TPInsights

Thursday, January 30, 2020

Econometrics in the Cloud: Two-Stage Least Squares in BigQuery ML

This is part three in a series about how to extend cloud-based data analysis tools – such as Google's BigQuery ML – to handle specific econometrics requirements. In part 1, I showed how to compute coefficient standard errors in BigQuery and in part 2, I showed how to compute robust standard errors in BigQuery.

This post shows how to perform Two-Stage Least Squares (2SLS) in BigQuery. 2SLS is used to identify an endogenous regressor of interest—that is, when a regressor is correlated with the error terms of the regression. Instead of running a single OLS regression, you estimate two regressions: one to generate a predicted value of the endogenous variable, and a second using the predicted variable instead of the actual values, and adjusting the standard errors appropriately. Here at TPI, we used 2SLS in our recent paper on internet streaming piracy.

Implementing 2SLS correctly requires calculating the coefficients and then calculating the corrected standard errors. Both aspects are needed because the standard errors generated by estimating on OLS regression using the predicted values of the endogenous variable will be incorrect, as the residuals will be based off the predicated values of the instrument and not the real values.

Let's start with the easier part: estimating the coefficients. Create the first-stage model with the endogenous variable as the dependent variable:

```
CREATE OR REPLACE MODEL `<dataset>.stage1`
OPTIONS(model_type = 'linear_reg',
        input_label_cols=[<endogenous variable>]) AS
SELECT
        < endogenous variable>
        <instrument>
        <other variables>
FROM
`<dataset>.<data>`
WHERE
        <variables> is not NULL
```

Add a new column, pred_endogenous, to hold the predicted values. Calculate these using ML.WEIGHTS for the coefficeints as we did for the robust coefficients. Next, we can run the second-stage model:

```
CREATE OR REPLACE MODEL `<dataset>.stage2`
OPTIONS(model_type = 'linear_reg', input_label_cols=[<dependent variable>]) AS
SELECT
        <dependent variable>
        pred_<endogenous variable>
        <other variables>
```

Economics Staff

Scott Wallsten, PhD
(202) 828-4405
swallsten@techpolicyinstitute.org

Robert Hahn, PhD
(202) 828-4405
rhahn@techpolicyinstitute.org

Thomas Lenard, PhD
(202) 828-4405
tlenard@techpolicyinstitute.org

Sarah Oh, JD, PhD
(202) 425-7725
soh@techpolicyinstitute.org

Lindsay Poss, MS
(202)-828-4405
lposs@techpolicyinstitute.org

Nathaniel Lovin
(202) 828-4405
nlovin@techpolicyinstitute.org

For Press Inquiries
David Fish
(571) 389-4446
dfish@techpolicyinstitute.org

Technology Policy Institute
409 12th Street, SW
Suite 700
Washington, D.C. 20024

```
FROM
ML.PREDICT(MODEL `<dataset>.stage1`, (SELECT * FROM `<dataset>.<data>`)
WHERE
        <variables> is not NULL
```

And then the coefficients can be taken from ML.WEIGHTS as before.

To correct the standard errors, we need to calculate the corrected residual mean error (based off the endogenous variable, not the predicted variable), and then multiply the incorrect standard errors by the corrected root mean squared error (rmse) divided by the original rmse. This process works for robust and non-robust standard errors.

First, we add a new function to the python code to compute the corrected rmse:

```
def realrmse(dataset, model_name, endogenous, regressand, data, n):
  #add columns to table (the 2 is needed since
  #predicted_regressand exists in robust for first stage)
  table_ref = client.dataset(dataset).table(data)
  table = client.get_table(table_ref)

  original_schema = table.schema
  new_schema = original_schema[:]

  new_schema.append(bigquery.SchemaField("predicted_" + regressand + "2", "FLOAT"))
  new_schema.append(bigquery.SchemaField("residual_" + regressand + "2", "FLOAT"))
  table.schema = new_schema
  table = client.update_table(table, ["schema"])
  assert len(table.schema) == len(original_schema) + 2 == len(new_schema)

  #find first stage coefficents
  coeffs = {}
  query = ("SELECT processed_input, weight FROM ML.WEIGHTS(MODEL `"
    + dataset + "." + model_name + "`)")
  query_job = client.query(query)
  result = query_job.result()
  for row in result:
    coeffs[row.processed_input] = {}
    coeffs[row.processed_input]['coefficient'] = row.weight

  #prediction
  regression = ""
  for coeff in coeffs.keys():
    if coeff != "__INTERCEPT__":
      if coeff[0:4]=="pred":
        regression += str(coeffs[coeff]['coefficient']) + "*" + endogenous + " + "
      else:
        regression += str(coeffs[coeff]['coefficient']) + "*" + coeff + " + "
    else:
      regression += str(coeffs[coeff]['coefficient']) + " + "
  regression = regression[:-3]
  query = ("UPDATE `" + dataset + "." + data + "` SET predicted_" + regressand
    + "2 = " + regression + " WHERE predicted_" + regressand + "2 is null")
  query_job = client.query(query)
  result = query_job.result()

  #residuals
  query = ("UPDATE `" + dataset + "." + data + "` SET residual_" + regressand
```

```
    + "2 = predicted_" + regressand + "2 - " + regressand + " WHERE residual_"
      + regressand + "2 is null")
  query_job = client.query(query)
  result = query_job.result()

  #Compute rmse
  query = ("SELECT SQRT(SUM(POW(residual_" + regressand + "2, 2)) / "
    + str(n-len(coeffs)) + ") FROM `" + dataset + "." + data + "`")
  query_job = client.query(query)
  result = query_job.result()
  for row in result:
    return row.f0_
```

