Time Series Forecasting for a Local Restaurant's Demand Using Seasonal ARIMA And Holt-Winters Methods

Carmen María Pelayo Fernández, April 2024

1. Introduction

Demand forecasting plays a key role in every business, as it can dramatically help in optimizing costs. Restaurants particularly benefit from forecasting tools, as they deal with perishable goods that require rapid consumption. Other advantages include maximizing customer satisfaction, optimizing labor allocation or enhanced menu planning [1]. As a business graduate and current data science student, I am interested in exploring forecasting methods to assist in strategic planning. Therefore, I decided to use my time series knowledge to analyze a real dataset from a local café in Pomfret, CT.

2. Data set

I am in contact with the Vanilla Bean Café's [2] owner, who provided a raw dataset of their demand spanning over two decades (01/01/2000 to 12/31/2023). The set includes data about the number of daily breakfast, lunch and dinner servings, as well as the revenue earned and some notes about the weather, day of the week and general events. Figure 1 displays the dataset after some cleaning and preprocessing (see Appendix.1).

Date	Day	Breakfast	Lunch	Dinner	Total	Weather	Notes	Revenue	Avg Spend
2000-08-18	Friday	10	168	87	265.0	Overcast	NaN	2768.99	10.449019
2000-08-19	Saturday	57	211	115	383.0	Sunny	Terry Kitchen 12- People	4433.37	11.575379
2000-08-20	Sunday	113	328	92	533.0	Sunny	Beautiful Day	4875.03	9.146398
2000-08-21	Monday	6	167	0	173.0	Sunny	NaN	1883.77	10.888844
2000-08-22	Tuesday	8	167	0	175.0	Sunny	NaN	1809.92	10.342400

Figure 1. Vanilla Bean Café's dataset.

Some variables in the dataset are linearly dependent on others. For instance, the meal count in a day corresponds to the sum of breakfast, lunch, and dinner servings; and the average customer spending is obtained by dividing the revenue by the meal counts. Therefore, the most value from the forecasts could be obtained by choosing the total count of meals (i.e., 'Total') as the target variable.

3. Results

3.1. Visualizing and Evaluating the Data

To first understand the structure of the process, the evolution of servings over the years was plotted. Since the dataset provided daily counts, it had to be resampled to quarterly observations for better visualization and analysis. As a result, Figure 2 was obtained.

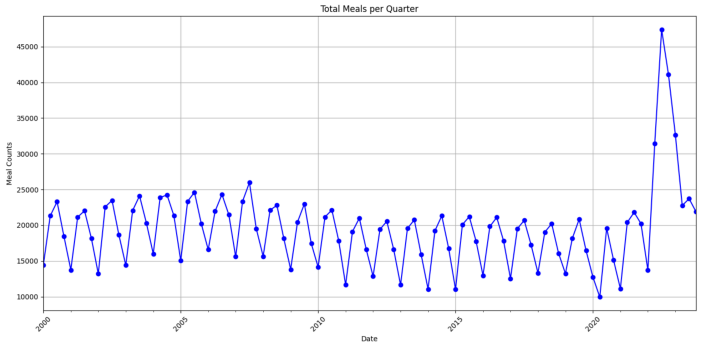


Figure 2. Total Meals per Quarter (before adjustments).

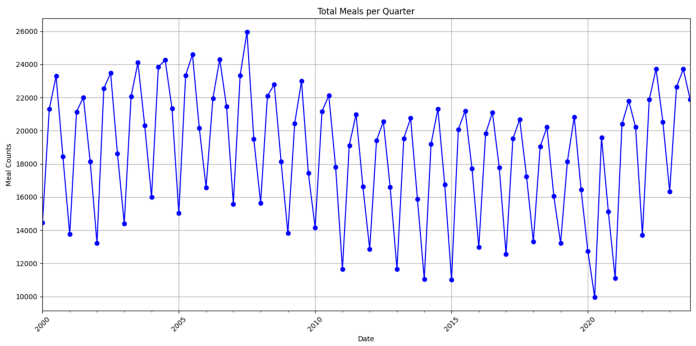


Figure 3. Total Meals per Quarter (after adjustments).

Interestingly, the process showed a clear outlier between the years 2022 and 2023. Checking for anomalies, I found the dataset concentrated 314 duplicates among those years, which were dropped. Additionally, there were 164 missing values, which seemed to be the result of the business being closed those days (e.g., on holidays) rather than data entry errors. To fix the series and prevent those missing values from affecting the pattern of the data, I used *spline interpolation* [3], which fits a flexible, curved line through the data points. These adjustments led to the series in Figure 3.

To identify its stationarity, the process was decomposed into its trend, seasonal component and residuals using additive decomposition (see Figure 4).

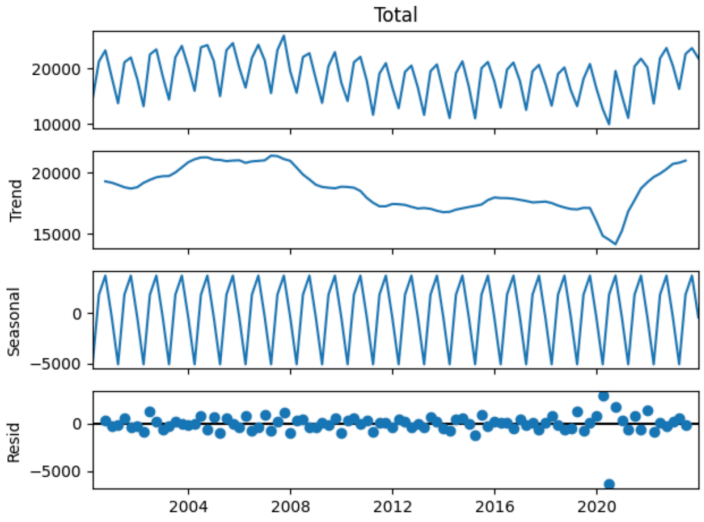


Figure 4. Classical Additive Decomposition.

The plot showed a varying trend —with values over 200,000 in the years between 2004 and 2008, and below 150,000 on 2020—, discarding the hypothesis of the process being stationary. Additionally, from the decomposition plot, a clear seasonality could be identified. Since the data was quarterly and there were twenty-three years of observations, the seasonal period was 4. To confirm for the non-stationarity of the process, the Augmented Dickey-Fuller test was conducted, returning a critical value of -1.58 (which was greater than the t-values at 1%, 5%, and 10% confidence intervals) and a p-value of 0.49. As a result, the null hypothesis of the process being non-stationary could not be rejected, thus confirming the process was non-stationary (see analysis in Appendix.2).

3.2. Transformations

In order to be properly forecasted, a time series process must become stationary first. With that intent, our series was seasonally differentiated once (see Appendix.3). After that, the critical value was -3.75, which was lower than the t-value at the 1% confidence interval (-3.51), thus guaranteeing the process' stationarity. The resulting process (Figure 5) shows a constant trend and a fairly-constant variance —except for the observations in 2021 and 2022, which will need to be accounted separately as a temporary intervention (likely due to the COVID-10 pandemic)—.

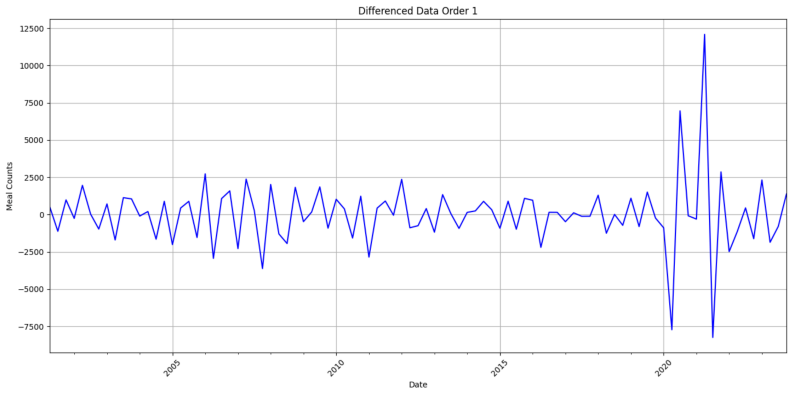


Figure 5. Differentiated Data Order 1.

3.3. Model-based Forecast: ARIMA

To forecast future values of the time series, I fitted a model-based seasonal ARIMA model (see Appendix.4). From the ACF and PACF plots (see Figures 6 and 7), it seems reasonable to consider a low-order ARMA model. The PACF cuts off near lag 2 decaying to 0 and the ACF cuts off after lag 2 decaying to 0. However, although there is an early cutoff, the lag significance returns around lag 4. This periodic pattern is a characteristic of seasonality with period 4 (stochastic part), confirming what was observed in the seasonal component from Figure 4.

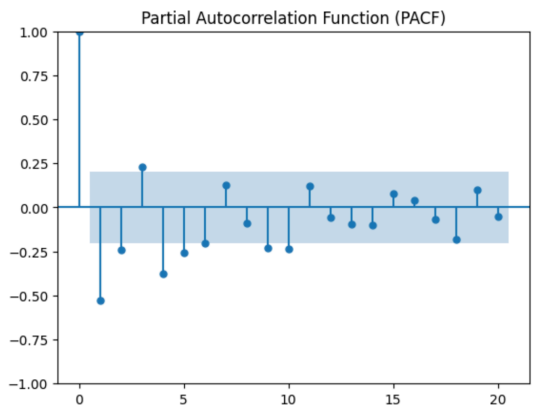


Figure 6. Partial Autocorrelation Function

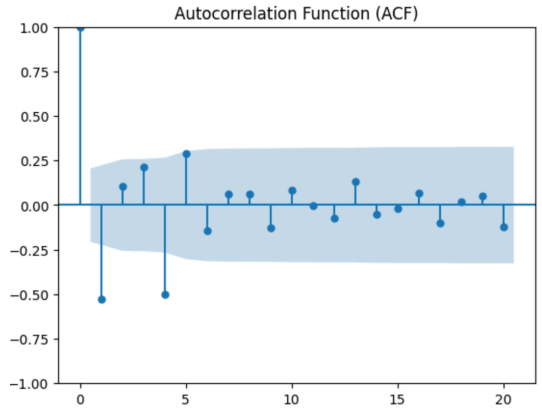


Figure 7. Autocorrelation Function

Model	AIC (without intervention)	AIC (with intervention)
ARMA (2,1,2) x SARMA (1,1,1)[4]	**1551.71***	1583.86
ARMA (2,1,1) x SARMA (1,1,1)[4]	1552.54	1550.94
ARMA (2,1,0) x SARMA (1,1,1)[4]	1573.81	1596.20
ARMA (1,1,2) x SARMA (1,1,1)[4]	1567.62	1543.98
ARMA (0,1,2) x SARMA (1,1,1)[4]	1554.08	**1543.20***
ARMA (2,1,2) x SARMA (0,1,1)[4]	1565.92	1567.82

Table 1. ARIMA Model Evaluation (With and Without Intervention).

For the initial model, I chose an ARMA (2,1,2) x SARMA (1,1,1)[4] model. First, for the non-seasonal ARMA(p, d, q) model, there is a decaying pattern in the ACF and PACF. Second, for the seasonal ARMA(P, D, Q) model, there is strong autocorrelation and strong partial autocorrelation around lag 4 and both cutoff decaying to 0. As there are multiple interpretations to take, I generated six additional variations and compared performance. Although the different models had roughly comparable AIC (Akaike Information Criterion [4]) values (see Table 1), the initial model had the smallest one, and thus was used for forecasting the time series. Additionally, I evaluated a set of ARIMA models accounting for the intervention described earlier. The ARMA (0,1,2) x SARMA (1,1,1)[4] was found to have the lowest AIC, so it was thus chosen as

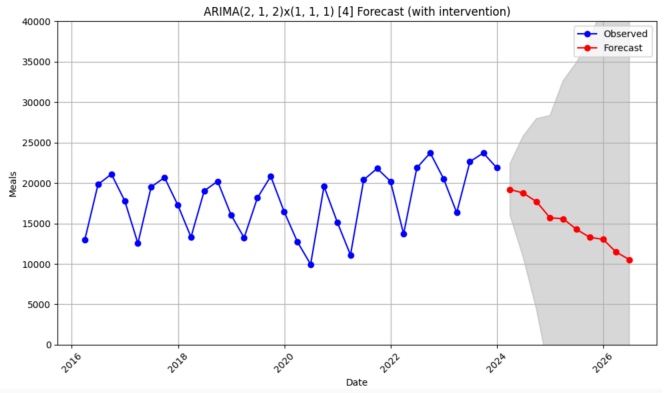


Figure 8. ARIMA Forecast (with intervention).

