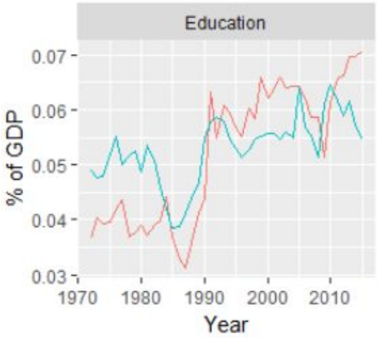
Time series analysis

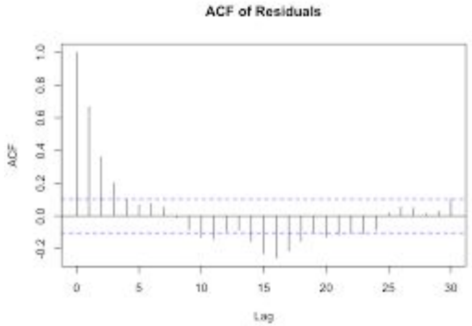
Angel Marchev, Jr.

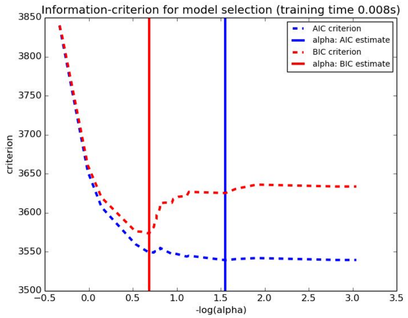
Kaloyan Haralampiev

Key topics

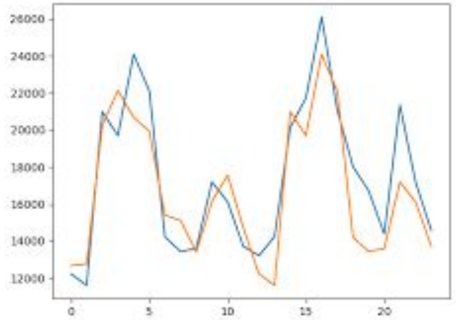
Comparability

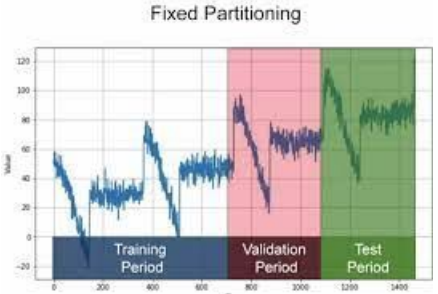




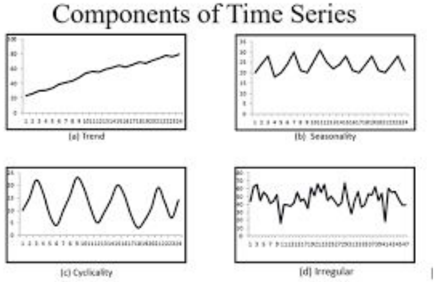


Stationarity

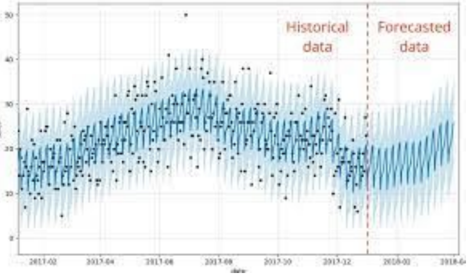




Components



Date	Value	$Value_{t-1}$	$Value_{t-2}$
1/1/2017	200	NA	NA
1/2/2017	220	200	NA
1/3/2017	215	220	200
1/4/2017	230	215	220
1/5/2017	235	230	215
1/6/2017	225	235	230
1/7/2017	220	225	235
1/8/2017	225	220	225
1/9/2017	240	225	220
1/10/2017	245	240	225



