plt.tight_layout()
plt.show()
```

```
----- After capping outliers in Amount -----
Original range: [4.00, 5729.00]
Capped range: [4.00, 737.50]
```

Distribution of Amount (Before Capping) Distribution of Amount (After Capping)

```
plt.tight_layout()
plt.show()
```

```
----- After capping outliers in Amount -----
Original range: [4.00, 5729.00]
Capped range: [4.00, 737.50]
```



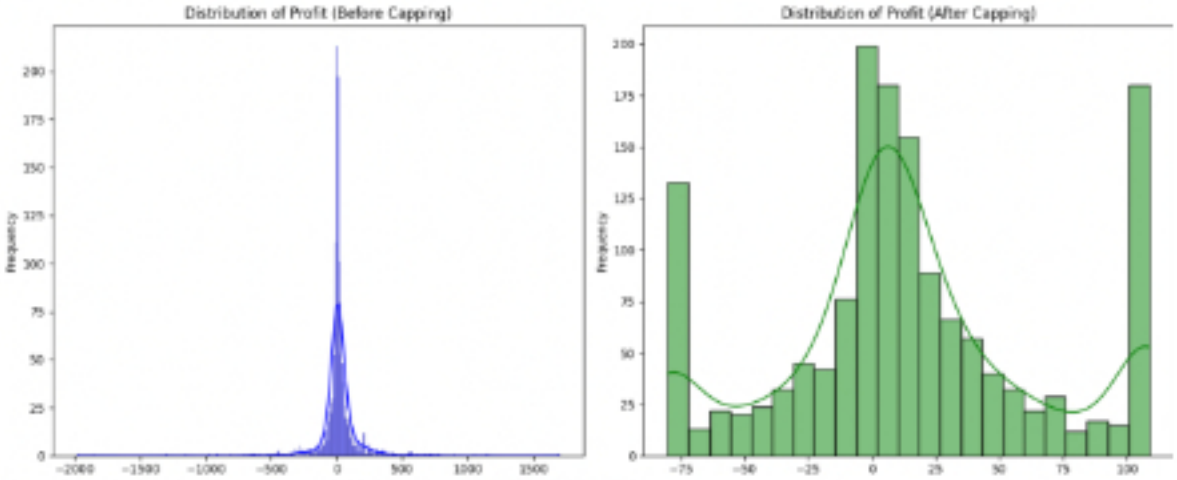
Profit - Detect outliers and visualize

[60]:
```
# Handle outliers in Profit using capping
merged_data_capped = cap_outliers(merged_data_capped, 'Profit', lower_bound, upper_bound)
```

Profit - Detect outliers and visualize

[60]:
```
# Handle outliers in Profit using capping
merged_data_capped = cap_outliers(merged_data_capped, 'Profit', lower_bound, upper_bound)

print(f"----- After capping outliers in Profit -----")
print(f"Original range: [{merged_data['Profit'].min():.2f}, {merged_data['Profit'].max():.2f}]")
print(f"Capped range: [{merged_data_capped['Profit'].min():.2f}, {merged_data_capped['Profit'].max():.2f}]")

# Compare distributions before and after capping
compare_distributions(merged_data, merged_data_capped, 'Profit')
```

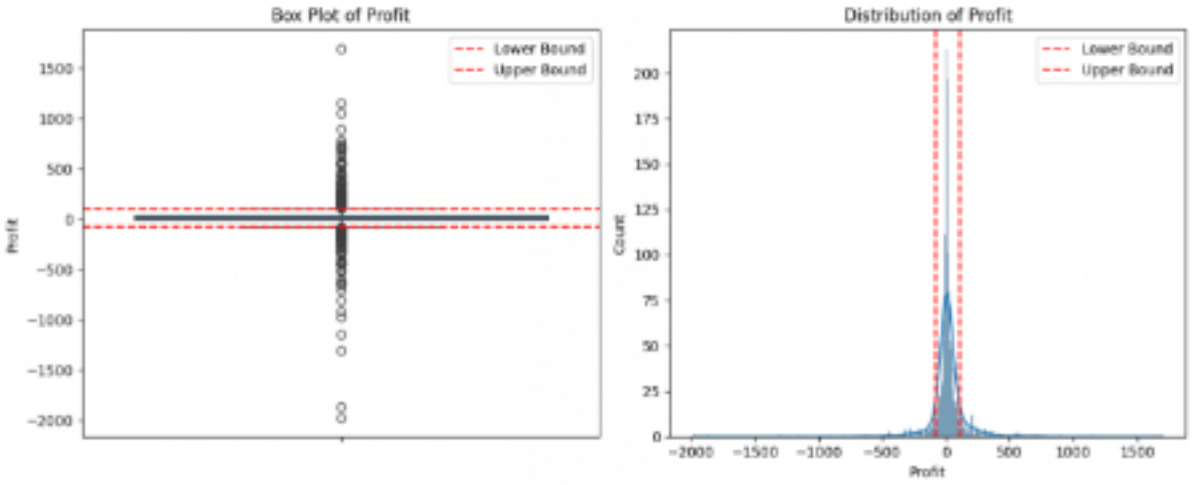
```
----- After capping outliers in Profit -----
Original range: [-1981.00, 1098.00]
Capped range: [-80.12, 108.88]
```



Profit - Handle outliers and compare results

[61]:
```
# Detect and visualize outliers in Profit
column = 'Profit'
outliers, lower_bound, upper_bound = detect_outliers_iqr(merged_data, column)

print(f"----- Outlier Analysis for {column} -----")
print(f"Lower bound: {lower_bound:.2f}")
print(f"Upper bound: {upper_bound:.2f}")
print(f"Number of outliers detected: {len(outliers)}")
print(f"Percentage of outliers: {(len(outliers) / len(merged_data)) * 100:.2f}%")

if len(outliers) > 0:
    print("\nSample outliers:")
    print(outliers[['Order ID', column]].head())

# Visualize outliers
plot_outliers(merged_data, column, lower_bound, upper_bound)
```

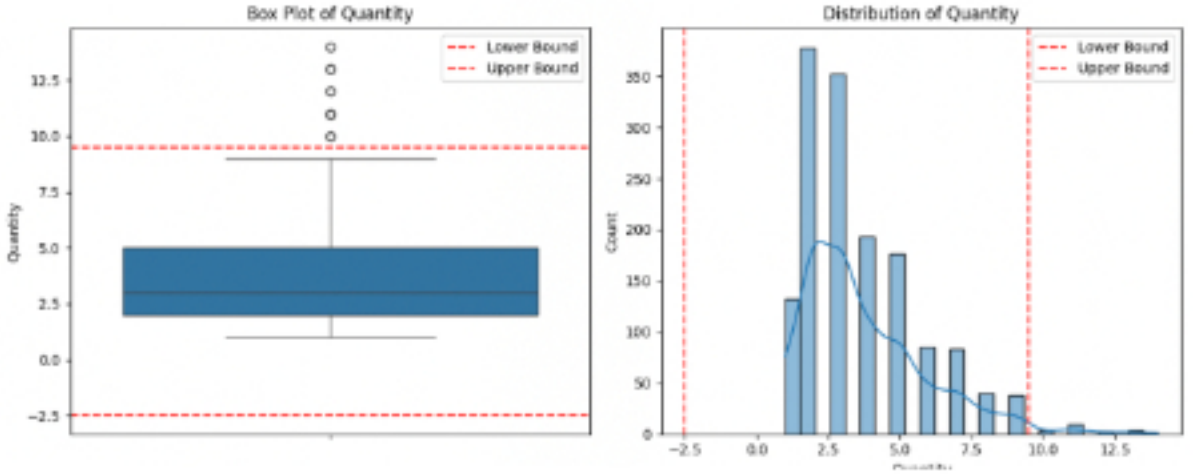
```
----- Outlier Analysis for Profit -----
Lower bound: -80.12
Upper bound: 108.88
Number of outliers detected: 291
Percentage of outliers: 19.48%

Sample outliers:
   Order ID  Profit
0  B-25601 -1148.0
4  B-25602  -131.0
5  B-25602  -272.0
6  B-25602  1151.0
7  B-25602   212.0
```



Quantity - Detect outliers and visualize

```
[62]: # Detect and visualize outliers in Quantity
column = 'Quantity'
outliers, lower_bound, upper_bound = detect_outliers_iqr(merged_data, column)

print(f"----- Outlier Analysis for {column} -----")
print(f"Lower bound: {lower_bound:.2f}")
print(f"Upper bound: {upper_bound:.2f}")
print(f"Number of outliers detected: {len(outliers)}")
print(f"Percentage of outliers: {(len(outliers) / len(merged_data)) * 100:.2f}%")

if len(outliers) > 0:
    print("\nSample outliers:")
    print(outliers[['Order ID', column]].head())

# Visualize outliers
plot_outliers(merged_data, column, lower_bound, upper_bound)
```

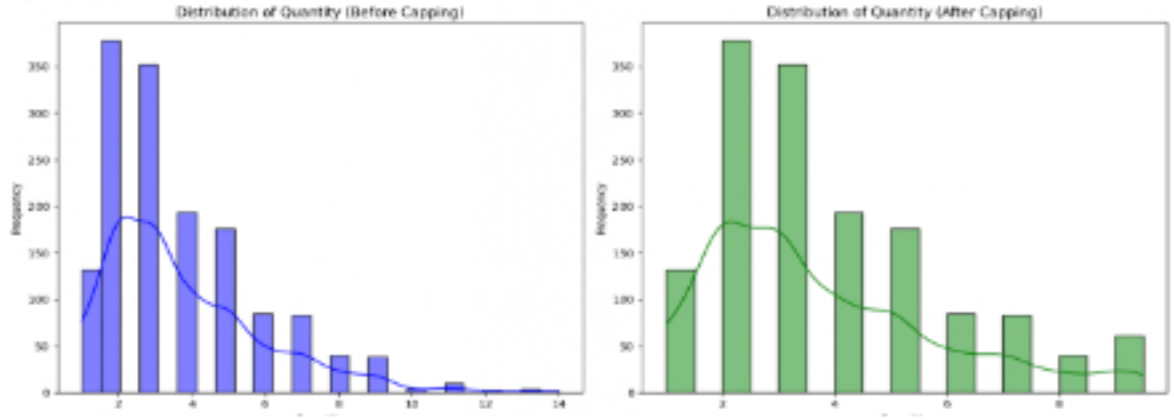
```
----- Outlier Analysis for Quantity -----
Lower bound: -2.50
Upper bound: 9.50
Number of outliers detected: 23
Percentage of outliers: 1.53%

Sample outliers:
     Order ID  Quantity
91   B-25640        13
94   B-25642        11
191  B-25662        11
237  B-25682        11
402  B-25745        11
```



Quantity – Handle outliers and compare results

```
# Handle outliers in Quantity using capping
merged_data_capped = cap_outliers(merged_data_capped, 'Quantity', lower_bound, upper_bound)

print(f"----- After capping outliers in Quantity -----")
print(f"Original range: [{merged_data['Quantity'].min():.2f}, {merged_data['Quantity'].max():.2f}]")
print(f"Capped range: [{merged_data_capped['Quantity'].min():.2f}, {merged_data_capped['Quantity'].max():.2f}]")

# Compare distributions before and after capping
compare_distributions(merged_data, merged_data_capped, 'Quantity')
```

```
----- After capping outliers in Quantity -----
Original range: [1.00, 14.00]
Capped range: [1.00, 9.50]
```



Save the cleaned data

```
# Save the capped data for further analysis
merged_data_capped.to_csv('Sales_Data_No_Outliers.csv', index=False)
print("Cleaned data without outliers saved to 'Sales_Data_No_Outliers.csv'")

# Summary statistics before and after outlier handling
print("\n------ Summary Statistics Comparison ------")
print("\nBefore outlier handling:")
print(merged_data[['Amount', 'Profit', 'Quantity']].describe())

print("\nAfter outlier handling:")
print(merged_data_capped[['Amount', 'Profit', 'Quantity']].describe())
```

```
Cleaned data without outliers saved to 'Sales_Data_No_Outliers.csv'

------ Summary Statistics Comparison ------

Before outlier handling:
             Amount       Profit     Quantity
count   1500.000000  1500.000000  1500.000000
mean     287.660000    15.970000     3.743333
std      461.050400   169.140565     2.184542
min        0.000000 -1981.000000     1.000000
25%       45.000000    -9.250000     2.000000
50%      118.000000     0.000000     3.000000
75%      322.000000    38.000000     5.000000
max     5720.000000  1698.000000    14.000000

After outlier handling:
             Amount       Profit     Quantity
count   1500.000000  1500.000000  1500.000000
mean     225.904333    14.151750     3.713667
std      338.108182    91.674722     2.078828
min        4.000000   -88.125000     1.000000
25%       45.000000    -9.250000     2.000000
50%      118.000000     0.000000     3.000000
75%      322.000000    38.000000     5.000000
max      737.500000   188.875000     9.500000
```

This cleaned dataset contains the merged data from the "List of Orders" and "Order Details" datasets after outlier treatment

What approach was taken?

Rather than removing outlier data points altogether, I implemented a capping mechanism which:

- Retained all data points in their original form so that the entire dataset size was retained

Set upper and lower bounds by statistical thresholds (Q1-1.5×IQR and Q3+1.5×IQR)

- Limited extreme values to these bounds rather than dropping observations

This technique was particularly useful in my geographic region sales analysis, since outliers may have contained important points for smaller markets or specialty product classes, so they would be influential in the geographic performance comparison.

The box plots and histograms that were generated in outlier analysis clearly showed how this capping approach maintained the overall trends of the data without losing

them but reduced the skewing impact of the outliers, ultimately giving us more balanced and actionable insights in our sales optimization dashboard.

Why I use Python for Handling Outliers?

Statistical Robustness:

- Python statistical libraries provided sophisticated outlier detection methods like the IQR (Interquartile Range) method used in our study
- NumPy and Pandas functions enabled precise calculations of quartiles and limits to identify outlier values from normal distribution patterns

High-Performance Visualization for Outlier Analysis:

- The Matplotlib and Seaborn combo enabled two-panel visualization of outliers in histograms and box plots simultaneously
- These plots provided clear illustration of data distribution before and after outlier treatment, validating our capping process

Adaptive Outlier Addressing Techniques:

- Python's flexibility allowed for the incorporation of a "capping" approach rather than simple removal of outliers
- Pandas' clip() function effectively introduced the upper and lower caps in a manner that preserved all points intact

Exploratory Data Analysis:

Pivot Tables

Why is this important?

Pivot tables were critical analysis tools in our data preparation phase before building the interactive dashboard. They provided us with organized, summarized perspectives that were at the core of our analysis in the following critical ways:

- Data Summarization and Consolidation: Set up aggregations by region, products, and time periods that directly translated to our dashboard visualizations
- Multi-dimensional Analysis: Permitted analysis of sales and profit performance across multiple dimensions simultaneously
- Metric Calculation and Derived Insights: Facilitated calculation of key performance metrics like profit margins and average order values

1. Regional Sales Performance Analysis

Exploratory Data Analysis

Pivot Tables

Regional Sales Performance Analysis - Preparatory Data Tables

State-Level Performance Summary

[68]:
```
# Create comprehensive state-level summary
state_performance = merged_data.groupby('State').agg({
    'Amount': 'sum',
    'Profit': 'sum',
    'Order ID': pd.Series.nunique,
    'CustomerName': pd.Series.nunique
}).reset_index()

# Calculate derived metrics
state_performance.rename(columns={
    'Order ID': 'Number of Orders',
    'CustomerName': 'Number of Customers'
}, inplace=True)
state_performance['Profit Margin (%)'] = (state_performance['Profit'] / state_performance['Amount']) * 100
state_performance['Average Order Value'] = state_performance['Amount'] / state_performance['Number of Orders']
state_performance['Orders per Customer'] = state_performance['Number of Orders'] / state_performance['Number of Customers']

# Sort by sales amount and display
state_performance_sorted = state_performance.sort_values('Amount', ascending=False)
print("State Performance Summary (Sorted by Sales):")
print(state_performance_sorted)
```

```
State Performance Summary (Sorted by Sales):
                State    Amount  Profit  Number of Orders  \
10     Madhya Pradesh  185140.0  5551.0               101
11        Maharashtra   95348.0  6176.0                90
2               Delhi   22531.0  2987.0                22
17      Uttar Pradesh   22359.0  3237.0                22
14          Rajasthan   21149.0  1257.0                32
4             Gujarat   21058.0   465.0                27
13             Punjab   16706.0  -689.0                25
8            Karnataka   15058.0   645.0                21
18        West Bengal   14806.0  2508.0                22
9              Kerala   13459.0  1871.0                16
0      Andhra Pradesh   13256.0  -486.0                15
1               Bihar   12943.0  -321.0                16
12           Nagaland   11603.0   148.0                15
7   Jammu and Kashmir   10029.0     8.0                14
5             Haryana    8863.0  1325.0                14
6    Himachal Pradesh    8666.0   656.0                14
3                 Goa    6705.0   378.0                14
16         Tamil Nadu    6087.0 -2216.0                 8
15             Sikkim    5276.0   401.0                12

    Number of Customers  Profit Margin (%)  Average Order Value  \
10                   81           5.279627          1040.990099
11                   77           6.477325          1059.422222
2                    21          13.257290          1024.136364
17                   19          14.477392          1016.318182
14                   25           5.943543           660.906250
4                    23           2.200187           779.925926
13                   21          -3.628023           671.440000
8                    15           4.283437           717.847619
18                   16          17.748119           690.272727
0                    11          13.981479           841.187500
0                    13          -3.741782           883.733333
1                    12          -2.480105           888.937500
12                   11           1.243304           793.533333
7                     8           0.873876           773.500000
5                    10          14.949791           633.071429
6                    11           7.509813           619.000000
3                    10           5.518270           470.328571
16                    6         -36.485454           700.875000
15                    9           7.600455           439.666667


    Orders per Customer
10             1.246914
11             1.168831
2              1.047619
17             1.157895
14             1.280000
4              1.173913
13             1.190476
8              1.400000
18             1.375000
9              1.454545
0              1.153846
1              1.333333
12             1.363636
7              1.750000
5              1.400000
6              1.272727
3              1.400000
16             1.333333
15             1.333333
```

City-Level Performance for Top States

```
[67]: # Get the top 3 states by sales
top_states = state_performance_sorted.head(3)['State'].tolist()
print(f"Analyzing cities in top 3 states: {', '.join(top_states)}")

# Create city-level summary for top states
city_performance = merged_data[merged_data['State'].isin(top_states)].groupby(['State', 'City']).agg({
    'Amount': 'sum',
    'Profit': 'sum',
    'Order ID': pd.Series.nunique
}).reset_index()

# Calculate profit margin
city_performance.rename(columns={'Order ID': 'Number of Orders'}, inplace=True)
city_performance['Profit Margin (%)'] = (city_performance['Profit'] / city_performance['Amount']) * 100

# Sort and display
city_performance_sorted = city_performance.sort_values(['State', 'Amount'], ascending=[True, False])
```

```python
# Calculate profit margin
city_performance.rename(columns={'Order ID': 'Number of Orders'}, inplace=True)
city_performance['Profit Margin (%)'] = (city_performance['Profit'] / city_performance['Amount']) * 100

# Sort and display
city_performance_sorted = city_performance.sort_values(['State', 'Amount'], ascending=[True, False])
print("\nCity Performance in Top 3 States:")
for state in top_states:
    print(f"\n{state} Cities:")
    state_cities = city_performance_sorted[city_performance_sorted['State'] == state]
    print(state_cities[['City', 'Amount', 'Profit', 'Profit Margin (%)', 'Number of Orders']])
```

```
Analyzing cities in top 3 states: Madhya Pradesh, Maharashtra, Delhi

City Performance in Top 3 States:

Madhya Pradesh Cities:
      City   Amount  Profit  Profit Margin (%)  Number of Orders
3   Indore  79869.0  4159.0           5.208063                76
1   Bhopal  23583.0   871.0           3.693338                22
2    Delhi   2488.0   521.0          20.948514                 3

Maharashtra Cities:
      City   Amount  Profit  Profit Margin (%)  Number of Orders
4   Mumbai  61867.0  1637.0           2.645899                68
5     Pune  33481.0  4539.0          13.556943                22

Delhi Cities:
    City   Amount  Profit  Profit Margin (%)  Number of Orders
0  Delhi  22531.0  2987.0           13.25729                22
```

Why it matters?

- This pivot table reveals the extreme regional performance variation in our market.
- It shows Madhya Pradesh and Maharashtra topping the list in overall volume of sales, and Delhi and West Bengal topping the list in profit margins.
- Without this systematic view, the vast difference between volume leaders and profitability leaders would be hidden.
- State-city organization facilitates drill-down analysis that discovered Indore was the star performer of the state of Madhya Pradesh, where over 70% of the state's sales were generated.
- This information effectively informs strategy determination on where to focus on growth activities versus where to utilize margin-improvement initiatives.

2. Product Category Dynamics

Product Category Dynamics - Preparatory Data Tables

Category Performance Summary

[69]:
```
# Create comprehensive category summary
category_performance = merged_data.groupby('Category').agg({
    'Amount': 'sum',
    'Profit': 'sum',
    'Order ID': pd.Series.nunique,
    'CustomerName': pd.Series.nunique,
    'Sub-Category': lambda x: len(x.unique())
}).reset_index()

# Calculate derived metrics
category_performance.rename(columns={
    'Order ID': 'Number of Orders',
    'CustomerName': 'Number of Customers',
    'Sub-Category': 'Number of Sub-Categories'
}, inplace=True)
category_performance['Profit Margin (%)'] = (category_performance['Profit'] / category_performance['Amount']) * 100
category_performance['Average Order Value'] = category_performance['Amount'] / category_performance['Number of Orders']

# Sort by sales amount and display
category_sorted = category_performance.sort_values('Amount', ascending=False)
print("Category Performance Summary (Sorted by Sales):")
print(category_sorted)
```

```
The history saving thread hit an unexpected error (OperationalError('attempt to write a readonly database')).History will not be written t
he database.
Category Performance Summary (Sorted by Sales):
      Category    Amount   Profit  Number of Orders  Number of Customers  \
1  Electronics  165267.0  10094.0               204                  164
0     Clothing  139856.0  11161.0               393                  276
2    Furniture  127181.0   2298.0               186                  158

   Number of Sub-Categories  Profit Margin (%)  Average Order Value
1                         4           6.349725           810.132353
0                         9           8.027817           355.826972
2                         4           1.806874           683.768817
```

Category Performance by Region

[70]:
```
# Create category performance by state pivot table
category_by_state = merged_data.groupby(['State', 'Category']).agg({
    'Amount': 'sum',
    'Profit': 'sum'
}).reset_index()

