Machine Learning I

Chapter 12 - Statistical Models For Sequential Data

Prof. Dr. Sandra Eisenreich

January 18 2023

Hochschule Landshut

Time series are data which are sequences of one or more values at certain time intervals.
Examples for time series:

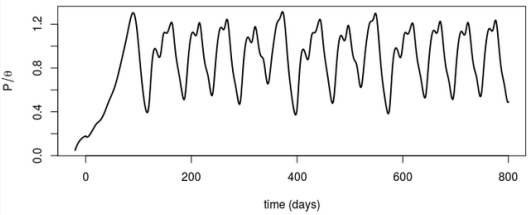
- the number of active users per hour on your website
- the daily temperature in your city
- medicine: blood pressure/weight/heart rate tracking
- economics: GDP, unemployment rates, the development of stock price over time
- epidemiology: disease/mortality rates, mosquito populations in malaria regions
- industry, the performance of production over time in a plant, or the health of production equipment, eg a robot, over time.

Difference to other ML methods: ?

The task and random processes

Tasks for time series/sequential models:

- forecasting: to predict future values. (often via Regression)
- imputation: to fill in missing values from the past. (often via Regression).



Quelle: Wikimedia Commons

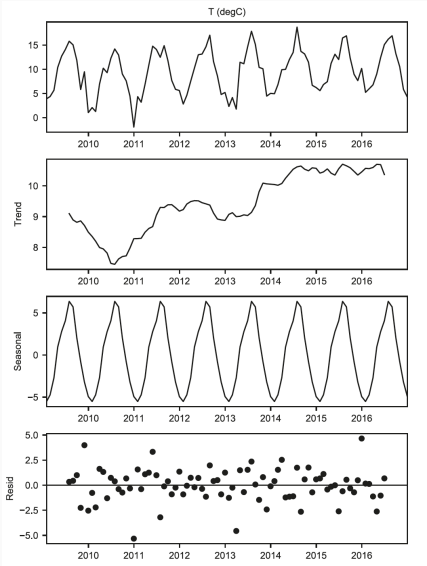
Time series are called

- univariate if the value at each time step is just one real value $\in \mathbb{R}$, and
- multivariate if there are multiple values at each time step (i.e. a vector $\mathbf{x} \in \mathbb{R}^m$).

The previous values of the time series $x_{t-1}, x_{t-2}, \ldots$ are called lags (e.g. you would call the previous value of the time series "lag 1", the value of 12 steps earlier "lag 12", etc...).

Trend and seasonality

- trend: the development of the average of your data over time (like global warming affecting temperature).
- seasonality: repeated patterns over time (like seasonal changes in weather and temperature)
- residuals: the errors of the model prediction.
- Data without seasonalty or a trend, whose behaviour, average and standard deviation are constant over time, are called stationary.



Quelle: Hirschle, Machine Learning für Zeitreihen

Trend and seasonality with Statsmodels

You can get trend and seasonality with:

```
from statsmodels.tsa.seasonal import seasonal_decompose
decomposition = seasonal_decompose(data)
decomposition.plot()
```

Lags and order

- A lag is a past time value a fixed numer of time steps in the past.
- The order of a lag is the number of time steps it is in the past, e.g. the order 1 lag at time t is simply the previous value x_{t-1}, the order 4 lag is x_{t-4}.

Lags with pandas:

You can add a column of the lag 1 values (i.e. the shifted values) of a time series in pandas using

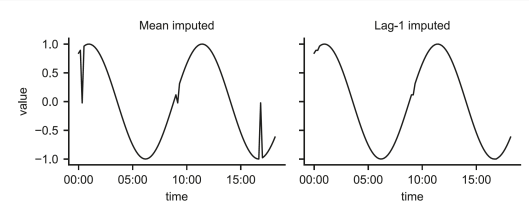
```
df['lag_1'] = df['value'].shift(periods=1)
```

If you change periods to a higher number d, it will add a column of the order d lags to the dataframe.

Preparing time series data

Question: How should you deal with missing data in time series? Is it a good idea to delete instances or fill in the mean or median?

Answer: ?



Instead:?