Comparability

Basic

- By territory
- By time
- By methodology

Additional

- By prices
- By coverage
- By measurement units

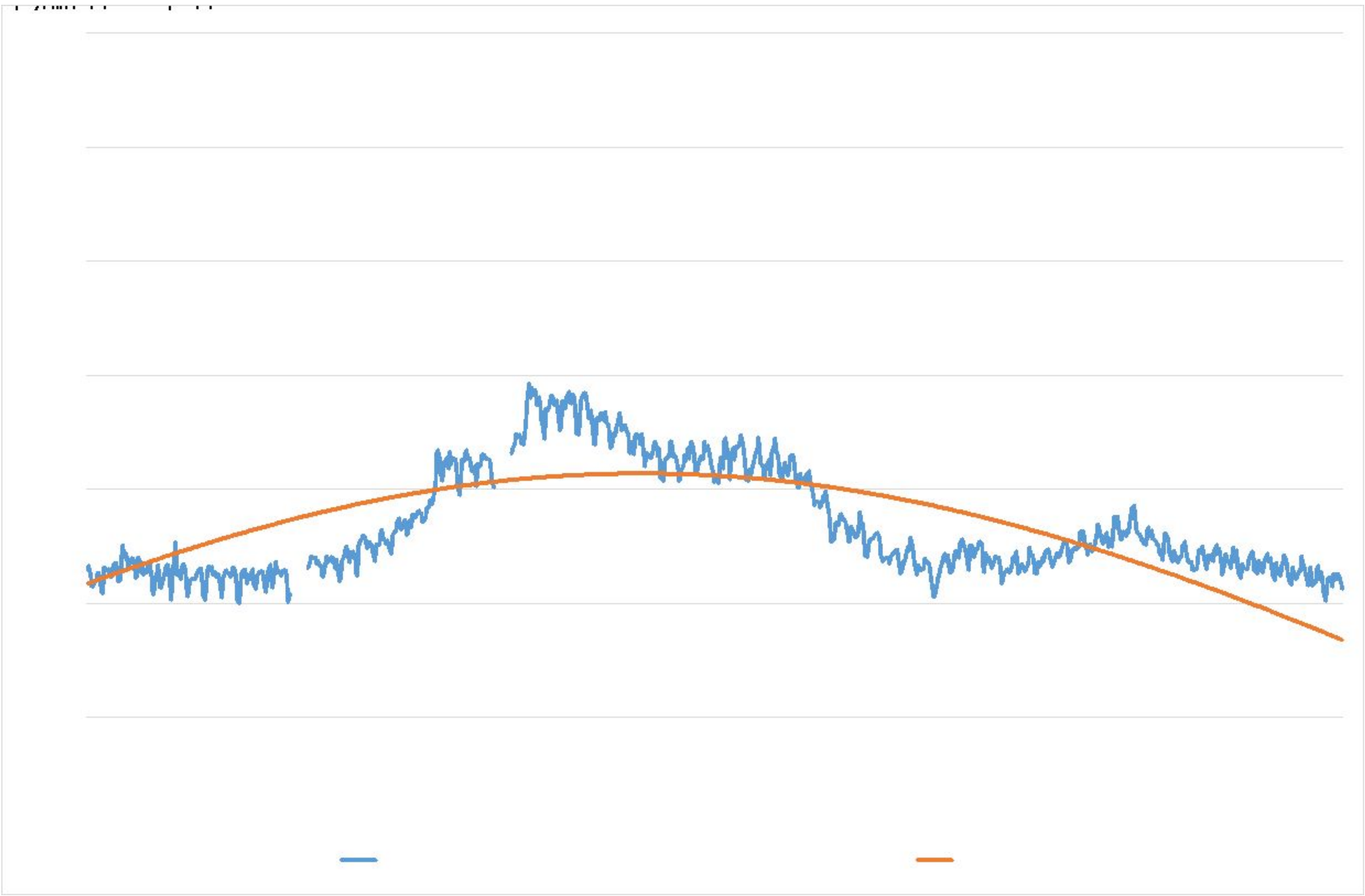
Stationarity

- Constant distribution
- i.e.
- Constant mean
- Constant variance
- etc…

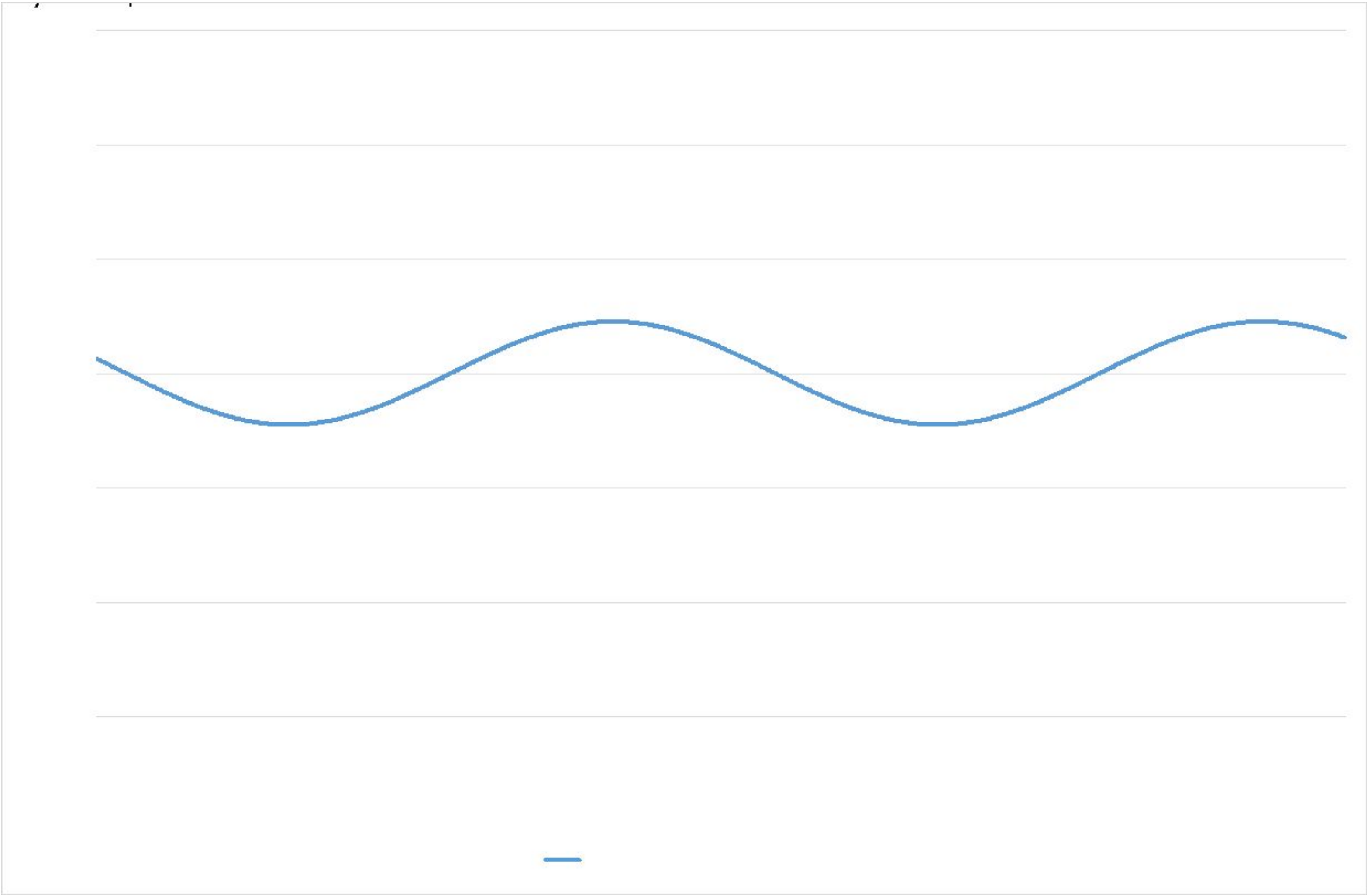
Components of dynamics

- Trend
- Cycle
- Seasonality
- Residuals

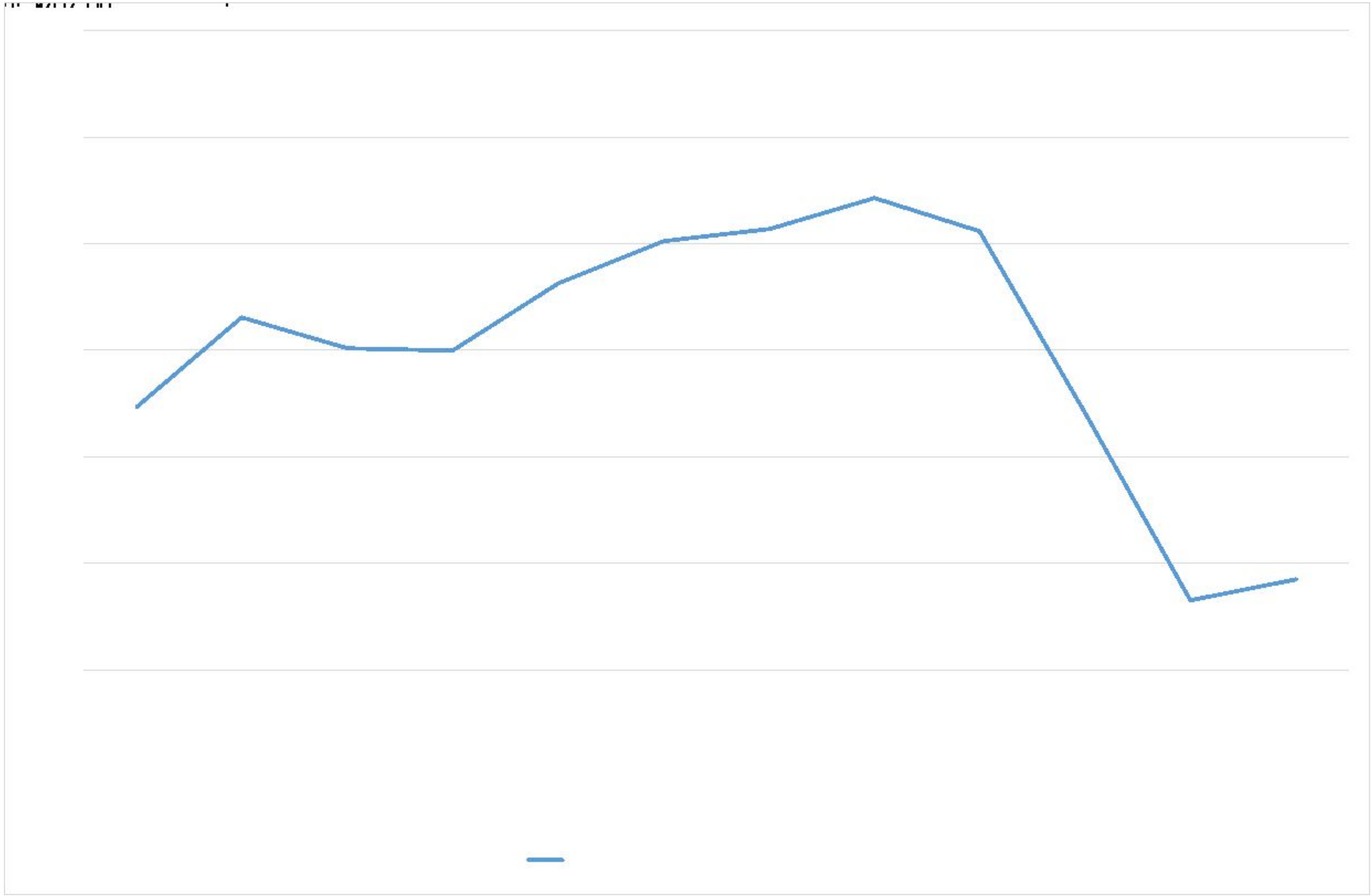
Trend



Cycle



Seasonality



Autocorrelation

- Autocorrelation function (ACF)

$$R_{y_t, y_{t-i}}$$

- Partial autocorrelation function (PACF)

$$R_{y_t, y_{t-i} | y_{t-j}}, j < i$$

Main models

- Regression

$$\hat{y}_t = f(t)$$

$$\hat{y}_t = f(t, x)$$

- Autoregression

$$\hat{y}_t = f(y_{t-i})$$

$$\hat{y}_t = f\left(y_{t-i}, x_{t-j}\right)$$

- Mixed models of regression and autoregression

$$\hat{y}_t = f(t, y_{t-i}, x_{t-j})$$

Feature engineering

Most often operations

- lags
- rolling window statistics
- datetime
- outliers low frequency filter
- Harmonic decomposition

Deriving lagged variables

Variables with a time delay compared to the others. Variable shifted in time.

- used in time series analysis to model the relationships between variables over time
- used to analyze the relationship between a variable and its past values

Methods

- **shift function in pandas**
- **Henkel matrix** - Strongly recommended universal method

In [200]:
```python
# create a lagged variable with a time shift of 1 day
df['lagged'] = df['value'].shift(1)

print(df)
```

```
   value  lagged
0      1     NaN
1      2     1.0
2      3     2.0
3      4     3.0
4      5     4.0
```

```python
import numpy as np

# Generate random time series data with 20 observations
data = np.random.rand(20)

# Define the maximum lag we want to include in our lagged features
max_lag = 5

# Create a Henkel matrix with lagged features
henkel_matrix = np.zeros((len(data), max_lag+1))

for i in range(max_lag+1):
    henkel_matrix[i:len(data), i] = data[0:len(data)-i]
henkel_matrix=henkel_matrix.round(3)
```

```python
# Print the Henkel matrix
print(henkel_matrix)
```

```
[[0.74  0.    0.    0.    0.    0.   ]
 [0.497 0.74  0.    0.    0.    0.   ]
 [0.586 0.497 0.74  0.    0.    0.   ]
 [0.061 0.586 0.497 0.74  0.    0.   ]
 [0.617 0.061 0.586 0.497 0.74  0.   ]
 [0.657 0.617 0.061 0.586 0.497 0.74 ]
 [0.859 0.657 0.617 0.061 0.586 0.497]
 [0.569 0.859 0.657 0.617 0.061 0.586]
 [0.905 0.569 0.859 0.657 0.617 0.061]
 [0.834 0.905 0.569 0.859 0.657 0.617]
 [0.568 0.834 0.905 0.569 0.859 0.657]
 [0.847 0.568 0.834 0.905 0.569 0.859]
 [0.026 0.847 0.568 0.834 0.905 0.569]
