Lab 4 : AutoRegressive Integrated Moving Average

Jeroen Olieslagers
Richard-John Lin

Center for Data Science

09/27/2020

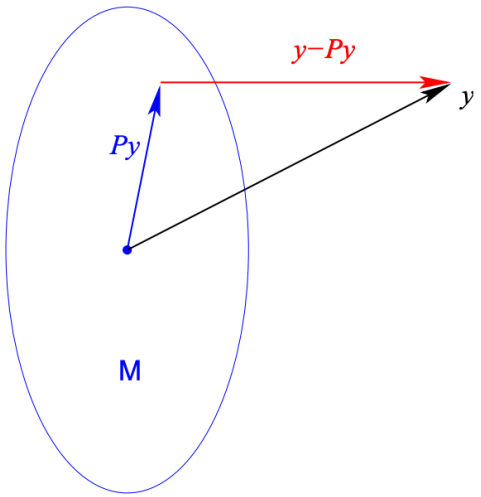
Projection theorem

Theorem

Let $\mathcal{H}$ a Hilbert space, $\mathcal{M}$ a closed subset of $\mathcal{H}$ and $y \in \mathcal{H}$. Then $Py \in \mathcal{M}$ is the orthogonal projection of y on $\mathcal{M}$ iif :

$$\forall w \in \mathcal{M}, \quad \|Py - y\| \leq \|w - y\|,$$
$$< y - Py, w >= 0.$$

This projection is unique.



Application

Conditions of best linear predictor

Let $n, m \in \mathbb{N}$.
The best predictor of x_{n+m} :

$$x_{n+m}^{n} = \alpha_0 + \sum_{k=1}^{n} \alpha_k x_k, \tag{1}$$

must satisfy the following conditions :

$$\mathbb{E}\left[x_{n+m} - x_{n+m}^{n}\right] = 0 \tag{2}$$

$$\forall k \in [\![1, n]\!], \qquad \mathbb{E}\left[\left(x_{n+m} - x_{n+m}^{n}\right) x_k\right] = 0 \tag{3}$$

Proof

Rewriting eq.2, we have :

$$\mu = \alpha_0 + \sum_{k=1}^{n} \alpha_k \mu.$$

Then, by replacing α_0 in 1, the form of the best linear predictor is :

$$x_{n+m}^{n} = \mu + \sum_{k=1}^{n} \alpha_k (x_k - \mu).$$

This means that without loss of generality, we can consider $\mu = 0$.

Method of moments : Yule-Walker equations

We can rewrite the condition 3 for a single step in the future :

$$x_{n+1}^n := \phi_{n,1} x_n + \phi_{n,2} x_{n-1} + \cdots + \phi_{n,n} x_1, \qquad (\phi_{n,n+1-k} := \alpha_k)$$

$$\forall k \in [\![1, n]\!], \quad \mathbb{E}\left[\left(x_{n+1} - \sum_{j=1}^{n} \phi_{n,j} x_{n+1-j}\right) x_{n+1-k}\right] = 0$$

$$\sum_{j=1}^{n} \phi_{n,j} \gamma(k-j) = \gamma(k)$$

Method of moments : Yule-Walker equations

Writing the condition in matrix form, we have :

$$\begin{bmatrix} \gamma(0) & \gamma(-1) & \dots & \gamma(1-n) \\ \gamma(1) & \gamma(0) & \dots & \gamma(2-n) \\ \vdots & \dots & \ddots & \vdots \\ \gamma(n-1) & \gamma(n-2) & \dots & \gamma(0) \end{bmatrix} \begin{bmatrix} \phi_{n,1} \\ \phi_{n,2} \\ \vdots \\ \phi_{n,n} \end{bmatrix} = \begin{bmatrix} \gamma(1) \\ \gamma(2) \\ \vdots \\ \gamma(n) \end{bmatrix}$$

Method of moments : Yule-Walker equations

Writing the condition in matrix form, we have :

$$\begin{bmatrix} \gamma(0) & \gamma(1) & \dots & \gamma(n-1) \\ \gamma(1) & \gamma(0) & \dots & \gamma(n-2) \\ \vdots & \dots & \ddots & \vdots \\ \gamma(n-1) & \gamma(n-2) & \dots & \gamma(0) \end{bmatrix} \begin{bmatrix} \phi_{n,1} \\ \phi_{n,2} \\ \vdots \\ \phi_{n,n} \end{bmatrix} = \begin{bmatrix} \gamma(1) \\ \gamma(2) \\ \vdots \\ \gamma(n) \end{bmatrix}$$

$$\Gamma_n \phi_n = \gamma_n.$$

As x_{n+1}^n is unique by the projection theorem, then the matrix is invertible.

