Lecture 3: ARIMA Models

Contents

Lecture Learning Objectives

- Define and explain what it means to say that a time series is stationary.
- Use differencing to coerce a non-stationary series to stationarity.
- Explain what a autoregressive process is (`AR(p)`), what its ACF looks like, and under what conditions it is stationary.
- Explain what a moving average process is (`MA(q)`), what its ACF looks like, and under what conditions it is stationary.
- Explain what are `ARIMA`, `SARIMA` and `ARIMAX` models are.
- Implement an `AR(p)`, `MA(q)`, and `ARIMA()` in `statsmodels` using the `ARIMA` class.

- Use the ACF and PACF to help select the order of an `AR(p)`, `MA(q)`, or `ARIMA(p,d,q)` model.

Imports

```
import warnings
warnings.filterwarnings('ignore')
import numpy as np
import pandas as pd
import plotly.express as px
import plotly.graph_objects as go
from plotly.subplots import make_subplots
import matplotlib.pyplot as plt
from statsmodels.tsa.stattools import adfuller
from statsmodels.graphics.tsaplots import plot_acf, plot_pacf
from statsmodels.tsa.arima.model import ARIMA
from statsmodels.tsa.arima_process import ArmaProcess
from statsmodels.tsa.api import SimpleExpSmoothing, Holt, ExponentialSmoothing, ET
from sklearn.metrics import mean_absolute_error, mean_squared_error, mean_absolute
from sklearn.linear_model import LinearRegression
import yfinance as yf

# Custom modules
from scripts.plotting import *
from scripts.utils import *

# Plot defaults
px.defaults.height = 400; px.defaults.width = 620
plt.style.use("bmh"); plt.rcParams.update({"font.size": 16, "figure.figsize": (7,5
```

1. Recap week 1

- Time-series EDA with decomposition
- Baseline models (naive, drift, seasonal naive, average)
- Simple exponential smoothing (level), Holt (level + trend), Holt-Winters (level+trend+seasonality)
- ETS models focusing on describing the trend and seasonality in the data

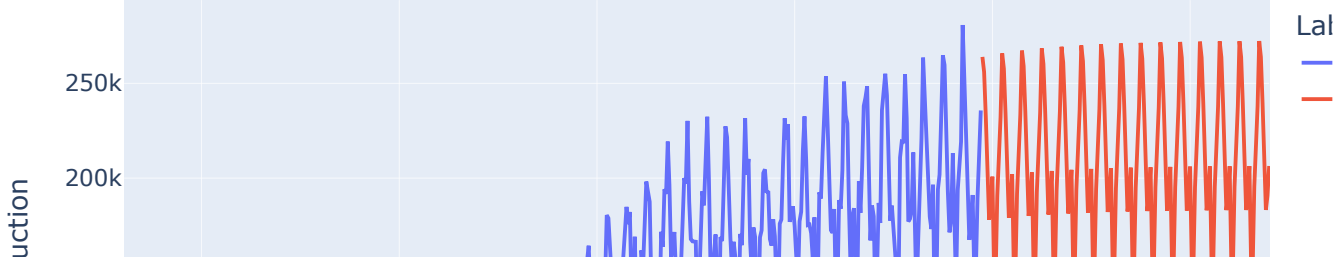
Trend Component	Seasonal Component
None `(N)`	None `(N)`
Additive `(A)`	Additive `(A)`
Additive damped `(Ad)`	Multiplicative `(M)`

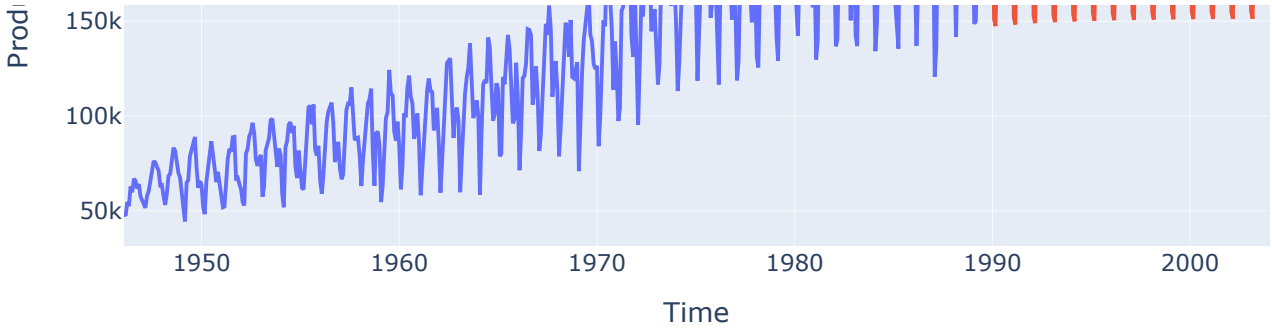
Notation	Method
`(N,N)`	Simple exponential smoothing
`(A,N)`	Holt's method
`(A,A)`	Additive Holt-Winter's method

```
def get_beer(split="train", label="Observed"):
    df = pd.read_csv(f"data/beer_{split}.csv", index_col=0, parse_dates=["Time"]).
    df.index.freq = "M"
    return df
```

```
# Load data
df = get_beer()
forecast_index = create_forecast_index(df.index[-1], 175)

# Fit AAdM
model = ETSModel(df["Production"], error="add", trend="add",
                 damped_trend=True, seasonal="mul", seasonal_periods=12).fit(dis

# Make forecast
ets = pd.DataFrame({"Production": model.forecast(len(forecast_index)),
                    "Label": "ETS (AAdM)"},
                   index=forecast_index)
```

```
px.line(pd.concat((df, ets)), y="Production", color="Label", width=820)
```





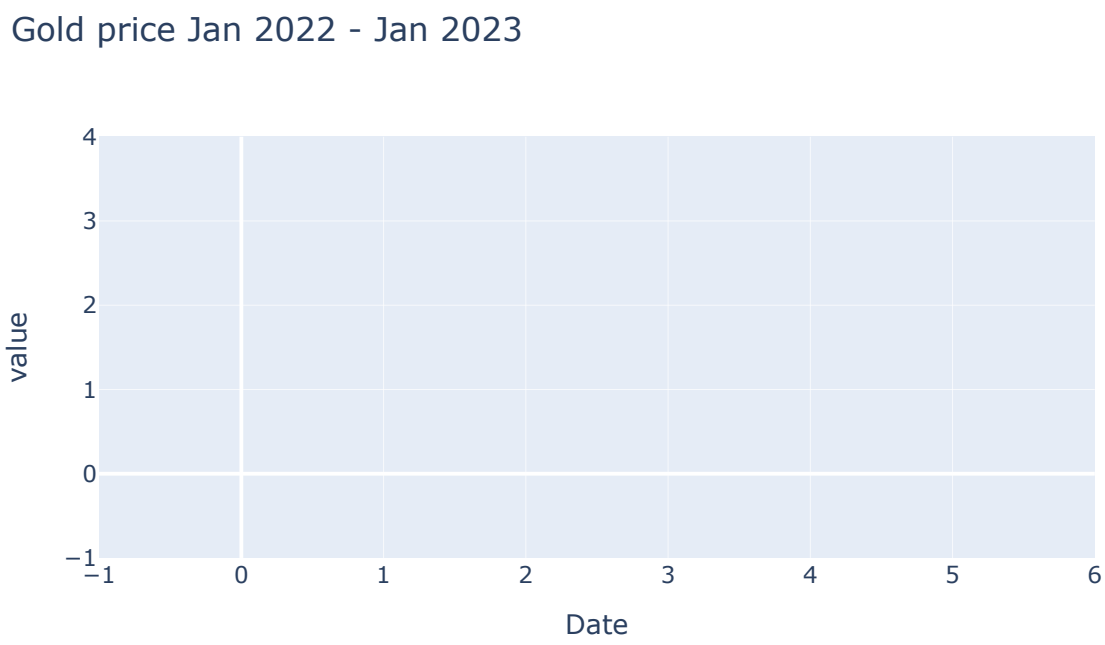
What if we have a time-series like this

- No obvious seasonality
- Trends are changing rapidly

```
df = yf.download('GC=F', start='2022-01-01', end='2023-01-01', progress=False).Clo

px.line(df, width=680, title='Gold price Jan 2022 - Jan 2023')
```

```
1 Failed download:
```

```
['GC=F']: JSONDecodeError('Expecting value: line 1 column 1 (char 0)')
```

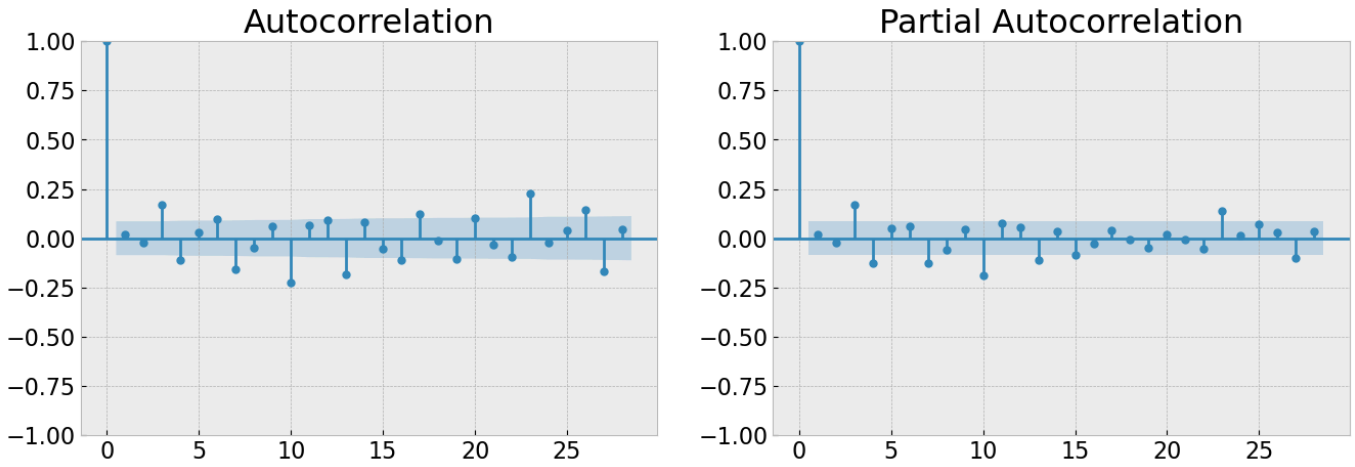


What we have done with ETS model:

- We model level + trend + seasonality
- But we have not modeled the autocorrelation explicitly (e.g., y_t is highly correlated with y_{t-1} for example)

Residuals diagnostics

```
fig, axes = plt.subplots(nrows=1, ncols=2, figsize=(16, 5))

plot_acf(model.resid, ax=axes[0])
plot_pacf(model.resid, ax=axes[1]);
```



- We still see a lot of auto-correlation
- That means y_t is still highly correlated with past values y_{t-1}, y_{t-2}, ...
- How can we take auto-correlation into account in our model?

Intuition behind ARIMA Models

- ARIMA models provide another approach to time series forecasting based on leveraging past values to inform future values
- Exponential smoothing and ARIMA models are the two most commonly used classical approaches to time series forecasting - and they are often difficult to beat!
- While exponential smoothing is based on a description of the trend and seasonality in the data, ARIMA models are based on autocorrelations in the data
- Because ARIMA models don't explicitly model trend or seasonality, we typically constrain them to modelling what we call *stationary* data

2. What is stationarity?

Why stationarity is important in ARIMA model?

- Recall that with most of the classical statistics model, we assume our data are **independent** and indentically distributed (i.i.d)
- However, as we know, time series data are **dependent**, that means y_t tend to correlate with some lagged versions of itself (y_{t-1}, y_{t-2}, etc.)
- Now independence is a unique property, **two random variables can be independent only in one way, but they can be dependent in various ways.**
- Stationarity is one way of modeling the dependence structure

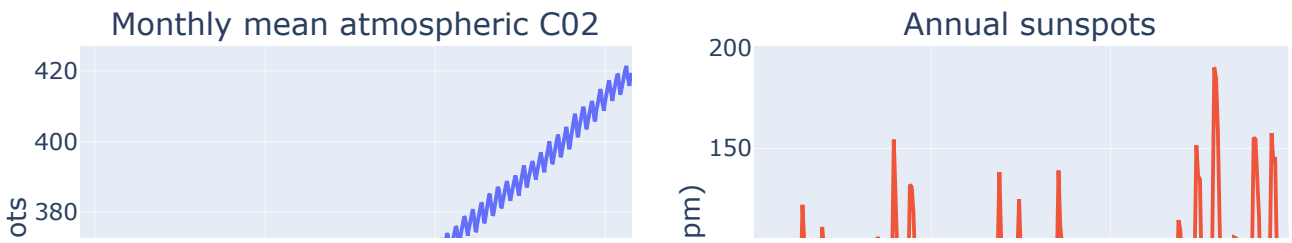
What is stationarity?

