Module 10: Time Series Analysis and Forecasting

Quick Reference Guide

Learning Outcomes:

1. Describe the forecasting problem.
2. Use modeling in time series forecasting with the effect of autocorrections.
3. Compute the sample autocorrelation and partial autocorrelation functions for a time series.
4. Construct a model using classical time series decomposition.
5. Differentiate between key characteristics of the autoregressive model and moving average model.
6. Use the ACF and PACF to select the orders of an ARMA model.
7. Interpret a forecast and its uncertainty.
8. Determine the invertibility and stationarity of an ARMA model and use ACF and PACF to select the order of ARMA model.

Introduction

Machine learning models leverage the defining property of time series data, which is that the data is **sequential**. To analyze time series data, you need tools such as the autocorrelation and partial autocorrelation functions. These tools involve three concepts:

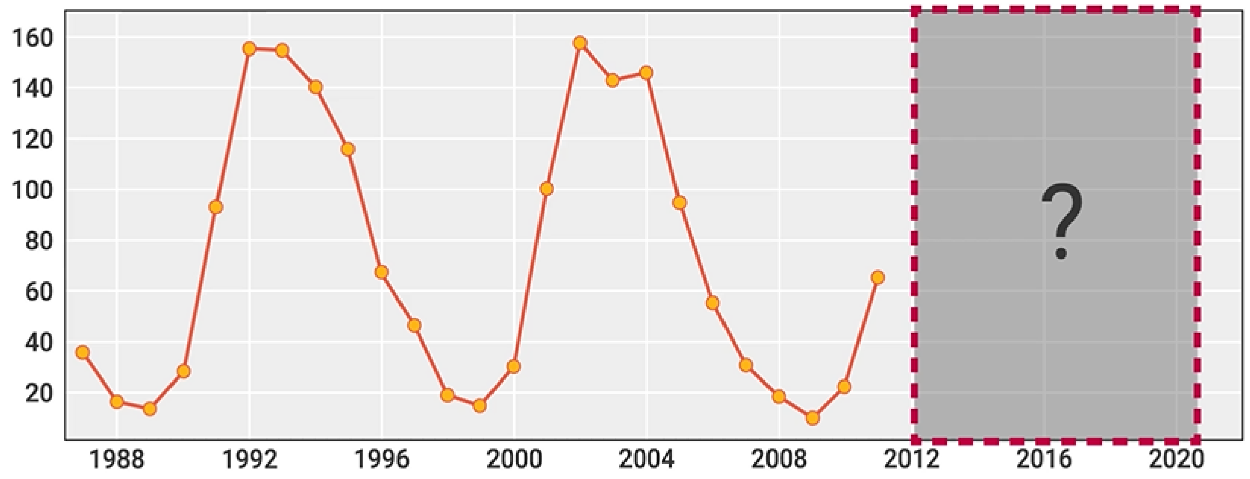
- trends
- seasonality
- stationarity

Problem Statement

There are many problems involving time series data. The most important of these is **forecasting**.

The forecasting problem is as follows: Given some historical data, can you try to predict, or forecast, what will happen over some future time window?

Consider this plot of yearly data that rises and falls over a period of about a decade:

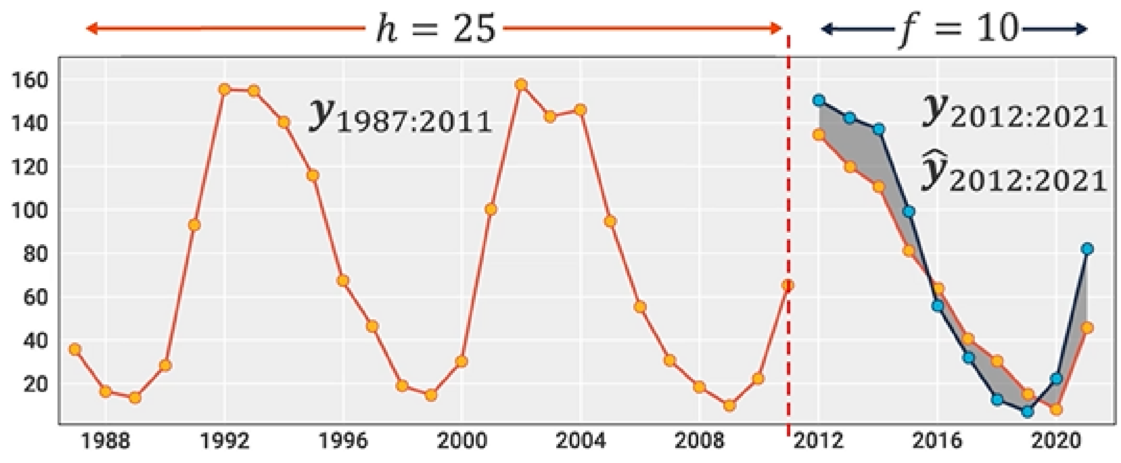


Assuming that you are now in the year 2011, you are tasked with predicting the future evolution of this time series over the next ten years from 2012 to 2021. This is yearly data. However, the methods for time series forecasting do not depend on the duration of the time step, only on the amount of data. So, it makes no difference whether you are making a ten-year prediction with 100 years of data, or a ten-second prediction with 100 seconds of second-by-second data. Because these two situations involve the same number of datapoints, the results will be the same.

Approaching the forecasting problem proceeds along four steps:

1. Collect historical data
2. Train a model
3. Use the model to make a forecast
4. Score or evaluate the performance of the model

Suppose you have made your ten-year forecasts back in 2011 and it is now 2021. Next, you need to assess the model's performance.



You can use this notation:

- Value of the quantity at time t: y_t
- Size of the historical dataset: h
- Number of time steps being forecast: f
- Historical data as an array: $\boldsymbol{y}_{t-h:t} = [y_{t-h+1}, y_{t-h+2}, \dots, y_t]$
- Forecast: $\hat{\boldsymbol{y}}_{t:t+f} = [\hat{y}_{t+1}, \hat{y}_{t+2}, \dots, \hat{y}_{t+f}]$
- Future data: $\boldsymbol{y}_{t:t+f} = [y_{t+1}, y_{t+2}, \dots, y_{t+f}]$
- Forecast error: $\boldsymbol{e}_t = \boldsymbol{y}_{t:t+f} - \hat{\boldsymbol{y}}_{t:t+f}$

The bold font indicates that it is a vector, not a scalar. The prediction error is the difference between the prediction and the actual data for that period.

Reducing an array to a single number

The error, $\boldsymbol{e}_t$, is an array, so it needs to be reduced to a single number.

There are two common ways to do this:

- **Mean absolute error** (MAE): The mean of the absolute values of the elements
- **Root mean squared error** (RMSE): The square root of the mean of the squared errors

The equations for MAE and RMSE, respectively, are as follows.

$$\text{MAE} = \|\mathbf{e}_t\|_1 = \sum_{\tau=t+1}^{t+f} |e(\tau)|$$

$$\text{RMSE} \; = \; \|\mathbf{e}_t\|_2 = \sqrt{\frac{1}{f}\sum_{\tau=t+1}^{t+f} e(\tau)^2}$$

When choosing a **forecasting model**, this means deciding on any equation or algorithm that can generate a forecast from historical data. To resolve which model is better, you compare models' scalar prediction errors.

Modeling (Part 1)

Probability theory can be used to help analyze time series data. Assuming that the data was generated by an unseen probability density function (PDF), your task is to infer something about that PDF from the data. But for this approach, the random variables are now organized in a sequential manner into a stochastic process.

A **stochastic process** is an ordered sequence of random variables. It uses this notation:

$$(Y_t)_{1:T} = (Y_1, Y_2, \ldots, Y_T)$$

T is the length of the stochastic process and it could be infinite.

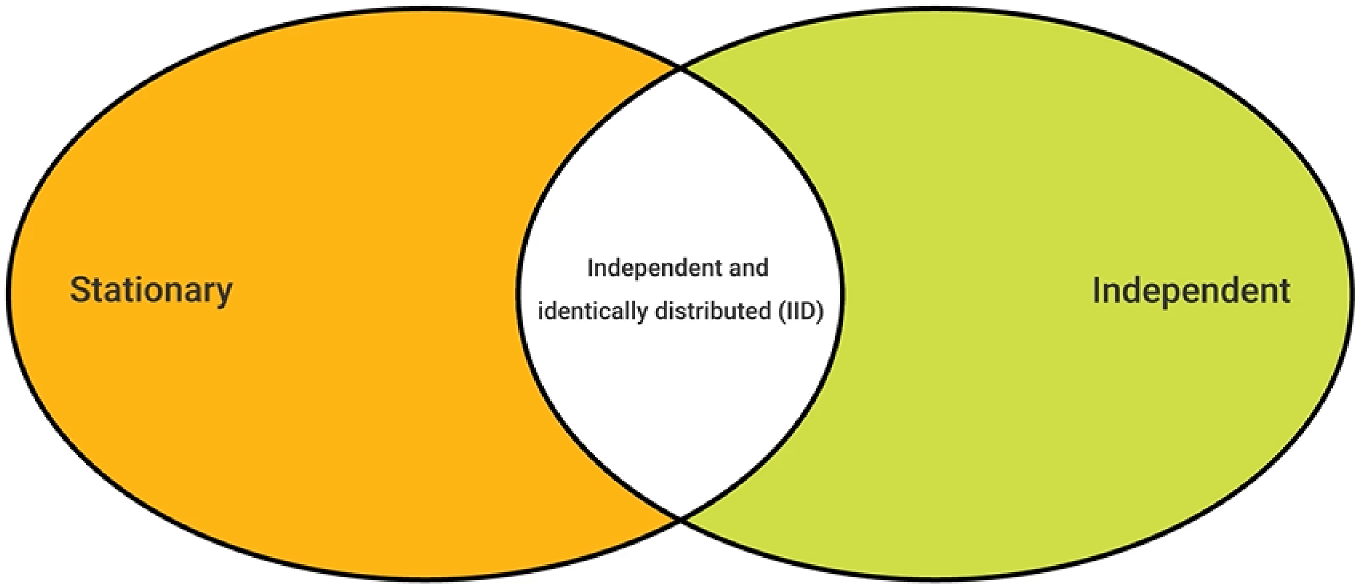
The term **stochastic process** applies to a sequence of random variables. **A time series** is a single sample from the stochastic process. Different time series can be produced from the same stochastic process.

Stochastic process properties

Two important properties of stochastic processes are **stationarity** and **independence**.

- **Stationarity**: A process is stationary when its statistical properties—its mean, variance, and the correlations between different points in time—remain constant over time
  $p(Y_{t_1}) = p(Y_{t_2})$
  where t_1 is equal to the distribution at any other time t_2 and all Y_t are identically distributed
- **Independence**: A process is independent when all of its constituent random variables, Y_t, are mutually independent
  $p(Y_1,\ Y_2,\ \ldots,\ Y_T) = \prod_{t=1}^{T} p(Y_t)$
  so past values do not influence current or future values
  $p(Y_t |\ Y_{t-1},\ Y_{t-2},\ \ldots) = p(Y_t)$
  where the distribution of y at time t, given the historic information, is equal to the distribution of Y_t without that information

Stationarity and independence are distinct from one another. A stochastic process can be stationary and not independent, independent and non-stationary, neither, or both. A process that is both stationary and independent is **independent and identically distributed** (IID).



Modeling (Part 2)

When trying to build a forecasting model, it is useful to estimate the correlations between pairs of random variables. These correlations are usually organized into a matrix, the **autocorrelation** matrix.

Shown here is the autocorrelation matrix, P, of a stochastic process (Y_t).

$$\mathrm{P} = \begin{bmatrix} 1 & \rho(Y_1, Y_2) & \rho(Y_1, Y_3) & \rho(Y_1, Y_4) & \dots \\ \rho(Y_2, Y_1) & 1 & \rho(Y_2, Y_3) & \rho(Y_2, Y_4) & \dots \\ \rho(Y_3, Y_1) & \rho(Y_3, Y_2) & 1 & \rho(Y_3, Y_4) & \dots \\ \rho(Y_4, Y_1) & \rho(Y_4, Y_2) & \rho(Y_4, Y_3) & 1 & \dots \\ \vdots & \vdots & \vdots & \vdots & \ddots \end{bmatrix}$$

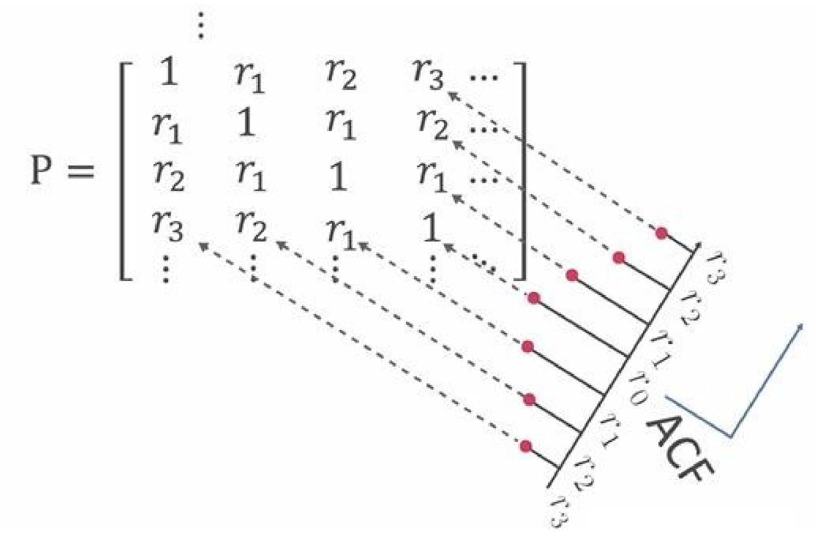
Along the diagonal are values of 1. These are correlations of Y_t with itself.

$$P = \begin{bmatrix} 1 & \rho(Y_1,Y_2) & \rho(Y_1,Y_3) & \rho(Y_1,Y_4) & \cdots \\ \rho(Y_2,Y_1) & 1 & \rho(Y_2,Y_3) & \rho(Y_2,Y_4) & \cdots \\ \rho(Y_3,Y_1) & \rho(Y_3,Y_2) & 1 & \rho(Y_3,Y_4) & \cdots \\ \rho(Y_4,Y_1) & \rho(Y_4,Y_2) & \rho(Y_4,Y_3) & 1 & \cdots \\ \vdots & \vdots & \vdots & \vdots & \ddots \end{bmatrix}$$

Then on the diagonals directly above and below that central diagonal, you get autocorrelations between time steps that are one time step apart, Y_1 and Y_2, Y_2, and Y_3, et cetera.

Recall that a stationary process is one for which the statistical properties of any window of data are the same no matter where the window is placed in time.

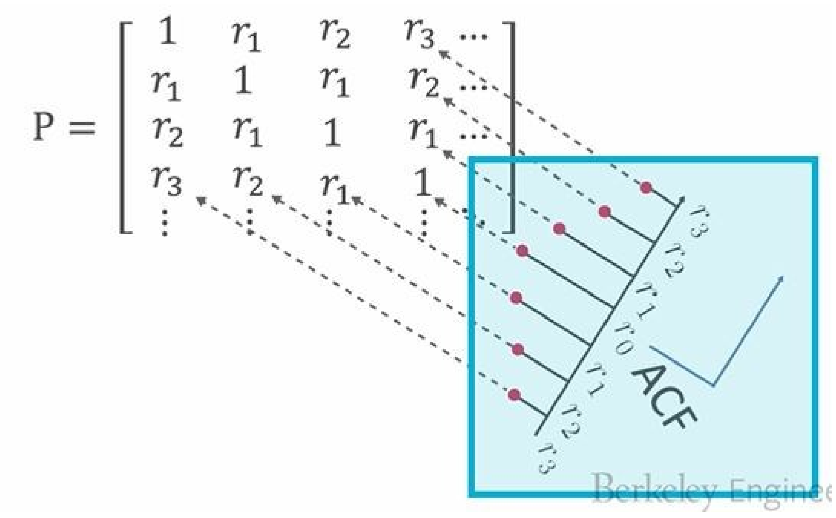
$$\text{stationary} \Rightarrow \text{lag 1: } r_1 = \rho(Y_2,\ Y_1) = \rho(Y_3,Y_2) = \rho(Y_4,Y_3) = \ \ldots$$
$$\text{lag 2: } r_2 = \rho(Y_3,\ Y_1) = \rho(Y_4,Y_2) = \rho(Y_5,Y_3) = \ \ldots$$
$$\vdots$$



For example, considering a window of size two, the correlations between Y_1 and Y_2 should be equal, as are the correlations for Y_2 and Y_3, et cetera. All of the values, r_1, above and below the central diagonal must be the same.

