pyRVtest

Marco Duarte, Lorenzo Magnolfi, Mikkel Solvsten, Christopher Su

Jan 26, 2023

CONTENTS:

The documentation can be navigated with the sidebar, the links below, or the index.

CHAPTER

ONE

INTRODUCTION

Note: This package is in beta. In future versions, the API may change substantially. Please use the GitHub issue tracker to report bugs or to request features.

This code was written to perform the procedure for testing firm conduct developed in "Testing Firm Conduct" by Marco Duarte, Lorenzo Magnolfi, Mikkel Sølvsten, and Christopher Sullivan. It builds on the PyBLP source code (see Conlon and Gortmaker (2020)) - to do so.

The code implements the following features:

- Computes Rivers and Vuong (2002) (RV) test statistics to test a menu of two or more models of firm conduct allowing for the possibility that firms or consumers face per unit or ad-valorem taxes.
- Implements the RV test using the variance estimator of Duarte, Magnolfi, Sølvsten, and Sullivan (2023), including options to adjust for demand estimation error and clustering
- Computes the effective F-statistic proposed in Duarte, Magnolfi, Sølvsten, and Sullivan (2023) to diagnose instrument strength with respect to worst-case size and best-case power of the test, and reports appropriate critical values
- Reports Hansen, Lunde, and Nason (2011) MCS p-values for testing more than two models
- Compatible with PyBLP Conlon and Gortmaker (2020), so that demand can be estimated with PyBLP, and the estimates are an input to the test for conduct

For a full list of references, see the references in Duarte, Magnolfi, Sølvsten, and Sullivan (2023).

1.1 Install

First, you will need to download and install python, which you can do from this link.

You will also need to make sure that you have all package dependencies installed.

To install the pyRVtest package, use pip:

```
pip install pyRVtest
```

This should automatically install the python packages on which pyRVtest depends: numpy, pandas, statsmodels, pyblp

To update to a newer version of the package use:

```
pip install --upgrade pyRVtest
```

1.2 Using the package

For a detailed tutorial about how to set up and run the testing procedure, see Tutorial.

1.3 Citing the package

When using the package, please include the following citation:

Duarte, M., L. Magnolfi, M. Sølvsten, C. Sullivan, and A. Tarascina (2023): "pyRVtest: A Python package for testing firm conduct," https://github.com/anyatarascina/pyRVtest.

@misc{ pyrvtest, author={Marco Duarte and Lorenzo Magnolfi and Mikkel S{o}lvsten and Christopher Sullivan and Anya Tarascina}, title={texttt{pyRVtest}: A Python package for testing firm conduct}, howpublished={url{https://github.com/anyatarascina/pyRVtest}}, year={2023}

1.4 Bugs and Requests

Please use the GitHub issue tracker to submit bugs or to request features.

CHAPTER

TWO

TUTORIAL

This section uses a series of Jupyter Notebooks to explain how pyRVtest can be used to solve example problems, compute post-estimation outputs, and simulate problems.

2.1 Testing Firm Conduct with Cereal Data

[1]:
```
import numpy as np
import pandas as pd
import pyblp
import pyRVtest

pyblp.options.digits = 2
pyblp.options.verbose = False
pyRVtest.options.digits = 2
pyRVtest.__version__
```

[1]:
```
'0.2.0'
```

2.1.1 Overview

In this tutorial, we are going to use the *Nevo (2000)* fake cereal data which is provided in the PyBLP package. PyBLP has excellent documentation including a thorough tutorial for estimating demand on this dataset which can be found here.

Note that the data are originally designed to illustrate demand estimation. Thus, the following caveats should be kept in mind:

- The application in this tutorial only serves the purpose to illustrate how pyRVtest works. The results we show should not be used to infer conduct or any other economic feature about the cereal industry.
- To test conduct, a researcher generally needs data on both cost shifters, and strong excluded instruments. As the data was designed to perform demand estimation, it does not necessarily have the features that are desirable to test conduct in applications.
- Hence, the specifications that we use below, including the specification of firm cost and the candidate conducts models, are just shown for illustrative purposes and may not be appropriate for the economic context of the cereal industry.

The tutorial proceeds in the following steps:

- *Load the main dataset*
- *Estimate demand with PyBLP*

- *Test firm conduct with pyRVtest*

2.1.2 Load the main dataset

First we will use pandas to load the necessary datasets from PyBLP. We call the main data containing information on markets and product characteristics `product_data`. The important product characteristics for demand estimation in this data set are `price`, `sugar`, and `mushy`.

[2]:
```
product_data = pd.read_csv(pyblp.data.NEVO_PRODUCTS_LOCATION)
product_data.head()
```

[2]:
```
  market_ids  city_ids  quarter product_ids  firm_ids  brand_ids    shares  \
0      C01Q1         1        1       F1B04         1          4  0.012417
1      C01Q1         1        1       F1B06         1          6  0.007809
2      C01Q1         1        1       F1B07         1          7  0.012995
3      C01Q1         1        1       F1B09         1          9  0.005770
4      C01Q1         1        1       F1B11         1         11  0.017934

     prices  sugar  mushy  ...  demand_instruments10  demand_instruments11  \
0  0.072088      2      1  ...              2.116358             -0.154708
1  0.114178     18      1  ...             -7.374091             -0.576412
2  0.132391      4      1  ...              2.187872             -0.207346
3  0.130344      3      0  ...              2.704576              0.040748
4  0.154823     12      0  ...              1.261242              0.034836

   demand_instruments12  demand_instruments13  demand_instruments14  \
0             -0.005796              0.014538              0.126244
1              0.012991              0.076143              0.029736
2              0.003509              0.091781              0.163773
3             -0.003724              0.094732              0.135274
4             -0.000568              0.102451              0.130640

   demand_instruments15  demand_instruments16  demand_instruments17  \
0              0.067345              0.068423              0.034800
1              0.087867              0.110501              0.087784
2              0.111881              0.108226              0.086439
3              0.088090              0.101767              0.101777
4              0.084818              0.101075              0.125169

   demand_instruments18  demand_instruments19
0              0.126346              0.035484
1              0.049872              0.072579
2              0.122347              0.101842
3              0.110741              0.104332
4              0.133464              0.121111

[5 rows x 30 columns]
```

It is possible to estimate demand and test conduct with other data, provided that they have the `product_data` structure described here.

We will also load market demographics data from PyBLP and call this `agent_data`. This data contains draws of `income`, `income_squared`, `age`, and `child`.

[3]:
```
agent_data = pd.read_csv(pyblp.data.NEVO_AGENTS_LOCATION)
agent_data.head()
```

[3]:
```
  market_ids  city_ids  quarter  weights    nodes0    nodes1    nodes2  \
0      C01Q1         1        1     0.05  0.434101 -1.500838 -1.151079
1      C01Q1         1        1     0.05 -0.726649  0.133182 -0.500750
2      C01Q1         1        1     0.05 -0.623061 -0.138241  0.797441
3      C01Q1         1        1     0.05 -0.041317  1.257136 -0.683054
4      C01Q1         1        1     0.05 -0.466691  0.226968  1.044424

     nodes3    income  income_squared       age     child
0  0.161017  0.495123        8.331304 -0.230109 -0.230851
1  0.129732  0.378762        6.121865 -2.532694  0.769149
2 -0.795549  0.105015        1.030803 -0.006965 -0.230851
3  0.259044 -1.485481      -25.583605 -0.827946  0.769149
4  0.092019 -0.316597       -6.517009 -0.230109 -0.230851
```

2.1.3 Estimate demand with PyBLP

Next, we set up the demand estimation problem using *Conlon and Gortmaker (2020)*.

In this example the linear characteristics in consumer preferences are price and product fixed effects. Nonlinear characteristics with random coefficients include a constant, `price`, `sugar`, and `mushy`. For these variables, we are going to estimate both the variance of the random coefficient as well as demographic interactions for `income`, `income_squared`, `age`, and `child`. More info can be found here.

Running the problem setup yields output which summarizes the dimensions of the problem (see here) for description of each variable. Also reported are the Formulations, i.e., the linear characteristics or non-linear characteristics for which we are estimating either variances or demographic interactions, and a list of the demographics being used.

[4]:
```
pyblp_problem = pyblp.Problem(
    product_formulations=(
        pyblp.Formulation('0 + prices ', absorb='C(product_ids)'),
        pyblp.Formulation('1 + prices + sugar + mushy'),
    ),
    agent_formulation=pyblp.Formulation('0 + income + income_squared + age + child'),
    product_data=product_data,
    agent_data=agent_data
)
pyblp_problem
```

[4]:
```
Dimensions:
================================================
 T    N     F    I     K1    K2    D    MD    ED
---  ----  ---  ----  ----  ----  ---  ----  ----
94   2256   5   1880   1     4     4    20    1
================================================

Formulations:
=============================================================================
       Column Indices:           0           1           2      3
---------------------------  ------  --------------  -----  -----
 X1: Linear Characteristics    prices
```

(continues on next page)

(continued from previous page)

```
X2: Nonlinear Characteristics    1         prices      sugar  mushy
       d: Demographics         income  income_squared   age   child
=================================================================
```

Next, we solve the demand estimation and store the results for use with the testing module.

