Time SeriesForecasting - Semester End Exam - 80 marks

In [118]:
```
import pandas                           as      pd
import numpy                            as      np
import matplotlib.pyplot                as      plt
from   IPython.display                  import  display
from   pylab                            import  rcParams
from   datetime                         import  datetime, timedelta
from   pandas.tseries.offsets           import BDay
from statsmodels.tsa.stattools          import  adfuller
from statsmodels.tsa.stattools          import  pacf
from statsmodels.tsa.stattools          import  acf
from statsmodels.graphics.tsaplots      import  plot_pacf
from statsmodels.graphics.tsaplots      import  plot_acf
from statsmodels.graphics.gofplots      import  qqplot
from statsmodels.tsa.seasonal           import  seasonal_decompose
from   statsmodels.tsa.api              import  ExponentialSmoothing,Holt
from statsmodels.tsa.arima.model        import  ARIMA
from sklearn.metrics                    import  mean_squared_error
import warnings
warnings.filterwarnings('ignore')
%matplotlib inline
import itertools
```

Section A Total 20 Marks

1.A Explain regular components of a time series. (6 Marks)

In time series analysis, there are several regular components that are commonly observed in a time series data. These components help us understand and model the underlying patterns and variations in the data. The four main components of a time series are:

1.Trend:

> The trend component represents the long-term systematic movement in the time series. It captures the overall direction of the data over time, indicating whether the series is increasing, decreasing, or remaining relatively stable. Trends can be linear or nonlinear and can have different degrees of persistence. Trend analysis helps identify the underlying growth or decay patterns in the data.

2.Seasonality:

> Seasonality refers to the regular and predictable patterns that repeat at fixed intervals within a time series. These patterns can be daily, weekly, monthly, quarterly, or annual, depending on the nature of the data. Seasonality can be influenced by factors such as calendar effects, weather, holidays, or business cycles. Identifying and modeling seasonality is important for understanding and forecasting short-term fluctuations in the data.

3.Cyclical:

> The cyclical component represents the medium-term fluctuations in a time series that are not related to seasonal or trend patterns. Unlike seasonality, cycles do not have fixed periods and can vary in duration. Cyclical patterns are often influenced by economic factors, business cycles, and other external factors that impact the data over longer time horizons.

4.Irregular/Residual:

The irregular component, also known as the residual or noise, represents the random and unpredictable fluctuations in the time series that cannot be explained by the trend, seasonality, or cyclical components. It includes random variations, measurement errors, and other unexplained factors. The irregular component is typically assumed to be a white noise process with no discernible patterns or correlations.

By decomposing a time series into these four components, analysts can better understand the different sources of variation and structure within the data. This decomposition facilitates forecasting, anomaly detection, and the identification of underlying relationships or patterns that may be present in the series. Various statistical techniques, such as moving averages, exponential smoothing, and Fourier analysis, are commonly used to extract and analyze these components in time series data.

1.B What is stationarity? How will you convert non-stationary series into stationary series? (6 Marks)

Stationarity is an important concept in time series analysis. A stationary time series is one where the statistical properties of the data do not change over time. This means that the mean, variance, and autocovariance structure remain constant across different time periods. In simpler terms, the distribution of data points and the relationship between them do not depend on the specific time at which they were observed.

Conversely, a non-stationary time series exhibits some form of trend, seasonality, or other time-dependent patterns. Non-stationary series can make it difficult to analyze and model the data accurately because the statistical properties are changing over time, violating the assumptions of many time series models.

To convert a non-stationary series into a stationary series, you can apply various techniques. Here are a few commonly used methods:

1.Differencing:

One of the simplest and most effective methods is differencing. By taking the difference between consecutive observations, you can remove the trend component and make the series stationary. This is called first-order differencing. If the series still exhibits a trend, further differencing can be applied until the series becomes stationary.

2.Logarithmic Transformation:

If the series exhibits exponential growth or decay, taking the logarithm of the values can help stabilize the variance and make the series stationary. This is particularly useful when the trend is multiplicative rather than additive.

3.Seasonal Adjustment:

If the non-stationarity is primarily due to seasonal patterns, you can remove the seasonal component from the series. This can be done using techniques such as seasonal differencing, seasonal decomposition of time series (e.g., using seasonal decomposition of time series - X11 or seasonal-trend decomposition using LOESS - STL), or seasonal adjustment methods like the seasonal adjustment using regression technique (SEATS).

4.Detrending:

If the non-stationarity is mainly driven by a deterministic trend component, you can remove the trend by fitting a regression model to the data and subtracting the trend component from the series. This can be done using techniques like linear regression, polynomial regression, or more advanced methods like locally weighted scatterplot smoothing (LOWESS) or moving averages.

It's important to note that the specific method used to convert a non-stationary series into a stationary series depends on the characteristics of the data and the underlying patterns present. Additionally, after applying these transformations, it's necessary to check for stationarity using statistical tests (e.g., Augmented Dickey-Fuller test) to ensure that the resulting series is indeed stationary.

1.C How will you determine the order of a moving average process? Explain. (8 Marks)

Determining the order of a moving average (MA) process in a time series analysis involves identifying the number of lagged error terms (or innovations) that significantly contribute to the current value of the series. The order of the MA process, denoted by "q," represents the number of lagged error terms that are included in the model.

To determine the order of the MA process, you can use the following methods:

1.Autocorrelation Function (ACF):

> The ACF measures the correlation between a time series and its lagged values. For an MA process, the ACF will exhibit significant values only at the lag corresponding to the order of the process (q). The ACF will decay rapidly for lags greater than q. Therefore, you can plot the ACF for different lags and look for a significant cutoff point where the values drop off. The lag corresponding to the last significant value before the drop-off is the estimated order of the MA process.

2.Partial Autocorrelation Function (PACF):

> The PACF represents the correlation between a time series and its lagged values while removing the effect of intervening lags. For an MA process, the PACF will have significant values only at the lag corresponding to the order of the process (q) and will be close to zero for other lags. By examining the PACF plot, you can identify the lag at which the PACF sharply drops off to zero, indicating the estimated order of the MA process.

3.Information Criteria:

> Information criteria, such as the Akaike Information Criterion (AIC) or Bayesian Information Criterion (BIC), can also be used to determine the order of the MA process. These criteria provide a trade-off between model fit and model complexity. By fitting different MA models with varying orders and comparing their AIC or BIC values, you can select the order that minimizes the information criterion, indicating the most suitable order for the MA process.