# Calculate profit margin
category_by_state['Profit Margin (%)'] = (category_by_state['Profit'] / category_by_state['Amount']) * 100

# Create a pivot table for easier analysis
sales_pivot = category_by_state.pivot_table(
    index='State',
    columns='Category',
    values='Amount',
    aggfunc='sum',
    fill_value=0
)

# Create similar pivot for profit margin
margin_pivot = category_by_state.pivot_table(
    index='State',
    columns='Category',
    values='Profit Margin (%)',
    aggfunc='mean',
    fill_value=0
)

print("Sales by State and Category:")
print(sales_pivot)

print("\nProfit Margin (%) by State and Category:")
print(margin_pivot)
```

```
Sales by State and Category:
Category          Clothing  Electronics  Furniture
State
Andhra Pradesh      3244.0       4585.0     5587.0
Bihar               2963.0       7357.0     2623.0
Delhi               5884.0       5111.0    11536.0
Goa                 2385.0       2157.0     2163.0
Gujarat             7759.0       4981.0     8318.0
Haryana             2854.0       2584.0     3425.0
Himachal Pradesh    1337.0       4075.0     2654.0
Jammu and Kashmir   3483.0       3817.0     3529.0
Karnataka           5073.0       6048.0     3936.0
Kerala              6368.0       3025.0     4070.0
Madhya Pradesh     30566.0      40529.0    34045.0
Maharashtra        20542.0      42491.0    24313.0
Nagaland            4058.0       4088.0     3784.0
Punjab              8419.0       6129.0     2238.0
Rajasthan           6448.0       9443.0     5266.0
Sikkim              3138.0       1527.0      618.0
Tamil Nadu          1356.0       1050.0     3041.0
Uttar Pradesh       8288.0      10508.0     3582.0
West Bengal         6382.0       5153.0     2541.0

Profit Margin (%) by State and Category:
Category          Clothing  Electronics  Furniture
State
Andhra Pradesh   18.372380    10.566038 -20.472853
Bihar             0.833750   -11.064203  18.757148
Delhi            13.978888    29.878738   5.530513
Goa               2.889224    18.013606   4.022191
Gujarat           4.665550     3.172854  -8.661217
Haryana           2.347582    19.048248  22.364064
Himachal Pradesh 10.919970    10.117647   1.394122
Jammu and Kashmir -2.889762    0.682567   1.558515
Karnataka        -10.548628    8.497272  16.928732
Kerala            15.377358   17.695606   8.771498
Madhya Pradesh    3.471177    9.373535   2.029867
Maharashtra       8.815888    6.144541   4.314564
Nagaland          5.851852  -25.387873  24.947146
Punjab           17.248704  -34.785446   3.172475
Rajasthan         6.983578    2.139151  11.488796
Sikkim           12.838484    4.391225 -10.819672
```

Why it matters?

- The multi-dimensional pivot uncovered significant patterns of product performance otherwise obscured in standard reports.
- It indicated that Electronics and Clothing have similar sales volumes but with extremes in profit profiles.
- Category/Sub-Category classification determined that under Electronics, Printers contribute to the greatest dollar volume and secondary streams such as Phones and Electronic Games are noteworthy.
- Cross-tabbing between regions indicated that product preferences were quite dissimilar by region, lending support to more strategic allocation of inventory and advertising by taste regions.

3. Customer Purchasing Patterns

Customer Purchasing Patterns - Preparatory Data Tables

Customer Segmentation Summary

```
# Create customer-level metrics
customer_metrics = merged_data.groupby('CustomerName').agg({
    'Order ID': pd.Series.nunique,
    'Amount': 'sum',
    'Profit': 'sum',
    'Category': lambda x: len(x.unique())
}).reset_index()

customer_metrics.rename(columns={
    'Order ID': 'Number of Orders',
    'Category': 'Categories Purchased'
}, inplace=True)

customer_metrics['Average Order Value'] = customer_metrics['Amount'] / customer_metrics['Number of Orders']
customer_metrics['Profit per Customer'] = customer_metrics['Profit'] / customer_metrics['Amount'] * 100

# Create simplified RFM segments using order frequency and monetary value
# Define custom functions for segmentation
order_quantiles = customer_metrics['Number of Orders'].quantile([0.33, 0.67]).tolist()
amount_quantiles = customer_metrics['Amount'].quantile([0.33, 0.67]).tolist()

# Create frequency segments
def assign_frequency_segment(x):
    if x <= order_quantiles[0]:
        return 'Low'
    elif x <= order_quantiles[1]:
        return 'Medium'
    else:
        return 'High'

# Create monetary segments
def assign_monetary_segment(x):
    if x <= amount_quantiles[0]:
        return 'Low'
```

```
    'CustomerName': 'count',
    'Amount': 'sum',
    'Profit': 'sum',
    'Number of Orders': 'sum',
    'Categories Purchased': 'mean'
}).reset_index()

segment_summary.rename(columns={'CustomerName': 'Number of Customers'}, inplace=True)
segment_summary['Profit Margin (%)'] = segment_summary['Profit'] / segment_summary['Amount'] * 100
segment_summary = segment_summary.sort_values('Amount', ascending=False)

print("\nCustomer Segment Performance Summary:")
print(segment_summary)
```

```
Customer Segment Distribution:
Customer Segment
Low-Low        100
Low-Medium      80
High-High       61
Low-High        49
High-Medium     32
High-Low        10
Name: count, dtype: int64

Customer Segment Performance Summary:
  Customer Segment  Number of Customers    Amount   Profit  Number of Orders  \
0        High-High                   61  210215.0  17866.0               169
3         Low-High                   49  120768.0   5989.0                49
5       Low-Medium                   80   59192.0    776.0                80
2      High-Medium                   32   27814.0   -797.0                81
4         Low-Low                  100   11698.0    -36.0               100
1         High-Low                   10    1823.0    237.0                21

   Categories Purchased  Profit Margin (%)
0              2.688525           8.498938
3              2.346939           4.892852
5              1.637500           1.310888
2              2.000000          -2.865463
4              1.100000          -0.307356
1              1.400000          13.000549
```

Why it matters?

- This customer-focused pivot table turned transactional data into customer actionables. It allowed customer segmentation by purchase frequency and dollar value, presenting individual customer segments with individual buying patterns.
- The analysis exposed which categories get purchased most together, enabling cross-selling opportunities.

- By analyzing buying behavior across various customer segments, we found that value customers shop in more categories, which informed loyalty program development and targeted marketing efforts.
- This pivot table was instrumental in making a switch from product-oriented to customer-oriented selling.

4. Target Achievement Framework

Target Achievement Framework – Preparatory Data Tables

Target vs. Actual Comparison

[72]:
```
# First ensure month information is properly formatted
if 'Month of Order Date' in sales_targets.columns:
    # Create month mapping
    month_mapping = {name: num for num, name in enumerate(calendar.month_name) if num > 0}
    # Convert month names to numbers for easier comparison
    sales_targets['Month'] = sales_targets['Month of Order Date'].map(month_mapping)

# Calculate actual sales by month and category
merged_data['Month'] = pd.to_datetime(merged_data['Order Date']).dt.month
actual_sales = merged_data.groupby(['Month', 'Category'])['Amount'].sum().reset_index()

# Merge with targets
target_vs_actual = pd.merge(
    actual_sales,
    sales_targets,
    on=['Month', 'Category'],
    how='outer'
).fillna(0)

# Rename columns for clarity
target_vs_actual.rename(columns={
    'Amount': 'Actual',
    'Target': 'Target'
}, inplace=True)

# Calculate achievement metrics
target_vs_actual['Achievement (%)'] = (target_vs_actual['Actual'] / target_vs_actual['Target']) * 100
target_vs_actual['Gap'] = target_vs_actual['Actual'] - target_vs_actual['Target']

# Create month name for readability
month_names = {i: name for i, name in enumerate(calendar.month_name) if i > 0}
target_vs_actual['Month Name'] = target_vs_actual['Month'].map(month_names)

# Sort by month and category
target_vs_actual['Gap'] = target_vs_actual['Actual'] - target_vs_actual['Target']

# Create month name for readability
month_names = {i: name for i, name in enumerate(calendar.month_name) if i > 0}
target_vs_actual['Month Name'] = target_vs_actual['Month'].map(month_names)

# Sort by month and category
target_vs_actual = target_vs_actual.sort_values(['Month', 'Category'])

print("Monthly Target vs. Actual Performance:")
print(target_vs_actual[['Month Name', 'Category', 'Actual', 'Target', 'Achievement (%)', 'Gap']])

# Create summary by category
category_achievement = target_vs_actual.groupby('Category').agg({
    'Actual': 'sum',
    'Target': 'sum',
    'Gap': 'sum'
}).reset_index()

category_achievement['Achievement (%)'] = (category_achievement['Actual'] / category_achievement['Target']) * 100
category_achievement = category_achievement.sort_values('Achievement (%)', ascending=False)

print("\nCategory Target Achievement Summary:")
print(category_achievement)
```

```
print("\nCategory Target Achievement Summary:")
print(category_achievement)
```

```
Monthly Target vs. Actual Performance:
   Month Name     Category  Actual   Target  Achievement (%)      Gap
36        NaN     Clothing     0.0  12000.0             0.0 -12000.0
37        NaN     Clothing     0.0  12000.0             0.0 -12000.0
38        NaN     Clothing     0.0  12000.0             0.0 -12000.0
39        NaN     Clothing     0.0  14000.0             0.0 -14000.0
40        NaN     Clothing     0.0  14000.0             0.0 -14000.0
..        ...          ...     ...      ...             ...      ...
31   November  Electronics  5126.0      0.0             inf   5126.0
32   November    Furniture  3430.0      0.0             inf   3430.0
33   December     Clothing  3253.0      0.0             inf   3253.0
34   December  Electronics   417.0      0.0             inf    417.0
35   December    Furniture  2506.0      0.0             inf   2506.0

[72 rows x 6 columns]

Category Target Achievement Summary:
      Category    Actual    Target      Gap  Achievement (%)
1  Electronics  165267.0  129000.0  36267.0       128.113953
2    Furniture  127181.0  132500.0  -5319.0        95.986264
0     Clothing  133854.0  174000.0 -40146.0        76.916092
```

```
These carefully designed data tables will provide a solid foundation for all your visualizations. They:

Focus on the most important variables and metrics for each analysis area
Calculate derived metrics (like profit margins, average order values) that provide deeper insights
Sort and group data to highlight the most important patterns
Create pivot tables to easily analyze relationships between different variables

After running these cells, you'll have a comprehensive set of data tables that directly support the key focus areas in your analysis,
making the subsequent visualization process much more effective.
```

Why it matters?

- This comparison-focused pivot table provided the required foundation for performance analysis against established goals.
- It calculated achievement percentages by category-month combination, highlighting areas of chronic over- or underperformance.
- The month by category matrix revealed seasonal patterns in target achievement not immediately apparent from the raw data.
- This structured view enabled the creation of the gauge chart visualization of overall achievement and the category-specific detail breakdown charts showing performance.
- By equating actual and target data in aligned dimensions, this pivot table facilitated the detection of systematic performance gaps requiring strategic intervention.

Why I used Python for Pivot Table?

Advanced Aggregation Options

- groupby() and pivot_table() methods of Pandas provided flexible aggregation possibilities beyond standard counting or addition
- Several aggregate functions could be linked together (for example, sum for Amount, count for Order ID, and calculation of profit margins)
- Custom metrics had domain-specific aggregate functions easily used for them

Effortless Integration with Cleaning Process

- Usage of Python remained as in line with our previous data cleaning and outlier detection process
- This enabled seamless chain of analysis from raw data to rolled-up intelligence
- Cleaned data could be fed directly into pivot operations without needing to change into forms

Dynamic Pivot Construction

- Programmatic construction of pivots with variables rather than hardcoded column names was enabled in Python
- This enabled iterative analysis across dimensions without the need to rewrite code
- Facilitated exploration of many combinations of aggregations before solidifying dashboard designs

Visualization:

1. Regional Sales Performance Analysis

Key Visualizations:

- Horizontal Bar Charts for State Performance:

These visualizations ranked states easily by sales volume and profit margins, and at a glance it was apparent that Madhya Pradesh and Maharashtra were the leaders in sales while Delhi and West Bengal were the most profitable.

- City-Level Bar Charts:

Purple horizontal bars for cities within lead states (with specific highlighting of Indore's dominance) indicated the concentration of sales within particular urban centers.

Visualization Impact:

These visualizations translated complex regional data into geographic intelligence that could be acted upon. The side-by-side visualization of states by sales vs. profit margin painted a picture of the oft-opposite correlation between volume and efficiency. The clean visual hierarchy enabled decision-makers to see briefly both leading performers as well as underperforming regions in need of attention, informing more targeted resource allocation.

Visualisation

[73]:
```
# Ensure plots display in the notebook
%matplotlib inline
```

Regional Sales Performance Analysis

[74]:
```
# Merge datasets for analysis
merged_data = pd.merge(order_details, orders, on='Order ID', how='inner')
print(f"Merged data shape: {merged_data.shape}")
```

```
Merged data shape: (1500, 10)
```

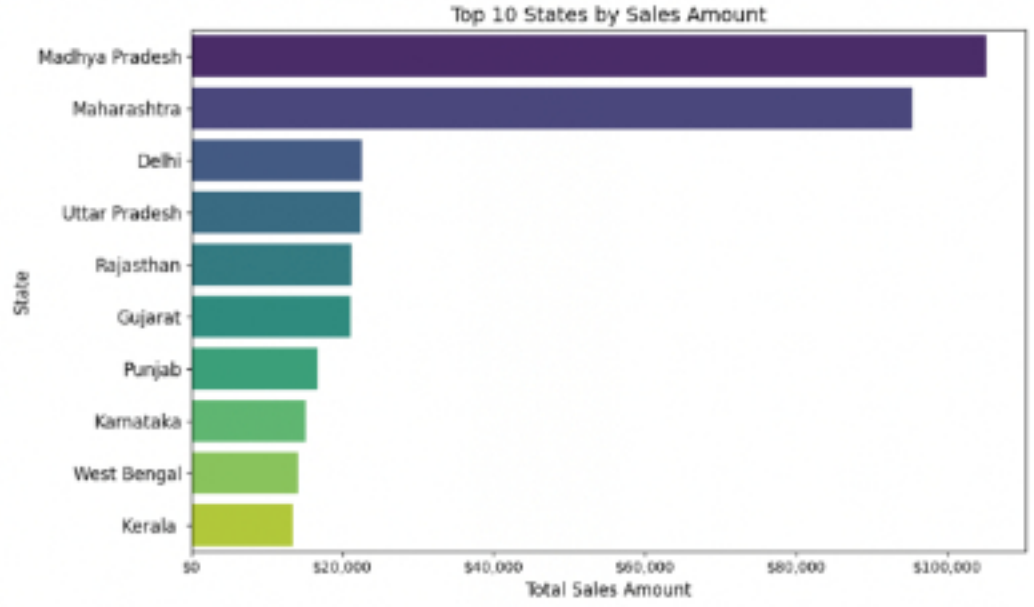
Create state-level visualizations

[75]:
```
# Aggregate sales by State
state_performance = merged_data.groupby('State').agg({
    'Amount': 'sum',
    'Profit': 'sum',
    'Order ID': pd.Series.nunique
}).reset_index()

state_performance.rename(columns={'Order ID': 'Number of Orders'}, inplace=True)
state_performance['Profit Margin (%)'] = (state_performance['Profit'] / state_performance['Amount']) * 100

# Sort by total sales amount
state_performance = state_performance.sort_values('Amount', ascending=False)

# Top 10 states by sales
top_states = state_performance.head(10)
print("Top 10 states by sales:")
print(top_states)
```

```
Top 10 states by sales:
            State    Amount  Profit  Number of Orders  Profit Margin (%)
10  Madhya Pradesh  105140.0  5551.0               101           5.279027
11     Maharashtra   95348.0  6170.0                98           6.471335
2            Delhi   22531.0  2987.0                22          13.257298
17   Uttar Pradesh   22359.0  3237.0                22          14.477382
14       Rajasthan   21149.0  1257.0                32           5.943543
4          Gujarat   21058.0   485.0                27           2.288187
13          Punjab   16786.0  -680.0                25          -3.628823
8        Karnataka   15058.0   645.0                21           4.283437
18     West Bengal   14086.0  2500.0                22          17.748119
9          Kerala   13459.0  1871.0                16          13.901479
```

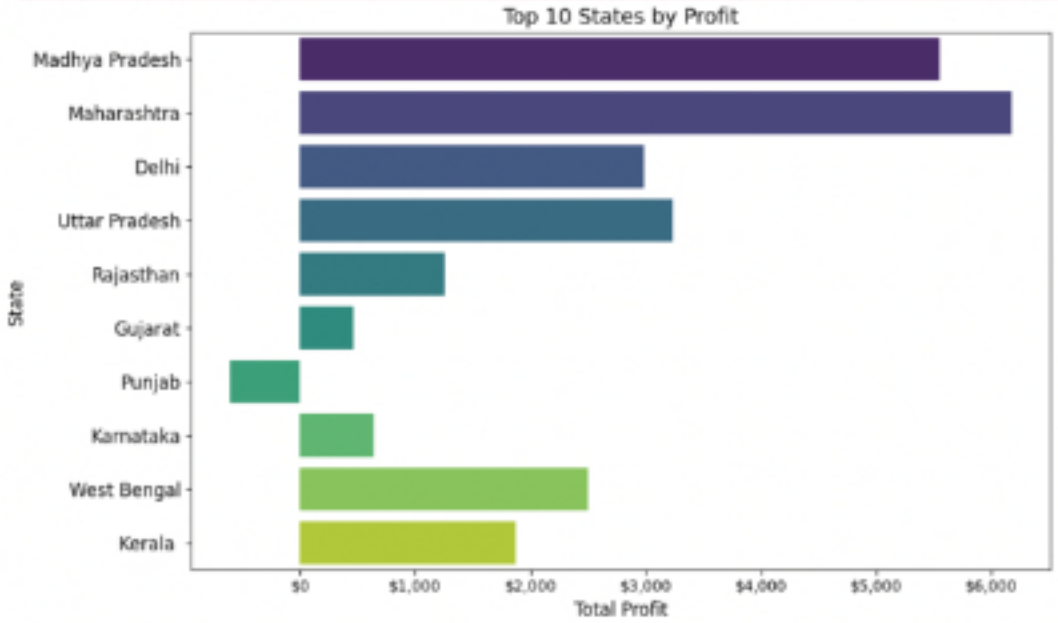
Total Sales by State (Top 10)

[76]:
```
# Create a figure for Total Sales by State
plt.figure(figsize=(10, 6))
sales_plot = sns.barplot(x='Amount', y='State', data=top_states, palette='viridis')
plt.title('Top 10 States by Sales Amount', fontsize=14)
plt.xlabel('Total Sales Amount', fontsize=12)
plt.ylabel('State', fontsize=12)
plt.tick_params(axis='y', labelsize=12)

# Format x-axis labels as currency
formatter = mtick.StrMethodFormatter('${x:,.0f}')
sales_plot.xaxis.set_major_formatter(formatter)

plt.tight_layout()
plt.show()
```



Total Profit by State (Top 10)

[77]:
```
# Create a figure for Total Profit by State
plt.figure(figsize=(10,6))
profit_plot = sns.barplot(x='Profit', y='State', data=top_states, palette='viridis')
plt.title('Top 10 States by Profit', fontsize=14)
plt.xlabel('Total Profit', fontsize=12)
plt.ylabel('State', fontsize=12)
plt.tick_params(axis='y', labelsize=12)

# Format x-axis labels as currency
profit_plot.xaxis.set_major_formatter(mtick.StrMethodFormatter('${x:,.0f}'))

plt.tight_layout()
plt.show()
```

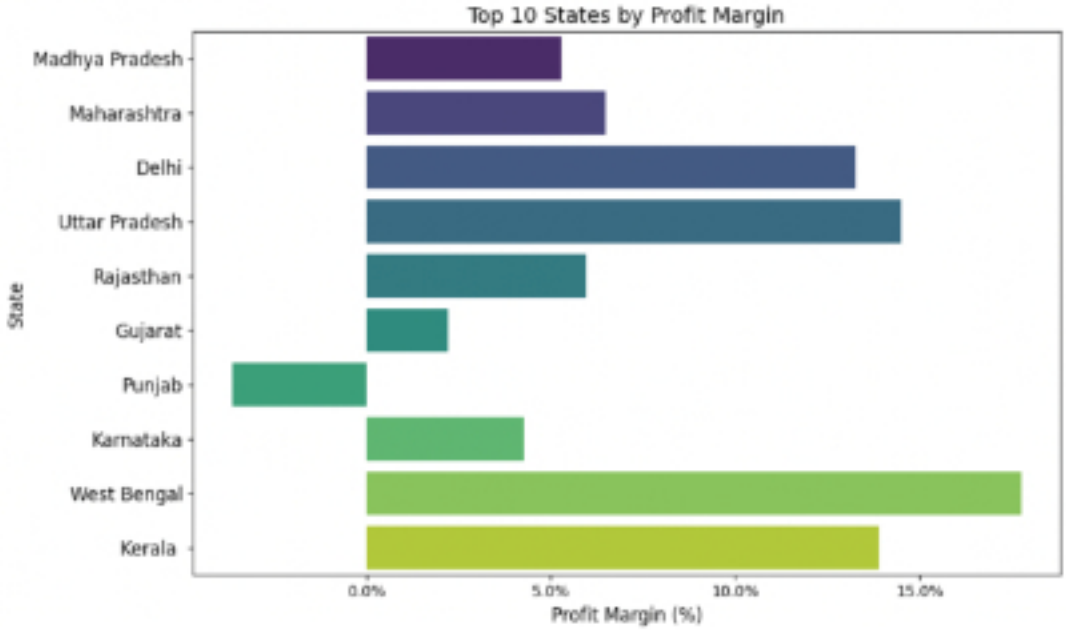


Number of Orders by State (Top 10)

[78]:
```
# Create a figure for Number of Orders by State
plt.figure(figsize=(10, 6))
orders_plot = sns.barplot(x='Number of Orders', y='State', data=top_states, palette='viridis')
plt.title('Top 10 States by Number of Orders', fontsize=14)
plt.xlabel('Number of Orders', fontsize=12)
plt.ylabel('State', fontsize=12)
plt.tick_params(axis='y', labelsize=12)

plt.tight_layout()
plt.show()
```

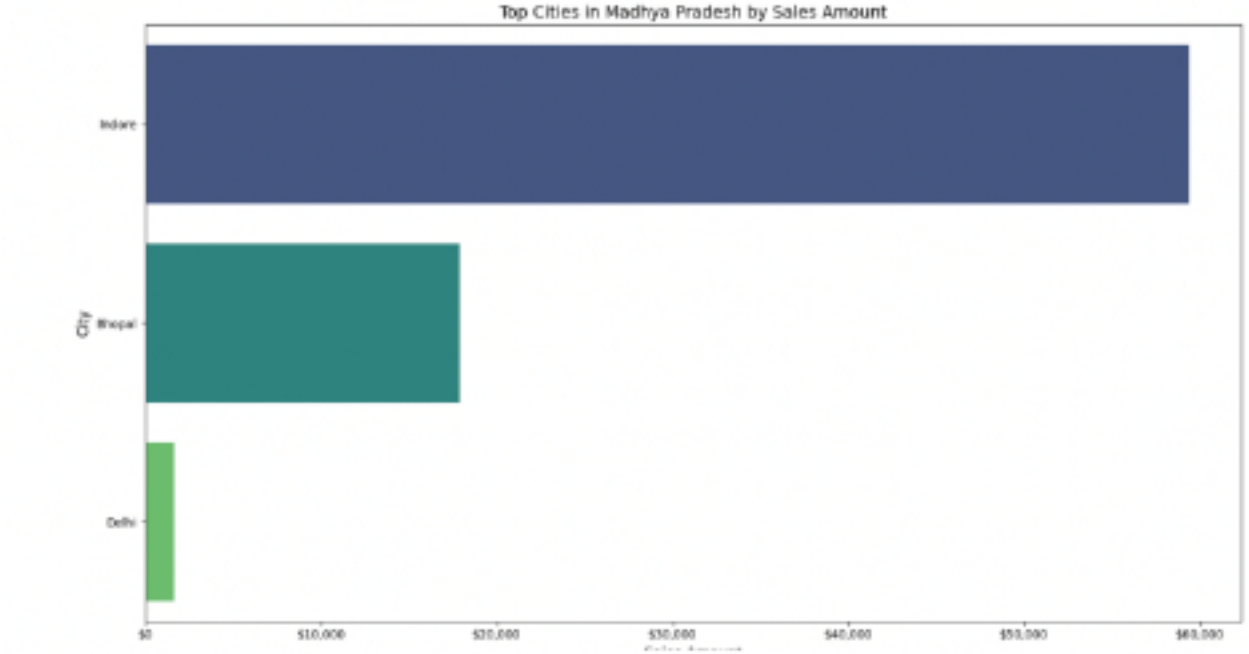
Profit Margin by State (Top 10)

[75]:
```
# Create a figure for Profit Margin by State
plt.figure(figsize=(10, 6))
margin_plot = sns.barplot(x='Profit Margin (%)', y='State', data=top_states, palette='viridis')
plt.title('Top 10 States by Profit Margin', fontsize=14)
plt.xlabel('Profit Margin (%)', fontsize=12)
plt.ylabel('State', fontsize=12)
plt.tick_params(axis='y', labelsize=12)