The output includes information on computation progress, as well as tables with the parameter estimates. A full list of the post-estimation output which can be queried is found here.

[5]:
```
pyblp_results = pyblp_problem.solve(
  sigma=np.diag([0.3302, 2.4526, 0.0163, 0.2441]),
  pi=[
      [5.4819,   0.0000,  0.2037, 0.0000],
      [15.8935, -1.2000,  0.0000, 2.6342],
      [-0.2506,  0.0000,  0.0511, 0.0000],
      [1.2650,   0.0000, -0.8091, 0.0000]
  ],
  method='1s',
  optimization=pyblp.Optimization('bfgs', {'gtol': 1e-5})
)
pyblp_results
```

[5]:
```
Problem Results Summary:
=========================================================================================================================
GMM   Objective  Gradient      Hessian         Hessian     Clipped  Weighting Matrix ␣
↪Covariance Matrix
Step    Value      Norm    Min Eigenvalue  Max Eigenvalue  Shares   Condition Number ␣
↪Condition Number
----  ---------  --------  --------------  --------------  -------  ----------------  ---
↪--------------
 1    +4.6E+00   +6.9E-06     +2.4E-05        +1.6E+04        0         +6.9E+07        ␣
↪ +8.4E+08
=========================================================================================================================

Cumulative Statistics:
=========================================================================
Computation  Optimizer  Optimization   Objective   Fixed Point  Contraction
   Time      Converged   Iterations   Evaluations  Iterations   Evaluations
-----------  ---------  ------------  -----------  -----------  -----------
 00:00:25       Yes           51           57          46384       143967
=========================================================================

Nonlinear Coefficient Estimates (Robust SEs in Parentheses):
=========================================================================================================================
Sigma:      1         prices      sugar       mushy     |   Pi:      income    income_
↪squared     age         child
------  ----------  ----------  ----------  ----------  |  ------  ----------  ----------
↪----  ----------  ----------
  1      +5.6E-01                                       |    1      +2.3E+00      +0.
↪0E+00      +1.3E+00    +0.0E+00
        (+1.6E-01)                                     |          (+1.2E+00)           ␣
↪       (+6.3E-01)
                                                       |
```

(continues on next page)

(continued from previous page)

```
prices    +0.0E+00     +3.3E+00                                |  prices    +5.9E+02       -3.
↪0E+01       +0.0E+00     +1.1E+01
                    (+1.3E+00)                                |            (+2.7E+02)     (+1.
↪4E+01)                 (+4.1E+00)
                                                              |
sugar     +0.0E+00     +0.0E+00     -5.8E-03                  |  sugar     -3.8E-01       +0.
↪0E+00       +5.2E-02     +0.0E+00
                                  (+1.4E-02)                  |            (+1.2E-01)            ␣
↪       (+2.6E-02)
                                                              |
mushy     +0.0E+00     +0.0E+00     +0.0E+00     +9.3E-02    |  mushy     +7.5E-01       +0.
↪0E+00       -1.4E+00     +0.0E+00
                                                 (+1.9E-01)   |            (+8.0E-01)            ␣
↪       (+6.7E-01)
=============================================================================================================

Beta Estimates (Robust SEs in Parentheses):
==========
  prices
----------
 -6.3E+01
(+1.5E+01)
==========
```

2.1.4 Test firm conduct with pyRVtest

pyRVtest follows a similar structure to PyBLP. First, you set up the testing problem, then you run the test.

Setting Up Testing Problem

Here is a simple example of the code to set up the testing problem for testing between two models: 1. manufacturers set retail prices according to Bertrand vs 2. manufacturers set retail prices according to monopoly (i.e., perfect collusion).

We set up the testing problem with `pyRVtest.problem` and we store this as a variable `testing_problem`:

[6]:
```
testing_problem = pyRVtest.Problem(
    cost_formulation = (
        pyRVtest.Formulation('0 + sugar', absorb = 'C(firm_ids)' )
    ),
    instrument_formulation = (
        pyRVtest.Formulation('0 + demand_instruments0 + demand_instruments1')
    ),
    model_formulations = (
        pyRVtest.ModelFormulation(model_downstream='bertrand', ownership_downstream=
↪'firm_ids'),
        pyRVtest.ModelFormulation(model_downstream='monopoly', ownership_downstream=
↪'firm_ids')
    ),
    product_data = product_data,
    demand_results = pyblp_results
)
```

Detailed information on the input accepted by *Problem* and how to specify them can be found in the API documentation. Below we clarify the inputs (observed exogenous cost shifters, instruments for testing conduct, and models to be tested) in this particular example:

- `cost_formulation`: Here, the researcher specifies the observable linear shifters of marginal cost (in the current version of the package, these must be exogenous variables). In this example, we have defined the cost formulation as `pyRVtest.Formulation('0 + sugar', absorb = 'C(firm_ids)' )`. Here, `0` means no constant. We are also including the variable `sugar` as an observed cost shifter and this variable must be in the `product_data`. Finally `absorb = 'C(firm_ids)'` specifies that we are including firm fixed effects which will be absorbed. The variable `firm_ids` must also be in the `product_data`.
- `instrument_formulation`: Here, the researcher specifies the set of instruments she wants to use to test conduct. In this example, we will use one set of instruments to test conduct which contains two variables, `demand_instruments0` and `demand_instruments1`. Thus, we have defined the instrument formulation as `pyRVtest.Formulation('0 + demand_instruments0 + demand_instruments1')`. Here, `0` means no constant and this should always be specified as a 0. Both `demand_instruments0` and `demand_instruments1` must also be in the `product_data`. It is possible to test conduct separately for more than one set of instruments as shown in the example below.
- `model_formulations`: Here, we have specified two ModelFormulations and therefore two models to test. The first model is specified as `pyRVtest.ModelFormulation(model_downstream='bertrand', ownership_downstream='firm_ids')` `model_downstream = 'bertrand'` indicates that retail prices are set according to Bertrand. `ownership_downstream='firm_ids'` specifies that the ownership matrix in each market should be built from the variable `firm_id` in the `product_data`. For testing more than two models, see the example below. To find the full list of models supported by pyRVtest and their associated *ModelFormulation* see the Library of Models page.

[7]:
```
testing_problem
```

[7]:
```
Dimensions:
=========================
 T    N     M    L   d_z0
---  ----  ---  ---  ----
94   2256   2    1    2
=========================

Formulations:
============================================================
Column Indices:            0                     1
---------------  --------------------  --------------------
w: Marginal Cost         sugar
z0: Instruments   demand_instruments0  demand_instruments1
============================================================

Models:
=========================================
                          0         1
--------------------  --------  --------
 Model - Downstream    bertrand  monopoly
  Model - Upstream       None      None
Firm IDs - Downstream  firm_ids  monopoly
 Firm IDs - Upstream     None      None
      VI Index           None      None
 Cost Scaling Column     None      None
      Unit Tax           None      None
```

(continues on next page)

(continued from previous page)

```
    Advalorem Tax        None      None
   Advalorem Payer       None      None
User Supplied Markups    None      None
========================================
```

The first table `Dimensions` reports the following statistics:

- T = number of markets
- N = number of observations
- M = number of models (each specified by a model formulation)
- L = number of instrument sets (each specified by an instrument formulation)
- d_z_0 = number of instruments in the first instrument set (with more than one instrument formulation, additional columns d_z_1, d_z_2, ... $d\_z_{(L-1)}$ would be reported)

The second table `Formulations` reports the variables specified as observed cost shifters and excluded instruments. The first row indicates that `sugar` is the only included observed cost shifter (ignoring the fixed effects). The second row indicates that `demand_instruments0` and `demand_instruments1` are the excluded instruments for testing each model.

The third table `Models` specifies the models being tested, where each model is a column in the table

- **Model-Downstream** reports the name of the model of firm conduct. For vertical models of wholesaler and retailer behavior, this reports the nature of retailer conduct.
- **Model-Upstream** reports, for vertical models with wholesalers and retailers, the nature of wholesaler conduct. In this example, we are ignoring upstream behavior and assuming manufacturers set retail prices directly as in *Nevo (2001)*.
- **Firm Ids - Downstream** is the variable in `product_data` used to make the ownership matrix for setting retail conduct (prices or quantities). If monopoly is specified as Model-Downstream, then Firm id - Downstream will default to monopoly and the ownership matrix in each market will be a matrix of ones.
- **Firm Ids - Upstream** is the same as **Firm IDs - Downstream** but for wholesale price or quantity behavior.
- **VI Index** is the name of the dummy variable indicating whether retailer and wholesaler are vertically integrated.
- **User Supplied Markups** indicates if the user has chosen to input their own markup computation instead of choosing a prespecified model for which the package will compute markups.

