Advanced Time Series For Everyone

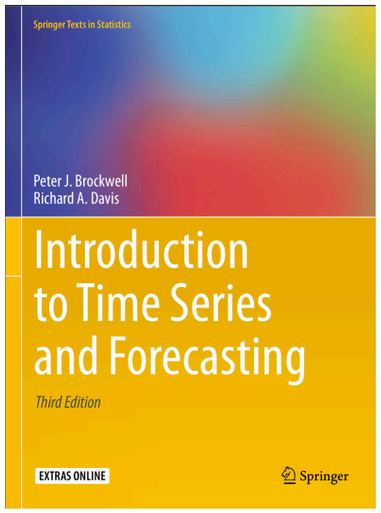
Bruno Gonçalves

www.data4sci.com/newsletter

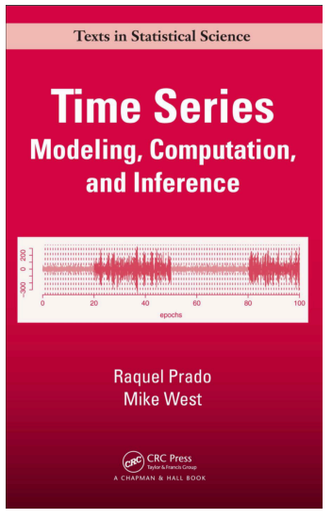
https://github.com/DataForScience/AdvancedTimeseries

References

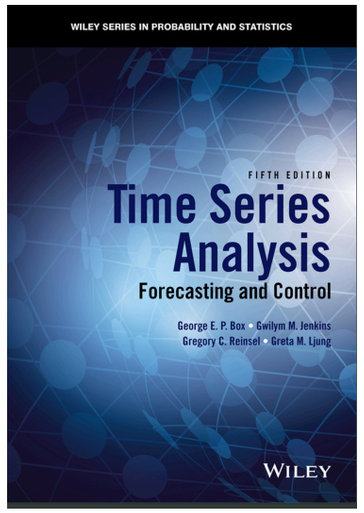
https://github.com/DataForScience/AdvancedTimeseries



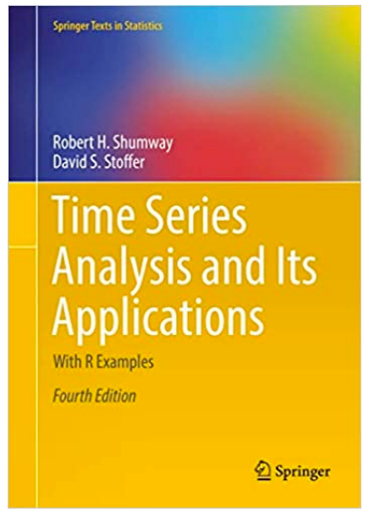
https://amzn.to/30yEp8I



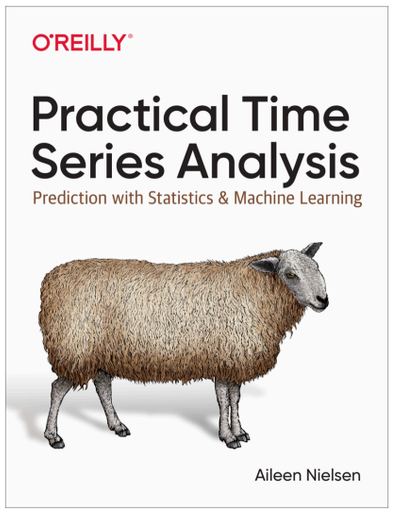
https://amzn.to/2AhvCxt



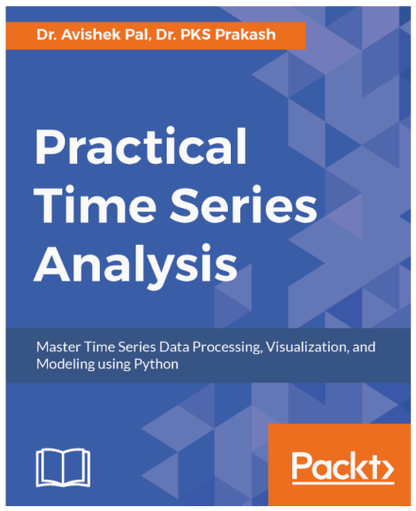
https://amzn.to/30wY5db



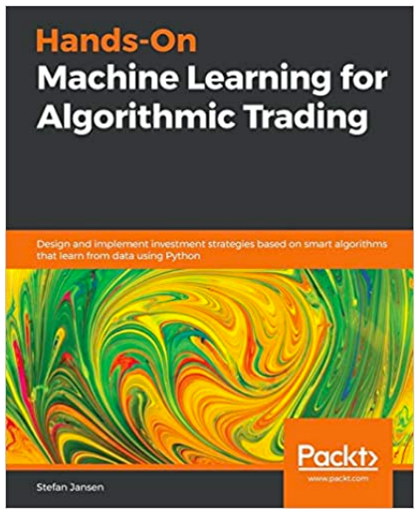
https://amzn.to/3fOVLlM



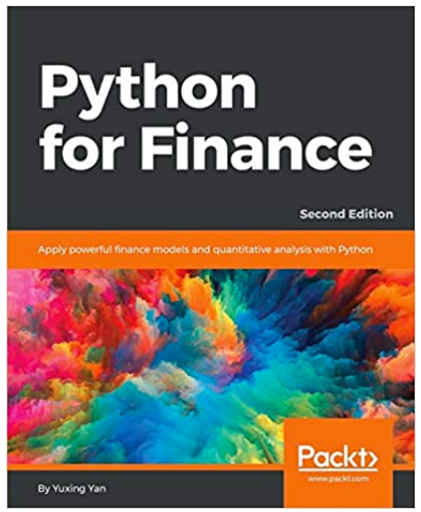
https://amzn.to/2YoEKs6



https://amzn.to/30wY8Wp



https://amzn.to/2Ppvuj8



https://amzn.to/3fxzlp6

Lesson I:
Pandas for Time Series

Time Series

https://github.com/DataForScience/AdvancedTimeseries

- A set of values measured sequentially in time
- Values are typically (but not always) measured at equal intervals, x_1, x_2, x_3, x_3, etc…
- Values can be:
  - continuous
  - discrete or symbolic (words).
- Associated with empirical observation of time varying phenomena:
  - Stock market prices (day, hour, minute, tick, etc…)
  - Temperatures (day, minute, second, etc…)
  - Number of patients (week, month, etc…)
  - GDP (quarter, year, etc…)
- Forecasting requires predicting future values based on past behavior

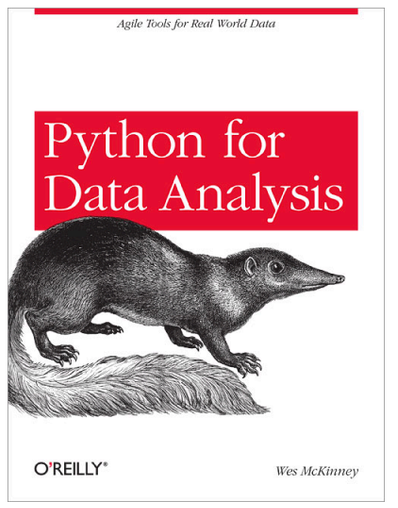
pandas

pandas.pydata.org

- Created by **Wes McKinney** in 2008 while working for **AQR Capital Management**, currently working at **Two Sigma** and **Ursa Labs**
- Pandas is specifically designed to handle time series and data frames
- High performance, flexible tool for quantitative analysis on financial data
- The functionality is built on top **numpy** and **matplotlib** (see next lecture)

- The fundamental data structures are **Series** (1-dimensional) and **DataFrame** (2-dimensional).
- Each column of a **DataFrame** can have a different **dtype** but all the elements in a column (or **Series**) must be of the same **dtype**
- Conventionally imported as

```
import pandas as pd
```



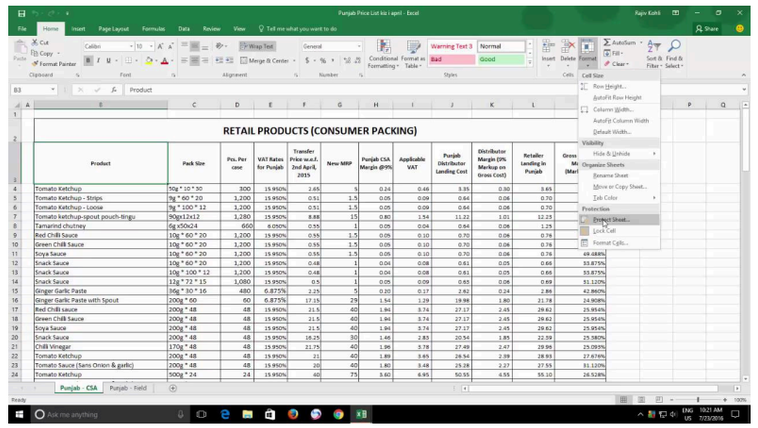
https://amzn.to/2Pmg6UX

Series and Data Frames

- A **Series** is conceptually equivalent to a **numpy** array with some extra information (column and row names, etc)

- A **DataFrame** can be thought of as the union of several **Series**, with names associated.

- Similarly, you can think of a **DataFrame** as an Excel **sheet** and of a **Series** as an individual **column**



- A minimal **DataFrame** implementation would be a dict where each element is a list

Series

	id
0	23
1	42
2	12
3	86

+

Series

	Name
0	“Bob”
1	“Karen”
2	“Kate”
3	“Bill”

=

DataFrame

	id	Name
0	23	“Bob”
1	42	“Karen”
2	12	“Kate”
3	86	“Bill”

DataFrame Fields

- DataFrames contain a great deal of extra information in addition to just the data values. In particular, they include:
    - columns - column names
    - index - row names
    - dtypes - dtype associated with each column
    - shape - number of rows and columns
    - ndim - the number of dimensions
- DataFrames make it easy to add new columns.
- DataFrame elements can be accessed and modified in various ways

DataFrame Exploration

- We get information about the contents of a DataFrame using several simple methods:
  - head() - display the top 5 rows
  - tail() - display the bottom 5 rows
  - info() - Print a concise summary of a DataFrame
  - describe() - Generate descriptive statistics that summarize the data

Indexing and Slicing

- pandas supports various ways of indexing the contents of a DataFrame
- [<column name>] - select a given column by it's name. column names can also be used as field names
- .loc[<row name>] - select a given row by it's (Index) name
- .iloc[<position>] - select a given row by it's position in the Index, starting at 0
- .loc[<row name>, <column name>] - select an individual element by row and column name
- .iloc[<row position>, <row position>] - select an individual element by row and column position (starting at 0)
- .iloc also supports ranges and slices similarly to python lists or numpy arrays

Importing and Exporting Data

- pandas has powerful methods to read and write data from multiple sources
  - pd.read_csv() - Read a comma-separated values file
    - **sep=' '** - Define the separator to use. ',' is the default
    - **header=0** - Row number to use as column names
  - pd.read_excel() - Read an Excel file
    - **sheet_name** - The sheet name to load
  - pd.read_html() - Read tabular data from a URL (or local html file)
  - pd.read_pickle() - Read a Pickle file
- Each of these functions accepts a large number of options and parameters controlling its behavior. Use help(<function name>) to explore further.
- Each read_*() function has a complementary to_*() function to write out a DataFrame to disk. The to_*() functions are members of the DataFrame object

Time Series

- pandas was originally developed to handle financial data
- Temporal sequences (time series) are a common type of data encountered in financial applications, so, naturally, pandas has good support for the most common time series operations.
- pd.to_datetime - converts a Series or value into a date time timestamp.
    - Format is inferred by looking at the entire Series data
    - Format can also be manually specified using specific arguments:
        - format - detailed string specifying the full format
        - dayfirst=True - parse 10/11/12 as Nov 10, 2012
        - yearfirst=True - parse 10/11/12 as Nov 12, 2010
- pd.to_timedelta - converts a Series into an absolute difference in time
- Dates can also be parse automatically at read time by passing a list of the columns containing dates to the parse_dates argument of pd.read_csv

Time Series

- Sequences of dates/times can be created using pd.date_range(), similar to np.arange()
  - start/end - specify the limits
  - freq - specify frequency (step)
    - B - business day frequency
    - D - calendar day frequency
    - W - weekly frequency
    - M - month end frequency
    - Q - quarter end frequency
    - A - year end frequency
    - H - hourly frequency
    - T - minutely frequency
    - S - secondly frequency
- By setting the index to be a date time, we turn the DataFrame into a Time Series.
- Pandas has several methods that are specialized to this case
- index.day/index.month/index.year - return the day, month and year of the time series index value

Transformed values

- We often need to apply some simple transformation to the values in our original Series. Pandas offers several methods to accomplish this goal:
  - map(func) - Map values of Series according to input correspondence.
    - func - function, dict or Series to use for mapping
  - transform(func) - Transform each column/row of the DataFrame using a function. Output must have the same shape as the original
    - axis = 0 - apply function to columns
    - axis = 1 - apply function to rows
- apply(func) - Apply a function along an axis of the DataFrame
  - Similar to transform() but without the limitation of preserving the shape

Lagged values and differences

- While analyzing time series, we often refer to values that our time series took 1, 2, 3, etc time steps in the past
- These are known as lagged values and denoted:

$$x_{t-l}$$

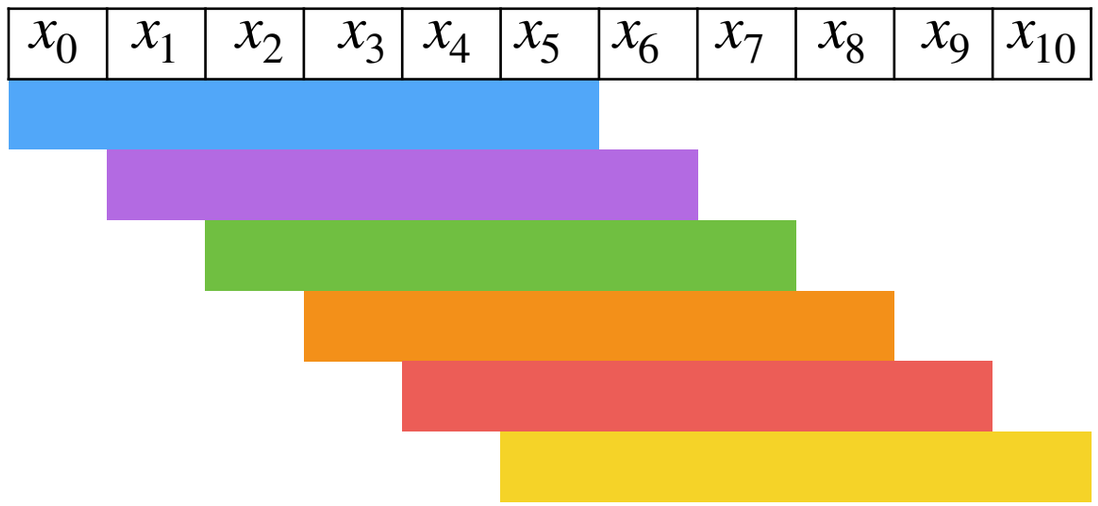
- where l is the value of the lag we are considering.
- pandas supports lagged values with the shift(l) DataFrame method
- Perhaps the most common use case for lagged values is for the calculation of differences of the form:

$$x_t - x_{t-l}$$

- Where $l \geq 1$ is the value of the lag we are interested in.
- Differences are also a particularly simple way to detrend a time series
- pandas supports lagged values with the diff(l) DataFrame method

Windowing

- When analyzing the temporal behavior of of a signal, we often need to evaluate if specific quantities are **time varying** or not
- A common approach is to use **sliding windows** of a given length to evaluate the required values
- So a sliding window of width 6 on a series of length 11 would look like:



$$= \begin{matrix} x_0 & x_1 & x_2 & x_3 & x_4 & x_5 \\ x_1 & x_2 & x_3 & x_4 & x_5 & x_6 \\ x_2 & x_3 & x_4 & x_5 & x_6 & x_7 \\ x_3 & x_4 & x_5 & x_6 & x_7 & x_8 \\ x_4 & x_5 & x_6 & x_7 & x_8 & x_9 \\ x_5 & x_6 & x_7 & x_8 & x_9 & x_{10} \end{matrix}$$

- and we would calculate the metric of interest **within each window**.
- In Pandas we use the method **rolling(w)** that returns dynamic view of the data allowing us to daisy-chain other common **numpy** operators. **rolling(w).mean()**, **rolling(w).max()**, etc...

groupby

- Sometimes we need to calculate statistics for subsets of our data
- groupby() allows us to group data based on the values of a column
- groupby() returns a GroupBy object that supports several aggregations functions, including:
    - max()/min()/mean()/median()
    - transform()/apply()
    - sum()/cumsum()
    - prod()/cumprod()
    - quantile()
- Each of these functions is applied to the contents of each group

Pivot Tables

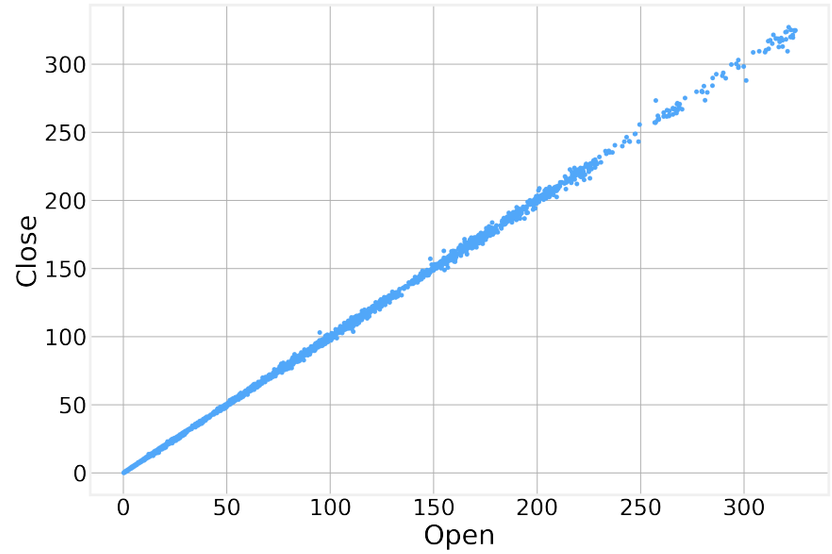
- Pandas provides an extremely flexible pivot table implementation
- pd.pivot_table() - Creates a spreadsheet-style pivot table as a DataFrame.
- Three important arguments:
    - values - column to aggregate
    - index - Keys to group by on the pivot table index. If an array is passed, it is being used as the same manner as column values.
    - columns - Keys to group by on the pivot table column.
    - aggfunc - Function to use for aggregation. Defaults to np.mean()
- index, columns and aggfunc can also be lists of keys or functions to use.

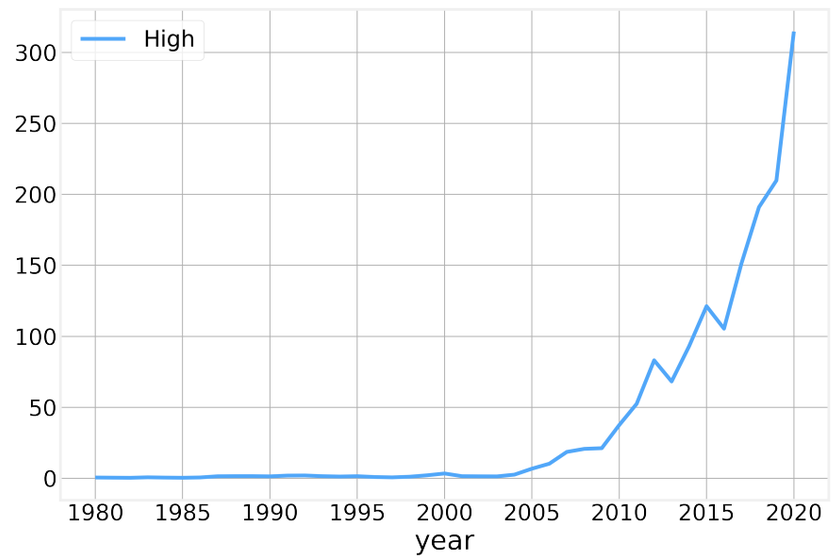
merge/join

- merge() and join() allows us to perform database-style join operation by columns or indexes (rows)
- Some of the most important arguments are common to both methods:
    - on - Column(s) to use for joining, otherwise join on index. Can pass an array as the join key if not already contained in the calling DataFrame. Like an Excel VLOOKUP operation
    - how - Type of join to perform: {'left', 'right', 'outer', 'inner'}
- merge() is more sophisticated and flexible. It also allows us to specify:
    - left_on/right_on - Field names to join on for each DataFrame
    - left_index/right_index - Whether or not to use the left/right index as join key(s)

plotting

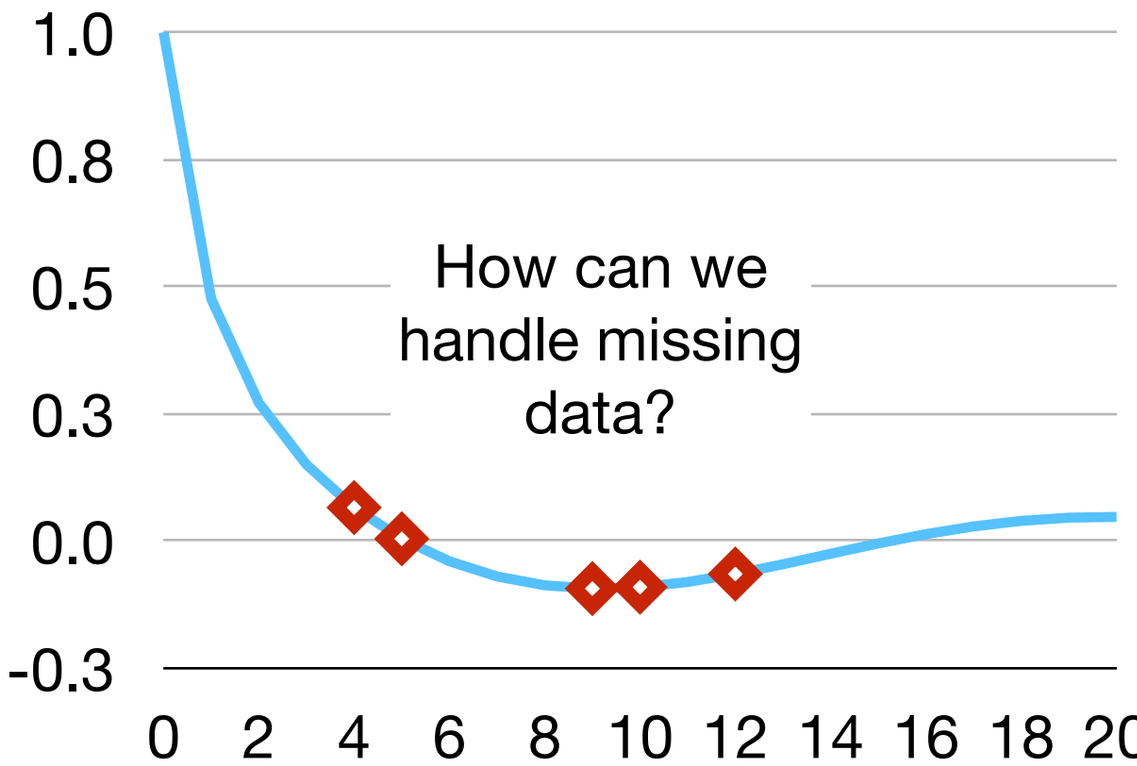
- Finally, pandas also provides a simple interface for basic plotting through the **plot()** function
- By specifying some basic parameters the variables plotted and even the kind of plot can be easily modified:
  - **x/y** -column name to use for the **x/y** axis
  - **kind** - type of plot
    - **'line'** - line plot (default)
    - **'bar'**/**'barh'** - vertical/horizontal bar plot
    - **'hist'** - histogram
    - **'box'** - boxplot
    - **'pie'** - pie plot
    - **'scatter'** - scatter plot
- The **plot()** function returns a **matplotlib Axes** object