 [0.818 0.026 0.847 0.568 0.834 0.905]
 [0.961 0.818 0.026 0.847 0.568 0.834]
 [0.207 0.961 0.818 0.026 0.847 0.568]
 [0.57  0.207 0.961 0.818 0.026 0.847]
 [0.954 0.57  0.207 0.961 0.818 0.026]
 [0.237 0.954 0.57  0.207 0.961 0.818]
 [0.474 0.237 0.954 0.57  0.207 0.961]]
```

Rolling window statistics

Sample windows

- used in time series analysis to reduce the dimensionality of the data
- capture relevant patterns over a specific time interval

Method

- defining a fixed-length sample window
- extract a set of features from each window
- size of the sample window is an important hyperparameter
- it should be chosen based on the characteristics of the time series data and the specific prediction problem at hand.

```python
# Define the window size for the rolling statistics
window_size = 3
# Calculate rolling mean, standard deviation, and maximum
rolling_mean = series.rolling(window_size).mean()
rolling_std = series.rolling(window_size).std()
rolling_max = series.rolling(window_size).max()
```

	Original data	Rolling mean	Rolling standard deviation	Rolling maximum
0	0.076313	NaN	NaN	NaN
1	0.264040	NaN	NaN	NaN
2	0.675782	0.338712	0.306631	0.675782
3	0.068876	0.336233	0.309826	0.675782
4	0.806467	0.517042	0.393585	0.806467
5	0.705469	0.526937	0.399894	0.806467
6	0.756620	0.756185	0.050500	0.806467
7	0.018057	0.493382	0.412437	0.756620
8	0.089027	0.287901	0.407471	0.756620
9	0.579511	0.228865	0.305734	0.579511
10	0.527292	0.398610	0.269375	0.579511
11	0.970188	0.692330	0.242044	0.970188
12	0.485930	0.661137	0.268444	0.970188
13	0.957106	0.804408	0.275888	0.970188
14	0.128065	0.523700	0.415809	0.957106
15	0.372937	0.486036	0.425935	0.957106

Datetime index operations

Re-scaling

- manipulating the index of DataFrame to a new scale of dates

```
import pandas as pd

# create a DataFrame with a datetime index
date_rng = pd.date_range(start='1/1/2020', end='1/20/2020', freq='D')
df = pd.DataFrame(date_rng, columns=['date'])
df['data'] = np.random.randint(0,100,size=(len(date_rng)))

# change the frequency to weekly and take the mean of each group
df = df.set_index('date')
weekly_df = df.resample('W').mean()
weekly_df
```

	data
date	
2020-01-05	59.600000
2020-01-12	70.857143
2020-01-19	42.857143
2020-01-26	95.000000

Datetime index operations

Re-framing

- fill in the missing dates with some specified fill value.

```
# fill in the missing dates with NaN values
df = df.set_index('date')
df_new = df.asfreq('D')
df_new
```

	data
date	
2020-01-01	54.0
2020-01-02	67.0
2020-01-03	42.0
2020-01-04	NaN
2020-01-05	60.0
2020-01-06	22.0
2020-01-07	99.0