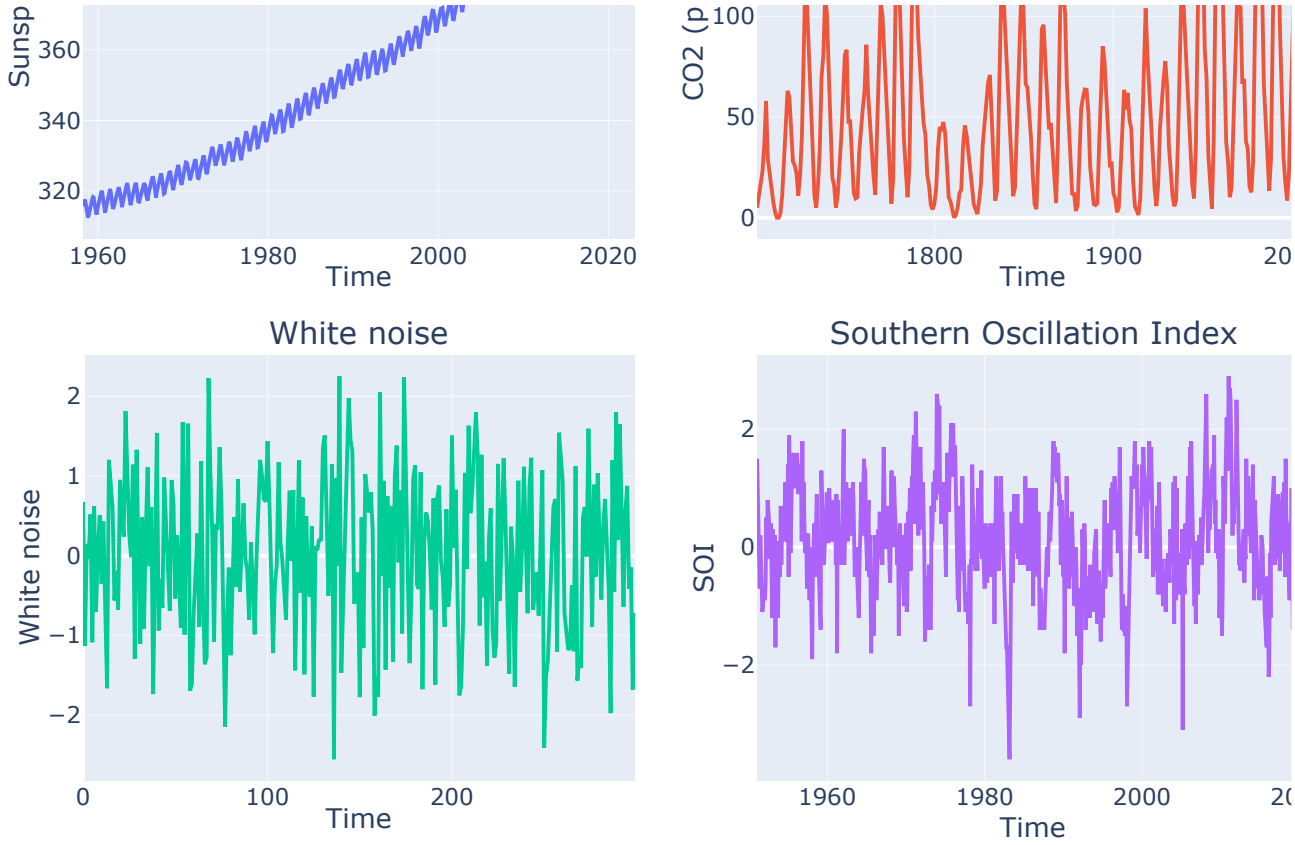
- Stationarity means that the statistical properties of a process generating a time series do not change over time. It is characterized by:
  - A constant mean & variance
  - Is roughly horizontal (no strong up or down trends)
  - Does not show predictable patterns (e.g., seasonality)
- It does not mean that the series does not change over time, just that the way it changes remains constant over time. Stationary data is often easier to model and ensures that the past is informative of the future.

In-Class Exercise

Looking at the following plots, discuss whether you think they are stationary.

```
plot_stationary()
```





- Are the above series stationary?
    1. **CO2**: No - obvious trend and seasonality.
    2. **Sunspots**: No - seasonality (although it could be argued that the period is of 9-11 years, so is not exactly predictable and therefore the series is stationary...)
    3. **White noise**: Yes - no obvious trend or seasonality here, it looks completely random.
    4. **SOI**: Hard to tell it looks pretty random but we know that SOI has some seasonality. So it's not stationary

Weak stationarity vs. strong stationarity

Strong stationarity, also called strict stationarity, requires that all moments of the time series, including its joint distribution, remain constant over time. This means that the statistical properties of the time series remain exactly the same over time, including the shape of the distribution, and it is often assumed in theoretical and mathematical analyses of time series. This type of series is rarely seen in real-life practice.

Weak stationarity, also called second-order stationarity, is a less strict form of stationarity than strong stationarity. A time series is considered weakly stationary if its mean, variance, and autocovariance are constant over time, but its higher-order moments (such as skewness and kurtosis) may not be constant. This means that the statistical properties of the time series remain

the same over time on average, but there may be some changes in the higher moments. In practice, weak stationarity is often sufficient for many time series analysis and forecasting applications.

In practice, when people refer to `stationarity`, they usually mean 'weak stationarity'.

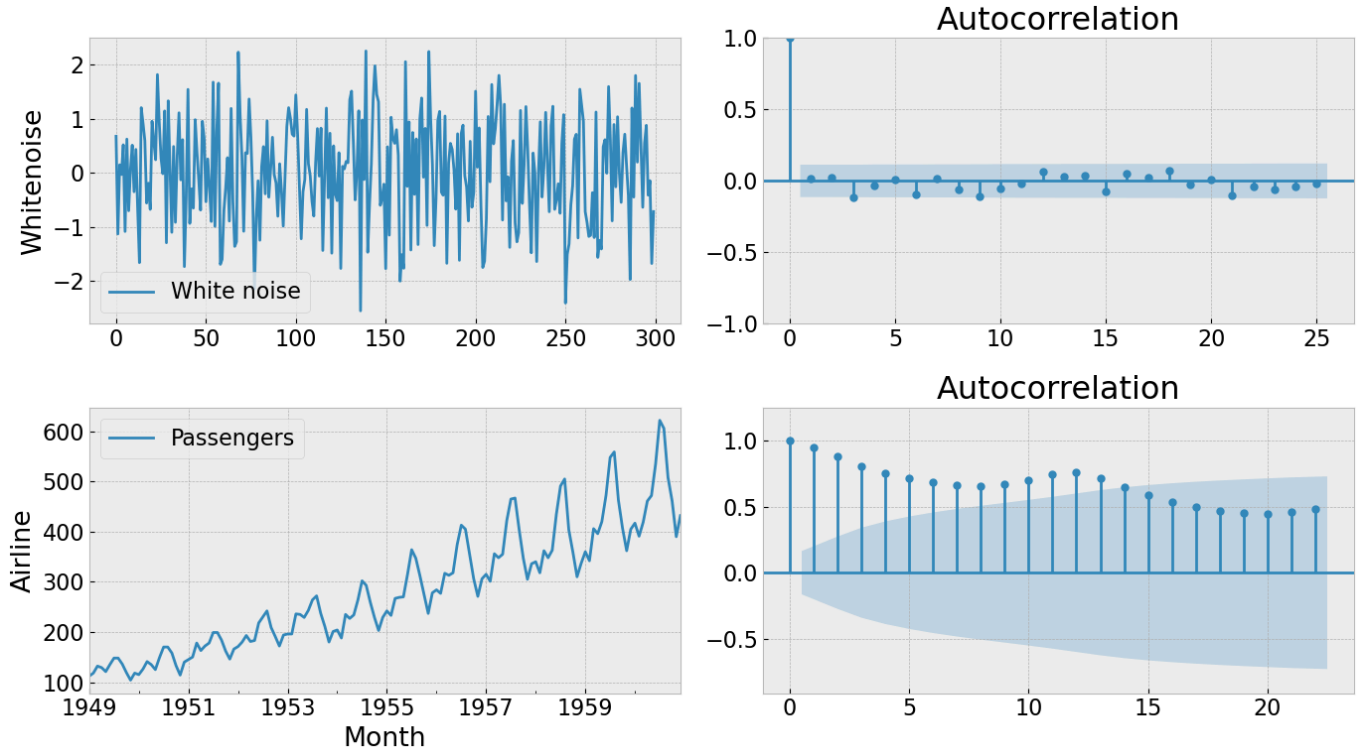
How to check whether a time-series is stationary?

- Visual inspections
  - Line plot
  - Correlogram
- Summary statistics
  - Rolling mean
  - Rolling standard deviation
- Hypothesis test
  - Augmented Dickey- fuller Test (ADCF)

Visual inspections

- Line plot: Look for trends or patterns that might indicate non-stationarity, such as a noticeable trend, seasonality, or a changing variance over time
- Correlogram: In a stationary time series, the ACF plot should decay rapidly to 0, indicating that there is no significant correlation between observations at different time points. Non-stationary time-series will have ACF decays slowly

```
fig, axes = plt.subplots(nrows=2, ncols=2, figsize=(14, 8))

noise = pd.read_csv("data/white-noise.csv", index_col=0)
noise.plot(ax=axes[0,0])
plot_acf(noise,ax=axes[0,1]);
axes[0,0].set(ylabel='Whitenoise')

airline = pd.read_csv("data/airline.csv", index_col=0, parse_dates=True)
airline.plot(ax=axes[1,0])
plot_acf(airline,ax=axes[1,1], auto_ylims=True);
axes[1,0].set(ylabel='Airline')
plt.tight_layout()
plt.show()
```

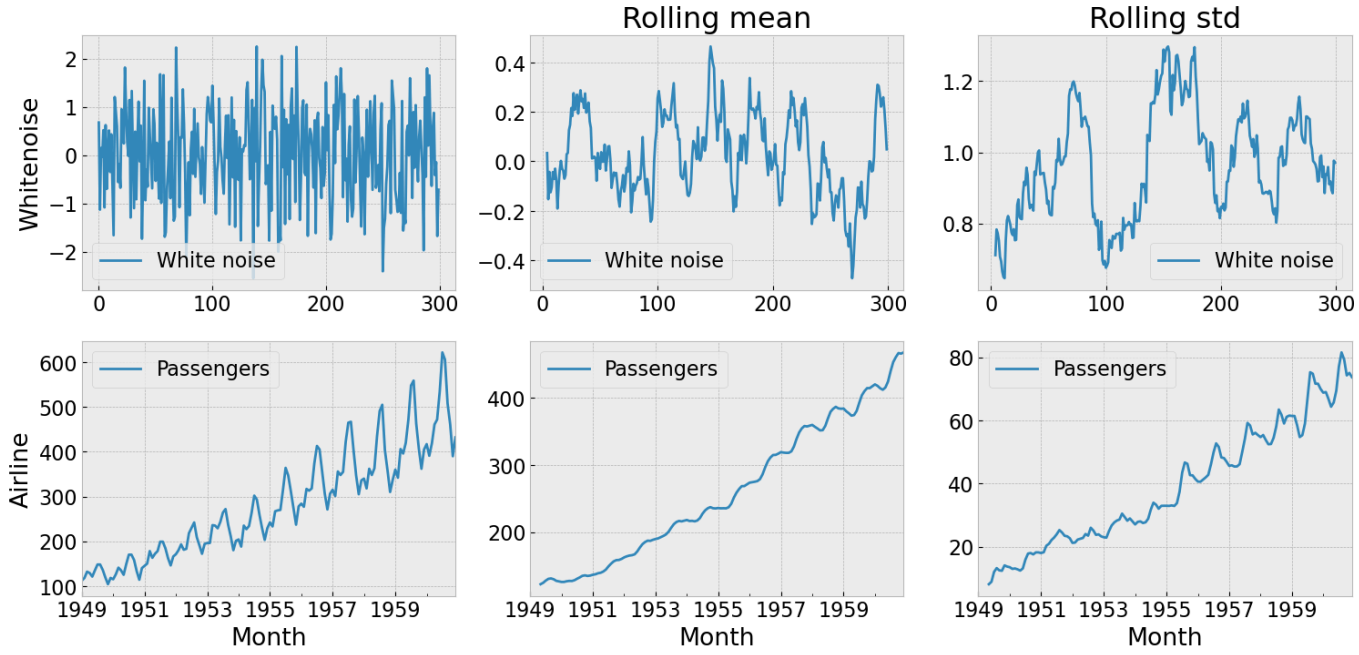


Summary statistics

For a stationary time series, the mean and variance should be approximately constant over time.

```
fig, axes = plt.subplots(nrows=2, ncols=3, figsize=(18, 8))
wd_size = 20

noise = pd.read_csv("data/white-noise.csv", index_col=0)
noise.plot(ax=axes[0,0])
axes[0,0].set(ylabel='Whitenoise')
noise.rolling(wd_size, min_periods=5).mean().plot(ax=axes[0,1])
axes[0,1].set_title('Rolling mean')
noise.rolling(wd_size, min_periods=5).std().plot(ax=axes[0,2])
axes[0,2].set_title('Rolling std')

airline = pd.read_csv("data/airline.csv", index_col=0, parse_dates=True)
airline.plot(ax=axes[1,0])
airline.rolling(wd_size, min_periods=5).mean().plot(ax=axes[1,1])
airline.rolling(wd_size, min_periods=5).std().plot(ax=axes[1,2])
axes[1,0].set(ylabel='Airline')
```

```
[Text(0, 0.5, 'Airline')]
```



Hypothesis tests

1. Augmented Dickey Fuller ("ADF") test: Null hypothesis H_0: The time series is non-stationary, small p indicates stationary
2. Kwiatkowski-Phillips-Schmidt-Shin ("KPSS") test: Null hypothesis H_0: The time series is stationary, small p indicates non-stationary

- Both are available in `statsmodels`. Let's try an ADF test on our weight data (from `statsmodels.tsa.stattools.adfuller`):

The Augmented Dickey- fuller Test (ADCF) belongs to a category of tests called 'Unit Root Test', a common statistical test for time-series stationarity.

- Null hypothesis: the data is non-stationary.

```
result = adfuller(noise)
print('ADF Statistic: %f' % result[0])
print('p-value: %f' % result[1])
```

```
ADF Statistic: -10.785624
p-value: 0.000000
```

```
result = adfuller(airline)
print('ADF Statistic: %f' % result[0])
print('p-value: %f' % result[1])
```

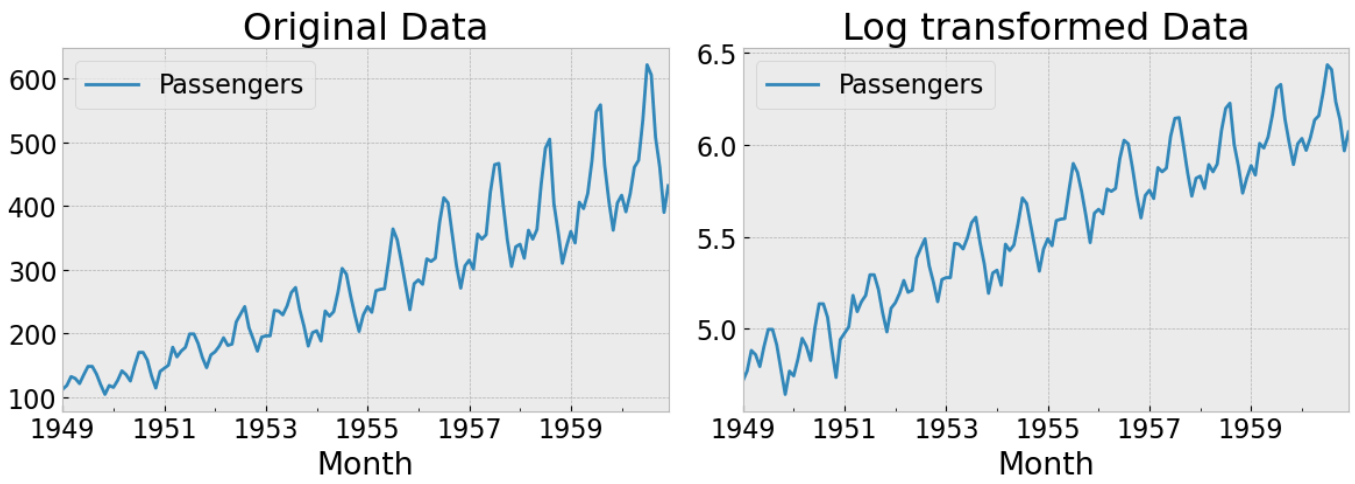
```
ADF Statistic: 0.815369
p-value: 0.991880
```

How to make time-series data stationary?

