Exercise 6: Linear models and machine learning

Kristian Urup Olesen Larsen[1]

[1]kuol@econ.ku.dk

March 26, 2020

Problem 6.1.1 - Wrapping in a function

This first question should be straight forward to complete, we simply wrap the code in a `def` and `return` statement

```
def simulate(n = 1000, m = 1500, k = 10, plot = False):
    # [function code]
    return (d, z, Pi, gamma, nu), (y, x, delta, alpha, u)
```

with an `if`-statement ensuring that the plotting code is only run when `plot = True`. Notice that the function returns two tuples of data. This is simply to make unpacking results more intuitive, as we can call

```
aux, main = simulate()
d, z, Pi, gamma, nu = aux
y, X, delta, alpha, u = main
```

Now let us quickly discuss what the code is actually doing. The main part of the code is straight forward, it simulates data according to the two equations

$$d_i = x_i'\gamma_0 + z_i'\delta_0 + u_i \qquad \text{(Auxilliary)}$$

$$y_i = \alpha_0 d_i + x_i'\beta_0 + \epsilon_i \qquad \text{(Main)}$$

The important part happens in one of the first lines, namely

```
errors = np.random.multivariate_normal(mean = [0,0], cov = [[5, 5], [5, 5]], size = n)
```

which constructs u_i, ν_i as correlated random normals, as evident from figure 1. The consequence of this is that whenever u_i is high we also expect ϵ_i to be high and vice versa. As we will see shortly this becomes an issue if we want to estimate the actual impact of d_i on y_i. i.e. α_0.

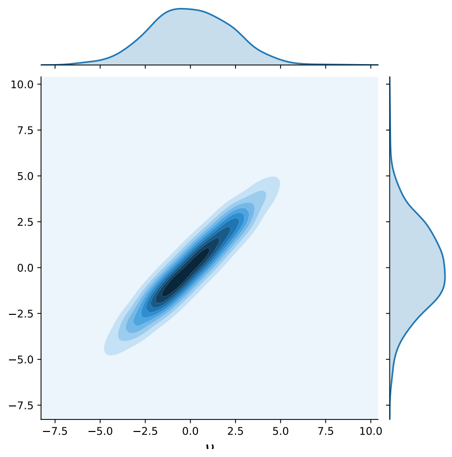

Fig 1. Correlation structure of ν_i and u_i.

Problem 6.1.2 - The naive regression

In this problem we will see what quantity we can expect to estimate by running the regression

$$y_i \sim \beta_0 + \beta_1 d_i + \eta_i \tag{1}$$

If this regression provides unbiased estimates on d_i we will expect $\hat{\beta}_1 \approx \alpha_0$. Repeating the regression many times should reveal that $\hat{\beta}_1$ falls evenly on either side of the true value α_0. However, this is not what we will find, and the reason lies in the correlated errors.

Because u_i and ϵ_i are positively correlated $u_i > 0 \Rightarrow \mathbb{E}[\epsilon_i|u_i] > 0$ - i.e. positive errors in the main equation are associated with positive errors in the auxilliary equation. Assume for a moment that u_i and ϵ_i are correlated such that $\partial\epsilon_i/\partial u_i = \sigma$. Then increasing u_i dicretly increases ϵ_i by a proportional amount. In this case what happens when $u_i = 1$? d_i increases by 1 above $\mathbb{E}[d_i|z_i, x_i]$, and y_i increase by $\alpha_0 + \sigma$ above $\mathbb{E}[y_i|x_i]$. From observing only y_i and d_i we cannot separate out what changes in y_i were due to changes in d_i (α_0), and what changes were due to changes in u_i affecting ϵ_i (σ). For this reason we should expect a bias in our naive regression results. The code for this problem starts off with setting

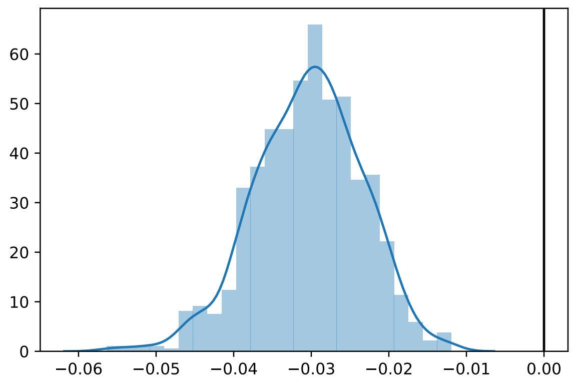

Fig 2. Distribution of estimated values for $\hat{\beta}_1$'s deviation from truth in the naive regression.