Quelle: Hirschle, ML für Zeitreihen

Rolling mean with pandas:

```
df['rm'] = df['value'].rolling(window=9,
                               min_periods=3,
                               center=True).mean()
```

Here, `min_periods` specifies how many valid values you want in a window to compute and impute the average (e.g. for null values). `Center=False` only considers past values. To use this for missing value imputation:

```
df['rm_imput'] = df['value'].fillna(
                            value=df['value'].rolling(
                            window=3,
                            center=True,
                            min_periods=2).mean())
```

Problem: The above only work if we have individual missing values. What if entire windows of data are missing (e.g. because a sensor stopped working for a while), but we know there is a seasonality (e.g. the temperature over a year)? Then we could just impute the value from 12 months earlier for each missing value.

```
df_lag12 = df['value'].fillna(
                                value=df['value'].shift(periods=12))
```

Train-Test-Split: Attention!

Statistical models: Overview

One could extract features like year/season/month/weekday/hour as individual feautures and use models we already know, e.g. linear regression, where every time step of the time series is considered independet from each other. However, it is better to view time series as values depending on each other and use other methods like the following:

We only look at univariate models here; multivariate versions exist. Statistical models to forecast univariate stationary time series:

- Autoregressive (AR) model
- Moving average (MA) model
- A combination: Autoregressive moving average (ARMA) model

Statistical model to forecast time series with a trend = a combination of all the above: autoregressive integrated moving average (ARIMA) model

Statistical models: Overview

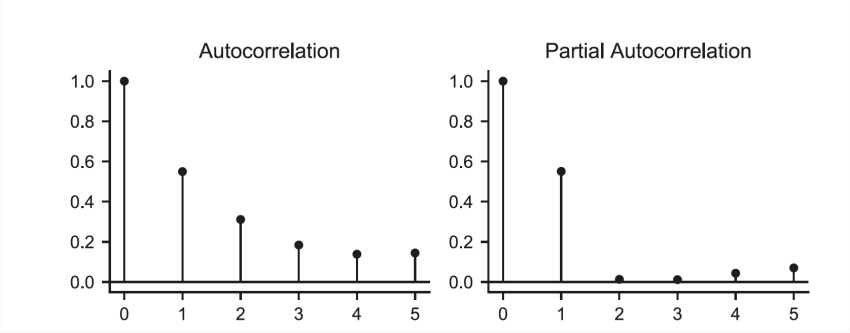
When to use which?

- AR is better than MA for time series where the effect of one lag is carried over to all later values.
- MA is better for time series where lags have shorter, restricted effects (shocks) to a single value.
- Better: use ARMA or ARIMA.

For general time series data, one can also use neural network methods (Recurrent Neural Networks, Transformers)... these are even more powerful, but you need a lot of data $\rightarrow$ see ML2.

The autocorrelation and partial autocorrelation function

The autocorrelation of a time series is the correlation matrix of the lags, i.e. of x_t with $x_{t+1}, \ldots$, i.e.: how much is a time series value correlated with the following value, the value after that, etc. . .



Quelle: Hirschle, ML für Zeitreihen

Since each value of a time series somehow depends on the previous values, the correlation of x_t with x_{t+1} leads to a correlation of x_{t+2} with x_t, etc. So a valid question would be: How much are x_t and x_{t+2} correlated independet of the value of x_{t+1}? The value which describes this (not just for t and $t+2$, but any time steps) is the partial autocorrelation.

The ACF and PACF function with statsmodels

Given a stationary time series $\{x_t\}$, for $u \geq 1$ the partial autocorrelation ϕ_u at lag u is defined as the correlation between x_t and x_{t-u} when the random variables $x_{t-1}, \ldots, x_{t-u+1}$ are held constant. Then the sequence $\{\phi_u : u \geq 1\}$ is known as the partial autocorrelation function (PACF).

You get plots of the autocorrelation function (ACF) and partial autocorrelation function (PACF) with statsmodels:

```
from statsmodels.graphics.tsaplots import plot_acf, plot_pacf


plot_acf(X, lags=5)
plot_pacf(X, lags=5)
plt.show()
```

Autoregressive model (AR)

AR is a regression model where the features of the training data are the past q lags. q is called the order of the model.