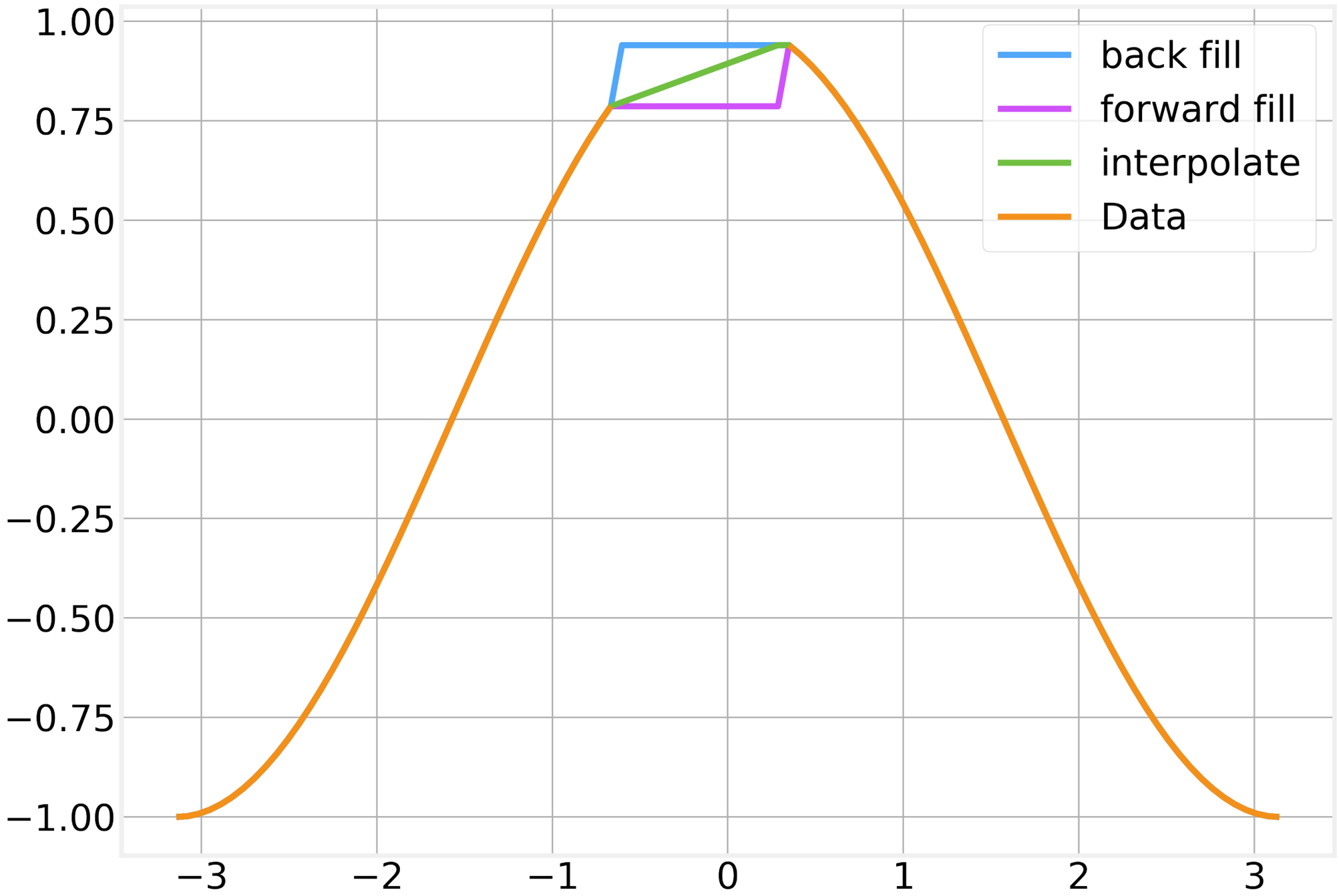


Fill methods

- Often the time series is incomplete.
- Missing data points can be due to data corruption, data collection issues, etc.
- Missing values are represented as nan
- Several techniques have been developed to handle this case:
  - fillna() - replace all nan values with a specific value
  - ffill() - keep the last valid value
  - bfill() - keep the next valid value
  - interpolate() - add values by interpolating between the previous and the next value:
    - the default method is "linear" interpolation, but pandas supports many others, including "time" to scale the values based on time interval.



Fill methods



Resampling

- Time Series typically have an **intrinsic time scale** at which the data was collected: ticks, seconds, days, months, etc

- In many cases, our analysis requires that we **resample** the data to a different time scale

- Resampling to a longer timescale is relatively simple and similar to aggregation:

    - Transforming from daily to weekly frequency requires simply aggregating by week

- Resampling to shorter timescales requires **interpolation or imputation** to make up for the missing values

    - Going from weekly to daily frequency requires **specifying how to allocate** the values for each day of the week

Code - Pandas for Time Series

https://github.com/DataForScience/AdvancedTimeSeries

Lesson II:
statsmodels for Time Series

statsmodels.org

- Python module built on top of numpy and scipy
- Provides classes and functions for:
  - estimations of various statistical models
  - statistical tests
  - data exploration
- Extensive statistics are available for each estimate produced
- Particularly useful for:
  - Linear Regression
  - Generalized Linear Models
  - Survival Analysis
  - Time Series!

statsmodels.org

- Conventionally imported as:

```
import statsmodels.api as sm
```

- Supports **R-style** formula notation for statistical models
- For **R-style** formula notation support, conventionally imported as:

```
import statsmodels.formula.api as smf
```

- And for timeseries:

```
import statsmodels.api as sm
```

-

statsmodels.org

statsmodels.	api.	OLS()	.fit() .predict() .summary()		
		tsa.	seasonal_decompose()		
			stattools.	acf() pacf() adfuller()	
			arima.	ARIMA()	.fit() .predict() .forecast() .plot_predict() .summary()
			statespace.	SARIMAX()	.fit() .predict() .forecast() .summary()
		graphics.	tsa.	plot_acf() plot_pacf()	

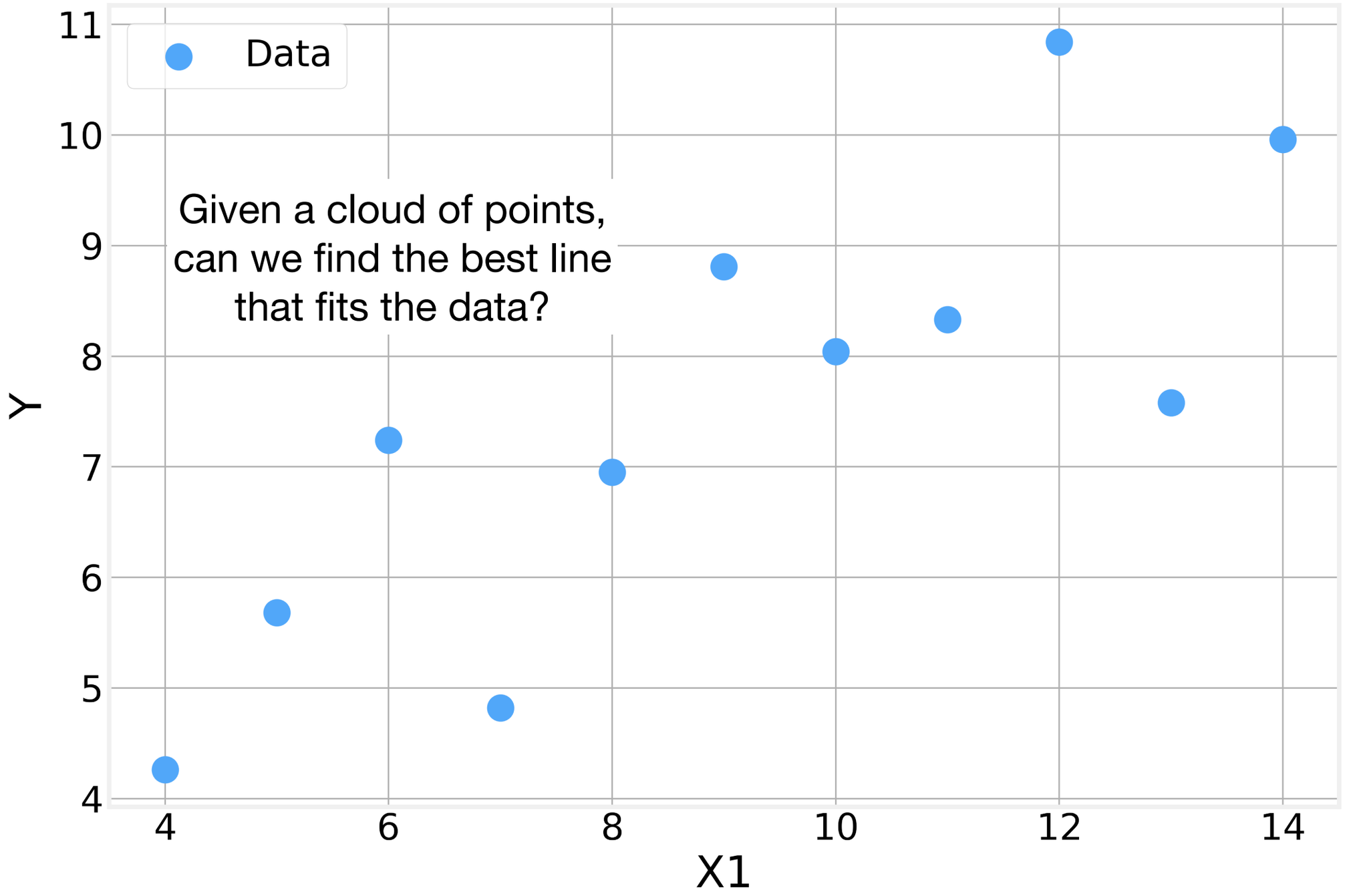
statsmodels.org

- statsmodels refers to variables as:
  - "endogenous" - caused by factors within the system
  - "exogenous" - caused by factors outside the system

endog	exog
y	x
y variable	x variable
left hand side (LHS)	right hand side (RHS)
dependent variable	independent variable
regressand	regressors
outcome	design
response variable	explanatory variable

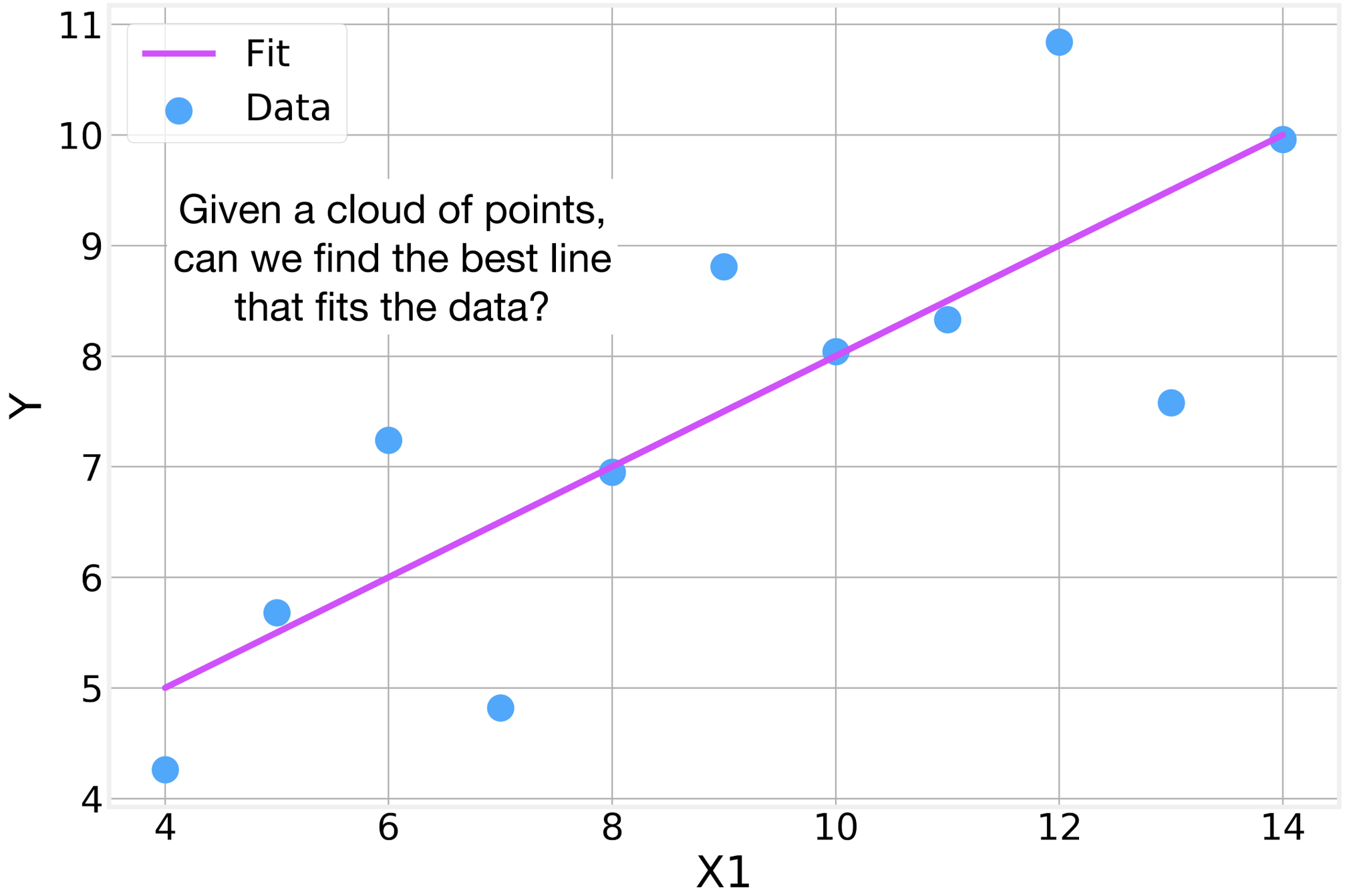
Linear Regression

https://en.wikipedia.org/wiki/Linear_regression



Linear Regression

https://en.wikipedia.org/wiki/Linear_regression



Linear Regression

https://www.statsmodels.org/devel/generated/statsmodels.regression.linear_model.OLS.html

- statsmodels returns:
  - a description of the model and fitting procedure,
  - the coefficients of the fitted function,
  - estimates of the error,
  - statistical significance,
  - and a wealth of statistical test results.

OLS Regression Results

Dep. Variable:	y	**R-squared:**	0.667
Model:	OLS	**Adj. R-squared:**	0.629
Method:	Least Squares	**F-statistic:**	17.99
Date:	Sun, 09 Aug 2020	**Prob (F-statistic):**	0.00217
Time:	12:37:16	**Log-Likelihood:**	-16.841
No. Observations:	11	**AIC:**	37.68
Df Residuals:	9	**BIC:**	38.48
Df Model:	1		
Covariance Type:	nonrobust		

	coef	std err	t	P>|t|	[0.025	0.975]
const	3.0001	1.125	2.667	0.026	0.456	5.544
x1	0.5001	0.118	4.241	0.002	0.233	0.767

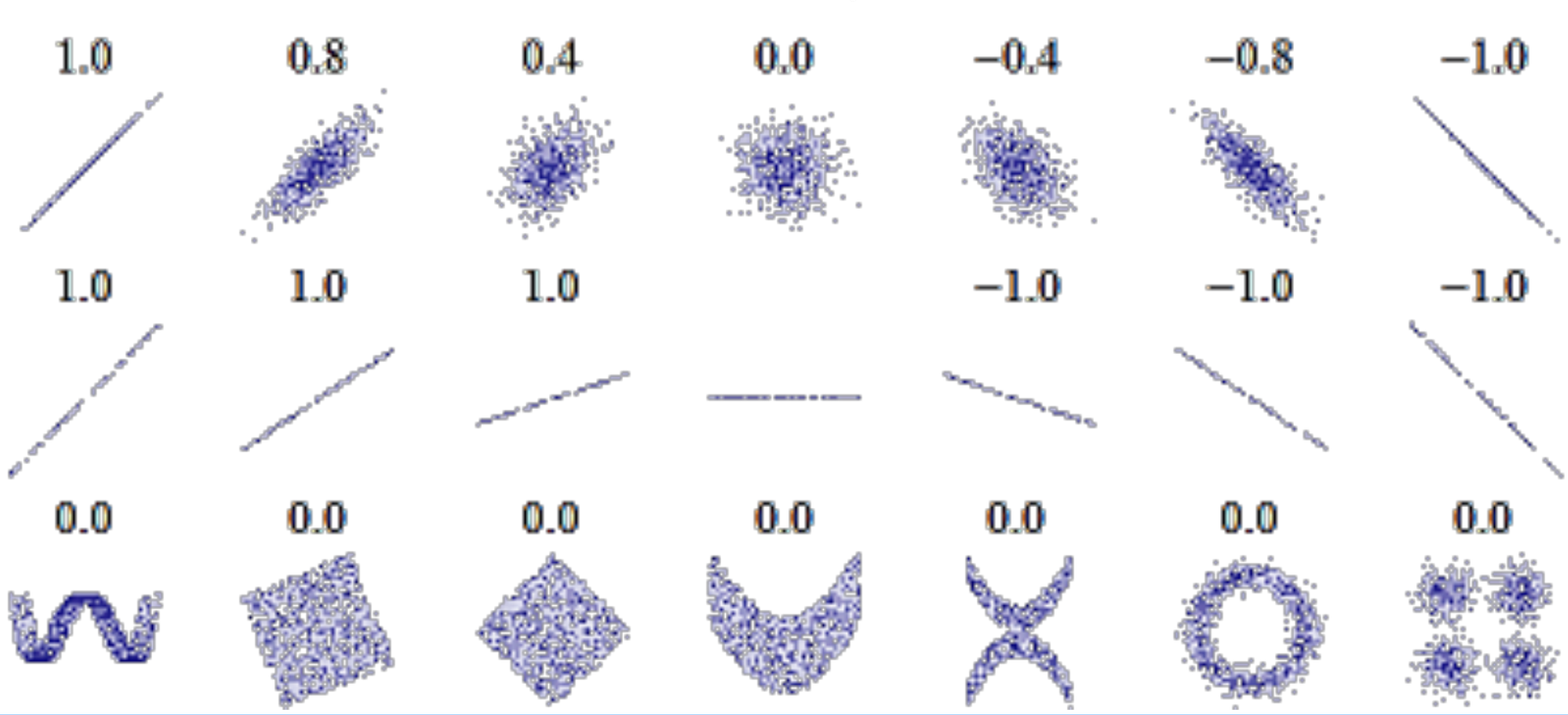
Omnibus:	0.082	**Durbin-Watson:**	3.212
Prob(Omnibus):	0.960	**Jarque-Bera (JB):**	0.289
Skew:	-0.122	**Prob(JB):**	0.865
Kurtosis:	2.244	**Cond. No.**	29.1

Pearson Correlation

- Many correlation measures have been proposed over the years
- The most well known one is the Pearson Correlation

$$\rho\left(x,y\right) = \sum_{i=1}^{N} \frac{\left(x_i - \mu_x\right)\left(y_i - \mu_y\right)}{\sigma_x \sigma_y}$$

- Assumes a linear relationship between x and y.



Auto-correlation

- It follows from the previous slide that a series will have a perfect correlation with itself, but what about lagged versions of itself?

- The Auto-correlation function is simply the Pearson correlation between values of the time series at different lags, as a function of the lag:

$$ACF_x\left(l\right) = \rho\left(x_t, x_{t-l}\right)$$

- By definition, $ACF_x(0) \equiv 1$

- And as the lag l increases the value of the ACF tends to decrease.

- We can calculate the confidence interval for the ACF using:

$$CI = \pm z_{1-\alpha/2}\sqrt{\frac{1}{N}\left(1 + 2\sum_{l=1}^{k} r_l^2\right)}$$

- where $z_{1-\alpha/2}$ is the quantile of the normal distribution corresponding to significance level α and r_l are the values of the ACF for a specific lag l

Auto-correlation

https://www.statsmodels.org/devel/generated/statsmodels.tsa.stattools.acf.html
https://www.statsmodels.org/devel/generated/statsmodels.graphics.tsaplots.plot_acf.html

- statsmodels provides us with the acf() and plot_acf() methods in the tsa.statstools and the graphics.tsa modules, respectively.

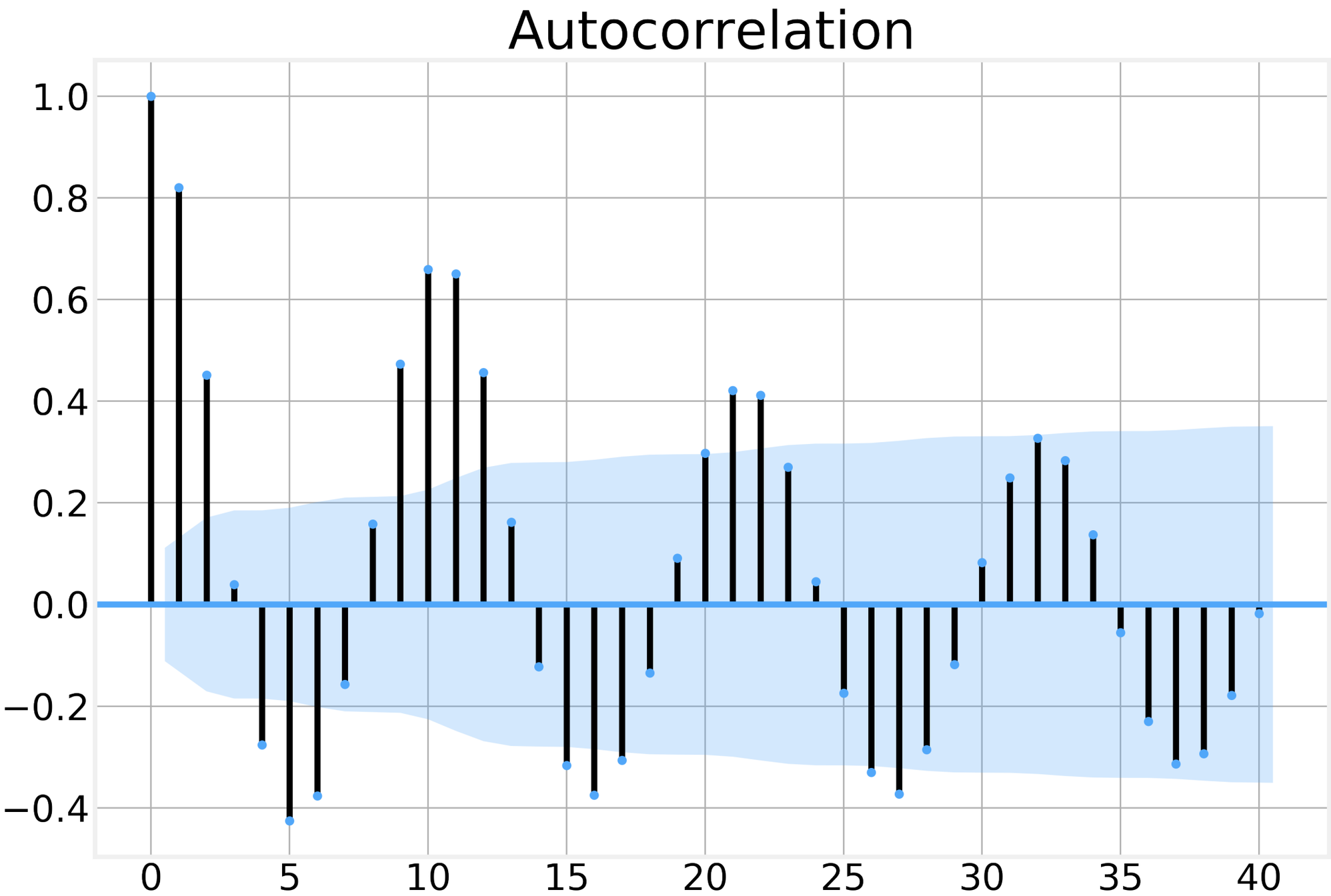
- acf() calculates the values of the auto-correlation function up to nlags (default=40)

- plot_acf() calculates the auto-correlation function and the respective confidence interval up to lags (default=the number of significant auto-correlation values)

- Both of these methods include various optional arguments to help customize their behavior.