Additionally, the table contains outputs that are relevant when taxes are levied in the markets being studied (**Unit Tax**, **Advalorem Tax**, **Advalorem Payer**). In this example, there are no taxes in the market for cereal. These will be discussed in the testing with taxes example below.

Running the Testing Procedure

Now that the problem is set up, we can run the test, which we do with the following code.

Given that we define the variable `testing_problem` as pyRVtest.problem, we must write `testing_problem.solve` in the first line. There are two user specified options in running the test:

- `demand_adjustment`: `False` indicates that the user does not want to adjust standard errors to account for two-step estimation with demand. `True` indicates standard errors should be adjusted to account for demand estimation.

- `clustering_adjustment`: `False` means no clustering. `True` indicates that all standard errors should be clustered. In this case, a variable called `clustering_ids` which indicates the cluster to which each group belongs needs to appear in the `product_data`. See example below.

Both of these adjustments are implemented according to the formulas in Appendix C of Duarte, Magnolfi, Sølvsten, and Sullivan (2023).

[8]:
```
testing_results = testing_problem.solve(
    demand_adjustment=False,
    clustering_adjustment=False
)
testing_results
```

```
Computing Markups ...
Total Time is ... 0.1552119255065918
```

[8]:

```
Testing Results - Instruments z0:
================================================================================
  TRV:                 |   F-stats:                 |    MCS:
--------  ---  ------  |  ----------  ---  -------  |  --------  ------------
 models    0     1     |    models     0      1     |   models   MCS p-values
--------  ---  ------  |  ----------  ---  -------  |  --------  ------------
   0      nan  -1.144  |      0       nan   13.3    |     0          1.0
                       |                    *** ^^^  |
   1      nan   nan    |      1       nan    nan    |     1          0.252
                       |                            |
================================================================================
Significance of size and power diagnostic reported below each F-stat
*, **, or *** indicate that F > cv for a worst-case size of 0.125, 0.10, and 0.075 given␣
↪d_z and rho
^, ^^, or ^^ indicate that F > cv for a best-case power of 0.50, 0.75, and 0.95 given d_
↪z and rho
appropriate critical values for size are stored in the variable F_cv_size_list of the␣
↪pyRVtest results class
appropriate critical values for power are stored in the variable F_cv_power_list of the␣
↪pyRVtest results class
================================================================================
```

The result table is split into three parts: the pairwise RV test statistics, the pairwise F-statistics, and the model confidence set p-values. Details on these objects and how they are computed can be found in Duarte, Magnolfi, Sølvsten, and Sullivan (2023).

- The first part of the table reports the pairwise RV test statistic given the specified adjustments to the standard errors. In this example, there is one RV test statistic as we are only testing two models. Elements on and below the main diagonal in the RV and F-statistic block are set to "nan" since both RV tests and F-statistics are symmetric for a pair of models. A negative test statistic suggests better fit of the row model, in this case, Bertrand. The critical values for each T^{RV} are $\pm$ 1.96, so the RV test cannot reject the null of equal fit of the two models.
- The second part reports the pairwise F-statistics, again with the specified adjustments to the standard errors. The symbols underneath each F-statistic indicate whether the corresponding F-statistic is weak for size or for power. The appropriate critical values for each F-statistic depend on the number of instruments the researcher is using, as well as the value of the scaling parameter ρ. In the above table, we see that the F-statistic is associated with a symbol of "***" for size. This means that the probability of incorrectly rejecting the null if it is true (Type-I error) is at most 7.5%. The same F-statistic is associated with "^" for power, meaning that the probability of

rejecting the null if it is false is at least 95%. Therefore the instruments in this example are neither weak for size nor for power.

- Finally, the p-values associated with the model confidence set are reported. To interpret these p-values, a researcher chooses a significance level α. Any model whose p-value is below α is rejected. The models with p-values above α cannot be rejected by the instruments and the researcher should conclude that they have equal fit. Thus, in this example, for an $\alpha = 0.05$, the researcher is left with a model confidence set containing both models.

The testing procedure also stores additional output which the user can access after running the testing code. A full list of available output can be found in *ProblemResults*.

As an example, the procedure stores the markups for each model. In the above code, we stored `testing_problem.solve()` as the variable `testing_results. Thus, to access the markups, you type

[9]:
```
testing_results.markups
```

[9]:
```
[array([[0.03616274],
        [0.02752501],
        [0.04300875],
        ...,
        [0.03948227],
        [0.02837644],
        [0.04313683]]),
 array([[0.07839009],
        [0.04966324],
        [0.08338118],
        ...,
        [0.08728693],
        [0.05866822],
        [0.09141367]])]
```

where the first array, `testing_results.markups[0]` stores the markups for model 0 and the second array `testing_results.markups[1]` stores the markups for model 1.

Example with more than two models and more than one instrument set

Here we consider a more complicated example to illustrate more of the features of pyRVtest. Here, we are going to test five models of vertical conduct: 1. manufacturers set monopoly retail prices 2. manufacturers set Bertrand retail prices 3. manufacturers set Cournot retail quantities 4. manufacturers set Bertrand wholesale prices and retailers set monopoly retail prices 5. manufacturers set monopoly wholesale prices and retailers set monopoly retail prices

To accumulate evidence, we are going to separatley use two different sets of instruments to test these models.

We are also going to adjust all standard errors to account for two-step estimation coming from demand, as well as cluster standard errors at the market level. To implement these adjustments, we need to add a variable to the `product_data` called `clustering_ids`:

[10]:
```
product_data['clustering_ids'] = product_data.market_ids
```

Next, we can initialize the testing problem.

Notice that to add more models or more instrument sets, we add model formulations and instrument formulations. Further notice that by specifying `demand_adjustment = True` and `clustering_adjustment = True` we turn on two-step adjustments to the standard errors as well as clustering at the level indicated by `product_data.clustering_ids`.

[11]:
```
testing_problem = pyRVtest.Problem(
    cost_formulation=(
        pyRVtest.Formulation('1 + sugar', absorb='C(firm_ids)')
    ),
    instrument_formulation=(
        pyRVtest.Formulation('0 + demand_instruments0 + demand_instruments1'),
        pyRVtest.Formulation('0 + demand_instruments2 + demand_instruments3 + demand_
↪instruments4')
    ),
    model_formulations=(
        pyRVtest.ModelFormulation(model_downstream='monopoly', ownership_downstream=
↪'firm_ids'),
        pyRVtest.ModelFormulation(model_downstream='bertrand', ownership_downstream=
↪'firm_ids'),
        pyRVtest.ModelFormulation(model_downstream='cournot', ownership_downstream='firm_
↪ids'),
        pyRVtest.ModelFormulation(
            model_downstream='monopoly',
            ownership_downstream='firm_ids',
            model_upstream='bertrand',
            ownership_upstream='firm_ids'
        ),
        pyRVtest.ModelFormulation(
            model_downstream='monopoly',
            ownership_downstream='firm_ids',
            model_upstream='monopoly',
            ownership_upstream='firm_ids'
        )
    ),
    product_data=product_data,
    demand_results=pyblp_results
)
testing_problem
```

[11]:
```
Dimensions:
================================
 T    N     M    L   d_z0  d_z1
---  ----  ---  ---  ----  ----
94   2256   5    2    2     3
================================

Formulations:
=============================================================================
Column Indices:            0                    1                    2
----------------  -------------------  -------------------  -------------------
w: Marginal Cost         sugar
z0: Instruments   demand_instruments0  demand_instruments1
z1: Instruments   demand_instruments2  demand_instruments3  demand_instruments4
=============================================================================

Models:
=========================================================================
                          0         1         2         3         4
```

(continues on next page)

(continued from previous page)

```
--------------------  --------  --------  --------  --------  --------
 Model - Downstream    monopoly  bertrand  cournot   monopoly  monopoly
  Model - Upstream       None      None      None    bertrand  monopoly
Firm IDs - Downstream  monopoly  firm_ids  firm_ids  monopoly  monopoly
 Firm IDs - Upstream     None      None      None    firm_ids  monopoly
      VI Index           None      None      None      None      None
 Cost Scaling Column     None      None      None      None      None
      Unit Tax           None      None      None      None      None
    Advalorem Tax        None      None      None      None      None
   Advalorem Payer       None      None      None      None      None
User Supplied Markups    None      None      None      None      None
==========================================================================
```

Now, we can run `testing_problem.solve` to output the testing results. Each table here shows the results for each instrument set. The tables appear in the order of the instrument sets specified by the user in `instrument_formulations`.

