Topics in Time Series Analysis: Assignment 9

Lucas Cruz Fernandez

1544674, lcruzfer@mail.uni-mannheim.de

24th of May 2020

Q17

Data generation

The data we want to generate to use it in our replication study is modeled in the following way:

$$
\begin{aligned}
y_t &= (1-L)^{-d} x_t \\
&= \sum_{j=1}^{\infty} \psi_j x_{t-j}
\end{aligned}
$$

where we define $\psi_j = \frac{j-1+d}{j}\psi_{j-1}$ for $j = 0, 1, \ldots$. However, as we do not have infinite observations we will generate 3000 thousand observations and then cut off the first 2000 to mimic the behavior of an already existing process in a far horizon.
The process x_t in turn is generated as an ARMA(1, 1) process where the underlying shocks are $\epsilon_t \sim N(0, 1)$ and parameters are as given in the assignment. The data generating process is described in the functions below.

```python
def get_psis(j_max, d):
    '''
    Get an array of the psi_j/pi_j, which are MA coefficients of fractional
        process
    '''
    psi = np.empty(j_max)
    psi[0] = 1
    for j in range(1, j_max):
        psi[j] = (j-1+d)/j * psi[j-1]
    return(psi)
```

```
def data_gen(total_len, d):
    '''
    This function creates T observation array of y_t as defined in the Assignment.
    '''
    #first, get x_t
    eps = rd.normal(0,1,total_len+1)
    x = np.empty(total_len)
    x[0] = eps[1] + 0.3*eps[0]
    #epsilon is one step ahead of xt
    for t in range(1, total_len):
        x[t] = 0.5*x[t-1] + eps[t+1] + 0.3*eps[t]
    #now, get the psis corresponding to d supplied
    psis_s = get_psis(j_max=total_len, d=d)
    #lastly, calculating the y's
    y = np.empty(total_len)
    for t in range(total_len):
        psi_oi = psis_s[0:t+1]
        x_oi = x[0:t+1][::-1]
        y[t] = np.sum(psi_oi * x_oi)
    return(y)
```

Local Whittle Estimation

We now turn to the functions we use in the estimation procedure, which for the Local Whittle is quite similar to the one used in Question 15. The objective function we want to maximize for the Local Whittle estimator is now given by:

$$R(d) = log\left(\frac{1}{m}\sum_{j}^{m}\frac{I_y(\lambda_j}{\lambda_j\hat{}-2d}\right) - \frac{2d}{m}\sum_{j}^{m}log(\lambda_j)$$

Minimizing this function numerically using a grid of values for $d \in (-0.5, 0.5]$ will yield the estimate $\hat{d_{LW}}$.

```
def obj_fct(lambdas, Iy, d, m):
    '''
    Generate the value of the objective function for given m (which implictly
        defines lambdas, and thus Iy) and d
    '''
    lambdas_transform = lambdas**(-2*d)
    first = np.sum(Iy/lambdas_transform)
    second = np.sum(np.log(lambdas))
    final = np.log(1/m*first) - 2*d/m * second
    return(final)

def optimization(d_grid, lambdas, Iy, m):
    '''
    Minimize objective function on d grid
    '''
    values = np.asarray([obj_fct(d = d, lambdas=lambdas, Iy = Iy, m = m) for d in
        d_grid])
    min_i = np.where(values == min(values))
    result = d_grid[min_i]
    return(result)
```

The Python implementation of the objective function and the optimization process are shown above. *obj_fct* calculates the value of the objective function given the list of λ_j's and list of corresponding $I_y(\lambda_j)$ as well as a given guess for d. The function we define for $I_y(lambda_j)$ in Python is shown in the Appendix below. It is the same as in Question 15 again, this time only using our series y_t as an input.
However, the list of λ_j we use is now depending on the choice of bandwidth $m = \lfloor T \rfloor^{\alpha}$. Since we later want to look at the behavior of the estimator depending on m we define a function to calculate the corresponding series of λ_j (see Appendix).
Lastly, the optimization function simply applies this function for each value of d on the grid of d we supply, finds the smallest value and then returns the corresponding d on the supplied grid.