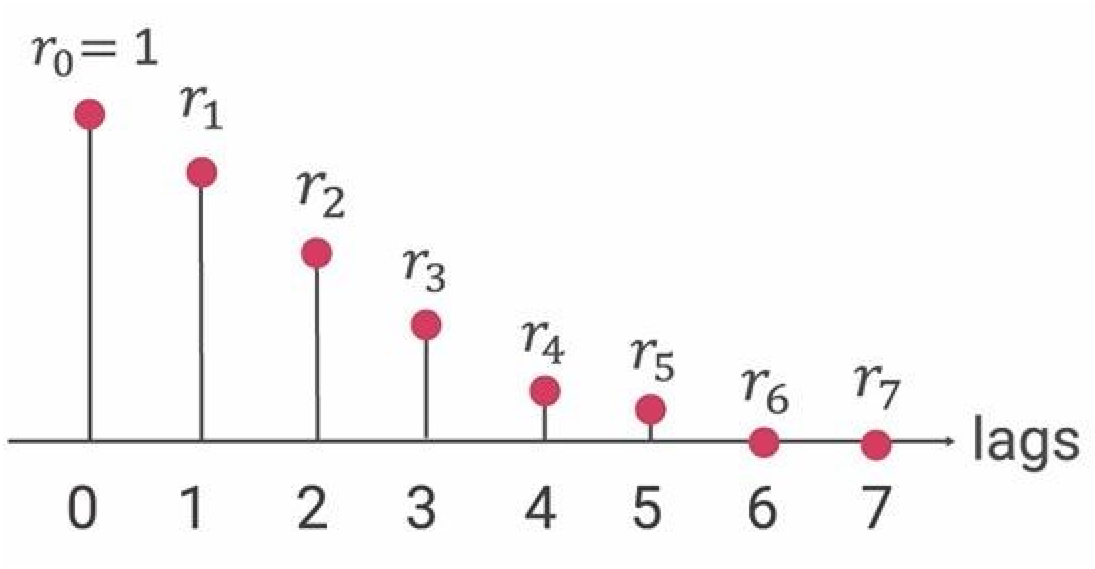
Similarly, with a window of size three, conclude that all of the entries that are two steps above or below the central diagonals, r_2, should be equal, and so on, as you move further away from the central diagonal.

For stationary processes, the autocorrelation matrix has a very simple structure that is constant along diagonals. This observation motivates the use of the **autocorrelation function (ACF)**, instead of the autocorrelation matrix, for stationary processes. The ACF, shown here at a 45° angle, collects the values along the diagonals and plots them as a function of the lag, or the distance between points in time.



$P = \begin{bmatrix} 1 & r_1 & r_2 & r_3 & \cdots \\ r_1 & 1 & r_1 & r_2 & \cdots \\ r_2 & r_1 & 1 & r_1 & \cdots \\ r_3 & r_2 & r_1 & 1 & \\ \vdots & \vdots & \vdots & \vdots & \end{bmatrix}$
r_3
r_2
r_1
r_0
r_1
r_2
r_3
ACF
Berkeley Enginee

This is a typical autocorrelations plot. The first entry, r_0, corresponds to a lag of 0 and equals 1 since it is the correlation of Y_t with itself.



For many processes, ACF will decay with lag, meaning that the data is more strongly correlated with recent measurements than with measurements further back in time.

Just as you can compute sample correlations between random variables from data, you can also compute the sample autocorrelation function for a stochastic process from a time series. This is done by first computing the mean, $\bar{y}$, of the time series. Then compute the sample covariances, y_k , for each lag value, k, with the formula shown here. And then divide each c_k by c_0 so that the first sample autocorrelation, $\hat{\rho}_0$, equals 1.

$$\bar{y} = \frac{1}{T}\sum_{t=1}^{T} y_t$$

$$C_k = \frac{1}{T}\sum_{t=1}^{T-k} (y_t - \bar{y})(y_{t+k} - \bar{y})$$

$$\hat{\rho}_k = \frac{c_k}{c_0}$$

Autocorrelations

You can analyze time series using the statsmodels package, which provides functionality for doing statistical inferences, in Python. Statsmodels shares some functionality with scikit-learn but it also adds time series analysis, which scikit-learn does not have. Like scikit-learn, statsmodels comes with a series of datasets you can use to practice and test different models with.

The time series datasets provided with statsmodels include CO2 observations from the Mauna Loa observatory in Hawaii, sea surface temperatures related to El Niño, and yearly sunspots data from 1700 to 2008, among others.

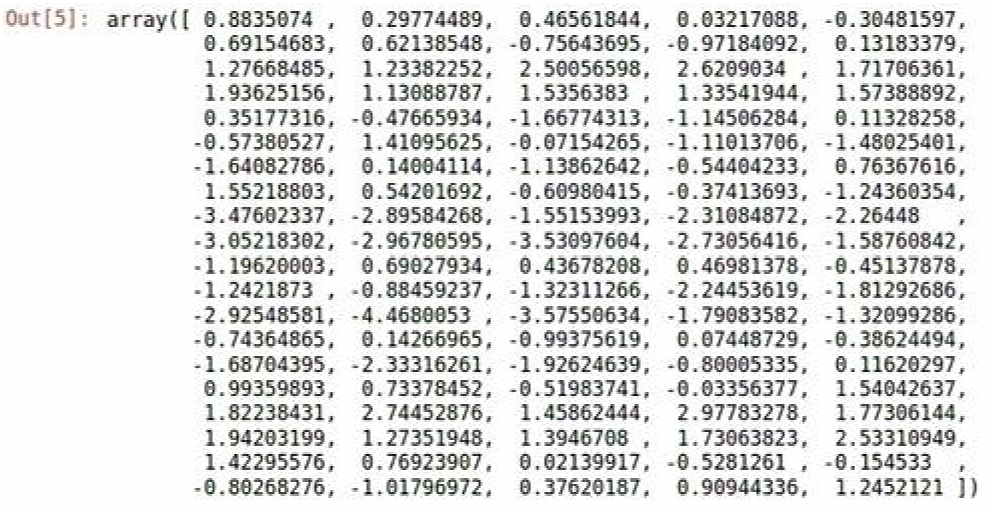
For generating a time series, you can use the ARIMA process module from the time series analysis (TSA) package of statsmodels.
import statsmodels.tsa.arima_process as arima_process

To start, you need to create an autoregressive moving average (ARMA) process, and it takes two parameters in the inputs. One is the autoregressive part, AR, and the other is the moving average part, MA.

In this instance, you set AR to 1 and −0.8, and MA to 1.
process = arima_process.ArmaProcess(ar=[1, −0.8], ma=[1])

Once you have the process object, you can take a sample of the process. You do that by calling the generate_sample() function on the ArmaProcess object. You pass into that function the size or the number of datapoints in the sample, which in this case is 100.
z = process.generate_sample(nsample=100)

When you run this, you get an array with 100 numbers in it.

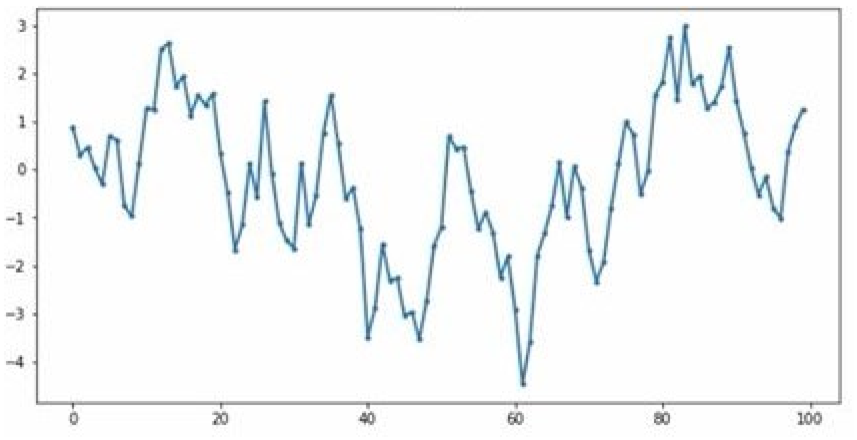


```
Out[5]: array([ 0.8835074 ,  0.29774489,  0.46561844,  0.03217088, -0.30481597,
                0.69154683,  0.62138548, -0.75643695, -0.97184092,  0.13183379,
                1.27668485,  1.23382252,  2.50056598,  2.6209034 ,  1.71706361,
                1.93625156,  1.13088787,  1.5356383 ,  1.33541944,  1.57388892,
                0.35177316, -0.47665934, -1.66774313, -1.14506284,  0.11328258,
               -0.57380527,  1.41095625, -0.07154265, -1.11013706, -1.48025401,
               -1.64082786,  0.14004114, -1.13862642, -0.54404233,  0.76367616,
                1.55218803,  0.54201692, -0.60980415, -0.37413693, -1.24360354,
               -3.47602337, -2.89584268, -1.55153993, -2.31084872, -2.26448    ,
               -3.05218302, -2.96780595, -3.53097604, -2.73056416, -1.58760842,
               -1.19620003,  0.69027934,  0.43678208,  0.46981378, -0.45137878,
               -1.2421873 , -0.88459237, -1.32311266, -2.24453619, -1.81292686,
               -2.92548581, -4.4680053 , -3.57550634, -1.79083582, -1.32099286,
               -0.74364865,  0.14266965, -0.99375619,  0.07448729, -0.38624494,
               -1.68704395, -2.33316261, -1.92624639, -0.80005335,  0.11620297,
                0.99359893,  0.73378452, -0.51983741, -0.03356377,  1.54042637,
                1.82238431,  2.74452876,  1.45862444,  2.97783278,  1.77306144,
                1.94203199,  1.27351948,  1.3946708 ,  1.73063823,  2.53310949,
                1.42295576,  0.76923907,  0.02139917, -0.5281261 , -0.154533   ,
               -0.80268276, -1.01796972,  0.37620187,  0.90944336,  1.2452121 ])
```

You can plot this using matplotlib in the usual way.

plt.figure(figsize=(10,5))

plt.plot(z, linewidth=2,marker=' . ')



If you took another random sample and plotted it, it would look different. It would be different each time for a different random sample from that process. The process depends on the parameters that you pass in.

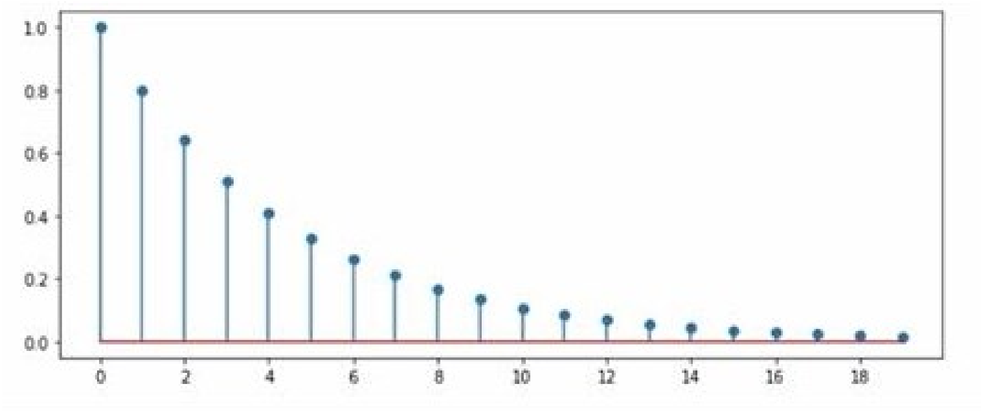
Note: You could, if you are not careful, pass in numbers that would produce an unstable process.

Next, you need to plot the theoretical autocorrelations for this process.

So to do that, you call the ACF function on the process to return the autocorrelations. You pass in the number of lags or the number of coefficients that you want, in this case 20.

acf = process.acf(lags=20)

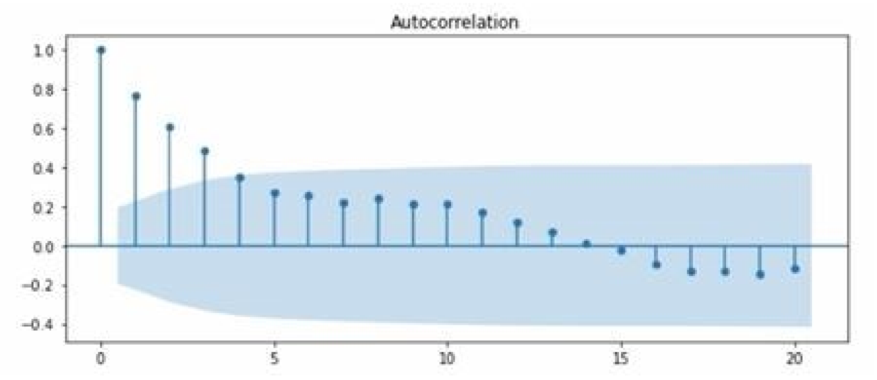
This produces an array of 20 numbers. As expected, it starts with 1 and thereafter the autocorrelations decay exponentially with the number of lags.



This is typical of a stationary process.

Next, you plot the sample autocorrelation of the time series using the TSA module of statsmodels.

import statsmodels.graphics.tsaplots as tsaplots
fig, ax = plt.subplots(figsize=(10,4))
tsaplots.plot_acf(z, lags=20, ax=ax)
plt.show()



Next, you can explore how these functions work on real data. First, you load the sunspots data that is available from statsmodels.

sunspots = pd.read_csv('Sunspots.csv')

sunspots.head()

	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

It has a column called Date that has the date of each observation. The first date is the 31st of January, 1749 and it is a string.

Next, you can convert dates into a timestamp object using the to_datetime() function in pandas.

pd.to_datetime(sunspots.loc[0, 'Date'])

The resulting output is a timestamp: Timestamp('1749-01-31 00:00:00').

Next, you would like to use timestamps for your index so pandas will be able to plot it nicely.

sunspots = sunspots.set_index(pd.to_datetime(sunspots['Date']))

Date	Unnamed: 0	Date	Monthly Mean Total Sunspot Number
1749-01-31	0	1749-01-31	96.7
1749-02-28	1	1749-02-28	104.3
1749-03-31	2	1749-03-31	116.7
1749-04-30	3	1749-04-30	92.8
1749-05-31	4	1749-05-31	141.7

As expected, you get a date as the index, but there are two columns that you are not very interested in. So you call drop(columns) to get rid of them.

sunspots.drop(columns=['Unnamed: 0', 'Date'], inplace=True)

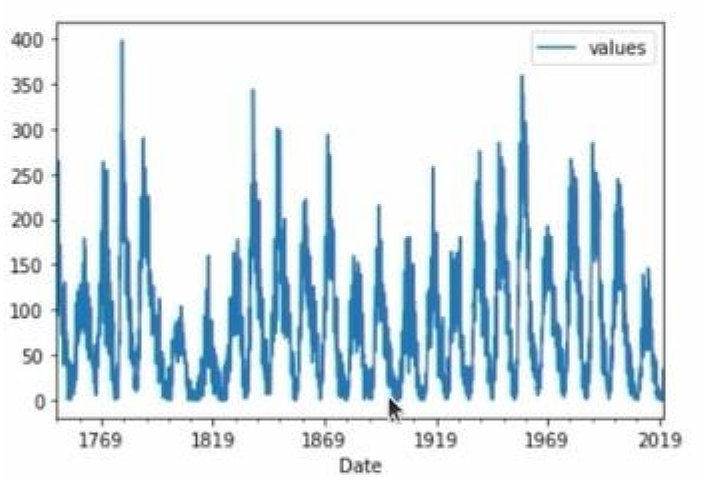
Date	Monthly Mean Total Sunspot Number
1749-01-31	96.7
1749-02-28	104.3
1749-03-31	116.7
1749-04-30	92.8
1749-05-31	141.7
...	...
2020-09-30	0.6
2020-10-31	14.4

2020-11-30	34.0
2020-12-31	21.8
2021-01-31	10.4

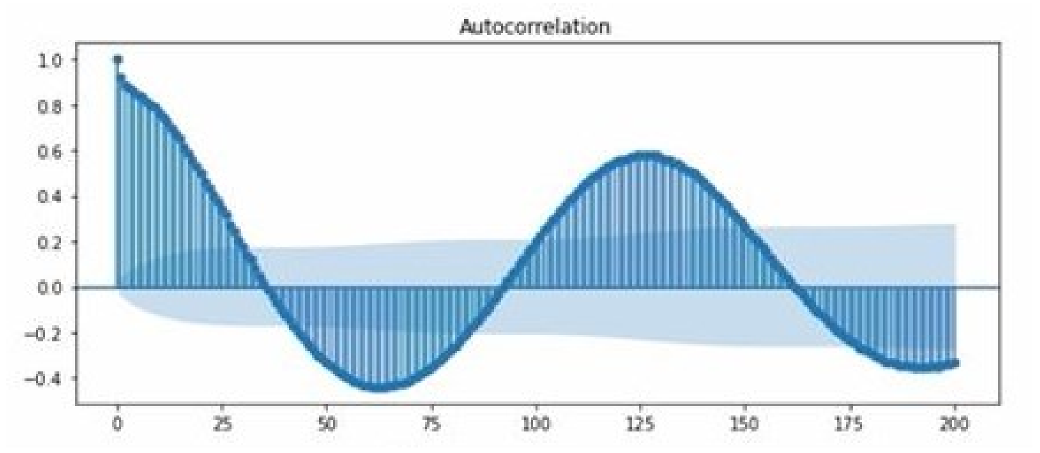
But the column header, Monthly Mean Total Sunspot Number, is really long. You can rename it 'values' or similar to make it easier to work with. Then you have a dataframe with timestamps in the index and a values column.

