The George Washington University

Laboratory Eight:

Estimating Model Orders with Generalized Partial Autocorrelation

Fernando Zambrano

DATS 6450: Multivariate Modeling

Dr. Reza Jafari

1 April 2020

Abstract:

An autoregressive model predicts the future behavior of a series based upon its past behavior. There are three models that are discussed the autoregressive, AR(na), moving average, MA(nb), and Autoregressive Moving Average, ARMA(na,nb). The order of these processes is not always known. Using Python, the Generalized Partial Autocorrelation or GPAC, is a method that estimates the order of an ARMA process.

Introduction:

ARMA models are composed of both an AR(na), and an MA(nb) models. ARMA models are popular and powerful in the forecasting industry due to its reliability and flexibility in predicting future values of single variable base on past behavior. However, one of the shortcomings of using ARMA models for experimental and research purposes is that their performance is dependent on using the proper order or lag value of the AR and MA components for a dataset. This information is not always known nor is it trivial to detect. For this reason, the GPAC method is used to estimate the best order of an ARMA process for a given dataset. Once order determination is established parameter estimation can then be achieved.

Methods & Theory:

The autoregressive models are similar to multivariable linear regression, where the predictors are lagged versions of the series with their own coefficients. This model can have up to *na* predictors or coefficients which depends on the number of lags taken into the model. This also determines the order of the model. An AR(2) model will take into account the series lagged at t -1 and t -2 steps. This concept often used to refer to AR models as "AR (na) models." One of the principle requirements of AR models is that they require stationarity – no trend or seasonality. There needs to be a constant level of variance and autocorrelation throughout the entire series.

A powerful characteristic of AR models is that they have "long memory." Each observation at k lags before have an effect on the current observation, and on each succeeding observation, at either large or small quantiles. Hence the information from the very beginning is stored through the entire data. The effects of the first values have little effect on the current observations IF the coefficient of the first observation is less than 1.

The formula for AR(na) is below.

$$y(t) + a_1 y(t-1) + a_2 y(t-2) + \ldots + a_{n_a} y(t-n_a) = \epsilon(t)$$

where $y(t)$ is the variable of interest and $\epsilon(t)$ is white noise $(WN \sim (0, \sigma_\epsilon^2)$.

Since AR models are implementing regression, the predictors or coefficients for each lagged series can be estimated using Least Square Estimate (LSE). The optimal coefficients for a certain AR(na) model can be solved through matrix algebra through the following formula:

Moving Average models or MA models are similar to AR models, but rather than making forecasts with pervious based on past values it uses the pervious forecaster errors. The order of the MA model depends on the lags between forecast errors, or q. Hence moving average models are referred to as MA(nb) models. The equation for MA(nb) models is below, followed by its backshift operator formula.

$$y(t) = \epsilon(t) + b_1\epsilon(t-1) + b_2\epsilon(t-2) + \ldots + b_{n_b}\epsilon(t-n_b)$$

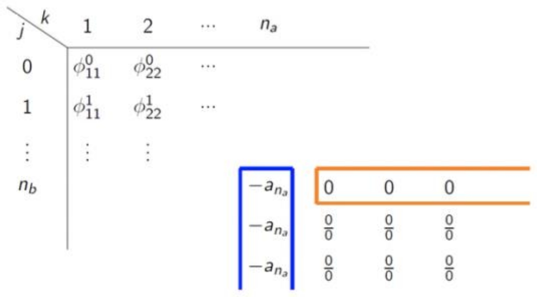
where $\epsilon(t)$ is white noise $WN \sim (0, \sigma_\epsilon^2)$. This is called a moving average model of order n_b , $\boldsymbol{MA(n_b)}$.

The Autoregressive Moving Average (ARMA) is a combination of AR(na) and MA(nb) models. The ARMA model is one the most popular methods used for time stationary time series analysis since it offers the minimum number of parameters to forecast the unknown. The equation for the ARMA(na,nb) model is below.

$$\begin{aligned} y(t) + a_1y(t-1) + a_2y(t-2) + \ldots + a_{n_a}y(t-n_a) = \epsilon(t) + \\ b_1\epsilon(t-1) + b_2\epsilon(t-2) + \ldots + b_{n_b}\epsilon(t-n_b) \end{aligned}$$

GPAC uses the Autocorrelation Function or ACF of a dataset to estimate the order of the ARMA process by constructing a table with the possible combinations for the ARMA orders. The parameters of this this method are “j”,”k”, and “phi”. Ry(j) represents the estimated autocorrelation at lag j. “j” also denotes the estimated value nb for the MA process. “k” denotes the estimated value for na for the AR process. Phi is the result the division between the determinate of the matrices of the given formula below:

$$\phi_{kk}^{j} = \frac{\begin{vmatrix} \hat{R}_y(j) & \hat{R}_y(j-1) & \ldots & \hat{R}_y(j+1) \\ \hat{R}_y(j+1) & \hat{R}_y(j) & \ldots & \hat{R}_y(j+2) \\ \vdots & \vdots & \vdots & \vdots \\ \hat{R}_y(j+k-1) & \hat{R}_y(j+k-2) & \ldots & \hat{R}_y(j+k) \end{vmatrix}}{\begin{vmatrix} \hat{R}_y(j) & \hat{R}_y(j-1) & \ldots & \hat{R}_y(j-k+1) \\ \hat{R}_y(j+1) & \hat{R}_y(j) & \ldots & \hat{R}_y(j-k+2) \\ \vdots & \vdots & \vdots & \vdots \\ \hat{R}_y(j+k-1) & \hat{R}_y(j+k-2) & \ldots & \hat{R}_y(j) \end{vmatrix}}$$



