Machine Learning for Time Series Analysis

Forecasting

Fortunato Nucera[1]

[1]Department of Mathematics
Imperial College London

Final Project Preparation - Week 1-2, 9th-16th May 2023

Table of Contents

Table of Contents

Time Series Fundamentals

The main resource for this section is [1].

Definition 1

A time series is a set of observations $x \in \{x_1, x_2, \ldots, x_T\}$ recorded at time $t \in T_0 = \{t_1, t_2, \ldots, T\}$. If t belongs to a countable set, then the time series is **discrete**, otherwise the time series is **continuous**. In literature, continuous time series are indicated with the notation $x(t)$ whereas discrete time series are indicated with x_t.

Importantly, given the set of observations constituting a time series, this set is **ordered**. That is, two time series are different unless all elements *at each time step* match.

The class of *stochastic processes* is useful for the analysis of time series.

Definition 2

A stochastic process is a family of random variables $\{X_t, t \in T_0\}$ defined on a probability space.

In particular, given the time set T_0, we can regard a time series as the realization of an underlying stochastic process defined as the joint probability distribution over $t \in T_0$. For example, if $T_0 = \{t_1, t_2, \ldots, T\}$, then the time series is the realization of the probability distribution

$$\mathrm{P}^X_{t_1,t_2,\ldots,T} = \mathrm{P}(X_{t_1}, X_{t_2}, \ldots, X_T)$$

where the superscript X indicates the considered random variable whereas the subscript indicates the time steps.

The order of the elements in the time series matters. The elements in the time series are not *exchangeable*. Since exchangeability is a necessary condition for independence, then the elements of the time series are not independent. Thus, $\mathrm{P}^X_{t_1,t_2,\ldots,T}$ **cannot be factorized**.

Table of Contents

Covariance & Correlation

Definition 3

Let $\{X_t\}, \{Y_t\}$ with $t \in T_0$ be two stochastic processes defined on the same time set. The autocovariance, is the covariance between the random variables of a stochastic process at different time steps:

$$C_{XX}(t_1, t_2) = \text{cov}[X_{t_1}, X_{t_2}], \text{ where } t_1, t_2 \in T_0$$

The cross-covariance is the covariance between the random variables of two distinct stochastic processes at different time steps:

$$C_{XY}(t_1, t_2) = \text{cov}[X_{t_1}, Y_{t_2}], \text{ where } t_1, t_2 \in T_0$$

Definition 4

Autocorrelation and cross-correlation are defined from autocovariance and cross-covariance by normalizing through the standard deviation of each stochastic process becoming, respectively:

$$\rho_{XX}(t_1, t_2) = \frac{C_{XX}(t_1, t_2)}{\sigma_{X,t_1}\sigma_{X,t_2}}, \text{ where } t_1, t_2 \in T_0$$

$$\rho_{XY}(t_1, t_2) = \frac{C_{XY}(t_1, t_2)}{\sigma_{X,t_1}\sigma_{Y,t_2}}, \text{ where } t_1, t_2 \in T_0$$

Stationarity: Weak and Strong

Stationarity is a property time series inherit from stochastic processes. The word *stationarity* is used, in literature, to refer to *weak stationarity* although two distinct concepts exist.

Weak Stationarity (or Stationarity)

A time series $\{X_t\}$ is weakly-stationary if *all* the conditions below are met

- Constant and finite first-order and second-order moments:

$$\mathbb{E}[X_{t_1}] = \mathbb{E}[X_{t_2}] < \infty, \text{ where } t_1, t_2 \in T_0$$

$$\text{Var}[X_{t_1}] = \text{Var}[X_{t_2}] < \infty, \text{ where } t_1, t_2 \in T_0$$

- Constant autocovariance (or autocorrelation) at a given lag h:

$$C_{XX}(t_1, t_2) = C_{XX}(t_1, t_1 + h) = C_{XX}(t_1 + t, t_2 + t) = C_{XX}(0, h)$$

where the lag $h = t_2 - t_1$ and $\forall t \in T_0$.

Strong Stationarity

A time series $\{X_t\}$ is strongly-stationary if its underlying joint distribution is constant for constant shift of the indices.

$$\mathrm{P}^X_{t_1,t_2,\ldots,t_n} = \mathrm{P}^X_{t_{1+h},t_{2+h},\ldots,t_{n+h}}$$

where $h \in \mathbb{Z}$.

In general, if a strongly-stationary time series has finite second-order moment $\mathbb{E}[X_t^2] < \infty$, then the time series is also weakly-stationary. The reverse is not necessarily true (unless an assumption on the underlying distribution of the stochastic process is made - i.e. Gaussian Stochastic Processes).
Weak Stationarity is normally assessed via the Augmented Dickey Fuller Test (H_0: non-stationarity of the time series. Test available in `statsmodels.tsa.stattools.adfuller`).

Table of Contents

Time Series Decomposition

A given time series $\{X_t\}$ can be decomposed into 3 distinct components:

- m_t: trend
- s_t: seasonal component
- Y_t: noise component (RV)

Decomposition can be:

- *additive*: $X_t = m_t + s_t + Y_t$
- *multiplicative* : $X_t = m_t \times s_t \times Y_t$

The multiplicative formulation is preferred when the seasonality amplitude and/or the trend slope are not constant.

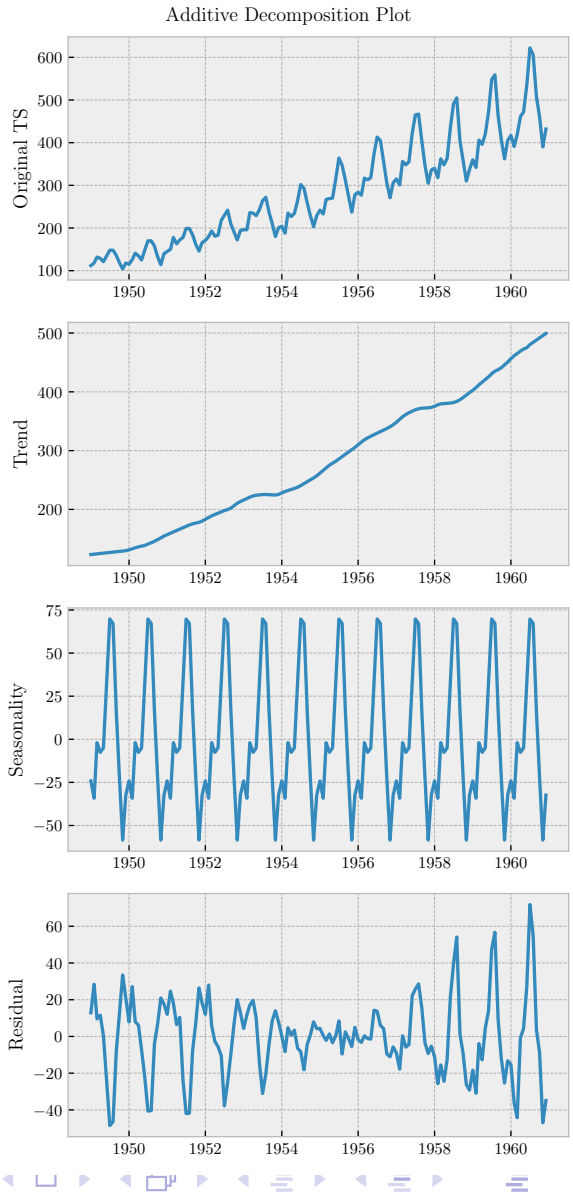


Table of Contents

In this section, we introduce the Backshift and the difference operators, which will be useful for the description of the ARIMA models.

Definition 5

The backshift operator B maps the value of a time series at step t to the value of the same series at step $t-1$.

$$Bx_t := x_{t-1}$$

The backshift operator can be used to compactly write time series expressions.

Examples

$$x_{t-12} + x_t - 2x_{t-2} = B^{12}x_t + x_t - 2B^2x_t = (B^{12} + 1 - 2B^2)x_t$$

The backshift operator is also used to define the difference operator.

Differencing in time series is essential to remove trends and/or seasonalities (thus making the time series stationary)

Definition 6

The difference operator ∇ represents the difference between the current and previous value of a time series.

$$\nabla x_t := x_t - x_{t-1} = (1 - B)x_t$$

The Box & Jenkins seasonal differencing allows to perform differencing s time steps apart.

$$\nabla_s x_t := x_t - x_{t-s} = (1 - B^s)x_t$$

The backshift notation allows to write differenced time series expressions easily.