AR For a stationary univariate time series x_t, AR = autoregressive model AR(p) is the probabilistic model

$$y_t = \theta_0 + \theta_1 x_{t-1} + \theta_2 x_{t-2} + \ldots + \theta_p x_{t-p} + \epsilon_t$$

with parameters $(\theta_0, \ldots, \theta_p)$, for some random white noise ϵ_t (i.e. a sample from the Gaussian normal distribution $N(0, \sigma^2)$). Here, y_t is the prediction of the model for the value x_t.
Objective function: minimize negative log likelihood. (i.e. training = maximum likelihood)

Implementation of AR with statsmodels

One can use the class `AutoReg` from the python-module `statsmodels`. It works similar to Scikit-Learn classes: you train it by calling the fit-method, which returns the trained model. The `summary`-method returns the coefficients of the lags. With the trained model you can make predictions for $t + 1$.

```
from statsmodels.tsa.ar_model import AutoReg
model = AutoReg(X, lags=p, trend='n')
model_fit = model.fit()
```

With `.summary()` you get a summary of the training and results. Like in Scikit-Learn, you can get predictions with the `.predict()` method.

Moving-averages-model (MA) - Motivation

$AR(q)$: use p past values to predict the next one.
$MA(p)$: use past q prediction errors to predict the next value: for each time step t: ???

But: This is not a regression model, because...?

Moving-averages-model (MA)

For a stationary univariate time series x_t, Moving Average (MA(q)) is the probabilistic model with

$$y_t = \mu + \theta_1 \epsilon_{t-1} + \theta_2 \epsilon_{t-2} + \dots \theta_q \epsilon_{t-q}$$

where μ is the average of the time series values, θ_i are parameters that are learned by the model, and ϵ_{t-i} are the error-terms computed recursively: For each time step t

- compute the prediction $\widehat{\mathbf{y}}_t$ of the model using the previous error terms $\epsilon_{t-1}, \dots, \epsilon_{t-q}$
- compute $\epsilon_i = \widehat{x}_t - x_t$

Implementing MA with statsmodels

To train a model, use the `statsmodels`-class `ARMA` which can be used to train ARIMA as well as moving-average methods. Here, the `order` parameter taked a tupel (p=number of lags for autoregression, q=number of lags for moving-average-model). Setting `order=(0,n)` gives rise to a moving-average-model with n lags (error terms).

```
from statsmodels.tsa.arima_model import ARMA
model = ARMA(X, order=(0,1))
model_fit = model.fit(trend='nc')
model_fit.summary()
```

Autoregressive moving average (ARMA) model

One can combine the above two methods, AR(p) and MA(q), into a single model called ARMA(p,q), where p and q are the lags of both methods: First, we use the last p terms of the time series to predict the current value of the time series like we did in AR(p), and then "specify" the current value by predicting the current error term like we did in MA(q):

For a stationary univariate time series x_t, Autoregressive moving average (ARMA(p,q)) is the probabilistic model

$$y_t = \varphi_1 x_{t-1} + \ldots + \varphi_{t-p} x_p + \theta_1 \epsilon_{t-1} + \theta_2 \epsilon_{t-2} + \ldots \theta_q \epsilon_{t-q}$$

where φ_i, θ_i are the parameters of the model, the ϵ_i are the error terms computed recursively like in the MA model.
Objective function: minimize negative log likelihood. (i.e. training = maximum likelihood)

Implementing ARMA with statsmodels

```
from statsmodels.tsa.arima_model import ARMA
model = ARMA(X, order=(3,4))
model_fit = model.fit()
model_fit.summary()
```

Non-stationary data: Seasonality

If the data have seasonality of a certain length N, e.g. a yearly recurring pattern in mean and variance. So in time step t (e.g. the month of April), you expect the same or similar mean as one season ago at $t - N$.

There are two steps to get rid of seasonality: ??

$\Rightarrow$ Get a time series z_t which doesn't have seasonality. Make predictions for z_t and recover the predicitions for x_t by reversing the above calculations.

Non-stationary data: Trend

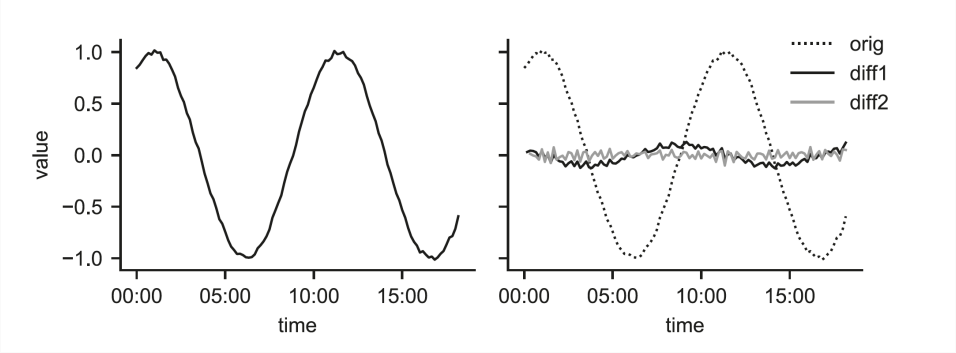
If the data have a linear trend, you can get rid of it by considering the differences between two consecutive time steps $z_t := x_t - x_{t-1}$ ($\sim$ like computing the first differential of a function).
Note: you can recover the predictions for x_t from predictions for z_t!
However, if the trend is quadratic, there will still be a linear trend in the "differenced" time series z_t. So, again, you could compute the differences of this time series $z'_t = z_t - z_{t-1}$ to get rid of the remaining trend...

The time series given by the differences between two consecutive time lags

$$Dx_t = x_t - x_{t-1}$$

is called First Difference ($\sim$ kind of like a first differential for a function). Computing this is called Differencing. The d-th Difference (or: Difference of order d) is the time series obtained by differencing a time series d times.

So: Keep differencing until you reach a stationary time series!



Quelle: Hirschle, ML für Zeitreihen

Differencing with pandas:

You could either compute differences "by hand" using the shift method with

```
df['diff1']=df['value'] − df['value'].shift(periods=1)
```

or use the pandas operator `.diff()`:

```
df['diff1']=df['value'].diff().
```

Logging

Sometimes it doesn't suffice to simply take differences: If you don't just have a trend but different sizes of "spikes" within a trend it might be beneficial to smooth things out. In this case it can help to consider the logarithm of the time series:

$$z_t = \log x_t$$

Since $\log x^n = n \cdot \log x$, This can make non-linear trends linear and also smooths out "spikes".

Autoregressive integrated moving average (ARIMA) model

A generalization of ARMA to non-stationary time series is given by ARIMA or SARIMA:

For a univariate time series x_t (possibly with a trend or seasonality), (Seasonal) Autoregressive integrated moving average ((S)ARIMA(p,d,q)$_m$) is the following model:

- (For SARIMA: First apply seasonal differencing (with m time steps) to remove seasonality.)
- Then apply differencing d times until you arrive at a stationary time series z_t.
- Use ARMA(p+d,q) to make predictions for z_t.

Implementing (S)ARIMA with statsmodels

```
from statsmodels.tsa.statespace.sarimax import SARIMAX

arima = SARIMAX(df_train['values'], order=(p,d,q))
result = arima.fit()
result.summary()
```

To predict future time series values, you can use

- `.forecast(1)` gives exactly the one next value.
- `.predict(start = 'start-datum', end = 'end-datum')` gives predictions for the time interval betwen start and end.

Performance Measures

Summary gives the following performance measures:

- Log-Likelihood (the logarithm of the likelihood to get the data given the statistical model, see maximum likelihood) $\widehat{L}$.
- AIC = Akaike Information criterion derived from log likelihood: evaluates models based on their complexity: less complex models are better as long as their performance is not much worse.

$$AIC = 2k - 2\widehat{L}$$

  where k is the number of parameters of the model.
- BIC = Bayesian information criterion:

$$BIC = \ln(m)k - 2\widehat{L}$$

  where m is the number of instances.
- HQIC = Hannan-Quinn information criterion :

$$HQIC = 2k \ln\ln(m) - 2\widehat{L}$$

Validation

If we want to evaluate a model, we have to predict all values for the test data in the future. To generate a series of predictions, we cannot use the fitted model to make all predicions, because each later prediction depends on the earlier ground truth values.

Idea: Get a new instance of the model with all data (training and test) but don't fit it, but instead transfer the learned parameters of the "correct" model to this one. Adavantage: it "knows" all the data, but the parameters were only trained on the training set. Use this to make predictions by specifying the start date of the test data.

```
model = SARIMAX(df['values'], order=(0,1,1))
result_new = model.filter(result.params)
y_pred_test = result_new.predict(start=datum)
```

Then you can plot the real test values vs. the predictions or compute MAE or MSE.

Box-Jenkins approach

The Box-Jenkins approach has three parts:

- model identification:
  - plot trend and seasonality, use differencing if necessary, make sure the time series obtained is stationary
  - plot ACF and PACF to decide which AR or MA component should be used. (choose p and q such that the highly correlated lags are part of the model!)
- parameter estimation: training with maximum likelihood
- verification: statistical model checking: plot the residuals (i.e. prediction - real value) over time to see if they are independent, or plot their mean and variance over time to see if they are constant. If so, the model is fine. If not, return to step 1.