Project title: Food demand forecasting for a meal delivery service

Project overview

Description

This project focuses on demand forecasting for a meal delivery company operating across multiple cities, which relies on fulfillment centers for dispatching meal orders. The goal is to predict meal demand for the next 10 weeks, helping centers plan their stock of perishable raw materials and staffing needs more efficiently.

The dataset is sourced from Kaggle provides details on cities where the meal delivery service operates, category of meal, pricing, and promotions.

Key steps and analysis

1. Data reading and cleaning

- Loaded the dataset using Pandas and explored initial data characteristics.
- Addressed inconsistencies, such as swapping `base_price` and `checkout_price` values when needed to ensure logical pricing.

2. Exploratory data analysis (EDA)

- Price analysis: Analyzed distributions of base and checkout prices, uncovering pricing trends and the impact of discounts.
- Demand patterns: Visualized demand fluctuations across weeks and explored correlations with pricing and promotions.
- Category & Cuisine Analysis: Identified the most popular categories (e.g., Beverages, Rice Bowl, and Sandwich) and cuisines (e.g., Italian and Thai).

3. Feature engineering

- Created features such as month, quarter, discount ratios, and interaction terms for promotions.
- One-hot encoded categorical variables to prepare data for machine learning models.

4. Modeling approach

Baseline models

- Linear regression: Initial R^2 score of 0.3, improved to 0.5 with feature engineering.

Advanced techniques

- Ridge and lasso regression: Implemented for regularization, achieving an R^2 of around 0.59.
- Random forest: Achieved the best performance with an R^2 of 0.79 on cross-validation.
- Gradient boosting: Moderate performance with an R^2 of 0.61, requiring further tuning.

Final model

- Random forest: Chosen for its ability to capture complex relationships, achieving an R^2 of 0.809 on the test set with a Mean Absolute Error (MAE) of 75.53.

Feature importance

- Discount effects: Discounts were found to positively correlate with increased demand.

- Promotional strategies: Analyzed the impact of email and homepage promotions, both individually and combined, on order volumes.

Visual insights

- Weekly demand trends: Visualized average orders per week to compare actual versus predicted values.
- Promotion effectiveness: Bar plots highlighted the boost in orders due to email and homepage promotions.

1. Data reading

The first step is to import the necessary Python libraries to handle the dataset:

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

After downloading the dataset from Kaggle, I used the Pandas read_csv function to load it into a DataFrame:

```
df = pd.read_csv("house_price_bd.csv")
```

To get a quick understanding of the dataset, I used the head() function to see the first few rows:

The train dataset:

```
df.head()
```

output:

	id	week	center_id	meal_id	checkout_price	base_price	emailer_for_promotion	homepage_featured	num_orders
0	1379560	1	55	1885	136.83	152.29	0	0	177
1	1466964	1	55	1993	136.83	135.83	0	0	270
2	1346989	1	55	2539	134.86	135.86	0	0	189
3	1338232	1	55	2139	339.50	437.53	0	0	54
4	1448490	1	55	2631	243.50	242.50	0	0	40

```
df.info()
```

output:

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 456548 entries, 0 to 456547
Data columns (total 9 columns):
 #   Column                 Non-Null Count   Dtype  
---  ------                 --------------   -----  
 0   id                     456548 non-null  int64  
 1   week                   456548 non-null  int64  
 2   center_id              456548 non-null  int64  
 3   meal_id                456548 non-null  int64  
 4   checkout_price         456548 non-null  float64
 5   base_price             456548 non-null  float64
 6   emailer_for_promotion  456548 non-null  int64  
 7   homepage_featured      456548 non-null  int64  
 8   num_orders             456548 non-null  int64  
dtypes: float64(2), int64(7)
memory usage: 31.3 MB
```

The test dataset:

```
df_test.head()
```

output:

	id	week	center_id	meal_id	checkout_price	base_price	emailer_for_promotion	homepage_featured
0	1028232	146	55	1885	158.11	159.11	0	0
1	1127204	146	55	1993	160.11	159.11	0	0
2	1212707	146	55	2539	157.14	159.14	0	0
3	1082698	146	55	2631	162.02	162.02	0	0
4	1400926	146	55	1248	163.93	163.93	0	0

```
df_test.info()
```

output:

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 32573 entries, 0 to 32572
Data columns (total 8 columns):
 #   Column                 Non-Null Count  Dtype  
---  ------                 --------------  -----  
 0   id                     32573 non-null  int64  
 1   week                   32573 non-null  int64  
 2   center_id              32573 non-null  int64  
 3   meal_id                32573 non-null  int64  
 4   checkout_price         32573 non-null  float64
 5   base_price             32573 non-null  float64
 6   emailer_for_promotion  32573 non-null  int64  
 7   homepage_featured      32573 non-null  int64  
dtypes: float64(2), int64(6)
memory usage: 2.0 MB
```

The meal dataset that we are going to merge with the train, test datasets

```
meal.head()
```

output:

	meal_id	category	cuisine
0	1885	Beverages	Thai
1	1993	Beverages	Thai
2	2539	Beverages	Thai
3	1248	Beverages	Indian
4	2631	Beverages	Indian

```
meal.info()
```

output:

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 51 entries, 0 to 50
Data columns (total 3 columns):
 #   Column    Non-Null Count  Dtype
---  ------    --------------  -----
 0   meal_id   51 non-null     int64
 1   category  51 non-null     object
 2   cuisine   51 non-null     object
dtypes: int64(1), object(2)
memory usage: 1.3+ KB
```

Quick statistical overview:

```
df.describe()
```

output:

	id	week	center_id	meal_id	checkout_price	base_price	emailer_for_promotion	homepage_featured	num_orders
count	4.565480e+05	456548.000000	456548.000000	456548.000000	456548.000000	456548.000000	456548.000000	456548.00000	456548.000000
mean	1.250096e+06	74.768771	82.105796	2024.337458	332.238933	354.156627	0.081152	0.10920	261.872760
std	1.443548e+05	41.524956	45.975046	547.420920	152.939723	160.715914	0.273069	0.31189	395.922798
min	1.000000e+06	1.000000	10.000000	1062.000000	2.970000	55.350000	0.000000	0.00000	13.000000
25%	1.124999e+06	39.000000	43.000000	1558.000000	228.950000	243.500000	0.000000	0.00000	54.000000
50%	1.250184e+06	76.000000	76.000000	1993.000000	296.820000	310.460000	0.000000	0.00000	136.000000
75%	1.375140e+06	111.000000	110.000000	2539.000000	445.230000	458.870000	0.000000	0.00000	324.000000
max	1.499999e+06	145.000000	186.000000	2956.000000	866.270000	866.270000	1.000000	1.00000	24299.000000

```
df_test.describe()
```

output:

	id	week	center_id	meal_id	checkout_price	base_price	emailer_for_promotion	homepage_featured
count	3.257300e+04	32573.000000	32573.000000	32573.000000	32573.000000	32573.000000	32573.000000	32573.000000
mean	1.248476e+06	150.477819	81.901728	2032.067909	341.854440	356.493615	0.066435	0.081356
std	1.441580e+05	2.864072	45.950455	547.199004	153.893886	155.150101	0.249045	0.273385
min	1.000085e+06	146.000000	10.000000	1062.000000	67.900000	89.240000	0.000000	0.000000
25%	1.123969e+06	148.000000	43.000000	1558.000000	214.430000	243.500000	0.000000	0.000000
50%	1.247296e+06	150.000000	76.000000	1993.000000	320.130000	321.130000	0.000000	0.000000
75%	1.372971e+06	153.000000	110.000000	2569.000000	446.230000	455.930000	0.000000	0.000000
max	1.499996e+06	155.000000	186.000000	2956.000000	1113.620000	1112.620000	1.000000	1.000000

Joining the datasets:

```
df = df.merge(meal,
          how = "left",
          on = "meal_id")

df_test = df_test.merge(meal,
                  how = "left",
                  on = "meal_id")
```

There was a mistake filling the data with base price and checkout price in some instances we notice base price smaller than checkout price which is impossible

```
# Function to check and swap values if needed
def correct_prices(row):
    if row['base_price'] < row['checkout_price']:
        # Swap the values to make base_price greater
        row['base_price'], row['checkout_price'] = row['checkout_price'], row['base_price']
    return row

# Apply the function to each row of the DataFrame
df = df.apply(correct_prices, axis = 1)
df_test = df_test.apply(correct_prices, axis = 1)
```

2. Exploratory data analysis

Our exploratory data analysis aims to uncover key trends and relationships within our dataset, providing insight into the factors influencing demand.

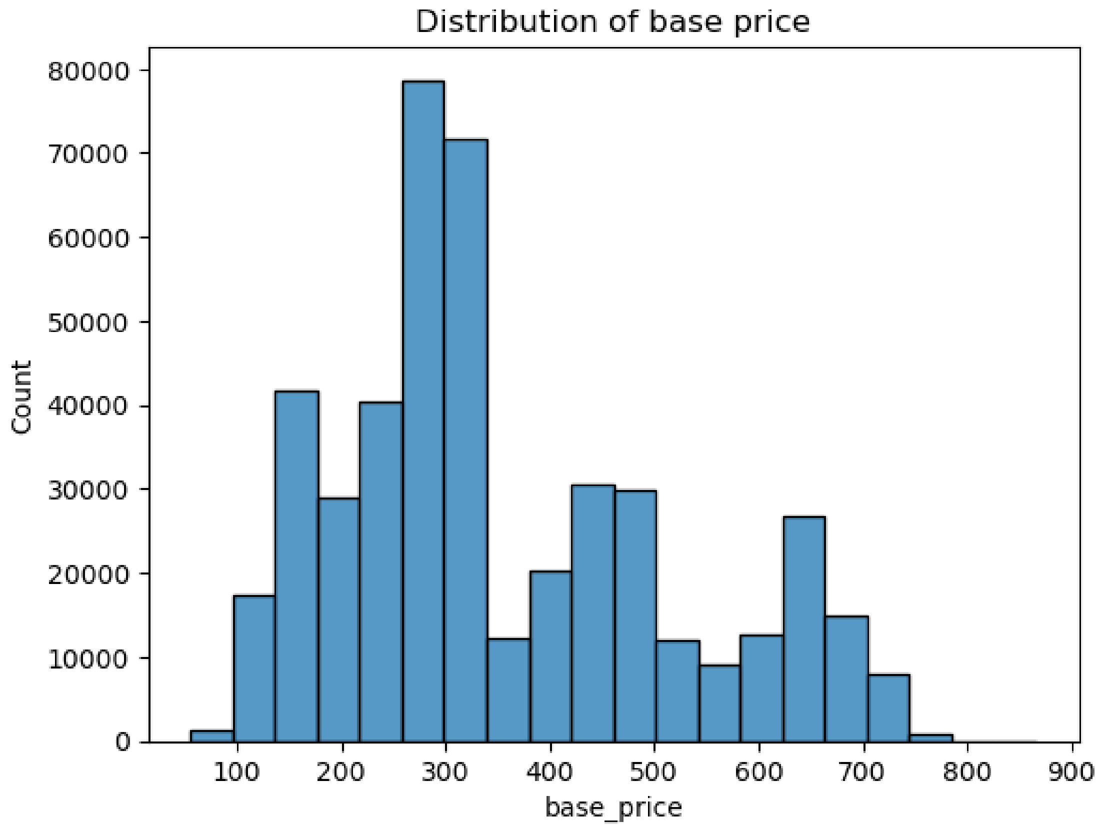
2.1 Summary statistics

Since the dataset is clean, we began with a high-level overview of our key features, examining their distributions to understand their spread and identify any anomalies.

2.1.1 Distribution of base_price