Examples

A 12-period differencing followed by a 1-period differencing could be written as:

$$\nabla\nabla_{12}x_t = (1 - B)(1 - B^{12})x_t = (1 - B - B^{12} - B^{13})x_t =$$
$$= x_t - x_{t-1} - x_{t-12} - x_{t-13}$$

De-trending via Differencing

Examples

Suppose a time series can be described via the following function:

$$x_t = q + mt + \epsilon_t \quad \text{with } \epsilon_t \sim \mathcal{N}(0, \sigma^2)$$

Then $\mathbb{E}[x_t] = q + mt$, which means that the process cannot be stationary as the mean depends on time. However, using the difference operator:

$$\nabla x_t = x_t - x_{t-1} = q + mt + \epsilon_t - q - m(t-1) - \epsilon_{t-1} = m + \nabla\epsilon_t$$

From this, we can see that $\mathbb{E}[\nabla x_t] = m$ and:

$$C_{xx}(0, h) = \begin{cases} \sigma^2 & \text{if } h = 0 \\ 0 & \text{otherwise} \end{cases}$$

Therefore the series becomes stationary.

Table of Contents

Forecasting can be carried out in a multitude of ways:

- one-step forecasting returns one prediction at a time
- *n*-step forecasting returns *n* steps, where the prediction at each time step is plugged into the model to obtain the prediction at the following time step (serial in nature). Errors tend to propagate.
- multi-output forecast: a number of models are trained in parallel and produce *n* predictions *simultaneously*.

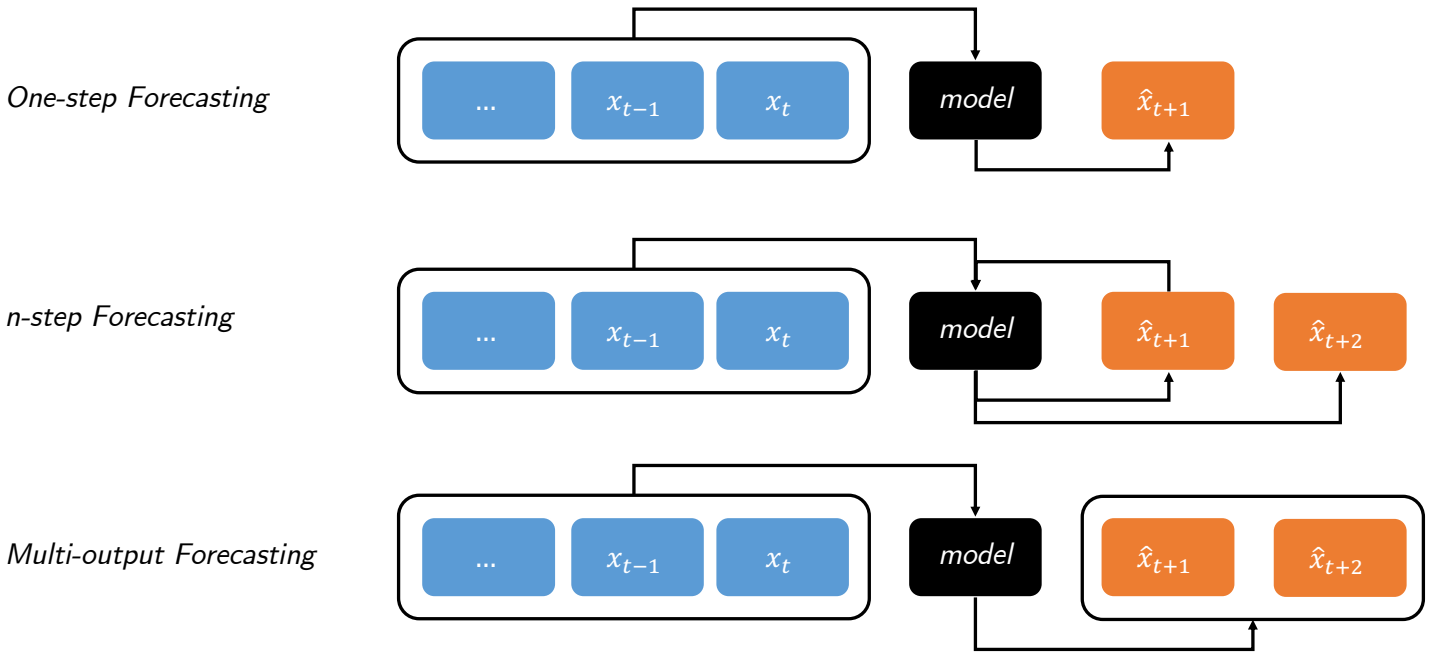


Table of Contents

Forecast Models: An introduction

The following sections contain information and examples of Exponential Smoothing Models (among which, the popular Holt-Winters model) and ARIMA models. Understanding these popular architectures will allow us to generalize the time series forecast models to other model types commonly used outside the scope of time series analysis (Random Forest, Bayesian Additive Regression Trees, Support Vector Regressors etc). Finally, we will provide an example of Deep Learning architectures using a Convolutional Neural Network (CNN). CNN could be easily replaced with Long-Short Term Memory networks or Transformer models (we did not include those owing to lack of time).
The Exponential Smoothing Models are based on interpolations of past values for level, trend, and seasonality. The models' theory can be found at [2].

Simple Exponential Smoothing (SES) - 1

The future prediction is the weighted average of the previous exponentially weighted moving average (EWMA) and the current value.

$$\hat{y}_{t+1|t} = \alpha y_t + (1-\alpha)\hat{y}_{t|t-1}$$

α is the weight given to the most recent observation, normally tuned by minimizing the in-sample mean squared error. The forecast equation can be rewritten as:

$$\text{forecast:} \quad \hat{y}_{t+h|t} = l_t \quad \text{with } h = 1, 2, 3 \tag{1}$$

$$\text{level:} \quad l_t = \alpha y_t + (1-\alpha) l_{t-1} \tag{2}$$

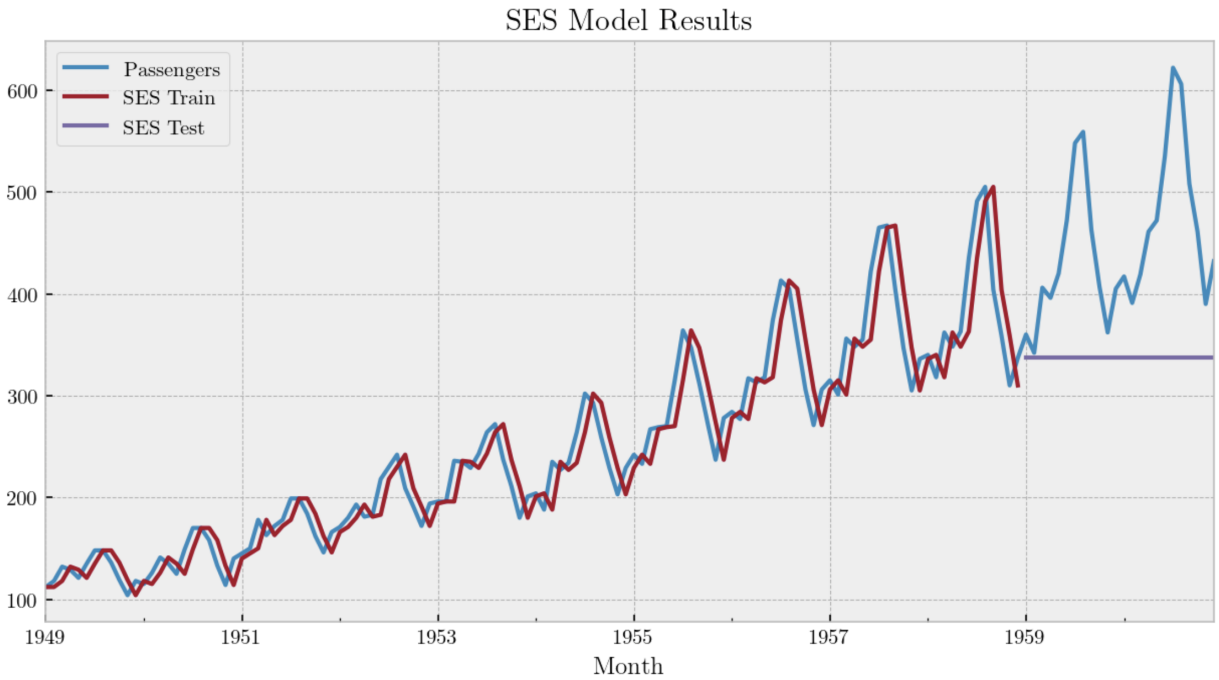
l_t is called the *level*, the average value of the signal in time. h is the forecast horizon. Regardless of h, the future value is constant and corresponding to the most recently observed value of l_t, which in turn depends on y_t. $\hat{y}$ indicates the model's prediction. SES can be easily fit by using the dedicated `statsmodels` class `statsmodels.tsa.holtwinters.SimpleExpSmoothing`.
In the following slide, please note the delay in the forecast. This is not a mistake: SES is copying the past values of the time series into the future (common issue in time series forecast models). This means that the forecast is *delayed*.