Dividing by the covariance, the same system can be written with autocorrelations, and for the Yule-Walker equations.

Note that by construction, as the best linear predictor, $\phi_{n,n}$ is the PACF for a lag n.

Durvin-Levinson method

The naive way to solve the system using matrix inversion has a complexity of $O(n^3)$.
Durvin-Levinson method allows to solve this problem in $O(n^2)$ by proceeding iteratively.

$$\left[\begin{array}{ccc|c} \gamma(0) & \dots & \gamma(n-2) & \gamma(n-1) \\ \vdots & \ddots & \vdots & \vdots \\ \gamma(n-2) & \dots & \gamma(0) & \gamma(1) \\ \hline \gamma(n-1) & \dots & \gamma(1) & \gamma(0) \end{array}\right] \left[\begin{array}{c} \phi_{n,1} \\ \vdots \\ \phi_{n,n-1} \\ \hline \phi_{n,n} \end{array}\right] = \left[\begin{array}{c} \gamma(1) \\ \vdots \\ \gamma(n-1) \\ \hline \gamma(n) \end{array}\right].$$

Durvin-Levinson method

The naive way to solve the system using matrix inversion has a complexity of $O(n^3)$.
Durvin-Levinson method allows to solve this problem in $O(n^2)$ by proceeding iteratively.

$$\begin{bmatrix} \Gamma_{n-1} & \tilde{\gamma}_{n-1} \\ \tilde{\gamma}_{n-1}^\top & \gamma(0) \end{bmatrix} \begin{bmatrix} \phi_{n-1} - \phi_{n,n}\tilde{\phi}_{n-1} \\ \phi_{n,n} \end{bmatrix} = \begin{bmatrix} \gamma_{n-1} \\ \gamma(n) \end{bmatrix}.$$

We can then invert this matrix by block, which gives :

$$\phi_{n,n} = \frac{\gamma(n) - \tilde{\gamma}_{n-1}^\top \Gamma_{n-1}^{-1} \gamma_{n-1}}{\gamma(0) - \tilde{\gamma}_{h-1}^\top \Gamma_{n-1}^{-1} \tilde{\gamma}_{n-1}}.$$

Durvin-Levinson method

Finally, we obtain :

$$\begin{aligned}\phi_{n,n} &= \frac{\gamma(n) - \phi_{n-1}^{\top}\gamma_{n-1}}{\gamma(0) - \phi_{n-1}^{\top}\tilde{\gamma}_{n-1}} \\ &= \frac{\gamma(n) - \sum_{k=1}^{n-1}\phi_{n-1,k}\gamma(n-k)}{\gamma(0) - \sum_{k=1}^{n-1}\phi_{n-1,k}\gamma(k)},\end{aligned}$$

where

$$\phi_{n,k} = \phi_{n-1,k} - \phi_{n,n}\phi_{n-1,n-k}.$$

As per the previous remark, the same equation can be written with autocorrelations instead.

Maximum likelihood

Consider an ARMA(p,q) model :

$$x_t - \sum_{h=1}^{p} \phi_h x_{t-h} = w_t + \sum_{h=1}^{q} \theta_h w_{t-h},$$

where $w_t \underset{i.i.d.}{\sim} \mathcal{N}(0, \sigma^2)$.
By independence of w_t, we have :

$$x_{t+1} | x_{1:t} \sim \mathcal{N}\left(0, \sigma^2\left(1 + \sum_{h=1}^{q} \theta_h^2\right)\right).$$

Maximum likelihood

Hence the negative log-likelihood function :

$$NLL(\phi_1, \ldots, \phi_p, \theta_1, \ldots, \theta_q, \sigma) = \frac{1}{2} \sum_{t=p+1}^{T} \log\left(2\pi\tilde{\sigma}^2\right) + \left(\frac{x_t - \sum_{h=1}^{p} \phi_h x_{t-h}}{\tilde{\sigma}^2} \right)^2,$$

with

$$\tilde{\sigma}^2 = \sigma^2 \left(1 + \sum_{h=1}^{q} \theta_h^2 \right).$$

Parameter estimation : Remarks

- Moments estimators are often used as initial estimates for iterative procedures (including MLE). The estimates gets better as the number of points increase.
- For AR models, the Yule-Walker equations are used for this initial guess.
- For general ARMA processes, Newton-Raphson method is often used.
- In general solutions obtained by least squares estimation are satisfying approximations with a big sample size. For smaller sizes, the maximum likelihood is preferred especially if the MA process is close to the invertibility region.