```
sns.histplot(df['base_price'], bins = 20)
plt.title('Distribution of base price')
plt.show()
```

output:

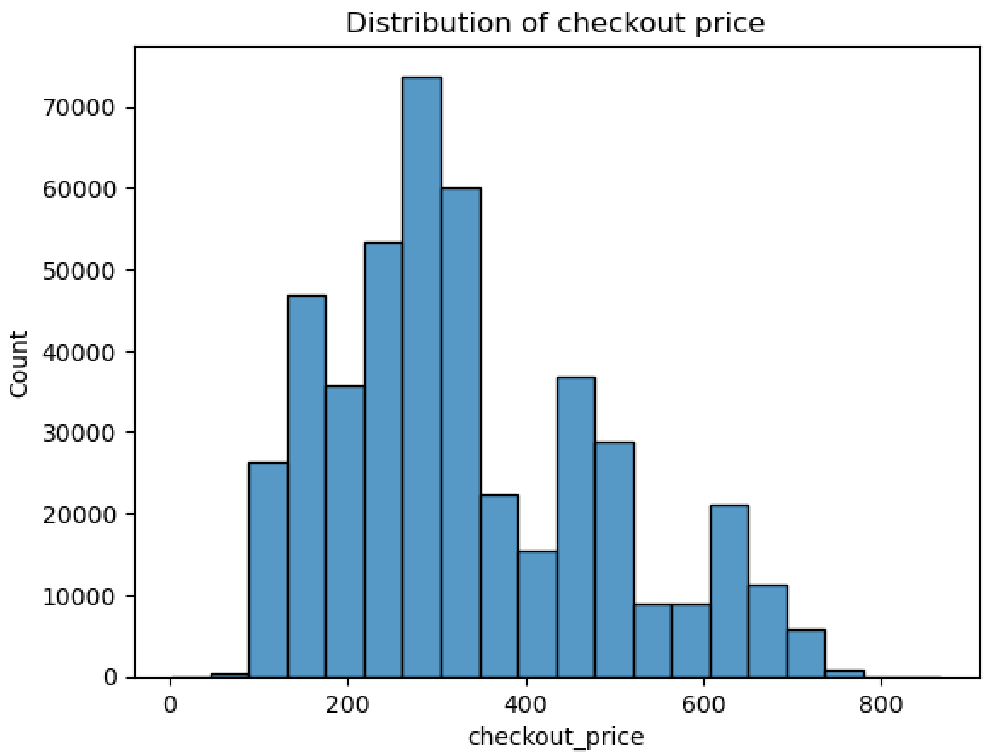


The histogram of base_price shows a right-skewed distribution, indicating a higher concentration of items in the lower price range. The presence of high-value outliers suggests the need for further analysis to understand pricing disparities.

2.1.2 Distribution of the checkout price

```
sns.histplot(df['checkout_price'], bins = 20)
plt.title('Distribution of checkout price')
plt.show()
```

output:

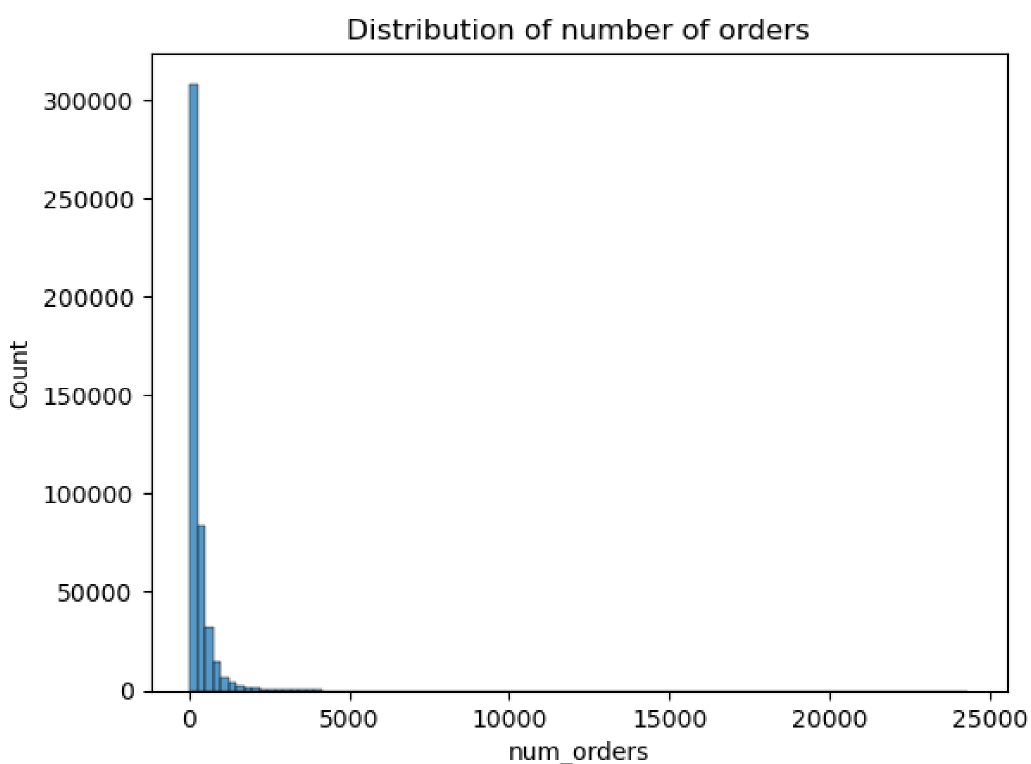


The checkout_price distribution helps identify discrepancies between listed and final prices, reflecting the effects of discounts or promotions. Notable deviations may warrant investigation into pricing strategies.

2.1.3 Distribution of number of orders

```
sns.histplot(df['num_orders'], bins = 100)
plt.title('Distribution of number of orders')
plt.show()
```

output:



The num_orders histogram reveals significant variability in demand, with a long tail suggesting that while most items have moderate demand, a few are extremely popular.

2.2 Category and Cuisine Analysis

Analyzing the breakdown of item categories and cuisines provides insights into consumer preferences and high-demand segments.

2.2.1 Category analysis

```
df["category"].value_counts()
```

output:

```
category
Beverages       127890
Rice Bowl        33408
Sandwich         33291
Pizza            33138
Starters         29941
Other Snacks     29379
Desert           29294
Salad            28559
Pasta            27694
Seafood          26916
Biryani          20614
Extras           13562
Soup             12675
Fish             10187
```

Beverages dominate in total orders, indicating high overall demand. Rice Bowl and Sandwich categories are also popular, suggesting a strong preference for quick and convenient meals. Categories like Soup and Fish have comparatively lower demand.

2.2.2 Cuisine analysis

```
df["cuisine"].value_counts()
```

output:

```
cuisine
Italian        122925
Thai           118216
Indian         112612
Continental    102795
```

Italian and Thai cuisines lead in popularity, closely followed by Indian cuisine. Continental cuisine has the lowest total orders, presenting an opportunity for targeted marketing or menu diversification.

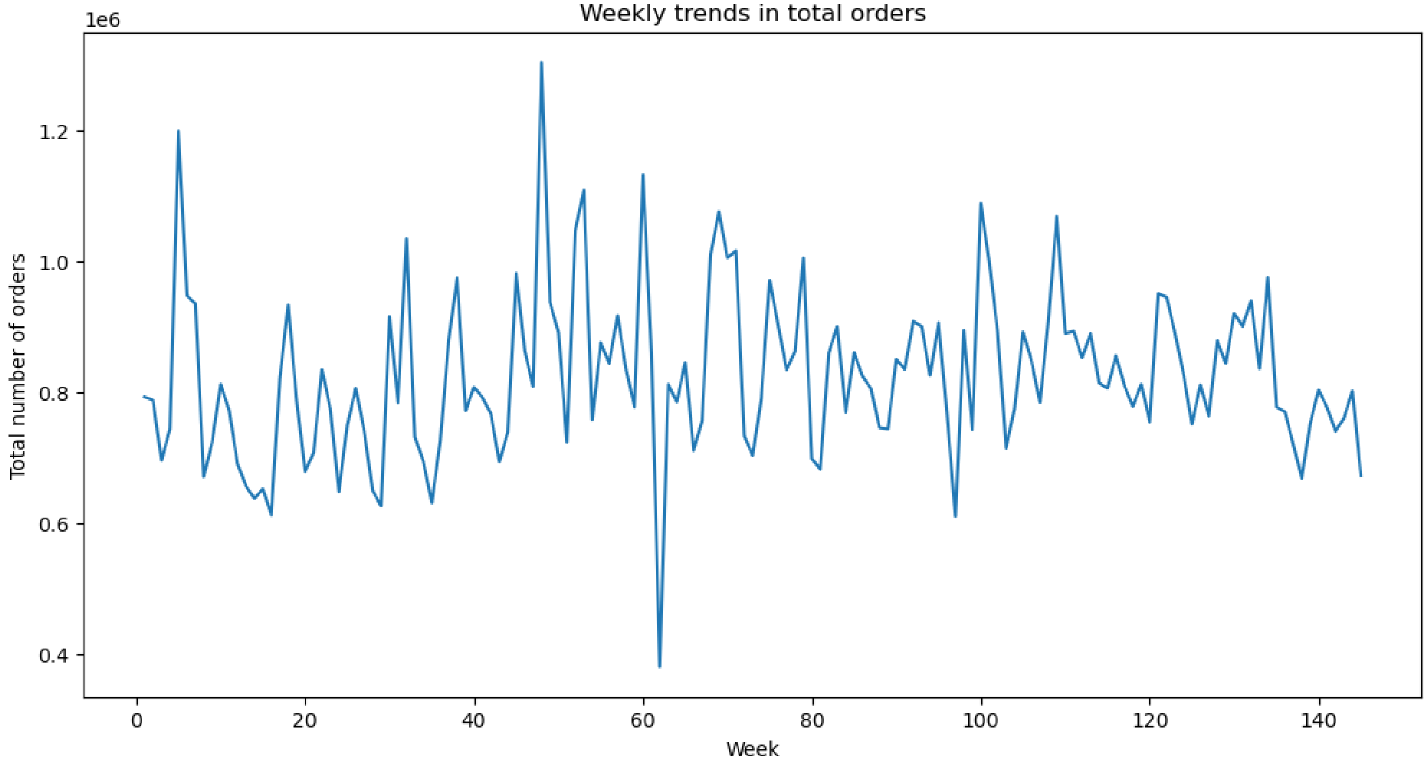
2.3 Time series analysis

Analyzing demand trends over time can reveal seasonal patterns or growth trends.

2.3.1 Weekly trend