$j \backslash k$	1	2	$\cdots$	n_a		
0	ϕ_{11}^0	ϕ_{22}^0	$\cdots$			
1	ϕ_{11}^1	ϕ_{22}^1	$\cdots$			
$\vdots$	$\vdots$	$\vdots$				
n_b				$-a_{n_a}$	0	0	0
				$-a_{n_a}$	$\frac{0}{0}$	$\frac{0}{0}$	$\frac{0}{0}$
				$-a_{n_a}$	$\frac{0}{0}$	$\frac{0}{0}$	$\frac{0}{0}$

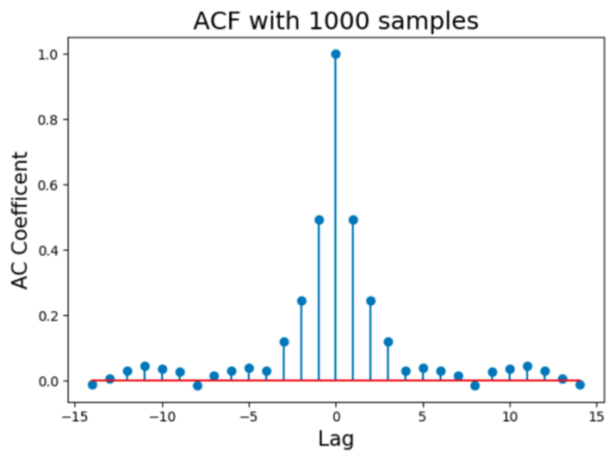
Where $\hat{R}_y(j)$ is the estimated autocorrelation of y(t) at lag j.

Implementation and Results:

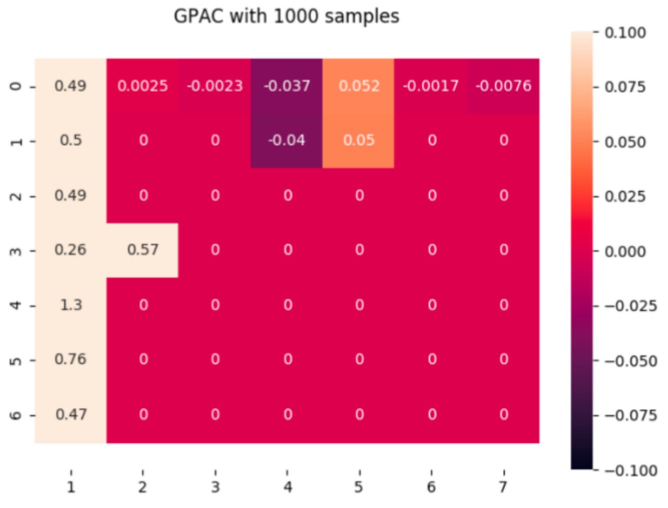
1. Create a program that generates an ARMA(na,nb) process which asks for user input which asks for number of samples, orders for AR and MA, and their corresponding parameters.
2. Add a GPAC program to the ARMA generator.
3. Using the developed code above, simulate ARMA(1,0) for 1000 samples as follows:

$$y(t) - 0.5y(t-1) = e(t) \text{ Where e(t) is WN(0,1)}$$

4. Estimate ACF with 15 lags.



5. Plot the GPAC 7x7 table

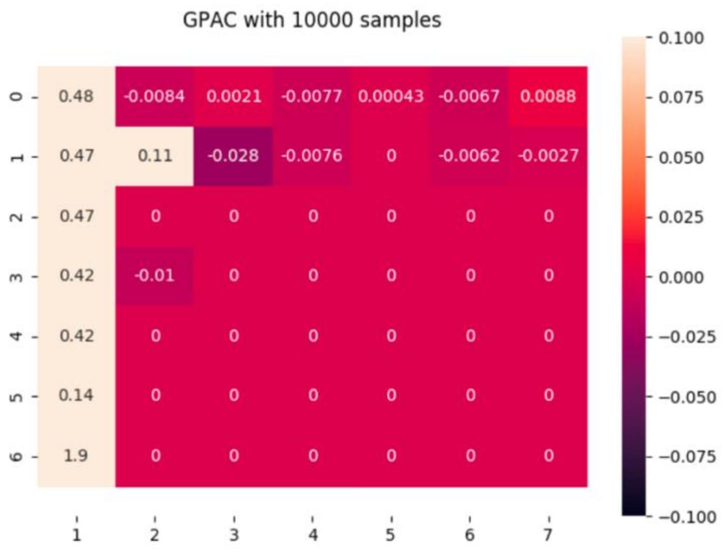


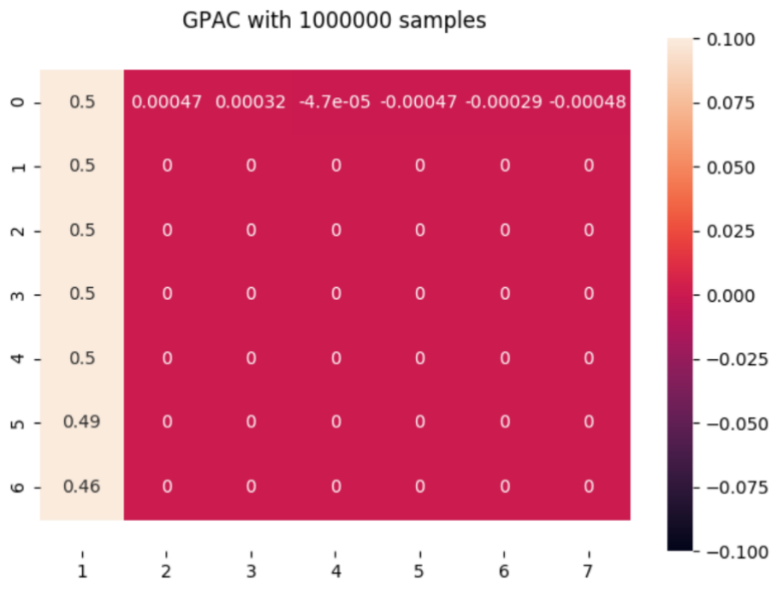
	1	2	3	4	5	6	7
0	0.49	0.0025	-0.0023	-0.037	0.052	-0.0017	-0.0076
1	0.5	0	0	-0.04	0.05	0	0
2	0.49	0	0	0	0	0	0
3	0.26	0.57	0	0	0	0	0
4	1.3	0	0	0	0	0	0
5	0.76	0	0	0	0	0	0
6	0.47	0	0	0	0	0	0

a.

b. There is an obvious pattern here of an ARMA (1,0) since the first column are values similar to the AR (1) coefficient –(-.5) in the process with mostly zeros everywhere to the right of this column which means an MA(0).

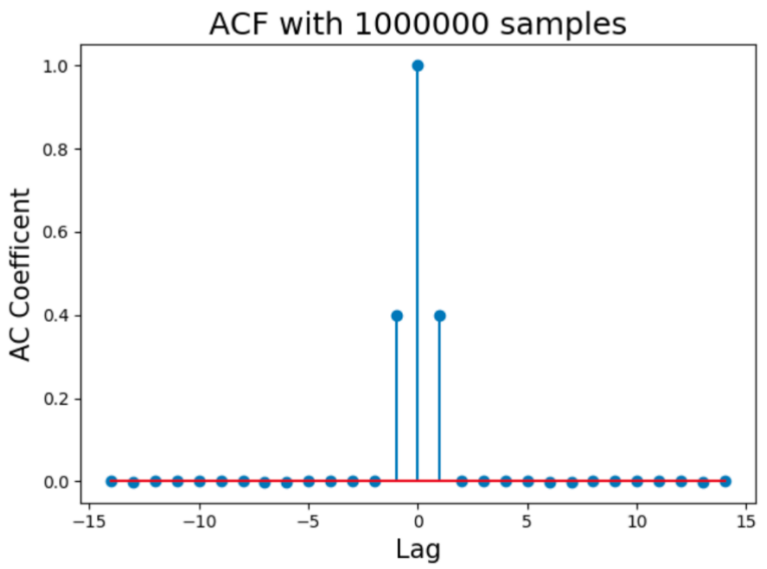
6. Increase sample sizes to 10,000 and 1,000,000 samples

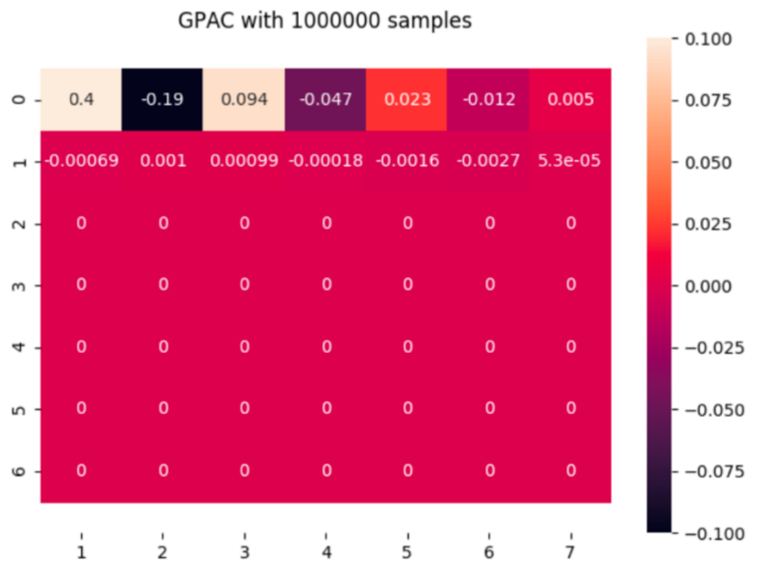




a. By increasing the sample sizes, the pattern becomes clearer and the values of the AR parameters are more similar to the actual parameters.

