

# Week2\_Project

January 14, 2022

```
In [1]: import math
import warnings
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
import statsmodels.api as sm

sns.set_style("darkgrid")
warnings.filterwarnings('ignore')
```

## 0.1 Problem 1

Compare the conditional distribution of the Multivariate Normal, to the OLS equations. Are these values the same? Why?

Use the data in problem1.csv to prove your answer empirically.

### 0.1.1 Answer

Suppose there are two variables  $x$  and  $y$ . We can either deem  $x$  and  $y$  as jointly bivariate normal, or use  $x$  to explain  $y$  based on OLS. Under both cases,  $y$  is a normally distributed variable given  $x$  as a condition. Denote these distributions as  $N(\mu_1, \sigma_1)$  and  $N(\mu_2, \sigma_2)$ .

We can calculate the unbiased estimator of  $E(y|x)$  and  $Var(y|x)$  as parameters of  $y$ 's distribution under these two cases and compare their difference.

**Case 1. Conditional Distribution of the Multivariate Normal** We assume that the  $(x, y)$  data-points in problem1.csv are independent variables under the same bivariate normal distribution. The conditional distribution of  $y$  given knowledge of  $x$  is a normal distribution  $N(\mu, \sigma)$  with:

$$\mu = \mu_y + \frac{\sigma_{xy}}{\sigma_x^2}(x - \mu_x)$$
$$\sigma = \sigma_y - \frac{\sigma_{xy}^2}{\sigma_x}$$

where  $\mu_y, \mu_x$  are mean value of  $y$  and  $x$ ,  $\sigma_y, \sigma_x, \sigma_{xy}$  are variance of  $y$  and  $x$  and covariance between them.

First, we need to calculate  $E(x)$ ,  $E(y)$ , and the covariance matrix. This can be done by calling APIs of Pandas.

```
In [2]: data1 = pd.read_csv("problem1.csv").sort_values('x')

# mean
mean = data1.mean()
mean_x, mean_y = mean.loc['x'], mean.loc['y']
print(f"\mu_1 = {mean_x}, \mu_2 = {mean_y}\n")

# covariance
print("Covariance Matrix of x, y:")
cov = data1.cov()
print(cov)
var_x, var_y, cov_xy = cov.loc['x', 'x'], cov.loc['y', 'y'], cov.loc['x', 'y']

\mu_1 = -0.14054562796809722, \mu_2 = -0.0222771880352644
```

Covariance Matrix of x, y:

	x	y
x	1.315195	0.562908
y	0.562908	0.898883

We want to make sure that the variance estimator provided by pandas is unbiased. That is to say, the sum of squared deviation is divided by  $n-1$  but not  $n$ .

```
In [3]: n = len(data1.x)
my_var_x = np.square(data1.x - np.mean(data1.x)).sum() / (n-1)
var_x_unbiased = math.isclose(my_var_x, var_x, rel_tol=0.000001)
if var_x_unbiased:
    print("Var(x) provided by Pandas is unbiased.")
else:
    print("Var(x) provided by Pandas is biased.")
```

Var(x) provided by Pandas is unbiased.

Then we define functions that calculate  $\mu$  and  $\sigma$ .

```
In [4]: def conditional_mean(x2, mu1, mu2, var2, cov12):
    """
    calculate conditional mean of x1 given the knowledge of x2,
    if x1 and x2 follow a bivariate normal distribution.

    params:
        - x2
        - mu1: mean of x1
        - mu2: mean of x2.
        - var2: variance of x2
        - cov12: covariance of x1 and x2
    """
```

```

    return mu1 + cov12 / var2 * (x2 - mu2)

def conditional_var(var1, var2, cov12):
    """
    calculate conditional variance of x1 given the knowledge of x2,
    if x1 and x2 follow a bivariate normal distribution.

    params:
        - var1: variance of x1
        - var2: variance of x2
        - cov12: covariance of x1 and x2
    """
    return var1 - cov12**2 / var2

cond_mean_y = conditional_mean(data1.x, mean_y, mean_x, var_x, cov_xy)
cond_var_y = conditional_var(var_y, var_x, cov_xy)