Next, you can plot it.
sunspots.plot()

The plot indicates that the sunspots on the surface of the sun increase and decrease in number with a period of about a decade.



Next, you plot the sample autocorrelation function.
fig, ax = plt.subplots(figsize=(10,4))
tsa.plots.plot_acf(sunspots, lags=200, ax=ax)
plt.show()

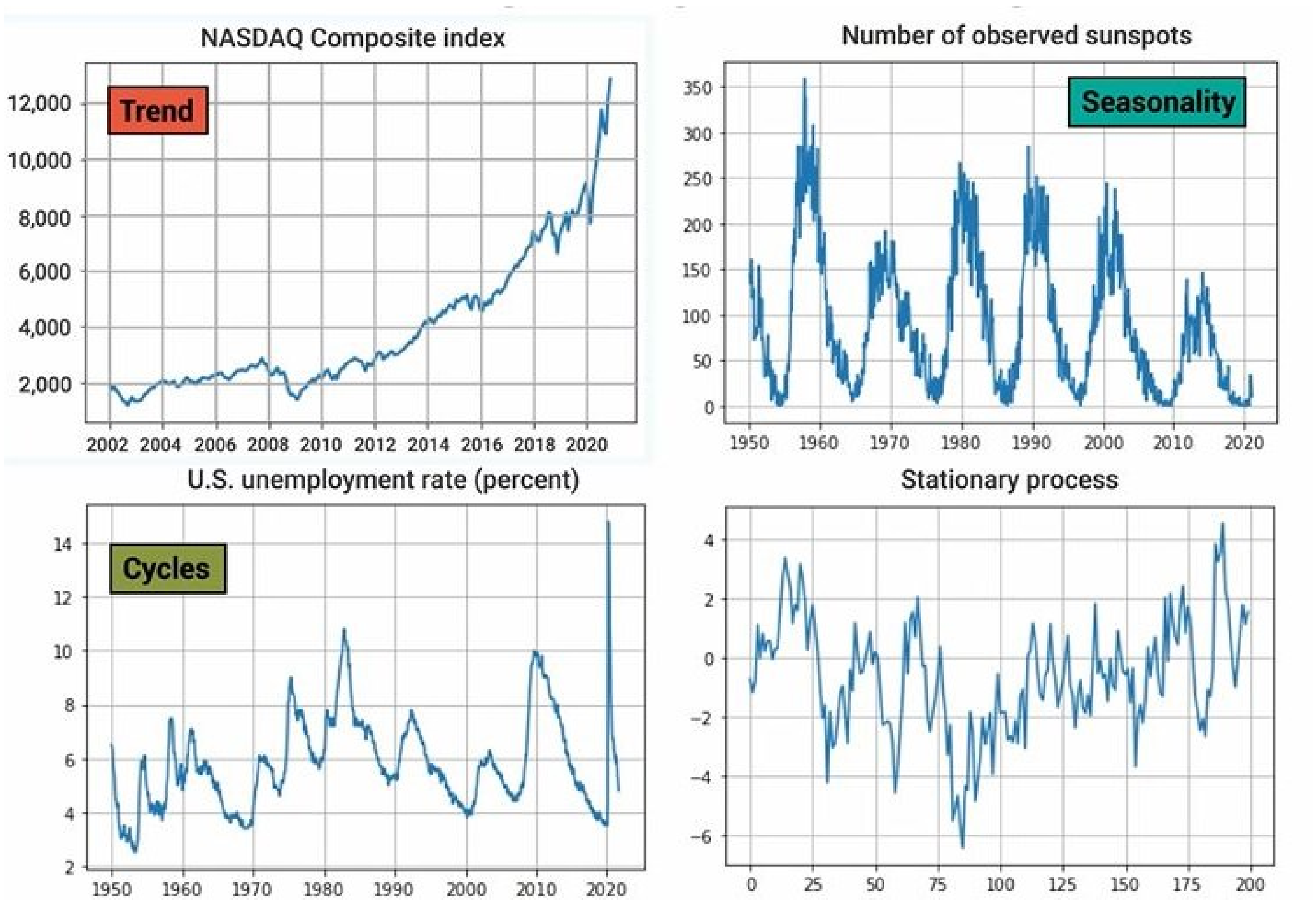


It is evident from the plot that the autocorrelations are not decaying at all, the time series behaves sort of sinusoidally and even after 200 lags is not seeming to decay. So this confirms that the sunspots data is not stationary.

Although the sunspots data is not stationary, the difference of sunspots from one month to the next is believed to be stationary.

Decomposition (Part 1)

Consider some examples of time series.



Stationary processes are more difficult to predict than non-stationary processes since they lack a trend, seasonality, or cycles.

However, stationary processes may still contain structure in the form of the autocorrelation function, which can be estimated and used to make short-term predictions.

Long-term behavior that can be modeled, include:

- Trend
- Seasonality
- Cycles

You can model a time series, y, as a composition of four terms:

$$\boldsymbol{y} = \boldsymbol{t} + \boldsymbol{c} + \boldsymbol{s} + \boldsymbol{r}$$

- Trend ($\boldsymbol{t}$): Long-term behavior
- Cycles ($\boldsymbol{c}$): Variations that break with the trend, but do not happen with a set period i.e. random low-frequency variations
- Seasonality ($\boldsymbol{s}$): Known periodic behavior
- Residue ($\boldsymbol{r}$): Any other behavior

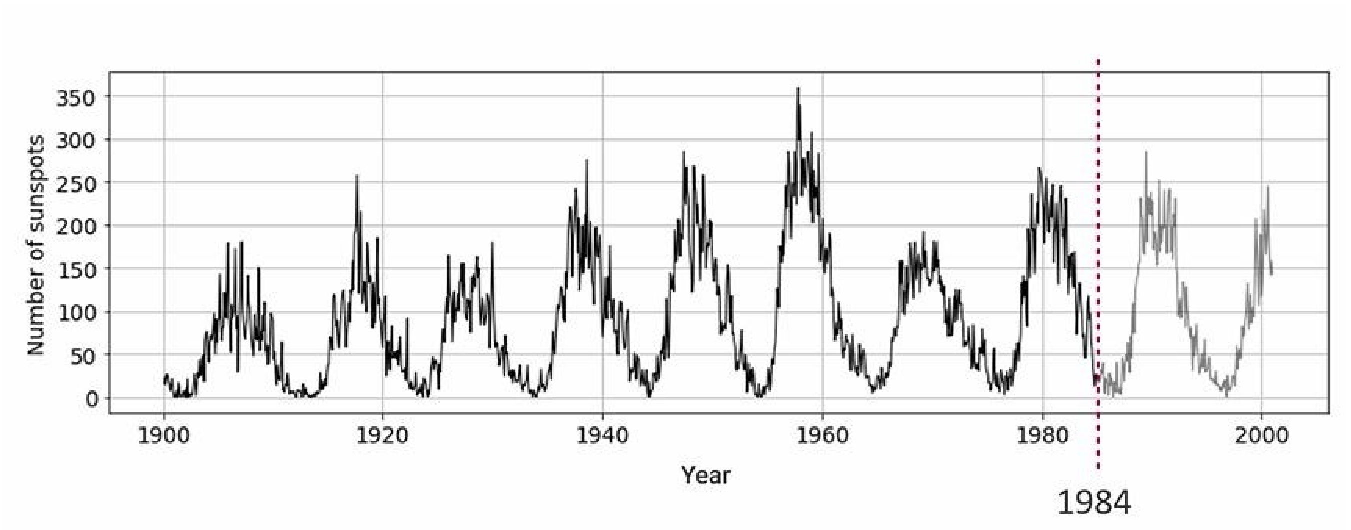
The symbols in this formula are in bold because they are vectors.

Decomposition (Part 2)

The procedure for time series decomposition has three steps:

1. Smooth the data to compute the trend and cycles
2. Compute the seasonal component
3. Check the stationarity of the residue

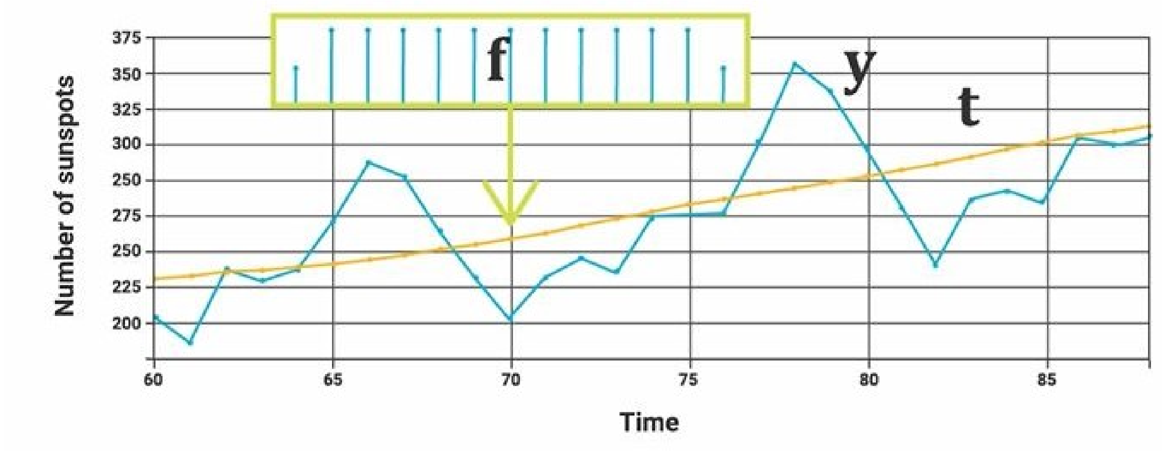
Imagine the year is 1984 and you have been tasked with predicting the next 15 years of sunspot activity. You are to base your prediction on the observations of sunspots from 1900 to 1984.



As the first step, you need to extract the trend and cycles by smoothing the data. To do so, you run $\boldsymbol{y}$ through a filter, $\boldsymbol{f}$ – f is an array of positive numbers that add up to one. The length of f should be chosen to be similar but greater than the period of the data. In this case, the period is about 128 months and so f has a length of about 129.

$$\mathbf{t} = \mathrm{conv}(\mathbf{y}, \mathbf{f})$$

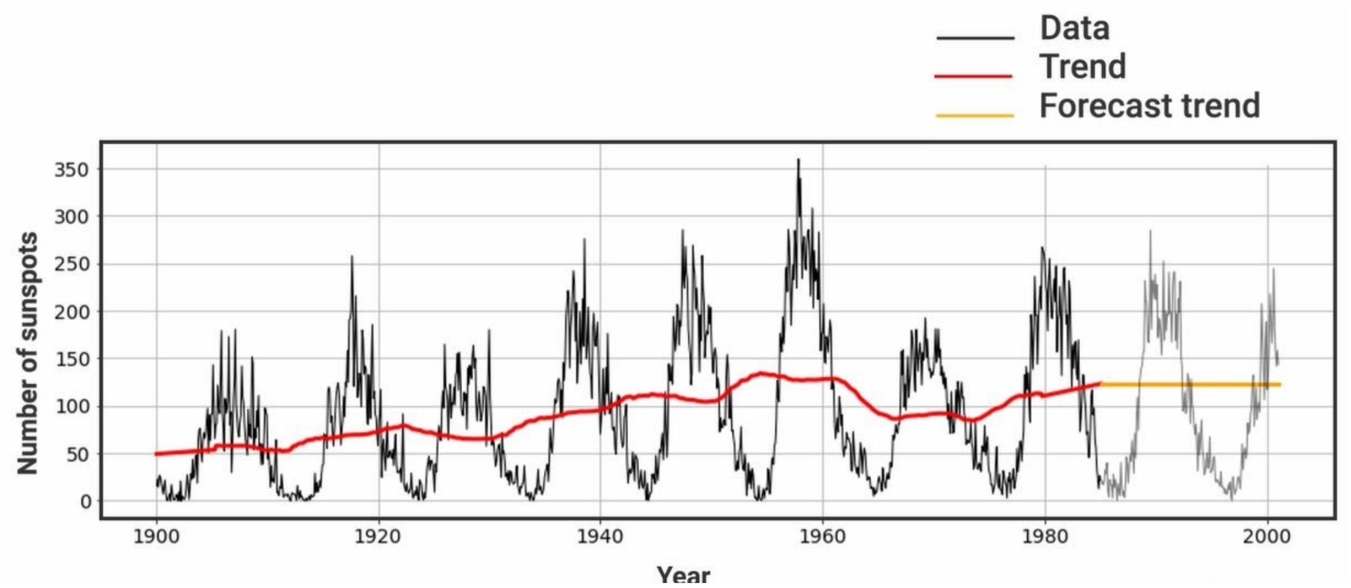
$$t_t = f_0 y_{t-6} + f_1 y_{t-5} + \ \dots\ + \ f_{12} y_{t+6}$$



To compute $\boldsymbol{t}$, at some time you take a weighted-average of nearby values of $\boldsymbol{y}$ with weights given by $\boldsymbol{f}$.

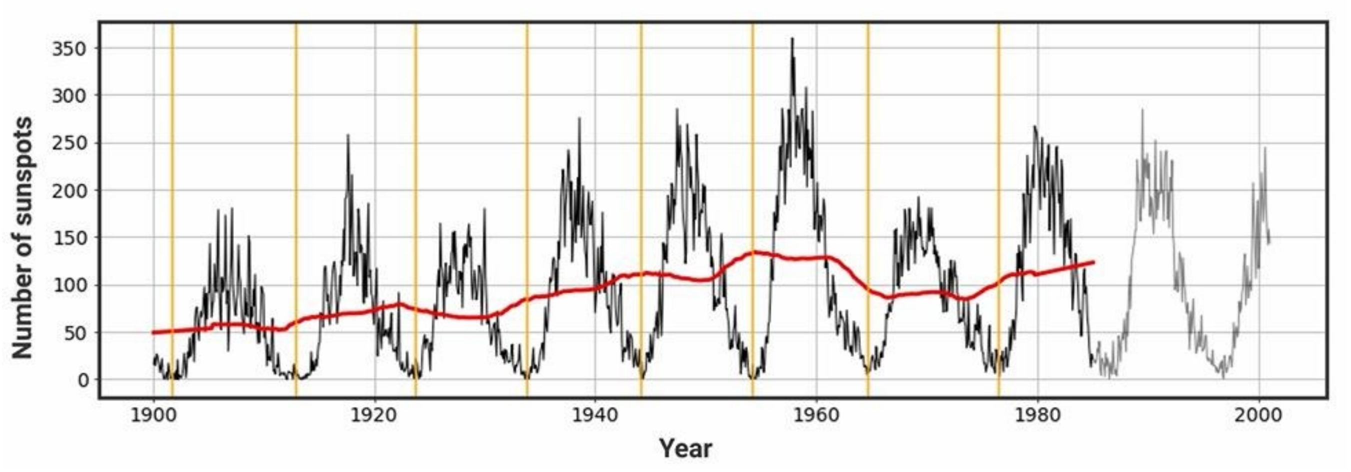
Because the filter is longer than the period, the seasonal component is removed completely – this leaves only trend and cycles, assuming the cycles are longer than the seasonality and not too severe.

Here is the resulting trend in red.



It looks pretty flat, so at this point you can make a decision to extrapolate it into the future with a horizontal line, as shown here in orange. Had you observed an increasing or decreasing trend, you might have used a different extrapolation technique. For example, you may have used a linear regression with linear, quadratic, or exponential basis functions.

The next step in the process is to compute the seasonal component. To do this, you chop the historical time series into segments of length one period.

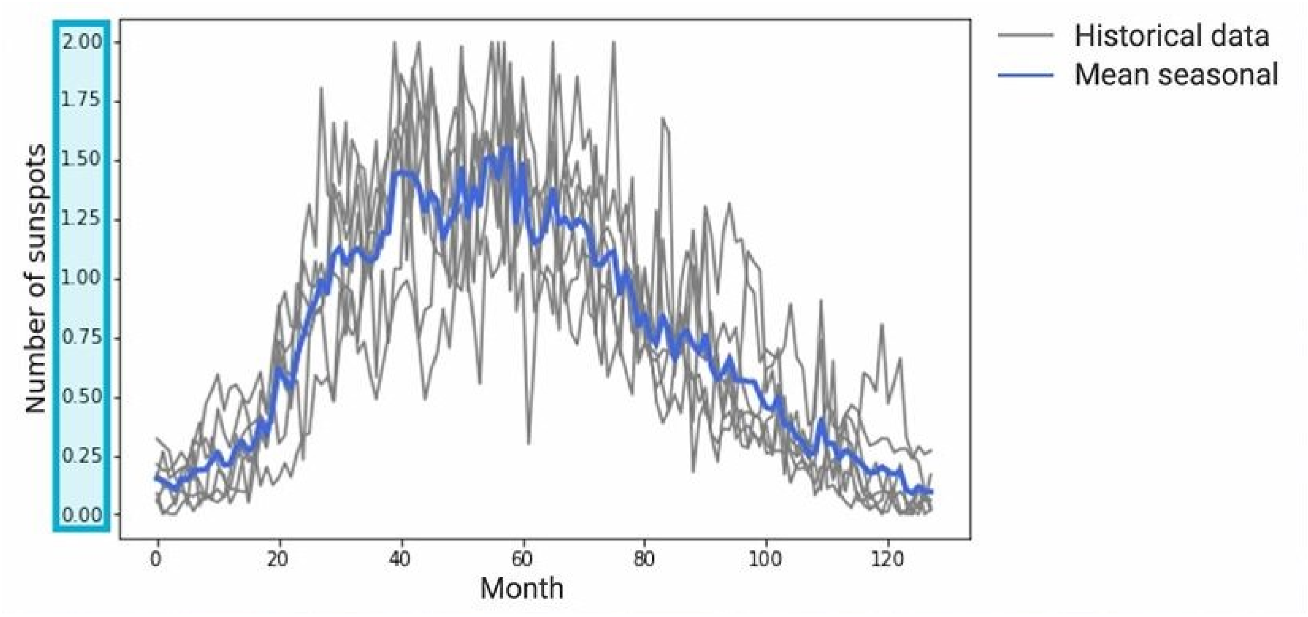


Then, you overlap those segments and take their average. At this point, you notice that the period of the data changes slightly from one season to the next. And so you cannot define a single overall period. You also notice there is significant variation in the amplitude of the oscillations. However, throughout the historical data, a) the minimum number of sunspots in every season is very nearly zero. And b) the peak is about twice the trend.