Auto-correlation

https://www.statsmodels.org/devel/generated/statsmodels.tsa.stattools.acf.html
https://www.statsmodels.org/devel/generated/statsmodels.graphics.tsaplots.plot_acf.html



Partial Autocorrelation

https://en.wikipedia.org/wiki/Partial_autocorrelation_function

- One of the disadvantages of the Autocorrelation function is that it still considers the intermediate values

- The Partial Autocorrelation function calculate the correlation function between x_t and x_{t-l} after explaining away all the intermediate values $x_{t-1} \cdots x_{t-l+1}$

- Intermediate values are "explained away" by fitting a linear model of the form:

$$\hat{x}_t = f\left(x_{t-1} \cdots x_{t-l+1}\right)$$

$$\hat{x}_{t-l} = f\left(x_{t-1} \cdots x_{t-l+1}\right)$$

- And then calculating the Pearson correlation function between the values and their residuals:

$$PACF_x\left(l\right) = \rho\left(x_t - \hat{x}_t, x_{t-l} - \hat{x}_{t-l}\right)$$

- Confidence intervals can be computed using the same formula used for the ACF

Partial Autocorrelation

https://www.statsmodels.org/devel/generated/statsmodels.tsa.stattools.pacf.html
https://www.statsmodels.org/devel/generated/statsmodels.graphics.tsaplots.plot_pacf.html

- statsmodels uses a similar structure for the partial auto-correlation function with a pacf() and plot_pacf() functions provided by the tsa.statstools and the graphics.tsa modules, respectively.

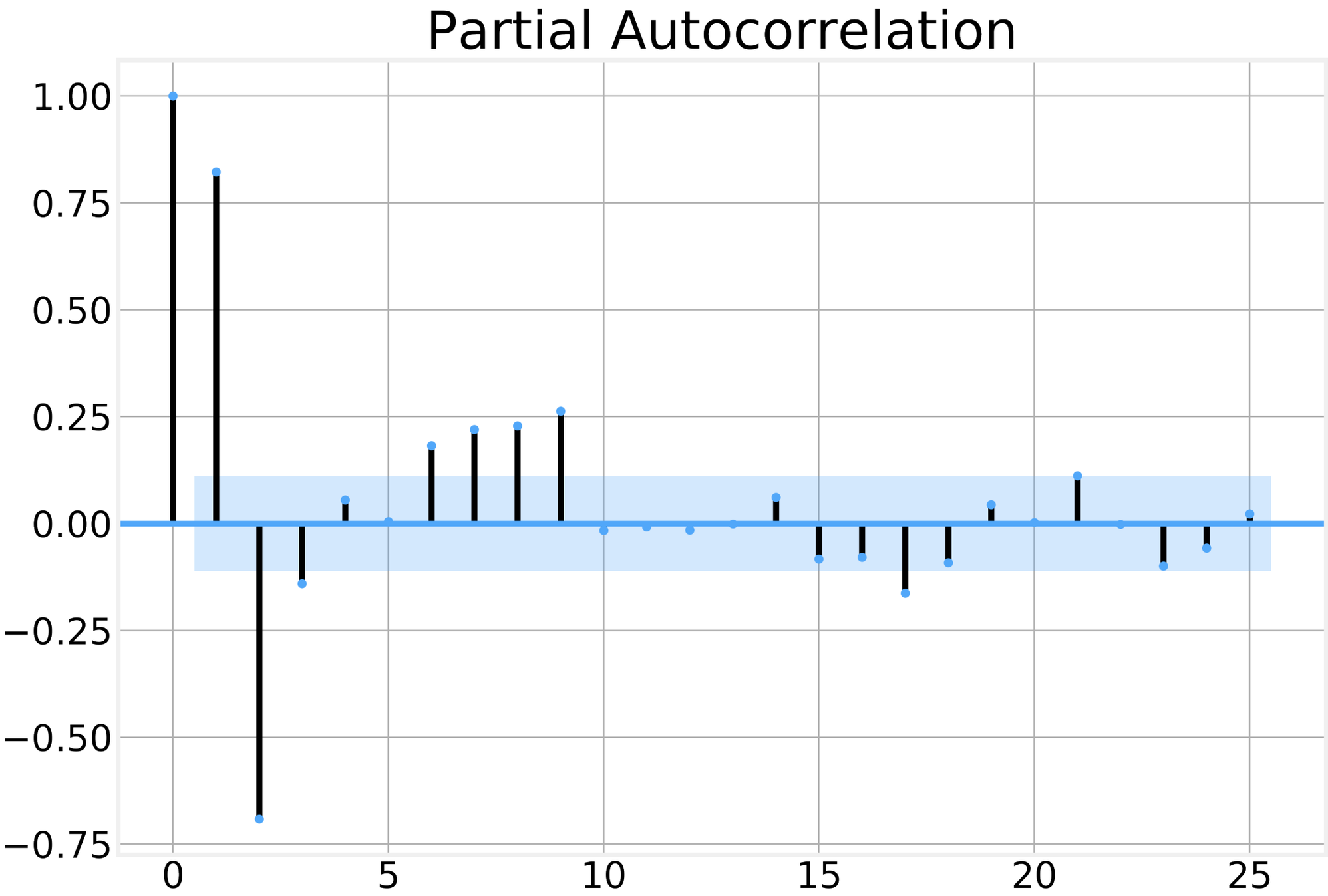
- pacf() calculates the values of the partial auto-correlation function up to nlags (default=40)

- plot_acf() calculates the auto-correlation function and the respective confidence interval up to lags (default=the number of significant auto-correlation values)

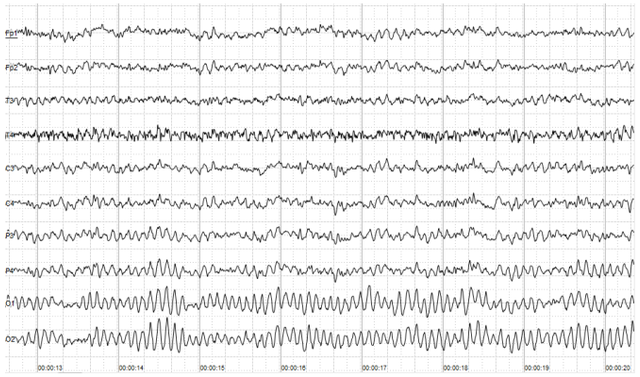
- And as with the acf/plot_acf duo, both of these methods include various optional arguments to help customize their behavior.

Partial Autocorrelation

https://www.statsmodels.org/devel/generated/statsmodels.tsa.stattools.pacf.html
https://www.statsmodels.org/devel/generated/statsmodels.graphics.tsaplots.plot_pacf.html

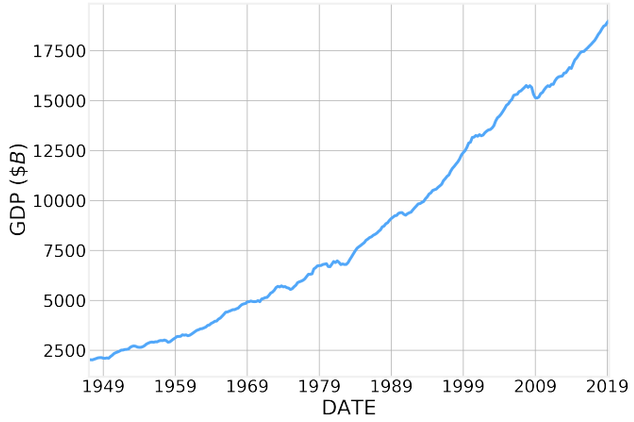


Three fundamental behaviors



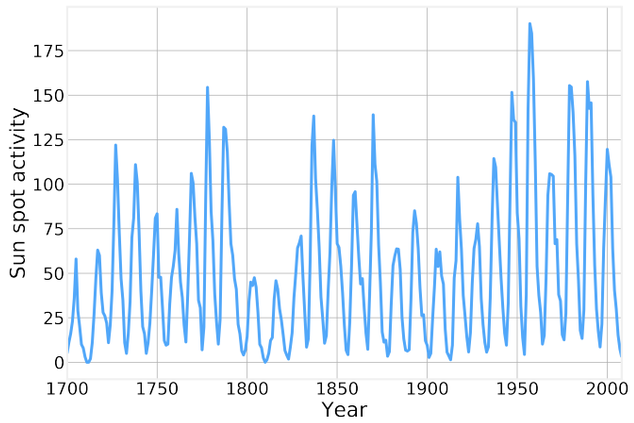
Stationarit

$$\langle x_t \rangle \approx \text{constant}$$



Trend

$$\langle x_t \rangle \approx ct$$



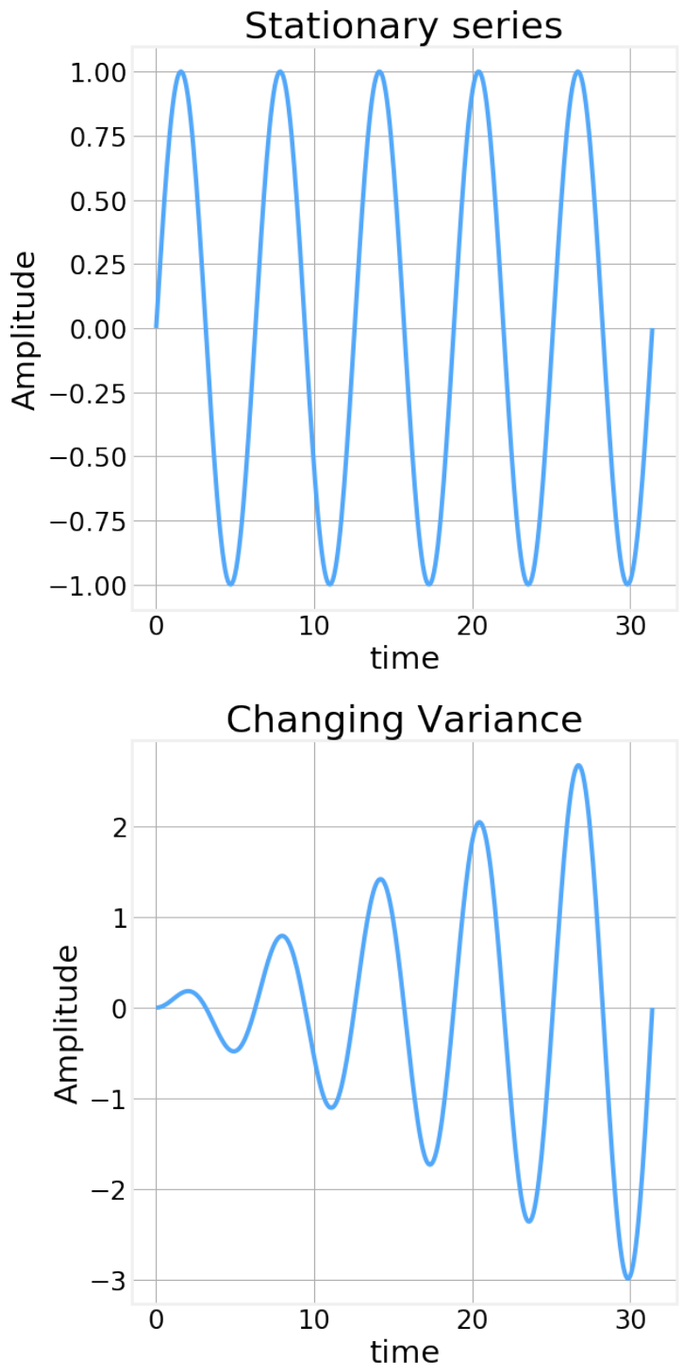
Seasonalit

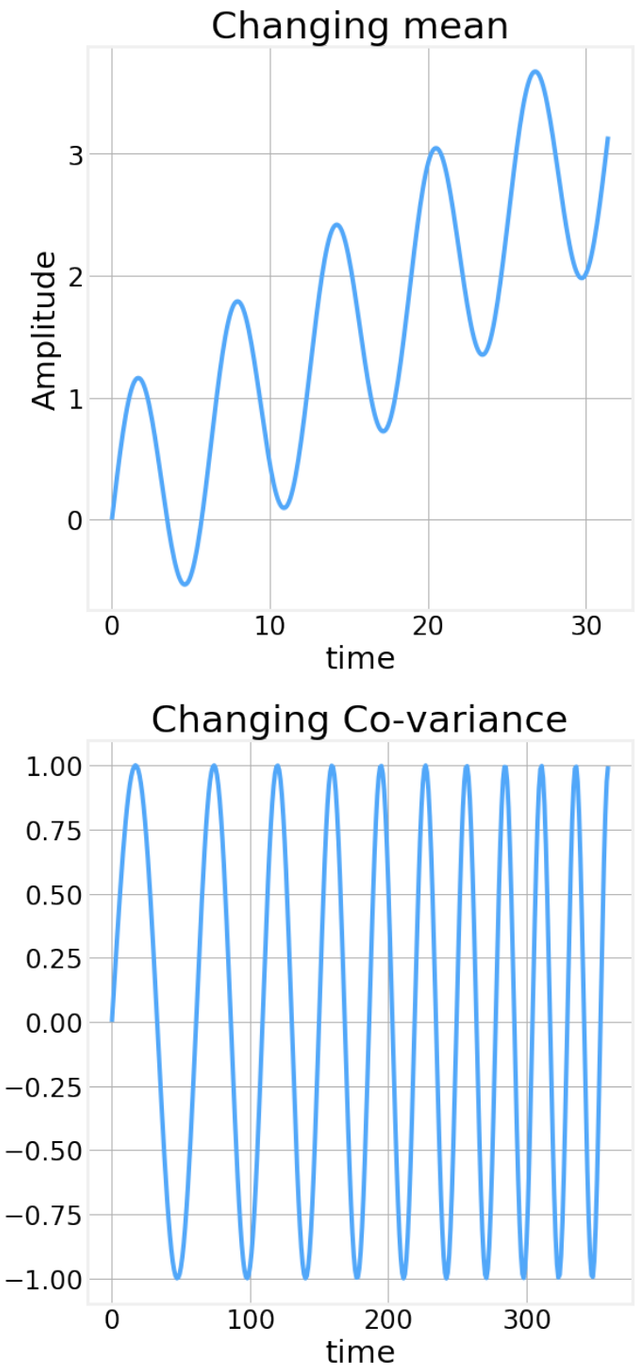
$$x_{t+T} \approx x_t$$

Stationarity

- A time series is said to be stationary if its basic statistical properties are independent of time
- In particular:
    - Mean - Average value stays constant
    - Variance - The width of the curve is bounded
    - Covariance - Correlation between points is independent of time
- Stationary processes are easier to analyze. If the parameters of the model vary in time, the number of parameters we must estimate increases significantly.
- Many time series analysis algorithms assume stationarity.
- Several rigorous tests for stationarity have been developed such as the (Augmented) Dickey-Fuller and Hurst Exponent

Stationarity





Dickey-Fuller Test

- The **Dickey-Fuller Test** is a test of stationarity inspired by a simple: Can we show that

$$\rho \neq 1$$

- with some degree of certainty?

- Numerically, we can express this as a **linear regression** fit

$$x_t - x_{t-1} = \gamma \; x_{t-1} + \epsilon_t$$

- where $\gamma = \rho - 1$

- The slope of the regression is then our expected value for $\rho - 1$.

- If the process is **non-stationary** then we expect $\gamma \neq 0$.

Dickey-Fuller Test

- statsmodels provides an implementation of the Augmented Dickey-Fuller test with the function adftest() in the tsa.stattools module
- adftest() is able to automatically handle multiple lags, adjust for trends, etc.
- adftest(), when using the parameter regresults=True, returns a tuple with 4 elements:
    - adfstat - the value of the augmented dickey fuller statistic
    - p-value - the p-value (statistical significance) of the result
    - criticalvalues - the critical values for the test statistic at the 1%, 5%, and 10% levels
    - results - an object containing more information, including the results for the linear regression
- We encourage you to explore the various arguments to this function to fully understand its functionality.

Time series decomposition

- A time series can be decomposed into three components:
  - Trend, T_t
  - Seasonality, S_t
  - Residuals, R_t
- Decompositions can be
  - additive - $x_t = T_t + S_t + R_t$
  - multiplicative - $x_t = T_t \cdot S_t \cdot R_t$
- The residuals are simply what is left of the original signal after we remove the trend and the seasonality
- Residuals are typically stationary

Time series decomposition

https://www.statsmodels.org/dev/generated/statsmodels.tsa.seasonal.seasonal_decompose.html

- statsmodels provide the seasonal_decompose() function in the tsa.seasonal module to perform the seasonal decomposition described above.

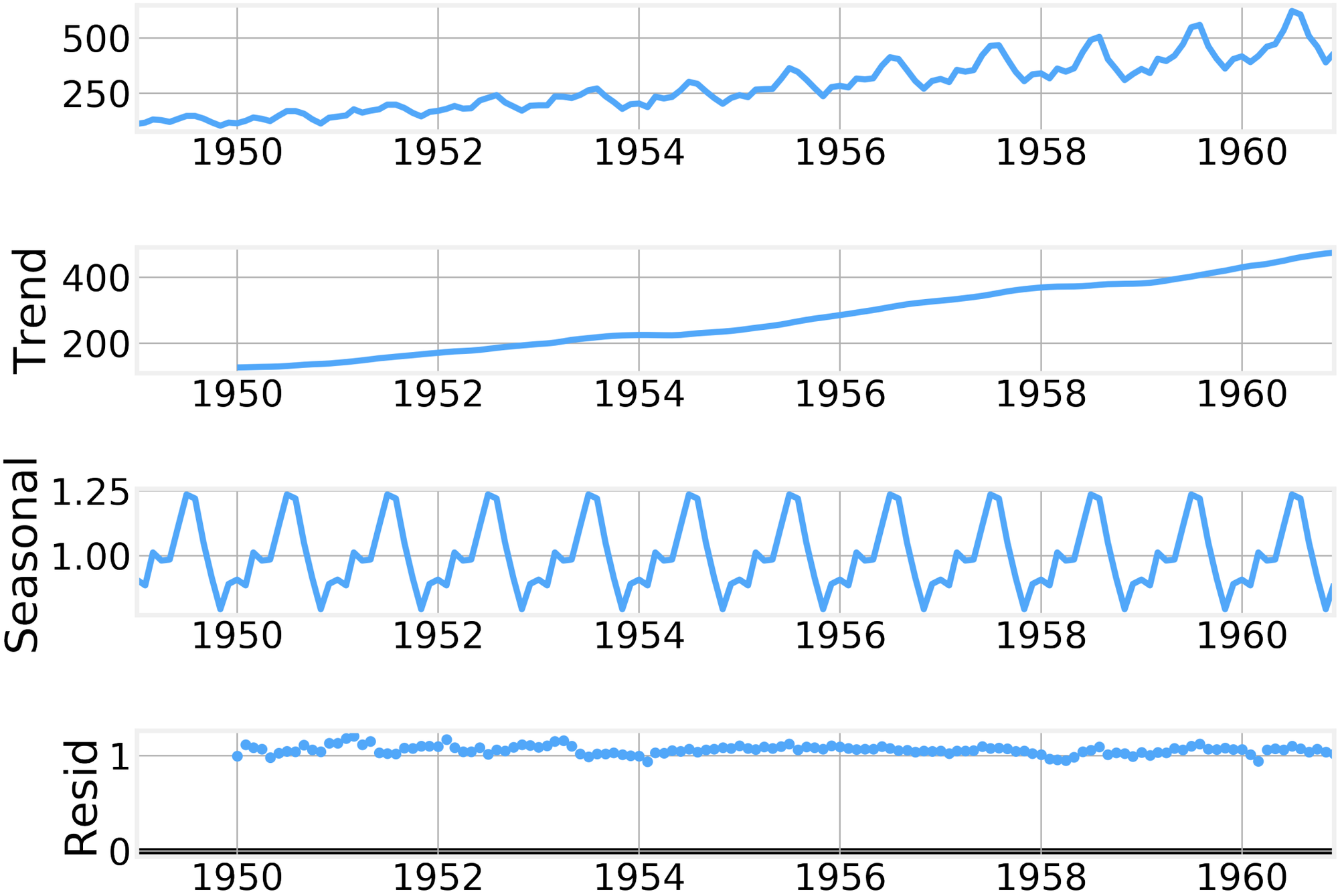
- The model argument allows us to select between “additive” or “multiplicative” decomposition.

- The two_sided argument lets us decide how to calculate the moving average. We should set it to False to make sure the moving average values are computed only past values.

- It returns an object with observed, seasonal, trend, and resid attributes as pandas DataFrames. The plot() method generates the usual decomposition plot.

Time series decomposition

https://www.statsmodels.org/dev/generated/statsmodels.tsa.seasonal.seasonal_decompose.html



Code - statsmodels

https://github.com/DataForScience/AdvancedTime Series

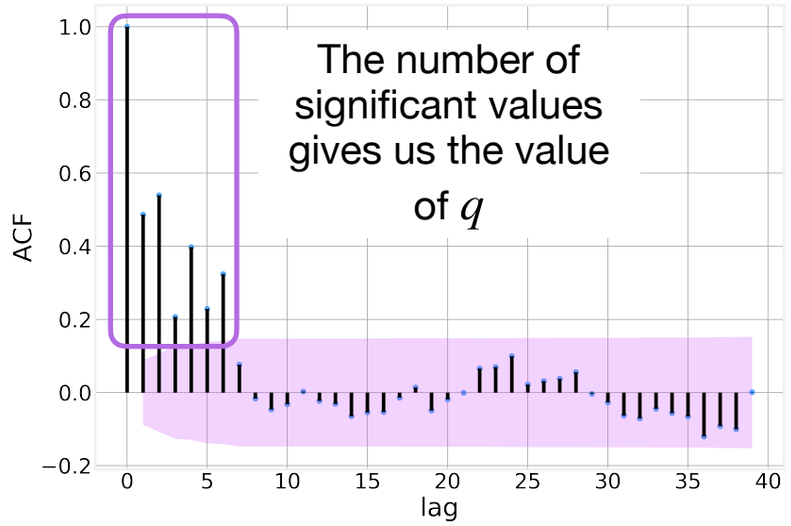
Lesson III:
ARIMA Models

Moving Average (MA) Model

- A more general case can be written as:

$$x_t = \beta + \sum_{l=0}^{q} \omega_l \, \epsilon_{t-l}$$

- where β is a constant offset, ω_l are the weights for the values at lag l and q is the moving window size. $\omega_0 \equiv 1$.



