Final Project: Daily Visits to Website

File: website_traffic_data.csv

This dataset records data related to website traffic.

1.0 Preparing Jupyter Notebook

Package Downloads

In [ ]:
```
#Import necessary packages and set options for jupyter notebook
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from statsmodels.tsa.seasonal import seasonal_decompose
from sklearn.metrics import mean_absolute_error, mean_squared_error
from statsmodels.graphics.tsaplots import plot_acf, plot_pacf
from sklearn.linear_model import LinearRegression
from statsmodels.tsa.stattools import adfuller
from itertools import product
from statsmodels.tsa.arima.model import ARIMA
import warnings
```

Jupyter Notebook Settings

In [ ]:
```
#Set the maximum number of rows that can be observed
pd.set_option('display.max_rows', 45)
pd.set_option('display.max_columns',100)
```

Function Creation

In [ ]:
```
#Create a function that is going to plot data
#It is called time series because that is the type of data it is going to be primarily be dealing with
def time_series(xdata,ydata,title,xlabel,ylabel):
    plt.figure(figsize=(10,6))
    plt.plot(xdata,ydata)
    plt.title(title)
    plt.xlabel(xlabel)
    plt.ylabel(ylabel)
```

In [ ]:
```
#Create a scatterplot graph
def scatter(xdata,ydata,title,xlabel,ylabel):
    plt.figure(figsize=(14,6))
    plt.scatter(xdata,ydata)
    plt.title(title)
    plt.xlabel(xlabel)
    plt.ylabel(ylabel)
```

In [ ]:
```
#Create a seasonal decomposition of the different parts that make up the data
#Creating a subplot where the values are stacked up on one another makes it easier to read

def plot_seasonal_decomposition(data, period):
    decomposition = seasonal_decompose(data, period=period)

    plt.figure(figsize=(12, 8))

    plt.subplot(411)
    plt.plot(data, label='Original')
    plt.legend()

    plt.subplot(412)
    plt.plot(decomposition.trend, label='Trend')
    plt.legend()

    plt.subplot(413)
    plt.plot(decomposition.seasonal, label='Seasonal')
    plt.legend()

    plt.subplot(414)
    plt.plot(decomposition.resid, label='Residual')
    plt.legend()

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

In [ ]:
```
#Create a rolling mean of the data to understand the trend

def roll_mean(df_column,window,title,xlabel,ylabel):

    plt.figure(figsize=(10,6))
    rolmean = df_column.rolling(window).mean()

    # Plot the original data and the rolling mean
    plt.figure(figsize=(10, 6))
    plt.plot(df_column, color='blue', label='Original')
    plt.plot(rolmean, color='red', label=f'Rolling Mean (window={window})')

    # Add title and labels
    plt.title(title)
    plt.xlabel(xlabel)
    plt.ylabel(ylabel)

    # Add legend and display plot
    plt.legend(loc='best')
    plt.show()
```

In [ ]:
```python
#Get the decomposed values which make up the time series
def decompose_seasonality(data, period):

    decomposition = seasonal_decompose(data, period=period)

    original = data
    trend = decomposition.trend
    seasonal = decomposition.seasonal
    residual = decomposition.resid

    return original, trend, seasonal, residual
```

In [ ]:
```python
#Create a least squares line going through the data values

def plot_least_squares_line(x_value, y_value, title, x_label, y_label):
    plt.figure(figsize=(10,6))
    # Extracting x and y values from the dataframe
    x = x_value.values.reshape(-1, 1).astype(float)
    y = y_value.astype(float)

    # Fitting the linear regression model
    model = LinearRegression()
    model.fit(x, y)

    # Predicting y values using the model
    y_pred = model.predict(x)

    # Plotting the original data and the least squares line
    plt.plot(x, y, label='original data', color='blue')
    plt.plot(x, y_pred, label='least squares', color='red')

    plt.legend()
    plt.title(title)
    plt.xlabel(x_label)
    plt.ylabel(y_label)
    plt.show()
```

In [ ]:
```python
#Create the Dickey Fuller test to use on data
def adf_test(data):
    """Using the ADF test to determine if a series is stationary"""
    test_results = adfuller(data)
    print('ADF Statistic: ',test_results[0])
    print('P-Value: ',test_results[1])
    print('Critical Value: ',)
    for thres,adf_stat in test_results[4].items():
       print('\t%s: %.2f' % (thres,adf_stat))
```

In [ ]:
```python
warnings.filterwarnings("ignore", message="No frequency information was provided, so inferred frequency")
warnings.filterwarnings("ignore", category=UserWarning)
warnings.filterwarnings("ignore", category=RuntimeWarning)

def evaluate_time_series_models(data, ar_orders, ma_orders, d_orders):
    results = []

    # Evaluate AR models
    for p in ar_orders:
        try:
            model = ARIMA(data, order=(p, 0, 0)).fit()
            results.append(('AR', (p, 0, 0), model.aic, model.bic))
        except Exception as e:
            continue

    # Evaluate MA models
    for q in ma_orders:
        try:
            model = ARIMA(data, order=(0, 0, q)).fit()
            results.append(('MA', (0, 0, q), model.aic, model.bic))
        except Exception as e:
            continue

    # Evaluate ARMA models
    for p, q in product(ar_orders, ma_orders):
        try:
            model = ARIMA(data, order=(p, 0, q)).fit()
            results.append(('ARMA', (p, 0, q), model.aic, model.bic))
        except Exception as e:
            continue

    # Evaluate ARIMA models
    for p, d, q in product(ar_orders, d_orders, ma_orders):
        try:
            model = ARIMA(data, order=(p, d, q)).fit()
            results.append(('ARIMA', (p, d, q), model.aic, model.bic))
        except Exception as e:
            continue

    result_df = pd.DataFrame(results, columns=['Model', 'Order', 'AIC', 'BIC'])
    return result_df.sort_values(by=['AIC'])
```

In [ ]:
```python
from pmdarima import auto_arima

def select_best_sarima_model(data, p_list, d_list, q_list, P_list, D_list, Q_list, s):
    """
    Fits a SARIMA model to the provided data, searching over given parameter lists.
    It returns the best model based on AIC and also provides its BIC for comparison.

    Parameters:
    - data: Pandas Series of the time series data.
    - p_list, d_list, q_list: Lists of integers for the AR (p), differencing (d), and MA (q) parameters.
    - P_list, D_list, Q_list: Lists of integers for the seasonal AR (P), seasonal differencing (D),
                              and seasonal MA (Q) parameters.
    - s: Integer representing the length of the seasonal cycle.

    Returns:
    - A dictionary with the best model's AIC and BIC, along with the corresponding parameters.
```

```
    """

    # Expand the parameter space for auto_arima to search
    model = auto_arima(data, start_p=min(p_list), d=min(d_list), start_q=min(q_list),
                       max_p=max(p_list), max_d=max(d_list), max_q=max(q_list),
                       start_P=min(P_list), D=min(D_list), start_Q=min(Q_list),
                       max_P=max(P_list), max_D=max(D_list), max_Q=max(Q_list),
                       seasonal=True, m=s, stepwise=True,
                       suppress_warnings=True, trace=False,
                       error_action='ignore', information_criterion='aic')

    # Extract the AIC and BIC from the selected model
    results = {
        'aic': model.aic(),
        'bic': model.bic(),
        'pdq': model.order,
        'seasonal_pdq': model.seasonal_order
    }

    return results
```

In [ ]:
```
def test_training_model(train_data, test_data, predictions,
                        x_label, y_label, title,prediction_label):
    # Plot original data
    plt.figure(figsize=(10, 6))
    plt.plot(train_data.index, train_data, label='Train Data', color='cornflowerblue')
    plt.plot(test_data.index, test_data, label='Test Data', color='mediumseagreen')
    plt.plot(predictions.index, predictions, label=prediction_label, color='darkorange',linestyle='--')

    # Add labels and title
    plt.xlabel(x_label)
    plt.ylabel(y_label)
    plt.title(title)

    # Add legend
    plt.legend()

    # Show plot
    plt.show()
```

In [ ]:
```
def model_forecast(original_data, original_data_sales, predictions,
                        x_label, y_label, title, prediction_label):
    # Plot original data
    plt.figure(figsize=(10, 6))
    plt.plot(original_data.index, original_data_sales, label='Original Data', color='cornflowerblue')
    plt.plot(predictions.index, predictions, label=prediction_label, color='mediumseagreen', linestyle='--')

    # Add labels and title
    plt.xlabel(x_label)
    plt.ylabel(y_label)
    plt.title(title)

    # Add legend
    plt.legend()

    # Show plot
    plt.show()
```

1.1 Data Preprocessing:

1. Load the dataset and convert the date information into a datetime object to facilitate time series analysis.

2. Check for missing values and anomalies, and handle them appropriately.

Import data

In [ ]:
```
#Import the data and save it as website
website = pd.read_csv('/Users/ben_nicholson/Visual_Code_Projects/Applied Statistics & Modeling/Final Project/Data Files/w
website
```

Out[ ]:

	Date	Daily_Visitors
0	2020-01-01	202
1	2020-01-02	204
2	2020-01-03	197
3	2020-01-04	197
4	2020-01-05	239
...	...	...
1456	2023-12-27	185
1457	2023-12-28	202
1458	2023-12-29	234
1459	2023-12-30	194
1460	2023-12-31	208

1461 rows × 2 columns

Preprocess dataframe

In [ ]:
```
#Rename column 'Daily_Visitors' to 'Daily Visitors'
website.rename(columns={'Daily_Visitors': 'Daily Visitors'}, inplace=True)
```

In [ ]:
```
#Set the 'Date' column to datetime
website['Date'] = pd.to_datetime(website['Date'])
```

In [ ]:
```
#Set the index to the 'Date' column
website.set_index('Date',inplace=True)
website
```

Out[ ]:

Date	Daily Visitors
2020-01-01	202
2020-01-02	204
2020-01-03	197
2020-01-04	197
2020-01-05	239
...	...
2023-12-27	185
2023-12-28	202
2023-12-29	234
2023-12-30	194
2023-12-31	208

1461 rows × 1 columns

In [ ]:
```
#Double check the columns are in the right format
#Expect '<M8[ns]' for index
#Expect 'float64' or 'int64' for monthly mean
print('Date dtype: ',website.index.dtype)
print('')
print('Daily Visitors dtype: ',website['Daily Visitors'].dtype)
```