[12]:
```
testing_results = testing_problem.solve(
    demand_adjustment=True,
    clustering_adjustment=True
)
testing_results
```

```
Computing Markups ...
Total Time is ... 19.744781017303467
```

[12]:

```
Testing Results - Instruments z0:
==========================================================================================================================
  TRV:                                    |   F-stats:                                                                  ␣
↪    |    MCS:
--------  ---  ----  -----  ------  ------  |  ----------  ---  -----  -----  -----  ----
↪-  |  --------  ------------
 models    0    1      2      3       4     |    models     0     1      2      3      4␣
↪    |   models   MCS p-values
--------  ---  ----  -----  ------  ------  |  ----------  ---  -----  -----  -----  ----
↪-  |  --------  ------------
   0      nan  0.49  0.618  -0.006  -1.339  |      0       nan   2.8    2.4    0.0    0.
↪6   |     0         0.801
                                           |                    ***    ***    ***    ***␣
↪    |
   1      nan  nan   0.294  -0.028  -1.402  |      1       nan   nan    4.6    0.0    0.
↪3   |     1         0.932
                                           |                          ***    ***    ***␣
↪    |
   2      nan  nan    nan   -0.03   -1.406  |      2       nan   nan    nan    0.0    0.
↪3   |     2          1.0
                                           |                                 ***    ***␣
↪    |
   3      nan  nan    nan    nan    -0.444  |      3       nan   nan    nan    nan    0.
↪1   |     3         0.976
                                           |                                        ***␣
↪    |
```

(continues on next page)

(continued from previous page)

```
   4      nan  nan    nan    nan     nan    |      4       nan   nan    nan    nan   ␣
↪nan   |     4         0.418
                                            |                                        ␣
↪    |
================================================================================================================

Testing Results - Instruments z1:
================================================================================================================
  TRV:                                      |   F-stats:                                ␣
↪      |    MCS:
--------  ---  -----  -----  ------  ------  |  ----------  ---  -----  -----  -----  ---
↪---  |  --------  ------------
 models    0      1      2      3       4    |    models     0      1      2      3     ␣
↪4     |   models   MCS p-values
--------  ---  -----  -----  ------  ------  |  ----------  ---  -----  -----  -----  ---
↪---  |  --------  ------------
   0      nan  0.047  0.243  -0.248  -1.601  |      0       nan   2.6    2.4    0.0    0.
↪8     |     0         0.808
                                            |                    ***    ***    ***    ␣
↪***     |
   1      nan   nan   1.663  -0.219  -1.523  |      1       nan   nan    3.3    0.1    0.
↪6     |     1         0.183
                                            |                           ***    ***    ␣
↪*** ^^  |
   2      nan   nan    nan   -0.25   -1.537  |      2       nan   nan    nan    0.0    0.
↪6     |     2          1.0
                                            |                                  ***    ␣
↪*** ^   |
   3      nan   nan    nan    nan    -1.971  |      3       nan   nan    nan    nan    2.
↪5     |     3         0.913
                                            |                                        ␣
↪***     |
   4      nan   nan    nan    nan     nan    |      4       nan   nan    nan    nan   ␣
↪nan    |     4         0.128
                                            |                                        ␣
↪     |
================================================================================================================
Significance of size and power diagnostic reported below each F-stat
*, **, or *** indicate that F > cv for a worst-case size of 0.125, 0.10, and 0.075 given␣
↪d_z and rho
^, ^^, or ^^ indicate that F > cv for a best-case power of 0.50, 0.75, and 0.95 given d_
↪z and rho
appropriate critical values for size are stored in the variable F_cv_size_list of the␣
↪pyRVtest results class
appropriate critical values for power are stored in the variable F_cv_power_list of the␣
↪pyRVtest results class
================================================================================================================
```

This output has a similar format to Table 5 in Duarte, Magnolfi, Sølvsten, and Sullivan (2023). Each testing results panel, corresponding to a set of instruments, reports in three separate blocks the RV test statistics for each pair of models, effective F-statistic for each pair of models, and MCS p-value for the row model. See the above description for an explanation of each.

In interpreting these results, note that instruments z_0 and z_1 contain 2 and 3 instruments respectively. Since the number of instruments is between 2-9, there are no size distortions above 0.025 for any model pair and instruments are strong for size. However, the instruments are weak for power as each pairwise F-statistic is below the critical value corresponding to best-case power of 0.95. Unsurprisingly then, no model is ever rejected from the MCS. These results reflect the fact that the data used in the tutorial do not provide the appropriate variation to test conduct. As such, the results should not be interpreted as to draw any conclusion about the nature of conduct in this empirical environment. We plan to include a more economically interesting example with future releases.

Testing with Taxes

Suppose now that we want to run the testing procedure in a setting with taxes (specifically ad valorem or unit taxes). Having taxes in the testing set up changes the first order conditions faced by the firms. To incorporate them using pyRVtest, we can specify additional options in the *ModelFormulation*:

- `unit_tax`: a vector of unit taxes, measured in the same units as price, which should correspond to a data column in the `product_data`.
- `advalorem_tax`: a vector of ad valorem tax rates, measured between 0 and 1, which should correspond to a data column in the `product_data`.
- `advalorem_payer`: party responsible for paying the advalorem tax. In our example this is the consumer (other options are 'firm' and 'None').

Here, we add ad valorem and unit tax data to the `product_data` for illustrative purposes.

[13]:
```
# additional variables for tax testing
product_data['unit_tax'] = .5
product_data['advalorem_tax'] = .5
product_data['lambda'] = 1
```

Next, initialize the testing problem. Note here that the first model formulation now specifies our additional variables.

[14]:
```
testing_problem_taxes = pyRVtest.Problem(
    cost_formulation=(
        pyRVtest.Formulation('0 + sugar', absorb='C(firm_ids)')
    ),
    instrument_formulation=(
        pyRVtest.Formulation('0 + demand_instruments0 + demand_instruments1')
    ),
    model_formulations=(
        pyRVtest.ModelFormulation(
            model_downstream='bertrand',
            ownership_downstream='firm_ids',
            cost_scaling='lambda',
            unit_tax='unit_tax',
            advalorem_tax='advalorem_tax',
            advalorem_payer='consumer'),
        pyRVtest.ModelFormulation(
            model_downstream='bertrand',
            ownership_downstream='firm_ids'
        ),
        pyRVtest.ModelFormulation(
            model_downstream='perfect_competition',
            ownership_downstream='firm_ids'
        )
```

(continues on next page)

(continued from previous page)

```
    ),
    product_data=product_data,
    demand_results=pyblp_results
)
testing_problem_taxes
```

[14]:
```
Dimensions:
=========================
 T    N     M    L   d_z0
---  ----  ---  ---  ----
94   2256   3    1    2
=========================

Formulations:
============================================================
Column Indices:            0                     1
----------------  ------------------  ------------------
w: Marginal Cost         sugar
z0: Instruments   demand_instruments0  demand_instruments1
============================================================

Models:
========================================================================
                            0           1              2
--------------------  -------------  --------  ------------------
 Model - Downstream      bertrand     bertrand  perfect_competition
  Model - Upstream         None         None           None
Firm IDs - Downstream    firm_ids     firm_ids       firm_ids
 Firm IDs - Upstream       None         None           None
      VI Index             None         None           None
 Cost Scaling Column      lambda        None           None
      Unit Tax           unit_tax       None           None
    Advalorem Tax      advalorem_tax    None           None
   Advalorem Payer       consumer       None           None
User Supplied Markups      None         None           None
========================================================================
```

As mentioned above, the Models table includes additional details related to testing with taxes:

- Unit Tax: the column in `product_data` corresponding to the unit tax
- Advalorem Tax : the column in `product_data` corresponding to the ad valorem tax
- Advalorem Payer: who is responsible for paying the tax

Finally, output the testing results.

[15]:
```
testing_results = testing_problem_taxes.solve(
    demand_adjustment=False,
    clustering_adjustment=False
)
testing_results
```

```
Computing Markups ...
Total Time is ... 0.17354202270507812
```

[15]:

```
Testing Results - Instruments z0:
=============================================================================================
  TRV:                          |   F-stats:                      |    MCS:
--------  ---  ------  ------   |  ----------  ---  ----  -----   |  --------  ------------
 models    0     1       2     |    models     0    1      2     |   models   MCS p-values
--------  ---  ------  ------   |  ----------  ---  ----  -----   |  --------  ------------
   0      nan  -1.936  -2.059   |      0       nan  -0.0   0.3    |     0          1.0
                                |                          ***    |
   1      nan   nan    -1.53    |      1       nan  nan    4.4    |     1          0.053
                                |                          ***    |
   2      nan   nan     nan     |      2       nan  nan    nan    |     2          0.078
                                |                                 |
=============================================================================================
Significance of size and power diagnostic reported below each F-stat
*, **, or *** indicate that F > cv for a worst-case size of 0.125, 0.10, and 0.075 given␣
↪d_z and rho
^, ^^, or ^^ indicate that F > cv for a best-case power of 0.50, 0.75, and 0.95 given d_
↪z and rho
appropriate critical values for size are stored in the variable F_cv_size_list of the␣
↪pyRVtest results class
appropriate critical values for power are stored in the variable F_cv_power_list of the␣
↪pyRVtest results class
=============================================================================================
```

Storing Results

If you are working on a problem that runs over the course of several days, PyBLP and pyRVtest make it easy for you to store your results.

