Topics in Time Series Analysis: Assignment 6

Lucas Cruz Fernandez

1544674, lcruzfer@mail.uni-mannheim.de

24th of April 2020

Q11

Our goal is to estimate the size of the "normal" t-test and different specifications of the Phillips-Perron test - that is adjusting for the serial correlation in the error term as well as the constant mean β_0 -. First, we will define some functions that we will later put to use to calculate the respective t-statistics.
We first specify the functions necessary to estimate $\gamma_e(0)$ and ω_e^2. Since we want to use the Andrews estimator for the long run variance we define two functions for the long run variance - the kernel function and the long-run variance estimator itself.

```
    def kernel(z):
        c = 6 * np.pi * z / 5
        value = 3/c**2 * (np.sin(c)/c - np.cos(c))
        return(value)

    def long_run_var(variable, T, q, k):
        factor = T/(T-k)
        gamma0 = np.var(variable)
        autocov = acovf(variable, missing = 'drop')
        cov_sum = np.sum([kernel(h/(q+1)) * 2 * autocov[h] for h in range(1, T-1)])
        lr_var = factor * (gamma0 + cov_sum)
        return(lr_var)
```

The function *acovf* is part of the *statsmodels* module, which provides a wide field of predefined functions for econometrics (or statistics in general). *acovf* calculates the auto-covariance $\gamma(h)$ for all $h = 0, ..., T-1$. We estimate all these covariances, save them as a list and then access the h^{th} element when calculating the summation term in the Andrews estimator.
Next up, we define the different test statistics. First, the simple t-test.

```
    def t_test(coeff_hat, se_hat, H_0):
        return((coeff_hat - H_0)/se_hat)
```

And then the Phillips-Perron statistic (henceforth, PP).

```
def PP_test(t_val, se_hat, omega_e, gamma_e_0, T):
    stat1 = np.sqrt(gamma_e_0/omega_e)*t_val
    stat2 = (omega_e - gamma_e_0)/(2*np.sqrt(omega_e))
    stat3 = (T * se_hat)/gamma_e_0
    return(stat1 - stat2*stat3)
```

Note that the PP-function uses an already calculated t-statistic as an argument. This saves computing time later on when we run the simulations since we apply the usual t-test anyway and therefore only need to calculate it once.
We define three different, empty lists to which the respective test value will be added later

on. The digit in the name of the lists refers to the respective testing strategy as defined in the assignment.

```
t_stat_1 = []
t_stat_2 = []
t_stat_3 = []
```

The following code blocks are all written inside a for-loop that repeats the inside described process 1000 times.
The first step inside the for-loop is to generate the data.

```
#generate an empty dataframe
    data_df = pd.DataFrame()
#start with epsilon
#we need 1002 epsilons so we have the lags to construct e_0
    data_df['epsilon_t'] = rd.normal(0, 1, 1002)
#create the lags
    data_df['epsilon_t_1'] = data_df['epsilon_t'].shift(1)
    data_df['epsilon_t_2'] = data_df['epsilon_t'].shift(2)
```

The *shift()* property shifts the column it is called for by the number of steps provided as an argument. To shift it one row "up" (i.e. generating a lag) we need positive numbers as an argument.

```
#epsilon[2] is the first period, epsilon[1] and [0] are the 0th and -1st period,
     respectively
    data_df['e_t'] = data_df['epsilon_t'] - 5/6 * data_df['epsilon_t_1'] + 1/6 *
        data_df['epsilon_t_2']
```

Since we only have 1000 observations of e due to the lag structure but our dataframe has length 1002, we can drop the first two rows.

```
#remove first 2 rows since they have missing e values and reset the index to
     start at 0 again
    data_df = data_df[2:].reset_index()
    data_df = data_df.drop(['index'], axis = 1)
```

We now create an empty column for the u_t and calculate them. To get 1000 x_t, we set $u_0 = e_0$.

```
#first set all to zero
    data_df['u_t'] = [0] * len(data_df)
#set the first value equal to first value of e, since there is no u_0
    data_df.loc[0, 'u_t'] = data_df.loc[0, 'e_t']
    for i in range(1, len(data_df)):
        data_df.loc[i, 'u_t'] = data_df.loc[i-1, 'u_t'] + data_df.loc[i, 'e_t']
```

Finally, we can create the x_t.

```
    data_df['x_t'] = 1 + data_df['u_t']
    data_df['x_t_1'] = data_df['x_t'].shift(1)
```

As the data is now created, we can run an OLS regression of x_t on its lag x_{t-1}, get the residuals, coefficient estimates and standard errors to apply our above defined functions.

```
#now do the regression of x_t on x_t_1 and a constant
#this is the same as regressing x on a constant to get u_hat and then regressing
     u_hat on its lag
    model = sm.ols('x_t ~ x_t_1', data = data_df, missing = 'drop')
    model = model.fit()

#get the necessary estimates for calculating the t statistics we are interested in
    residuals = model.resid
#get a_hat
    param = model.params[1]
#get std. error of a_hat
    std_err = model.bse[1]
#get gamma_0 for e_hat (i.e. residuals)
    gamma_e_0 = np.var(residuals)
    omega_e_1 = long_run_var(residuals, T = 1000, q = int(1000**(4/5)), k = 2)
    omega_e_2 = long_run_var(residuals, T = 1000, q = int(1000**(1/3)), k = 2)
```

