Final Project: Sunspots Monthly Average

File: Sunspots.csv

This dataset contains historical records of sunspots. Typical sunspot datasets include measurements such as the date of observation, sunspot numbers, and sometimes details about solar cycles. Such data is crucial for studying solar activity and its cycles over time.

Sunspots are classified as areas where the magnetic field is about 2500 times stronger than Earth's.

1.0 Preparing Jupyter Notebook

Package Downloads

In [ ]:
```
#Import necessary packages
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
from pmdarima import auto_arima
from statsmodels.tsa.holtwinters import SimpleExpSmoothing
from statsmodels.tsa.holtwinters import ExponentialSmoothing
```

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

Graphing Function

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)
```

Seasonality Functions

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 [ ]:
```
#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
```

Least Squares Functions

In [ ]:
```
#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 [ ]:
```
from sklearn.linear_model import LinearRegression

def get_least_squares_line(x_value, y_value):
    # 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)

    # Getting the coefficients of the least squares line
    slope = model.coef_[0]
    intercept = model.intercept_

    return slope, intercept
```

In [ ]:
```
def calculate_residuals(x_value, y_value, slope, intercept):
    # Extracting x and y values from the dataframe
    x = x_value.values.reshape(-1, 1).astype(float)
    y = y_value.astype(float)

    # Predicting y values using the provided slope and intercept
    y_pred = slope * x + intercept

    # Calculating residuals
    residuals = y - y_pred

    return residuals
```

ARIMA Modeling Functions

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

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 [ ]:
```
#Find the best SARIMA Model
def select_best_sarima_model(data, p_list, d_list, q_list, P_list, D_list, Q_list, s):
    # 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 [ ]:
```
from pmdarima import auto_arima

def sarima_model_definite_order(data, p, d, q, P, D, Q, s):

    # Fit SARIMA model using specified parameters
    model = auto_arima(data, start_p=p, d=d, start_q=q,
                       max_p=p, max_d=d, max_q=q,
                       start_P=P, D=D, start_Q=Q,
                       max_P=P, max_D=D, max_Q=Q,
                       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
```

Forecasting Functions

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:

Load the dataset and convert the date information into a datetime object to facilitate time series analysis. Check for missing values and anomalies, and handle them appropriately.

Import Data

In [ ]:
```
#Import the data using pandas
sunspots = pd.read_csv('/Users/ben_nicholson/Visual_Code_Projects/Applied Statistics & Modeling/Final Project/Data Files/S
sunspots
```

Out[ ]:

	Unnamed: 0	Date	Monthly Mean Total Sunspot Number
0	0	1749-01-31	96.7
1	1	1749-02-28	104.3
2	2	1749-03-31	116.7
3	3	1749-04-30	92.8
4	4	1749-05-31	141.7
...	...	...	...
3260	3260	2020-09-30	0.6
3261	3261	2020-10-31	14.4
3262	3262	2020-11-30	34.0
3263	3263	2020-12-31	21.8
3264	3264	2021-01-31	10.4

3265 rows × 3 columns

Preprocess dataframe

In [ ]:
```
#First drop the unecessary columns
sunspots_cleaned = sunspots.copy()
sunspots_cleaned.drop(columns='Unnamed: 0',inplace=True)
```

In [ ]:
```
#Rename column 'Monthly Mean Total Sunspot Number' to 'Monthly Mean'
sunspots_cleaned.rename(columns={'Monthly Mean Total Sunspot Number': 'Monthly Mean'}, inplace=True)
```

In [ ]:
```
#Convert date to datetime then make it so that values are the first of every month rather than the last
sunspots_cleaned['Date'] = pd.to_datetime(sunspots_cleaned['Date'])
sunspots_cleaned['Date']= sunspots_cleaned['Date'] - pd.offsets.MonthBegin(1)
```

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

Out[ ]:

	Monthly Mean
Date	
1749-01-01	96.7
1749-02-01	104.3
1749-03-01	116.7
1749-04-01	92.8
1749-05-01	141.7
...	...
2020-09-01	0.6
2020-10-01	14.4
2020-11-01	34.0
2020-12-01	21.8
2021-01-01	10.4

3265 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: ',sunspots_cleaned.index.dtype)
print('')
print('Monthly Mean dtype: ',sunspots_cleaned['Monthly Mean'].dtype)
```

```
Date dtype:  datetime64[ns]

Monthly Mean dtype:  float64
```

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
sunspots_cleaned['Monthly Mean'].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
sunspots_cleaned['zscore'] = (sunspots_cleaned['Monthly Mean'] - sunspots_cleaned['Monthly Mean'].mean()) / sunspots_clea

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

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

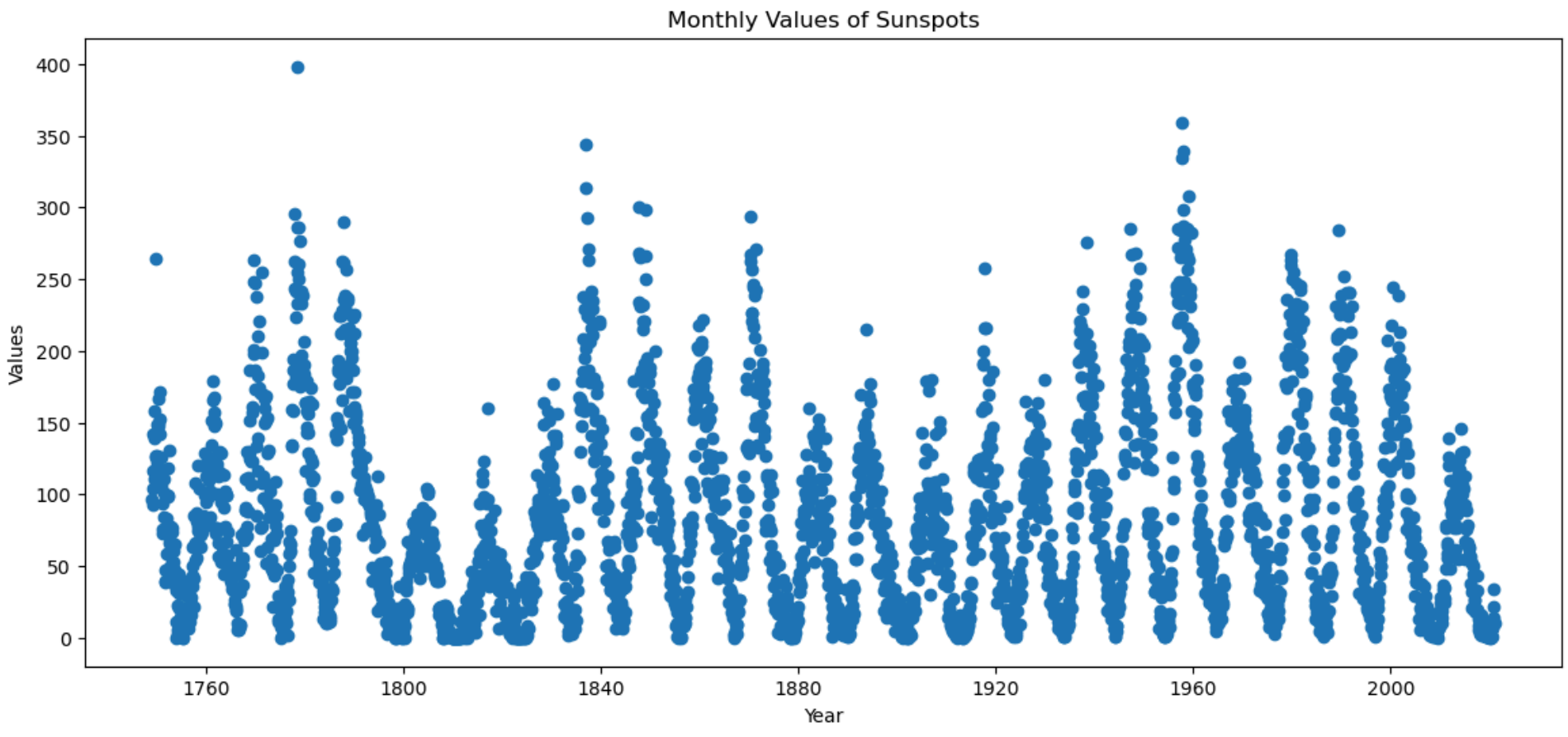
```
Anomalies detected:
            Monthly Mean    zscore
Date
1778-01-01         295.5  3.148085
1778-05-01         398.2  4.660842
1778-06-01         286.0  3.008151
1778-09-01         286.2  3.011097
1787-12-01         290.0  3.067071
1836-12-01         343.8  3.859538
1837-01-01         313.4  3.411750
1837-02-01         292.6  3.105369
1847-10-01         300.6  3.223207
1849-01-01         298.3  3.189329
1870-05-01         293.6  3.120098
1957-09-01         334.0  3.715185
1957-10-01         359.4  4.089324
1957-11-01         298.6  3.193748
1957-12-01         339.0  3.788834
1958-01-01         286.7  3.018462
1959-01-01         307.7  3.327790
```

Observe that the anomalies are concentrated around certain time periods

- 1778
- 1836-1837
- 1957-1959

These values can be taken out during certain periods of analysis to improve understanding of seasonality and trends. Make two different copies of the data, one with the anomalies and one without the anomalies.

In [ ]:
```
#Plot the data to see how much these values actually influence the data
scatter(sunspots_cleaned.index,sunspots_cleaned['Monthly Mean'],'Monthly Values of Sunspots','Year','Values')
```



From the scatterplot you can see that a majority of these anomalies work with the movement. The anomalies that do exist will only need to be removed when looking at the trend. So I will create a new dataframe that removes these rows. However overall there is no need to remove any of the anomalies.

In [ ]:
```
#Drop the anomalies indexes from the df from above
#Label it as trend as this is the only time that this data will need to be removed
sunspots_cleaned_no_anomalies = sunspots_cleaned.drop(anomalies.index)
```

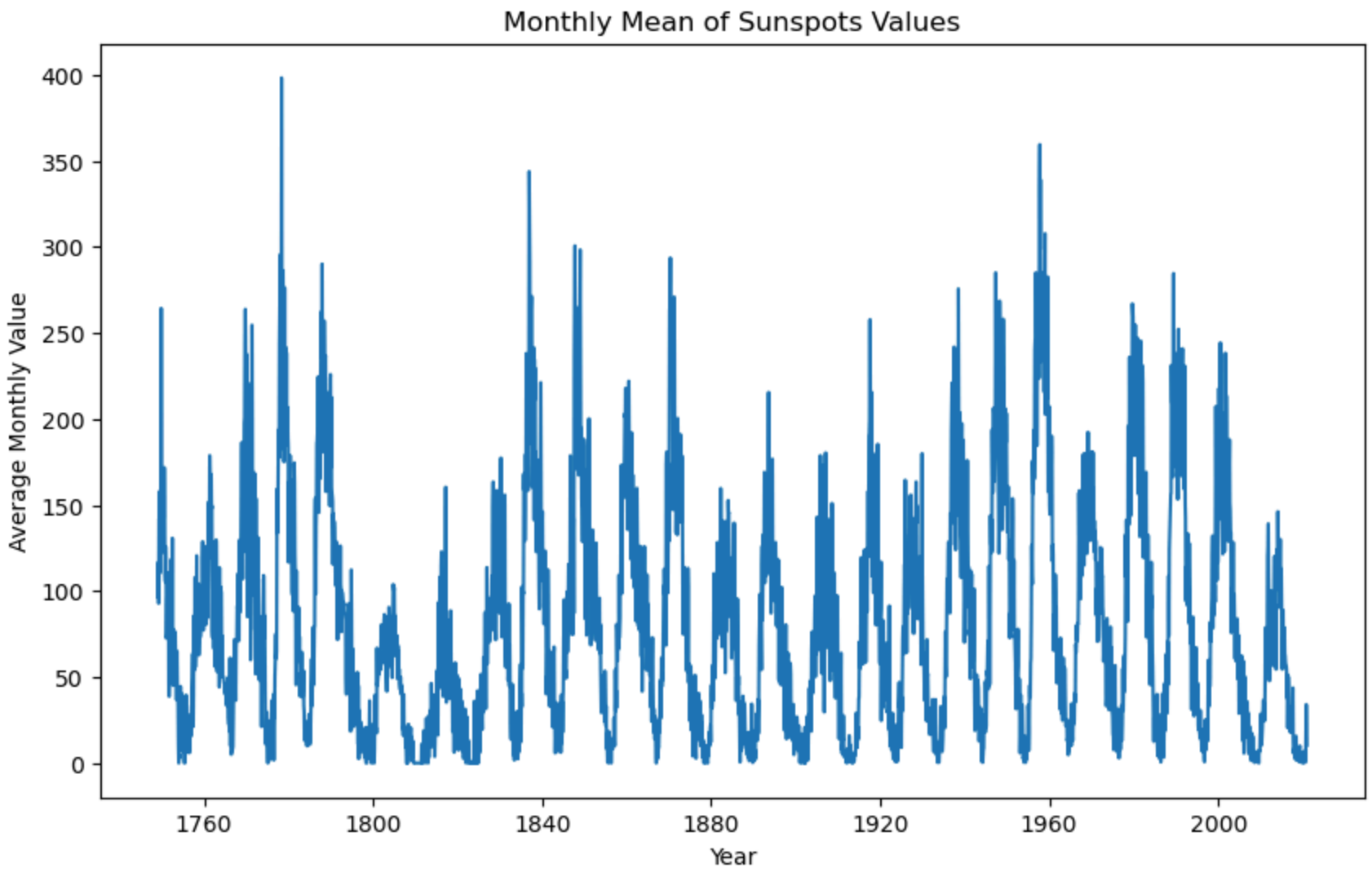
1.2 Exploratory Data Analysis (EDA):

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

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

Visualise Sunspots Data

In [ ]:
```
#Create a scatterplot of the data
time_series(sunspots_cleaned.index,sunspots_cleaned['Monthly Mean'],'Monthly Mean of Sunspots Values','Year','Average Mon
```



You can observe that there are some clear cycles that take place across certain time periods. These values will also drop back down to zero during certain times. There does not seem to be a massive trend movement, this will be explored more in the seasonal decomposition

Exploration of Seasonal Decomposition of Sunspots Data

Use the data that does not include anomalies as it is easier to understand trends this way.

Seasonality Period Decision

There is no given seasonality so there are 2 methods that I am going to use to determine the seasonal period

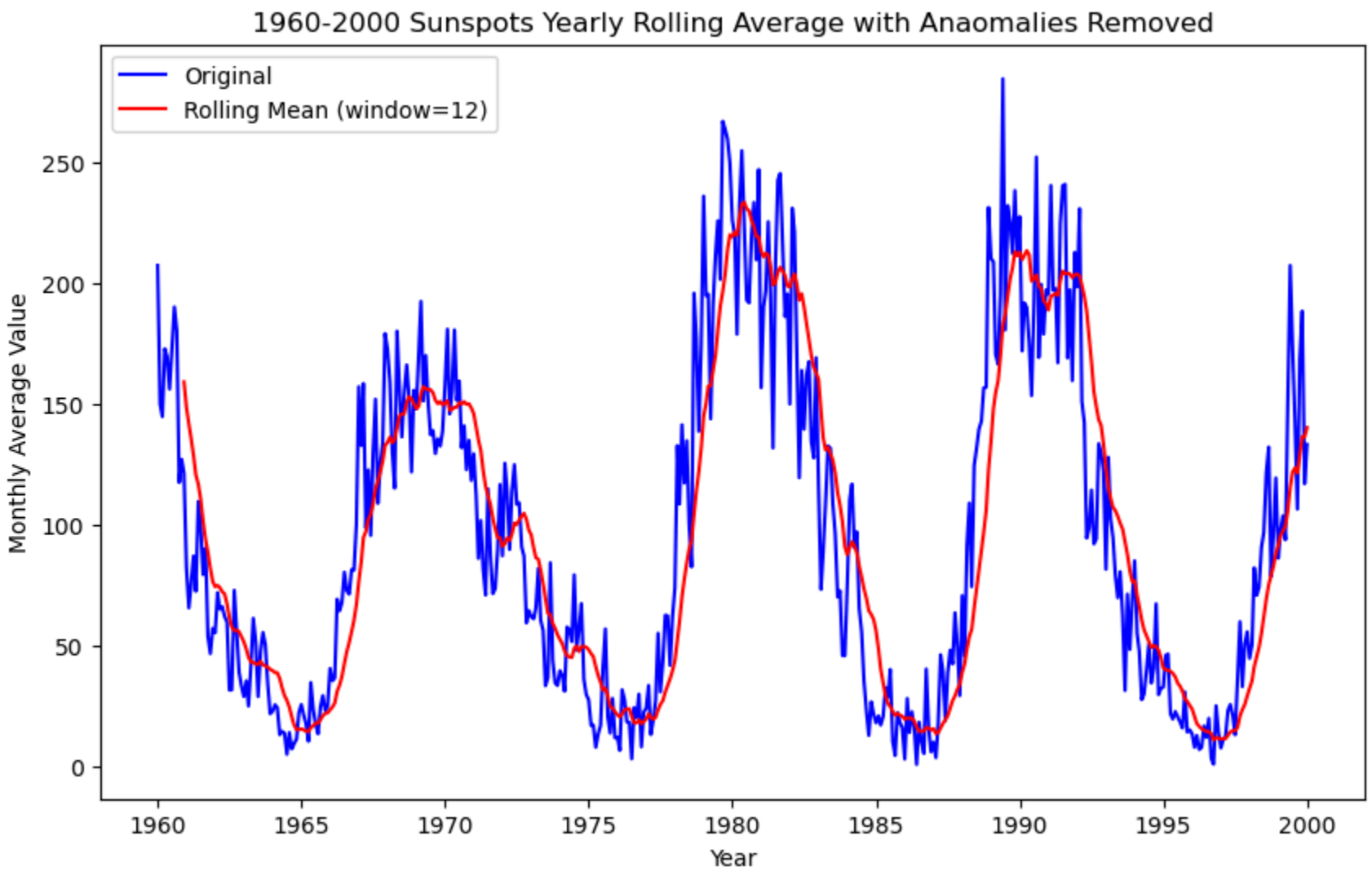
1. Visualisation
2. ACF & PACF Plots

1. Visualisation Review of Seasonality Period Decision

To understand the seasonal period take the rolling average of specific time periods. This reduces the noise and makes the changes easier to see.

In [ ]:
```
#Create a graph of the sunspots value from 1960 to 2000 and observe how the data moves
sunspots_cleaned_no_anomalies_time_period_1 = pd.DataFrame(sunspots_cleaned_no_anomalies['Monthly Mean'].loc['1960-01-01'
roll_mean(sunspots_cleaned_no_anomalies_time_period_1['Monthly Mean'],12,'1960-2000 Sunspots Yearly Rolling Average with A
```

```
<Figure size 1000x600 with 0 Axes>
```



It seems to be that every 10 years there is a fluctuation where the maximum values happen around the turn of the decade. I am going to use 120 as the period as this is 10 years of 12 values per year (monthly recordings)

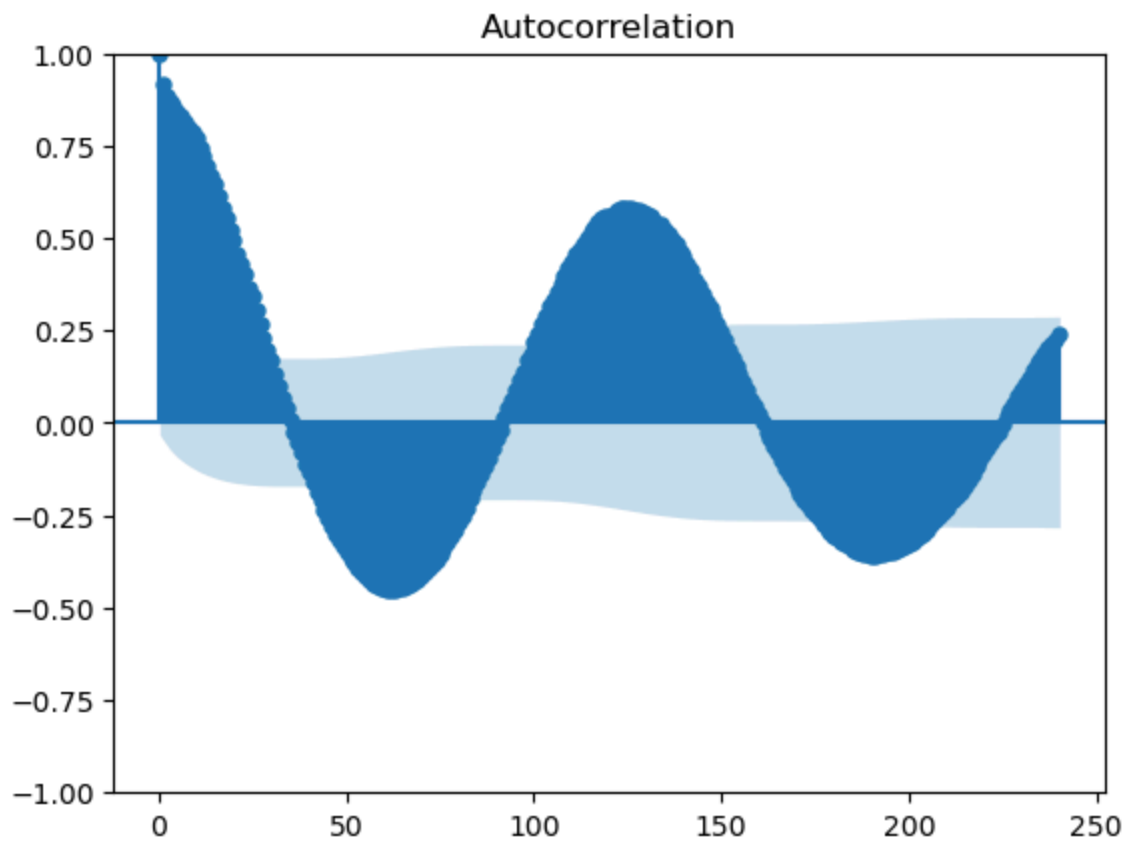
2. ACF & PACF Plots

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.

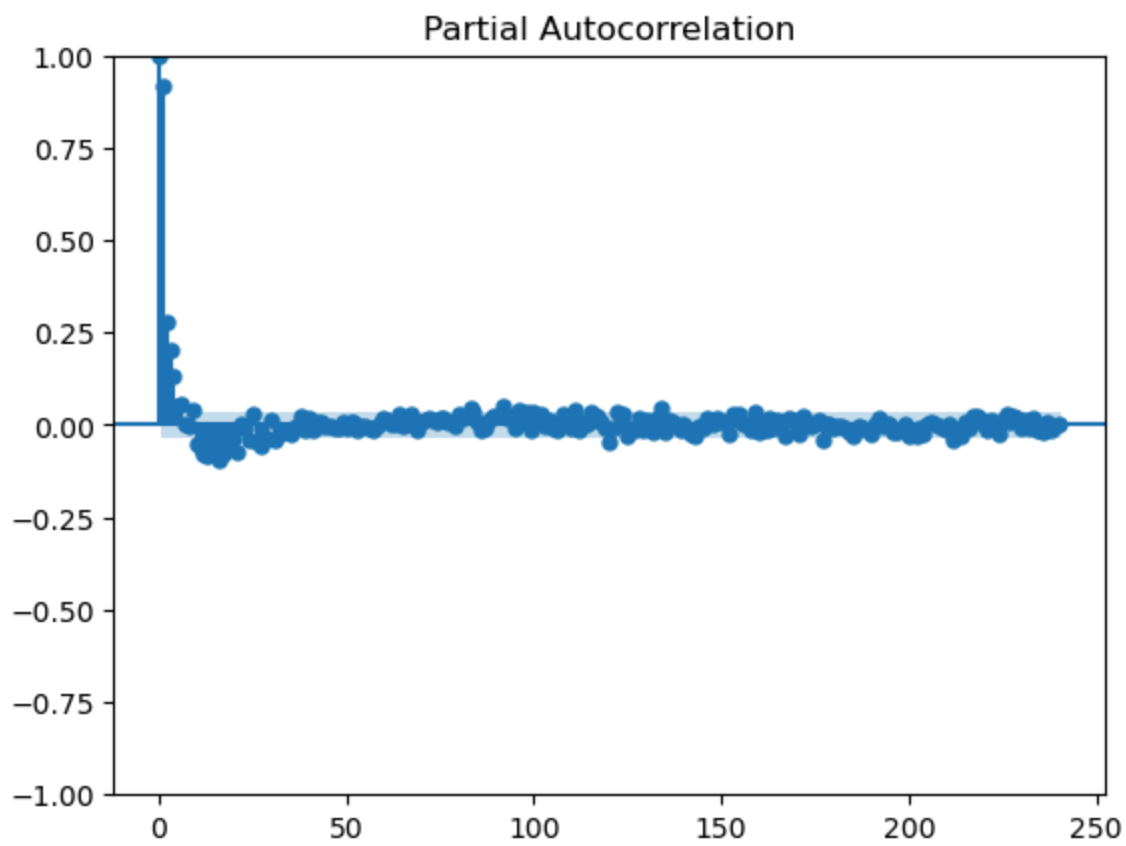
2. PACF plot - Is looking at the autoregressive nature of the graph, this is how values from lag k impact the current value

In [ ]:
```
#Look at the acf plot to determine the moving average (trend)
sunspots_cleaned_no_anomalies_acf = plot_acf(sunspots_cleaned_no_anomalies['Monthly Mean'],lags=240)
```



This only shows that there is a decreaing autocorrelation as time continues but does not show any further detail.

In [ ]:
```
#Look at the PACF plot to determine the autoregressive nature (seasonality)
sunspots_cleaned_no_anomalies_pacf = plot_pacf(sunspots_cleaned_no_anomalies['Monthly Mean'],lags=240)
```



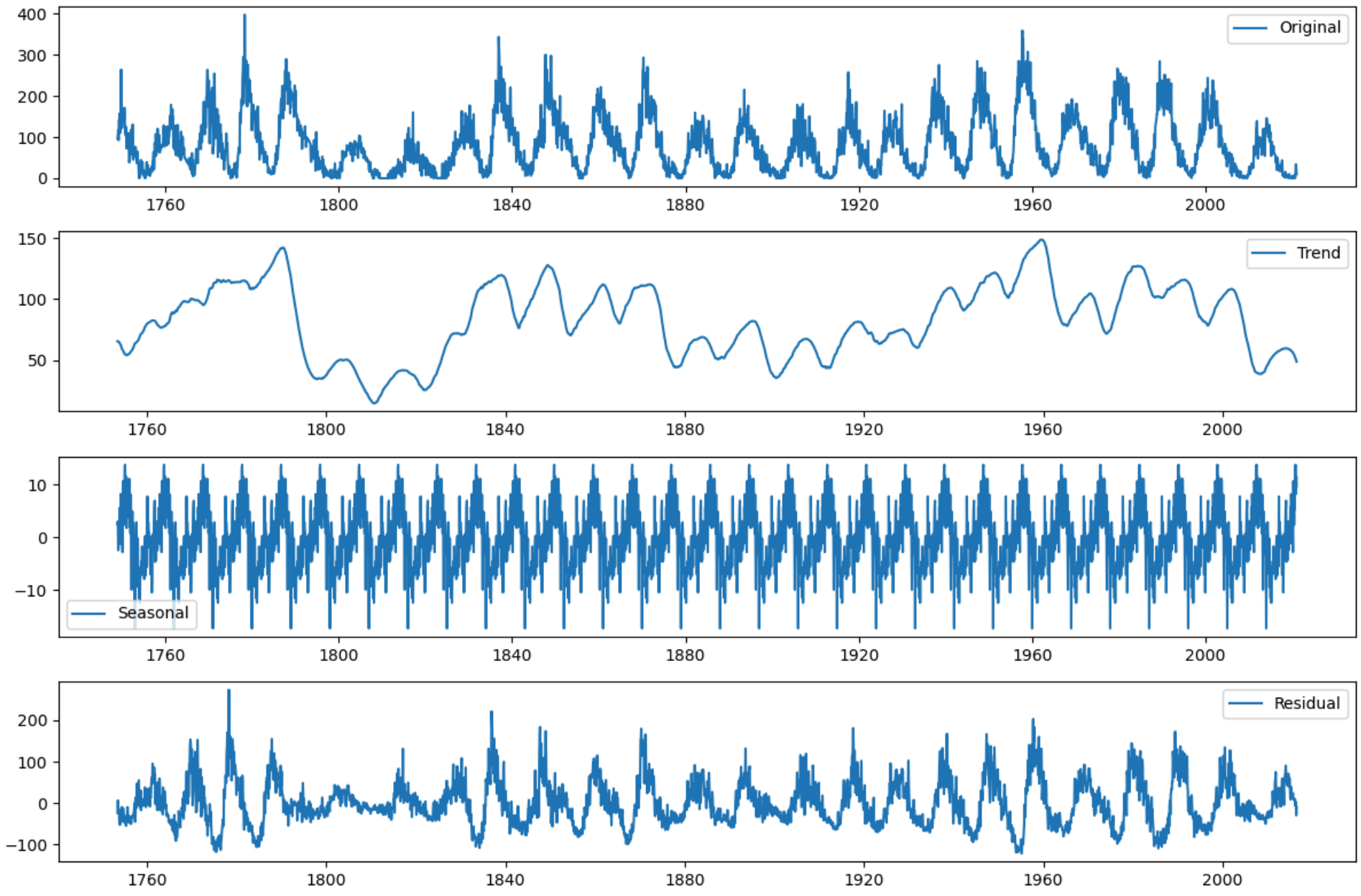
The PACF values do not show any lag values which should be looked at specifically. The ACF and PACF values do not show as useful information as the visualisation of data from 1960 to 2000. I am going to look at period values between 9 and 11 years.

Seasonal Decomposition of Different Periods

Looking at the different seasonal decompositions ranging from years 9 to 12 is going to help determine the most accurate model. This is an important step as it sets up the analysis for the rest of the project.

Seasonal Decomposition - Period of 9 years

In [ ]:
```
#plot the seasonal decomposition where the period is 9 years long
plot_seasonal_decomposition(sunspots_cleaned['Monthly Mean'],(12*9))
```



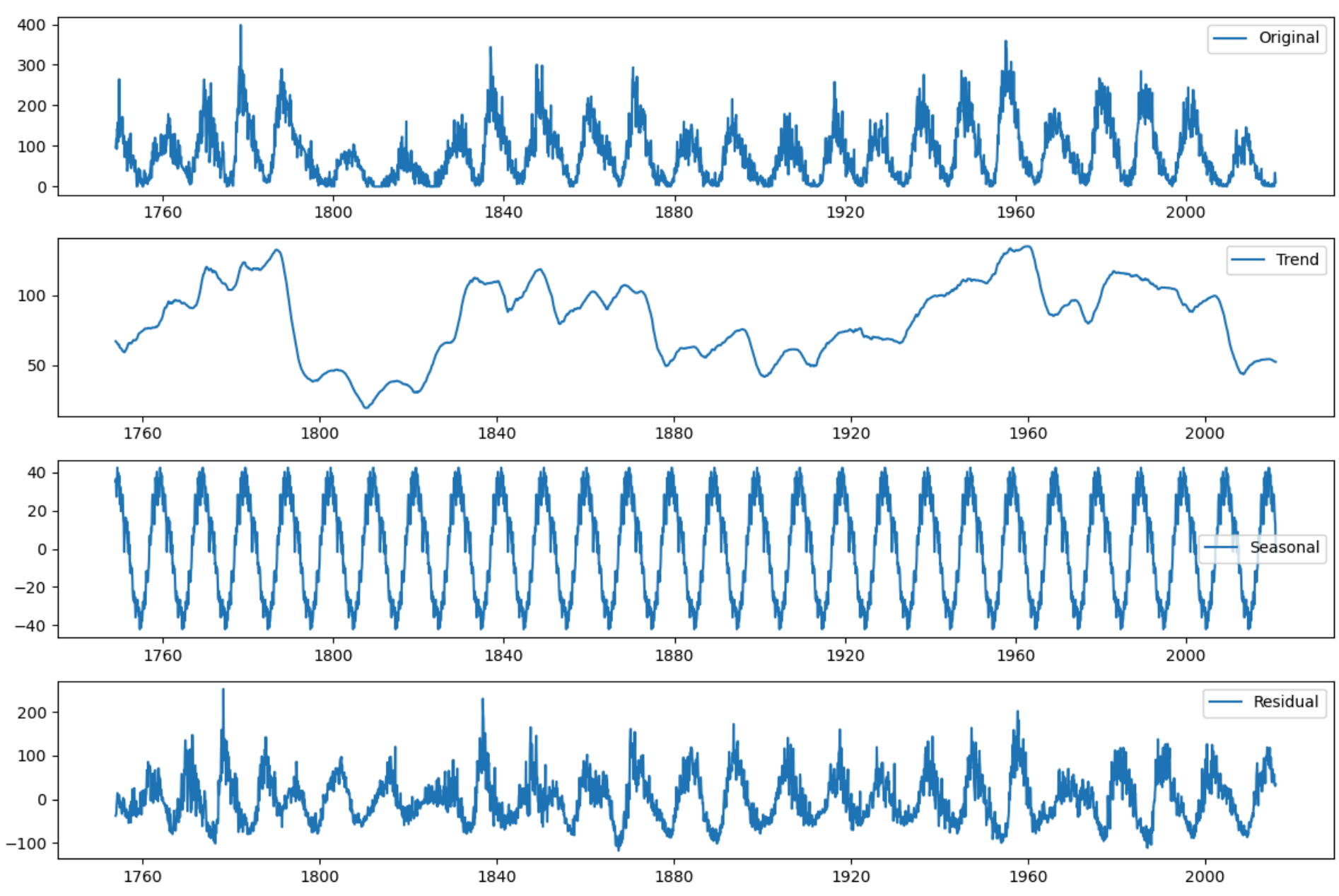
In [ ]:
```
#Find the mean absolute error of nine year period
nine_year_sunspot_decomposition = decompose_seasonality(sunspots_cleaned['Monthly Mean'],(12*9))
nine_year_sunspot_decomposition_residual = nine_year_sunspot_decomposition[3]
nine_year_sunspot_decomposition_residual.dropna(inplace=True)
zeros = np.zeros(len(nine_year_sunspot_decomposition_residual))
mean_absolute_error(zeros, nine_year_sunspot_decomposition_residual)
```

Out[ ]:
```
43.1113434310481
```

The seasonal values do seem to jump around, the seasonal values are also very small which does not suggest that it is going to be the best decision to choose 9 years as the seasonal period. The mae is quite small though which may mislead you to think that it is the best model.

Seasonal Decomposition - Period of 10 years

In [ ]:
```
#plot the seasonal decomposition where the period is 10 years long
plot_seasonal_decomposition(sunspots_cleaned['Monthly Mean'],(12*10))
```



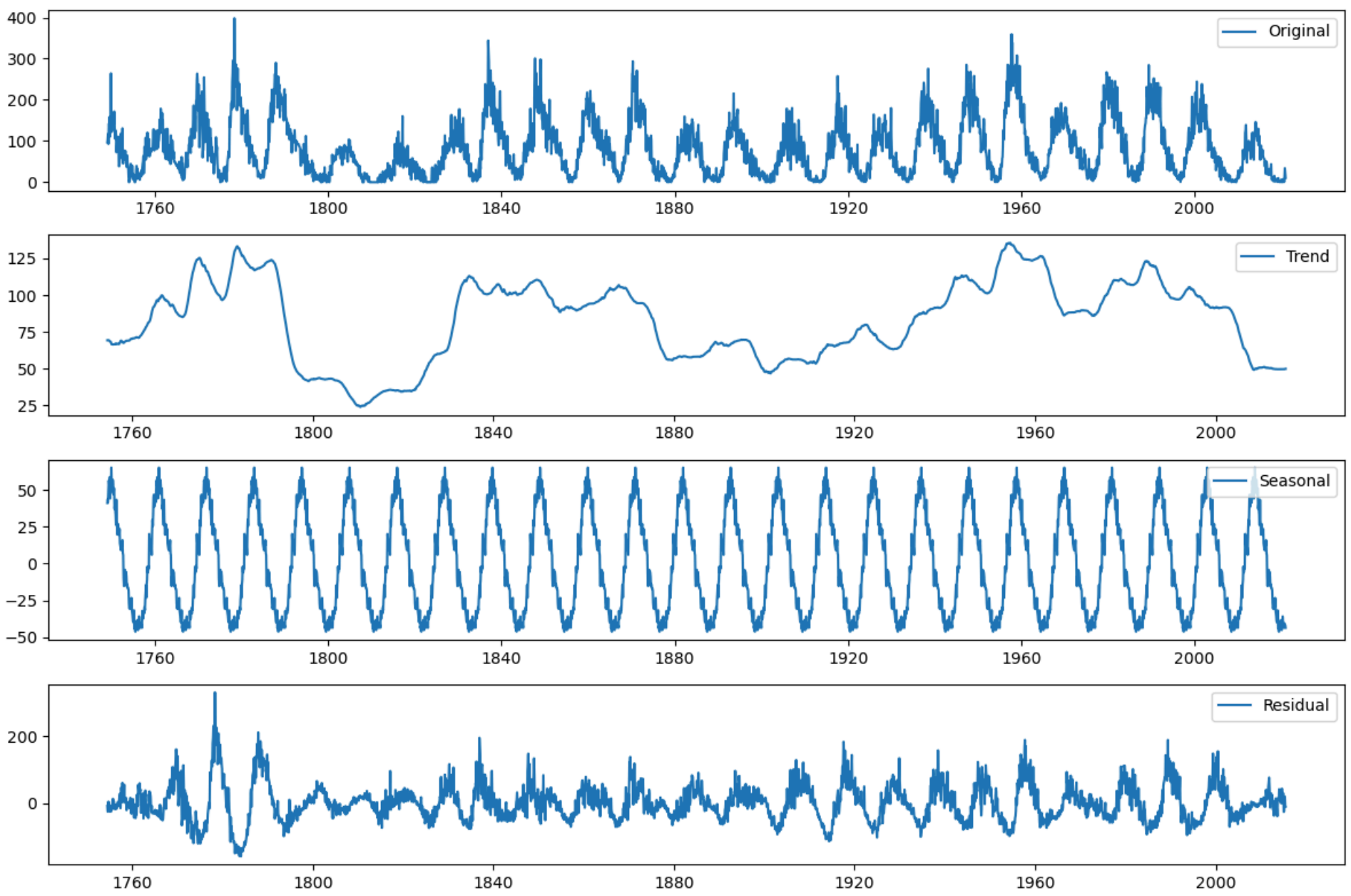
In [ ]:
```
#Find the mean absolute error of ten year period
ten_year_sunspot_decomposition = decompose_seasonality(sunspots_cleaned['Monthly Mean'],(12*10))
ten_year_sunspot_decomposition_residual = ten_year_sunspot_decomposition[3]
ten_year_sunspot_decomposition_residual.dropna(inplace=True)
zeros_2 = np.zeros(len(ten_year_sunspot_decomposition_residual))
mean_absolute_error(zeros_2, ten_year_sunspot_decomposition_residual)
```

Out[ ]:
```
44.05606900747373
```

Using a seasonal period of 10 years shows a more clear seasonal trend with a larger movement from 0. This does have a larger mae but might be the correct seasonal period.

Seasonal Decomposition - Period of 11 years

In [ ]:
```
#plot the seasonal decomposition where the period is 11 years long
plot_seasonal_decomposition(sunspots_cleaned['Monthly Mean'],(132))
```



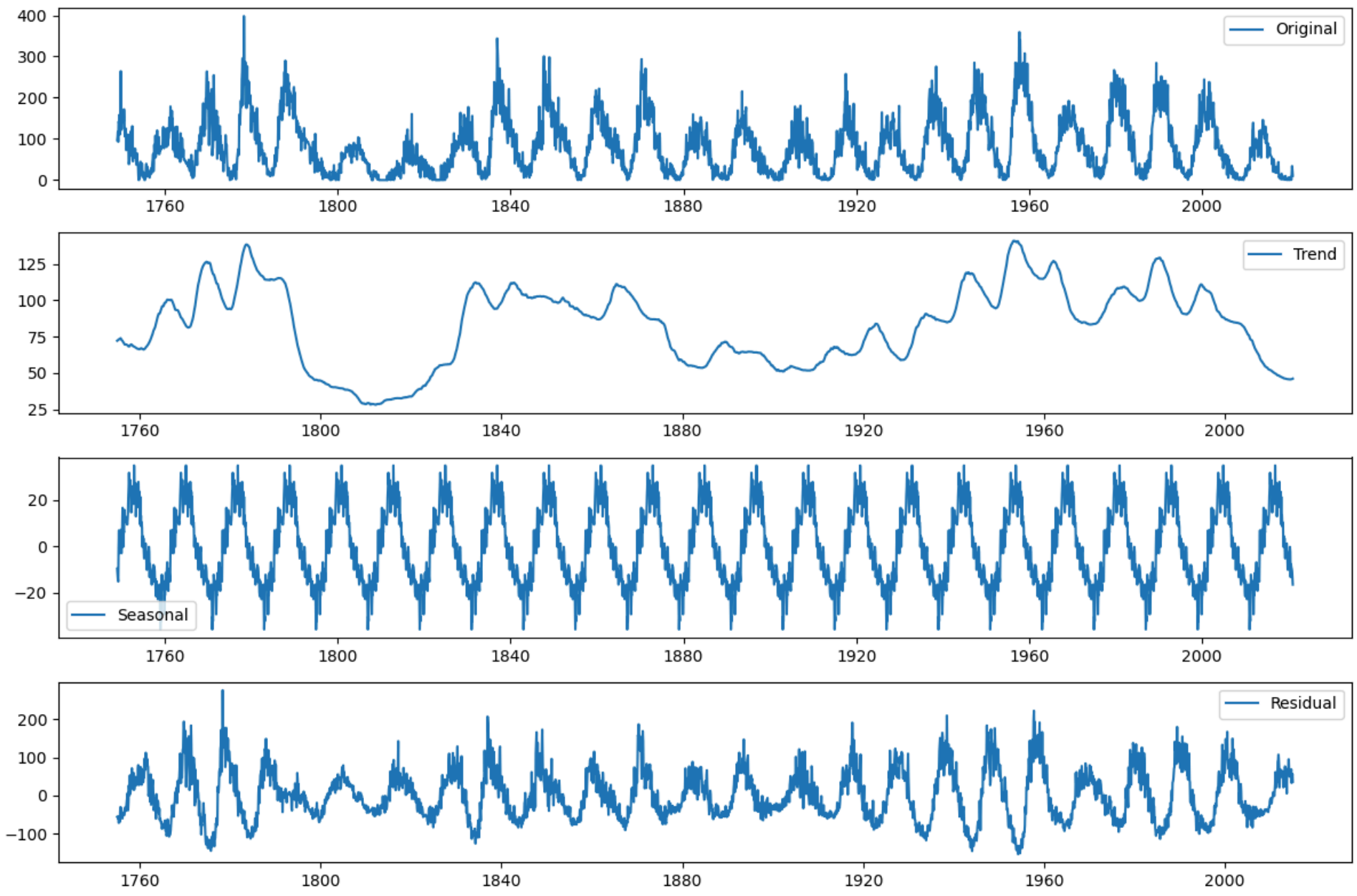
In [ ]:
```
#Find the mean absolute error of eleven year period
eleven_year_sunspot_decomposition = decompose_seasonality(sunspots_cleaned['Monthly Mean'],(12*11))
eleven_year_sunspot_decomposition_residual = eleven_year_sunspot_decomposition[3]
eleven_year_sunspot_decomposition_residual.dropna(inplace=True)
zeros_3 = np.zeros(len(eleven_year_sunspot_decomposition_residual))
mean_absolute_error(zeros_3, eleven_year_sunspot_decomposition_residual)
```

Out[ ]:
```
41.34308950683846
```

The MAE value is low, there is a clear seasonal movement and the residuals patterns are minimised, this is likely going to be the seasonal period

Seasonal Decomposition - Period of 12 Years

In [ ]:
```
#plot the seasonal decomposition where the period is 12 years long
plot_seasonal_decomposition(sunspots_cleaned['Monthly Mean'],(144))
```



In [ ]:
```
#Find the mean absolute error of twelve year period
twelve_year_sunspot_decomposition = decompose_seasonality(sunspots_cleaned['Monthly Mean'],(12*12))
twelve_year_sunspot_decomposition_residual = twelve_year_sunspot_decomposition[3]
twelve_year_sunspot_decomposition_residual.dropna(inplace=True)
zeros_4 = np.zeros(len(twelve_year_sunspot_decomposition_residual))
mean_absolute_error(zeros_4, twelve_year_sunspot_decomposition_residual)
```

Out[ ]:
```
53.81079465932563
```

The residual shows a clear pattern with less clear patterns in the seasonal period so this is not going to be selected.

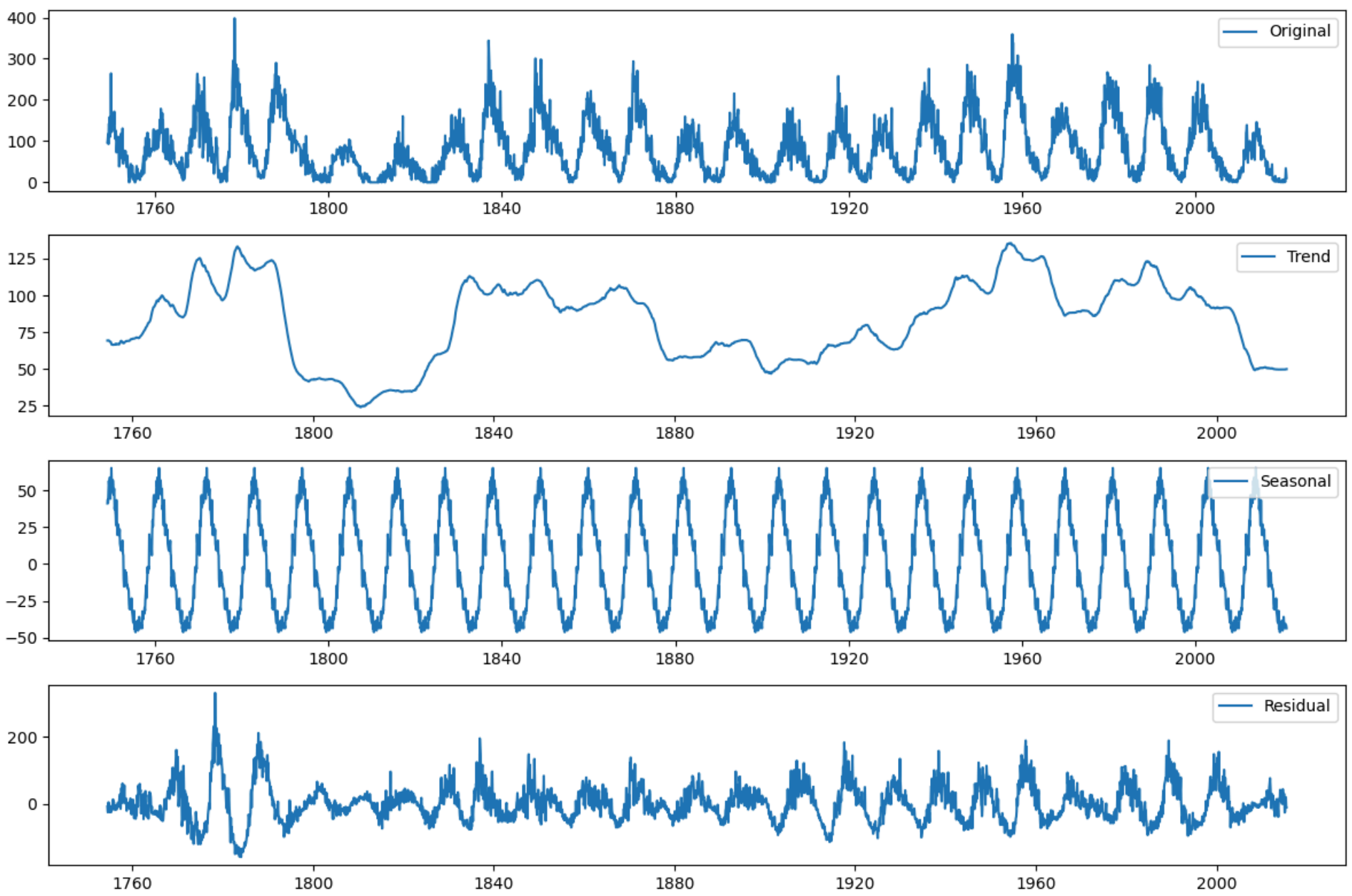
Seasonal Period Selection

The best seasonality is going to be 11 years (132 data values). This has the lowest residual as well as the smallest residual pattern. The seasonal decomposition can now take place for year 11.

Seasonal Decomposition of Sunspots

Using a period of 11 years, break down the data into its individual components to understand how the data moves over time

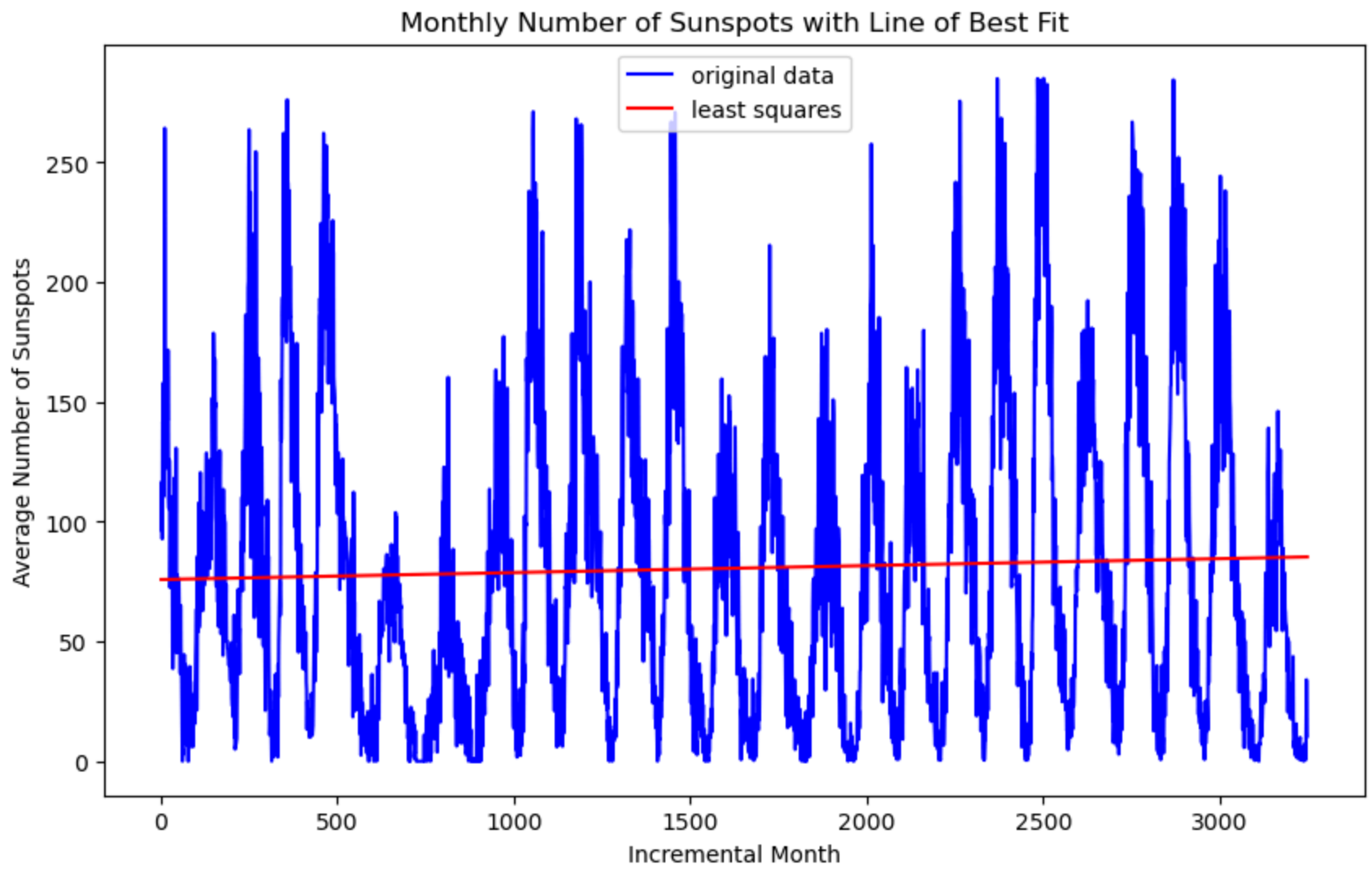
In [ ]:
```
#Create a variable which can be referrenced to get each of the different series
#Plot the decomposition with a period of 132 months (11 years)
sunspots_seasonal_decomposition = decompose_seasonality(sunspots_cleaned['Monthly Mean'],132)
plot_seasonal_decomposition(sunspots_cleaned['Monthly Mean'],(132))
```



Trend of Sunspots

In [ ]:
```
#Create an incrementing months column so that a line of best fit can go through the data
sunspots_cleaned_no_anomalies['Incrementing Month'] = range(1,len(sunspots_cleaned_no_anomalies) + 1)
sunspots_cleaned['Incrementing Month'] = range(1,len(sunspots_cleaned) + 1)
```

In [ ]:
```
#Plot the original data with the least squares line
plot_least_squares_line(sunspots_cleaned_no_anomalies['Incrementing Month'],sunspots_cleaned_no_anomalies['Monthly Mean']
```



In [ ]:
```
#Look at the ADF statistic to determine stationarity
adf_test(sunspots_cleaned['Monthly Mean'])
```

```
ADF Statistic:  -10.497051662546138
P-Value:  1.1085524921956742e-18
Critical Value:
        1%: -3.43
        5%: -2.86
        10%: -2.57
```

There is an overall constant movement in the data. This is solidified by the ADF statistic being well above the threshold to have a 99% confidence that the graph is stationary. This means that the value of the time series is not dependent on the time.

Seasonality of sunspots

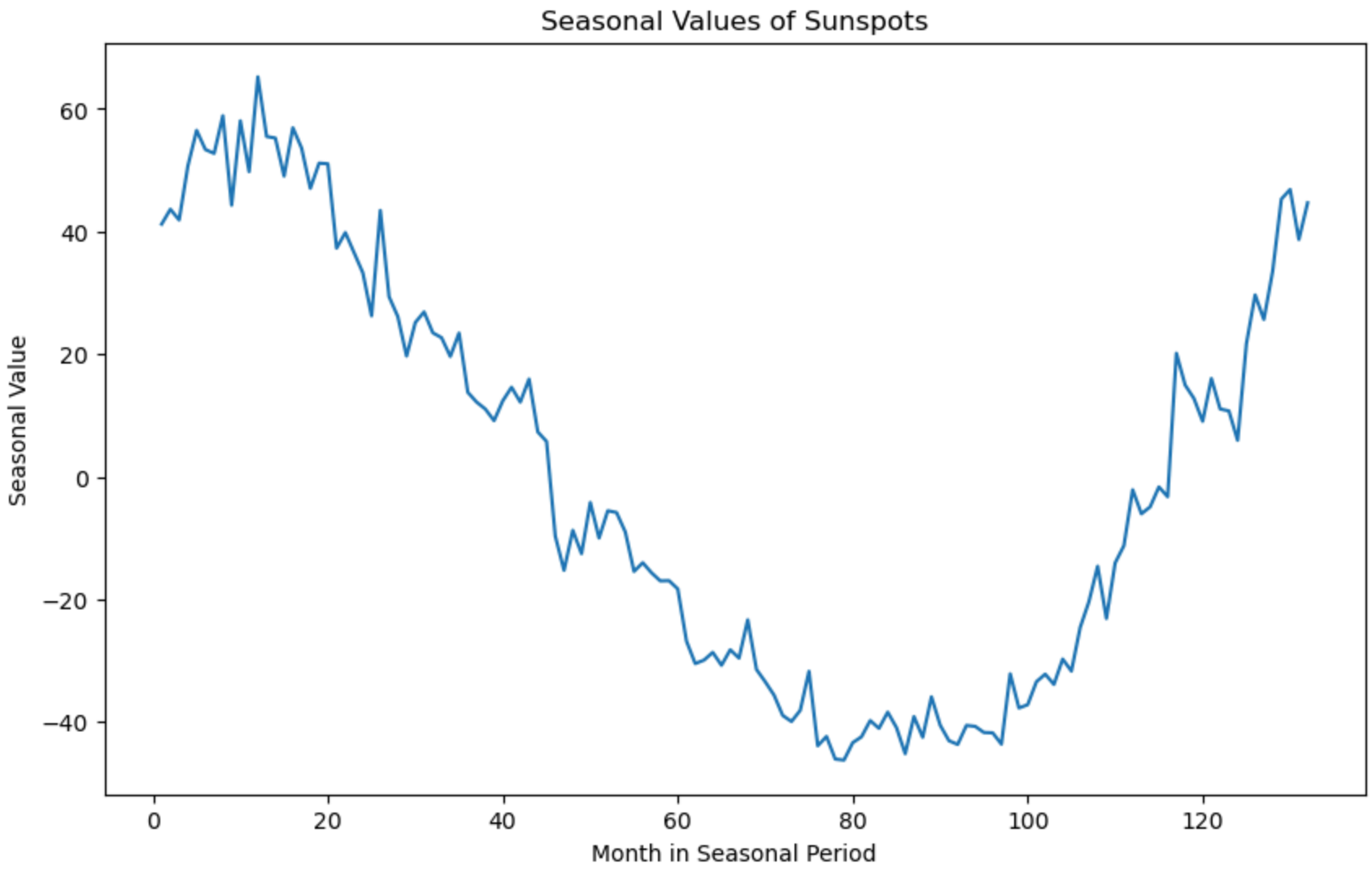
In [ ]:
```
#Create a dataframe for 11 years of the seasonality rather than 1750-2020
#This will be 132 monthly values
#These values can observe how values are expected to change based on their time within the 11 years
sunspots_cleaned_seasonal_values = pd.DataFrame(sunspots_seasonal_decomposition[2].iloc[0:132])
sunspots_cleaned_seasonal_values['Incrementing Month'] = range(1, len(sunspots_cleaned_seasonal_values)+1)
sunspots_cleaned_seasonal_values
```

Out[ ]:

Date	seasonal	Incrementing Month
1749-01-01	41.202966	1
1749-02-01	43.634516	2
1749-03-01	41.888019	3
1749-04-01	50.691066	4
1749-05-01	56.499336	5
...	...	...
1759-08-01	33.495485	128
1759-09-01	45.294632	129
1759-10-01	46.859374	130
1759-11-01	38.694112	131
1759-12-01	44.685384	132

132 rows × 2 columns

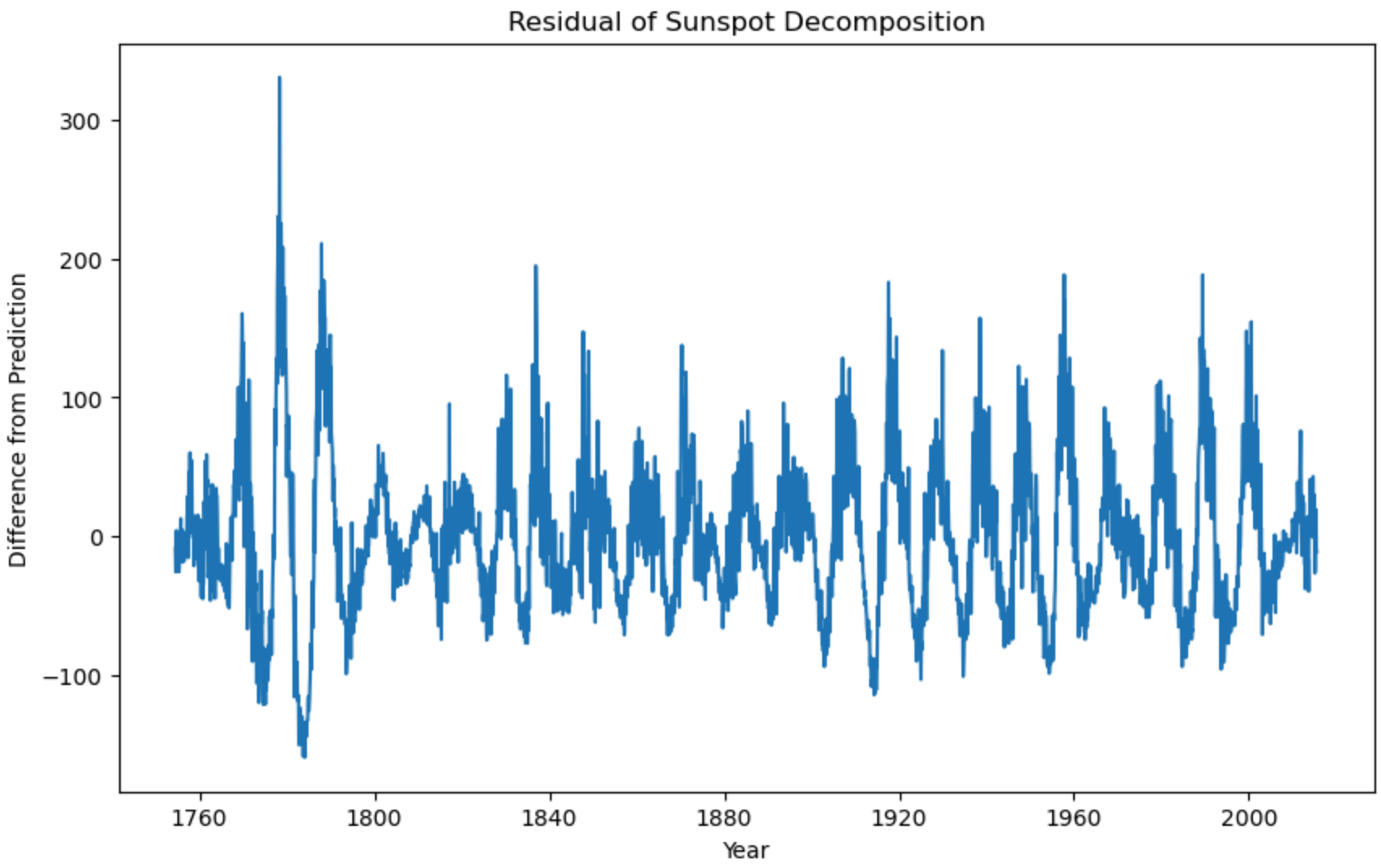
In [ ]:
```
#Plot the seasonal values across one seasonal period
time_series(sunspots_cleaned_seasonal_values['Incrementing Month'],sunspots_cleaned_seasonal_values['seasonal'],'Seasonal
```



The seasonal pattern shows that there is a spike around the start of the seasonal period with a downturn to about 80-100 months onwards (years 7-8), before returning back to a maximum.

Residual of sunspots

In [ ]:
```
#Plot the residual of sunspots
time_series(sunspots_seasonal_decomposition[3].index,sunspots_seasonal_decomposition[3],'Residual of Sunspot Decompositio
```



As found before the mean absolute error is about 41. From the graph you can see that there is no real clear pattern and it seems to be relative white noise about a mean of 0 which means there isn't any missing component of the data that has not been included.

1.3 ARIMA Model Building:

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

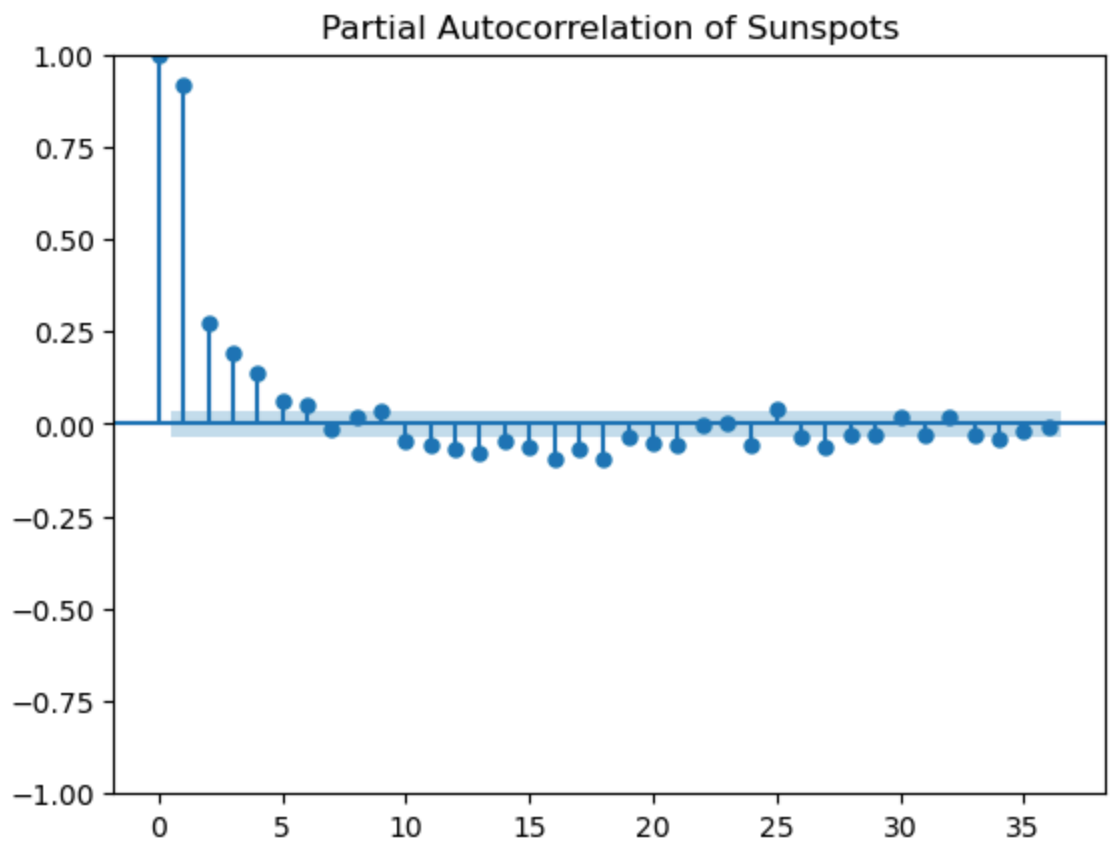
ARIMA Modeling

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)
- SARIMA: Seasonal Autoregressive Integrated Moving Average (p value, d value, q value, P value, D value, Q value, seasonal). P value found through seasonal PACF, D value found through seasonal ADF statistic, Q value found through seasonal ACF

p,d,q orders value

In [ ]:
```
#Use the pacf plot to find the p model
sunsports_cleaned_pacf = plot_pacf(sunspots_cleaned['Monthly Mean'],title='Partial Autocorrelation of Sunspots')
```



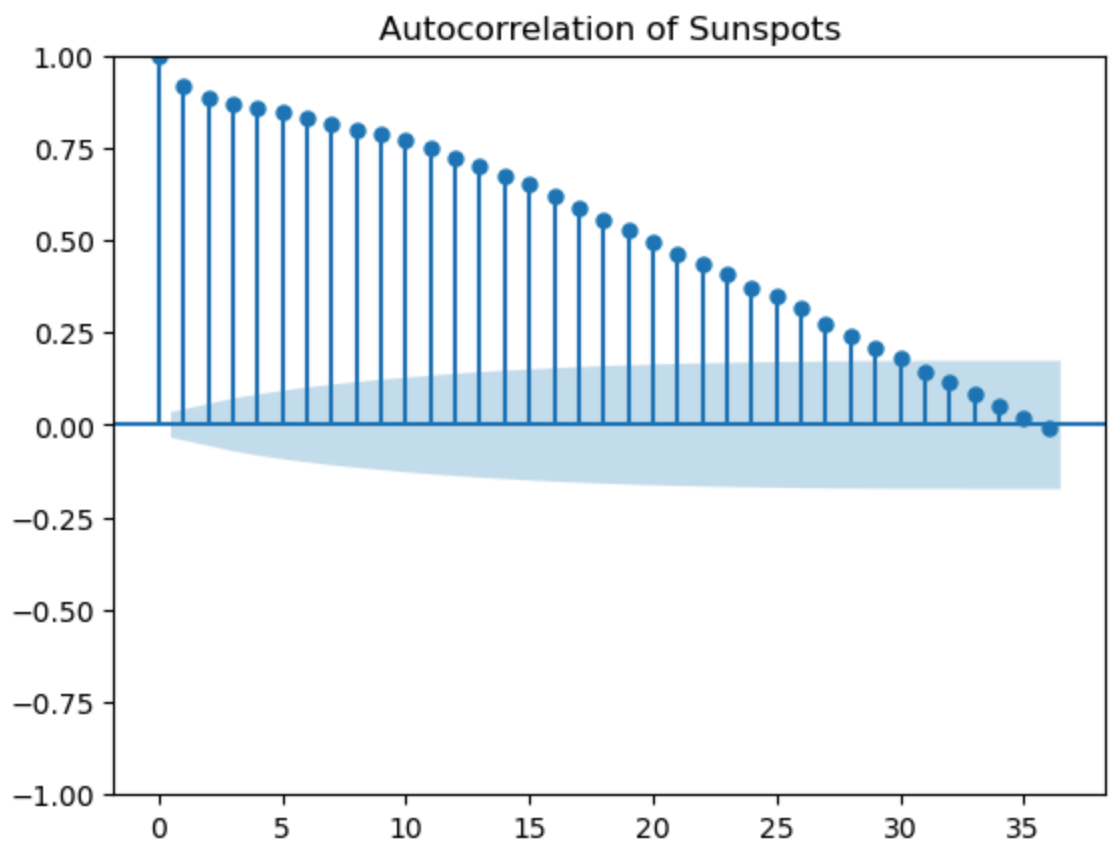
In [ ]:
```
#Create a list of p orders
p_orders_value = [1,2]
```

In [ ]:
```
#Look at the ADF statistic to determine the d value
adf_test(sunspots_cleaned['Monthly Mean'])
```

```
ADF Statistic:  -10.497051662546138
P-Value:  1.1085524921956742e-18
Critical Value:
        1%: -3.43
        5%: -2.86
        10%: -2.57
```

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

In [ ]:
```
#Look at the ACF plot to find q
sunspots_cleaned_acf = plot_acf(sunspots_cleaned['Monthly Mean'],title='Autocorrelation of Sunspots')
```



In [ ]:
```
#Create a list of q orders
q_orders_value = [1,2]
```

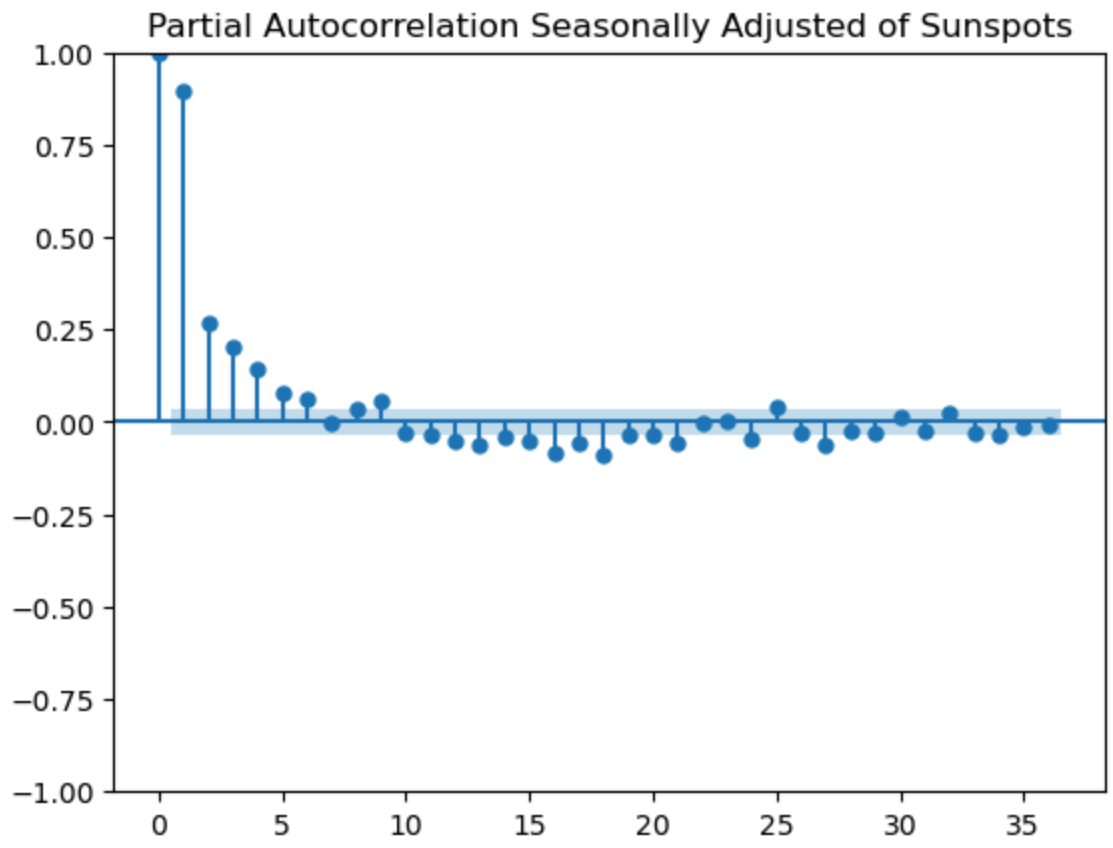
Because of the complexity of the seasonal period of the data, the sarima optimisation function was struggling to find appropriate values, even when given just a few options. I am going to choose the most appropriate value based on the ACF and PACF models and find the SARIMA AIC and BIC that way.

P,D,Q orders value

In [ ]:
```
#Remove the seasonal values from the cleaned data
sunspots_cleaned_seasonally_adjusted = sunspots_seasonal_decomposition[0] - sunspots_seasonal_decomposition[2]
sunspots_cleaned_seasonally_adjusted
```

Out[ ]:
```
Date
1749-01-01    55.497034
1749-02-01    60.665484
1749-03-01    74.811981
1749-04-01    42.108934
1749-05-01    85.200664
                ...
2020-09-01    41.133193
2020-10-01    55.089427
2020-11-01    75.692852
2020-12-01    63.546387
2021-01-01    53.990658
Length: 3265, dtype: float64
```

In [ ]:
```
#Use the PACF to find P values
sunspots_cleaned_seasonally_adjusted_pacf = plot_pacf(sunspots_cleaned_seasonally_adjusted,title='Partial Autocorrelation
```



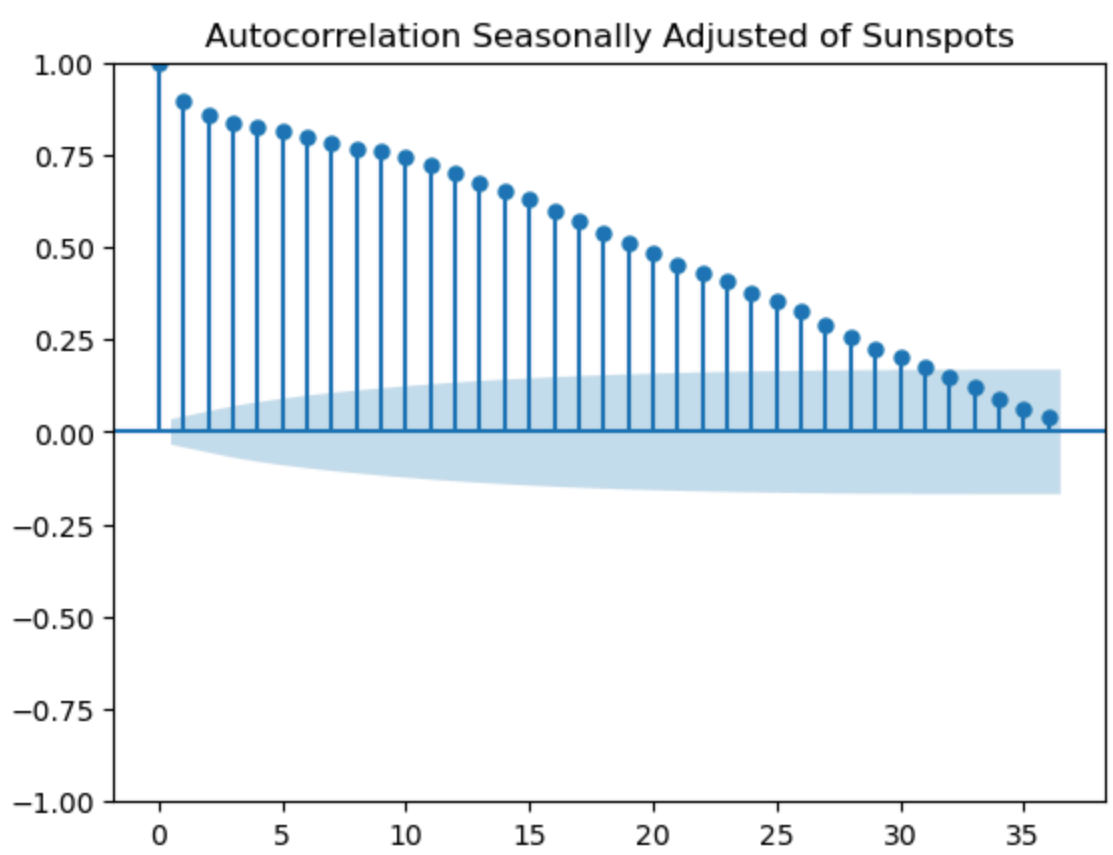
In [ ]:
```
#Create an instance of P orders
P_orders_value = 1
```

In [ ]:
```
#Look at the ADF statistic of the sesonally adjusted value to find D value
adf_test(sunspots_cleaned_seasonally_adjusted)
```

```
ADF Statistic:  -9.502626339936468
P-Value:  3.412819308375638e-16
Critical Value:
        1%: -3.43
        5%: -2.86
        10%: -2.57
```

In [ ]:
```
#Create an instance of D orders
D_orders_value = 0
```

In [ ]:
```
#Use the ACF plot to find Q value
sunspots_cleaned_seasonally_adjusted_acf = plot_acf(sunspots_cleaned_seasonally_adjusted,title='Autocorrelation Seasonall
```



In [ ]:
```
#Create an instance of Q orders
Q_orders_value = 1
```

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 [ ]:
```
#Create a list of different ARIMA models to choose
evaluate_time_series_models(sunspots_cleaned['Monthly Mean'],p_orders_value,q_orders_value,d_orders_value)
```

Out[ ]:

	Model	Order	AIC	BIC
15	ARIMA	(2, 1, 2)	30250.957980	30281.411524
5	ARMA	(1, 0, 2)	30316.781285	30347.236360
9	ARIMA	(1, 0, 2)	30316.781285	30347.236360
7	ARMA	(2, 0, 2)	30318.779675	30355.325766
13	ARIMA	(2, 0, 2)	30318.779675	30355.325766
6	ARMA	(2, 0, 1)	30321.223487	30351.678562
12	ARIMA	(2, 0, 1)	30321.223487	30351.678562
14	ARIMA	(2, 1, 1)	30329.219256	30353.582091
11	ARIMA	(1, 1, 2)	30331.183799	30355.546633
10	ARIMA	(1, 1, 1)	30333.403911	30351.676037
4	ARMA	(1, 0, 1)	30360.870706	30385.234767
8	ARIMA	(1, 0, 1)	30360.870706	30385.234767
1	AR	(2, 0, 0)	30515.008081	30539.372141
0	AR	(1, 0, 0)	30766.404603	30784.677648
3	MA	(0, 0, 2)	32960.383449	32984.747509
2	MA	(0, 0, 1)	34223.100649	34241.373694

With a seasonal value of 132 it is taking too long for usual models to incorporate seasonality. One way of over coming this is understanding that the values are really not changing much from year to year and you could continue to project the most recent seasonal values and continue that. Understanding that there is going to be some change from year to year. Use SES or single exponential modeling as well as Holt Winter's Seasonal Method or triple exponential smoothing to determine its accuracy.

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. SES - Single Exponential Smoothing is going to continue the most recent trend to forecast
3. Holt Winter's Seasonal Method - Triple exponential smoothing is going to incorporate seasonality into the system

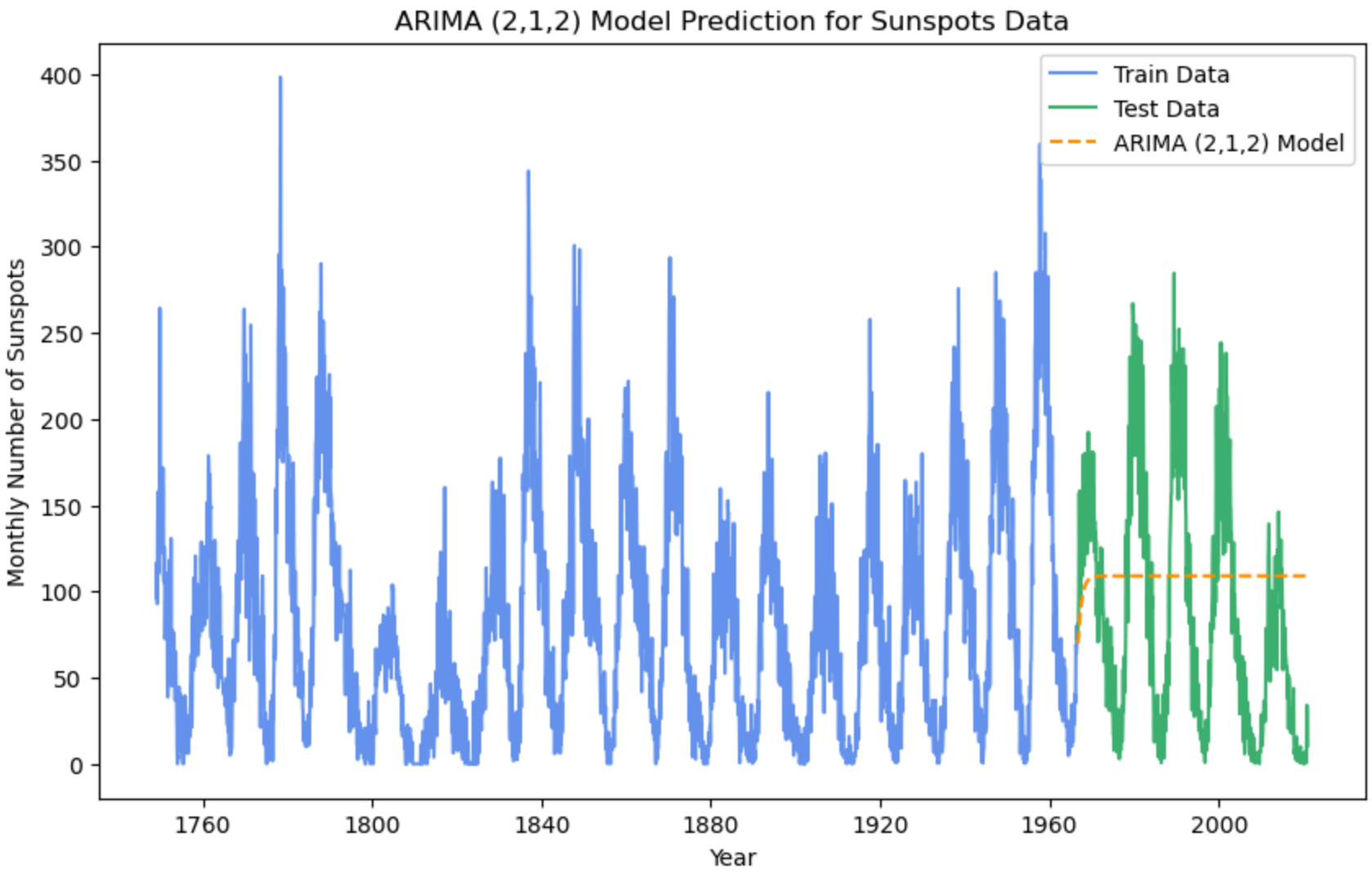
In [ ]:
```
#Get the index value to split the data 0.8 to 0.2 for train and testing respectively.
split_index = int(len(sunspots_cleaned)*0.8)

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

1. ARIMA Modeling

- ARIMA (2,1,2)

In [ ]:
```
# Fit ARIMA model to the training data
model_arima_1 = ARIMA(train_data['Monthly Mean'], order=(2, 1, 2))
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[0], 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['Monthly Mean'], test_data['Monthly Mean'], test_predictions_arima_1,
                    'Year', 'Monthly Number of Sunspots', 'ARIMA (2,1,2) Model Prediction for Sunspots Data','ARIMA (2,1,
```



The ARIMA model does not take into account those fluctuations that are observed in the test data. This is likely not going to be a good modeling forecasting for future events.

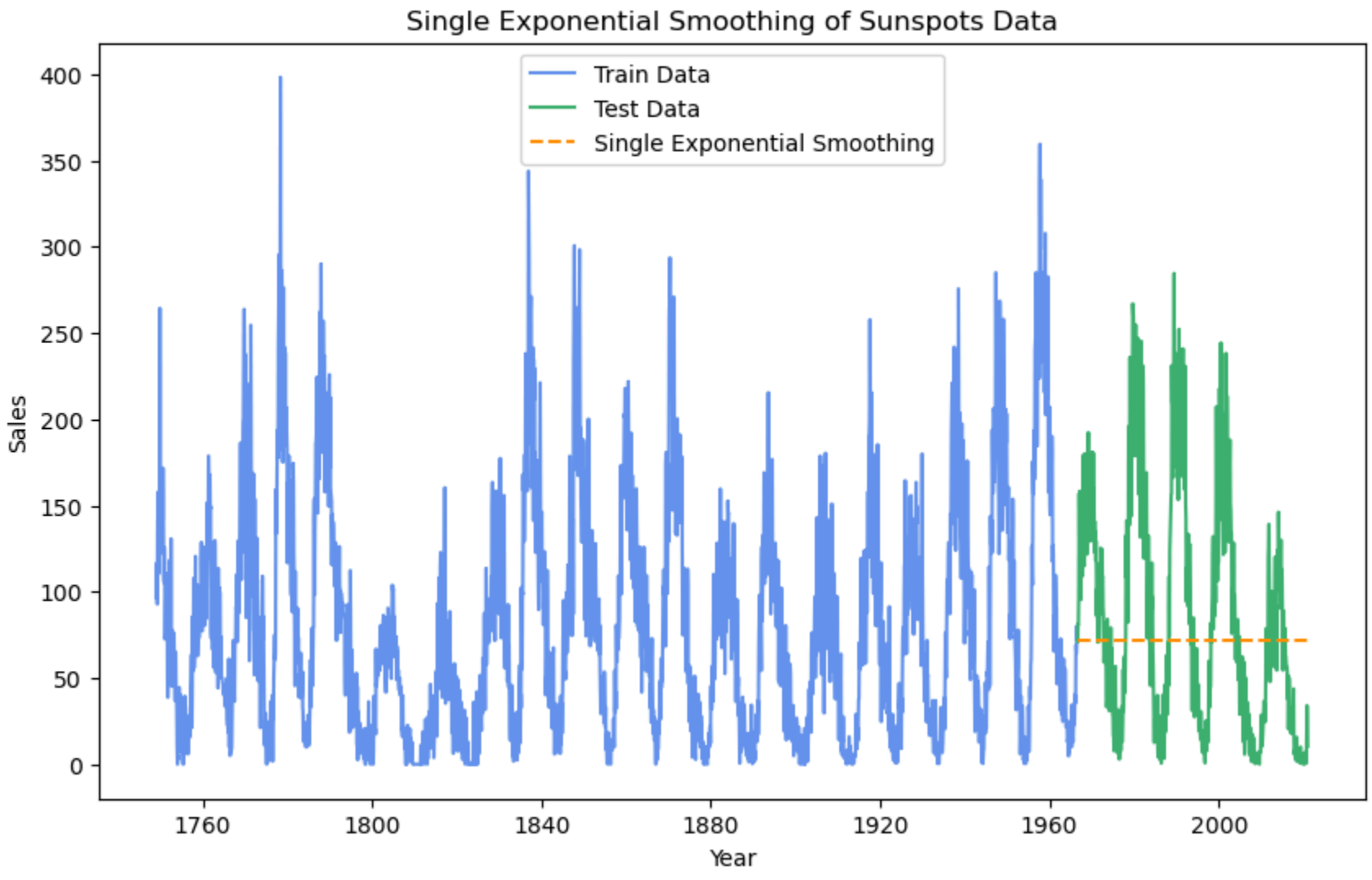
In [ ]:
```
#Get the mean absolute error for arima model
arima_1_mae = mean_absolute_error(test_data['Monthly Mean'],test_predictions_arima_1)
arima_1_mae
```

Out[ ]:
```
63.86739600092766
```

2. SES Model

In [ ]:
```
#Create SES model
ses_model=SimpleExpSmoothing(train_data['Monthly Mean'])

#Fit the model
ses_model = ses_model.fit(smoothing_level=0.5)

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

test_training_model(train_data['Monthly Mean'], test_data['Monthly Mean'], ses_predictions,
                    'Year', 'Sales', 'Single Exponential Smoothing of Sunspots Data', 'Single Exponential Smoothing')
```

```
/Users/ben_nicholson/opt/anaconda3/lib/python3.9/site-packages/pandas/util/_decorators.py:207: EstimationWarning: Model h
as no free parameters to estimate. Set optimized=False to suppress this warning
  return func(*args, **kwargs)
```



In [ ]:
```
#Find the absolute mean error for the single exponential smoothing
ses_mae = mean_absolute_error(test_data['Monthly Mean'],ses_predictions)
ses_mae
```

Out[ ]:
```
57.97045909511802
```

Adding seasonal values to SES

You can see from the graph above that the straight line of the SES is a good way of understanding the stationarity of the graph. The next component will be to add the seasonal values and see how close that is to being accurate.

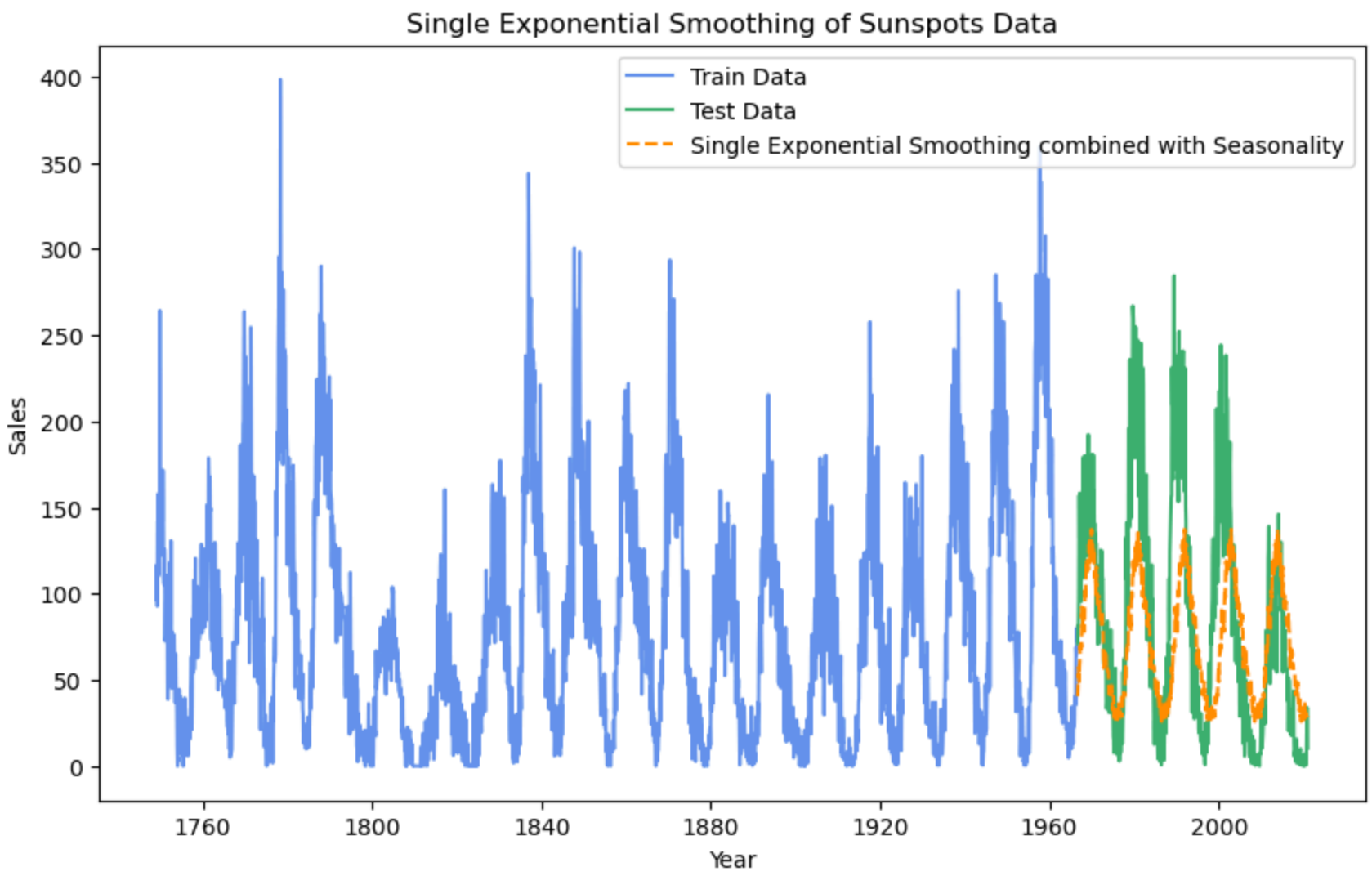
In [ ]:
```
#Create a dataframe of the SES predictions
ses_df = pd.DataFrame({'SES Predictions': ses_predictions})

#Create a column which has the seasonal values
ses_df['Seasonal Values'] = sunspots_seasonal_decomposition[2][-len(ses_df)::]

#Create a combined column which includes the SES prediction and the seasonal values
ses_df['Final Prediction'] = ses_df['SES Predictions'] + ses_df['Seasonal Values']

ses_df
```

Out[ ]:

	SES Predictions	Seasonal Values	Final Prediction
1966-09-01	72.078042	-31.673534	40.404509
1966-10-01	72.078042	-24.562123	47.515920
1966-11-01	72.078042	-20.357435	51.720607
1966-12-01	72.078042	-14.571845	57.506197
1967-01-01	72.078042	-23.094982	48.983060
...	...	...	...
2020-09-01	72.078042	-40.533193	31.544850
2020-10-01	72.078042	-40.689427	31.388615
2020-11-01	72.078042	-41.692852	30.385190
2020-12-01	72.078042	-41.746387	30.331655
2021-01-01	72.078042	-43.590658	28.487384

653 rows × 3 columns

In [ ]:
```
#Plot the graph
test_training_model(train_data['Monthly Mean'], test_data['Monthly Mean'], ses_df['Final Prediction'],
                    'Year', 'Sales', 'Single Exponential Smoothing of Sunspots Data', 'Single Exponential Smoothing combi
```



In [ ]:
```
#Find the mean absolute error of the ses with sesonality included
mae_ses_with_seasonality = mean_absolute_error(test_data['Monthly Mean'],ses_df['Final Prediction'])
mae_ses_with_seasonality
```

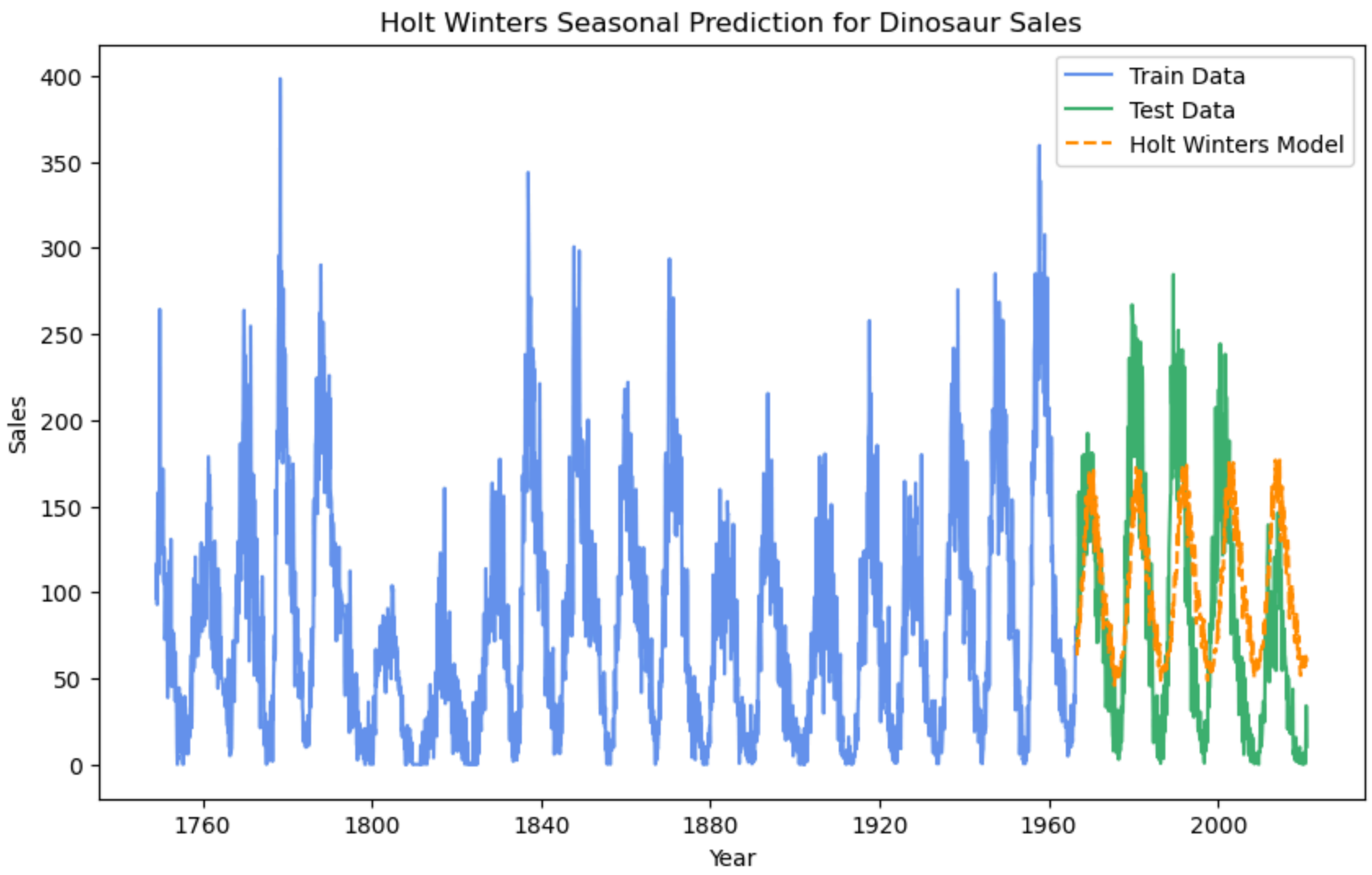
Out[ ]:
```
40.83485855936685
```

The mean absolute error for ses with seasonality improves upon the ARIMA and the normal SES. It is quite accurate apart from being off with some of the higher spikes that are taking place. It gets the overall pattern of a years movement quite well.

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 [ ]:
```
# Fit Holt-Winters method to the training data with multiplicative seasonality
holt_winters_1_model = ExponentialSmoothing(train_data['Monthly Mean'], seasonal_periods=132, trend='add', seasonal='add'
holt_winters_1_model = holt_winters_1_model.fit(smoothing_level=0.2, smoothing_trend=0, smoothing_seasonal=0.15)

# 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['Monthly Mean'], test_data['Monthly Mean'], holt_winters_test_predictions,
                    'Year', 'Sales', 'Holt Winters Seasonal Prediction for Dinosaur Sales', 'Holt Winters Model')
```



In [ ]:
```
holt_winters_mae = mean_absolute_error(test_data['Monthly Mean'],holt_winters_test_predictions)
holt_winters_mae
```

Out[ ]:
```
49.145153591694005
```

Model Evaluation Comparison

In [ ]:
```
print(f'SES mae: {ses_mae}')
print(f'SES with Seasonal Components mae: {mae_ses_with_seasonality}')
print(f'ARIMA mae: {arima_1_mae}')
print(f'Holt Winters mae: {holt_winters_mae}')
```

```
SES mae: 57.97045909511802
SES with Seasonal Components mae: 40.83485855936685
ARIMA mae: 63.86739600092766
Holt Winters mae: 49.145153591694005
```

SES with Seasonal Component performs the best out of the models when comparising the training model with the test data.

1.5 Forecasting:

Forecast the future for what seems reasonable based upon the data.

Plot the predictions against the actual data (if available) to visualize the forecast's accuracy.

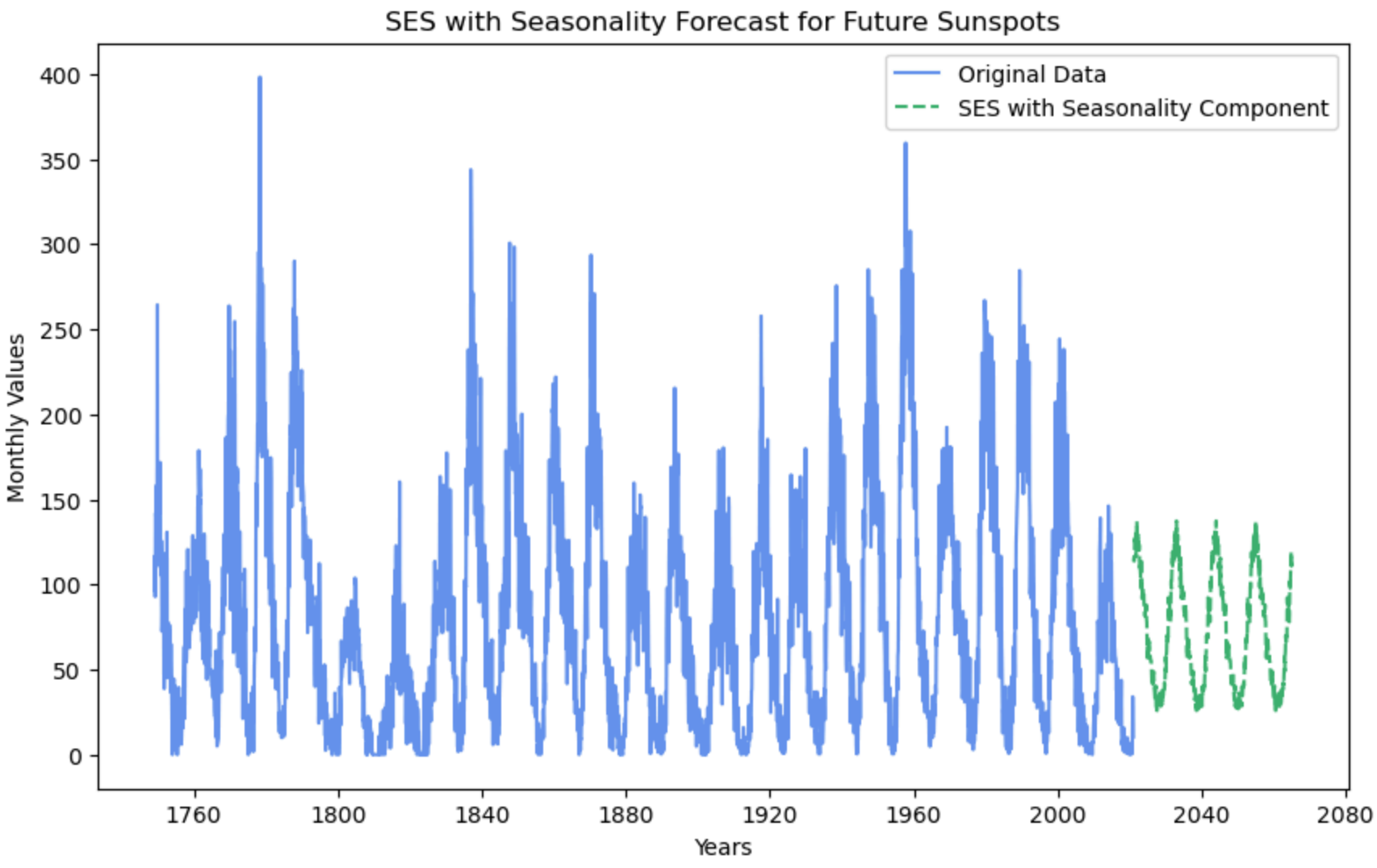
In [ ]:
```
#Load in the dataframe and forecast 4 seasons or 44 years out into the future using the ses with seasonality forecast mod
seasonal_values_forecast = sunspots_seasonal_decomposition[2][0:(132*4)]
ses_values_forecast = ses_df['SES Predictions'][0:(132*4)]

#Create a dataframe
ses_with_seasonality_forecast = pd.DataFrame(seasonal_values_forecast.values + ses_values_forecast.values)
ses_with_seasonality_forecast['Date'] = pd.date_range(start='2021-01-01',periods=(132*4),freq='M')

#Set the date column to be the index
ses_with_seasonality_forecast.set_index('Date',inplace=True)

ses_with_seasonality_forecast.columns = ['Monthly Mean']
ses_with_seasonality_forecast
```

Out[ ]:

Date	Monthly Mean
2021-01-31	113.281008
2021-02-28	115.712558
2021-03-31	113.966062
2021-04-30	122.769108
2021-05-31	128.577378
...	...
2064-08-31	105.573527
2064-09-30	117.372675
2064-10-31	118.937416
2064-11-30	110.772154
2064-12-31	116.763426

528 rows × 1 columns

In [ ]:
```
model_forecast(sunspots_cleaned,sunspots_cleaned['Monthly Mean'],ses_with_seasonality_forecast['Monthly Mean'],'Years','M
```


SES with Seasonality Forecast for Future Sunspots
Original Data
SES with Seasonality Component
Monthly Values
Years
0
50
100
150
200
250
300
350
400
1760
1800
1840
1880
1920
1960
2000
2040
2080