7. Repeat the process for various examples and estimate the order of the ARMA process.
   a. *ARMA(0,1) : y(t) = e(t) + 0.5e(t-1)*

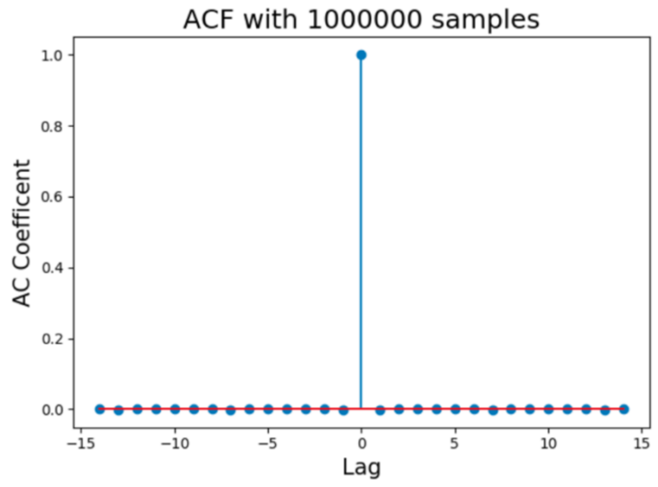


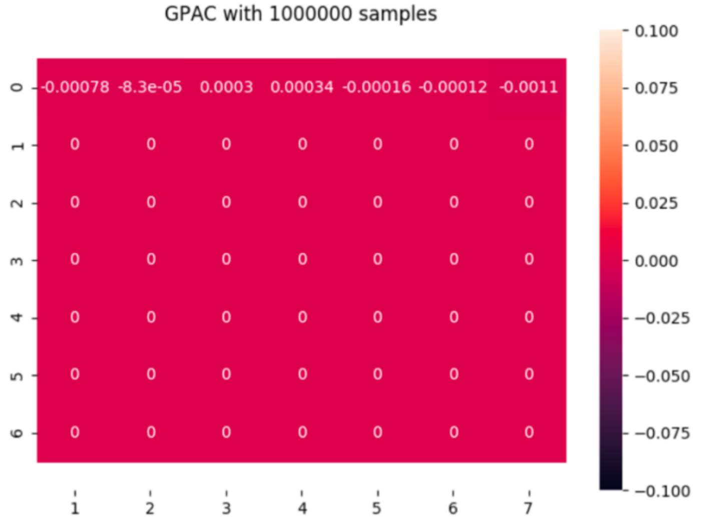


	1	2	3	4	5	6	7
0	0.4	-0.19	0.094	-0.047	0.023	-0.012	0.005
1	-0.00069	0.001	0.00099	-0.00018	-0.0016	-0.0027	5.3e-05
2	0	0	0	0	0	0	0
3	0	0	0	0	0	0	0
4	0	0	0	0	0	0	0
5	0	0	0	0	0	0	0
6	0	0	0	0	0	0	0

      i. Because the clear line of non-zero values at j=0 and almost zeros at j =1 this GPAC shows an ARMA(0,1).

b. ARMA(1,1) yt + 0.5y(t-1) = et + 0.5e(t-1)

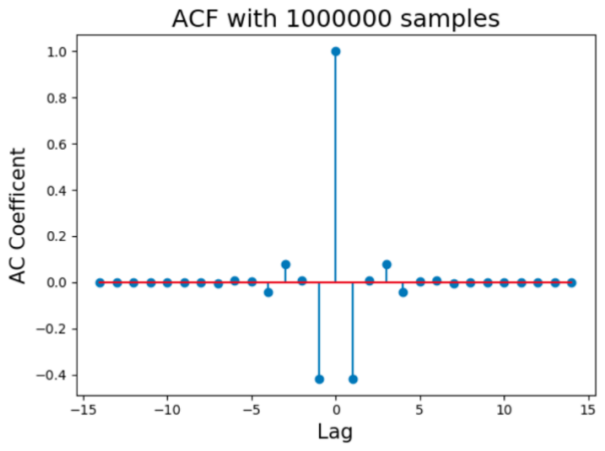
   i. 

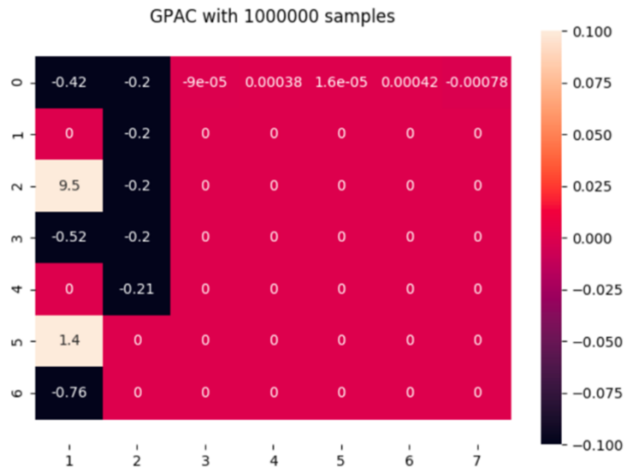
   ii. 