Datetime index operations

Extracting datetime features

- using the full datetime string to brake down into features

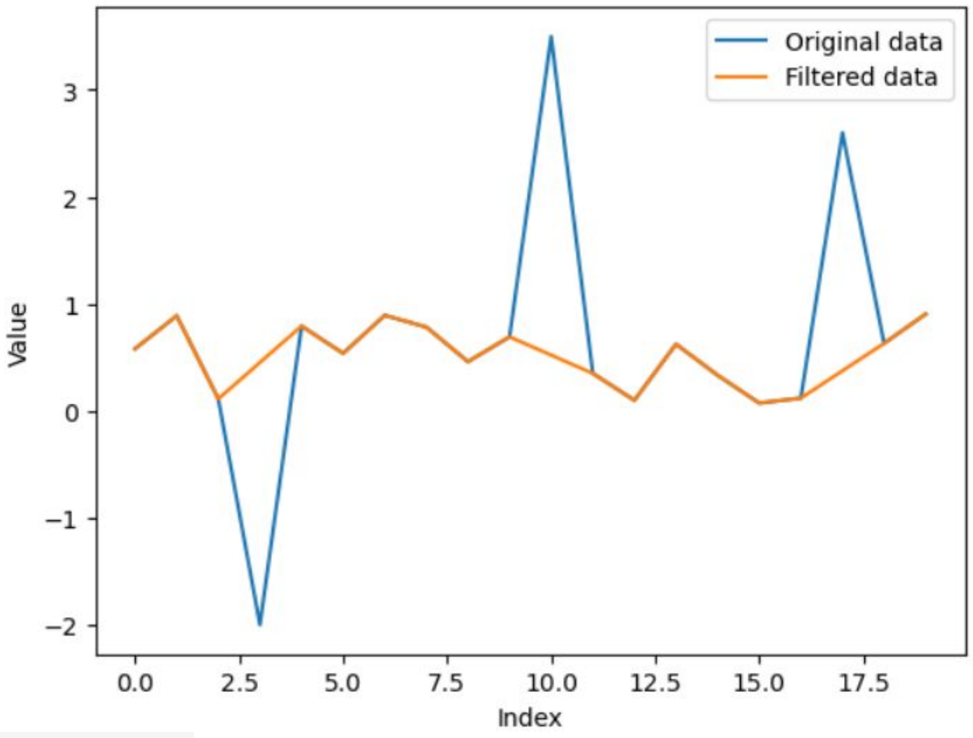
```
# Convert the data to a Pandas Series with DatetimeIndex
series = pd.Series(data, index=date_range)

# Extract calendar and time base features from the index
year = series.index.year
month = series.index.month
day = series.index.day
hour = series.index.hour
minute = series.index.minute
```

	Date	Data	Year	Month	Day	Hour	Minute
0	2022-01-01 00:00:00	0.114295	2022	1	1	0	0
1	2022-01-01 01:00:00	0.499400	2022	1	1	1	0
2	2022-01-01 02:00:00	0.316746	2022	1	1	2	0
3	2022-01-01 03:00:00	0.901192	2022	1	1	3	0
4	2022-01-01 04:00:00	0.531030	2022	1	1	4	0
5	2022-01-01 05:00:00	0.792617	2022	1	1	5	0
6	2022-01-01 06:00:00	0.100412	2022	1	1	6	0
7	2022-01-01 07:00:00	0.187317	2022	1	1	7	0
8	2022-01-01 08:00:00	0.786790	2022	1	1	8	0
9	2022-01-01 09:00:00	0.497147	2022	1	1	9	0
10	2022-01-01 10:00:00	0.138009	2022	1	1	10	0

Outliers low frequency filter

- Similar to panel data case
- but it could be implemented to be a streaming process
- IQR




```
# Convert the data to a Pandas Series
series = pd.Series(data)

# Calculate the first and third quartiles
q1 = series.quantile(0.25)
q3 = series.quantile(0.75)

# Define the filter based on the interquartile range (IQR)
iqr = q3 - q1
filter = (series >= q1 - 1.5*iqr) & (series <= q3 + 1.5*iqr)

# Filter the data
filtered_data = series[filter]
```

Harmonics decomposition

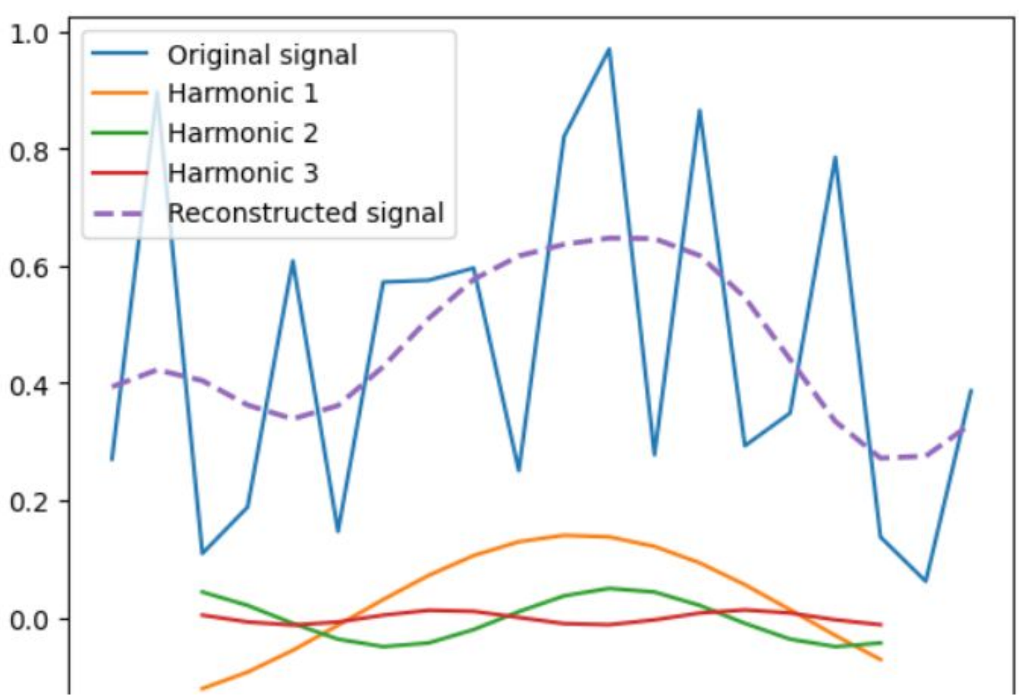
Extract seasonality from a time series, decomposing them into its trend, seasonal, and residual components.

Fourier

- Decompose a signal into its frequency components
- based on the Fourier series
- any periodic function can be represented as a sum of sine and cosine waves of different frequencies, phases, and amplitudes
- the time series data is first transformed into the frequency domain using a Fourier transform
- The amplitudes and phases of these waves are then estimated using a least-squares regression

```python
# Calculate the Fourier coefficients for each harmonic separately
num_harmonics = 3
all_coeffs = np.fft.fft(series)
coeffs = []
for i in range(1, num_harmonics+1):
    coeffs.append(np.zeros(len(all_coeffs), dtype=complex))
    coeffs[-1][i] = all_coeffs[i]
    coeffs[-1][-i] = all_coeffs[-i]

# Reconstruct the signal using the first 3 harmonics
reconstructed_coeffs = np.zeros(len(all_coeffs), dtype=complex)
for i in range(num_harmonics):
    reconstructed_coeffs += coeffs[i]
reconstructed_signal = np.fft.ifft(reconstructed_coeffs).real
reconstructed_signal += series.mean()
```