For example, suppose that you want to estimate one demand system and then run multiple testing scenarios. In this case, you can simply use the PyBLP pickling method.

[16]:
```
pyblp_results.to_pickle('my_demand_estimates')
```

And read them in similarly:

[17]:
```
old_pyblp_results = pyblp.read_pickle('my_demand_estimates')
```

You can now use these demand estimates to run additional iterations of testing without having to re-estimate demand each time.

Additionally, if you want to save your testing results so that you can access them in the future, you can do so with the same method in pyRVtest.

[18]:
```
testing_results.to_pickle('my_testing_results')
```

And read them back in to analyze:

[19]:
```
old_testing_results = pyRVtest.read_pickle('my_testing_results')
old_testing_results.markups
```

[19]:
```
[array([[0.03616274],
        [0.02752501],
        [0.04300875],
        ...,
        [0.03948227],
        [0.02837644],
        [0.04313683]]),
 array([[0.03616274],
        [0.02752501],
        [0.04300875],
        ...,
        [0.03948227],
        [0.02837644],
        [0.04313683]]),
 array([[0.],
        [0.],
        [0.],
        ...,
        [0.],
        [0.],
        [0.]])]
```

2.2 Library of Models

[1]:
```
import pyRVtest
pyRVtest.__version__
```

[1]:
```
'0.2.0'
```

Here, we document the library of models that is currently supported by pyRVtest and how the user can specify them as a *ModelFormulation*. The current library of models includes:

- Bertrand Competition with Differentiated Products
- Cournot Competition with Differentiated Products
- Monopoly (i.e., Perfect Collusion)
- Bertrand and Cournot Competition with Profit Weights
- Non-Profit Conduct Models
- Marginal Cost Pricing (i.e., zero markup models)
- Rule-of-Thumb Models (i.e., markups as a fixed percentage of price or cost)
- Bertrand with Scaled Costs
- Constant Markup Models (that do not vary with demand or cost)
- Vertical Models

We also detail two options for how a user can test models outside this class.

2.2.1 Preliminaries and notation

Throughout, we consider settings in which potentially multi-product firms, indexed by f compete across markets, indexed by t. In each market, there are J_t products offered. Each firm f offers a distinct subset of those products, $\mathcal{J}_{ft}$. We use the jt index to denote observations at the product-market level, and we use the t index to denote the vector which stacks all J_t observations in market t. Demand for product j in market t, which depends on p_t, the price of all products in the market, is denoted $s_{jt}(p_t)$. Realizations of market shares across products in market t at the equilibrium prices p_t are denoted s_t. We denote the $J_t \times J_t$ matrix of own- and cross-price derivatives as $\frac{\partial s_t}{\partial p_t}$, so that the (j,k)-th element denotes the marginal effect of an increase in the price of product k on the market share of product j.

The researcher observes, in each market t, realizations of equilibrium prices and shares for a true model of conduct which generated the data: p_t and s_t. The researcher does not know the true model, but wishes to test a menu of models, M. For each model m in the menu, the stacked first order condition in market t can be expressed as:

$$p_t - c_{mt} = \Delta_{mt} \tag{2.1}$$

Here, Δ_{mt} are the stacked markups implied by model m in market t. For the models we consider, the markups Δ_{mt} can be expressed as known functions of prices and exogenous variables and typically depend on the demand system. Thus, given equilibrium outcomes, the known market structure (i.e., which firm sells which products), and the demand system, Δ_{mt} can be computed. c_{mt} are the marginal costs implied by the model which satisfy the system of first order conditions.

To specify a menu of models in pyRVtest, the researcher creates a *ModelFormulation* for each of the models in the menu when defining the testing problem. In what follows, we show how to specify the *ModelFormulation* for the class of models that the code can currently handle.

2.2.2 Bertrand Competition: $m = B$

Suppose the researcher wants to include in the menu of models the Bertrand-Nash model of competition in prices. In market t, firm f sets prices for all $j \in \mathcal{J}_{ft}$ to maximize the sum of its profits across those products. Letting p_{ft} be the vector of those prices, the firm solves:

$$\max_{p_{ft}} \sum_{j \in \mathcal{J}_{ft}} (p_{jt} - c_{jt}) s_{jt}(p_t) \tag{2.2}$$

The stacked first-order conditions across firms in market t can be written as $(p_t - c_t) = \Delta_{Bt}$ where Δ_{Bt} is:

$$\Delta_{Bt} \quad = \quad \left(\Omega_t \odot \frac{\partial s_t}{\partial p_t} \right)^{-1} s_t. \tag{2.3}$$

Here, Ω_t is the standard $J_t \times J_t$ ownership matrix so that the (j,k)-th element is 1 if products j and k are sold by the same firm, and 0 otherwise. Furthermore, $\odot$ denotes element-by-element multiplication.

To include Bertrand as one of the models to test with pyRVtest, the researcher must include in the `product_data` a column for which each row indicates the identity of the firm selling the corresponding product. If that column is named `firm_ids`, then the Bertrand model can be specified with the following `ModelFormulation`.

[2]:
```
model_formulation = pyRVtest.ModelFormulation(model_downstream='bertrand', ownership_
↪downstream='firm_ids')
```

Note that we label the options `model_downstream` and `ownership_downstream` as the code also accommodates vertical models. See the example below.

2.2.3 Monopoly (i.e., Perfect Collusion): $m = M$

Now suppose that in market t, prices p_{jt} are chosen to maximize the sum of its profits across all products in market t, or:

$$\max_{p_t} \sum_{j \in \mathcal{J}_t} (p_{jt} - c_{jt}) s_{jt}(p_t) \tag{2.4}$$

This can arise in settings with more than one firm if the firms are perfectly colluding. The stacked first-order conditions across firms in market t can be written as $(p_t - c_t) = \Delta_{Mt}$ where Δ_{Mt} is:

$$\Delta_{Mt} \quad = \quad \left(1_t \odot \frac{\partial s_t}{\partial p_t}\right)^{-1} s_t. \tag{2.5}$$

Here, 1_t is a $J_t \times J_t$ matrix of ones.

The researcher can specify Monopoly as one of the models to test with pyRVtest, using the following `ModelFormulation`. Note that here, if the researcher was to include a ownership_downstream option, the package will override this and build the ownership matrix in each market as a matrix of ones.

[3]:
```
model_formulation = pyRVtest.ModelFormulation(model_downstream='monopoly')
```

2.2.4 Cournot Competition with Differentiated Products: $m = C$

Suppose the researcher wants to include in the menu of models Cournot competition (quantity-setting) with differentiated products, in which each firm simultaneously chooses market shares for its $\mathcal{J}_{ft}$ products, s_{ft}, to maximize:

$$\max_{s_{ft}} \sum_{j \in \mathcal{J}_{ft}} s_{jt}(p_{jt}(s_t) - c_{jt}) \tag{2.6}$$

where $p_{jt}(s_t)$ represents inverse demand for product j in market t. The stacked first-order conditions are $p_{jt} - c_{jt} = \Delta_{Ct}$ where Δ_{Ct} is:

$$\Delta_{Ct} \quad = \quad \left(\Omega_t \odot \left(\frac{\partial s_t}{\partial p_t}\right)^{-1}\right) s_t. \tag{2.7}$$

Here, Ω_t is the standard $J_t \times J_t$ ownership matrix so that the (j, k)-th element is 1 if product j and k are sold by the same firm, and 0 otherwise. Furthermore, $\odot$ denotes element-by-element multiplication.

To include Cournot as one of the models to test with pyRVtest, the researcher must include in the `product_data` a column for which the i-th row indicates the identity of the firm selling the corresponding product. If that column is named `firm_ids`, then the Cournot model can be specified with the following `ModelFormulation`.

[4]:
```
model_formulation = pyRVtest.ModelFormulation(model_downstream='cournot', ownership_
↪downstream='firm_ids')
```

2.2.5 Bertrand and Cournot with Profit Weights: $m = PW(\lambda)$

Suppose the researcher wants to include in the menu of models either price or quantity competition in which the firms partially internalize the effect of their actions on their rivals' profits. This can occur, for example, in a model of common ownership (e.g., *Backus, Conlon, and Sinkinson (2022)*) or imperfect collusion (e.g., *Miller and Weinberg (2017)*). Now, the first order conditions of the Bertrand and Cournot models contain an ownership matrix specified by the user for which the (j, k)-th element is λ_{jk}.