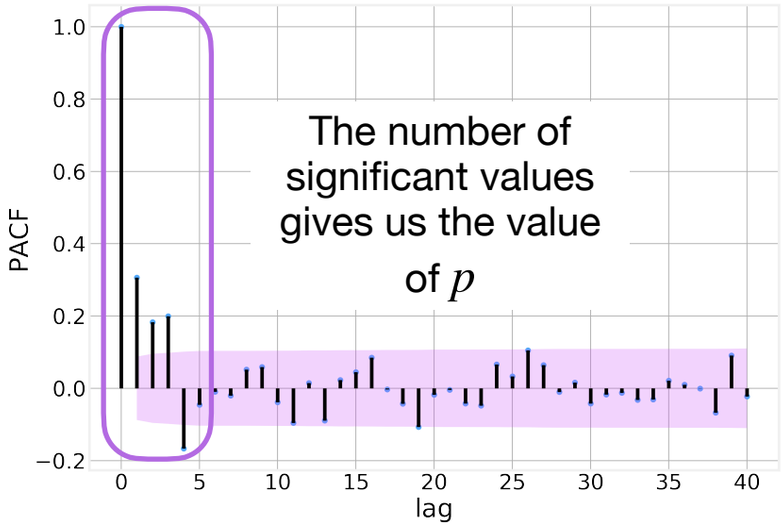
- The ϵ_t values are stochastic variables (often referred to as “errors") rather than the actual observed values x_t
- The generated x_t is uncorrelated with itself for any lag $l > q$

Auto-Regressive (AR) Models

- Auto-Regressive models can be defined as:

$$x_t = \alpha + \epsilon_t \sum_{l=1}^{p} \phi_l \; x_{t-l}$$

- Where the constant c represents the process' average value and the x_{t-l} are the observed values at a given lag l and ϕ_l are the corresponding weights.



- The generated x_t has negligible partial correlation (uncorrelated) for any lag $l > p$
- The series "reverts to the mean".

Integrative (I) "model"

- We already saw that we can take differences to "stationarize" the time series.
- To recover the original values, we must then integrate
- While not a model by itself, it is often an important first step in modeling time series

ARIMA model

- The ARIMA class of models is a combination of these three distinct models, each of which accounts for a specific feature of classical time series:

Auto Regressive Integrated Moving Average

$$x_t = c + \sum_{i}^{p} \phi_i x_{t-i} + \epsilon_t$$

Regression to the mean - relying on positions up to lag p

Integrated - considering only differences up to order d

$$x_t = \mu + \epsilon_t + \sum_{j}^{q} \theta_j \epsilon_{t-j}$$

Random Walk - relying on random variables up to lag q

- The complete model can be written as:

$$\hat{x}_t = c + \mu + \sum_{i}^{p} \phi_i x_{t-i} + \sum_{j}^{q} \theta_j \epsilon_{t-j} + \epsilon_t$$

- where $\hat{x}_t$ is the properly differentiated time series

- Naturally, by selecting the proper parameters, we're able to recover any combination of the three models considered

ARIMA model

- $ARIMA\,(0,1,0)$ - Random Walk (with or without drift)
  - $x_t - x_{t-1} = c + \epsilon_t$
- $ARIMA\,(0,0,0)$ - White noise (the sequence of stochastic variables)
  - $x_t = \epsilon_t$
- $ARIMA\,(0,1,1)$ - Exponential Smoothing
  - $x_t - x_{t-1} = \epsilon_t + \theta_1 \epsilon_{t-1}$
- $ARIMA\,(0,2,2)$ - Double exponential Smoothing
  - $x_t - 2x_{t-1} + x_{t-2} = \epsilon_t + \theta_1 \epsilon_{t-1} + \theta_2 \epsilon_{t-2}$

ARIMA model

https://www.statsmodels.org/devel/generated/statsmodels.tsa.arima.model.ARIMA.html

- statsmodels provides the ARIMA object in the tsa.arima module
- This object allows us to fit a specific model (specified by the argument order=(p, q, d)) to a given dataset, i.e. find the specific values of θ_j and ϕ_i
- The procedure to fit the model is similar to that of sklearn, relying two specific methods:
  - fit() - Determine the parameters of the model
  - predict() - "Run" the model to determine the predicted values of the "position" x_t
- Plus our friendly summary() to summarize the results and plot_predict() to compare the original data with the fitted model.

How can we find the values of p, q, and

Fitting ARIMA models

The general procedure to fit an ARIMA model was originally proposed by Box-Jenkins

- d - degree of differencing
- p - number of lag observations included in the model ($PACF$)
- q - size of the moving average window (ACF)

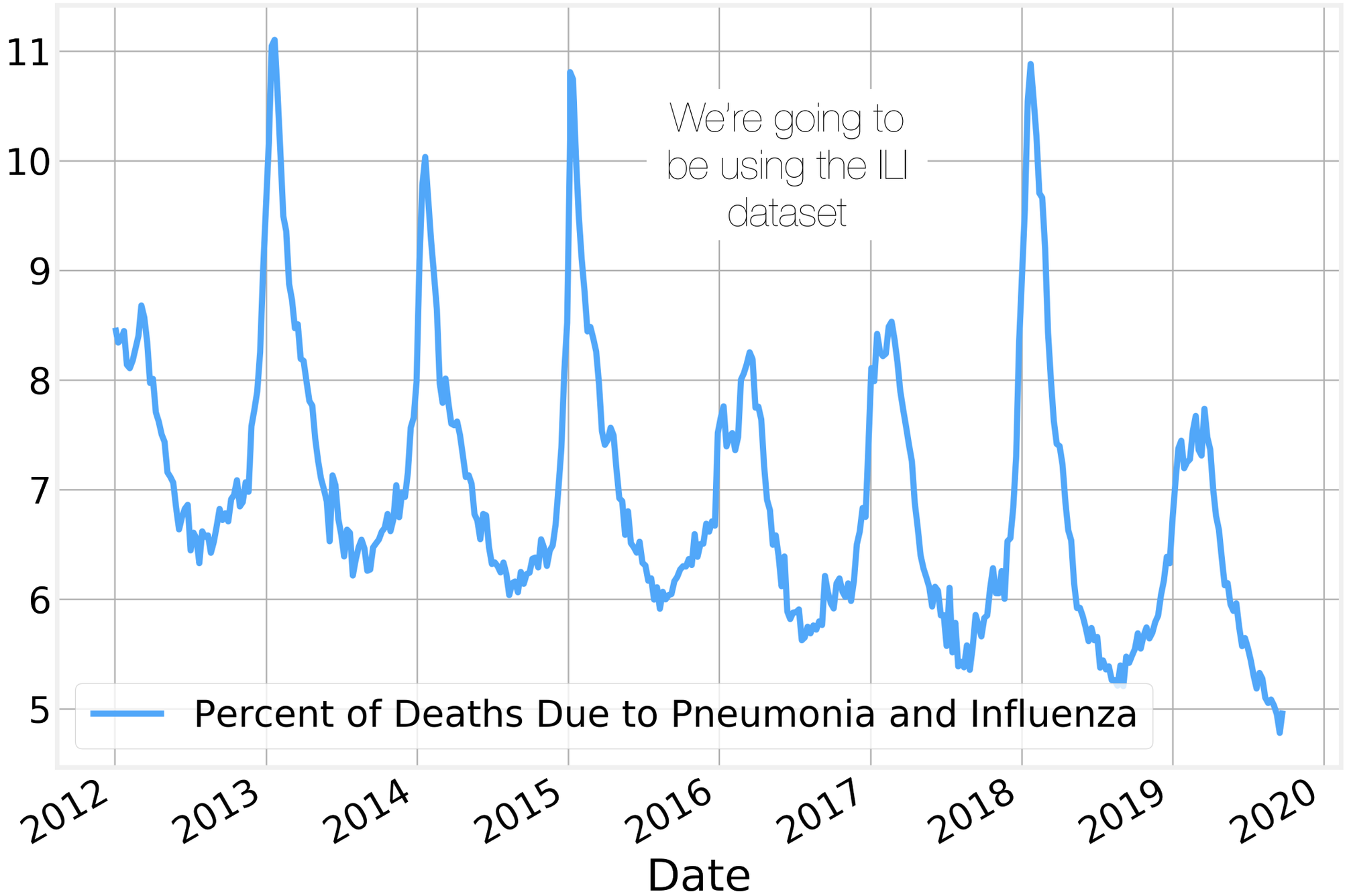
Fitting ARIMA models

- d - degree of differencing
- p - number of lag observations included in the model ($PACF$)
- q - size of the moving average window (ACF)

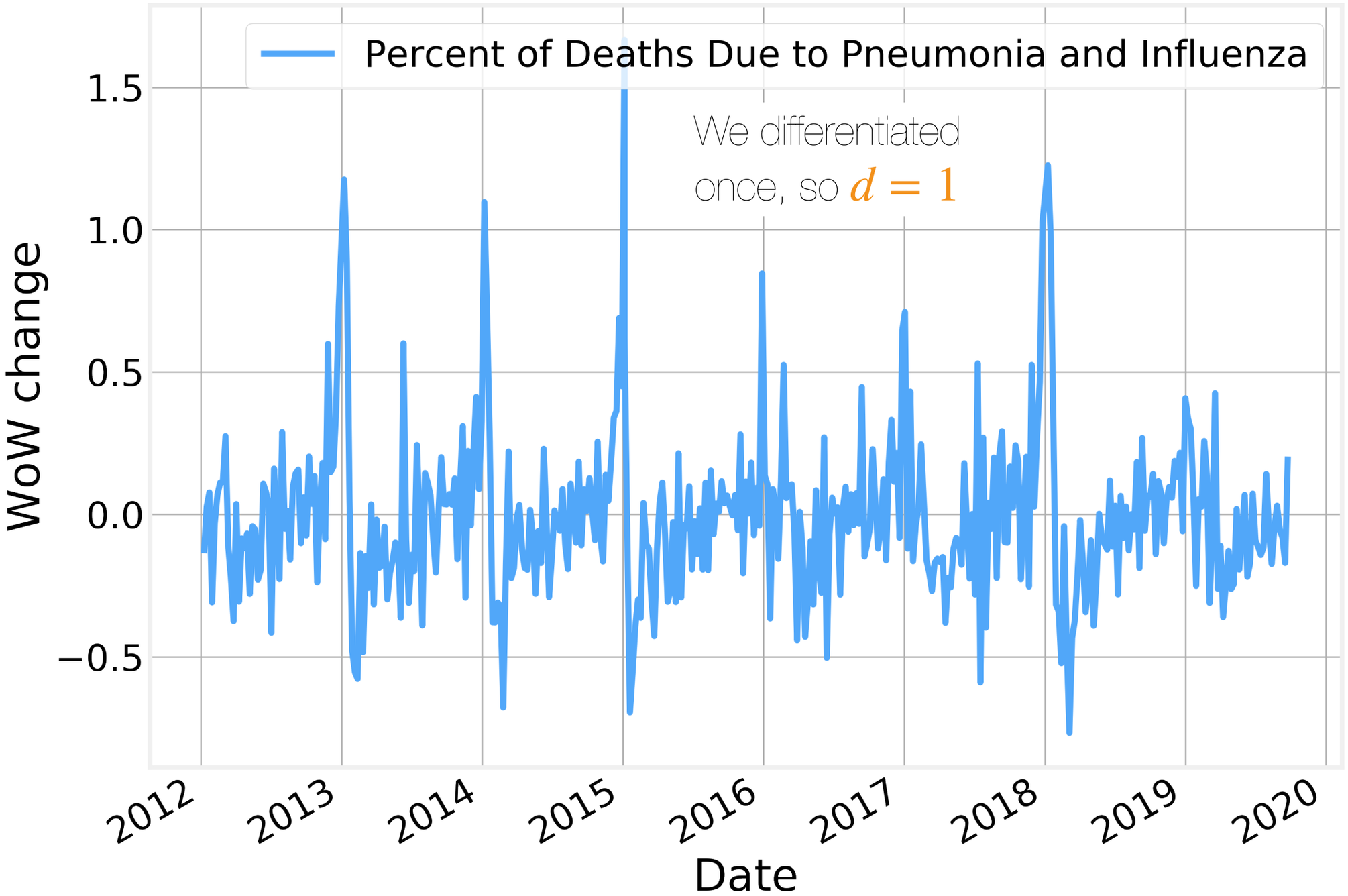
The general procedure to fit an ARIMA model was originally proposed by Box-Jenkins

Let us consider an example

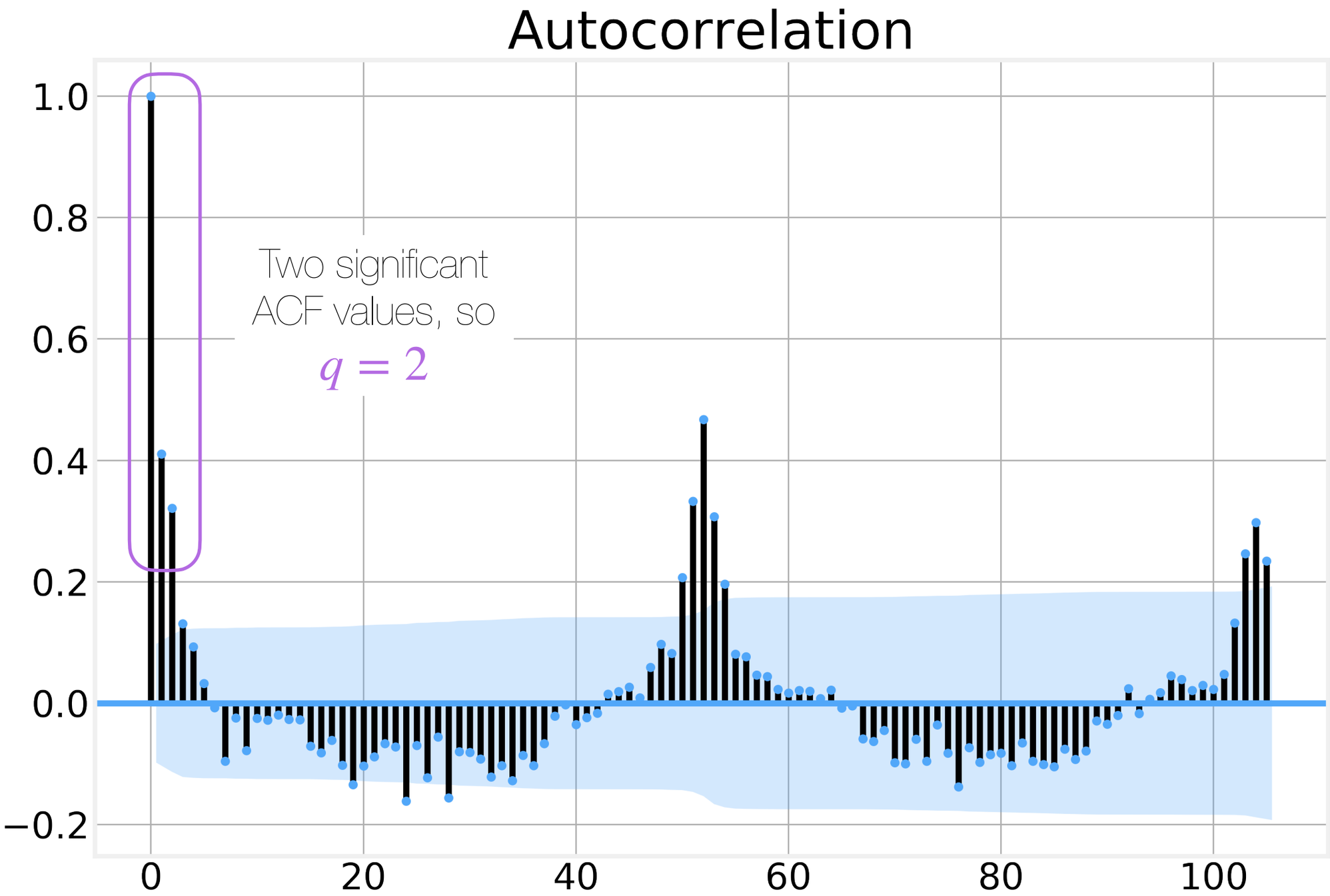
Visualize Time Series



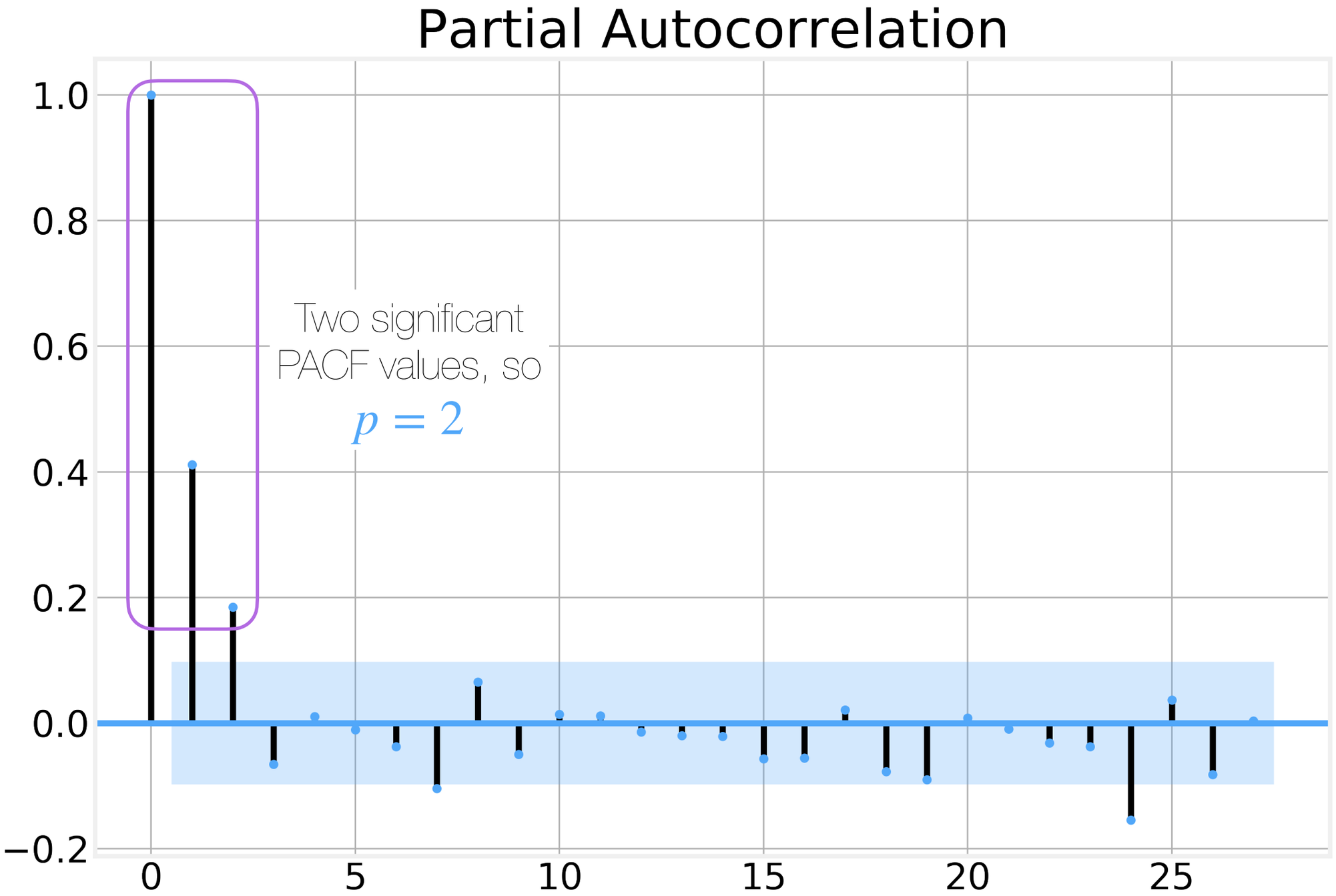
Stationarize Time Series



Plot ACF


Autocorrelation
1.0
0.8
0.6
0.4
0.2
0.0
−0.2
0
20
40
60
80
100
Two significant
ACF values, so
$q = 2$

Plot PACF


Partial Autocorrelation
1.0
0.8
0.6
0.4
0.2
0.0
−0.2
0
5
10
15
20
25
Two significant
PACF values, so
$p = 2$

Build ARIMA Model

- Using statsmodels we just have to do:

```
model = sm.tsa.ARIMA(ILI, (2, 1, 2))
results = model.fit()
```

Build ARIMA Model

- Using statsmodels we just have to do:

```
model = sm.tsa.ARIMA(ILI, (2, 1, 2))
results = model.fit()
```

- From where we can easily extract the parameter values using summary():

- and, finally, using plot_predict() we obtain

ARIMA Model Results

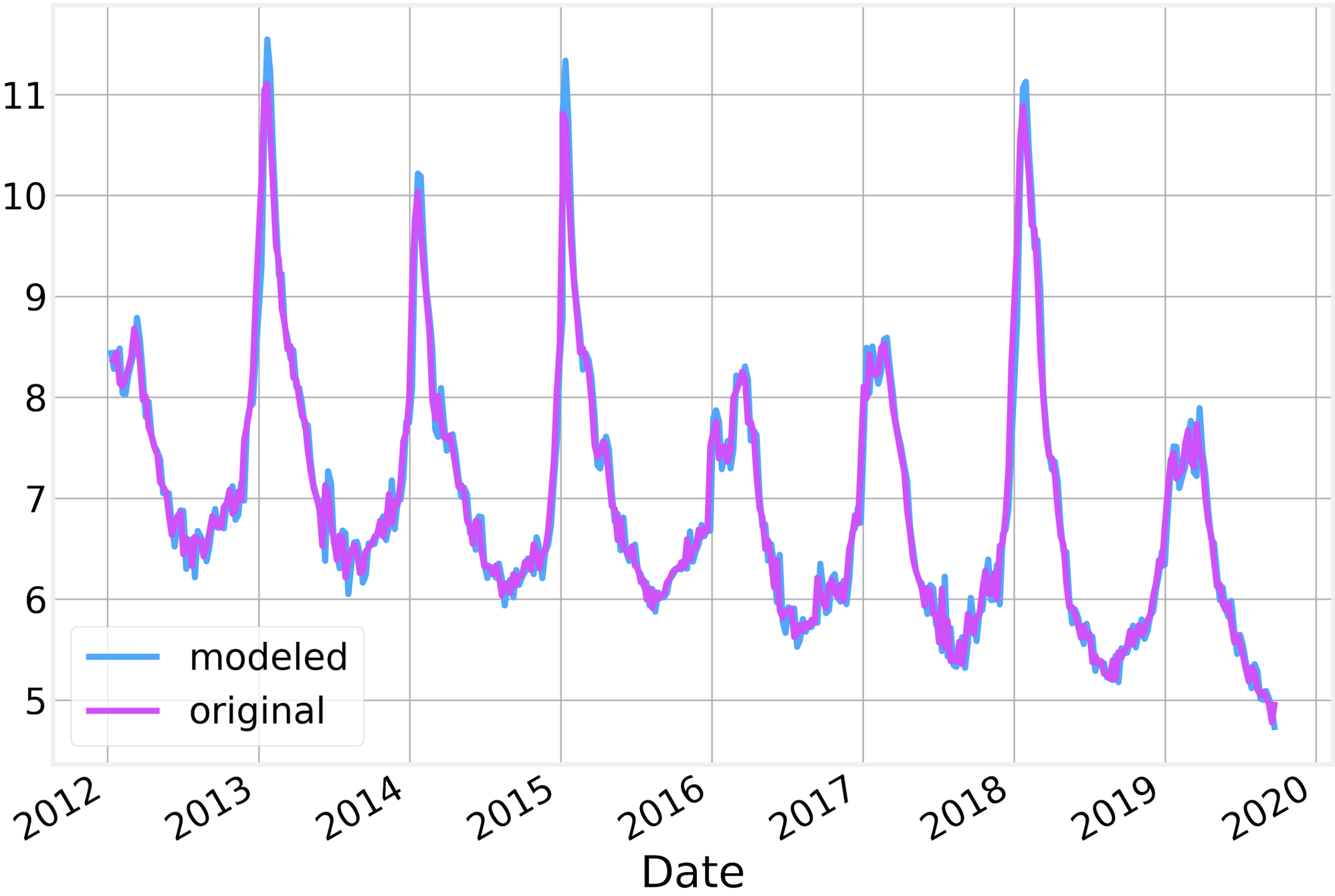
Dep. Variable:	D.Percent of Deaths Due to Pneumonia and Influenza	**No. Observations:**	402
Model:	ARIMA(2, 1, 2)	**Log Likelihood**	-18.777
Method:	css-mle	**S.D. of innovations**	0.253
Date:	Sun, 09 Aug 2020	**AIC**	49.553
Time:	20:15:20	**BIC**	73.532
Sample:	1	**HQIC**	59.047

	coef	std err	z	P>|z|	[0.025	0.975]
const	-0.0086	0.025	-0.346	0.729	-0.057	0.040
ar.L1.D.Percent of Deaths Due to Pneumonia and Influenza	0.0506	0.561	0.090	0.928	-1.050	1.151
ar.L2.D.Percent of Deaths Due to Pneumonia and Influenza	0.2785	0.476	0.585	0.559	-0.655	1.212
ma.L1.D.Percent of Deaths Due to Pneumonia and Influenza	0.2971	0.564	0.527	0.598	-0.808	1.403
ma.L2.D.Percent of Deaths Due to Pneumonia and Influenza	0.0261	0.318	0.082	0.935	-0.597	0.649