Simple Exponential Smoothing (SES) - 2

```python
import pandas as pd
from statsmodels.tsa.holtwinters import SimpleExpSmoothing

df = pd.read_csv("airline_passengers.csv", parse_dates=True, index_col=0)
df.index.freq = "MS" # set frequency

# define training and test set
Ntest = 24 # predict last two years
training = df.iloc[:-Ntest,:]
test = df.iloc[-Ntest:,:]
training_idx = df.index <= training.index[-1]
test_idx = ~training_idx
# fit model
model = SimpleExpSmoothing(training["Passengers"])
result = model.fit()
# predict in-sample and out-of-sample
df.loc[training_idx,"SES Train"] = result.fittedvalues
df.loc[test_idx,"SES Test"] = result.forecast(steps=Ntest)
# produce plot
df.plot(title="SES Model Results");
```



Holt's Linear Model -1

Holt's Linear Model adds the trend equation to the forecast and level equations, thus allowing a closer fit to the time series:

$$\begin{aligned} \text{forecast:} \quad & \hat{y}_{t+ht} = l_t + hb_t \quad \text{with } h = 1, 2, 3 \\ \text{level:} \quad & l_t = \alpha y_t + (1-\alpha)(l_{t-1} + b_{t-1}) \quad \text{(1-step ahead prediction)} \\ \text{trend:} \quad & b_t = \beta(l_t - l_{t-1}) + (1-\beta)b_{t-1} \end{aligned}$$

The forecast is now a line which has, as trend, the final observed trend b_t. The intercept is, instead, the final observed level l_t. Importantly, we first solve the level equation, obtain l_t, and then solve the trend equation to obtain b_t. Both α and β are tuned so that the in-sample sum of squared errors (RSS) is minimized. Holt's Linear model can be fit using `statsmodels.tsa.holtwinters.Holt` .

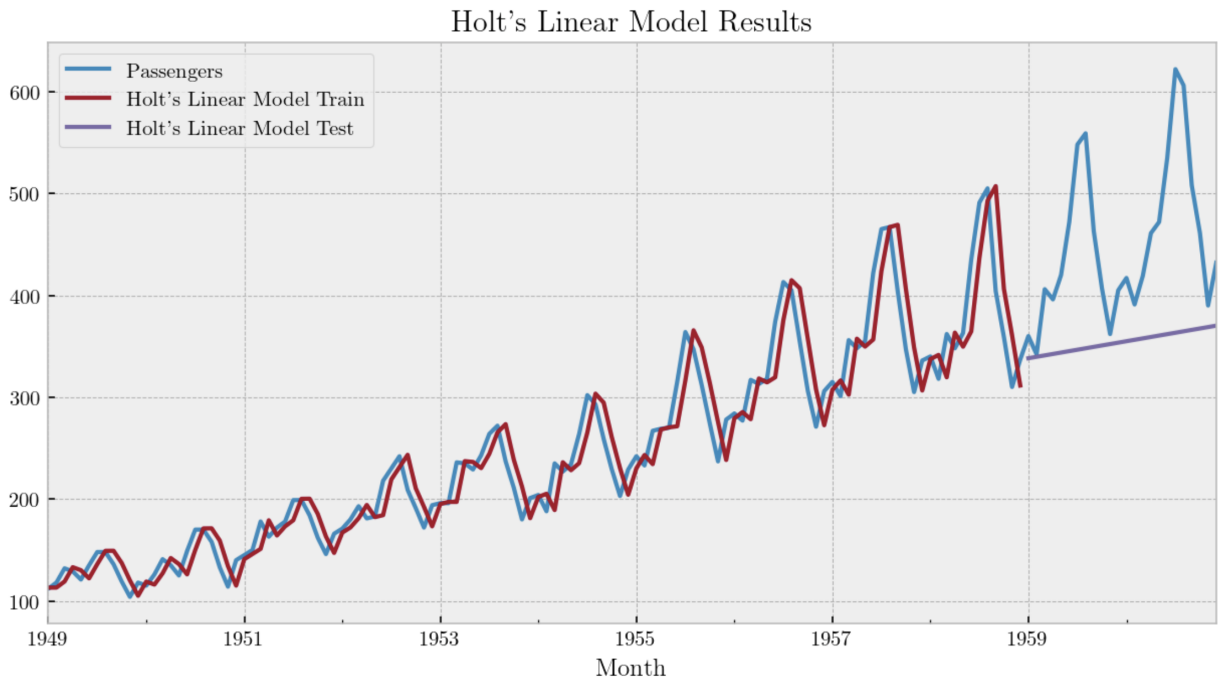
The prediction is now a line going upward (not a constant anymore)

Holt's Linear Model - 2

```python
import pandas as pd
from statsmodels.tsa.holtwinters import Holt

df = pd.read_csv("airline_passengers.csv", parse_dates=True, index_col=0)
df.index.freq = "MS" # set frequency

# define training and test set
Ntest = 24 # predict last two years
training = df.iloc[:-Ntest,:]
test = df.iloc[-Ntest:,:]
training_idx = df.index <= training.index[-1]
test_idx = ~training_idx
# fit model
model = Holt(training["Passengers"])
result = model.fit()
# predict in-sample and out-of-sample
df.loc[training_idx,"Holt's Linear Model Train"] = result.fittedvalues
df.loc[test_idx,"Holt's Linear Model Test"] = result.forecast(steps=Ntest)
# produce plot
df.plot(title="Holt's Linear Model Results");
```



Holt-Winters Model - 1

SES and Holt's Linear model do not account for the *seasonality* in the time series.
Holt-Winters improves on both of them.
Seasonality may be either *additive* or *multiplicative*.

- **Additive Seasonality Model:** in this model, the forecast is $y = level + trend + seasonality$. The equations for the model are below.

$$\begin{aligned}
\text{forecast:} \quad & \hat{y}_{t+ht} = l_t + hb_t + s_{t+h-mk} \quad \text{with } h = 1, 2, 3 \\
\text{level:} \quad & l_t = \alpha(y_t - s_{t-m}) + (1-\alpha)(l_{t-1} + b_{t-1}) \quad \text{(1-step ahead prediction)} \\
\text{trend:} \quad & b_t = \beta(l_t - l_{t-1}) + (1-\beta)b_{t-1} \\
\text{seasonality:} \quad & s_t = \gamma(y_t - l_{t-1} - b_{t-1}) + (1-\gamma)s_{t-m}
\end{aligned}$$

where:

- m is the period of the cycle in terms of the time step frequency (say, for example, if the seasonality of the time series is 12 months, then $m = 12$).
- t is the last known time for which we have observations.
- h is the forecast horizon.
- k finds the matching seasonal component $k = \text{floor}\left(\frac{h-1}{m}\right) + 1$

Holt-Winters Model - 2

- **Multiplicative Seasonality Model:** in this model, the forecast is $y = (level + trend) \times seasonality$

$$\begin{aligned}
\text{forecast:} \quad & \hat{y}_{t+ht} = (l_t + hb_t)s_{t+h-mk} \quad \text{with } h = 1, 2, 3 \\
\text{level:} \quad & l_t = \alpha(y_t / s_{t-m}) + (1-\alpha)(l_{t-1} + b_{t-1}) \quad \text{(1-step ahead prediction)} \\
\text{trend:} \quad & b_t = \beta(l_t - l_{t-1}) + (1-\beta)b_{t-1} \\
\text{seasonality:} \quad & s_t = \gamma \left(\frac{y_t}{l_{t-1} + b_{t-1}} \right) + (1-\gamma)s_{t-m}
\end{aligned}$$

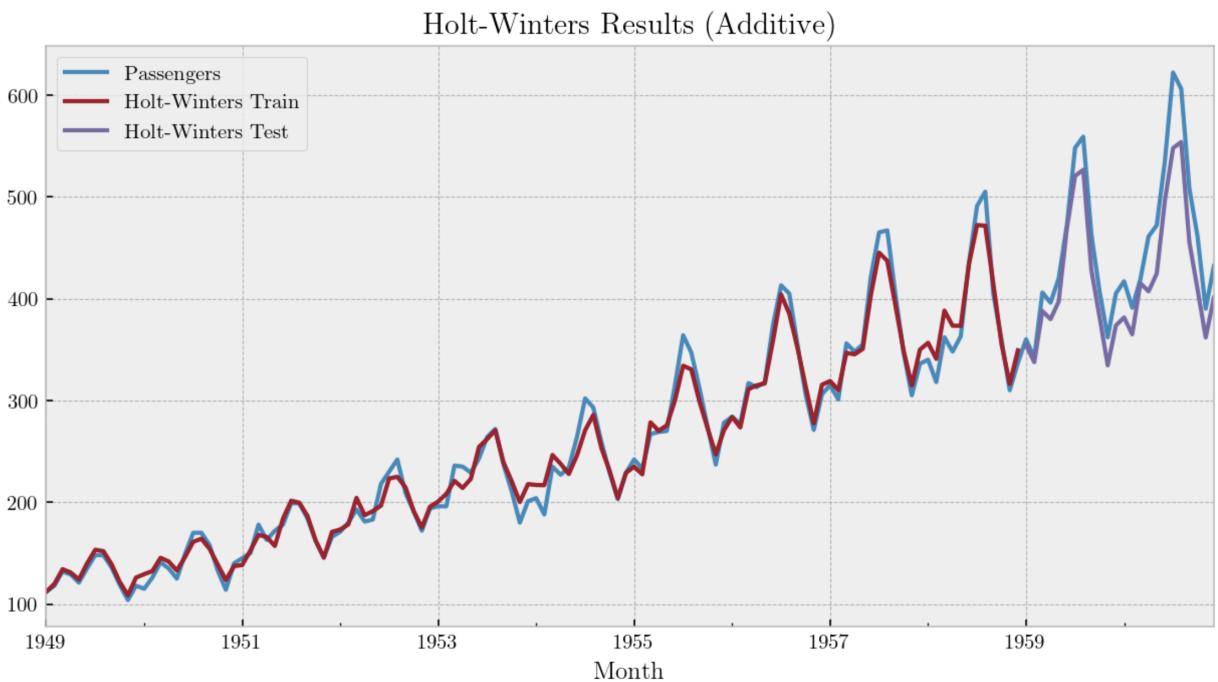
The Holt-Winters model can be implemented by using `statsmodels.tsa.holtwinters.ExponentialSmoothing`.