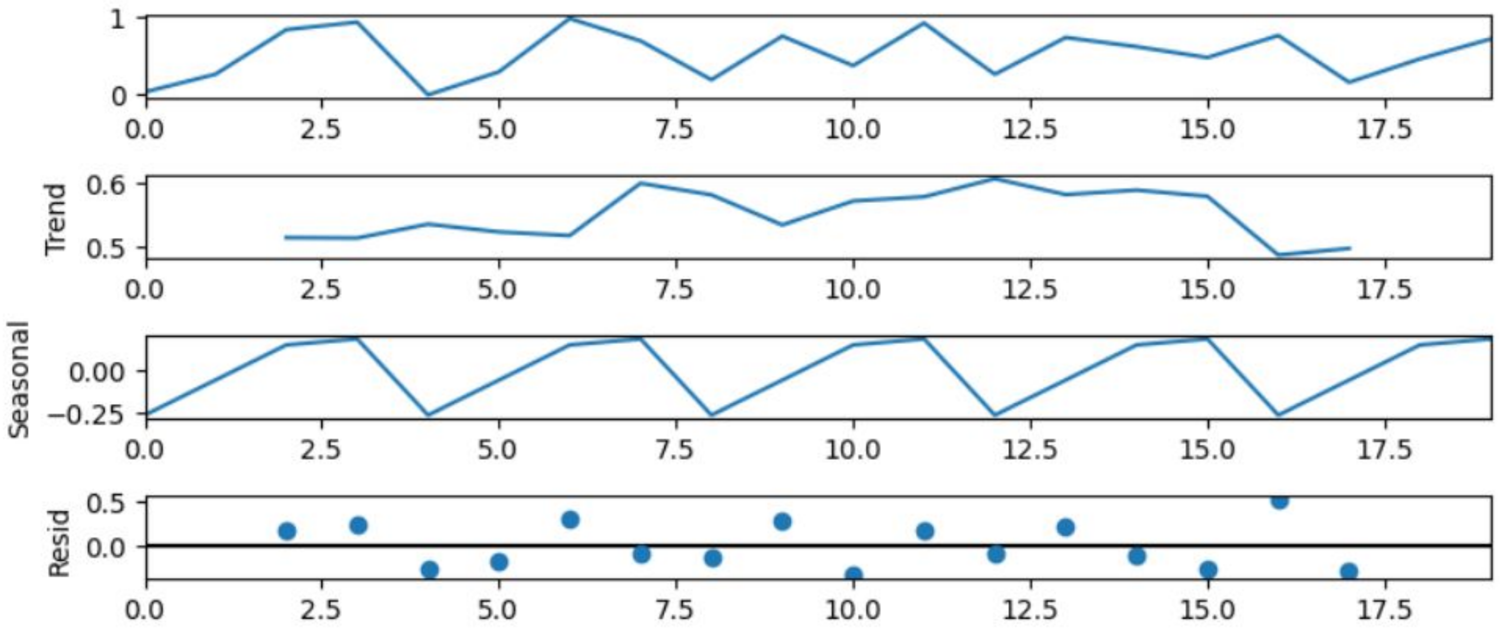
Harmonics decomposition

Seasonality analysis

- uses the classical time series decomposition method based on moving averages

```
# Perform the decomposition
decomposition = sm.tsa.seasonal_decompose(series, model='additive', per

fig=decomposition.plot();
fig.set_size_inches((8, 3.5));
fig.tight_layout();
```



Approaches for estimation of coefficients

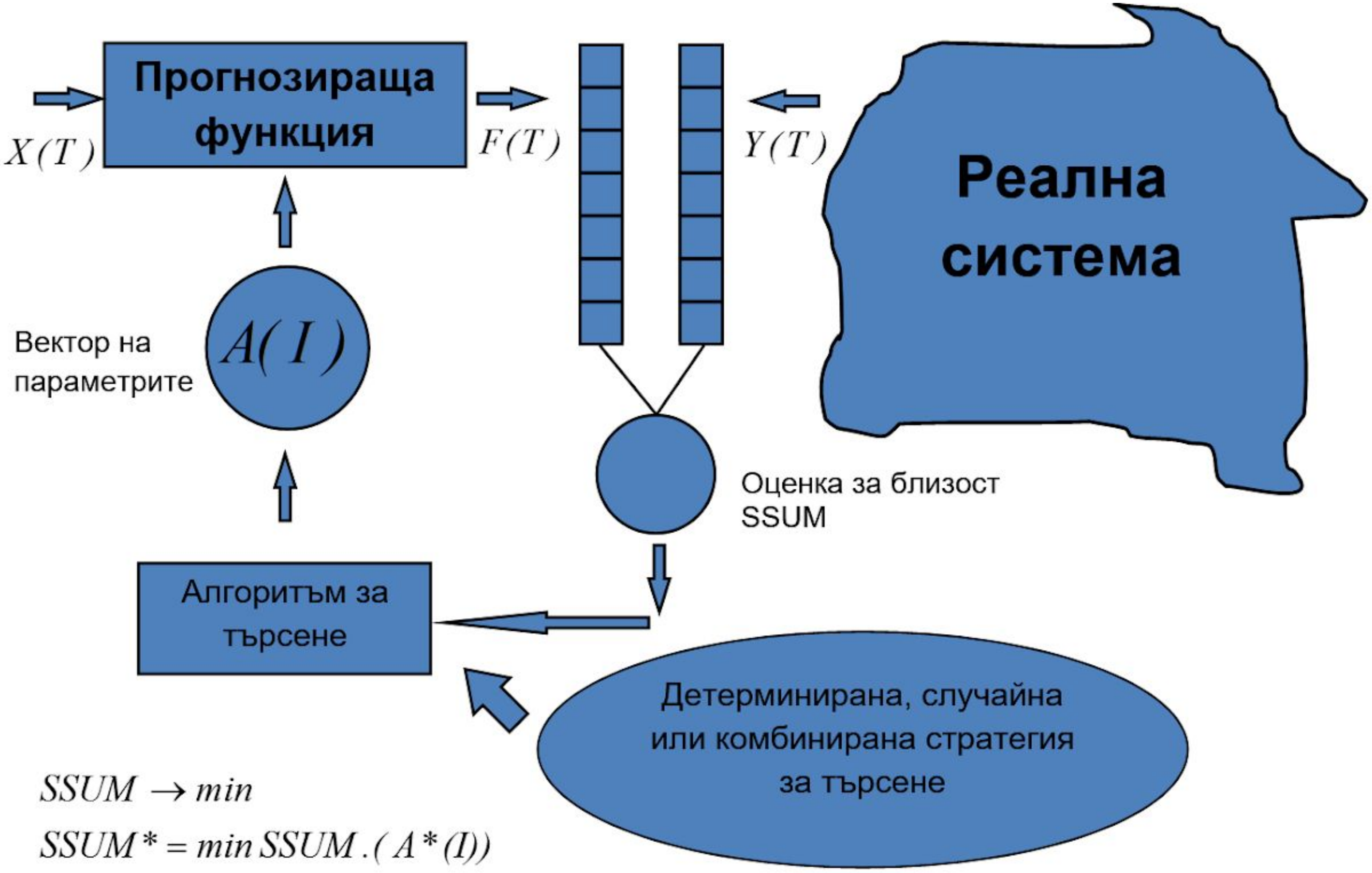
- Analytical
  - Ordinary least squares (OLS)
  - Maximum likelihood (ML)
  - Bayesian