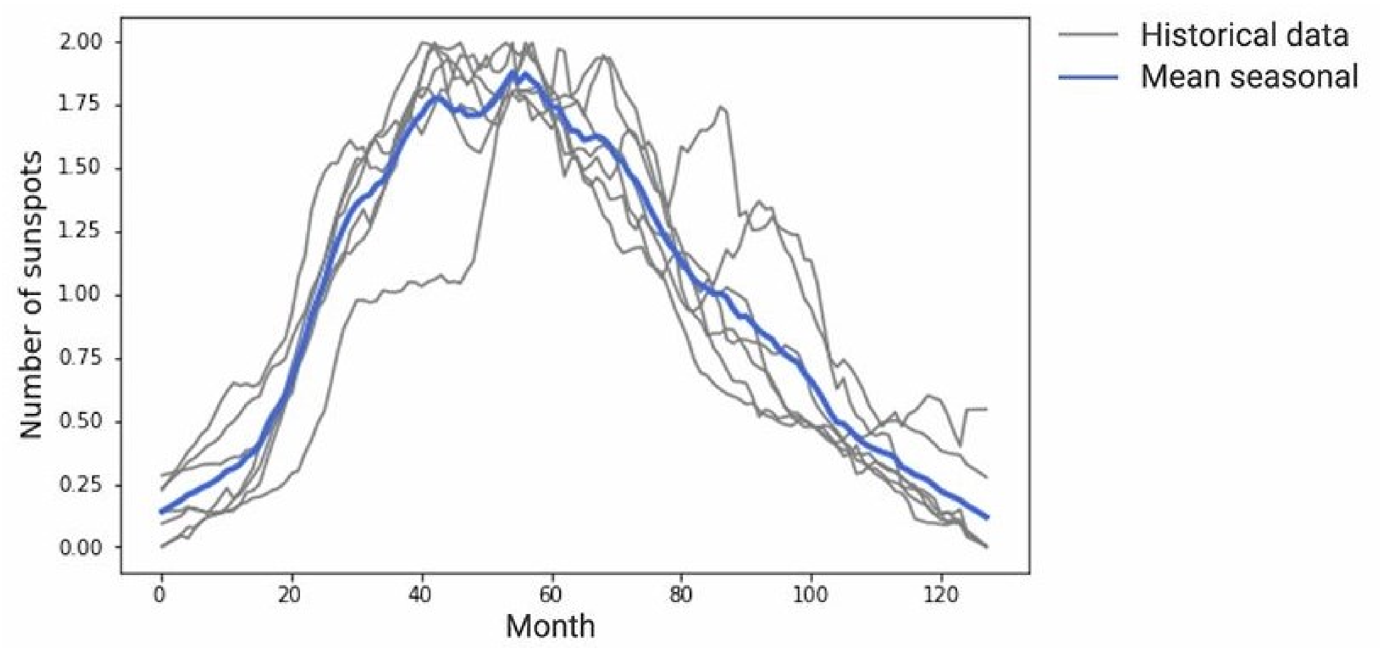
To incorporate these observations into your model, you can replace the additive decomposition, $\hat{y} = t + s$, with a multiplicative model, $\hat{y} = t \times s$, and scale the seasonal component so that it oscillates between zero and two. Then $\hat{y}$ will oscillate between zero and twice the trend as desired.

Decomposition (Part 3)

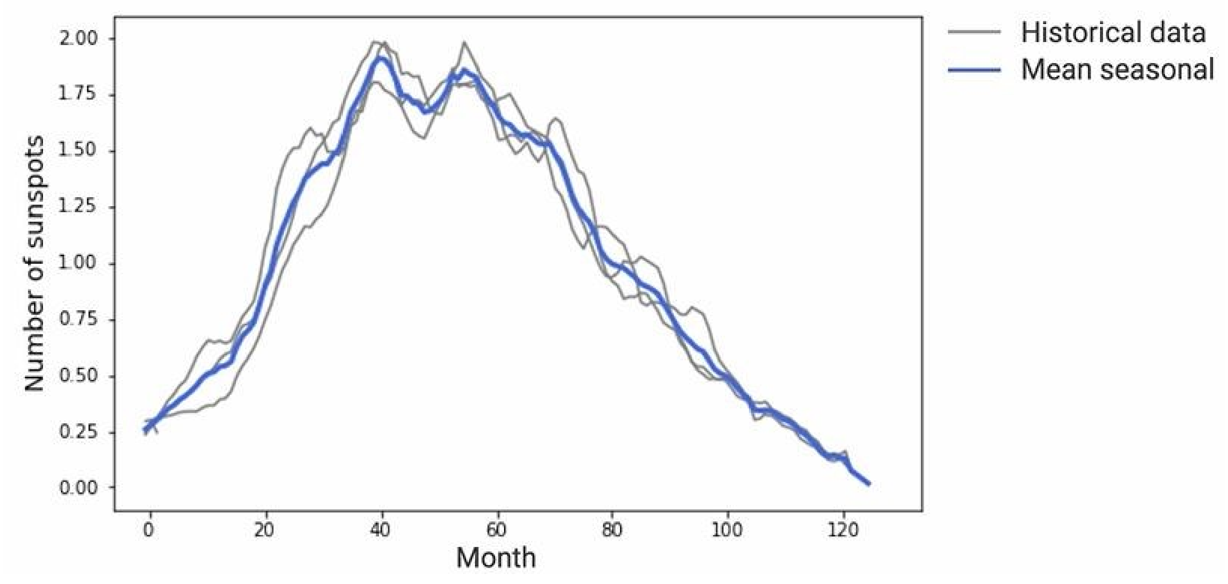
Suppose you divided the time series into seven seasons, each lasting about a decade. And you scaled each season so its minimum and maximum values are zero and two. Those seven lines are plotted here in gray.



Then you took the average of the seven gray lines. The resulting blue line is a bit too noisy to serve as your seasonal template. To deal with this, you can put the original data through a filter just like you did to extract the trend. This filter is shorter so it only smooths high-frequency noise without affecting seasonal behavior.

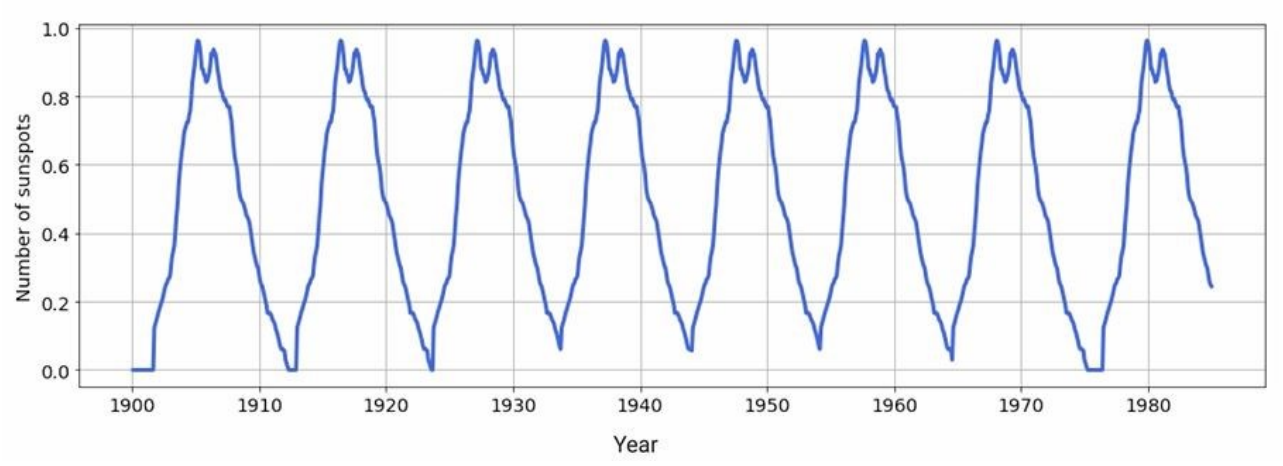


But there are some outliers in the data. So you remove those and try again.



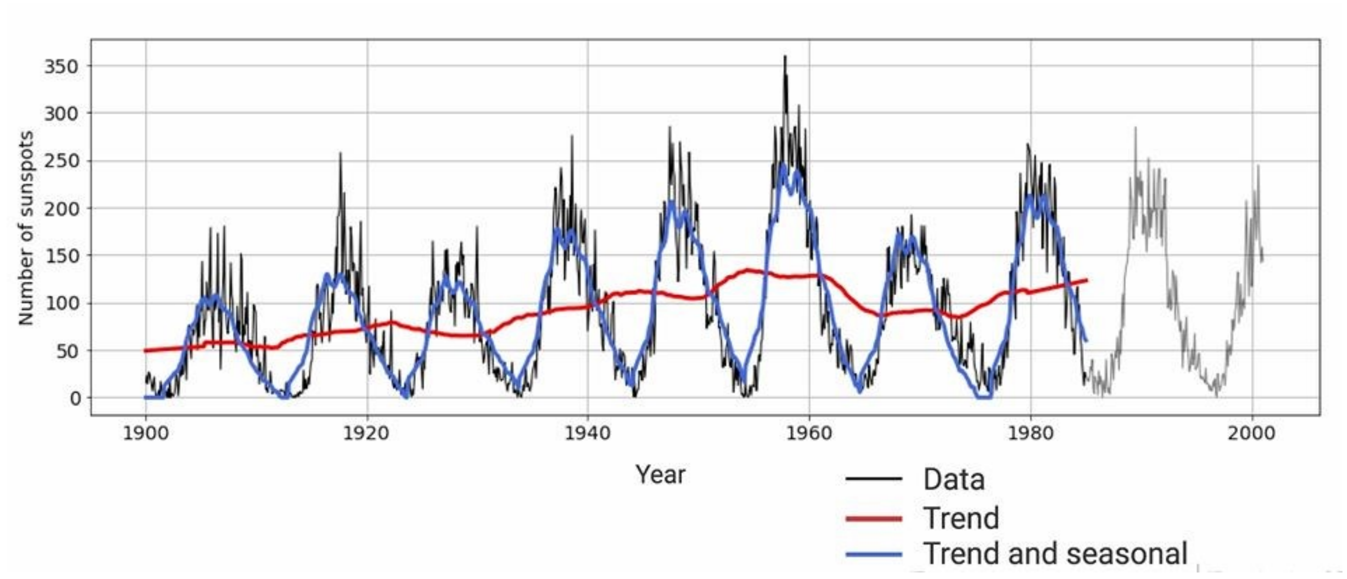
That looks much better. You are left with only three seasons, but the average now follows them quite closely.

Next, you can construct the seasonal signal by repeating the template that you have just constructed. And it looks like this.



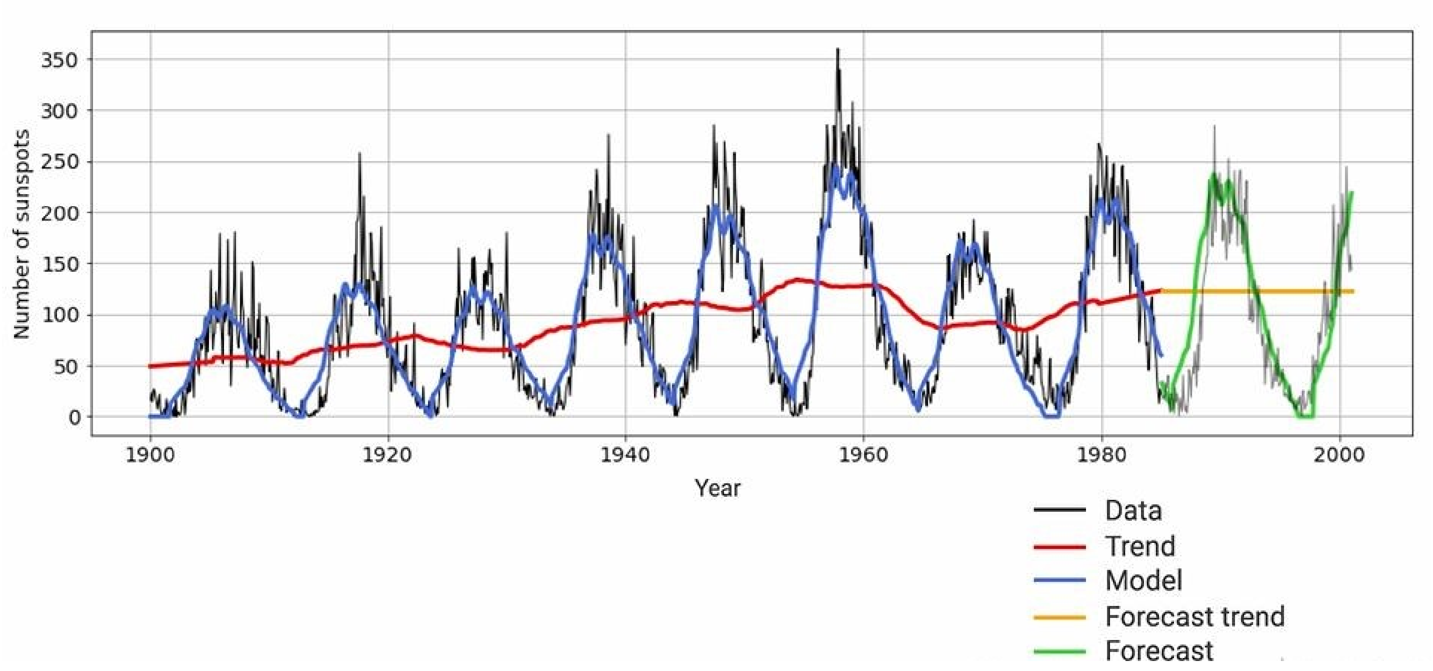
You can now plug this into the formula, $\hat{y} = t \times s$.

This is the result and it seems to follow the historical data pretty well.

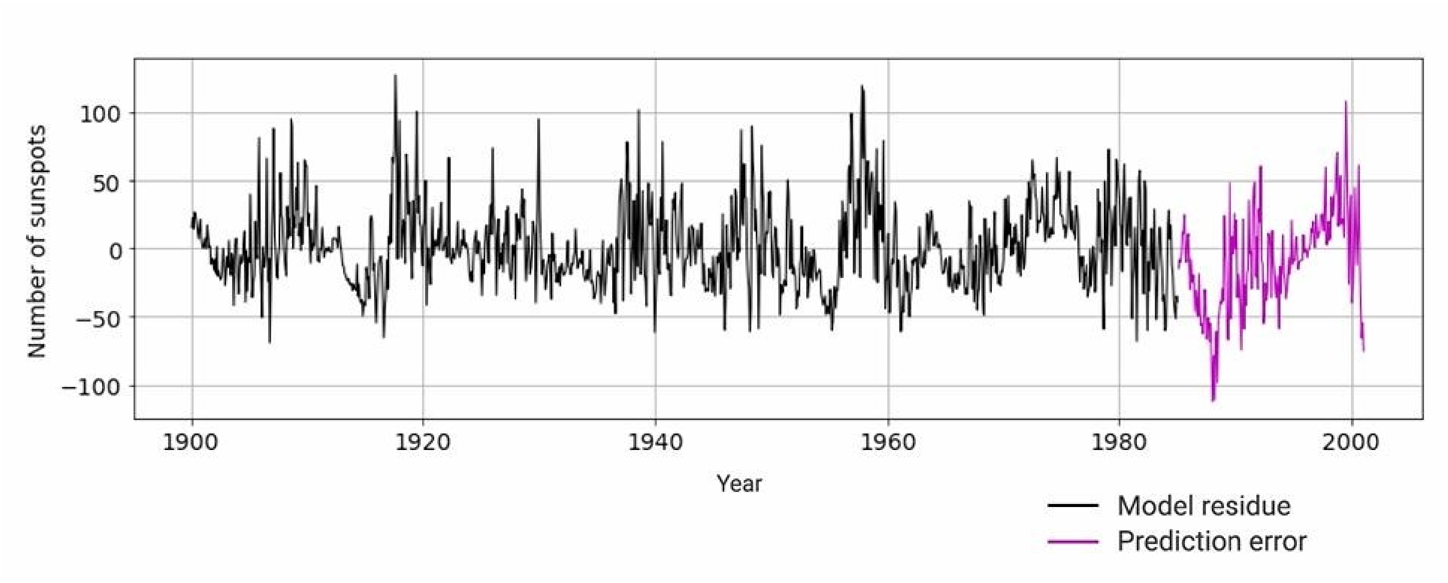


The final step in the process is to produce the residue of the model by subtracting $\hat{y}$ from y and to check whether this signal is stationary. This is done by checking the autocorrelation coefficients. If the residue is not stationary, this might suggest that some long-term trend, or seasonality, may have remained unmodeled. However, if the residue is stationary, then there is not much more structure that can be extracted using decomposition techniques.