ARIMA modeling

To model a time series as an ARIMA(p,d,q) :

1. Stationarize the time series (d).
2. Identify reasonable values for p, q using the ACF and PACF.
3. Estimate the parameters of the ARIMA model given previous parameters/
4. Assess if the residuals are white noise and select the best model/
5. Compute the loss on a test set.

Some general considerations

- Beware of overdifferencing ! The optimal order of differencing is often the one where the standard deviation is the lower, but not always.
  Differencing tend to introduce negative correlation.
- Slightly too much or little difference can be corrected with AR and MA terms.
- Always check for parameter redundancy.
- More rules here.

Part I: Applying an AR(p) model

Apply AR Model

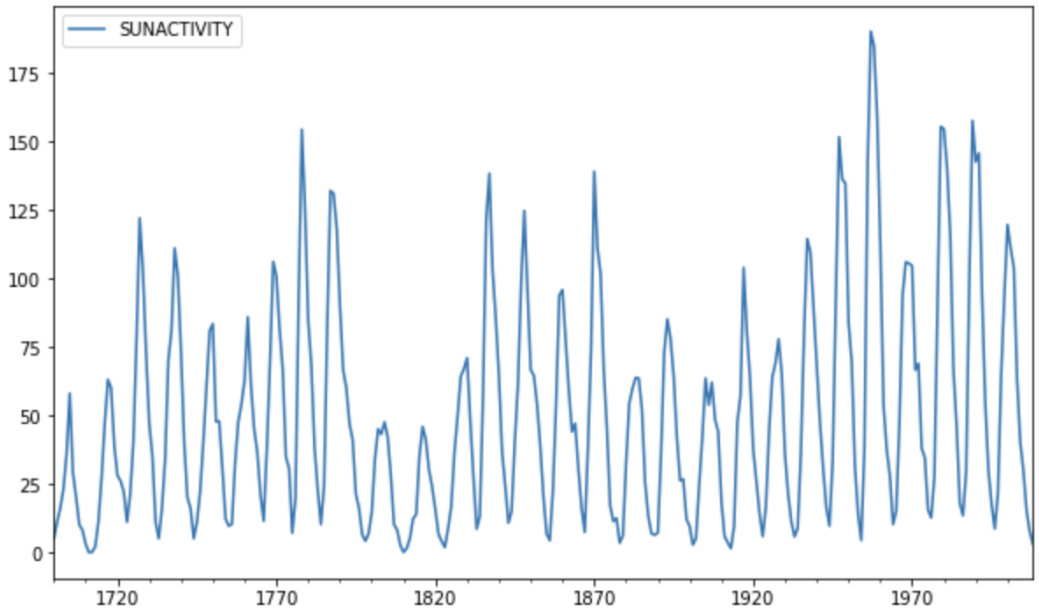
In this example, we will fit an AR(p) model to the SunActivity data, which denotes the number of sunspots for each year.

We will determine p, fit the model, compute the roots and the lag 0 to p components of the ACF.

Wikipedia for sunspots: https://en.wikipedia.org/wiki/Sunspot

The code in this section is selected from the tutorial specified in the reference section.

```
dta = sm.datasets.sunspots.load_pandas().data
dta.index = pd.Index(sm.tsa.datetools.dates_from_range('1700', '2008'))
del dta["YEAR"]
dta.plot(figsize=(10,6))
plt.show()
```



Part I: Fitting the model parameters and calculating ACF and roots

Fit AR Model of order p

```
# TODO: chose p appropriately
p =

arma_mod = sm.tsa.ARIMA(dta, order=(p,0,0)).fit()
print(arma_mod.params)

# TODO: predict ACF of model at lag 0, 1, ..., p
# Hint: use arma_mod.params[n] to access the nth parameter of the AR model
rho = np.zeros(p+1)

# TODO: compute roots
roots = np.zeros(2)

print('roots: ', roots)
```

```
const      49.746198
ar.L1       1.390633
ar.L2      -0.688573
sigma2    274.727181
dtype: float64
roots:  [0. 0.]
```