Then we simply need to compute the corrected standard errors:

```
def correctstandarderrors(coeffs, orgrmse, corrrmse):
  ratio =  (corrrmse/orgrmse)
  print(ratio)
  for coeff in coeffs.keys():
    coeffs[coeff]['2SLS se'] = ratio * coeffs[coeff]['standard error']
```

We need to make a few small changes to the body of our program for these functions to work. For the non-robust errors, we need to add the first-stage model and the residual to the input call arguments. Then we need to add calls to the two new functions, and change the standard errors to calculate the t-stats to the corrected ones:

```
dataset = sys.argv[1]
data = sys.argv[2]
model_name = sys.argv[3]
endogenous = sys.argv[4]
regressand = sys.argv[5]
n = int(sys.argv[6])
sqrtn = math.sqrt(n-5)
root_mean = rmse(model_name)
coeffs = standef(model_name)
coeffs = rsquared(data, coeffs)
standarderror(root_mean, sqrtn, coeffs)
root_mean2 = realrmse(dataset, model_name, endogenous, regressand, data, n)
correctstandarderrors(coeffs, root_mean, root_mean2)

#The t-stat for the null hypothesis Beta_hat = 0 is Beta_hat/se(Beta_hat)
coefficients(coeffs, dataset, model_name)
for coeff in coeffs.keys():
  coeffs[coeff]['tstat'] = coeffs[coeff]['coefficient']/coeffs[coeff]['2SLS se']
  coeffs[coeff]['pvalue'] = 2*t.sf(abs(coeffs[coeff]['tstat']), n-len(coeffs.keys())-1)

for coeff in coeffs.keys():
  print(coeff + " coefficient: " + str(coeffs[coeff]['coefficient']))
  print(coeff + " standard error: " + str(coeffs[coeff]['2SLS se']))
  print(coeff + " t-stat: " + str(coeffs[coeff]['tstat']))
  print(coeff + " p-value: " + str(coeffs[coeff]['pvalue']))
```

We have to do the same for robust as well, although the regressand is already there:

```
dataset = sys.argv[1]
data = sys.argv[2]
model_name = sys.argv[3]
endogenous = sys.argv[4]
regressand = sys.argv[5]
n = int(sys.argv[6])
coeffs = {}
coeffs = coefficients(dataset, model_name)
addColumns(dataset, data, coeffs, regressand)
predict(dataset, data, regressand, coeffs)
residuals(dataset, data, regressand)
coeffs.pop("__INTERCEPT__")
coeffs = regressions(dataset, data, coeffs, regressand)
root_mean = handrmse(dataset, data, coeffs, regressand, n)
root_mean2 = realrmse(dataset, model_name, endogenous,
  regressand, data, n)
correctstandarderrors(coeffs, root_mean, root_mean2)

for coeff in coeffs.keys():
  coeffs[coeff]['tstat'] = coeffs[coeff]['coefficient']
    /coeffs[coeff]['2SLS se']
  coeffs[coeff]['pvalue'] = 2*t.sf(abs(coeffs[coeff]['tstat']),
    n-len(coeffs.keys())-1)

for coeff in coeffs.keys():
  print(coeff + " coefficient: "
    + str(coeffs[coeff]['coefficient']))
  print(coeff + " standard error: "
    + str(coeffs[coeff]['2SLS se']))
  print(coeff + " t-stat: " + str(coeffs[coeff]['tstat']))
  print(coeff + " p-value: " + str(coeffs[coeff]['pvalue']))
```

Then the programs can be run as

```
python se2sls.py <dataset> <data> <model_name>
        <endogenous><regressand> <n>
```

where <dataset>is the BigQuery dataset where your model and data are located, <data>is the BigQuery table with your data, <model_name>is the name of the original BigQuery ml model, <endogenous>is the endogenous variable you are predicting in stage 1, <regressand>is the dependent variable of the original regression, and <n>is the size of the sample.

Let's compare the output of this program with the Stata output to show that it works. We'll use the "CollegeDistance" dataset from Applied Economics in R (https://cran.r-project.org/web/packages/AER/AER.pdf) again. The "CollegeDistance" dataset has 4,739 observations so the degrees of freedom correction is smaller and the comparison should be close. We run a two-stage regression that in Stata would be:

Recent TPInsights

The 2020 Annual Economics Meetings: Tacos, GDPR, Machine Learning, and New Ways to Randomize (Jan 16, 2020)

What Tech is at CES and Where is It From? (Jan 10, 2020)

Econometrics in the Cloud: Robust Standard Errors in BigQuery ML (Dec 10, 2019)

Econometrics in the Cloud: Extending Google BigQuery ML (Nov 6, 2019)

```
ivreg2 wage (education = distance) unemp tuition
```

		Unemployment	Distance	Tuition
Stata	Coefficient	.1095696	.324572	1.025165
	Standard Error	.0075151	.1269422	.0660714
	Robust Standard Error	.0074349	.1268149	.0523126
BigQuery	Coefficient	0.11049387	0.4681203	0.984163005
	Standard Error	0.0081373	0.162673	0.0761612
	Robust Standard Error	0.00806	0.162078	0.0589027