```

**Case2. OLS** We assume that

$$y_i = \beta_0 + \beta_1 x_i + \epsilon_i$$

and all assumptions of OLS are satisfied. Then the unbiased estimator for  $\mu_2$  is  $\hat{\beta}_0 + \hat{\beta}_1 x$ . Since  $Var(y|x) = Var(\beta_1 x + \epsilon|x) = Var(\epsilon)$ , the unbiased estimator for  $\sigma_2$  is  $\frac{SSR}{n-2}$ .

```

In [5]: results1 = sm.OLS(data1.y, sm.add_constant(data1.x)).fit()
        ols_mean_y = results1.fittedvalues
        ols_var_y = results1.ssr / results1.df_resid

```

**Conclusion** We can draw y's conditional mean calculated in case1 and y's OLS fitted value on the same plot.

These two lines overlap each other, which means  $\mu_1 = \mu_2$ . This is because the error term in OLS equation has zero mean value.

```

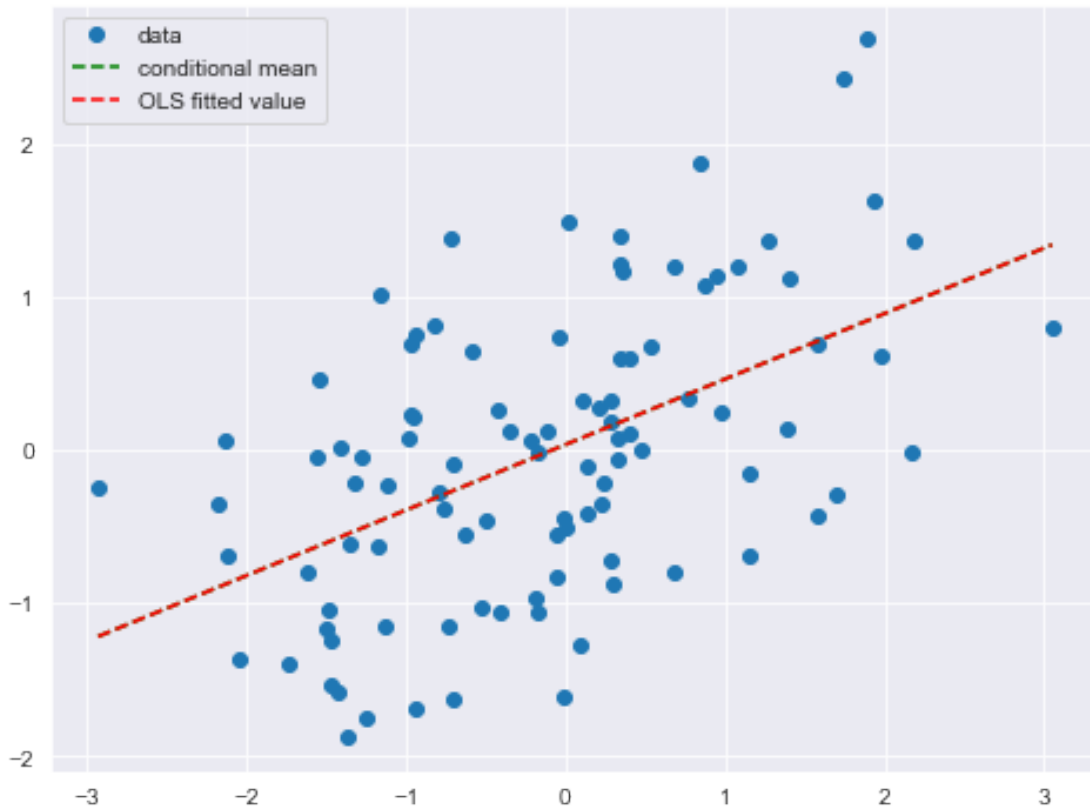
In [6]: fig, ax = plt.subplots(figsize=(8,6))
        ax.plot(data1.x, data1.y, 'o', label="data")
        ax.plot(data1.x, cond_mean_y, 'g--', label="conditional mean")
        ax.plot(data1.x, results1.fittedvalues, 'r--', label="OLS fitted value")
        ax.legend(loc='best')