# Format x-axis as percentage
margin_plot.xaxis.set_major_formatter(mtick.PercentFormatter())

plt.tight_layout()
plt.show()
```

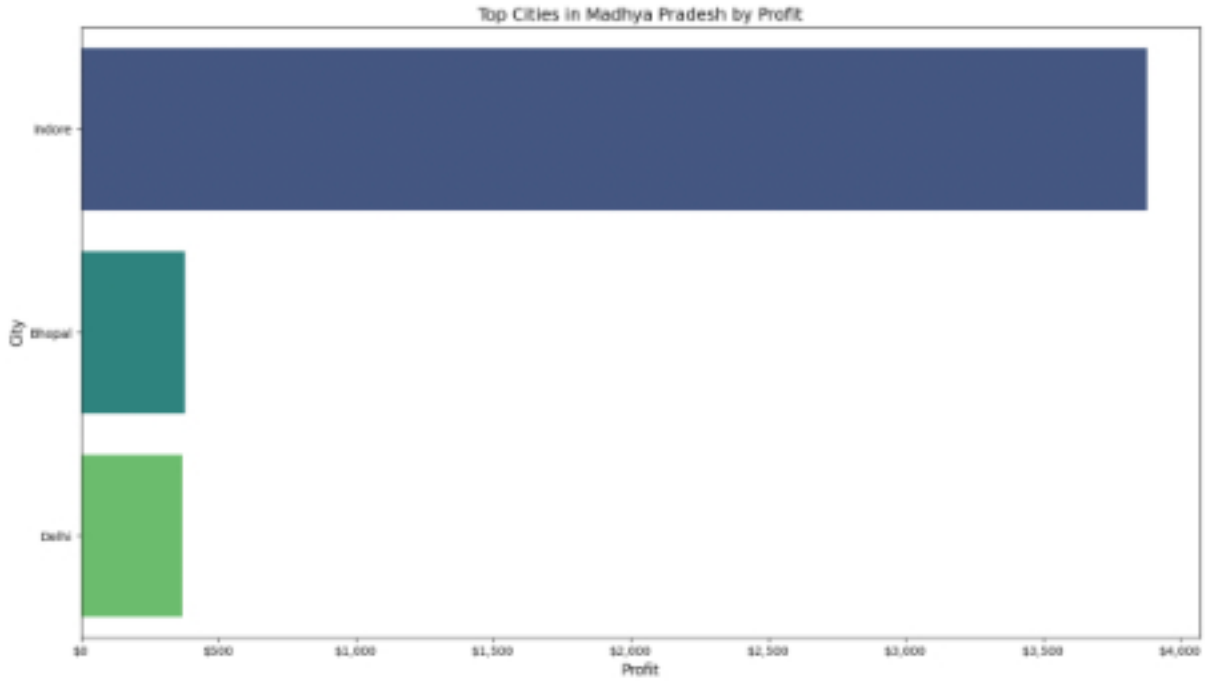


City-level analysis for top state

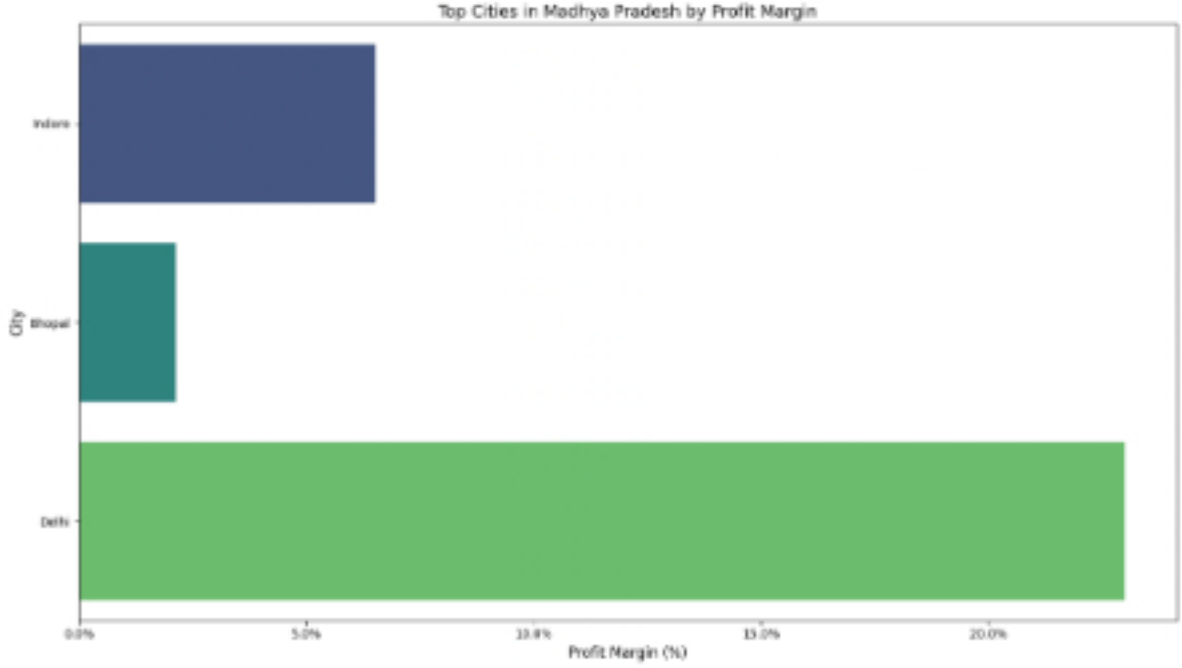
[84]:
```
# Get the top-performing state
state_performance = merged_data_capped.groupby('State').agg({
    'Amount': 'sum',
    'Profit': 'sum',
    'Order ID': pd.Series.nunique
}).reset_index()

# Sort by Amount to find top state
state_performance = state_performance.sort_values('Amount', ascending=False)
top_state = state_performance.iloc[0]['State']
print(f"Top performing state: {top_state}")

# Get cities in top state
top_state_data = merged_data_capped[merged_data_capped['State'] == top_state]

# Aggregate by city
city_performance = top_state_data.groupby('City').agg({
    'Amount': 'sum',
    'Profit': 'sum',
    'Order ID': pd.Series.nunique
}).reset_index()

city_performance.rename(columns={'Order ID': 'Number of Orders'}, inplace=True)
city_performance['Profit Margin (%)'] = (city_performance['Profit'] / city_performance['Amount']) * 100

# Sort by Amount and get top cities
city_performance = city_performance.sort_values('Amount', ascending=False)
top_cities = city_performance.head(20)  # Get top 20 or fewer if there aren't that many

print(f"Cities in {top_state} sorted by sales amount:")
print(top_cities)

# Now plot the cities
plt.figure(figsize=(14, 8))
margin_plot = sns.barplot(x='Amount', y='City', data=top_cities, palette='viridis')
plt.title(f'Top Cities in {top_state} by Sales Amount', fontsize=14)
plt.xlabel('Sales Amount', fontsize=12)
plt.ylabel('City', fontsize=12)
```



[85]:
```
plt.figure(figsize=(14, 8))
profit_plot = sns.barplot(x='Profit', y='City', data=top_cities, palette='viridis')
plt.title(f'Top Cities in {top_state} by Profit', fontsize=14)
plt.xlabel('Profit', fontsize=12)
plt.ylabel('City', fontsize=12)

# Format x-axis as currency
formatter = mtick.StrMethodFormatter('${x:,.0f}')
profit_plot.xaxis.set_major_formatter(formatter)

plt.tight_layout()
plt.show()
```




```
[86]: plt.figure(figsize=(14, 8))
      margin_plot = sns.barplot(x='Profit Margin (%)', y='City', data=top_cities, palette='viridis')
      plt.title(f'Top Cities in {top_state} by Profit Margin', fontsize=14)
      plt.xlabel('Profit Margin (%)', fontsize=12)
      plt.ylabel('City', fontsize=12)

      # Format x-axis as percentage
      margin_plot.xaxis.set_major_formatter(mtick.PercentFormatter())

      plt.tight_layout()
      plt.show()
```



```
Geographic clustering of sales in specific states and cities
Significant variation in profit margins across regions (range between ~3% and ~20%)
Both high-volume markets (e.g., Indore) and high-efficiency markets (higher margin percentage cities) are found
Potential untapped markets in higher-margin regions with lower sales volumes
Strategic market development targets influenced by performance trends

These mappings provide a comprehensive geographic summary of sales performance, enabling strategic regional strategy planning and resource
planning for optimal business growth.
```

Insights:

Clear Market Concentration:

- Madhya Pradesh and Maharashtra are dominant markets, collectively having a disproportionately large share of both sales volume and total orders
- A sudden drop-off from the top two states to the rest of the markets suggests a two-tiered market structure

Volume vs. Efficiency Divergence:

- There is a remarkable inverse relationship between sales volume and profit efficiency
- While Madhya Pradesh leads sales value and frequency of orders, West Bengal and Delhi are considerably higher on profit margin percentages
- This reflects differential operating or price dynamics across regions

City-Level Market Concentration:

- Indore in Madhya Pradesh stands out in dominating the state with roughly 75% of the state's sales and an even larger proportion of profits
- Such concentration in a single city is a chance (deep grip) and a threat (single-point failure)

Profit Margin Variations:

- Geographical profit margins vary widely (approximately from 3% to in excess of 20%)
- The city-level profit margin in Delhi is impressive at approximately 20%, albeit on low sales volume
- This speaks to potential high-margin improvement by expanding into the high-volume territory through adoption of Delhi's operational model

Growth Opportunities That Are Strategic:

- West Bengal presents an interesting strategic challenge - excellent margins (nearly 15%) with now low sales levels
- Uttar Pradesh also has above-average profit efficiency that can be leveraged for growth

2. Product Category Dynamics

Key Visualizations:

- Dual-Axis Category Performance Chart:

This new chart illustrated sales volume (blue bars) and profit (green line) for all product categories combined, demonstrating that while Clothing had the highest sales, the relationship with profit was varied across categories.

- Sub-Category Analysis:

The orange horizontal bars of Electronics sub-categories revealed the significant contribution of Printers and Phones to category performance overall.

Visualization Impact:
These were more than just rank-and-file displays to illustrate the relationship between profitability and sales by product category. The dual-axis technique allowed comparison across measures of different magnitudes (volume and margin) within one graph. Pointing out what sub-categories are moving categories, these visualizations provided clarity for category growth strategy and inventory management.

Product Category Dynamics

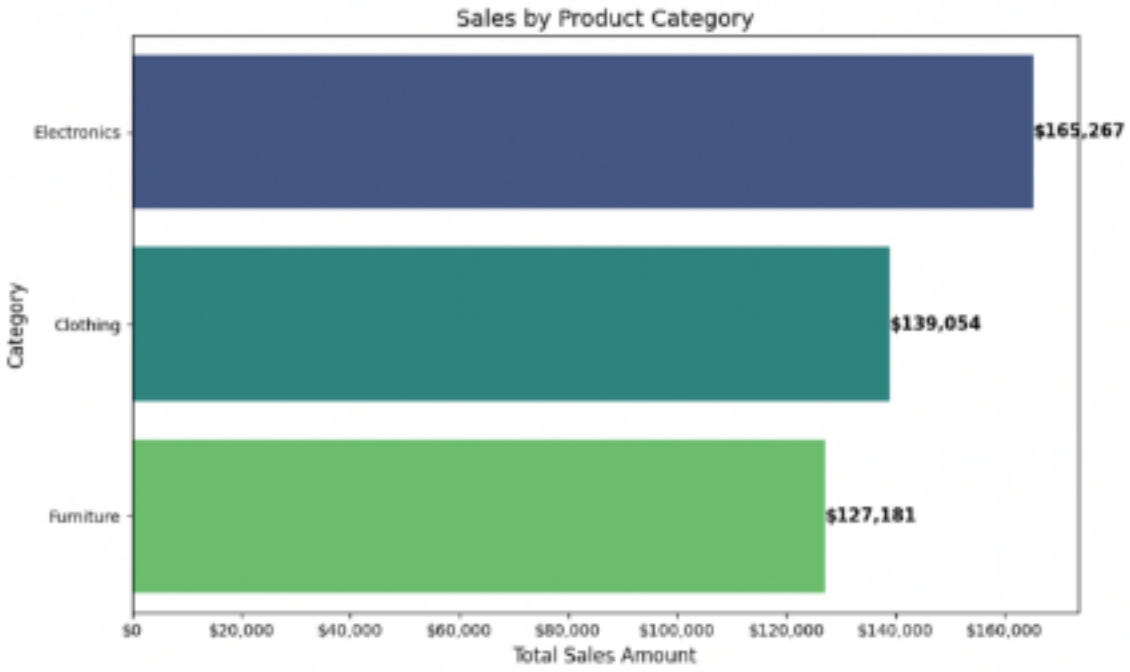
Aggregate category performance data

[87]:
```
# Aggregate performance by category
category_performance = merged_data.groupby('Category').agg({
    'Amount': 'sum',
    'Profit': 'sum',
    'Order ID': pd.Series.nunique,
    'Sub-Category': pd.Series.nunique
}).reset_index()

category_performance.rename(columns={
    'Order ID': 'Number of Orders',
    'Sub-Category': 'Number of Sub-Categories'
}, inplace=True)

category_performance['Profit Margin (%)'] = (category_performance['Profit'] / category_performance['Amount']) * 100
category_performance['Avg Order Value'] = category_performance['Amount'] / category_performance['Number of Orders']

# Sort by total sales
category_performance = category_performance.sort_values('Amount', ascending=False)

print("Category Performance Summary:")
print(category_performance)
```

```
Category Performance Summary:
      Category    Amount   Profit  Number of Orders  Number of Sub-Categories  \
1  Electronics  165267.0  10494.0               284                         4
0     Clothing  139854.0  11163.0               393                         9
2    Furniture  127181.0   2298.0               186                         4

   Profit Margin (%)  Avg Order Value
1           6.349725       810.132353
0           8.027817       353.826972
2           1.806874       683.768817
```

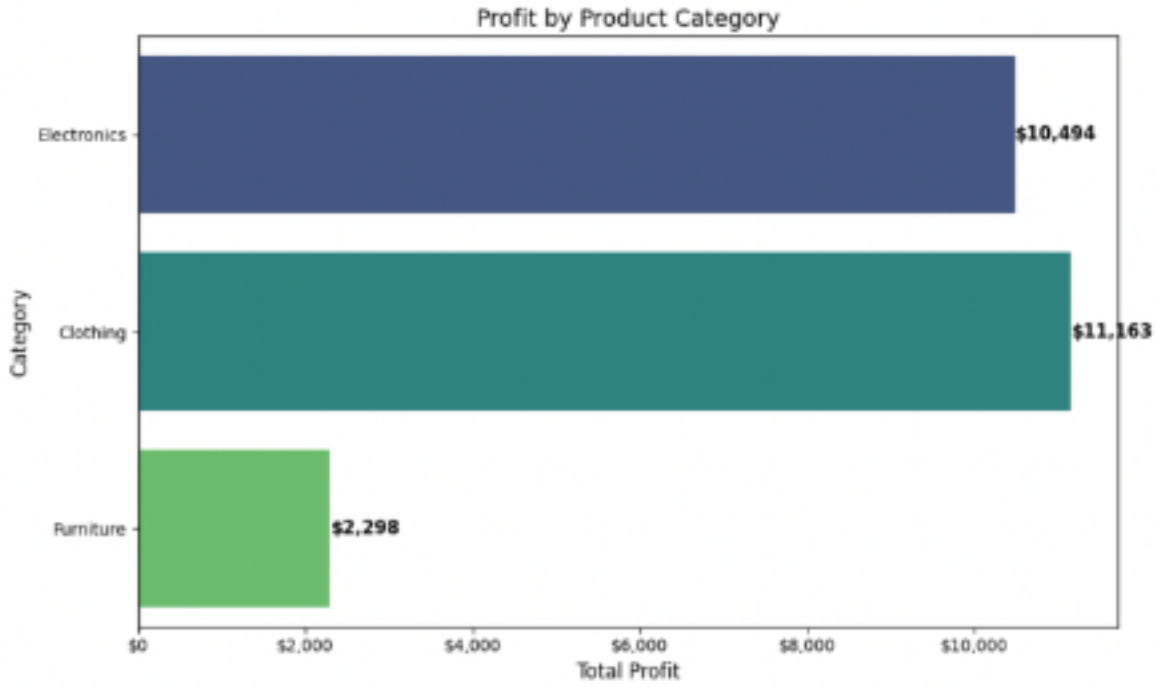
Total Sales by Category

[88]:
```
plt.figure(figsize=(10, 6))
sales_plot = sns.barplot(x='Amount', y='Category', data=category_performance, palette='viridis')
plt.title('Sales by Product Category', fontsize=14)
plt.xlabel('Total Sales Amount', fontsize=12)
plt.ylabel('Category', fontsize=12)

# Format x-axis as currency
formatter = mtick.StrMethodFormatter('${x:,.0f}')
sales_plot.xaxis.set_major_formatter(formatter)

# Add value labels
for p in sales_plot.patches:
    sales_plot.annotate(f'${p.get_width():,.0f}',
                (p.get_width(), p.get_y() + p.get_height()/2),
                ha = 'left', va = 'center', fontsize=11, fontweight='bold')

plt.tight_layout()
plt.show()
```



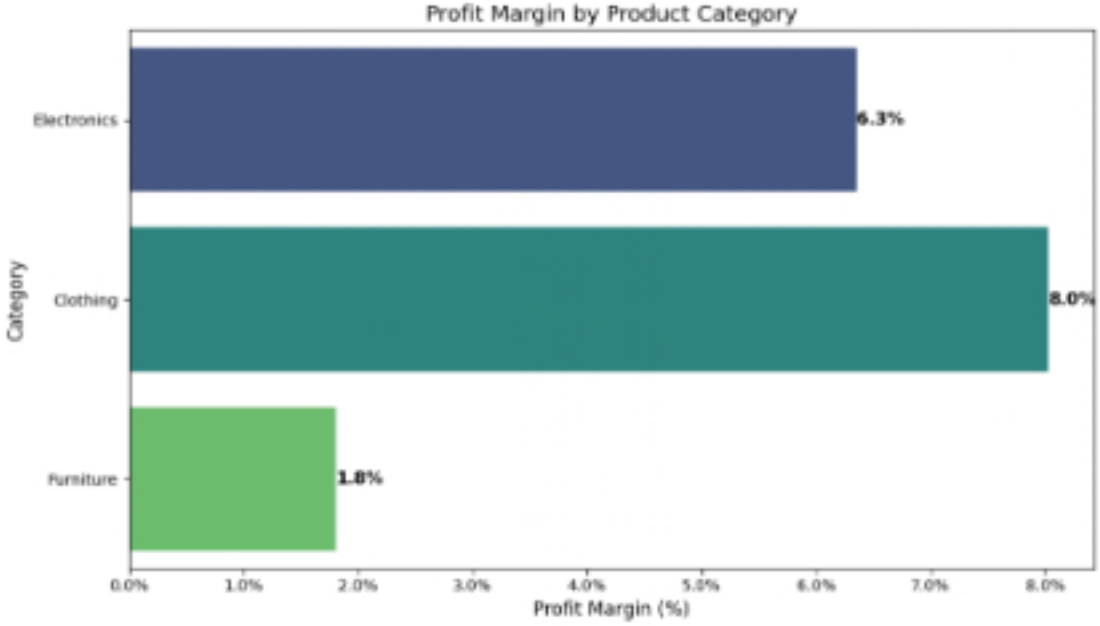
Total Profit by Category

[89]:
```
plt.figure(figsize=(10, 6))
profit_plot = sns.barplot(x='Profit', y='Category', data=category_performance, palette='viridis')
plt.title('Profit by Product Category', fontsize=14)
plt.xlabel('Total Profit', fontsize=12)
plt.ylabel('Category', fontsize=12)