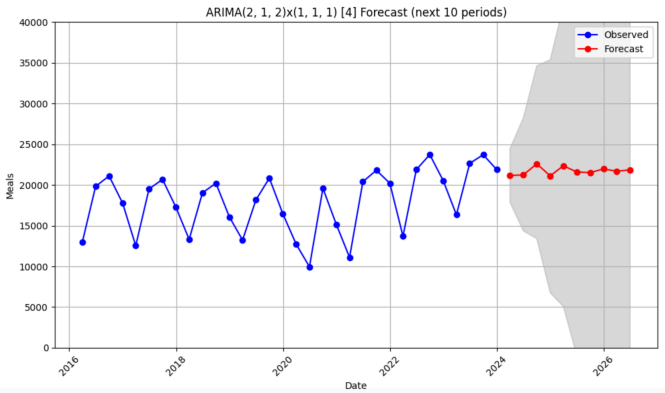


Figure 9. ARIMA Forecast (without intervention).

the model accounting for the intervention in 2020 and 2021. To get the final predictions, the results of the forecast had to be integrated, as the model had been fitted on differentiated data (see resulting forecasts, with and without accounting the effect of the intervention in 2020 and 2021 in Figures 8 and 9).

3.4. Model Diagnostics

By analyzing the model diagnostics, we can check whether the models selected (ARMA (2,1,2) x SARMA (1,1,1)[4] not accounting for the intervention, and ARMA (0,1,2) x SARMA (1,1,1)[4] accounting for the intervention) are the one that best fitted our time series. We can see in Figures 10 and 11 that:

- The standardized residuals are centered around zero, and do not display obvious patterns.
- The histograms of the standardized residuals follow a normal distribution.
- The Quantile-Quantile plots show that the residuals are coming close from a normal population.
- The ACFs of the residuals (as shown in the *Correlograms*) do not show significant correlations.

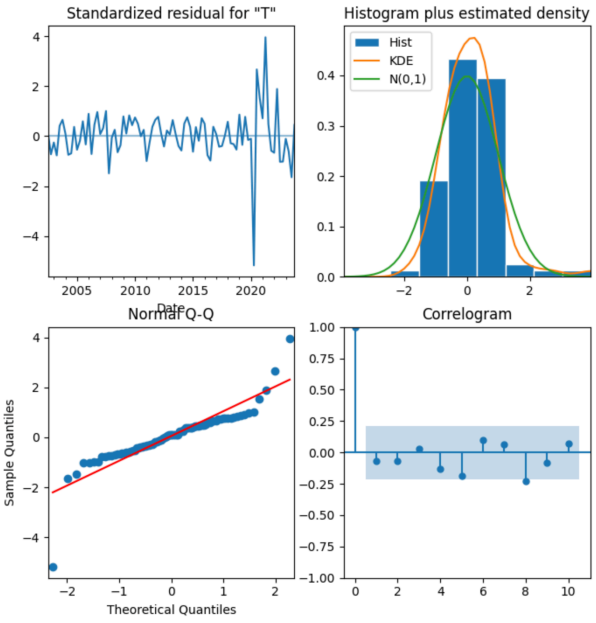


Figure 10. Model Without Intervention Diagnostics.

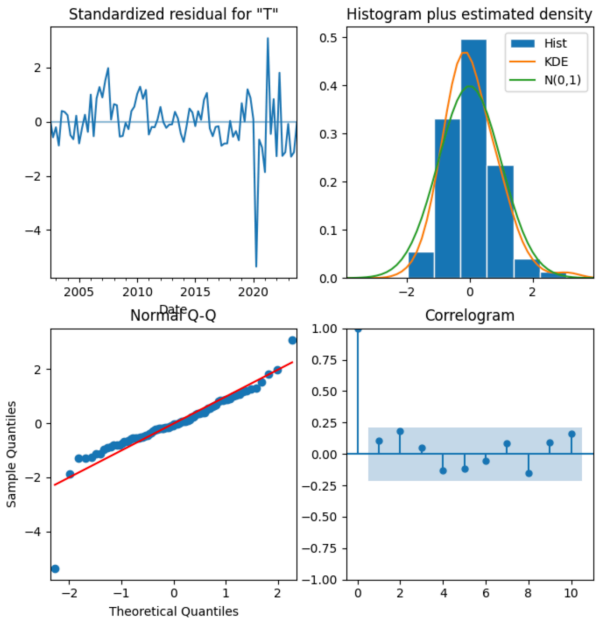


Figure 11. Model With Intervention Diagnostics.

This confirmed that both models fitted well the process. It can also be observed that the model accounting for the intervention (Fig. 11) has a slightly better behavior, as the residuals follow a more normal distribution than in the model not accounting for the intervention, and all its autocorrelations lay within the significance interval.

3.5. Smoothing-based Forecast: Holt-Winters

Another widely used and successful forecasting method is the seasonal Holt-Winters method. The smoothing-based method uses the pattern of the data to extrapolate the forecast using double exponential smoothing. I performed the additive version, as the seasonal pattern remained roughly the same for the range of the data. Additionally, I applied the *Basin-Hopping* optimizer, as it returned the most accurate results during validation when compared against others optimizers, like *Powell's Method* [5] (see Appendix 5.1 and 5.2). The resulting forecast looks reasonable —it is smooth as expected (see Figure 12).

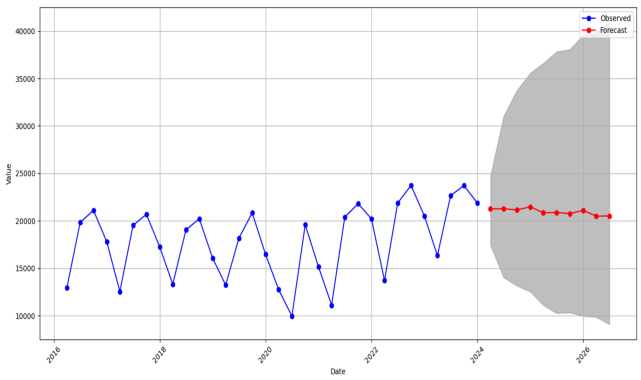


Figure 12. Holt-Winters Forecast (next 10 periods).

3.6. Forecast Evaluation

To evaluate the accuracy of the fitted forecasting models, I used the Mean Absolute Error (MAE) and Mean absolute Percentage Error (MAPE) metrics. I performed an out-of-sample forecast validation to compare the models, using validation samples of different sizes so that the effect of the observed intervention (all observations in 2020 and 2021) in different scenarios could be evaluated. The results can be consulted in Table 3 (see complete evaluation in Appendix 4.1.2, 4.2.2 and 5.2).

- 4 most recent observations (4% of total data approx.): observations from 03/31/2023 to 12/31/2023.
- 6 most recent observations (6% of total data approx.): observations from 09/30/2022 to 12/31/2023.
- 12 most recent observations (13% of total data approx.): observations from 03/31/2021 to 12/31/2023.

Validation Set Size	Model	MAE	RMSE	MAPE
4	ARIMA with intervention	5067.099	5611.257	0.236
	ARIMA without intervention	5415.258	5657.276	0.255
	Holt-Winters	**2505.577**	**2934.481**	**0.130**
6	ARIMA with intervention	10065.404	10875.456	0.492
	ARIMA without intervention	**1790.191**	2793.624	**0.097**
	Holt-Winters	2087.960	**2566.606**	0.106
12	ARIMA with intervention	3866.174	4573.490	0.224
	ARIMA without intervention	**3461.997**	**4522.323**	**0.219**
	Holt-Winters	3675.649	4668.942	0.222

Table 3. Forecasting Models Evaluation.

From the results in Table 3 we can observe that:

- The Holt-Winters method performs significantly better than any ARMA method (13% vs. 23-25% MAPEs) when the validation set is small (i.e., the training set covers most observations in the original dataset).
- Interestingly, when the validation set is close to (but not covering) the intervention (i.e., 6 most recent observations), the ARMA model *not accounting for the intervention* performs dramatically better than the ARMA model *accounting for the intervention* (9.7% vs. 49.2% MAPE).
- In general, the model being the most stable and accurate in all scenarios is the Holt-Winters forecast (as can be seen with all MAPEs being in the range of 10% to 22% in all tests).

4. Conclusions

The goal of this time series analysis was to forecast the meals that would be served in a local restaurant in the next few quarters of year, for strategic planning purposes. Given a raw, non-stationary dataset on daily meal counts spanning from 2000 to 2023, I cleaned, resampled it to quarterly data, transformed it into stationary via differencing, and encoded it to account for the intervention that occurred in the series between 2020 and 2021 due to the COVID-19 pandemic. Then, I chose the suitable forecasting method by considering the AIC values among the ARIMA models (with and without accounting for the intervention observed) and the RMSE, MAE, and MAPE when comparing them against the Holt-Winters method (using Basin-Hoppin optimization). The results of testing the accuracy of the models over a validation sample of 4, 6, and 12 periods showed that the seasonal Holt-Winters method (which does not account for the intervention) is the most plausible forecasting method for restaurant servings, as it returns the most stable and accurate results overall. This analysis could be extended to account for other socioeconomic and environmental variables influencing customer decisions, such as the weather or price levels.

References

[1] https://www.5out.io/post/demand-forecasting-importance-for-restaurants#:~:text=By forecasting demand, restaurant owners can gain a better understanding,planning and optimize their spending

[2] https://thevanillabeancafe.com/

[3] https://www.geeksforgeeks.org/cubic-spline-interpolation/

[4] https://www.sciencedirect.com/topics/earth-and-planetary-sciences/akaike-information-criterion

[5] https://www.statsmodels.org/stable/optimization.html

Appendix

April 29, 2024

[1]:
```
import pandas as pd
import numpy as np
import seaborn as sns
import matplotlib.pyplot as plt
import meteostat
import datetime as dt
```

0.1 Appendix.1: Exploratory Data Analysis & Data Preprocessing

I will begin the project with a Exploratory Data Analysis performing some simple **statistics** and **visualizations** to observe the nature of the data and gain insights about it.

[2]:
```
data = pd.read_excel("VBCData.xlsx")
data
```

[2]:

	Unnamed: 0	Unnamed: 1	Date	Day	Breakfast	Lunch	Dinner	Total	Weather	Notes	Unnamed: 10	Avg Spend
0	NaN	NaN	2023-12-31	Sunday	88	105	0	193	Overcast	NaN	3649.27	18.908135
1	NaN	NaN	2023-12-30	Saturday	56	149	28	233	PM Rain	NaN	4643.05	19.927253
2	NaN	NaN	2023-12-29	Friday	30	154	17	201	PM Rain	NaN	3715.30	18.484080
3	NaN	NaN	2023-12-28	Thursday	26	154	21	201	PM Rain	NaN	3677.01	18.293582
4	NaN	NaN	2023-12-27	Wednesday	43	138	13	194	PM Rain	NaN	3239.03	16.696031
…	…	…	…	…	…	…	…	…	…	…	…	…
9074	NaN	NaN	2000-01-05	Wednesday	8	110	20	138	NaN	NaN	1320.27	9.567174
9075	NaN	NaN	2000-01-04	Tuesday	5	92	0	97	NaN	NaN	1077.91	11.112474
9076	NaN	NaN	2000-01-03	Monday	3	79	0	82	Part Sunny	Warm	840.04	10.244390
9077	NaN	NaN	2000-01-02	Sunday	55	161	29					
9078	NaN	NaN	2000-01-01	Saturday	2	91	0					

```
9077    245         NaN                    NaN      2259.14   9.220980
9078     93       Sunny  Open 11-5 Slow Start      1064.21  11.443118

[9079 rows x 12 columns]
```

The following actions were taken to preprocess the data before we begin the analysis: * Dropped unnecessary columns. * Named column 'Revenue'. * Checked data types.

