Consequences of Autocorrelation for OLS regression

Andrija Djurovic[*]

[*]www.linkedin.com/in/andrija-djurovic

In an ideal regression model, it is assumed that the residuals are independent of one another, indicating the absence of any systematic pattern or correlation between observations. When there is a serial correlation among the residuals in OLS regression, we refer to it as an autocorrelation problem. An acknowledged consequence of autocorrelation is the generation of inefficient parameter estimates, thereby compromising the validity of statistical inference. In addition to this commonly accepted consequence, some practitioners argue that the presence of autocorrelated errors suggests model misspecification, omitted variable bias, or an incorrect functional form.

The following example illustrates a framework for exploring the implications of autocorrelation in OLS regression residuals. It is important to note that this provided example should not be construed as a broad generalization of autocorrelation consequences but rather as a foundation for tailoring the process to meet specific requirements and address unique use cases. Towards the conclusion of the exercise, potential adjustments to the presented design are mentioned.

To commence, let's establish the simulation parameters and the data-generating process in `R`.

```r
library(lmtest)
library(sandwich)

#simulation parameters
n <- 45              #number of observations
iv <- 2              #number of independent variables

#simulate the x
set.seed(123)
x <- replicate(n = iv,
               expr = rnorm(n = n)
               )
x <- data.frame(x)
names(x) <- paste0("x", 1:iv)

#true regression parameters
intercept <- 0
beta1 <- 0.50
beta2 <- -0.45

#simulate the target
y <- intercept + beta1*x$x1 + beta2*x$x2

#store the inputs into data frame
db <- data.frame(y = y, x)
```

Based on the provided inputs, we will conduct a simulation examining the potential bias resulting from correlated error terms. We will also compare the power based on the traditional p-value with one calculated using the correction of HAC (Heteroskedasticity-and-Autocorrelation-Consistent) standard errors. Additional simulations' assumptions include a significance level (`alpha`) set at 5% and an autoregressive coefficient (`phi`) fixed at 0.70.

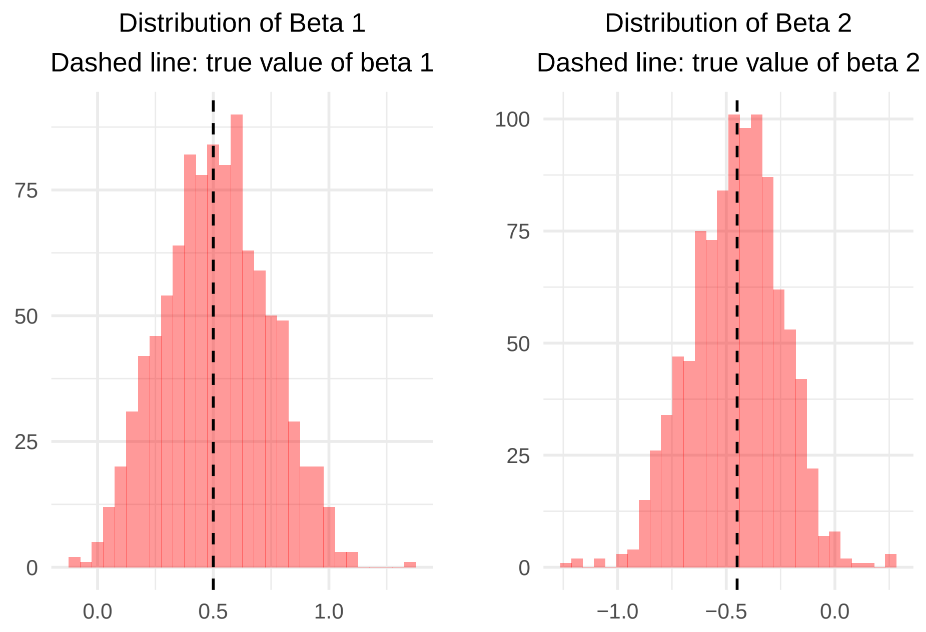
```r
#-------------------------run the simulations-------------------------#
#mc parameters
alpha <- 0.05        #significance level
phi <- 0.70          #autocorrelation coefficient
B <- 1000            #number of mc simulations

res <- data.frame(b1 = rep(NA, B),
                  b2 = rep(NA, B),
                  b1.p = rep(NA, B),
                  b2.p = rep(NA, B),
                  b1.p.hac = rep(NA, B),
                  b2.p.hac = rep(NA, B))

for   (i in 1:B) {
      #random seed
      set.seed(i*3)

      #simulate the (centered) correlated errors
      error.i <- c(arima.sim(model = list(ar = phi),
                             n = n,
                             sd = 1))
      error.i <- error.i - mean(error.i)

      #simulate the target
      db$y.sim <- db$y + error.i

      #run the regression
      lr <- lm(formula = y.sim ~ x1 + x2,
               data = db)
      lr.s <- summary(lr)

      #extract the estimates and p-values
      est.p.i <-  c(lr.s$coefficients[-1, c(1, 4)])
      #store the results
      res[i, 1:4] <-  est.p.i

      #extract hac p-values
      hac.p <- coeftest(x = lr,
                        vcov. = NeweyWest(x = lr,
                                          lag = 1,
                                          prewhite = FALSE)
                        )

      #store the results
      res[i, 5:6] <- unname(hac.p[-1, 4])

      }
```