```

```

Out[6]: <matplotlib.legend.Legend at 0x1b1c7c30>

```



However  $\sigma_1$  is different from  $\sigma_2$ .

```
In [7]: print(f"\sigma_1 = {cond_var_y}, \sigma_2 = {ols_var_y}")
```

```
\sigma_1 = 0.6579563030192093, \sigma_2 = 0.6646701428459357
```

## 0.2 Problem 2

Fit the data in problem2.csv using OLS and calculate the error vector. Look at it's distribution.

How well does it fit the assumption of normally distributed errors?

Fit the data using MLE given the assumption of normality. Then fit the MLE using the assumption of a T distribution of the errors. Which is the best fit?

What are the fitted parameters of each and how do they compare? What does this tell us about the breaking of the normality assumption in regards to expected values in this case?

```
In [8]: import scipy.stats as stats
import scipy.optimize as optimize
```

### 0.2.1 Answer

We assume that

$$y_i = b_0 + b_1 x_i + \epsilon_i$$

and all assumptions of OLS are satisfied, then fit the data in problem2.csv using OLS and calculate the error vector as follows.

The summary shows that the estimated value for  $b_0$  and  $b_1$  is 0.1198 and 0.6052.

```
In [9]: data2 = pd.read_csv("problem2.csv").sort_values('x')
        results = sm.OLS(data2.y, sm.add_constant(data2.x)).fit()
        error = data2.y - results.fittedvalues

        results.summary()
```

```
Out[9]: <class 'statsmodels.iolib.summary.Summary'>
        """
```

```

                                OLS Regression Results
=====
Dep. Variable:                  y      R-squared:                0.195
Model:                            OLS      Adj. R-squared:         0.186
Method:                 Least Squares      F-statistic:             23.68
Date:                Fri, 14 Jan 2022      Prob (F-statistic):       4.34e-06
Time:                        03:06:57      Log-Likelihood:          -159.99
No. Observations:                100      AIC:                     324.0
Df Residuals:                     98      BIC:                     329.2
Df Model:                           1
Covariance Type:                nonrobust
=====

```

	coef	std err	t	P> t	[0.025	0.975]
const	0.1198	0.121	0.990	0.325	-0.120	0.360
x	0.6052	0.124	4.867	0.000	0.358	0.852

```

=====
Omnibus:                 14.146      Durbin-Watson:           1.931
Prob(Omnibus):            0.001      Jarque-Bera (JB):        43.673
Skew:                     -0.267      Prob(JB):                3.28e-10
Kurtosis:                  6.193      Cond. No.                 1.03
=====

```

Notes:

```
[1] Standard Errors assume that the covariance matrix of the errors is correctly specified
"""
```

**1. Are errors normally distributed?** The errors are supposed to be normally distributed, whose the mean value is 0 and variance can be estimated by  $\frac{SSR}{n-2}$ .

We use these parameters to generate a series of random normal variables and compare its distribution to that of the errors.