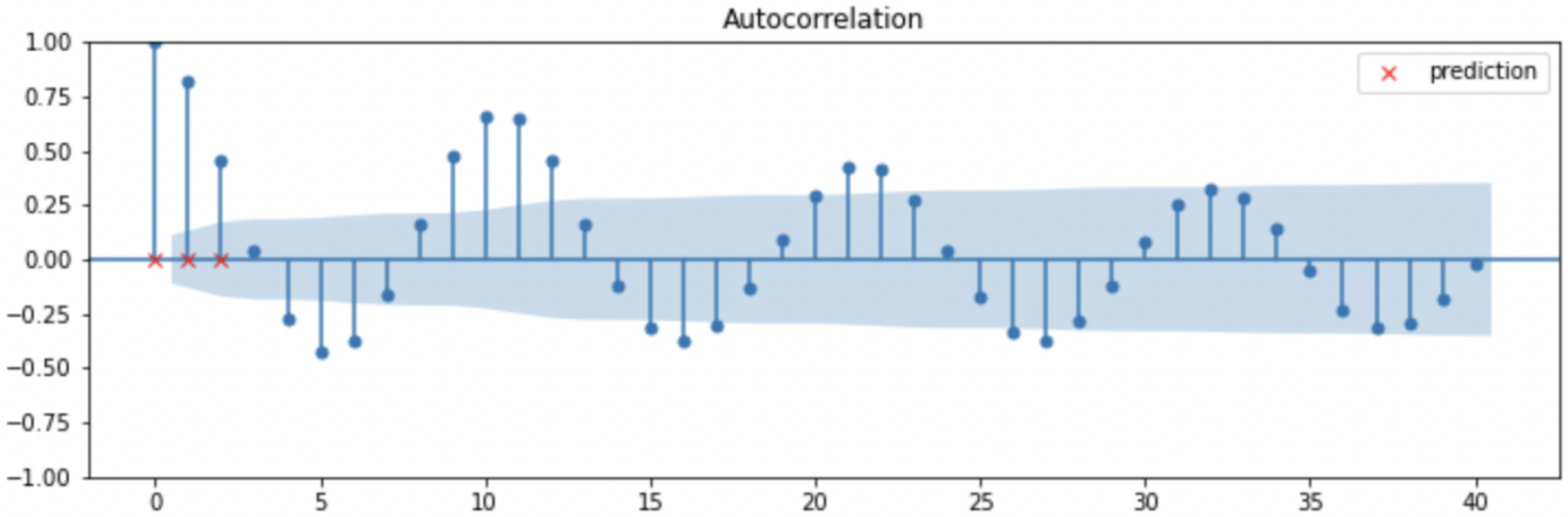
Part I: Predicting the future

prediction

```
# TODO: try to predict further into the future by increasing tsteps
tsteps=

fig, ax = plt.subplots(figsize=(12, 8))
ax = dta.loc['1900':].plot(ax=ax)
T = np.arange(1930, 2030, tsteps)
for tt in range(len(T)-1):
    pred = arma_mod.predict(str(T[tt]), str(T[tt+1]), ax=ax, plot_insample=False)
    pred.plot(ax=ax)
plt.show()
```

Part I: Checking the ACF is correct



Part II: Creating your own AR model

Implement AR model from Scratch

Section I: Implementing the AR Model

Recall that the negative log-likelihood function takes as input the parameter values and returns the negative log probability of the observed data, under the assumption that those were the parameters used to generate the data.

For an AR(p) model, we have:

$$NLL(\phi_1, \phi_2, \dots, \phi_p, \sigma\,; x_1, x_2, \dots, x_n) = \sum_{t=p+1}^{n} \left(\log\left(\sigma\sqrt{2\pi}\right) + \frac{1}{2} \cdot \left(\frac{x_t - \left(\sum_{i=1}^{p} \phi_i x_{t-i} \right)}{\sigma} \right)^2 \right)$$

Optional: We use Nelder-Mead Algorithm to find the minimum of the loss. Check this link if you're interested (https://machinelearningmastery.com/how-to-use-nelder-mead-optimization-in-python/)

Part II: Creating the model class

```
class ARModel:
    """Class that implements an ARMA Model. Its functions are as follows:
    1. Maximum Likelihood estimation of parameters
    2. Inference/prediction of future states
    3. Data simulation
    """
    def __init__(self, p, data, p_params = None, sigma = None):
        """Initialize the network state
        @param p: the number of time steps to include in the AR process
        @param p_params: the initialization for the AR parameters
        """
        if (p_params is None):
            p_params = np.zeros(p)
        if (sigma is None):
            sigma = 1

        assert p == len(p_params)

        #assign parameter values
        self.p = p
        self.p_params = p_params
        self.sigma = sigma
        #store the data within the object
        self.data = data
```