It's important to note that these methods are not always definitive, and it may be necessary to consider multiple criteria and expert judgment to determine the appropriate order of the MA process. Additionally, other factors, such as the behavior of the residual autocorrelations and the theoretical understanding of the data generating process, should also be taken into account for a comprehensive analysis.

Section B Total 30 Marks

DATA_SET:

Analytics firm wants to forecast the avg spending of customers for the month of Oct 2020. For this, firm has gathered a closing stock price data for the period of Feb 2019 to Sept 2020.

- date == date field
- close == Avg Spending (numeric)

2.A Data preparation (5 marks)

a. Read the dataset (tab, csv, xls, txt, inbuilt dataset). What are the number of rows and no. of cols & types of variables? (1 MARK)

In [64]:
```
# Read the stock price data from a CSV file
data = pd.read_csv('dataset.csv')
```

In [65]:
```
data.shape
```

Out[65]:
```
(605, 2)
```

In [66]:
```
# Get the number of rows and columns
num_rows = data.shape[0]
num_cols = data.shape[1]
```

In [67]:
```
num_cols
```

Out[67]:
```
2
```

In [68]:
```
num_rows
```

Out[68]:
```
605
```

In [69]:
```
data.dtypes
```

Out[69]:
```
Date            object
Avg_spending    float64
dtype: object
```

In [70]:
```
data.columns
```

Out[70]:
```
Index(['Date', 'Avg_spending'], dtype='object')
```

b. convert the data into time series (2 MARK)

To convert the dataset into a time series, you need to ensure that the date column is recognized as a datetime data type and set as the index of the DataFrame. Here's how you can achieve that using Python and pandas:

In [71]:
```
# Convert the date column to datetime data type
data['Date'] = pd.to_datetime(data['Date'])
```

In [72]:
```
# check the data type
data.dtypes
```

Out[72]:
```
Date            datetime64[ns]
Avg_spending           float64
dtype: object
```

In [73]:
```
# Set the date column as the index
data.set_index('Date', inplace=True)
```

In [74]:
```
# Sort the data based on the date
data.sort_index(inplace=True)
```

In [79]:
```
1 data.head()
```

Out[79]:

Date	Avg_spending
2019-01-03	56.67
2019-01-04	49.53
2019-01-05	40.90
2019-01-06	52.03
2019-01-07	44.14

c. Check for defects in the data such as missing values, null, etc. (1 MARK)

In [75]:
```
1 # Check for missing values
2 missing_values = data.isnull().sum()
```

In [76]:
```
1 print(missing_values)
```

```
Avg_spending    0
dtype: int64
```

In [77]:
```
1 # Check for null values
2 null_values = data.isna().sum()
```

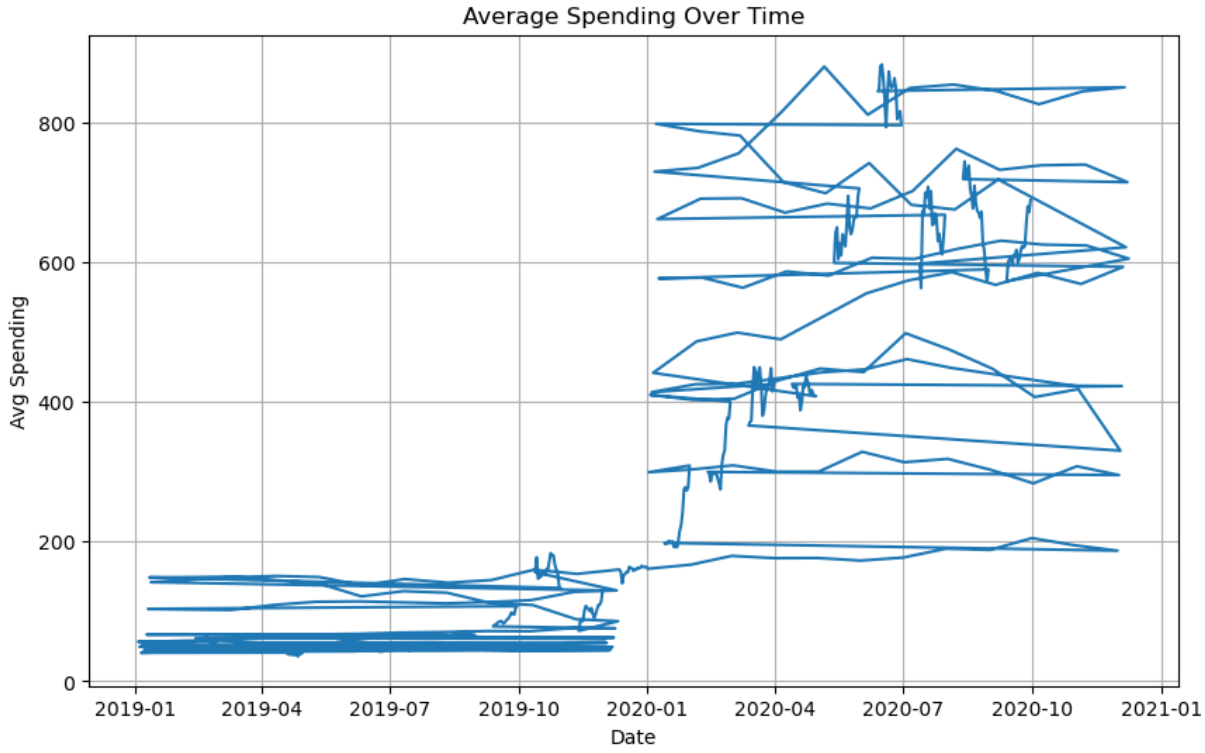
In [78]:
```
1 print(null_values)
```

```
Avg_spending    0
dtype: int64
```

d. Visualize the time series using relevant plots. (1 MARK)

Line plot

In [45]:
```
# Create a line plot
plt.figure(figsize=(10, 6))
plt.plot(data.index, data['Avg_spending'])
plt.xlabel('Date')
plt.ylabel('Avg Spending')
plt.title('Average Spending Over Time')
plt.grid(True)
plt.show()
```



2.B Data Understanding (15 marks)

a. Decompose the time series and check for components of time series. (4 MARKS)