- Iterative…

- but…

- All of these are in fact optimization

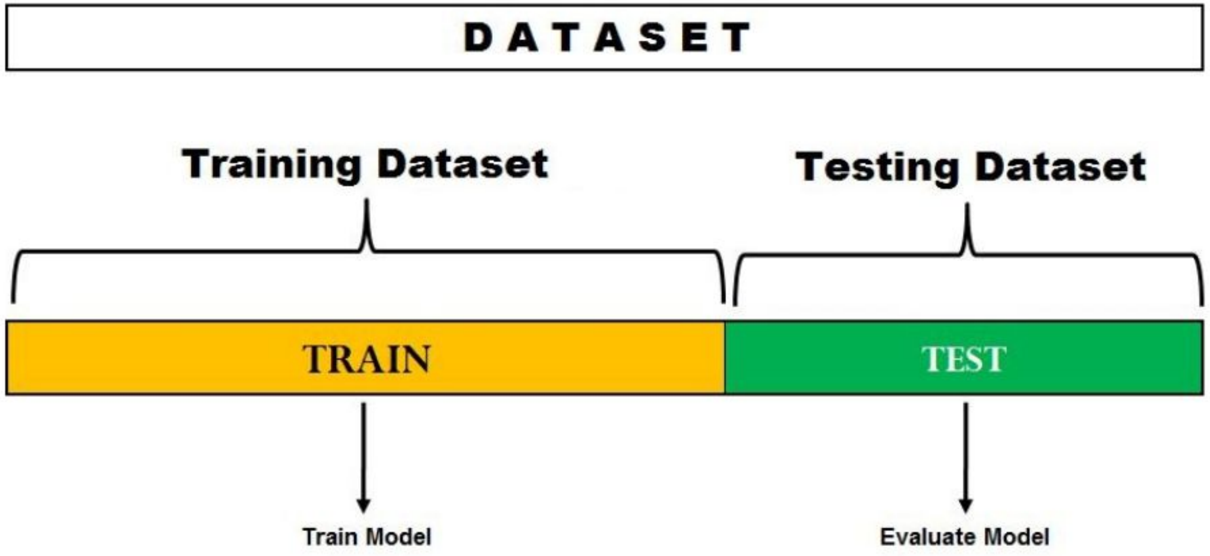
Optimization



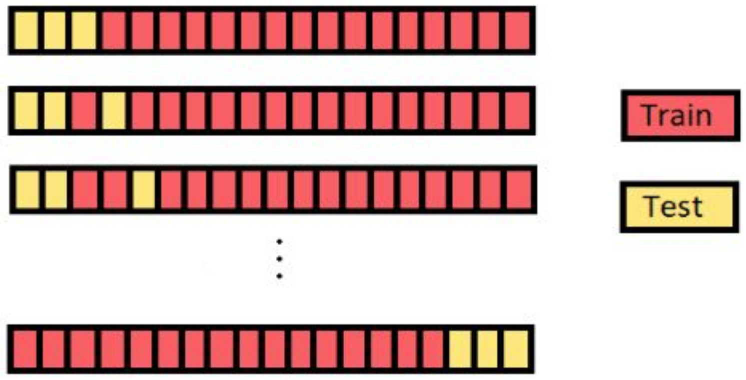
Validation

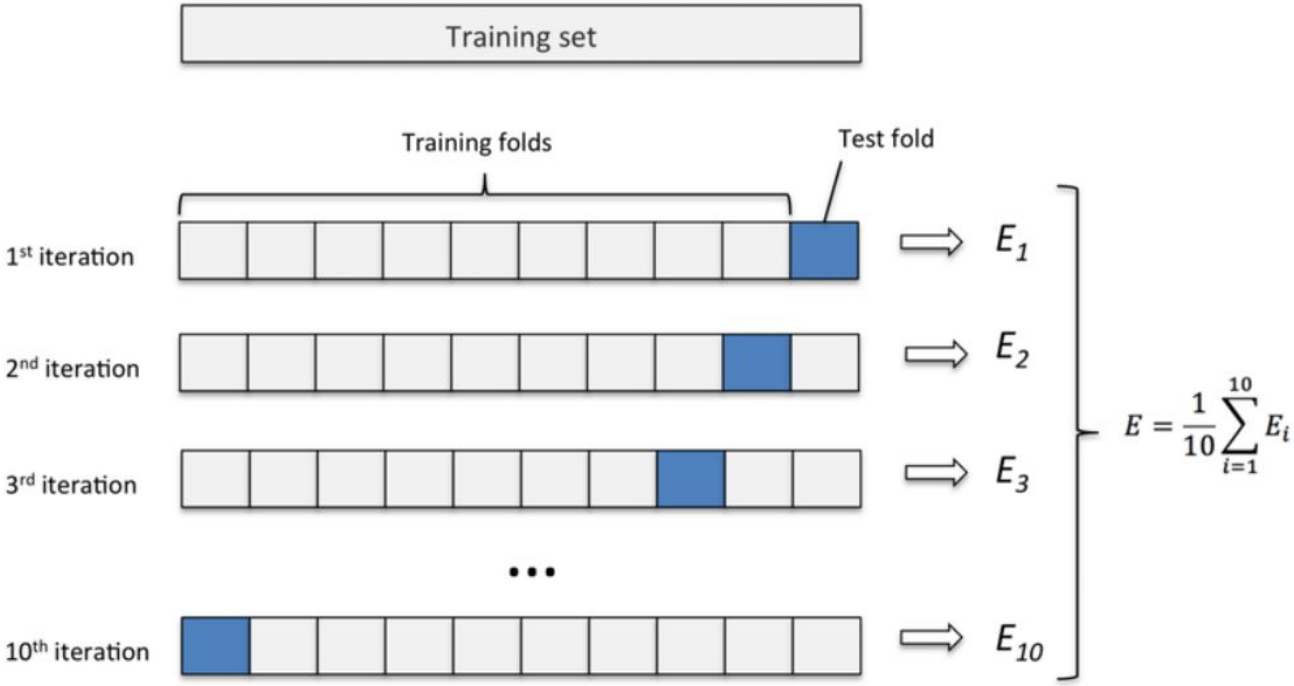
- Split sample validation
  - Training set
  - Test set

