Cafe Sales Prediction

Predicting cafe sales using historical transaction data.

In [304]:

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

plt.style.use('dark_background')
df = pd.read_csv('/content/dirty_cafe_sales.csv')
df.head(100)
```

Out[304]:

	Transaction ID	Item	Quantity	Price Per Unit	Total Spent	Payment Method	Location	Transaction Date
0	TXN_1961373	Coffee	2	2.0	4.0	Credit Card	Takeaway	2023-09-08
1	TXN_4977031	Cake	4	3.0	12.0	Cash	In-store	2023-05-16
2	TXN_4271903	Cookie	4	1.0	ERROR	Credit Card	In-store	2023-07-19
3	TXN_7034554	Salad	2	5.0	10.0	UNKNOWN	UNKNOWN	2023-04-27
4	TXN_3160411	Coffee	2	2.0	4.0	Digital Wallet	In-store	2023-06-11
...	...	...	...	...	...	...	...	...
95	TXN_8268061	Salad	3	5.0	15.0	ERROR	Takeaway	2023-08-20
96	TXN_5220895	Salad	5	5.0	25.0	Cash	In-store	2023-06-10
97	TXN_3085509	Coffee	4	2.0	8.0	Digital Wallet	In-store	2023-04-15
98	TXN_9999113	Juice	4	3.0	12.0	Cash	Takeaway	2023-05-27
99	TXN_8779771	Coffee	4	2.0	8.0	Cash	In-store	2023-07-25

100 rows × 8 columns

Data Preprocessing and EDA

In [305]:

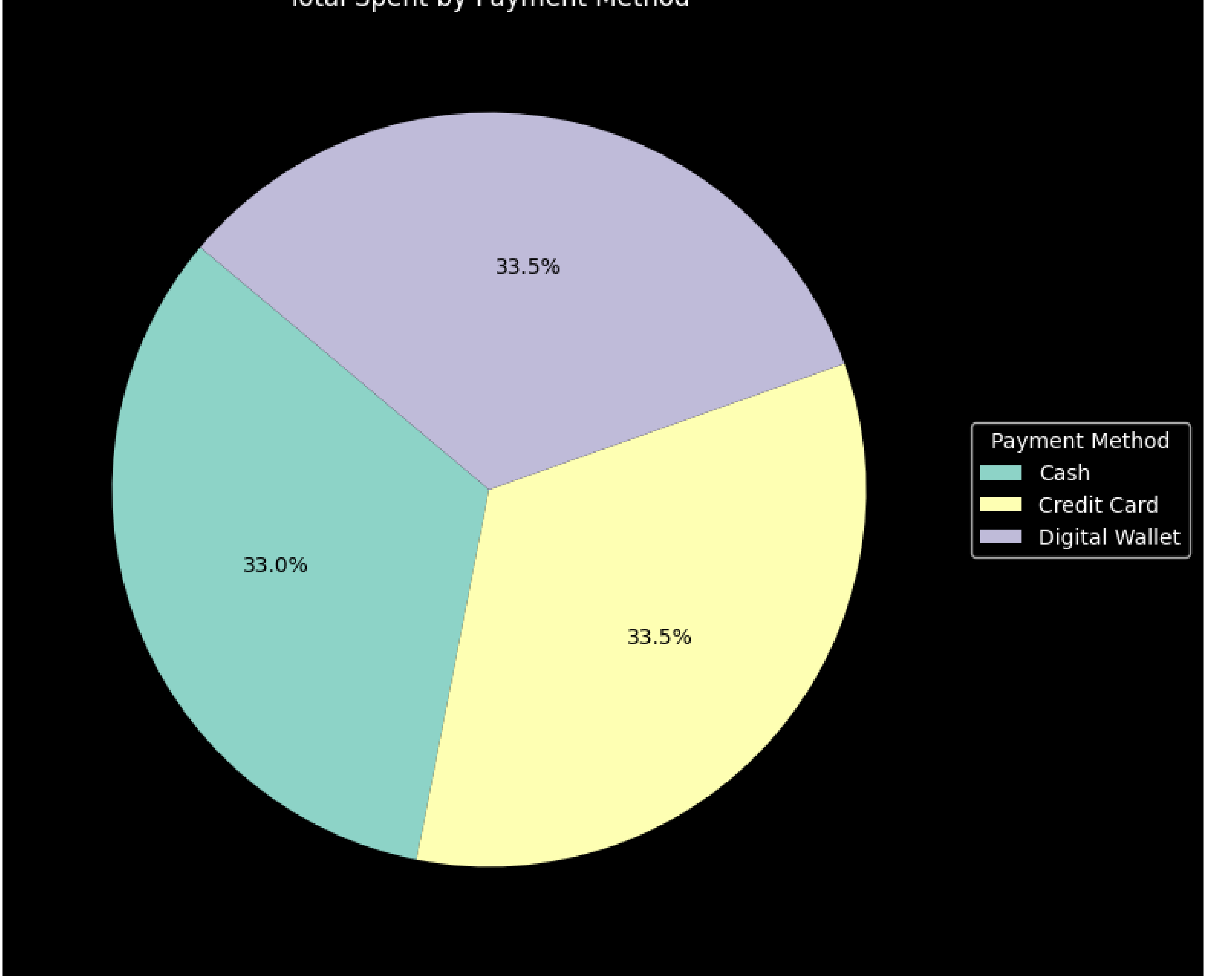
```
import numpy as np
df.replace(['ERROR', 'UNKNOWN'], np.nan, inplace=True)
```

In [306]:

```
df.columns = df.columns.str.replace(' ', '_')
```

In [307]:

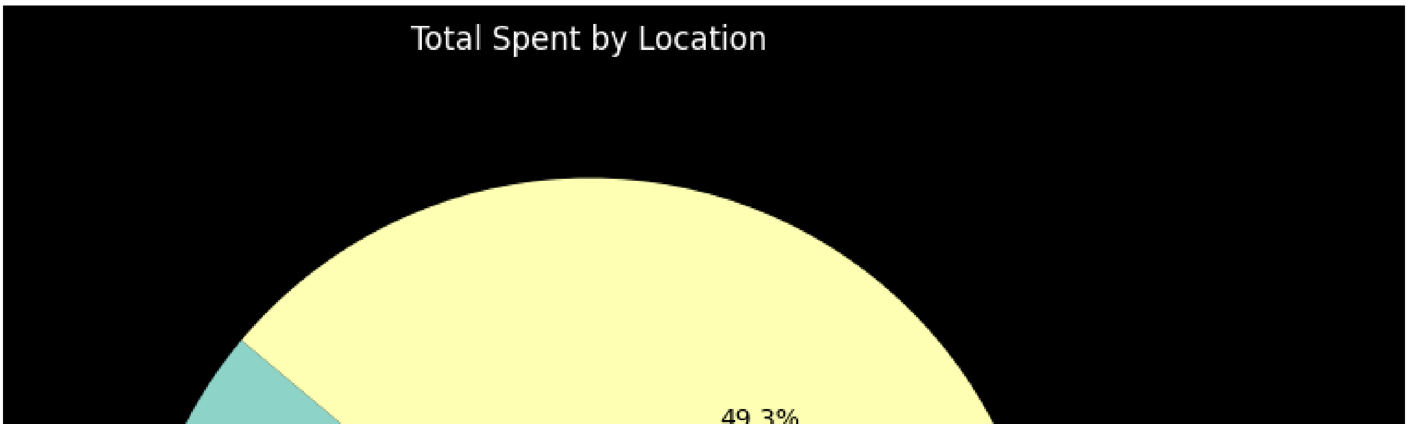
```
df_copy = df.copy()
df_copy['Total_Spent'] = pd.to_numeric(df_copy['Total_Spent'], errors='coerce')
df_copy.dropna(subset=['Payment_Method', 'Total_Spent'], inplace=True)
payment_method_totals = df_copy.groupby('Payment_Method')['Total_Spent'].sum()
plt.figure(figsize=(8, 8))
wedges, texts, autotexts = plt.pie(payment_method_totals, labels=payment_method_totals.ind
ex, autopct='%1.1f%%', startangle=140, textprops={'color': 'black'})
plt.title('Total Spent by Payment Method')
plt.legend(wedges, payment_method_totals.index, title="Payment Method", loc="center left",
bbox_to_anchor=(1, 0, 0.5, 1))
plt.show()
```

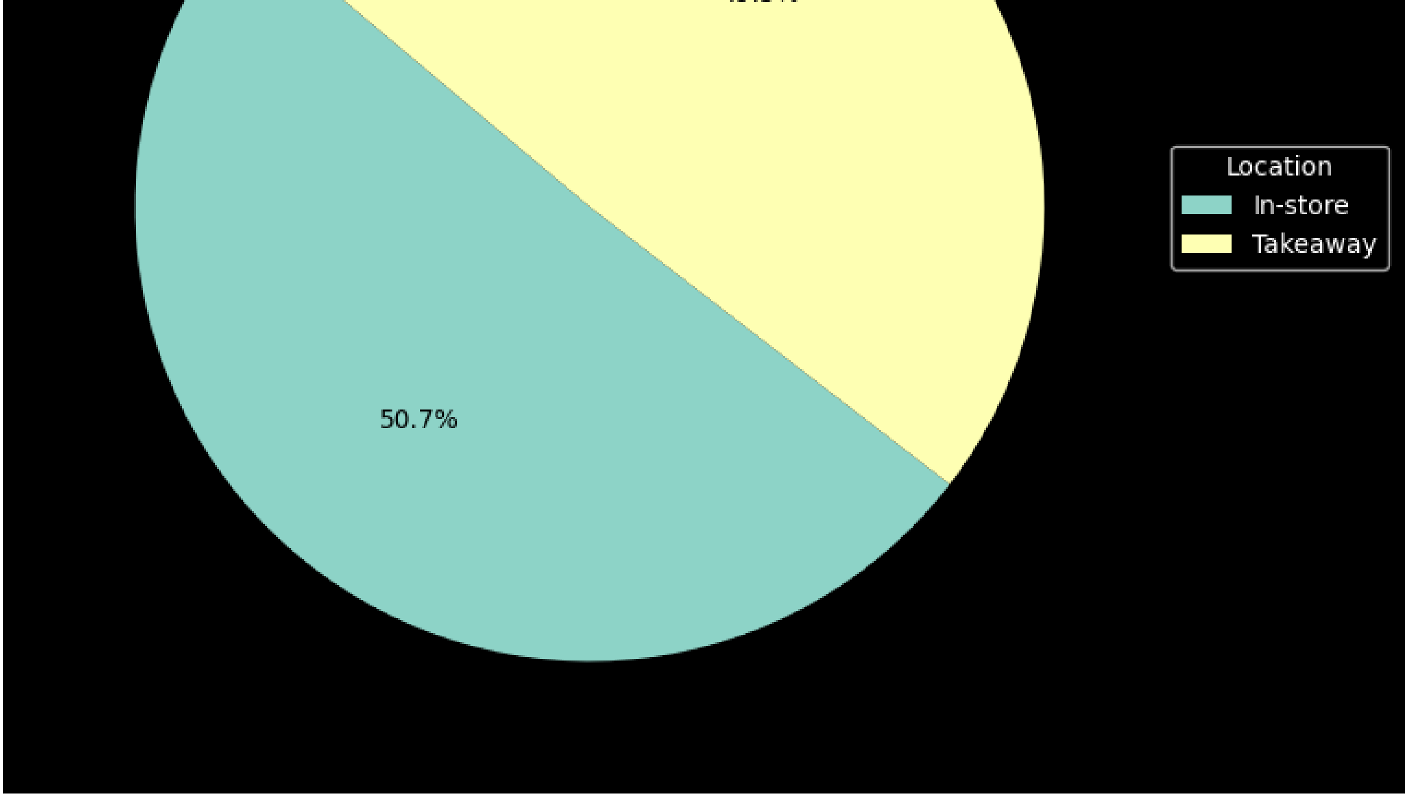


After analyzing the distribution of the Payment Method feature, we observe that the total spending is almost evenly divided among the three methods (approximately 33%, 33.5%, and 33.5%). This indicates that the Payment Method does not significantly impact the target variable (Total Spent), and therefore, it can be considered for removal as a non-informative feature.

In [308]:

```
df_copy = df.copy()
df_copy['Total_Spent'] = pd.to_numeric(df_copy['Total_Spent'], errors='coerce')
df_copy.dropna(subset=['Location', 'Total_Spent'], inplace=True)
location_totals = df_copy.groupby('Location')['Total_Spent'].sum()
plt.figure(figsize=(8, 8))
wedges, texts, autotexts = plt.pie(location_totals, labels=location_totals.index, autopct=
'%1.1f%%', startangle=140, textprops={'color': 'black'})
plt.title('Total Spent by Location')
plt.legend(wedges, location_totals.index, title="Location", loc="center left", bbox_to_anc
hor=(1, 0, 0.5, 1))
plt.show()
```

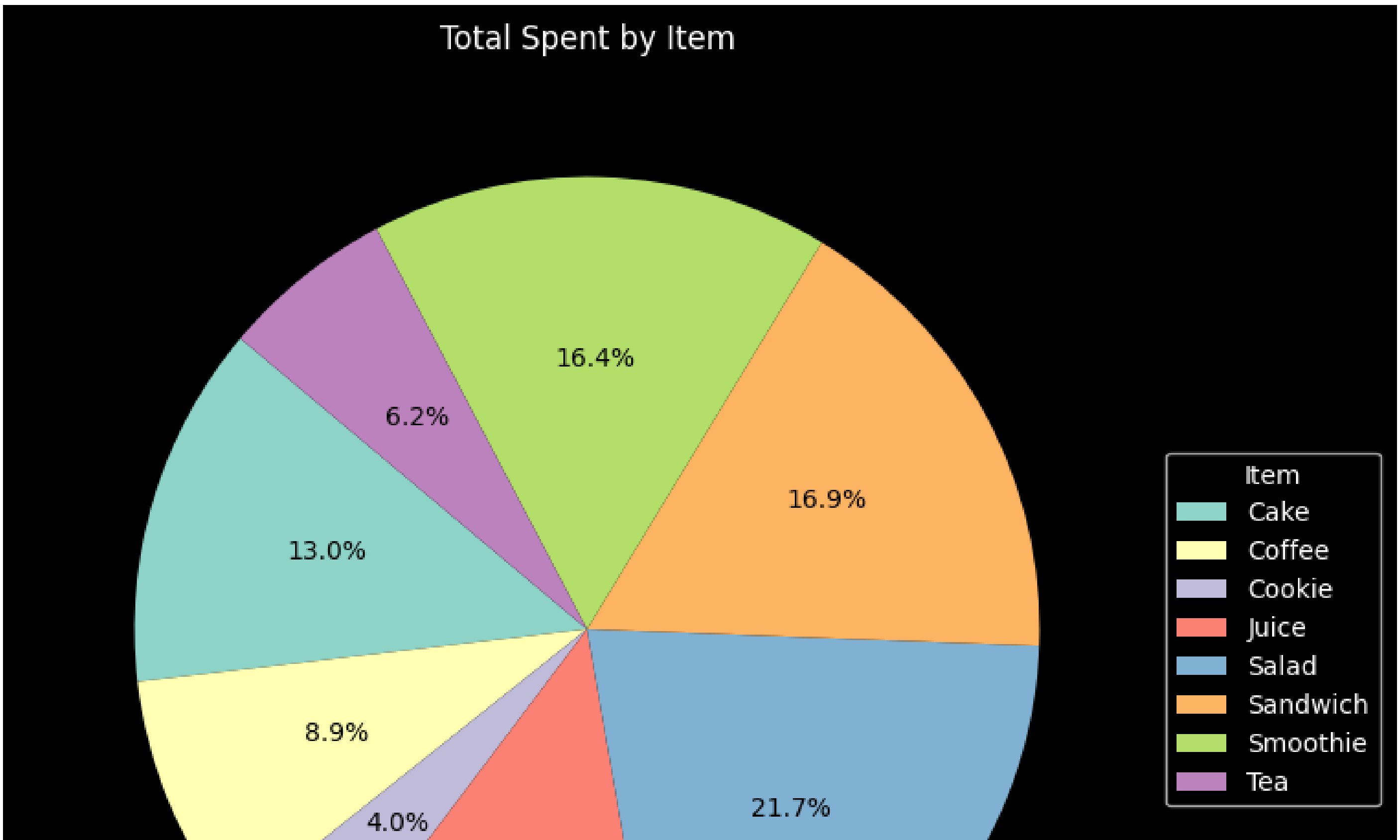


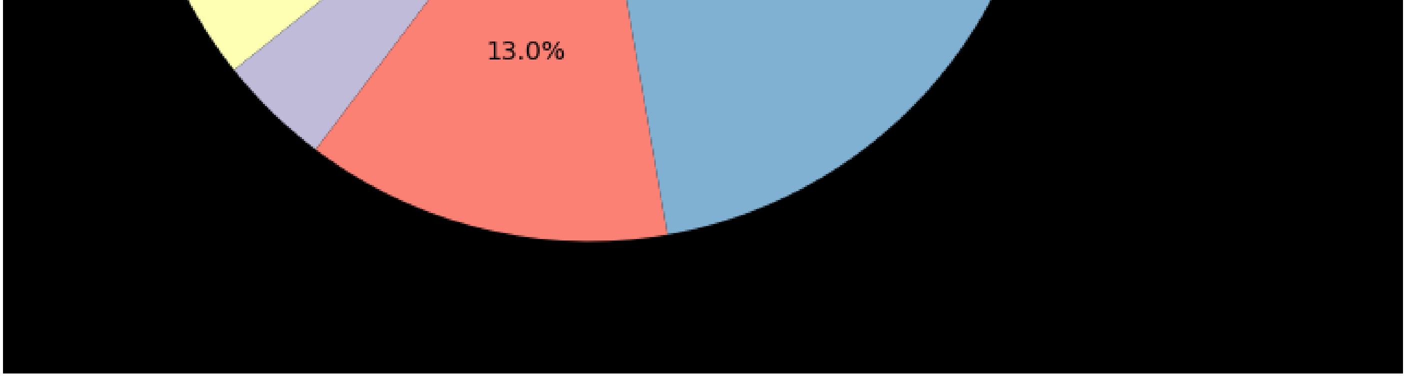


The analysis of the Location feature shows that total spending is nearly evenly split between In-Store (50.7%) and Takeaway (49.3%). This minimal difference suggests that Location has little to no impact on the target variable (Total Spent), and thus, it can be considered for removal as an uninformative feature.

In [309]:

```
df_copy = df.copy()
df_copy['Total_Spent'] = pd.to_numeric(df_copy['Total_Spent'], errors='coerce')
df_copy.dropna(subset=['Item', 'Total_Spent'], inplace=True)
location_totals = df_copy.groupby('Item')['Total_Spent'].sum()
plt.figure(figsize=(8, 8))
wedges, texts, autotexts = plt.pie(location_totals, labels=location_totals.index, autopct=
'%1.1f%%', startangle=140, textprops={'color': 'black'})
plt.title('Total Spent by Item')
plt.legend(wedges, location_totals.index, title="Item", loc="center left", bbox_to_anchor=
(1, 0, 0.5, 1))
plt.show()
```

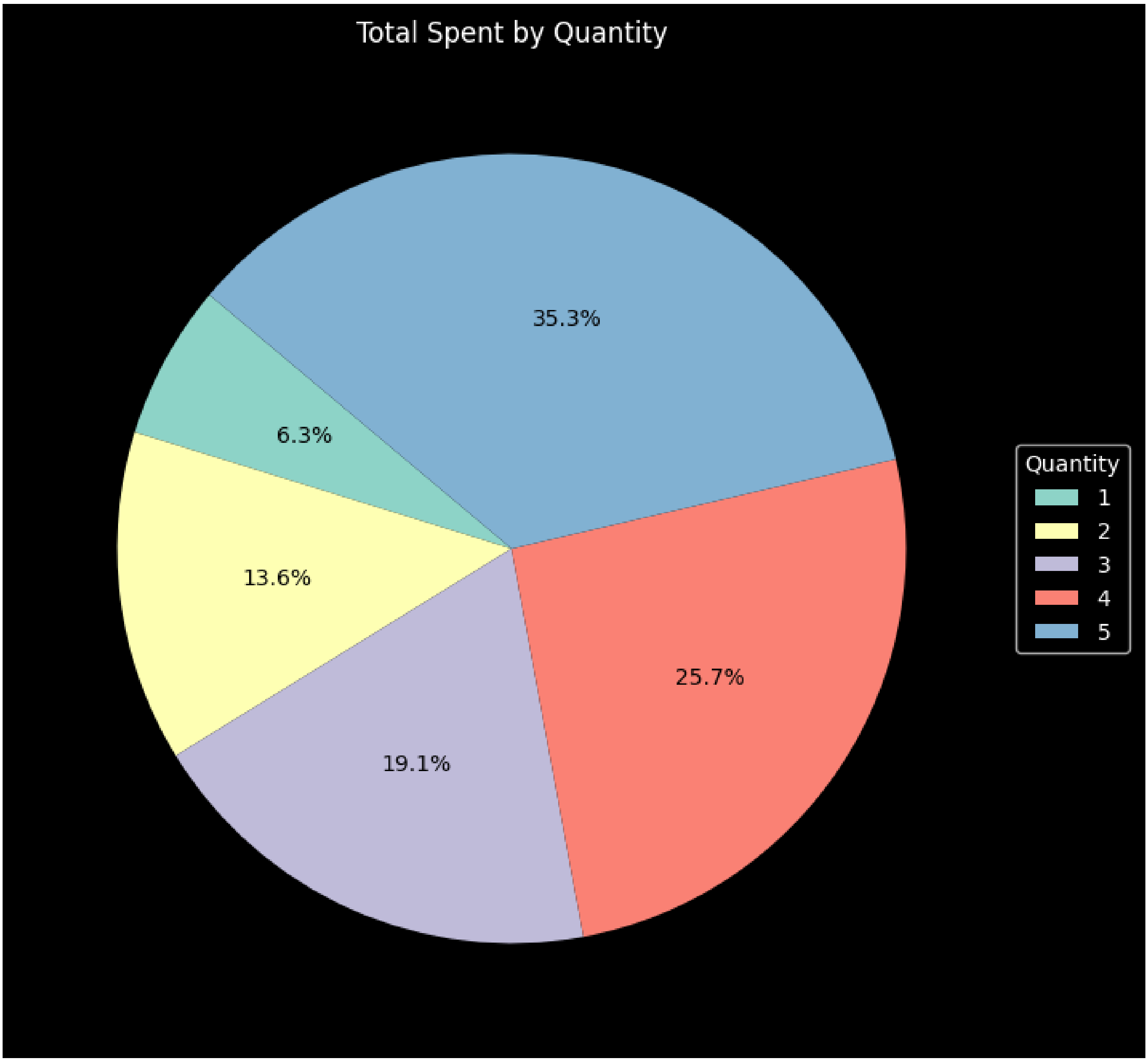




We observe that there are 8 different items, and the percentage of total spending across these items varies significantly — ranging from 4% to 21.7%. This high variability indicates that the Item feature plays an important role in our analysis and should be retained for further study.

In [310]:

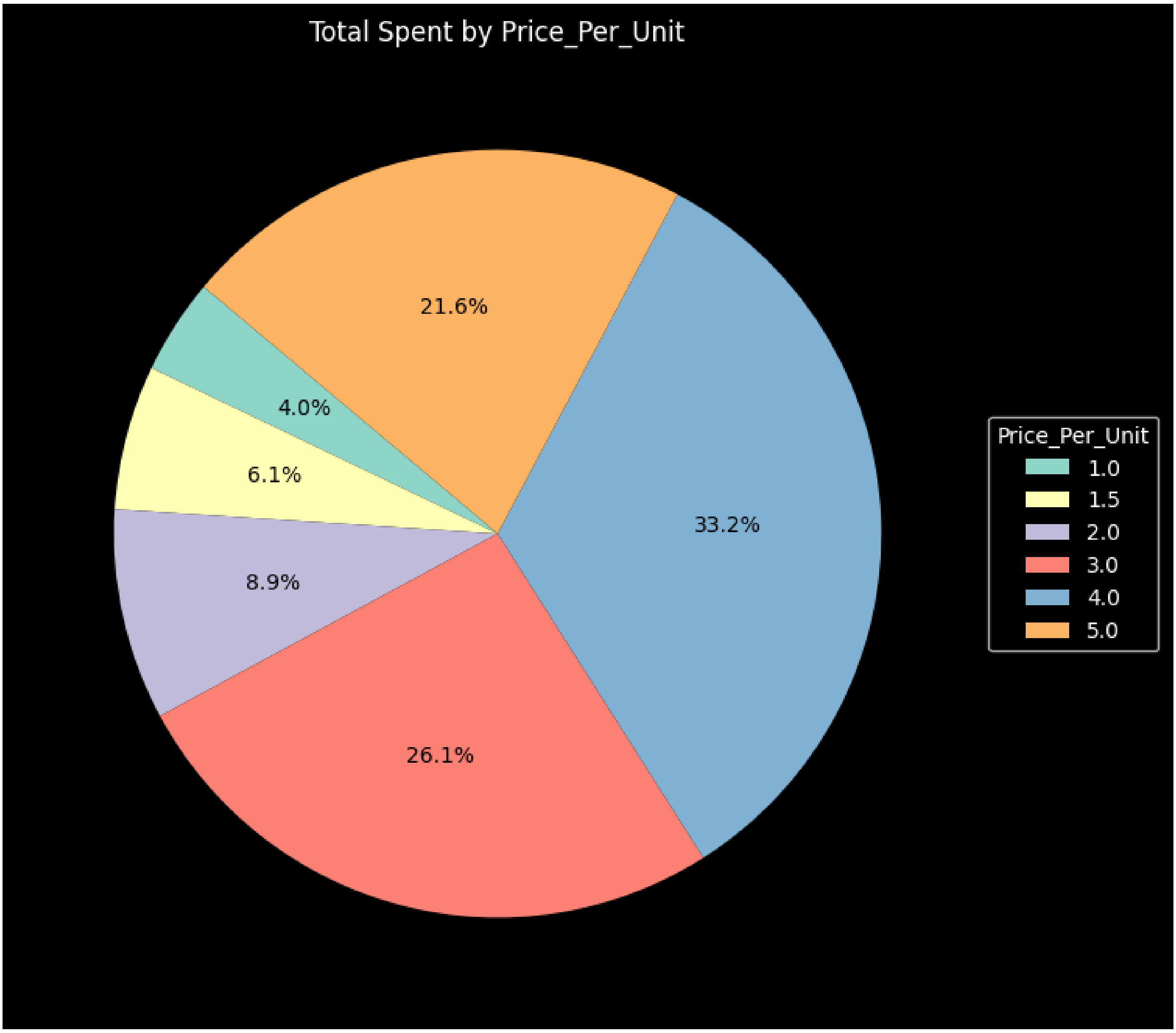
```
df_copy = df.copy()
df_copy['Total_Spent'] = pd.to_numeric(df_copy['Total_Spent'], errors='coerce')
df_copy.dropna(subset=['Quantity', 'Total_Spent'], inplace=True)
location_totals = df_copy.groupby('Quantity')['Total_Spent'].sum()
plt.figure(figsize=(8, 8))
wedges, texts, autotexts = plt.pie(location_totals, labels=location_totals.index, autopct=
'%1.1f%%', startangle=140, textprops={'color': 'black'})
plt.title('Total Spent by Quantity')
plt.legend(wedges, location_totals.index, title="Quantity", loc="center left", bbox_to_anc
hor=(1, 0, 0.5, 1))
plt.show()
```



The quantity feature significantly influences the target variable, total spent. When the total spent percentage is divided into five levels ranging from 1 to 5, it varies substantially—from as low as 6.3% up to 35.3%. This indicates that as the quantity increases, the total spent tends to rise noticeably, highlighting a strong positive relationship between quantity and total Spent.

In [311]:

```
df_copy = df.copy()
df_copy['Total_Spent'] = pd.to_numeric(df_copy['Total_Spent'], errors='coerce')
df_copy.dropna(subset=['Price_Per_Unit', 'Total_Spent'], inplace=True)
location_totals = df_copy.groupby('Price_Per_Unit')['Total_Spent'].sum()
plt.figure(figsize=(8, 8))
wedges, texts, autotexts = plt.pie(location_totals, labels=location_totals.index, autopct=
'%1.1f%%', startangle=140, textprops={'color': 'black'})
plt.title('Total Spent by Price_Per_Unit')
plt.legend(wedges, location_totals.index, title="Price_Per_Unit", loc="center left", bbox_
to_anchor=(1, 0, 0.5, 1))
plt.show()
```



The price per unit feature plays an important role in determining the total spent. When the total spent percentage is divided into five levels from 1 to 5, it varies widely—from 6.3% up to 35.3%. This variation shows that changes in the price per unit are closely associated with changes in total spending, indicating that higher unit prices generally lead to higher total expenditures.

In [312]:

```
nan_transaction_date_count = df['Transaction_Date'].isna().sum()
print(f"Number of rows where 'transaction_date' is NaN: {nan_transaction_date_count}")
```

```
Number of rows where 'transaction_date' is NaN: 460
```

Out of 10,000 rows, 460 have a missing transaction_date value, representing only 4.6% of the data. Since this is a relatively small proportion, these rows can be safely dropped without significantly impacting the overall analysis.

In [313]:

```
df.dropna(subset=['Transaction_Date'], inplace=True)
```

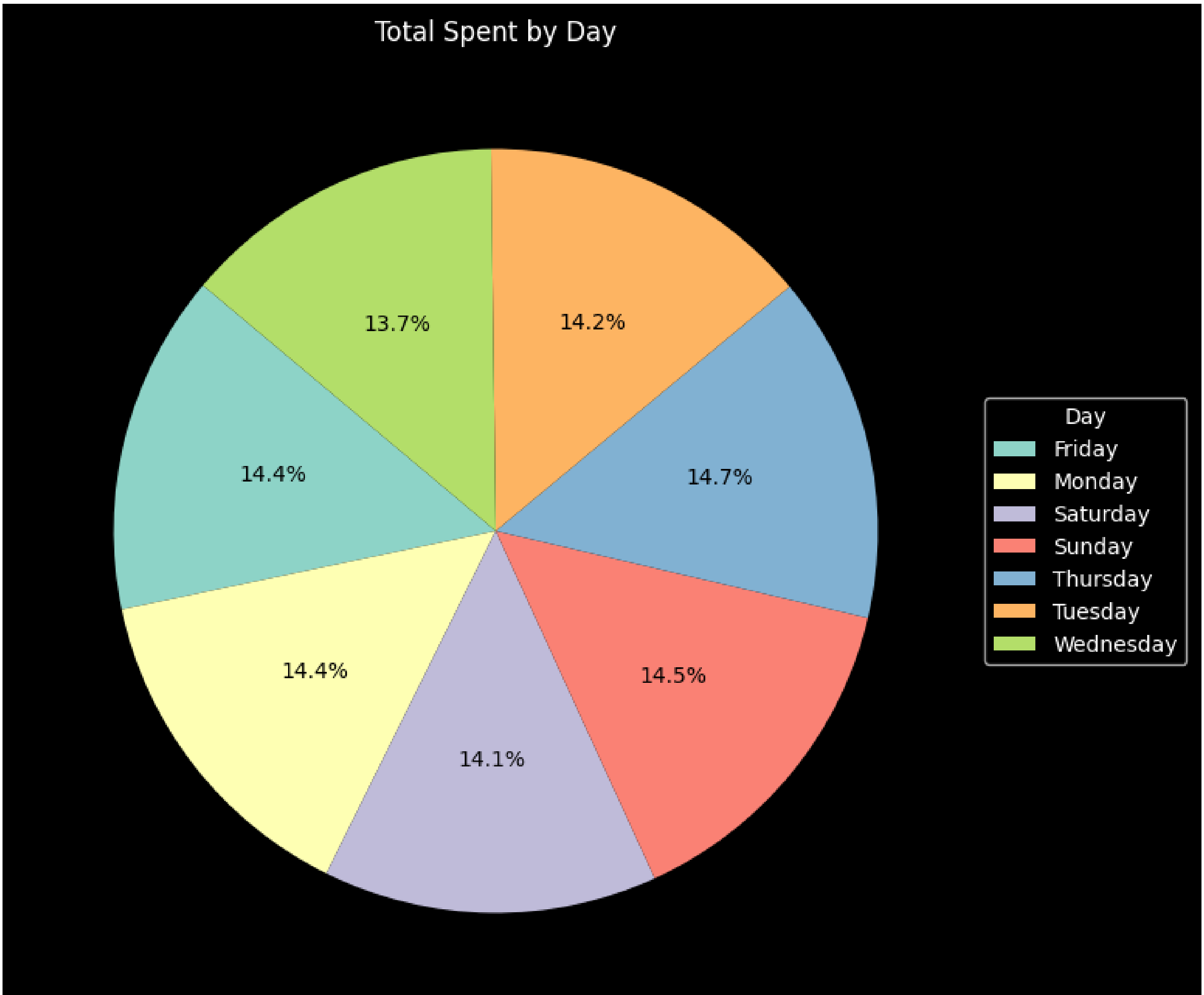
In [314]:

```
df['Transaction_Date'] = pd.to_datetime(df['Transaction_Date'])
df['Day'] = df['Transaction_Date'].dt.day_name()
df['Month'] = df['Transaction_Date'].dt.month
```

To enhance the analysis, I plan to extract the day of the week and month from the transaction_date feature. These time-based features are likely to be important, as customer behavior can vary depending on the day or month. For example, sales may be higher on weekends or during specific months. By creating these features, we can later visualize the data to verify whether such patterns exist.

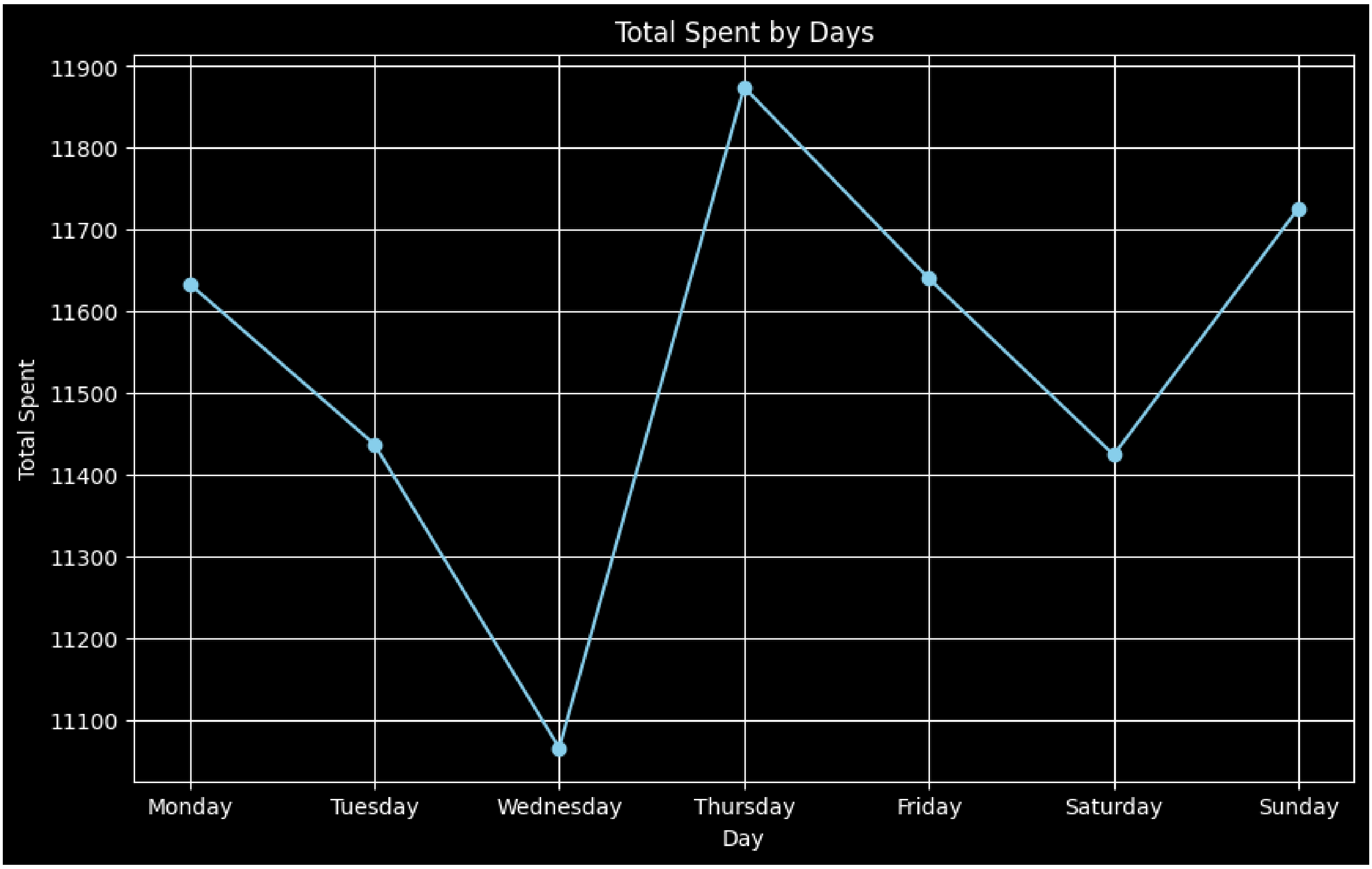
In [315]:

```
df_copy = df.copy()
df_copy['Total_Spent'] = pd.to_numeric(df_copy['Total_Spent'], errors='coerce')
df_copy.dropna(subset=['Day', 'Total_Spent'], inplace=True)
location_totals = df_copy.groupby('Day')['Total_Spent'].sum()
plt.figure(figsize=(8, 8))
wedges, texts, autotexts = plt.pie(location_totals, labels=location_totals.index, autopct=
'%1.1f%%', startangle=140, textprops={'color': 'black'})
plt.title('Total Spent by Day')
plt.legend(wedges, location_totals.index, title="Day", loc="center left", bbox_to_anchor=(
1, 0, 0.5, 1))
plt.show()
```



In [316]:

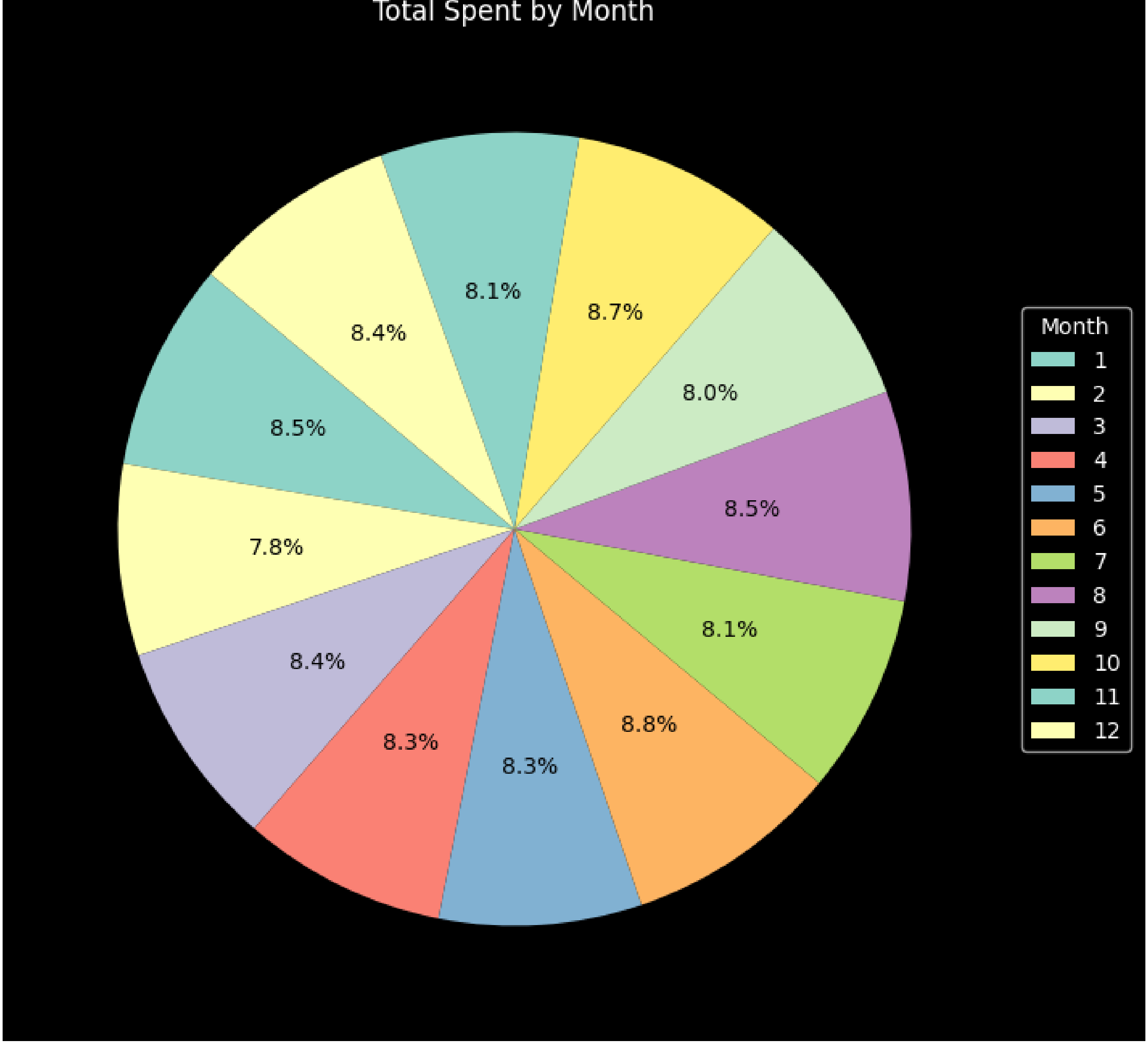
```
df_copy = df.copy()
df_copy['Total_Spent'] = pd.to_numeric(df_copy['Total_Spent'], errors='coerce')
df_copy.dropna(subset=['Day', 'Total_Spent'], inplace=True)
month_totals = df_copy.groupby('Day')['Total_Spent'].sum()

# Define the desired order of the days
day_order = ['Monday', 'Tuesday', 'Wednesday', 'Thursday', 'Friday', 'Saturday', 'Sunday']

# Reindex the month_totals Series to reflect the desired order
month_totals = month_totals.reindex(day_order)

plt.figure(figsize=(10, 6))
plt.plot(month_totals.index, month_totals.values, marker='o', linestyle='-', color='skyblu
e')
plt.title('Total Spent by Days')
plt.xlabel('Day')
plt.ylabel('Total Spent')
plt.xticks(month_totals.index)
plt.grid(True)
plt.show()
```



The total spent is not very affected by which day it is. But it is still considerable.

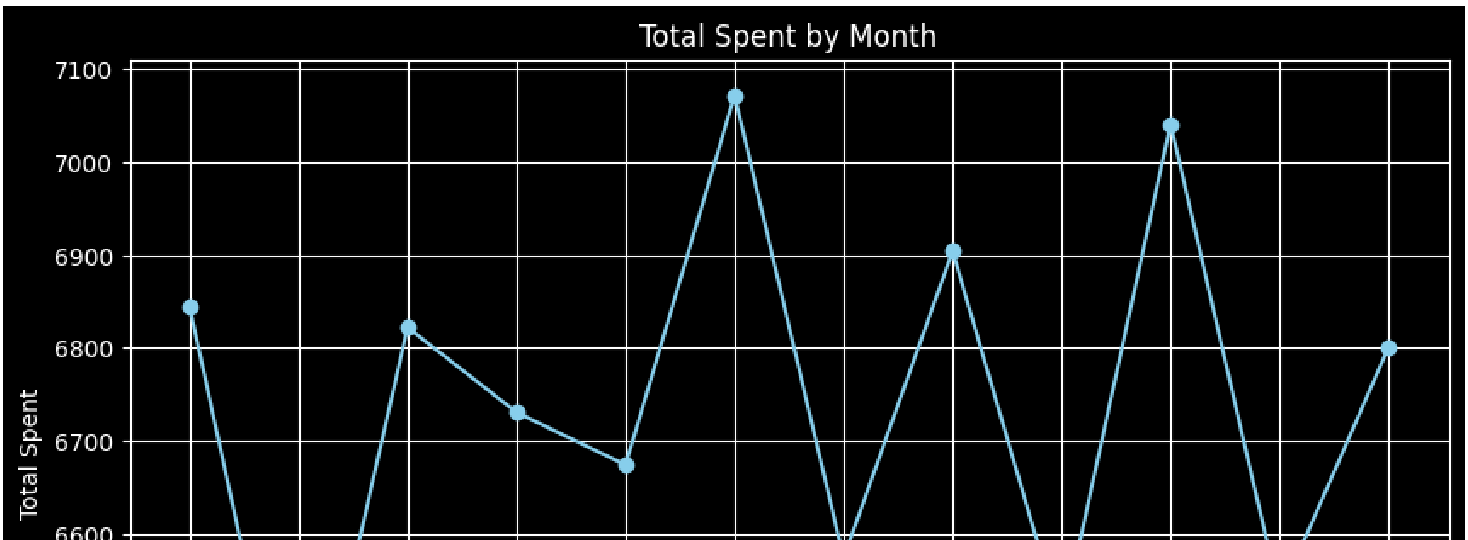
In [317]:

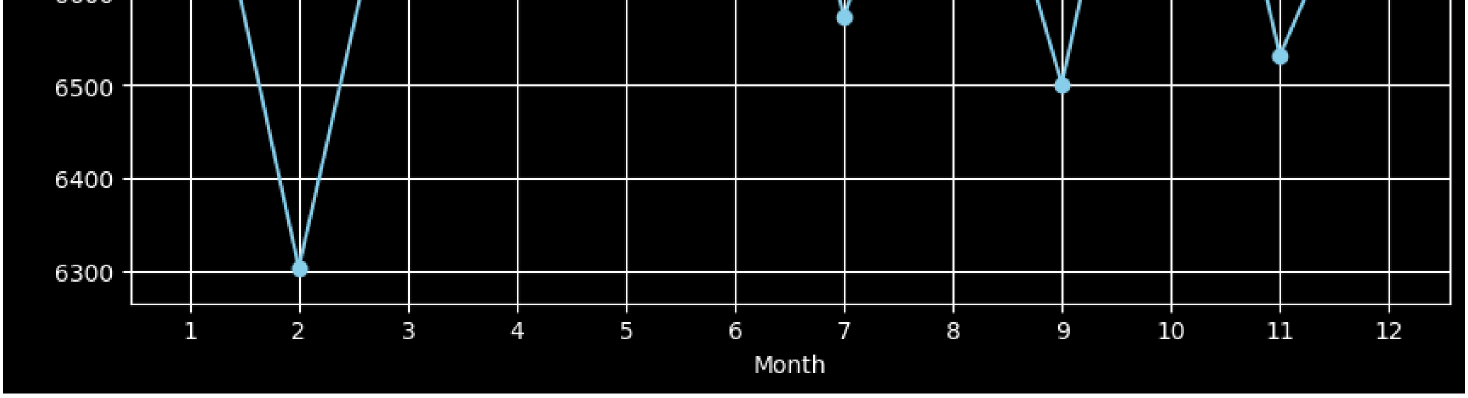
```
df_copy = df.copy()
df_copy['Total_Spent'] = pd.to_numeric(df_copy['Total_Spent'], errors='coerce')
df_copy.dropna(subset=['Month', 'Total_Spent'], inplace=True)
location_totals = df_copy.groupby('Month')['Total_Spent'].sum()
plt.figure(figsize=(8, 8))
wedges, texts, autotexts = plt.pie(location_totals, labels=location_totals.index, autopct=
'%1.1f%%', startangle=140, textprops={'color': 'black'})
plt.title('Total Spent by Month')
plt.legend(wedges, location_totals.index, title="Month", loc="center left", bbox_to_anchor
=(1, 0, 0.5, 1))
plt.show()
```



In [318]:

```
df_copy = df.copy()
df_copy['Total_Spent'] = pd.to_numeric(df_copy['Total_Spent'], errors='coerce')
df_copy.dropna(subset=['Month', 'Total_Spent'], inplace=True)
month_totals = df_copy.groupby('Month')['Total_Spent'].sum()
plt.figure(figsize=(10, 6))
plt.plot(month_totals.index, month_totals.values, marker='o', linestyle='-', color='skyblu
e')
plt.title('Total Spent by Month')
plt.xlabel('Month')
plt.ylabel('Total Spent')
plt.xticks(month_totals.index)
plt.grid(True)
plt.show()
```





The total spent is not very affected by which month we are in. But it is still considerable.

In [319]:

```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Index: 9540 entries, 0 to 9999
Data columns (total 10 columns):
 #   Column            Non-Null Count  Dtype
---  ------            --------------  -----
 0   Transaction_ID    9540 non-null   object
 1   Item              8613 non-null   object
 2   Quantity          9086 non-null   object
 3   Price_Per_Unit    9034 non-null   object
 4   Total_Spent       9064 non-null   object
 5   Payment_Method    6525 non-null   object
 6   Location          5761 non-null   object
 7   Transaction_Date  9540 non-null   datetime64[ns]
 8   Day               9540 non-null   object
 9   Month             9540 non-null   int32
dtypes: datetime64[ns](1), int32(1), object(8)
memory usage: 782.6+ KB
```

In [320]:

```
df.drop(['Location', 'Payment_Method'], axis=1, inplace=True)

df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Index: 9540 entries, 0 to 9999
Data columns (total 8 columns):
 #   Column            Non-Null Count  Dtype
---  ------            --------------  -----
 0   Transaction_ID    9540 non-null   object
 1   Item              8613 non-null   object
 2   Quantity          9086 non-null   object
 3   Price_Per_Unit    9034 non-null   object
 4   Total_Spent       9064 non-null   object
 5   Transaction_Date  9540 non-null   datetime64[ns]
 6   Day               9540 non-null   object
 7   Month             9540 non-null   int32
dtypes: datetime64[ns](1), int32(1), object(6)
memory usage: 633.5+ KB
```

The following is a code to check price per each item.

In [321]:

```
if 'Price_Per_Unit' in df.columns and 'Item' in df.columns:
    item_prices = df[['Item', 'Price_Per_Unit']].copy()
    item_prices.drop_duplicates(inplace=True)
    item_prices['Price_Per_Unit'] = pd.to_numeric(item_prices['Price_Per_Unit'], errors='c
oerce')
    item_prices.dropna(subset=['Price_Per_Unit'], inplace=True)
    print("Item Name and Price Per Unit:")
```

```
    for index, row in item_prices.iterrows():
        print(f"Item: {row['Item']}, Price Per Unit: {row['Price_Per_Unit']:.2f}")
else:
    print("The dataframe does not contain both 'Item' and 'Price_Per_Unit' columns.")
```

```
Item Name and Price Per Unit:
Item: Coffee, Price Per Unit: 2.00
Item: Cake, Price Per Unit: 3.00
Item: Cookie, Price Per Unit: 1.00
Item: Salad, Price Per Unit: 5.00
Item: Smoothie, Price Per Unit: 4.00
Item: nan, Price Per Unit: 3.00
Item: Sandwich, Price Per Unit: 4.00
Item: nan, Price Per Unit: 1.50
Item: Juice, Price Per Unit: 3.00
Item: nan, Price Per Unit: 2.00
Item: nan, Price Per Unit: 1.00
Item: Tea, Price Per Unit: 1.50
Item: nan, Price Per Unit: 5.00
Item: nan, Price Per Unit: 4.00
```

Price Per Unit of Cookie is uniquely shared with one nan. Therefore we can use this info to fix the nan value that has Price Per Unit = 1. Similarly, we apply the same procedure for Tea and Salad.

In [322]:

```
df.loc[(df['Price_Per_Unit'] == 1) & (df['Item'].isna()), 'Item'] = 'Cookie'
```

In [323]:

```
df.loc[(df['Price_Per_Unit'] == 1.5) & (df['Item'].isna()), 'Item'] = 'Tea'
```

In [324]:

```
df.loc[(df['Price_Per_Unit'] == 5) & (df['Item'].isna()), 'Item'] = 'Salad'
```

In [325]:

```
if 'Price_Per_Unit' in df.columns and 'Item' in df.columns:
    item_prices = df[['Item', 'Price_Per_Unit']].copy()
    item_prices['Price_Per_Unit'] = pd.to_numeric(item_prices['Price_Per_Unit'], errors='c
oerce')
    item_prices.dropna(subset=['Price_Per_Unit'], inplace=True)
    nan_item_prices = item_prices[item_prices['Item'].isna()]['Price_Per_Unit'].unique()
    for price in nan_item_prices:
        possible_items = item_prices[(item_prices['Price_Per_Unit'] == price) & (item_pric
es['Item'].notna())]['Item'].unique()
        if len(possible_items) > 0:
            item_distribution = item_prices[(item_prices['Price_Per_Unit'] == price) & (it
em_prices['Item'].notna())]['Item'].value_counts(normalize=True)
            nan_indices = df.index[(df['Item'].isna()) & (pd.to_numeric(df['Price_Per_Unit
'], errors='coerce') == price)]
            if not item_distribution.empty and len(nan_indices) > 0:
                df.loc[nan_indices, 'Item'] = np.random.choice(item_distribution.index,
                                                               size=len(nan_indices),
                                                               p=item_distribution.values
)
        else:
             print(f"Warning: No valid items found for Price_Per_Unit: {price}. NaNs for t
his price will remain.")
    print("Unique Item Name and Price Per Unit after filling NaNs:")
    item_prices_filled = df[['Item', 'Price_Per_Unit']].copy()
    item_prices_filled['Price_Per_Unit'] = pd.to_numeric(item_prices_filled['Price_Per_Uni
t'], errors='coerce')
    item_prices_filled.dropna(subset=['Item', 'Price_Per_Unit'], inplace=True)
    item_prices_filled['Item_lower'] = item_prices_filled['Item'].str.lower()
    unique_item_prices = item_prices_filled.drop_duplicates(subset=['Item_lower', 'Price_P
er_Unit'])
    for index, row in unique_item_prices.iterrows():
        print(f"Item: {row['Item']}, Price Per Unit: {row['Price_Per_Unit']:.2f}")
else:
```

```
    print("The dataframe does not contain both 'Item' and 'Price_Per_Unit' columns.")
```

```
Unique Item Name and Price Per Unit after filling NaNs:
Item: Coffee, Price Per Unit: 2.00
Item: Cake, Price Per Unit: 3.00
Item: Cookie, Price Per Unit: 1.00
Item: Salad, Price Per Unit: 5.00
Item: Smoothie, Price Per Unit: 4.00
Item: Juice, Price Per Unit: 3.00
Item: Sandwich, Price Per Unit: 4.00
Item: Tea, Price Per Unit: 1.50
```

Now the items are filled successfully when one of price per unit or item is nan.

In the following we will deal with price per unit and item being both nan at the same time in an instance.

In [326]:

```
df[df['Item'].isna()]
```

Out[326]:

	Transaction_ID	Item	Quantity	Price_Per_Unit	Total_Spent	Transaction_Date	Day	Month
118	TXN_4633784	NaN	5	NaN	15.0	2023-02-06	Monday	2
151	TXN_4031509	NaN	4	NaN	16.0	2023-01-04	Wednesday	1
289	TXN_3495950	NaN	4	NaN	6.0	2023-02-19	Sunday	2
334	TXN_2523298	NaN	4	NaN	6.0	2023-03-25	Saturday	3
550	TXN_4186681	NaN	4	NaN	6.0	2023-05-24	Wednesday	5
750	TXN_5787508	NaN	3	NaN	9.0	2023-07-23	Sunday	7
818	TXN_7940202	NaN	1	NaN	4.0	2023-07-23	Sunday	7
1154	TXN_2473090	NaN	2	NaN	3.0	2023-03-03	Friday	3
1337	TXN_5031214	NaN	5	NaN	5.0	2023-07-29	Saturday	7
1377	TXN_8396271	NaN	2	NaN	2.0	2023-09-12	Tuesday	9
1589	TXN_5245399	NaN	5	NaN	10.0	2023-12-25	Monday	12
1761	TXN_3611851	NaN	4	NaN	NaN	2023-02-09	Thursday	2
1786	TXN_1923349	NaN	4	NaN	6.0	2023-07-06	Thursday	7
2002	TXN_5206049	NaN	3	NaN	3.0	2023-06-24	Saturday	6
2227	TXN_3200203	NaN	2	NaN	8.0	2023-12-04	Monday	12
2289	TXN_7524977	NaN	4	NaN	NaN	2023-12-09	Saturday	12
2596	TXN_4844386	NaN	5	NaN	15.0	2023-10-28	Saturday	10
2610	TXN_8266689	NaN	5	NaN	15.0	2023-12-03	Sunday	12
2962	TXN_9702662	NaN	4	NaN	16.0	2023-07-10	Monday	7
3013	TXN_1842697	NaN	5	NaN	15.0	2023-10-25	Wednesday	10
3404	TXN_7797231	NaN	5	NaN	10.0	2023-05-05	Friday	5
3434	TXN_6457997	NaN	1	NaN	4.0	2023-12-12	Tuesday	12
3666	TXN_8616276	NaN	2	NaN	3.0	2023-07-22	Saturday	7
3739	TXN_4849180	NaN	5	NaN	15.0	2023-10-14	Saturday	10
3779	TXN_7376255	NaN	NaN	NaN	25.0	2023-05-27	Saturday	5
3900	TXN_5093855	NaN	4	NaN	6.0	2023-11-23	Thursday	11
4092	TXN_1840897	NaN	1	NaN	5.0	2023-06-03	Saturday	6
4152	TXN_9646000	NaN	2	NaN	NaN	2023-12-14	Thursday	12
4621	TXN_7844352	NaN	2	NaN	6.0	2023-07-26	Wednesday	7
5039	TXN_9514452	NaN	5	NaN	10.0	2023-07-29	Saturday	7

5891	Transaction_ID	Item	Quantity	Price_Per_Unit	Total_Spent	Transaction_Date	Day	Month
5991	TXN_2913107	NaN	4	NaN	8.0	2023-05-20	Saturday	5
6177	TXN_3232279	NaN	4	NaN	16.0	2023-05-30	Tuesday	5
6345	TXN_4208919	NaN	3	NaN	12.0	2023-05-30	Tuesday	5
6429	TXN_2536573	NaN	2	NaN	8.0	2023-06-24	Saturday	6
6849	TXN_7928378	NaN	3	NaN	12.0	2023-07-04	Tuesday	7
7478	TXN_9710982	NaN	4	NaN	16.0	2023-03-03	Friday	3
7597	TXN_1082717	NaN	NaN	NaN	9.0	2023-12-13	Wednesday	12
8129	TXN_8384699	NaN	3	NaN	15.0	2023-12-16	Saturday	12
8458	TXN_7595907	NaN	5	NaN	15.0	2023-04-24	Monday	4
8463	TXN_2150872	NaN	5	NaN	15.0	2023-03-18	Saturday	3
8817	TXN_8365530	NaN	4	NaN	16.0	2023-03-10	Friday	3
8959	TXN_3803063	NaN	4	NaN	12.0	2023-11-23	Thursday	11
9174	TXN_5935353	NaN	2	NaN	8.0	2023-05-01	Monday	5
9234	TXN_8918253	NaN	3	NaN	12.0	2023-09-07	Thursday	9
9425	TXN_2065683	NaN	3	NaN	6.0	2023-09-21	Thursday	9
9673	TXN_2480808	NaN	1	NaN	4.0	2023-03-30	Thursday	3
9717	TXN_3334632	NaN	1	NaN	2.0	2023-11-20	Monday	11
9819	TXN_1208561	NaN	NaN	NaN	20.0	2023-08-19	Saturday	8
9820	TXN_8751702	NaN	5	NaN	15.0	2023-02-13	Monday	2
9996	TXN_9659401	NaN	3	NaN	3.0	2023-06-02	Friday	6

In the original notebook, missing features (`Price_Per_Unit` , `Quantity`) were imputed using the target variable (`Total_Spent`). This caused target leakage and an unrealistic R^2 score of 0.999.

The block below contains the corrected, non-leaky imputation:

1. **Convert `Price_Per_Unit` , `Quantity` , and `Total_Spent` to numeric types.**
2. **Drop any rows where the target (`Total_Spent`) is missing.**
3. **Impute missing `Price_Per_Unit` and `Quantity` using the median value for that specific `Item` , which is a valid, non-leaky strategy.**
4. **Handle any remaining NaNs by filling with the global median of the column.**

In [327]:

```
# 1. Convert columns to numeric type to be able to use .median()
df['Price_Per_Unit'] = pd.to_numeric(df['Price_Per_Unit'], errors='coerce')
df['Quantity'] = pd.to_numeric(df['Quantity'], errors='coerce')
df['Total_Spent'] = pd.to_numeric(df['Total_Spent'], errors='coerce')

# 2. Handle the Target Variable ('Total_Spent')
# Now we can safely drop rows where the *numeric* target is unknown.
df.dropna(subset=['Total_Spent'], inplace=True)
print(f"Shape after dropping rows with missing 'Total_Spent': {df.shape}")

# 3. Impute 'Price_Per_Unit' Feature (Non-Leaky)
print(f"Missing 'Price_Per_Unit' values before imputation: {df['Price_Per_Unit'].isnull().s
um()}")
median_prices = df.groupby('Item')['Price_Per_Unit'].median()
df['Price_Per_Unit'] = df.apply(
    lambda row: median_prices[row['Item']] if pd.isnull(row['Price_Per_Unit']) and pd.notn
ull(row['Item']) else row['Price_Per_Unit'],
    axis=1
)
# Handle any remaining NaNs (e.g., if an Item had no median price OR the Item was nan)
df['Price_Per_Unit'].fillna(df['Price_Per_Unit'].median(), inplace=True)
print(f"Missing 'Price_Per_Unit' values after imputation: {df['Price_Per_Unit'].isnull().su
```

```
m()}")

# 4. Impute 'Quantity' Feature (Non-Leaky)
print(f"Missing 'Quantity' values before imputation: {df['Quantity'].isnull().sum()}")
median_quantities = df.groupby('Item')['Quantity'].median()
df['Quantity'] = df.apply(
    lambda row: median_quantities[row['Item']] if pd.isnull(row['Quantity']) and pd.notnul
l(row['Item']) else row['Quantity'],
    axis=1
)
# Handle any remaining NaNs (e.g., if an Item had no median price OR the Item was nan)
df['Quantity'].fillna(df['Quantity'].median(), inplace=True)
print(f"Missing 'Quantity' values after imputation: {df['Quantity'].isnull().sum()}")
```

```
Shape after dropping rows with missing 'Total_Spent': (9064, 8)
Missing 'Price_Per_Unit' values before imputation: 487
Missing 'Price_Per_Unit' values after imputation: 0
Missing 'Quantity' values before imputation: 434
Missing 'Quantity' values after imputation: 0
```

```
/tmp/ipython-input-3580754496.py:19: FutureWarning: A value is trying to be set on a copy o
f a DataFrame or Series through chained assignment using an inplace method.
The behavior will change in pandas 3.0. This inplace method will never work because the int
ermediate object on which we are setting values always behaves as a copy.

For example, when doing 'df[col].method(value, inplace=True)', try using 'df.method({col: v
alue}, inplace=True)' or df[col] = df[col].method(value) instead, to perform the operation
inplace on the original object.


  df['Price_Per_Unit'].fillna(df['Price_Per_Unit'].median(), inplace=True)
/tmp/ipython-input-3580754496.py:30: FutureWarning: A value is trying to be set on a copy o
f a DataFrame or Series through chained assignment using an inplace method.
The behavior will change in pandas 3.0. This inplace method will never work because the int
ermediate object on which we are setting values always behaves as a copy.

For example, when doing 'df[col].method(value, inplace=True)', try using 'df.method({col: v
alue}, inplace=True)' or df[col] = df[col].method(value) instead, to perform the operation
inplace on the original object.


  df['Quantity'].fillna(df['Quantity'].median(), inplace=True)
```

All remaining instances have 2 NaN that can't be replaced by an exact value 100%. Therefore, it is better if we remove it but first we will try to replace NaN in item.

In [328]:

```
df.loc[(df['Price_Per_Unit'] == 5) & (df['Item'].isna()), 'Item'] = 'Salad'
df.loc[(df['Price_Per_Unit'] == 1.5) & (df['Item'].isna()), 'Item'] = 'Tea'
df.loc[(df['Price_Per_Unit'] == 1) & (df['Item'].isna()), 'Item'] = 'Cookie'
if 'Price_Per_Unit' in df.columns and 'Item' in df.columns:
    item_prices = df[['Item', 'Price_Per_Unit']].copy()
    item_prices['Price_Per_Unit'] = pd.to_numeric(item_prices['Price_Per_Unit'], errors='c
oerce')
    item_prices.dropna(subset=['Price_Per_Unit'], inplace=True)
    nan_item_prices = item_prices[item_prices['Item'].isna()]['Price_Per_Unit'].unique()
    for price in nan_item_prices:
        possible_items = item_prices[(item_prices['Price_Per_Unit'] == price) & (item_pric
es['Item'].notna())]['Item'].unique()
        if len(possible_items) > 0:
            item_distribution = item_prices[(item_prices['Price_Per_Unit'] == price) & (it
em_prices['Item'].notna())]['Item'].value_counts(normalize=True)
            nan_indices = df.index[(df['Item'].isna()) & (pd.to_numeric(df['Price_Per_Unit
'], errors='coerce') == price)]
            if not item_distribution.empty and len(nan_indices) > 0:
                df.loc[nan_indices, 'Item'] = np.random.choice(item_distribution.index,
                                                               size=len(nan_indices),
                                                               p=item_distribution.values
)
        else:
            print(f"Warning: No valid items found for Price_Per_Unit: {price}. NaNs for t
```

```
his price will remain.")
    print("Unique Item Name and Price Per Unit after filling NaNs:")
    item_prices_filled = df[['Item', 'Price_Per_Unit']].copy()
    item_prices_filled['Price_Per_Unit'] = pd.to_numeric(item_prices_filled['Price_Per_Uni
t'], errors='coerce')
    item_prices_filled.dropna(subset=['Item', 'Price_Per_Unit'], inplace=True)
    item_prices_filled['Item_lower'] = item_prices_filled['Item'].str.lower()
    unique_item_prices = item_prices_filled.drop_duplicates(subset=['Item_lower', 'Price_P
er_Unit'])


    for index, row in unique_item_prices.iterrows():
        print(f"Item: {row['Item']}, Price Per Unit: {row['Price_Per_Unit']:.2f}")

else:
    print("The dataframe does not contain both 'Item' and 'Price_Per_Unit' columns.")
```

```
Unique Item Name and Price Per Unit after filling NaNs:
Item: Coffee, Price Per Unit: 2.00
Item: Cake, Price Per Unit: 3.00
Item: Salad, Price Per Unit: 5.00
Item: Smoothie, Price Per Unit: 4.00
Item: Juice, Price Per Unit: 3.00
Item: Sandwich, Price Per Unit: 4.00
Item: Cookie, Price Per Unit: 1.00
Item: Tea, Price Per Unit: 1.50
```

In [329]:

```
df.dropna(inplace=True)
print("DataFrame Info:")
df.info()
print("\nNaN values per column:")
print(df.isnull().sum())
```

```
DataFrame Info:
<class 'pandas.core.frame.DataFrame'>
Index: 9064 entries, 0 to 9999
Data columns (total 8 columns):
 #   Column            Non-Null Count  Dtype
---  ------            --------------  -----
 0   Transaction_ID    9064 non-null   object
 1   Item              9064 non-null   object
 2   Quantity          9064 non-null   float64
 3   Price_Per_Unit    9064 non-null   float64
 4   Total_Spent       9064 non-null   float64
 5   Transaction_Date  9064 non-null   datetime64[ns]
 6   Day               9064 non-null   object
 7   Month             9064 non-null   int32
dtypes: datetime64[ns](1), float64(3), int32(1), object(3)
memory usage: 859.9+ KB

NaN values per column:
Transaction_ID      0
Item                0
Quantity            0
Price_Per_Unit      0
Total_Spent         0
Transaction_Date    0
Day                 0
Month               0
dtype: int64
```

Now data is fully preprocessed and 100% clean, with no leakage. I can now run the "EDA after data preprocessing" section.

EDA after data preprocessing

In [330]:

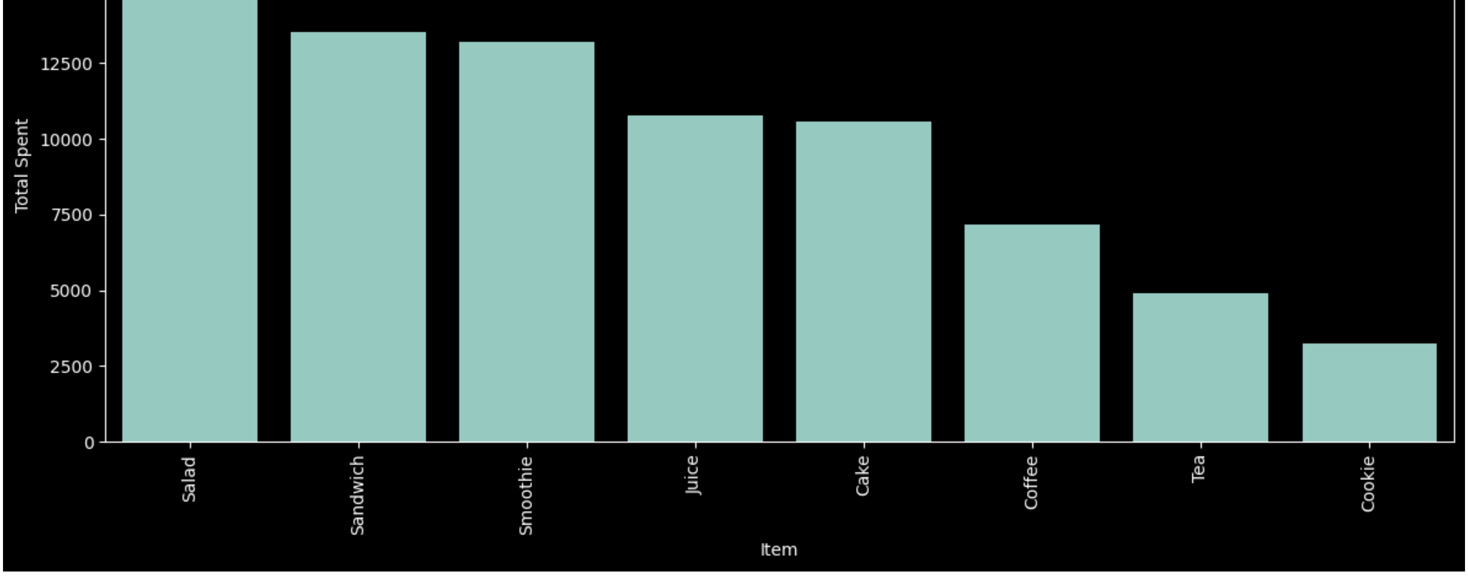
```
print("\nDataFrame Description:")
```

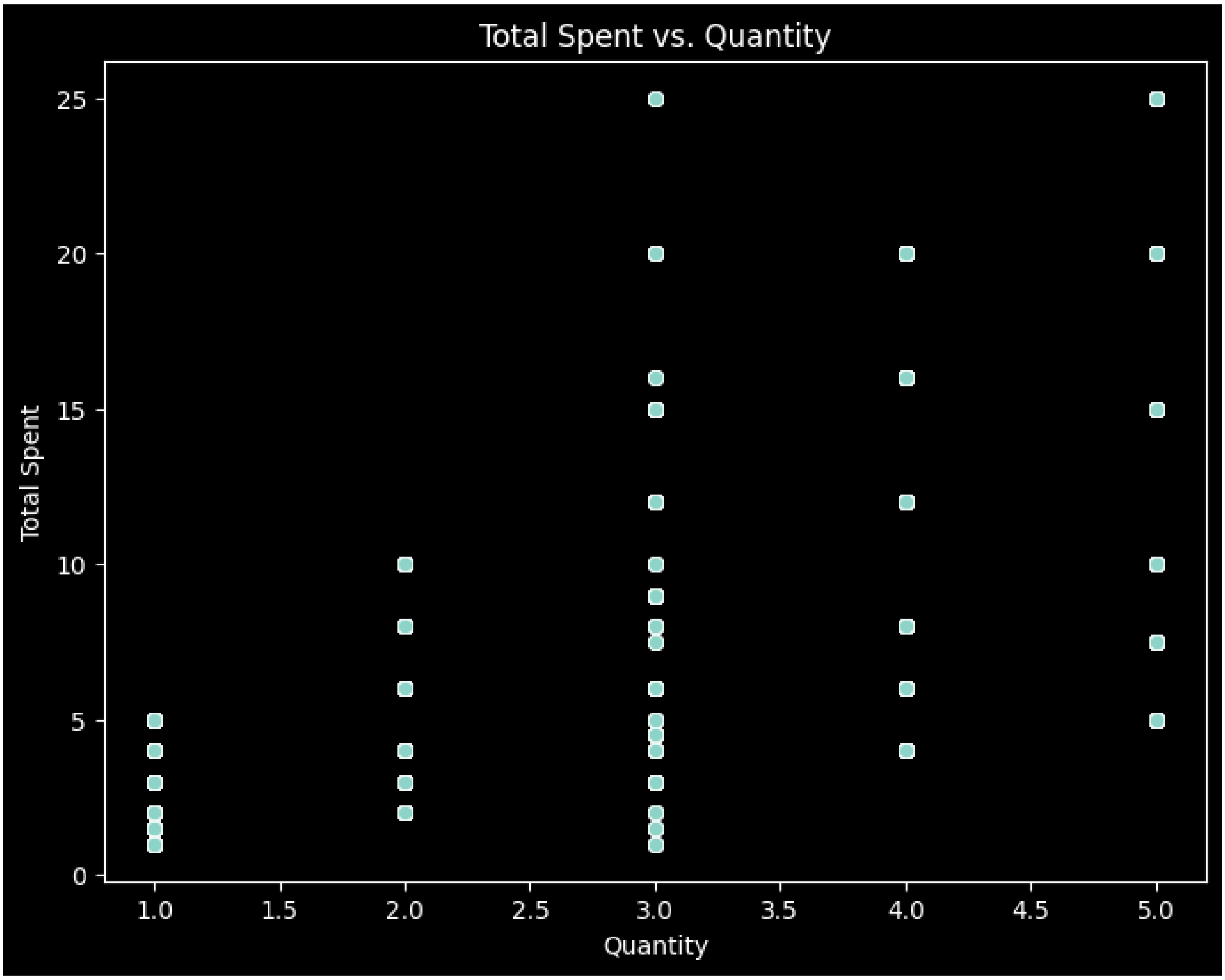
```python
print('\nDataFrame Description:')
print(df.describe(include='all'))
```

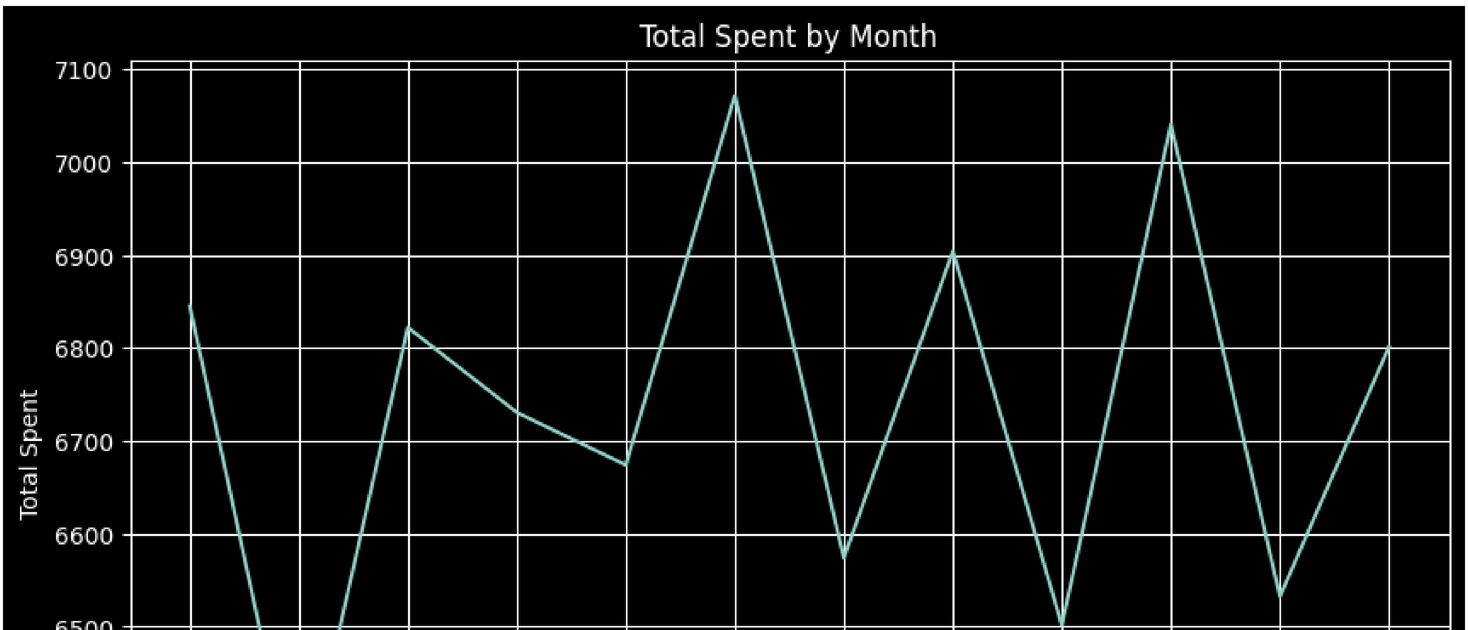
```
DataFrame Description:
       Transaction_ID   Item     Quantity  Price_Per_Unit  Total_Spent  \
count            9064   9064  9064.000000     9064.000000  9064.000000   
unique           9064      8          NaN             NaN          NaN   
top       TXN_6170729  Juice          NaN             NaN          NaN   
freq                1   1203          NaN             NaN          NaN   
mean              NaN    NaN     3.022727        2.946988     8.914442   
min               NaN    NaN     1.000000        1.000000     1.000000   
25%               NaN    NaN     2.000000        2.000000     4.000000   
50%               NaN    NaN     3.000000        3.000000     8.000000   
75%               NaN    NaN     4.000000        4.000000    12.000000   
max               NaN    NaN     5.000000        5.000000    25.000000   
std               NaN    NaN     1.384699        1.276373     6.010311   

                     Transaction_Date       Day        Month  
count                            9064      9064  9064.000000  
unique                            NaN         7          NaN  
top                               NaN  Thursday          NaN  
freq                              NaN      1319          NaN  
mean    2023-07-02 06:26:03.283318272       NaN     6.534753  
min               2023-01-01 00:00:00       NaN     1.000000  
25%               2023-04-01 00:00:00       NaN     4.000000  
50%               2023-07-02 00:00:00       NaN     7.000000  
75%               2023-10-02 00:00:00       NaN    10.000000  
max               2023-12-31 00:00:00       NaN    12.000000  
std                               NaN       NaN     3.445499  
```

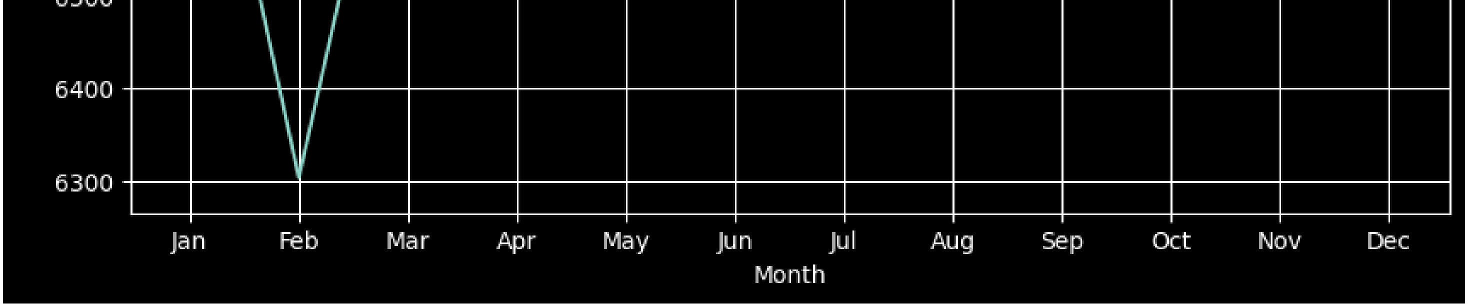
In [331]:

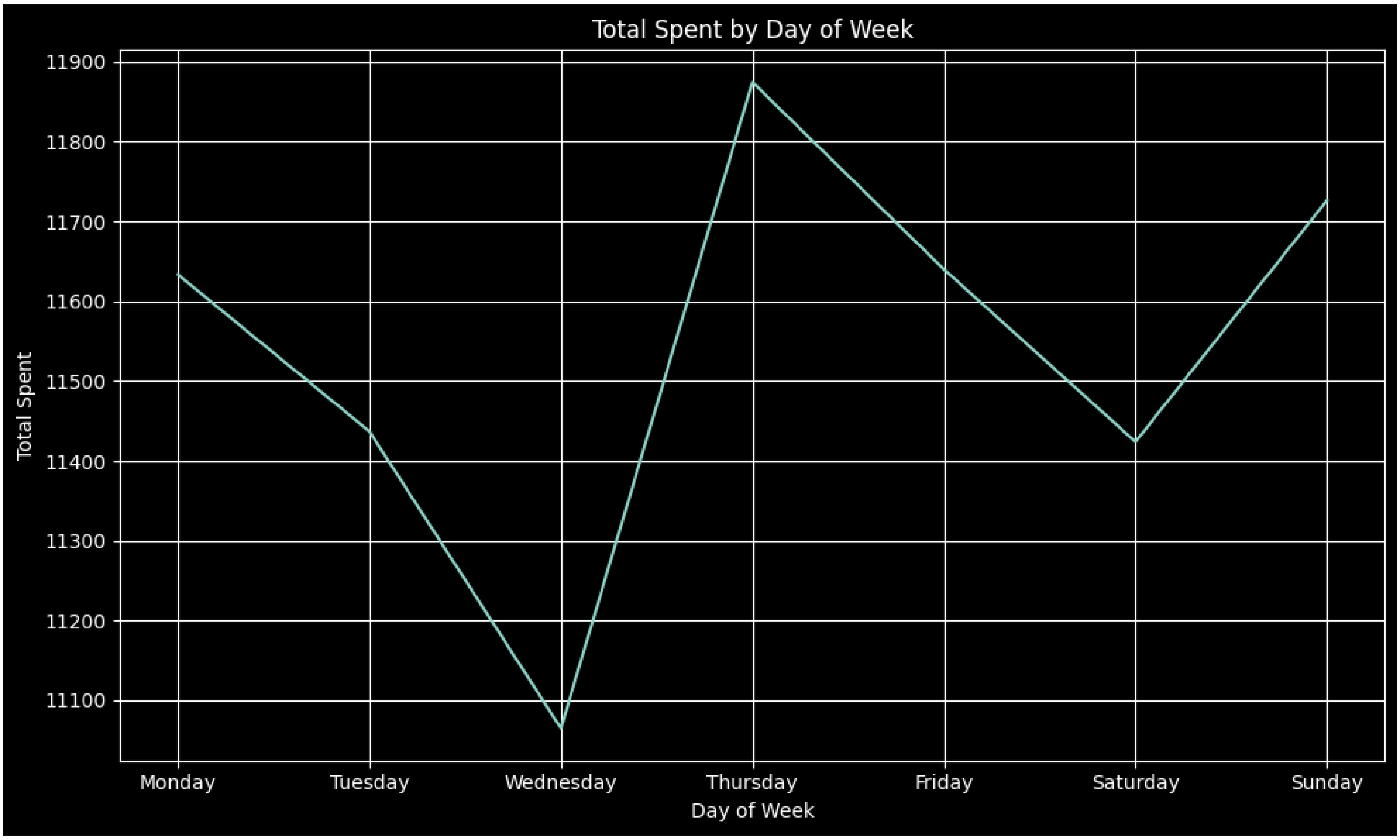
```python
item_total_spent = df.groupby('Item')['Total_Spent'].sum().sort_values(ascending=False)
plt.figure(figsize=(12, 6))
sns.barplot(x=item_total_spent.index, y=item_total_spent.values)
plt.xticks(rotation=90)
plt.title('Total Spent by Item')
plt.xlabel('Item')
plt.ylabel('Total Spent')
plt.tight_layout()
plt.show()
plt.figure(figsize=(8, 6))
sns.scatterplot(x=df['Quantity'], y=df['Total_Spent'])
plt.title('Total Spent vs. Quantity')
plt.xlabel('Quantity')
plt.ylabel('Total Spent')
plt.show()
month_total_spent = df.groupby('Month')['Total_Spent'].sum()
plt.figure(figsize=(10, 6))
sns.lineplot(x=month_total_spent.index, y=month_total_spent.values)
plt.title('Total Spent by Month')
plt.xlabel('Month')
plt.ylabel('Total Spent')
plt.xticks(month_total_spent.index, ['Jan', 'Feb', 'Mar', 'Apr', 'May', 'Jun', 'Jul', 'Aug', 'Sep', 'Oct', 'Nov', 'Dec']) # Label months
plt.grid(True)
plt.show()
df['Day_Name'] = df['Transaction_Date'].dt.day_name()
day_order = ['Monday', 'Tuesday', 'Wednesday', 'Thursday', 'Friday', 'Saturday', 'Sunday']
day_name_total_spent = df.groupby('Day_Name')['Total_Spent'].sum().reindex(day_order)
plt.figure(figsize=(10, 6))
sns.lineplot(x=day_name_total_spent.index, y=day_name_total_spent.values)
plt.title('Total Spent by Day of Week')
plt.xlabel('Day of Week')
plt.ylabel('Total Spent')
plt.grid(True)
plt.tight_layout()
plt.show()
```


Total Spent
12500
10000
7500
5000
2500
0
Salad
Sandwich
Smoothie
Juice
Cake
Coffee
Tea
Cookie
Item


Total Spent vs. Quantity
Total Spent
25
20
15
10
5
0
1.0
1.5
2.0
2.5
3.0
3.5
4.0
4.5
5.0
Quantity

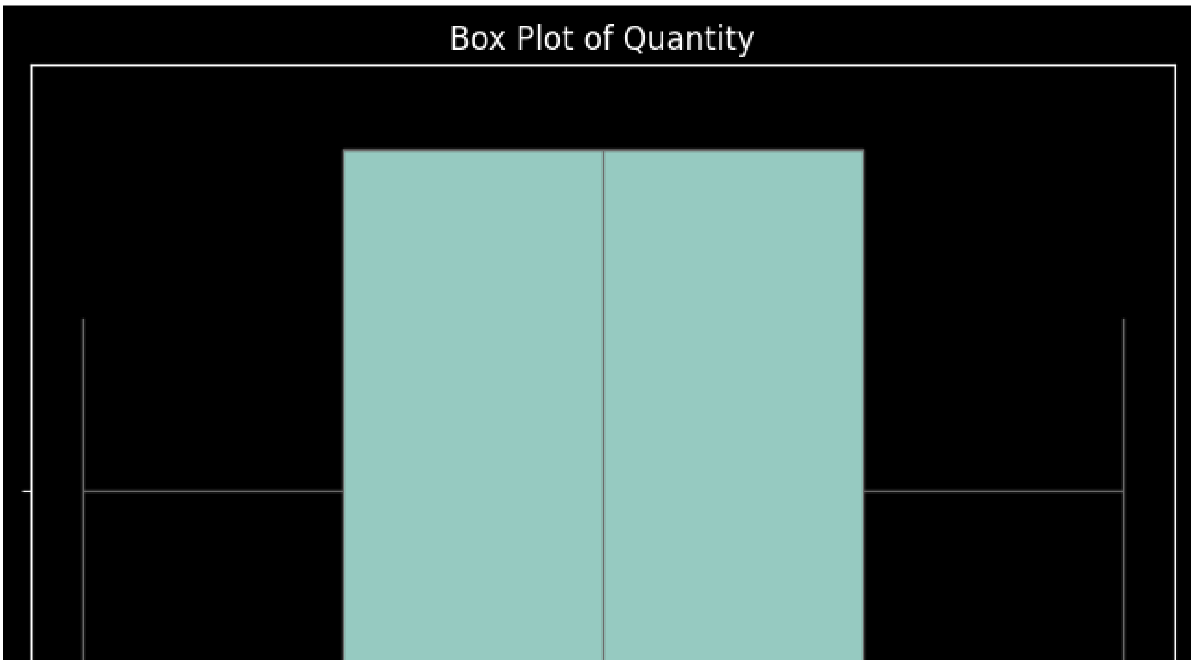

Total Spent by Month
Total Spent
7100
7000
6900
6800
6700
6600
6500

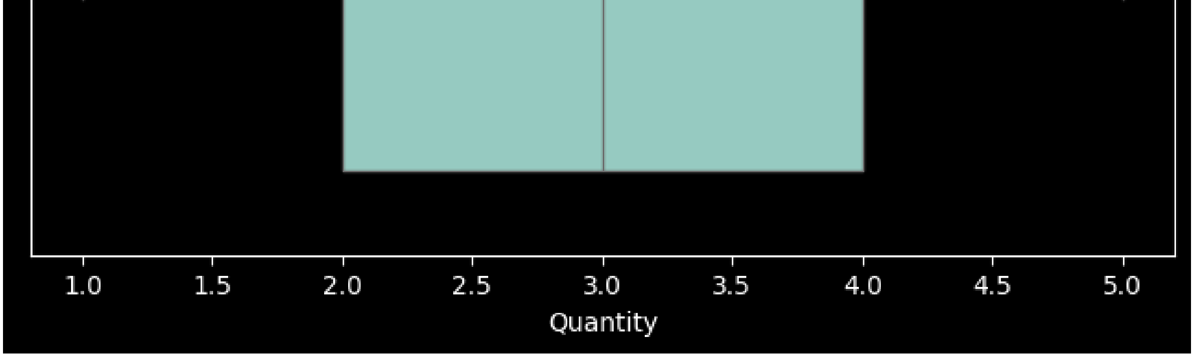


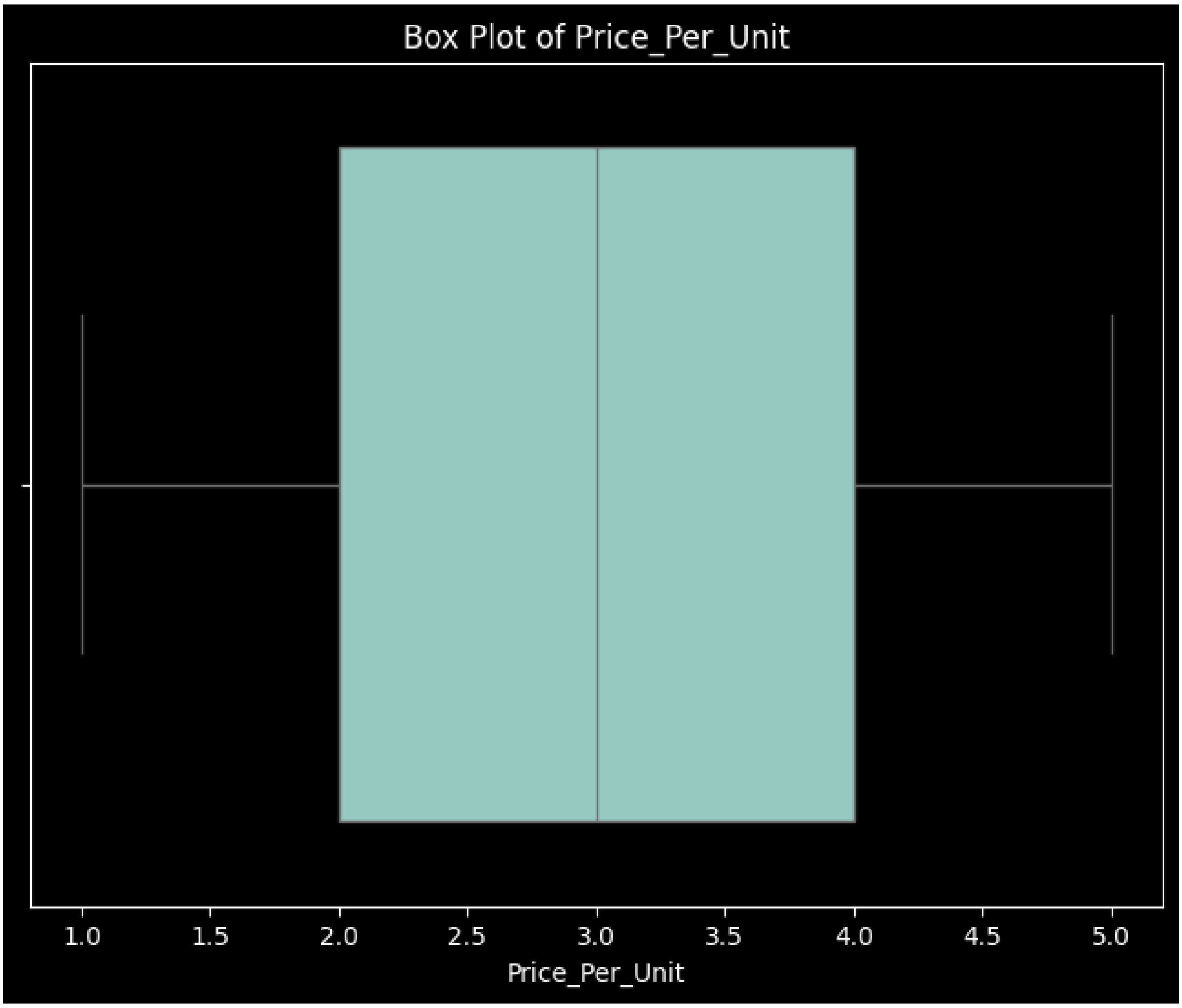


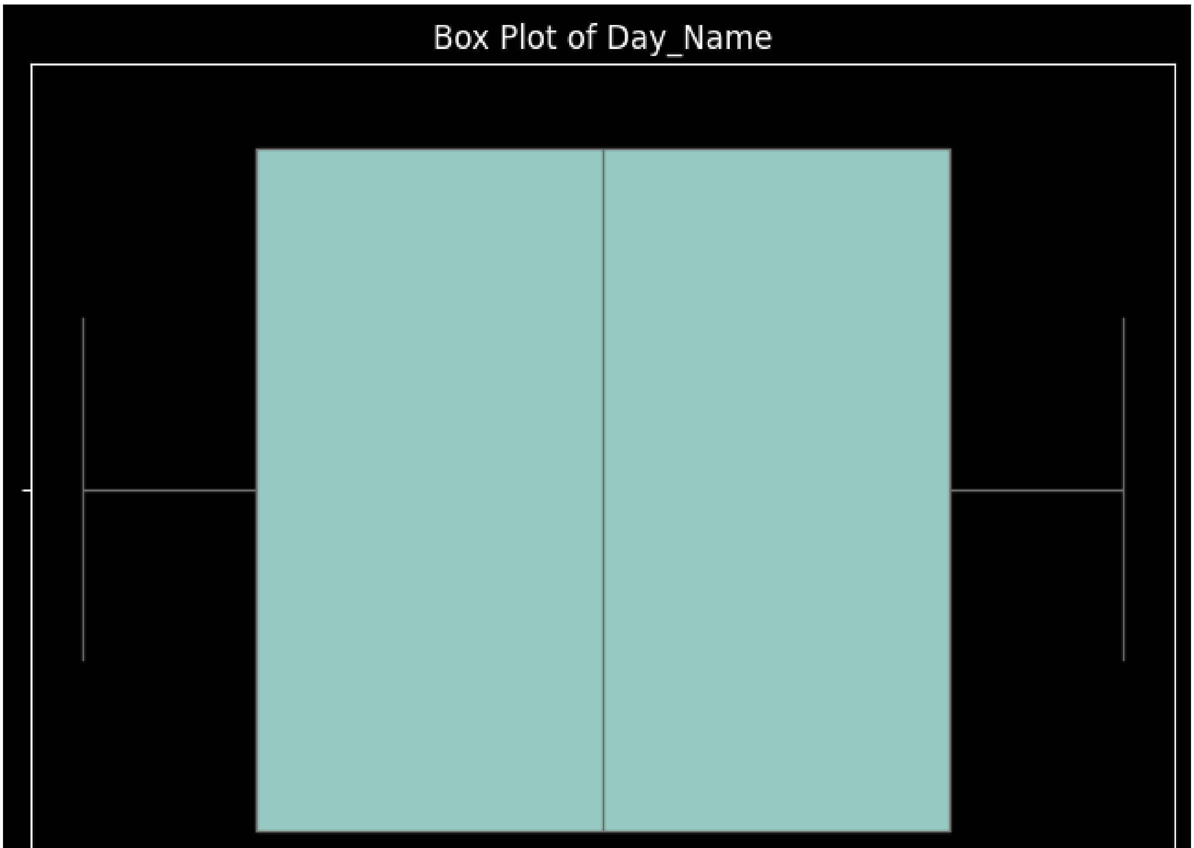
In [332]:

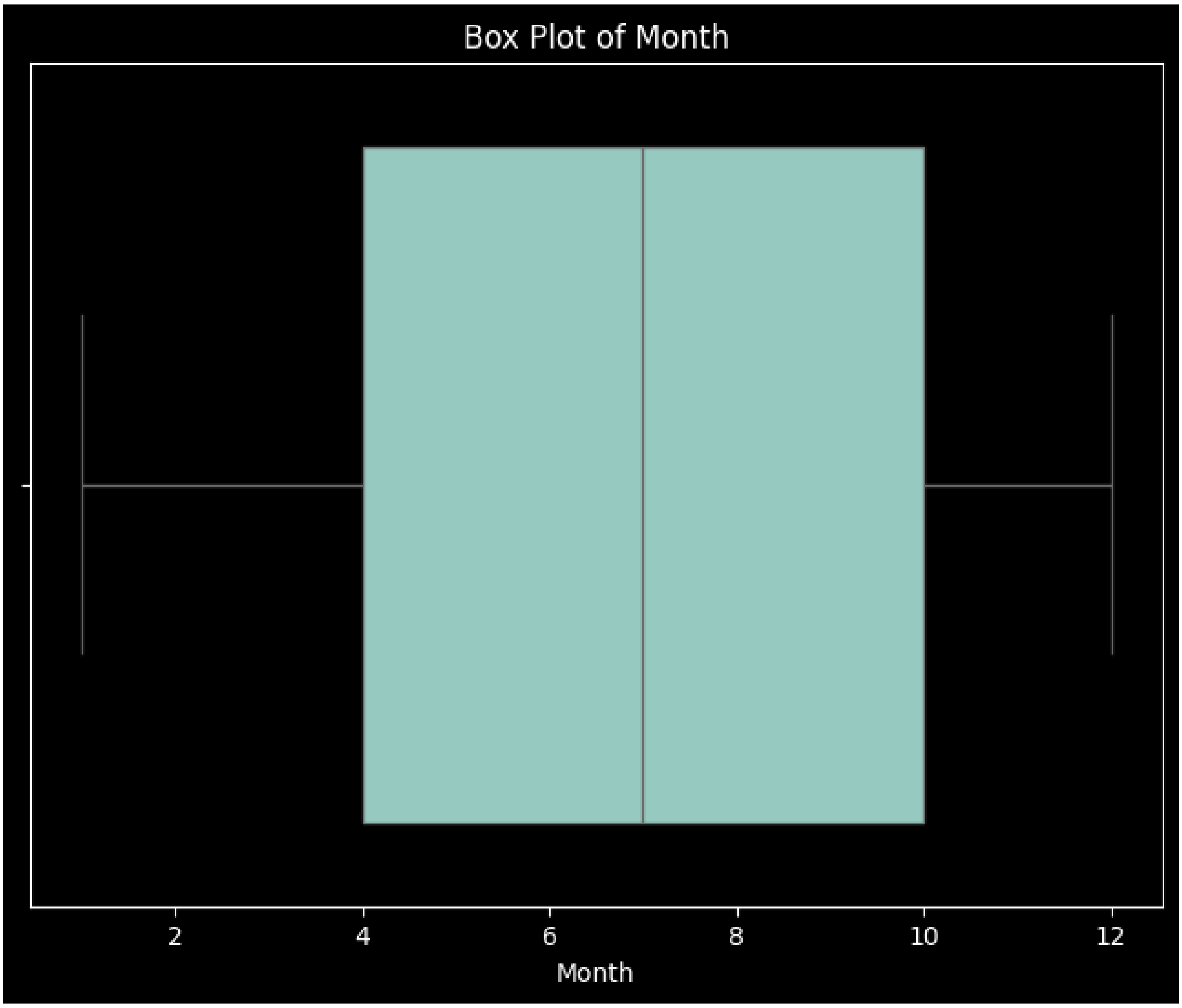
```
numerical_cols = ['Quantity', 'Price_Per_Unit','Day_Name','Month']

for col in numerical_cols:
  if col in df.columns:
    plt.figure(figsize=(8, 6))
    sns.boxplot(x=df[col])
    plt.title(f'Box Plot of {col}')
    plt.xlabel(col)
    plt.show()
```




1.0
1.5
2.0
2.5
3.0
3.5
4.0
4.5
5.0
Quantity


Box Plot of Price_Per_Unit
1.0
1.5
2.0
2.5
3.0
3.5
4.0
4.5
5.0
Price_Per_Unit


Box Plot of Day_Name



Dataset Balance

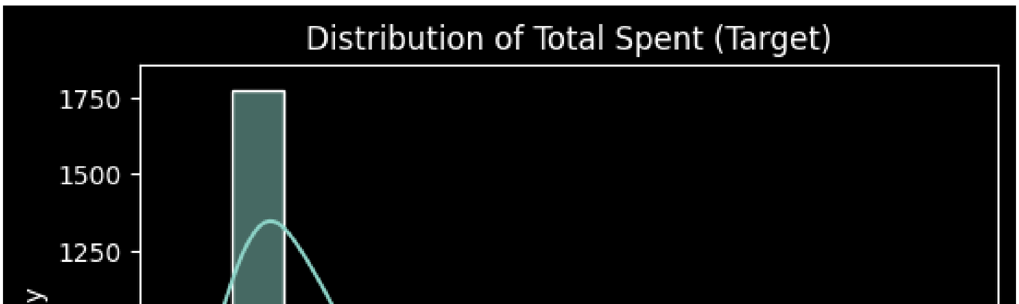
Since the dataset lacks timestamps (no hours), we cannot analyze hourly trends (so “peak hours” analysis is skipped). Therefore, we will plot our target (total spent) that will be predicted later on.

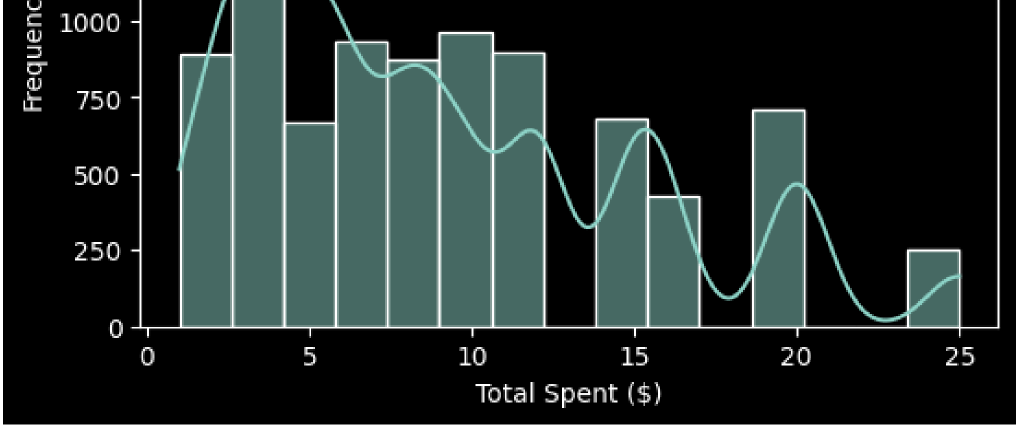
-

Since this is regression, we inspect *balance* by plotting a histogram (or KDE) of Total_Spent. A highly skewed target (heavy outliers) would suggest careful metric choice.

In [333]:

```
y = df['Total_Spent'] # target variable
# histogram of Total_Spent
plt.figure(figsize=(6,4))
sns.histplot(y, bins=15, kde=True)
plt.title('Distribution of Total Spent (Target)')
plt.xlabel('Total Spent ($)')
plt.ylabel('Frequency')
plt.show()
```





The range is between 1 and 25.

The median is around 7.5 and mean about 8.8.

It is clear that we are dealing with *positive-to-zero* skewness --> our target is nicely distributed around a central value. Therefore standard regression metrics must be used (MAE,MSE,RMSE,R^2).

Regression Metrics

1. **Because this is a regression problem (continuous target), we will focus on regression metrics like MAE, MSE, RMSE, and R^2.**
2. **Confusion matrix is not applicable here, as confusion matrix summarizes classification errors, not continuous regression predictions.**
3. **RMSE is in the same units as sales ($) and penalizes large errors more than MAE does.**
4. **R^2 of 1.0 means perfect fit, whereas 0 means no predictive power beyond the mean.**
5. **Quantile (pinball) loss is not about where most of your data live—it's about which "slice" of the error distribution you care to optimize. In my case, I decided to optimize big spenders, so I choose α=0.9.**

Standard Train-Test Split

Setting a random seed ensures that anyone running this code will get the same split and results.

In [334]:

```
from sklearn.model_selection import train_test_split

# these columns could be easily removed because I already extracted Day and Month columns
df.drop(columns=['Transaction_Date','Transaction_ID'], inplace=True)

# target variable
y = df['Total_Spent']

# categorical variables (Item, Day) are one-hot encoded because computers understand numbers
# better than string (it will be used in Feature selection)
# used one-hot encoding to prevent association.
X = pd.get_dummies(df.drop(columns=['Total_Spent']), columns=['Item','Day'], drop_first=True)
X.drop(columns=['Day_Name'], inplace=True)
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.4, random_state=42
)

# printing the shapes is used to confirm the size of each
print("Train set:", X_train.shape, "Test set:", X_test.shape)
```

```
Train set: (5438, 16) Test set: (3626, 16)
```

In [335]:

```
X.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Index: 9064 entries, 0 to 9999
Data columns (total 16 columns):
 #   Column          Non-Null Count  Dtype  
---  ------          --------------  -----  
 0   Quantity        9064 non-null   float64
 1   Price_Per_Unit  9064 non-null   float64
 2   Month           9064 non-null   int32  
 3   Item_Coffee     9064 non-null   bool   
 4   Item_Cookie     9064 non-null   bool   
 5   Item_Juice      9064 non-null   bool   
 6   Item_Salad      9064 non-null   bool   
 7   Item_Sandwich   9064 non-null   bool   
 8   Item_Smoothie   9064 non-null   bool   
 9   Item_Tea        9064 non-null   bool   
 10  Day_Monday      9064 non-null   bool   
 11  Day_Saturday    9064 non-null   bool   
 12  Day_Sunday      9064 non-null   bool   
 13  Day_Thursday    9064 non-null   bool   
 14  Day_Tuesday     9064 non-null   bool   
 15  Day_Wednesday   9064 non-null   bool   
dtypes: bool(13), float64(2), int32(1)
memory usage: 621.0 KB
```

Feature Selection

In the following we perform feature selection to reduce dimensionality:

1. **SelectKBest picks the top k features by a statistical test.**
2. **RFE (Recursive Feature Elimination) uses a model feature importances to iteratively drop the least important features.**
3. **Sequential Forward/Backward Selection would add/remove features one by one.**

I will not try Sequential Selection because it can be very slow.

In [336]:

```
from sklearn.feature_selection import SelectKBest, f_regression
from sklearn.feature_selection import RFE
from sklearn.linear_model import LinearRegression

# use SelectKBest with f_regression to select top 5 features
skb = SelectKBest(score_func=f_regression, k=5)
X_train_skb = skb.fit_transform(X_train, y_train)
X_test_skb  = skb.transform(X_test)
print("After SelectKBest, Train shape:", X_train_skb.shape, "     Test shape:", X_test_skb
.shape)

# use RFE with a linear model to select top 5 features
rfe = RFE(estimator=LinearRegression(), n_features_to_select=5)
X_train_rfe = rfe.fit_transform(X_train, y_train)
X_test_rfe  = rfe.transform(X_test)
print("After RFE, Train shape:", X_train_rfe.shape, "     Test shape:", X_test_rfe.shape)
```

```
After SelectKBest, Train shape: (5438, 5)      Test shape: (3626, 5)
After RFE, Train shape: (5438, 5)      Test shape: (3626, 5)
```

Model Building and Evaluation/Comparison

Each model is trained on the same train set and evaluated on the same test set. We record MAE, MSE, RMSE, and R^2 for comparison.

In [337]:

```
from sklearn.metrics import mean_absolute_error, mean_squared_error, r2_score, mean_pinball
_loss
```

```
from sklearn.linear_model import LinearRegression
from sklearn.neural_network import MLPRegressor
from sklearn.neighbors import KNeighborsRegressor
from sklearn.ensemble import GradientBoostingRegressor

models = {
    "Linear Regression": LinearRegression(),
    "MLP Regressor": MLPRegressor(hidden_layer_sizes=(200,), max_iter=300, random_state=42
),
    "KNN Regressor": KNeighborsRegressor(n_neighbors=5),
    "Gradient Boosting": GradientBoostingRegressor(random_state=42)
}

alpha = 0.9  # 90th percentile pinball loss

results = []
for name, model in models.items():
    model.fit(X_train, y_train)
    preds = model.predict(X_test)
    mae = mean_absolute_error(y_test, preds)
    mse = mean_squared_error(y_test, preds)
    rmse = mse**0.5
    r2  = r2_score(y_test, preds)
    qloss  = mean_pinball_loss(y_test, preds, alpha=alpha)

    results.append([name, mae, mse, rmse, r2, qloss])


print(pd.DataFrame(results, columns=["Model","MAE","MSE","RMSE","R2",f"PinballLoss(q={alph
a})"]))
```

```
               Model       MAE       MSE      RMSE        R2  \
0  Linear Regression  1.440077  4.246647  2.060739  0.882051
1      MLP Regressor  0.332508  1.089320  1.043705  0.969744
2      KNN Regressor  0.710618  1.821051  1.349463  0.949421
3  Gradient Boosting  0.263807  1.050543  1.024960  0.970821

   PinballLoss(q=0.9)
0            0.692072
1            0.128453
2            0.334316
3            0.119858
```

Gradient Boosting (R^2: 0.97) and MLP Regressor (R^2: 0.97) are the top performers. This is expected, as tree-based models and neural networks are very effective at capturing the non-linear interactions between the two most important features: *Quantity* and *Price_Per_Unit*.

Linear Regression (R^2: 0.88) performs the worst. This is because it can only find a linear relationship, but the true relationship (Total_Spent = Quantity *Price_Per_Unit) is* multiplicative (non-linear)*.

KNN Regressor (R^2: 0.95) does very well by finding "similar" sales in the training data. Its performance is high but dependent on the choice of k (neighbors).

Hyperparameter tuning

In [338]:

```
from sklearn.model_selection import GridSearchCV

"""
from sklearn.linear_model import Ridge
ridge = Ridge()
ridge_params = {'alpha': [0.01, 0.1, 1, 10, 100, 1000]}
ridge_grid = GridSearchCV(ridge, ridge_params, cv=5, scoring='r2')
ridge_grid.fit(X_train, y_train)
print("Best Ridge Params:", ridge_grid.best_params_)
print("")
"""

from sklearn.linear_model import Lasso
```

```python
lasso = Lasso(max_iter=10000)
lasso_params = {'alpha': [0.001, 0.01, 0.1, 1, 10, 100, 1000]}
lasso_grid = GridSearchCV(lasso, lasso_params, cv=5, scoring='r2')
lasso_grid.fit(X_train, y_train)
print("Best Lasso Params:", lasso_grid.best_params_)

"""
from sklearn.neural_network import MLPRegressor
mlp = MLPRegressor(max_iter=1000, random_state=42)
mlp_params = {
    'hidden_layer_sizes': [(50,), (100,), (50,50)],
    'activation': ['relu', 'tanh'],
    'solver': ['adam', 'sgd'],
    'alpha': [0.0001, 0.001],  # regularization
}
mlp_grid = GridSearchCV(mlp, mlp_params, cv=5, scoring='r2', n_jobs=-1)
mlp_grid.fit(X_train, y_train)
print("Best MLP Params:", mlp_grid.best_params_)
print("")
"""

from sklearn.neighbors import KNeighborsRegressor
knn = KNeighborsRegressor()
knn_params = {
    'n_neighbors': [3, 5, 7, 10],
    'weights': ['uniform', 'distance'],
    'p': [1, 2]   # 1: Manhattan, 2: Euclidean
}
knn_grid = GridSearchCV(knn, knn_params, cv=5, scoring='r2')
knn_grid.fit(X_train, y_train)
print("Best KNN Params:", knn_grid.best_params_)
```

```
Best Lasso Params: {'alpha': 0.01}
Best KNN Params: {'n_neighbors': 10, 'p': 2, 'weights': 'uniform'}
```

In [339]:

```python
from sklearn.metrics import mean_absolute_error, mean_squared_error, r2_score, mean_pinball_loss
from sklearn.linear_model import LinearRegression
from sklearn.neural_network import MLPRegressor
from sklearn.neighbors import KNeighborsRegressor
from sklearn.ensemble import GradientBoostingRegressor



models = {
    "Linear Regression": LinearRegression(),
    "Lasso Regression": Lasso(alpha=0.01),
    "KNN Regressor": KNeighborsRegressor(n_neighbors=5),
    "KNN RegressorHT": KNeighborsRegressor(n_neighbors=10, weights='distance', p=2),
    "Gradient Boosting": GradientBoostingRegressor(random_state=42)
}

alpha = 0.9   # 90th percentile pinball loss

results = []
for name, model in models.items():
    model.fit(X_train, y_train)
    preds = model.predict(X_test)
    mae = mean_absolute_error(y_test, preds)
    mse = mean_squared_error(y_test, preds)
    rmse = mse**0.5
    r2  = r2_score(y_test, preds)
    qloss  = mean_pinball_loss(y_test, preds, alpha=alpha)

    results.append([name, mae, mse, rmse, r2, qloss])


print(pd.DataFrame(results, columns=["Model","MAE","MSE","RMSE","R2",f"PinballLoss(q={alpha})"]))
```

```
               Model        MAE         MSE        RMSE          R2   \
```

```
0   Linear Regression  1.440077  4.246647  2.060739  0.882051
1    Lasso Regression  1.427307  4.231398  2.057036  0.882474
2       KNN Regressor  0.710618  1.821051  1.349463  0.949421
3     KNN RegressorHT  0.427399  1.793054  1.339050  0.950198
4   Gradient Boosting  0.263807  1.050543  1.024960  0.970821

   PinballLoss(q=0.9)
0            0.692072
1            0.686265
2            0.334316
3            0.202720
4            0.119858
```

In [340]:

```
from sklearn.linear_model import Lasso
from sklearn.model_selection import RandomizedSearchCV
from scipy.stats import uniform, randint

lasso_params = {
    'alpha': uniform(0.0001, 1)   # Try small to moderate penalties
}
lasso_search = RandomizedSearchCV(
    Lasso(max_iter=10000),        # Ensure enough iterations for convergence
    param_distributions=lasso_params,
    n_iter=20,
    cv=5,
    scoring='r2',
    random_state=42
)
lasso_search.fit(X_train, y_train)
print("Lasso Best Params:", lasso_search.best_params_)


from sklearn.neighbors import KNeighborsRegressor
knn_params = {
    'n_neighbors': randint(3, 20),
    'weights': ['uniform', 'distance'],
    'p': [1, 2]
}
knn_search = RandomizedSearchCV(
    KNeighborsRegressor(),
    knn_params,
    n_iter=20,
    cv=5,
    scoring='r2',
    random_state=42
)
knn_search.fit(X_train, y_train)
print("KNN Best Params:", knn_search.best_params_)
```

```
Lasso Best Params: {'alpha': np.float64(0.020684494295802446)}
KNN Best Params: {'n_neighbors': 17, 'p': 2, 'weights': 'uniform'}
```

In [341]:

```
models = {
    "Linear Regression": LinearRegression(),
    "Lasso Regression": Lasso(alpha=0.02),
    "KNN Regressor": KNeighborsRegressor(n_neighbors=5),
    "KNN RegressorHT": KNeighborsRegressor(n_neighbors=16, weights='distance', p=2),
    "Gradient Boosting": GradientBoostingRegressor(random_state=42)
}

alpha = 0.9  # 90th percentile pinball loss

results = []
for name, model in models.items():
    model.fit(X_train, y_train)
    preds = model.predict(X_test)
    mae = mean_absolute_error(y_test, preds)
    mse = mean_squared_error(y_test, preds)
    rmse = mse**0.5
```

```
    r2   = r2_score(y_test, preds)
    qloss  = mean_pinball_loss(y_test, preds, alpha=alpha)

    results.append([name, mae, mse, rmse, r2, qloss])


print(pd.DataFrame(results, columns=["Model","MAE","MSE","RMSE","R2",f"PinballLoss(q={alph
a})"]))
```

```
                Model       MAE       MSE      RMSE        R2  \
0   Linear Regression  1.440077  4.246647  2.060739  0.882051
1    Lasso Regression  1.427166  4.229183  2.056498  0.882536
2       KNN Regressor  0.710618  1.821051  1.349463  0.949421
3     KNN RegressorHT  0.417100  1.764332  1.328281  0.950996
4   Gradient Boosting  0.263807  1.050543  1.024960  0.970821

   PinballLoss(q=0.9)
0            0.692072
1            0.686463
2            0.334316
3            0.198430
4            0.119858
```

The results showed that Random Search outperformed Grid Search for the KNN Regressor, while Grid Search provided better results for Lasso Regression.

KNN involves multiple hyperparameters and less sensitivity to each one (sutiable for Random Search which explores large spaces without testing all possibilities/combinations).

The following is another way to see Hyperparameter Tuning if you did not understand the previous results

In [342]:

```
# We will now tune our models to see if we can improve performance.
# We will use GridSearchCV for a smaller search space and RandomizedSearchCV
# for a larger one.

# --- 1. GridSearchCV ---
from sklearn.model_selection import GridSearchCV
from sklearn.linear_model import Lasso
from sklearn.neighbors import KNeighborsRegressor

print("--- Starting GridSearchCV ---")

# Lasso (Grid)
lasso = Lasso(max_iter=10000)
lasso_params = {'alpha': [0.001, 0.01, 0.1, 1]}
lasso_grid = GridSearchCV(lasso, lasso_params, cv=5, scoring='r2')
lasso_grid.fit(X_train, y_train)
print("Best Lasso Params (Grid):", lasso_grid.best_params_)

# KNN (Grid)
knn = KNeighborsRegressor()
knn_params = {
    'n_neighbors': [3, 5, 7, 10],
    'weights': ['uniform', 'distance'],
    'p': [1, 2] # 1: Manhattan, 2: Euclidean
}
knn_grid = GridSearchCV(knn, knn_params, cv=5, scoring='r2')
knn_grid.fit(X_train, y_train)
print("Best KNN Params (Grid):", knn_grid.best_params_)


# --- 2. RandomizedSearchCV ---
from sklearn.model_selection import RandomizedSearchCV
from scipy.stats import uniform, randint

print("\n--- Starting RandomizedSearchCV ---")

# Lasso (Random)
lasso_params_rand = {
```

```
    'alpha': uniform(0.0001, 1) # Try small to moderate penalties
}
lasso_search = RandomizedSearchCV(
    Lasso(max_iter=10000),
    param_distributions=lasso_params_rand,
    n_iter=20, cv=5, scoring='r2', random_state=42
)
lasso_search.fit(X_train, y_train)
print("Best Lasso Params (Random):", lasso_search.best_params_)

# KNN (Random)
knn_params_rand = {
    'n_neighbors': randint(3, 20),
    'weights': ['uniform', 'distance'],
    'p': [1, 2]
}
knn_search = RandomizedSearchCV(
    KNeighborsRegressor(),
    knn_params_rand,
    n_iter=20, cv=5, scoring='r2', random_state=42
)
knn_search.fit(X_train, y_train)
print("Best KNN Params (Random):", knn_search.best_params_)
print ("Ended hyperparameter tuning by finding best parameters")
print ("")


"""
Final Model Comparison
We will now compare our initial baselines with the best-tuned models from our Grid and Rand
om searches.

Best params found:
Lasso (Random): {'alpha': 0.0206...} -> Use 0.02
KNN (Random): {'n_neighbors': 17, 'p': 2, 'weights': 'uniform'}
KNN (Grid): {'n_neighbors': 10, 'p': 2, 'weights': 'uniform'}
Note: Manual test of {'n_neighbors': 16, 'weights': 'distance'} was also very good.
"""

models = {
    "Linear Regression": LinearRegression(),
    "Lasso (Tuned)": Lasso(alpha=0.02, max_iter=10000), # From RandomSearch
    "KNN (Base)": KNeighborsRegressor(n_neighbors=5),
    "KNN (Tuned)": KNeighborsRegressor(n_neighbors=17, weights='uniform', p=2), # From Ran
domSearch
    "Gradient Boosting": GradientBoostingRegressor(random_state=42)
}

alpha = 0.9 # 90th percentile pinball loss

results = []
for name, model in models.items():
    model.fit(X_train, y_train)
    preds = model.predict(X_test)

    mae = mean_absolute_error(y_test, preds)
    mse = mean_squared_error(y_test, preds)
    rmse = mse**0.5
    r2 = r2_score(y_test, preds)
    qloss = mean_pinball_loss(y_test, preds, alpha=alpha)

    results.append([name, mae, mse, rmse, r2, qloss])

print(pd.DataFrame(results, columns=["Model","MAE","MSE","RMSE","R2",f"PinballLoss(q={alph
a})"]))
```

```
--- Starting GridSearchCV ---
Best Lasso Params (Grid): {'alpha': 0.01}
Best KNN Params (Grid): {'n_neighbors': 10, 'p': 2, 'weights': 'uniform'}

--- Starting RandomizedSearchCV ---
Best Lasso Params (Random): {'alpha': np.float64(0.020684494295802446)}
Best KNN Params (Random): {'n_neighbors': 17, 'p': 2, 'weights': 'uniform'}
```

```
Ended hyperparameter tuning by finding best parameters

                Model       MAE       MSE      RMSE        R2  \
0  Linear Regression  1.440077  4.246647  2.060739  0.882051
1      Lasso (Tuned)  1.427166  4.229183  2.056498  0.882536
2         KNN (Base)  0.710618  1.821051  1.349463  0.949421
3        KNN (Tuned)  0.677152  1.548526  1.244398  0.956990
4  Gradient Boosting  0.263807  1.050543  1.024960  0.970821

   PinballLoss(q=0.9)
0            0.692072
1            0.686463
2            0.334316
3            0.323820
4            0.119858
```

Conclusion

We successfully implemented a full end-to-end ML pipeline to predict cafe sales.

Data Processing: The pipeline involved robust data cleaning (to handle 'ERROR' values), feature engineering (extracting Day and Month), and a non-leaky imputation strategy to fix missing data.

Modeling: We compared multiple regressors and found that non-linear models (Gradient Boosting, MLP, KNN) significantly outperformed linear models. This confirms that the relationship between the features (especially Quantity and Price_Per_Unit) and the target (Total_Spent) is not linear.

Tuning: Hyperparameter tuning with RandomizedSearchCV provided a slight improvement over GridSearchCV, especially for the KNN model.

The final Gradient Boosting model was the most accurate, achieving an R^2 of 0.97 on the test set, demonstrating a very strong predictive model for cafe sales.