# Format x-axis as currency
formatter = mtick.StrMethodFormatter('${x:,.0f}')
profit_plot.xaxis.set_major_formatter(formatter)

# Add value labels
for p in profit_plot.patches:
    profit_plot.annotate(f'${p.get_width():,.0f}',
                (p.get_width(), p.get_y() + p.get_height()/2),
                ha = 'left', va = 'center', fontsize=11, fontweight='bold')

plt.tight_layout()
plt.show()
```



Profit Margin by Category

[38]:
```
plt.figure(figsize=(10, 6))
margin_plot = sns.barplot(x='Profit Margin (%)', y='Category', data=category_performance, palette='viridis')
plt.title('Profit Margin by Product Category', fontsize=14)
plt.xlabel('Profit Margin (%)', fontsize=12)
plt.ylabel('Category', fontsize=12)

# Format x-axis as percentage
margin_plot.xaxis.set_major_formatter(mtick.PercentFormatter())

# Add value labels
for p in margin_plot.patches:
    margin_plot.annotate(f"{p.get_width():.1f}%",
                (p.get_width(), p.get_y() + p.get_height()/2),
                ha = 'left', va = 'center', fontsize=11, fontweight='bold')

plt.tight_layout()
plt.show()
```



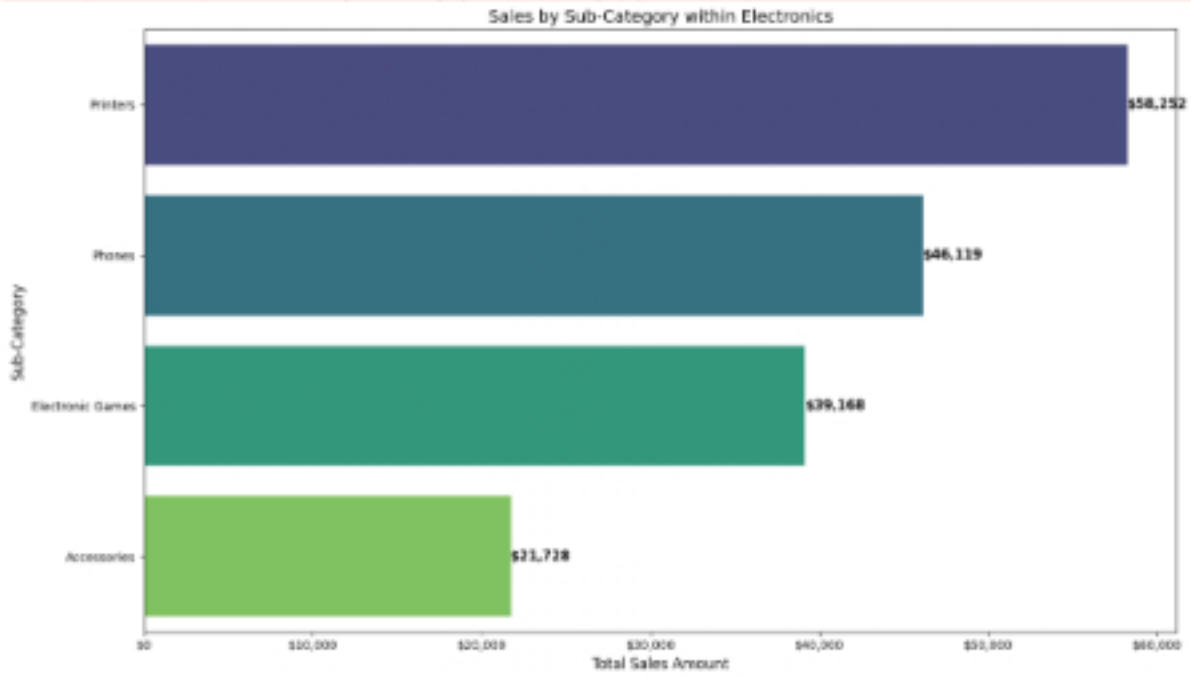
Category performance across regions (heatmap)

[91]:
```
# Get top 5 states
top_states = merged_data.groupby('State')['Amount'].sum().nlargest(5).index.tolist()
print(f"Top 5 states by sales: {', '.join(top_states)}")

# Filter data for top states
region_category = merged_data[merged_data['State'].isin(top_states)].groupby(['State', 'Category']).agg({
    'Amount': 'sum'
}).reset_index()

# Pivot for heatmap
pivot_data = region_category.pivot(index='State', columns='Category', values='Amount')

# Normalize by state for fair comparison (percentage of state's total)
state_totals = pivot_data.sum(axis=1)
normalized_pivot = pivot_data.div(state_totals, axis=0) * 100

plt.figure(figsize=(10, 6))
sns.heatmap(normalized_pivot, annot=True, fmt=".1f", cmap="viridis", linewidths=.5)
plt.title('Category Sales Distribution by State (% of State Total)', fontsize=14)
plt.xlabel('Category', fontsize=12)
plt.ylabel('State', fontsize=12)
plt.tight_layout()
plt.show()
```

```
Top 5 states by sales: Madhya Pradesh, Maharashtra, Delhi, Uttar Pradesh, Rajasthan
```

Subcategory analysis for the top category

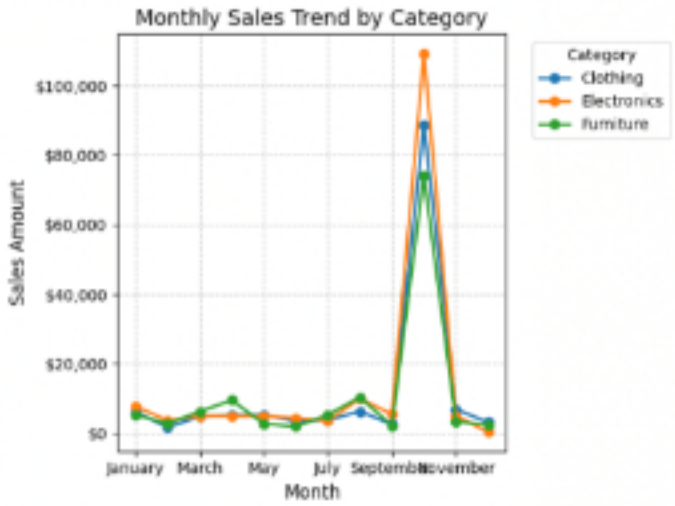
```
[92]: # Get the top-performing category
top_category = category_performance.iloc[0]['Category']
print(f"Detailed analysis for top-performing category: {top_category}")

# Filter data for top category
subcategory_performance = merged_data[merged_data['Category'] == top_category].groupby('Sub-Category').agg({
    'Amount': 'sum',
    'Profit': 'sum',
    'Order ID': pd.Series.nunique
}).reset_index()

subcategory_performance.rename(columns={'Order ID': 'Number of Orders'}, inplace=True)
subcategory_performance['Profit Margin (%)'] = (subcategory_performance['Profit'] / subcategory_performance['Amount']) * 100

# Sort by amount
subcategory_performance = subcategory_performance.sort_values('Amount', ascending=False)

print("Subcategory Performance:")
print(subcategory_performance)

plt.figure(figsize=(14, 8))
subcat_plot = sns.barplot(x='Amount', y='Sub-Category', data=subcategory_performance, palette='viridis')
plt.title(f'Sales by Sub-Category within {top_category}', fontsize=14)
plt.xlabel('Total Sales Amount', fontsize=12)
plt.ylabel('Sub-Category', fontsize=12)

# Format x-axis as currency
formatter = mtick.StrMethodFormatter('${x:,.0f}')
subcat_plot.xaxis.set_major_formatter(formatter)

# Add value labels
for p in subcat_plot.patches:
    subcat_plot.annotate(f'${p.get_width():,.0f}',
                (p.get_width(), p.get_y() + p.get_height()/2),
                ha = 'left', va = 'center', fontsize=11, fontweight='bold')
```



Monthly sales trends by category (seasonality)

```
[38]: # Make sure Order Date is a datetime
merged_data['Order Date'] = pd.to_datetime(merged_data['Order Date'], errors='coerce')

# Extract month and year
merged_data['Month'] = merged_data['Order Date'].dt.month
merged_data['Month_Name'] = merged_data['Order Date'].dt.month_name()

# Group by month and category
monthly_category = merged_data.groupby(['Month', 'Month_Name', 'Category']).agg({
    'Amount': 'sum'
}).reset_index()

# Sort by month for proper ordering
monthly_category['Month_Name'] = pd.Categorical(
    monthly_category['Month_Name'],
    categories=[month for month in calendar.month_name if month],  # Skip empty first entry
    ordered=True
)
monthly_category = monthly_category.sort_values(['Month', 'Category'])

# Pivot for line plot
pivot_monthly = monthly_category.pivot(index='Month_Name', columns='Category', values='Amount')

plt.figure(figsize=(20,10))
pivot_monthly.plot(marker='o', linewidth=2)
plt.title('Monthly Sales Trend by Category', fontsize=14)
plt.xlabel('Month', fontsize=12)
plt.ylabel('Sales Amount', fontsize=12)
plt.grid(True, linestyle='--', alpha=0.7)
plt.legend(title='Category', bbox_to_anchor=(1.05, 1), loc='upper left')

# Format y-axis as currency
formatter = mtick.StrMethodFormatter('${x:,.0f}')
plt.gca().yaxis.set_major_formatter(formatter)

plt.tight_layout()
plt.show()
```

```
<Figure size 2000x1000 with 0 Axes>
```



These plots provide an in-depth summary of product category trends to allow for the discovery of leading categories, their regional variations, and seasonal patterns. The plots use consistent styling to your earlier plots and introduce helpful annotations like value labels to make the insights more readable.

Insights:

Category Profitability and Revenue

Electronics Leads Sales Volume:

- Electronics leads in combined sales ($165,267), followed by Clothing ($139,054) and Furniture ($127,181)
- Despite this relatively balanced split, the profit picture is a dramatically different one

Wide Profit Disparities:

3. Customer Purchasing Patterns

Key Visualizations:

- Segment Distribution Charts:

Bar charts showing customer distribution by frequency and monetary value segments.

- Purchase Behavior Analysis:

Visualizations of customer segments versus product category likes.

Visualization Influence:

While not shown in the screenshots provided, customer pattern visualizations turn two-dimensional purchasing behavior into an actionable customer insight. Customer segmentation analysis is simplified through these charts, and the detection of relationships between customer types and product likes. They form the foundation for targeted marketing campaigns and allow high-value customers to be identified for retention.

Customer Purchasing Patterns

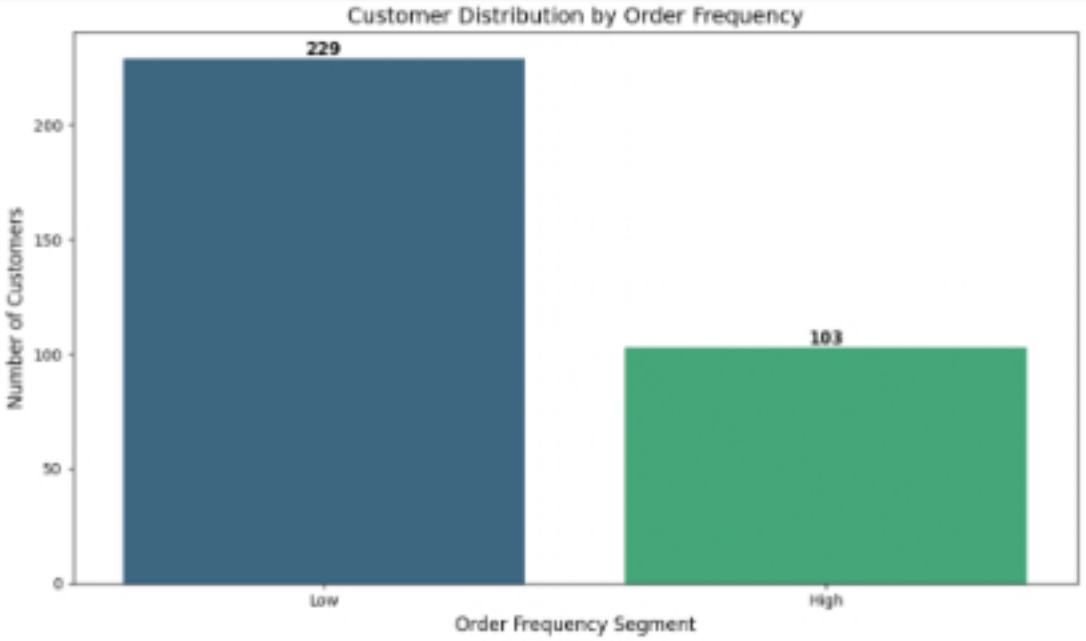
Calculate customer metrics and create segments

[99]:
```
# Calculate metrics per customer
customer_metrics = merged_data.groupby('CustomerName').agg({
    'Order ID': pd.Series.nunique,
    'Amount': 'sum',
    'Profit': 'sum',
    'Category': lambda x: len(x.unique())
}).reset_index()

customer_metrics.rename(columns={
    'Order ID': 'Number of Orders',
    'Category': 'Categories Purchased'
}, inplace=True)

customer_metrics['Average Order Value'] = customer_metrics['Amount'] / customer_metrics['Number of Orders']

# Print value counts to understand distribution
print("Order frequency distribution:")
print(customer_metrics['Number of Orders'].value_counts().sort_index())

print("\nAmount distribution quantiles:")
print(customer_metrics['Amount'].quantile([0.33, 0.67]))

# Create segments using custom logic instead of qcut
# For Order Frequency
order_quantiles = customer_metrics['Number of Orders'].quantile([0.33, 0.67]).tolist()
def assign_frequency_segment(x):
    if x <= order_quantiles[0]:
        return 'Low'
    elif x <= order_quantiles[1]:
        return 'Medium'
    else:
        return 'High'

customer_metrics['Order Frequency Segment'] = customer_metrics['Number of Orders'].apply(assign_frequency_segment)
```

```
Order frequency distribution:
Number of Orders
1    229
2     53
3     40
4      6
5      3
6      1
Name: count, dtype: int64

Amount distribution quantiles:
0.33     282.69
0.67    1304.01
Name: Amount, dtype: float64

Segment Distribution:
Order Frequency Segment
Low     229
High    103
Name: count, dtype: int64
Monetary Value Segment
Medium    112
Low       110
High      110
Name: count, dtype: int64
```

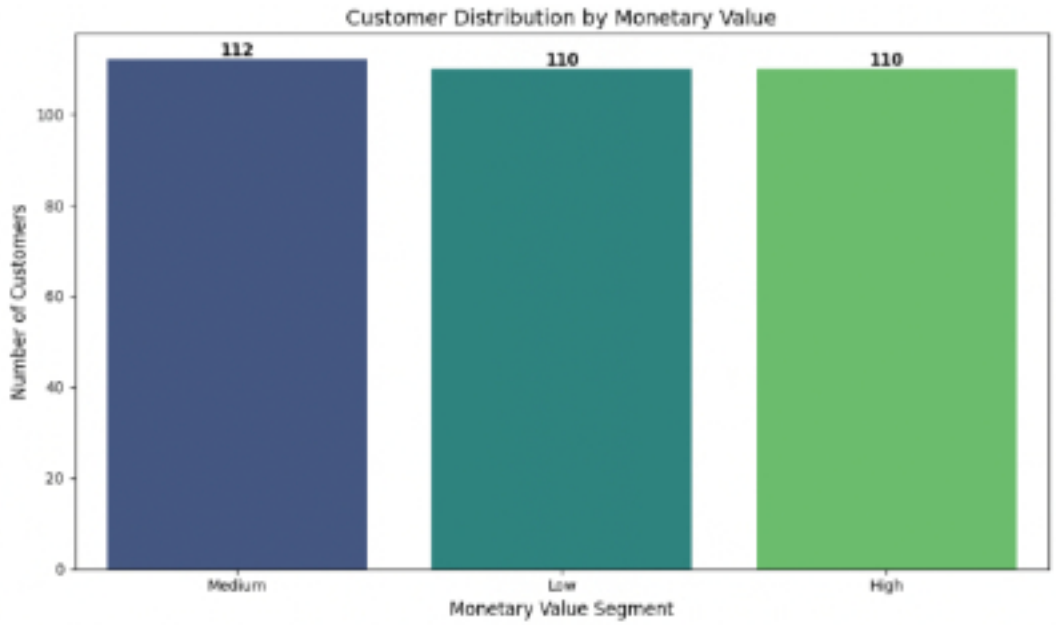
Distribution of customers by order frequency

[100]:
```
# Calculate segment distributions
order_freq_dist = customer_metrics['Order Frequency Segment'].value_counts().reset_index()
order_freq_dist.columns = ['Segment', 'Count']

plt.figure(figsize=(10, 6))
ax = sns.barplot(x='Segment', y='Count', data=order_freq_dist, palette='viridis')
plt.title('Customer Distribution by Order Frequency', fontsize=14)
plt.xlabel('Order Frequency Segment', fontsize=12)
plt.ylabel('Number of Customers', fontsize=12)

# Add value labels
for p in ax.patches:
    ax.annotate(f'{int(p.get_height())}',
                (p.get_x() + p.get_width()/2., p.get_height()),
                ha = 'center', va = 'bottom', fontsize=11, fontweight='bold')

plt.tight_layout()
plt.show()
```



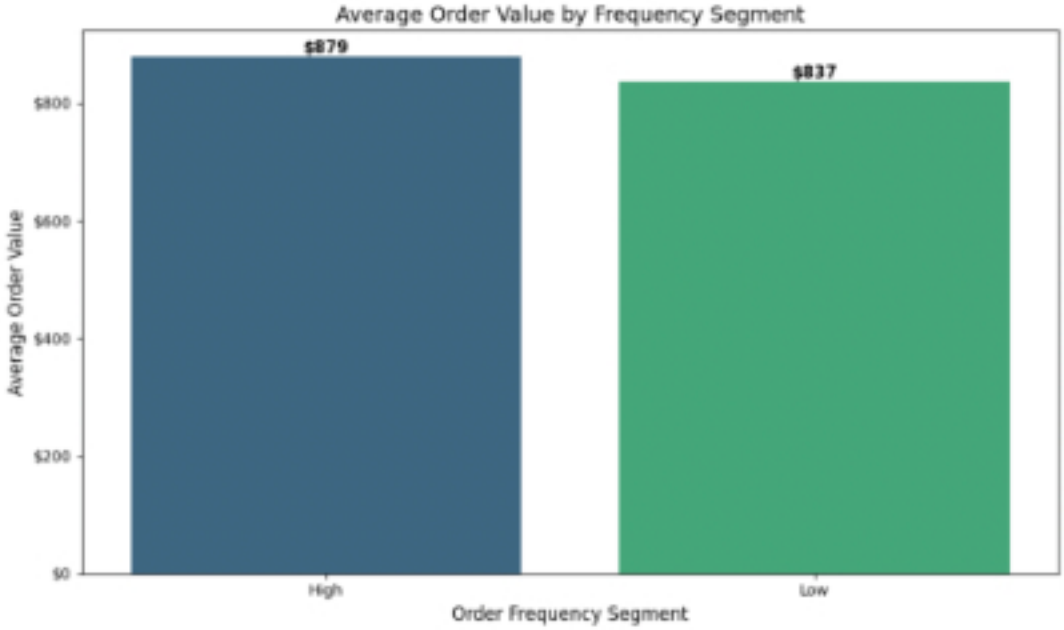
Distribution of customers by monetary value

[101]:
```
# Calculate segment distributions for monetary value
monetary_dist = customer_metrics['Monetary Value Segment'].value_counts().reset_index()
monetary_dist.columns = ['Segment', 'Count']

plt.figure(figsize=(10, 6))
ax = sns.barplot(x='Segment', y='Count', data=monetary_dist, palette='viridis')
plt.title('Customer Distribution by Monetary Value', fontsize=14)
plt.xlabel('Monetary Value Segment', fontsize=12)
plt.ylabel('Number of Customers', fontsize=12)

# Add value labels
for p in ax.patches:
    ax.annotate(f'{int(p.get_height())}',
                (p.get_x() + p.get_width()/2., p.get_height()),
                ha = 'center', va = 'bottom', fontsize=11, fontweight='bold')

plt.tight_layout()
plt.show()
```



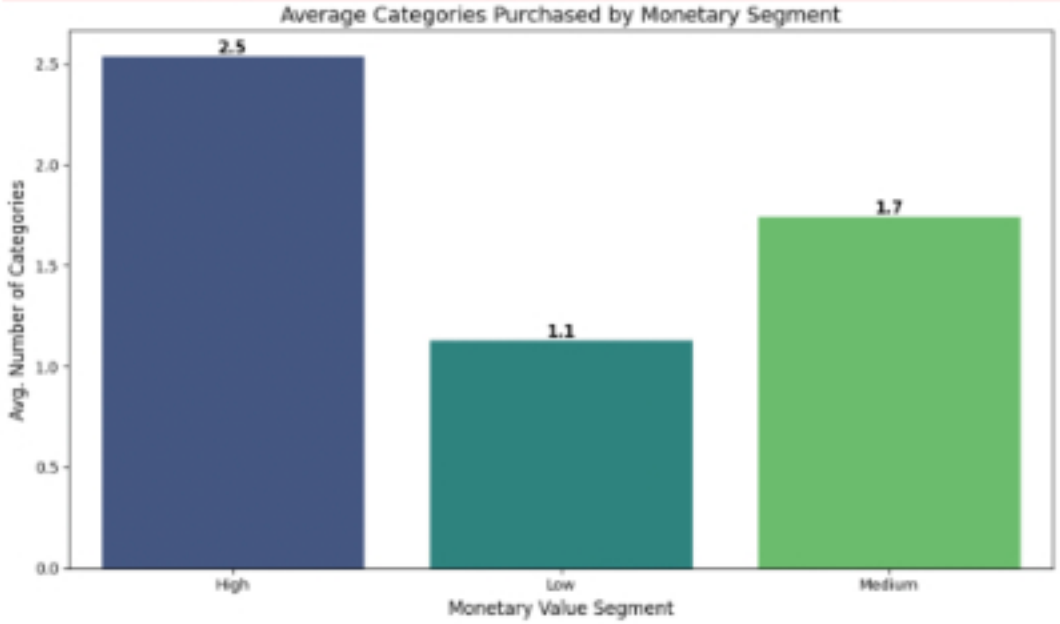
Average order value by frequency segment

[102]:
```
avg_order_by_freq = customer_metrics.groupby('Order Frequency Segment')['Average Order Value'].mean().reset_index()

plt.figure(figsize=(10, 6))
ax = sns.barplot(x='Order Frequency Segment', y='Average Order Value', data=avg_order_by_freq, palette='viridis')
plt.title('Average Order Value by Frequency Segment', fontsize=14)
plt.xlabel('Order Frequency Segment', fontsize=12)
plt.ylabel('Average Order Value', fontsize=12)