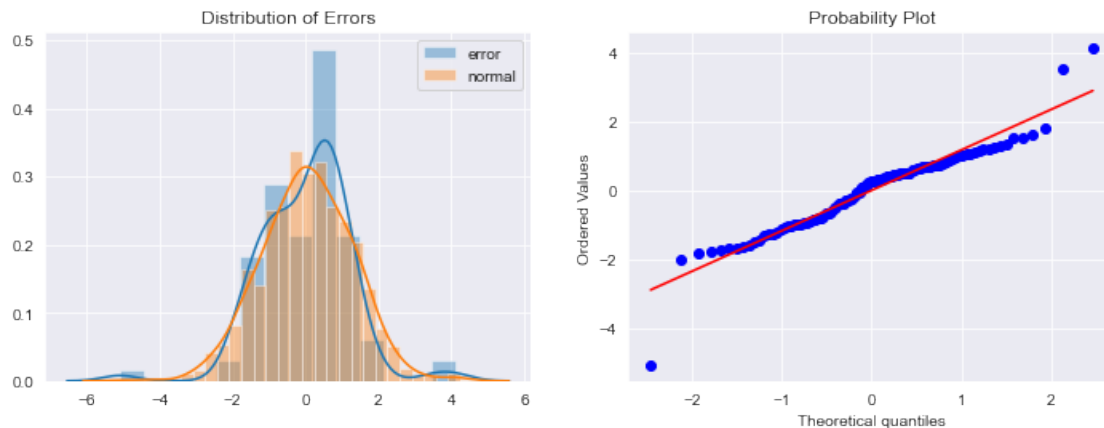
It seems that the error vector has more values in  $[0, 1]$  than it should.

We can also draw the quantile-quantile plot for the errors. If they are normally distributed, the points on the plot will form a line that's roughly straight. In this case, I think it's not.

```
In [10]: sigma = np.sqrt(results.ssr / results.df_resid)
         normal_dist = np.random.normal(loc=0.0, scale=sigma, size=1000)
```

```
In [11]: fig, axes = plt.subplots(1, 2, figsize=(12, 4))
pic0 = sns.distplot(error, ax=axes[0], label="error")
pic0 = sns.distplot(normal_dist, ax=axes[0], label="normal")
axes[0].set_title("Distribution of Errors")
axes[0].legend()

pic1 = stats.probplot(error, dist=stats.norm, plot=axes[1])
```



In addition, we can use the Shapiro-Wilks method to examine whether errors are normally distributed.

The test suggests that we should reject the hypothesis that errors are normally distributed.

```
In [12]: # Shapiro-Wilks test
print("Null Hypothesis: errors are normally distributed.")
alpha = 0.05
test_stat, p = stats.shapiro(error)
print(f"w-test statistic = {test_stat}, pvalue = {p}")

if p < alpha:
    print("Hypothesis rejected. Errors are probably not normally distributed.")
else:
    print("Hypothesis not rejected. Errors are probably normally distributed.")
```

Null Hypothesis: errors are normally distributed.

w-test statistic = 0.9383853077888489, pvalue = 0.00015388526662718505

Hypothesis rejected. Errors are probably not normally distributed.

**2. Fit the data using MLE** In case1 ,we fit the data under the assumption of normality, then fit the data under the assumption of t distribution of errors in case2.

They generated different results for parameters (in case 1, the results are the same as that produced by OLS).

Even though errors may not be normally distributed, SSE in case 1 is still lower than that in case2.