	1	2	3	4	5	6	7
0	-0.00078	-8.3e-05	0.0003	0.00034	-0.00016	-0.00012	-0.0011
1	0	0	0	0	0	0	0
2	0	0	0	0	0	0	0
3	0	0	0	0	0	0	0
4	0	0	0	0	0	0	0
5	0	0	0	0	0	0	0
6	0	0	0	0	0	0	0

   iii. ARMA(1,1) essentially becomes ARMA(0,0) because the AR and MA share the same root. Hence the GPAC correctly shows a AR(0) and MA(0) because the entire table is essentially zeros.

c. ARMA (2,0) : y(t) + 0.5y(t-1) + 0.2y(t-2) = e(t)

   i. 

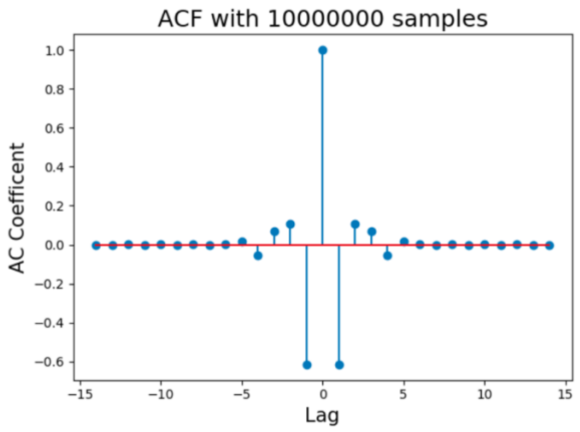


ii.

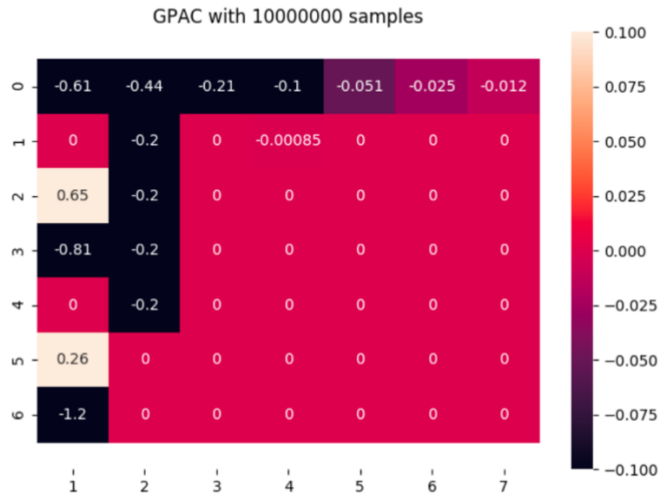
iii. There is a clear patter after at k=2 and j=0 demonstrating an ARMA(2,0) also that that at k =2 the column is -0.2 which is the negative coefficient of the second parameter of AR(2) process.

d. ARMA(2,1) : y(t) + 0.5y(t-1) + 0.2y(t-2) = e(t) - 0.5e(t-1)

i.

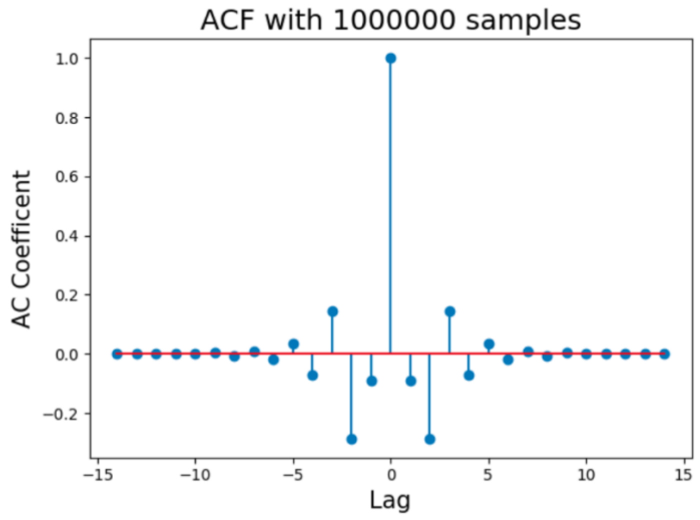


ii.