```
weekly_trends = df.groupby("week")["num_orders"].sum()

plt.figure(figsize=(12, 6))
weekly_trends.plot()
plt.title("Weekly trends in total orders")
plt.xlabel("Week")
```

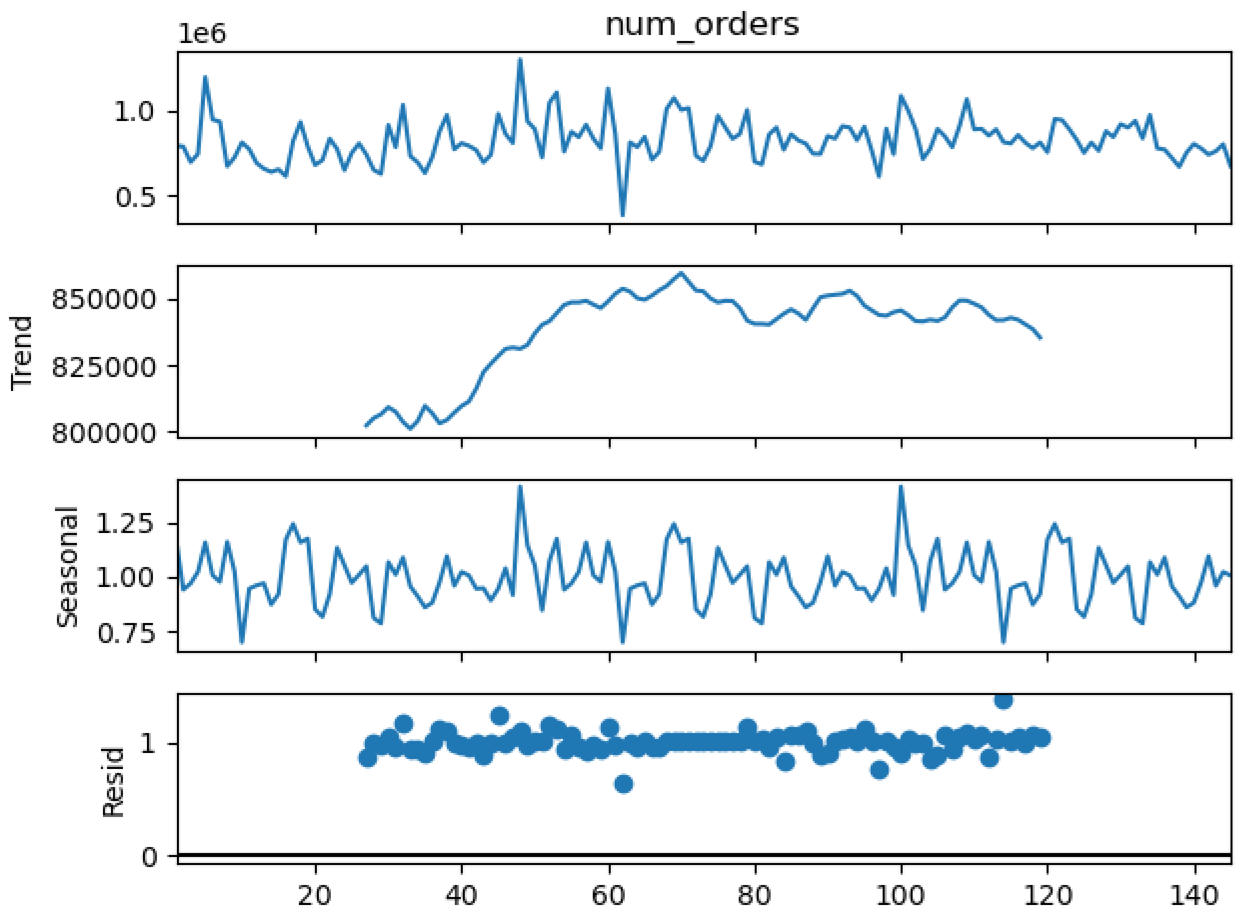
```
plt.ylabel("Total number of orders")
plt.show()
```
output:



The weekly trend analysis displays demand fluctuations. Peaks could indicate specific events or promotions, and troughs might suggest lower consumer activity periods.

2.3.2 Decomposition of Weekly Orders

```
df_weekly = df.groupby('week')['num_orders'].sum().sort_index()

decomposition = seasonal_decompose(df_weekly, model='multiplicative', period=52)  # period=52 for weekly seasonality over a year

decomposition.plot()
plt.show()
```
output:



The decomposition highlights underlying patterns: seasonality, trend, and residual noise, helping us understand long-term demand behavior.

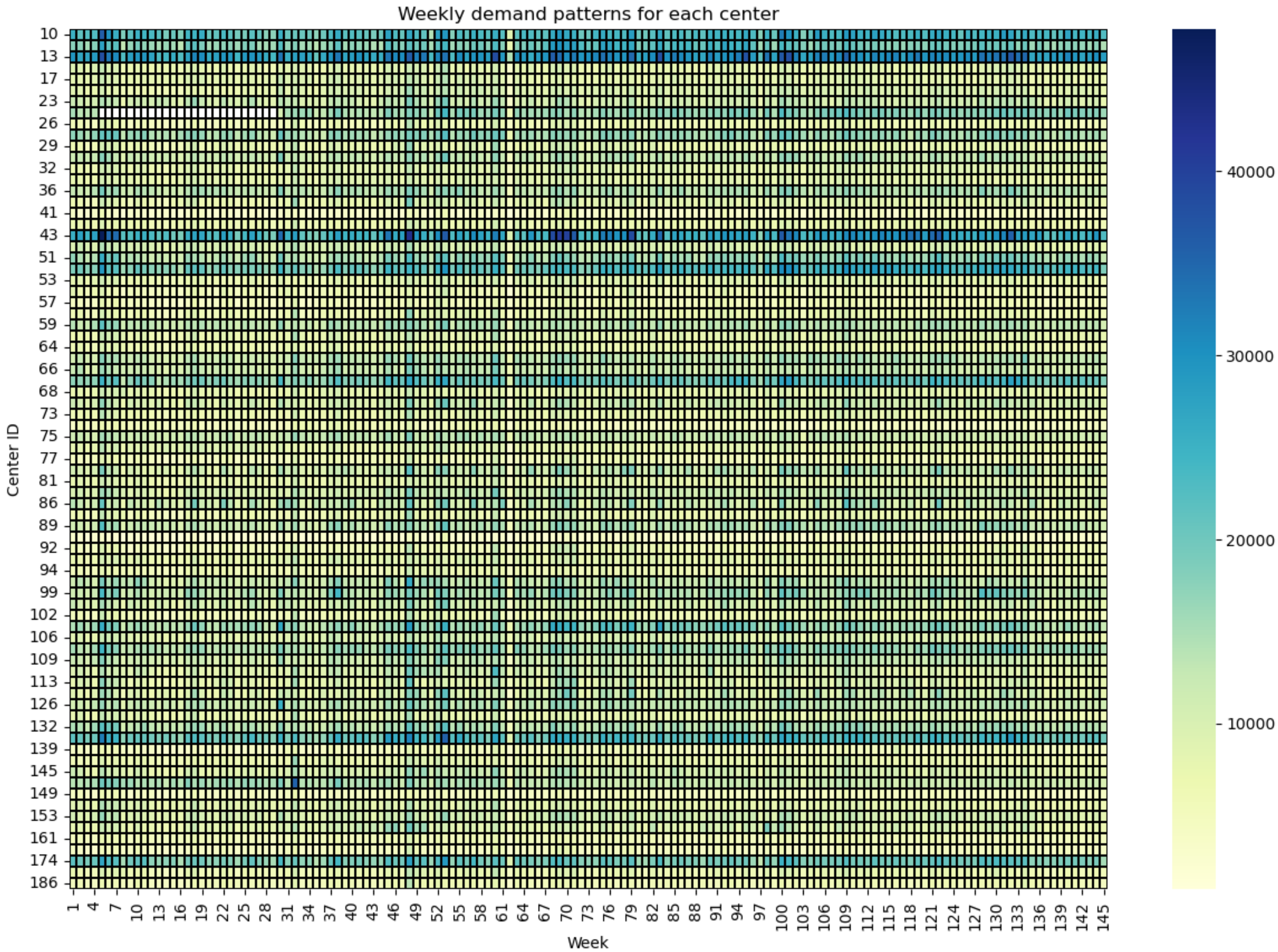
2.4 Demand patterns analysis

Visualizing demand across centers helps identify regional trends and optimize operations.

2.4.1 Heatmap of weekly demand patterns

```
demand_heatmap_data = df.pivot_table(
    index = 'center_id',
    columns = 'week',
    values = 'num_orders',
    aggfunc = 'sum')

plt.figure(figsize=(15, 10))
sns.heatmap(demand_heatmap_data, cmap="YlGnBu", linewidths=0.1, linecolor="black")
plt.title("Weekly demand patterns for each center")
plt.xlabel("Week")
plt.ylabel("Center ID")
plt.show()
```

output:



The heatmap illustrates demand levels across different centers over time. Notably, centers 10, 11, 13, and 43 exhibit high demand, while centers 51, 52, 67, 137, and 174 show medium demand. The remaining centers demonstrate low to medium demand in comparison.

By visualizing these demand patterns, we can effectively identify which centers require more resources and where adjustments can be made to optimize operations.

2.5 Price analysis

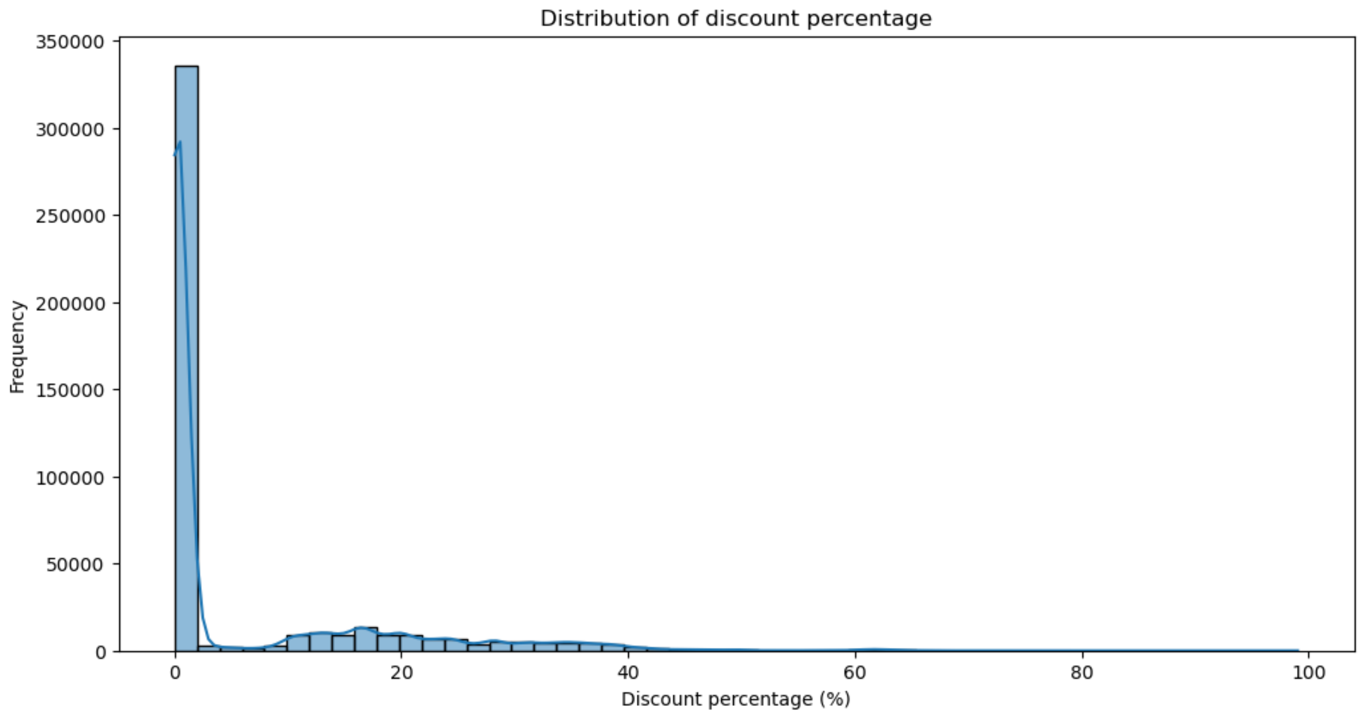
Exploring the relationship between base_price, checkout_price, and discounts offers insights into pricing strategies.

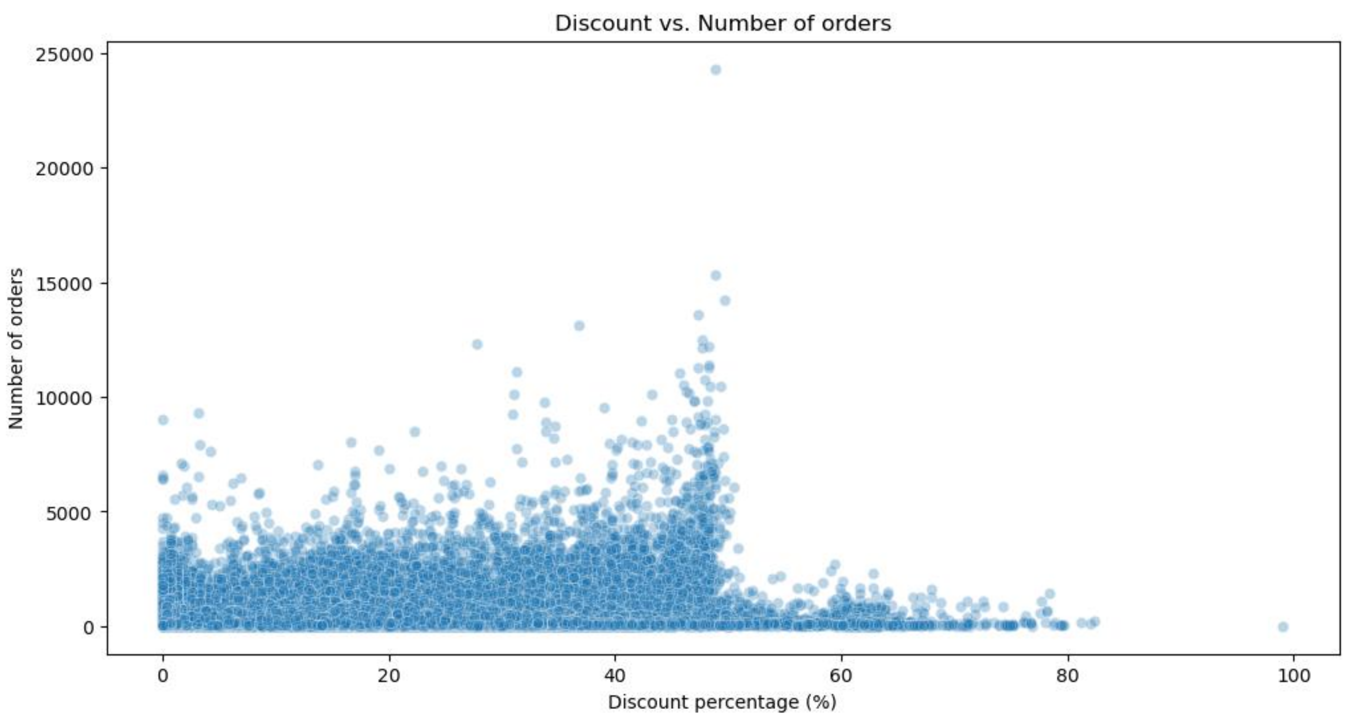
2.5.1 Discount calculation and distribution