For example, to include Bertrand with a given set of profit weights as one of the models to test with pyRVtest, the researcher specifies within the `ModelFormulation`, a `kappa_specification` as used in the `build_ownership` function in PyBLP:

[5]:
```
kappa_specification = 1
model_formulation = pyRVtest.ModelFormulation(model_downstream='bertrand', ownership_
↪downstream='firm_ids', kappa_specification_downstream = kappa_specification)
```

Likewise, to include Cournot with the given profit weights, the `ModelFormulation` is:

[6]:
```
model_formulation = pyRVtest.ModelFormulation(model_downstream='cournot', ownership_
↪downstream='firm_ids', kappa_specification_downstream = kappa_specification)
```

2.2.6 Non-profit Conduct: $m = N(\lambda)$

Suppose the researcher wants to model non-profit conduct where firms choose their own prices to maximize a weighted sum of profit and consumer surplus as in (*Duarte, Magnolfi, and Roncoroni (2022)*) where non-profit firms place a weight of $1 - \lambda \in (0, 1)$ on welfare relative to profit. This can be achieved by augmenting the first order conditions of the Bertrand model above. Specifically, one adjusts Ω_t by setting the (j, k)-th element of the ownership matrix to $1/\lambda_{jk}$ if the firm selling products j and k is a non-profit.

For example, to test a model in which non-profit firms have given welfare weights with pyRVtest, the researcher specifies within the *ModelFormulation*, a `kappa_specification` encoding those weights as used in the `build_ownership` function in PyBLP:

[7]:
```
model_formulation = pyRVtest.ModelFormulation(model_downstream='bertrand', ownership_
↪downstream='firm_ids', kappa_specification_downstream = kappa_specification)
```

2.2.7 Marginal Cost Pricing (Perfect Competition / Zero Markup): $m = PC$

Consider a class of models where firms selling differentiated products set prices equal to marginal costs so that markups are zero, $\Delta_{PCt} = 0$.

For example, to include marginal cost pricing in the menu of models to test with pyRVtest, the researcher includes the following *ModelFormulation*. Note that here, if the researcher was to include a `ownership_downstream` option, the package will override this set markups to zero.

[8]:
```
model_formulation = pyRVtest.ModelFormulation(model_downstream='perfect_competition')
```

2.2.8 Rule of Thumb Models: $m = R(\lambda)$

Consider a class of models where markups are a fraction λ of price or cost. For example, suppose the firm sets prices according to the rule-of-thumb: $p_t = (1 + \lambda)c_t$. In this case, the stacked first order conditions are $p_t - c_t = \Delta_{Rt}$ where

$$\Delta_{Rt} = \lambda c_t \tag{2.8}$$

Equivalently, markups can be equivalently expressed as a function of prices, or

$$\Delta_{Rt} = \frac{\lambda}{1 + \lambda} p_t \tag{2.9}$$

For now, pyRVtest only accommodates models where λ is constant across firms and markets.

For example, to include model of rule-of-thumb λ, the researcher specifies within the *ModelFormulation*, a `cost_scaling` option equal to a column in the `product_data` containing the scalar value `lmbda` (If markups are equal to cost or equivalently 50% of price, then `lmbda` = 1. Instead, if markups are equal to 50% of cost or equivalently 1/3 of price, then `lmbda` = 0.5)

[9]:
```
lmbda = 'cost_scaling_column'
model_formulation = pyRVtest.ModelFormulation(model_downstream='perfect_competition',␣
↪cost_scaling=lmbda)
```

2.2.9 Bertrand with Scaled Costs: $m = SC(\lambda)$

Next, consider a class of models where firms choose their own prices in market t to solve:

$$\max_{p_{ft}} \sum_{j \in \mathcal{J}_{ft}} (p_{jt} - \lambda c_{jt}) s_{jt}(p_t) \tag{2.10}$$

The stacked first-order conditions across firms in market t can be written as $(p_t - c_t) = \Delta_{SCt}$ where

$$\Delta_{SCt} = \Delta_{Bt} + (1 - \lambda) c_t \tag{2.11}$$

and Δ_{Bt} are the Bertrand markups.

Markups of this form arise in the model of collusion considered in *Harrington (2023)*, where firms collude via cost coordination. These markups also arise in settings where two firms compete a la Bertrand in prices, but each one maximizes a weighted sum of profits and revenues, where $\frac{1-\lambda}{\lambda}$ is the weight put on revenue relative to profit (e.g., *Baumol (1958)*).

For now, pyRVtest only accommodates models where λ is constant across firms and markets.

For example, to include model with cost-scaling λ, the researcher specifies within the `ModelFormulation`, a `cost_scaling` option equal to the scalar value `lmbda`:

[10]:
```
model_formulation = pyRVtest.ModelFormulation(model_downstream='bertrand', ownership_
↪downstream='firm_ids', cost_scaling=lmbda)
```

2.2.10 Constant Markup Models: $m = CM(\eta)$

Next, consider a class of models where the markup for each product j in each market t is equal to a constant η_{jt} which does not depend on demand or cost, or

$$\Delta_{CMjt} = \eta_{jt} \tag{2.12}$$

The researcher can include a constant markup model in the menu to be tested by using within `ModelFormulation` the `user_specified_markups` option. Here, the user creates a column in `product_data` where each row is the value η_{jt} corresponding to product j in market t. If this column is named `eta`, the `ModelFormulation` is:

[11]:
```
model_formulation = pyRVtest.ModelFormulation(user_supplied_markups='eta')
```

2.2.11 Vertical Models with unobserved wholesale costs

Now we consider models with vertical interactions between retailers and wholesalers (we denote each with superscript r and w respectively). Consider the simple linear pricing model from *Villas-Boas (2007)* where wholesalers individually set wholesale prices p_t^w to maximize their profits in market t, and, given p_t^w, retailers choose retailer prices p_t^r to maximize their profits in the same market. Assuming that wholesale prices are unobserved, we can sum the stacked first order conditions for wholesalers and retailers to obtain:

$$p_t^r - c_t^r - c_t^w = \Delta_{Bt}^r + \Delta_{Bt}^w \tag{2.13}$$

where Δ^r_{Bt}, the vector of retailer Bertrand markups, are:

$$\Delta^r_{Bt} = -\left(\Omega^r_t \odot \frac{\partial s_t}{\partial p^r_t}\right)^{-1} s_t \tag{2.14}$$

and Δ^w_{Bt}, the vector of wholesaler Bertrand markups, are:

$$\Delta^w_{Bt} - \left(\Omega^w_t \odot \frac{\partial s_t}{\partial p^w_t}\right)^{-1} s_t \tag{2.15}$$

If a researcher wants to include linear pricing in the menu of models to be tested, she must specify as columns of `product_data` both retailer ids and wholesaler ids. Assuming these are called, respectively, `retailer_ids` and `wholesaler_ids`, the following `ModelFormulation` is used:

[12]:
```
model_formulation = pyRVtest.ModelFormulation(model_downstream='bertrand', ownership_
↪downstream='retailer_ids', model_upstream='bertrand', ownership_upstream='wholesaler_
↪ids')
```

One could allow for a different model upstream or downstream by changing the name of `model_downstream` and `model_upstream` (for example, in the context of consumer packaged goods market, if each store is a market, then the downstream model is monopoly). One can also allow for profit weights or non-profit conduct in Ω^r_t and Ω^w_t by specifying `kappa_specification_downstream` and `kappa_specification_upstream`. A *ModelFormulation* can also accommodate partial vertical integration in a market using the `vertical_integration` option. In this case, the `product_data` must contain a column which equals one if the product is vertically integrated in the given market and zero otherwise. Supposing this column is named vi_id, the linear pricing model with partial vertical integration can be specified with the following *ModelFormulation*

[13]:
```
model_formulation = pyRVtest.ModelFormulation(model_downstream='bertrand', ownership_
↪downstream='retailer_ids', model_upstream='bertrand', ownership_upstream='wholesaler_
↪ids', vertical_integration='vi_id')
```

2.2.12 Options to Test Other Models

If the user wants to include in the menu models not included in the current library, there are two options.

First, if the model of conduct implies a vector of markups which are a known function of the ownership matrix, the response matrix, or shares, then the user can pass that function using the `custom_model_specification` input for a *ModelFormulation*. This input takes a dictionary where the key is the custom model name, and the value is a string formula to be evaluated. For example, if we wanted to test the Bertrand markups using this option (assuming Bertrand was not already part of our menu of models), we would write:

[14]:
```
model_formulation = pyRVtest.ModelFormulation(model_downstream='other', custom_model_
↪specification={'bertrand': '-inv(ownership_matrix * response_matrix) @ shares'})
```

Otherwise, if the markups the user wishes to test are not a known function of the ownership matrix, the response matrix, or shares, the user can pass an arbitrary vector using the `user_specified_markups` option as illustrated for the Constant Markup Models above. For example, if the user creates a column in the `product_data` called `my_markups`, the user can test these markups by defining the following *ModelFormulation*:

[15]:
```
model_formulation = pyRVtest.ModelFormulation(user_supplied_markups='my_markups')
```

Please note, when using the option `user_specified_markups`, **the code is not able to adjust** the pairwise RV test statistics, the model confidence set, and the pairwise F-statistics for the errors coming from demand estimation or for clustering.