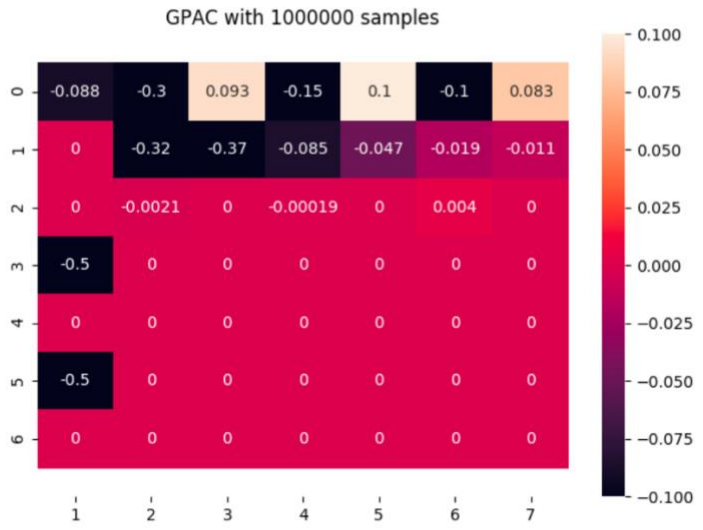


iii. There is a clear patter after at k=2 and j=1 demonstrating an ARMA(2,1) also that that at k =2 the column is -0.2 which is the negative coefficient of the parameter of AR(2) process.

e. ARMA(1,2) : y(t) + 0.5y(t-1) = e(t) + 0.5e(t-1) - 0.4e(t-2)



i.



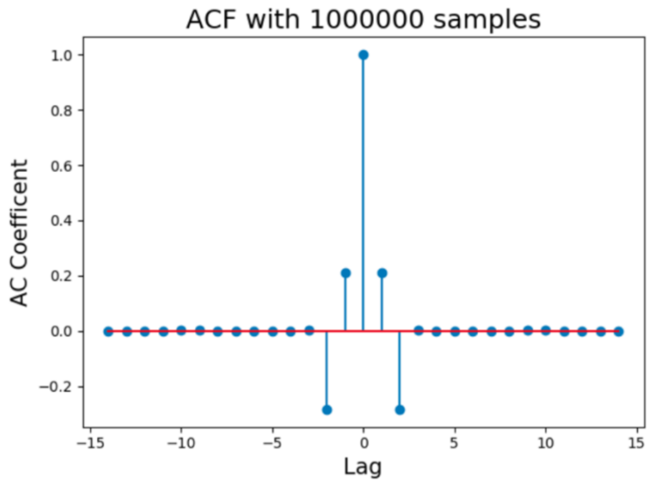
	1	2	3	4	5	6	7
0	-0.088	-0.3	0.093	-0.15	0.1	-0.1	0.083
1	0	-0.32	-0.37	-0.085	-0.047	-0.019	-0.011
2	0	-0.0021	0	-0.00019	0	0.004	0
3	-0.5	0	0	0	0	0	0
4	0	0	0	0	0	0	0
5	-0.5	0	0	0	0	0	0
6	0	0	0	0	0	0	0

ii.

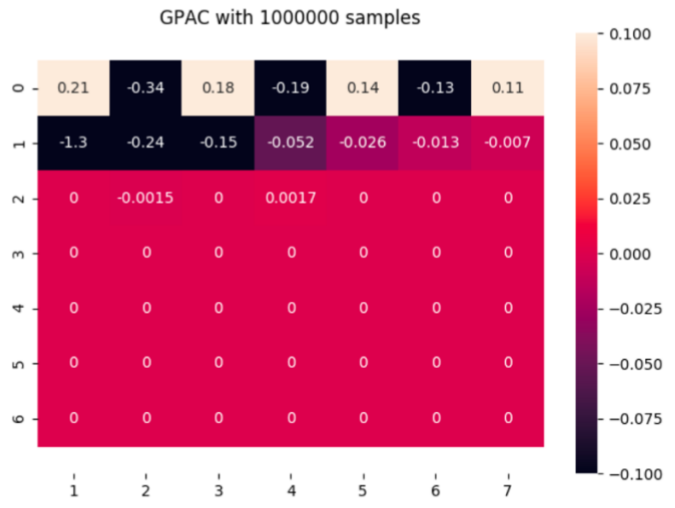
iii. There is a clear pattern at j =2 and k = 1 since afterwards all values are zeros. At k = 1 the distinct values are -.5 which is the coefficient of the AR process. The estimated ARMA order is (1,2).

f. ARMA(0,2) : y(t) = e(t)+ 0.5e(t-1) - 0.4e(t-2)

i.



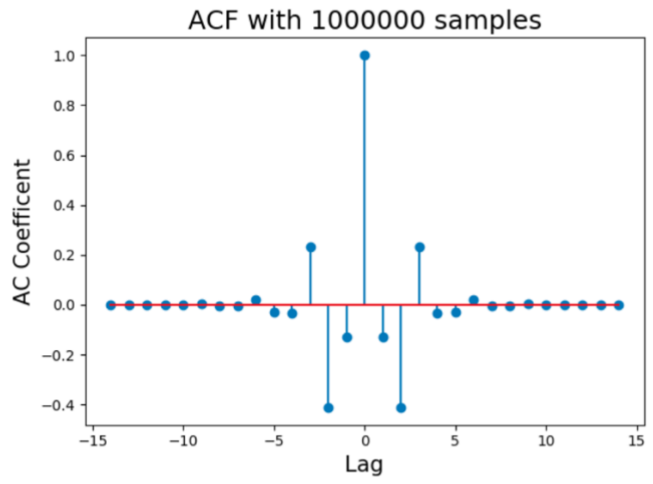
ii.