Part II: Creating the model class

```
    def loss(self, params):
        """
        params: array of parameters, elements 0:p = p_params, element p = sigma
        returns: loss
        """
        assert len(params) == self.p + 1
        N = self.data.shape[0]
        p_params = params[0:self.p]
        sigma = params[self.p]
        loss = 0

        #TODO: calculate the NLL of the data for the purposes of optimization and store it in loss

        return loss

    def fit(self):
        # Minimize the loss function, given the dataset
        params = np.concatenate((self.p_params, np.array([self.sigma])))
        res = minimize(self.loss, params, method='nelder-mead',
               options={'xatol': 1e-8, 'disp': True})
        self.p_params = res.x[0:self.p]
        self.sigma = res.x[self.p]
```

Part II: Creating the model class

```
    def predict(self,data, N):
        """Method that predicts N timesteps in the future given input data
        @params data: p data points used to form the prediction
        @params N: number of time steps to predict in the future

        returns:
        prediction: predicted future value
        conf: variance of the estimated future value
        """
        assert len(data) == self.p
        predction = np.zeros(N)
        conf = np.zeros(N)

        #TODO: predict N time steps in advance, given an input.
        #The inference can be specific to your choice of p, no need to worry about general inference here

        return prediction, conf

    def simulate(self,N):
        """Method that stimulates data given the p_params and q_params
        @param N: number of datapoints to simulate
        returns: N sampled datapoints
        """
        transient = 100 # length of time to run the simulation to wash out initial conditions
        w_t = self.sigma * np.random.normal(size = (N + transient,))
        x_t = np.zeros(N + transient)

        # TODO: generate data x_t given the parameters and white noise w_t

        return x_t[transient::] #discard the transient when returning simulated data
```

Part II: Fitting your model

Section II: Fitting the AR Model

In this section, we will load some data from an unknown source, look at its ACF and PACF plots to determine an appropriate AR(p) order, and fit the AR(p) model to the data to determine the coefficients of the AR model as well as the standard deviation of the driving white noise process.

Part II: Fitting your model

```
# fit the model
p = 0 #TODO: choose a 'p' value

# set p_params and sigma to an educated guess for parameter values
data_fitter = ARModel(p, data, p_params = np.array([]), sigma = 0.5)
data_fitter.fit()
print('lambda = ' + str(data_fitter.p_params))
print('sigma = ' + str(data_fitter.sigma))
```

```
Optimization terminated successfully.
         Current function value: -1090.532908
         Iterations: 69
         Function evaluations: 134
lambda = [0.59225995]
sigma = 0.20359459057446244
```

Part II: Simulate synthetic data and compare

Section III: Simulating data

Now, we will use our fitted model to simulate a run of the AR model.

```
#Generate 1000 samples from the fit model
data_2 = data_fitter.simulate(1000)

#Compare the ACF from the fit model to the data ACF
lag = 20

plot_acf(x=data, lags=lag, title="ACF data")
plot_acf(x=data_2, lags=lag, title="ACF data_2")
plt.show()

#Compare the PACF from the fit model to the data ACF
plt.figure()
sm.graphics.tsa.plot_pacf(data, lags=20)
plt.title('PACF Data')
sm.graphics.tsa.plot_pacf(data_2, lags=20)
plt.title('PACF Data 2')
plt.show()
```

Part II: Predicting the future

Section IV: Using the AR Model for prediction

Finally, we will use some of the provided data as a starting point and predict the next 20 values based on our AR model's fitted parameters. This will be repeated for each of various starting points.

```
#for each of the given data points, generate predictions 20 time steps into the future

data_prediction = data[0:100:25]
predictions = np.zeros((len(data_prediction), 20))
mse = np.zeros((len(data_prediction), 20))
for ii in range(0, len(data_prediction)):
    for jj in range(0,20):
        #for each data point, predict for each of 20 time steps
        predictions[ii,jj], mse[ii,jj] = data_fitter.predict(data_prediction[[ii]],jj)
```

```
#plot the MSE bars of the estimate
plt.figure()
for ii in range(0, len(data_prediction)):
    plt.errorbar(np.arange(0,20), predictions[ii,:], yerr = mse[ii,:])
plt.legend(data_prediction)
plt.show()
```