Replication Study

For the replication study, we first set up the basic parameters as given in the assignment and generate some data. Instead of generating data in each iteration, we create a matrix of dimensions $S \times M + T$, where the $\hat{s}th$ row contains the observations used in the $\hat{s}t$ repetition. We then discard the first M columns of this data matrix, i.e. disposing the first M observations of each iteration dataset.

```
M = 2000
T = 1000
S = 10
alpha_list = np.arange(0.15, 0.8+0.01, 0.1)
d_grid = np.arange(-0.45, 0.5, 0.05)
#*Generating data before the actual loop
start = time.time()
data = np.empty((S, M+T))
for s in range(S):
    yt = data_gen(M+T, d=0.25)
    data[s, :] = yt
data = data[:, M:]
```

The nested loop structure of the replication study is as follows: the outer loop iterates over the supplied list of α we are interested in. We calculate our m and λ_j based on this. The inner loop then repeats the local whittle estimation 1000 times, using the pre-defined data and stores the results in a matrix, that has dimension $length(\alpha - list) \times S$, such that in the end the i^{th} row contains the 1000 estimates of $\hat{d}_{LW}$ for the corresponding element in the α-list.
The second step in the loop is to apply the log-Periodogram regression. We can simply calculate the necessary *pseudo*-observations by following the definitions from the lecture and then do a simple OLS regression of I_j on a constant and the R_j. The resulting estimate of d is stored in the same way as the Local Whittle estimates.

Results

Using the estimates from our replication study we can calculate the bias and variance as well as the coverage probability of each estimator. To calculate the latter we first need to define the theoretical confidence interval. Following the result of Robinson (1995) we can derive the asymptotic, theoretical variance of the Local Whittle estimator:

$$\begin{aligned} & \sqrt{m}(\hat{d}_{LW} - d) \xrightarrow{d} N\left(0, \frac{1}{4}\right) \\ \Leftrightarrow \qquad & \hat{d}_{LW} \xrightarrow{d} N\left(d, \frac{1}{4m}\right) \end{aligned}$$

Thus, the theoretical 95% confidence interval using the standard deviation of the estimator is given by:

$$CI_{0.95}^{LW} = \left[d \pm 1.96 \frac{1}{4\sqrt{m}\sqrt{T}}\right]$$

A similar exercise for the log-Periodogram yields (following the asymptotic results presented in the lecture):

$$CI_{0.95}^{PR} = \left[d \pm 1.96 \frac{\sqrt{2\pi}}{\sqrt{24m}\sqrt{T}} \right]$$

We can the define function in Python that given m calculates the theoretical standard deviation of the estimators and use a second function that uses the supplied standard deviation to calculate a $2 \times length(\alpha - list)$ matrix, where the first row is the lower and the second row the upper bound of the respective confidence interval for respective m (which we have as many α we are looking at. A function comparing the estimates of our replication study and the bounds of the CI's then calculates the coverage probability as defined in the assignment. For the corresponding functions in Python code, see Appendix. The Appendix also presents the function used to calculate the biases of the estimators for each α. For the variances we use the build-in function $numpy.var()$ from the numpy module.

The bias and variances for each estimator are presented in Figure (1) and (2). The coverage probabilities are depicted in Table (1).

As can be seen from Figure (1) the bias of the LW estimator is at first well above the bias of the PR estimator. We explain this in the following way: the assumption of the log-Periodogram estimator that indeed $f_y(\lambda_j)$ is a constant is true at harmonic frequencies very close to the origin. As we only consider very small λ_j for small α the functional form and assumptions of the PR estimator might indeed hold true, which makes it perform better in terms of bias. However, since we effectively only use a very small number of the generated *pseudo*-observations this comes at a cost of a very high variance. This can very well be seen in Figure (2). Meanwhile, the LW estimator might suffer in terms of bias in our setup at very low frequencies because the actual frequencies that lead to the persistence of the process are higher. Therefore, at first the bias is high but it decreases as the frequencies we consider increase as well. The PR estimator suffers from this increase when we look at the bias, but as the LW estimator its variance decreases (see Figure(2) again). Both estimators then suffer from the fact that our baseline assumption/approximation $f_y(\lambda_j) \sim \lambda_j^{2d} f_x(\lambda_j)$ does become worse as λ_j increases. As can be seen from Figure (1) the LW estimator is more or less unbiased around $\alpha = 0.5, 0.6$.