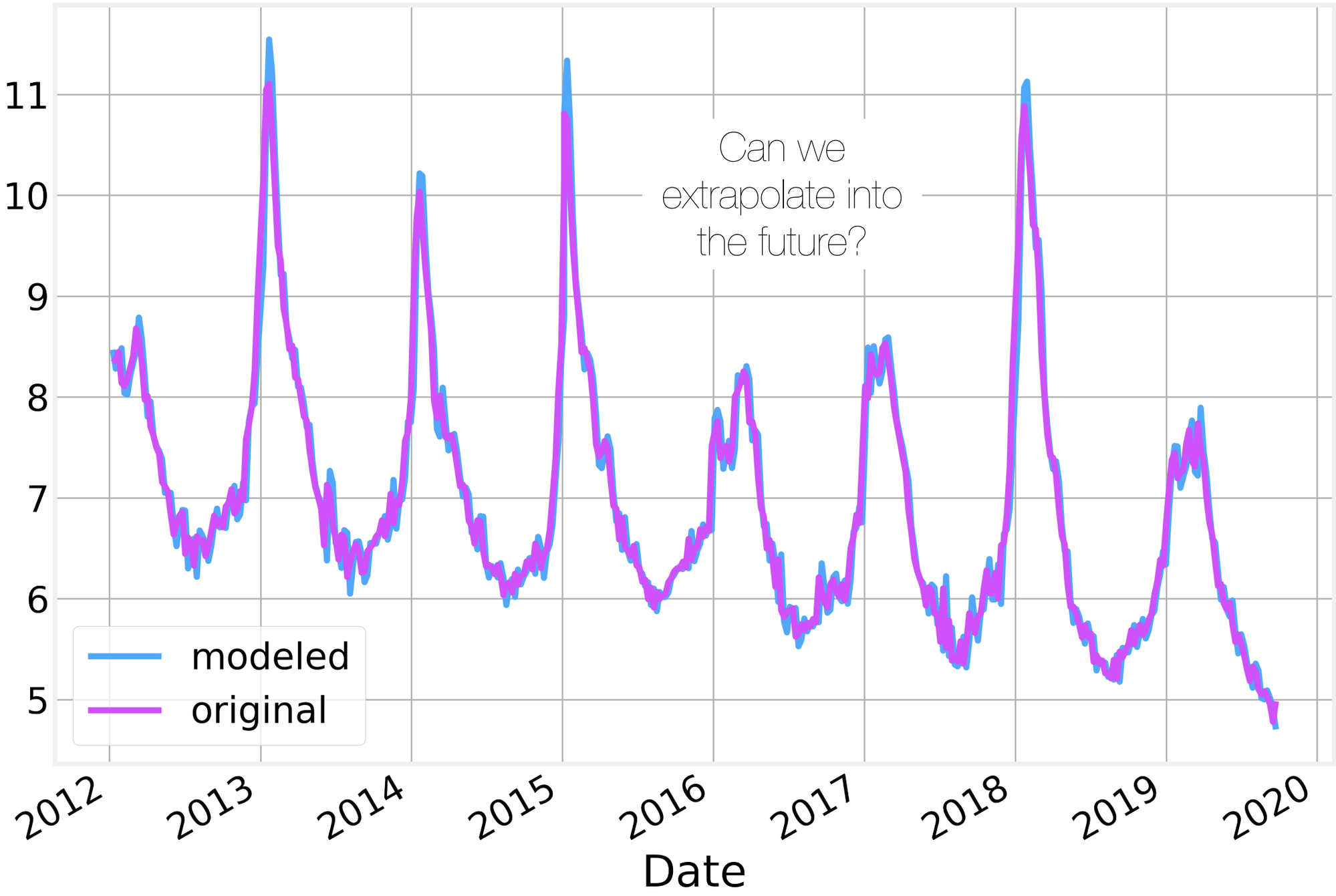
Roots

	Real	Imaginary	Modulus	Frequency
AR.1	1.8061	+0.0000j	1.8061	0.0000
AR.2	-1.9878	+0.0000j	1.9878	0.5000
MA.1	-5.7000	-2.4260j	6.1948	-0.4360
MA.2	-5.7000	+2.4260j	6.1948	0.4360

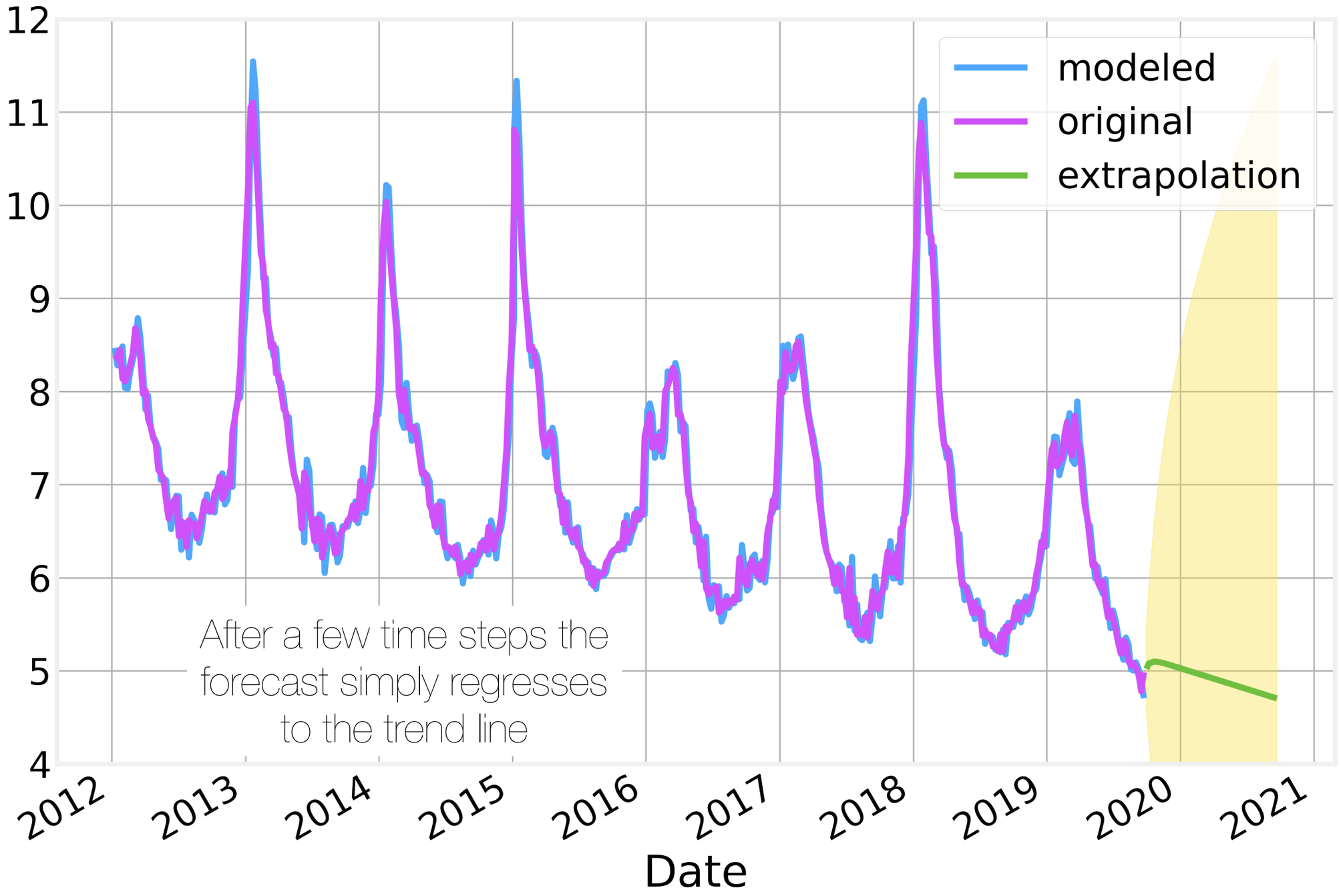
Forecast


11
10
9
8
7
6
5
modeled
original
2012
2013
2014
2015
2016
2017
2018
2019
2020
Date

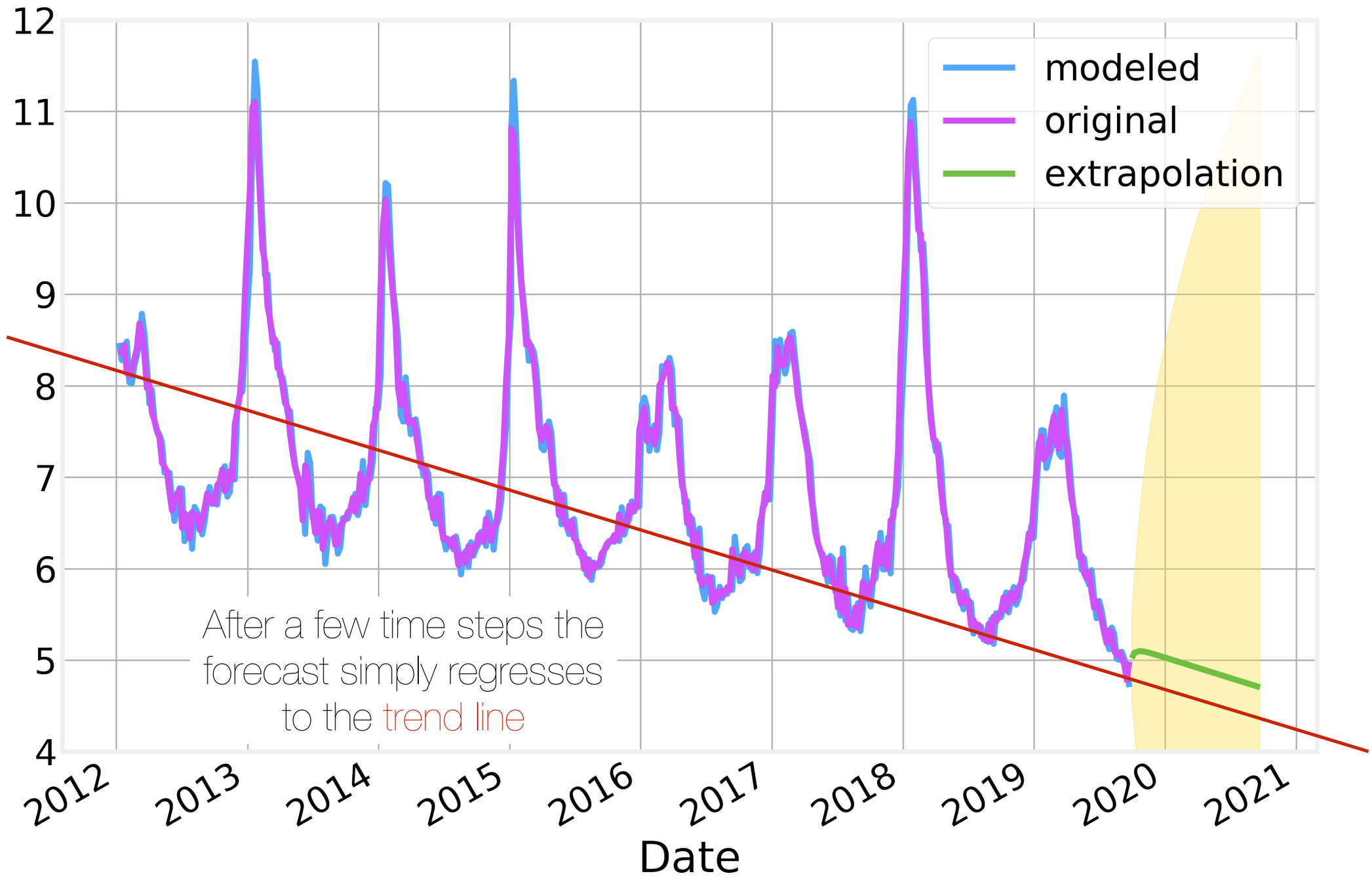
Forecast


Can we
extrapolate into
the future?
11
10
9
8
7
6
5
modeled
original
2012
2013
2014
2015
2016
2017
2018
2019
2020
Date

Forecast


12
11
10
9
8
7
6
5
4
2012
2013
2014
2015
2016
2017
2018
2019
2020
2021
Date
modeled
original
extrapolation
After a few time steps the forecast simply regresses to the trend line

Forecast


12
11
10
9
8
7
6
5
4
2012
2013
2014
2015
2016
2017
2018
2019
2020
2021
Date
modeled
original
extrapolation
After a few time steps the forecast simply regresses to the trend line

Seasonal Models

- The problem with our previous approach is that our model isn't taking into account the full seasonal structure of the data.

- A simple approach to taking seasonal variations into account is to train the ARIMA model just on the residuals of the time series, after properly decomposing it.

    - We can recover the original signal by "manually" adding the trend and seasonality we obtained in our decomposition

- A more sophisticated approach is to use the SARIMA (Seasonal ARIMA) class of models.

- SARIMA explicitly models the seasonal variations as a full fledged ARIMA model at the cost of 4 extra parameters: $SARIMA\left(p,d,q\right)\times\left(P,D,Q,s\right)$:

    - p, d and q, are defined exactly as above

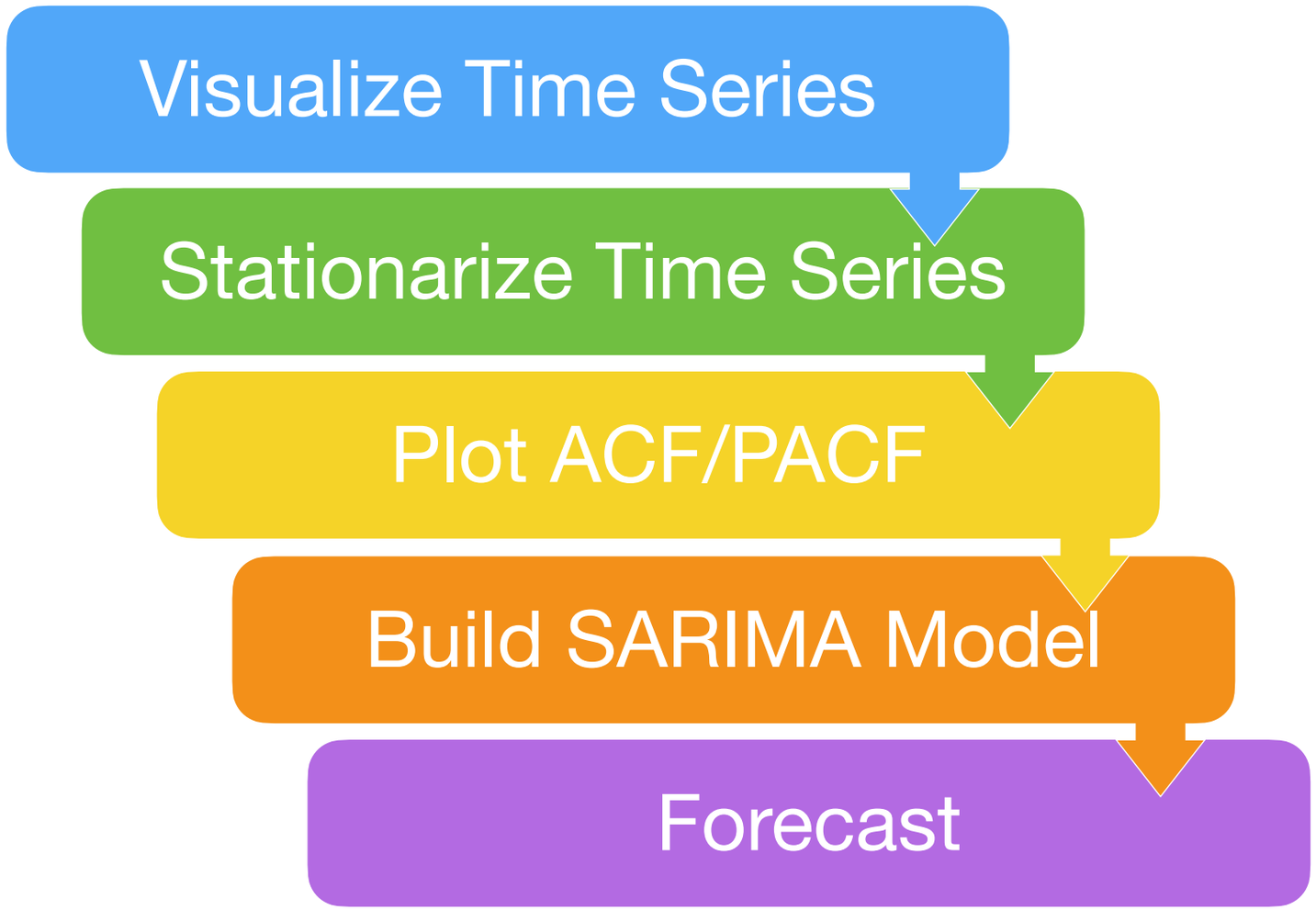
    - s, is the seasonal periodicity

    - P, D and Q, are the parameters of an ARIMA model trained on the s-lagged time series

SARIMA

https://www.statsmodels.org/devel/generated/statsmodels.tsa.statespace.sarimax.SARIMAX.html

- To fit a SARIMA model, we follow a similar procedure to the one described above:

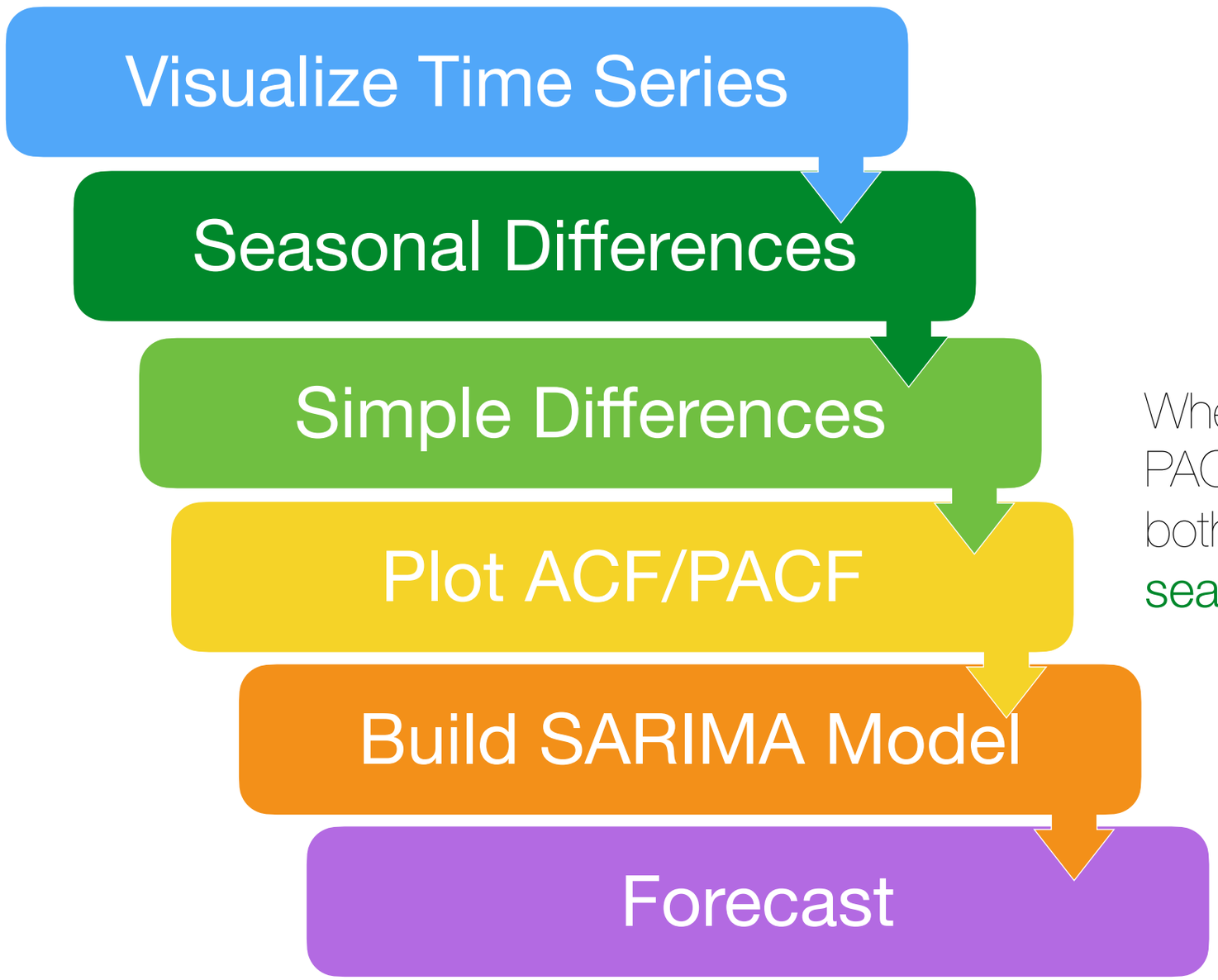


- With one important difference, in the Stationarize step, we differentiate on simple and seasonal lags

SARIMA

https://www.statsmodels.org/devel/generated/statsmodels.tsa.statespace.sarimax.SARIMAX.html

- Our procedure is then:

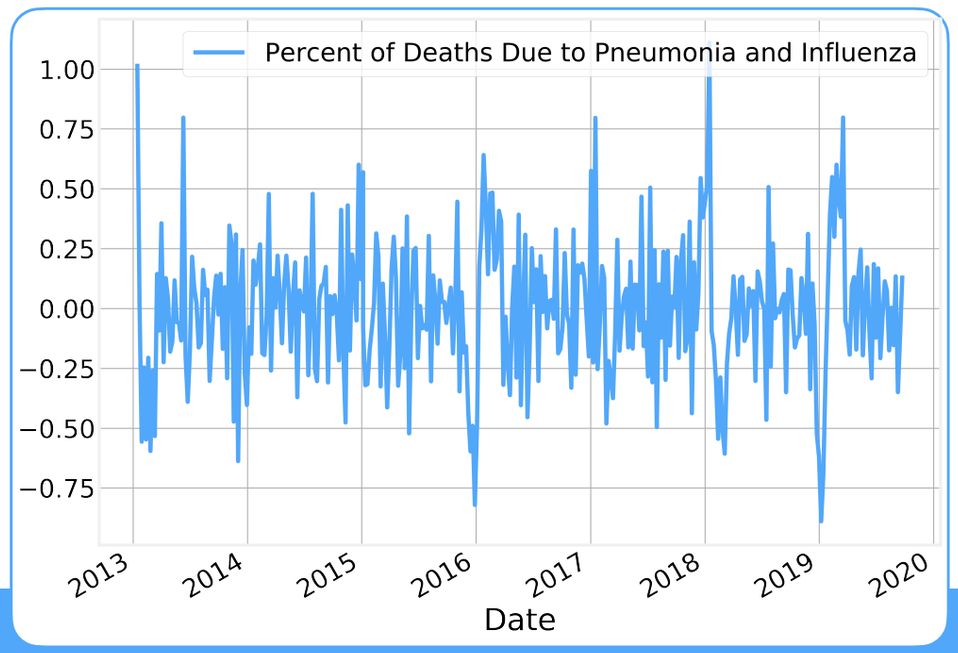


When plotting the ACD/PACF we must look for both simple and seasonal lags

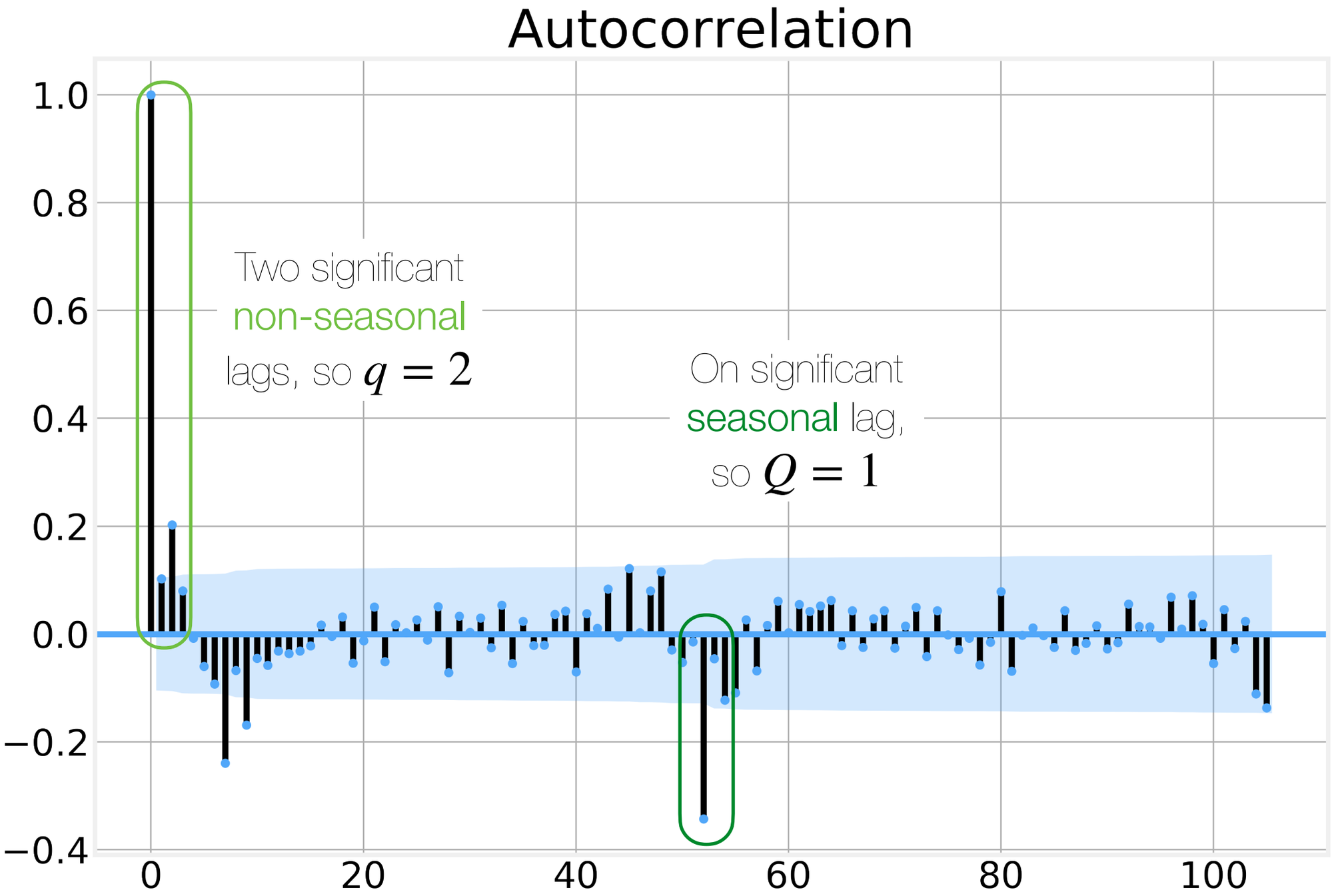
SARIMA

https://www.statsmodels.org/devel/generated/statsmodels.tsa.statespace.sarimax.SARIMAX.html

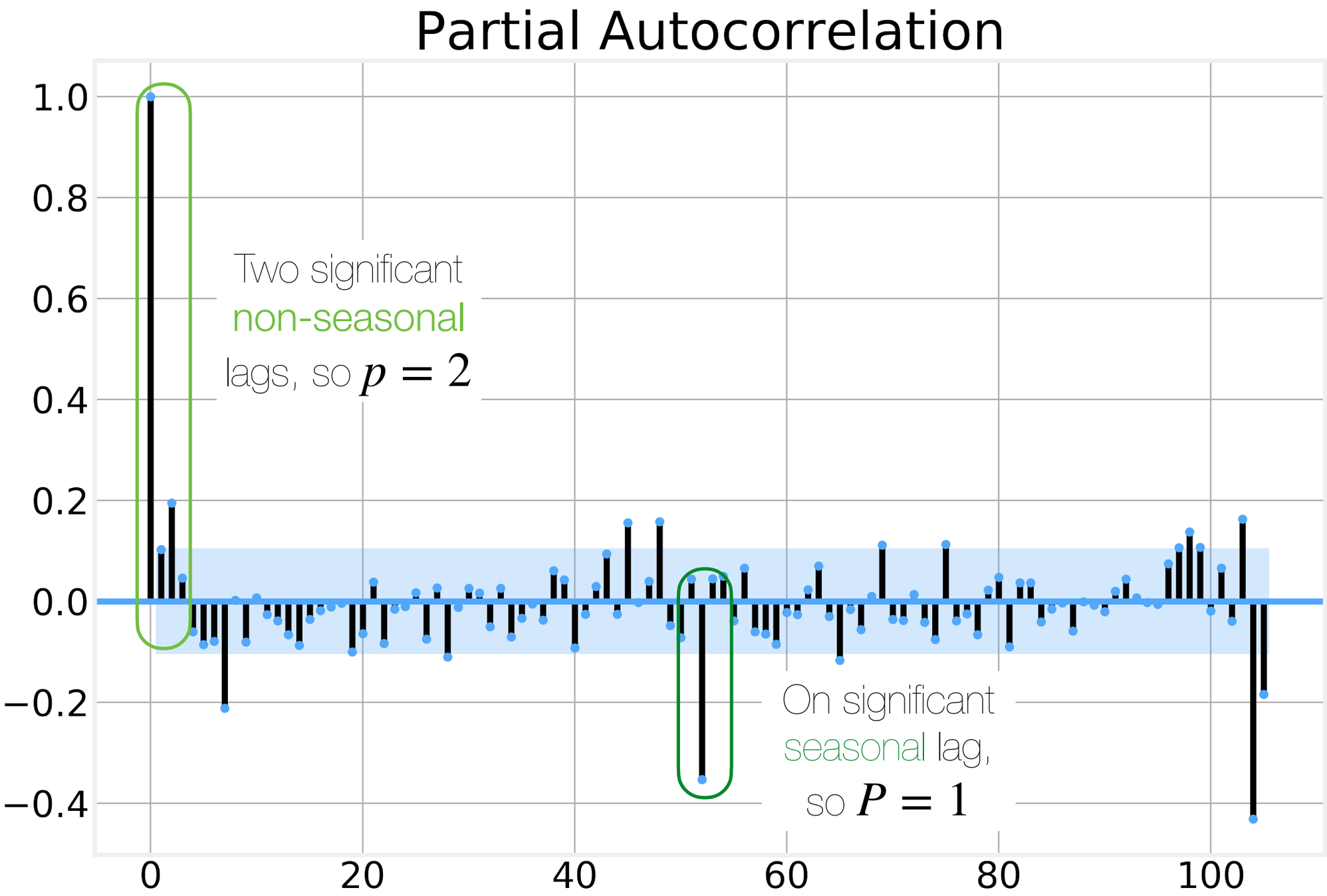
- statsmodels provides the SARIMAX model in the tsa.statemodels package.
- The API is similar to that of the ARIMA class of models, but now we must provide both:
  - order - $(p, d\ q)$ - the usual ARIMA parameters
  - seasonal_order - (P, D, Q, s) - the parameters specifying the seasonal component
- With this approach, we are now able to perform significantly better forecasts!
- The first step is to take the seasonal (52) followed but the non-seasonal (1) differences.



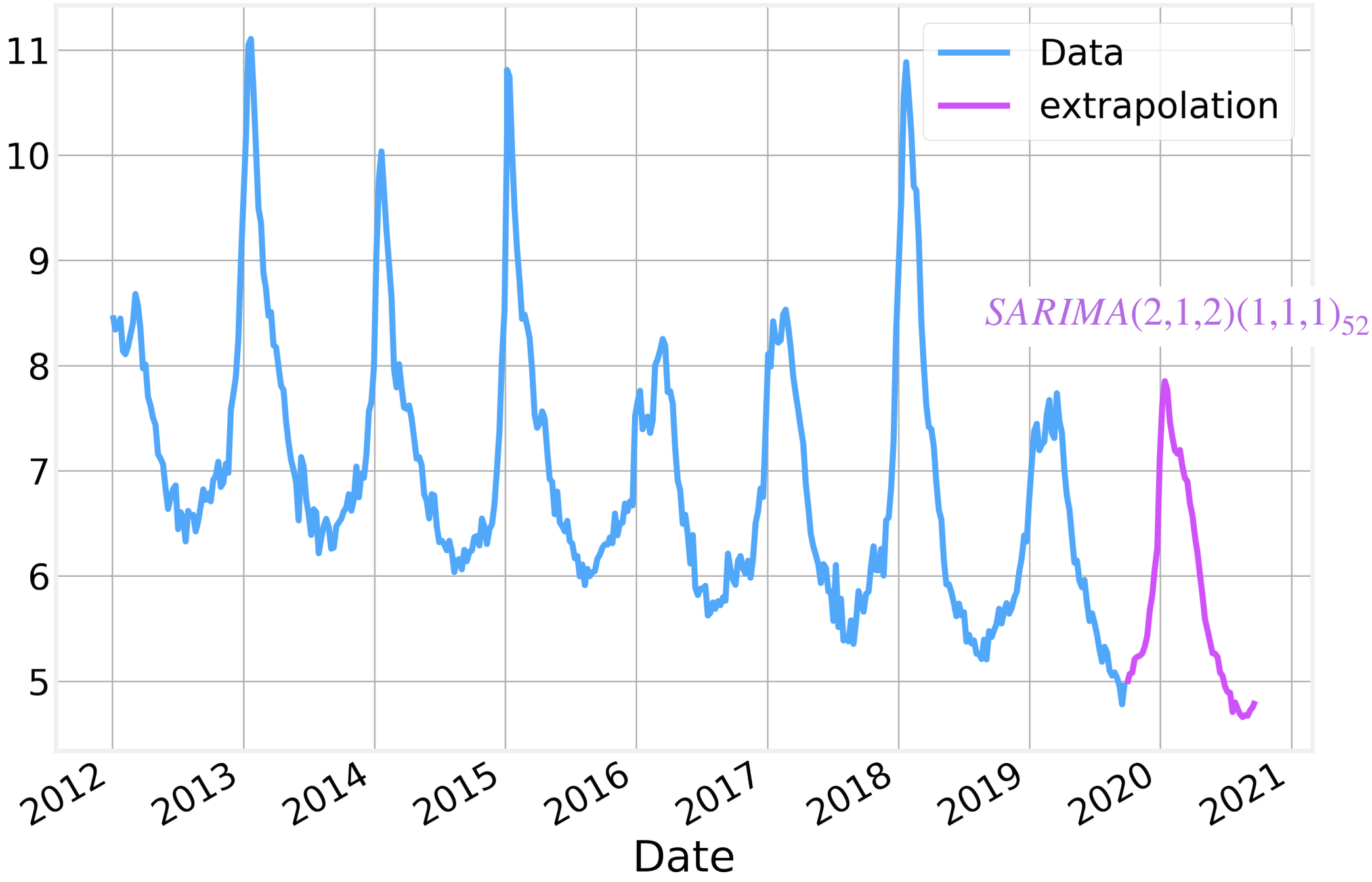
Plot ACF


Autocorrelation
1.0
0.8
0.6
0.4
0.2
0.0
−0.2
−0.4
0
20
40
60
80
100
Two significant non-seasonal lags, so $q = 2$
On significant seasonal lag, so $Q = 1$

Plot PACF


Partial Autocorrelation
1.0
0.8
0.6
0.4
0.2
0.0
−0.2
−0.4
0
20
40
60
80
100
Two significant non-seasonal lags, so $p = 2$
On significant seasonal lag, so $P = 1$

Forecast


Data
extrapolation
$SARIMA(2,1,2)(1,1,1)_{52}$
11
10
9
8
7
6
5
2012
2013
2014
2015
2016
2017
2018
2019
2020
2021
Date

Code - ARIMA Models

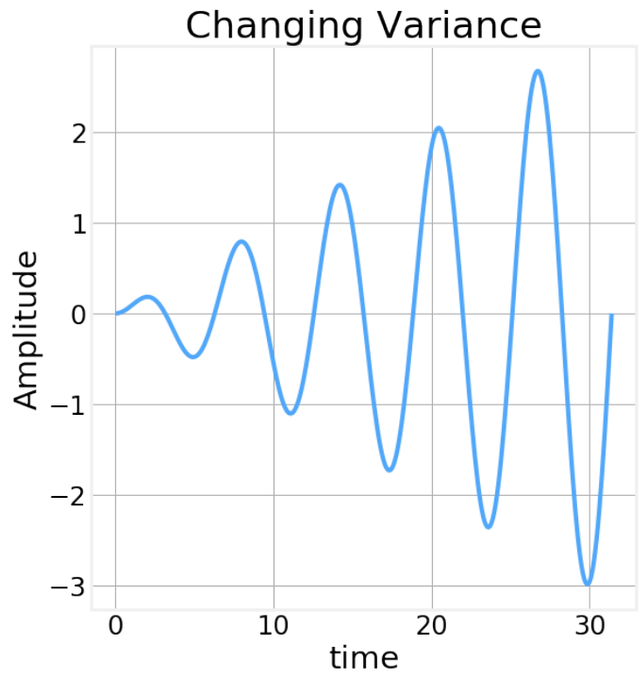
https://github.com/DataForScience/AdvancedTimeseries

Lesson IV:
ARCH Models

Heteroscedasticity

https://en.wikipedia.org/wiki/Heteroscedasticity

- Some time series display **variance that changes in time**:



The temporal behavior of Variance is not required to be monotonous.

- This phenomenon is called Heteroscedasticity

- This kind of time series required a specific class of models from the ones we have considered so far

Heteroscedastic Models

- ARIMA class models can be written as:

$$x_t = f\left(x_{t-1}, \cdots\right)$$

- Heteroskedastic Models, referred as ARCH models, can be interpreted as being a simple AR model applied to the variance of a series.

$$Var\left(x_t \,|\, x_{t-1}\right) \equiv \sigma_t^2 = f\left(x_{t-1}, \cdots\right)$$

- In the simplest case of the Autoregressive Model, AR(1) we have:

$$x_t = \mu + \sigma_t \epsilon_t$$

- With

$$\sigma_t^2 = \alpha_0 + \alpha_1 x_{t-1}^2$$

- We obtain the order one Autoregressive Conditionally Heteroscedastic Model, ARCH(1) model.

- We further impose the constraints $\alpha_0 > 0$, $\alpha_1 > 0$ and $\alpha_1^2 < 1/3$ to avoid negative variances.

ARCH Model

- The ARCH(1) Model can be written:

$$x_t = \sigma_t \epsilon_t$$

$$\sigma_t^2 = \alpha_0 + \alpha_1 x_{t-1}^2$$

- Where we assume that the stochastic variable ϵ_t is Normally distributed:

$$\epsilon_t \sim \mathcal{N}(0,1)$$

- Under these conditions, we have:

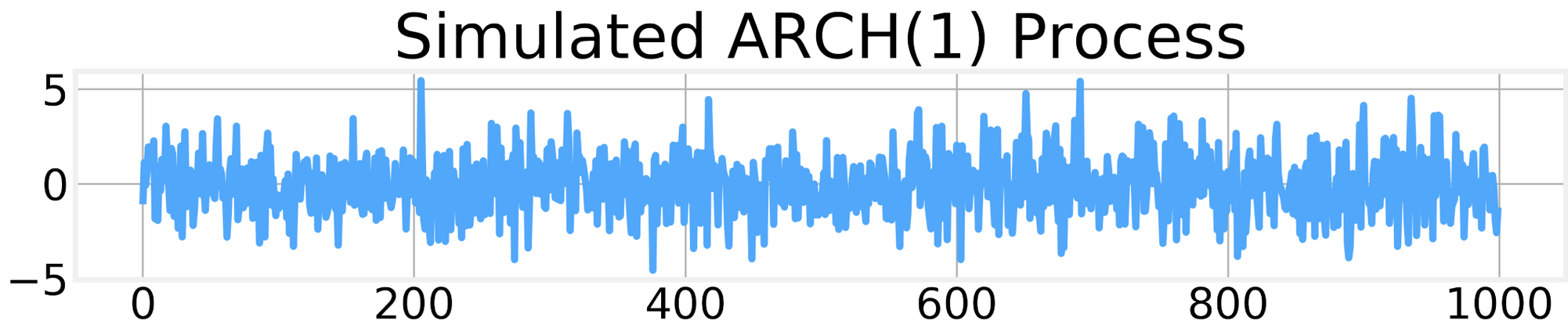
$$x_t^2 = \alpha_0 + \alpha_1 x_{t-1} + \text{error}$$

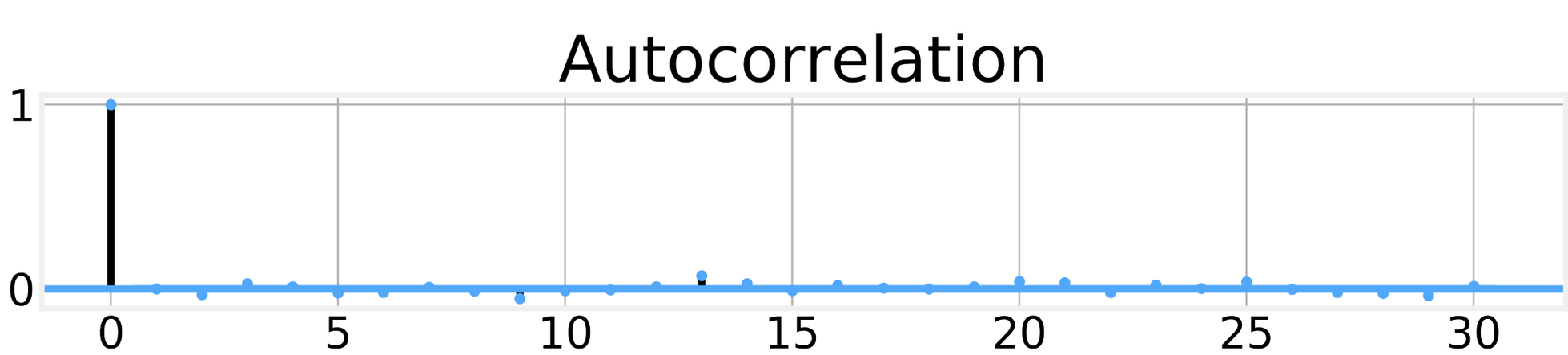
  or, in other words, an AR(1) model for x_t^2! But with a significant error term...

- And x_t is just White Noise!

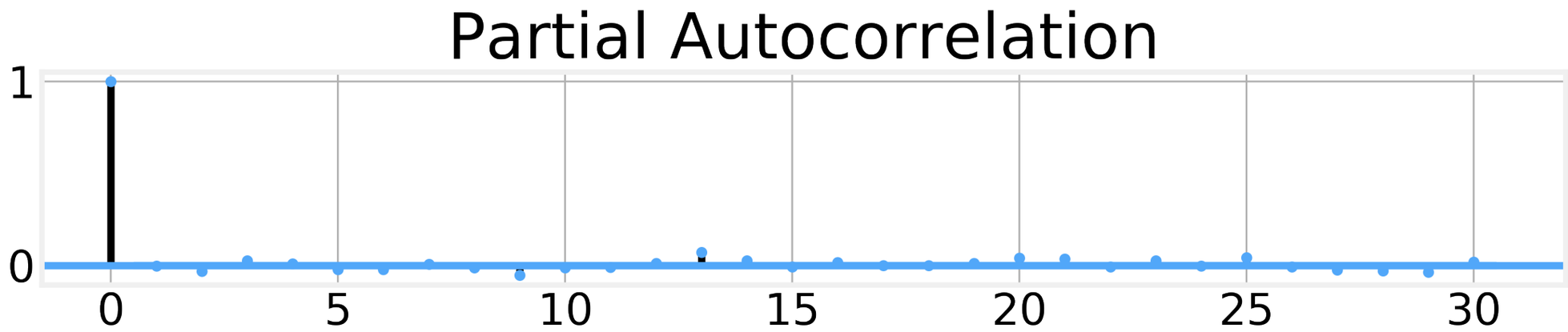
- Let's start by simulating this process in order to understand it better

ARCH Process

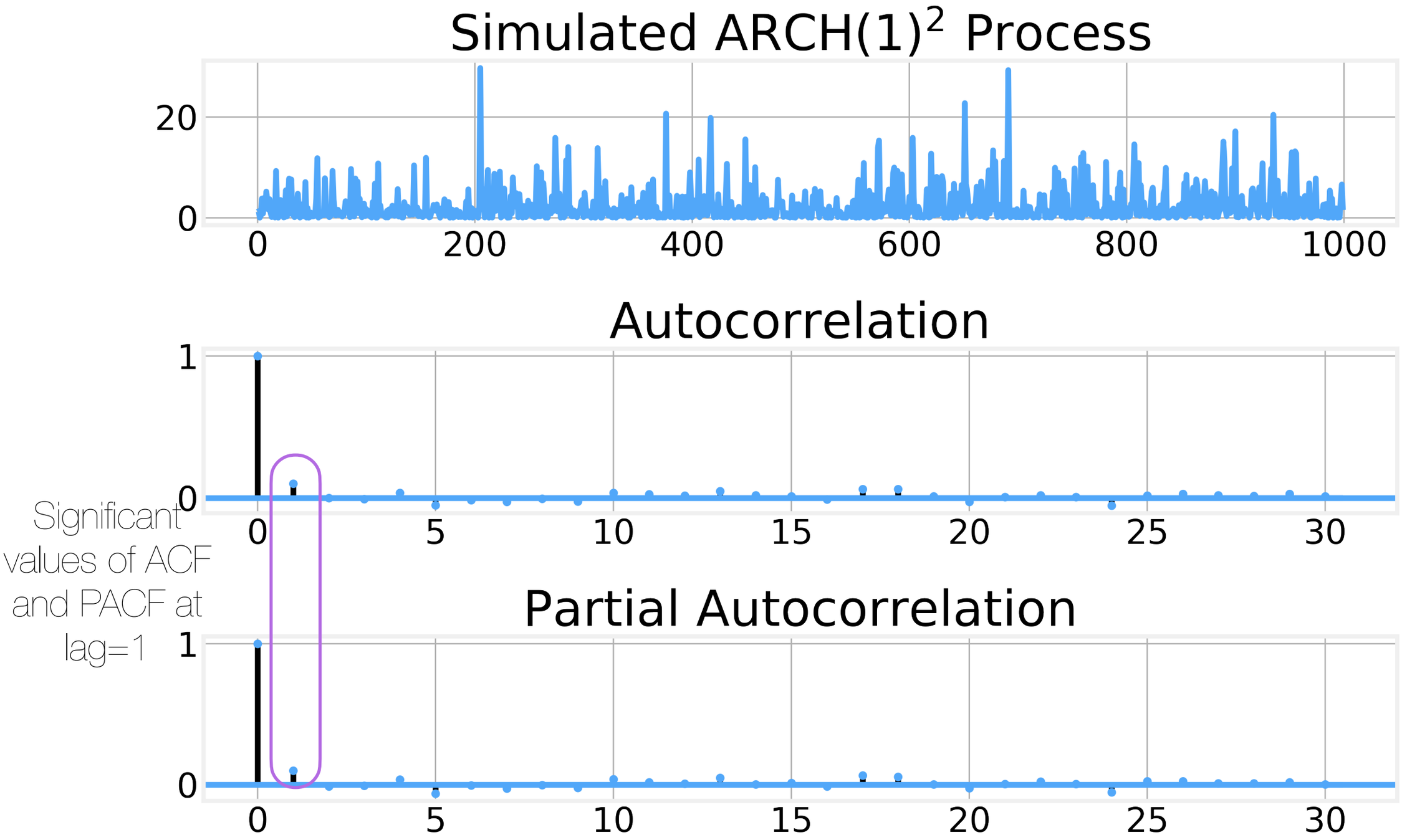




Apparently just White-Noise



ARCH Process



GARCH Model

- There are many variants of the ARCH model, such as the ARCH(m):

$$Var\left(x_t | x_{t-1}, \cdots, x_{t-m}\right) = \sigma_t^2 = \alpha_0 + \sum_{l=1}^{m} \alpha_l x_{t-l}^2$$

- But perhaps the best know variant is the Generalized Autoregressive Conditionally Heteroskedastic (GARCH) Model.