```
df['discount'] = np.abs(((df['base_price'] - df['checkout_price']) / df['base_price'])) * 100

plt.figure(figsize=(12, 6))
sns.histplot(df['discount'], bins=50, kde=True)
plt.title("Distribution of discount percentage")
plt.xlabel("Discount percentage (%)")
plt.ylabel("Frequency")
plt.show()

plt.figure(figsize=(12, 6))
```

```python
sns.scatterplot(x='discount', y='num_orders', data=df, alpha=0.3)
plt.title("Discount vs. Number of orders")
plt.xlabel("Discount percentage (%)")
plt.ylabel("Number of orders")
plt.show()

discount_correlation = df[['discount', 'num_orders']].corr().iloc[0, 1]
print(f"Correlation between discount and number of orders: {discount_correlation}")
```

output:





The discount distribution shows the extent and frequency of price reductions, indicating how often items are discounted and to what degree.

The scatter plot suggests a positive correlation between discount levels and order quantities, indicating that higher discounts generally boost demand.

A correlation of 0.21, though not very strong, shows that discounts have a noticeable impact on order numbers.

2.5.2 Analyzing orders by discount thresholds

```
df['discount_bin'] = pd.cut(df['discount'],
                           bins=[-np.inf, 0, 10, 20, 30, 50, np.inf],
                           labels=['No Discount', '0-10%', '10-20%', '20-30%', '30-50%', '50%+'])
avg_orders_by_discount = df.groupby('discount_bin')['num_orders'].mean()
print(avg_orders_by_discount)
```

output:

```
discount_bin
No Discount    207.538127
0-10%          225.528170
10-20%         347.311908
20-30%         353.669952
30-50%         546.420060
50%+           217.505357
```

Orders increase significantly in the 20-30% and 30-50% discount bins, indicating optimal discount ranges for boosting sales.

2.6 Promotion effectiveness

Understanding the impact of promotional strategies is crucial for optimizing marketing efforts and boosting demand. In this section, we analyze the influence of email promotions and homepage features, both independently and combined.

2.6.1 Impact of e-mail promotions and homepage features

First i evaluated how promotions affect the average number of orders by examining different scenarios: no promotion, email promotion, homepage feature, and both promotions combined.

```
promo_impact = df.groupby(['emailer_for_promotion', 'homepage_featured', 'discount_bin'])['num_orders'].mean().unstack()
print(promo_impact)
```

output:

```
discount_bin                           No Discount       0-10%      10-20%
emailer_for_promotion homepage_featured
0                     0                 205.741155  205.687197  263.519953
                      1                 268.758514  481.576769  460.953032
1                     0                 338.652778  356.260153  424.243573
                      1                 740.185714  641.145329  784.834038

discount_bin                                20-30%      30-50%        50%+
emailer_for_promotion homepage_featured
0                     0                 220.962354  199.054567  190.634670
                      1                 402.820567  552.708167  261.316547
1                     0                 357.026953  567.411153  216.153285
                      1                 652.844899 1021.747861  345.692810
```

The data reveal clear patterns:

- Email Promotion Impact: Implementing email promotions significantly increases the average number of orders, indicating the effectiveness of direct marketing strategies.
- Homepage Feature Impact: Featuring items on the homepage also results in a noticeable boost in order volume, demonstrating the importance of product visibility.

2.6.2 Email promotion impact

```
avg_orders_emailer = df.groupby('emailer_for_promotion')['num_orders'].mean()
print(avg_orders_emailer)
```

output:

```
emailer_for_promotion
0    229.262883
1    631.097544
```

Orders increase substantially when email promotions are activated compared to when they are not. This suggests that email marketing is a powerful tool for driving sales.

2.6.3 Home page featured impact

```
avg_orders_homepage = df.groupby('homepage_featured')['num_orders'].mean()
print(avg_orders_homepage)
```

output:

```
homepage_featured
0    221.050040
1    594.884786
```

Similar to email promotions, featuring items on the homepage yields a higher average order count, underscoring the value of strategic product placement.

2.6.4 Combined impact of both promotions

To understand how both promotional strategies work together, we use a pivot table to analyze their combined effect.

```
avg_orders_both_promotions = df.pivot_table(
   index = 'emailer_for_promotion',
   columns = 'homepage_featured',
   values = 'num_orders',
   aggfunc = "mean")
avg_orders_both_promotions
```

output:

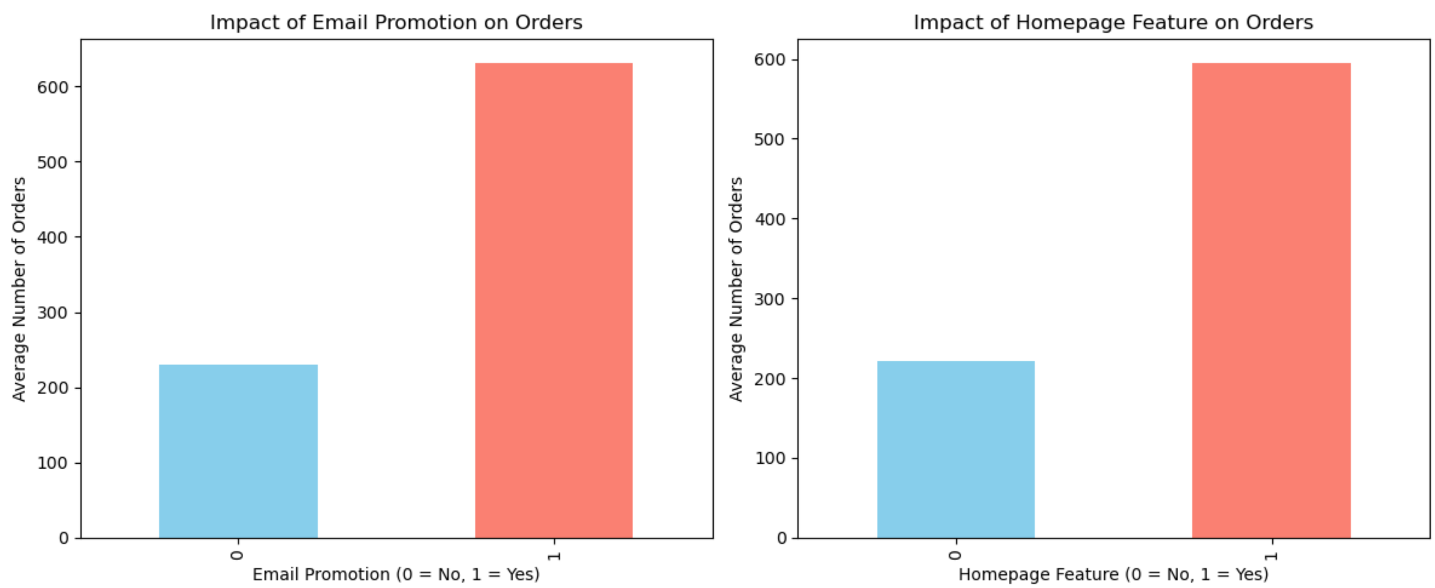
homepage_featured	0	1
emailer_for_promotion		
0	211.416983	455.876208
1	431.277625	816.246061

The combination of both email promotions and homepage features results in the highest average order volume, indicating a synergistic effect when both strategies are used simultaneously.

Visualization of promotion impact

i used bar plots to visualize the impact of each promotion type:

```
# Plotting for emailer_for_promotion
plt.figure(figsize=(12, 5))
plt.subplot(1, 2, 1)
avg_orders_emailer.plot(kind='bar', color=['skyblue', 'salmon'])
plt.title("Impact of Email Promotion on Orders")
plt.xlabel("Email Promotion (0 = No, 1 = Yes)")
plt.ylabel("Average Number of Orders")

# Plotting for homepage_featured
plt.subplot(1, 2, 2)
avg_orders_homepage.plot(kind='bar', color=['skyblue', 'salmon'])
plt.title("Impact of Homepage Feature on Orders")
plt.xlabel("Homepage Feature (0 = No, 1 = Yes)")
plt.ylabel("Average Number of Orders")

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



The bar plots provide a clear visual comparison of how each promotion strategy influences order numbers, reinforcing our earlier observations.

2.6.5 Promotion effectiveness by week

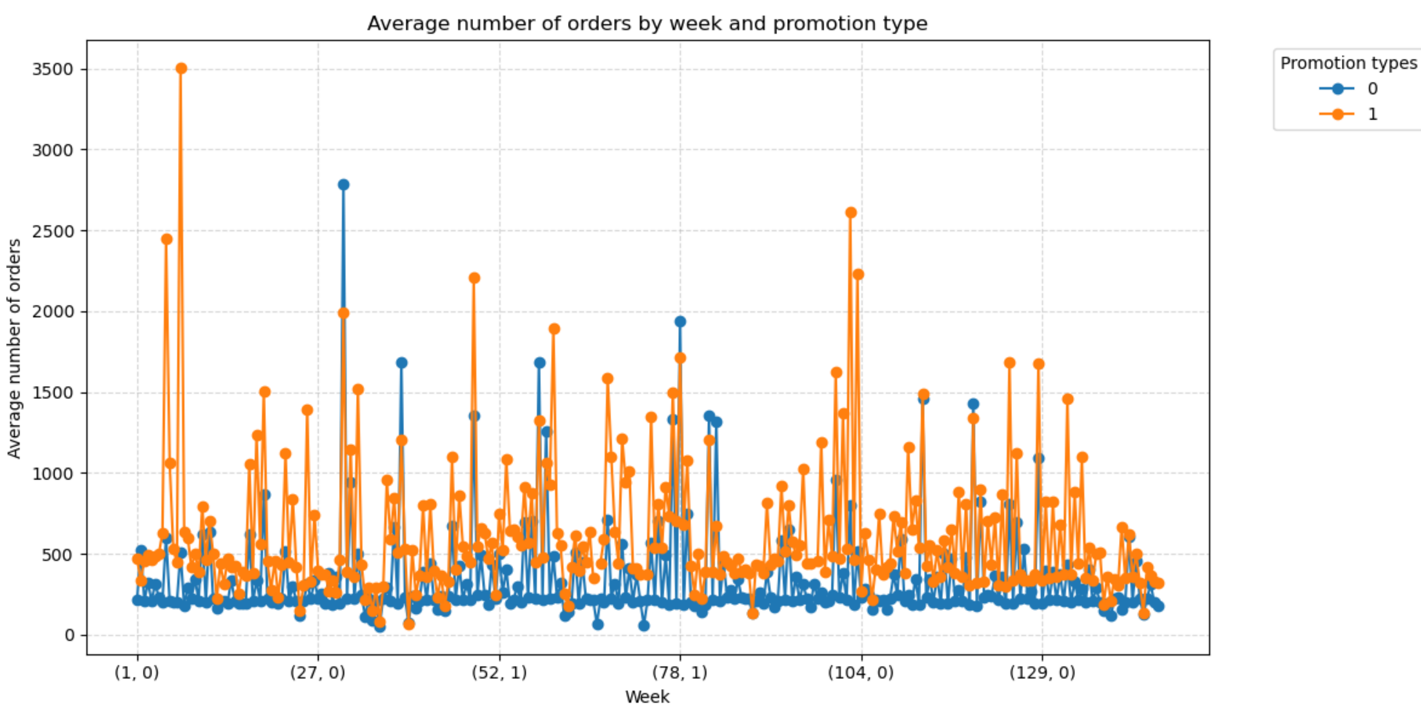
```
promotion_by_week = (df.groupby(['week', 'emailer_for_promotion', 'homepage_featured'])['num_orders']
                    .mean()
                    .unstack()
)
promotion_by_week.head(6)
```