The differences indicates that breaking of the normality assumption will result in different estimated values of parameters and thus a difference in the expected value of  $y$ .

```
In [13]: class RegressionMLEstimator:
        """
        An abstract class to estimate  $b_0$ ,  $b_1$  in regression

        
$$y = b_0 + b_1x + e$$


        using the MLE method.
        The distribution of  $e$  is to be defined.
        """

        def __init__(self, y, x):
            self.y = y
            self.x = x

            self.b0_hat = None
            self.b1_hat = None

        def residual(self, b0, b1):
            return self.y - b0 - b1*self.x

        @staticmethod
        def logpdf_sum(dist, x):
            """
            The log-likelihood of variable  $x$  given its distribution.

            params:
            - dist: a distribution object provided by scipy.stats
            - x: random variable under dist
            """
            return np.log(dist.pdf(x)).sum()

        def ll(self, args):
            """
            The negated log likelihood function.
            It negated so as to be minimized by scipy.optimize,
            which achieves the same effect as maximizing the log likelihood function.
            """
            raise NotImplementedError

        def estimate(self, **kwargs):
            result = optimize.minimize(self.ll, **kwargs).x
            self.b0_hat, self.b1_hat = result[0], result[1]

        def sse(self):
            """Calculate the sum of squared errors"""
```

```

        rsd = self.residual(self.b0_hat, self.b1_hat)
        return np.square(rsd).sum()

    def fitted_value(self):
        return self.b0_hat + self.b1_hat * self.x

class NormalEstimator(RegressionMLEstimator):
    """
    Estimate b0, b1 in regression

     $y = b0 + b1*x + e$ 

    using the MLE method, where e is normally distributed.
    """

    def ll(self, args):
        b0, b1, sigma = args
        dist = stats.norm(0, sigma)
        e = self.residual(b0, b1)
        return -self.logpdf_sum(dist, e)

class TEstimator(RegressionMLEstimator):
    """
    Estimate b0, b1 in regression

     $y = b0 + b1*x + e$ 

    using the MLE method, where e follows t distribution.
    """

    def ll(self, args):
        b0, b1 = args
        df_residual = len(self.y) - 2
        dist = stats.t(df_residual)
        e = self.residual(b0, b1)
        return -self.logpdf_sum(dist, e)

```

```

In [14]: norm_estimator = NormalEstimator(data2.y, data2.x)
norm_estimator.estimate(x0=(0, 1, 1), constraints={"type":"ineq", "fun":lambda x: x[2]})
norm_sse = norm_estimator.sse()
norm_fittedvalue = norm_estimator.fitted_value()
print(f"If errors follow a normal distribution, then")
print(f"b0 = {norm_estimator.b0_hat}, b1 = {norm_estimator.b1_hat}")
print(f"sse = {norm_sse}")

```

If errors follow a normal distribution, then  
b0 = 0.1198446690635319, b1 = 0.6051913741196576



```
sse = 143.6148485649267
```

```
In [15]: t_estimator = TEstimator(data2.y, data2.x)
t_estimator.estimate(x0=(0, 1))
t_sse = t_estimator.sse()
t_fittedvalue = t_estimator.fitted_value()
print(f"If errors follow a t distribution, then")
print(f"b0 = {t_estimator.b0_hat}, b1 = {t_estimator.b1_hat}")
print(f"sse = {t_sse}")
```

```
If errors follow a t distribution, then
b0 = 0.12325309763059471, b1 = 0.5951244491011526
sse = 143.62564595376156
```

```
In [16]: if t_sse < norm_sse:
    print("TEstimator gives the best fit.")
else:
    print("NormalEstimator gives the best fit.")
```

```
NormalEstimator gives the best fit.
```

### 0.3 Problem 3

Simulate AR(1) through AR(3) and MA(1) through MA(3) processes. Compare their ACF and PACF graphs. How do the graphs help us to identify the type and order of each process?

```
In [17]: from functools import partial
from statsmodels.tsa.arima_process import ArmaProcess
from statsmodels.graphics.tsaplots import plot_acf, plot_pacf

In [18]: def simulate_ARMA_process(ar, ma, nsample=300):
    """Simulate an ARMA process."""
    ARMA_object = ArmaProcess(ar=[1] + [coef * -1 for coef in ar],
                              ma=[1] + ma)
    simulated_data = ARMA_object.generate_sample(nsample=nsample)
    return simulated_data

simulate_AR_process = partial(simulate_ARMA_process, ma=[])
simulate_MA_process = partial(simulate_ARMA_process, ar=[])

def plot_processes(processes, titles):
    n_prcs = len(processes)
    fig, axes = plt.subplots(n_prcs, 3, figsize=(5*n_prcs, 3*n_prcs))
```

```

alpha = 0.05
lags = 30
for i, process in enumerate(processes):
    axes[i][0].plot(process)
    plot_acf(process, alpha=alpha, lags=lags, ax=axes[i][1])
    plot_pacf(process, alpha=alpha, lags=lags, ax=axes[i][2])

    # plot labels and titles
    axes[i][0].set_ylabel(ylabel=titles[i],labelpad=0.6,loc='center')
    if i == 0:
        axes[i][0].set_title('data')
        axes[i][1].set_title('ACF')
        axes[i][2].set_title('PACF')