You can use the model you have built to make a prediction. To do this, simply extend the seasonal component into the future. In this case, you multiply it by the extrapolation of the trend. The result is shown in green.



Imagine that 17 years have gone by and you have collected observations corresponding to the prediction you made back in 1984. Just as you computed the residue (as the historical data minus the model), you can compute the prediction error (as the observed data minus the forecast). Here is the prediction error, plotted in purple.



This table shows the two error metrics, mean absolute error (MAE) and the root mean squared error (RMSE) applied to both the residue and the prediction error.

	Residue	**Prediction Error**
MAE	22.7	29.0
RMSE	29.2	37.0

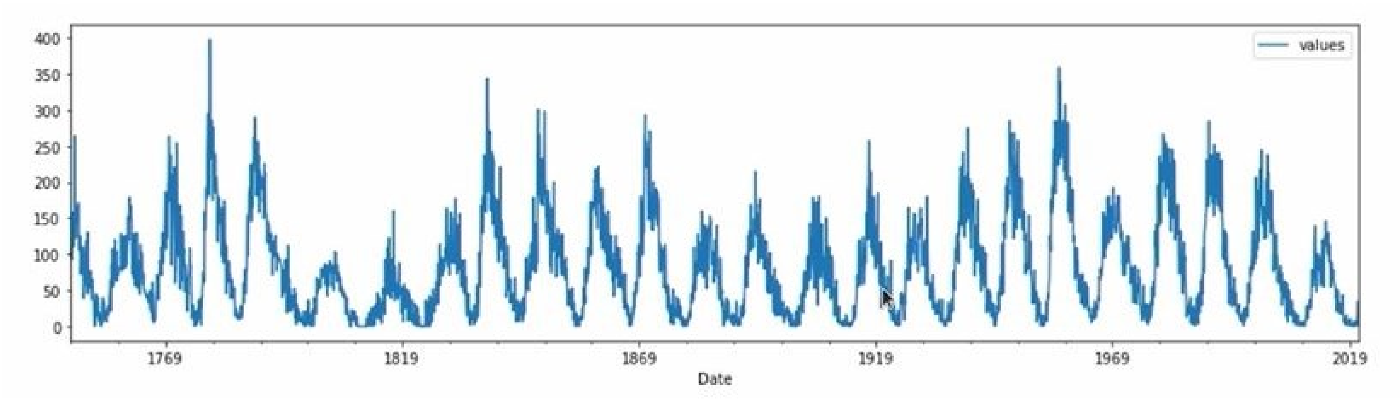
Both of these quantities represent estimates of the error, in units of number of sunspots predicted. Based on this, it is reasonable that the residue is smaller than the prediction error.

Decomposition (Part 4)

You can explore the code that was used to perform the time series decomposition of solar sunspots data.

To start, you load the data stored in the Sunspots.csv file and do some preprocessing: Set the index to a datetime series of dates, drop a couple of unwanted columns, and rename the last column to: values.

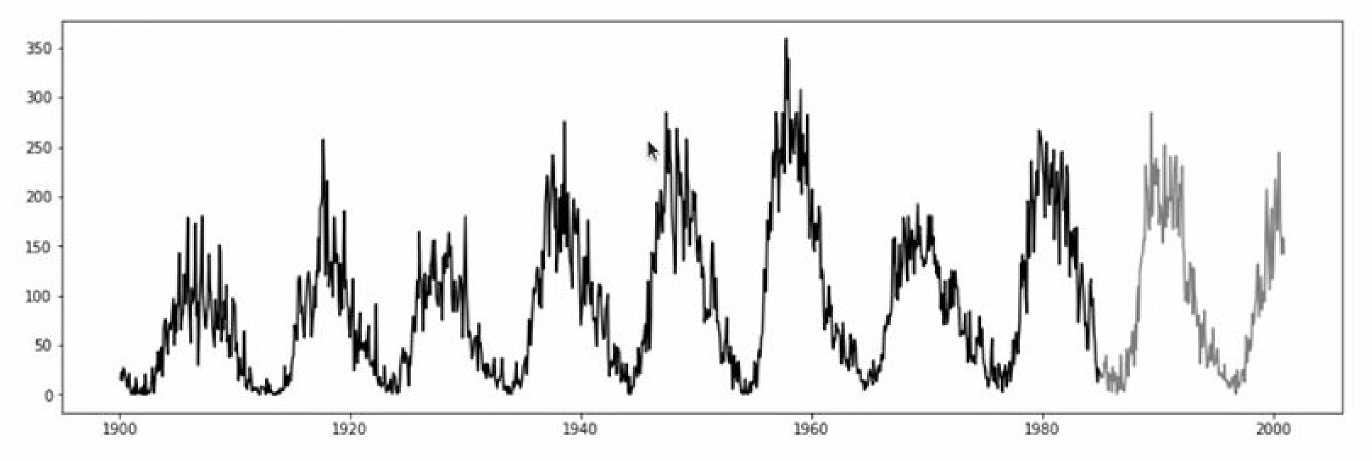
When plotted, this is what it looks like.



You can use this data for the prediction. In this instance, you want to use data from 1900 to 1984 to construct the model and then make a forecast from 1985 to 2000.

So you store that information into two series called y_hist and y_future.
y_hist = sunspots.loc['1900' : '1984', 'values']
y_future = sunspots.loc['1985' : '2000', 'values']

Next, you can plot the historical data, which appears in black from 1900 to 1984. From 1985 onward, that data is not available to the model training procedure.



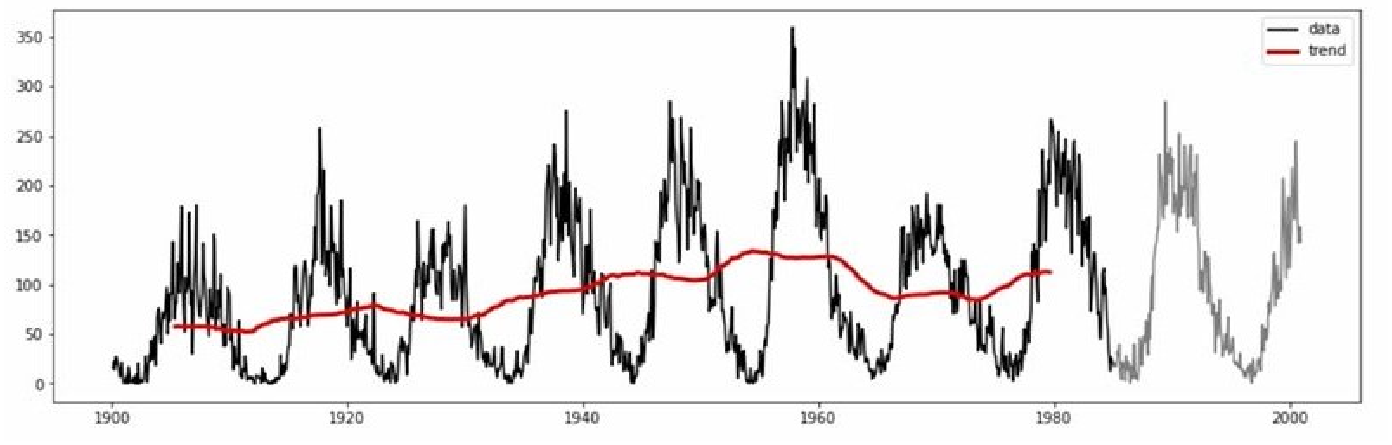
The first step in building the model is to extract the trend. You need a period to do that because you need to smooth the historical data with a filter with same length as the period. Since you know the period for solar sunspots is about 128 months, you can give the filter a length 129 (period+1).

```
period = 128

filt = np.ones(period+1)
filt[0] = 0.5
filt[-1] = 0.5
filt /= period
```

Then once you have the filter, you can run the historical data through the filter to extrapolate the trend.

```
trend = convolution_filter(y_hist, filt)
```

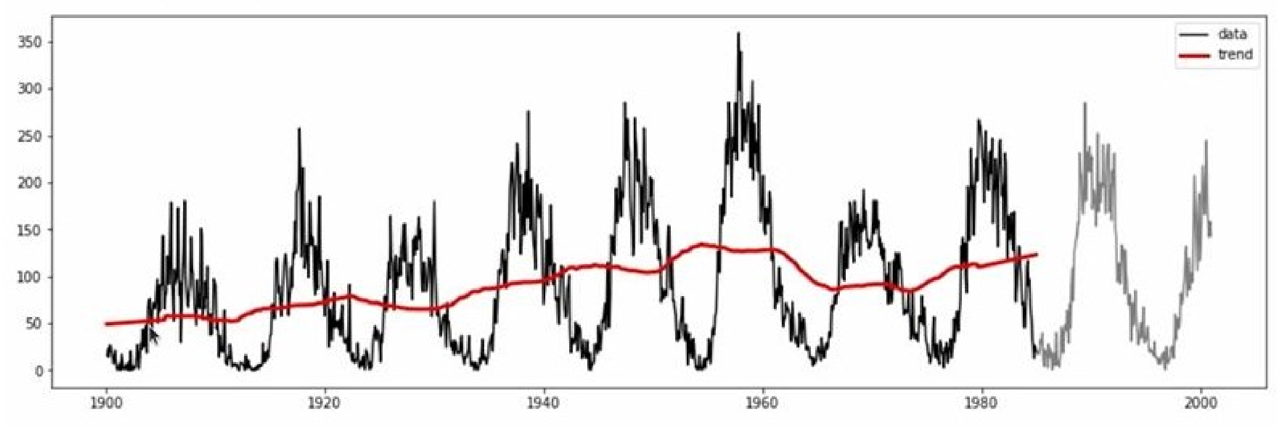
And if you plot that, this is what it looks like.



Now one problem here is that this trend is not going all the way to the beginning or to the end of the historical period. And you especially need it to reach the end. So you need to extrapolate this so that it reaches the end. And you do that with a function called extrapolate_trend().

trend = extrapolate_trend(trend, period + 1)

If you plot that, now the trend reaches both ends.

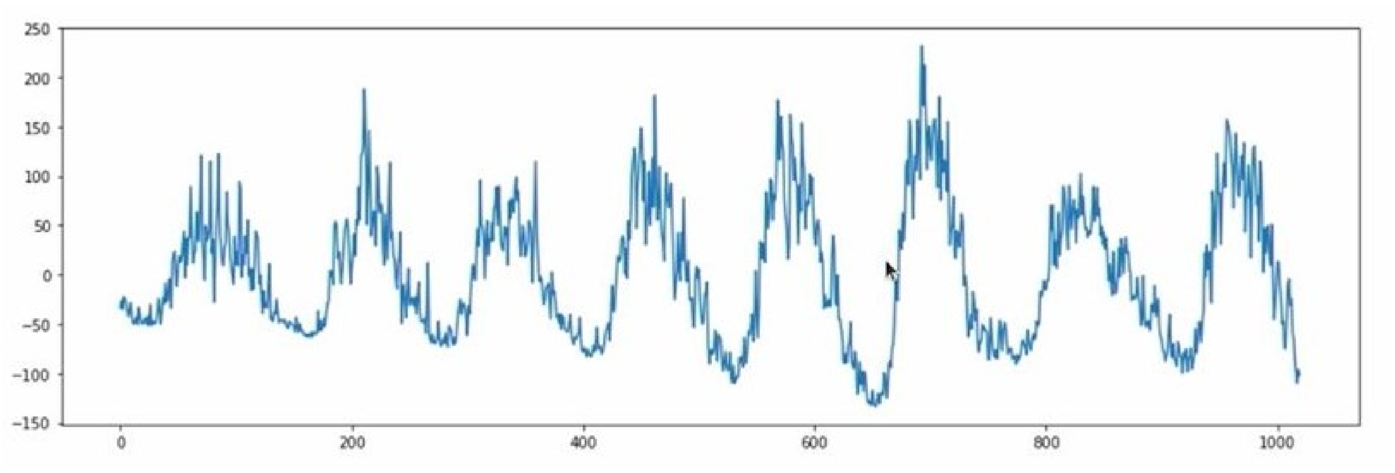


Next, you can compute the detrended data, which you get by subtracting the trend from the historical data.

detrended = y_hist - trend

detrended = detrended.to_numpy()

And this is what it looks like if you plot it.



The detrended data is similar to the historical data, except that the mean is at zero. The next step is going to be to split the historical data into seasons. And as you know, the seasons are of approximately 128 months in length. However, they are not always exactly 128 months.

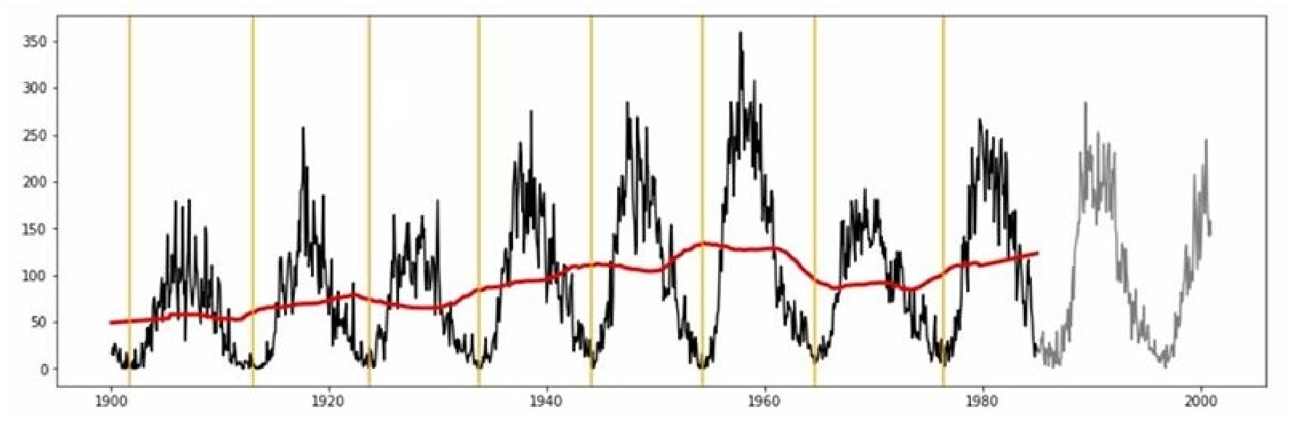
So you need to find, manually, the beginning and end of each season.

Having done that manually, you find these numbers.
lows_index = [20, 155, 284, 405, 529, 650, 775, 917]
lows = y_hist.index[lows_index]
lows

Passing those indices through the index, you get the dates for the low points. In this instance, the first low point is on September 30, 1901. The second is December 31, 1912. And so on.

You can plot that with the **axvline()** function to create vertical lines through the low points.

```
plt.figure(figsize=(16,5))
plt.plot(y_hist, 'k')
plt.plot(y_future, 'k', alpha=0.5)
plt.plot(trend, 'r', linewidth=3)
for low in lows:
        plt.axvline(x=pd.to_datetime(low),color='orange')
```



There are essentially seven seasons.

Next, you need to extract the detrended data for each of these seasons and find an average.