  - Validation subset/method



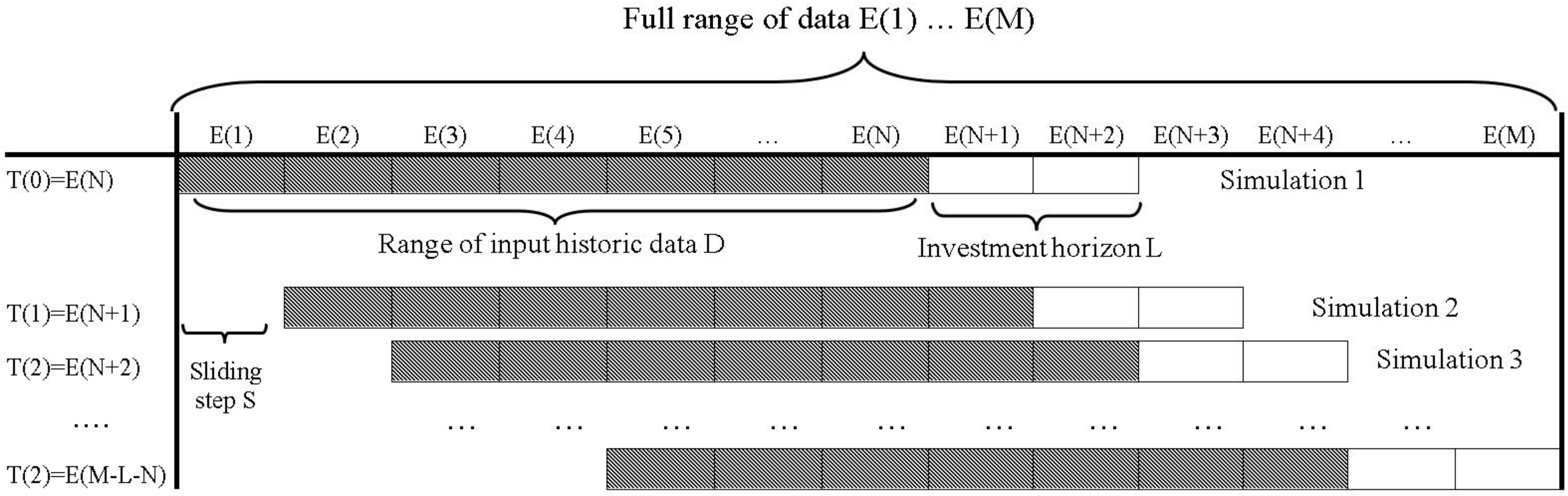
Leave P-out Cross Validation





Validation

- Validation with moving window



Overfitting

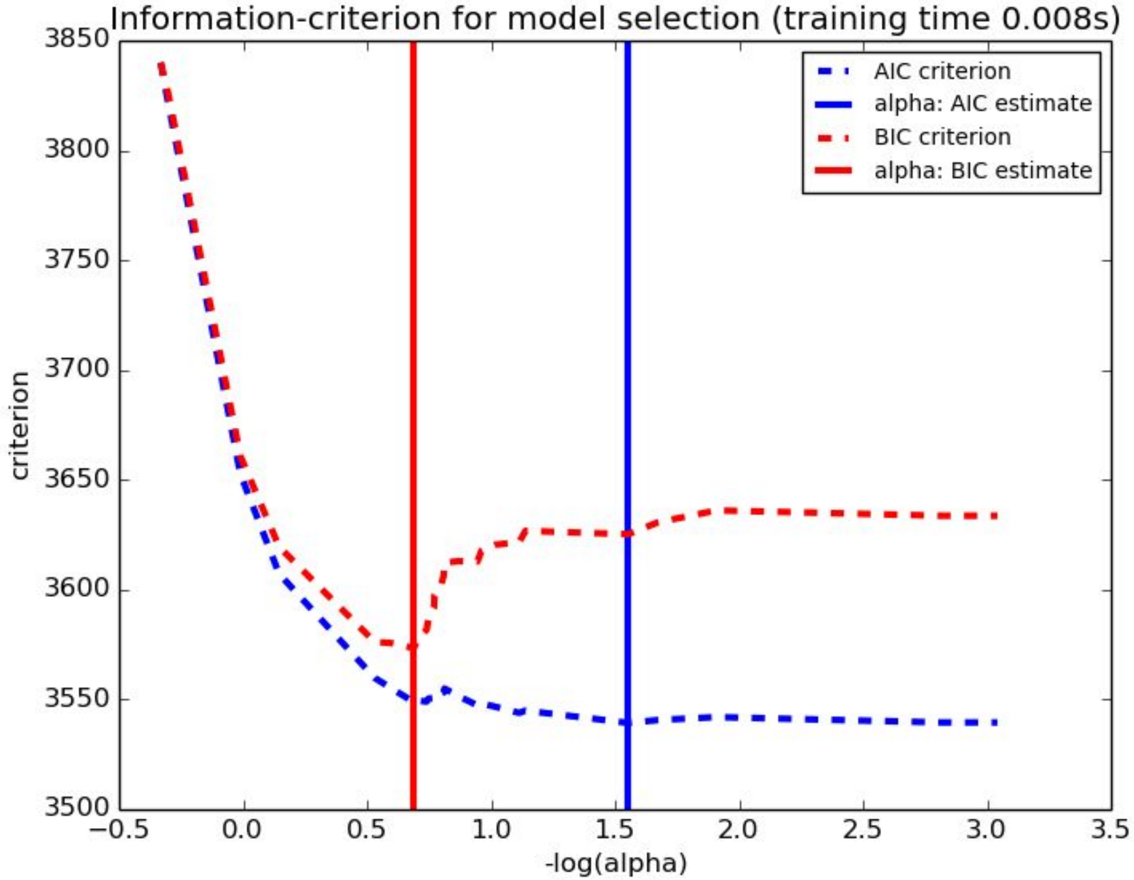
- Akaike information criterion (AIC)