The latter range is also where the LW estimator performs best when looking at the coverage probability. One can note that both estimators perform rather bad, but the PR estimator is constantly underperforming when compared to the LW estimator. Thus, one might argue that even though the estimator is unbiased at low frequencies, this comes at a high cost of a very large variance.

Lastly, looking at Figure (2) in general it is straightforward to see that the variance indeed decreases in the bandwidth m as is predicted by the theory.

α	LW	PR
0.2	0.030	0.025
0.3	0.060	0.025
0.4	0.098	0.033
0.5	0.184	0.037
0.6	0.223	0.038
0.7	0.020	0.009
0.8	0.000	0.000

Table 1: Coverage Probabilities

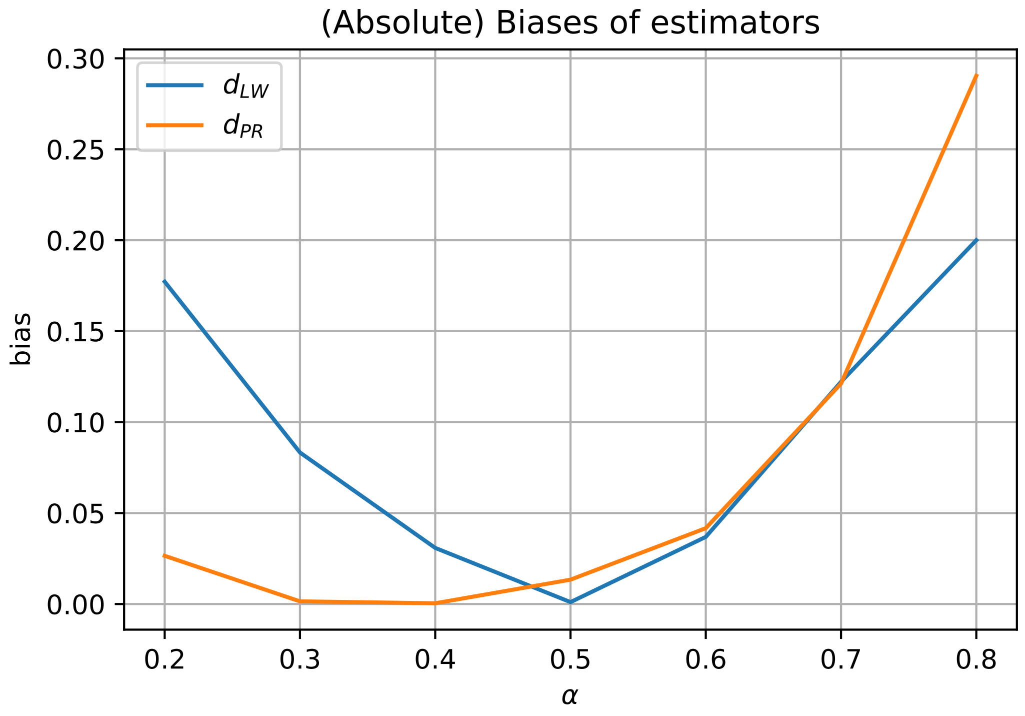

Figure 1: Biases of LW and PR estimators at different bandwidths

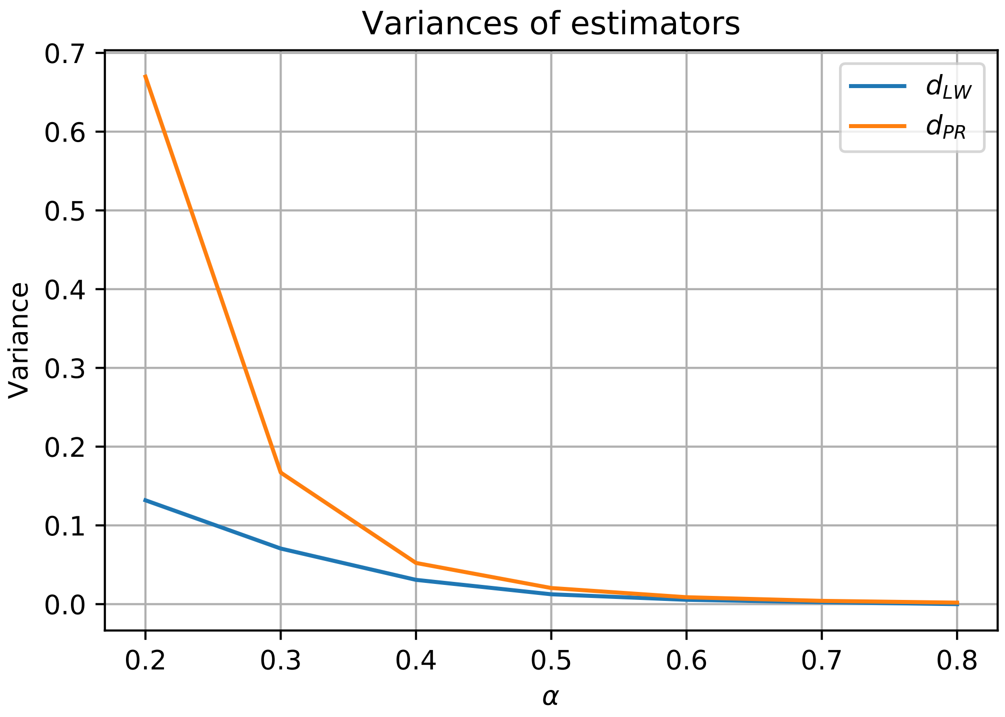

Figure 2: Variances of LW and PR estimators at different bandwidths

Appendix

```python
#Function for I_y, taken from Assignment 9
def Ix(x_t, lambda_j):
    '''
    I_x as defined on slide 144.
    '''
    T = len(x_t)
    factor = 1/(2*np.pi*T)
    cos_term = np.sum([x_t[t] * np.cos(lambda_j*t) for t in range(T)])
    sin_term = np.sum([x_t[t] * np.sin(lambda_j*t) for t in range(T)])
    return(factor * (cos_term**2 + sin_term**2))
#function to generate lambdas given m
def get_lambdas(m, T):
    '''
    Get lambda_j grid given m and T
    '''