up two lists and a for-loop. The first few lines of the loop simply simulates a new dataset and unpacks all of the return values.

```
A,B = [], []
for _ in range(1000):
    aux, main = simulate()
    d, z, Pi, gamma, nu = aux
    y, X, delta, alpha, u = main
```

Next we fit the naive regression

```
    model = OLS(y, add_constant(d))
    result = model.fit()
```

and store the true α_0 alongside $\hat{\alpha}_0$ in the two lists.

```
    A.append(alpha)
    B.append(result.params[1])
```

The histogram can easily be drawn with seaborn,

```
sns.distplot([(a - b) for a,b in zip(A,B)])
plt.axvline(0,color = 'black')
```

1 Problem 6.1.3 - counting non-zero elements in δ_0

In this question we are asked to verify that only a few values in δ_0 are indeed non-zero. One approach to this is simply to refer to the simulation code. By construction parameters are set to 0 in 90% of the cases, resulting in an expected number of non-zero parameters of $0.1 \cdot 1500 = 150$ in δ_0.

To solve this question via code, begin by simulating a new dataset. Note that your result will vary slightly between simulations.

```
aux, main = simulate()
d, z, Pi, gamma, nu = aux
y, x, delta, alpha, u = main
```

We can then quickly check the number of non-zero elements in δ_0 using a list comprehension

```
len([x for x in Pi if not x == 0])
```

2 Problem 6.1.4 - The ideal instrument

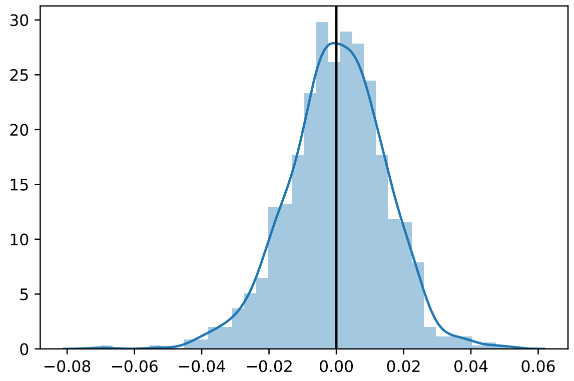

Fig 3. Distribution of estimated values for $\hat{\pi}_1$'s deviation from truth in the ideal second stage regression using $z_i'\delta_0$ as instrument.

The ideal instrument for d_i is exactly the variation induced by z_i, as this variation is going to be orthogonal to variation in d_i, y_i due to x_i or due to the error terms. Mathematically we can compute the ideal instrument as

$$\hat{d}_i^* = z_i' \cdot \delta_0 \tag{2}$$

Of course δ_0 would not be recoverable in practice, as it would require running a regression with more variables than observations. In python, we begin like we did in 6.1.2, by setting up two container lists, and in a loop simulating new data,

```
A, B = [], []
for _ in range(1000):
    aux, main = simulate()
    d, z, Pi, gamma, nu = aux
    y, x, delta, alpha, u = main
```

We then compute the ideal instrument z @ Pi and regress $y_i \sim \pi_0 + \pi_1 \hat{d}_i^*$

```
    d_hat = z @ Pi
    beta_1 = OLS(y, add_constant(d_hat)).fit().params[1]
```

After which we can close up our loop by appending the true value α_0 and our estimate $\hat{\pi}_1$ to A and B respectively.

```
A.append(alpha)
B.append(beta_1)
```

After the loop completes, the code is exactly like in 6.1.2, we draw a histogram of the differences

```
difference = [a - b for a,b in zip(A, B)]
sns.distplot(difference)
plt.axvline(0, color = 'black')
```

As you see in figure 4 this time estimates are centered around α_0, that is our estimate is in this case unbiased. Notice that this does not imply always estimating $\hat{\pi}_1 = \alpha_0$ and that there can be significant differences between the two.