```
Date dtype:  datetime64[ns]

Daily Visitors dtype:  int64
```

Check for Missing Values

In [ ]:
```
#Use isna is to create a series of true and false
#sum those to get a total number of missing values
#if it is 0 you can continue
website['Daily Visitors'].isna().sum()
```

Out[ ]:
```
0
```

Check for Anomalies

There are a number of ways to do this. I am going to use the statistical way by creating z scores and see if any of them lie outside of this

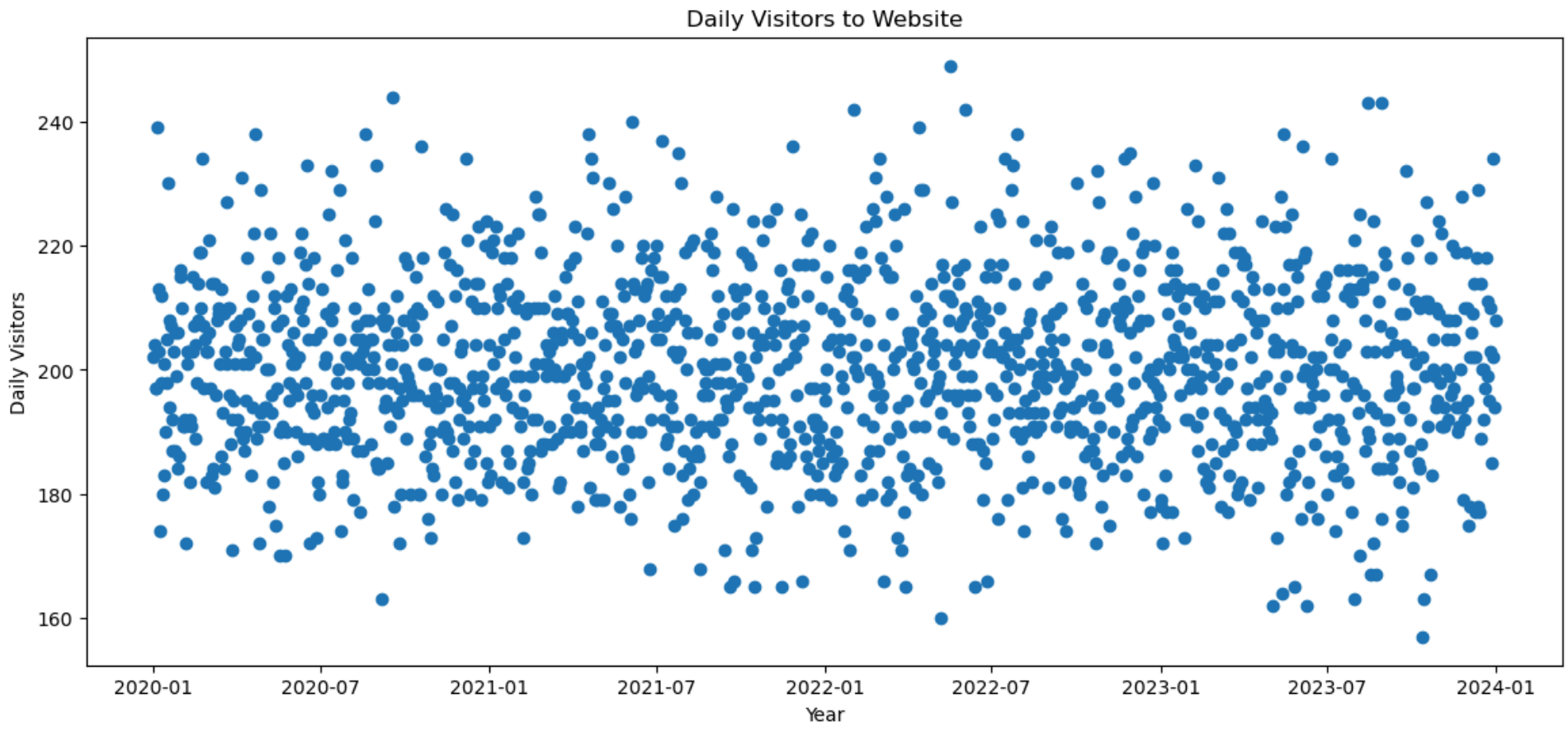
In [ ]:
```
#Create zscore value
website['zscore'] = (website['Daily Visitors'] - website['Daily Visitors'].mean()) / website['Daily Visitors'].std()

#Identify anomalies where their standard deviation is greater than 3
anomalies = website[abs(website['zscore']) > 3]

if len(anomalies) == 0:
    print("No anomalies detected.")
else:
    print("Anomalies detected:")
    print(anomalies)
```