```

We simulate AR processes

$$y_t = 0.3y_{t-1} + \epsilon_t$$

$$y_t = 0.3y_{t-1} + 0.3y_{t-2} + \epsilon_t$$

$$y_t = 0.3y_{t-1} + 0.3y_{t-2} + 0.3y_{t-3} + \epsilon_t$$

and plot their ACF and PACF.

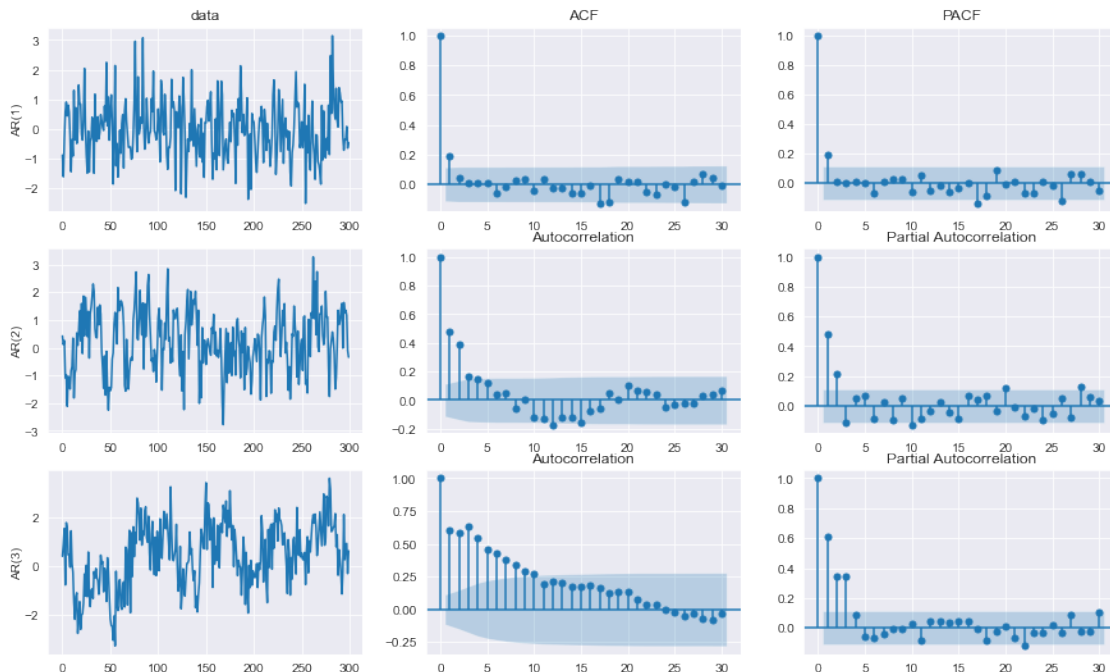
We can discover that for the AR(1) process, only the lag-1 PACF is significantly different from 0; for the AR(2) process, lag-1 and lag-2 PACF is significantly different from 0; for the AR(3) process, lag-1, lag-2 and lag-3 PACF is significantly different from 0. However, there is no obvious pattern with ACFs.