Stabilize the variance:

- Log transformation is easy to interpret. Changes in a log value are relative (percent) changes on the original scale.

```
fig, axes = plt.subplots(1, 2, figsize=(12, 4.5))
airline = pd.read_csv("data/airline.csv", index_col=0, parse_dates=True)
airline.plot(ax=axes[0], title='Original Data')
airline_log = np.log(airline)
airline_log.plot(ax=axes[1], title='Log transformed Data')
plt.tight_layout()
plt.show();
```



- Box-Cox transformation using the `scipy.stats.boxcox` function

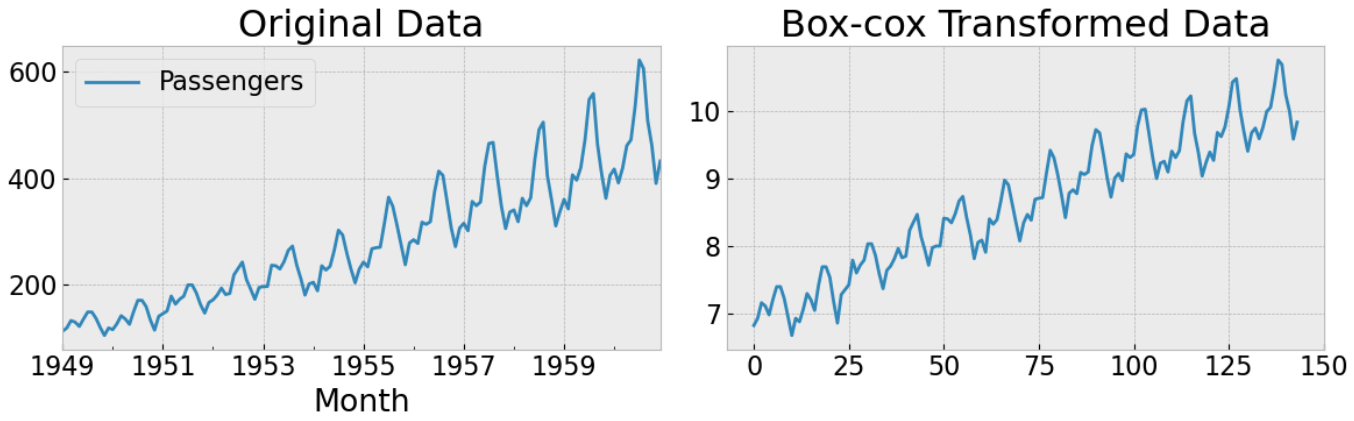
$$w_t = \begin{cases} \frac{y^\lambda - 1}{\lambda} & \text{if } \lambda \neq 0 \\ \ln y & \text{if } \lambda = 0 \end{cases}$$

where:

- y_t is the original data
- w_t is the transformed data

```
from scipy.stats import boxcox

# Apply Box-Cox transformation
transformed_data, lambda_val = boxcox(airline['Passengers'])

# Print the optimal lambda value
print('Lambda value: {:.3f}'.format(lambda_val))

# Plot the original and transformed data
fig, axs = plt.subplots(1, 2, figsize=(12, 4))
airline.plot(ax=axs[0], title='Original Data')
pd.Series(transformed_data).plot(ax=axs[1], title='Box-cox Transformed Data')

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

```
Lambda value: 0.148
```



As we can see above, the variance was stabilized after applying log transform or Box-cox transform. However, there's still an increasing trend and we need to stabilize the mean.

Stabilize the mean

- Differencing: calculate the differences between consecutive observations in the series. This removes any trend in the data, since it measures the change between two points in time rather than the absolute value.

1st order differencing:

$$y'_t = y_t - y_{t-1}$$

- In some situations, 1st order differenced data will not appear to be stationary and it may be necessary to difference the data a second time to obtain a stationary series.

2nd order differencing:

$$y_t'' = y_t' - y_{t-1}'$$

$$= y_t - 2y_{t-1} + y_{t-2}$$

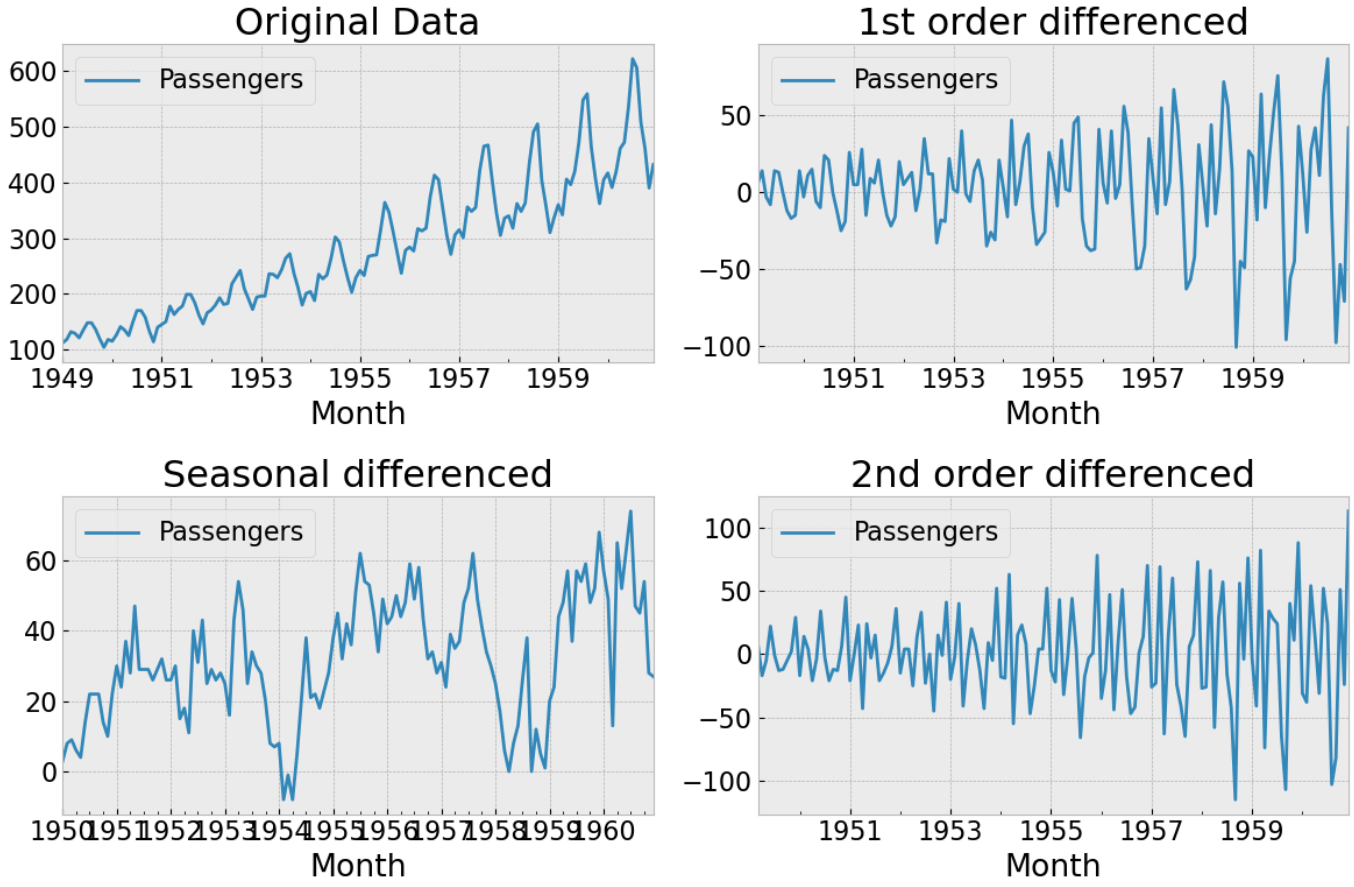
- In practice, we almost never go beyond 2nd order differencing. If your data still appear non-stationarity after 2nd order differencing, then you might want to check for seasonal differencing

Seasonal differencing:

$$y_t' = y_t - y_{t-m}$$

where m is the seasonal period

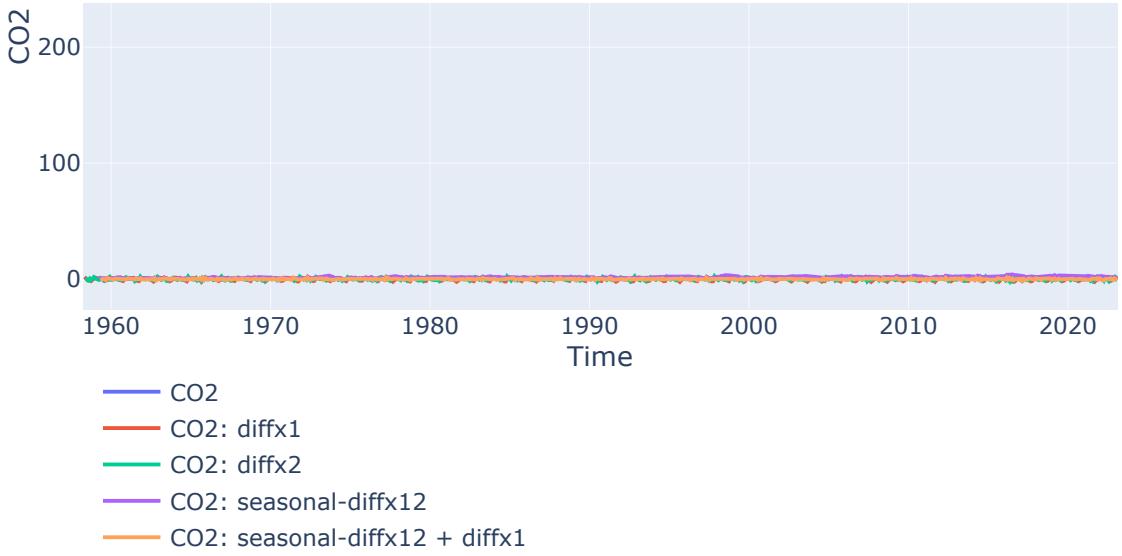
```
fig, axes = plt.subplots(2, 2, figsize=(12, 8))
airline = pd.read_csv("data/airline.csv", index_col=0, parse_dates=True)
airline.plot(ax=axes[0,0], title='Original Data')

airline_diff = airline.diff().dropna() # 1st order diff
airline_diff.plot(ax=axes[0,1], title='1st order differenced')

airline_diff2 = airline.diff().diff().dropna() # 2nd order diff
airline_diff2.plot(ax=axes[1,1], title='2nd order differenced')

airline_diff12 = airline.diff(12).dropna() # seasonal diff
airline_diff12.plot(ax=axes[1,0], title='Seasonal differenced')
plt.tight_layout()
plt.show();
```



Let's do another one with CO2 data

- Sometimes the differenced data will still not appear to be stationary
- In these cases, we have a few options:
  1. Differencing again (this is called "second-order differencing")
  2. Seasonal differencing (the difference between an observation and the previous observation from the same season)
  3. Applying a transformation (when there is changing variance in our series, we might want to apply a transform like a log, or Box-Cox transform)

```
df = pd.read_csv("data/co2.csv", index_col=0, parse_dates=["Time"])
plot_interactive_differencing(df['CO2'])
```



- As we see above, the data still have some seasonality patterns even after applying 2nd order differencing
- If we just apply the seasonal differencing only (e.g. diff(12)), then the data still display an increasing trend
- If we apply both seasonal differencing & 1st order differencing, then we managed to remove both trend & seasonality making the data more stationary

3. Autoregressive models

- The term autoregression indicates that it is a regression of the variable against itself. We simply forecast our series using a linear combination of past values of the series
- An AR model of order p, written `AR(p)`, looks like this:

$$y_t = c + \phi_1 y_{t-1} + \phi_2 y_{t-2} + \ldots + \phi_p y_{t-p} + \epsilon_t$$

- Where ϕ is the regression coefficient(s), c is a constant (think of it like the "intercept"), and ϵ_t is white noise.
- For example, an AR(1) model will be

$$y_t = c + \phi_1 y_{t-1} + \epsilon_t$$

- For example, an AR(2) model will be

$$y_t = c + \phi_1 y_{t-1} + \phi_2 y_{t-2} + \epsilon_t$$

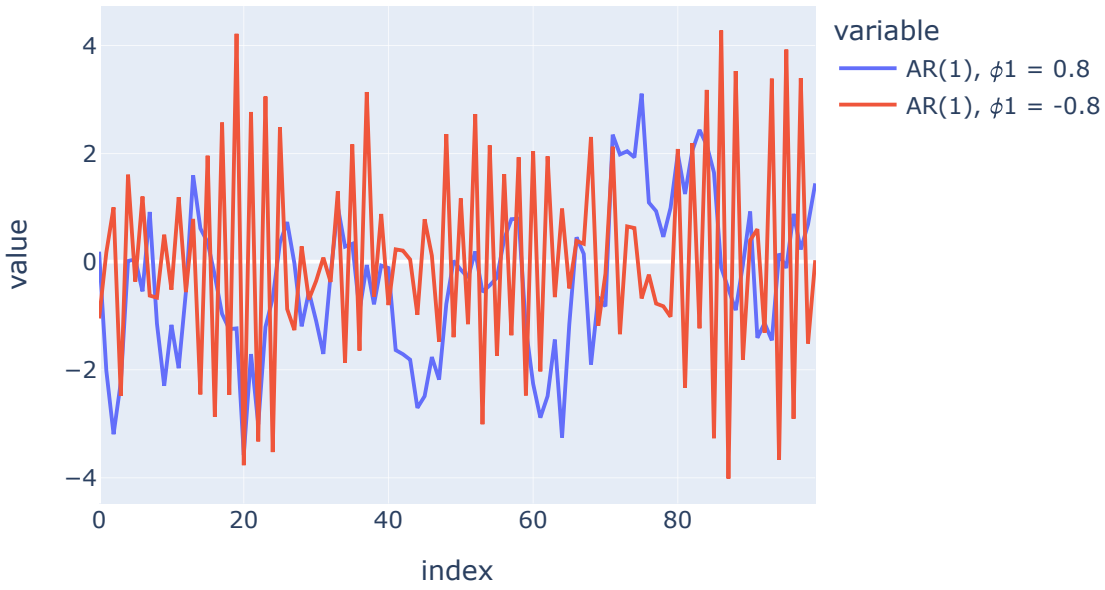
- Before we go about fitting an `AR(p)` model to some data, let's actually see what some `AR(p)` processes look like using the `statsmodels` class `ArmaProcess`:

> Note that in `statsmodels` you must specify the coefficient on the zero-lag (almost alwasy 1) and AR coefficients should have the opposite sign of what one would write in the ARMA representation (due to the way the code is run under-the-hood).

$$y_t = c + 0.8 y_{t-1} + \epsilon_t$$

$$y_t = c - 0.8 y_{t-1} + \epsilon_t$$