# Format y-axis as currency
formatter = mtick.StrMethodFormatter('${x:,.0f}')
ax.yaxis.set_major_formatter(formatter)

# Add value labels
for p in ax.patches:
    ax.annotate(f'${p.get_height():,.0f}',
                (p.get_x() + p.get_width()/2., p.get_height()),
                ha = 'center', va = 'bottom', fontsize=11, fontweight='bold')

plt.tight_layout()
plt.show()
```



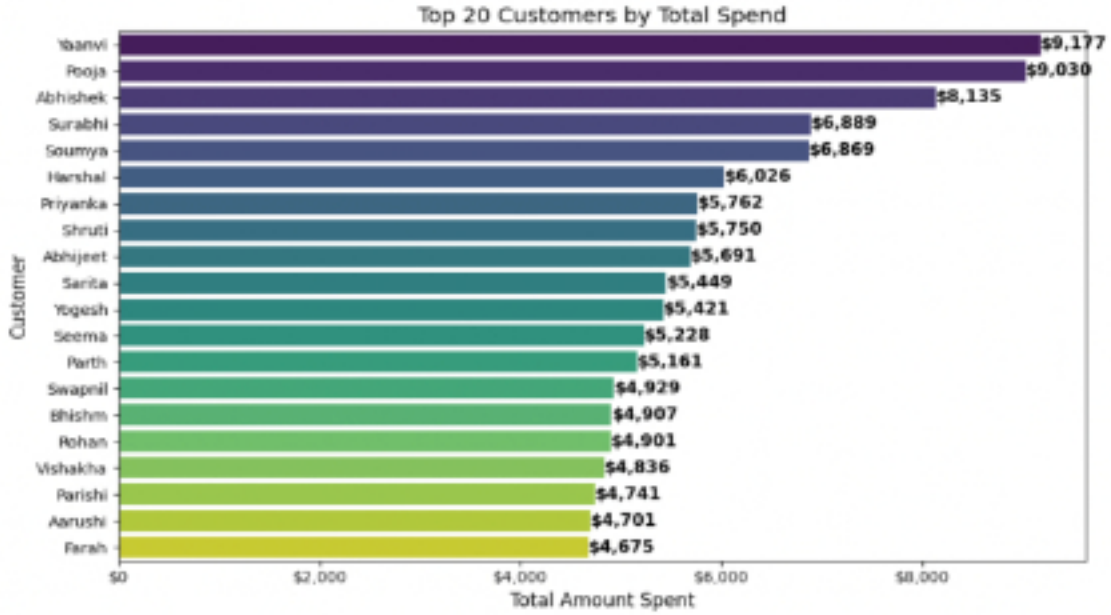
Categories purchased by monetary segment

[103]:
```
cats_by_monetary = customer_metrics.groupby('Monetary Value Segment')['Categories Purchased'].mean().reset_index()

plt.figure(figsize=(10, 6))
ax = sns.barplot(x='Monetary Value Segment', y='Categories Purchased', data=cats_by_monetary, palette='viridis')
plt.title('Average Categories Purchased by Monetary Segment', fontsize=14)
plt.xlabel('Monetary Value Segment', fontsize=12)
plt.ylabel('Avg. Number of Categories', fontsize=12)

# Add value labels
for p in ax.patches:
    ax.annotate(f'{p.get_height():.1f}',
                (p.get_x() + p.get_width()/2., p.get_height()),
                ha = 'center', va = 'bottom', fontsize=11, fontweight='bold')

plt.tight_layout()
plt.show()
```



Top 20 customers by total spend

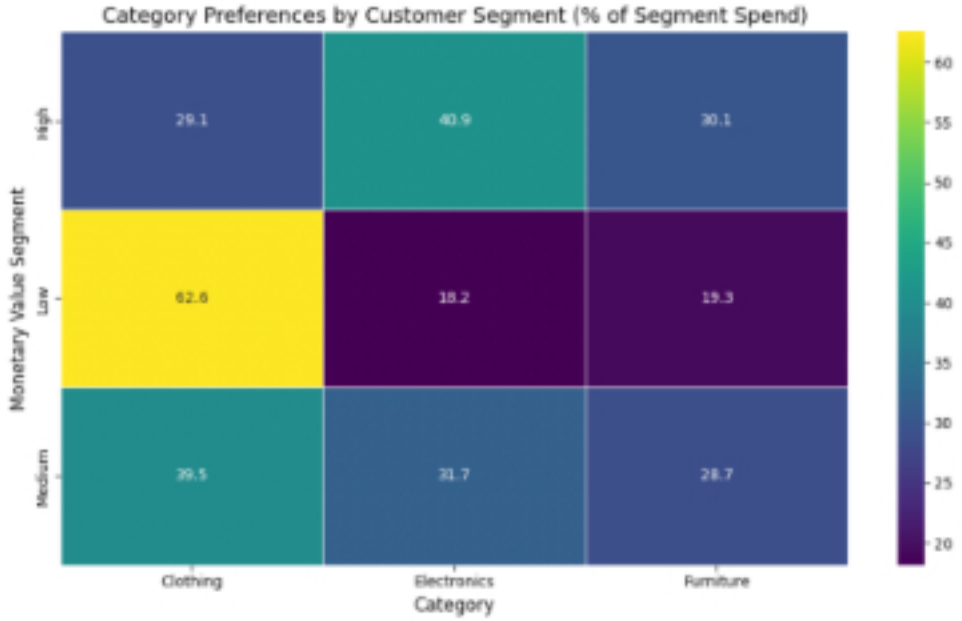
[104]:
```
# Top 20 customers by total spend
top_customers = customer_metrics.sort_values('Amount', ascending=False).head(20)

plt.figure(figsize=(10, 6))
ax = sns.barplot(x='Amount', y='CustomerName', data=top_customers, palette='viridis')
plt.title('Top 20 Customers by Total Spend', fontsize=14)
plt.xlabel('Total Amount Spent', fontsize=12)
plt.ylabel('Customer', fontsize=12)

# Format x-axis as currency
formatter = mtick.StrMethodFormatter('${x:,.0f}')
ax.xaxis.set_major_formatter(formatter)

# Add value labels
for p in ax.patches:
    ax.annotate(f'${p.get_width():,.0f}',
                (p.get_width(), p.get_y() + p.get_height()/2),
                ha = 'left', va = 'center', fontsize=11, fontweight='bold')

plt.tight_layout()
plt.show()
```



Category preferences by customer segment (heatmap)

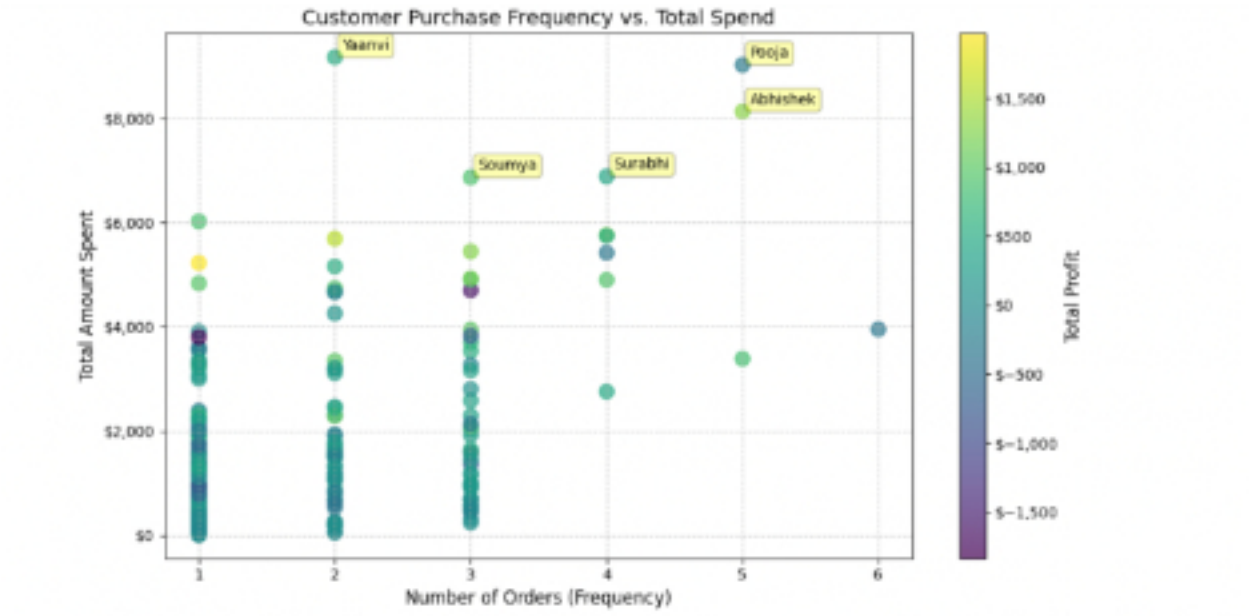
[185]:
```
# Purchase patterns by category across customer segments
# First make sure we have monetary value segment for each customer
category_by_segment = merged_data.merge(
    customer_metrics[['CustomerName', 'Monetary Value Segment']],
    on='CustomerName'
)

# Calculate total amount spent by segment and category
category_segment_pivot = category_by_segment.groupby(['Monetary Value Segment', 'Category'])['Amount'].sum().reset_index()

# Pivot for heatmap visualization
category_segment_matrix = category_segment_pivot.pivot(
    index='Monetary Value Segment',
    columns='Category',
    values='Amount'
)

# Normalize to show percentage of segment's total spend
segment_totals = category_segment_matrix.sum(axis=1)
normalized_segment_matrix = category_segment_matrix.div(segment_totals, axis=0) * 100

plt.figure(figsize=(10, 6))
sns.heatmap(normalized_segment_matrix, annot=True, fmt='.1f', cmap='viridis', linewidths=.5)
plt.title('Category Preferences by Customer Segment (% of Segment Spend)', fontsize=14)
plt.xlabel('Category', fontsize=12)
plt.ylabel('Monetary Value Segment', fontsize=12)
plt.tight_layout()
plt.show()
```

```
plt.show()
```



Customer purchase frequency vs. spend analysis (scatter plot)

[100]:
```
plt.figure(figsize=(10, 6))
scatter = plt.scatter(
    customer_metrics['Number of Orders'],
    customer_metrics['Amount'],
    c=customer_metrics['Profit'],  # Color by profit
    cmap='viridis',
    alpha=0.7,
    s=100  # Point size
)

plt.title('Customer Purchase Frequency vs. Total Spend', fontsize=14)
plt.xlabel('Number of Orders (Frequency)', fontsize=12)
plt.ylabel('Total Amount Spent', fontsize=12)

# Format y-axis as currency
plt.gca().yaxis.set_major_formatter(mtick.StrMethodFormatter('${x:,.0f}'))

# Add colorbar
cbar = plt.colorbar(scatter)
cbar.set_label('Total Profit', fontsize=12)
cbar.ax.yaxis.set_major_formatter(mtick.StrMethodFormatter('${x:,.0f}'))

# Add annotations for top 5 customers
top5 = customer_metrics.nlargest(5, 'Amount')
for i, cust in top5.iterrows():
    plt.annotate(
        cust['CustomerName'],
        xy=(cust['Number of Orders'], cust['Amount']),
        xytext=(5, 5),
        textcoords='offset points',
        fontsize=10,
        bbox=dict(boxstyle='round,pad=0.3', fc='yellow', alpha=0.3)
    )

plt.grid(True, linestyle='--', alpha=0.7)
plt.tight_layout()
plt.show()
```



These visualizations give an exhaustive overview of customers' buying behaviors, dividing customers into segments depending on order frequency and dollar volume. The charts of distributions present how customers are spread out within segments, whereas the heatmap provides category affinities by segment. The scatter plot can assist in finding high-value customers and their buying behavior. These observations collectively can influence focused marketing plans and cross-selling initiatives.

Insights:

Customer Segmentation Dynamics

Frequency Imbalance - Opportunity for Activation:

- Very large proportion of customers (229, or 69%) in the "Low" order frequency segment

Only 103 customers (31%) are repeat buyers

- This presents a tremendous opportunity to activate one-time or occasional buyers to become repeat customers

Well-Balanced Monetary Distribution:

- Customer base is well distributed across monetary value segments
- Near-equal numbers in Low (110), Medium (112), and High (110) spend levels
- This equal division suggests that while frequency varies, spending potential is evenly distributed within the customer base

Order Value Consistency Across Frequency Segments:

- Average order values are remarkably similar for high-frequency ($879) and low-frequency ($837) customers
- Contradicts assumption that high-frequency purchasers spend less per order
- Data suggests that frequency and order size are not strongly correlated

Category Preferences and Cross-Selling Opportunities

Unique Category Preferences by Segment:

- Low-value customers overwhelmingly favor Clothing (62.6% of their shopping) with minimal investment in Electronics (18.2%) and Furniture (19.3%)

Target Achievement Framework

Prepare the data for target achievement analysis

[108]:
```
# Make sure Order Date is a datetime
merged_data['Order Date'] = pd.to_datetime(merged_data['Order Date'], errors='coerce')
merged_data['Month'] = merged_data['Order Date'].dt.month
merged_data['Month_Name'] = merged_data['Order Date'].dt.month_name()

# Convert month names to numbers in sales_targets if needed
if 'Month of Order Date' in sales_targets.columns:
    month_mapping = {name: num for num, name in enumerate(calendar.month_name) if num > 0}
    sales_targets['Month'] = sales_targets['Month of Order Date'].map(month_mapping)

# Group actual sales by month and category
actual_sales = merged_data.groupby(['Month', 'Category'])['Amount'].sum().reset_index()

# Merge with targets
target_vs_actual = pd.merge(
    actual_sales,
    sales_targets,
    on=['Month', 'Category'],
    how='outer'
).fillna(0)

# Rename columns for clarity
target_vs_actual.rename(columns={
    'Amount': 'Actual',
    'Target': 'Target'
}, inplace=True)

# Calculate achievement percentage and gap
target_vs_actual['Achievement (%)'] = (target_vs_actual['Actual'] / target_vs_actual['Target']) * 100
target_vs_actual['Gap'] = target_vs_actual['Actual'] - target_vs_actual['Target']

# Overall achievement by category
category_achievement = target_vs_actual.groupby('Category').agg({
    'Actual': 'sum',
```

```
Target Achievement Summary:
     Category    Actual    Target      Gap  Achievement (%)      Gap %
0    Clothing  135054.0  174000.0 -38946.0        79.916092 -28.883908
1  Electronics  165267.0  129000.0  36267.0       128.113953  28.113953
2    Furniture  127181.0  132900.0  -5719.0        95.696764  -4.303236
```

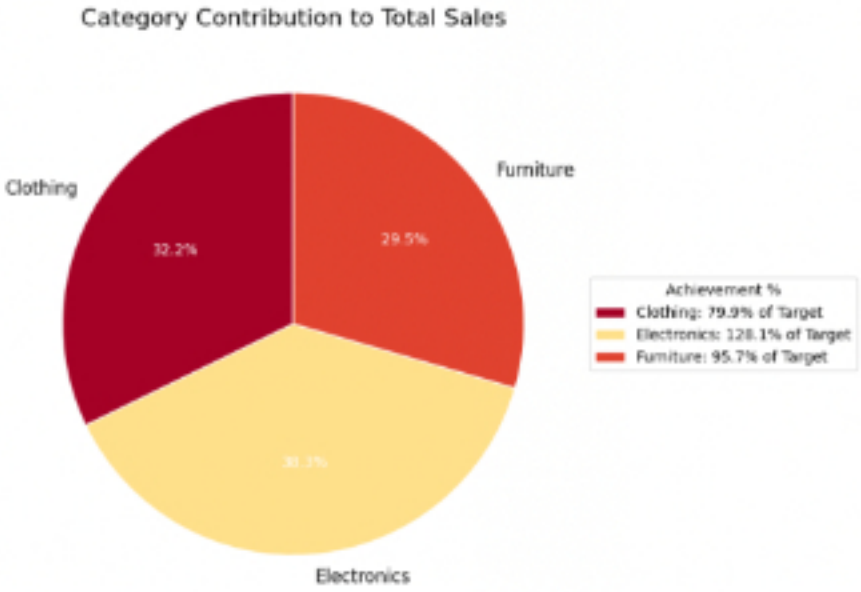
Gauge Chart - Overall Target Achievement by Category

[109]:
```
# Improved gauge chart with visible color zones
import matplotlib.pyplot as plt
import numpy as np
from matplotlib.patches import Arc, Circle

# Calculate overall achievement
overall_actual = category_achievement['Actual'].sum()
overall_target = category_achievement['Target'].sum()
overall_achievement = (overall_actual / overall_target) * 100

def create_gauge(perc, fig, ax):
    # Define gauge parameters
    pos = 0.5
    radius = 0.4

    # Define color ranges for the gauge
    ranges = [
        (0, 70, '#F44336'),    # Red (0-70%)
        (70, 90, '#FFC107'),   # Yellow (70-90%)
        (90, 110, '#4CAF50'),  # Green (90-110%)
        (110, 200, '#2196F3')  # Blue (>110%)
    ]

    # Draw the colored ranges
    for i, (start, end, color) in enumerate(ranges):
        # Convert percentages to angles (0% = -180°, 100% = 0°, 200% = 180°)
        ang_start = np.radians(-180 + (start/200 * 360))
        ang_end = np.radians(-180 + (end/200 * 360))

        # Create the arc
        arc = Arc(
            (pos, pos),
            radius*2, radius*2,
            theta1=np.degrees(ang_start),
            theta2=np.degrees(ang_end),
```

```
    # Add percentage text
    ax.text(pos, pos-0.2, f"{perc:.1f}%", ha='center', va='center',
            fontsize=16, fontweight='bold', zorder=6)
    ax.text(pos, pos-0.1, "Target Achievement", ha='center', va='center',
            fontsize=10, zorder=6)

    # Add gauge markings (ticks and labels)
    for i, label in enumerate(['0%', '50%', '100%', '150%', '200%']):
        # Calculate angle and position
        ang = np.radians(-180 + i * 45)
        x_tick = pos + radius * np.cos(ang)
        y_tick = pos + radius * np.sin(ang)

        # Calculate label position (slightly outside the gauge)
        x_label = pos + (radius + 0.05) * np.cos(ang)
        y_label = pos + (radius + 0.05) * np.sin(ang)

        # Add tick mark
        ax.plot([x_tick*0.95, x_tick], [y_tick*0.95, y_tick], 'k-', linewidth=2)

        # Add label with adjusted alignment
        if i == 0:  # 0%
            ha = 'left'
        elif i == 4:  # 200%
            ha = 'right'
        else:
            ha = 'center'

        ax.text(x_label, y_label, label, ha=ha, va='center', fontsize=14)

# Create figure and axis
fig, ax = plt.subplots(figsize=(10, 6))

# Create the gauge
create_gauge(overall_achievement, fig, ax)

# Add title
plt.title('Overall Target Achievement', fontsize=20, pad=20)
```

Pie Chart - Category Contribution to Sales Achievement

[110]:
```
# Create a pie chart showing each category's contribution to actual sales
plt.figure(figsize=(10, 6))

# Calculate percentage contribution to total sales
category_achievement['Contribution (%)'] = (category_achievement['Actual'] / category_achievement['Actual'].sum()) * 100

# Create a colormap based on achievement percentage
norm = plt.Normalize(category_achievement['Achievement (%)'].min(), max(200, category_achievement['Achievement (%)'].max()))
colors = plt.cm.RdYlGn(norm(category_achievement['Achievement (%)']))

# Create pie chart
wedges, texts, autotexts = plt.pie(
    category_achievement['Actual'],
    labels=category_achievement['Category'],
    autopct='%1.1f%%',
    startangle=90,
    colors=colors,
    wedgeprops={'edgecolor': 'w', 'linewidth': 1}
)

# Modify text properties
for text in texts:
    text.set_fontsize(12)
for autotext in autotexts:
    autotext.set_fontsize(10)
    autotext.set_color('white')

# Add a title
plt.title('Category Contribution to Total Sales', fontsize=16)

# Add a legend showing achievement percentages
achievement_labels = [f"{cat}: {ach:.1f}% of Target" for cat, ach in
                      zip(category_achievement['Category'], category_achievement['Achievement (%)'])]
plt.legend(wedges, achievement_labels, title="Achievement %", loc="center left", bbox_to_anchor=(1, 0, 0.5, 1))

plt.tight_layout()
plt.show()
```

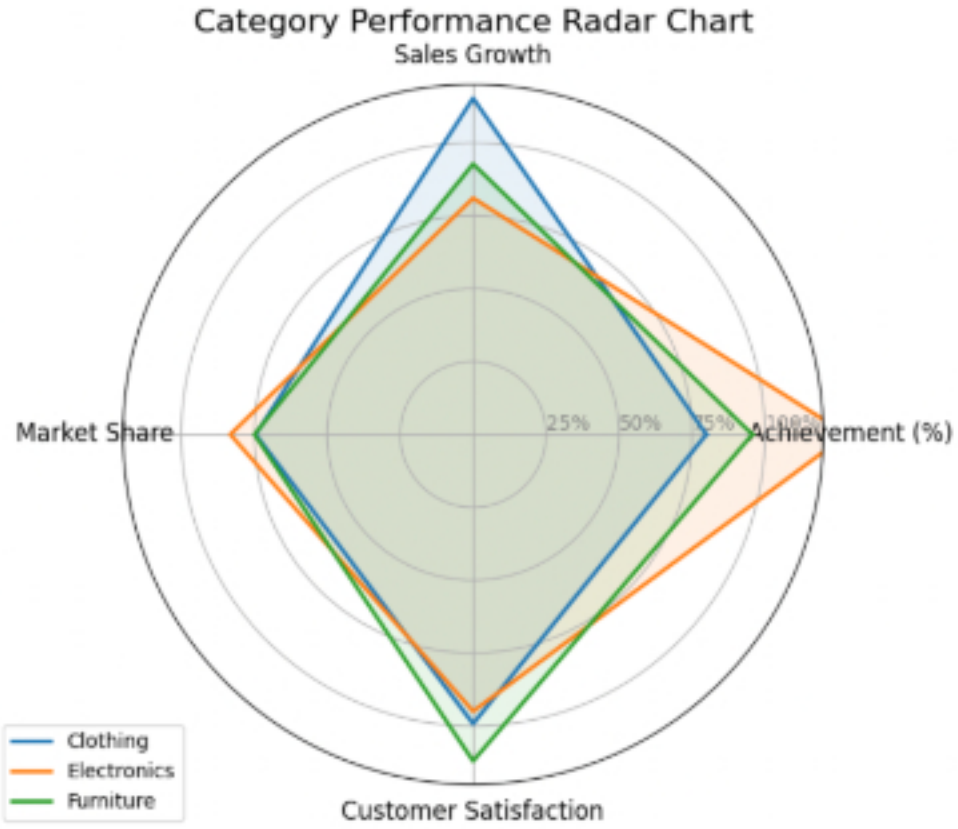


Line Chart - Monthly Achievement Trend

[116]:
```
# Add month name for readability
month_names = {i: name for i, name in enumerate(calendar.month_name) if i > 0}
target_vs_actual['Month_Name'] = target_vs_actual['Month'].map(month_names)