At this point, it is important to have a common period for every season so you can store the data in a matrix. You compute that period by taking the difference of the low index, which is the number of months in each season. Take the average of that, then round that number, and cast it as an integer.

```
period = int(np.round(np.mean(np.diff(lows_index))))
```

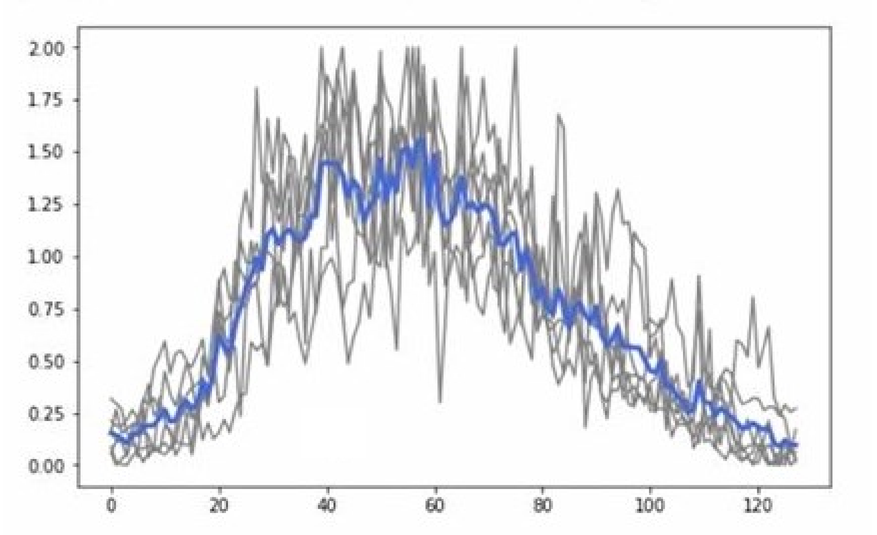
And that gives 128 – the number of seasons is the number of orange bars, minus one.

```
num_seasons = len(lows)-1
```

And now you are ready to extract the mean of the seasons.

```
seasonals = np.empty((period,num_seasons))
for p in range(num_seasons):

    s = detrended[lows_index[p]:lows_index[p]+period]
    s = 2*(s-np.min(s))/(np.max(s)-np.min(s))
    seasonals[:,p] = s

mean_seasonals = seasonals.mean(axis=1)
```

And when you plot that, this is what it looks like.



The seven gray lines are the seasons and the blue is the average. But this average is pretty jagged, so you should smooth it out. To smooth it out, you pass it again through a filter, a convolution filter, that is 9 months long.

```
filt_size = 9
filt = np.repeat(1.0 / filt_size, filt_size)
seasonals = np.empty((period, num_seasons))
for p in range(num_seasons):

    s = detrended[lows_index[p]:lows_index[p]+period]
```

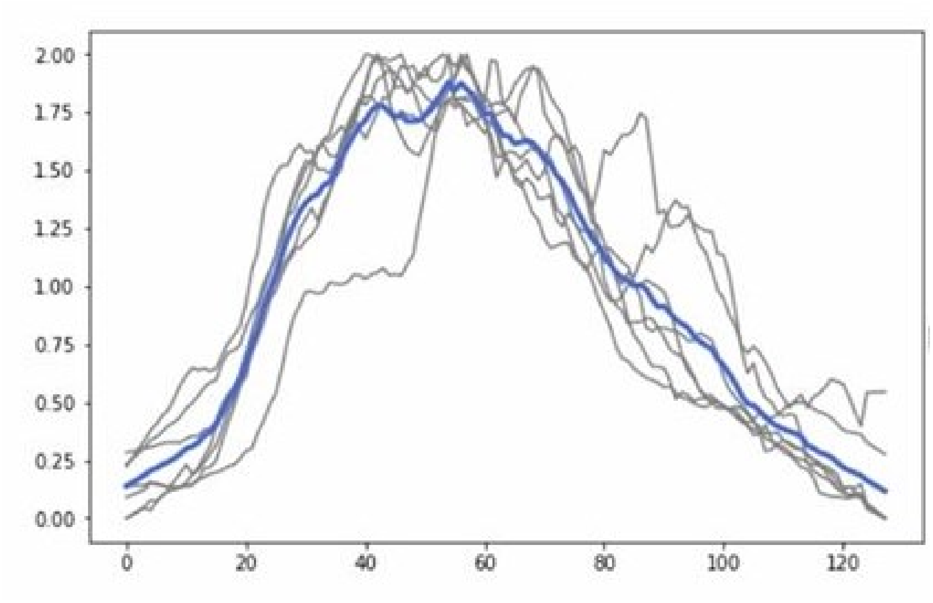
s = convolution_filter(s, filt)

s = extrapolate_trend(s, filt_size)

s = 2*(s-np.min(s))/(np.max(s)-np.min(s))

seasonals[:,p] = s
mean_seasonals = seasonals.mean(axis=1)

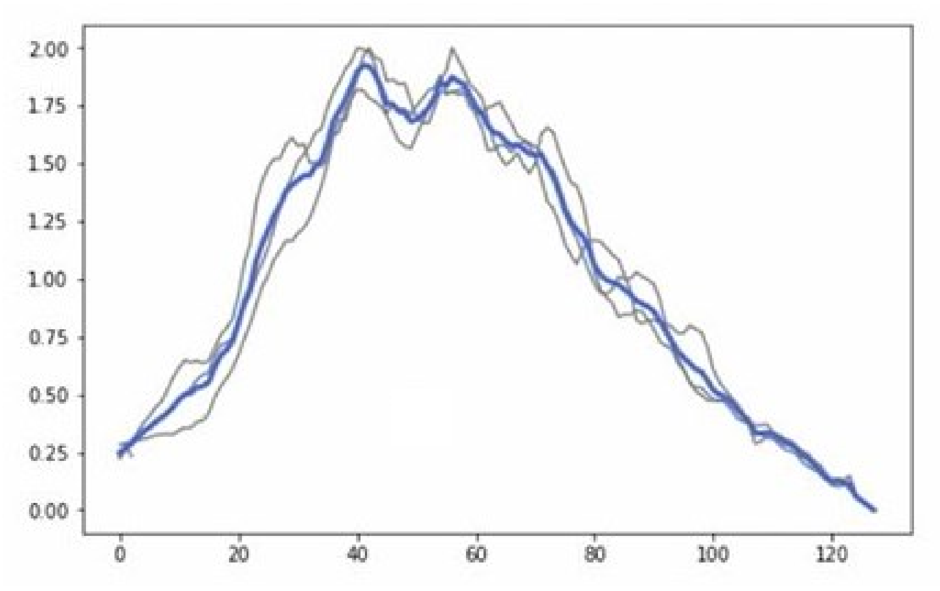
And then you plot it again.



You get the same number of seven gray lines, but the average looks a lot smoother than before. However, you have some outliers.

So as your next step, you remove the outliers. Since the outliers are seasons number zero, one, five, and six, you only keep seasons two, three, and four and then take their mean again.
seasonals = seasonals[:,[2,3,4]]
mean_seasonals = seasonals.mean(axis=1)

And then you plot it once more.

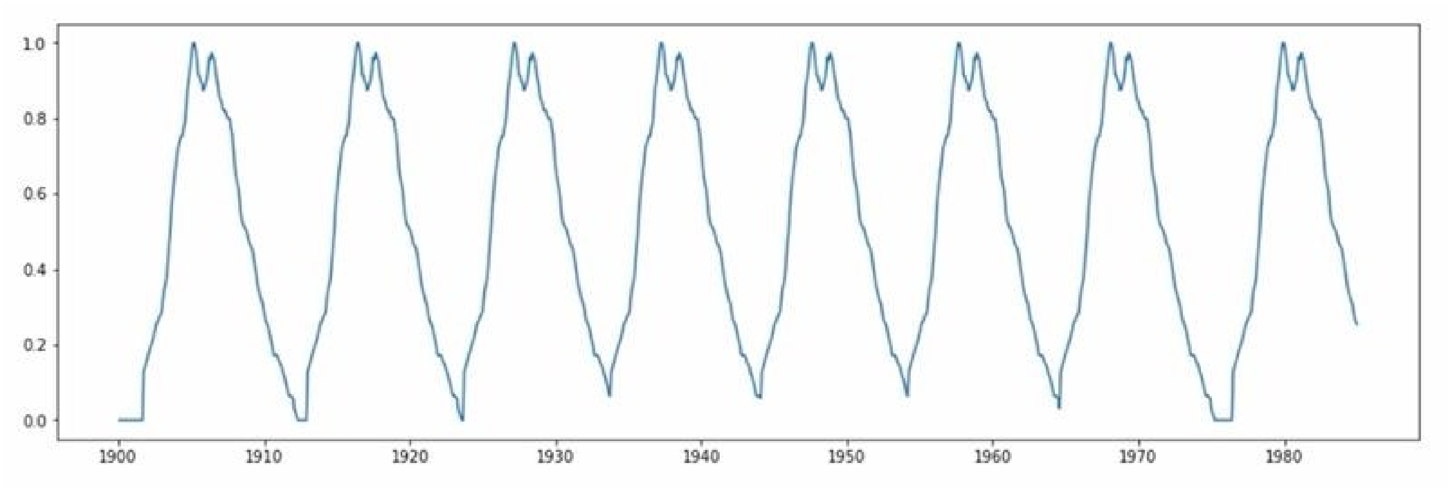


So this is ready to serve as your template for each season.

Next, you build the complete seasonal template. You construct a series whose index is the same as the historical dates. And you set all the data to zero. And then copy into that index the mean_seasonals, as much as will fit.

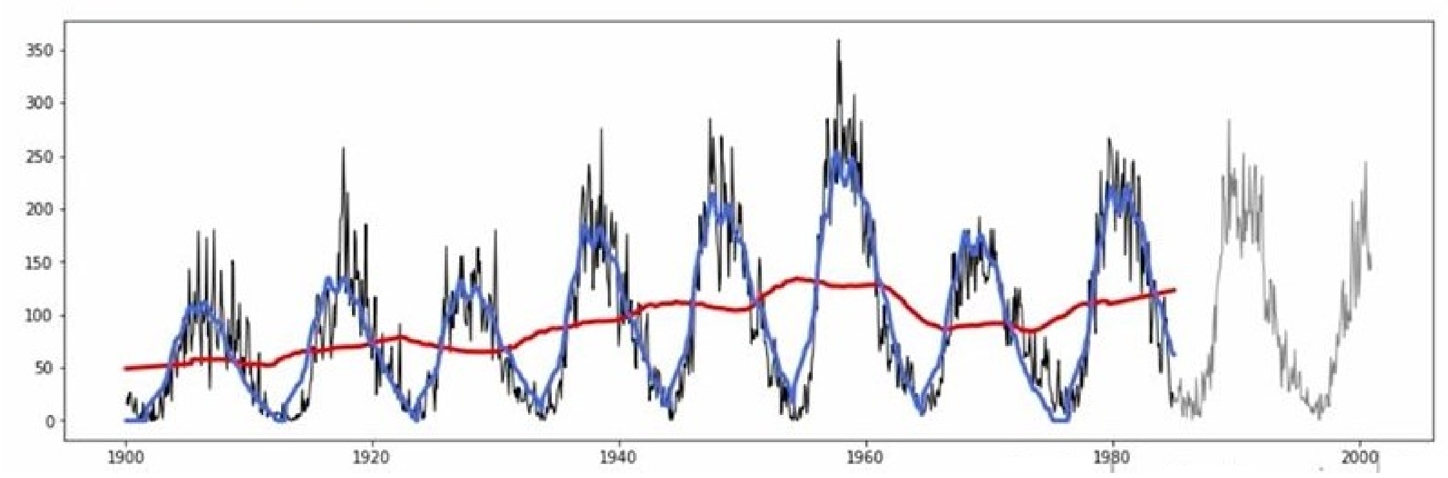
```
seasonal = pd.Series(index=y_hist.index,data=0)
for low in lows_index:

    if low+period<len(seasonal):

        seasonal[low:low+period] = mean_seasonals

    else:

        seasonal[low:] = mean_seasonals[:len(seasonal)-(low+period)]
seasonal = seasonal / np.max(seasonal)
```

When plotted, this the result.



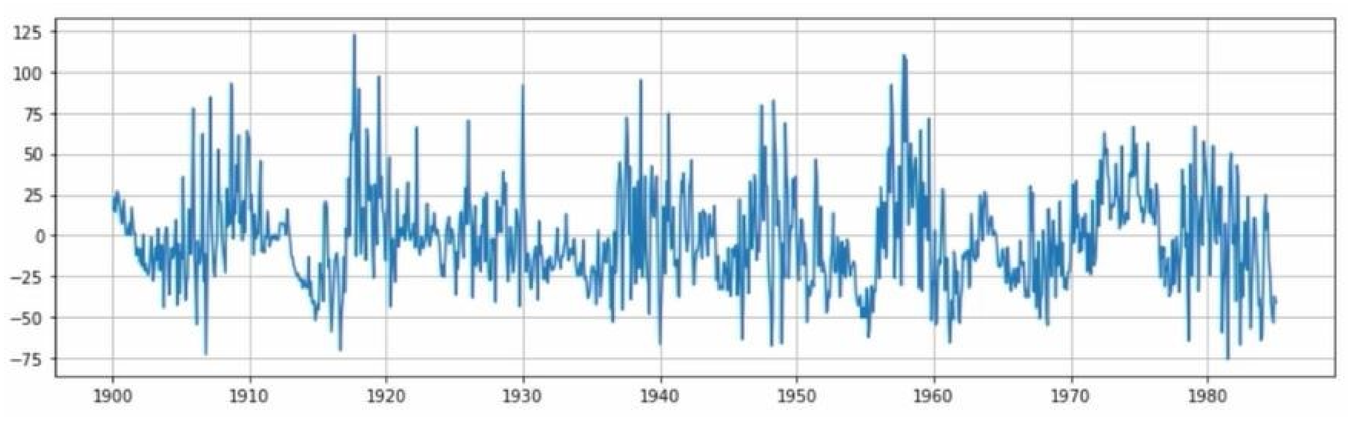
Last, to finalize the model, you take that template and multiply it by the trend, and then multiply that by two.
season_and_trend = 2*trend*seasonal

And this is your complete model, once plotted. It shows up in blue and it does look like it is following the data fairly well.



You can quantify how good this model is by computing the residue, which is the difference between the historical data and the model.
residue = y_hist - season_and_trend

So you computed that and then you can plot it.

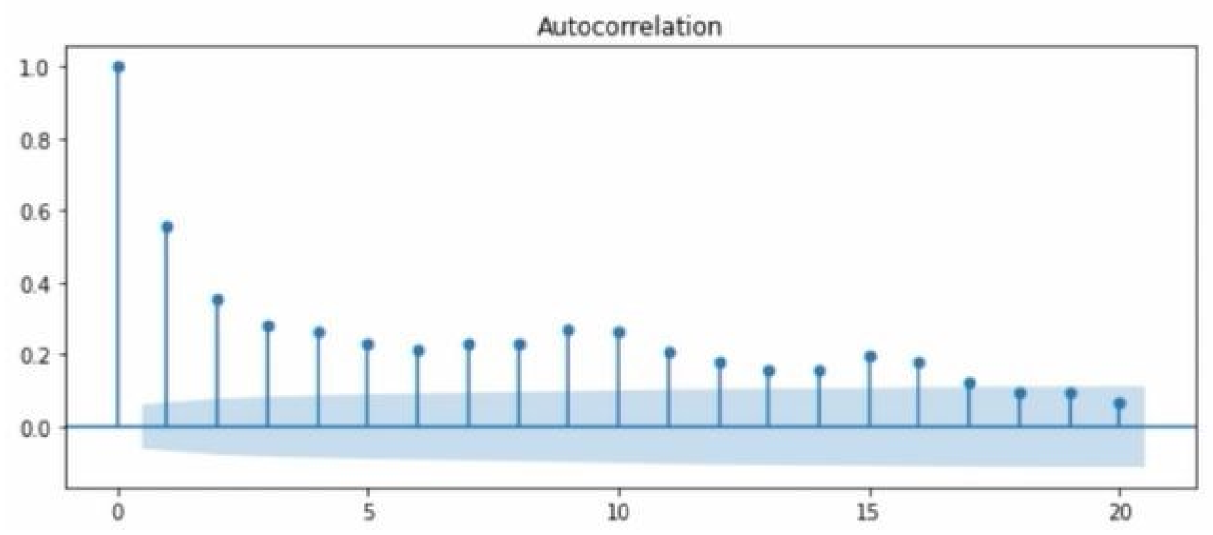


And you would like for this residue to be stationary, so you need to do autocorrelations to see if it is stationary. But first, what is the size of the residue? You can quantify that in terms of the mean absolute error (MAE) or the root mean squared error (RMSE).

MAEm = np.abs(residue).mean()
RMSEm = np.sqrt(np.square(residue.mean)))
MAEm, RMSEm

And as a result, you obtain 22.72 for the MAE and 29.21 for the RMSE.

Now back to stationarity, so you plot the autocorrelation function next.

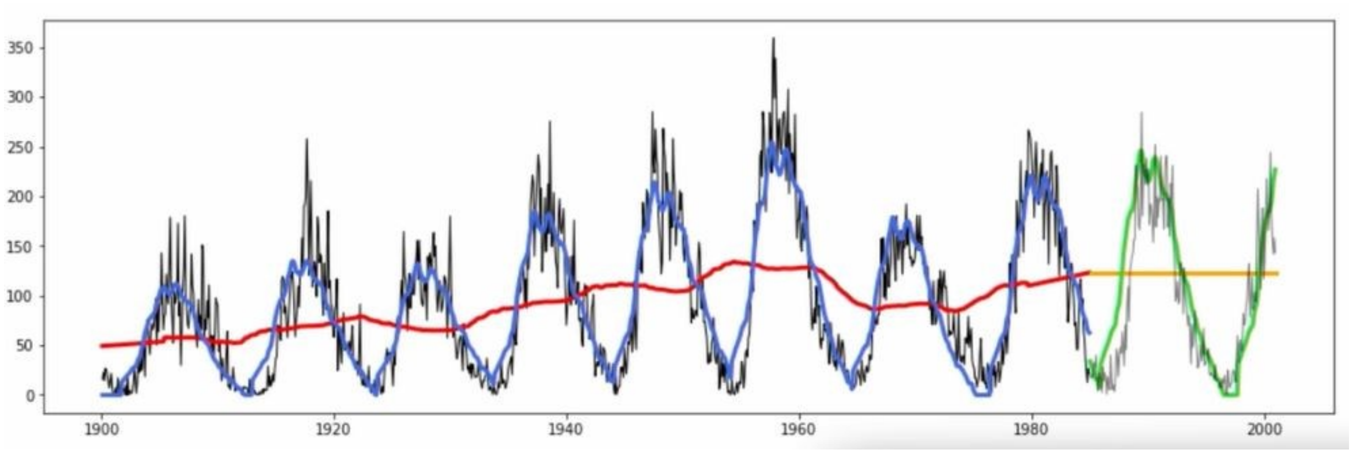


It does not look too stationary. If the residue were stationary, you would expect it to decay or maybe just cut off at some point. But it is not decaying exponentially, which is indicative of non-stationarity.