Note that we calculate two different versions of $\hat{\omega}_e^2$ since the difference between strategy (2) and (3) is the q chosen for the estimation of the long-run variance.
The last step is to calculate the t-statistics and append the value to the pre-defined empty lists.

```
    t_val = t_test(coeff_hat = param, se_hat = std_err, H_0 = 1)
    t_stat_1.append(t_val)
    t_stat_2.append(PP_test(t_val, std_err, omega_e_1, gamma_e_0, 1000))
    t_stat_3.append(PP_test(t_val, std_err, omega_e_2, gamma_e_0, 1000))
```

We repeat this while process 1000 times and end up with 3 lists, each containing 1000 t-statistics. We now define the function to calculate the empirical size of the tests with respect to the Dickey-Fuller critical value of -2.86.

```
     def size(t_statistics, iterations):
    P = 1/iterations * np.sum(np.asarray(t_statistics) < -2.86)
    return(P)
```

np.asarray(t_statistics) changes the *list* element to a so-called *numpy array*, which allows for vectorization of operations, e.g. such as comparison to another value like here. The

evaluated array of Boolean operators can then be summed and divided by the number of the iterations. We do this in the following step and get P_1, P_2 and P_3, which we print.

```
P1 = size(t_stat_1, 1000)
P2 = size(t_stat_2, 1000)
P3 = size(t_stat_3, 1000)
```

The values for P_1, P_2 and P_3 are: $P_1 = 0.945$, $P_2 = 0.971$ and $P_3 = 0.895$.
The empirical size of a test should ideally by very close to the nominal size, which in our case is $\alpha = 0.05$. All three empirical sizes are very far away from this with the third strategy yielding the "closest" value. Therefore, we should prefer this strategy but it seems to be the one-eyed amongst the blind. The second test strategy - although correcting for the serial correlation of the error term and the constant - does even worse than the uncorrected "usual" t-test. This suggests that the choice of q and therefore how the autocovariances are weighted through the kernel function in the Andrews estimator, strongly affects the performance of the PP test.

Some thoughts:

The figure below shows the kernel function, which is used as a weight on the covariances in the Andrews estimator and the estimated autocovariances for one of the iterations. One can see that the q related to the second strategy leads to a long hump shape at the beginning, i.e. that even long horizon autocovariances are weighted relatively strongly. The q for the third strategy, however, puts weights only on a very short horizon of autocovariances before becoming zero. Since we know the serial dependence structure of our simulated error terms, we know that any autocovariance beyond $h = 3$ dies off. As can be seen the estimated autocovariance does not. Combined these two effects might affect the bad outcome.
Moreover, the weakness of the PP test could stem from the existence of a constant in our process. When we estimate the coefficient $\hat{a}$ for the lag of x_t in our regression the constant we estimate is equal to $\mu = \beta(1 - a)$, therefore, this estimation does not properly identify the coefficient a. The ERS test, however, accounts for these identification issues. Using the ERS provided by the *arch* module we calculate its empirical size to be around 0.07, which is very close to the nominal size of the test.

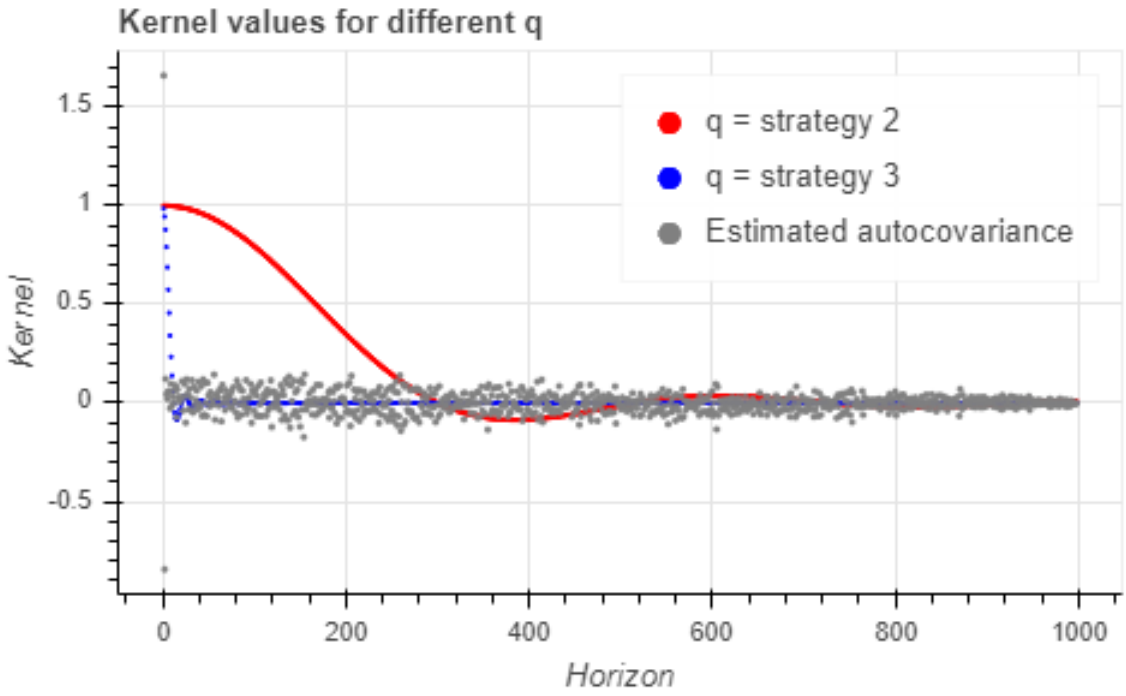
Kernel values for different q
q = strategy 2
q = strategy 3
Estimated autocovariance
Kernel
Horizon
1.5
1
0.5
0
-0.5
0
200
400
600
800
1000

Figure 1