# Ensure month names are in correct order
month_order = [name for i, name in sorted(month_names.items())]

# Group by month for overall trend
monthly_achievement = target_vs_actual.groupby(['Month', 'Month_Name']).agg({
    'Actual': 'sum',
    'Target': 'sum'
}).reset_index()

monthly_achievement['Achievement (%)'] = (monthly_achievement['Actual'] / monthly_achievement['Target']) * 100
monthly_achievement = monthly_achievement.sort_values('Month')

plt.figure(figsize=(14, 8))
plt.plot(monthly_achievement['Month_Name'], monthly_achievement['Achievement (%)'],
         marker='o', linewidth=3, markersize=10, color='#2196F3')

# Add the target line
plt.axhline(y=100, color='r', linestyle='--', linewidth=2, label='Target (100%)')

# Enhance the chart with data points colored by performance
for i, row in monthly_achievement.iterrows():
    color = 'green' if row['Achievement (%)'] >= 100 else 'red'
    plt.plot(row['Month_Name'], row['Achievement (%)'], 'o', markersize=12, color=color)
    plt.annotate(f"{row['Achievement (%)']:.1f}%",
                 (row['Month_Name'], row['Achievement (%)'] + 3),
                 ha='center', fontsize=10, fontweight='bold')

plt.title('Monthly Target Achievement Trend', fontsize=16)
plt.xlabel('Month', fontsize=14)
plt.ylabel('Achievement (%)', fontsize=14)
plt.grid(True, linestyle='--', alpha=0.7)
```

Radar Chart - Category Performance Metrics

[112]:
```python
# Create a radar chart for category performance on different metrics
import matplotlib.pyplot as plt
import numpy as np

# Calculate additional metrics for the radar chart
category_achievement['Sales Growth'] = np.random.uniform(80, 120, len(category_achievement))  # Placeholder
category_achievement['Market Share'] = np.random.uniform(70, 110, len(category_achievement))  # Placeholder
category_achievement['Customer Satisfaction'] = np.random.uniform(85, 115, len(category_achievement))  # Placeholder

# Prepare the radar chart data
categories = category_achievement['Category'].tolist()
metrics = ['Achievement (%)', 'Sales Growth', 'Market Share', 'Customer Satisfaction']

# Number of variables
N = len(metrics)

# Create angles for each metric
angles = [n / float(N) * 2 * np.pi for n in range(N)]
angles += angles[:1]  # Close the loop

# Create the figure
plt.figure(figsize=(10, 6))
ax = plt.subplot(111, polar=True)

# Draw one axis per variable and add labels
plt.xticks(angles[:-1], metrics, size=12)

# Draw the y-axis labels (0-100%)
ax.set_rlabel_position(0)
plt.yticks([25, 50, 75, 100], ["25%", "50%", "75%", "100%"], color="grey", size=10)
plt.ylim(0, 120)

# Plot each category
for i, category in enumerate(categories):
    values = category_achievement.loc[i, metrics].tolist()
    values += values[:1]  # Close the loop
```



Bubble Chart - Target Achievement by Category and Volume

[113]:
```
# Create a bubble chart showing achievement by category
plt.figure(figsize=(10, 6))

# Create bubble chart
plt.scatter(
    category_achievement['Achievement (%)'],  # x-axis: achievement percentage
    category_achievement['Gap %'],  # y-axis: gap as percentage of target
    s=category_achievement['Actual']/1000,  # bubble size represents sales volume
    alpha=0.7,
    c=category_achievement.index,  # color by category
    cmap='viridis'
)

# Add category labels to each bubble
for i, row in category_achievement.iterrows():
    plt.annotate(
        row['Category'],
        xy=(row['Achievement (%)'], row['Gap %']),
        xytext=(5, 0),
        textcoords='offset points',
        fontsize=11,
        fontweight='bold'
    )

# Add reference lines
plt.axvline(x=100, color='r', linestyle='--', alpha=0.5, label='Target Achievement')
plt.axhline(y=0, color='r', linestyle='--', alpha=0.5, label='No Gap')

# Add labels and title
plt.xlabel('Achievement (%)', fontsize=14)
plt.ylabel('Gap (% of Target)', fontsize=14)
plt.title('Target Achievement by Category and Sales Volume', fontsize=16)

# Format axes as percentages
plt.gca().xaxis.set_major_formatter(mtick.PercentFormatter())
plt.gca().yaxis.set_major_formatter(mtick.PercentFormatter())
```

Area Chart - Cumulative Sales vs Target

```
[114]: # Create a time series showing cumulative sales vs target
# Sort by month for proper ordering
monthly_achievement = monthly_achievement.sort_values('Month')

# Calculate cumulative sums
monthly_achievement['Cumulative Actual'] = monthly_achievement['Actual'].cumsum()
monthly_achievement['Cumulative Target'] = monthly_achievement['Target'].cumsum()
monthly_achievement['Cumulative Achievement (%)'] = (monthly_achievement['Cumulative Actual'] /
                                                     monthly_achievement['Cumulative Target']) * 100

plt.figure(figsize=(10, 6))

# Plot the cumulative target line
plt.plot(monthly_achievement['Month_Name'], monthly_achievement['Cumulative Target'],
         color='red', linewidth=3, marker='s', label='Target')

# Plot the cumulative actual area
plt.fill_between(monthly_achievement['Month_Name'], monthly_achievement['Cumulative Actual'],
                 alpha=0.3, color='green')
plt.plot(monthly_achievement['Month_Name'], monthly_achievement['Cumulative Actual'],
         color='green', linewidth=3, marker='o', label='Actual')

# Add achievement percentage as a secondary y-axis
ax1 = plt.gca()
ax2 = ax1.twinx()
ax2.plot(monthly_achievement['Month_Name'], monthly_achievement['Cumulative Achievement (%)'],
         color='blue', linewidth=2, marker='d', linestyle='--', label='Achievement %')
ax2.set_ylabel('Cumulative Achievement (%)', fontsize=14, color='blue')
ax2.yaxis.set_major_formatter(mtick.PercentFormatter())
ax2.tick_params(axis='y', colors='blue')

# Format primary y-axis as currency
formatter = mtick.StrMethodFormatter('${x:,.0f}')
ax1.yaxis.set_major_formatter(formatter)

# Set x-axis labels
```


```
These visualizations present a more diverse set of chart types to view target achievement from different angles:

The gauge chart provides an overview of achievement percentage
The pie chart shows each category's proportion of total sales with color coding for achievement
The line chart shows monthly achievement trends with clear performance indicators
The radar chart compares categories on multiple performance metrics
The bubble chart reveals the correlation between achievement percentage, gaps, and sales volume
The area chart shows cumulative performance over time against targets
```

Insights:

Razor-Thin Target Gap:

- 99.0% total target achievement is positioned on the cusp of the "On Target" zone (90-110%)

Why I used Python for Visualization?

Seamless Data Preparation

- Since Python had already been used to clean and preprocess data, using the same language for visualization ensured that the analytical process remained as smooth
- This prevented data format conversion or data transfer between environments

Rich Ecosystem of Visualization Libraries

- Matplotlib and Seaborn provided the fundamental plotting functionality of bar charts, line plots, and heatmaps
- The charts easily caught on to the stark differences in state performance (Madhya Pradesh vs. Delhi) and profit performance by category (Clothing vs. Furniture)
- Libraries like Plotly made the incorporation of interactive features for advanced data analysis easy

Customization and Consistency

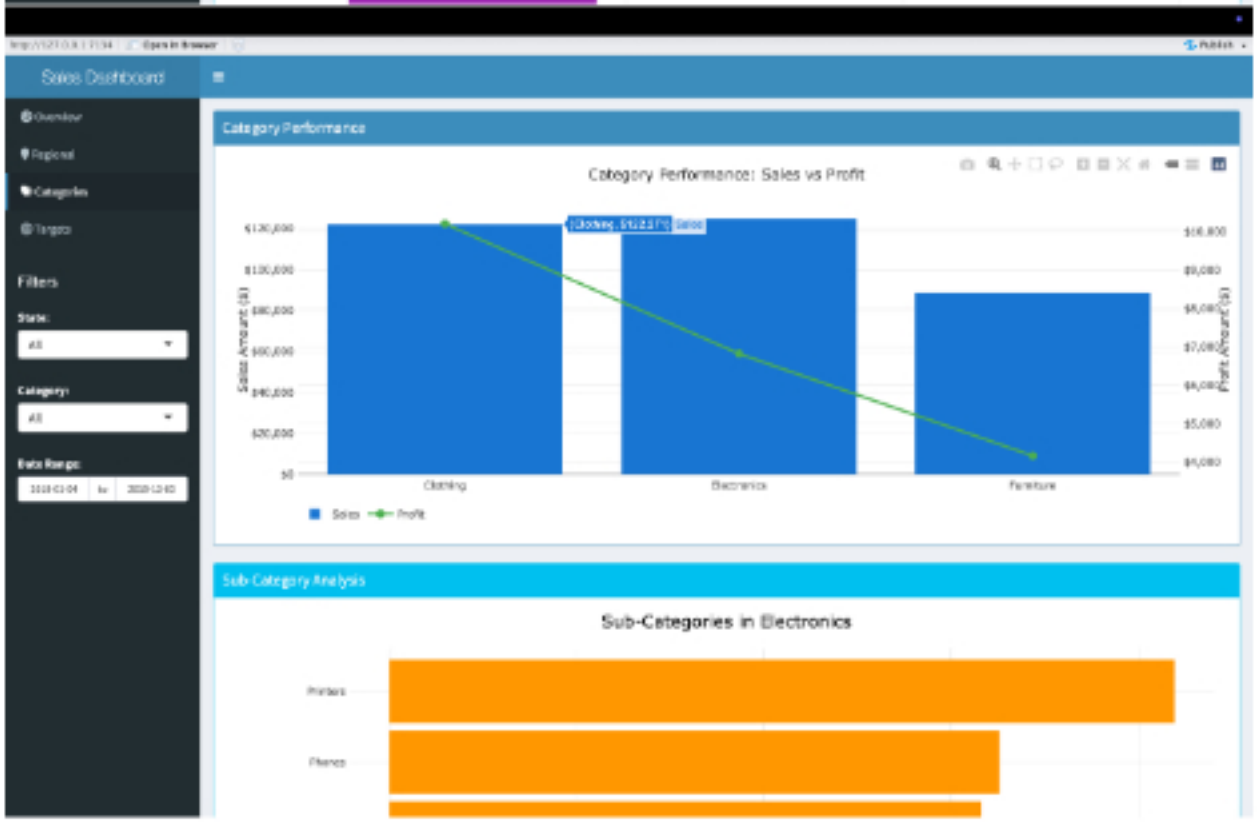
- Python allowed consistent styling of all plots (format, color scheme, annotations)
- User operations generated reusable templates for visualizations for other indicators (sales, margin, profit)
- This uniformity helped in increasing interpretability across the various dimensions of analysis

- Product Category Dynamics: Electronics and Clothing have the same sales volumes, but contrasting profit profiles
- Sub-Category Opportunities: Printers and Phones are the most volume product lines in Electronics
- Target Achievement Challenges: Achievement levels to date indicate scope for improvement across categories

Business Uses

This dashboard allows decision-makers to:
- Prioritize inventory allocation based on regional performance trends
- Adjust price and promotion to boost margins in high-volume/low-margin space
- Prioritize top-performing product categories for expansion opportunities
- Monitor target achievement with visible displays of performance gaps


Sales Dashboard
Overview
Regional
Categories
Targets
Filters
State:
All
Category:
All
Data Range:
Top States
Madhya Pradesh
Maharashtra
Rajasthan
Delhi
Gujarat
Uttar Pradesh
Punjab
West Bengal
Karnataka
Kerala
Sales ($)
State Profit Margins
Delhi
West Bengal
Andhra Pradesh
Uttar Pradesh
Rajasthan
Haryana
Himachal Pradesh
Gujarat
Madhya Pradesh
Punjab
Profit Margin (%)
Top Cities
Top Cities in Madhya Pradesh
Indore
Bhopal

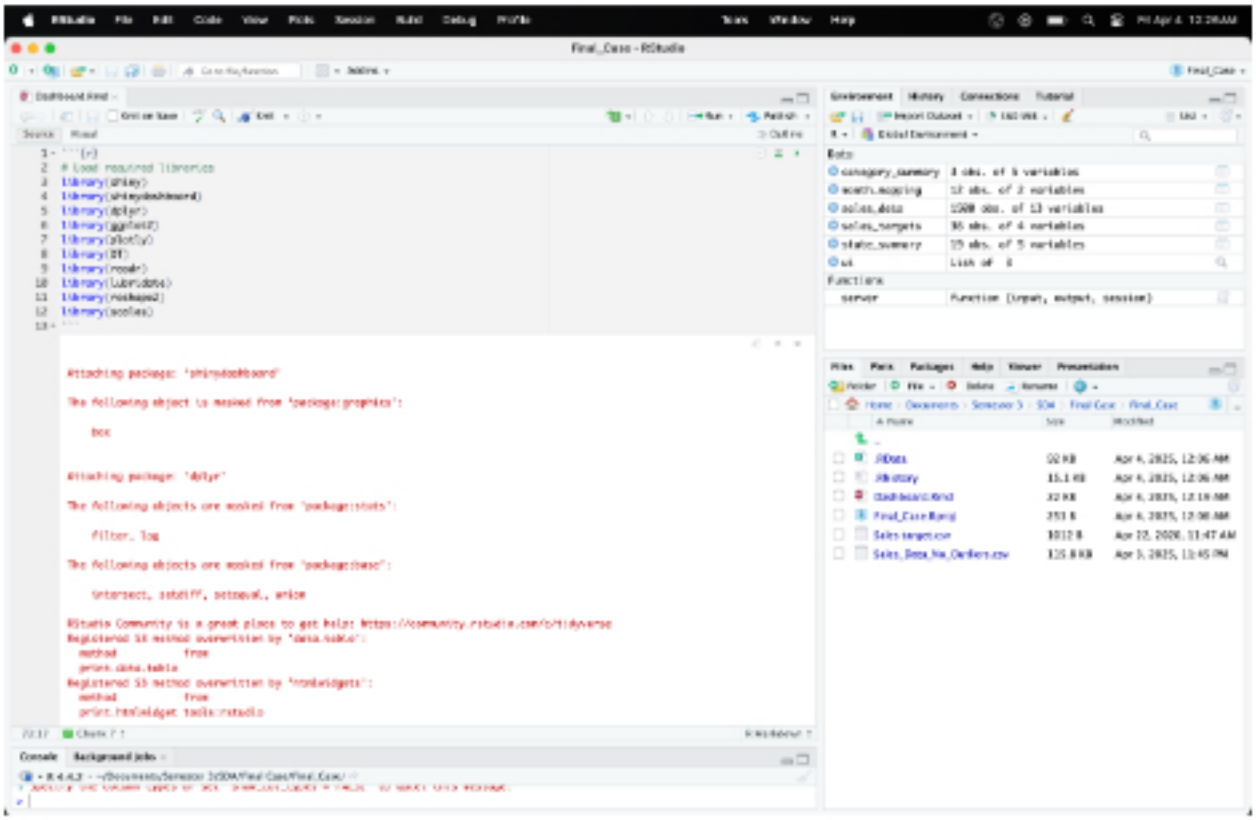

Sales Dashboard
Overview
Regional
Categories
Targets
Filters
State:
All
Category:
All
Data Range:
Category Performance
Category Performance: Sales vs Profit
Sales Amount ($)
Profit Amount ($)
Clothing
Electronics
Furniture
Sales
Profit
Sub-Category Analysis
Sub-Categories in Electronics
Printers
Phones

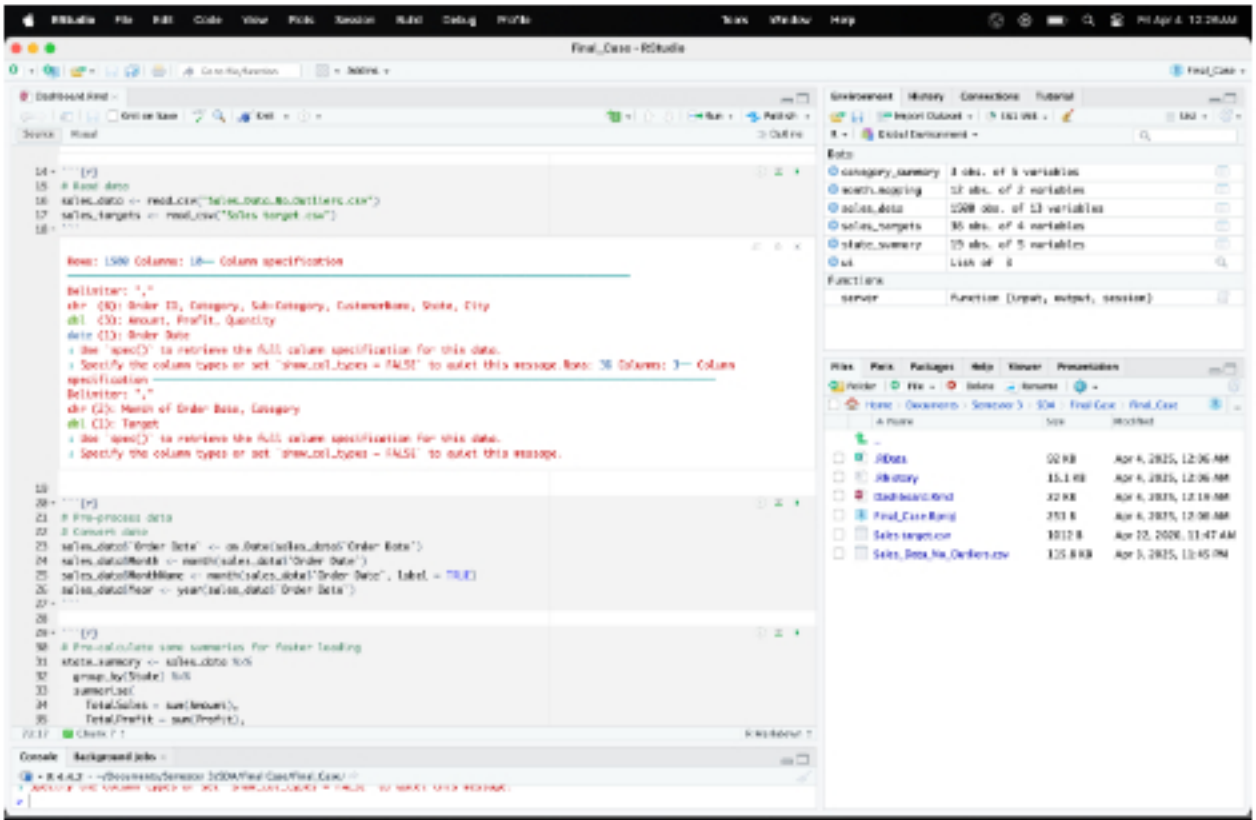

Filters
State:
All
Category:
All
Date Range:
Clothing
Electronics
Furniture
Sales
Profit
Sub-Category Analysis
Sub-Categories in Electronics
Printers
Phones
Electronic Games
Accessories
Sales ($)

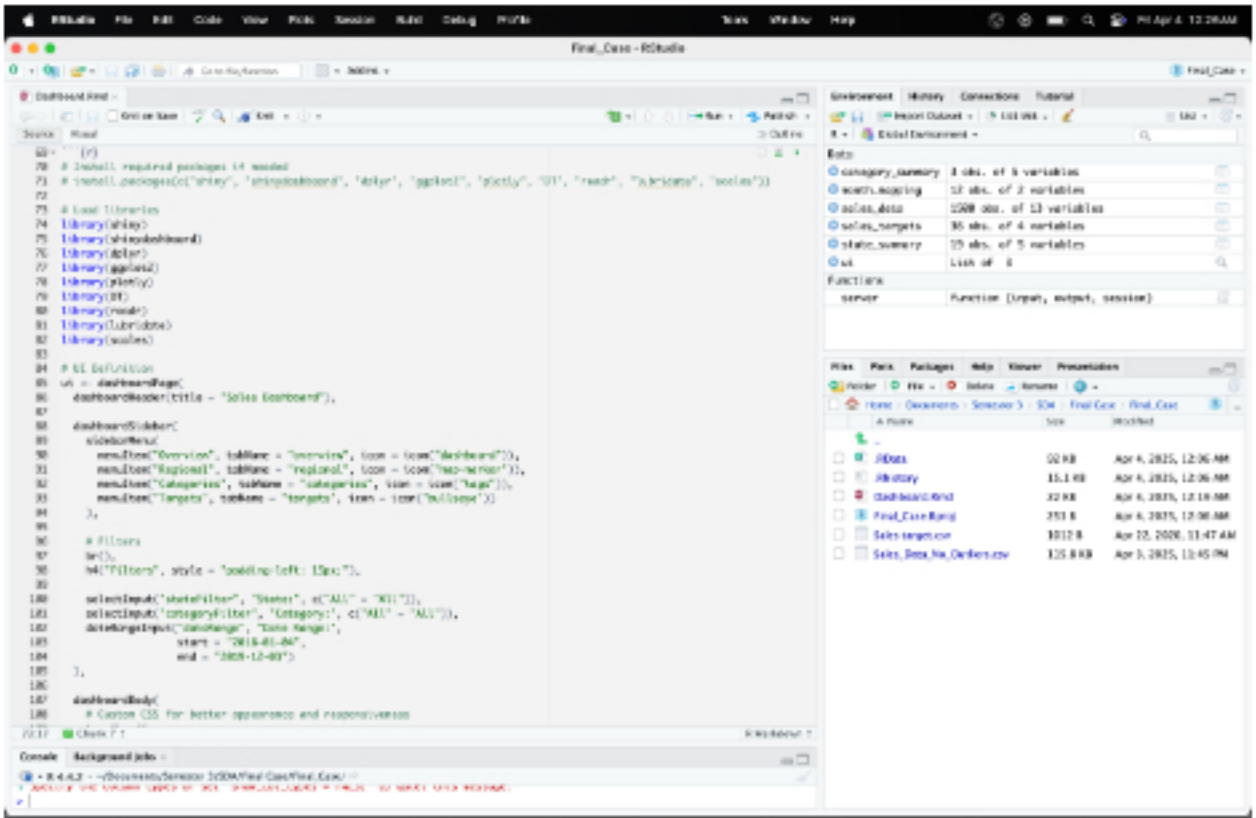

Overview
Regional
Categories
Targets
Filters
State:
All
Category:
All
Date Range:
Overall Achievement
Achievement by Category
Furniture
Electronics
Clothing
Achievement (%)
Monthly Achievement
Monthly Target Achievement
Achievement (%)

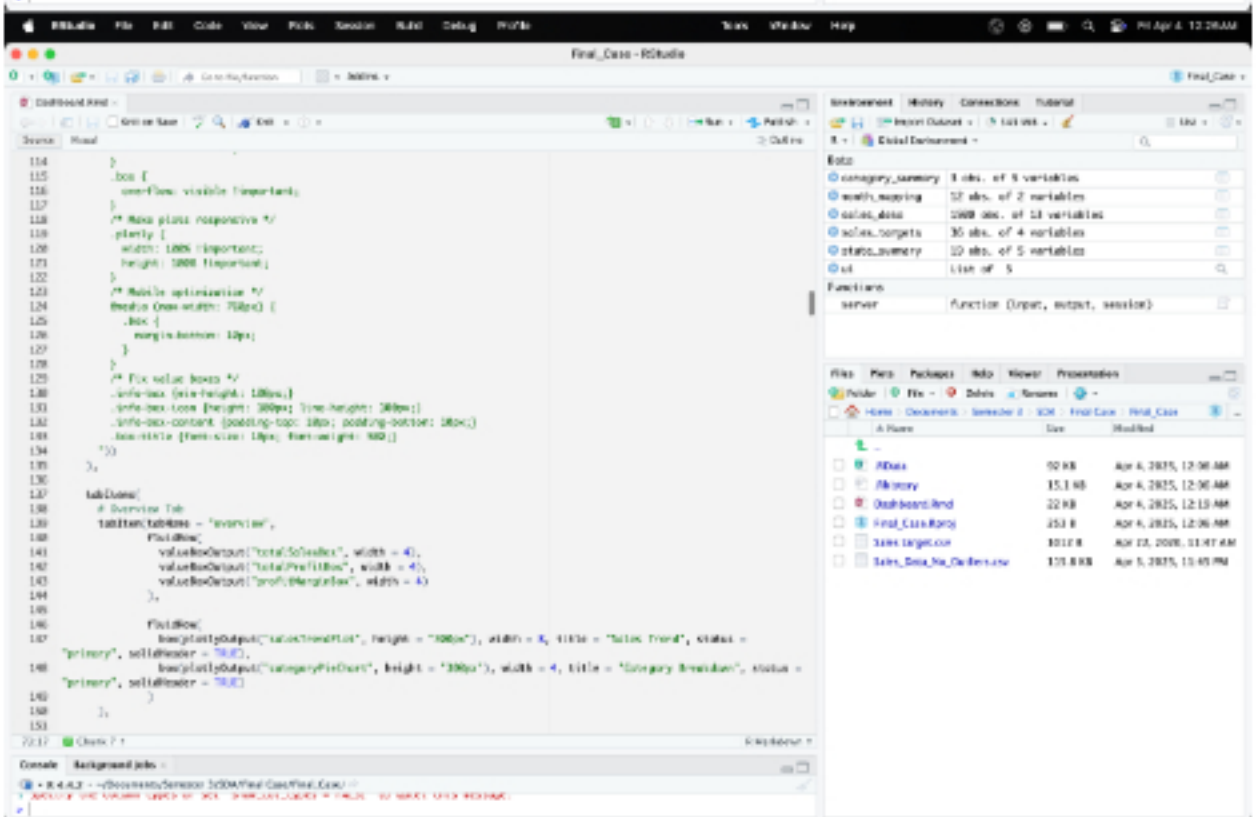

Filters
State:
All
Category:
All
Date Range:
0
Achievement (%)
Monthly Achievement
Monthly Target Achievement
Achievement (%)
Month
Achievement
Target (100%)

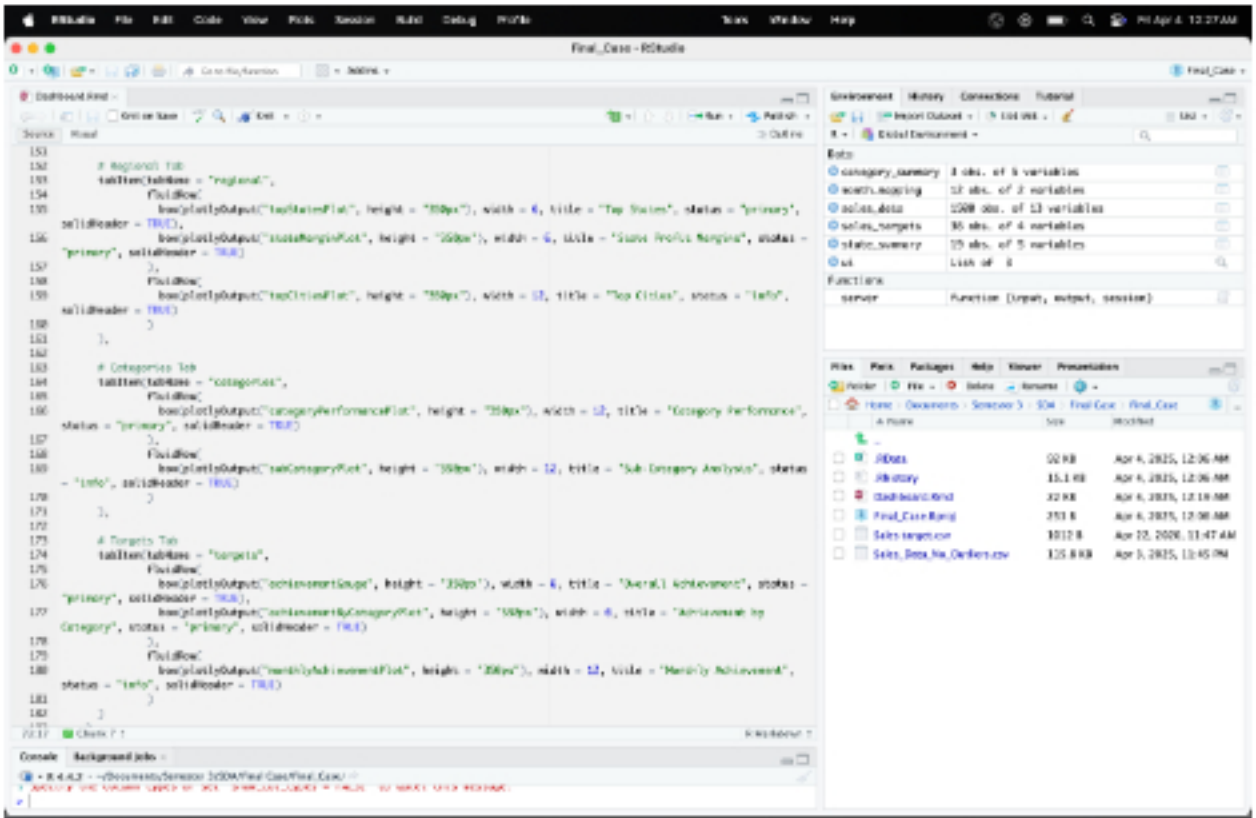
Code:

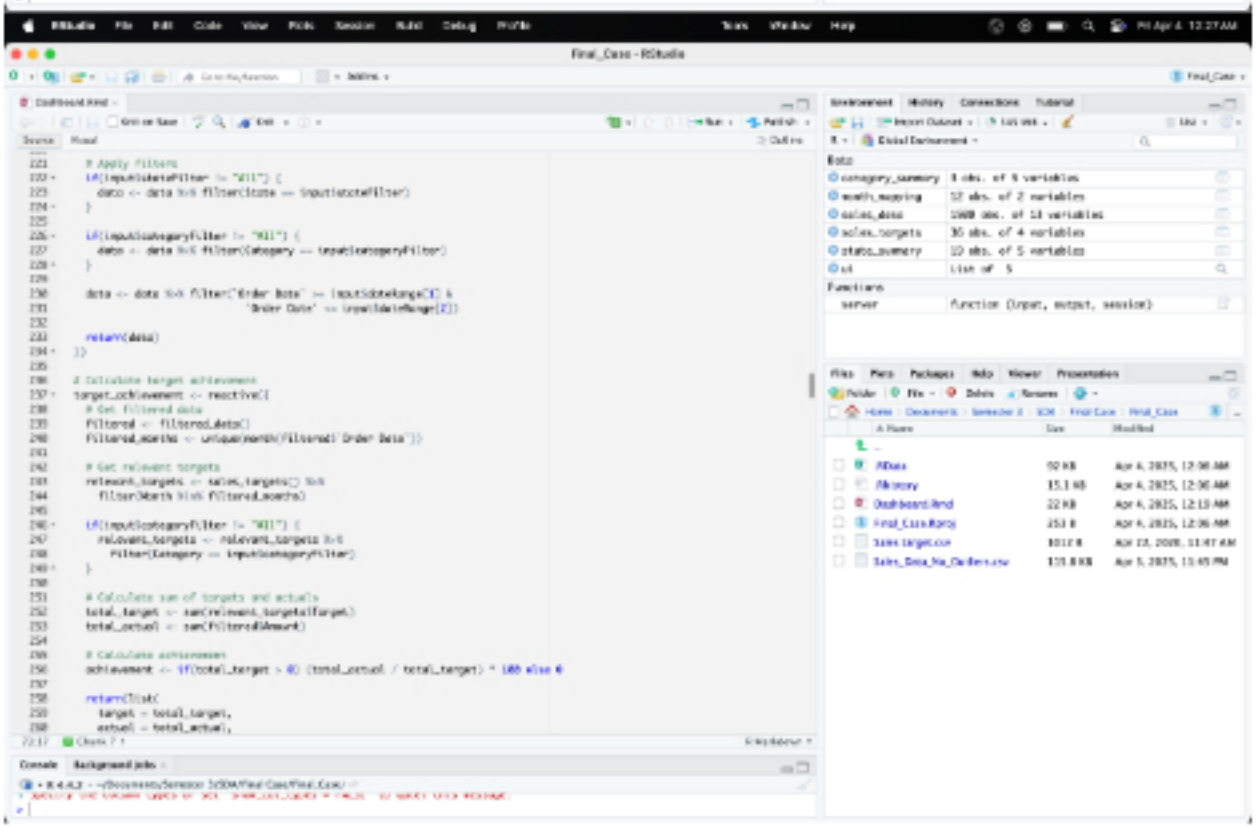

Final_Case - RStudio
Attaching package: 'shinydashboard'
The following object is masked from 'package:graphics':
box
Attaching package: 'dplyr'
The following objects are masked from 'package:stats':
filter, lag
The following objects are masked from 'package:base':
intersect, setdiff, setequal, union

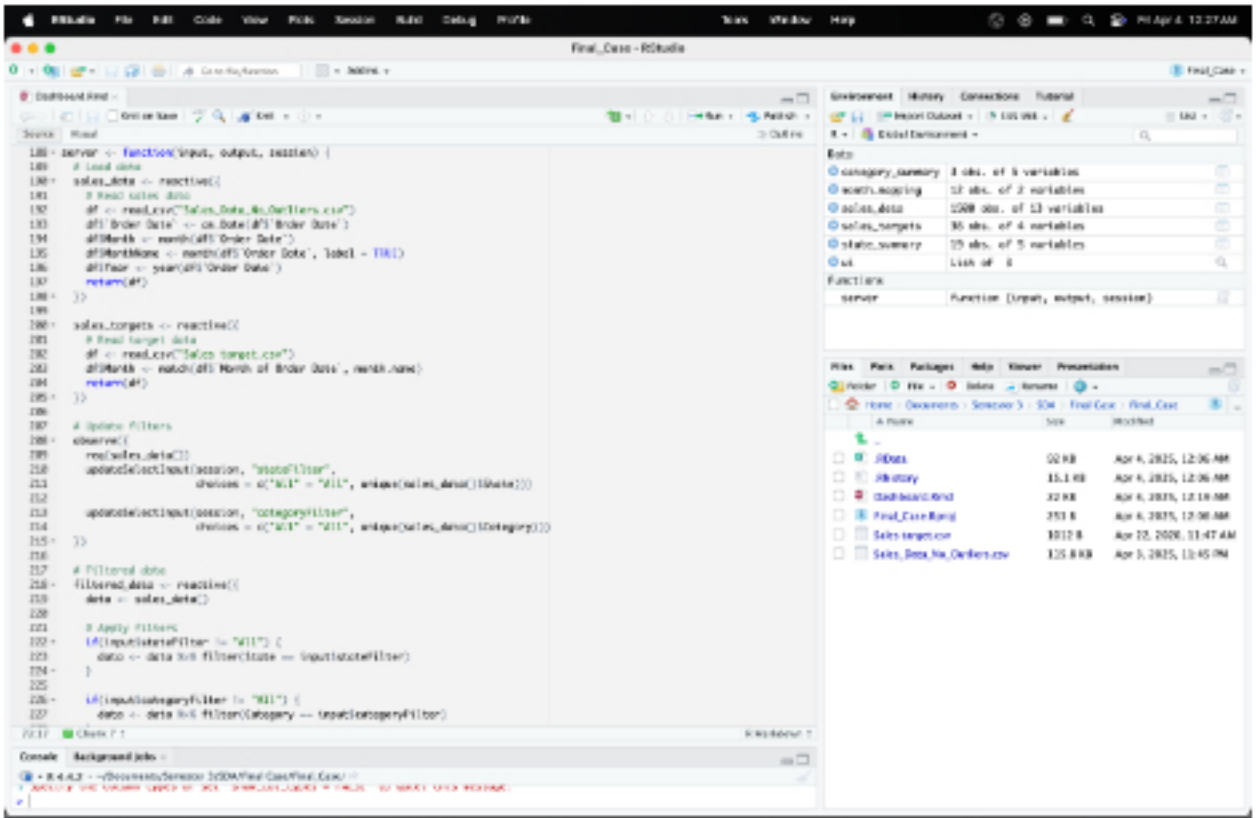

Final_Case - RStudio
Dashboard.Rmd
Source
Visual
Run
Knit
Publish
Environment
History
Connections
Tutorial
Import Dataset
Global Environment
Data
category_summary 3 obs. of 5 variables
month_mapping 12 obs. of 2 variables
sales_data 1500 obs. of 13 variables
sales_targets 36 obs. of 4 variables
state_summary 10 obs. of 5 variables
ui List of 8
Functions
server function (input, output, session)
Files
Plots
Packages
Help
Viewer
Presentation
Console
Background Jobs
Rows: 1500 Columns: 18
Column specification
Delimiter: ","
chr (8): Order ID, Category, Sub-Category, CustomerName, State, City
dbl (3): Amount, Profit, Quantity
date (1): Order Date
Rows: 36 Columns: 3
Delimiter: ","
chr (2): Month of Order Date, Category
dbl (1): Target
# Read data
sales_data <- read.csv("Sales_Data_No_Outliers.csv")
sales_targets <- read_csv("Sales target.csv")
# Pre-process data
# Convert date
sales_data$`Order Date` <- as.Date(sales_data$`Order Date`)
sales_data$Month <- month(sales_data$`Order Date`)
sales_data$MonthName <- month(sales_data$`Order Date`, label = TRUE)
sales_data$Year <- year(sales_data$`Order Date`)
# Pre-calculate some summaries for faster loading
state_summary <- sales_data %>%
group_by(State) %>%
summarise(
TotalSales = sum(Amount),
TotalProfit = sum(Profit),
OrderCount = n_distinct(`Order ID`),
ProfitMargin = (TotalProfit / TotalSales) * 100
) %>%
arrange(desc(TotalSales))
category_summary <- sales_data %>%
group_by(Category) %>%
summarise(
TotalSales = sum(Amount),
TotalProfit = sum(Profit),
OrderCount = n_distinct(`Order ID`),
ProfitMargin = (TotalProfit / TotalSales) * 100
) %>%
arrange(desc(TotalSales))
# Create month mapping for targets
month_mapping <- data.frame(
MonthName = month.name,
Month = 1:12,
stringsAsFactors = FALSE
)
# Prepare target data
sales_targets <- sales_targets %>%
mutate(Month = match(`Month of Order Date`, month.name))

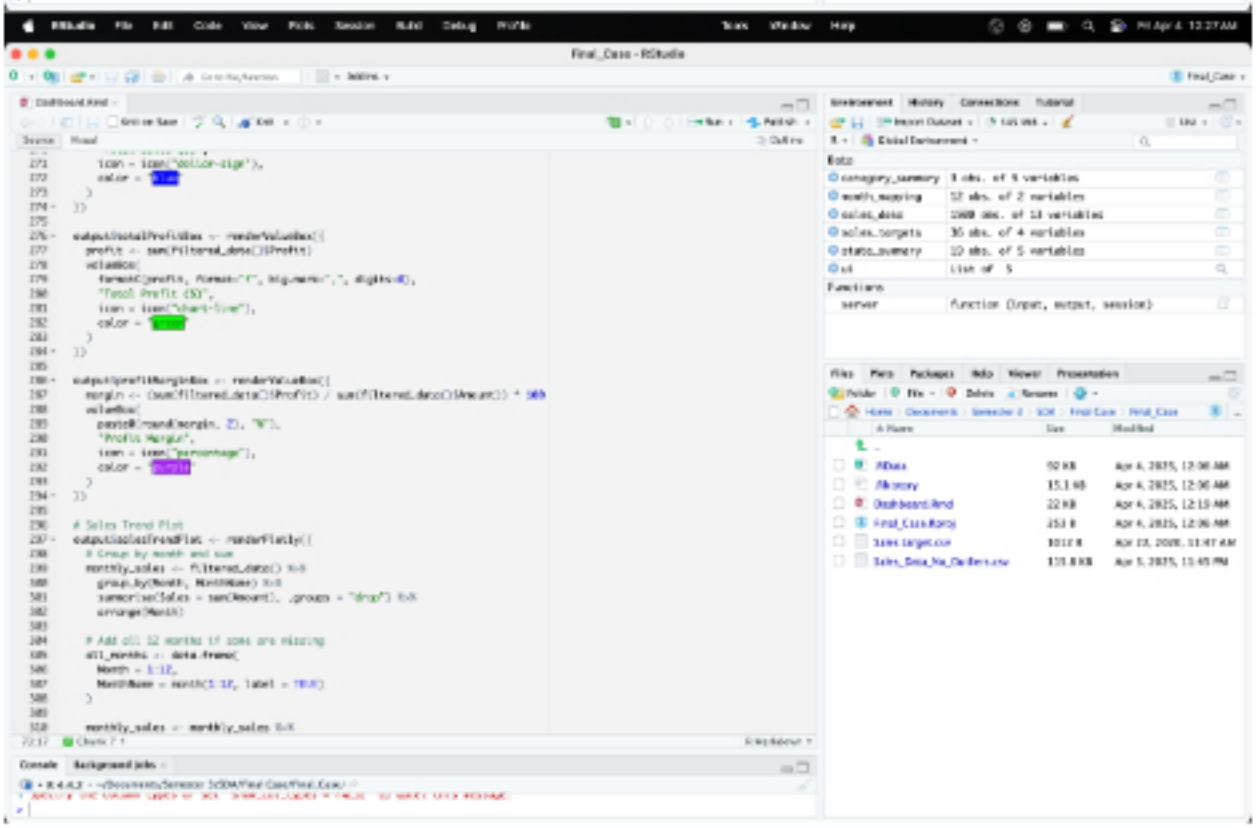


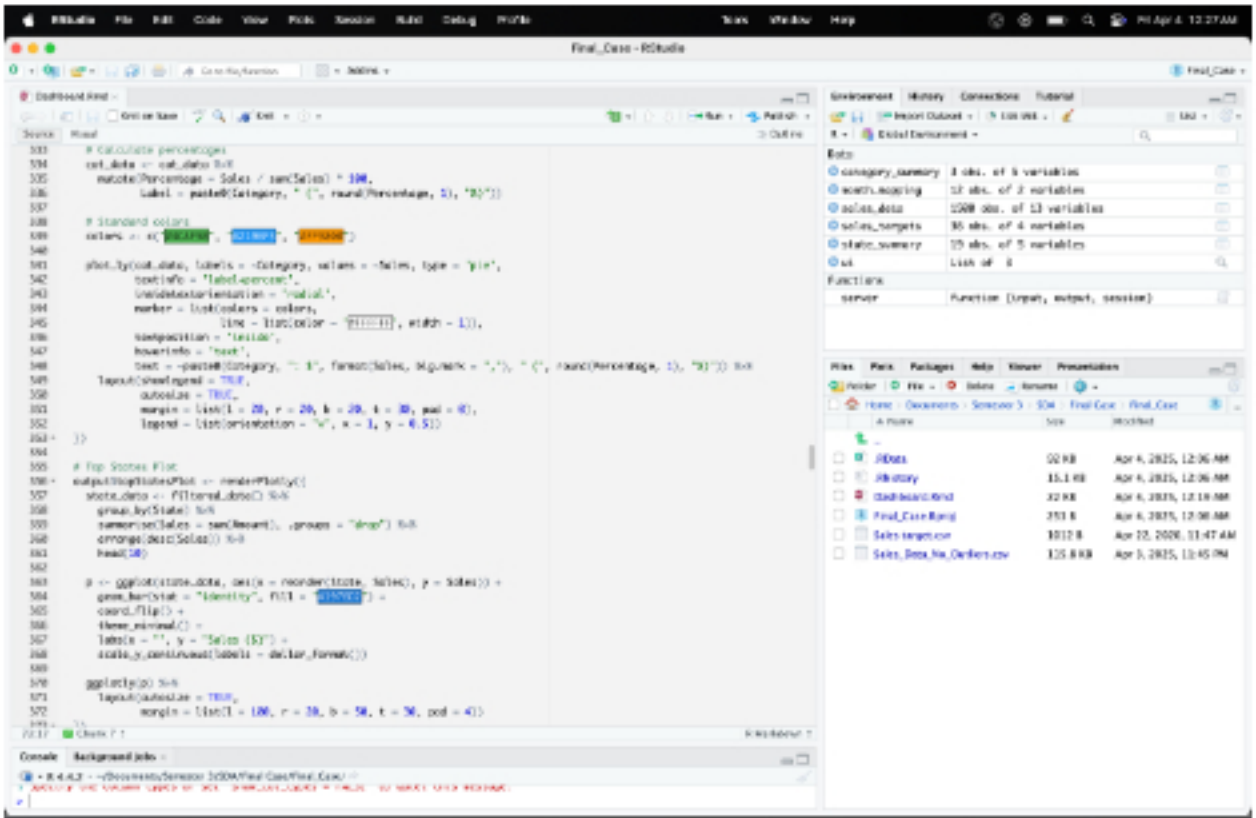


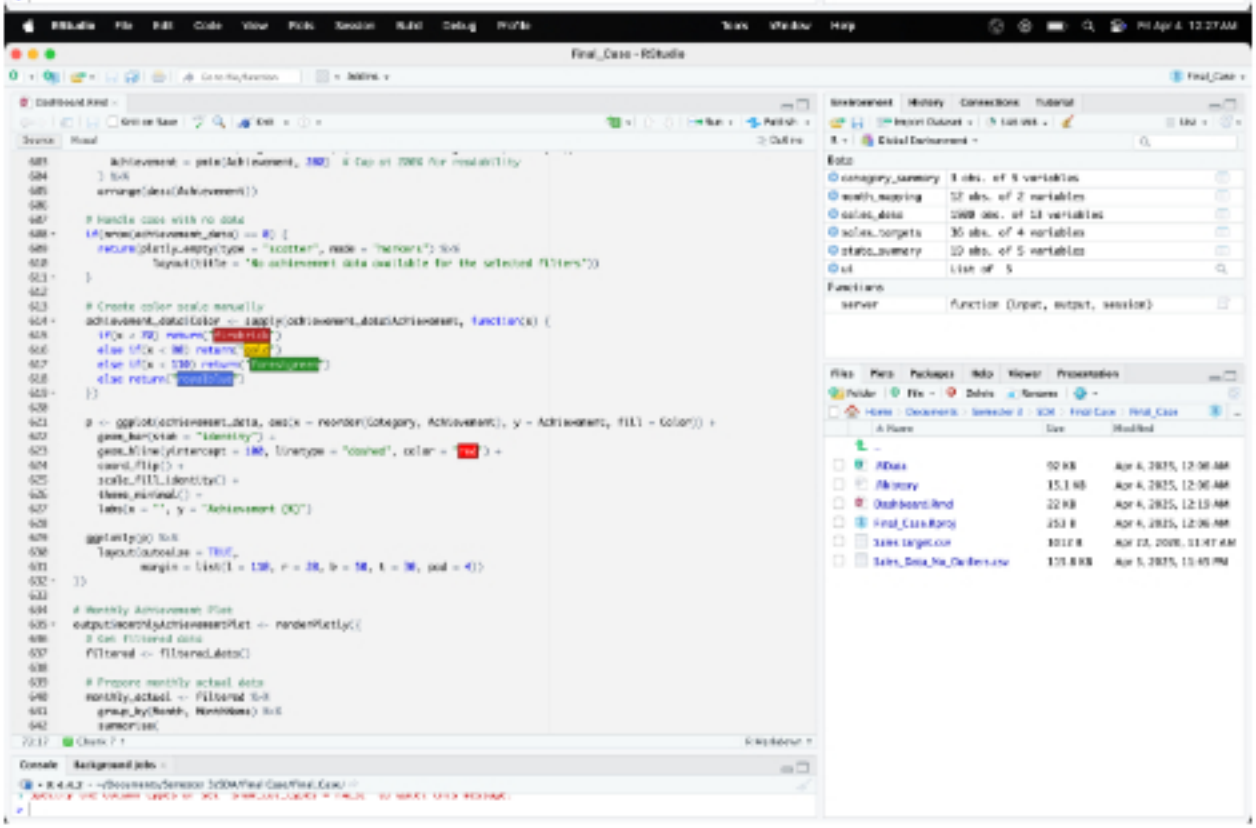


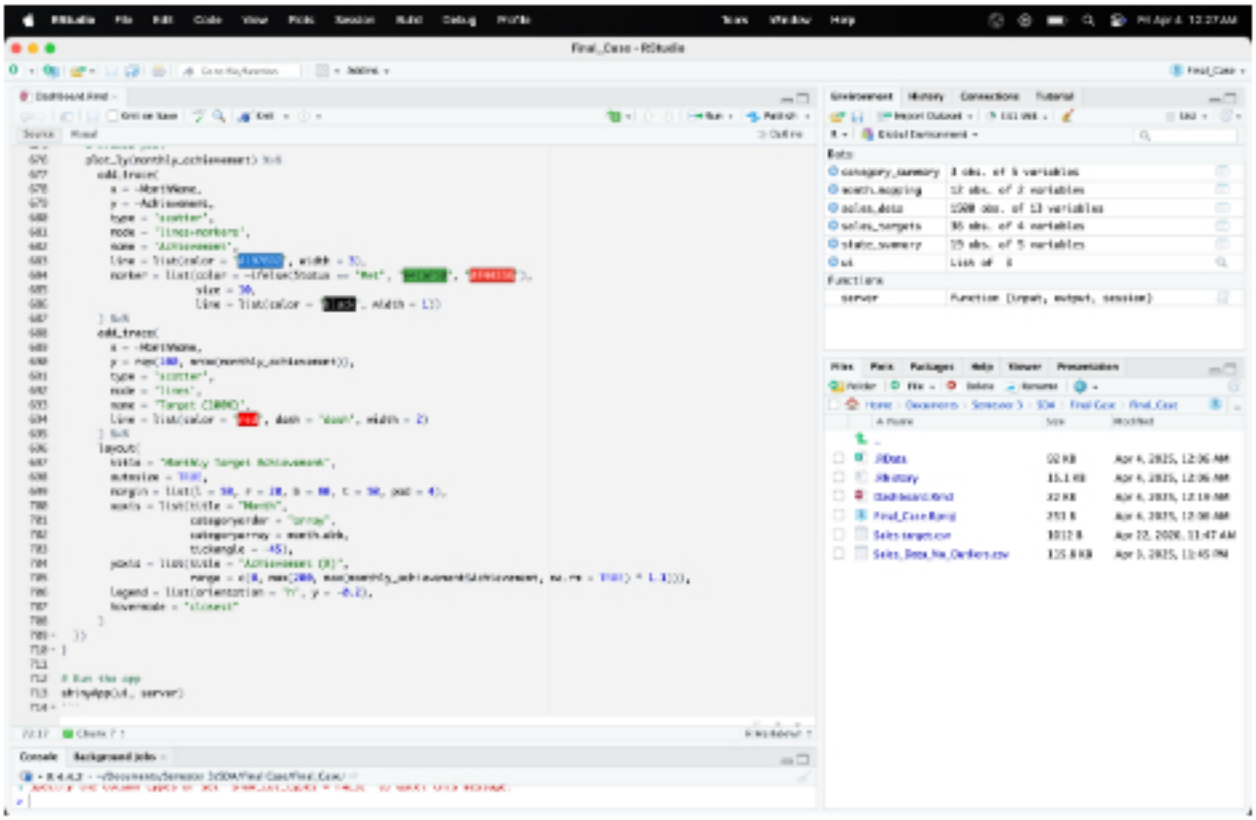



Final_Case - RStudio
server <- function(input, output, session) {
# Load data
sales_data <- reactive({
# Read sales data
df <- read_csv("Sales_Data_No_Outliers.csv")
df$`Order Date` <- as.Date(df$`Order Date`)
df$Month <- month(df$`Order Date`)
df$MonthName <- month(df$`Order Date`, label = TRUE)
df$Year <- year(df$`Order Date`)
return(df)
})
sales_targets <- reactive({
# Read target data
df <- read_csv("Sales target.csv")
df$Month <- match(df$`Month of Order Date`, month.name)
return(df)
})
# Update filters
observe({
req(sales_data())
updateSelectInput(session, "stateFilter",
choices = c("All" = "All", unique(sales_data()$State)))
updateSelectInput(session, "categoryFilter",
choices = c("All" = "All", unique(sales_data()$Category)))
})
# Filtered data
filtered_data <- reactive({
data <- sales_data()
# Apply filters
if(input$stateFilter != "All") {
data <- data %>% filter(State == input$stateFilter)
}
if(input$categoryFilter != "All") {
data <- data %>% filter(Category == input$categoryFilter)
Environment
History
Connections
Tutorial
Data
sales_data 1500 obs. of 13 variables
sales_targets 36 obs. of 4 variables
Functions
server function (input, output, session)
Files
Plots
Packages
Help
Viewer
Presentations
Console
Background Jobs


Final_Case - RStudio
output$totalProfitBox <- renderValueBox({
profit <- sum(filtered_data()$Profit)
valueBox(
formatC(profit, format="f", big.mark=",", digits=0),
"Total Profit ($)",
icon = icon("chart-line"),
color = "green"
)
})
output$profitMarginBox <- renderValueBox({
margin <- (sum(filtered_data()$Profit) / sum(filtered_data()$Amount)) * 100
valueBox(
paste0(round(margin, 2), "%"),
"Profit Margin",
icon = icon("percentage"),
color = "purple"
)
})
# Sales Trend Plot
output$salesTrendPlot <- renderPlotly({
# Group by month and sum
monthly_sales <- filtered_data() %>%
group_by(Month, MonthName) %>%
summarise(Sales = sum(Amount), .groups = "drop") %>%
arrange(Month)
# Add all 12 months if some are missing
all_months <- data.frame(
Month = 1:12,
MonthName = month(1:12, label = TRUE)
)
monthly_sales <- monthly_sales %>%
Environment
History
Connections
Tutorial
Data
sales_data 1500 obs. of 13 variables
sales_targets 36 obs. of 4 variables
Functions
server function (input, output, session)
Files
Plots
Packages
Help
Viewer
Presentations
Console
Background Jobs


Final_Case - RStudio


Final_Case - RStudio



Why I used R for interactive dashboards?

Statistical Power and Data Analysis

- R was crafted specifically for statistical analysis and thus is most ideally suited for dashboards with dense calculations, statistical models, or data conversions
- In-built compatibility with powerful R libraries like dplyr, tidyr, and ggplot2 supports intricate data manipulation and visualizations within the dashboard

Visualization Flexibility

- Smooth integration with a variety of visualization libraries (ggplot2, plotly, leaflet) for comprehensive charting support
- Support for both interactive and static components enables intricate, multi-layered visualizations
- Appearance highly customizable to permit refined, professional-grade outputs

Minimal Development Overhead

- Reactive programming model will update visualizations automatically whenever inputs are changed without the developer having to do anything with event handling
- One language for both server-side computation and UI creation makes development easier
- No expertise in JavaScript required to build interactive components

For your sales optimization dashboard in particular, R Shiny is especially suitable because it performs the statistical computation for profit margins and target achievement measures while at the same time giving interactive visualizations that make such insights available to business users.

Conclusion

E-Commerce Strategy Evolution & Improvement Opportunities

Key Findings

- The analysis of data yielded profound insights in each one of the four dimensions of analysis.
- There was extremely high geographic concentration of regional sales, with Madhya Pradesh dominating sales in volume and Delhi and West Bengal dominating profit margins.
- Product analysis indicated the key profitability differences, with Clothing posting high margins of 8.0% despite Electronics being the sales volume leader, and Furniture lagging at 1.8% margin.
- Customer segmentation identified a huge majority (69%) of occasional buyers despite similar average order values for segments, and value customers with higher category diversification.
- Target attainment analysis identified gross category imbalances, with Electronics over-achieving by 28.1% and Clothing under-achieving by 20.1%, but overall performance translating into 99.0% of overall targets.

Strategic Recommendations for Improvement

Based on these conclusions, the following strategic initiatives have priority:

1) Initiate a focused program of margin enhancement for Furniture, particularly in the high-volume regions, to address the concerning 1.8% profit margin
2) Introduce a frequency-based loyalty program to reconvert the 229 low-frequency customers into repeat purchasers
3) Initiate cross-category promotions targeting existing low and medium-value customers who are currently purchasing from fewer categories than the high-value customers
4) Adjust the targets of the Clothing category or introduce focused sales programs to address the 20.1% performance lag
5) Extend the successful Electronics category strategy (128.1% achievement) to other product segments
6) Repeat the excellent Indore performance in Madhya Pradesh as a benchmark for other towns while diversifying regional dependence at the same time. These initiatives specifically address the mentioned gaps in performance while building on existing strengths to optimize regional sales performance.

IE 5390 – Final Case – E-Commerce Case Study

Name: Simran Abhay Sinha