output:

	homepage_featured	0	1
week	emailer_for_promotion		
1	0	217.996935	469.174377
	1	525.843750	336.308271
2	0	211.675184	455.366667
	1	323.302594	492.156934
3	0	209.444972	465.955390
	1	314.894737	487.510638

```
promotion_by_week.plot(kind='line', figsize=(12, 6), marker='o')

plt.title('Average number of orders by week and promotion type')
plt.xlabel('Week')
plt.ylabel('Average number of orders')
plt.legend(title='Promotion types', bbox_to_anchor=(1.05, 1), loc='upper left')
plt.grid(visible=True, linestyle='--', alpha=0.5)
plt.tight_layout()

plt.show()
```

output:



The line plot illustrates how the effectiveness of promotional strategies evolves over time. Peaks in the graph may correspond to specific marketing campaigns or seasonal fluctuations, providing valuable insights for future promotional planning.

2.7 Category and cuisine analysis

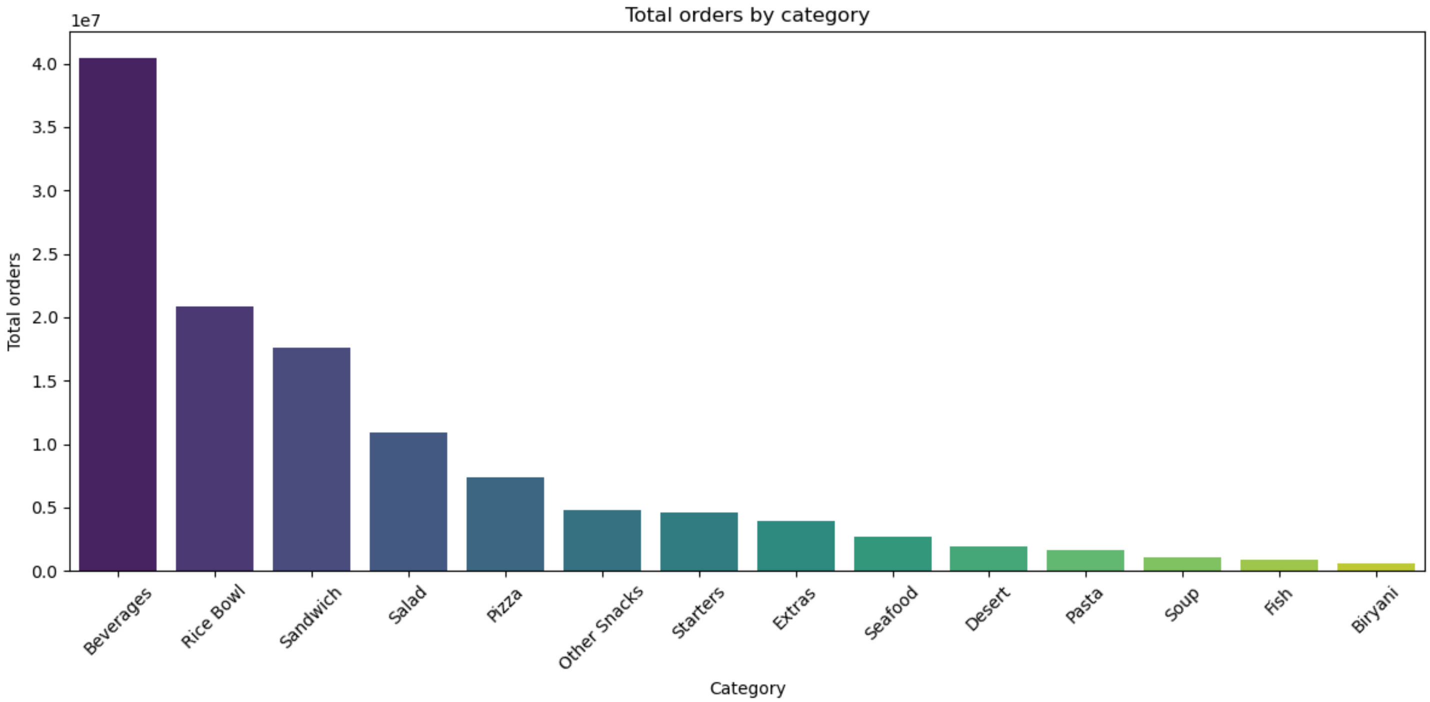
We begin by analyzing the demand across different item categories and cuisines to identify trends and high-performing segments.

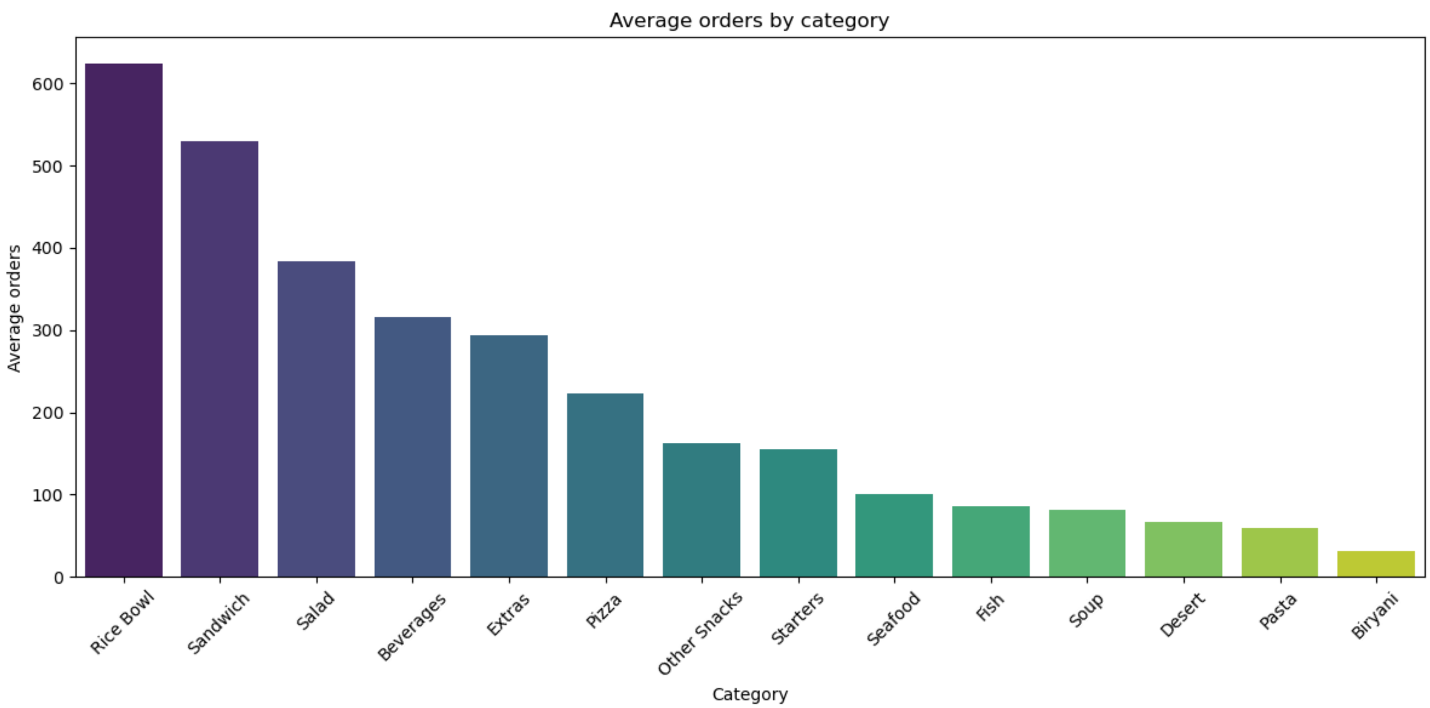
2.7.1 Total and average orders by category

To understand which item categories are most in demand, we calculate both the total and average number of orders:

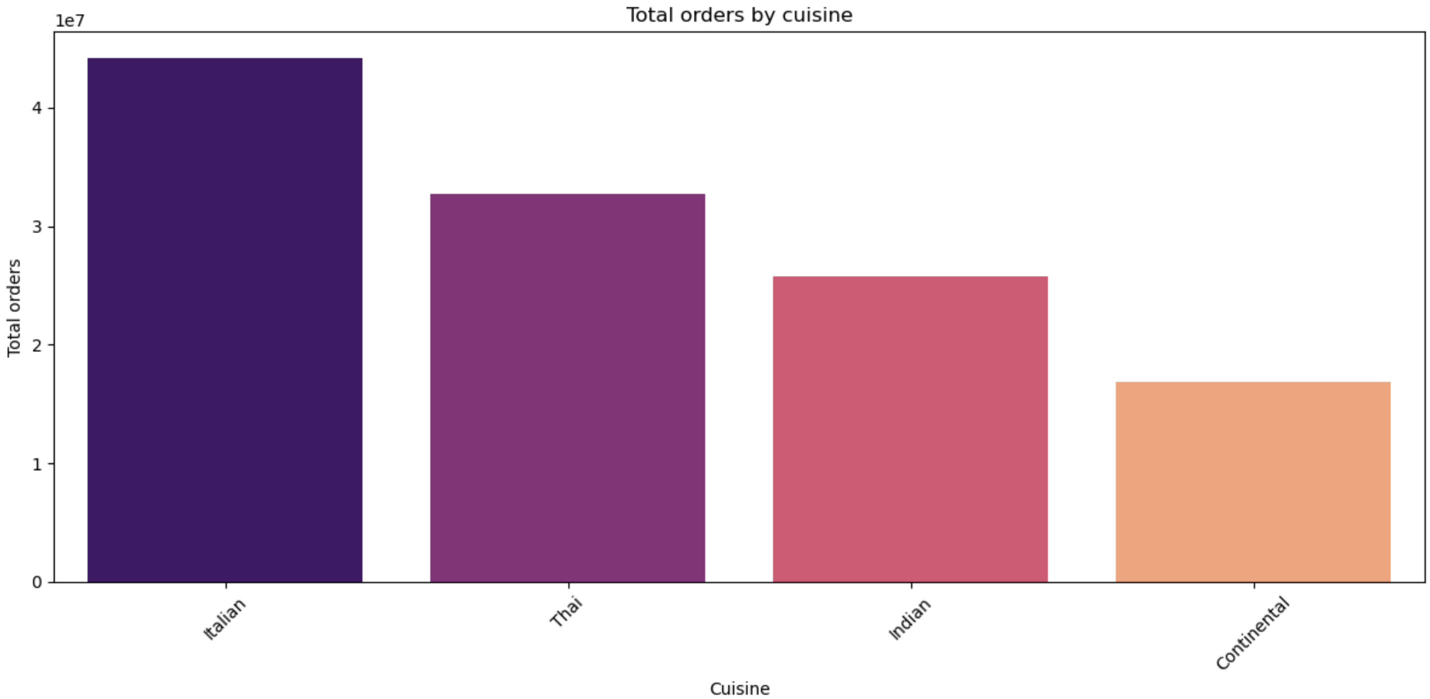
```
# Calculate total orders by category and cuisine
orders_by_category = df.groupby('category')['num_orders'].sum().sort_values(ascending=False)
orders_by_cuisine = df.groupby('cuisine')['num_orders'].sum().sort_values(ascending=False)

# Calculate average orders by category and cuisine
avg_orders_by_category = df.groupby('category')['num_orders'].mean().sort_values(ascending=False)
avg_orders_by_cuisine = df.groupby('cuisine')['num_orders'].mean().sort_values(ascending=False)

# Plot for total orders by category
sns.barplot(x=orders_by_category.index, y=orders_by_category.values, palette='viridis')
plt.xticks(rotation=45, fontsize=10)
plt.title('Total orders by category', fontsize=12)
plt.xlabel('Category', fontsize=10)
plt.ylabel('Total orders', fontsize=10)
plt.tight_layout()  # Adjust layout to avoid clipping
plt.show()

# Plot for total orders by cuisine
sns.barplot(x=orders_by_cuisine.index, y=orders_by_cuisine.values, palette='magma')
plt.xticks(rotation=45, fontsize=10)
plt.title('Total orders by cuisine', fontsize=12)
plt.xlabel('Cuisine', fontsize=10)
plt.ylabel('Total orders', fontsize=10)
plt.tight_layout()
plt.show()

# Plot for average orders by category
sns.barplot(x=avg_orders_by_category.index, y=avg_orders_by_category.values, palette='viridis')
plt.xticks(rotation=45, fontsize=10)
plt.title('Average orders by category', fontsize=12)
plt.xlabel('Category', fontsize=10)
plt.ylabel('Average orders', fontsize=10)
plt.tight_layout()
plt.show()

# Plot for average orders by cuisine
sns.barplot(x=avg_orders_by_cuisine.index, y=avg_orders_by_cuisine.values, palette='magma')
plt.xticks(rotation=45, fontsize=10)
plt.title('Average orders by cuisine', fontsize=12)
plt.xlabel('Cuisine', fontsize=10)
plt.ylabel('Average orders', fontsize=10)
plt.tight_layout()
plt.show()
```