- In GARCH models, we allow for both a AR and a MA components.

- The GARCH model can be defined as:

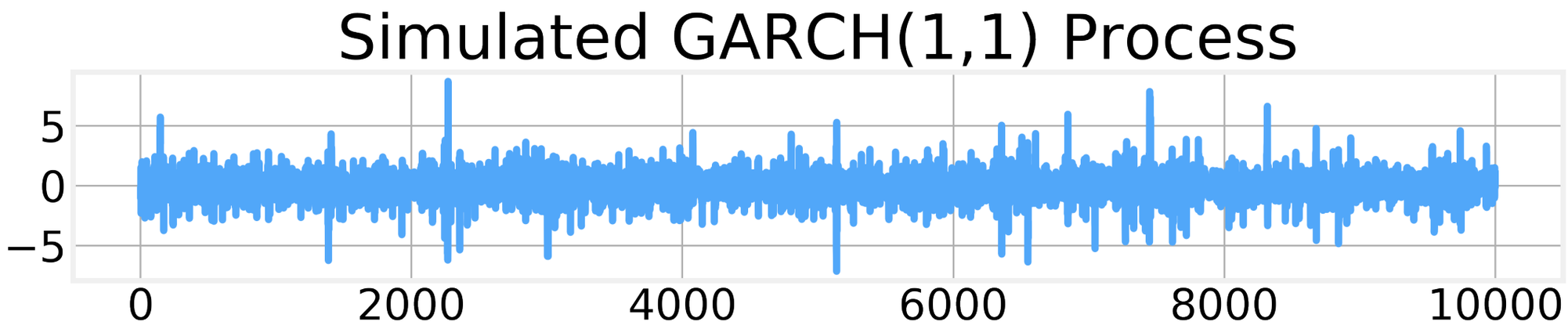
$$x_t = \sigma_t \epsilon_t$$

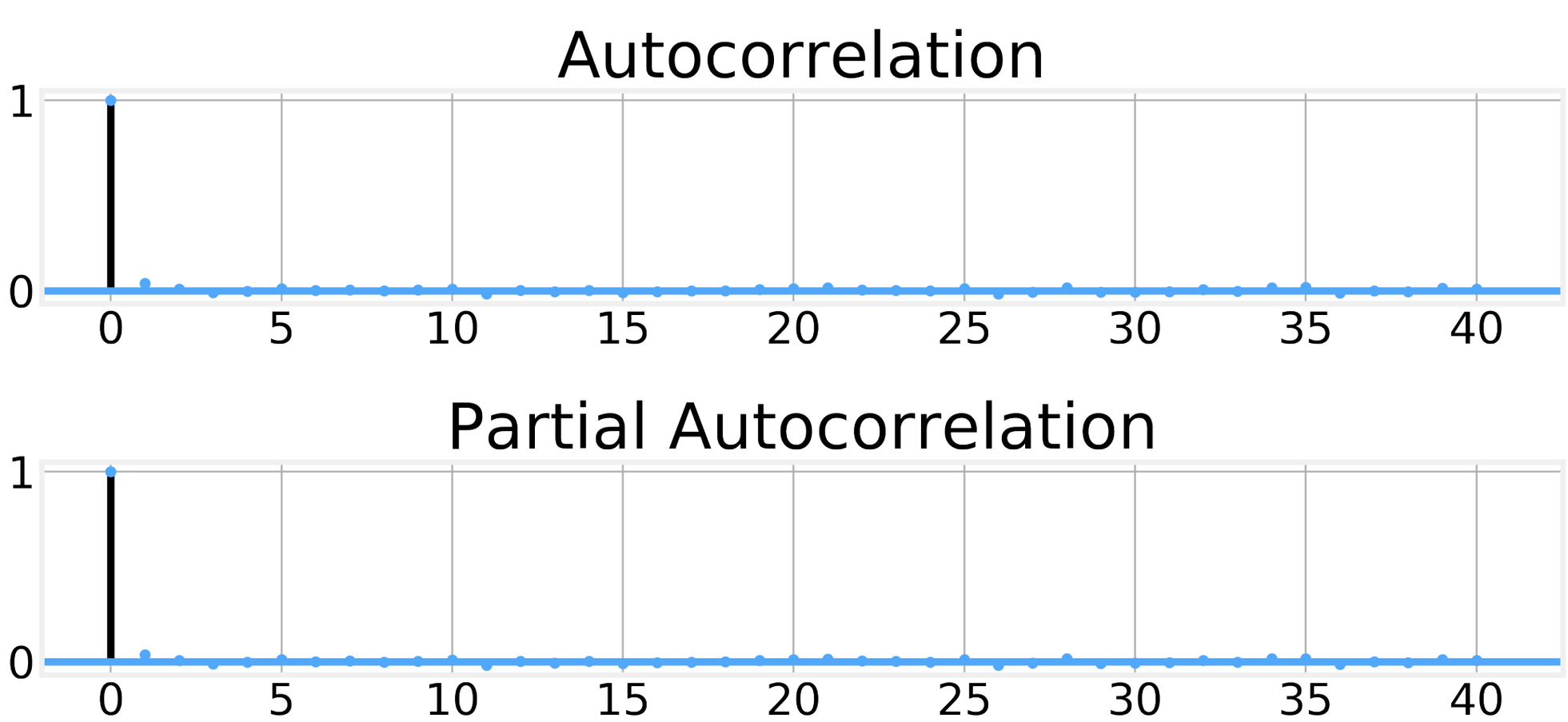
$$\sigma_t^2 = \alpha_0 + \alpha_1 x_{t-1}^2 + \beta_1 \sigma_{t-1}^2$$

- where the β term represent the MA component.

- If we simulate this process, we obtain

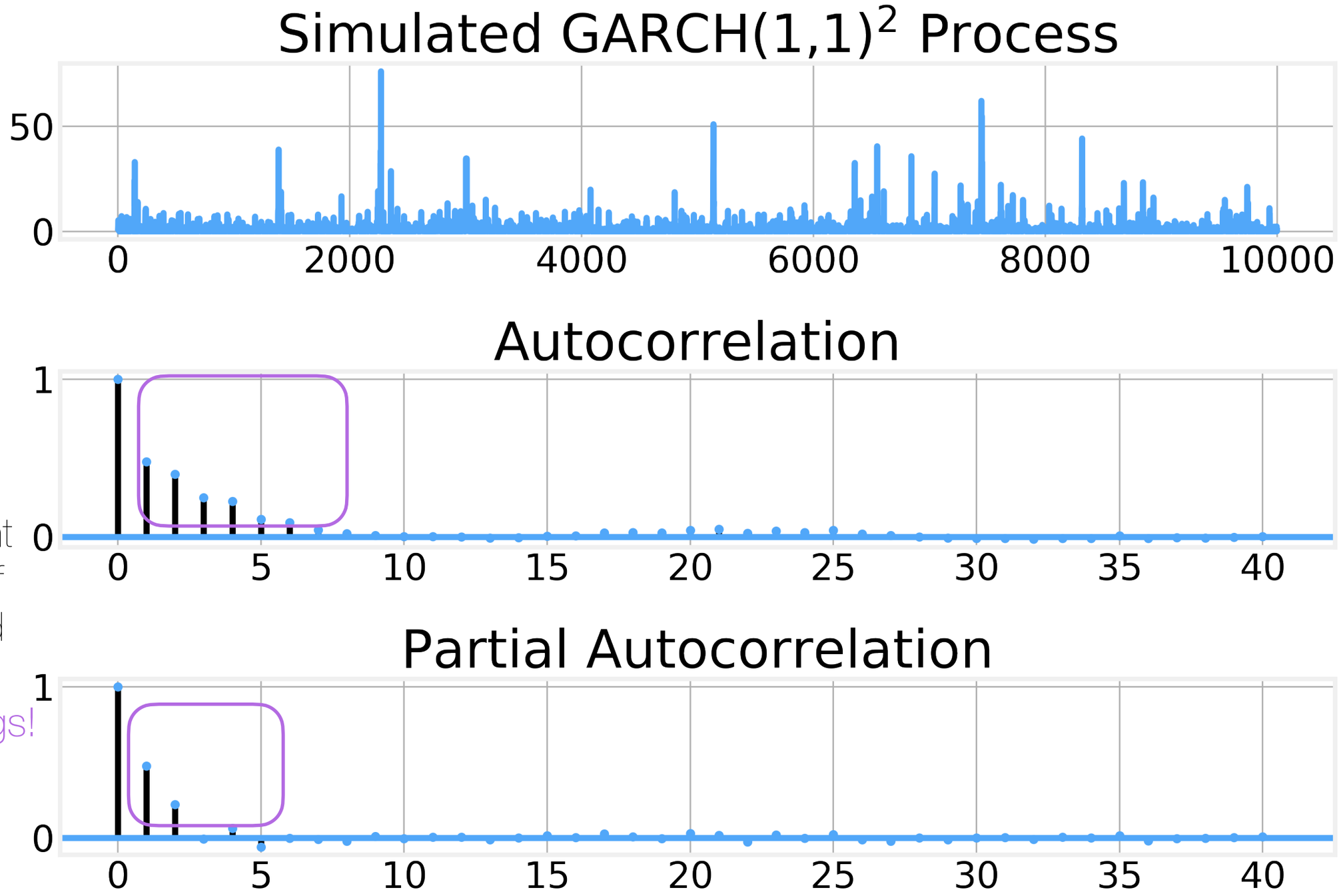
GARCH Model





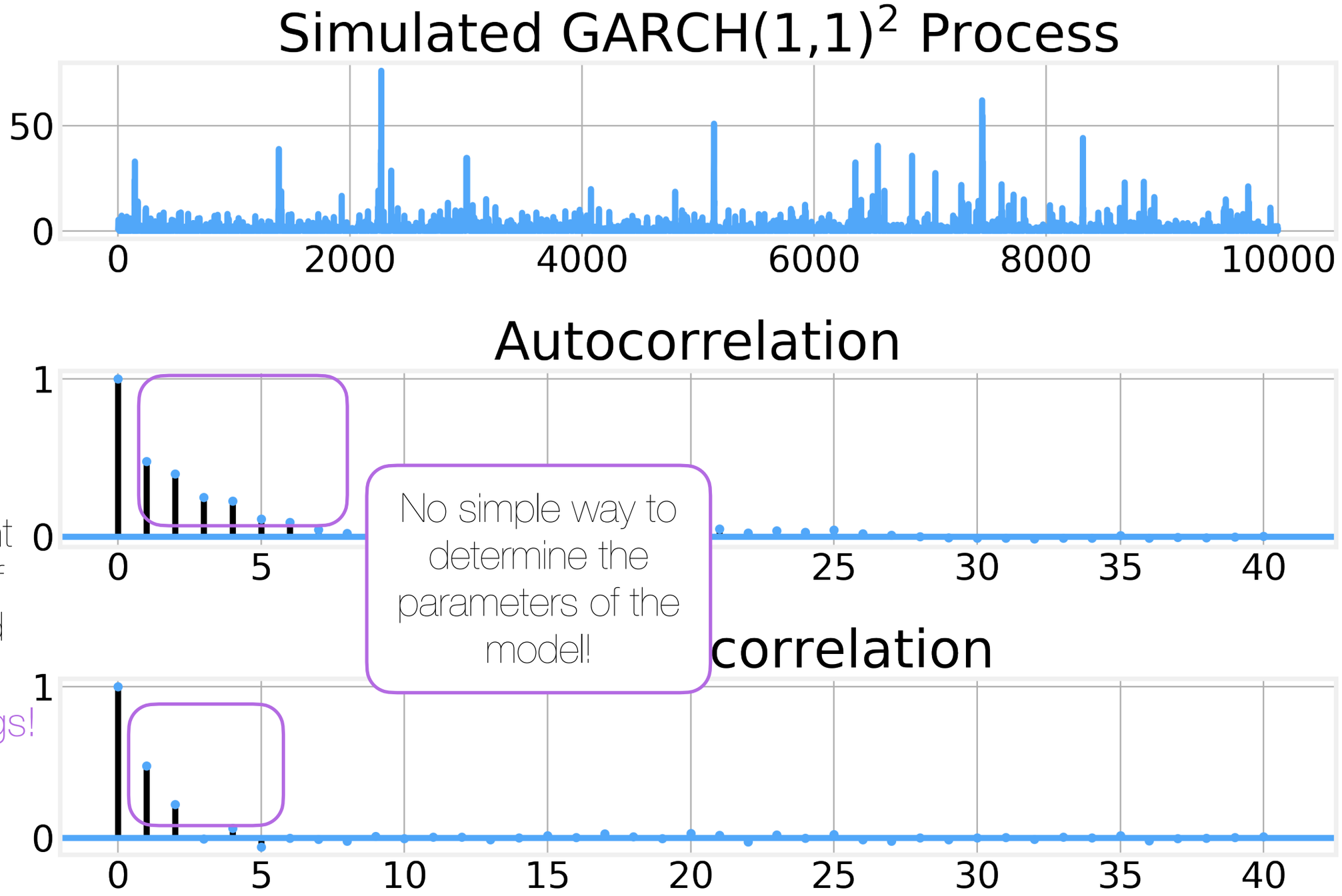
Again just White-Noise

GARCH Model



Significant values of ACF and PACF at various lags!

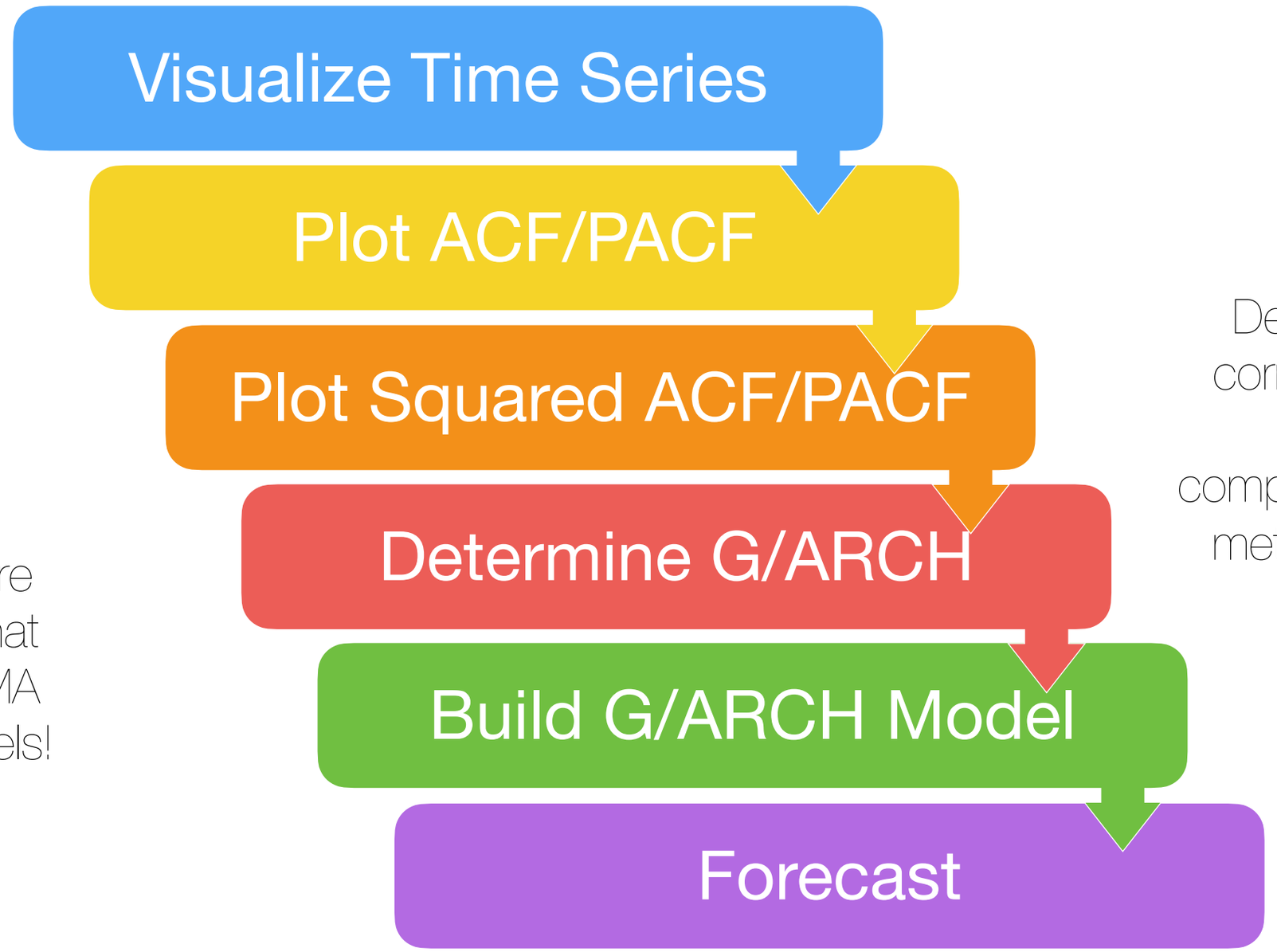
GARCH Model



Significant values of ACF and PACF at various lags!

No simple way to determine the parameters of the model!

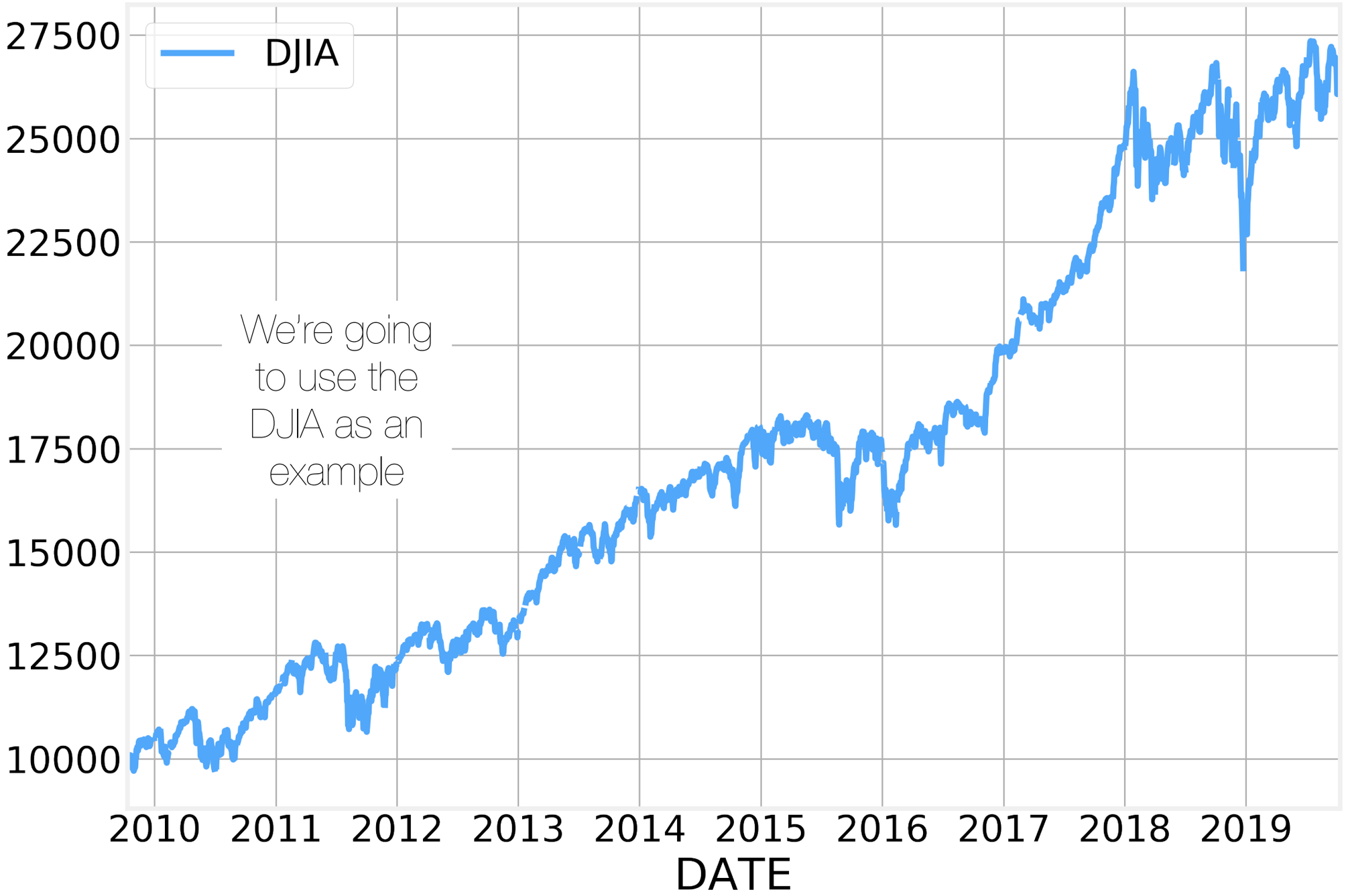
Fitting G/ARCH models



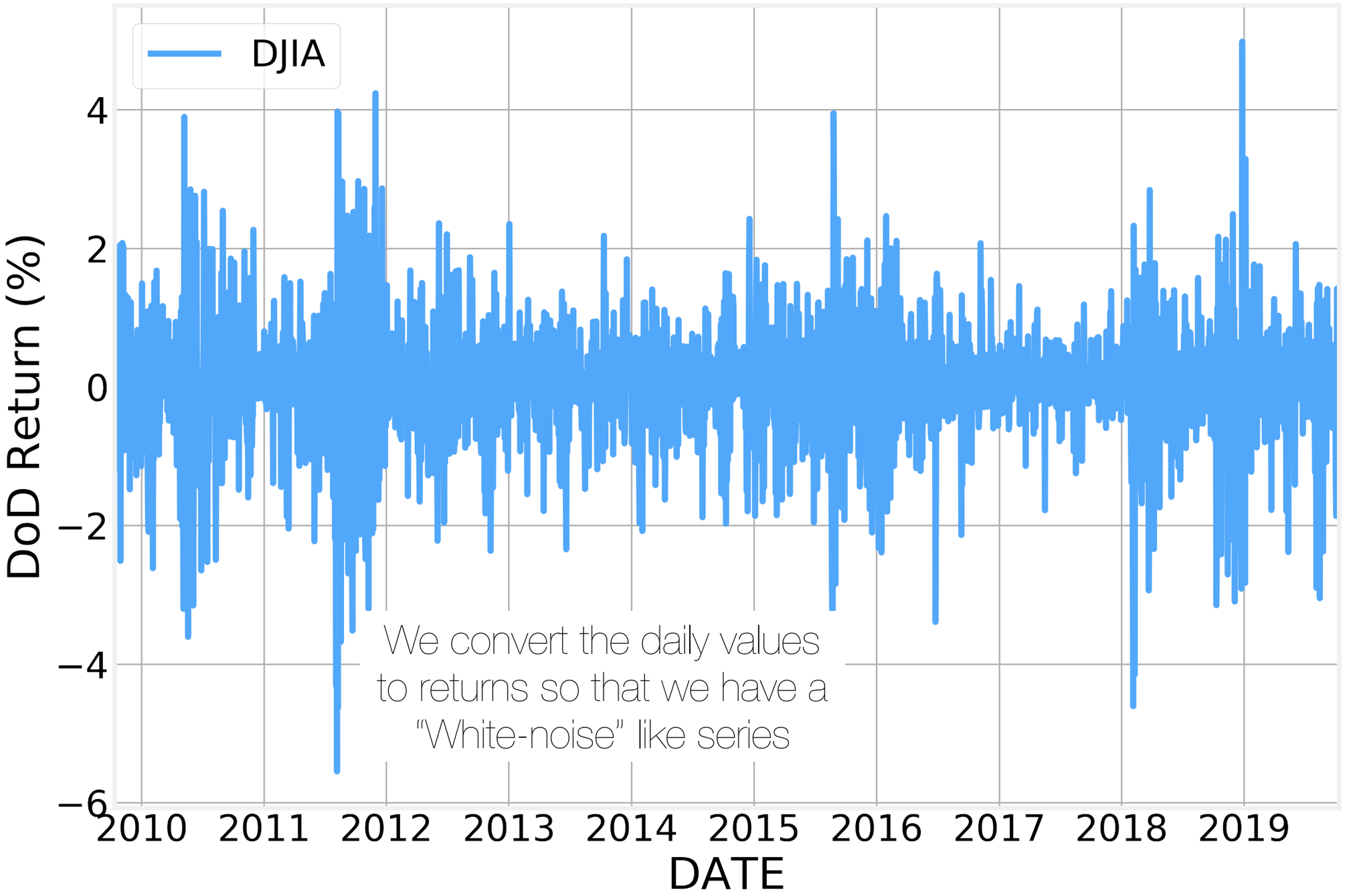
Much more involved that fitting ARIMA class models!