(lag-0 ACFs and PACFs are not considered since they are always 1.)

```

In [19]: ar_list = [[0.3], [0.3]*2, [0.3]*3]
         ar_processes = [simulate_AR_process(ar=ar_coeffs) for ar_coeffs in ar_list]
         plot_processes(ar_processes, ['AR(1)', 'AR(2)', 'AR(3)'])

```



Then we simulate MA processes

$$y_t = \epsilon_t + 0.3\epsilon_{t-1}$$

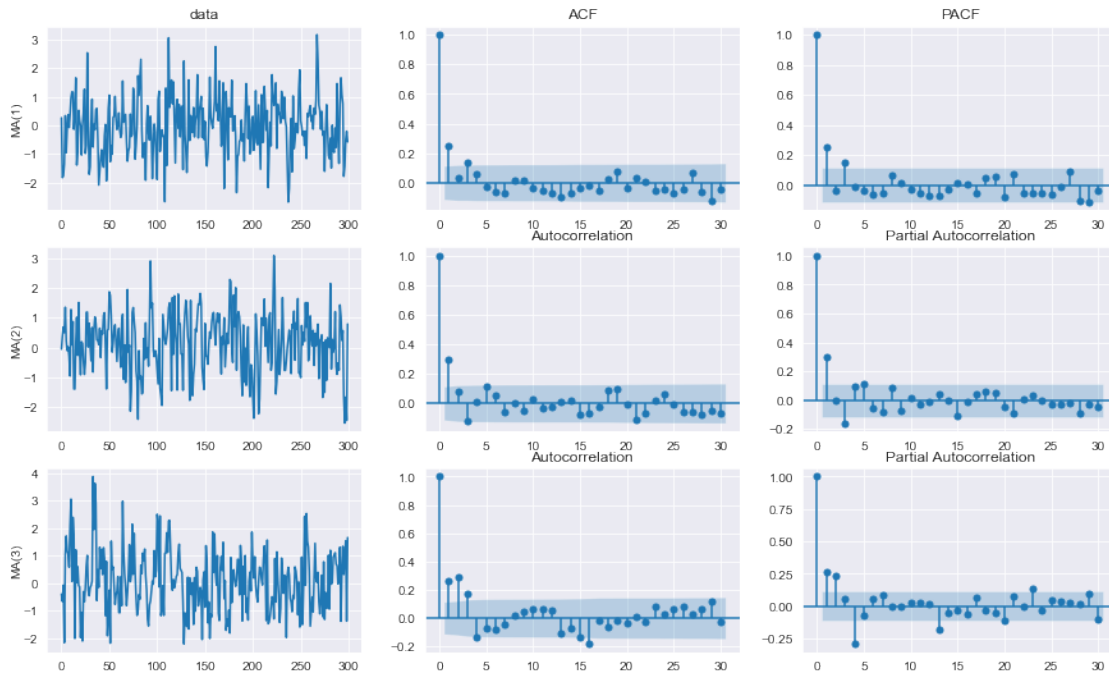
$$y_t = \epsilon_t + 0.3\epsilon_{t-1} + 0.3\epsilon_{t-2}$$

$$y_t = \epsilon_t + 0.3\epsilon_{t-1} + 0.3\epsilon_{t-2} + 0.3\epsilon_{t-3}$$

and plot their ACF and PACF.

We can discover that for the MA(1) process, only the lag-1 ACF is significantly different from 0; for the MA(2) process, lag-1 and lag-2 ACF is significantly different from 0; for the MA(3) process, lag-1, lag-2 and lag-3 ACF is significantly different from 0.

```
In [20]: ma_list = [[0.3], [0.3]*2, [0.3]*3]
ma_processes = [simulate_MA_process(ma=ma_coeffs) for ma_coeffs in ma_list]
plot_processes(ma_processes, ['MA(1)', 'MA(2)', 'MA(3)'])
```



As a result, we can use ACF graphs and PACF graphs to identify the type and order of each process by observing the number of lags they remain significant.

For example, if we find that only lag-1 to lag-4 PACF of a process is significantly different from 0, but the ACF remains significant for a large number of lags, we may consider it to be an AR(4). For another, if we find that only lag-1 to lag-3 ACF of a process is significantly different from 0, but no obvious pattern in PACF, we may think it to be a MA(3).