$$AIC = 2k - 2.\ln(\hat{L})$$

- Bayesian information criterion (BIC) or Schwarz information criterion

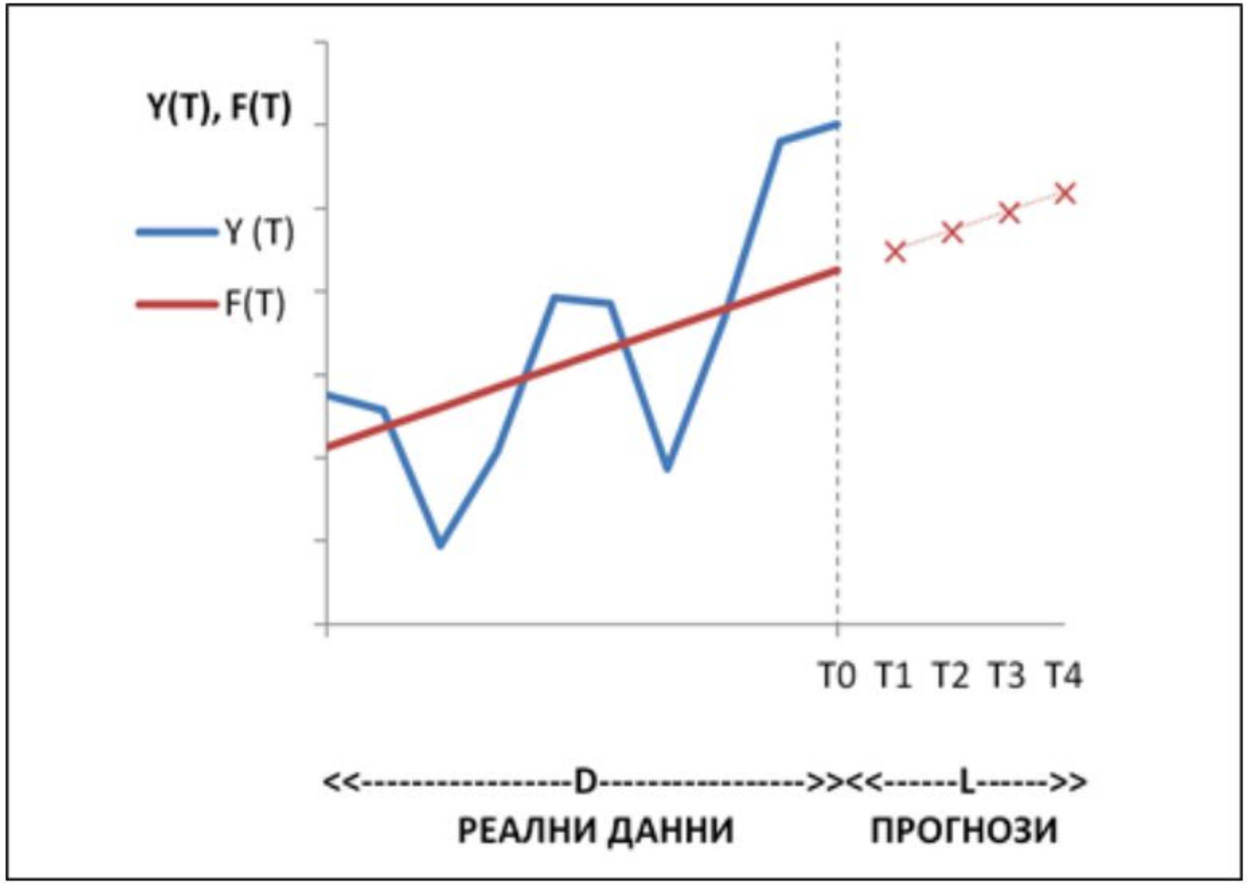
$$BIC = k.ln(n) - 2.\ln(\hat{L})$$

$$AIC/BIC = min$$



Forecasts

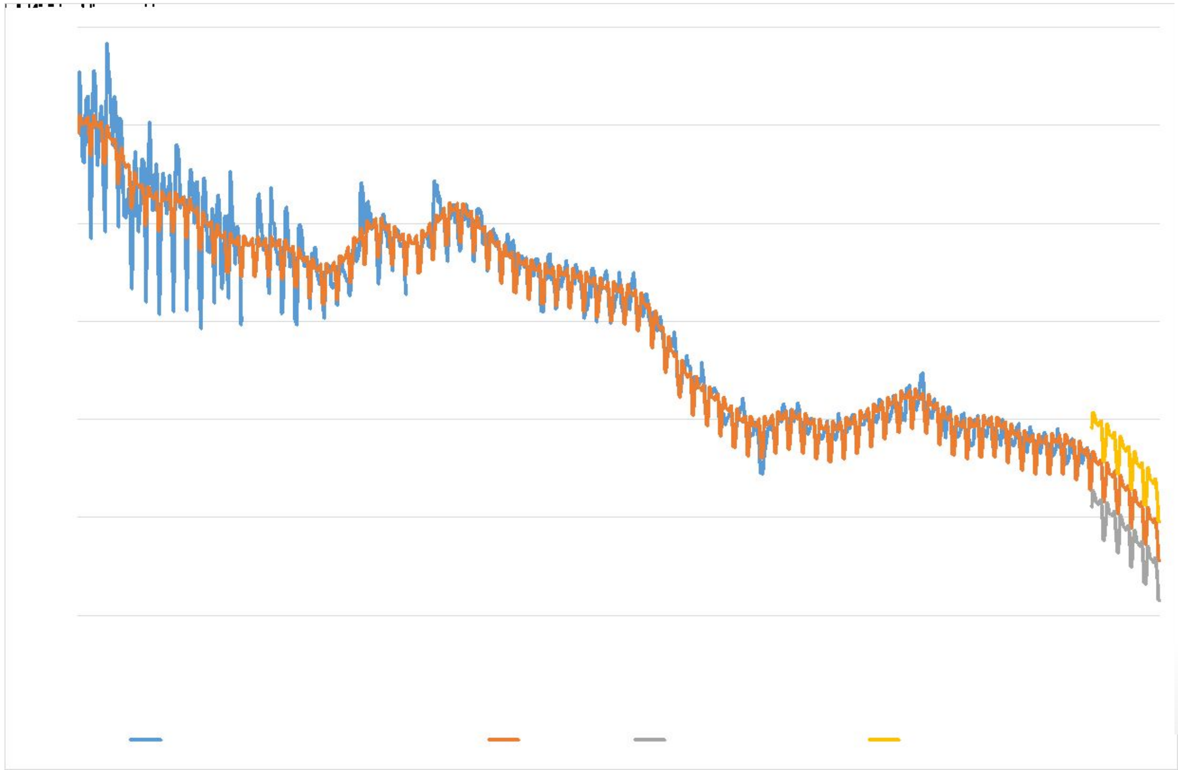
- Extrapolation/interpolation
- Analytical forecasts
- Target forecasts

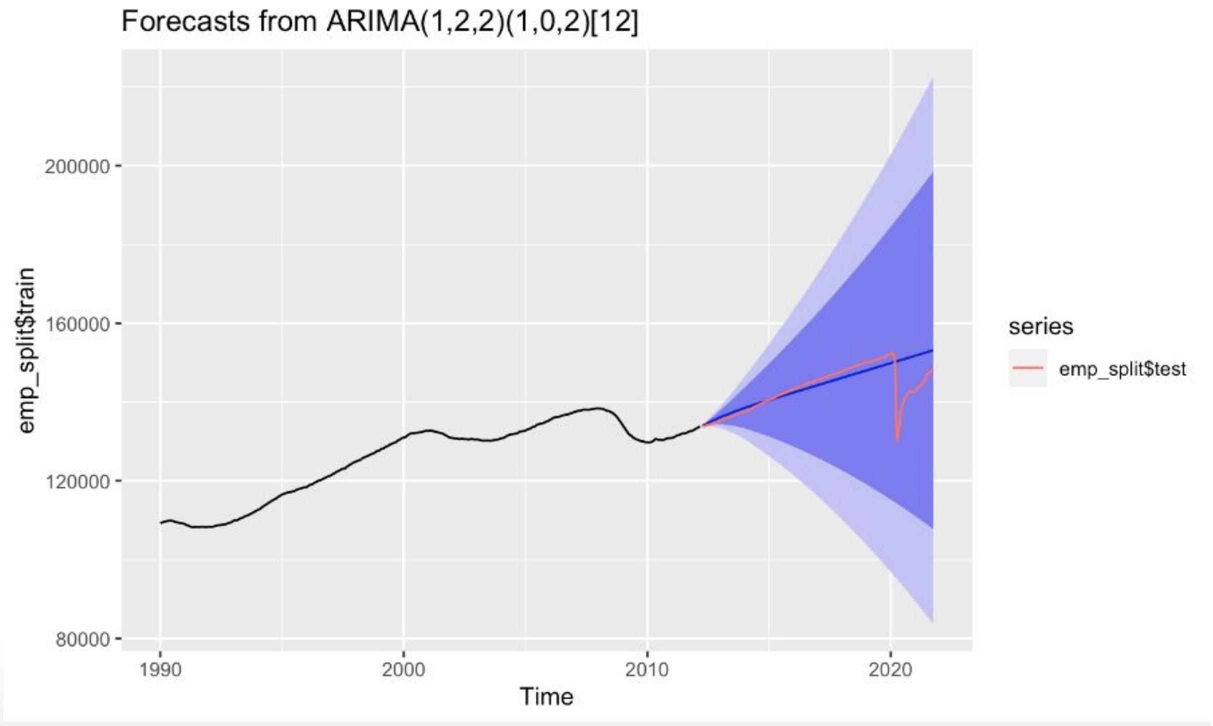


Forecast horizon

- Short term, medium term, long term
- Depending of time series length

Confidence interval of forecast





Evaluation of forecast

- Mean squared error (MSE)

$$MSE = \frac{\sum(y - \hat{y})^2}{n}$$

- Root mean squared error (RMSE)

$$RMSE = \sqrt{MSE} = \sqrt{\frac{\sum(y - \hat{y})^2}{n}}$$

- Mean absolute error (MAE)

$$MAE = \frac{\sum|y - \hat{y}|}{n}$$