```
df = pd.DataFrame({"AR(1), ϕ1 = 0.8": ArmaProcess(ar=[1, -0.8]).generate_sample(),
                   "AR(1), ϕ1 = -0.8": ArmaProcess(ar=[1, 0.8]).generate_sample()}
px.line(df)
```



- Notice how $\phi > 0$ generally leads to a "smoother" series and $\phi < 0$ leads to rapid, oscillating changes. This makes sense if you look at the equaton for an `AR(1)` model: $y_t = \phi_1 y_{t-1} + \epsilon_t$, see that $\phi < 0$ means that the next value in the series is some negative multiple of the previous value, plus some noise.

In-Class Exercise

Fill in the blank with the one of following answers: {random walk, random walk with drift, oscillate between positive and negative values, white noise}

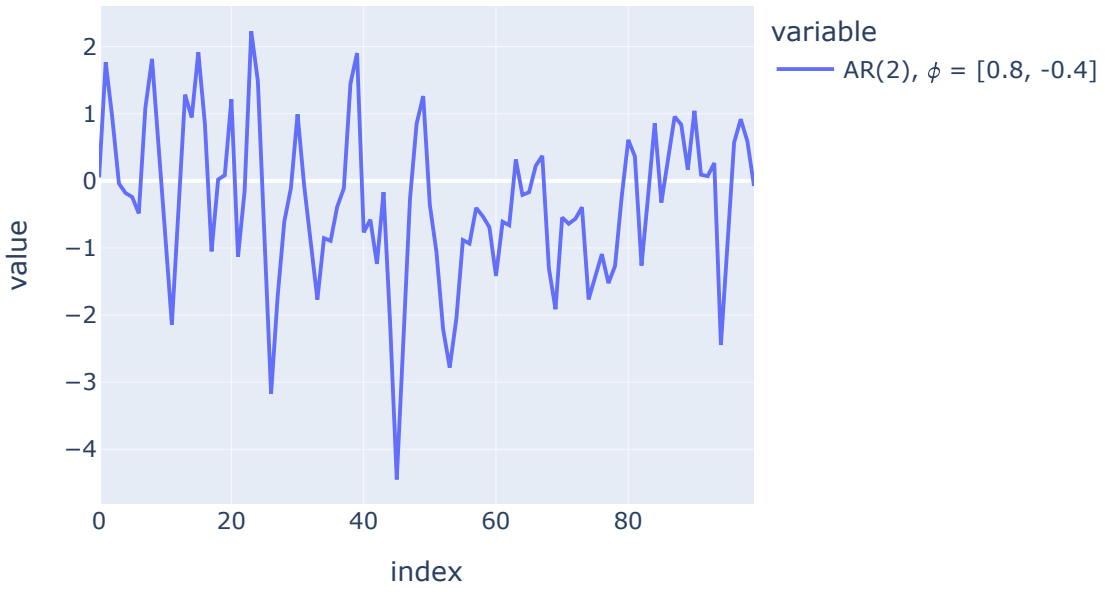
For an AR(1) model: $$y_t = c + \phi_1 y_{t-1} + \epsilon_t$$

- When ϕ_1 = 0, y_t is equivalent to a ...
- When ϕ_1 =1 and c = 0, y_t is equivalent to a ...
- When ϕ_1 =1 and c ≠ 0, y_t is equivalent to a ...
- When ϕ_1 < 0, y_t tends to ...

- We can add as many terms as we like to the model to come up with different patterns, let's try 2 terms, AR(2) model:

$$y_t = c + 0.8y_{t-1} - 0.4y_{t-2} + \epsilon_t$$

```
df = pd.DataFrame({"AR(2), ϕ = [0.8, -0.4]": ArmaProcess(ar=[1, -0.8, 0.4]).genera
px.line(df)
```



- We typically constrain the parameter values of an `AR(p)` model to restrict it to stationary data. I'll talk more about what that means shortly, but basically it means that the properties

(mean, variance, etc.) of the series don't change with time, meaning that the past remains informative of the future.

- For `AR(1)`, this means $|\phi_1| < 1$
- For `AR(2)`, this means $|\phi_2| < 1$, and $\phi_1 + \phi_2 < 1$, and $\phi_2 - \phi_1 < 1$
- For higher orders, things get more complicated, but `statsmodels` takes care of it for us

Why are AR models also called long-memory models?

Let's take an example of an AR(1) model

$$y_t = c + \phi_1 y_{t-1} + \epsilon_t (1)$$

$$y_{t-1} = c + \phi_1 y_{t-2} + \epsilon_{t-1} (2)$$

If we plug (2) into (1)

$$y_t = c + \phi_1 (c + \phi_1 y_{t-2} + \epsilon_{t-1}) + \epsilon_t$$

$$y_t = c^* + \phi_1^2 y_{t-2} + \phi_1 \epsilon_{t-1} + \epsilon_t$$

We could do the same by replacing $y_{t-2} = c + \phi_1 y_{t-3} + \epsilon_{t-2}$

We would get

$$y_t = c^* + \phi_1^3 y_{t-3} + \phi_1^2 \epsilon_{t-2} + \phi_1 \epsilon_{t-1} + \epsilon_t$$

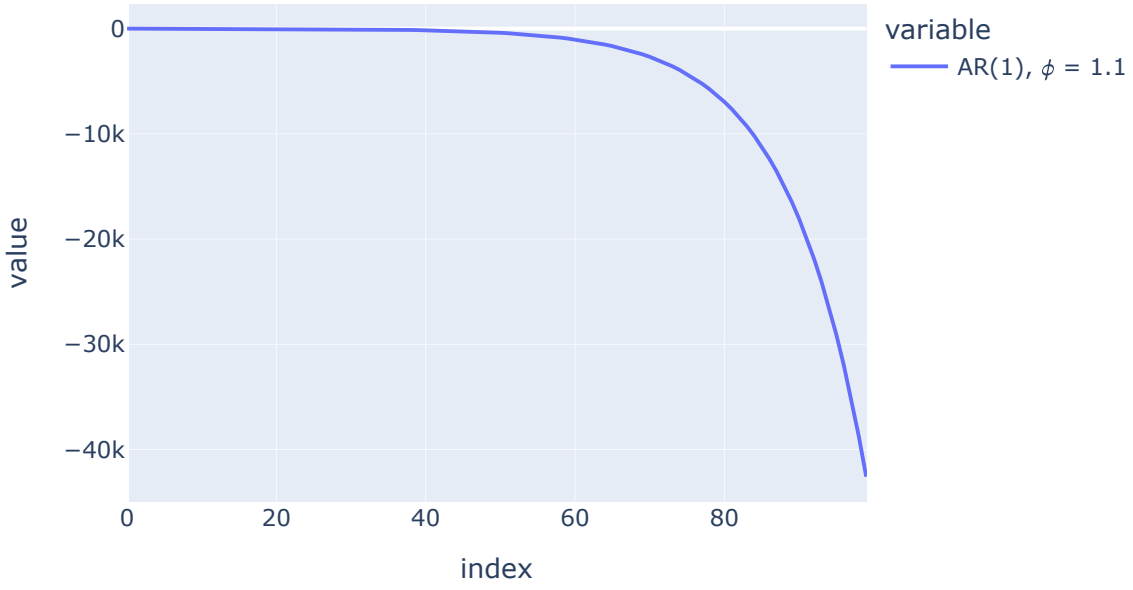
If we keep continue this recurssion process to the first observation, we would get:

$$y_t = \frac{c}{1-\phi_1} + \phi_1^t y_1 + \phi_1^{t-1} \epsilon_2 + \phi_1^{t-2} \epsilon_3 + \ldots + \epsilon_t$$

As we can see in the equation, the first observation y_1 still carry some effects ϕ_1^t until today, this is why we call AR a long-memery model. However, we want the effects to get smaller and smaller as we go back in the past, and that's why $|\phi_1| < 1$

- To understand why we do this, look what happens if we relax the stationarity restriction and just $\phi = 1.1$:

```
df = pd.DataFrame({"AR(1), ϕ = 1.1": ArmaProcess(ar=[1, -1.1]).generate_sample()})
px.line(df)
```



4. Moving average models

- Rather than using past values of the forecast variable in a regression, a moving average model uses **past forecast errors** in a regression-like model
- A moving average models should not be confused with the moving average smoothing we discussed in Lecture 1
- An MA model of order q, written `MA(q)`, looks like:

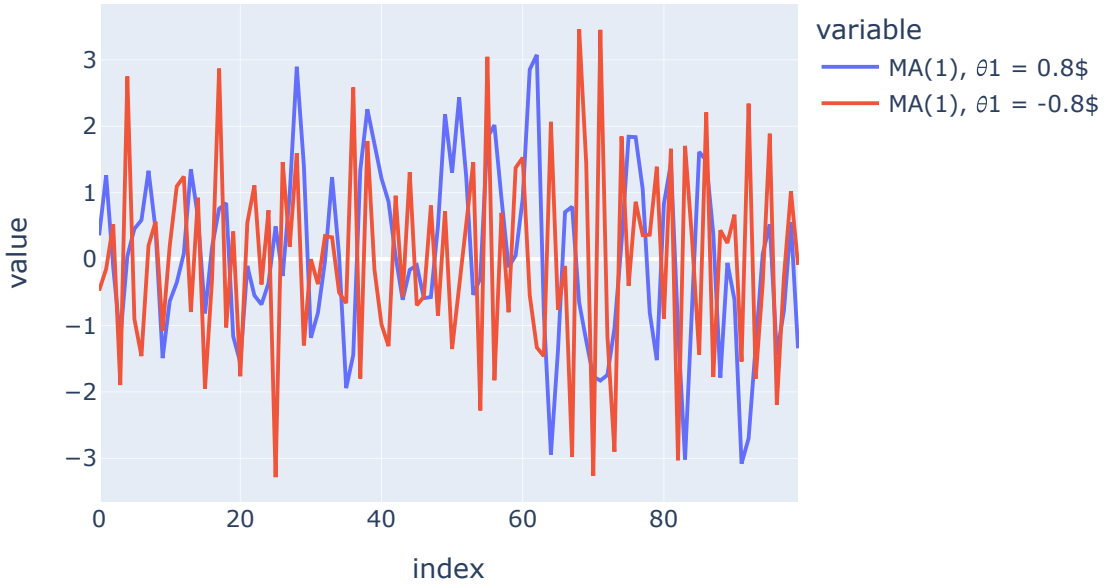
$$y_t = c + \epsilon_t + \theta_1\epsilon_{t-1} + \theta_2\epsilon_{t-2} + \ldots + \theta_q\epsilon_{t-q}$$

- Where θ is the regression coefficient(s), c is a constant (think of it like the "intercept"), and ϵ_t is white noise.

For example, an MA(1) model would take the form:

$$y_t = c + \epsilon_t + \theta_1\epsilon_{t-1}$$

```
df = pd.DataFrame({"MA(1), θ1 = 0.8$": ArmaProcess(ma=[1, 0.8]).generate_sample(),
                   "MA(1), θ1 = -0.8$": ArmaProcess(ma=[1, -0.8]).generate_sample(
px.line(df)
```



Why are MA models called short-memory models?

$y_{t-2} = c + \epsilon_{t-2} + \theta_1 \epsilon_{t-3}$

$y_{t-1} = c + \epsilon_{t-1} + \theta_1 \epsilon_{t-2}$ (no more ϵ_{t-3})

$y_t = c + \epsilon_t + \theta_1 \epsilon_{t-1}$ (no more ϵ_{t-2})

The effect of the error terms from the distant past don't last long into the future.

Unlike the AR process, MA processes are stationary for any values of the parameters. Although we usually constrain the parameters for a different reason (to enforce "invertibility" - you don't need to know about this, but it's an optional question in this week's lab if you're interested).

Summary

- AR models are often better at capturing long-term trends and patterns in the data, as they incorporate multiple lagged values of the time series.

- AR models can be useful for modeling time series that have a strong dependence on their own past values, which is often the case in financial time series and other economic data.
- However, AR models may be less effective at modeling short-term fluctuations or noise in the data, as they tend to focus more on longer-term patterns.
- MA models are often better at filtering out short-term fluctuations or noise in the data, as they only incorporate a small number of lagged forecast errors.
- MA models can be useful for modeling time series that have a lot of random variation or volatility. For example, if some external event impacts our series and increases it's value (imagine a big spike in a time plot), an MA model allows us to propagate that impact into the future for several time steps. After several time steps, that spike no longer has an influence. We'll explore this concept in lab.
- However, MA models may be more sensitive to the choice of lag length or window size, as different choices can lead to different results. MA models also may not be able to distinguish between short-term fluctuations and long-term trends.

This leads us to ARMA to achieve the best of both worlds

5. ARMA

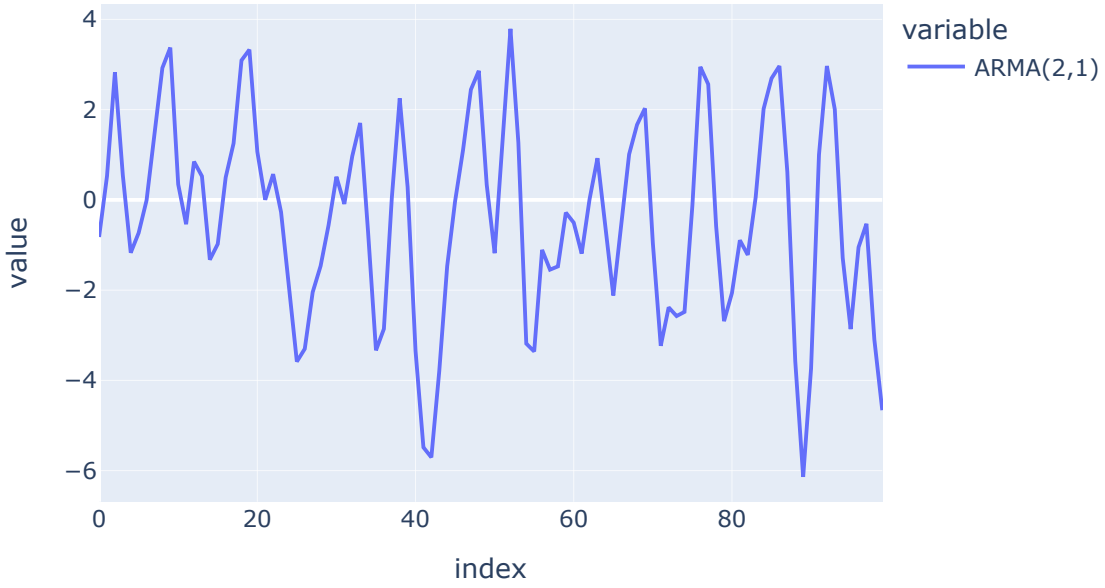
- We can combine autoregression + moving average, to get the dynamics of both models, into a so-called ARMA model:

$$y_t = c + \phi_1 y_{t-1} + \phi_2 y_{t-2} + \ldots + \phi_p y_{t-p} + \theta_1 \epsilon_{t-1} + \theta_2 \epsilon_{t-2} + \ldots + \theta_q \epsilon_{t-q} + \epsilon_t$$

- Where ϕ is the AR coefficient(s), θ is the MA coefficient(s), c is a constant (think of it like the "intercept"), and ϵ_t is white noise.
- We usually write an ARMA model as `ARMA(p, q)`
- The key idea behind ARMA is parsimony: it might be possible to fit a simpler, mixed model with fewer parameters, than either a pure AR or a pure MA model
- Let's try it out with

$$y_t = c + 0.8 y_{t-1} - 0.4 y_{t-2} + 0.8 \epsilon_{t-1} + \epsilon_t$$