CHAPTER

THREE

API DOCUMENTATION

The majority of the package consists of classes, which compartmentalize different aspects of testing models of firm conduct.

There are some convenience functions as well.

3.1 Configuration Classes

Formulation(formula[, absorb, ...])	Configuration for designing matrices and absorbing fixed effects.
ModelFormulation([model_downstream, ...])	Configuration for designing matrices and absorbing fixed effects.

3.1.1 pyRVtest.Formulation

class pyRVtest.**Formulation**(*formula*, *absorb=None*, *absorb_method=None*, *absorb_options=None*)

Configuration for designing matrices and absorbing fixed effects.

Note: This class is a copy of the Formulation class from PyBLP.

Internally, the patsy package is used to convert data and R-style formulas into matrices. All of the standard binary operators can be used to design complex matrices of factor interactions:

- `+` - Set union of terms.
- `-` - Set difference of terms.
- `*` - Short-hand. The formula `a * b` is the same as `a + b + a:b`.
- `/` - Short-hand. The formula `a / b` is the same as `a + a:b`.
- `:` - Interactions between two sets of terms.
- `**` - Interactions up to an integer degree.

However, since factors need to be differentiated (for example, when computing elasticities), only the most essential functions are supported:

- `C` - Mark a variable as categorical. See `patsy.builtins.C()`. Arguments are not supported.
- `I` - Encapsulate mathematical operations. See `patsy.builtins.I()`.
- `log` - Natural logarithm function.

- `exp` - Natural exponential function.

Data associated with variables should generally already be transformed. However, when encapsulated by `I()`, these operators function like normal mathematical operators on numeric variables: `+` adds, `-` subtracts, `*` multiplies, `/` divides, and `**` exponentiates.

Internally, mathematical operations are parsed and evaluated by the SymPy package, which is also used to symbolically differentiate terms when derivatives are needed.

Parameters

- **formula** (*str*) – R-style formula used to design a matrix. Variable names will be validated when this formulation and data are passed to a function that uses them. By default, an intercept is included, which can be removed with `0` or `-1`. If `absorb` is specified, intercepts are ignored.
- **absorb** (*str, optional*) – R-style formula used to design a matrix of categorical variables representing fixed effects, which will be absorbed into the matrix designed by `formula` by the PyHDFE package. Fixed effect absorption is only supported for some matrices. Unlike `formula`, intercepts are ignored. Only categorical variables are supported.
- **absorb_method** (*str, optional*) – Method by which fixed effects will be absorbed. For a full list of supported methods, refer to the `residualize_method` argument of `pyhdfe.create()`.

  By default, the simplest methods are used: simple de-meaning for a single fixed effect and simple iterative de-meaning by way of the method of alternating projections (MAP) for multiple dimensions of fixed effects. For multiple dimensions, non-accelerated MAP is unlikely to be the fastest algorithm. If fixed effect absorption seems to be taking a long time, consider using a different method such as `'lsmr'`, using `absorb_options` to specify a MAP acceleration method, or configuring other options such as termination tolerances.
- **absorb_options** (*dict, optional*) – Configuration options for the chosen `method`, which will be passed to the `options` argument of `pyhdfe.create()`.

3.1.2 pyRVtest.ModelFormulation

class pyRVtest.**ModelFormulation**(*model_downstream=None*, *model_upstream=None*, *ownership_downstream=None*, *ownership_upstream=None*, *custom_model_specification=None*, *vertical_integration=None*, *unit_tax=None*, *advalorem_tax=None*, *advalorem_payer=None*, *cost_scaling=None*, *kappa_specification_downstream=None*, *kappa_specification_upstream=None*, *user_supplied_markups=None*)

Configuration for designing matrices and absorbing fixed effects.

For each model, the user can specify the downstream and upstream (optional) models, the downstream and upstream ownership structure, a custom model and markup formula, and vertical integration. The user can also choose to forgo markup computation and specify their own markups with *user_supplied_markups*. Additionally, there are specifications related to testing conduct with taxes.

There is a built-in library of models that the researcher can choose from.

Here, we have another difference with PyBLP. In PyBLP, if one wants to build an ownership matrix, there must be a variable called *firm_id* in the *product_data*. With pyRVtest, the researcher can pass any variable in the *product_data* as *ownership_downstream* and from this, the ownership matrix in each market will be built.

Note: We are working on adding additional models to this library as well as options for the researcher to specify their own markup function.)

Parameters

- **model_downstream** (*str, optional*) – The model of conduct for downstream firms (or if no vertical structure, the model of conduct). One of "bertrand", "cournot", "monopoly", "perfect_competition", or "other".
- **model_upstream** (*str, optional*) – The model of conduct for upstream firms. One of "bertrand", "cournot", "monopoly", "perfect_competition", or "other".
- **ownership_downstream** (*str, optional*) – Column indicating which firm ids to use for ownership matrix construction for downstream firms.
- **ownership_upstream** (*str, optional*) – Column indicating which firm ids to use for ownership matrix construction for upstream firms.
- **custom_model_specification** (*dict, optional*) – A dictionary containing an optional custom markup formula specified by the user. The specified function must consist of objects computed within the package.
- **vertical_integration** (*str, optional*) – The column name for the data column which indicates the vertical ownership structure.
- **unit_tax** (*str, optional*) – The column name for the vector containing information on unit taxes.
- **advalorem_tax** (*str, optional*) – The column name for the vector containing information on advalorem taxes.
- **advalorem_payer** (*str, optional*) – A string indicating who pays for the advalorem tax in the given model.
- **cost_scaling** (*str, optional*) – The column name for the cost scaling parameter.
- **kappa_specification_downstream** (*Union[str, Callable[[Any, Any], float]]], optional*) – Information on the degree of cooperation among downstream firms for each market.
- **kappa_specification_upstream** (*Union[str, Callable[[Any, Any], float]]], optional*) – Information on the degree of cooperation among upstream firms for each market.
- **user_supplied_markups** (*str, optional*) – The name of the column containing user-supplied markups.

3.2 Data Manipulation Functions

There are also a number of convenience functions that can be used to compute markups, or manipulate other pyRVtest objects.

build_markups(products, demand_results, ...)	This function computes markups for a large set of standard models.
read_pickle(path)	Load a pickled object into memory.

3.2.1 pyRVtest.build_markups

pyRVtest.**build_markups**(*products*, *demand_results*, *model_downstream*, *ownership_downstream*, *model_upstream=None*, *ownership_upstream=None*, *vertical_integration=None*, *custom_model_specification=None*, *user_supplied_markups=None*)

This function computes markups for a large set of standard models.

The models that this package is able to compute markups for include:

- standard bertrand with ownership matrix based on firm id
- price setting with arbitrary ownership matrix (e.g. profit weight model)
- standard cournot with ownership matrix based on firm id
- quantity setting with arbitrary ownership matrix (e.g. profit weight model)
- monopoly
- bilateral oligopoly with any combination of the above models upstream and downstream
- bilateral oligopoly as above but with subset of products vertically integrated
- any of the above with consumer surplus weights

In order to compute markups, the products data and PyBLP demand estimation results must be specified, as well as at least a model of downstream conduct. If *model_upstream* is not specified, this is a model without vertical integration.

Parameters

- **products** (*recarray*) – The *product_data* containing information on markets and product characteristics. This should be the same as the data used for demand estimation. To compute markups, this data must include *prices*, *market_ids*, and *shares*.
- **demand_results** (*structured array-like*) – The results object obtained from using the pyBLP demand estimation procedure. We use built-in PyBLP functions to return the demand Jacobians and Hessians (first and second derivatives of shares with respect to prices).
- **model_downstream** (*ndarray*) – The model of conduct for downstream firms. Can be one of [*bertrand*, *cournot*, *monopoly*, *perfect_competition*, *other*]. Only specify option *other* if supplying a custom markup formula.
- **ownership_downstream** (*ndarray*) – The ownership matrix for price or quantity setting (optional, default is standard ownership).
- **model_upstream** (*ndarray, optional*) – Upstream firm model of conduct. Only specify option *other* if supplying a custom markup formula. Can be one of ['none' (default), *bertrand*, *cournot*, *monopoly*, *perfect_competition*, *other*].
- **ownership_upstream** (*ndarray, optional*) – Ownership matrix for price or quantity setting of upstream firms (optional, default is None).
- **vertical_integration** (*ndarray, optional*) – Vector indicating which *product_ids* are vertically integrated (ie store brands) (optional, default is None).
- **custom_model_specification** (*dict, optional*) – Dictionary containing a custom markup formula and the name of the formula (optional, default is None).
- **user_supplied_markups** (*ndarray, optional*) – Vector containing user-computed markups (optional, default is None). If user supplied own markups, this function simply returns them.