Determine the correct order for model by comparing statistics metrics like AIC!

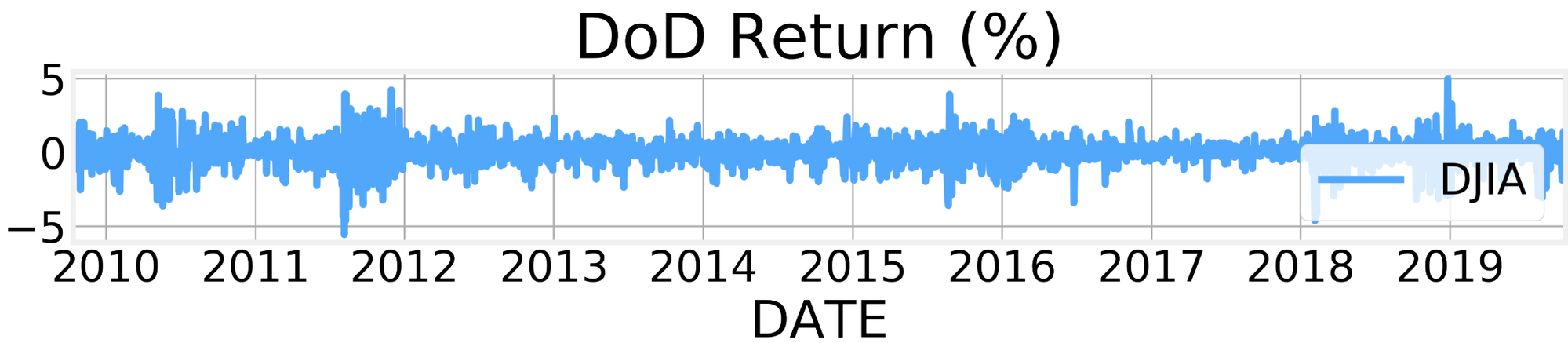
Visualize Time Series

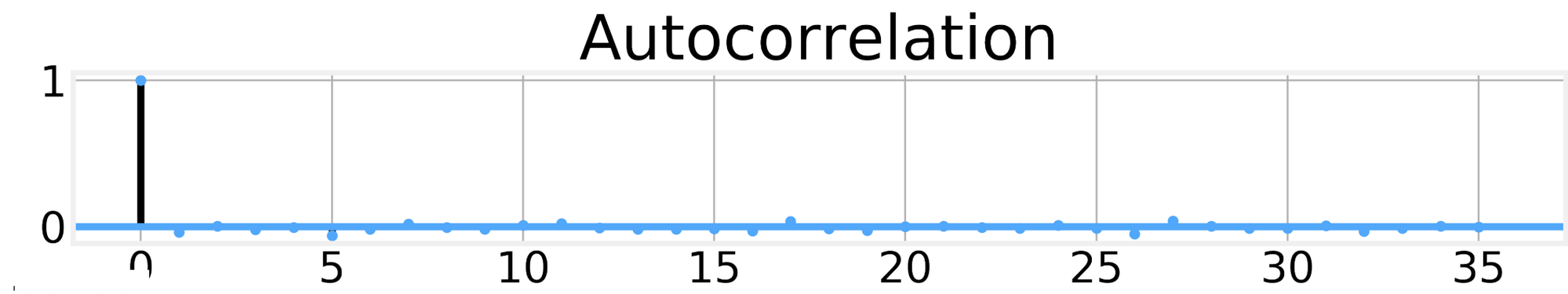


Visualize Time Series

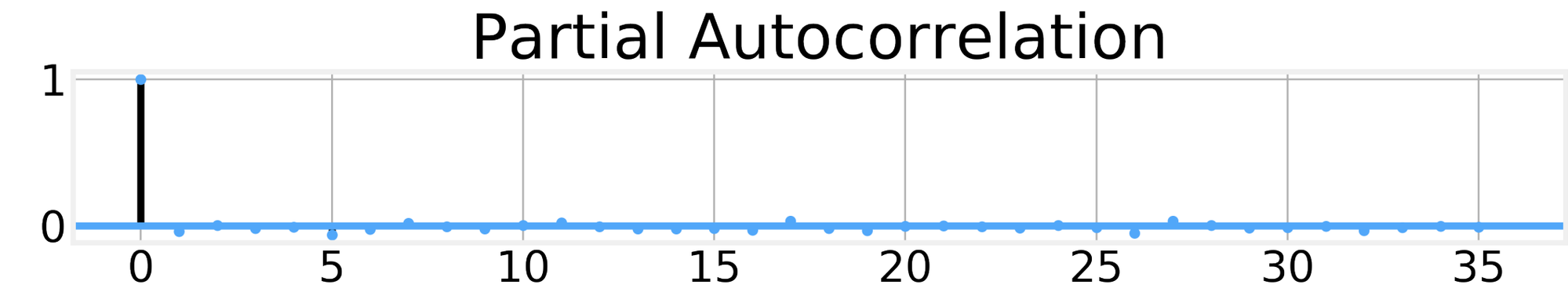


Plot ACF/PACF

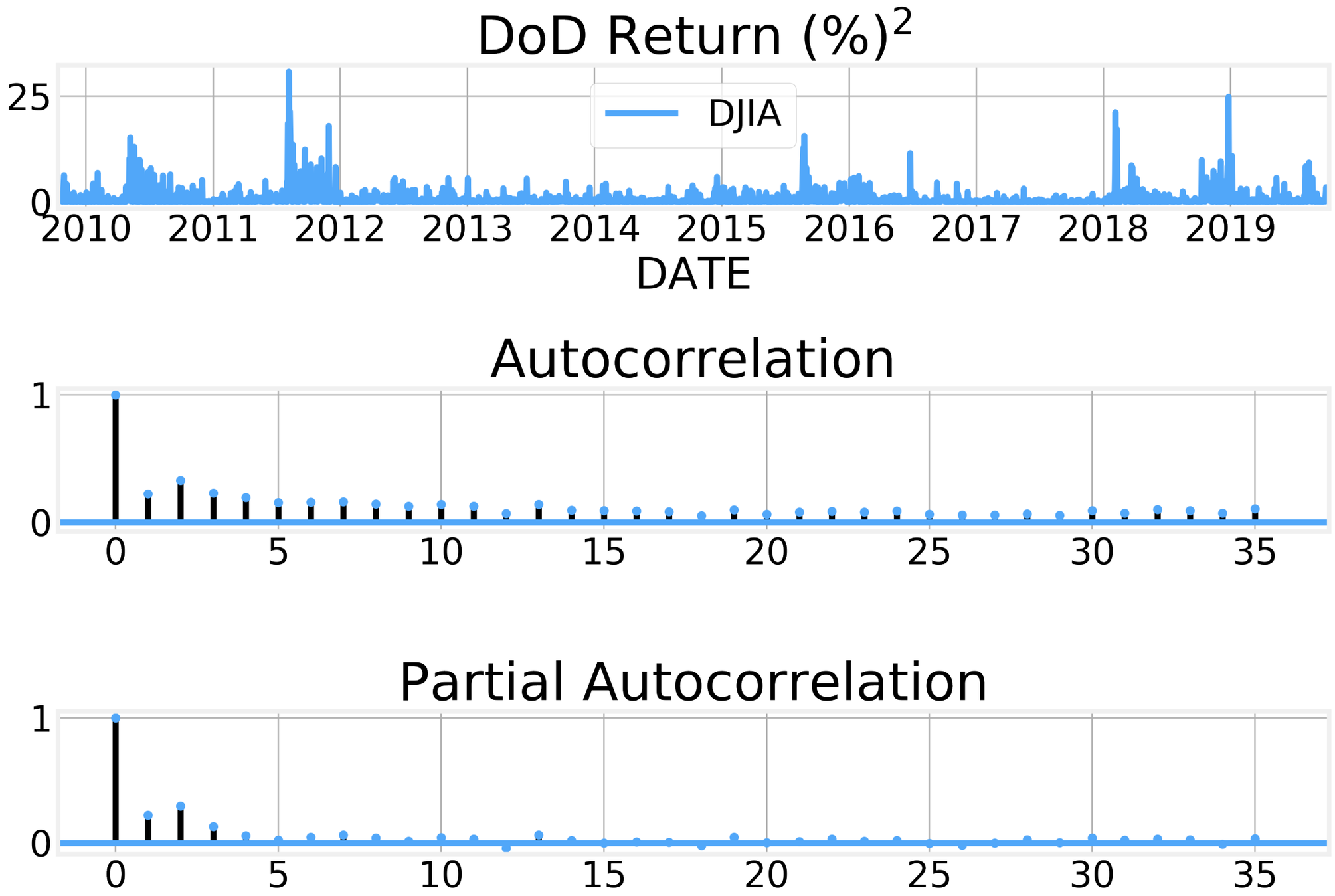




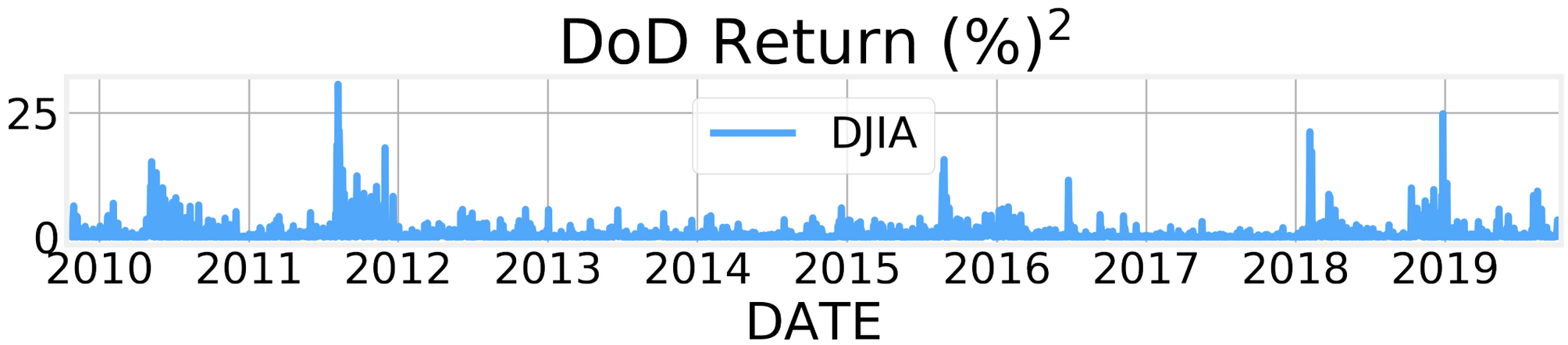
White Noise, as expected!

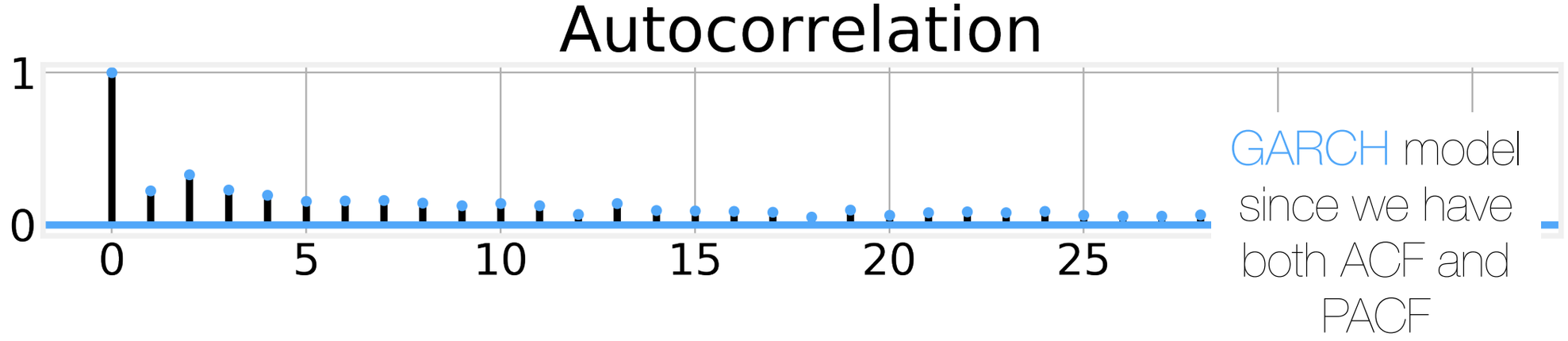


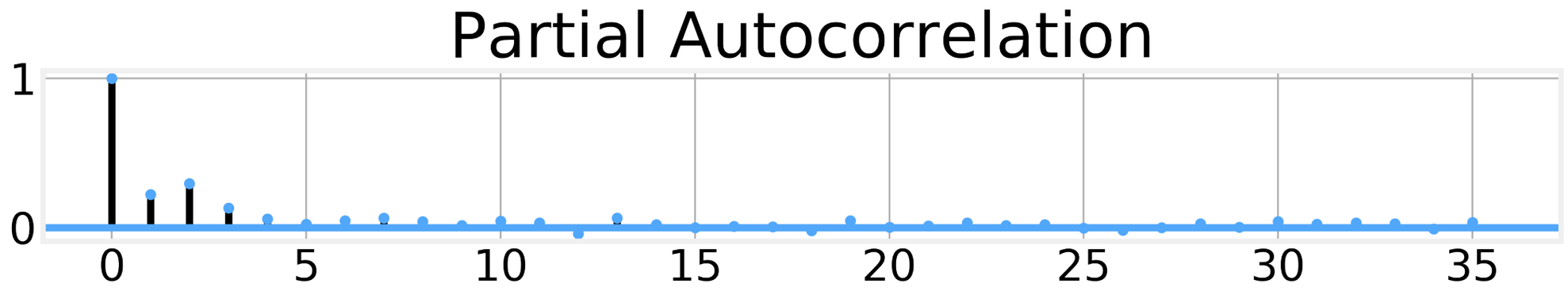
Plot Squared ACF/PACF



Determine G/ARCH







arch.readthedocs.io

- To fit and model ARCH and GARCH models, we will use the arch library
- arch includes a large amount of functionality, but we will focus on the univariate sub-package.
- arch.univariate provides the arch_model() function that allows us to define a wide variety of ARCH or GARCH models.
- While arch_model() can take many arguments, there's only a few that are relevant for our purposes:
  - y - the (dependent) variable data to train on
  - vol - the name of the volatility model, typically "ARCH" or "GARCH"
  - p, o, q - the model parameters

arch

https://arch.readthedocs.io/en/latest/univariate/generated/arch.univariate.base.ARCHModel.html

- When a arch_model() returns a ARCHModel object that represents our model. This object provides a great deal of functionality, but we're particularly interested in:
  - fit() - fit the parameters of the model (the values of α_i, β_j, etc.
  - simulate() - generate new values from the fitted model.
  - forecast() - forecast new values of the data.

And continuing with our example…

Build G/ARCH Model

- We use arch to generate a GARCH(1,1) model and train it on the DJIA returns dataset.
- arch provides us with a nice summary of the model parameters, similarly to statsmodels

Constant Mean - GARCH Model Results

Dep. Variable:	DJIA	**R-squared:**	-0.001
Mean Model:	Constant Mean	**Adj. R-squared:**	-0.001
Vol Model:	GARCH	**Log-Likelihood:**	-3000.92
Distribution:	Normal	**AIC:**	6009.85
Method:	Maximum Likelihood	**BIC:**	6033.31
		No. Observations:	2608
Date:	Mon, Aug 10 2020	**Df Residuals:**	2604
Time:	23:46:42	**Df Model:**	4

Can use AIC/BIC to compare various models and pick the best (smallest) one

Mean Model

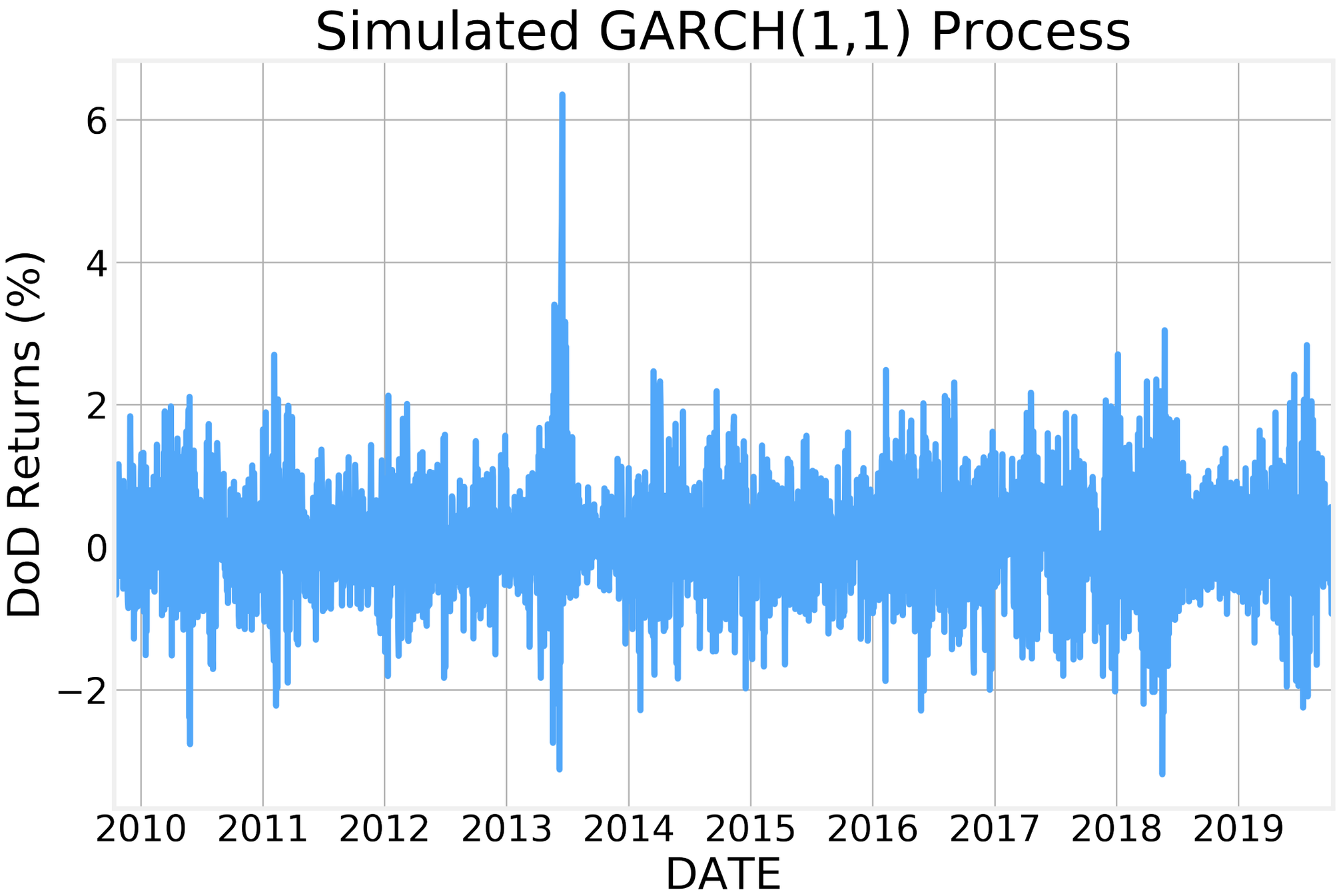
	coef	std err	t	P>|t|	95.0% Conf. Int.
mu	0.0744	1.327e-02	5.608	2.043e-08	[4.840e-02, 0.100]

Volatility Model

	coef	std err	t	P>|t|	95.0% Conf. Int.
omega	0.0318	6.646e-03	4.780	1.749e-06	[1.875e-02,4.480e-02]
alpha[1]	0.1587	2.316e-02	6.851	7.310e-12	[0.113, 0.204]
beta[1]	0.8033	2.280e-02	35.232	6.527e-272	[0.759, 0.848]

parameter coefficients

Forecast


Simulated GARCH(1,1) Process
DoD Returns (%)
6
4
2
0
−2
2010
2011
2012
2013
2014
2015
2016
2017
2018
2019
DATE

Code - ARCH Models

https://github.com/DataForScience/AdvancedTimeseries

Events

www.data4sci.com/newsletter

http://paypal.me/data4sci

Natural Language Processing (NLP) for Everyone

Jan 15, 2020 - 5am-9am (PST)

NLP with Deep Learning for Everyone

Jan 27, 2020 - 5am-9am (PST)

Natural Language Processing (NLP) from Scratch

http://bit.ly/LiveLessonNLP - On Demand