```
df = pd.DataFrame({"ARMA(2,1)": ArmaProcess(ar=[1, -0.8, 0.4], ma=[1, 0.8]).genera
px.line(df)
```

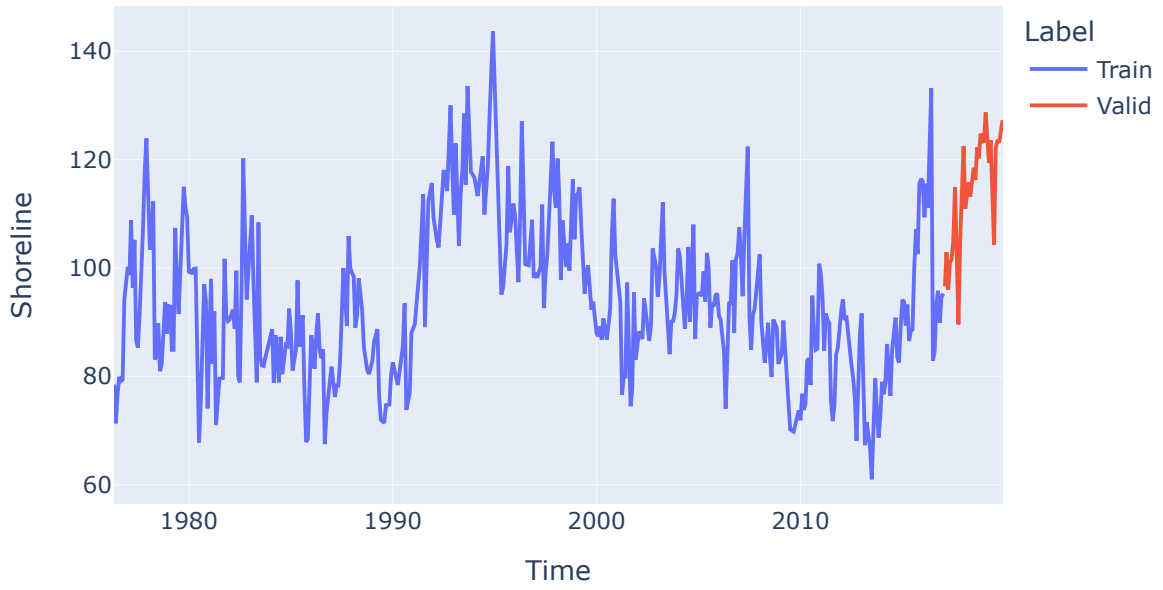


- Feel free to play around with the parameters above. The biggest takeaway I want you to have here is that ARMA models are very flexible models that can model a wide range of *stationary* data

6. ARIMA

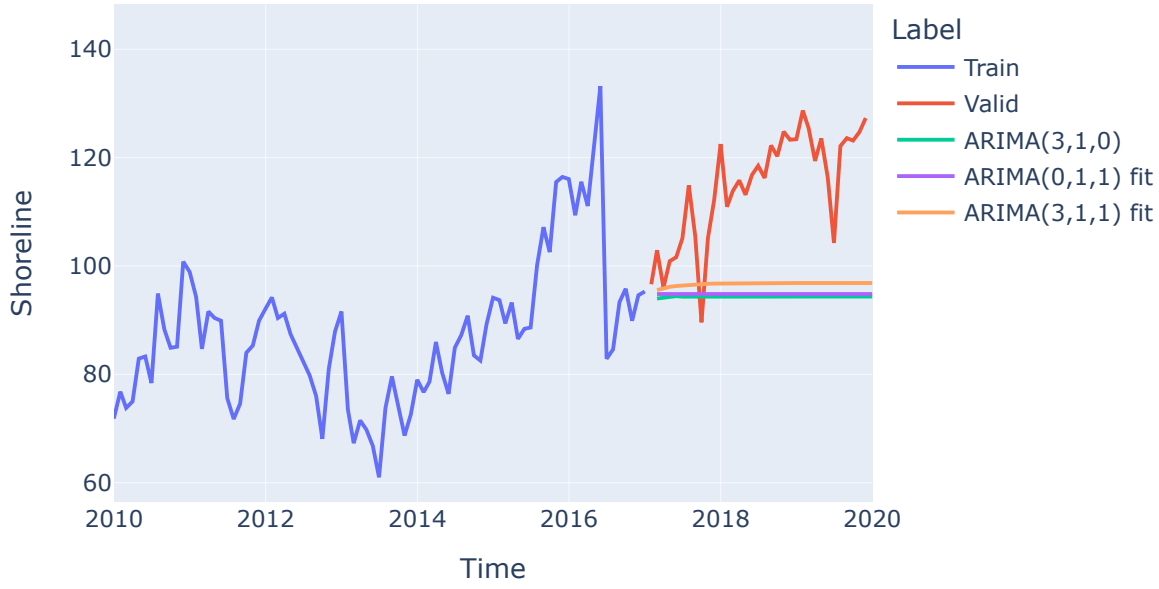
- An ARIMA model (Autoregressive Integrated Moving Average) is just an ARMA model with differencing built-in!
- We write it as `ARIMA(p,d,q)` where:
  - `p` is the autoregressive order
  - `d` is the order of differencing to apply (usually 1)
  - `q` is the moving average order
- Let's try all our models out on the beach data in lab 1

```
# Load data
beach_train = (pd.read_csv("data/beach_train.csv", index_col=0, parse_dates=True)
               .resample("1M").mean()
               .interpolate(method="linear")
               .assign(Label = "Train")
               .rename_axis(index="Time")
           )
beach_train.index.freq = "M"

beach_valid = (pd.read_csv("data/beach_valid.csv", index_col=0, parse_dates=True)
               .resample("1M").mean()
               .interpolate(method="linear")
               .assign(Label = "Valid")
               .rename_axis(index="Time")
           )
beach_valid.index.freq = "M"
# Forecast index
forecast_index = create_forecast_index(beach_train.index[-1], 36, freq="M")
px.line(pd.concat((beach_train, beach_valid)), y="Shoreline", width=640, color="La
```



- We can use the `statsmodels` class `ARIMA()` to fit an `AR(p)`, `MA(q)`, `ARMA(p,q)` or `ARIMA(p,d,q)` model
- We specify the order of our model using the `order` argument which accepts a tuple of 3 values: `(p, d, q)`

```
# Fit models
model_ar = ARIMA(beach_train["Shoreline"], order=(3, 1, 0)).fit()
model_ma = ARIMA(beach_train["Shoreline"], order=(0, 1, 1)).fit()
model_arima = ARIMA(beach_train["Shoreline"], order=(3, 1, 1)).fit()
# index = index=df_beach.index.join(forecast_index, how="outer")
# Make forecasts
ar = pd.DataFrame({"Shoreline": pd.concat((model_ar.fittedvalues, model_ar.forecas
                   "Label": "ARIMA(3,1,0)"},
                  index=forecast_index).iloc[1:]
ma = pd.DataFrame({"Shoreline": pd.concat((model_ma.fittedvalues, model_ma.forecas
                   "Label": "ARIMA(0,1,1) fit"},
                  index=forecast_index).iloc[1:]
arima = pd.DataFrame({"Shoreline": pd.concat((model_arima.fittedvalues, model_arim
                     "Label": "ARIMA(3,1,1) fit"},
                    index=forecast_index).iloc[1:]
fig = px.line(pd.concat((beach_train,beach_valid, ar, ma, arima)), y="Shoreline",
fig.update_xaxes(range=["2010", "2020"])
```



- But how do we choose how many lags to include?
- Well, we can treat the model order as a hyperparameter to optimize, or we can use the ACF/PACF to help us choose an order (more on this later)

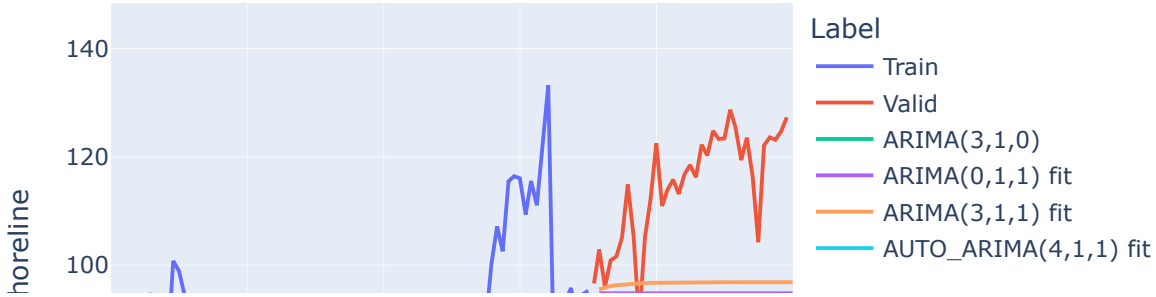
Let's use the function `auto_arima()` to perform an automatic grid search of p, d, q in ARIMA model

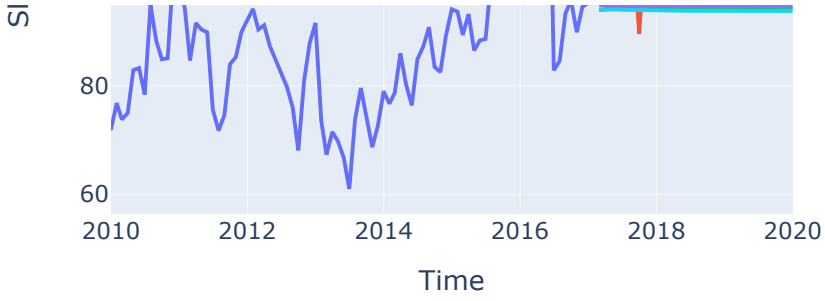
```
```

```
import pmdarima as pm
autoarima = pm.auto_arima(beach_train.Shoreline,
                          start_q=0,start_d=1,start_p=0,
                            max_q=5,  max_d=2,  max_p=5,
                          seasonal=False)
print(autoarima.summary())
```

```
                               SARIMAX Results
==============================================================================
Dep. Variable:                      y   No. Observations:                  489
Model:               SARIMAX(4, 1, 1)   Log Likelihood               -1693.006
Date:                Tue, 11 Mar 2025   AIC                           3398.013
Time:                        12:57:26   BIC                           3423.155
Sample:                    04-30-1976   HQIC                          3407.889
                         - 12-31-2016
Covariance Type:                  opg
==============================================================================
                 coef    std err          z      P>|z|      [0.025      0.975]
------------------------------------------------------------------------------
ar.L1          0.7181      0.048     14.878      0.000       0.623       0.813
ar.L2         -0.0928      0.060     -1.541      0.123      -0.211       0.025
ar.L3          0.0984      0.063      1.553      0.120      -0.026       0.223
ar.L4          0.0740      0.047      1.561      0.119      -0.019       0.167
ma.L1         -0.9615      0.025    -38.866      0.000      -1.010      -0.913
sigma2        60.3003      3.121     19.318      0.000      54.182      66.418
===================================================================================
Ljung-Box (L1) (Q):                   0.00   Jarque-Bera (JB):                 95.3
Prob(Q):                              0.99   Prob(JB):                          0.0
Heteroskedasticity (H):               0.80   Skew:                             -0.3
Prob(H) (two-sided):                  0.16   Kurtosis:                          5.0
===================================================================================

Warnings:
[1] Covariance matrix calculated using the outer product of gradients (complex-ste
```

```
autoarima = pd.DataFrame({"Shoreline": autoarima.predict(len(forecast_index)),
                          "Label": "AUTO_ARIMA(4,1,1) fit"},
                         index=forecast_index).iloc[1:]
fig = px.line(pd.concat((beach_train,beach_valid, ar, ma, arima, autoarima)), y="S
fig.update_xaxes(range=["2010", "2020"])
```



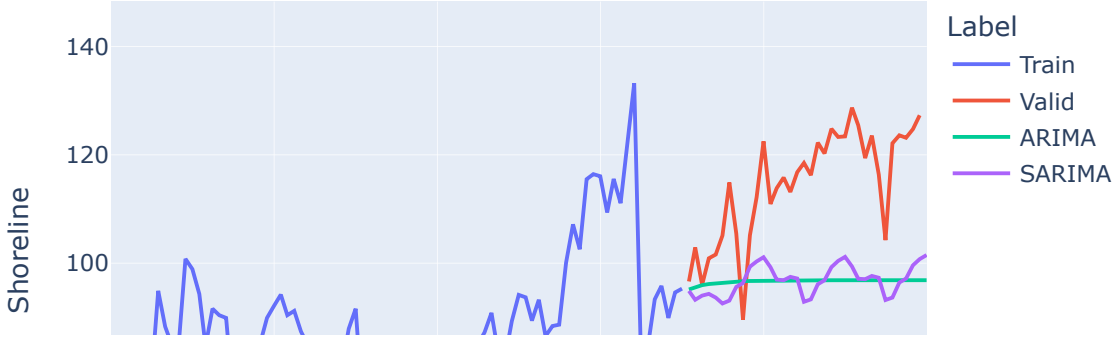


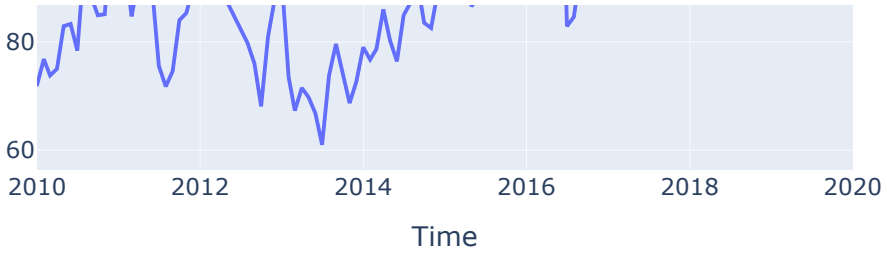
SARIMA

- So far we've only considered non-seasonal data, but don't worry, ARIMA can handle seasonality too. We call a seasonal ARIMA model, SARIMA
- SARIMA simply adds an ARIMA with seasonal terms, that is, seasonal differencing, and seasonally-based AR and MA models. It is often denoted `ARIMA(p,d,q)(P,D,Q,m)`), where `m` is the seasonal period.
- It's easiest to see by example so let's give it a shot with out beach shoreline data:

```
df_beach = beach_train
model_arima = ARIMA(df_beach["Shoreline"], order=(3, 1, 1)).fit()
model_sarima = ARIMA(df_beach["Shoreline"], order=(3, 1, 1), seasonal_order=(1, 1,
```

```
forecast_arima = pd.DataFrame({"Shoreline": model_arima.forecast(len(forecast_inde
                                "Label": "ARIMA"},
                               index=forecast_index)
forecast_sarima = pd.DataFrame({"Shoreline": model_sarima.forecast(len(forecast_in
                                 "Label": "SARIMA"},
                                index=forecast_index)
fig = px.line(pd.concat((beach_train, beach_valid, forecast_arima, forecast_sarima
fig.update_xaxes(range=["2010", "2020"])
fig
```





SARIMAX

- Finally, we can also add explanatory variables to the mix quite easily, by adding in the explanatory variable (x_t) to our ARMA equation:

$$y_t' = c + \theta x_t + \phi_1 y_{t-1}' + \phi_2 y_{2-1}' + \ldots + \phi_p y_{t-p}' + \theta_1 \epsilon_{t-1} + \theta_2 \epsilon_{t-2} + \ldots + \theta_q \epsilon_{t-q} + \epsilon_t$$

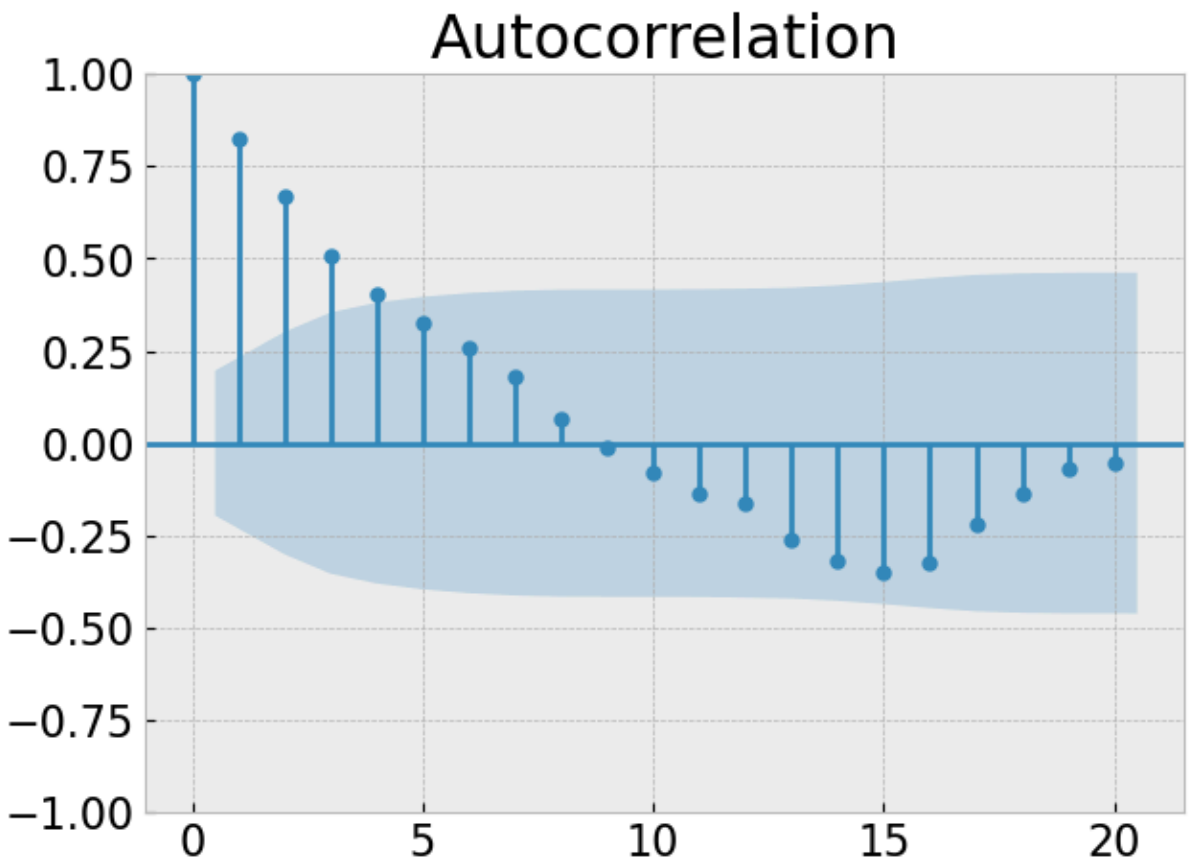
- In general, this might not be the most effective model, but it is possible by using the `exog` argument in the `ARIMA()` class.
- You can turn to ML to incorporate explanatory variables, but I'll touch on this topic again later in the course!

7. Choosing orders

Using correlograms ACF & PACF

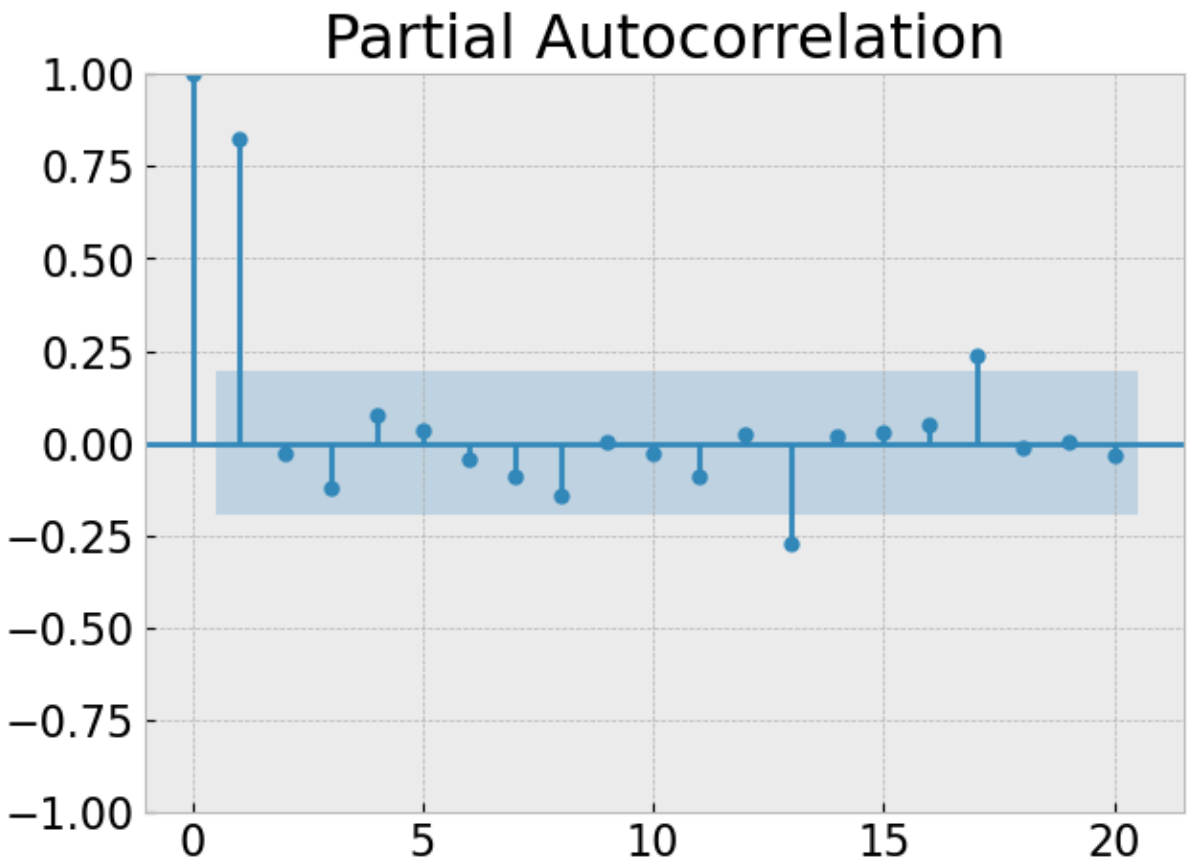
- Up until now, I've just been manually choosing the orders of my ARIMA models
- Usually I would treat these as hyperparameters and optimize them based on a validation set, but we can also use some other tools to help inform our choice of order them
- Recall that a corellogram (ACF plot) shows autocorrelations at different lags:

```
ar_data = ArmaProcess(ar=[1, -0.9]).generate_sample()
plot_acf(ar_data);
```



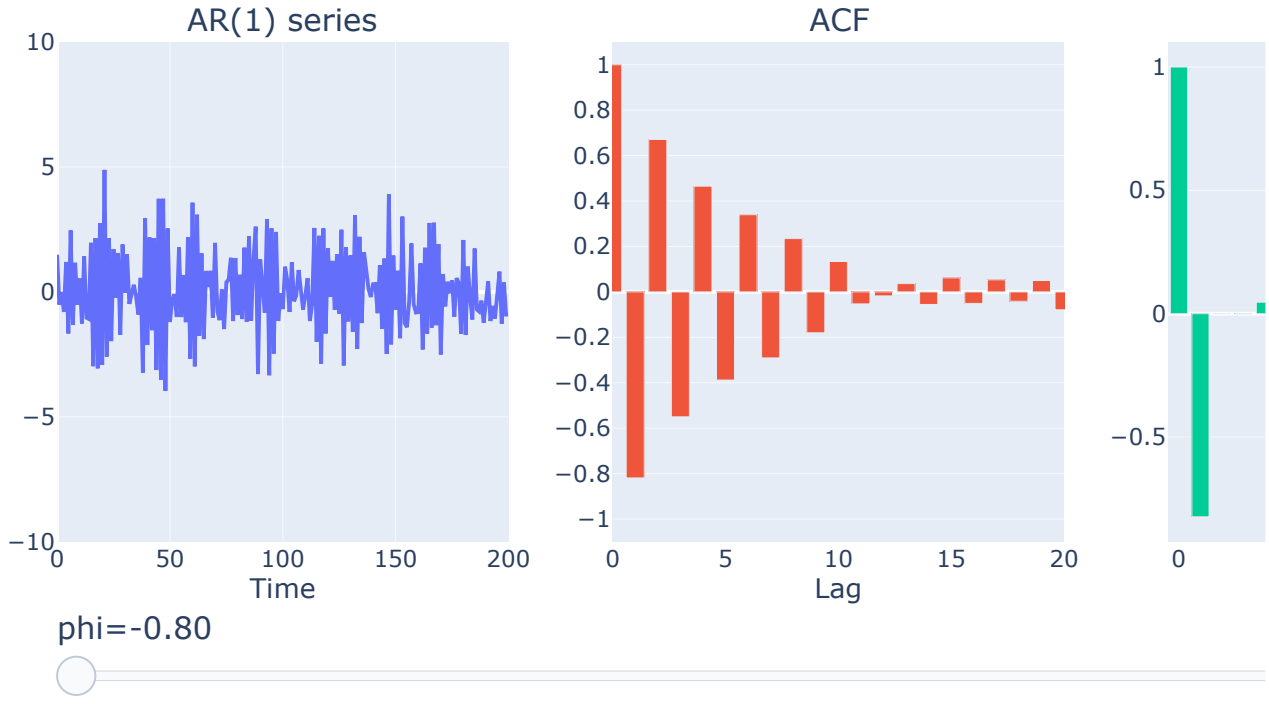
- To choose the order of an `AR(p)` model, this isn't all the helpful. We want to know which lags are important to include in our model, but a correlogram shows the correlation between y_t and y_{t-h}
- The problem is that if y_t and y_{t-1} are correlated, then y_{t-1} and y_{t-2} are also correlated, and therefore, y_t and y_{t-2} have indirect correlation (via y_{t-1}). This makes it hard to isolate exactly which lags are important in our series. We can clearly see this in the model above which should only show one significant lag.
- The solution to this problem is **partial autocorrelation** (PACF) which removes intermediate effects between y_t and y_{t-h}. The partial autocorrelation is estimated as the last coefficient in an autoregressive model. So $PACF(k)$ is the kth estimated coefficient in an `AR(k)` model.
- We can plot the PACF using the `statsmodels` function `plot_pacf()`:

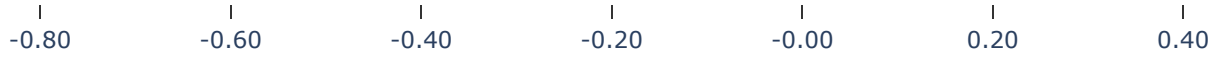
```
plot_pacf(ar_data);
```



- That looks much better! We can clearly see that one significant lag now
- In general, let's look at the behaviour of the ACF and PACF for an `AR(2)` model:

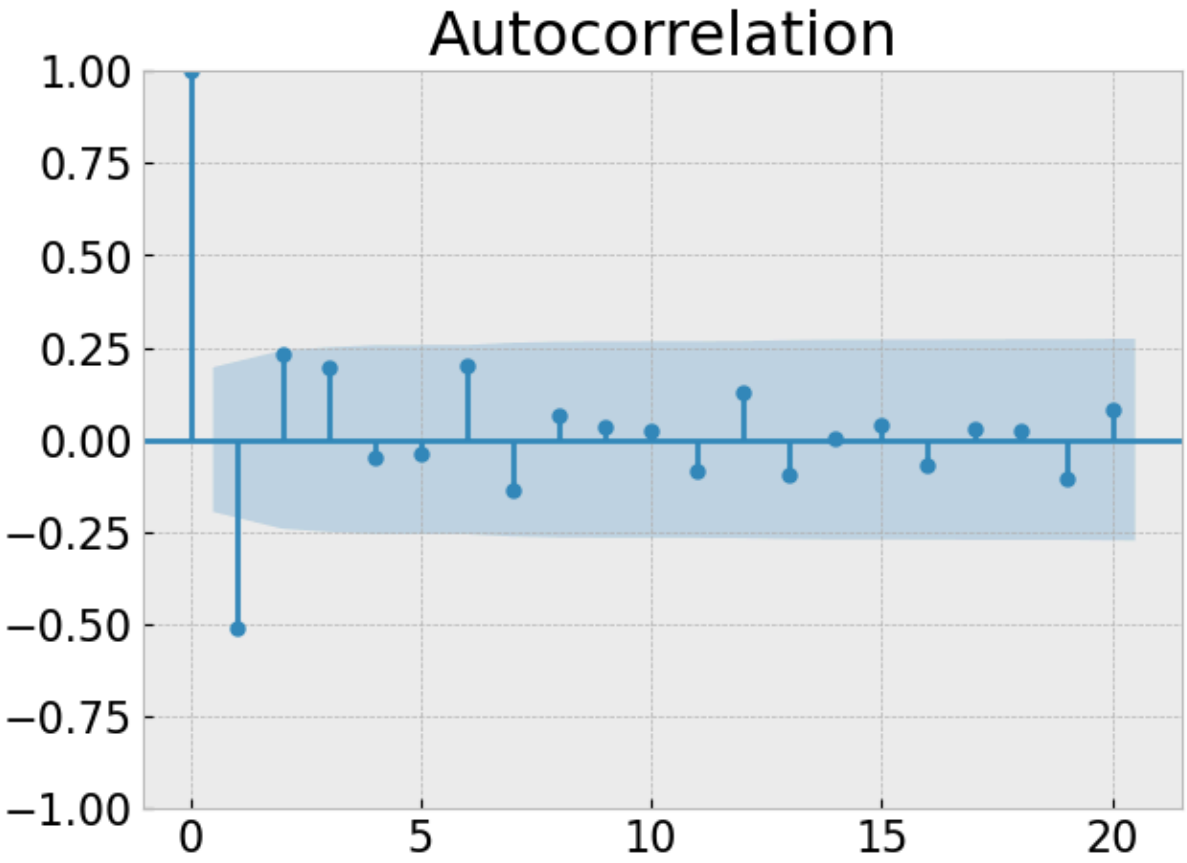
```
plot_acf_pacf("ar")
```





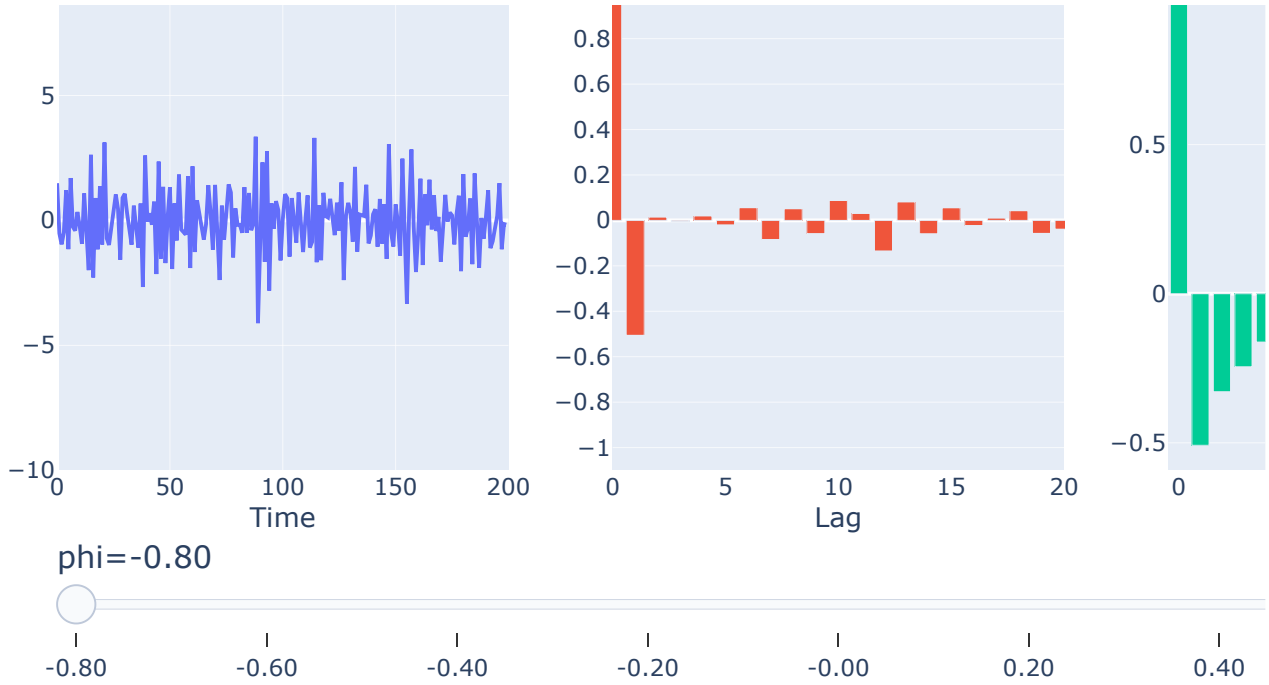
- What about the order of our `MA(q)` models?
- Well remember that `MA(q)` models are based on white noise. As we saw, the value of an `MA(q)` model at time t is a weighted sum of the last `q` values of the white noise process. But there is no dependence structure on values at lags higher than `q`, it's just white noise, so we would expect the ACF to "cut off" at lag `q`:

```
ma_data = ArmaProcess(ma=[1, -0.8, 0.8], nobs=500).generate_sample()
plot_acf(ma_data);
```



- That looks about right to me
- In general, the ACF of an `MA(q)` model reveals something about the order of the model, while the PACF usually doesn't tell us too much:

```
plot_acf_pacf("ma")
```



If you weren't quite following, here's a simple summary of all that:

Model	ACF	PACF
MA(q)	Cuts-off at lag q	Tails off, no pattern
AR(p)	Tails off (exponentially or like a "damped" sine wave)	Cuts-off at lag p

iClicker Question

Match the following time-series with one of the following options:

A. AR(1)

B. AR(2)

C. MA(1)

D. Whitenoise

Question 1

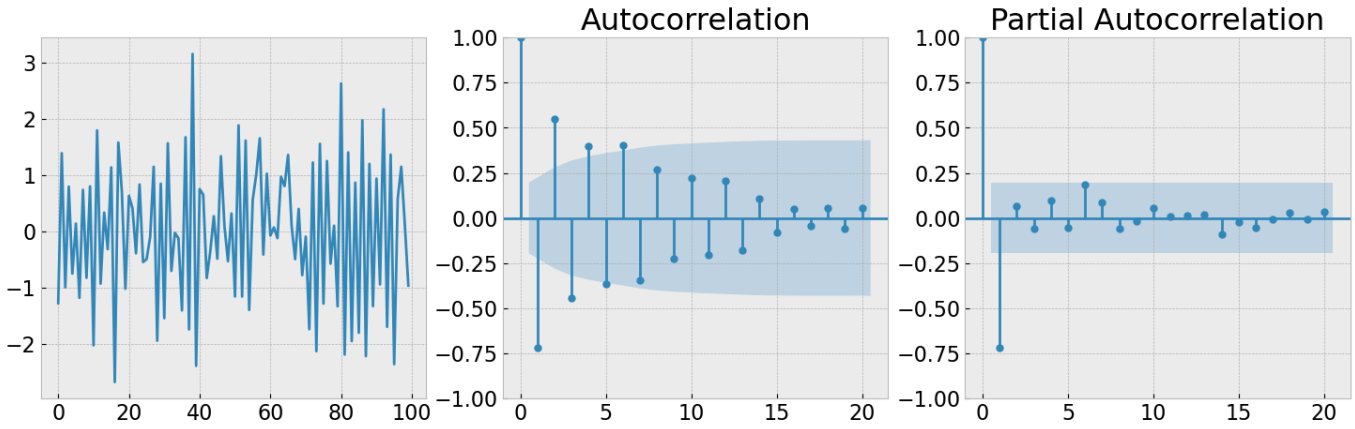
A. AR(1)

B. AR(2)

C. MA(1)

D. Whitenoise

▶ Show code cell source



Question 2

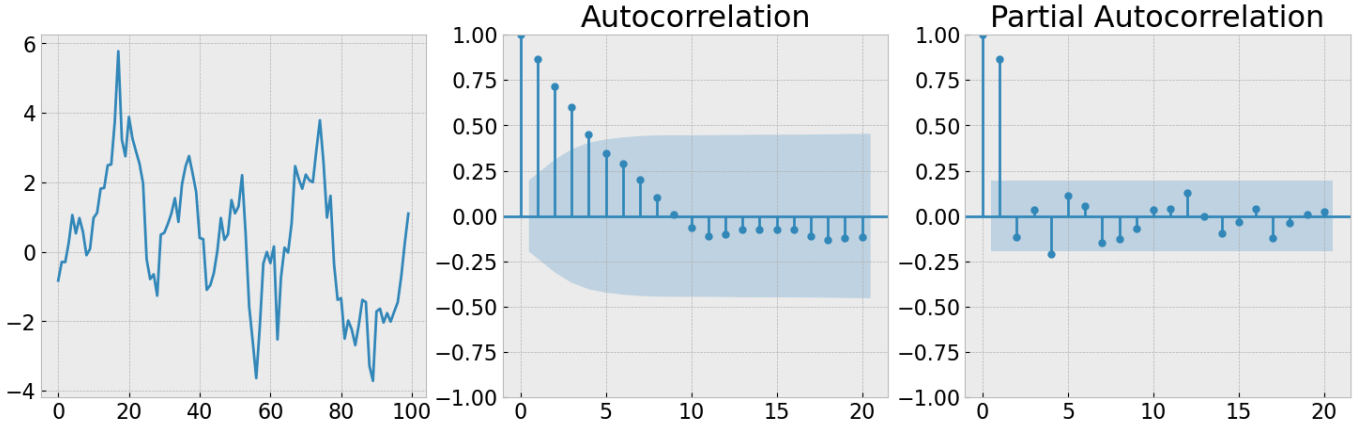
A. AR(1)

B. AR(2)

C. MA(1)

D. Whitenoise

▶ Show code cell source



Question 3

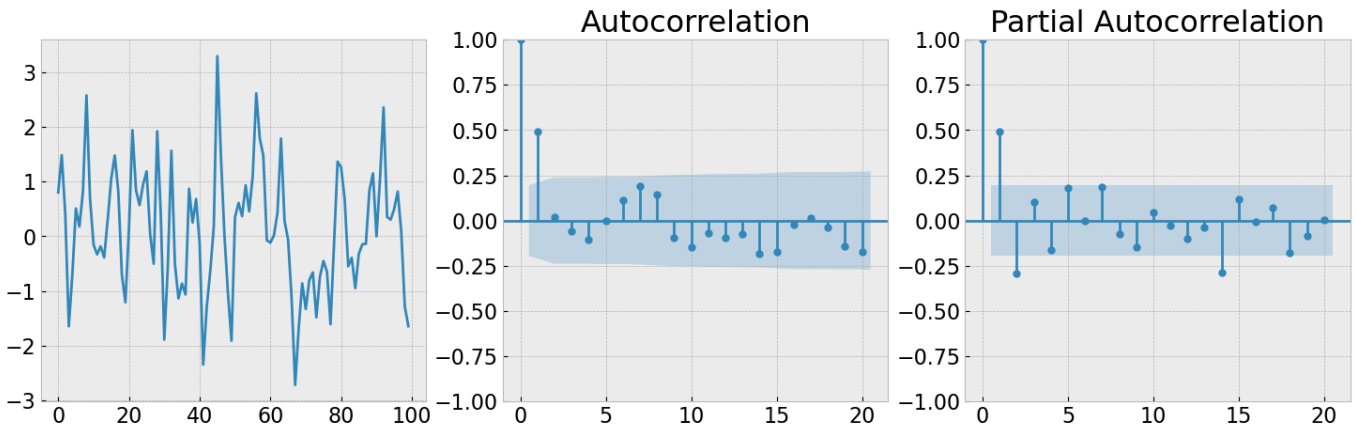
A. AR(1)

B. AR(2)

C. MA(1)

D. Whitenoise

▶ Show code cell source



Question 4

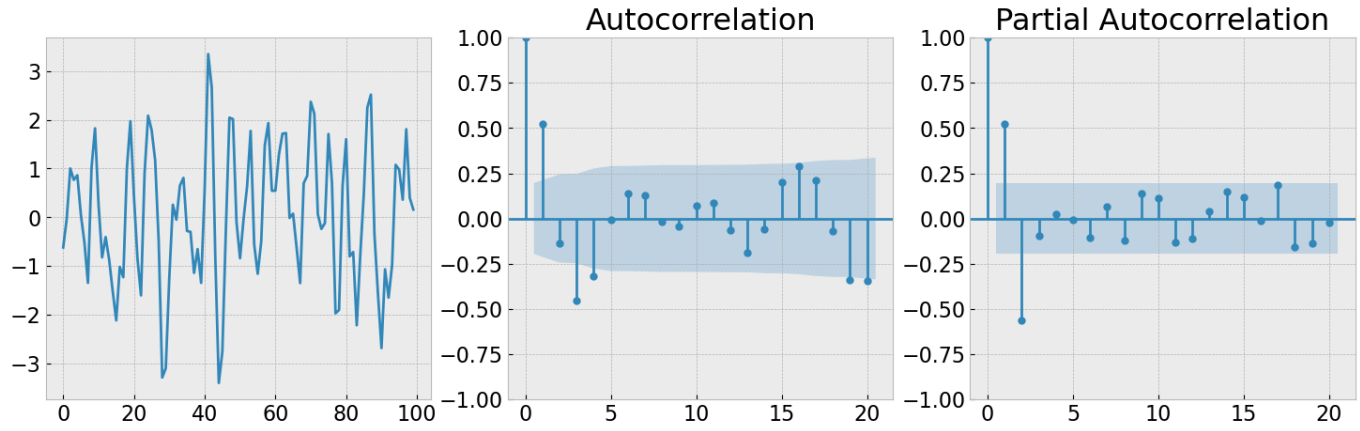
A. AR(1)

B. AR(2)

C. MA(1)

D. Whitenoise

▶ Show code cell source



Question 5

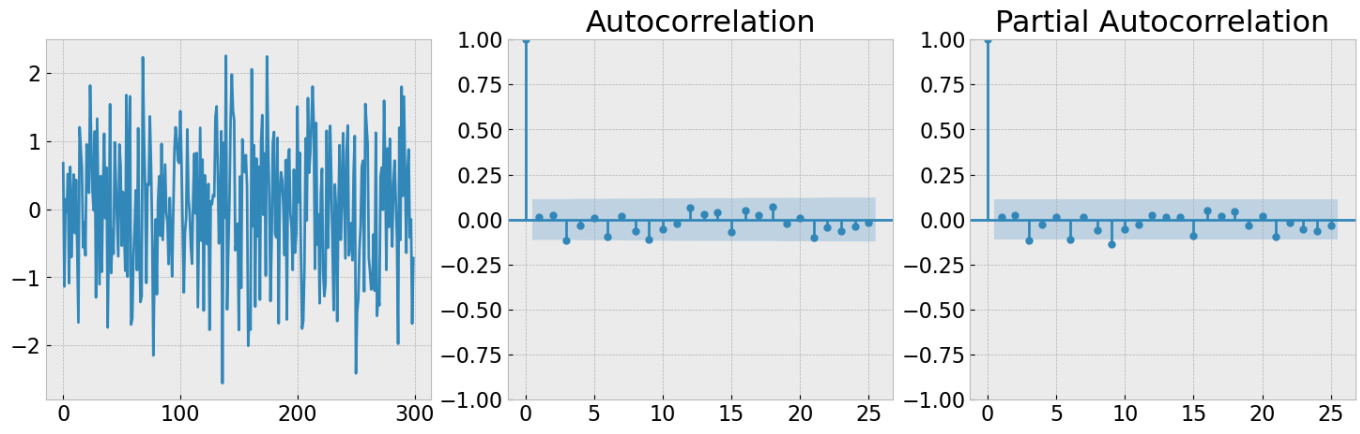
A. AR(1)

B. AR(2)

C. MA(1)

D. Whitenoise

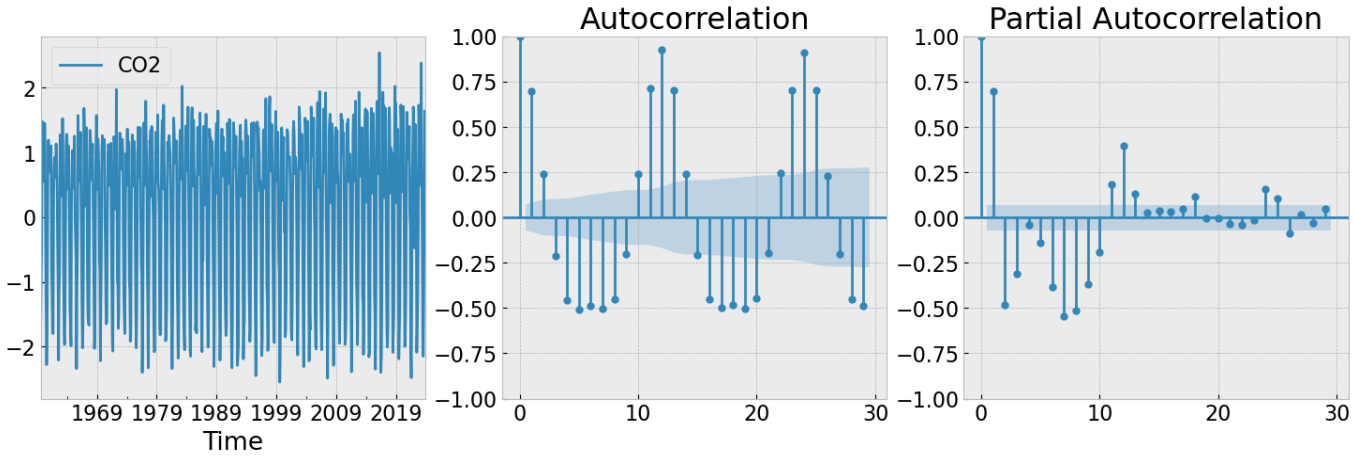
▶ Show code cell source



Auto-arima

The function performs a search (either stepwise or parallelized) over possible model & seasonal orders within the constraints provided, and selects the parameters that minimize the given metric.