Returns

. Computed markups, downstream markups, and upstream markups for each model.

Return type
tuple[list, list, list]

Notes

For models without vertical integration, firm_ids must be defined in product_data for vi models, and firm_ids_upstream and firm_ids (=firm_ids_downstream) must be defined.

3.2.2 pyRVtest.read_pickle

pyRVtest.**read_pickle**(*path*)

Load a pickled object into memory. This is a simple wrapper around *pickle.load*.

Parameters
path (*str or Path*) – File path of a pickled object.

Returns
The unpickled object.

Return type
object

3.3 Problem Class

Problem(cost_formulation, ...[, ...])	A firm conduct testing-type problem.

3.3.1 pyRVtest.Problem

class pyRVtest.**Problem**(*cost_formulation*, *instrument_formulation*, *product_data*, *demand_results*, *model_formulations=None*, *markup_data=None*)

A firm conduct testing-type problem.

This class is initialized using the relevant data and formulations, and solved with *Problem.solve()*.

Parameters

- **cost_formulation** (*Formulation*) – *Formulation* is a list of the variables for observed product characteristics. All observed cost shifters included in this formulation must be variables in the *product_data*. To use a constant, one would replace *0* with *1*. To absorb fixed effects, specify *absorb = 'C(variable)'*, where the *variable* must also be in the *product_data*. Including this option implements fixed effects absorption using [PYHDFE](https://github.com/jeffgortmaker/pyhdfe), a companion package to PyBLP.
- **instrument_formulation** (*Formulation or sequence of Formulation*) – *Formulation* is list of the variables used as excluded instruments for testing. For each instrument formulation, there should never be a constant. The user can specify as many instrument formulations as desired. All instruments must be variables in *product_data*.

Note: **Our instrument naming conventions differ from PyBLP**. With PyBLP, one specifies the excluded instruments for demand estimation via a naming convention in

the product_data: each excluded instrument for demand estimation begins with *"demand_instrument"* followed by a number (i.e., *demand_instrument0*). In pyRVtest, you specify directly the names of the variables in the *product_data* that you want to use as excluded instruments for testing (i.e., if you want to test with one instrument using the variable in the *product_data* named, "transportation_cost" one could specify *pyRVtest.Formulation('0 + transportation_cost')*.

- **model_formulations** (*sequence of ModelFormulation*) – *ModelFormulation* defines the models that the researcher wants to test. There must be at least two instances of *ModelFormulation* specified to run the firm conduct testing procedure.
- **product_data** (*structured array-like*) – This is the data containing product and market observable characteristics, as well as instruments.
- **demand_results`** (*structured array-like*) – The results object returned by *pyblp.solve*.

Methods

solve([demand_adjustment, clustering_adjustment])	Solve the problem.

Once initialized, the following method solves the problem.

Problem.solve([demand_adjustment, ...])	Solve the problem.

3.3.2 pyRVtest.Problem.solve

Problem.**solve**(*demand_adjustment=False*, *clustering_adjustment=False*)

Solve the problem.

Given demand estimates from PyBLP, we compute implied markups for each model m being tested. Marginal cost is a linear function of observed cost shifters and an unobserved shock.

The rest of the testing procedure is done for each pair of models, for each set of instruments. A GMM measure of fit is computed for each model-instrument pair. This measure of fit is used to construct the test statistic.

Parameters

- **demand_adjustment** (*Optional[bool]*) – (optional, default is False) Configuration that allows user to specify whether to compute a two-step demand adjustment. Options are True or False.
- **clustering_adjustment** (*Optional[str]*) – (optional, default is unadjusted) Configuration that specifies whether to compute clustered standard errors. Options are True or False.

Returns

ProblemResults of the solved problem.

Return type

ProblemResults

3.4 Problem Results Class

Solved problems return the following results class.

ProblemResults	Results of running the firm conduct testing procedures.

3.4.1 pyRVtest.ProblemResults

class pyRVtest.**ProblemResults**

Results of running the firm conduct testing procedures.

problem

An instance of the Problem class.

Type
ndarray

markups

Array of the total markups implied by each model (sum of retail and wholesale markups).

Type
ndarray

markups_downstream

Array of the retail markups implied by each model.

Type
ndarray

markups_upstream

Array of the manufacturer markups implied by each model of double marginalization.

Type
ndarray

taus

Array of coefficients from regressing implied marginal costs for each model on observed cost shifters.

Type
ndarray

mc

Array of implied marginal costs for each model.

Type
ndarray

g

Array of moments for each model and each instrument set of conduct between implied residualized cost unobservable and the instruments.

Type
ndarray

Q

Array of lack of fit given by GMM objective function with 2SLS weight matrix for each set of instruments and each model.

Type
ndarray

`RV_numerator`

Array of numerators of pairwise RV test statistics for each instrument set and each pair of models.

Type
ndarray

`RV_denominator`

Array of denominators of pairwise RV test statistics for each instrument set and each pair of models.

Type
ndarray

`TRV`

Array of pairwise RV test statistics for each instrument set and each pair of models.

Type
ndarray

`F`

Array of pairwise F-statistics for each instrument set and each pair of models.

Type
ndarray

`MCS_pvalues`

Array of MCS p-values for each instrument set and each model.

Type
ndarray

`rho`

Scaling parameter for F-statistics.

Type
ndarray

`unscaled_F`

Array of pairwise F-statistics without scaling by rho.

Type
ndarray

`F_cv_size_list`

Vector of critical values for size for each pairwise F-statistic.

Type
ndarray

`F_cv_power_list`

Vector of critical values for power for each pairwise F-statistic.

Type
ndarray

Methods

to_pickle(path)	Save these results as a pickle file.

The results can be pickled or converted into a dictionary.

ProblemResults.to_pickle(path)	Save these results as a pickle file.

3.4.2 pyRVtest.ProblemResults.to_pickle

ProblemResults.**to_pickle**(*path*)

Save these results as a pickle file. This function is copied from PyBLP.

Parameters

path (*str or Path*) – File path to which these results will be saved.

CHAPTER

FOUR

REFERENCES

This page contains a full list of references cited in the documentation. If you use this package in your research please also cite references: Duarte, Magnolfi, Sølvsten, and Sullivan (2023).

4.1 Duarte, Magnolfi, Sølvsten, and Sullivan (2023)

Duarte, Marco, Lorenzo Magnolfi, Mikkel Sølvsten, and Christopher Sullivan (2023): "Testing Firm Conduct," Working paper.

4.2 Other References

4.2.1 Baumol (1958)

Baumol, William (1958): "On the Theory of Oligopoly," *Economica,* 25(99), 187-198.

4.2.2 Backus, Conlon, and Sinkinson (2022)

Backus, Matthew, Christopher Conlon, and Michael Sinkinson (2021): "Common Ownership and Competition in the Ready-to-Eat Cereal Industry," NBER Working Paper No. 28350.

4.2.3 Conlon and Gortmaker (2020)

Conlon, Christopher, and Jeff Gortmaker (2020): "Best Practices for Differentiated Products Demand Estimation with PyBLP," *RAND Journal of Economics,* 51(4), 1108-1161.

4.2.4 Duarte, Magnolfi, and Roncoroni (2022)

Duarte, Marco, Lorenzo Magnolfi, and Camilla Roncoroni (2022): "The Competitive Conduct of Consumer Cooperatives," Working paper.

4.2.5 Hansen, Lunde, and Nason (2011)

Hansen, Peter, Asger Lunde, and James Nason (2011): "The Model Confidence Set," *Econometrica,* 79(2), 453–497.

4.2.6 Harrington (2023)

Harrington, Joseph (2023): "Cost Coordination," Working paper.

4.2.7 Miller and Weinberg (2017)

Miller, Nathan and Matthew Weinberg (2017): "Understanding the Price Effects of the MillerCoors Joint Venture," *Econometrica,* 85(6), 1763–1791.

4.2.8 Nevo (2000)

Nevo, Aviv (2000): "A Practitioner's Guide to Estimation of Random-Coefficients Logit Models of Demand," *Journal of Economics & Management Strategy,* 9(4), 513-548.

4.2.9 Nevo (2001)

Nevo, Aviv (2001): "Measuring Market Power in the Ready-to-Eat Cereal Industry," *Econometrica,* 69(2), 307-342.

4.2.10 Rivers and Vuong (2002)

Rivers, Douglas and Quang Vuong (2002): "Model Selection Tests for Nonlinear Dynamic Models," *Econometrics Journal,* 5(1), 1-39.

4.2.11 Villas-Boas (2007)

Villas-Boas, Sofia (2007): "Vertical Relationships between Manufacturers and Retailers: Inference with Limited Data," *Review of Economic Studies*, 74(2), 625–652.

CHAPTER

FIVE

LEGAL

CHAPTER

SIX

INDICES AND TABLES

- genindex
- modindex
- search

INDEX