    lambdas = np.array([2*np.pi*j/T for j in range(1, m+1)])
    return(lambdas)

#Functions for theoretical sigmas, CIs, and coverage probability
def sigma_theory_w(m):
    '''
    Get theoretical sigma given m following slide 175.
    '''
    sigma2 = 1/(4*m)
    sigma = np.sqrt(sigma2)
    return(sigma)

def sigma_theory_pr(m):
    '''
    get theoretical variance of PR from slide 184
    '''
    sigma2 = np.pi**2/(24*m)
    sigma = np.sqrt(sigma2)
    return(sigma)

def CI_theory(true_val, sigma, n):
    '''
    Get theoretical CI.
    '''
    upper = true_val + 1.96*sigma/np.sqrt(n)
    lower = true_val - 1.96*sigma/np.sqrt(n)
    CI = np.vstack((lower, upper))
    return(CI)
```

```
def coverage_prob(CIs, estimates):
    cov_prob = np.empty(CIs.shape[1])
    CIs = CIs_whittle
    for i in range(CIs.shape[1]):
        lower, upper = CIs[0, i], CIs[1, i]
        in_CI = np.asarray([(val >= lower) & (val <= upper) for val in
            estimates[i, :]])
        summation = np.sum(in_CI)
        cov_prob[i] = summation/estimates.shape[1]
    return(cov_prob)

#bias function
def bias(estimates, true_val):
    '''
    Calculate the (unscaled) bias of the estimates.
    *true_val
    *estimates
    '''
    #get number of estimates for a certain alpha
    S = estimates.shape[1]
    sum = np.sum(estimates, axis = 1)
    val = 1/S * sum - true_val
    return(val)
```