```
Anomalies detected:
            Daily Visitors    zscore
Date
2022-05-17             249  3.332062
```

This data only has one anomaly so there is no need to delete the value with a high zscore

Plot the scatterplot graph

In [ ]:
```
#Find the scatterplot
scatter(website.index,website['Daily Visitors'],'Daily Visitors to Website','Year','Daily Visitors')
```



Aggregate data

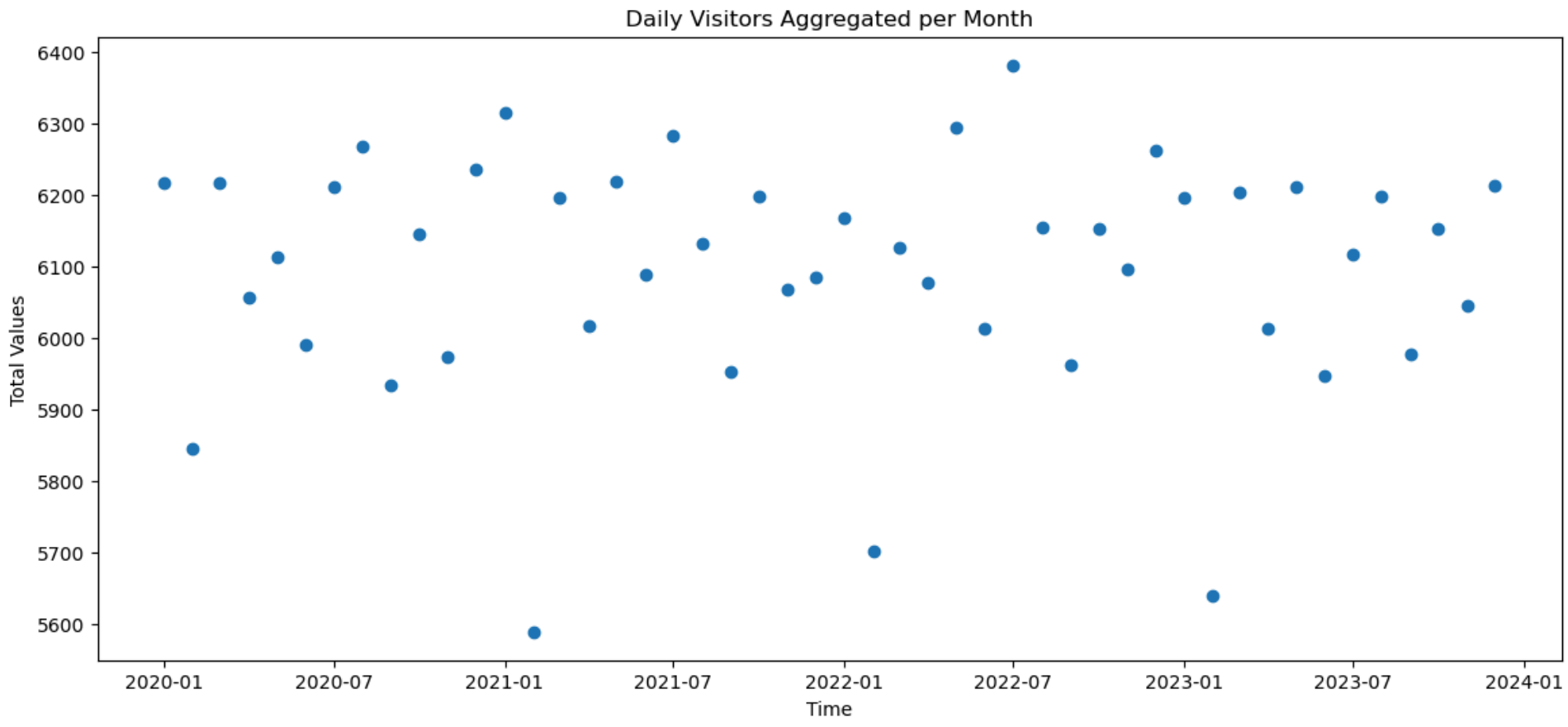
In [ ]:
```
#Create a dataframe which adds the day values based on the month
website_monthly_sum = website.groupby(pd.Grouper(freq='M')).sum()
website_monthly_sum.index= website_monthly_sum.index - pd.offsets.MonthBegin(1)
website_monthly_sum
```

Out[ ]:

Date	Daily Visitors	zscore
2020-01-01	6216	0.468965
2020-02-01	5845	2.493311
2020-03-01	6217	0.537378
2020-04-01	6056	3.225673
2020-05-01	6113	-6.577590
...	...	...
2023-08-01	6197	-0.830885
2023-09-01	5976	-2.247379
2023-10-01	6153	-3.841064
2023-11-01	6045	2.473128
2023-12-01	6213	0.263725

48 rows × 2 columns

In [ ]:
```
#Create a time series graphs of aggregated data based on the month
scatter(website_monthly_sum.index,website_monthly_sum['Daily Visitors'],'Daily Visitors Aggregated per Month','Time','Tot
```



The aggregated data that is summed based on the month of the datetime, shows more of a cycle and this will be the data that is going to be used in the exploratory data analysis

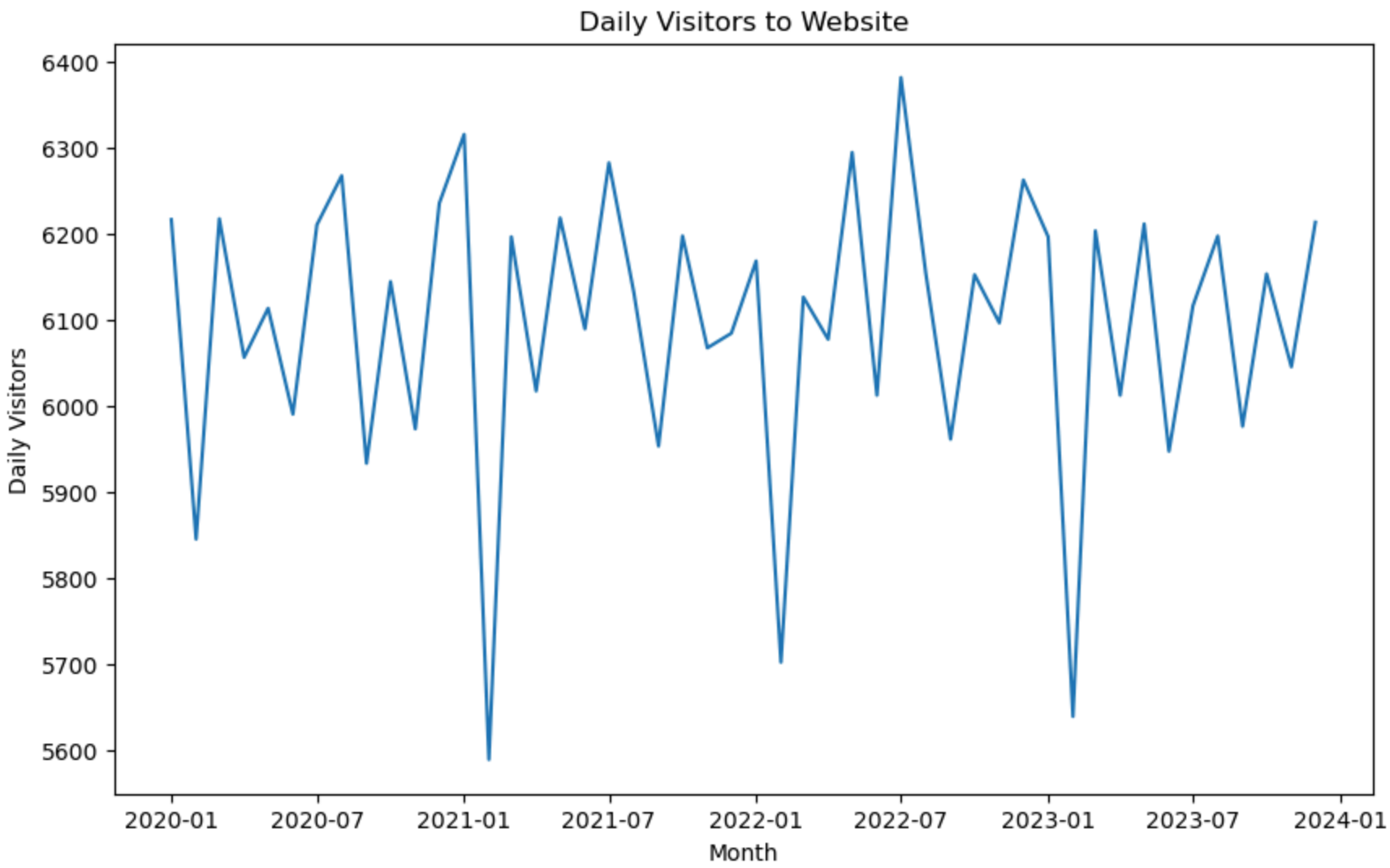
1.2 Exploratory Data Analysis:

1. Visualize the data to understand trends, seasonality, and other characteristics.

2. Decompose the series to observe its components: trend, seasonality, and residuals.

Visualise the data

In [ ]:
```
#Plot the time series graph of website visits data
time_series(website_monthly_sum.index,website_monthly_sum['Daily Visitors'],'Daily Visitors to Website','Month','Daily Vi
```



Exploration of Seasonal Decomposition for website

Seasonality Period Decision

The seasonality does not become very apparent so using the following methods, a seasonal value is going to be determined

1. Data Visualisation - Taking the rolling mean
2. ACF & PACF Plot

1. Data Visualisation

Look at the rolling average of the data to try and identify trends in the data which might lead to a better understanding seasonal time period.

There does seem to be some type of oscillations that take place when you look at the quarterly rolling average.

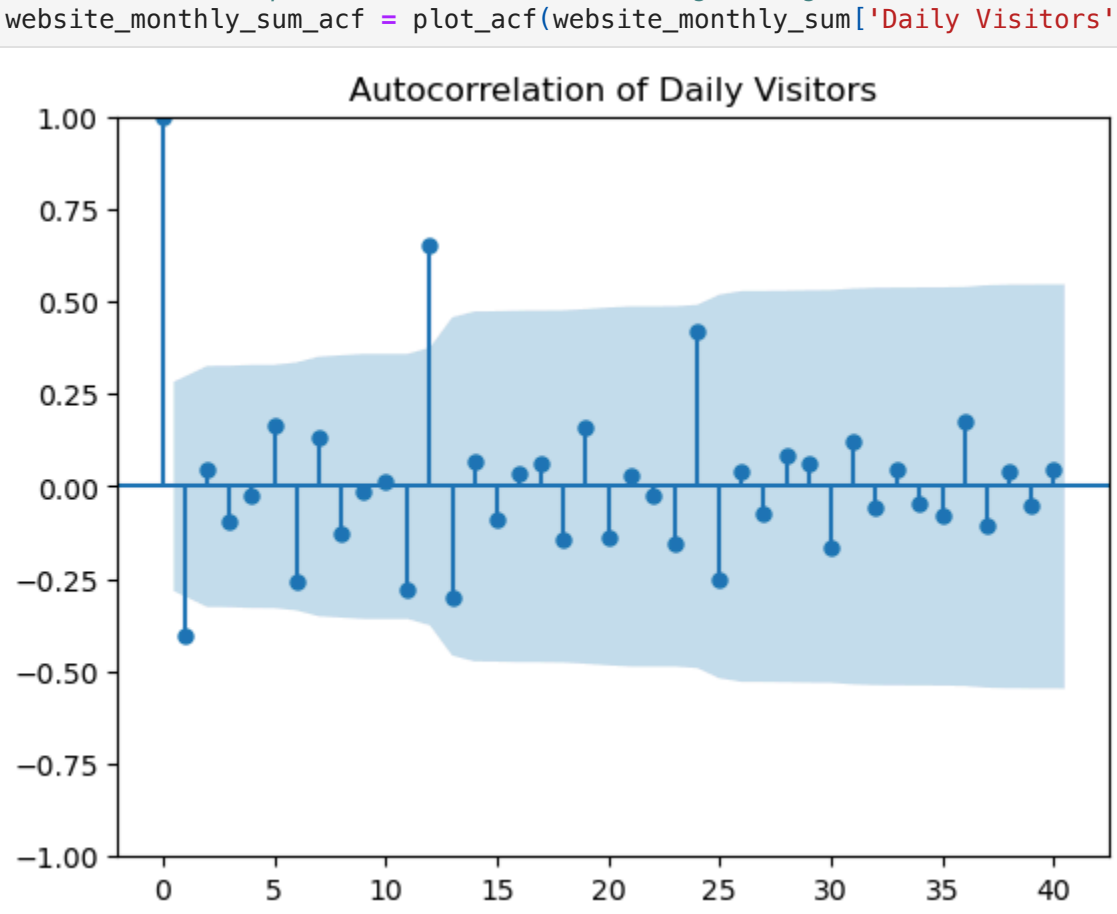
2. ACF & PACF Plots

Looking at the ACF and PACF plots is going to give insight for different things

1. ACF plot - Is looking at the moving average nature of the graph, this is how much values from lag k-1 impact lag k.

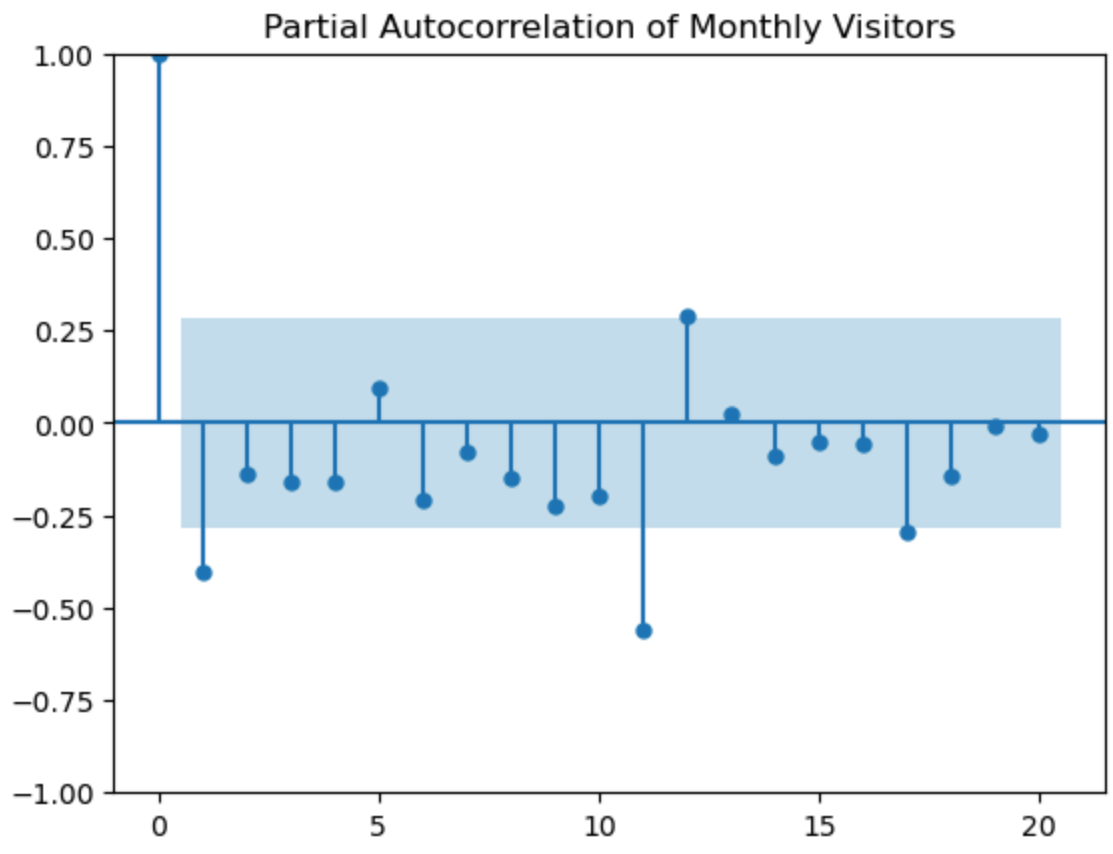
2. PACF plot - Is looking at the autoregressive nature of the graph, this is how values from lag k impact the current value

In [ ]:
```
#Look at the acf plot to determine the moving average
website_monthly_sum_acf = plot_acf(website_monthly_sum['Daily Visitors'],lags=40,title='Autocorrelation of Daily Visitors
```



You can observe the spike of autocorrelation at lag 12.

In [ ]:
```
#Look at the acf plot to determine the moving average
website_monthly_sum_pacf = plot_pacf(website_monthly_sum['Daily Visitors'],lags=20,title='Partial Autocorrelation of Montl
```

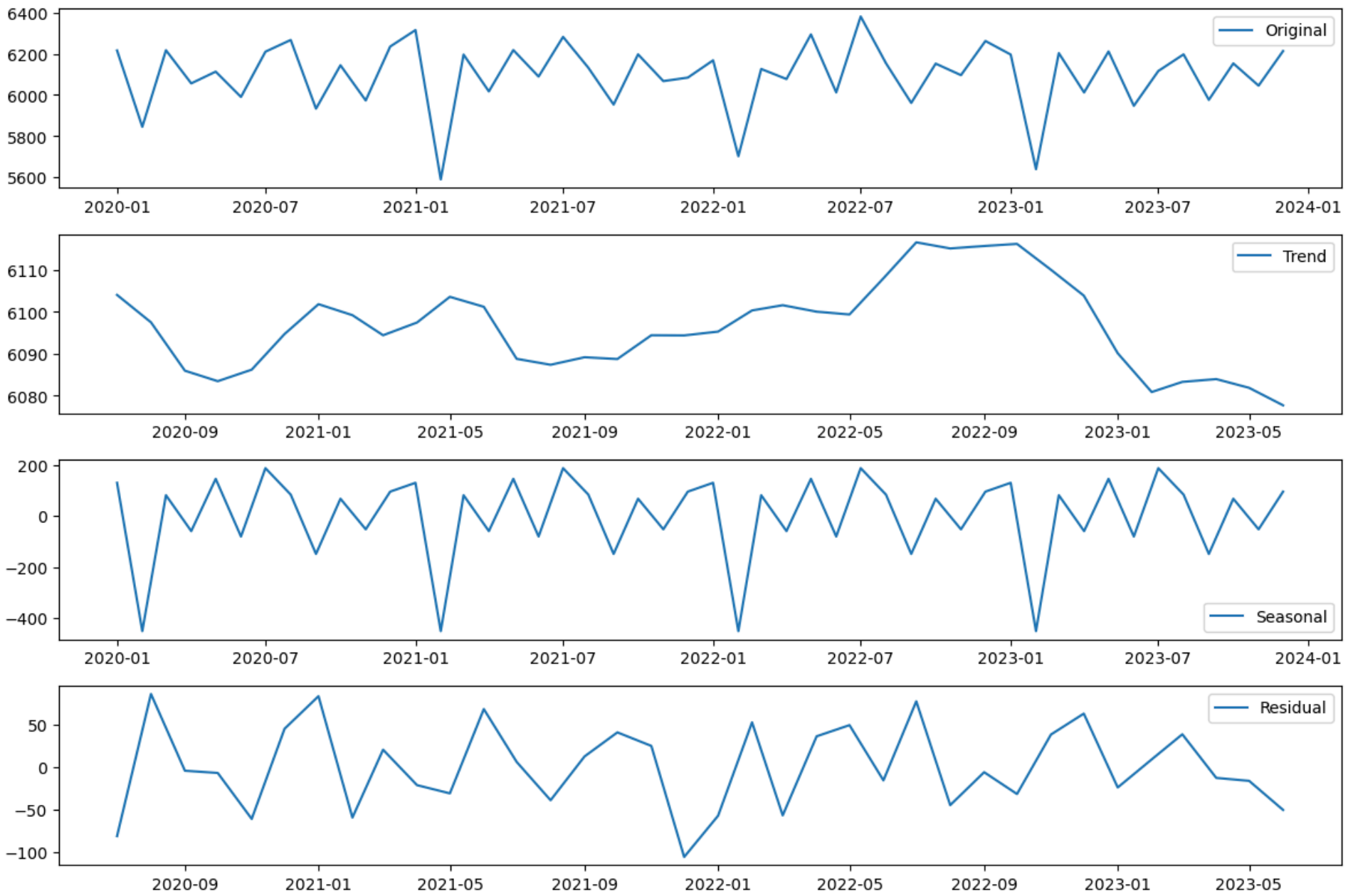


There is a spike at 12 again, I am going to use this as the seasonality value

Seasonal Decomposition of Website Daily Visitors

Using a period of 12 for the monthly summed data this is going to assume that the season is a year.

In [ ]:
```
#Create a variable which can be referrenced to get the different series below
#Plot the decomposition with a period of a year as the seasonality
website_seasonal_decomposition = decompose_seasonality(website_monthly_sum['Daily Visitors'],12)
plot_seasonal_decomposition(website_monthly_sum['Daily Visitors'],12)
```



Trend of Website Decomposition

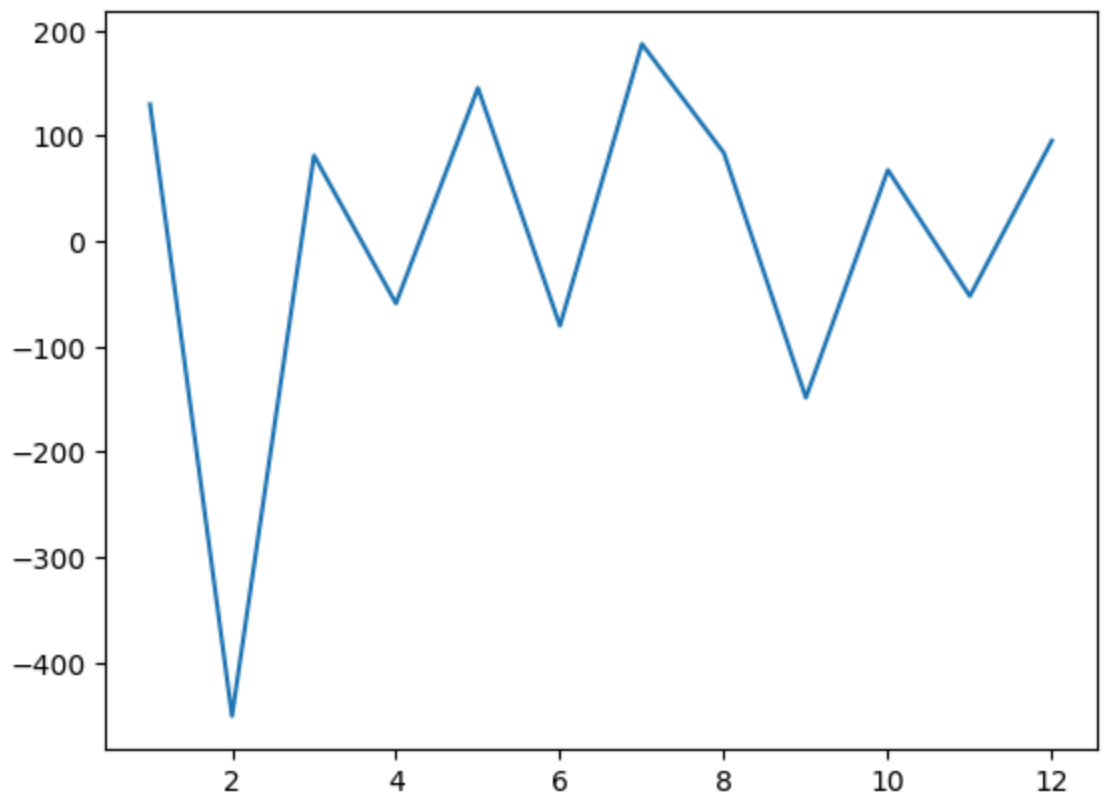
You can see that the difference is an extremely small movement so there is no overall change in values based on the time change

Seasonal of Website Decomposition

There is a seasonal value is quite small again, so it does not represent that signifigant of changes based upon the time of the year the data was recorded.

In [ ]:
```
#Create a dataframe for the first year of the seasonality
#This will return 12 values
#You can observe the seasonality by itself
months_list = [
    'January', 'February', 'March', 'April', 'May', 'June',
    'July', 'August', 'September', 'October', 'November', 'December'
]
website_seasonal_values = pd.DataFrame(website_seasonal_decomposition[2].iloc[0:12])
website_seasonal_values['Incrementing Month'] = range(1, len(website_seasonal_values)+1)
plt.plot(website_seasonal_values['Incrementing Month'],website_seasonal_values['seasonal'])
```

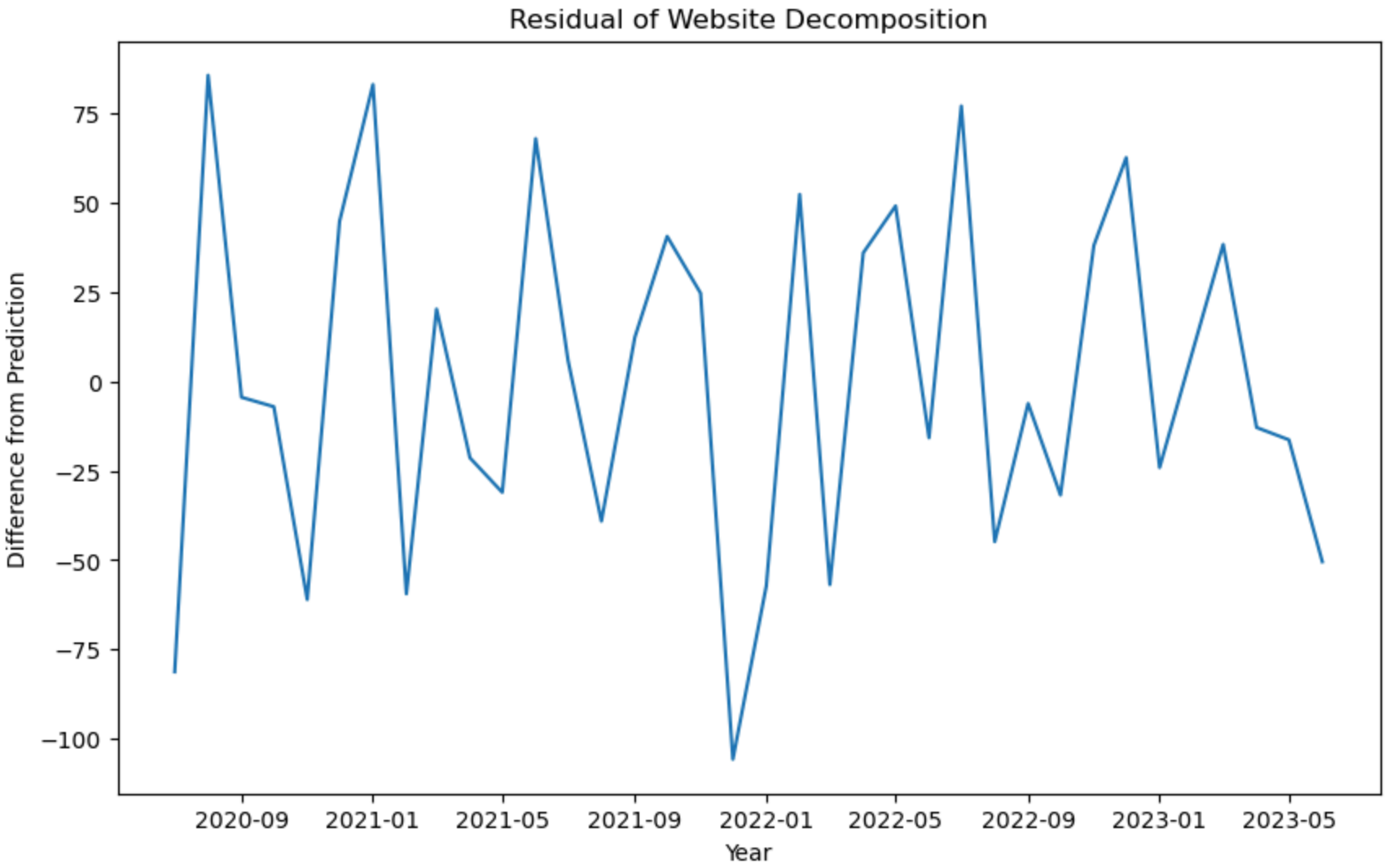
Out[ ]:
```
[<matplotlib.lines.Line2D at 0x7ff2d8b16bb0>]
```



You can see that during February of a calendar year the value is going to drop down before returning to inconsistient spikes and dips around a mean of 0.

Residual of Website Decomposition

In [ ]:
```
#Plot the residual of websites
time_series(website_seasonal_decomposition[3].index,website_seasonal_decomposition[3],'Residual of Website Decomposition'
```



Comparison of the axes of the seasonality graph and of the residual shows how the seasonality value is not that accurate. The differences almost match the residual difference which suggests that there is no real seasonality going on. This would be consistient with the fact that the model is stationary.

1.3 ARIMA Model Building:

1. Determine the order of differencing (d) needed to make the series stationary.
2. Identify the autoregressive term (p) and moving average term (q) using plots such as the Autocorrelation Function (ACF) and Partial Autocorrelation Function (PACF).
3. Construct and fit the ARIMA model to the historical data.

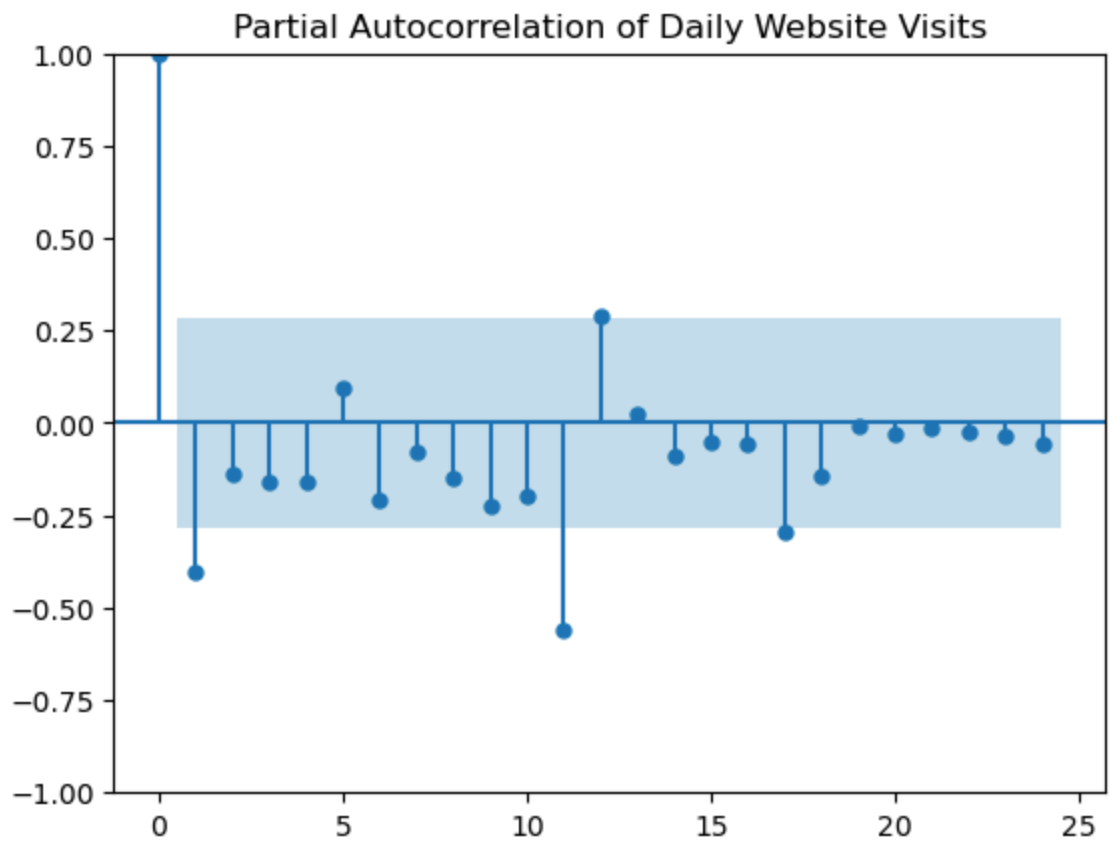
ARIMA Modeling

The following ARIMA modeling types are going to be explored

- AR: Autoregressive Model (p value - using PACF to find order)
- MA: Moving Average Model (q value - using ACF to find order)
- ARMA: Autoregressive Moving Average (p value, q value)
- ARIMA: Autoregressive Integrated Moving Average (p value, d value, q value)

p,d,q orders value

In [ ]:
```
#Use the pacf plot to find the p order
website_monthly_sum_pacf = plot_pacf(website_monthly_sum['Daily Visitors'],title='Partial Autocorrelation of Daily Websit
```



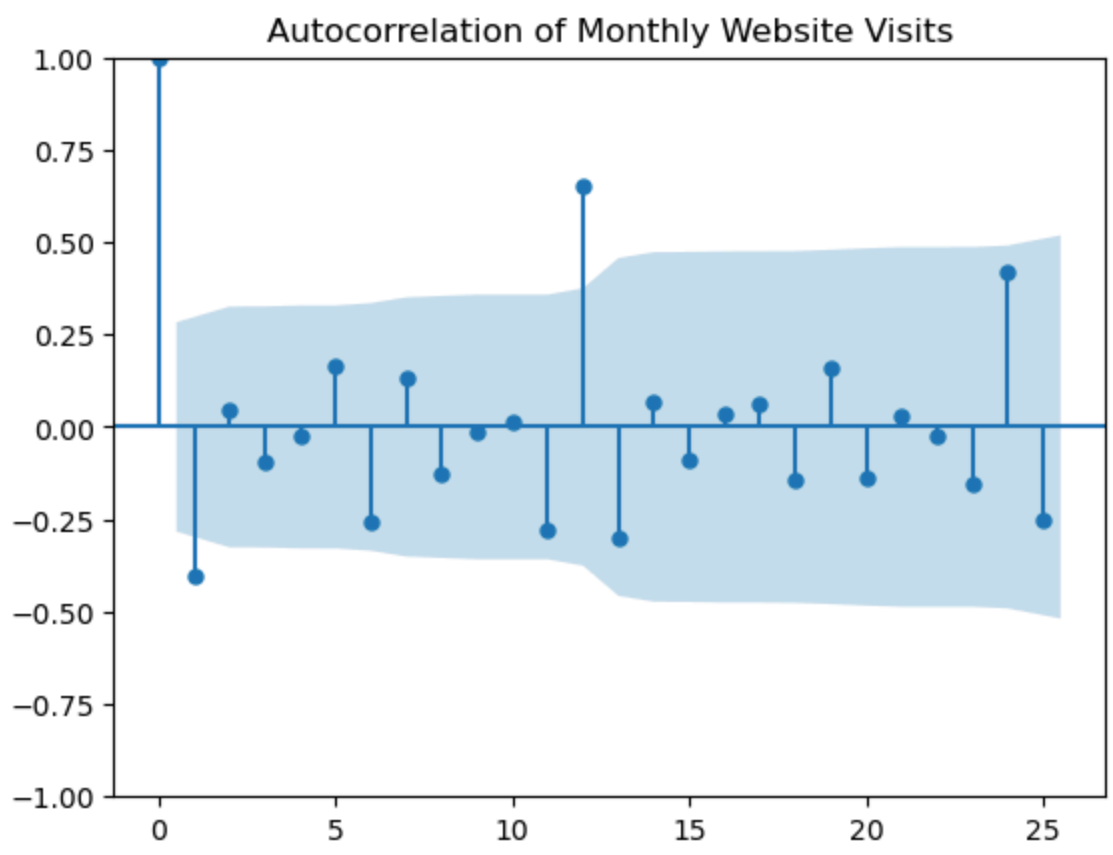
In [ ]:
```
#Create a list of p orders
p_orders_value = [1,2]
```

In [ ]:
```
#Look at the ADF statistic to determine the d value
adf_test(website_monthly_sum['Daily Visitors'])
```