Holt-Winters Model - 3

```python
import pandas as pd
from statsmodels.tsa.holtwinters import ExponentialSmoothing

df = pd.read_csv("airline_passengers.csv", parse_dates=True, index_col=0)
df.index.freq = "MS" # set frequency

# define training and test set
Ntest = 24 # predict last two years
training = df.iloc[:-Ntest,:]
test = df.iloc[-Ntest:,:]
training_idx = df.index <= training.index[-1]
test_idx = ~training_idx
# fit model
model = ExponentialSmoothing(training["Passengers"], trend='add', seasonal='add', seasonal_periods=12)
result = model.fit()
# predict in-sample and out-of-sample
df.loc[training_idx,"Holt-Winters Train"] = result.fittedvalues
df.loc[test_idx,"Holt-Winters Test"] = result.forecast(steps=Ntest)
# produce plot
df.plot(title="Holt-Winters Results (Additive)");
```

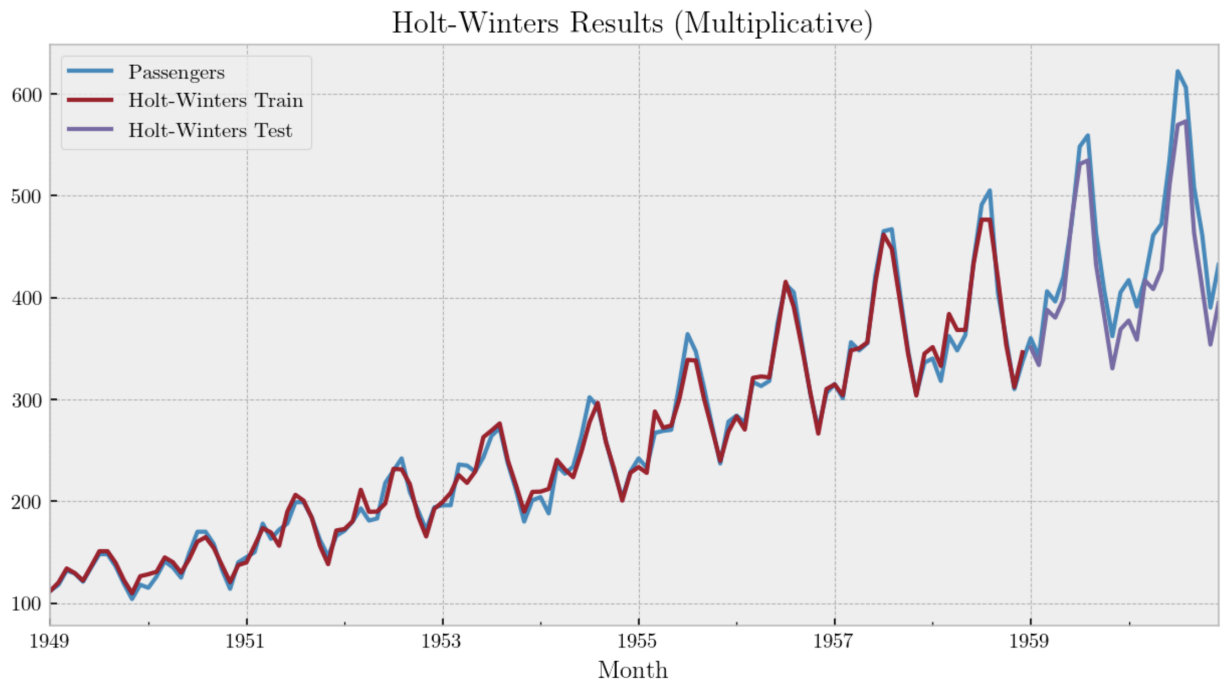


Holt-Winters Model - 4

```python
import pandas as pd
from statsmodels.tsa.holtwinters import ExponentialSmoothing

df = pd.read_csv("airline_passengers.csv", parse_dates=True, index_col=0)
df.index.freq = "MS" # set frequency

# define training and test set
Ntest = 24 # predict last two years
training = df.iloc[:-Ntest,:]
test = df.iloc[-Ntest:,:]
training_idx = df.index <= training.index[-1]
test_idx = ~training_idx
# fit model
model = ExponentialSmoothing(training["Passengers"], trend='add', seasonal='mul', seasonal_periods=12)
result = model.fit()
# predict in-sample and out-of-sample
df.loc[training_idx,"Holt-Winters Train"] = result.fittedvalues
df.loc[test_idx,"Holt-Winters Test"] = result.forecast(steps=Ntest)
# produce plot
df.plot(title="Holt-Winters Results (Multiplicative)");
```



ARIMA models

An Autoregressive Integrated Moving Average Model (ARIMA) is a model used to forecast future values of a time series based on information available at prior time steps. ARIMA is the most general model and, in order to discuss it, we might proceed by induction from simpler models:

- White Noise Process
- Random Walk Model
- Moving Average Model MA(q)
- Autoregressive Model AR(p)
- Autoregressive Moving Average Model ARMA(p,q)
- Autoregressive Integrated Moving Average Model ARIMA(p,d,q)

Most of the information contained in these slides is borrowed from [3].

White Noise Process - 1

Given an i.i.d. stochastic process $\{\epsilon_t\}$, where $\epsilon_t \sim \mathcal{N}(0, \sigma^2)$, it is evident that the process has constant mean and variance.

$$\mathbb{E}[\epsilon_t] = 0 \quad \text{Var}[\epsilon_t] = \sigma^2$$

Calculating the autocovariance is easy as well, since the RVs are i.i.d.

$$C_{XX}(h) = \begin{cases} \sigma^2 & \text{if } h = 0 \\ 0 & \text{if } h \neq 0 \end{cases}$$

Which leads to autocorrelation:

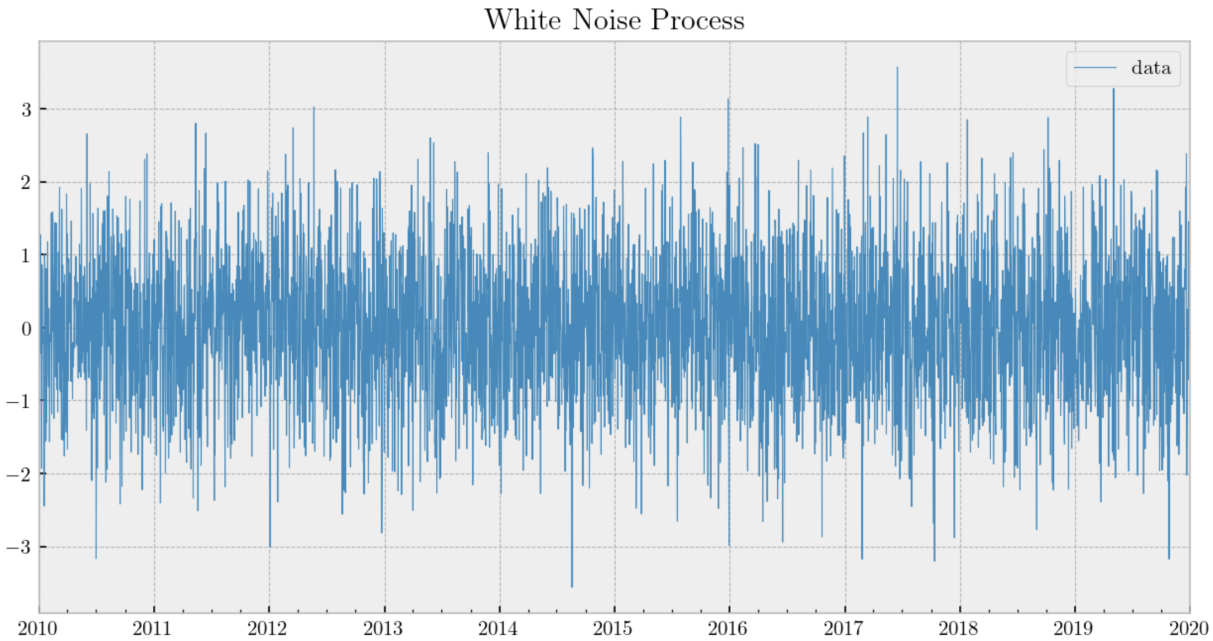
$$\rho_{XX}(h) = \begin{cases} 1 & \text{if } h = 0 \\ 0 & \text{if } h \neq 0 \end{cases}$$

where h represents the lag. The white noise process is therefore stationary.