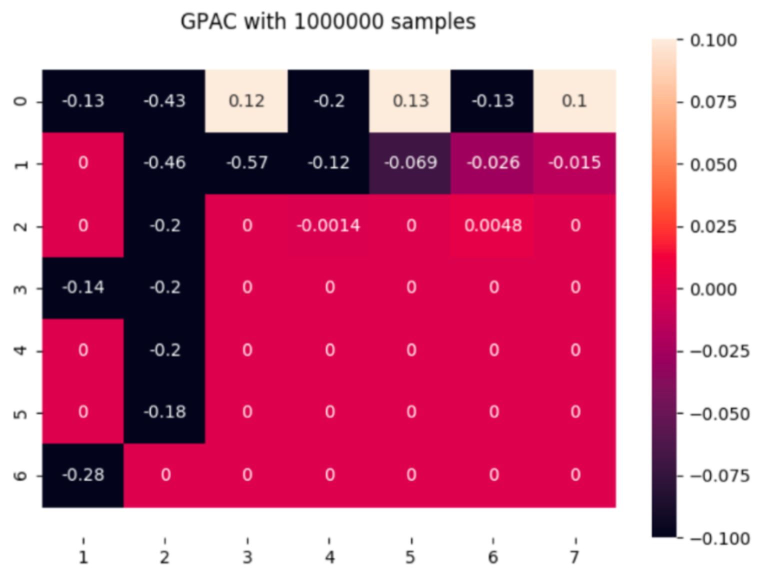


iii. There is a clear pattern at j =2 since afterwards all values are zeros. There is no clear pattern or value at k = 1. Hence the estimated order for this process is ARMA (0,2).

g. *ARMA(2,2):y(t)+0.5y(t-1)+0.2y(t-2)=e(t)+0.5e(t-1)- 0.4e(t-2)*

i.





	1	2	3	4	5	6	7
0	-0.13	-0.43	0.12	-0.2	0.13	-0.13	0.1
1	0	-0.46	-0.57	-0.12	-0.069	-0.026	-0.015
2	0	-0.2	0	-0.0014	0	0.0048	0
3	-0.14	-0.2	0	0	0	0	0
4	0	-0.2	0	0	0	0	0
5	0	-0.18	0	0	0	0	0
6	-0.28	0	0	0	0	0	0

ii.

iii. There is a clear pattern at j = 2 and k =2 for an order determination of ARMA(2,2). K =2 is filled with -0.2 which is the negative value of the second process of the AR process.

Conclusion:

The GPAC implementation results in a visual estimation of the order determination of a given ARMA process. As demonstrated with the early examples, the accuracy of the GPAC increases as sample size increased from 1,000 to 1,000,000. The easiest order determinations were for ARMA process that only have an MA process since the ACF of theses process become zero after the lag number exceeds the order. Furthermore, these processes show as bright rows equal to the MA order. For ARMA process with only an AR component, the GPAC is a bit more useful as its more difficult to determine the order strictly form the ACF. Instead with GPAC, AR orders are determined by the columns that become highlighted. The last column to the right is mainly filled with the negative value of the last AR parameter. Finally, the GPAC is most useful for ARMA processes with both AR and MA components as its incredibly difficult to determine the order bases off the ACF as it may also look like an AR process. The GPAC will highlight the rows and columns for the AR and MA processes.