```
ADF Statistic:  -6.230698158447822
P-Value:  4.9617716583331424e-08
Critical Value:
        1%: -3.62
        5%: -2.94
        10%: -2.61
```

In [ ]:
```
#Create a list of d orders
d_orders_value = [0,1]
```

In [ ]:
```
#Look at the ACF plot to find q
website_monthly_sum_acf = plot_acf(website_monthly_sum['Daily Visitors'],title='Autocorrelation of Monthly Website Visits
```



In [ ]:
```
#Create a list of q orders
q_orders_value = [1,5]
```

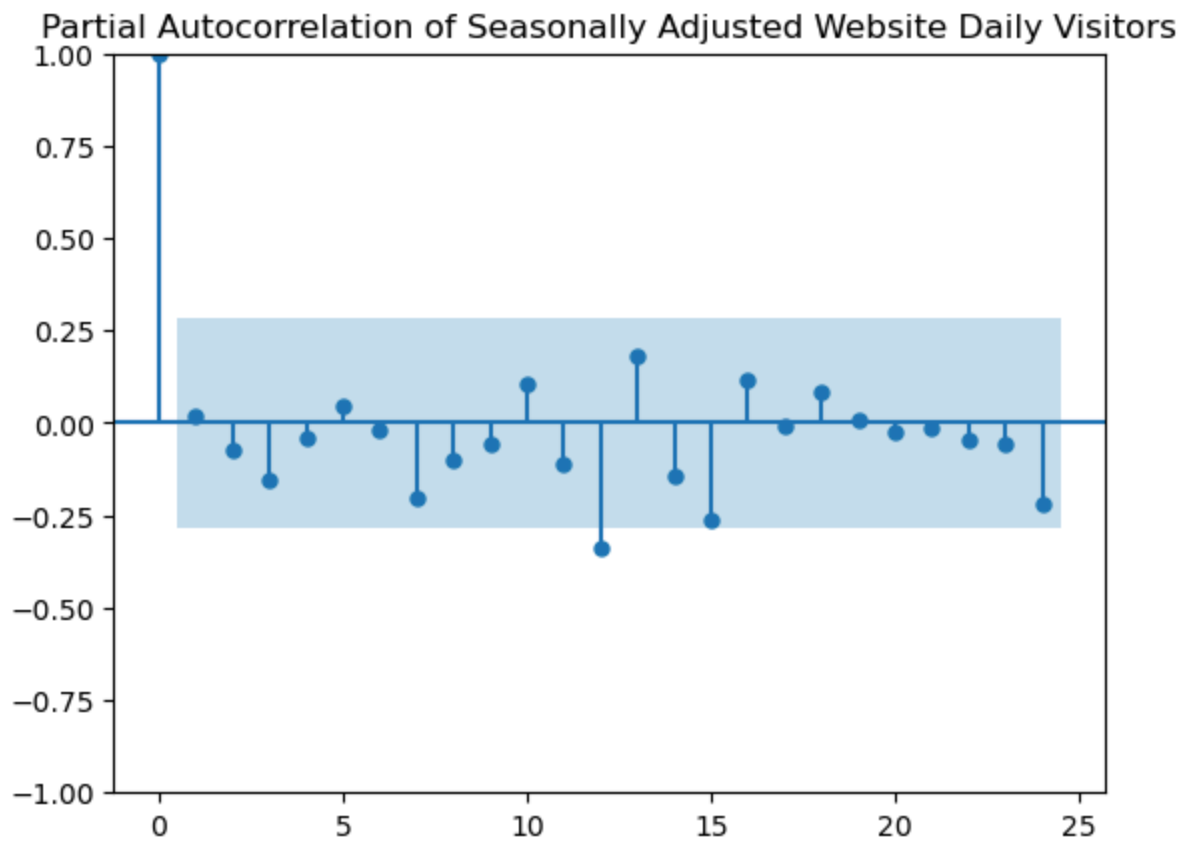
P,D,Q Values

Take the seasonality values out of the data so that pacf,acf and adf statistics can be done to figure out the P,D,Q values

In [ ]:
```
#Remove seasonal variations from data
website_seasonally_adjusted = website_seasonal_decomposition[0] - website_seasonal_decomposition[2]
website_seasonally_adjusted
```

Out[ ]:
```
Date
2020-01-01    6086.015046
2020-02-01    6295.737269
2020-03-01    6135.695602
2020-04-01    6115.084491
2020-05-01    5967.542824
                 ...
2023-08-01    6113.265046
2023-09-01    6124.542824
2023-10-01    6085.403935
2023-11-01    6097.153935
2023-12-01    6117.584491
Length: 48, dtype: float64
```

In [ ]:
```
#Use the PACF to find P values
website_seasonally_adjusted_pacf = plot_pacf(website_seasonally_adjusted,lags=24,title='Partial Autocorrelation of Seasona
```



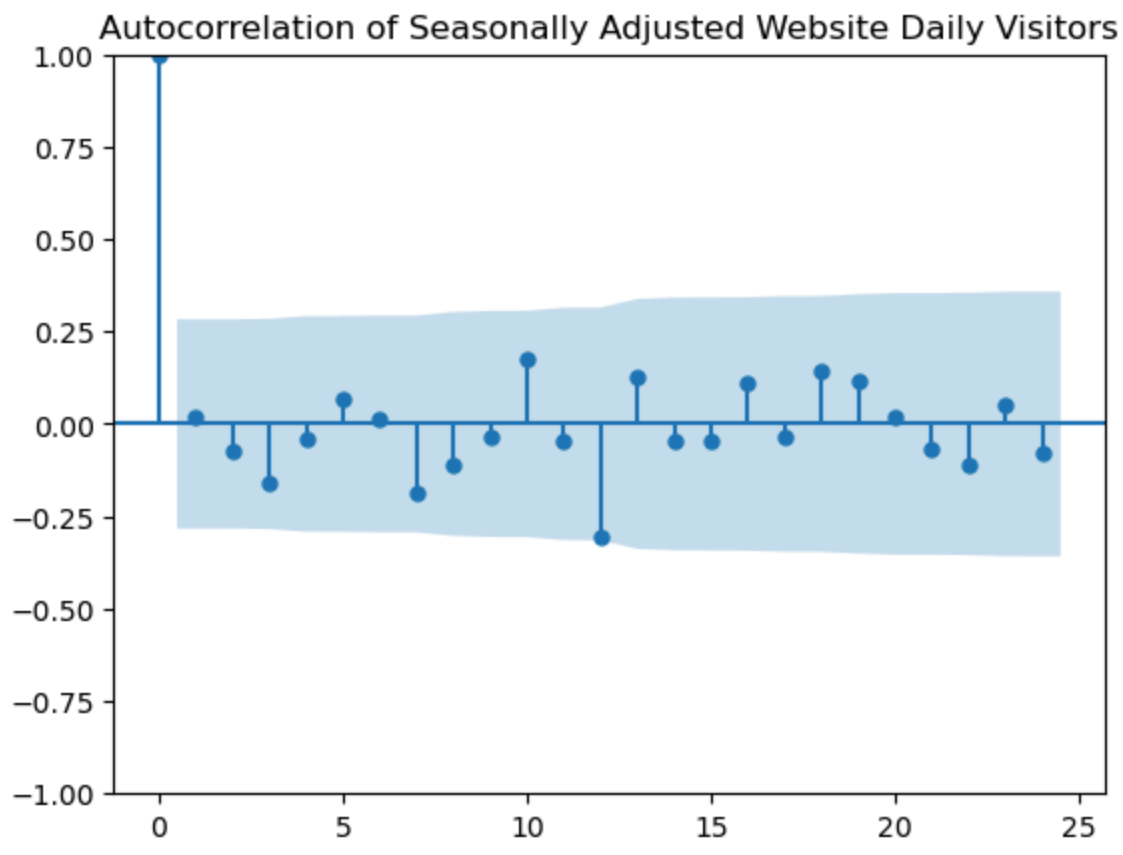
In [ ]:
```
#List of P orders
P_orders_value = [1,2,3]
```

In [ ]:
```
#Use the ADF statistic to solve for D list
adf_test(website_seasonally_adjusted)
```

```
ADF Statistic:  -3.0256856463231645
P-Value:  0.03256431803499405
Critical Value:
        1%: -3.61
        5%: -2.94
        10%: -2.61
```

In [ ]:
```
#D orders values
D_orders_value = [0,1]
```

In [ ]:
```
#Use the ACF to find Q values
website_seasonally_adjusted_acf = plot_acf(website_seasonally_adjusted,lags=24,title='Autocorrelation of Seasonally Adjus
```



In [ ]:
```
#Q orders value list
Q_orders_value = [1,2,3]
```

ARIMA Model Evaluation

Using the function above, there is going to be an ordered list of most effective (using AIC) order for different models.

In [ ]:
```
evaluate_time_series_models(website_monthly_sum['Daily Visitors'],p_orders_value,q_orders_value,d_orders_value)
```

Out[ ]:

	Model	Order	AIC	BIC
15	ARIMA	(2, 1, 5)	611.801164	626.602345
10	ARIMA	(1, 1, 1)	614.966322	620.516765
7	ARMA	(2, 0, 5)	615.061804	631.902613
13	ARIMA	(2, 0, 5)	615.061804	631.902613
4	ARMA	(1, 0, 1)	616.132897	623.617701
8	ARIMA	(1, 0, 1)	616.132897	623.617701
14	ARIMA	(2, 1, 1)	616.611260	624.011851
11	ARIMA	(1, 1, 5)	617.468058	630.419092
6	ARMA	(2, 0, 1)	617.561590	626.917596
12	ARIMA	(2, 0, 1)	617.561590	626.917596
2	MA	(0, 0, 1)	619.949810	625.563413
3	MA	(0, 0, 5)	621.026591	634.124998
0	AR	(1, 0, 0)	622.604891	628.218494
5	ARMA	(1, 0, 5)	622.679063	637.648671
9	ARIMA	(1, 0, 5)	622.679063	637.648671
1	AR	(2, 0, 0)	623.757966	631.242770

In [ ]:
```
select_best_sarima_model(website_monthly_sum['Daily Visitors'],p_orders_value,d_orders_value,q_orders_value,P_orders_valu
```

Out[ ]:
```
{'aic': 602.34679553842,
 'bic': 609.8315995820516,
 'pdq': (0, 0, 1),
 'seasonal_pdq': (0, 0, 1, 12)}
```

The values are 0 for both p and d as the data is stationary and has no autoregressive (moving average) and no need to difference data (d) as it is stationary.

1.4 Model Evaluation:

Use metrics such as Mean Absolute Error (MAE), Root Mean Squared Error (RMSE), and others to evaluate the model.

Discuss the model's limitations and any discrepancies observed between the predicted and actual values.

There are 3 different methods of modeling I am going to use.

1. ARIMA Model - Using the orders from the list above forecast the data
2. SARIMA Model - Using the orders from the list above forecast the data and include the seasonal impact
3. Holt Winter's Seasonal Method - Triple exponential smoothing is going to incorporate seasonality into the system

In [ ]:
```
#Get the index value to split the data 0.7 to 0.3 for train and testing respectively.
split_index = int(len(website_monthly_sum)*0.7)

#Apply the index to train and test data
train_data = website_monthly_sum.iloc[:split_index+1]
test_data = website_monthly_sum.iloc[split_index:]
```

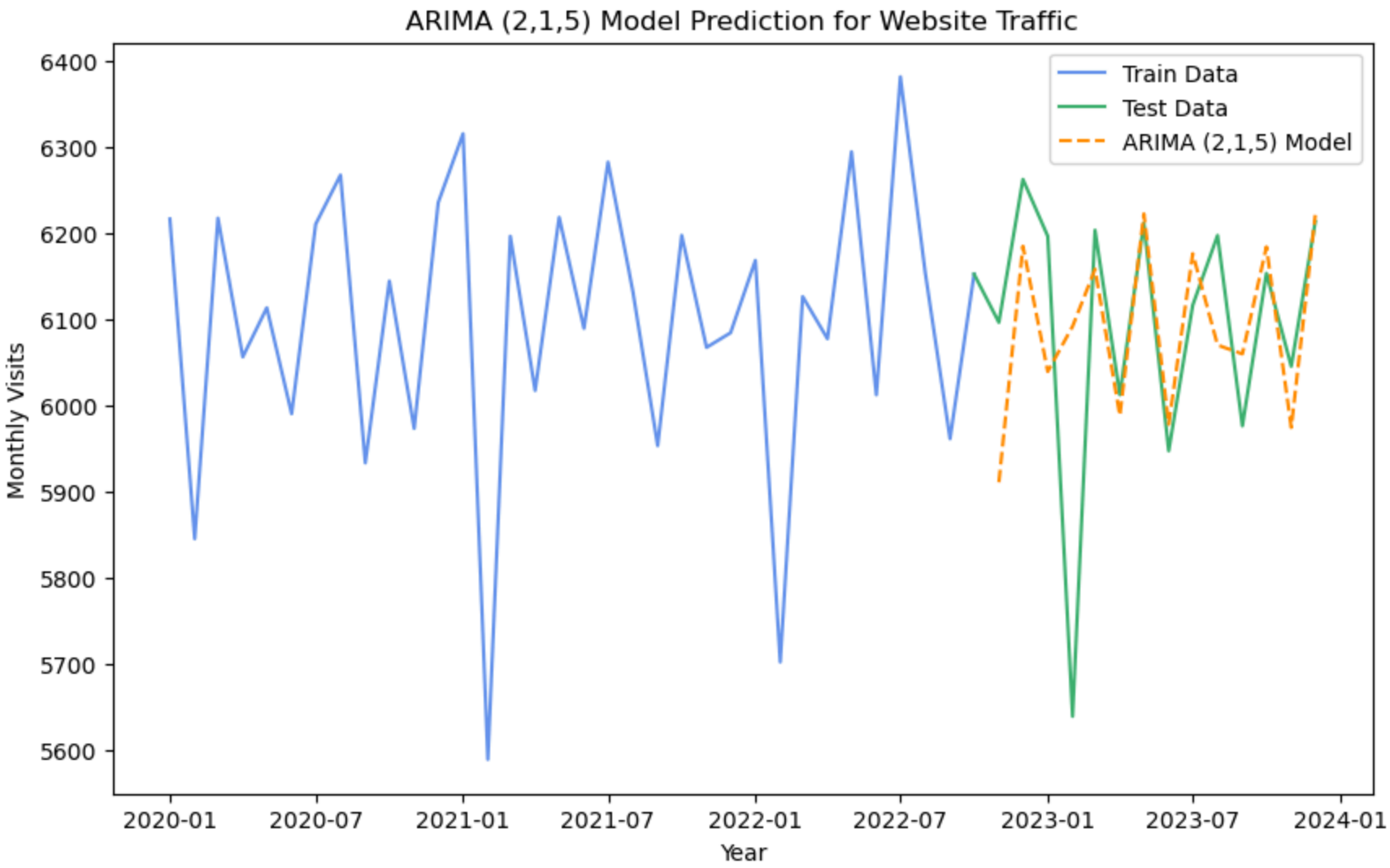
In [ ]:
```
#Add an incrementing month
test_data['Incrementing Month'] = range(len(train_data),len(train_data)+len(test_data))
```

```
/var/folders/6q/lw13gkln44z1r6ncfrmcbc5w0000gn/T/ipykernel_38134/4043535707.py:2: SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame.
Try using .loc[row_indexer,col_indexer] = value instead

See the caveats in the documentation: https://pandas.pydata.org/pandas-docs/stable/user_guide/indexing.html#returning-a-view-versus-a-copy
  test_data['Incrementing Month'] = range(len(train_data),len(train_data)+len(test_data))
```

ARIMA (2,1,5) Model

In [ ]:
```
# Fit ARIMA model to the training data
model_arima_1 = ARIMA(train_data['Daily Visitors'], order=(2,1,5))
model_arima_1 = model_arima_1.fit()

# Make predictions on the test data
test_predictions_arima_1 = model_arima_1.predict(start=test_data.index[1], end=test_data.index[-1], dynamic=False)
```

In [ ]:
```
#Plot the arima model next to the original data with the 80/20 data split
test_training_model(train_data['Daily Visitors'], test_data['Daily Visitors'], test_predictions_arima_1,
                    'Year', 'Monthly Visits', 'ARIMA (2,1,5) Model Prediction for Website Traffic','ARIMA (2,1,5) Model')
```



The ARIMA model does a good job of capturing how the data fluctuates around a mean however it does not capture the large drop that takes place in February.

In [ ]:
```
#Get the mean absolute error for arima model
arima_1_mae = mean_absolute_error(test_data['Daily Visitors'][0:-1],test_predictions_arima_1)

actual_mean = test_data['Daily Visitors'].mean()

arima_1_mae_percentage = (arima_1_mae / actual_mean) * 100

arima_1_mae_percentage
```

Out[ ]:
```
3.2919496259278516
```

In [ ]:
```
#Create a forecast model for the next 5 years for ARIMA (2,1,5)
#This will be modeled in the forecast section
forecast_arima_1 = model_arima_1.predict(start=test_data['Incrementing Month'][-1], end=test_data['Incrementing Month'][-
```

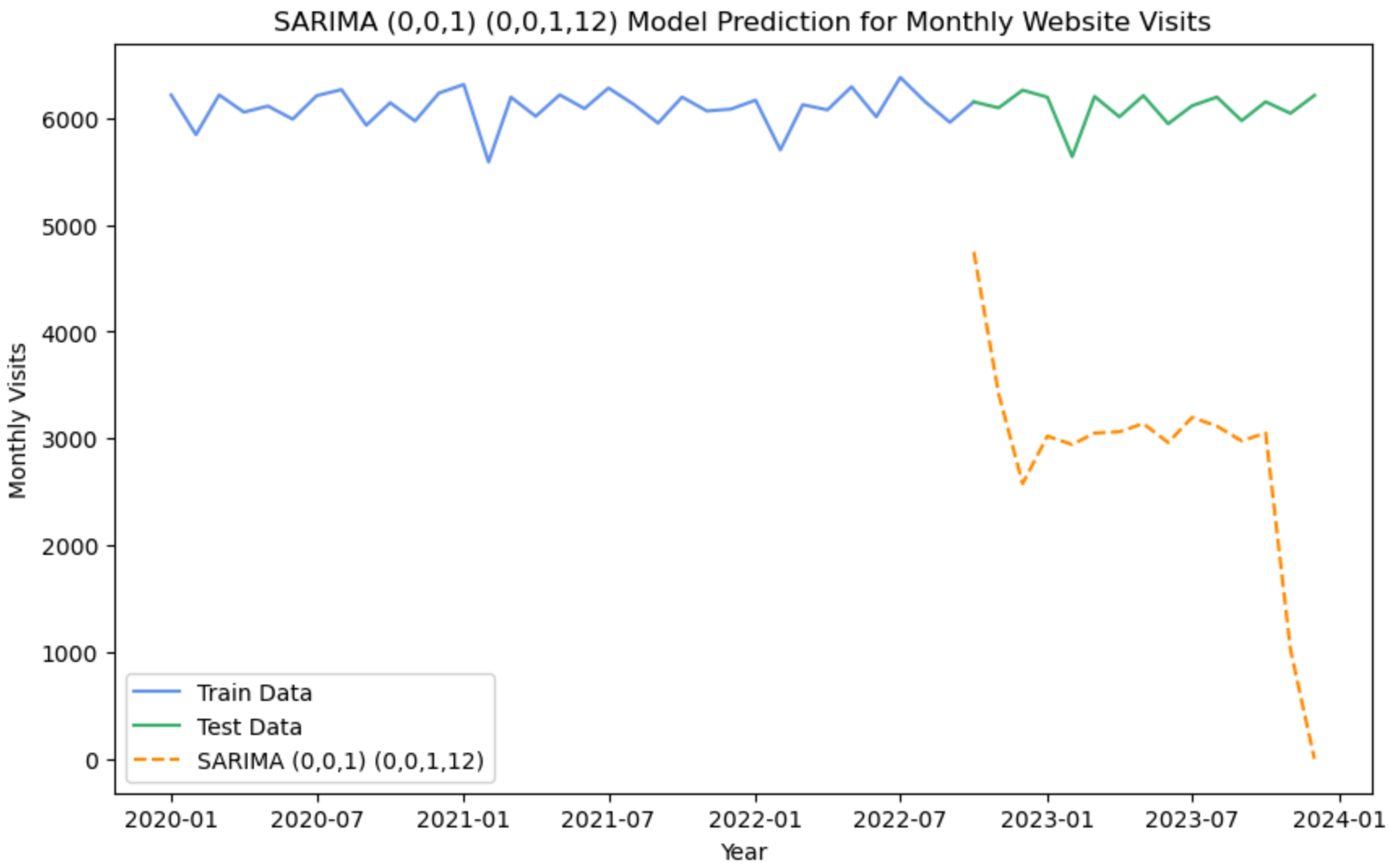
2. SARIMA Model (0, 0, 1), (0, 0, 1, 12)

In [ ]:
```
from statsmodels.tsa.statespace.sarimax import SARIMAX
#Fit SARIMA model to the training data
model_sarima_1 = SARIMAX(train_data['Daily Visitors'], order=(0,0,1), seasonal_order=(0, 0, 1, 12))
model_sarima_1 = model_sarima_1.fit()

# Make predictions on the test data
test_predictions_sarima_1 = model_sarima_1.predict(start=test_data.index[0], end=test_data.index[-1], dynamic=False)
```

```
RUNNING THE L-BFGS-B CODE

           * * *

Machine precision = 2.220D-16
 N =            3     M =           10
At X0         0 variables are exactly at the bounds

At iterate    0    f=  1.03734D+01    |proj g|=  1.65631D+00

At iterate    5    f=  9.77460D+00    |proj g|=  7.20205D-03

At iterate   10    f=  9.77341D+00    |proj g|=  1.31195D-04

At iterate   15    f=  9.77338D+00    |proj g|=  2.44862D-03

At iterate   20    f=  9.77004D+00    |proj g|=  1.66412D-02

At iterate   25    f=  9.20758D+00    |proj g|=  6.50319D-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
    3     29     38      1     0     0   6.212D-06   9.208D+00
  F =   9.2075176034862256

CONVERGENCE: NORM_OF_PROJECTED_GRADIENT_<=_PGTOL
 This problem is unconstrained.
```

In [ ]:
```
#Plot the arima model next to the original data with the 80/20 data split
test_training_model(train_data['Daily Visitors'], test_data['Daily Visitors'], test_predictions_sarima_1,
```

```
                    'Year', 'Monthly Visits', 'SARIMA (0,0,1) (0,0,1,12) Model Prediction for Monthly Website Visits','SA
```



In [ ]:
```
#Get the mean absolute error for sarima model
sarima_1_mae = mean_absolute_error(test_data['Daily Visitors'],test_predictions_sarima_1)

actual_mean = test_data['Daily Visitors'].mean()

sarima_1_mae_percentage = (sarima_1_mae / actual_mean) * 100

sarima_1_mae_percentage
```

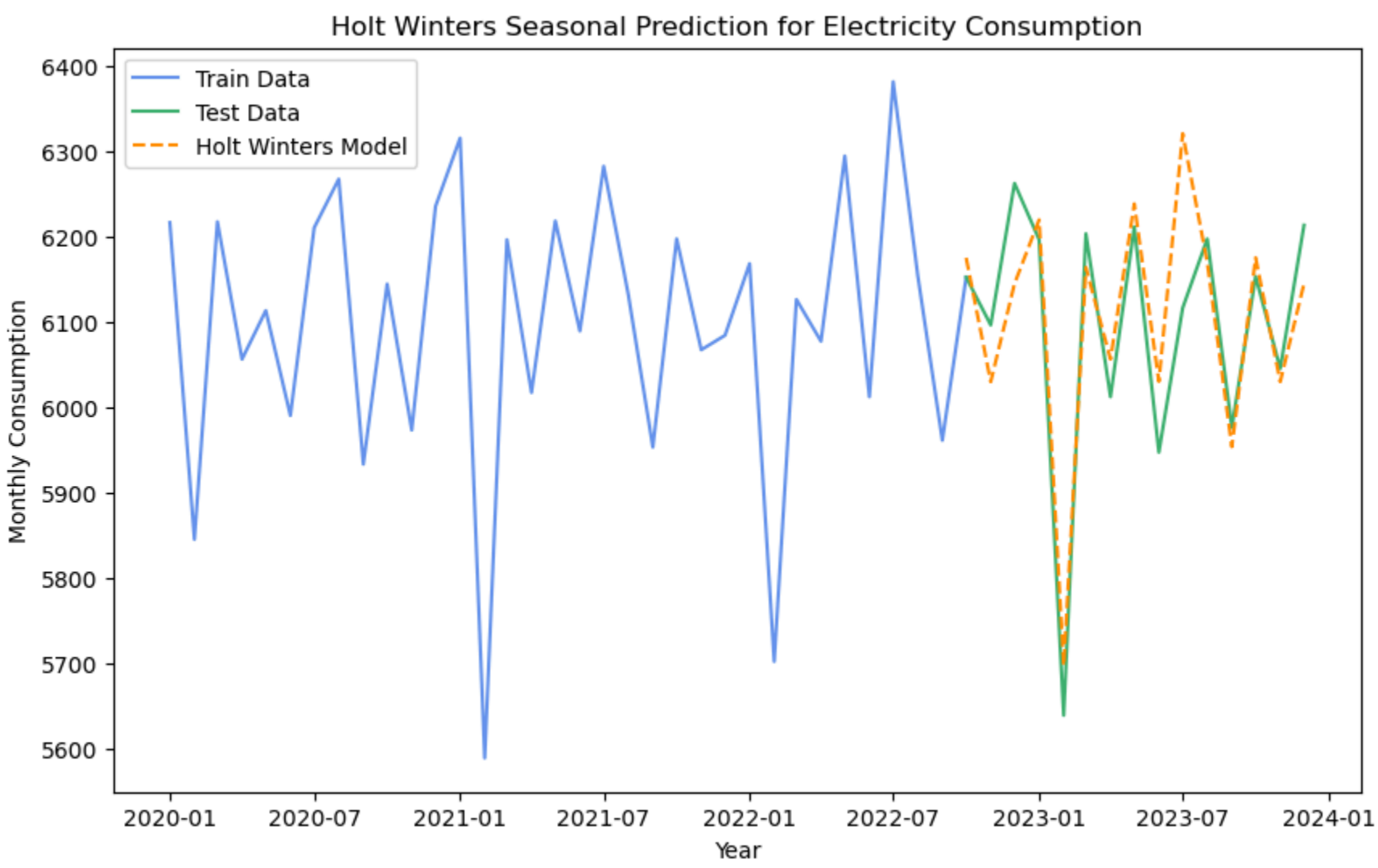
Out[ ]:
```
53.741490957028034
```

The SARIMA model does not appropriately find the mean value. This is likely because the autoregression is 0 but the moving average is 1 so it continues the downwards trend. As it is 50% wrong I will not be using the SARIMA model for future forecasting.

3. Holt Winter's Seasonal Method

This is commonly known as triple exponential smoothing and is used when the data exhibits seasonality. It incorporates SES, trend and seasonality which is why it is triple exponential smoothing.

In [ ]:
```
from statsmodels.tsa.holtwinters import ExponentialSmoothing
# Fit Holt-Winters method to the training data with multiplicative seasonality
holt_winters_1_model = ExponentialSmoothing(train_data['Daily Visitors'], seasonal_periods=12, trend='add', seasonal='add
holt_winters_1_model = holt_winters_1_model.fit(smoothing_level=0, smoothing_trend=1, smoothing_seasonal=0.5)

# Make predictions on the test data
holt_winters_test_predictions = holt_winters_1_model.predict(start=test_data.index[0], end=test_data.index[-1])

test_training_model(train_data['Daily Visitors'], test_data['Daily Visitors'], holt_winters_test_predictions,
                    'Year', 'Monthly Consumption', 'Holt Winters Seasonal Prediction for Electricity Consumption', 'Holt \
```



In [ ]:
```
#Get the mean absolute error for Holt Winter's Seasonal Method
holt_winters_mae = mean_absolute_error(test_data['Daily Visitors'],holt_winters_test_predictions)
```

```
actual_mean = test_data['Daily Visitors'].mean()

holt_winters_mae_percentage = (holt_winters_mae / actual_mean) * 100

holt_winters_mae_percentage
```

Out[ ]:
```
0.9205907328945953
```

In [ ]:
```
# Make predictions based on the test data
forecast_holt_winters = holt_winters_1_model.predict(start=test_data['Incrementing Month'][-1], end=test_data['Incrementi
```

This is a very accurate model and will be the one that is used in future forecasting.

1.5 Model Evaluation

1. Use metrics such as Mean Absolute Error (MAE), Root Mean Squared Error (RMSE), and others to evaluate the model.

2. Discuss the model's limitations and any discrepancies observed between the predicted and actual values.

In [ ]:
```
plt.figure(figsize=(20,6))

# Plot original data
plt.plot(website_monthly_sum['Daily Visitors'], color='steelblue', label='Original Data')

# Plot forecasted values from Holt Winters Seasonal Method
plt.plot(forecast_holt_winters.index, forecast_holt_winters, linestyle='-', color='orange', label='Holt Winters Forecast'

# Add labels and title
plt.xlabel('Date')
plt.ylabel('Number of Daily Visitors')
plt.title('Forecast of Website Traffic Using Holt Winters Seasonal Method')

# Add legend
plt.legend()

# Show plot
plt.show()
```