White Noise Process - 2

```python
import numpy as np
import pandas as pd

sigma2 = 1.0
idxs = pd.date_range(start="2010-01-01", end="2020-01-01")
df = pd.DataFrame(data = np.random.randn(len(idxs))*np.sqrt(sigma2), columns=["data"], index=idxs)
df.plot(title="White Noise Process", lw=0.5);
```



Random Walk Model - 1

The basic equation for the Random Walk model is the following:

$$X_t = X_{t-1} + \epsilon_t$$

where $\epsilon_t \sim \mathcal{N}(\mu, \sigma^2)$. We usually set $X_0 = 0$ therefore, using recursive replacement:

$$X_t = \sum_{j=1}^{t} \epsilon_j$$

As a result, $\mathbb{E}[X_t] = t\mu$ and $\text{Var}[X_t] = t\sigma^2$. We see that both variance and mean depend on the time t, therefore they are not constant and the process is non-stationary.
If we use the differencing operator ∇:

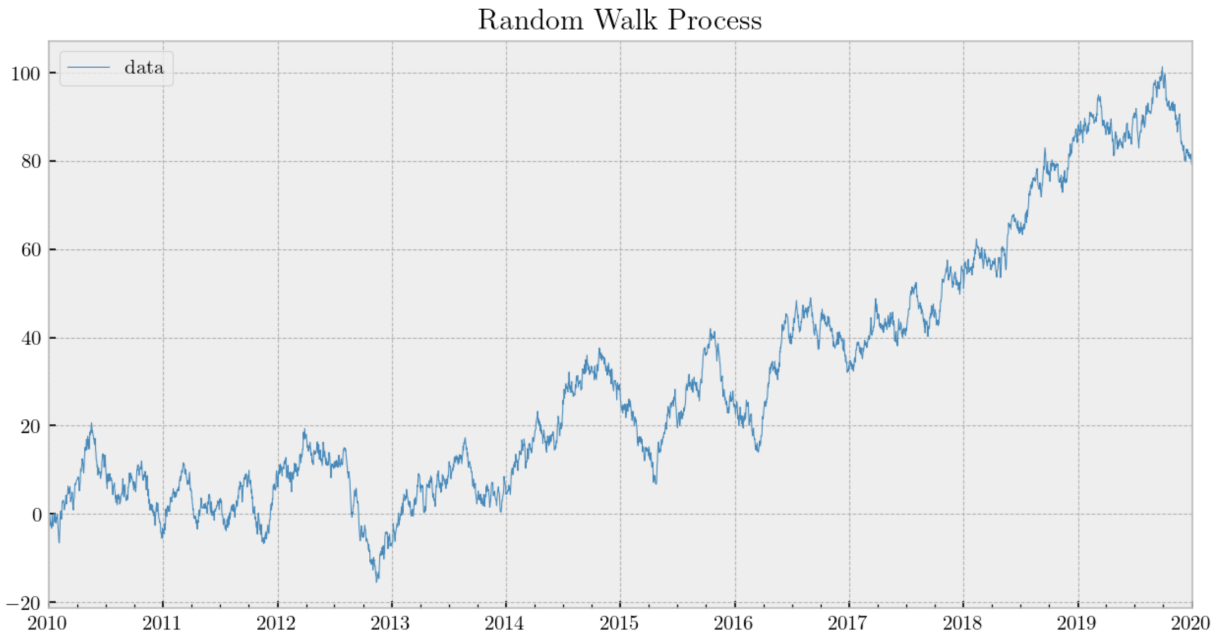
$$\nabla X_t = X_t - X_{t-1} = \epsilon_t$$

$\mathbb{E}[\nabla X_t] = \mu$ and $\text{Var}[\nabla X_t] = \sigma^2$, constant mean and variance. It can be shown that the autocovariance depend only on the lag, which implies that the differenced time series is stationary. Random Walk models well approximate the behavior of stock returns, although stock returns also exhibit volatility clustering (regions of similar variance tend to group together).

Random Walk Model - 2

```python
import numpy as np
import pandas as pd

sigma2 = 1.0
mu = 0.01
idxs = pd.date_range(start="2010-01-01", end="2020-01-01")
x = np.zeros((len(idxs)))
for i in range(1, len(x)):
    x[i] = x[i-1] + np.random.normal(loc=mu, scale=np.sqrt(sigma2))
df = pd.DataFrame(data = x, columns=["data"], index=idxs)
df.plot(title="Random Walk Process", lw=0.5);
```



Moving Average Model - 1

A moving average model of order q is indicated with the abbreviation MA(q). If we assume $\epsilon_t \sim \mathcal{N}(0, \sigma^2)$, the model can be written in the following form:

$$X_t = \beta_0 \epsilon_t + \beta_1 \epsilon_{t-1} + \cdots + \beta_q \epsilon_{t-q}$$

Usually, the coefficients are scaled so that $\beta_0 = 1$. The linearity of expectations implies that $\mathbb{E}[X_t] = 0$ and $\text{Var}[X_t] = \sigma^2 \sum_{j=0}^{q} \beta_j^2$
The autocovariance can be written as:

$$C_{XX}(X_t, X_{t+h}) = \begin{cases} 0 & h > q \\ \sigma^2 \sum_{j=0}^{q-h} \beta_j \beta_{j+h} & 0 \leq h \leq q \\ \sigma^2 \sum_{j=0}^{q+h} \beta_j \beta_{j-h} & -q \leq h < 0 \\ 0 & h < -q \end{cases}$$

For lag values $h > q$, C_{XX} is null. Thus, we can look at the autocorrelation function and find the highest value of the lag h with significant autocorrelation to determine q.
In backshift notation, we can write:

$$X_t = \epsilon_t + \beta_1 \epsilon_{t-1} + \cdots + \beta_q \epsilon_{t-q} = (1 + \beta_1 B + \beta_2 B^2 + \cdots + \beta_q B^q)\epsilon_t = \theta(B)\epsilon_t$$

where the coefficients have been normalized so that $\beta_0 = 1$.

Moving Average Model - 2

No restrictions on the coefficients are required for stationarity, but some are needed for *invertibility*, which allows to rewrite the MA model as an autoregressive model.

Theorem 1

the MA(q) process is invertible if and only if the roots of the polynomial $\theta(B)$ are located outside the unit circle in the complex plane $\mathbb{C}$.

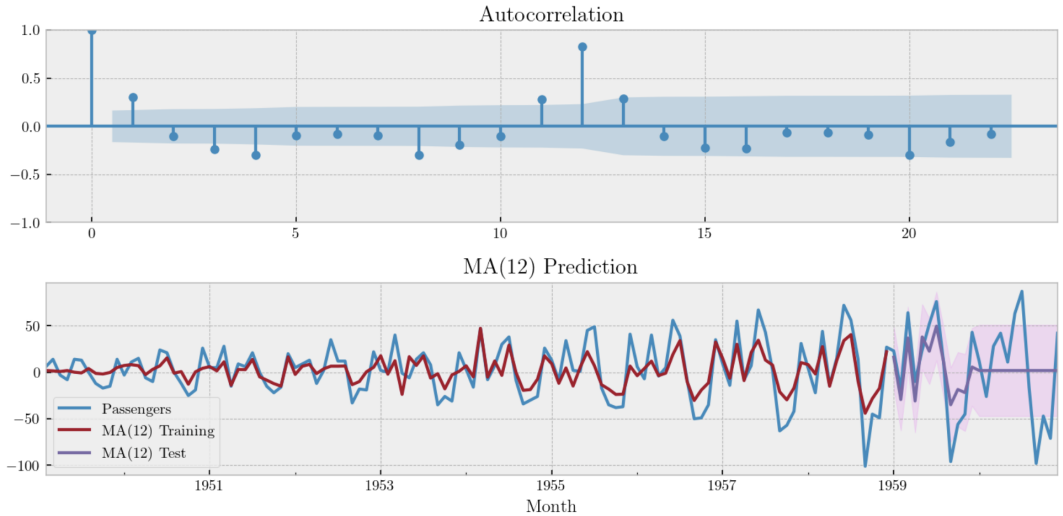
Theorem 1 can be used to assess whether a given MA process is invertible.
Mind that the errors ϵ are not observed. This means that, even though simulating from an MA model is easy (it involves just sampling the errors from a distribution and then obtain the estimates of X at time step t) fitting the model is done via Iterative Reweighted Least Squares (IRWLS, which we have seen in the context of Generalized Linear Models training in the Supervised Learning module).

On the other hand, in Python one would instead resort to the class `statsmodels.tsa.arima.model.ARIMA` with the setting `order=(0,0,q)`.