However, in this instance, you forge ahead and produce a forecast with this model. So to start, you project the trend forward, then go step-by-step through the future and copy in the value from the seasonal. With that seasonal projection, you then apply the model to the forecast.

```
yhat_trend = pd.Series(index=y_future.index, data=trend[-1])
yhat_seasonal = pd.Series(index=y_future.index)
for i in range(len(yhat_seasonal)):

    yhat_seasonal[i] = seasonal[-(2*len(mean_seasonals)-i)]
yhat = 2*yhat_trend*yhat_seasonal
```

Once plotted, this is the projection into the future.

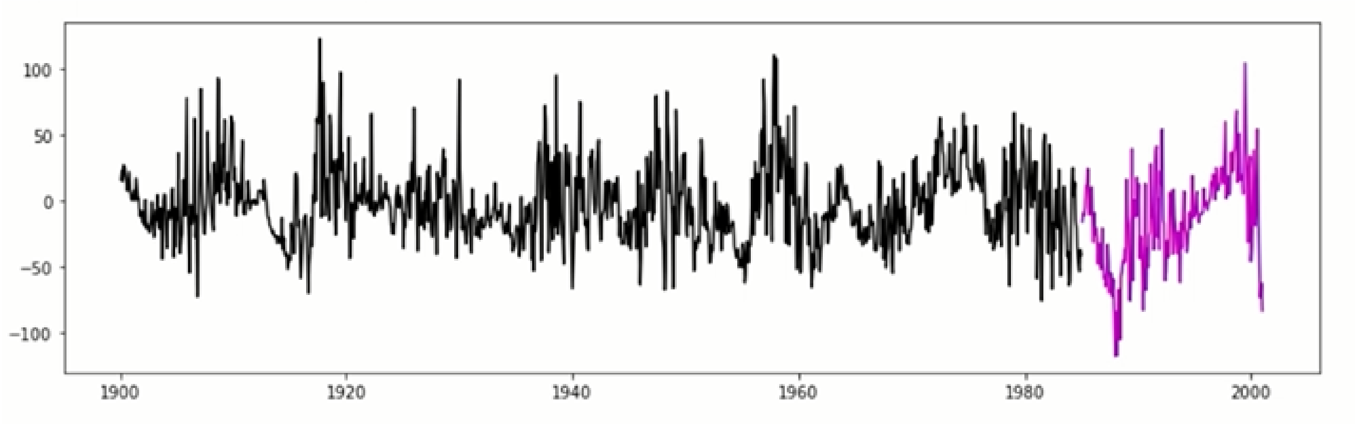


Next, you compute the error for this forecast. The error is the future data minus the model, minus the forecast. And this you could only do after you reached the year 2001 where you have access to this data.

But for now, assume that is the case.

pred_error = y_future - yhat

Once you computed the prediction error, you can put it into the same plot as the residuals. In this instance, it has been plotted in pink.



Next, you compute the size of the prediction error. You apply the same metrics, the MAE and the RMSE.

MAEp = np.abs(pred_error).mean()

RMSEp = np.sqrt(np.square(pred_error).mean())

MAEp, RMSEp

And you get 30 and 39, respectively.

Now if you compare that to the metrics for the residual, which are 22 and 29, it is evident that they are quite a bit larger and that is very reasonable. It is like saying that the error in the testing dataset is larger than the error in the training dataset. Which is what you would expect, since the training was done, in a way, to minimize the error. So this is all very reasonable.

ARMA (Part 1)

The autoregressive moving average (ARMA) family of models are designed to capture the time invariant structure exhibited in stationary time series.

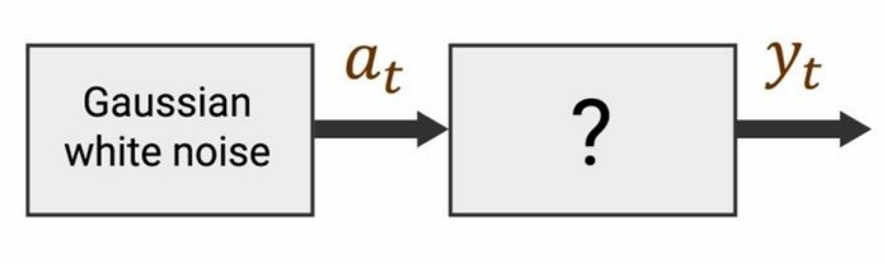
Once this has been done, you will be left with a final residue of structureless and useless white noise.

Two of the most useful models for representing stationary processes are:

- The **moving average (MA)** process (q)
- The **autoregressive (AR)** process (p)

Here the q and the p are the orders of the MA and AR process, respectively.

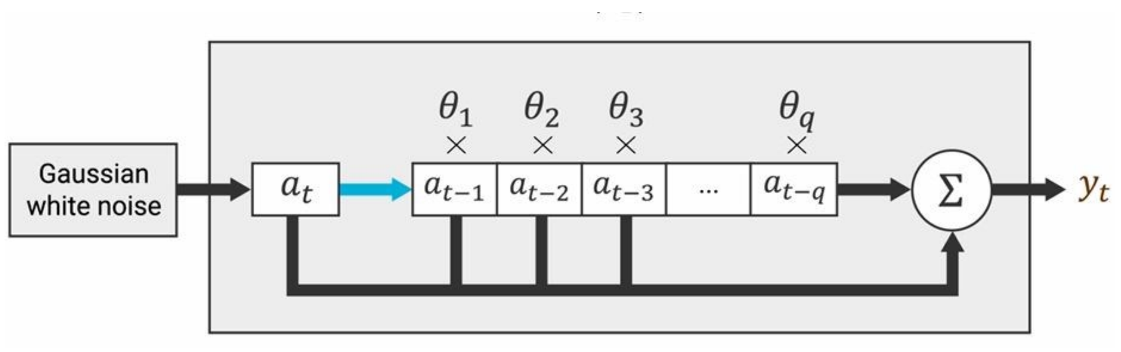
So, for example, MA(3) is a moving average process of order three.



Both the MA and AR models regard the stationary process as the output, y_t of a procedure that is fed with Gaussian white noise, which you denote by a_t.

MA(q)

For a moving average process of order q, the white noise is put through a filter with coefficients $\theta_1$through θ_q.

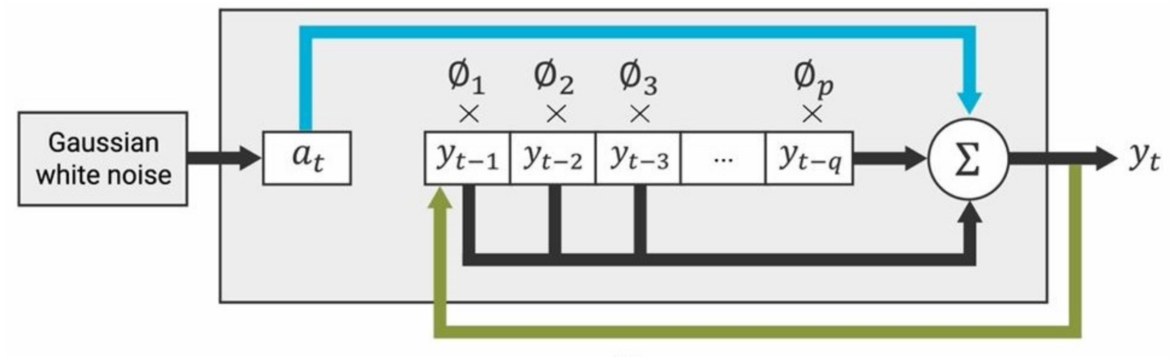


At each time, t, the output, y_t, is computed by multiplying the θ coefficients by the past q-values of the input noise and then adding a_t. This is summarized in the formula for the moving average model.

$$y_t = a_t + \sum_{j=1}^{q} \theta_j a_{t-j}$$

AR(p)

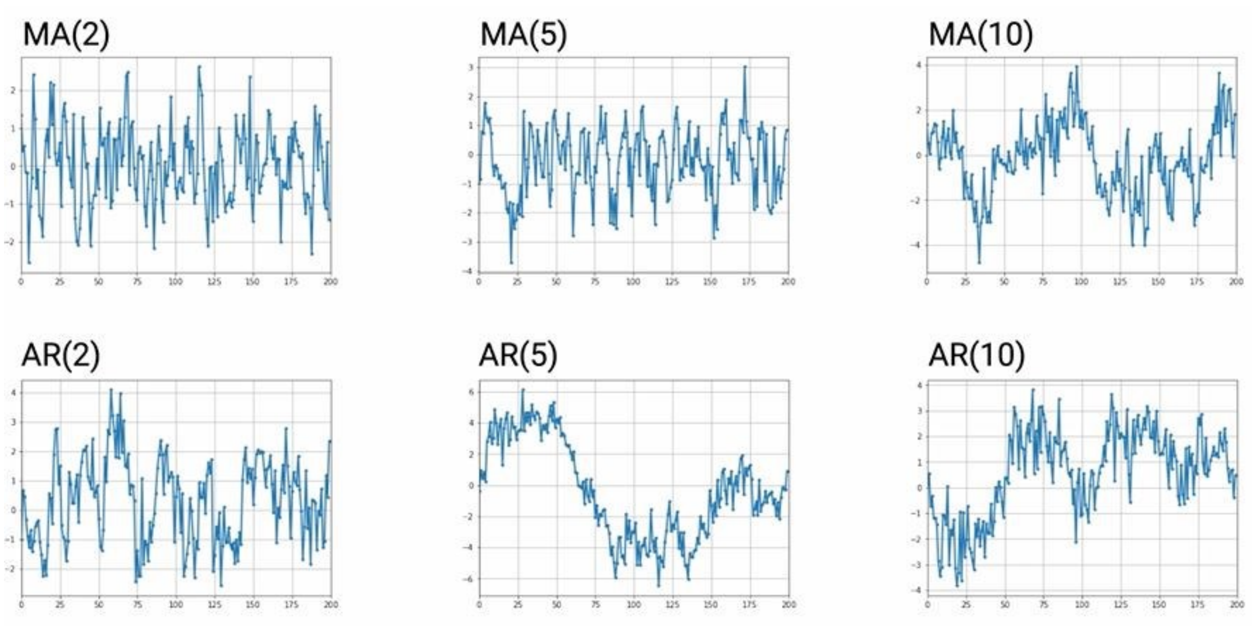
The autoregressive process of order p is a little different.



Instead of θ, here you use the letter $\emptyset$ to represent the p coefficients of the model. And instead of feeding the white noise to the coefficients, you bring back the output y_t and use that. Hence the name autoregressive, the procedure feeds back on itself. The mathematical formula for the AR(p) process is shown below.

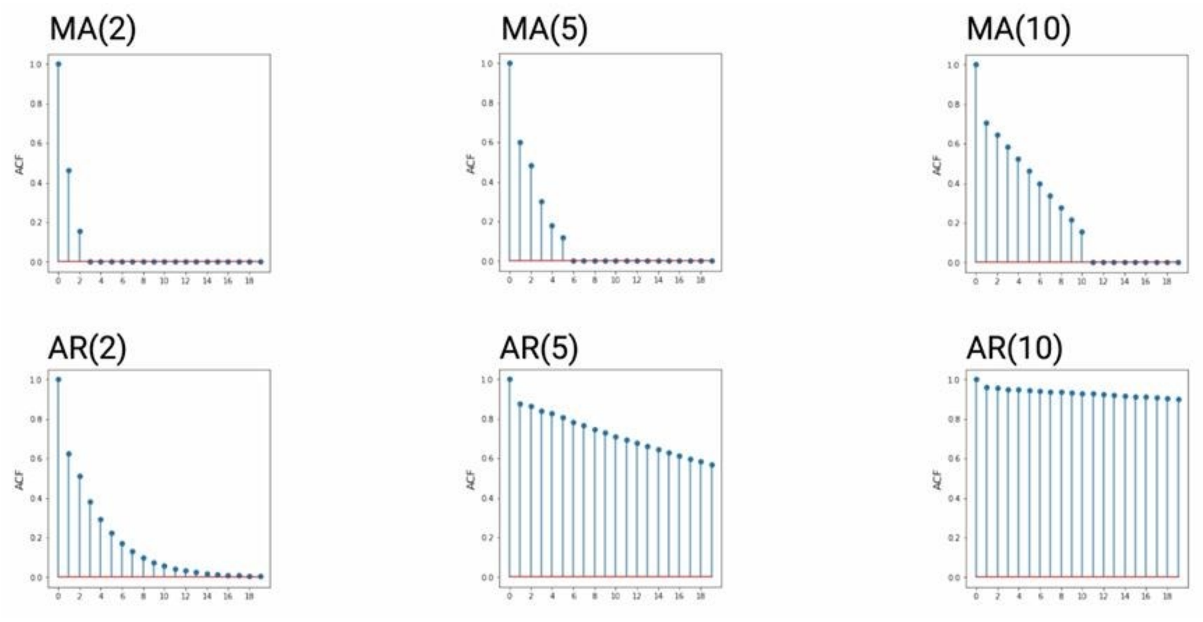
$$y_t = a_t + \sum_{j=1}^{p} \emptyset_j y_{t-j}$$

Consider six samples of moving average and autoregressive processes of order 2, 5, and 10, respectively.



A 0th order MA or AR process is just white noise. The higher the order, the more structured the signal becomes, generally speaking. However, just looking at these samples, it is difficult to guess whether the time series is MA or AR, let alone the order of the process.

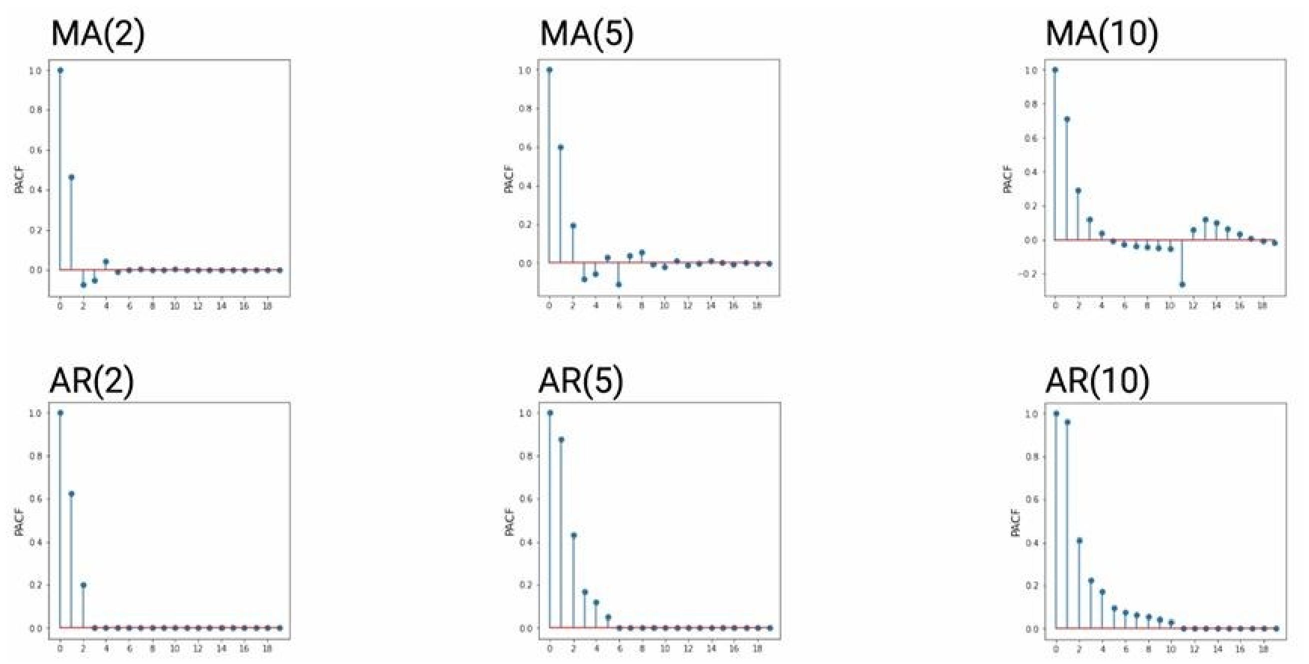
So you need better tools for analyzing stationary processes. These plots (below) show the autocorrelation functions for each of the processes whose samples you just saw.