[3]:
```
data = data.drop(columns=["Unnamed: 0", "Unnamed: 1"]).rename(columns={"Unnamed:
  ↪ 10": "Revenue"})
data.dtypes
```

[3]:
```
Date          datetime64[ns]
Day                   object
Breakfast              int64
Lunch                  int64
Dinner                 int64
Total                  int64
Weather               object
Notes                 object
Revenue              float64
Avg Spend            float64
dtype: object
```

All columns seem to have the proper data type. Therefore, no formatting is required here. The preprocessed dataset to be used will be the following:

[4]:
```
data.describe()
```

[4]:
```
                                Date   Breakfast         Lunch       Dinner  \
count                           9079  9079.000000  9079.000000  9079.000000
mean   2012-05-15 19:43:12.840621312    31.088115   133.909131    37.508316
min              2000-01-01 00:00:00     0.000000     0.000000     0.000000
25%              2006-03-19 12:00:00     8.000000    87.000000     0.000000
50%              2012-06-05 00:00:00    15.000000   119.000000    32.000000
75%              2018-08-22 12:00:00    40.000000   162.000000    59.000000
max              2023-12-31 00:00:00   222.000000   585.000000   209.000000
std                              NaN    36.934156    72.448446    37.483778

              Total       Revenue    Avg Spend
count  9079.000000   9070.000000  8915.000000
mean    202.517678   2985.114173    14.730858
min       0.000000      0.000000     0.000000
25%     114.000000   1561.997500    12.609058
50%     172.000000   2508.100000    14.318143
75%     260.500000   3929.772500    16.126611
max     821.000000  14833.490000   568.090000
std     123.561512   1961.484184     6.894470
```

From this summary table we can observe there are **missing values.** The dataset contains records from 9,079 days (i.e. 9078/366 = 24.8 years). Interestingly, we only have 'Avg Spend' (average expenditure) data from 8915 days. That means 164 values are missing and should be handled appropriately. Similarly, we only have 'Revenue' data from 9070 days, meaning 9 values are missing. Missing values should either be dropped or filled in with interpolated data to prevent the impact of null values on predictions. Given the time series nature of the problem, dropping values would discontinue the series and this could cause problems. Therefore, we will fill in the missing values. There are multiple methods to do so, including the *mean imputation*, *median imputation*, *Last Observation Carried Forward (LOCF)*, *Next Observation Carried Backward (NOCB)*, *linear interpolation* or the *spline imputation* (GeeksForGeeks, accessed March 2024). Given that there are only a small number of missing values, we can perform **spline interpolation** on them, which is the most computationally expensive method but also the most accurate for capturing complex trends and subtle changes in time series data. It estimates missing values by fitting a flexible, curved line through the data points.

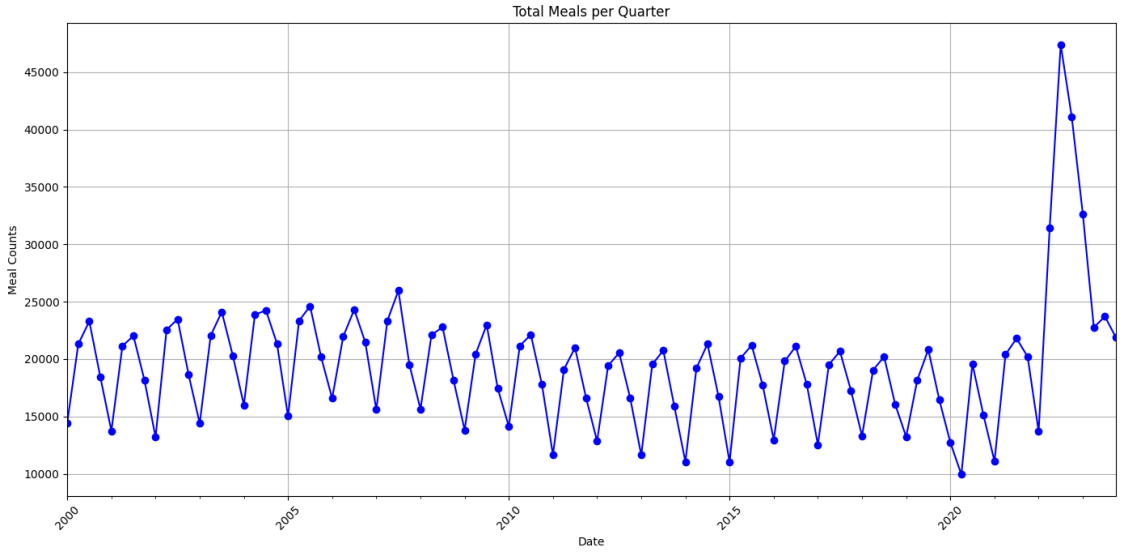
[5]:
```
data["Total"] = data["Total"].replace(0, np.nan).interpolate(option='spline')
```

By doing this, I am **dealing with the days where the restaurant was closed** (i.e., the count of meals served was 0), so that it does not impact predictions.

0.1.1 Demand Evolution

The **total demand per quarter** will now be plotted to observe how meal servings have evolved over the years.

[6]:
```
# Data Resampling (to obtain quarterly observations)
data.set_index('Date', inplace=True)
quarterly_data = data['Total'].resample('QE').sum()

# Plotting
plt.figure(figsize=(14, 7))
quarterly_data.plot(marker='o', linestyle='-', color='b')
plt.title('Total Meals per Quarter')
plt.xlabel('Date')
plt.ylabel('Meal Counts')
plt.grid(True)
plt.xticks(rotation=45)
plt.tight_layout()
plt.show()
```



The data seems to have a clear **yearly seasonality**. Throughout the entire series, we can see that the **first quarter** of the year (i.e. the months of January, February, March) produces the **lowest earnings** in the year, with the second (April, May, June) and third (July, August, September) quarters (the **warmer months**) being the **most profitable** ones.

0.1.2 Outliers and Data Duplicates

An **outlier during 2022** can be easily observed. This year, revenue almost reached 3 times the value of other years' revenue. This could be due to either the effect of external factors or issues in the data collection process. To check for this, we will first check for **duplicated records** in the data:

[7]:
```
duplicate_dates = data[data.index.duplicated()]
duplicate_dates
```

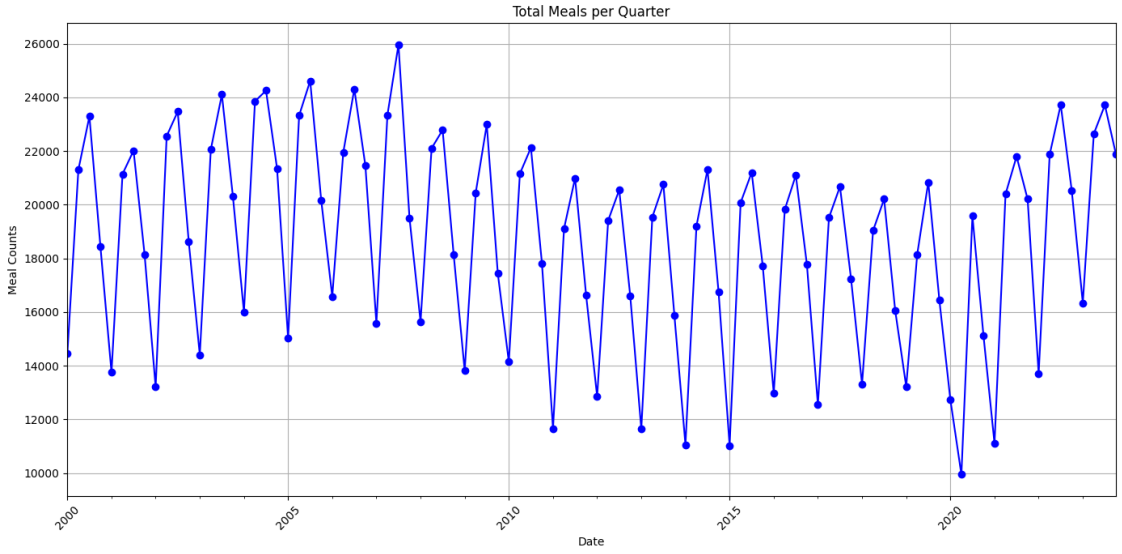
[7]:

Date	Day	Breakfast	Lunch	Dinner	Total	Weather	Notes	\
2023-04-25	Tuesday	21	84	0	105.0	Cold As F	NaN	
2023-03-31	Friday	25	112	59	196.0	NaN	NaN	
2023-03-30	Thursday	36	99	25	160.0	NaN	NaN	
2023-03-29	Wednesday	8	107	25	140.0	NaN	NaN	
2023-03-28	Tuesday	12	87	0	99.0	NaN	NaN	
…	…	…	…	…	…	…	…	
2022-05-27	Friday	21	133	95	249.0	NaN	NaN	
2022-05-26	Thursday	24	154	40	218.0	NaN	NaN	
2022-05-25	Wednesday	17	140	52	209.0	Beautiful	NaN	
2022-05-24	Tuesday	13	100	0	113.0	NaN	NaN	
2022-05-23	Monday	22	120	0	142.0	NaN	NaN	

```
            Revenue  Avg Spend
Date
2023-04-25  2127.22  20.259238
2023-03-31  3760.30  19.185204
2023-03-30  3238.70  20.241875
2023-03-29  2470.46  17.646143
2023-03-28  1858.06  18.768283
…              …         …
2022-05-27  4586.17  18.418353
2022-05-26  4139.37  18.987936
2022-05-25  3877.41  18.552201
2022-05-24  1958.33  17.330354
2022-05-23  2353.95  16.577113

[314 rows x 9 columns]
```

As suspected, there are **314 duplicated values** in the dataset. We will then proceed to remove them and replot the to quarterly revenues:

[8]:
```
data = (data.reset_index().drop_duplicates(subset='Date', keep='last').
  ↪set_index('Date').sort_index())
quarterly_meals = data['Total'].resample('QE').sum()

plt.figure(figsize=(14, 7))
quarterly_meals.plot(marker='o', linestyle='-', color='b')
plt.title('Total Meals per Quarter')
plt.xlabel('Date')
plt.ylabel('Meal Counts')
plt.grid(True)
plt.xticks(rotation=45)
plt.tight_layout()
plt.show()
```



This seems much more reasonable. With a clean dataset now, we can now proceed to perform the analysis and forecast.

0.2 Appendix.2: Stationarity

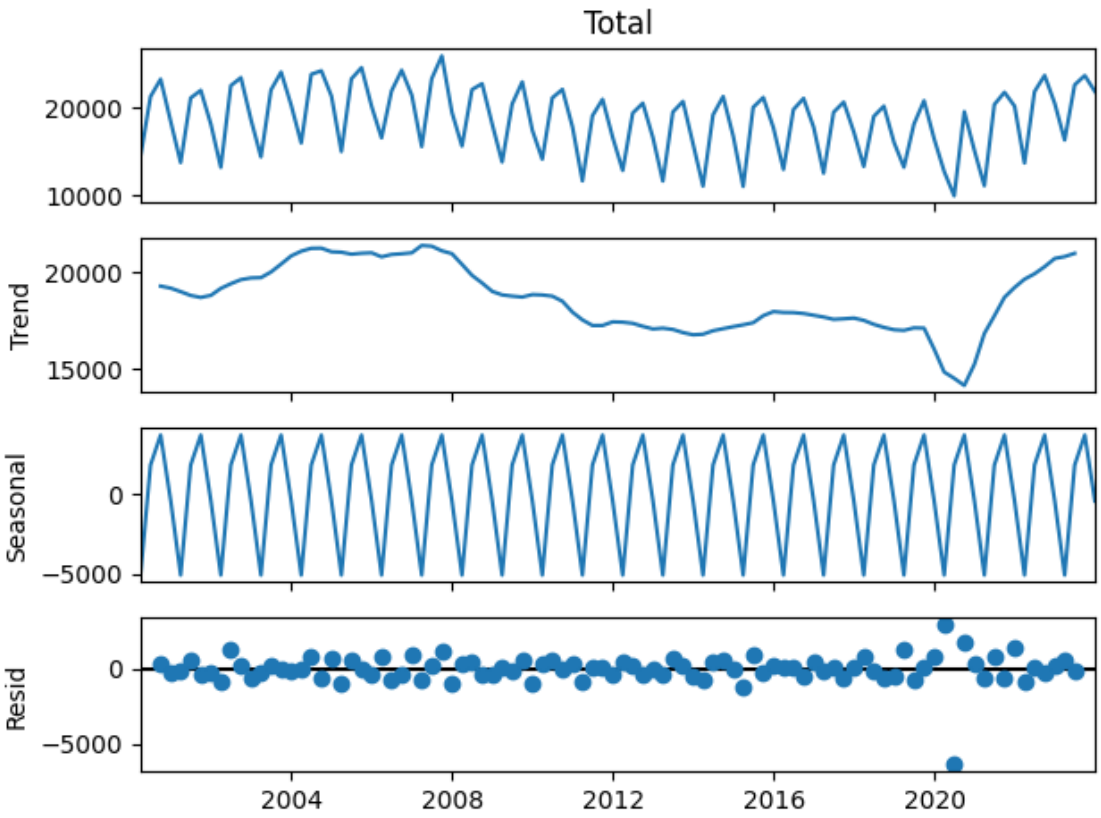
[9]:
```
from statsmodels.tsa.stattools import adfuller
from statsmodels.tsa.statespace.tools import diff
```