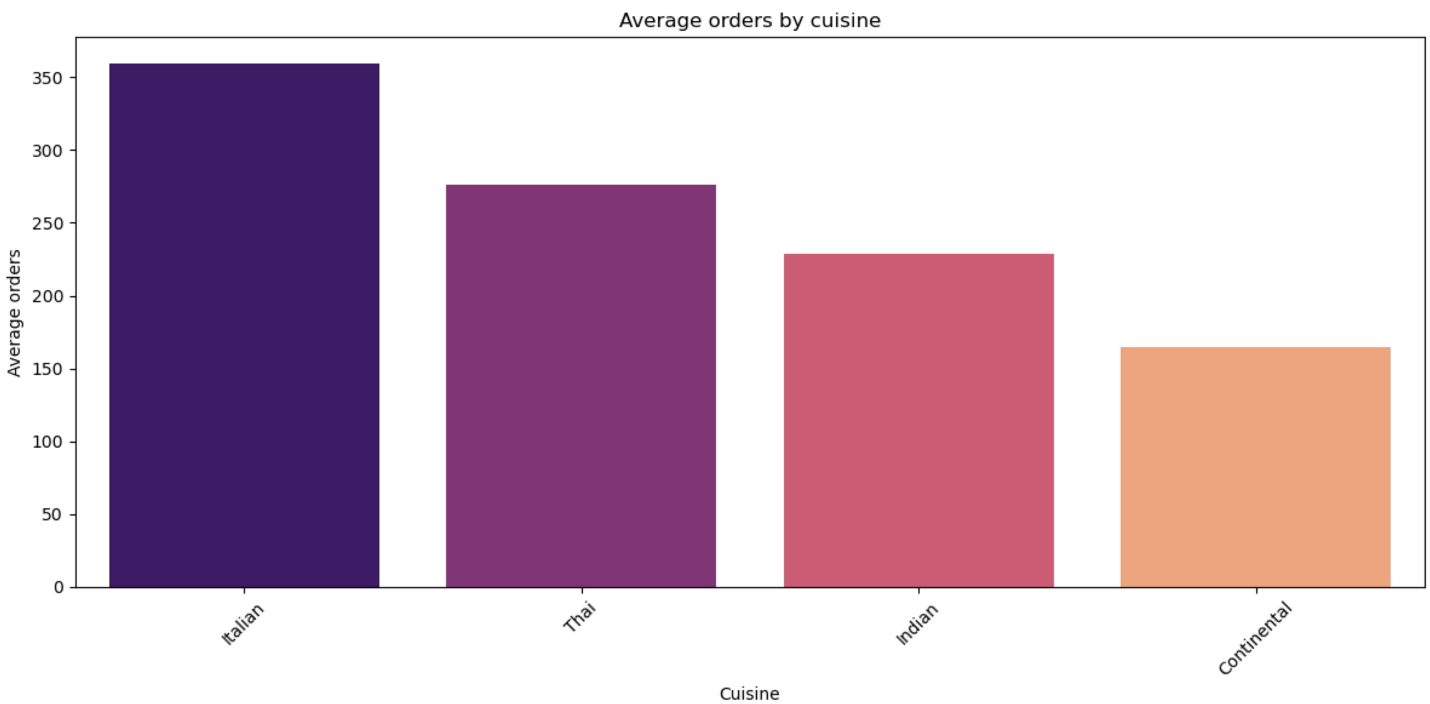
output:





- **Total orders**: The Beverages category leads by a substantial margin, indicating its high overall demand. Rice Bowl and Sandwich follow closely, suggesting a preference for quick and convenient meal options. Categories like Soup and Fish show comparatively lower demand.
- **Average orders**: When looking at average orders per item, Rice Bowl and Sandwich categories are again prominent, emphasizing their consistent popularity. Beverages and Extras have moderate average demand, while Other Snacks and Starters have similar lower average order counts.





- **Total orders**: `Italian` cuisine leads in total orders, followed by `Thai` and `Indian`, both of which have similar demand. `Continental` cuisine has the lowest total orders, indicating potential areas for menu diversification or marketing.
- **Average orders**: The ranking of cuisines remains consistent, with `Italian` cuisine maintaining its lead in average orders, suggesting strong and stable customer preferences.

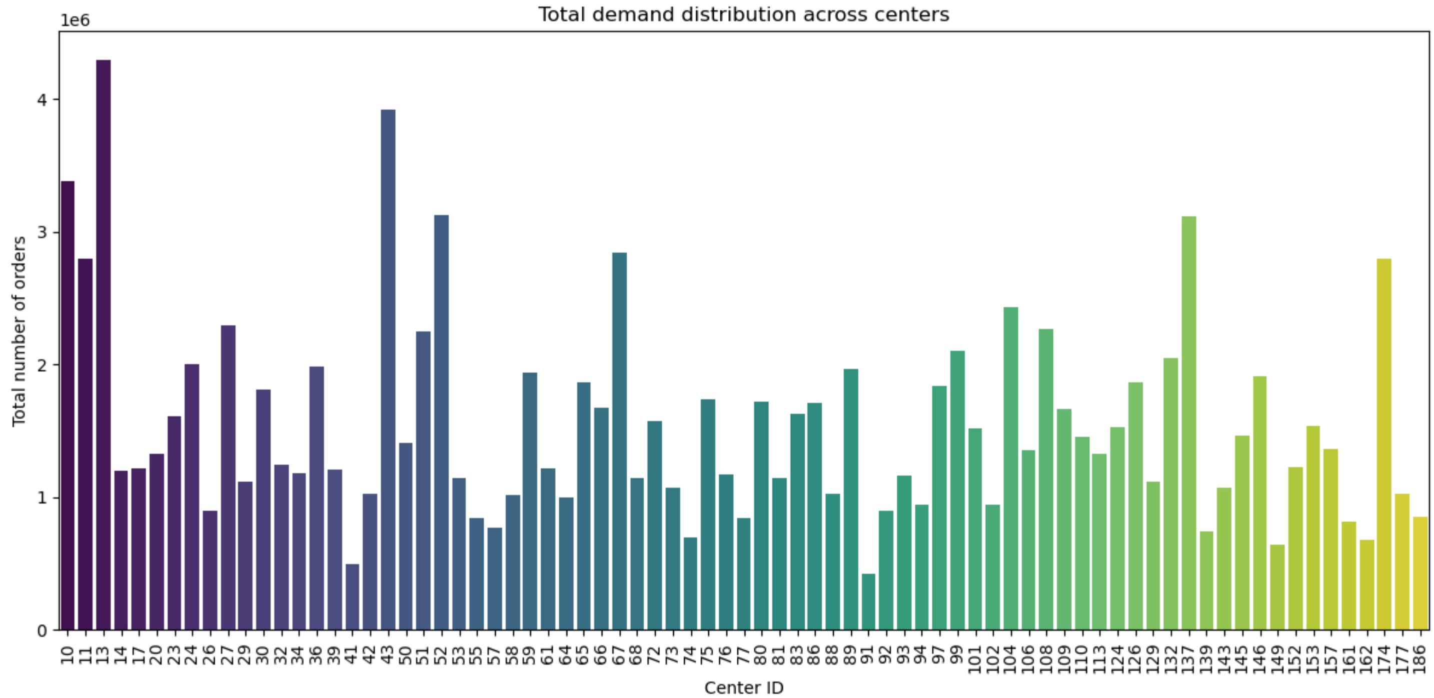
2.8 Center level analysis

I evaluated how demand varies across different centers, which helps in understanding regional preferences and optimizing resource distribution.

2.8.1 Demand distribution across centers

```
demand_by_center = df.groupby('center_id')['num_orders'].sum().reset_index()

# Sort centers by total demand
demand_by_center = demand_by_center.sort_values(by='num_orders', ascending=False)

# Plot demand distribution across centers
sns.barplot(x='center_id', y='num_orders', data=demand_by_center, palette='viridis')
plt.xticks(rotation=90)
plt.title('Total demand distribution across centers')
plt.xlabel('Center ID')
plt.ylabel('Total number of orders')
plt.tight_layout()
plt.show()
```

output:



The bar plot highlights which centers have the highest demand, allowing for better allocation of resources and targeted marketing efforts.

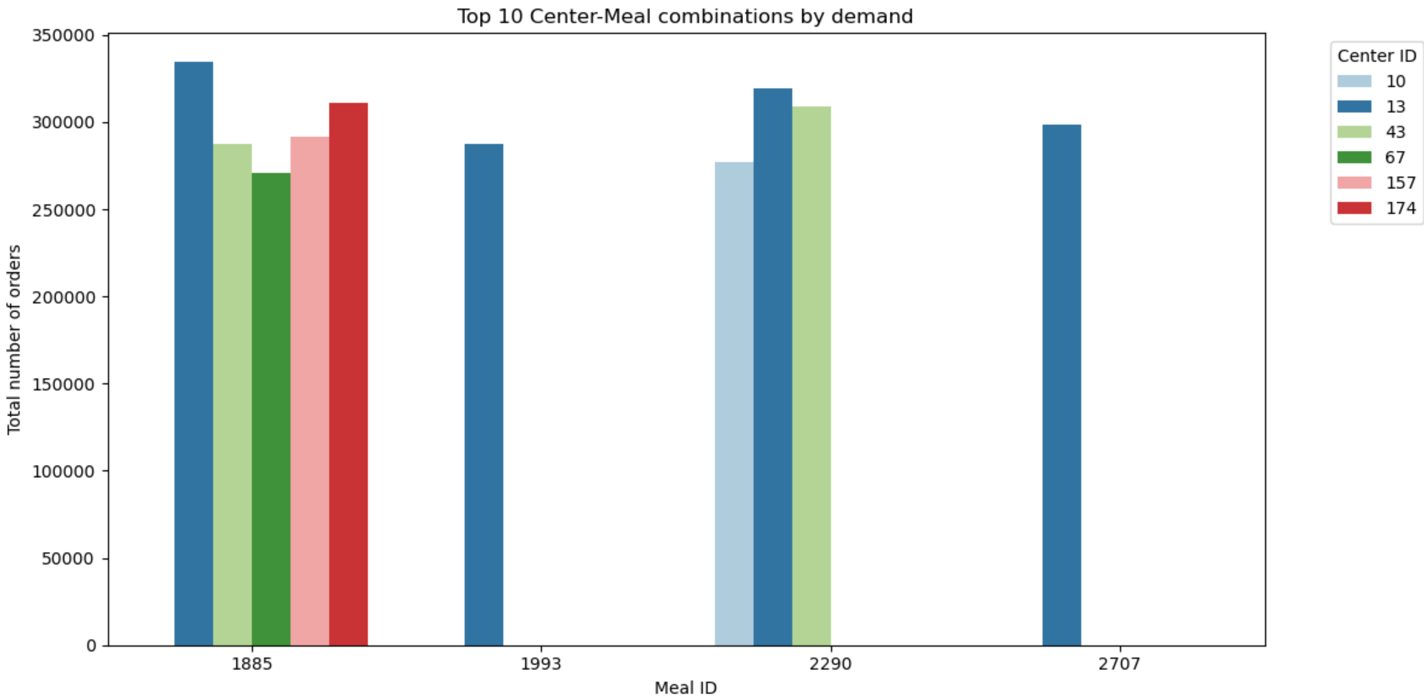
2.8.2 High-Demand Center-Meal combinations

Identifying popular center-meal combinations can help streamline operations and focus on high-demand pairings.

```
# Group by center_id and meal_id, and sum the number of orders
demand_by_center_meal = df.groupby(['center_id', 'meal_id'])['num_orders'].sum().reset_index()

# Sort by total demand to find high-demand center-meal combinations
demand_by_center_meal = demand_by_center_meal.sort_values(by='num_orders', ascending=False)

# Optionally, you can filter for the top N combinations
top_n = 10  # Change this to the number of top combinations you want
top_demand_pairs = demand_by_center_meal.head(top_n)
```

```
# Plot the top center-meal combinations
sns.barplot(x='meal_id', y='num_orders', hue='center_id', data=top_demand_pairs, palette='Paired')
plt.title(f'Top {top_n} Center-Meal combinations by demand')
plt.xlabel('Meal ID')
plt.ylabel('Total number of orders')
plt.legend(title='Center ID', bbox_to_anchor=(1.05, 1), loc='upper left')
plt.tight_layout()
plt.show()
```

output:



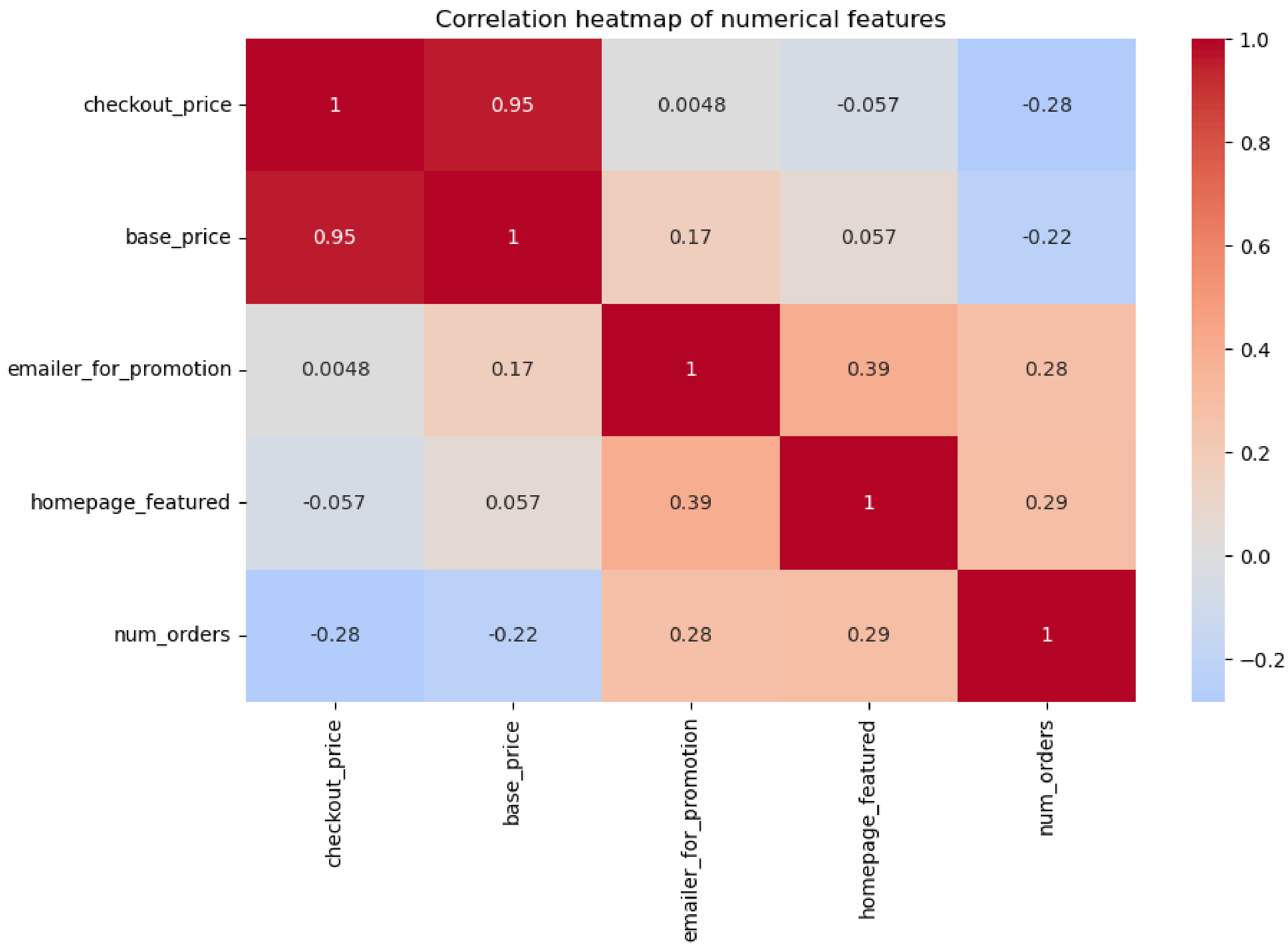
The plot of top center-meal combinations reveals which meals are most popular at specific centers, providing actionable insights for menu optimization and inventory management.

2.9 Correlation analysis

To understand relationships between numerical features, i created a correlation heatmap

```
# Plot the correlation heatmap
plt.figure(figsize=(10, 6))
sns.heatmap(df.drop(["id", "week", "center_id", "meal_id"], axis = 1).corr(numeric_only = True), annot=True,
cmap='coolwarm', center=0)
plt.title('Correlation heatmap of numerical features')
plt.show()
```

output:



Promotional efforts (like email and homepage features) are positively associated with an increase in the number of orders.

Price variables have a negative association with the number of orders, reflecting the general consumer tendency to prefer lower-priced items.

The high correlation between checkout_price and base_price warrants careful consideration to avoid redundancy or multicollinearity in your predictive modeling.

1. Modeling

I started by creating a copy of the training dataset and removing the non-essential id column

```
df_model = df.copy()
df_model = df_model.drop(["id"], axis = 1)
```

This allowed me to focus on features that would contribute to model performance.

Initial Model Performance:

- **Baseline Linear Regression**: Training a simple multiple linear regression model without feature engineering resulted in an R^2 score of only 0.3.

- **Feature Engineering**: By enhancing the data with new features (explained below), the R^2 improved to 0.5. Ridge and Lasso regressions provided similar results. Logging the target variable, num_orders, further increased the R^2 to 0.59.

Advanced Modeling:

- Applying ensemble models yielded significantly better performance:
  - **Random Forest**: Achieved an R^2 of 0.79, selected as the final model.
  - **Gradient Boosting**: Achieved an R^2 of 0.61.

To optimize features, I used Lasso regression for feature selection and excluded non-significant ones.

1.1 Feature engineering

I added new features to enhance model performance:

1. Date Features, created month and quarter features from the week data, calculated dates using the start of the year

```
year = 2023
start_date = pd.to_datetime(f'{year}-01-01')

df_model['date'] = start_date + pd.to_timedelta(df_model['week'] - 1, unit='W')

df_model['month'] = df_model['date'].dt.month
df_model['quarter'] = df_model['date'].dt.quarter

df_model.drop(columns='date', inplace=True)
```

2. Pricing and Discount Features, computed discount ratios and price interactions

```
df_model = df_model.assign(
   discount_ratio = df_model["discount"] / df_model["base_price"],
   price_diff = np.abs(df_model["base_price"] - df_model["checkout_price"]),
   price_discount_interaction = df_model['checkout_price'] * df_model['discount'],
   promotion_discount_interaction = df_model['emailer_for_promotion'] * df_model['discount'],
   homepage_discount_interaction = df_model['homepage_featured'] * df_model['discount'],
   num_orders_log = np.log(df_model["num_orders"])
)
```

3. One-hot encoding, converted categorical features into dummy variables

```
df_model = pd.get_dummies(df_model, drop_first = True)

bool_columns = df_model.select_dtypes(include='bool').columns

df_model[bool_columns] = df_model[bool_columns].astype(int)
```

1.2 Linear regression model

I used statsmodels to build and evaluate a multiple linear regression model

```
import statsmodels.api as sm
from sklearn.metrics import mean_absolute_error as mae
from sklearn.metrics import r2_score as r2
from sklearn.model_selection import train_test_split
from sklearn.model_selection import KFold
```

```
X = sm.add_constant(df_model.drop(["num_orders", "num_orders_log", "meal_id", "base_price", "discount", "price_diff",
"avg_demand_by_cuisine", "discount_ratio"], axis = 1))
y = df_model["num_orders_log"]

X, X_test, y, y_test = train_test_split(X, y, test_size = 0.2, random_state = 2024)

kf = KFold(n_splits = 5, shuffle = True, random_state = 2024)

# Create a list to store validation scores for each fold
cv_lm_r2s = []
cv_lm_mae = []

# Loop in each fold in X and y
for train_ind, val_ind in kf.split(X, y):
    # Subset data based on CV folds
    X_train, y_train = X.iloc[train_ind], y.iloc[train_ind]
    X_val, y_val = X.iloc[val_ind], y.iloc[val_ind]
    # Fit the model on fold's training data
    model = sm.OLS(y_train, X_train).fit()
    # Append validation score to list
    cv_lm_r2s.append(r2(y_val, model.predict(X_val)))
    cv_lm_mae.append(mae(y_val, model.predict(X_val)))

print("All validation R2s: ", [round(x, 3) for x in cv_lm_r2s])
print(f"Cross validation R2s: {round(np.mean(cv_lm_r2s), 3)} +- {round(np.std(cv_lm_r2s), 3)}")

print("All validation MAEs: ", [round(x, 3) for x in cv_lm_mae])
print(f"Cross validation MAEs: {round(np.mean(cv_lm_mae), 3)} +- {round(np.std(cv_lm_mae), 3)}")
```

output:

```
All validation R2s:  [0.599, 0.596, 0.596, 0.602, 0.6]
Cross validation R2s: 0.599 +- 0.002
All validation MAEs:  [135.583, 134.901, 135.8, 137.157, 136.918]
Cross validation MAEs: 136.072 +- 0.846
```

The linear model showed moderate predictive power, indicating the need for more complex models to capture non-linear patterns.

1.3 Ridge regression

Implemented Ridge regression with cross-validation

```
from sklearn.linear_model import RidgeCV
from sklearn.preprocessing import StandardScaler

X = df_model.drop(["num_orders", "num_orders_log", "meal_id", "base_price", "discount", "price_diff",
"avg_demand_by_cuisine", "discount_ratio"], axis = 1)
y = df_model["num_orders_log"]

X, X_test, y, y_test = train_test_split(X, y, test_size = 0.2, random_state = 2024)

std = StandardScaler()
```

```
X_tr = std.fit_transform(X.values)

n_alphas = 200
alphas = 10 ** np.linspace(-3, 3, n_alphas)

ridge_model = RidgeCV(alphas = alphas, cv = 5)

ridge_model.fit(X_tr, y)
print(f"Alpha: {ridge_model.alpha_}")
print(f"Train R²: {ridge_model.score(X_tr, y)}")
print(f"Mean absolute error: {mae(np.exp(y), np.exp(ridge_model.predict(X_tr)))}")
```

output:

```
Alpha: 4.769

Train R²: 0.599

Mean absolute error: 136.053
```

Ridge regression helped mitigate multicollinearity but didn't significantly boost R^2 compared to the linear model.

1.4 Lasso regression

Utilized Lasso regression to select features

```
from sklearn.linear_model import LassoCV

X = df_model.drop(["num_orders", "num_orders_log", "meal_id", "base_price", "discount", "price_diff",
"avg_demand_by_cuisine", "discount_ratio"], axis = 1)
y = df_model["num_orders_log"]

X, X_test, y, y_test = train_test_split(X, y, test_size = 0.2, random_state = 2024)

std = StandardScaler()
X_tr = std.fit_transform(X.values)*

n_alphas = 200
alphas = 10 ** np.linspace(-3, 3, n_alphas)

Lasso_model = LassoCV(alphas = alphas, cv = 5)

Lasso_model.fit(X_tr, y)
print(f"Alpha: {Lasso_model.alpha_}")
print(f"Train R²: {Lasso_model.score(X_tr, y)}")
print(f"Mean absolute error: {mae(np.exp(y), np.exp(Lasso_model.predict(X_tr)))}")
```

output:

```
Alpha: 0.001

Train R²: 0.598

Mean absolute error: 136.302
```

Lasso regression identified and removed non-influential features, simplifying the model but providing similar R^2 performance.

1.5 Random forest & Gradient boosting

Evaluated ensemble methods for improved performance

```
from sklearn.model_selection import cross_val_score
from sklearn.ensemble import RandomForestRegressor, GradientBoostingRegressor
from sklearn.metrics import make_scorer

X = df_model.drop(["num_orders", "num_orders_log", "meal_id", "base_price", "discount", "price_diff",
"avg_demand_by_cuisine", "discount_ratio"], axis = 1)
y = df_model["num_orders_log"]

X, X_test, y, y_test = train_test_split(X, y, test_size = 0.2, random_state = 2024)

scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)

cv = KFold(n_splits=5, shuffle=True, random_state=42)

# Random Forest Regressor
rf_model = RandomForestRegressor(n_estimators=100, random_state=42)
rf_scores = cross_val_score(rf_model, X_scaled, np.exp(y), cv=cv, scoring='r2')
print("Random Forest Regressor Cross-Validation R² Scores:", rf_scores)
print("Random Forest Regressor Mean R² Score:", np.mean(rf_scores))

# Gradient Boosting Regressor
gb_model = GradientBoostingRegressor(n_estimators=100, learning_rate=0.1, random_state=42)
gb_scores = cross_val_score(gb_model, X_scaled, np.exp(y), cv=cv, scoring='r2')
print("Gradient Boosting Regressor Cross-Validation R² Scores:", gb_scores)
print("Gradient Boosting Regressor Mean R² Score:", np.mean(gb_scores))
```

output:

```
Random Forest Regressor Cross-Validation R² Scores: [0.791 0.787 0.775 0.805 0.775]
Random Forest Regressor Mean R² Score: 0.786
Gradient Boosting Regressor Cross-Validation R² Scores: [0.616 0.610 0.606 0.625 0.596]
Gradient Boosting Regressor Mean R² Score: 0.611
```

Random Forest outperformed all other models, capturing complex relationships between features.
Gradient Boosting showed moderate improvement but required more fine-tuning.

Next after chosing the random forest model I fited the model and validated with test portion

```
# Fit the model on the training data
rf_model.fit(X_scaled, np.exp(y))

# Scale the test set
X_test_scaled = scaler.transform(X_test)
```

```python
# Make predictions on the test set
y_test_pred_exp = rf_model.predict(X_test_scaled)

# Calculate evaluation metrics
mae_test = mean_absolute_error(np.exp(y_test), y_test_pred_exp)
r2_test = r2(np.exp(y_test), y_test_pred_exp)

# Print the results
print("Test Set Mean Absolute Error (MAE):", mae_test)
print("Test Set R² Score:", r2_test)
```

output:

```
Test Set Mean Absolute Error (MAE): 75.527
Test Set R² Score: 0.809
```

2. Test Data Predictions

Feature Engineering for Test Data: Applied the same transformations to the test set

```python
df_test["discount"] = ((df_test['base_price'] - df_test['checkout_price']) / df_test['base_price']) * 100

df_test['discount_bin'] = pd.cut(df_test['discount'],
                        bins=[-np.inf, 0, 10, 20, 30, 50, np.inf],
                        labels=['No Discount', '0-10%', '10-20%', '20-30%', '30-50%', '50%+'])

price_variability = df_test.groupby('meal_id')['checkout_price'].std().rename('price_std')
df_test = df_test.merge(price_variability, on='meal_id', how='left')

year = 2023
start_date = pd.to_datetime(f'{year}-01-01')

df_test['date'] = start_date + pd.to_timedelta(df_test['week'] - 1, unit='W')

df_test['month'] = df_test['date'].dt.month
df_test['quarter'] = df_test['date'].dt.quarter

df_test.drop(columns='date', inplace=True)

df_test = df_test.assign(
    discount_ratio = df_test["discount"] / df_test["base_price"],
    price_diff = np.abs(df_test["base_price"] - df_test["checkout_price"]),
    price_discount_interaction = df_test['checkout_price'] * df_test['discount'],
    promotion_discount_interaction = df_test['emailer_for_promotion'] * df_test['discount'],
    homepage_discount_interaction = df_test['homepage_featured'] * df_test['discount'],
)

df_test = pd.get_dummies(df_test, drop_first = True)

bool_columns = df_test.select_dtypes(include='bool').columns

df_test[bool_columns] = df_test[bool_columns].astype(int)
df_test.drop("id", axis = 1, inplace = True)
```

Prediction

```
new_data_prepared = df_test.drop(["base_price", "checkout_price", "price_diff", "discount_ratio"], axis=1)

new_data_scaled = scaler.transform(new_data_prepared)

predictions = rf_model.predict(new_data_scaled)
```

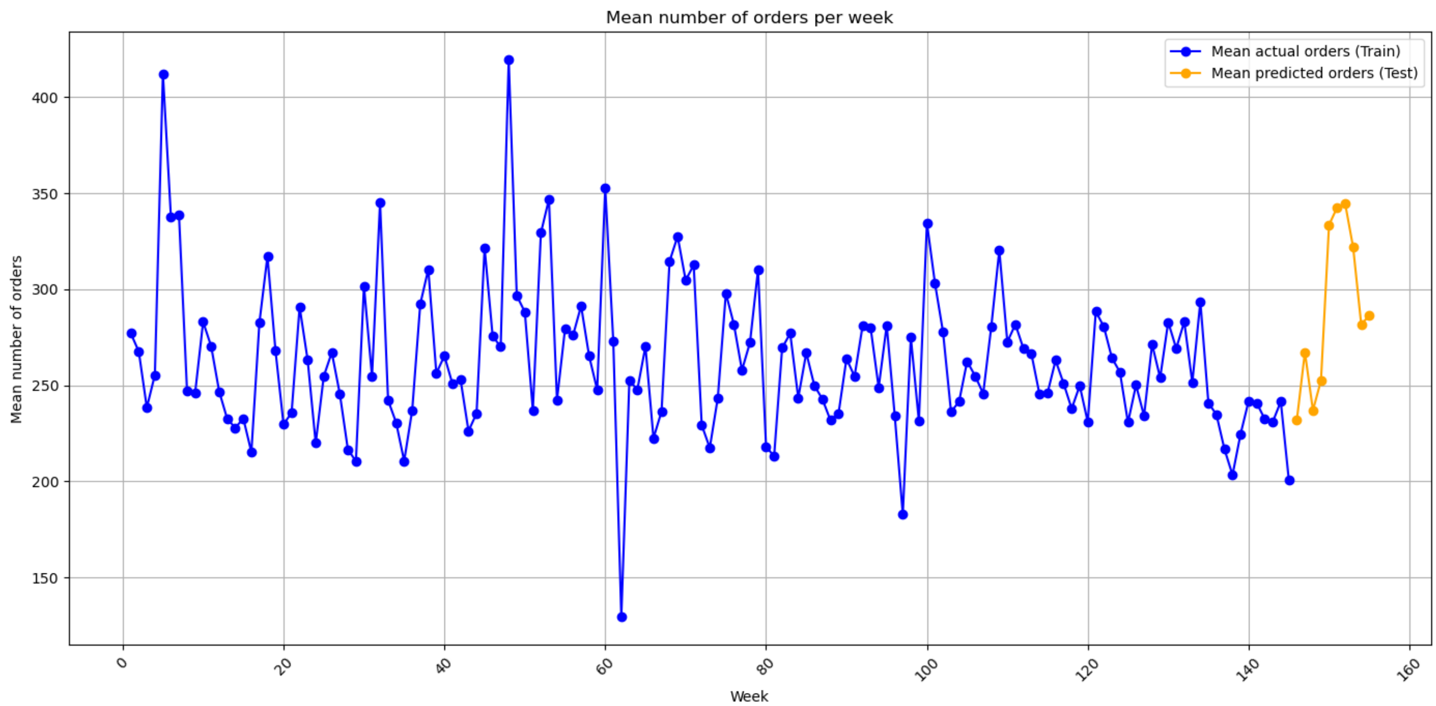
```
# Prepare actual values from training data
y_train_actual = np.exp(y)
df['num_orders'] = y_train_actual

# Calculate weekly means for training data
weekly_mean_train = df.groupby('week')['num_orders'].mean().reset_index()

# Calculate weekly means for predicted orders in test data
df_test['predicted_orders'] = predictions  # Add predictions to test DataFrame
weekly_mean_test = df_test.groupby('week')['predicted_orders'].mean().reset_index()
```

3. Visual Analysis

I visualized weekly average orders to compare actual versus predicted values

```
# Plotting
plt.figure(figsize=(14, 7))

# Plot the mean number of orders from training data
plt.plot(weekly_mean_train['week'], weekly_mean_train['num_orders'], marker='o', color='blue', label='Mean actual orders (Train)')

# Plot the mean predicted orders from test data
plt.plot(weekly_mean_test['week'], weekly_mean_test['predicted_orders'], marker='o', color='orange', label='Mean predicted orders (Test)')

# Plot customization
plt.title('Mean number of orders per week')
plt.xlabel('Week')
plt.ylabel('Mean number of orders')
plt.xticks(rotation=45)
plt.legend()
plt.grid()

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

output:



The visual analysis shows that:

1. Actual orders trend: The mean actual orders display significant fluctuations with some initial volatility, followed by a more stable pattern with periodic variations. A sharp dip occurs around week 60, likely influenced by external factors.
2. Predicted orders performance: The predictions align well with the overall trend of actual orders, showing consistency. However, the predictions are smoother, lacking the ability to fully capture sharp spikes or dips. The model indicates an upward trend in demand toward the end of the test period.
3. Model Insights: While the model captures general trends effectively, further improvements may be needed to account for sudden variations. Adding external factors or using more complex models could enhance accuracy.