Following the simulation, let's plot the distribution of estimated betas alongside their true values. Furthermore, we perform a comparison of statistical power between two different approaches to calculate p-values.

```r
#confidence intervals
ci.bs <- sapply(X = res[, c("b1", "b2")],
                FUN = function(x) {
                      quantile(x = x,
                               probs = c(0.025, 0.50, 0.975))
                      }
                )
ci.bs
```

```
##               b1          b2
## 2.5%  0.08698038 -0.85764567
## 50%   0.51242020 -0.44694846
## 97.5% 0.95819140 -0.08816721
```

```r
#histogram of betas
```



```r
#power calculation
p.bs <- sapply(X = res[, -c(1, 2)],
               FUN = function(x) mean(x < alpha))
p.bs
```

```
##     b1.p     b2.p b1.p.hac b2.p.hac
##    0.658    0.463    0.676    0.566
```

We will export the data generated in the above simulation to replicate the simulation in `Python`.

```r
#export the db to replicate the simulation in python
write.csv(x = db[, -ncol(db)],
          file = "db.csv",
          row.names = FALSE)
```

The following code snippet demonstrates the `Python` implementation of the same exercise.

```python
import pandas as pd
import numpy as np
import statsmodels.api as sm
import statsmodels.formula.api as smf

#import db
db = pd.read_csv("db.csv")

#------------------------run the simulations------------------------#
```

```python
#mc parameters
n = db.shape[0]      #number of observations
alpha = 0.05         #significance level
phi = 0.70           #autocorrelation coefficient
B = 10000            #number of mc simulations

columns = ["b1", "b2", "b1.p", "b2.p", "b1.p.hac", "b2.p.hac"]
res = pd.DataFrame(index = range(1, B + 1),
                   columns = columns)

#arma simulation ar input
arparams = np.array([phi])
ar = np.r_[1, -arparams]

for i in range(1, B + 1):

    #random seed
    np.random.seed(i * 3)

    #simulate (centered) correlated errors
    error_i = sm.tsa.ArmaProcess(ar = ar).generate_sample(nsample = n)
    error_i = error_i - np.mean(error_i)

    #simulate the target
    db["y_sim"] = db["y"] + error_i

    #run the regression
    lr = smf.ols(formula = "y_sim ~ x1 + x2",
                 data = db).fit()

    #extract estimates and p-values
    est_p_i = lr.params.values[1:]
    p_values_i = lr.pvalues.values[1:]
    est_p = np.r_[est_p_i, p_values_i]

    #store the results
    res.loc[i, ["b1", "b2", "b1.p", "b2.p"]] = np.r_[est_p_i, p_values_i]

    #extract hac p-values
    hac = lr.get_robustcov_results(cov_type = "HAC", maxlags = 1)
    res.loc[i, ["b1.p.hac", "b2.p.hac"]] = hac.pvalues[1:]



#confidence intervals
ci_bs = res[["b1", "b2"]].apply(lambda x:
                                np.percentile(a = x,
                                              q = [alpha*100/2,
                                                   50,
                                                   (1 - alpha/2)*100]))
ci_bs
```

```
##          b1        b2
## 0  0.058771 -0.861513
## 1  0.499371 -0.448229
## 2  0.942962 -0.036201
```

```
#power calculation
p_bs = (res[["b1.p", "b2.p", "b1.p.hac", "b2.p.hac"]] < alpha).mean()
p_bs
```

```
## b1.p        0.6463
## b2.p        0.4807
## b1.p.hac    0.6658
## b2.p.hac    0.5762
## dtype: float64
```

While the above simulations affirm the primary understanding of the implications of serially correlated errors, they should not be regarded as a universal generalization for all designs. In this context, readers are encouraged to explore additional designs, such as different values of the autocorrelation coefficient, incorporating lag of the dependent variable, experimenting with sample sizes for consistency checks, testing alternative methods for correcting autocorrelation, or investigating scenarios where HAC errors may surpass traditional ones. Additionally, one can explore combining autocorrelation errors with endogeneity problems.