Notice something interesting about the top row. The autocorrelation functions for MA processes begin with a one like all autocorrelation functions, and then is followed by as many non-zero entries as the order q of the process. And beyond that, it is all zeros.

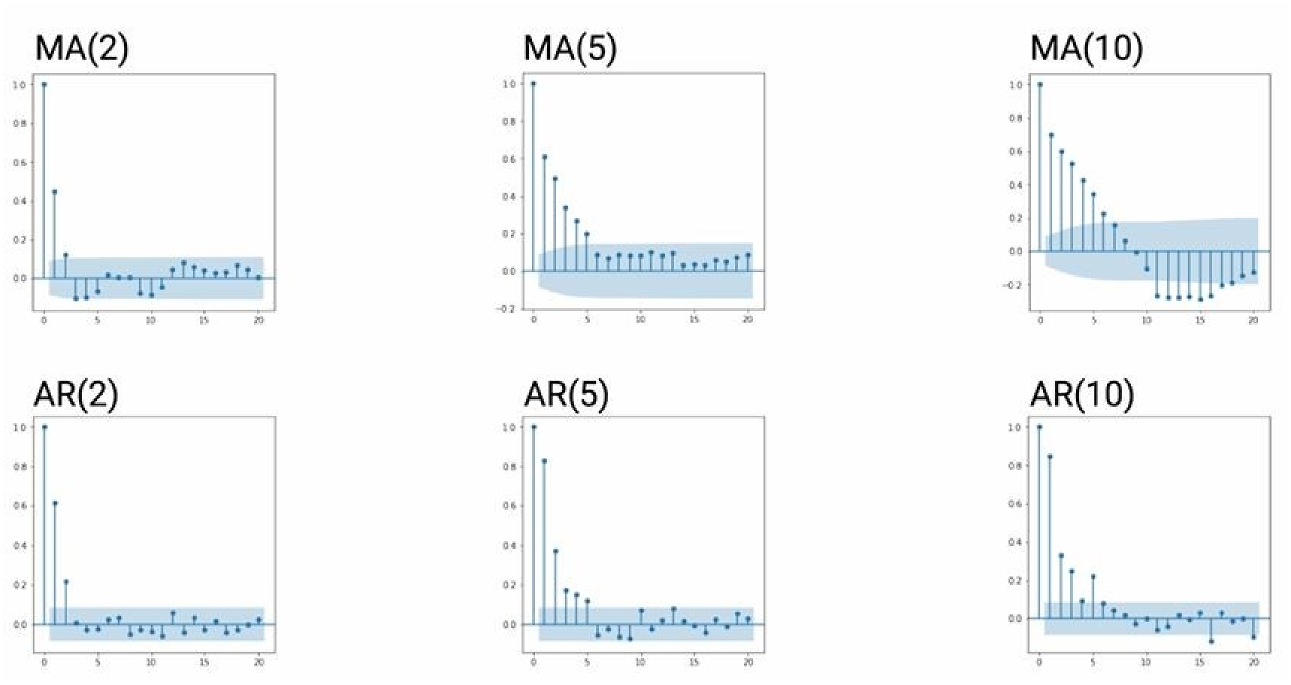
This observation gives you a direct method for identifying the order of an MA process. Just look at the lag of the largest non-zero entry in the ACF. The bottom row shows that this does not work for autoregressive processes. In this case, the 0th entry is again one. And after that the entries decay more or less exponentially without ever reaching absolute zero.

These next plots show a different analytical tool called the partial autocorrelation function, or PACF.



Inspecting the PACF, you notice that the situation is now the reverse. The order p of the AR process shows up as the largest lag with a non-zero entry. So the AR(2) process has a non-zero PACF for lag two and zeros beyond that. And, similarly, for AR(5) and AR(10).

So these are two of the tools that can help you to determine the orders p and q of the AR and MA processes. The problem is that you do not have these plots. All you have is the data from which you can calculate sample autocorrelation and partial autocorrelation functions.



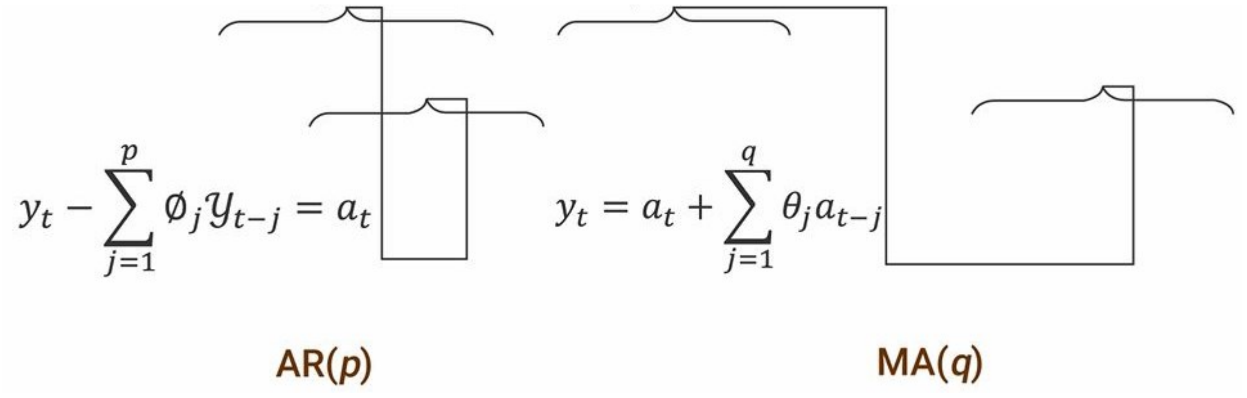
The top row shows sample autocorrelation functions for the MA processes. And in the bottom row, the sample PACF for the AR processes. For MA and AR processes of order 2 and 5, counting the number of non-zero coefficients before the plot dips into the blue region provides the correct answer.

However, this is not the case for MA(10) and AR(10). If you were to guess the order of these processes on the basis of these plots, you would have guessed MA(6) and AR(5). For this reason, you will rarely encounter AR and MA models of such high order.

ARMA (Part 2)

Having understood the autoregressive (AR) and moving average (MA) processes, consider the ARMA process, which is a simple combination of these two elements.

$$y_t - \sum_{j=1}^{p} \emptyset_j y_{t-j} = a_t + \sum_{j=1}^{q} \theta_j a_{t-j}$$


$y_t - \sum_{j=1}^{p} \emptyset_j y_{t-j} = a_t$
$y_t = a_t + \sum_{j=1}^{q} \theta_j a_{t-j}$
AR(p)
MA(q)

Here, on the bottom left, is the formula for the AR process of order p. The summation is on the left-hand side of the formula in order to gather all of the y's on the left and the a's on the right. Then, on the bottom right, is the formula for the MA process of order q. Again, the y's are on the left and the a's are on the right.

An ARMA(p,q) process is obtained by taking the left-hand side of the AR(p) and equating it to the right-hand side of the MA(q). ARMA(p,q) is a strictly larger class of models than both AR and MA, since you can get any AR model by setting $q = 0$ and any MA model by setting $p = 0$. The ARMA family of models is very general. It can replicate any zero mean stationary process to a desired precision by setting p and q large enough and carefully selecting the $\emptyset$ and θ coefficients.

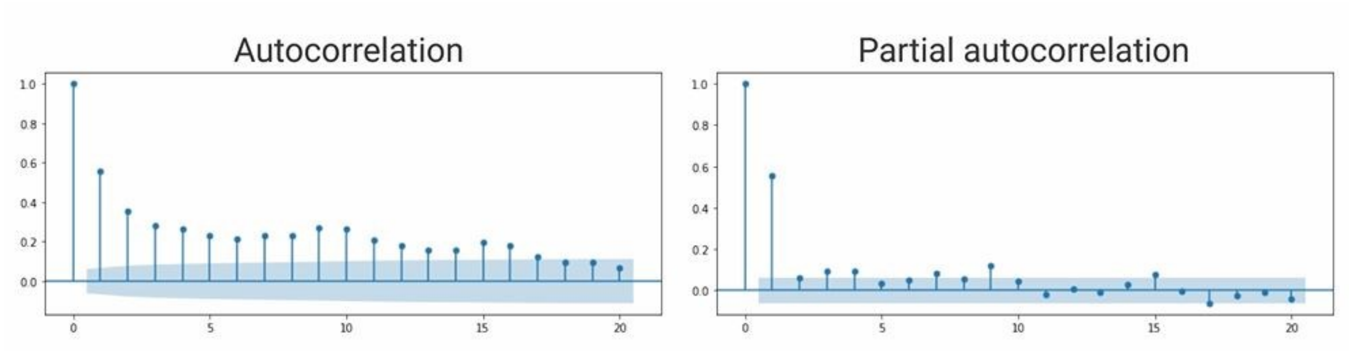
ARMA (Part 3)

The steps for modeling and forecasting time series data with autoregressive moving average (ARMA) are as follows:

1. Check that the signal is stationary.
2. Use the sample autocorrelation function (SACF) and sample partial autocorrelation function (SPACF) to select p and q
3. Compute the θ and $\emptyset$ coefficients of the model MA(q) and AR(p) parts
4. Compute the residuals
5. Make a forecast

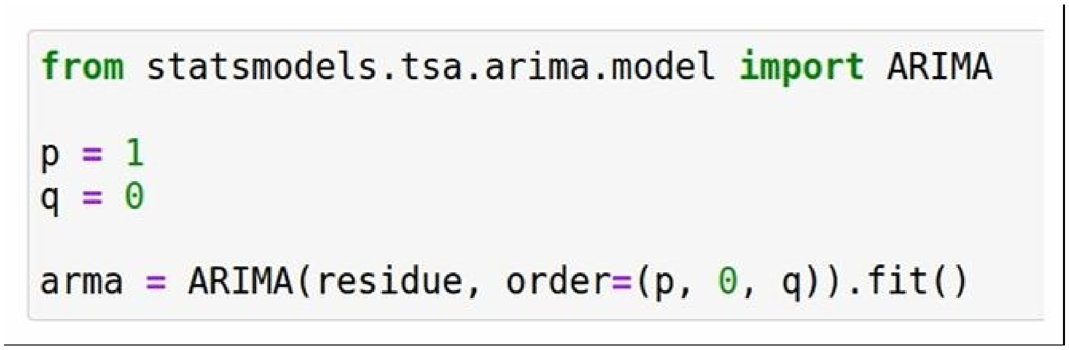
Previously, you saw the autocorrelation function (ACF) of the residue from the model of the sunspots data. And the conclusion was that the residue was likely non-stationary.

Next, you are going to apply the ARMA model to this data. Below, you review both the autocorrelation function and partial autocorrelation function (PACF) of the residue.



The PACF decays very quickly and suggests that you should consider an AR model of order one.

To build the model, you can use the ARIMA class from statsmodels. The constructor for the ARIMA class takes the time series to be modeled and the order of the model. It uses this code:



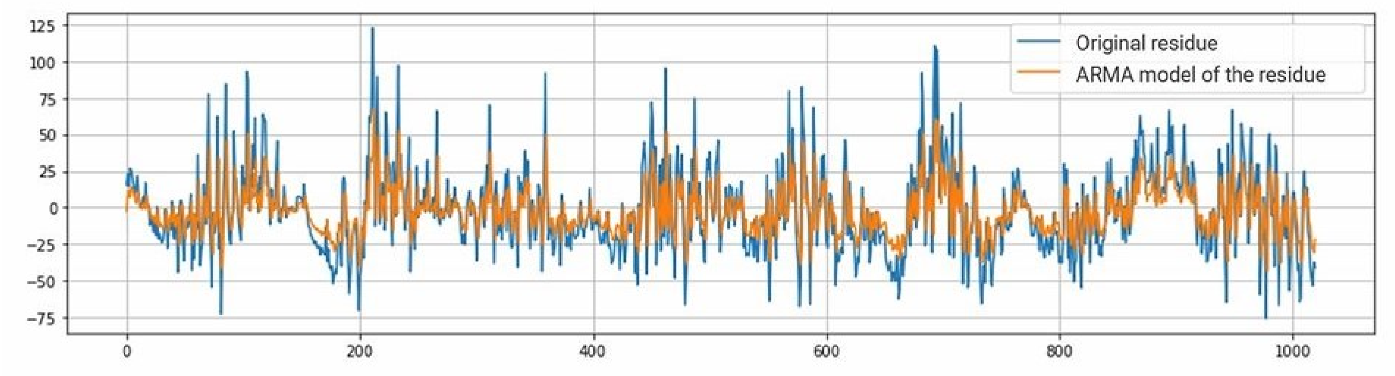
```
from statsmodels.tsa.arima.model import ARIMA

p = 1
q = 0

arma = ARIMA(residue, order=(p, 0, q)).fit()
```

Here you are building a model of order $p = 1$ and $q = 0$. The 0 between those two components corresponds to the I in the abbreviation ARIMA. Setting this to some number, d, will cause the model to first take d differences of the input data in order to achieve stationarity.

Having built the ARIMA object, you can then call the fit() method on that object. This invokes the parameter training procedure. In this case, you have only a single Ø parameter to train. The model can now be applied to the training data; that is, the residue from the decomposition model.

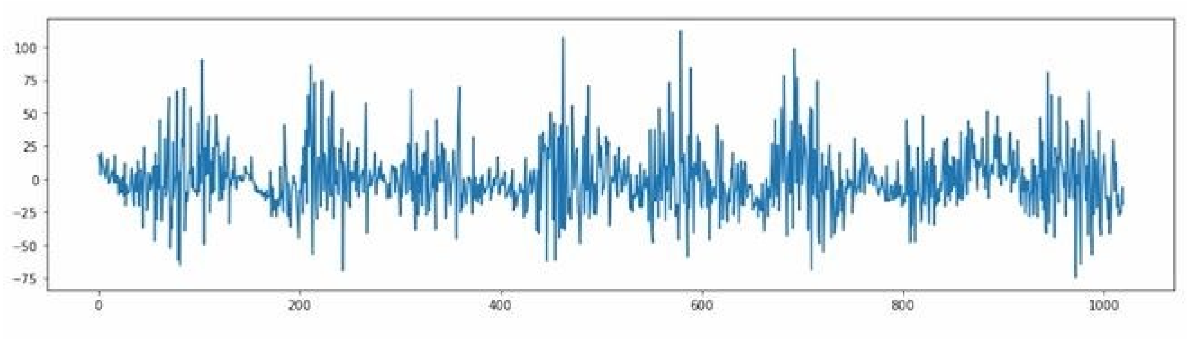
Confusingly, perhaps, this is done with the predict() method of the model.
arma_history = arma.predict()

The result, plotted, is seen here in orange.

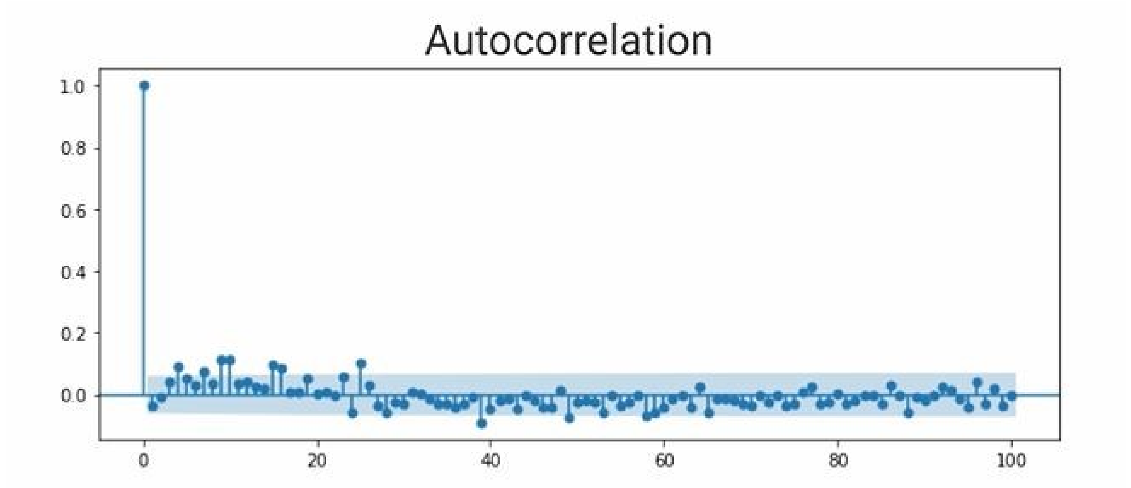


Compared to the original residue, plotted in blue, there is a pretty tight correspondence. But do not be fooled. This is because the ARMA model is making only a one-step prediction from each point in the original residue.

If you now subtract the ARMA prediction from the original residue, you get the residue of the ARMA model; shown here in blue on the plot.



Looking at the autocorrelation of that (below), it is now quite close to white noise and hence, there is no further stationary structure to extract from the sunspots time series.



So ARMA significantly reduced the size of the model residues, but will it help with the ultimate goal of prediction? The answer in this case is no.