```
df = pd.read_csv("data/temperature.csv", index_col=0, parse_dates=True)
fig, axes = plt.subplots(nrows=1, ncols=3, figsize=(18, 5))
df = df.diff().dropna()
df.plot(ax=axes[0])
plot_acf(df, ax=axes[1])
plot_pacf(df, ax=axes[2]);
```



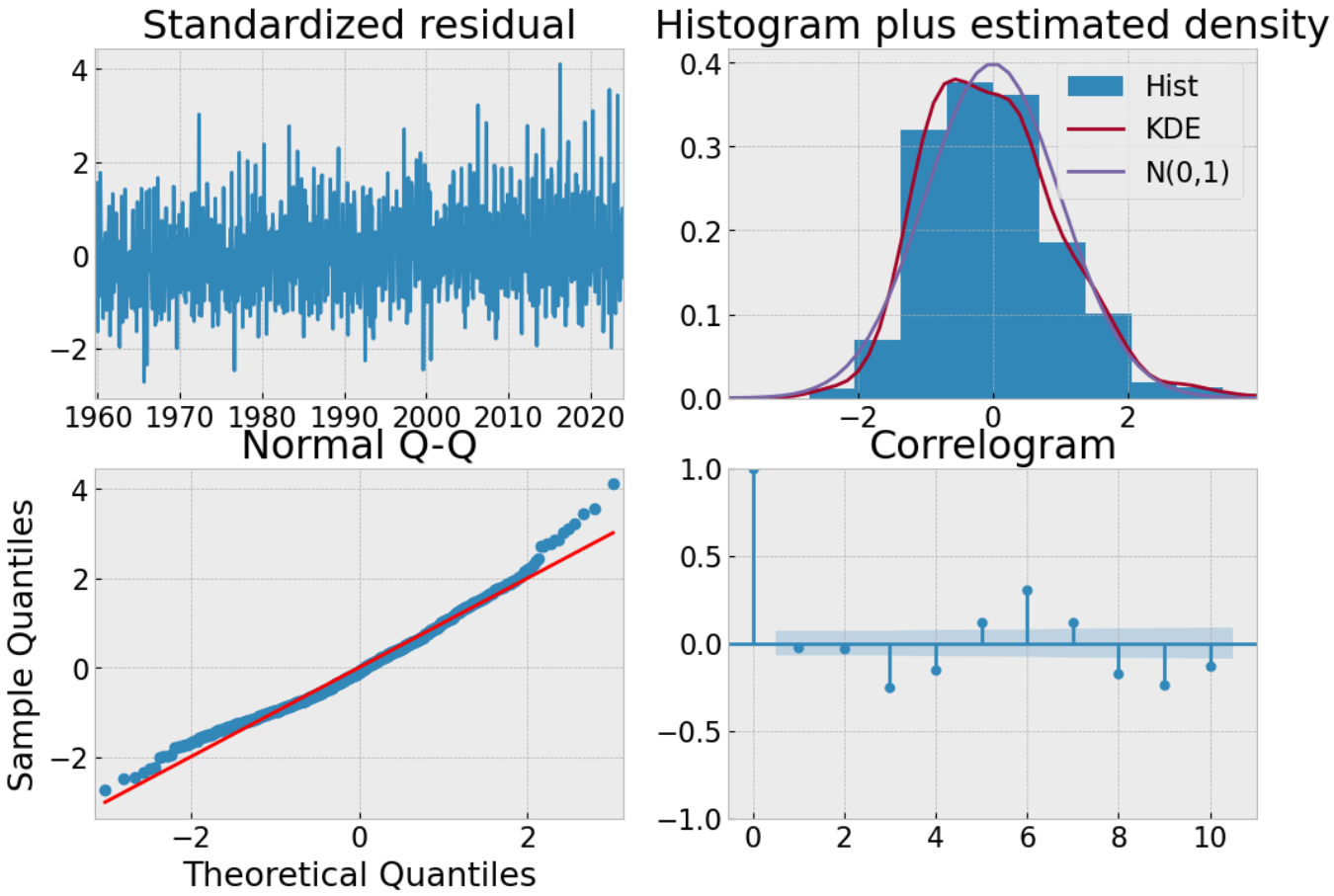

```
df = pd.read_csv("data/temperature.csv", index_col=0, parse_dates=True)
results = pm.auto_arima(df, start_q=0,start_d=1,start_p=0,
                        max_q=5, max_d=2, max_p=5)
print(results.summary())
```

```
                               SARIMAX Results
==============================================================================
Dep. Variable:                           y   No. Observations:                  774
Model:                 SARIMAX(2, 1, 1)   Log Likelihood                -771.864
Date:                  Tue, 11 Mar 2025   AIC                           1553.727
Time:                          12:57:30   BIC                           1576.979
Sample:                      07-31-1959   HQIC                          1562.674
                           - 12-31-2023
Covariance Type:                    opg
==============================================================================
                 coef    std err          z      P>|z|      [0.025      0.975]
------------------------------------------------------------------------------
intercept      0.0416      0.003     11.963      0.000       0.035       0.048
ar.L1          1.5425      0.022     70.305      0.000       1.499       1.585
ar.L2         -0.8459      0.022    -37.666      0.000      -0.890      -0.802
ma.L1         -0.8987      0.018    -49.618      0.000      -0.934      -0.863
sigma2         0.4297      0.022     19.908      0.000       0.387       0.472
===================================================================================
Ljung-Box (L1) (Q):                   0.51   Jarque-Bera (JB):                 44.6
Prob(Q):                              0.47   Prob(JB):                          0.0
Heteroskedasticity (H):               1.29   Skew:                              0.5
Prob(H) (two-sided):                  0.04   Kurtosis:                          3.4
===================================================================================

Warnings:
[1] Covariance matrix calculated using the outer product of gradients (complex-ste
```

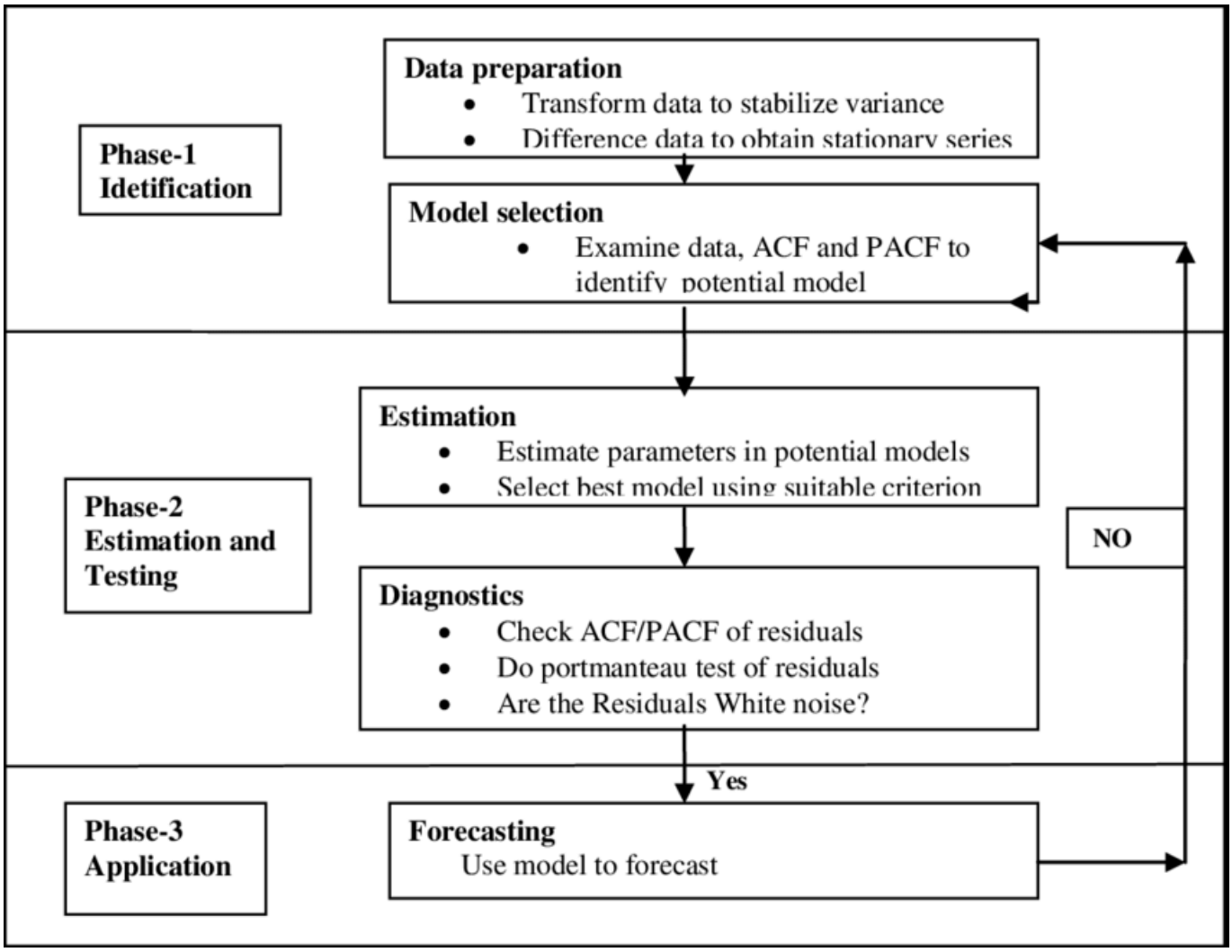
Residuals diagnostics

- Are the residuals normally distributed?
- Do the residuals have constant mean and variance?
- Are the residuals not auto-correlated?

```
results.plot_diagnostics(figsize=(12,8));
```



Box-Jenkins approach



Practical guidance

- In my opinion, time series forecasting is as much an art as a science
- It requires a good understanding of the data and plenty of EDA
- My typical approach is as follows
  - Naive (seasonally-adjusted if necessary) / Seasonal naive
  - ETS (for SES, or Holt, seasonally-adjust if necessary)
  - ARIMA
  - ML (next lecture)
  - Combinations of the above
- By the way, after your ML courses you're probably thinking that more data is always better. But for time series, this is not always the case, because sometimes, data well in the past are not representative of the data-generating process in the present, and a particular model may

not be the best fit for all of the data. It's often useful to consider what happens to your model and forecasts if you base it on only a subset of the most recent data.