Moving Average Model - 3

```python
import pandas as pd
from statsmodels.tsa.holtwinters import SimpleExpSmoothing
from statsmodels.graphics.tsaplots import plot_acf
from statsmodels.tsa.arima.model import ARIMA

df = pd.read_csv("airline_passengers.csv", parse_dates=True, index_col=0).diff().dropna()
df.index.freq = "MS" # set frequency
fig, ax = plt.subplots(nrows=2, ncols=1, figsize=(10,5))
# plot autocorrelation
plot_acf(df["Passengers"], ax=ax[0]);
# define training and test set
Ntest = 24 # predict last two years
training, test = df.iloc[:-Ntest,:], df.iloc[-Ntest:,:]
training_idx = df.index <= training.index[-1]
test_idx = ~training_idx
# fit model
modelString = "MA(12)"
model = ARIMA(training['Passengers'], order=(0,0,12))
result = model.fit(method_kwargs={"warn_convergence": False});
df.loc[training_idx, modelString + ' Training'] = result.predict(start=training.index[0],
                                                                 end=training.index[-1])
# plot
prediction_result = result.get_forecast(Ntest)
confInt = prediction_result.conf_int(alpha=0.05).values
df.loc[test_idx, modelString + ' Test'] = prediction_result.predicted_mean
df.plot(ax=ax[1], title=modelString+" Prediction")
plt.fill_between(test.index, confInt[:,0], confInt[:,1], color="magenta", alpha=0.1)
plt.tight_layout()
```



The autocorrelation function exhibits a peak at 12, therefore we will fit a MA(12) model on the differenced data.
Differencing the time series is essential since, without de-trending, the time series would not be stationary and the MA prediction would be extremely unreliable.

Also note that the 95% confidence interval contains the true time series value at most time steps.

Autoregressive Model - 1

An autoregressive model is a model for time series where the covariates of the regression with output X_t are the same values of X_t at previous time steps.

$$X_t = \alpha_1 X_{t-1} + \alpha_2 X_{t-2} + \cdots + \alpha_p X_{t-p} + \epsilon_t$$

where $\epsilon_t \sim \mathcal{N}(0, \sigma^2)$. This model can again be expressed via the backshift operator B.

$$(1 - \alpha_1 B - \alpha_2 B^2 + \cdots - \alpha_p B^p) X_t = \epsilon_t \implies \phi(B) X_t = \epsilon_t$$

Theorem 2

An AR(p) is stationary if and only if the roots of $\phi(B)$ fall outside the unit circle in the complex plane $\mathbb{C}$.

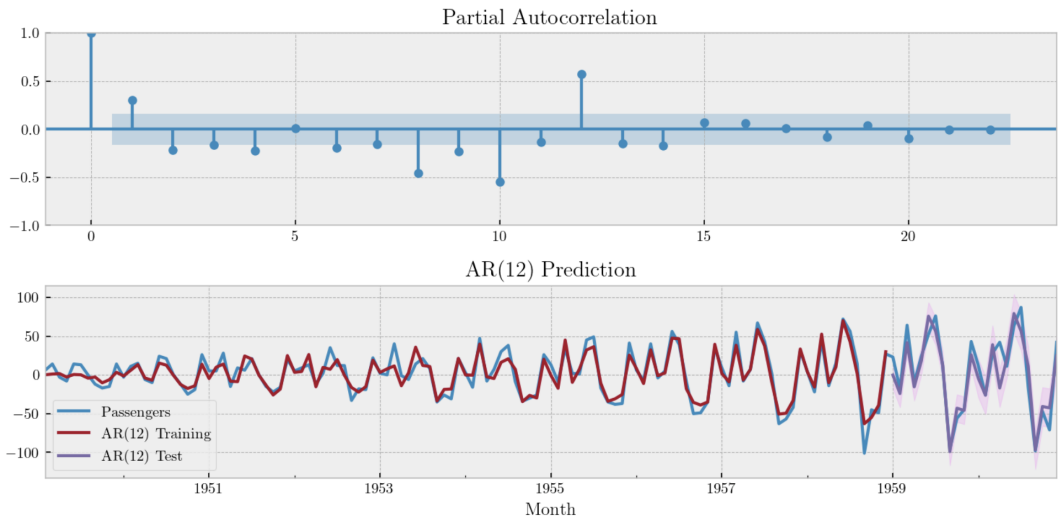
The order of the AR(p) model can be found by looking at the partial correlation plot. The partial correlation plot shows the residual correlation explained by each previous time step when used as predictor.

The AR model can be easily fit by generating a data set with a sliding window and training an Ordinary Least Squares model on it. In practice, this is best done by `statsmodels.tsa.arima.model.ARIMA` with `order=(1,0,0)`.

Autoregressive Model - 2

```python
import pandas as pd
from statsmodels.tsa.holtwinters import SimpleExpSmoothing
from statsmodels.graphics.tsaplots import plot_pacf
from statsmodels.tsa.arima.model import ARIMA

df = pd.read_csv("airline_passengers.csv", parse_dates=True, index_col=0).diff().dropna()
df.index.freq = "MS" # set frequency
fig, ax = plt.subplots(nrows=2, ncols=1, figsize=(10,5))
# plot autocorrelation
plot_pacf(df["Passengers"], ax=ax[0], method='ywm');
# define training and test set
Ntest = 24 # predict last two years
training, test = df.iloc[:-Ntest,:], df.iloc[-Ntest:,:]
training_idx = df.index <= training.index[-1]
test_idx = ~training_idx
# fit model
modelString = "AR(12)"
model = ARIMA(training['Passengers'], order=(12,0,0), )
result = model.fit(method_kwargs={"warn_convergence": False}, start_params=[0]*13+[1]); # to turn off warning
df.loc[training_idx, modelString + ' Training'] = result.predict(start=training.index[0],
                                                                 end=training.index[-1])
# plot
prediction_result = result.get_forecast(Ntest)
confInt = prediction_result.conf_int(alpha=0.05).values
df.loc[test_idx, modelString + ' Test'] = prediction_result.predicted_mean
df.plot(ax=ax[1], title=modelString+" Prediction")
plt.fill_between(test.index, confInt[:,0], confInt[:,1], color="magenta", alpha=0.1)
plt.tight_layout()
```



The partial autocorrelation function exhibits a peak at 12, therefore we will fit a AR(12) model on the differenced data. Note that this model provides a great improvement over the MA(12) model. AR(12) seems to be tracing the time series very well. The 95% confidence intervals are also much tighter than in the MA(12) case.

Autoregressive (Integrated) Moving Average Model - 1

The ARMA(p,q) model is a model combining both AR(p) and MA(q).

$$X_t = \alpha_1 X_{t-1} + \alpha_2 X_{t-2} + \cdots + \alpha_p X_{t-p} + \epsilon_t + \beta_1 \epsilon_{t-1} + \cdots + \beta_q \epsilon_{t-q}$$

This model can easily rewritten using again the backshift operator B:

$$\phi(B) X_t = \theta(B) \epsilon_t$$

Theorem 3

An ARMA(p,q) model is stationary and invertible if and only if the roots of the polynomials $\phi(B)$ and $\theta(B)$ fall outside the unit circle in the complex plane $\mathbb{C}$.

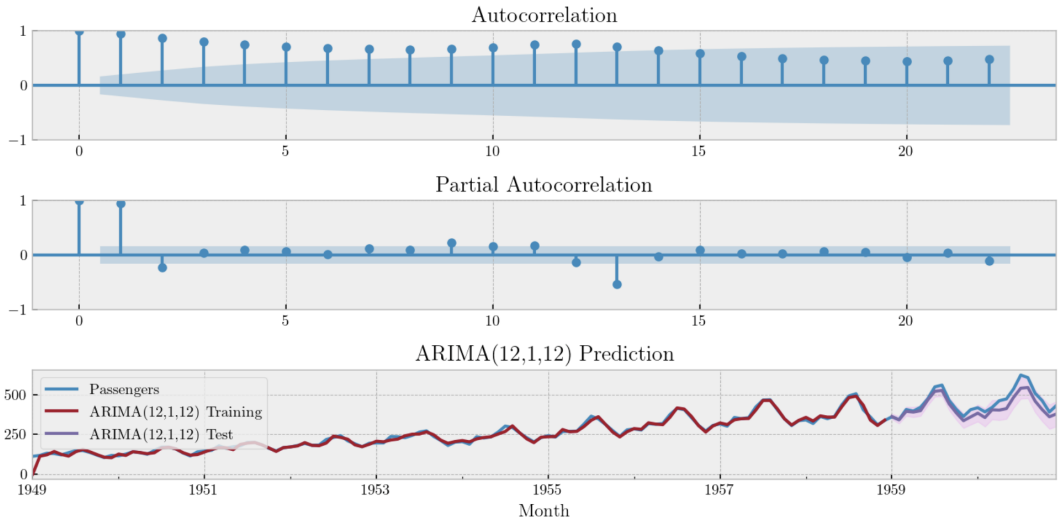
The ARIMA(p,d,q) model fits an ARMA(p,q) model on a time series which has been differenced d times.

Given ∇X_t, an ARMA(p,q) on ∇X_t is equivalent to an ARIMA(p,1,q) on X_t.

Autoregressive (Integrated) Moving Average Model - 2

```python
import pandas as pd
from statsmodels.tsa.holtwinters import SimpleExpSmoothing
from statsmodels.graphics.tsaplots import plot_pacf
from statsmodels.tsa.arima.model import ARIMA

df = pd.read_csv("airline_passengers.csv", parse_dates=True, index_col=0)
df.index.freq = "MS" # set frequency
# plot autocorrelation
fig, ax = plt.subplots(nrows=3, ncols=1, figsize=(10,5))
plot_acf(df["Passengers"], ax=ax[0])
plot_pacf(df["Passengers"], ax=ax[1], method='ywm');
# define training and test set
Ntest = 24 # predict last two years
training, test = df.iloc[:-Ntest,:], df.iloc[-Ntest:,:]
training_idx = df.index <= training.index[-1]
test_idx = ~training_idx
# fit model
modelString = "ARIMA(12,1,12)"
model = ARIMA(training['Passengers'], order=(12,1,12), )
result = model.fit(method_kwargs={"warn_convergence": False}, start_params=[0]*24+[1]); # to turn off warning
df.loc[training_idx, modelString + ' Training'] = result.predict(start=training.index[0],
                                                                 end=training.index[-1])
# plot
prediction_result = result.get_forecast(Ntest)
confInt = prediction_result.conf_int(alpha=0.05).values
df.loc[test_idx, modelString + ' Test'] = prediction_result.predicted_mean
df.plot(ax=ax[2], title=modelString+" Prediction")
plt.fill_between(test.index, confInt[:,0], confInt[:,1], color="magenta", alpha=0.1)
plt.tight_layout()
```

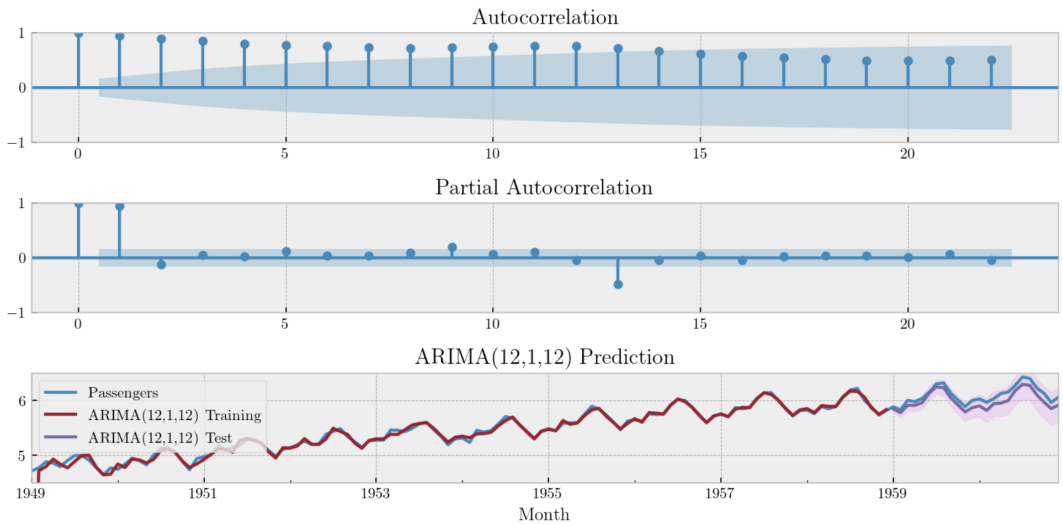


The autocorrelation and the partial autocorrelation exhibit significant peaks up to lag 13. However, since the integrated part accounts for 1 time step, we may set p and q to 12. The plot shows a decent agreement between the ground truth and the prediction. However, note that the time series constantly underestimate the true value, which means that a transformation might be needed before fitting the ARIMA model.

Autoregressive (Integrated) Moving Average Model - 3

```python
import pandas as pd
import numpy as np
from statsmodels.tsa.holtwinters import SimpleExpSmoothing
from statsmodels.graphics.tsaplots import plot_pacf, plot_acf
from statsmodels.tsa.arima.model import ARIMA

df = pd.read_csv("airline_passengers.csv", parse_dates=True, index_col=0)
df["Passengers"] = np.log(df["Passengers"])
df.index.freq = "MS" # set frequency
# plot autocorrelation
fig, ax = plt.subplots(nrows=3, ncols=1, figsize=(10,5))
plot_acf(df["Passengers"], ax=ax[0])
plot_pacf(df["Passengers"], ax=ax[1], method='ywm');
# define training and test set
Ntest = 24 # predict last two years
training, test = df.iloc[:-Ntest,:], df.iloc[-Ntest:,:]
training_idx = df.index <= training.index[-1]
test_idx = ~training_idx
# fit model
modelString = "ARIMA(12,1,12)"
model = ARIMA(training['Passengers'], order=(12,1,12), )
result = model.fit(method_kwargs={"warn_convergence": False}, start_params=[0]*24+[1]); # to turn off warning
df.loc[training_idx, modelString + ' Training'] = result.predict(start=training.index[0],
                                                                 end=training.index[-1])
# plot
prediction_result = result.get_forecast(Ntest)
confInt = prediction_result.conf_int(alpha=0.05).values
df.loc[test_idx, modelString + ' Test'] = prediction_result.predicted_mean
df.plot(ax=ax[2], title=modelString+" Prediction")
plt.fill_between(test.index, confInt[:,0], confInt[:,1], color="magenta", alpha=0.1)
ax[2].set_ylim([4.5,6.5])
plt.tight_layout()
```



The autocorrelation and the partial autocorrelation exhibit significant peaks up to lag 13. However, since the integrated part accounts for 1 time step, we may set p and q to 12. The application of the log function to the time series makes it easier for the ARIMA model to fit, however, other transformations, if possible, should be considered (for example, the more general Box-Cox transformation).

Machine Learning Models - 1

In the previous sections about the AR(p) model, we wrote the model's equation as:

$$X_t = \alpha_1 X_{t-1} + \alpha_2 X_{t-2} + \cdots + \alpha_p X_{t-p} + \epsilon_t$$

Now, this model makes the assumption that the future values of the time series are *linearly* related to the previous values. This requirement is rather strict and we can relax it by writing the more general equation:

$$X_t = f(X_{t-1}, X_{t-2}, \ldots, X_{t-p}) + \epsilon_t$$

This model is much more powerful, since f can be any predictive model. A few examples of such model can therefore be:

- **Non-DL Models**:
  - Linear Regression Model (this assumption leads to the already studied AR(p) model)
  - Support Vector Regression Model
  - K-Nearest Neighbor Regression Model
  - Random Forest Model
  - Bayesian Additive Regression Tree Model
  - Tree Bosting Models (XGBoost, LightGBM, CatBoost)

Machine Learning Models - 2

- **DL Models**:
  - Convolutional Neural Network (CNN)
  - Long-short Term Memory Model (LSTM)
  - Transformer Model

There are no one-size-fits-all solutions. The quality of a fit depends on the time series at hand. As pointed out in [4], deep learning models may not be the most suitable solution for time series (whereas they exhibit state-of-the-art performance in Natural Language Processing, for example). A more recent study [5] hypothesizes that DL models may be more prone to overfitting than more traditional methods and just as sensitive to the signal pre-processing.