3 Problem 6.1.5 - The PostLasso class

Let us discuss the PostLasso class method by method:

- `__init__(self, formula = None)` sets up an instance when the class is called. In the code you will see that it only requires one optional argument, the formula, which is only used in `__repr__`.
- `__repr__(self)` simply returns a string, which you will see if you run `print(PostLasso('hello'))` this method is mainly here for easier debugging.
- `fit(self, X, y, force_include_idx = None)` fits a post-lasso model given two matrices X and y. `force_include_idx` specify column indexes that must always be included in the OLS step. The methods first line estimates a Lasso model on X, y. Second line inserts the intersept in the 0th position of the coefficients, to store them all in a single numpy array.[1] Third line creates a mask, that picks out only the coefficients(/columns) that have non-zero values. These are the columns we will cary over to the OLS step. Lines 4 and 5 combines the column indices identified by the lasso with the ones given in `force_include_idx`, by taking the union. Line 6 creates a new matrix `relevant_x` which is the original X masked to only include the columns identified in the lasso, or force included. Finally line 7 estimates a linear regression using OLS, on this subset of X.
- `predict(self, X = None)` predicts values using the estimated model. If `X` is left as it default value the method reuse `self.relevant_x` which was stored during fitting. Otherwise, if `X` has the same shape as `self.relevant_x`, it assumes that this is new data, already filtered by the lasso, and it simply uses the estimated OLS model to predict values. In the final case, we assume as a fallback that `X` has the same shape as the original X matrix used for fitting, and mask it with the estimated `self.subset_cols` before using OLS to predict.

[1]sklearn by default stores the intercept separately from the other coefficients.

4 Problem 6.1.6 - Using the post-lasso

Finally we are ready to attempt to solve our $m \gg n$ inference problem without taking advantage of the fact that we know how the data were simulated.

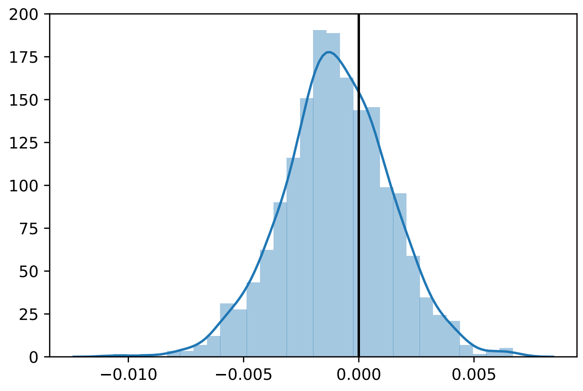

Fig 4. Distribution of estimated values for $\hat{\pi}_1$'s deviation from truth in the ideal second stage regression using post-lasso to estimate the instrument.

Our solution begins like the two other estimation problems, by setting up storage and a loop

```
A, B = [], []
for _ in range(1000):
    # Simulate a new dataset
    aux, main = simulate()
    d, z, Pi, gamma, nu = aux
    y, x, delta, alpha, u = main
```

This time, we use the post lasso to estimate a spare representation of $\hat{d}_i$. This is the main point of PostLasso, as this step involves regressing $m = 1500$ covariates with only $n = 1000$ observerations, yet we can do so, because the lasso first selects a subset of the m columns to include in the regression,

```
    dxz = PostLasso('di ~ xi + zi')
    dxz.fit(np.c_[x, z], d, force_include_idx = np.arange(x.shape[1]))
```

where, we concatenate x and z into a single matrix $[x|z]$ before passing them to .fit, we then use force_include_idx to specify that we always want the x part of this joint matrix included in the OLS step, resulting in OLS running on a matrix of the form $[x|z_a\ z_b\ z_c\ ....]$ where there may be gaps between $a, b, c, ...$ In this first step we (naturally) take d_i as the outcome variable. With the post-lasso fitted we can then compute $\hat{d}_i$, which we will concatenate with x once again to get a matrix $[x|\hat{d}]$. In the second stage we then regress $y_i \sim [x|\hat{d}_i]\beta$ and recover the PostLasso-IV estimate of α_0 from this model

```
    x_dhat = np.c_[x, dxz.predict()]
    stage_2 = OLS(y, x_dhat).fit()
    bhat = stage_2.params[-1]
```

As previously we end the loop by storing the relevant values in the storage lists we set up in the beginning

```
    A.append(alpha)
    B.append(bhat)
```

Plotting the figure is done with identical code to the one used above. From it we clearly observe an improvement over the naive regression, but some bias remains. Some of this is probably because we dont tune the Lasso, in fact there are fixed theoretical values of the Lasso hyperparameter that gives optimal performance of the PostLasso model.