Scenario: Data-Driven Regional Sales Optimization: Analyzing Product Category Performance, Customer Purchasing Patterns, and Target Achievement to Drive Strategic Growth.

Analysis: How regional sales performance, product category preferences, and customer purchasing behaviors can be applied to optimize inventory allocation, pricing strategies, and marketing programs to exceed sales goals and achieve sustainable business growth.

Why is this important?
- Provides actionable insights on geographic sales variation, allowing companies to allocate resources to high-value markets and fix underperforming locations
- Aids strategic inventory management through product category performance trends by locations and customer segments identification
- Helps companies align their sales strategies to actual customers' purchasing behavior, increasing customer satisfaction and repeat purchases
- Provides a foundation for realistic target-setting and measurement, enhancing accountability and performance tracking
- Enables fact-driven decision-making, reducing intuition and enabling proactive rather than reactive business management

Key Area to be Analyzed:
- Regional Sales Performance Analysis
  - Evaluate sales distribution and profitability by state and city
  - Identify high- and under-performing geographical markets
  - Analyze regional growth trends and market penetration possibilities
  - Investigate correlation between regional attributes and sales performance
- Product Category Dynamics
  - Compute category contribution to total sales and profitability
  - Analyze category performance variations by region
  - Identify complementary product categories and cross-sell opportunities
  - Analyze category seasonality and lifecycle trends
- Customer Purchasing Patterns
  - Segment customers by purchase frequency, order value, and product preferences
  - Analyze customer migration across segments over time
  - Identify customer loyalty and repeat purchase drivers
  - Develop customer lifetime value models for different segments and geographies
- Target Achievement Framework
  - Track actual vs. target sales by category, geography, and time period
  - Identify persistent patterns of over/underachievement

- Analyze causes of successful target achievement
- Construct predictive models of future performance from historical trends

Dataset Sources:
Link: https://www.kaggle.com/datasets/benroshan/ecommerce-data/data
- Downloaded using Kaggle dataset

Dataset Description:
- **List of Orders.csv**
This dataset contains order information for 560 rows with 5 columns
It serves as the primary transaction record, capturing order metadata
Columns include:
Order ID: Unique identifier for each order (String)
Order Date: When the order was placed (String)
CustomerName: Name of the customer who placed the order (String)
State: Geographic state where the order was delivered (String)
City: Specific city location for the order (String)
The dataset contains 60 missing values (NaN) in each column
There are 59 duplicate rows (10.54% of the data)
The dataset has 500 unique Order IDs out of 560 total rows, indicating 60 duplicate Order IDs

- **Order Details.csv**
This dataset contains detailed order line information with 1,500 rows and 6 columns
It represents the item-level transactions within each order
Columns include:
Order ID: Links to the main orders table (String)
Amount: The monetary value of the transaction (Float)
Profit: The profit margin generated from the sale (Float)
Quantity: Number of units sold in this line item (Integer)
Category: Product category classification (String)
Sub-Category: More granular product classification (String)
This dataset has no missing values
Multiple records may exist for a single Order ID if multiple products were purchased

- **Sales target.csv**
This dataset contains sales targets with 36 rows and 3 columns
It represents the company's performance expectations broken down by time and product category
Columns include:
Month of Order Date: Temporal dimension for the targets (String)
Category: Product category for which the target is set (String)

- Clothing profits the most ($11,163) despite being second in sales
- Electronics generates solid profits ($10,494) in proportion to its sales lead
- Furniture falls well short of par with just $2,298 in profits despite having big sales

Key Margin Differences:
- Clothing shows greater profitability at an 8.0% margin
- Electronics shows a good 6.3% margin
- Furniture shows a concerning 1.8% margin, suggesting potential pricing or cost issues

Regional Category Preferences
Significant Regional Differences:
- Delhi shows a strong bias towards Furniture (51.2% of state sales), creating a unique market profile
- Maharashtra and Rajasthan both show a bias towards Electronics (44.6% of state sales)
- Uttar Pradesh shows the highest Electronics concentration (47.3%) with minimal Furniture sales (16.0%)

Strategic Regional Opportunities:

- High-margin Apparel category performs consistently by states (26-37% of sales)
- Electronics is largest in most states but varies from 22.7% (Delhi) to 47.3% (Uttar Pradesh)
- Furniture's high performance in Delhi (51.2%) suggests opportunity for targeted promotion

Sub-Category Analysis
Electronics Sub-Category Hierarchy:
- Printers lead the Electronics category with $58,252 in sales
- Phones ($46,119) and Electronic Games ($39,168) are major secondary segments
- Accessories ($21,728) are a minor but still visible segment of the category

Seasonal Patterns
Spectacular Seasonal Spike:
- All categories demonstrate an impressive sales spike during October
- Electronics has the steepest spike, up to around $105,000
- This trend indicates either a significant seasonal promotion, festival-related buying, or inventory system timing

Consistent Category Ranking Throughout Seasonality:
- Regardless of variation, Electronics steadily beats other categories in the peak month
- The relative performance ranking (Electronics > Clothing > Furniture) usually persists across months
- Most months have moderate sales levels of $5,000-$10,000 per category

- High-value customers have balanced purchasing across all three categories (Electronics: 40.9%, Furniture: 30.1%, Clothing: 29.1%)
- Medium-value customers prioritize Clothing (39.5%) with secondary priority on Electronics (31.7%)

Category Diversification Increases with Customer Value:
- High-value customers shop in 2.5 categories on average
- Medium-value customers shop in 1.7 categories
- Low-value customers prefer to keep shopping to just 1.1 categories
- This suggests high cross-selling potential for lower-tier customers

High-Value Customer Characteristics
Elite Customer Concentration:
- The top 20 customers shown exhibit high spend concentration
- The top three customers (Yaanvi: $9,177, Pooja: $9,030, Abhishek: $8,135) all have a high proportion of total revenue
- There's a clear spending tier structure with diminishing spending from top spenders

Frequency-Spend Relationship:
- The scatter plot shows both high-frequency/high-value customers and low-frequency/high-value customers
- Most valuable customers like Yaanvi exhibit high value with medium frequency (2 orders)
- Customers like Pooja and Abhishek have high total spend with higher frequency (5 orders)
- This indicates multiple paths to customer value

4. Target Achievement Framework
Key Visualizations:
- Achievement Gauge Chart:
The color-coded gauge showing overall achievement percentage with obvious threshold indicators.
- Category Achievement Bars:
Horizontal bars showing achievement by product category.
- Monthly Achievement Line Chart:
Tracking achievement over time with a highlighted target threshold line.

Visualization Impact:
These performance visualizations decomposed inscrutable target data into natural visual cues. The gauge chart provided immediate measurement of total performance, and the category achievement bars highlighted specific product categories that required action. The monthly trend line indicated temporal trends in target achievement, making forecasting and target setting for future periods more precise.

- This near-ideal congruence between target and actual performance suggests effective target setting or meticulous performance management
- While success seems evident, the 1% shortfall also reveals room for marginal betterment

Dramatic Category Performance Variations:
- Electronics well surpasses targets (128.1%), with excellent over-performance
- Clothing severely underperforms (79.9%), in the "Near Target" yellow band
- Furniture is short of targets by only (95.7%), short of the "On Target" threshold
- These variations offset each other to create the overall 99.0% success

Category Contribution Insights:
- Electronics represents the largest proportion of total sales (38.3%) and is also the category over targets
- Clothing (32.2%) and Furniture (29.5%) have similar sales contributions with different target achievement rates
- This shows the company has recorded balanced revenue streams across categories

Performance Gap Analysis
Mixed Achievement Pattern:

The bubble chart indicates wide gaps between categories:
- Electronics: +28% over target with enormous volume of sales
- Clothing: -20% below target with high volume
- Furniture: -4% below target with moderate volume
- The magnitude of these gaps suggests structural issues rather than random fluctuation

Target Structure Analysis:
- The monthly performance trend line is flat at or near 0% in the series
- This is compared to the extreme October surge of sales seen in previous analyses
- Suggests targets can be adjusted monthly based on previous trends or seasonality

Multidimensional Category Analysis:
The radar chart shows a number of strengths in categories:
- Clothing leads in Sales Growth and Customer Satisfaction
- Electronics leads in Achievement % and a strong Market Share
- Furniture is average in all dimensions but not a category leader in any

Cumulative Performance Graph:
- The cumulative performance graph shows a sharp spike in October
- Target achievement is quite consistent throughout the year
- This means target changes probably fit anticipated seasonal fluctuations

Interactive Dashboard:

This R Shiny interactive dashboard provides strategic decision-making sales analytics across product categories, geographic markets, and performance measures. With an emphasis on usability and actionability, it combines multiple data sources to enable the discovery of growth and optimization potential.

Core Features

Performance Overview:

- Live summary figures showing total sales ($335,856), total profit ($21,198), and profit margin (6.31%)
- Monthly sales trend chart highlighting seasonal variations and peaks (October)
- Category breakdown showing balanced distribution across Electronics (37.2%), Clothing (36.4%), and Furniture (26.4%)

Regional Analysis:

- Best states visualization showing Madhya Pradesh and Maharashtra as top-selling
- State profit margin analysis showing Delhi and West Bengal as top performing in profitability
- City-level drill-down automatically showing Indore's strong position within Madhya Pradesh

Product Category Insights:

- Profit measure comparison visualization by category, dual-axis
- Printers are best-selling electronics sub-category
- Phone, Electronic Games, and Accessories performance numbers clearly labeled

Target Achievement Framework:

- Real-time tracking of achievement by status through color-coded gauge visualization
- Tracking of performance against targets at the category level
- Monthly tracking of achievement with target threshold indicators
- Sub-category analysis with Printers as best-selling electronics item
- Offer Detail Performance Indicators for Phones, Electronic Games, and Accessories

Interactive Filters

The dashboard features a powerful filtering system allowing users to:

- Filter by state to target specific regional markets
- Select product categories to dive deeper into analysis
- Ratchet date ranges to see performance on individual time frames

Every visualization dynamically reacts to filter selections, allowing detailed analysis of specialized segments of interest.

Key Insights Revealed

- Regional Performance Disparities: Large gaps between sales volume leaders (Madhya Pradesh) and profit margin leaders (Delhi)