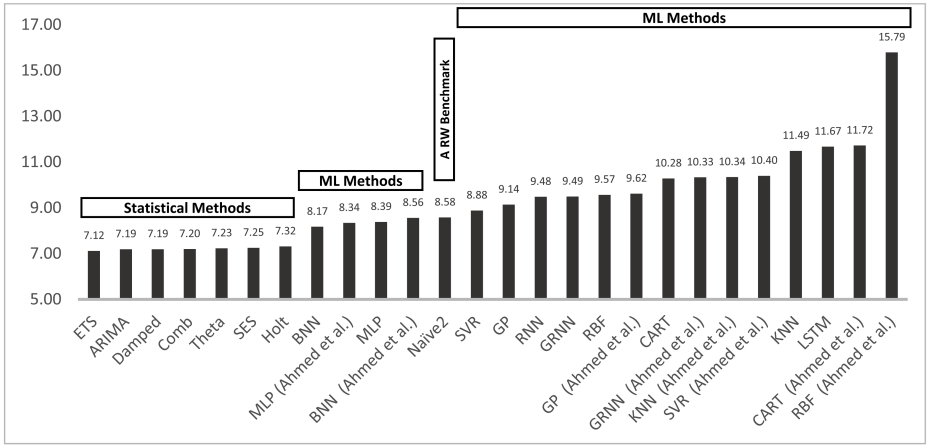


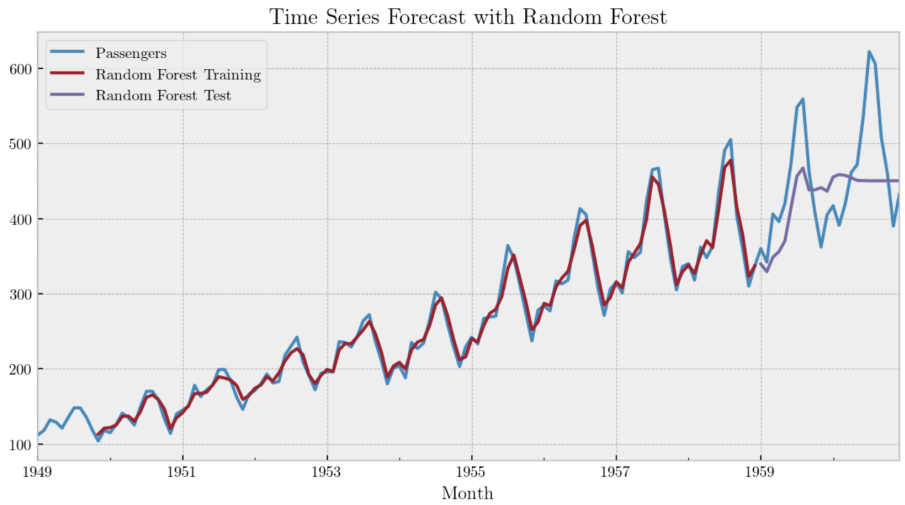
Fig 2. Forecasting performance (sMAPE) of the ML and statistical methods included in the study. The results are reported for one-step-ahead forecasts having applied the most appropriate preprocessing alternative.

An evident issue of Deep Learning models lies in the lack of interpretability. Although many methods for XAI have been proposed in literature (SHAP values, LIME etc.) convincing the stakeholders in high-risk environments (for example, health care) of the forecast reliability is problematic.

Machine Learning Models - 3

```python
import pandas as pd
from sklearn.ensemble import RandomForestRegressor

df = pd.read_csv("airline_passengers.csv", parse_dates=True, index_col=0)
df.index.freq = "MS" # set frequency
# define training and test set
Ntest = 24 # predict last two years
T = 10 # number of time steps used for prediction
# create training and test set
X_train = []
y_train = []
for i in range(df.shape[0]-Ntest-T):
    X_train.append(df.values[i:i+T,0])
    y_train.append(df.values[i+T,0])
X_train = np.array(X_train, dtype=float)
y_train = np.array(y_train, dtype=float)
y_test = []
for i in range(df.shape[0]-Ntest-T,df.shape[0]-T):
    y_test.append(df.values[i+T,0])
y_test = np.array(y_test, dtype=float)
# generate model
model = RandomForestRegressor() # leave classifier as default
model.fit(X_train, y_train)
# get in-sample predictions
y_fitted = model.predict(X_train)
# get out-of-sample predictions
X_start = np.roll(X_train[-1], shift=-1)
X_start[-1] = y_fitted[-1]
y_prediction_test = []
for i in range(Ntest):
    y_prediction_test.append(model.predict(X_start[None,:]).item())
    X_start = np.roll(X_start, shift=-1)
    X_start[-1] = y_prediction_test[-1]
y_prediction_test = np.array(y_prediction_test)
# set the values at the right place in the data frame
train_idx = (np.arange(df.shape[0]) >= T) & (np.arange(df.shape[0]) < df.shape[0]-Ntest)
test_idx = (np.arange(df.shape[0]) >= df.shape[0]-Ntest)
df.loc[train_idx, "Random Forest Training"] = y_fitted
df.loc[test_idx, "Random Forest Test"] = y_prediction_test
df.plot(title="Time Series Forecast with Random Forest");
```



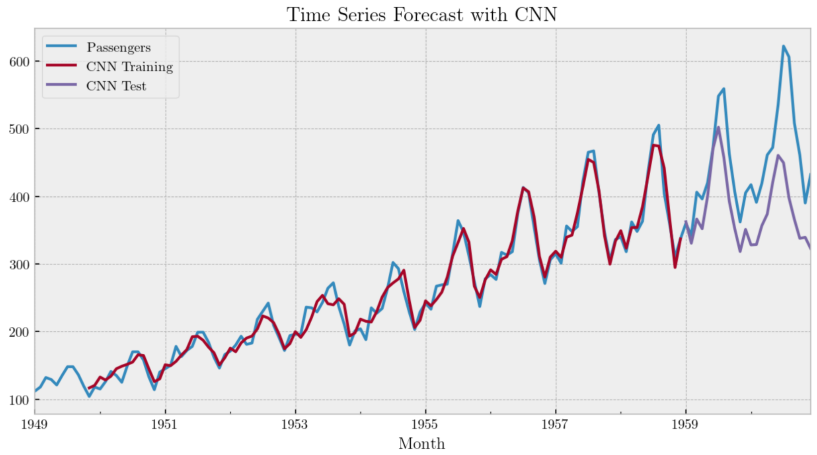
The Random Forest Regressor performs very well on the training set. However, on the test set, the errors propagate and the model is unable to capture both trend and seasonality. There is a large amount of overfit, which could be mitigated, in part, by fine-tuning the model parameters (which we didn't do in this case).

Also note that we could easily replace the model with another (for example, a Support Vector Regressor) by simply reassigning the `model` variable. This is the inherent benefit of using `scikit-learn`'s unified API.

Machine Learning Models - 4

```python
import pandas as pd
import tensorflow as tf
from tensorflow.keras.models import Model
from tensorflow.keras.layers import Input, Convolution1D, MaxPooling1D, Dense, GlobalAvgPool1D
from sklearn.preprocessing import StandardScaler

df = pd.read_csv("airline_passengers.csv", parse_dates=True, index_col=0)
df.index.freq = "MS" # set frequency
# define training and test set
Ntest = 24 # predict last two years
T = 10 # number of time steps used for prediction
# create training and test set
X_train = []
y_train = []
for i in range(df.shape[0]-Ntest-T):
    X_train.append(df.values[i:i+T,0])
    y_train.append(df.values[i+T,0])
X_train = np.array(X_train, dtype=float)
scaler = StandardScaler()
scaler.fit(df["Passengers"].values[:,None])
for j in range(X_train.shape[1]):
    X_train[:,j] = scaler.transform(X_train[:,[j]])[:,0]
y_train = np.array(y_train, dtype=float)
y_train = scaler.transform(y_train[:,None])[:,0]
y_test = []
for i in range(df.shape[0]-Ntest-T,df.shape[0]-T):
    y_test.append(df.values[i+T,0])
y_test = np.array(y_test, dtype=float)
# create model
inputs = Input(shape=(T,1))
x = Convolution1D(16, 5, activation="relu")(inputs)
x = MaxPooling1D()(x)
x = Convolution1D(32, 3, activation="relu")(x)
x = GlobalAvgPool1D()(x)
outputs = Dense(1)(x)
model = Model(inputs=inputs, outputs=outputs)
model.compile(loss="mse", optimizer="adam")
history = model.fit(X_train, y_train, epochs=100, verbose=False)
# predict in-sample
y_fitted = model.predict(X_train)
# predict out-of-sample
X_start = np.roll(X_train[-1], shift=-1)
X_start[-1] = y_fitted[-1]
y_prediction_test = []
for i in range(Ntest):
    y_prediction_test.append(model.predict(X_start[None,:]).item())
    X_start = np.roll(X_start, shift=-1)
    X_start[-1] = y_prediction_test[-1]
y_prediction_test = np.array(y_prediction_test)
train_idx = (np.arange(df.shape[0]) >= T) & (np.arange(df.shape[0]) < df.shape[0]-Ntest)
test_idx = (np.arange(df.shape[0]) >= df.shape[0]-Ntest)
df.loc[train_idx, "CNN Training"] = scaler.inverse_transform(y_fitted)
df.loc[test_idx, "CNN Test"] = scaler.inverse_transform(y_prediction_test[:,None])
df.plot(title="Time Series Forecast with CNN");
```



The forecast using 1D CNN tends to underestimate the future values for the time series. Both the normalization and the hyperparameters of the Neural Network play a key role in the determination of the fit quality, so more time should be spent in the tuning phase.

In addition, we might consider fitting the differenced data series instead of the raw one.

References


P. J. Brockwell and R. A. Davis, *Time series theory and methods*, 2nd ed. New York: Springer, 2006.


R. J. Hyndman and G. Athanasopoulos, *Forecasting : principles and practice*. Melbourne: OTexts, 2021.


C. Chatfield, *The analysis of time series : an introduction*, 6th ed. Boca Raton, FL ;: Chapman and Hall/CRC, 2004.


S. Makridakis, E. Spiliotis, and V. Assimakopoulos, "Statistical and machine learning forecasting methods: Concerns and ways forward," *PloS one; PLoS One*, vol. 13, no. 3, p. e0194889, 2018.


B. Lim and S. Zohren, "Time-series forecasting with deep learning: a survey," *Philosophical transactions of the Royal Society of London.Series A: Mathematical, physical, and engineering sciences; Philos Trans A Math Phys Eng Sci*, vol. 379, no. 2194, p. 20200209, 2021.