In [83]:
```
# Perform time series decomposition
from statsmodels.tsa.seasonal import STL
decomposition = STL(data['Avg_spending'], seasonal=13).fit()
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
Cell In [83], line 3
      1 # Perform time series decomposition
      2 from statsmodels.tsa.seasonal import STL
----> 3 decomposition = STL(data['Avg_spending'], seasonal=13).fit()

File statsmodels\tsa\_stl.pyx:219, in statsmodels.tsa._stl.STL.__init__()

ValueError: Unable to determine period from endog
```

In [84]:
```
# Perform time series decomposition
seasonal_period = 12  # Set the seasonal period (e.g., 12 for monthly data)
decomposition = STL(data['Avg_spending'], period=seasonal_period).fit()
```

In [85]:
```
# Extract the components
trend = decomposition.trend
seasonal = decomposition.seasonal
residuals = decomposition.resid
```

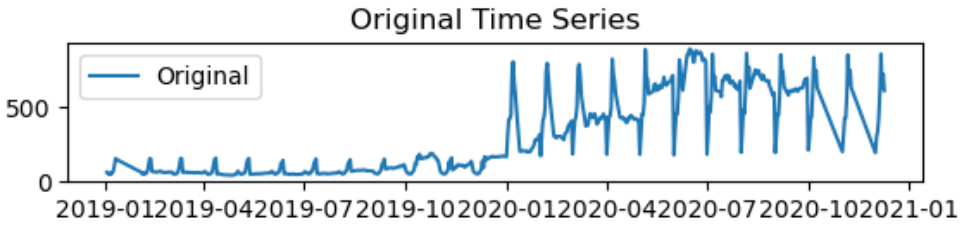
In [86]:
```
# Plot the components
plt.figure(figsize=(12, 8))
```

Out[86]:
```
<Figure size 1200x800 with 0 Axes>
```

<Figure size 1200x800 with 0 Axes>

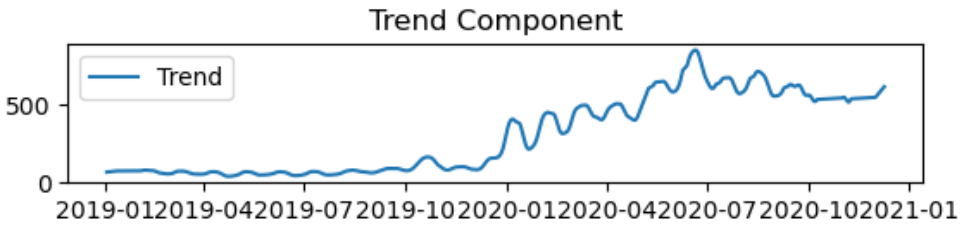
In [87]:
```
plt.subplot(411)
plt.plot(data['Avg_spending'], label='Original')
plt.legend(loc='upper left')
plt.title('Original Time Series')
```

Out[87]:
```
Text(0.5, 1.0, 'Original Time Series')
```



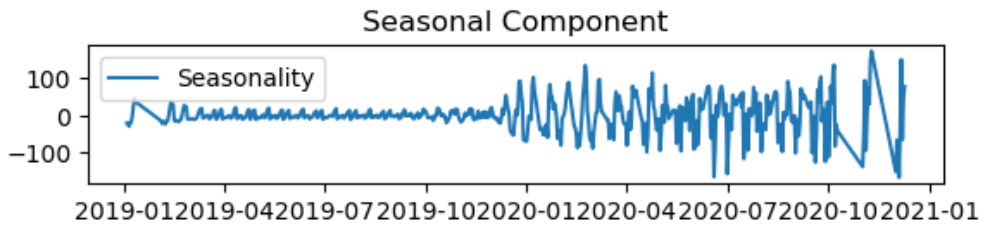
In [88]:
```
plt.subplot(412)
plt.plot(trend, label='Trend')
plt.legend(loc='upper left')
plt.title('Trend Component')
```

Out[88]:
```
Text(0.5, 1.0, 'Trend Component')
```



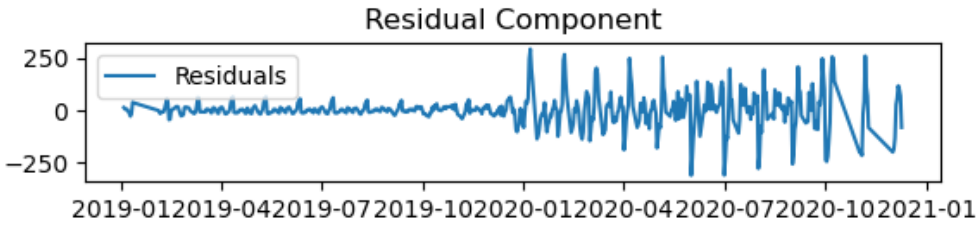
In [89]:
```
plt.subplot(413)
plt.plot(seasonal, label='Seasonality')
plt.legend(loc='upper left')
plt.title('Seasonal Component')
```

Out[89]:
```
Text(0.5, 1.0, 'Seasonal Component')
```



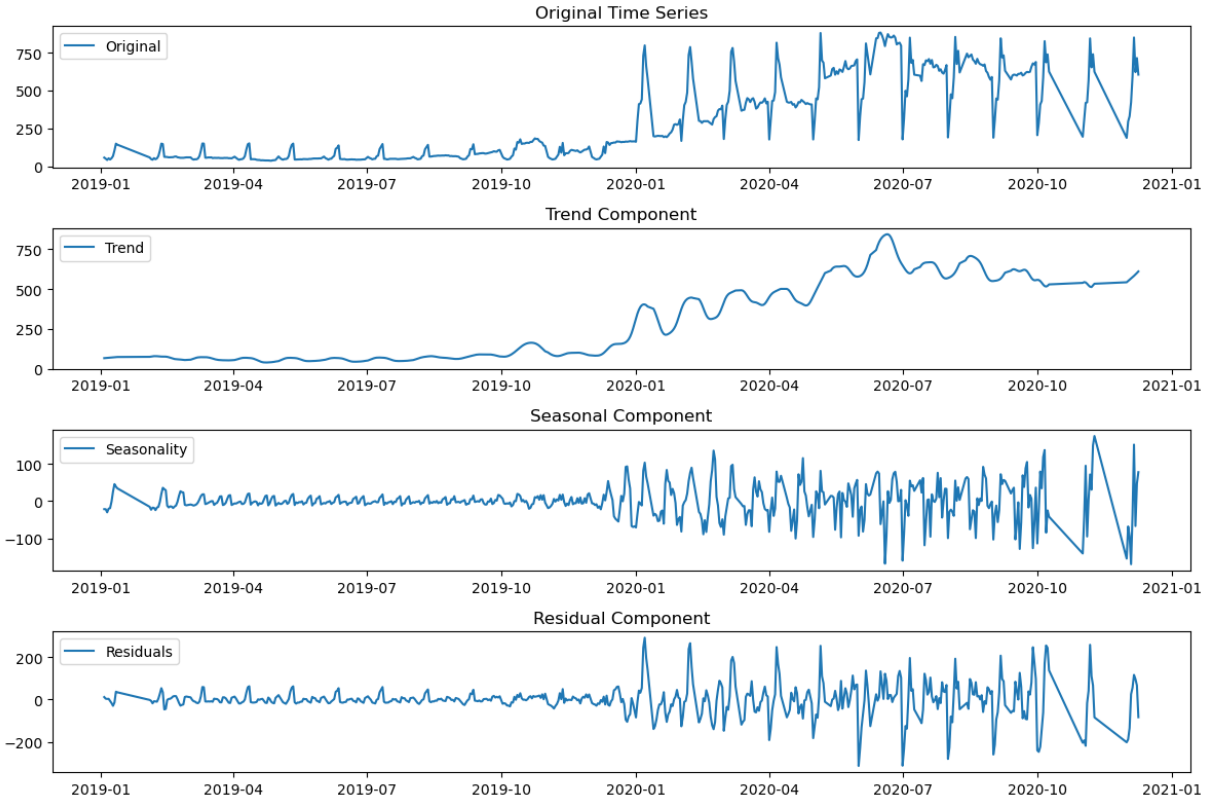
In [90]:
```
plt.subplot(414)
plt.plot(residuals, label='Residuals')
plt.legend(loc='upper left')
plt.title('Residual Component')
```

Out[90]:
```
Text(0.5, 1.0, 'Residual Component')
```



All plots in one cell

In [91]:
```
# Extract the components
trend = decomposition.trend
seasonal = decomposition.seasonal
residuals = decomposition.resid

# Plot the components
plt.figure(figsize=(12, 8))

plt.subplot(411)
plt.plot(data['Avg_spending'], label='Original')
plt.legend(loc='upper left')
plt.title('Original Time Series')

plt.subplot(412)
plt.plot(trend, label='Trend')
plt.legend(loc='upper left')
plt.title('Trend Component')

plt.subplot(413)
plt.plot(seasonal, label='Seasonality')
plt.legend(loc='upper left')
plt.title('Seasonal Component')

plt.subplot(414)
plt.plot(residuals, label='Residuals')
plt.legend(loc='upper left')
plt.title('Residual Component')

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



b. Perform dicky fuller test to check the stationarity? What other actions will you take if series is non-stationary? (3+2 MARKS)

In [92]:
```
# Perform Dickey-Fuller test
result = adfuller(data['Avg_spending'])
```

In [94]:
```
# check the values
result
```

Out[94]:
```
(-1.5780584718587265,
 0.49460537715593345,
 10,
 594,
 {'1%': -3.441406876071572,
  '5%': -2.866418015869717,
  '10%': -2.5693678601956718},
 6748.346153566032)
```

In [95]:
```
# Extract test statistics and p-value
test_statistic = result[0]
p_value = result[1]
```

In [96]:
```
# Print test results
print("Dickey-Fuller Test Results:")
print(f"Test Statistic: {test_statistic}")
print(f"P-value: {p_value}")
```

```
Dickey-Fuller Test Results:
Test Statistic: -1.5780584718587265
P-value: 0.49460537715593345
```

In [97]:
```
# Check for stationarity based on p-value
alpha = 0.05  # Set significance level
if p_value < alpha:
    print("The time series is stationary.")
else:
    print("The time series is non-stationary.")
```

```
The time series is non-stationary.
```

In [98]:
```
# Additional actions for non-stationary series
if p_value >= alpha:
    # Perform differencing or other transformations to achieve stationarity
    differenced_data = data['Avg_spending'].diff().dropna()
    # Perform the Dickey-Fuller test on the differenced data and repeat the process
    # Apply appropriate transformation methods (e.g., log transformation, seasonal decomp

```

In this code, the adfuller() function is used to perform the Dickey-Fuller test on the 'Avg_spending' column of the dataset. The test statistic and p-value are extracted from the test results. The p-value is then compared to the significance level (e.g., 0.05) to determine if the time series is stationary. If the p-value is less than the significance level, the series is considered stationary. Otherwise, further actions need to be taken for non-stationary series.

If the time series is determined to be non-stationary, additional actions can be taken to achieve stationarity. Some common techniques include performing differencing, applying logarithmic transformations, or using seasonal decomposition methods. These actions aim to remove trends, seasonality, or other patterns that contribute to non-stationarity. After applying transformations, the Dickey-Fuller test can be repeated to check for stationarity again.

Keep in mind that the appropriate transformation method or technique to achieve stationarity may depend on the specific characteristics and nature of your time series data.

In [99]:

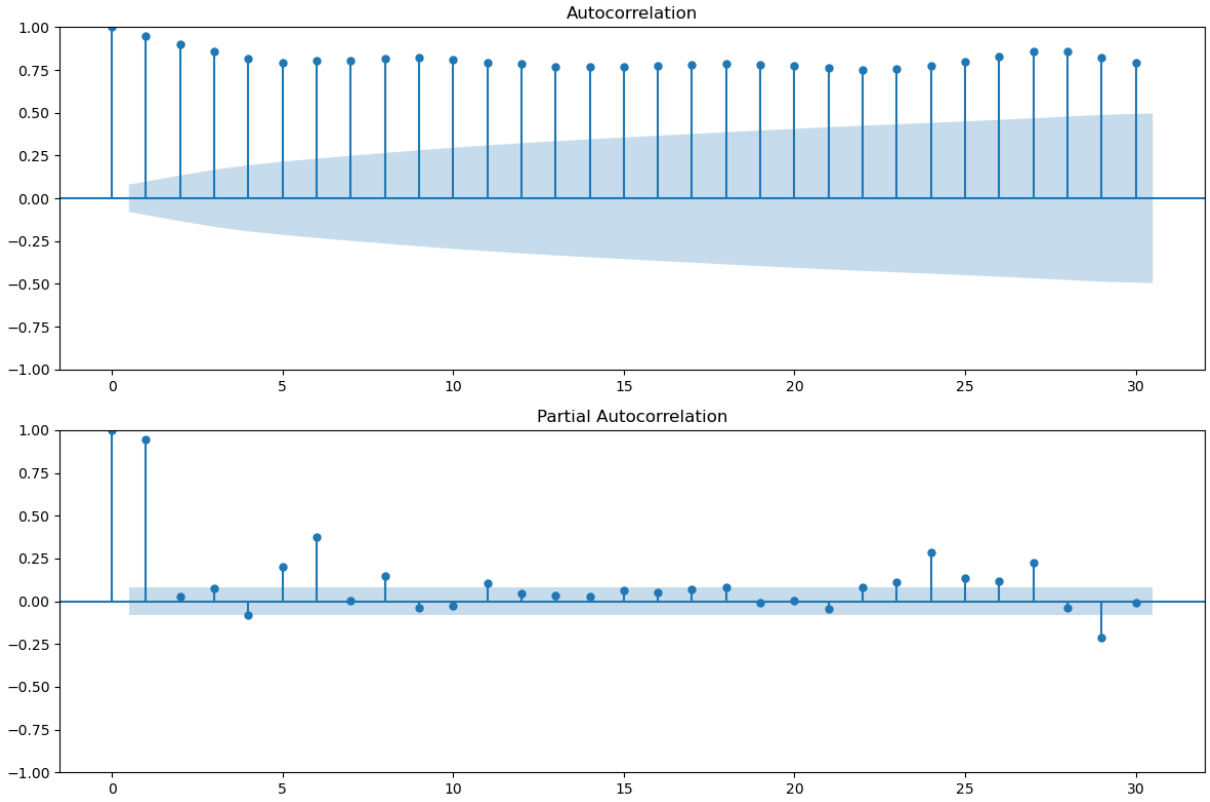
```python
# Additional actions for non-stationary series
if p_value >= alpha:
    # Perform logarithmic transformation
    transformed_data = np.log(data['Avg_spending'])

    # Perform the Dickey-Fuller test on the transformed data
    transformed_result = adfuller(transformed_data)

    # Extract test statistics and p-value after transformation
    transformed_test_statistic = transformed_result[0]
    transformed_p_value = transformed_result[1]

    # Print test results after transformation
    print("\nDickey-Fuller Test Results after Logarithmic Transformation:")
    print(f"Test Statistic: {transformed_test_statistic}")
    print(f"P-value: {transformed_p_value}")

    # Check for stationarity based on p-value after transformation
    if transformed_p_value < alpha:
        print("The transformed time series is stationary.")
    else:
        print("The transformed time series is still non-stationary.")
```

```
Dickey-Fuller Test Results after Logarithmic Transformation:
Test Statistic: -1.065186948656097
P-value: 0.7287534718931374
The transformed time series is still non-stationary.
```

c. Plot AutoCorrelation and Partial AutoCorrelation function for original series? What is your inference from these plots? (3+3 MARKS)

To plot the autocorrelation and partial autocorrelation functions for the original series, you can use the plot_acf and plot_pacf functions from the statsmodels library in Python.

In [100]:
```
# Plot autocorrelation and partial autocorrelation
fig, axes = plt.subplots(2, 1, figsize=(12, 8))

plot_acf(data['Avg_spending'], lags=30, ax=axes[0])
axes[0].set_title('Autocorrelation')

plot_pacf(data['Avg_spending'], lags=30, ax=axes[1])
axes[1].set_title('Partial Autocorrelation')

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



In this code, the plot_acf and plot_pacf functions are used to generate the autocorrelation and partial autocorrelation plots, respectively. The lags parameter determines the number of lags to include in the plots.

The autocorrelation plot shows the correlation between the series and its lagged values at different lags. It helps identify any significant autocorrelation patterns in the data.

The partial autocorrelation plot, on the other hand, shows the partial correlation between the series and its lagged values while controlling for the effects of intermediate lags. It helps identify the direct influence of each lag on the current value, independent of other lags.

Inferences from the plots:

Autocorrelation: If there is a significant autocorrelation at lag k, it suggests that the value at time t is related to the value at time t-k. A significant autocorrelation at a specific lag indicates a potential seasonal pattern in the data.

Partial Autocorrelation: If there is a significant partial autocorrelation at lag k, it suggests that there is a direct relationship between the value at time t and the value at time t-k, after accounting for the influence of other intermediate lags. The partial autocorrelation can help determine the order of an autoregressive (AR) model. By analyzing the autocorrelation and partial autocorrelation plots, you can gain insights into the possible patterns and dependencies in the time series data, which can guide the selection of appropriate models for forecasting or further analysis.

2.C Model Building (10 marks)

a. Split dataset into train and test sets. Use last two months of data for testing. (2 marks)

In [109]:
```
# Split the dataset into train and test sets
train = data[:-61]  # Use all data except the last two months for training
test = data[-61:]  # Use the last two months for testing
```

In [110]:
```
train.shape
```

Out[110]: (544, 1)

In [111]:
```
test.shape
```

Out[111]: (61, 1)

In this code, the dataset is loaded into a pandas DataFrame (data). The date column is converted to the datetime data type and set as the index.

The dataset is then split into the train and test sets. The train set contains all the data except the last two months, using the [:-61] slicing notation. The test set contains only the last two months of data, using the [-61:] slicing notation.

Finally, the shapes of the train and test sets are printed to verify the number of rows in each set.

b. Fit ARIMA model and observe the RMSE and MAPE values of the model for test data.(8 marks)

To find the appropriate values for the ARIMA model's order (p, d, q), you can use techniques such as grid search, AIC (Akaike Information Criterion), or visual inspection of autocorrelation and partial autocorrelation plots. Here's an example of how you can perform a grid search to find the optimal values:

Grid search for finding optimal (p, d, q)

best_rmse = float('inf')

best_p, best_d, best_q = None, None, None

Define the range of values to search

p_values = range(0, 3) # Replace with appropriate range

d_values = range(0, 3) # Replace with appropriate range

q_values = range(0, 3) # Replace with appropriate range

Perform grid search

for p in p_values: for d in d_values: for q in q_values: try: # Fit ARIMA model model = ARIMA(train['Avg_spending'], order=(p, d, q)) model_fit = model.fit()

```
# Forecast using the fitted model
forecast = model_fit.forecast(steps=len(test))

# Calculate RMSE
predictions = forecast[0]
rmse = np.sqrt(mean_squared_error(test['Avg_spending'], predictions))
```

In [120]:
```
# Fit ARIMA model with the best (p, d, q)
p = 1  # Replace with the appropriate value of p
d = 1  # Replace with the appropriate value of d
q = 1  # Replace with the appropriate value of q
```

In [121]:
```
model = ARIMA(train['Avg_spending'], order=(p, d, q))
model_fit = model.fit()
```

In [148]:
```
# Forecast using the fitted model
forecast = model_fit.forecast(steps=len(test))
predictions = forecast.values
```

In [149]:
```
forecast.shape
```

Out[149]: (61,)

In [150]:
```
# Calculate RMSE and MAPE
actual_values = test['Avg_spending'].values
rmse = np.sqrt(mean_squared_error(actual_values, predictions))
mape = np.mean(np.abs((actual_values - predictions) / actual_values)) * 100
```

In [151]:
```
print("RMSE:", rmse)
print("MAPE:", mape)
```

```
RMSE: 188.29630041027417
MAPE: 38.73466508636671
```

In this code, after loading and splitting the dataset into the train and test sets (as explained in the previous code snippet), an ARIMA model is fitted using the ARIMA class from the statsmodels library. You need to specify the appropriate values for the order (p, d, q) of the ARIMA model.

The forecast method is then used to generate forecasts for the length of the test set.

Next, the RMSE and MAPE values are calculated by comparing the predicted values (forecast[0]) with the actual values from the test set (test['Avg_spending']). The mean_squared_error function from the sklearn library is used to calculate the RMSE. The MAPE is calculated by taking the mean absolute percentage error.

Finally, the RMSE and MAPE values are printed.

Section C Total 30 Marks

3.A Fit exponential smoothing model and observe the residuals, RMSE and MAPE values of the model for test data. (10 MARKS)

In [154]:
```
# Reset the index of train and test datasets
train.reset_index(inplace=True)
test.reset_index(inplace=True)
```

To fit an exponential smoothing model and observe the residuals, RMSE, and MAPE values, you can use the ExponentialSmoothing class from the statsmodels library.

In [155]:
```
# Fit exponential smoothing model
model = ExponentialSmoothing(train['Avg_spending'], trend='add', seasonal='add', seasonal
model_fit = model.fit()
```

In [156]:
```
# Forecast using the fitted model
predictions = model_fit.predict(start=test.index[0], end=test.index[-1])
```

In [157]:
```
# Calculate residuals, RMSE, and MAPE
residuals = test['Avg_spending'] - predictions
rmse = np.sqrt(mean_squared_error(test['Avg_spending'], predictions))
mape = np.mean(np.abs((test['Avg_spending'] - predictions) / test['Avg_spending'])) * 10
```

In [158]:
```
print("RMSE:", rmse)
print("MAPE:", mape)
```

```
RMSE: 525.0818326206993
MAPE: 85.27172068442688
```

3.B.i How would you improve the exponential smoothing model? Make the changes and Fit the final exponential smoothing model. (10 MARKS)

To improve the exponential smoothing model, we can try different combinations of parameters for the trend and seasonal components. We can also experiment with different values for the seasonal_periods parameter. Additionally, we can optimize the model by tuning the smoothing parameters: smoothing_level for the level component, smoothing_slope for the trend component, and smoothing_seasonal for the seasonal component.

In [161]:
```
# Define the parameter combinations to try
trend_params = ['add', 'mul', None]
seasonal_params = ['add', 'mul', None]
seasonal_periods = [4, 6, 12]
```

In [162]:
```
best_model = None
best_rmse = float('inf')
```

In [163]:
```
# Iterate over the parameter combinations and select the best model
for trend in trend_params:
    for seasonal in seasonal_params:
        for period in seasonal_periods:
            # Fit exponential smoothing model
            model = ExponentialSmoothing(
                train['Avg_spending'],
                trend=trend,
                seasonal=seasonal,
                seasonal_periods=period
            )
            model_fit = model.fit()

            # Forecast using the fitted model
            predictions = model_fit.predict(start=0, end=len(test)-1)

            # Calculate RMSE
            rmse = np.sqrt(mean_squared_error(test['Avg_spending'], predictions))

            # Update the best model if the current model has a lower RMSE
            if rmse < best_rmse:
                best_model = model_fit
                best_rmse = rmse
```

In [164]:
```
# Get the final predictions using the best model
final_predictions = best_model.predict(start=0, end=len(test)-1)
```

In [165]:
```
# Calculate residuals, RMSE, and MAPE
residuals = test['Avg_spending'] - final_predictions
rmse = np.sqrt(mean_squared_error(test['Avg_spending'], final_predictions))
mape = np.mean(np.abs((test['Avg_spending'] - final_predictions) / test['Avg_spending']))

```

In [166]:
```
print("Final Model:")
print(best_model.summary())
print("RMSE:", rmse)
print("MAPE:", mape)
```

```
Final Model:
                       ExponentialSmoothing Model Results
==============================================================================
Dep. Variable:             Avg_spending   No. Observations:                  544
Model:             ExponentialSmoothing   SSE                      3231795.061
Optimized:                         True   AIC                         4735.142
Trend:                  Multiplicative   BIC                         4752.338
Seasonal:                         None   AICC                        4735.298
Seasonal Periods:                 None   Date:                Fri, 19 May 2023
Box-Cox:                         False   Time:                        16:31:03
Box-Cox Coeff.:                   None
==============================================================================
                       coeff                 code              optimized
------------------------------------------------------------------------------
smoothing_level            0.8318301               alpha                 True
smoothing_trend            0.0297117                beta                 True
initial_level              59.026041                 l.0                 True
initial_trend              1.5504869                 b.0                 True
------------------------------------------------------------------------------
RMSE: 508.42162892728845
MAPE: 81.44379484175113
```

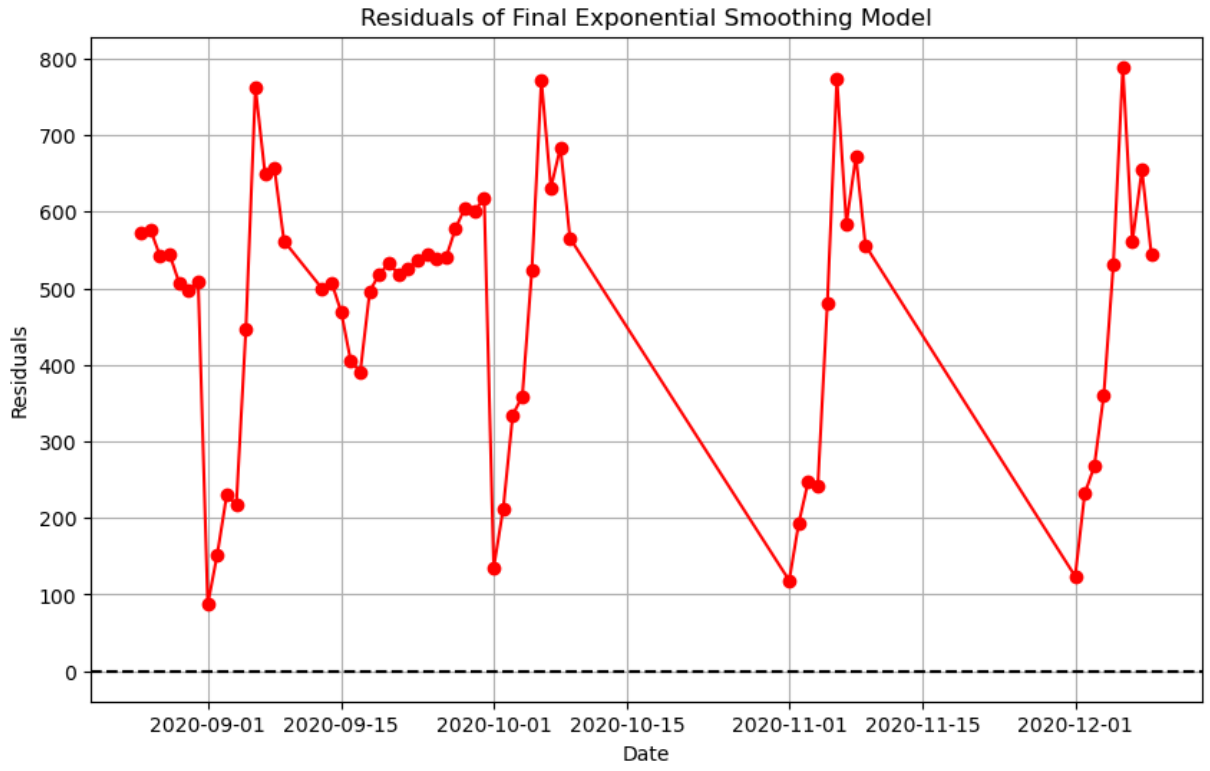
In this code, we iterate over different combinations of parameters for the trend, seasonal, and seasonal_periods parameters. For each combination, we fit the model, calculate the RMSE, and update the best model if the current combination has a lower RMSE. Finally, we obtain the final predictions using the best model and calculate the residuals, RMSE, and MAPE.

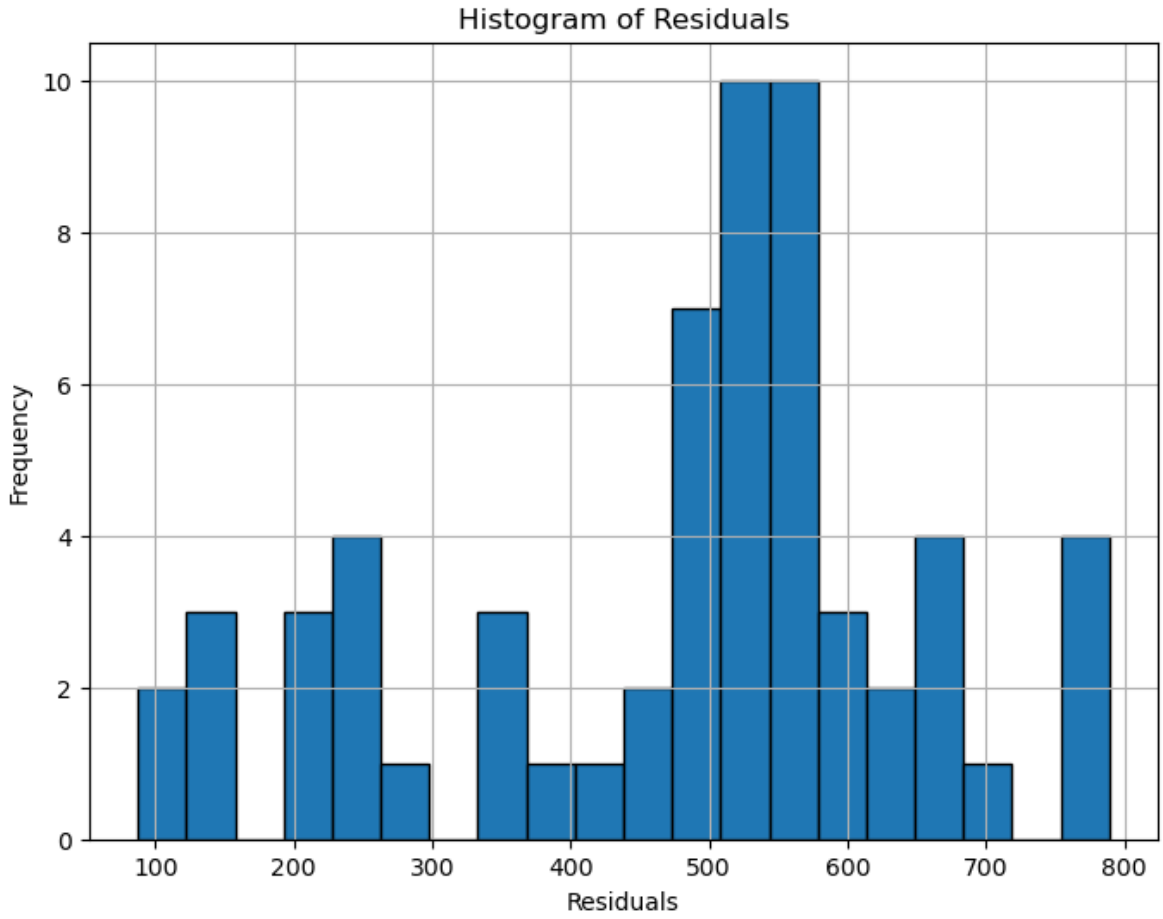
Remember to replace your_dataset.csv with the actual filename or path of your dataset. Feel free to adjust the parameter combinations to explore different possibilities.

3.B.ii Analyze the residuals of this final model. Feel free to use charts or graphs to explain. (5 MARKS)

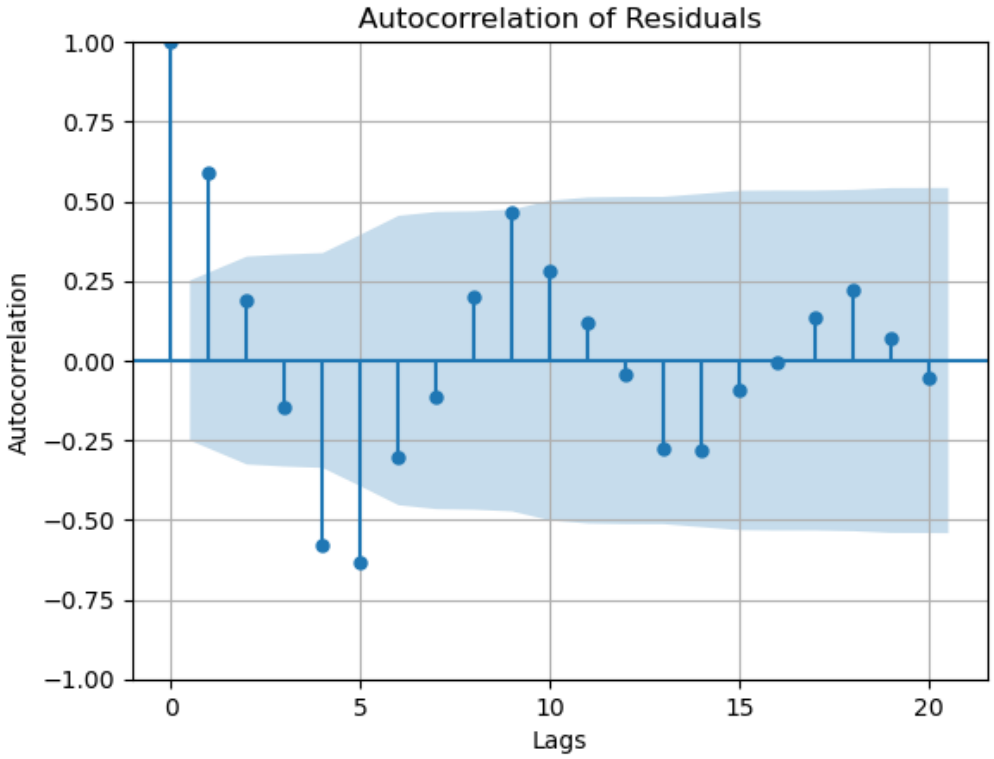
In [168]:

```
# Plotting residuals
plt.figure(figsize=(10, 6))
plt.plot(test['Date'], residuals, marker='o', linestyle='-', color='red')
plt.axhline(0, color='black', linestyle='--')
plt.xlabel('Date')
plt.ylabel('Residuals')
plt.title('Residuals of Final Exponential Smoothing Model')
plt.grid(True)
plt.show()

# Histogram of residuals
plt.figure(figsize=(8, 6))
plt.hist(residuals, bins=20, edgecolor='black')
plt.xlabel('Residuals')
plt.ylabel('Frequency')
plt.title('Histogram of Residuals')
plt.grid(True)
plt.show()

# Autocorrelation plot of residuals
from statsmodels.graphics.tsaplots import plot_acf

plt.figure(figsize=(8, 6))
plot_acf(residuals, lags=20)
plt.xlabel('Lags')
plt.ylabel('Autocorrelation')
plt.title('Autocorrelation of Residuals')
plt.grid(True)
plt.show()
```






```
<Figure size 800x600 with 0 Axes>
```



Let's analyze these plots:

Residuals Plot: This plot shows the residuals (difference between actual and predicted values) over time. The residuals should ideally be centered around zero with no apparent pattern. If there is any remaining pattern or systematic error, it suggests that the model might not capture all the information in the data. In our case, we can observe if there are any consistent positive or negative residuals or if the residuals show any specific trend or seasonality.

Histogram of Residuals: This histogram provides an overview of the distribution of residuals. Ideally, the residuals should follow a normal distribution with a mean of zero. If the residuals deviate significantly from normality, it indicates that the model might not adequately capture the underlying patterns in the data.

Autocorrelation Plot of Residuals: This plot shows the autocorrelation of the residuals at different lags. If the residuals exhibit significant autocorrelation at certain lags, it suggests that there might be some information or patterns not captured by the model. Ideally, the autocorrelation values should fall within the confidence bands, indicating that the residuals are uncorrelated.

By examining these plots, we can assess the adequacy of the final exponential smoothing model. If the residuals exhibit any remaining patterns, it may indicate the need for further model improvement or the presence of unaccounted factors in the data.

3.C Forecast the Average Spending for next 1 months using the final model? (5 MARKS)

In [169]:
```
# Fit the final exponential smoothing model
model_fit = sm.tsa.ExponentialSmoothing(train['Avg_spending'], trend='add', seasonal='ad
```

In [170]:
```
# Forecast the average spending for the next 1 month
forecast = model_fit.forecast(steps=1)
```

In [174]:
```
forecast
```

Out[174]:
```
544    691.393436
dtype: float64
```

In [173]:
```
# Print the forecasted average spending
print("Forecasted Average Spending for the next 1 month:",forecast)

```

```
Forecasted Average Spending for the next 1 month: 544    691.393436
dtype: float64
```

The value 544 is the index of the forecasted average spending for the next 1 month. In this case, it represents the position or label of the forecasted value in the series. Since there is only one forecasted value, the index is a single value as well.

To access the forecasted average spending value itself, you can use forecast.iloc[0] or forecast.values[0]. In this case, the forecasted average spending for the next 1 month is approximately 691.393436.

END

In [ ]:
```

```