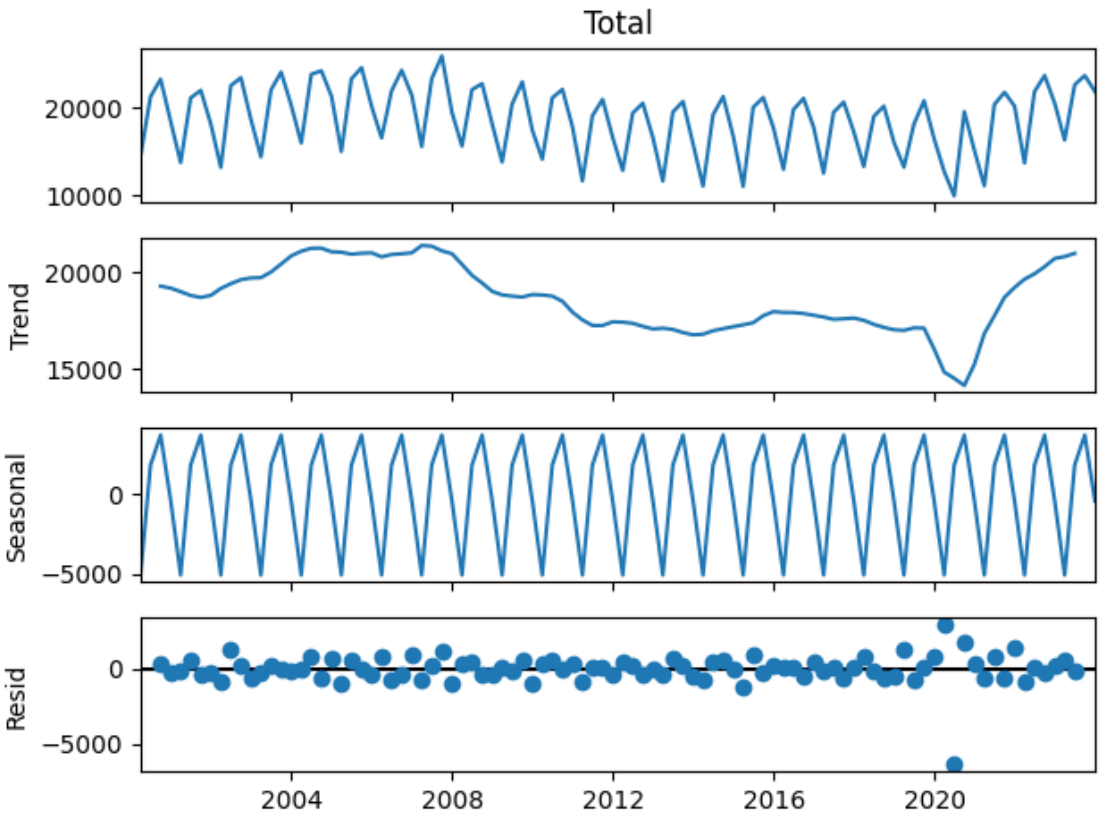
0.2.1 Classic Additive Decomposition

[10]:
```
import statsmodels
from statsmodels.tsa.seasonal import seasonal_decompose
```

[11]:
```
decomp_quarterly = statsmodels.tsa.seasonal.seasonal_decompose(quarterly_meals)
decomp_quarterly.plot()
```

[11]:


Total
20000
10000
Trend
20000
15000
Seasonal
0
−5000
Resid
0
−5000
2004
2008
2012
2016
2020



0.2.2 Augmented Dickey-Fuller Test

- **Null hypothesis**: Non-Stationarity exists in the series.
- **Alternative Hypothesis**: Stationarity exists in the series.

[12]:
```
adfuller(quarterly_meals)
```

[12]:
```
(-1.5826234123882856,
 0.49231620040665813,
 11,
 84,
 {'1%': -3.510711795769895,
  '5%': -2.8966159448223734,
  '10%': -2.5854823866213152},
 1475.1346689627142)
```

1. -1.5826234123882856 –> Critical value of the data.
2. 0.49231620040665813 –> Probability that null hypothesis will not be rejected(p-value)
3. 11 –> Number of lags used in regression to determine t-statistic.
4. 84 –> Number of observations used in the analysis.
5. {'1%': -3.510711795769895, '5%': -2.8966159448223734, '10%': -2.5854823866213152} –> T values corresponding to adfuller test. Since critical values -1.58 > -3.5, -2.8, -2.5 (t-values

at 1%, 5%, and 10% confidence intervals), the null hypothesis cannot be rejected. So there is non-stationarity in the data. Also p-value of $0.49 > 0.05$(if we take a 5% significance level or 95% confidence interval), the null hypothesis cannot be rejected. Hence the **data is nonstationary** (that means it has a relation with time).

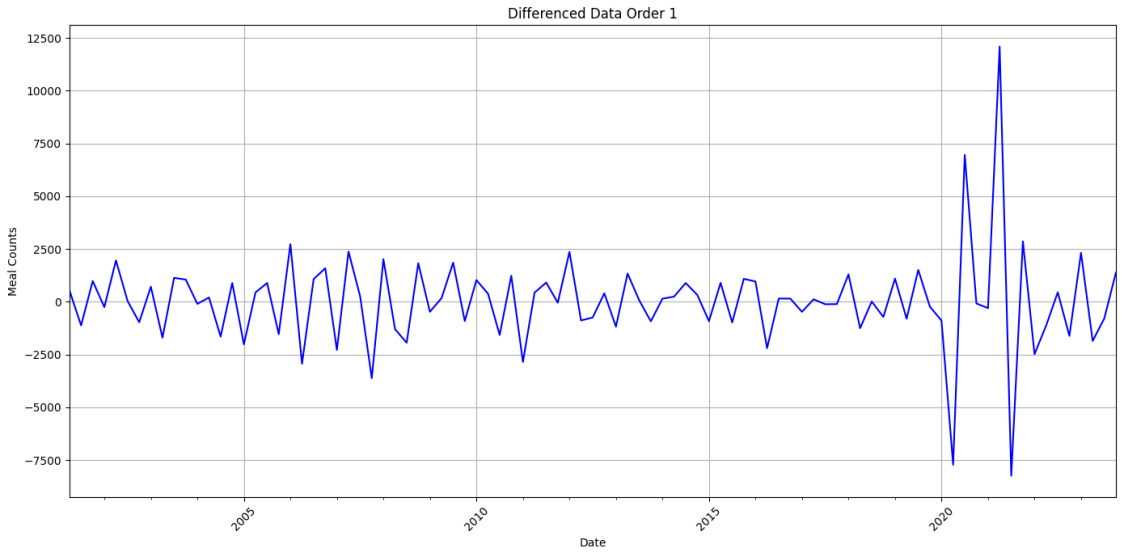
0.3 Appendix.3: Transformations

Since it is nonstationary, we can apply a **differencing transformation** to the data:

[13]:
```
diff_quarterly1 = diff(quarterly_meals, k_diff=1, k_seasonal_diff=1,␣
↪seasonal_periods=4)
adfuller(diff_quarterly1)
```

[13]:
```
(-3.7530530338166477,
 0.0034258692145842785,
 10,
 80,
 {'1%': -3.5148692050781247, '5%': -2.8984085156250003, '10%': -2.58643890625},
 1390.005167608742)
```

[14]:
```
plt.figure(figsize=(14, 7))
diff_quarterly1.plot(linestyle='-', color='b')
plt.title('Differenced Data Order 1')
plt.xlabel('Date')
plt.ylabel('Meal Counts')
plt.grid(True)
plt.xticks(rotation=45)
plt.tight_layout()
plt.show()
```

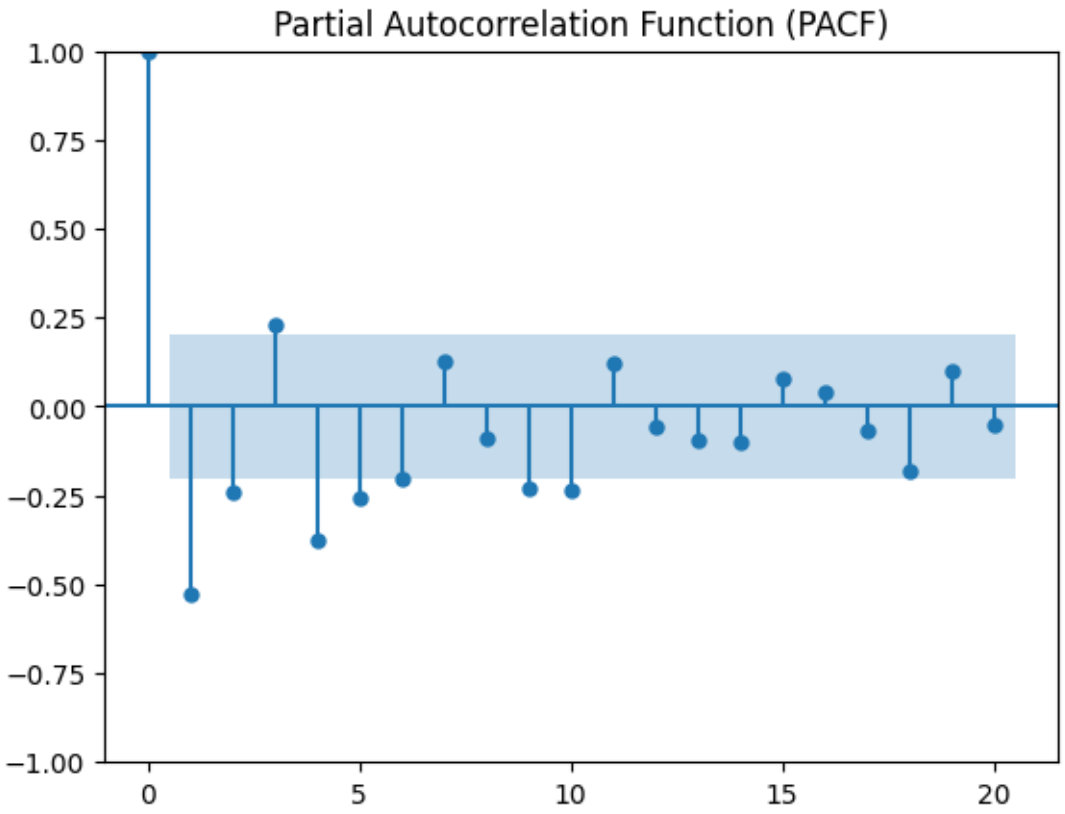


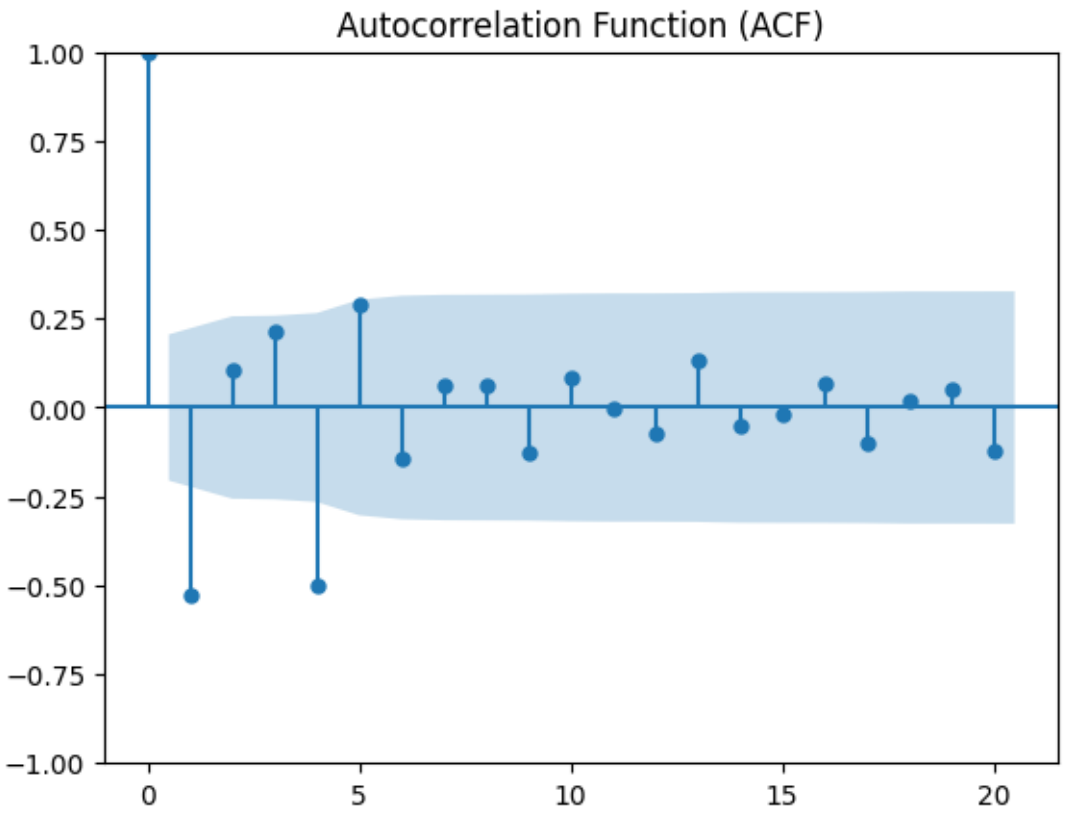
0.4 Appendix.4: Auto-Regressive Integrated Moving Average (ARIMA)

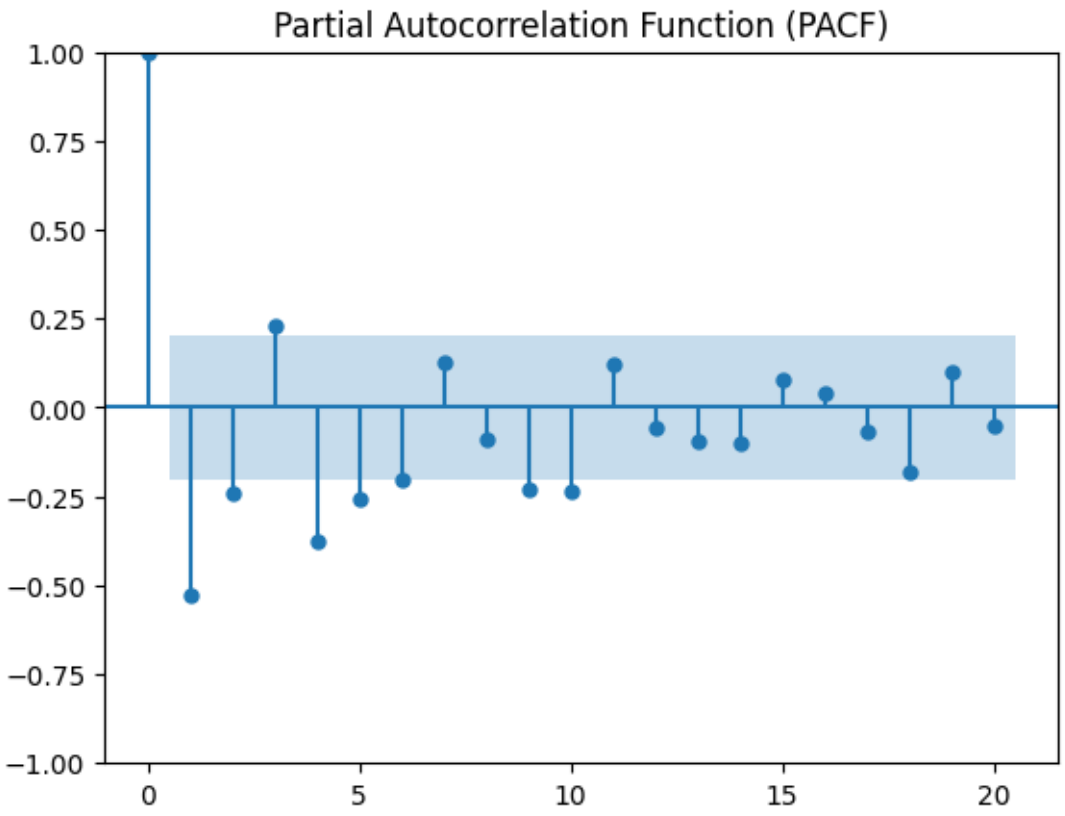
[15]:
```
from statsmodels.graphics.tsaplots import plot_pacf, plot_acf
from pmdarima import auto_arima
from statsmodels.tsa.arima.model import ARIMA
from sklearn.metrics import mean_squared_error
from math import sqrt
```

[16]:
```
plot_acf(diff_quarterly1, title='Autocorrelation Function (ACF)')
plot_pacf(diff_quarterly1, title='Partial Autocorrelation Function (PACF)')
```

[16]:




Autocorrelation Function (ACF)
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00
0
5
10
15
20



I will use the `auto_arima` function from the `pmdarima` package to **automatically estimate the proper order** of the ARIMA model we will fit afterwards.

[17]:
```
#arima_order = auto_arima(diff_quarterly1, seasonal=True, trace=True,␣
↪stationary=True)
```

The `auto_arima` function returned very high orders, so I will test different orders manually instead.

0.4.1 Appendix 4.1: Model without intervention

- **ARMA (2,1,2) x SARMA (1,1,1)[4] –> 1551.71**
- ARMA (2,1,1) x SARMA (1,1,1)[4] –> 1552.54
- ARMA (2,1,0) x SARMA (1,1,1)[4] –> 1573.81
- ARMA (1,1,2) x SARMA (1,1,1)[4] –> 1567.62
- ARMA (0,1,2) x SARMA (1,1,1)[4] –> 1554.08
- ARMA (2,1,2) x SARMA (0,1,1)[4] –> 1565.92
- ARMA (2,0,2) x SARMA (0,1,1)[4] –> 1563.52
- ARMA (0,0,5) x SARMA (0,0,0)[4] –> 1597.16

Appendix 4.1.1: Model fit.

[18]:
```
from statsmodels.tsa.statespace.sarimax import SARIMAX
```

```python
# Params
p = 2
d = 1
q = 2

# Seasonal params
P = 1
D = 1 # stable seasonal pattern
Q = 1
s = 4 # quarterly data

sarimax = SARIMAX(diff_quarterly1,
                  order=(p,d,q),
                  seasonal_order=(P,D,Q,s))

results = sarimax.fit(maxiter=1000)
```

```
RUNNING THE L-BFGS-B CODE

           * * *

Machine precision = 2.220D-16
 N =            7     M =           10

At X0         0 variables are exactly at the bounds

At iterate    0    f=  8.84902D+00    |proj g|=  1.44724D-01

At iterate    5    f=  8.78383D+00    |proj g|=  7.14758D-03

At iterate   10    f=  8.78322D+00    |proj g|=  1.33650D-03

At iterate   15    f=  8.78287D+00    |proj g|=  1.16053D-03

At iterate   20    f=  8.78286D+00    |proj g|=  3.38762D-04

At iterate   25    f=  8.78225D+00    |proj g|=  6.70825D-03

At iterate   30    f=  8.67416D+00    |proj g|=  3.92718D-01

At iterate   35    f=  8.50697D+00    |proj g|=  3.38210D-02
 This problem is unconstrained.

At iterate   40    f=  8.48230D+00    |proj g|=  7.05250D-04

At iterate   45    f=  8.45254D+00    |proj g|=  3.40408D-02
```

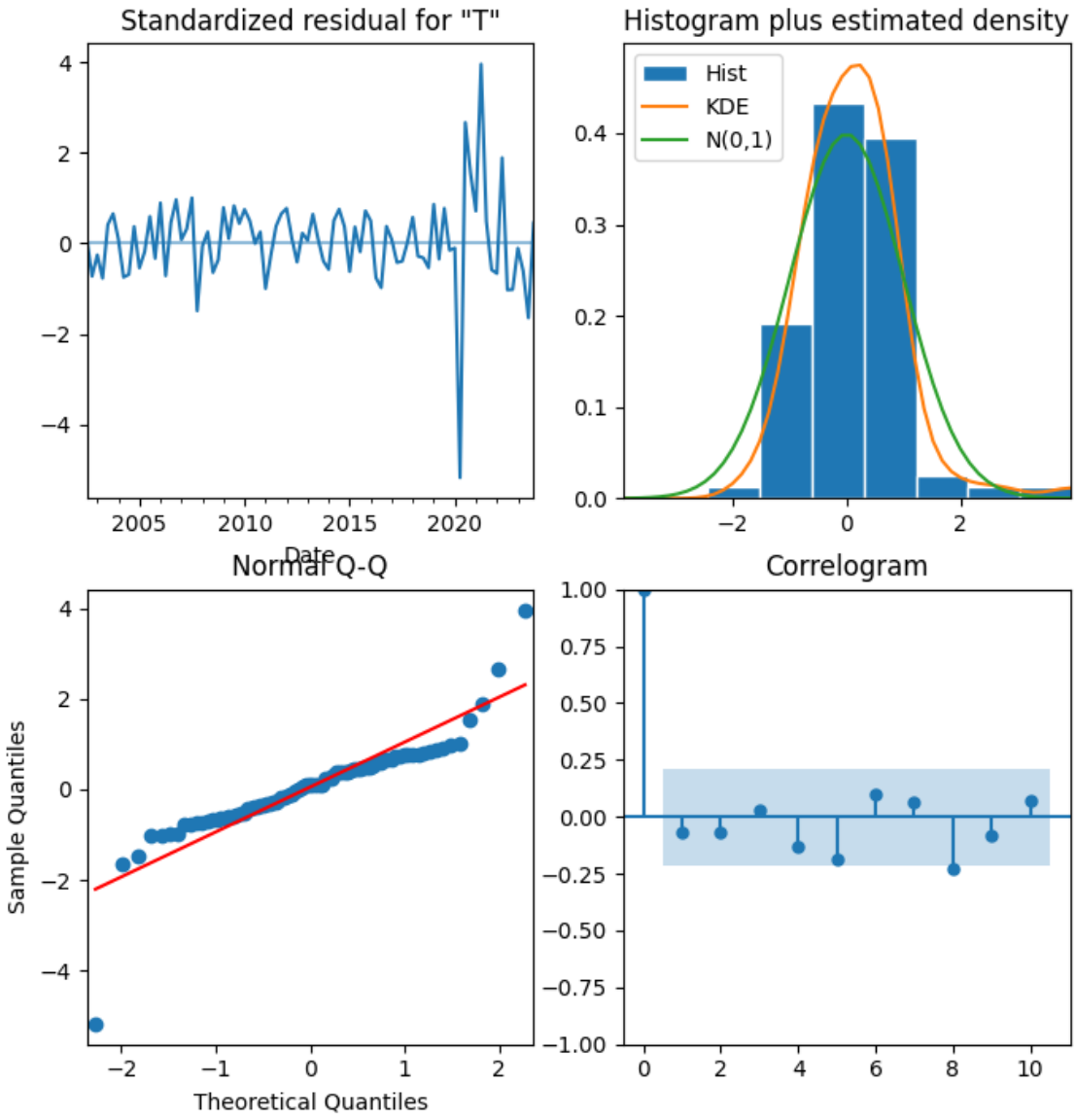
```
At iterate   50    f=  8.44897D+00    |proj g|=  1.49686D-05

At iterate   55    f=  8.44897D+00    |proj g|=  5.67305D-04

At iterate   60    f=  8.44896D+00    |proj g|=  6.32362D-05

           * * *

Tit   = total number of iterations
Tnf   = total number of function evaluations
Tnint = total number of segments explored during Cauchy searches
Skip  = number of BFGS updates skipped
Nact  = number of active bounds at final generalized Cauchy point
Projg = norm of the final projected gradient
F     = final function value

           * * *

   N    Tit     Tnf  Tnint  Skip  Nact     Projg        F
    7     61     93      1     0     0   5.784D-05   8.449D+00
  F =   8.4489636537938253

CONVERGENCE: REL_REDUCTION_OF_F_<=_FACTR*EPSMCH
```

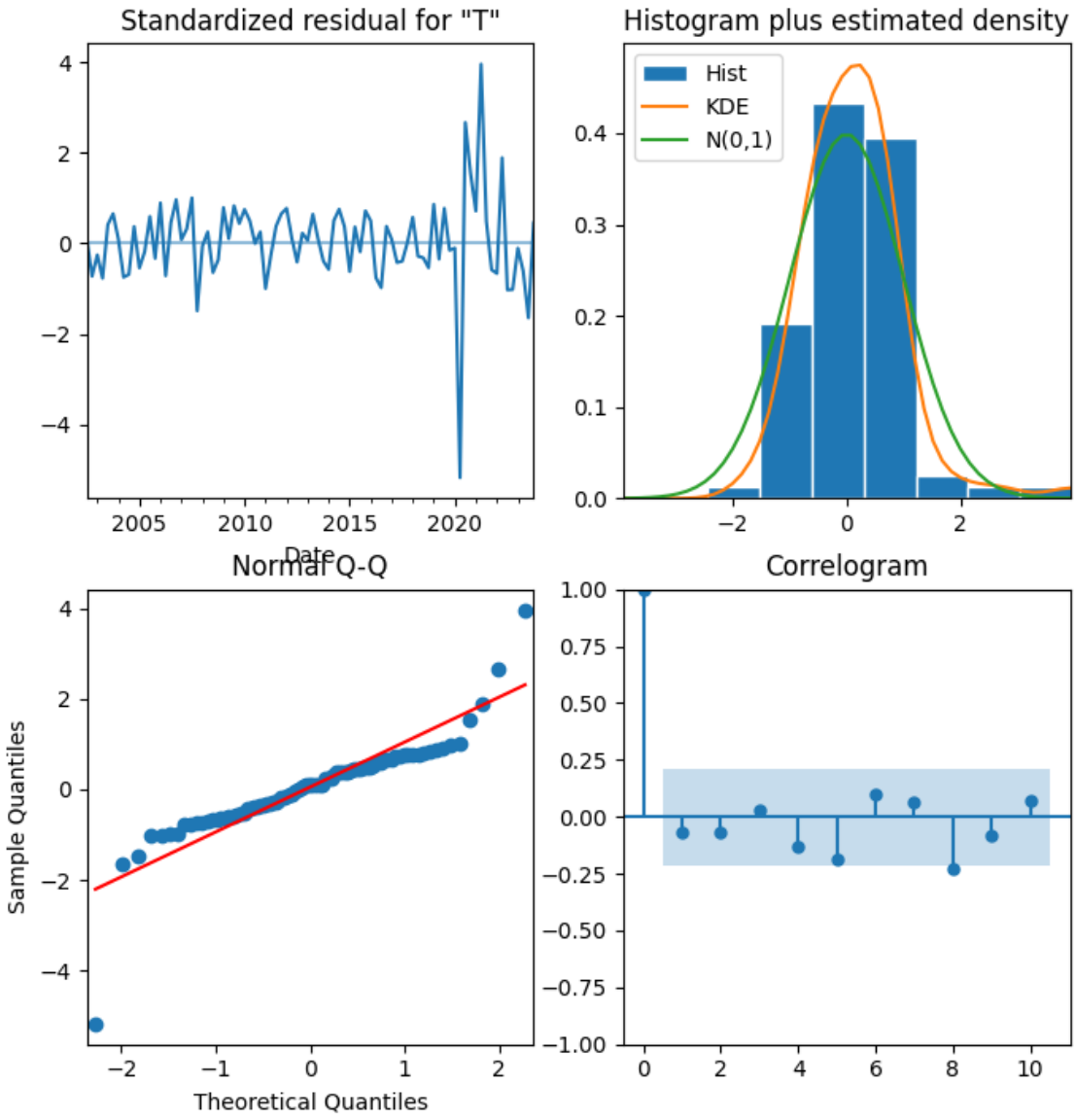
[19]:
```
results.aic
```

[19]: 1551.7113849904763

[20]:
```
results.plot_diagnostics(figsize=(8, 8.2))
```

[20]:


Standardized residual for "T"
Date
Histogram plus estimated density
Hist
KDE
N(0,1)
Normal Q-Q
Sample Quantiles
Theoretical Quantiles
Correlogram



Appendix 4.1.2: Evaluation

[69]:
```
# Train/validation split
train = diff_quarterly1[dt.datetime(2001,6,30,0,0):dt.datetime(2020,12,31,0,0)]
val = quarterly_meals.iloc[-12:]
```

[70]:
```
from sklearn.metrics import mean_absolute_error, mean_squared_error,␣
↪mean_absolute_percentage_error

# SARIMAX model
model = SARIMAX(train, order=(0,1,2), seasonal_order=(1,1,1,4))
results = model.fit(maxiter=1000)
```

```
forecast = results.get_forecast(steps=12)
forecast_diff = forecast.predicted_mean

# Reverse the differencing to get the forecast on the original scale
last_value = quarterly_meals.iloc[-1]  # The last value of the original␣
 ↪non-differenced data
val_arima = last_value + forecast_diff.cumsum()

# Evaluation (with train/val dataset)
mae = mean_absolute_error(val, val_arima)
print('MAE:', mae)
rmse = np.sqrt(mean_squared_error(val, val_arima))
print('RMSE:', rmse)
mape = mean_absolute_percentage_error(val, val_arima)
print('MAPE:', mape)
```

```
UserWarning: Non-invertible starting MA parameters found. Using zeros as
starting parameters.
 This problem is unconstrained.

RUNNING THE L-BFGS-B CODE

           * * *

Machine precision = 2.220D-16
 N =            5     M =           10

At X0         0 variables are exactly at the bounds

At iterate    0    f=  8.86329D+00    |proj g|=  2.81365D-01

At iterate    5    f=  8.71901D+00    |proj g|=  7.85570D-03

At iterate   10    f=  8.71735D+00    |proj g|=  1.97061D-04

At iterate   15    f=  8.71664D+00    |proj g|=  2.63318D-03

At iterate   20    f=  8.47744D+00    |proj g|=  5.15867D-03

At iterate   25    f=  8.45212D+00    |proj g|=  2.08229D-03
  ys=-4.573E-05  -gs= 7.869E-04 BFGS update SKIPPED

At iterate   30    f=  8.44277D+00    |proj g|=  2.55415D-03

At iterate   35    f=  8.18048D+00    |proj g|=  1.01947D-02

At iterate   40    f=  8.17810D+00    |proj g|=  1.17391D-05
```

```
           * * *

Tit   = total number of iterations
Tnf   = total number of function evaluations
Tnint = total number of segments explored during Cauchy searches
Skip  = number of BFGS updates skipped
Nact  = number of active bounds at final generalized Cauchy point
Projg = norm of the final projected gradient
F     = final function value

           * * *

   N    Tit     Tnf  Tnint  Skip  Nact     Projg        F
    5     40     70      1     1     0   1.174D-05   8.178D+00
  F =   8.1780967472160029

CONVERGENCE: REL_REDUCTION_OF_F_<=_FACTR*EPSMCH
MAE: 3461.9974959329143
RMSE: 4522.323508900008
MAPE: 0.21917908371018546
```

0.4.2 Appendix 4.2: Model with intervention

- ARMA (2,1,2) x SARMA (1,1,1)[4] –> 1583.86
- ARMA (2,1,1) x SARMA (1,1,1)[4] –> 1550.94
- ARMA (2,1,0) x SARMA (1,1,1)[4] –> 1596.20
- ARMA (1,1,2) x SARMA (1,1,1)[4] –> 1543.98
- **ARMA (0,1,2) x SARMA (1,1,1)[4] –> 1543.20**
- ARMA (2,1,2) x SARMA (0,1,1)[4] –> 1567.82
- ARMA (2,0,2) x SARMA (0,1,1)[4] –> 1577.51
- ARMA (0,0,5) x SARMA (0,0,0)[4] –> 1594.33

[35]:
```python
# Intervention (years 2020 & 2021)
intervention = [1 if '2020-01-01' <= str(date) <= '2022-12-31' else 0 for date␣
 ↪in diff_quarterly1.index]

# Params
p = 0
d = 1
q = 2

# Seasonal params
P = 1
D = 1 # stable seasonal pattern
Q = 1
s = 4 # quarterly data

sarimax_intervention = SARIMAX(diff_quarterly1,
```

```
                  order=(p,d,q),
                  seasonal_order=(P,D,Q,s),
                  exog=intervention)

results_intervention = sarimax_intervention.fit(maxiter=1000)
```

```
 This problem is unconstrained.
RUNNING THE L-BFGS-B CODE

           * * *

Machine precision = 2.220D-16
 N =            6     M =           10

At X0         0 variables are exactly at the bounds

At iterate    0    f=  9.20801D+00    |proj g|=  4.65458D-02

At iterate    5    f=  9.20012D+00    |proj g|=  1.29028D-04

At iterate   10    f=  9.19997D+00    |proj g|=  2.74862D-03

At iterate   15    f=  9.17988D+00    |proj g|=  2.56473D-02

At iterate   20    f=  8.85926D+00    |proj g|=  1.57082D-01

At iterate   25    f=  8.85762D+00    |proj g|=  6.84096D-02

At iterate   30    f=  8.72112D+00    |proj g|=  4.18608D-02

At iterate   35    f=  8.71834D+00    |proj g|=  3.35179D-04

At iterate   40    f=  8.68076D+00    |proj g|=  1.00205D-01

At iterate   45    f=  8.64929D+00    |proj g|=  3.46637D-03

At iterate   50    f=  8.64918D+00    |proj g|=  3.90042D-04

At iterate   55    f=  8.64904D+00    |proj g|=  7.82410D-03

At iterate   60    f=  8.64684D+00    |proj g|=  2.92970D-02

At iterate   65    f=  8.64530D+00    |proj g|=  7.20070D-04

At iterate   70    f=  8.64528D+00    |proj g|=  6.52141D-04

At iterate   75    f=  8.64290D+00    |proj g|=  1.71138D-02
```

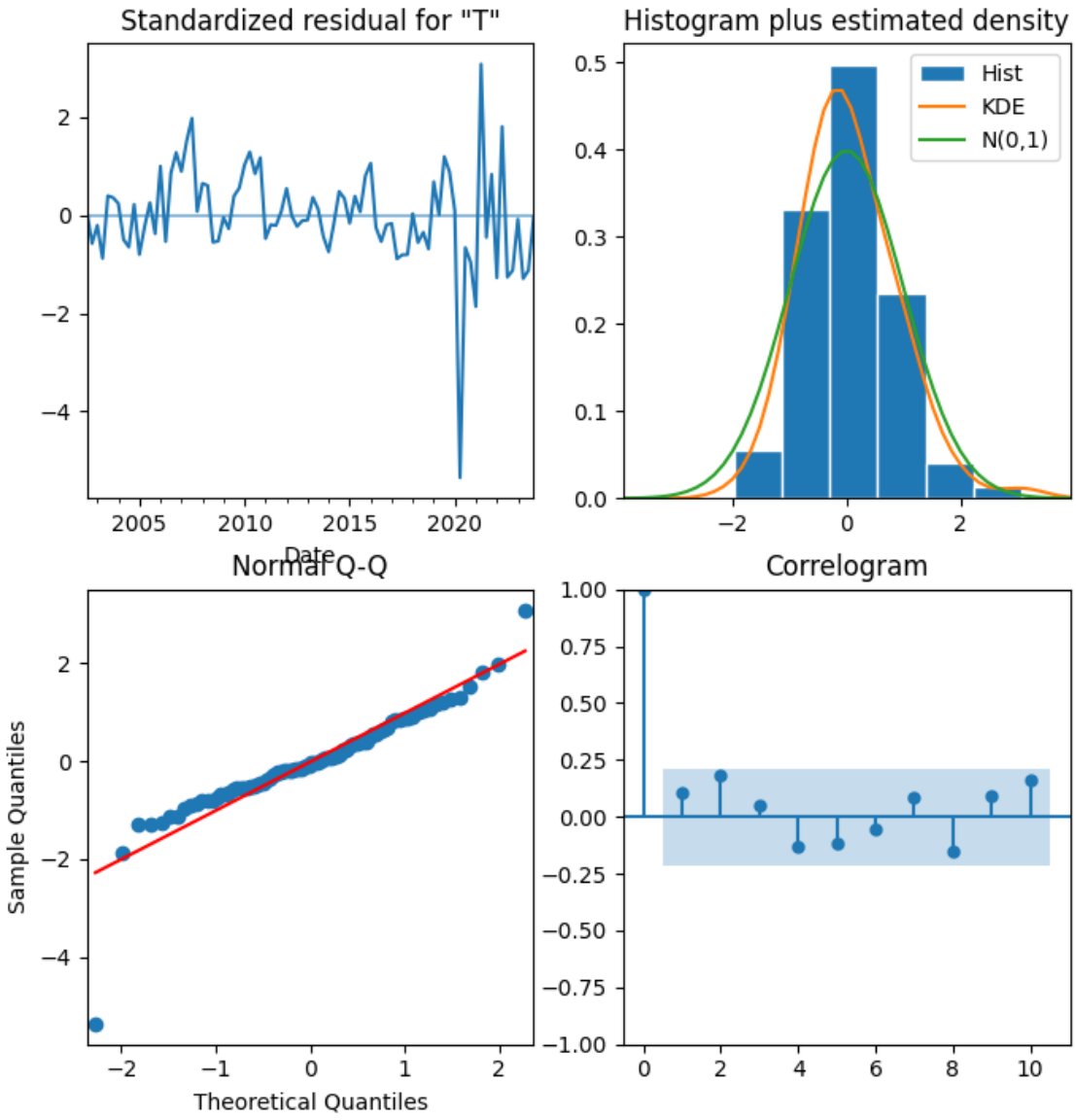
```
At iterate   80    f=  8.51503D+00    |proj g|=  1.10474D-01

At iterate   85    f=  8.44027D+00    |proj g|=  2.51435D-02

At iterate   90    f=  8.42347D+00    |proj g|=  2.96584D-02

At iterate   95    f=  8.41419D+00    |proj g|=  6.65112D-03

At iterate  100    f=  8.41331D+00    |proj g|=  1.43614D-03

At iterate  105    f=  8.41322D+00    |proj g|=  9.93097D-05

At iterate  110    f=  8.41320D+00    |proj g|=  9.93394D-05

At iterate  115    f=  8.41319D+00    |proj g|=  1.89082D-04

           * * *

Tit   = total number of iterations
Tnf   = total number of function evaluations
Tnint = total number of segments explored during Cauchy searches
Skip  = number of BFGS updates skipped
Nact  = number of active bounds at final generalized Cauchy point
Projg = norm of the final projected gradient
F     = final function value

           * * *

   N    Tit     Tnf  Tnint  Skip  Nact     Projg        F
    6    118    179      1     0     0   5.386D-06   8.413D+00
  F =   8.4131926297599744

CONVERGENCE: NORM_OF_PROJECTED_GRADIENT_<=_PGTOL
```

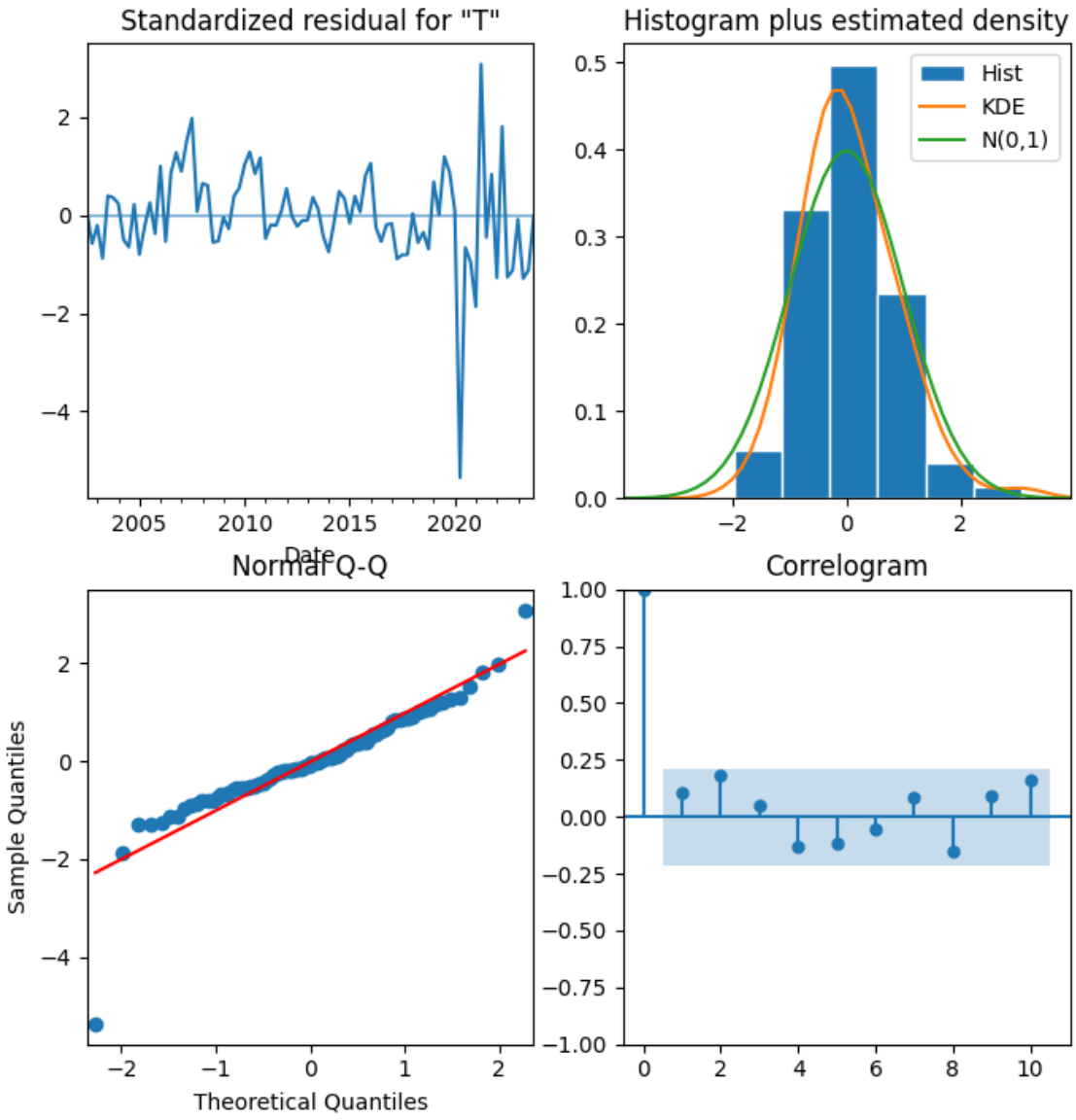
[24]:
```
results_intervention.aic
```

[24]: 1543.2010586163153

[25]:
```
results_intervention.plot_diagnostics(figsize=(8, 8.2))
```

[25]:


Standardized residual for "T"
Date
Histogram plus estimated density
Hist
KDE
N(0,1)
Normal Q-Q
Sample Quantiles
Theoretical Quantiles
Correlogram



- **Standardized Residuals Plot**: The top-left plot shows the standardized residuals over time. Ideally, you want to see no pattern, suggesting the residuals are white noise. If there's a clear pattern, especially systematic or periodic variations, it suggests the model is not capturing some aspect of the data. Here, there doesn't seem to be any obvious pattern, which is good.
- **Histogram plus KDE (Kernel Density Estimate) Plot**: The top-right plot compares the distribution of the standardized residuals to a normal distribution. The blue histogram represents the frequency of standardized residuals, the orange line is the KDE that estimates the density of the residuals, and the green line represents a normal distribution. Ideally, the KDE should follow closely with the normal distribution line. There's a slight deviation on the tails, suggesting the presence of outliers or heavy tails in the distribution of residuals.
- **Normal Q-Q Plot**: The bottom-left plot is a Quantile-Quantile plot. It compares the

quantiles of the standardized residuals with the quantiles of a normal distribution. If the points lie on the red line, it indicates that the residuals are normally distributed. In this plot, most of the data points fall along the line, but there are deviations on both ends, indicating that the residuals may not be perfectly normally distributed.

- **Correlogram (ACF Plot)**: The bottom-right plot shows the autocorrelation function (ACF) of the residuals. For a well-fitting model, you would expect that there would be no significant autocorrelation in the residuals. This plot typically has blue lines that represent confidence intervals – if the bars are within this area, it is assumed that there are no significant autocorrelations. The correlogram here looks good, as all the autocorrelations are within the confidence band, suggesting that there is no significant autocorrelation.

Appendix 4.2.1: Forecast (next 10 periods)

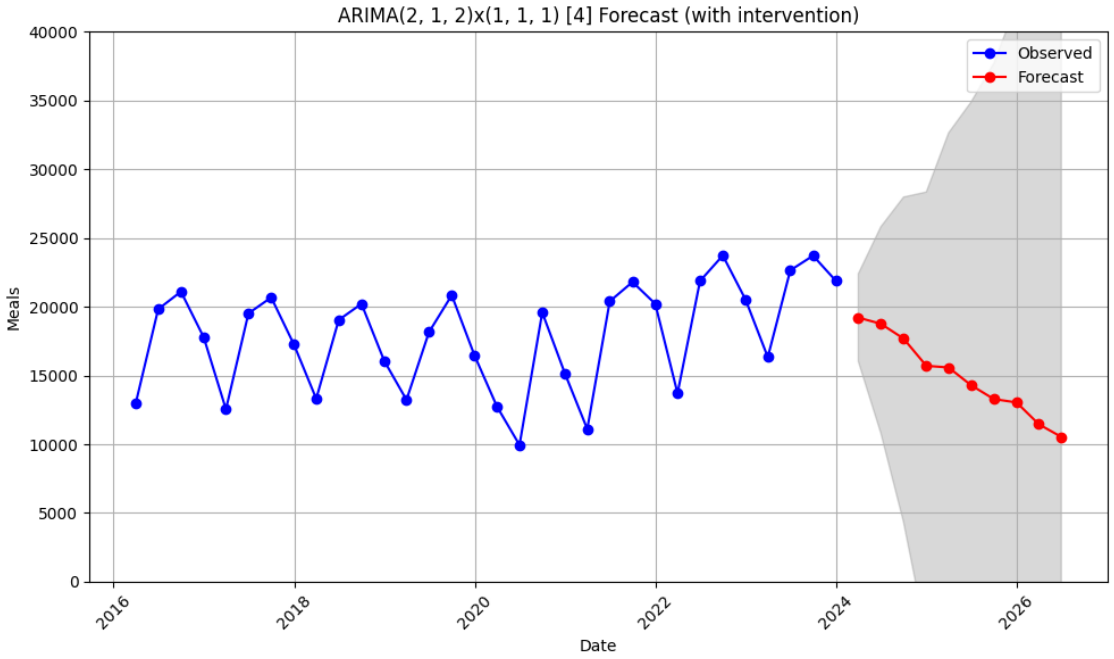
[26]:
```
# SARIMAX model for predicting next 10 periods
model = SARIMAX(diff_quarterly1, order=(0,1,2), seasonal_order=(1,1,1,4),␣
 ↪exog=intervention)
results = model.fit(maxiter=1000)
forecast = results.get_forecast(steps=10, exog=[0]*10)
forecast_diff = forecast.predicted_mean
forecast_conf_int = forecast.conf_int()

# Reverse the differencing to get the forecast on the original scale
last_value = quarterly_meals.iloc[-1]  # The last value of the original␣
 ↪non-differenced data
forecast_integrated = last_value + forecast_diff.cumsum()

# Calculate the cumulative sum of the forecasted lower and upper bounds.
integrated_lower_bound = forecast_integrated - (forecast_diff.iloc[0] -␣
 ↪forecast_conf_int.iloc[:, 0].cumsum())
integrated_upper_bound = forecast_integrated + (forecast_conf_int.iloc[:, 1].
 ↪cumsum() - forecast_diff.iloc[0])
```

```
RUNNING THE L-BFGS-B CODE

           * * *

Machine precision = 2.220D-16
 N =            6     M =           10

At X0         0 variables are exactly at the bounds

At iterate    0    f=  9.20801D+00    |proj g|=  4.65458D-02

At iterate    5    f=  9.20012D+00    |proj g|=  1.29028D-04

At iterate   10    f=  9.19997D+00    |proj g|=  2.74862D-03

At iterate   15    f=  9.17988D+00    |proj g|=  2.56473D-02
```

```
At iterate   20    f=  8.85926D+00    |proj g|=  1.57082D-01

At iterate   25    f=  8.85762D+00    |proj g|=  6.84096D-02

At iterate   30    f=  8.72112D+00    |proj g|=  4.18608D-02

At iterate   35    f=  8.71834D+00    |proj g|=  3.35179D-04

At iterate   40    f=  8.68076D+00    |proj g|=  1.00205D-01

At iterate   45    f=  8.64929D+00    |proj g|=  3.46637D-03
 This problem is unconstrained.

At iterate   50    f=  8.64918D+00    |proj g|=  3.90042D-04

At iterate   55    f=  8.64904D+00    |proj g|=  7.82410D-03

At iterate   60    f=  8.64684D+00    |proj g|=  2.92970D-02

At iterate   65    f=  8.64530D+00    |proj g|=  7.20070D-04

At iterate   70    f=  8.64528D+00    |proj g|=  6.52141D-04

At iterate   75    f=  8.64290D+00    |proj g|=  1.71138D-02

At iterate   80    f=  8.51503D+00    |proj g|=  1.10474D-01

At iterate   85    f=  8.44027D+00    |proj g|=  2.51435D-02

At iterate   90    f=  8.42347D+00    |proj g|=  2.96584D-02

At iterate   95    f=  8.41419D+00    |proj g|=  6.65112D-03

At iterate  100    f=  8.41331D+00    |proj g|=  1.43614D-03

At iterate  105    f=  8.41322D+00    |proj g|=  9.93097D-05

At iterate  110    f=  8.41320D+00    |proj g|=  9.93394D-05

At iterate  115    f=  8.41319D+00    |proj g|=  1.89082D-04

           * * *

Tit   = total number of iterations
Tnf   = total number of function evaluations
Tnint = total number of segments explored during Cauchy searches
```

```
Skip  = number of BFGS updates skipped
Nact  = number of active bounds at final generalized Cauchy point
Projg = norm of the final projected gradient
F     = final function value

           * * *

   N    Tit     Tnf  Tnint  Skip  Nact     Projg        F
    6    118    179      1     0     0   5.386D-06   8.413D+00
  F =   8.4131926297599744

CONVERGENCE: NORM_OF_PROJECTED_GRADIENT_<=_PGTOL
```

[27]:
```
# Plot the original data
plt.figure(figsize=(10, 6))
plt.plot(quarterly_meals[-32:].index, quarterly_meals[-32:], label='Observed',␣
 ↪marker='o', linestyle='-', color='b')

# Plot the integrated forecast
plt.plot(pd.date_range(start=quarterly_meals.index[-1], periods=11,␣
 ↪freq='QE')[1:], forecast_integrated, label='Forecast', marker='o',␣
 ↪linestyle='-', color='r')

# Plot the integrated confidence intervals
plt.fill_between(pd.date_range(start=quarterly_meals.index[-1],␣
 ↪periods=len(forecast_diff)+1, freq='QE')[1:], integrated_lower_bound,␣
 ↪integrated_upper_bound, color='grey', alpha=0.3)

plt.title('ARIMA(2, 1, 2)x(1, 1, 1) [4] Forecast (with intervention)')
plt.xlabel('Date')
plt.ylabel('Meals')
plt.legend()
plt.ylim(0, 40000)
plt.xticks(rotation=45)
plt.tight_layout()
plt.grid(True)
plt.show()
```



Appendix 4.2.2: Evaluation

[72]:
```
# SARIMAX model
model = SARIMAX(train, order=(0,1,2), seasonal_order=(1,1,1,4),␣
 ↪exog=intervention[:-12])
results = model.fit(maxiter=1000)
forecast = results.get_forecast(steps=12, exog=[0]*12)
forecast_diff = forecast.predicted_mean

# Reverse the differencing to get the forecast on the original scale
last_value = quarterly_meals.iloc[-1]  # The last value of the original␣
 ↪non-differenced data
val_arima = last_value + forecast_diff.cumsum()

# Evaluation (with train/val dataset)
mae = mean_absolute_error(val, val_arima)
print('MAE:', mae)
rmse = np.sqrt(mean_squared_error(val, val_arima))
print('RMSE:', rmse)
mape = mean_absolute_percentage_error(val, val_arima)
print('MAPE:', mape)
```

```
 This problem is unconstrained.
RUNNING THE L-BFGS-B CODE

           * * *
```

```
Machine precision = 2.220D-16
 N =            6     M =           10

At X0         0 variables are exactly at the bounds

At iterate    0    f=  8.73988D+00    |proj g|=  3.43805D-02

At iterate    5    f=  8.72862D+00    |proj g|=  4.02194D-03

At iterate   10    f=  8.72684D+00    |proj g|=  1.89869D-04

At iterate   15    f=  8.72672D+00    |proj g|=  2.06698D-03

At iterate   20    f=  8.70491D+00    |proj g|=  4.99898D-02

At iterate   25    f=  8.29582D+00    |proj g|=  3.04244D-02

At iterate   30    f=  8.29056D+00    |proj g|=  2.75226D-03

At iterate   35    f=  8.29052D+00    |proj g|=  8.76886D-05

At iterate   40    f=  8.29050D+00    |proj g|=  2.23880D-03

At iterate   45    f=  8.28789D+00    |proj g|=  2.50472D-02

At iterate   50    f=  8.20651D+00    |proj g|=  3.72701D-02

At iterate   55    f=  8.16863D+00    |proj g|=  3.11416D-03

At iterate   60    f=  8.16675D+00    |proj g|=  6.39821D-03

At iterate   65    f=  8.16129D+00    |proj g|=  2.22554D-02

At iterate   70    f=  8.15972D+00    |proj g|=  1.18877D-03

           * * *

Tit   = total number of iterations
Tnf   = total number of function evaluations
Tnint = total number of segments explored during Cauchy searches
Skip  = number of BFGS updates skipped
Nact  = number of active bounds at final generalized Cauchy point
Projg = norm of the final projected gradient
F     = final function value

           * * *
```

```
   N    Tit     Tnf  Tnint  Skip  Nact     Projg        F
    6     73      93      1     0     0   5.731D-06   8.160D+00
  F =   8.1597062598967014

CONVERGENCE: NORM_OF_PROJECTED_GRADIENT_<=_PGTOL
MAE: 3866.17459818228
RMSE: 4573.49047255581
MAPE: 0.22450930292360358
```

0.5 Appendix.5: Holt-Winters

0.5.1 Appendix 5.1: Powell optimization

Evaluation

[64]:
```
from statsmodels.tsa.holtwinters import ExponentialSmoothing

# Exponential Smoothing
model = ExponentialSmoothing(train, trend='additive', seasonal='additive',␣
 ↪seasonal_periods=4, initialization_method='heuristic')
results = model.fit(optimized=True,
                    remove_bias=False,
                    method='Powell'  # works with 'basinhopping' and 'Powell'
                    )

# Forecast the next 10 periods of the differenced data
forecast_diff = results.forecast(6)

# Reverse the differencing to get the forecast on the original scale
last_value = quarterly_meals.iloc[-1]  # The last value of the original␣
 ↪non-differenced data
val_powell = last_value + forecast_diff.cumsum()

# Evaluation
mae = mean_absolute_error(val, val_powell)
print('MAE Powell:', mae)
rmse = np.sqrt(mean_squared_error(val, val_powell))
print('RMSE Powell:', rmse)
mape = mean_absolute_percentage_error(val, val_powell)
print('MAPE Powell:', mape)
```

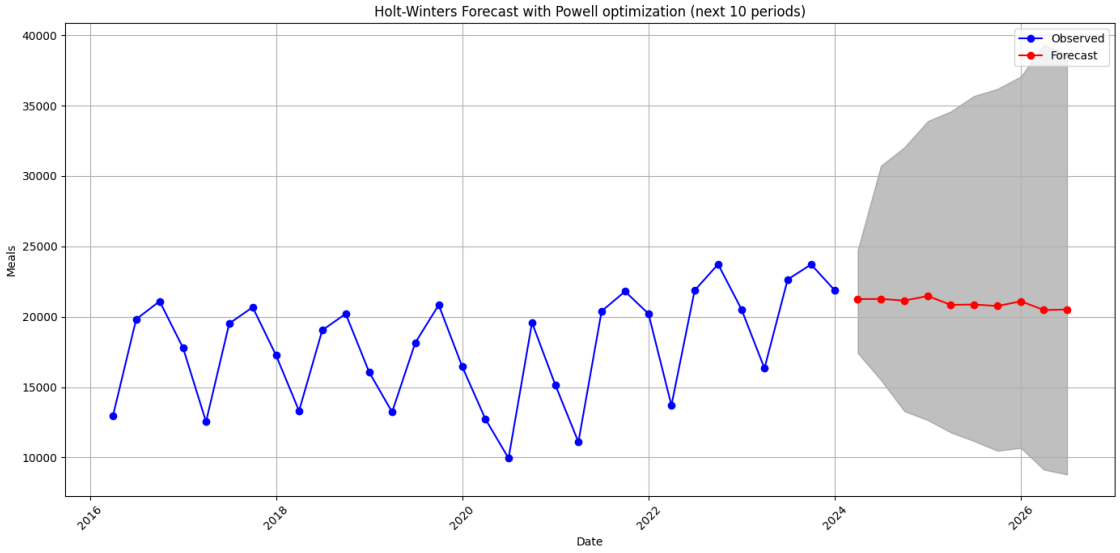
```
MAE Powell: 2088.6310724655036
RMSE Powell: 2566.6806783823945
MAPE Powell: 0.10644484178708101
```

Forecast

[40]:
```
# Fit the Exponential Smoothing model on the differenced data
model = ExponentialSmoothing(diff_quarterly1,
```

```
                              trend='additive',
                              seasonal='additive',
                              seasonal_periods=4,
                              initialization_method='heuristic')
results = model.fit(optimized=True,
                    remove_bias=False,
                    method='Powell')

# Forecast the next 10 periods of the differenced data
forecast_diff = results.forecast(10)

# Reverse the differencing to get the forecast on the original scale
last_value = quarterly_meals.iloc[-1]  # The last value of the original␣
 ↪non-differenced data
forecast_integrated = last_value + forecast_diff.cumsum()

# Simulate future residuals by bootstrapping the historical residuals
simulated_residuals = np.random.choice(results.resid, size=(10, 1000))
simulated_paths = np.cumsum(simulated_residuals, axis=0) + last_value

# Calculate the percentiles for each period across all simulated paths
lower_percentiles = np.percentile(simulated_paths, 2.5, axis=1)
upper_percentiles = np.percentile(simulated_paths, 97.5, axis=1)

# Calculate the lower and upper bounds for the forecasted values
lower_bounds = forecast_integrated - (last_value - lower_percentiles)
upper_bounds = forecast_integrated + (upper_percentiles - last_value)

# Plot the original data
plt.figure(figsize=(14, 7))
plt.plot(quarterly_meals[-32:].index, quarterly_meals[-32:], label='Observed',␣
 ↪marker='o', linestyle='-', color='b')

# Plot the integrated forecast
plt.plot(pd.date_range(start=quarterly_meals.index[-1], periods=11,␣
 ↪freq='QE')[1:], forecast_integrated, label='Forecast', marker='o',␣
 ↪linestyle='-', color='r')

# Plot the integrated confidence intervals
plt.fill_between(pd.date_range(start=quarterly_meals.index[-1], periods=11,␣
 ↪freq='QE')[1:], lower_bounds, upper_bounds, color='grey', alpha=0.5)

plt.title('Holt-Winters Forecast with Powell optimization (next 10 periods)')
plt.xlabel('Date')
plt.ylabel('Meals')
plt.legend()
plt.grid(True)
```

```
plt.xticks(rotation=45)
plt.tight_layout()
plt.show()
```



0.5.2 Appendix 5.2: Basin-Hoppin optimization

Evaluation

[65]:
```
# Exponential Smoothing
model = ExponentialSmoothing(train, trend='additive', seasonal='additive',␣
 ↪seasonal_periods=4, initialization_method='heuristic')
results = model.fit(optimized=True,
                    remove_bias=False,
                    method='basinhopping'  # works with 'basinhopping' and␣
 ↪'Powell'
                    )

# Forecast the next 10 periods of the differenced data
forecast_diff = results.forecast(6)

# Reverse the differencing to get the forecast on the original scale
last_value = quarterly_meals.iloc[-1]  # The last value of the original␣
 ↪non-differenced data
val_basinhopping = last_value + forecast_diff.cumsum()

# Evaluation
mae = mean_absolute_error(val, val_basinhopping)
print('MAE Basin-Hoppin:', mae)
```

```
rmse = np.sqrt(mean_squared_error(val, val_basinhopping))
print('RMSE Basin-Hoppin:', rmse)
mape = mean_absolute_percentage_error(val, val_basinhopping)
print('MAPE Basin-Hoppin:', mape)
```

```
MAE Basin-Hoppin: 2087.960226306207
RMSE Basin-Hoppin: 2566.6068101709434
MAPE Basin-Hoppin: 0.10641867298032619
```

Forecast

[32]:
```
# Fit the Exponential Smoothing model on the differenced data
model = ExponentialSmoothing(diff_quarterly1,
                             trend='additive',
                             seasonal='additive',
                             seasonal_periods=4,
                             initialization_method='heuristic')
results = model.fit(optimized=True,
                    remove_bias=False,
                    method='basinhopping')

# Forecast the next 10 periods of the differenced data
forecast_diff = results.forecast(10)

# Reverse the differencing to get the forecast on the original scale
last_value = quarterly_meals.iloc[-1]  # The last value of the original␣
 ↪non-differenced data
forecast_integrated = last_value + forecast_diff.cumsum()

# Simulate future residuals by bootstrapping the historical residuals
simulated_residuals = np.random.choice(results.resid, size=(10, 1000))
simulated_paths = np.cumsum(simulated_residuals, axis=0) + last_value

# Calculate the percentiles for each period across all simulated paths
lower_percentiles = np.percentile(simulated_paths, 2.5, axis=1)
upper_percentiles = np.percentile(simulated_paths, 97.5, axis=1)

# Calculate the lower and upper bounds for the forecasted values
lower_bounds = forecast_integrated - (last_value - lower_percentiles)
upper_bounds = forecast_integrated + (upper_percentiles - last_value)

# Plot the original data
plt.figure(figsize=(14, 7))
plt.plot(quarterly_meals[-32:].index, quarterly_meals[-32:], label='Observed',␣
 ↪marker='o', linestyle='-', color='b')

# Plot the integrated forecast
```

```
plt.plot(pd.date_range(start=quarterly_meals.index[-1], periods=11,␣
 ↪freq='QE')[1:], forecast_integrated, label='Forecast', marker='o',␣
 ↪linestyle='-', color='r')

# Plot the integrated confidence intervals
plt.fill_between(pd.date_range(start=quarterly_meals.index[-1], periods=11,␣
 ↪freq='QE')[1:], lower_bounds, upper_bounds, color='grey', alpha=0.5)

plt.title('Basin-Hopping Forecast with Basin-Hoppin optimization (next 10␣
 ↪periods)')
plt.xlabel('Date')
plt.ylabel('Value')
plt.legend()
plt.grid(True)
plt.xticks(rotation=45)
plt.tight_layout()
plt.show()
```