Appendix:

```
#!/usr/bin/env python

# coding: utf-8


import numpy as np

import pandas as pp

from scipy import signal

import matplotlib.pyplot as plt

import seaborn as sns

import math


def ACF(data,lags):

    # convert input data into a numpy array

    data = np.array(data)

    # acf will store the autocorreltion coefficent at each lag interval

    # the first datapoint is always 1.0 since anything correlated with itsself is = 1

    acf = [1.0]

    # calculate the mean for the entire dataset

    y_bar = data.mean()

    print("The mean of this dataset is: ",y_bar)

    # subtract the mean from each observation

    yy_bar = data - y_bar

    # clacualte the total variance for the data set

    total_variance = sum(np.square(yy_bar))

    #print("The total variance for this dataset is: ", total_variance)

    # perform a forloop over the dataset with the desired number of lags

    # range is 1,lags b/c the first iteration calcualtes T1

    for i in range(1,lags):

        # first nparray is removing the last element each iteration

        yy_bar_bottom = yy_bar[:-i]

        # second nparray removes the first element each interation

        yy_bar_top = yy_bar[i:]

        # take the sum of of the product of each nparray each iteration

        yy = sum(yy_bar_top * yy_bar_bottom)

        # divide the sum by total variance and append to resulting acf list

        acf.append(yy/total_variance)
    return acf

def acf_plot(y,a):
    #y = y.tolist()
    y_rev = y[::-1]
    y_rev.extend(y[1:])
    print(len(y_rev))
    lb = -(math.floor(len(y_rev)/2))
    hb = -(lb-1)
    x = np.array(list(range(lb,hb)))
    figure = plt.stem(x,y_rev,use_line_collection=True)
    plt.xlabel('Lag', fontsize=15)
    plt.ylabel('AC Coefficent', fontsize=15)
    plt.title('ACF with {} samples'.format(a),fontsize=18)
    plt.show()

    #return y_rev

def GPAC(y,a):
    acf = ACF(y, 30)
    acf_plot(ACF(y, 15),a)
    # construct den matrix
    den = np.zeros([14, 7])
    for j in range(0, 14):
        for k in range(1, 8):
            den[j][k - 1] = acf[abs(j - k + 1)]

    # GPAC matrix
    phikk = np.zeros([7, 7])
    for j in range(0, 7):
        for k in range(0, 7):
            if k == 0:
                d = den[j][k]
                n = den[j + 1][k]
                phi = n / d
                if d < 0.001:
                    phi = 0
                phikk[j][k] = phi
            else:
                d = den[j:j + k + 1, :k + 1]
                # capture the den info for num
                n1 = den[j:j + k + 1, :k]
                # create j+k column
                n2 = np.array(acf[j + 1:j + k + 2])
                num = np.concatenate([n1, n2], axis=1)
                phi = (np.linalg.det(num)) / (np.linalg.det(d))
                dt = (np.linalg.det(d))
                if dt < 0.001:
                    phi = 0
                phikk[j][k] = phi

    # Plot table
    sns.heatmap(phikk, annot=True, vmax=.1, vmin=-.1)
    b, t = plt.ylim()  # discover the values for bottom and top
    b += 0.5  # Add 0.5 to the bottom
    t -= 0.5  # Subtract 0.5 from the top
    plt.ylim(b, t)  # update the ylim(bottom, top) values
    plt.title("GPAC with {} samples".format(a))
    plt.xticks(np.arange(0.5, len(phikk), 1), np.arange(1, 8, 1))
    plt.show()

    # Plot table
    sns.heatmap(phikk,annot=True,vmax=.1,vmin=-.1)
    b, t = plt.ylim() # discover the values for bottom and top
    b += 0.5 # Add 0.5 to the bottom
    t -= 0.5 # Subtract 0.5 from the top
    plt.ylim(b, t) # update the ylim(bottom, top) values
    plt.title("GPAC with {} samples".format(a))
    plt.xticks(np.arange(0.5,len(phikk),1),np.arange(1,8,1))
    plt.show()

def ARMA(N,AR,MA,na,nb):
    # Setup the error signal with N # of samples
    mean = 0
    std = 1
    np.random.seed(42)
    e = std * np.random.randn(N) + mean

    print("\nARMA with AR order {0} and MA order {1}".format(AR,MA))
    num = nb
    print("MA",num)
    den = na
    print("AR",den)

    system = (num,den,1)
    x,y = signal.dlsim(system,e)
    plt.plot(y)
    plt.show()
    return y

def ARMA_gen():
    a = int(input("\nEnter the numbers of samples :"))
    b = int(input("\nEnter the order # of the AR process :"))
    c = int(input("\nEnter the order # of the MA process :"))
    print("\n The program will ask you to enter each parameter indivdually")
    na = np.zeros(b)
    for i in range(len(na)):
        na[i]= (float(input("\nEnter paramter {0} of AR({1}):".format(i+1,b))))
    if b < c:
        x = np.zeros(c-b)
        na = np.array(list(na) + list(x))
    d = np.array([1] + list(na))

    nb = np.zeros(c)
    for  i in range(len(nb)):
        nb[i] = (float(input("\nEnter parameter {0} of MA({1}):".format(i+1,c))))
    if c < b:
        z = np.zeros(b-c)
        nb = np.array(list(nb) + list(z))
    e = np.array([1] + list(nb))
    results = ARMA(a,b,c,d,e)

    GPAC(results,a)

    #return results

print("\nARMA(1,0) y(t)-0.5y(t-1)=e(t)")
print("1000 Samples")
ARMA_gen()

print("\nARMA(1,0) y(t)-0.5y(t-1)=e(t)")
print("10000 Samples")
ARMA_gen()

print("\nARMA(1,0) y(t)-0.5y(t-1)=e(t)")
print("1000000 Samples")
ARMA_gen()

print("\nARMA(0,1) yt = e(t) + 0.5(t-1)")
print("1000000 Samples")
ARMA_gen()

print("\n# ARMA(1,1) yt + 0.5y(t-1) = et + 0.5e(t-1)")
print("1000000 Samples")
ARMA_gen()

print("\nARMA(2,0) : y(t) + 0.5y(t-1) + 0.2y(t-2) = e(t)")
print("1000000 Samples")
ARMA_gen()

print("\nARMA(2,1) : y(t) + 0.5y(t-1) + 0.2y(t-2) = e(t) - 0.5e(t-1)")
print("1000000 Samples")
ARMA_gen()

print("\nARMA(1,2) : y(t) + 0.5y(t-1)  = e(t) + 0.5e(t-1) - 0.4e(t-2)")
print("1000000 Samples")
ARMA_gen()
```

```
print("\nARMA(0,2) : y(t) = e(t)+ 0.5e(t-1) - 0.4e(t-2)")
print("1000000 Samples")
ARMA_gen()

print("\nARMA(2,2) :y(t)+0.5y(t-1)+0.2y(t-2)=e(t)+0.5e(t-1) - 0.4e(t-2)")
print("1000000 Samples")
ARMA_gen()
```

References: