pyStoNED Documentation

Release 0.7.5

Sheng Dai, Yu-Hsueh Fang, Chia-Yen Lee, Timo Kuosmanen

Feb 22, 2025

CONTENTS

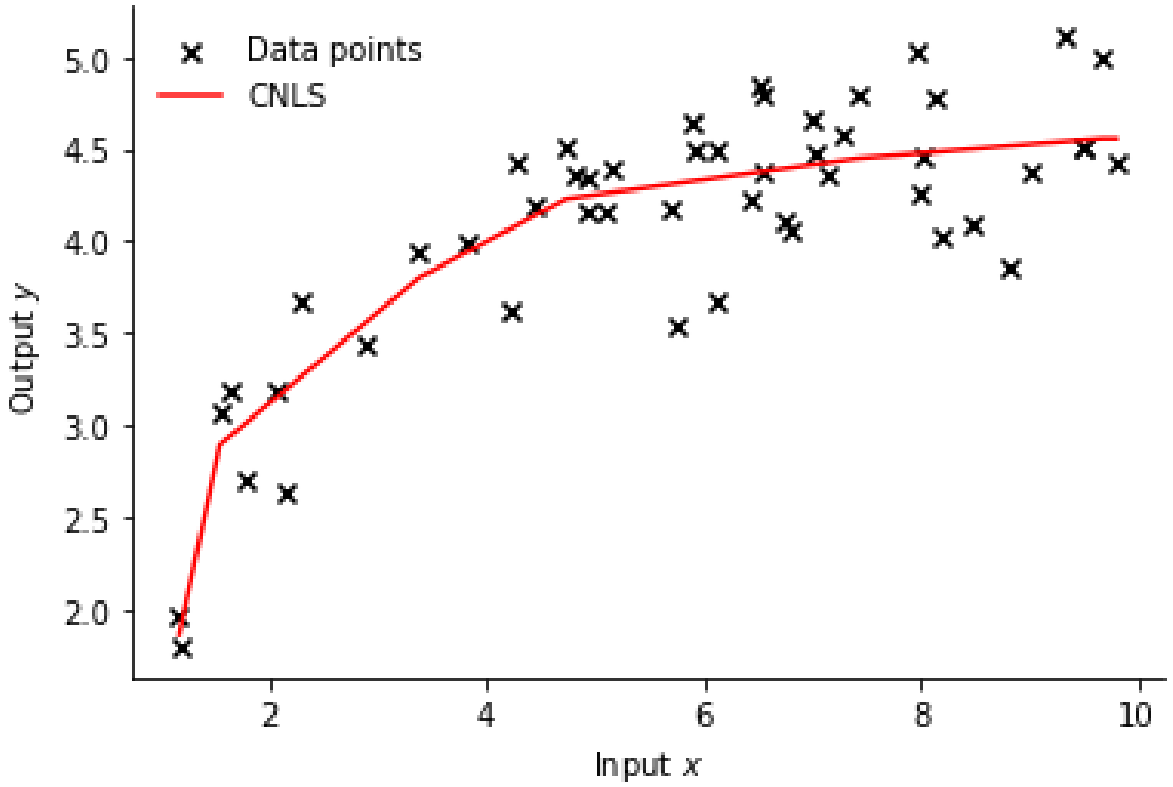
pyStoNED is a Python package that provides functions for estimating Convex Nonparametric Least Square (CNLS), Stochastic Nonparametric Envelopment of Data (StoNED), and other various StoNED-related variants such as Convex Quantile Regression (CQR), Convex Expectile Regression (CER), and Isotonic CNLS (ICNLS). It also provides efficiency measurement using Data Envelopement Analysis (DEA) and Free Disposal Hull (FDH). The pyStoNED package allows the user to estimate the CNLS/StoNED frontiers in an open-access environment and is built based on the Pyomo.

For example [.ipynb], the following code estimates the basic CNLS model and plot the production frontier.[1]

```
# import packages
import numpy as np
from pystoned import CNLS
from pystoned.plot import plot2d
from pystoned.constant import CET_ADDI, FUN_PROD, RTS_VRS

# set seed
np.random.seed(0)

# generate DMUs: DGP
x = np.sort(np.random.uniform(low=1, high=10, size=50))
u = np.abs(np.random.normal(loc=0, scale=0.7, size=50))
y_true = 3 + np.log(x)
y = y_true - u

# define the CNLS model
model = CNLS.CNLS(y, x, z=None, cet = CET_ADDI, fun = FUN_PROD, rts = RTS_VRS)
# solve the model with remote solver
model.optimize('email@address')

# display the residuals
model.display_residual()

# plot CNLS frontier
plot2d(model, x_select=0, label_name="CNLS", fig_name='CNLS_frontier')
```

The CNLS production frontier is shown as follows:

[1] Please read Solver in Installation before running the example.


Data points
CNLS
Output y
Input x
5.0
4.5
4.0
3.5
3.0
2.5
2.0
2
4
6
8
10

CHAPTER

ONE

INSTALLATION

pyStoNED supports Python 3.8 or later versions on Linux, macOS, and Windows. We can install the package using pip or GitHub.

1.1 PyPI

```
pip install pystoned
```

1.2 GitHub[1]

```
pip install -U git+https://github.com/ds2010/pyStoNED
```

1.3 Solver

All models supported by pyStoNED are either additive or multiplicative models, depending on the specification of the error term. From the perspective of optimization, the additive models are usually stated as QP problems with an exception of the CQR model, which is a linear programming (LP) problem, whereas all multiplicative models are NLP problems.

To solve the relevant QP or NLP problems, external off-the-shelf solvers are required. In our experience, `CPLEX` or `MOSEK` provide reliable and convenient platforms for solving the QP and LP problems. The NLP problem can be efficiently solved by `MINOS` or `KNITRO`. Note that the tailored algorithm for specific estimators could also be used to solve these models. However, tailored algorithms are not the most convenient choice for our general application purposes. With the help of Pyomo, all models supported by pyStoNED are computable by these off-the-shelf solvers.

- Remote solver

pyStoNED interfaces with Pyomo to access the network-enabled optimization system (NEOS) server that freely provides a large number of academic solvers for solving the additive or multiplicative models remotely. In this case, the users do not need to install solvers and corresponding licenses on their own computer. Here we have a model estimated by remote solver

```
model.optimize('email@address')
```

Replace the argument `email@address` with your email address.[2] By default, the additive and multiplicative models will be solved by `MOSEK` and `KNITRO`, respectively. In addition, the users can freely choose their preferred solver, e.g., `MINOS`, using

[1] The GitHub repo provides the latest development version.

[2] As of January 2021, NOES requires a valid email address in all submission; see NEOS Server FAQ.

```
model.optimize(email="email@address", solver='minos')
```

- Local solver

Pyomo also provides an application programming interface for pyStoNED to import the local solvers. In the pyStoNED package, `MOSEK` is attached as the internal dependency to solve the additive model. However, the `MOSEK` academic license is required to be installed.

```
## MOSEK Optimizer: License installation ##

1. Request Personal Academic License

   https://www.mosek.com/license/request/personal-academic/

2. Save the license file under the user's home directory
   with a folder named "mosek". e.g.,

   Windows: c:\users\xxxx\mosek\mosek.lic
   Unix/OS X: /home/xxxx/mosek/mosek.lic

   Note: xxxx is the User ID on the computer
```

Otherwise, the following error message will display on the screen when calculating CNLS/StoNED or other variants.

```
MOSEK error 1008: License cannot be located. The default search path is ':/
↪home/user/mosek/mosek.lic:'.
```

After that, we can use the following code to calculate the additive models.

```
model.optimize(OPT_LOCAL)
```

The parameter `OPT_LOCAL` is added in the function `.optimize(...)` to indicate that the model is computed locally. Similarly, one can use the parameter `solver` to select another solver when it is available.

Overall, the remote solver through the NEOS server is highly recommended for all light computing jobs, where, in general, the number of observations is no more than 500. The local solver for calculating the multiplicative model will be supported in pyStoNED when a free and stable NLP solver is available.

CHAPTER

TWO

EXAMPLES

Consider a standard multivariate, cross-sectional model in production economics:

$$\begin{aligned} y_i &= f(\boldsymbol{x}_i) + \varepsilon_i \\ &= f(\boldsymbol{x}_i) + v_i - u_i \quad \forall i \end{aligned} \tag{2.1}$$

where y_i is the output of unit i, $f : R_+^m \rightarrow R_+$ is the production function (cost function) that characterizes the production technology (cost technology), and $\boldsymbol{x}_i = (x_{i1}, x_{i2}, \cdots, x_{im})^{'}$ denotes the input vector of unit i. Similar to the literature in Stochastic Frontier analysis (SFA), the presented composite error term $\varepsilon_i = v_i - u_i$ consists of the inefficiency term $u_i > 0$ and stochastic noise term v_i. To estimate the function f, one could use the parametric and nonparametric methods or neoclassical and frontier models, of which methods are classified based on the specification of f and error term ε (see Kuosmanen and Johnson, 2010). In this paper, we assume certain axiomatic properties (e.g., monotonicity, concavity) instead of *a priori* functional form for the function f and apply the nonparametric methods to estimate the function f.

2.1 Convex Nonparametric Least Square

2.1.1 Additive CNLS model

Hildreth (1954) is the first to consider the nonparametric regression subject to monotonicity and concavity constraints in the case of a single input variable x. Kuosmanen (2008) extends Hildreth's approach to the multivariate setting with the multidimensional input $\boldsymbol{x}$, and refers it to as the Convex Nonparametric Least Squares (CNLS). CNLS builds upon the assumption that the true but unknown production function f belongs to the set of continuous, monotonic increasing and globally concave (convex) functions, imposing exactly the same production axioms as standard Data Envelopment Analysis (DEA) (see further discussion in Kuosmanen and Johnson, 2010). The additive multivariate CNLS formulations are defined as

- Esimating production function

$$\begin{aligned} \min_{\alpha,\boldsymbol{\beta},\varepsilon} & \sum_{i=1}^{n} \varepsilon_i^2 \\ \textit{s.t.} \quad & y_i = \alpha_i + \boldsymbol{\beta}_i^{'}\boldsymbol{x}_i + \varepsilon_i \qquad \forall i \\ & \alpha_i + \boldsymbol{\beta}_i^{'}\boldsymbol{x}_i \leq \alpha_j + \boldsymbol{\beta}_j^{'}\boldsymbol{x}_i \quad \forall i, j \\ & \boldsymbol{\beta}_i \geq \mathbf{0} \qquad \forall i \end{aligned} \tag{2.2}$$

- Estimating cost function

$$\min_{\alpha,\boldsymbol{\beta},\varepsilon} \sum_{i=1}^{n} \varepsilon_i^2 \tag{2.3}$$

$$\begin{aligned} s.t. \quad & y_i = \alpha_i + \boldsymbol{\beta}_i^{'}\boldsymbol{x}_i + \varepsilon_i && \forall i \\ & \alpha_i + \boldsymbol{\beta}_i^{'}\boldsymbol{x}_i \geq \alpha_j + \boldsymbol{\beta}_j^{'}\boldsymbol{x}_i && \forall i,j \\ & \boldsymbol{\beta}_i \geq \mathbf{0} && \forall i \end{aligned}$$

where α_i and $\boldsymbol{\beta}_i$ define the intercept and slope parameters of tangent hyperplanes that characterize the estimated piecewise linear frontier. ε_i denotes the CNLS residuals. The first constraint can be interpreted as a multivariate regression equation, the second constraint imposes convexity (concavity), and the third constraint imposes monotonicity. Similar to the DEA specification, other standard specification of returns to scale can be imposed by an additional constraint on the intercept term α_i. If $\alpha_i = 0$, then the problem (2.2) or (2.3) is a constant returns to scale (CRS) model, otherwise it is a variable returns to scale (VRS) model. Note that Problems (2.2) and (2.3) are the quadratic programming problem, so we resort to the MOSEK and CPLEX solver to solve them.

The additive CNLS model can be estimated in Python using the module CNLS(y, x, z, ...) in the package `pyStoNED` with the cet parameter set to CET_ADDI (additive model), rts parameter set to RTS_VRS (VRS model), and code{fun} parameter set to FUN_PROD (production function) or FUN_COST (cost function). Further, in this section we set the parameter z=None and introduce it in section (2.2). The results can be displayed in the screen directly using the .display_alpha() (i.e., display the coefficients $\hat{\alpha}_i$) or stored in the memory using the .get_alpha().

The following examples are to demonstrate how to estimate the VRS models:

Example: Estimating production function (additive CNLS model) [.ipynb]

```
# import packages
from pystoned import CNLS
from pystoned.constant import CET_ADDI, FUN_PROD, OPT_LOCAL, RTS_VRS
from pystoned.dataset import load_Finnish_electricity_firm

# import Finnish electricity distribution firms data
data = load_Finnish_electricity_firm(x_select=['OPEX', 'CAPEX'],
                                     y_select=['Energy'])

# define and solve the CNLS model
model = CNLS.CNLS(y=data.y, x=data.x, z=None,
                  cet = CET_ADDI, fun = FUN_PROD, rts = RTS_VRS)

# estimate the model using: 1) remote solver; 2) local solver
# Please replace with your own email address reqired by NEOS server (see https://neos-
↪guide.org/content/FAQ#email)
model.optimize('email@address')
#model.optimize(OPT_LOCAL)

# display the estimates
model.display_alpha()
model.display_beta()
model.display_residual()

# store the estimates
alpha = model.get_alpha()
beta = model.get_beta()
residuals = model.get_residual()
```

2.1.2 Multiplicative CNLS model

Similar to the most existing SFA literature, the Cobb-Douglas and translog functions are commonly assumed to be the functional form for the function f, where inefficiency u and noise v affect production in a multiplicative fashion. We, thus, further consider the multiplicative specification in the nonparametric models. Note that the assumption of CRS would also require multiplicative error structure. The multiplicative composite error structure CNLS model is rephrased as

$$y_i = f(\boldsymbol{x}_i) \cdot \exp(\varepsilon_i) = f(\boldsymbol{x}_i) \cdot \exp(v_i - u_i) \tag{2.4}$$

Applying the log-transformation to Eq.(2.4), we obtain

$$\ln y_i = \ln f(\boldsymbol{x}_i) + v_i - u_i \tag{2.5}$$

To estimate the Eq.(2.5), we reformulate the additive production model Eq.(2.2) and obtain the following log-transformed CNLS formulation:

$$\begin{aligned} \min_{\alpha,\boldsymbol{\beta},\varepsilon} & \sum_{i=1}^{n} \varepsilon_i^2 \\ \textit{s.t.} \quad & \ln y_i = \ln(\boldsymbol{\phi}_i + 1) + \varepsilon_i \quad \forall i \\ & \boldsymbol{\phi}_i = \alpha_i + \boldsymbol{\beta}_i^{'} \boldsymbol{x}_i - 1 \quad \forall i \\ & \alpha_i + \boldsymbol{\beta}_i^{'} \boldsymbol{x}_i \leq \alpha_j + \boldsymbol{\beta}_j^{'} \boldsymbol{x}_i \quad \forall i, j \\ & \boldsymbol{\beta}_i \geq 0 \quad \forall i \end{aligned} \tag{2.6}$$

where $\phi_i + 1$ is the CNLS estimator of $E[y_i \mid \boldsymbol{x}_i]$. The value of one is added here to make sure that the computational algorithms do not try to take logarithm of zero. The first equality can be interpreted as the log transformed regression equation (using the natural logarithm function $\ln(.)$). The rest of constraints are similar to additive models. The use of ϕ_i allows the estimation of a multiplicative relationship between output and input while assuring convexity of the production possibility set in original input-output space. Note that one could not apply the log transformation directly to the input data $\boldsymbol{x}$ due to the fact that the piece-wise log-linear frontier does not satisfy the axiomatic property (i.e., concavity or convexity) of function f. Since the multiplicative model (2.6) is a nonlinear programming problem, we have to use the nonlinear solver, e.g., MINOS, KNITRO.

Example: CNLS as production function [.ipynb]

```
# import packages
from pystoned import CNLS
from pystoned.constant import CET_MULT, FUN_PROD, FUN_COST, OPT_LOCAL, RTS_VRS, RTS_CRS
from pystoned.dataset import load_Finnish_electricity_firm

# import Finnish electricity distribution firms data
data = load_Finnish_electricity_firm(x_select=['OPEX', 'CAPEX'], y_select=['Energy'])

# define and solve the Multiplicative CNLS_vrs model
model1 = CNLS.CNLS(y=data.y, x=data.x, z=None, cet = CET_MULT, fun = FUN_PROD, rts = RTS_
↪VRS)
model1.optimize('email@address')

# define and solve the Multiplicative CNLS_crs model
model2 = CNLS.CNLS(y=data.y, x=data.x, z=None, cet = CET_MULT, fun = FUN_PROD, rts = RTS_
↪CRS)
model2.optimize('email@address')
```

(continues on next page)

(continued from previous page)

```
# display residuals in the VRS model
model1.display_residual()

# display residuals in the CRS model
model2.display_residual()
```

Example: CNLS as cost function [.ipynb]

```
# import packages
from pystoned import CNLS
from pystoned.constant import CET_MULT, FUN_PROD, FUN_COST, OPT_LOCAL, RTS_VRS, RTS_CRS
from pystoned.dataset import load_Finnish_electricity_firm

# import Finnish electricity distribution firms data
data = load_Finnish_electricity_firm(x_select=['Energy', 'Length', 'Customers'],
                                     y_select=['TOTEX'])

# define and solve the Multiplicative CNLS_vrs model
model1 = CNLS.CNLS(y=data.y, x=data.x, z=None, cet = CET_MULT, fun = FUN_COST, rts = RTS_
↪VRS)
model1.optimize('email@address')

# define and solve the Multiplicative CNLS_crs model
model2 = CNLS.CNLS(y=data.y, x=data.x, z=None, cet = CET_MULT, fun = FUN_COST, rts = RTS_
↪CRS)
model2.optimize('email@address')

# display residuals in the VRS model
model1.display_residual()

# display residuals in the CRS model
model2.display_residual()
```

2.1.3 Corrected CNLS model

Corrected convex nonparametric least squares (C^2NLS) is a variant of the corrected ordinary least squares (COLS) model in which nonparametric least squares subject to monotonicity and concavity constraints replace the first-stage parametric ordinary least squares (OLS) regression. To estimate the production models, the C^2NLS model assumes that the regression f is monotonic increasing and globally concave production function, the inefficiencies ε are identically and independently distributed with mean μ and a finite variance σ^2, and that the inefficiencies ε are uncorrelated with inputs $\boldsymbol{x}$.

Like COLS, the C^2NLS method is implemented in two stages, which can be stated as follows:

- Estimate $E(y_i|x_i)$ by solving the CNLS model, e.g., Problem (2.2). Denote the CNLS residuals by ε_i^{CNLS}.
- Shift the residuals analogous to the `COLS` procedure; the `C2NLS` efficiency estimator is

$$\hat{\varepsilon_i}^{C2NLS} = \varepsilon_i^{CNLS} \max_h \varepsilon_h^{CNLS},$$

where values of $\hat{\varepsilon_i}^{C2NLS}$ range from $[0, +\infty]$ with 0 indicating efficient performance. Similarly, we adjust the CNLS

intercepts α_i as

$$\hat{\alpha_i}^{C2NLS} = \alpha_i^{CNLS} + \max_h \varepsilon_h^{CNLS},$$

where α_i^{CNLS} is the optimal intercept for firmi in above CNLS problem and α_i^{C2NLS} is the C^{2}NLS estimator. Slope coefficients β_i for C^{2}NLS are obtained directly as the optimal solution to the CNLS problem.

Example: Corrected CNLS [.ipynb]

```
# import packages
from pystoned import CNLS
from pystoned.constant import CET_ADDI, FUN_PROD, OPT_LOCAL, RTS_VRS
from pystoned.dataset import load_Finnish_electricity_firm

# import Finnish electricity distribution firms data
data = load_Finnish_electricity_firm(x_select=['OPEX', 'CAPEX'],
                                     y_select=['Energy'])

# First stage: solve the CNLS model
model = CNLS.CNLS(y=data.y, x=data.x, z=None, cet = CET_ADDI, fun = FUN_PROD, rts = RTS_
↪VRS)
model.optimize(OPT_LOCAL)

# Second stage: shift the residuals/intercept
print(model.get_adjusted_residual())
print(model.get_adjusted_alpha())
```

2.2 Convex Quantile and Expectile Approaches

2.2.1 Convex quantile regression

While CNLS estimates the conditional mean $E(y_i|x_i)$, quantile regression aims at estimating the conditional median or other quantiles of the response variable (Koenker and Bassett 1978; Koenker 2005) and provides an overall picture of the conditional distributions at any given quantiles τ. In this section, we extend CNLS problem to estimate convex quantile regression (CQR) (Wang et al., 2014; Kuosmanen et al., 2015) and convex expectile regression (CER) (Kuosmanen et al., 2020; Dai et al., 2020; Kuosmanen and Zhou, 2021). Note that both quantile and expectile estimations are more robust to outliers and heteroscedasticity than the CNLS estimation.

Given a pre-specified quantile $\tau \in (0, 1)$, the CQR model is defined as

$$\begin{aligned} \min_{\alpha,\boldsymbol{\beta},\varepsilon^+,\varepsilon^-} \quad & \tau \sum_{i=1}^{n} \varepsilon_i^+ + (1-\tau) \sum_{i=1}^{n} \varepsilon_i^- \\ s.t. \quad & y_i = \alpha_i + \boldsymbol{\beta}_i^{'} x_i + \varepsilon_i^+ - \varepsilon_i^- \forall i \\ & \alpha_i + \boldsymbol{\beta}_i^{'} x_i \leq \alpha_h + \boldsymbol{\beta}_h^{'} x_i \quad \forall i, h \\ & \boldsymbol{\beta}_i \geq 0 \quad \forall i \\ & \varepsilon_i^+ \geq 0, \ \varepsilon_i^- \geq 0 \quad \forall i \end{aligned} \tag{2.7}$$

Where ε_i^+ and ε_i^- denote the two non-negative components. The last set of constraints is the sign constraint of the error terms. The other constraints are the same as in the CNLS problems.

Example: Quantile estimation [.ipynb]

```
# import packages
from pystoned import CQER
from pystoned.constant import CET_ADDI, FUN_PROD, OPT_LOCAL, RTS_VRS
from pystoned import dataset as dataset

# import the GHG example data
data = dataset.load_GHG_abatement_cost(x_select=['HRSN', 'CPNK', 'GHG'], y_select=['VALK
↪'])

# calculate the quantile model
model = CQER.CQR(y=data.y, x=data.x, tau=0.5, z=None, cet=CET_ADDI, fun=FUN_PROD,␣
↪rts=RTS_VRS)
model.optimize(OPT_LOCAL)

# display estimated alpha and beta
model.display_alpha()
model.display_beta()

# display estimated residuals
model.display_positive_residual()
model.display_negative_residual()
```

2.2.2 Convex expectile regression

As shown in Kuosmanen et al (2015), the convex quantile regression may suffer from the non-uniqueness problem due to that Problem (2.7) is a linear programming problem. To address this problem, Kuosmanen et al (2015) purpose a convex expectile regression (CER) approach, where a quadratic objective function is used to ensure unique estimates of the quantile functions.

$$\begin{aligned} \min_{\alpha,\boldsymbol{\beta},\varepsilon^+,\varepsilon^-} \quad & \tilde{\tau}\sum_{i=1}^{n}(\varepsilon_i^+)^2 + (1-\tilde{\tau})\sum_{i=1}^{n}(\varepsilon_i^-)^2 \\ s.t. \quad & y_i = \alpha_i + \boldsymbol{\beta}_i^{'}x_i + \varepsilon_i^+ - \varepsilon_i^- \qquad \forall i \\ & \alpha_i + \boldsymbol{\beta}_i^{'}x_i \leq \alpha_h + \boldsymbol{\beta}_h^{'}x_i \qquad \forall i,h \\ & \boldsymbol{\beta}_i \geq 0 \qquad \forall i \\ & \varepsilon_i^+ \geq 0,\ \varepsilon_i^- \geq 0 \qquad \forall i \end{aligned} \tag{2.8}$$

Example: Expectile estimation [.ipynb]

```
# import packages
from pystoned import CQER
from pystoned.constant import CET_ADDI, FUN_PROD, OPT_LOCAL, RTS_VRS
from pystoned import dataset as dataset

# import the GHG example data
data = dataset.load_GHG_abatement_cost(x_select=['HRSN', 'CPNK', 'GHG'], y_select=['VALK
↪'])

# calculate the expectile model
model = CQER.CER(y=data.y, x=data.x, tau=0.5, z=None, cet=CET_ADDI, fun=FUN_PROD,␣
```

(continues on next page)

(continued from previous page)

```
↪rts=RTS_VRS)
model.optimize(OPT_LOCAL)

# display estimated alpha and beta
model.display_alpha()
model.display_beta()

# display estimated residuals
model.display_positive_residual()
model.display_negative_residual()
```

2.3 Contextual Variables

2.3.1 CNLS with Z variables

In practice, a firm's ability to operate efficiently often depends on operational conditions and practices, such as the production environment and the firm specific characteristics for example technology selection or managerial practices. Banker and Natarajan (2008) refer to both variables that characterize operational conditions and practices as contextual variables.

- Contextual variables are often (but not always) **external factors** that are beyond the control of firms
  - Examples: competition, regulation, weather, location
  - Need to adjust efficiency estimates for operating environment
  - Policy makers may influence the operating environment
- Contextual variables can also be **internal factors**
  - Examples: management practices, ownership
  - Better understanding of the impacts of internal factors can help the firm to improve performance

By introducing the contextual variables $\boldsymbol{z}_i = (z_{i1}, ..., z_{ir})'$, the multiplicative model (2.5) is reformulated as an partial log-linear model to take the operational conditions and practices into account.

$$\ln y_i = \ln f(\boldsymbol{x_i}) + \boldsymbol{\delta}^{'} \boldsymbol{z}_i + \varepsilon_i \tag{2.9}$$

where parameter vector $\boldsymbol{\delta} = (\delta_1, ..., \delta_r)$ represents the marginal effects of contextual variables on output. All other variables maintain their previous definitions. Further, we can also introduce the contextual variables to additive model. In this section, we take the multiplicative production model as our stating point.

Following Johnson and Kuosmanen (2011), we incorporate the contextual variables in the multiplicative CNLS estimation and redefine it as follows:

$$\begin{aligned}
&\min_{\alpha,\boldsymbol{\beta},\boldsymbol{\delta},\varepsilon} \sum_{i=1}^{n} \varepsilon_i^2 && && (2.10)\\
s.t.\quad & \ln y_i = \ln(\phi_i + 1) + \boldsymbol{\delta}^{'} \boldsymbol{z}_i + \varepsilon_i && \forall i\\
& \phi_i = \alpha_i + \boldsymbol{\beta}_i^{'} \boldsymbol{x}_i - 1 && \forall i\\
& \alpha_i + \boldsymbol{\beta}_i^{'} \boldsymbol{x}_i \leq \alpha_j + \boldsymbol{\beta}_j^{'} \boldsymbol{x}_i && \forall i, j\\
& \boldsymbol{\beta}_i \geq \mathbf{0} && \forall i
\end{aligned}$$

Denote by $\hat{\boldsymbol{\delta}}$ the coefficients of the contextual variables obtained as the optimal solution to above nonlinear problem. Johnson and Kuosmanen (2011) examine the statistical properties of this estimator in detail, showing its unbiasedness, consistency, and asymptotic efficiency.

Example: CNLS-Z [.ipynb]

```
# import packages
from pystoned import CNLS
from pystoned.constant import CET_MULT, FUN_COST, RTS_CRS, RED_MOM
from pystoned.dataset import load_Finnish_electricity_firm

# import all data (including the contextual varibale)
data = load_Finnish_electricity_firm(x_select=['Energy', 'Length', 'Customers'],
                                     y_select=['TOTEX'],
                                     z_select=['PerUndGr'])

# define and solve the CNLS-Z model
model = CNLS.CNLS(y=data.y, x=data.x, z=data.z, cet = CET_MULT, fun = FUN_COST, rts =␣
↪RTS_CRS)
model.optimize('email@address')

# display the coefficient of contextual variable
model.display_lamda()
```

2.3.2 CER with Z variables

Following Kuosmanen et al. (2021), we can also incorporate the contextual variable in the multiplicative CER estimation.

$$\min_{\phi,\alpha,\boldsymbol{\beta},\varepsilon^{+},\varepsilon^{-}} \tilde{\tau}\sum_{i=1}^{n}(\varepsilon_i^{+})^2 + (1-\tilde{\tau})\sum_{i=1}^{n}(\varepsilon_i^{-})^2 \tag{2.11}$$

$$\begin{aligned}
s.t.\quad & \ln y_i = \ln(\phi_i + 1) + \boldsymbol{\delta}^{'}\boldsymbol{z}_i + \varepsilon_i^{+} - \varepsilon_i^{-} && \forall i \\
& \phi_i = \alpha_i + \boldsymbol{\beta}_i^{'}\boldsymbol{x}_i - 1 && \forall i \\
& \alpha_i + \boldsymbol{\beta}_i^{'}\boldsymbol{x}_i \leq \alpha_j + \boldsymbol{\beta}_j^{'}\boldsymbol{x}_i && \forall i,j \\
& \boldsymbol{\beta}_i \geq \mathbf{0} && \forall i \\
& \varepsilon_i^{+} \geq 0,\ \varepsilon_i^{-} \geq 0 && \forall i
\end{aligned}$$

Example: CER-Z [.ipynb]

```
# import packages
from pystoned import CQER
from pystoned.constant import CET_MULT, FUN_COST, RTS_CRS, RED_MOM
from pystoned.dataset import load_Finnish_electricity_firm

# import all data (including the contextual varibale)
data = load_Finnish_electricity_firm(x_select=['Energy', 'Length', 'Customers'],
                                     y_select=['TOTEX'],
                                     z_select=['PerUndGr'])

# define and solve the CER-Z model
model = CQER.CER(y=data.y, x=data.x, z=data.z, tau=0.5, cet = CET_MULT, fun = FUN_COST,␣
↪rts = RTS_CRS)
model.optimize('email@address')

# display the coefficient of contextual variable
model.display_lamda()
```

2.4 Multiple Outputs (DDF Formulation)

2.4.1 CNLS with multiple outputs

Until now, the convex regression approaches have been presented in the single output, multiple input setting. In this section, we describe the CNLS/CQR/CER approaches within the directional distance function (DDF) framework to handle with multiple input-multiple output data (Chambers et al., 1996, 1998).

Consider the following quadratic programming problem (Kuosmanen and Johnson, 2017)

$$
\begin{aligned}
\min_{\alpha,\boldsymbol{\beta},\boldsymbol{\gamma},\varepsilon} \quad & \sum_{i=1}^{n} \varepsilon_i^2 && (2.12)\\
s.t. \quad & \boldsymbol{\gamma}_i^{'}\boldsymbol{y}_i = \alpha_i + \beta_i^{'}\boldsymbol{x}_i - \varepsilon_i && \forall i\\
& \alpha_i + \boldsymbol{\beta}_i^{'}\boldsymbol{x}_i - \boldsymbol{\gamma}_i^{'}\boldsymbol{y}_i \leq \alpha_j + \boldsymbol{\beta}_j^{'}\boldsymbol{x}_i - \boldsymbol{\gamma}_j^{'}\boldsymbol{y}_i && \forall i,j\\
& \boldsymbol{\gamma}_i^{'}g^y + \boldsymbol{\beta}_i^{'}g^x = 1 && \forall i\\
& \boldsymbol{\beta}_i \geq \mathbf{0}, \boldsymbol{\gamma}_i \geq \mathbf{0} && \forall i
\end{aligned}
$$

where the residual ε_i represents the estimated value of $d\ (\vec{D}(x_i, y_i, g^x, g^y) + u_i)$. Besides the same notations as the CNLS estimator, we also introduce new firm-specific coefficients $\boldsymbol{\gamma_i}$ that represent marginal effects of outputs to the DDF.

The first constraint defines the distance to the frontier as a linear function of inputs and outputs. The linear approximation of the frontier is based on the tangent hyperplanes, analogous to the original CNLS formulation. The second set of constraints is the system of Afriat inequalities that impose global concavity. The third constraint is a normalization constraint that ensures the translation property. The last two constraints impose monotonicity in all inputs and outputs. It is straightforward to show that the CNLS estimator of function d satisfies the axioms of free disposability, convexity, and the translation property.

Example: CNLS-DDF [.ipynb]

```
# import packages
from pystoned import CNLSDDF
from pystoned.constant import FUN_PROD, OPT_LOCAL
from pystoned.dataset import load_Finnish_electricity_firm

# import Finnish electricity distribution firms data
data = load_Finnish_electricity_firm(x_select=['OPEX', 'CAPEX'],
                                     y_select=['Energy', 'Length', 'Customers'])

# define and solve the CNLS-DDF model
model = CNLSDDF.CNLSDDF(y=data.y, x=data.x, b=None, fun = FUN_PROD, gx= [1.0, 0.0],␣
↪gb=None, gy= [0.0, 0.0, 0.0])
model.optimize(OPT_LOCAL)

# display the estimates (alpha, beta, gamma, and residual)
model.display_alpha()
model.display_beta()
model.display_gamma()
model.display_residual()
```

2.4.2 CQR/CER with multiple outputs

Similarly to CNLS with DDF, we present another two approaches integrating DDF to convex quantile/expectile regression.

- CQR-DDF model

$$
\begin{aligned}
\min_{\alpha,\boldsymbol{\beta},\boldsymbol{\gamma},\varepsilon^+,\varepsilon^-} \quad & \tau\sum_{i=1}^{n}\varepsilon_i^+ + (1-\tau)\sum_{i=1}^{n}\varepsilon_i^- && (2.13)\\
s.t. \quad & \boldsymbol{\gamma}_i^{'}\boldsymbol{y}_i = \alpha_i + \boldsymbol{\beta}_i^{'}\boldsymbol{x}_i + \varepsilon_i^+ - \varepsilon_i^- && \forall i\\
& \alpha_i + \boldsymbol{\beta}_i^{'}\boldsymbol{x}_i - \boldsymbol{\gamma}_i^{'}\boldsymbol{y}_i \leq \alpha_j + \boldsymbol{\beta}_j^{'}\boldsymbol{x}_i - \boldsymbol{\gamma}_j^{'}\boldsymbol{y}_i && \forall i,j\\
& \boldsymbol{\gamma}_i^{'}g^y + \boldsymbol{\beta}_i^{'}g^x = 1 && \forall i\\
& \boldsymbol{\beta}_i \geq \mathbf{0}, \boldsymbol{\gamma}_i \geq \mathbf{0} && \forall i\\
& \varepsilon_i^+ \geq 0,\ \varepsilon_i^- \geq 0 && \forall i
\end{aligned}
$$

- CER-DDF model

$$
\begin{aligned}
\min_{\alpha,\boldsymbol{\beta},\boldsymbol{\gamma},\varepsilon^+,\varepsilon^-} \quad & \tilde{\tau}\sum_{i=1}^{n}(\varepsilon_i^+)^2 + (1-\tilde{\tau})\sum_{i=1}^{n}(\varepsilon_i^-)^2 && (2.14)\\
s.t. \quad & \boldsymbol{\gamma}_i^{'}\boldsymbol{y}_i = \alpha_i + \boldsymbol{\beta}_i^{'}\boldsymbol{x}_i + \varepsilon_i^+ - \varepsilon_i^- && \forall i\\
& \alpha_i + \boldsymbol{\beta}_i^{'}\boldsymbol{x}_i - \boldsymbol{\gamma}_i^{'}\boldsymbol{y}_i \leq \alpha_j + \boldsymbol{\beta}_j^{'}\boldsymbol{x}_i - \boldsymbol{\gamma}_j^{'}\boldsymbol{y}_i && \forall i,j\\
& \boldsymbol{\gamma}_i^{'}g^y + \boldsymbol{\beta}_i^{'}g^x = 1 && \forall i\\
& \boldsymbol{\beta}_i \geq \mathbf{0}, \boldsymbol{\gamma}_i \geq \mathbf{0} && \forall i\\
& \varepsilon_i^+ \geq 0,\ \varepsilon_i^- \geq 0 && \forall i
\end{aligned}
$$

Example: CQR-DDF [.ipynb]

```
# import packages
from pystoned import CQERDDF
from pystoned.constant import FUN_PROD, OPT_LOCAL
from pystoned import dataset as dataset

# import Finnish electricity distribution firms data
data = dataset.load_Finnish_electricity_firm(x_select=['OPEX', 'CAPEX'],
                                    y_select=['Energy', 'Length', 'Customers'])

# define and solve the CQR-DDF model
model = CQERDDF.CQRDDF(y=data.y, x=data.x, b=None, tau=0.9, fun = FUN_PROD, gx= [1.0, 0.
↪0], gb=None, gy= [0.0, 0.0, 0.0])
model.optimize(OPT_LOCAL)

# display the residual
model.display_positive_residual()
model.display_negative_residual()
```

2.5 Monotonic Models

2.5.1 Isotonic CNLS

This section introduces a variant of CNLS estimator, Isotonic Convex Nonparametric Least squares (ICNLS), where ICNLS only relies on the monotonic assumption. To relax the concavity assumption in CNLS estimation (i.e., estimating a production function), we have to rephrase the Afriat inequality constraint in Problem 2.2). To do that, we define a binary matrix $P = [p_{ij}]_{nxn}$ to represent the isotonicity (Keshvari and Kuosmanen, 2013).

Define the binary matrix $P = [p_{ij}]_{nxn}$ as follows

$$p_{ij} = \begin{cases} 1 & \text{if } x_i \preccurlyeq x_j \\ 0 & \text{otherwise} \end{cases}$$

Applying enumeration method to define the elements of matrix P, we solving the following QP problem

$$\underset{\alpha,\beta,\varepsilon}{min} \sum_{i=1}^{n} \varepsilon_i^2$$

s.t.

$$y_i = \alpha_i + \beta_i^{'} X_i + \varepsilon_i \quad \forall i$$

$$p_{ij}(\alpha_i + \beta_i^{'} X_i) \leq p_{ij}(\alpha_j + \beta_j^{'} X_i) \quad \forall i, j$$

$$\beta_i \geq 0 \quad \forall i$$

Note that the concavity constraints between units i and j is relaxed by the matrix $P_{ij} = 0$. If the $P_{ij} = 1$ for all i and j, then the above QP problem (i.e., ICNLS problem) reduces to CNLS problem.

Example: Isotonic CNLS(ICNLS) [.ipynb]

In the following code, we estimate an additive production function using ICNLS approach.

```
# import packages
from pystoned import ICNLS
from pystoned.constant import CET_ADDI, FUN_PROD, OPT_LOCAL, RTS_VRS
from pystoned.dataset import load_Finnish_electricity_firm

# import Finnish electricity distribution firms data
data = load_Finnish_electricity_firm(x_select=['OPEX', 'CAPEX'], y_select=['Energy'])

# define and solve the ICNLS model
model = ICNLS.ICNLS(y=data.y, x=data.x, z=None, cet = CET_ADDI, fun = FUN_PROD, rts =␣
↪RTS_VRS)
model.optimize(OPT_LOCAL)

# display residuals
model.display_residual()
```

2.5.2 Isotonic CQR/CER

Similarly to ICNLS, the Isotonic CQR and CER approaches are defined as follows

ICQR estimator:

$$\min_{\alpha,\beta,\varepsilon^+,\varepsilon^-} \tau \sum_{i=1}^{n} \varepsilon_i^+ + (1-\tau) \sum_{i=1}^{n} \varepsilon_i^-$$

s.t.

$$\begin{aligned}
& y_i = \alpha_i + \beta_i' x_i + \varepsilon_i^+ - \varepsilon_i^- && \forall i \\
& p_{ih}(\alpha_i + \beta_i' x_i) \leq p_{ih}(\alpha_h + \beta_h' x_i) && \forall i, h \\
& \beta_i \geq 0 && \forall i \\
& \varepsilon_i^+ \geq 0,\ \varepsilon_i^- \geq 0 && \forall i
\end{aligned}$$

ICER estimator:

$$\min_{\alpha,\beta,\varepsilon^+,\varepsilon^-} \tilde{\tau} \sum_{i=1}^{n} (\varepsilon_i^+)^2 + (1-\tilde{\tau}) \sum_{i=1}^{n} (\varepsilon_i^-)^2$$

s.t.

$$\begin{aligned}
& y_i = \alpha_i + \beta_i' x_i + \varepsilon_i^+ - \varepsilon_i^- && \forall i \\
& p_{ih}(\alpha_i + \beta_i' x_i) \leq p_{ih}(\alpha_h + \beta_h' x_i) && \forall i, h \\
& \beta_i \geq 0 && \forall i \\
& \varepsilon_i^+ \geq 0,\ \varepsilon_i^- \geq 0 && \forall i
\end{aligned}$$

Example: Isotonic CQR(ICQR) [.ipynb]

In the following code, we estimate an additive production function using ICQR approach.

```
# import packages
from pystoned import ICQER
from pystoned.constant import CET_ADDI, FUN_PROD, OPT_LOCAL, RTS_VRS
from pystoned.dataset import load_Finnish_electricity_firm

# import Finnish electricity distribution firms data
data = load_Finnish_electricity_firm(x_select=['OPEX', 'CAPEX'], y_select=['Energy'])

# define and solve the ICQR model
model = ICQER.ICQR(y=data.y, x=data.x, tau = 0.9, z=None, cet = CET_ADDI, fun = FUN_PROD,
↪ rts = RTS_VRS)
model.optimize(OPT_LOCAL)

# display residuals
model.display_residual()
```

Example: Isotonic CER(ICER) [.ipynb]

We next demostrate how to estimate an additive production function using ICER approach.

```
# import packages
from pystoned import ICQER
from pystoned.constant import CET_ADDI, FUN_PROD, OPT_LOCAL, RTS_VRS
from pystoned.dataset import load_Finnish_electricity_firm

# import Finnish electricity distribution firms data
```

(continues on next page)

(continued from previous page)

```
data = load_Finnish_electricity_firm(x_select=['OPEX', 'CAPEX'], y_select=['Energy'])

# define and solve the ICER model
model = ICQER.ICER(y=data.y, x=data.x, tau = 0.9, z=None, cet = CET_ADDI, fun = FUN_PROD,
↪ rts = RTS_VRS)
model.optimize(OPT_LOCAL)

# display residuals
model.display_residual()
```

2.6 Stochastic Nonparametric Envelopment of Data

2.6.1 The introduction to StoNED

Combing virtues of SFA and DEA in a unified framework, Stochastic Nonparametric Envelopment of Data (StoNED) (kuosmanen, 2006) uses a composed error term to model both inefficiency u and noise v without assuming a functional form of f. Analogous to the COLS/C [2] NLS estimators, the StoNED estimator consists of the following four steps:

- Step 1: Estimating the conditional mean $E[y_i \mid \boldsymbol{x}_i]$ using CNLS estimator
- Step 2: Estimating the expected inefficiency μ based on the residual ε_i^{CNLS}
- Step 3: Estimating the StoNED frontier $\hat{f}^{StoNED}$ based on the $\hat{\mu}$
- Step 4: Estimating firm-specific inefficiencies $E[u_i \mid \varepsilon_i^{CNLS}]$

Beside the CNLS estimator, we can apply other convex regression approaches such as ICNLS and CNLS-DDF to estimate the conditional mean in the first step (see Keshvari and Kuosmanen, 2013; Kuosmanen and Johnson, 2017). However, the quantile and expectile related estimators introduced in **Examples** can not be integrated into StoNED framework at present.

After obtaining the residuals (e.g., $\hat{\varepsilon}_i^{CNLS}$) from the convex regression approaches, one can estimate the expected value of the inefficiency term $\mu = E(u_i)$. In practice, three commonly used methods are available to estimate the expected inefficiency μ: method of moments (Aigner et al., 1977), quasi-likelihood estimation (Fan et al., 1996), and the kernel deconvolution estimation (Hall and Simar, 2002). We will next briefly review these three approaches and focus on demonstrating the application of *pyStoNED*; see more detailed theoretical introduction in Kuosmanen et al. (2015).

2.6.2 Method of moments

The method of moments requires some additional parametric distributional assumptions. Following Kuosmanen et al. (2015), under the maintained assumptions of half-normal inefficiency (i.e., $u_i \sim N^+(0, \sigma_u^2)$) and normal noise (i.e., $v_i \sim N(0, \sigma_v^2)$), the second and third central moments of the composite error (i.e., ε_i) distribution are given by

$$M_2 = \left[\frac{\pi - 2}{\pi}\right]\sigma_u^2 + \sigma_v^2$$

$$M_3 = \left(\sqrt{\frac{2}{\pi}}\right)\left[1 - \frac{4}{\pi}\right]\sigma_u^2$$

The second and third central moments can be estimated by using the CNLS residuals (i.e., $\hat{\varepsilon}_i^{CNLS}$)

$$\hat{M}_2 = \sum_{i=1}^{n}(\hat{\varepsilon}_i - \bar{\varepsilon})^2/n$$

$$\hat{M}_3 = \sum_{i=1}^{n}(\hat{\varepsilon}_i - \bar{\varepsilon})^3/n$$

Note that the third moment M_3 (which measures the skewness of the distribution) only depends on the standard deviation parameter σ_u of the inefficiency distribution. Thus, given the estimated $\hat{M}_3$ (which should be positive in the case of a cost frontier), we can estimate σ_u and σ_v by

$$\hat{\sigma}_u = \sqrt[3]{\frac{\hat{M}_3}{\left(\sqrt{\frac{2}{\pi}}\right)\left[1-\frac{4}{\pi}\right]}}$$

$$\hat{\sigma}_v = \sqrt[2]{\hat{M}_2 - \left[\frac{\pi-2}{\pi}\right]\hat{\sigma}_u^2}$$

Example: StoNED with CNLS [.ipynb]

```
# import packages
from pystoned import CNLS, StoNED
from pystoned.dataset import load_Finnish_electricity_firm
from pystoned.constant import CET_MULT, FUN_COST, RTS_VRS, RED_MOM

# import Finnish electricity distribution firms data
data = load_Finnish_electricity_firm(x_select=['Energy', 'Length', 'Customers'],
                                     y_select=['TOTEX'])

# build and optimize the CNLS model
model = CNLS.CNLS(data.y, data.x, z=None, cet=CET_MULT, fun=FUN_COST, rts=RTS_VRS)
model.optimize('email@address')

# calculate and print unconditional expected inefficiency (mu)
rd = StoNED.StoNED(model)
print(rd.get_unconditional_expected_inefficiency(RED_MOM))
```

2.6.3 Quassi-likelihood estimation

Quassi-likelihood approach is alternative to decomposing the σ_u and σ_v suggested by Fan et al. (1996). In this method we apply the standard maximum likelihood method to estimate the parameters σ_u and σ_v, taking the shape of CNLS curve as given. The quasi-likelihood function is formulated as

$$\ln L(\lambda) = -n\ln(\hat{\sigma}) + \sum \ln\Phi\left[\frac{-\hat{\varepsilon}_i\lambda}{\hat{\sigma}}\right] - \frac{1}{2\hat{\sigma}^2}\sum \hat{\varepsilon}_i^2$$

where

$$\hat{\varepsilon}_i = \hat{\varepsilon}_i^{CNLS} - (\sqrt{2}\lambda\hat{\sigma})/[\pi(1+\lambda^2)]^{1/2}$$

$$\hat{\sigma} = \left\{\frac{1}{n}\sum(\hat{\varepsilon}_i^{CNLS})^2 / \left[1 - \frac{2\lambda^2}{\pi(1+\lambda^2)}\right]\right\}$$

Note that the quasi-likelihood function only consists of a single parameter λ (i.e., the signal-to-noise ratio $\lambda = \sigma_u/\sigma_v$). The symbol Φ represents the cumulative distribution function of the standard normal distribution. In the *pyStoNED* package, we use the Broyden-Fletcher-Goldfarb-Shanno (BFGS) algorithm to solve the maximum likelihood function.

Example: StoNED with CNLS [.ipynb]

In the following code, we use the quassi-likelihood approach to decompose the CNLS residuals and display the StoNED frontier.

```
# import packages
from pystoned import CNLS, StoNED
from pystoned.dataset import load_Finnish_electricity_firm
from pystoned.constant import CET_MULT, FUN_COST, RTS_VRS, RED_QLE

# import Finnish electricity distribution firms data
data = load_Finnish_electricity_firm(x_select=['Energy', 'Length', 'Customers'],
                                     y_select=['TOTEX'])

# build and optimize the CNLS model
model = CNLS.CNLS(data.y, data.x, z=None, cet=CET_MULT, fun=FUN_COST, rts=RTS_VRS)
model.optimize('email@address')

# calculate and print unconditional expected inefficiency (mu)
rd = StoNED.StoNED(model)
print(rd.get_unconditional_expected_inefficiency(RED_QLE))
```

2.6.4 Kernel deconvolution estimation

while method of moments and quasi-likelihood approaches require additional distributional assumptions for the inefficiency and noise terms, a fully nonparametric estimation of the expected inefficiency μ is also available by applying nonparametric kernel deconvolution, proposed by Hall and Simar (2002). Note that the residuals $\hat{\varepsilon}_i^{CNLS}$ are consistent estimators of $e^o = \varepsilon_i + \mu$ (for production model). Following Kuosmanen and Johnson (2017), the density function of e^o

$$\hat{f}_{e^o}(z) = (nh)^{-1} \sum_{i=1}^{n} K\left(\frac{z - e_i^o}{h}\right)$$

where $K(\cdot)$ is a compactly supported kernel and h is a bandwidth. citet{Hall2002} show that the first derivative of the density function of the composite error term ($f_{\varepsilon}^{'}$)is proportional to that of the inefficiency term ($f_u^{'}$) in the neighborhood of μ. Therefore, a robust nonparametric estimator of expected inefficiency μ is obtained as

$$\hat{\mu} = \arg\max_{z \in C} (\hat{f'}_{e^o}(z)),$$

where C is a closed interval in he right tail of f_{e^o}.

Example: StoNED with CNLS [.ipynb]

In the following code, we use the kernel density approach to decompose the CNLS residuals and display the unconditional expected inefficiency.

```
# import packages
from pystoned import CNLS, StoNED
from pystoned.dataset import load_Finnish_electricity_firm
from pystoned.constant import CET_MULT, FUN_COST, RTS_VRS, RED_KDE

# import Finnish electricity distribution firms data
data = load_Finnish_electricity_firm(x_select=['Energy', 'Length', 'Customers'],
                                     y_select=['TOTEX'])

# build and optimize the CNLS model
model = CNLS.CNLS(data.y, data.x, z=None, cet=CET_MULT, fun=FUN_COST, rts=RTS_VRS)
model.optimize('email@address')
```

(continues on next page)

(continued from previous page)

```
# calculate and print unconditional expected inefficiency (mu)
rd = StoNED.StoNED(model)
print(rd.get_unconditional_expected_inefficiency(RED_KDE))
```

2.6.5 Estimating the StoNED frontier

Having estimated the expected inefficiency $\hat{\mu}$, we can shift the conditional mean function estimated by CNLS to the frontier in the case of the additive and multiplicative models under VRS and CRS. Given the fact that the function estimated by CNLS is only unique for the observed points $\boldsymbol{x}_i$, we then follow Kuosmanen and Kortelainen (2012) to estimate the unique conditional mean function $\hat{g}_{min}^{CNLS}(\boldsymbol{x})$ under VRS.

$$\hat{g}_{min}^{CNLS}(\boldsymbol{x}) = \min_{\alpha,\boldsymbol{\beta}}\{\alpha + \boldsymbol{\beta}'x \mid \alpha + \boldsymbol{\beta}'\boldsymbol{x}_i \geq \hat{g}^{CNLS}(\boldsymbol{x}_i), \forall i\}$$

where $\hat{g}^{CNLS}(\boldsymbol{x}_i)$‘ is the conditional mean function estimated by the CNLS estimator. We subsequently shift the conditional mean function $\hat{g}_{min}^{CNLS}(\boldsymbol{x})$ to the frontier using

- Production function.
  1. Additive model: $\hat{f}^{StoNED}(\boldsymbol{x}) = \hat{g}_{min}^{CNLS}(\boldsymbol{x}) + \hat{\mu}$;
  2. Multiplicative model: $\hat{f}^{StoNED}(\boldsymbol{x}) = \hat{g}_{min}^{CNLS}(\boldsymbol{x}) \cdot \exp(\hat{\mu})$.
- Cost function.
  1. Additive model: $\hat{f}^{StoNED}(\boldsymbol{x}) = \hat{g}_{min}^{CNLS}(\boldsymbol{x}) - \hat{\mu}$;
  2. Multiplicative model: $\hat{f}^{StoNED}(\boldsymbol{x}) = \hat{g}_{min}^{CNLS}(\boldsymbol{x}) \cdot \exp(-\hat{\mu})$.

Example: StoNED frontier [.ipynb]

In the following example, we demonstrate how to obtain the StoNED frontier via the package.

```
# import packages
from pystoned import CNLS, StoNED
from pystoned.dataset import load_Finnish_electricity_firm
from pystoned.constant import CET_MULT, FUN_COST, RTS_VRS, RED_MOM

# import Finnish electricity distribution firms data
data = load_Finnish_electricity_firm(x_select=['Energy', 'Length', 'Customers'],
                                     y_select=['TOTEX'])

# build and optimize the CNLS model
model = CNLS.CNLS(data.y, data.x, z=None, cet=CET_MULT, fun=FUN_COST, rts=RTS_VRS)
model.optimize('email@address')

# calculate and print the StoNED frontier
rd = StoNED.StoNED(model)
print(rd.get_stoned(RED_MOM))
```

2.6.6 JLMS estimator

After estimating the expected inefficiency μ using methods of moment (MOM) or quasi-likelihood estimation (QLE),[1] we then employ JLMS estimator proposed by Jondrow et al. (1982) to estimate the firm-specific inefficiencies (Johnson and Kuosmanen, 2015). Under the assumption of a normally distributed error term and a half-normally distributed inefficiency term, JLMS formulates the conditional distribution of inefficiency u_i, given ε_i, and propose the inefficiency estimator as the conditional mean $E[u_i|\varepsilon_i]$.

Following Kumbhakar & Lovell (2000), the conditional expected value of inefficiency $E[u_i|\varepsilon_i]$ for production function and cost function are shown as follow, respectively:

- Production function

$$E[u_i \mid \varepsilon_i] = \mu_{*i} + \sigma_* \left[\frac{\phi(-\mu_{*i}/\sigma_*)}{1 - \Phi(-\mu_{*i}/\sigma_*)} \right] = \sigma_* \left[\frac{\phi(\varepsilon_i \lambda/\sigma)}{1 - \Phi(\varepsilon_i \lambda/\sigma)} - \frac{\varepsilon_i \lambda}{\sigma} \right].$$

where $\mu_* = -\varepsilon\sigma_u^2/\sigma^2$, $\sigma_*^2 = \sigma_u^2\sigma_v^2/\sigma^2$, $\lambda = \sigma_u/\sigma_v$, and $\sigma^2 = \sigma_u^2 + \sigma_v^2$. The symbol ϕ is the standard normal density function, and the symbol Φ denotes the cumulative distribution function of the standard normal distribution.

- Cost function

$$E[u_i \mid \varepsilon_i] = \mu_{*i} + \sigma_* \left[\frac{\phi(-\mu_{*i}/\sigma_*)}{1 - \Phi(-\mu_{*i}/\sigma_*)} \right] = \sigma_* \left[\frac{\phi(\varepsilon_i \lambda/\sigma)}{1 - \Phi(-\varepsilon_i \lambda/\sigma)} + \frac{\varepsilon_i \lambda}{\sigma} \right].$$

where $\mu_* = \varepsilon\sigma_u^2/\sigma^2$, $\sigma_*^2 = \sigma_u^2\sigma_v^2/\sigma^2$, $\lambda = \sigma_u/\sigma_v$, and $\sigma^2 = \sigma_u^2 + \sigma_v^2$.

The firm-level technical efficiency (TE) is then measured based on the estimated conditional mean. For different model, the technical efficiency is calculated as

- Production function
  - Multiplicative model: TE $= \exp(-E[u_i \mid \varepsilon_i])$
  - Additive model: TE $= \frac{y - E[u_i|\varepsilon_i]}{y}$
- Cost function
  - Multiplicative model: TE $= \exp(E[u_i \mid \varepsilon_i])$
  - Additive model: TE $= \frac{y + E[u_i|\varepsilon_i]}{y}$

Example: CNLS [.ipynb]

```
# import packages
from pystoned import CNLS, StoNED
from pystoned.dataset import load_Finnish_electricity_firm
from pystoned.constant import CET_MULT, FUN_COST, RTS_VRS, RED_MOM
```

(continues on next page)

[1] For the expected inefficiency mu estimated by kernel deconvolution, Dai (2016) proposes a non-parametric strategy where the Richardson–Lucy blind deconvolution algorithm is used to identify firm-specific inefficiencies. However, the *pyStoNED* package only supports the parametric estimation of firm-specific inefficiencies due to the fact that the parametric method is more widely used in efficiency analysis literature.

(continued from previous page)

```
# import Finnish electricity distribution firms data
data = load_Finnish_electricity_firm(x_select=['Energy', 'Length', 'Customers'],
                                     y_select=['TOTEX'])

# build and optimize the CNLS model
model = CNLS.CNLS(data.y, data.x, z=None, cet=CET_MULT, fun=FUN_COST, rts=RTS_VRS)
model.optimize('email@address')

# print firm-level efficiency using MOM method
rd = StoNED.StoNED(model)
print(rd.get_technical_inefficiency(RED_MOM))
```

Example: CNLS with Z variable [.ipynb]

```
# import packages
from pystoned import CNLS, StoNED
from pystoned.dataset import load_Finnish_electricity_firm
from pystoned.constant import CET_MULT, FUN_COST, RTS_VRS, RED_MOM

# import Finnish electricity distribution firms data
data=load_Finnish_electricity_firm(x_select=['Energy','Length','Customers'],
                                   y_select=['TOTEX'],
                                   z_select=['PerUndGr'])

# build and optimize the CNLS model
model = CNLS.CNLS(y=data.y, x=data.x, z=data.z, cet=CET_MULT, fun=FUN_COST, rts=RTS_VRS)
model.optimize('email@address')

# calculate and print firm-level efficiency using MOM method
rd = StoNED.StoNED(model)
print(rd.get_technical_inefficiency(RED_MOM))
```

2.7 CNLS-G Algorithm (for large sample)

2.7.1 Solving CNLS model

Since convex regression approaches shape the convexity (concavity) of function using the Afrait inequality, the estimation becomes excessively expensive due to the $O(n^2)$ linear constraints. e.g., If the data samples have 500 observations, the total number of linear constraints is equal to 250,000. To speed up the computational time, Lee et al., (2013) propose a more efficient generic algorithm, CNLS-G, which uses the relaxed Afriat constraint set and iteratively adds violated constraints to the relaxed model as necessary. See more discussion in Lee et al., (2013).

To illustrate the CNLS-G algorithm, we follow Lee et al., (2013) to generate the input and output variables. In this section, we assume an additive production function with two-input and one-output, $y = x_1^{0.4} * x_2^{0.4} + u$. We randomly draw the inputs x_1 and x_2 from a uniform distribution, $x \sim U[1, 10]$, and the error term u from a normal distribution, $u \sim N(0, 0.7^2)$. Based on these specifications, we first generate 500 artificial observations and then estimate the CNLS problem eqref{eq:eq2} and the CER problem eqref{eq:eq8} using the CNLS-G algorithm.

Example: Solving CNLS [.ipynb]

```
# import packages
from pystoned import CNLSG, CNLS
from pystoned.constant import CET_ADDI, FUN_PROD, OPT_LOCAL, RTS_VRS
import numpy as np
import time

# set seed
np.random.seed(0)

# generate DMUs: DGP
x = np.random.uniform(low=1, high=10, size=(500, 2))
u = np.random.normal(loc=0, scale=0.7, size=500)
y = x[:, 0]**0.4*x[:, 1]**0.4+u

# solve CNLS model without algorithm
t1 = time.time()
model1 = CNLS.CNLS(y, x, z=None, cet = CET_ADDI, fun = FUN_PROD, rts = RTS_VRS)
model1.optimize(OPT_LOCAL)
CNLS_time = time.time() -t1

# solve CNLS model using CNLS-G algorithm
model2 = CNLSG.CNLSG(y, x, z=None, cet = CET_ADDI, fun = FUN_PROD, rts = RTS_VRS)
model2.optimize(OPT_LOCAL)

# display running time
print("The running time with algorithm is ", model2.get_runningtime())
print("The running time without algorithm is ", CNLS_time)

# display number of constraints
print("The total number of constraints is ", model2.get_totalconstr())
```

2.7.2 Solving CER model

Example: Solving CER [.ipynb]

```
# import packages
from pystoned import CQERG, CQER
from pystoned.constant import CET_ADDI, FUN_PROD, OPT_LOCAL, RTS_VRS
import numpy as np
import time

# set seed
np.random.seed(0)

# generate DMUs: DGP
x = np.random.uniform(low=1, high=10, size=(500, 2))
u = np.random.normal(loc=0, scale=0.7, size=500)
y = x[:, 0]**0.4*x[:, 1]**0.4+u

# solve CER model without algorithm
tau = 0.9
```

(continues on next page)

(continued from previous page)

```
t1 = time.time()
model1 = CQER.CER(y, x, tau, z=None, cet = CET_ADDI, fun = FUN_PROD, rts = RTS_VRS)
model1.optimize(OPT_LOCAL)
CER_time = time.time() -t1

# solve CER model using CNLS-G algorithm
model2 = CQERG.CERG(y, x, tau, z=None, cet = CET_ADDI, fun = FUN_PROD, rts = RTS_VRS)
model2.optimize(OPT_LOCAL)

# display running time
print("The running time with algorithm is ", model2.get_runningtime())
print("The running time without algorithm is ", CER_time)

# display number of constraints
print("The total number of constraints in CER model is ", model2.get_totalconstr())
```

2.7.3 Calculating firm-level efficiency

Example: Using StoNED with CNLSG [.ipynb]

```
# import packages
from pystoned import CNLSG, StoNED
from pystoned.dataset import load_Finnish_electricity_firm
from pystoned.constant import CET_MULT, FUN_COST, RTS_VRS, RED_MOM

# import Finnish electricity distribution firms data
data = load_Finnish_electricity_firm(x_select=['Energy', 'Length', 'Customers'],
                                     y_select=['TOTEX'])

# build and optimize the CNLS model
model = CNLSG.CNLSG(data.y, data.x, z=None, cet=CET_MULT, fun=FUN_COST, rts=RTS_VRS)
model.optimize('email@address')

# Calculate firm-level efficiency
rd = StoNED.StoNED(model)
print(rd.get_technical_inefficiency(RED_MOM))
```

2.8 Plot of estimated function

2.8.1 Plot of estimated function/frontier: one-input and one-output

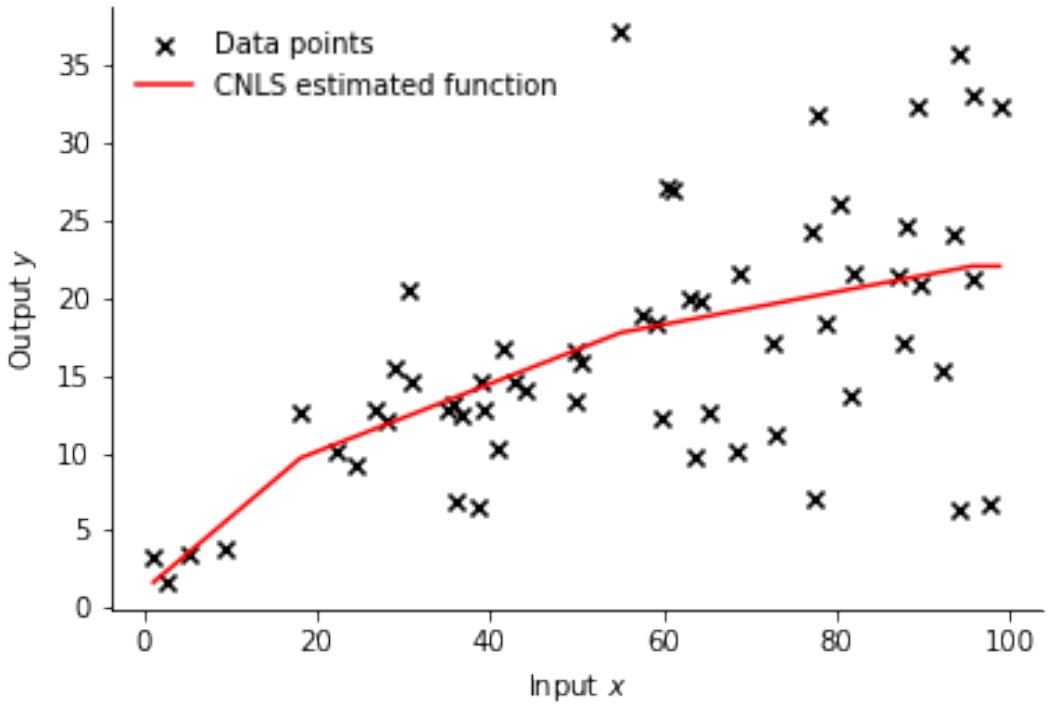
Example: 2D plot with CNLS and CQR [.ipynb]

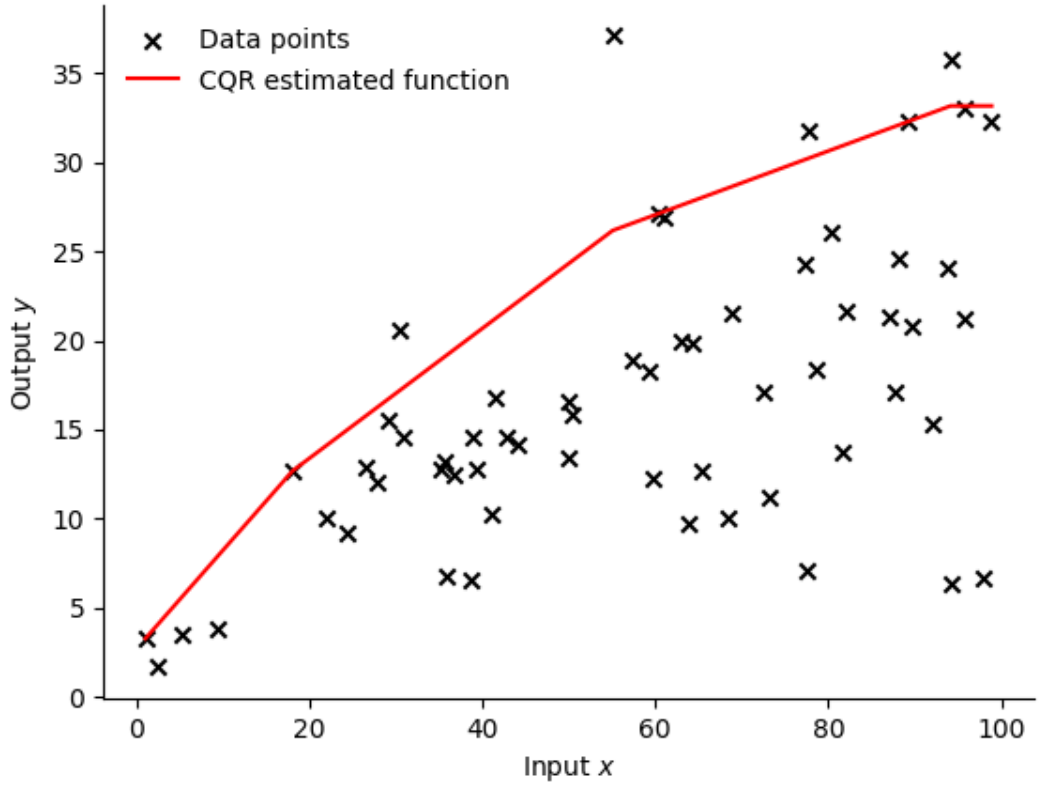
```
# import packages
from pystoned import CNLS, CQER
from pystoned.plot import plot2d
from pystoned.constant import CET_ADDI, FUN_PROD, OPT_LOCAL, RTS_VRS
from pystoned.dataset import load_Tim_Coelli_frontier

# import the data from Tim Coelli Frontier 4.1
data = load_Tim_Coelli_frontier(x_select=['labour'],
```

(continues on next page)

(continued from previous page)

```
                            y_select=['output'])

# define and solve the CNLS model
CNLS_model = CNLS.CNLS(y=data.y, x=data.x, z=None, cet = CET_ADDI, fun = FUN_PROD, rts =␣
↪RTS_VRS)
CNLS_model.optimize(OPT_LOCAL)

# Plot the estimated function
plot2d(CNLS_model, x_select=0, label_name="CNLS estimated function", fig_name="CNLS_2d")

# define and solve the CQR model
CQR_model=CQER.CQR(y=data.y, x=data.x, tau=0.9, z=None, cet=CET_ADDI, fun=FUN_PROD,␣
↪rts=RTS_VRS)
CQR_model.optimize(OPT_LOCAL)

# Plot the estimated function
plot2d(CQR_model, x_select=0, label_name="CQR estimated function", fig_name="CQR_2d")
```

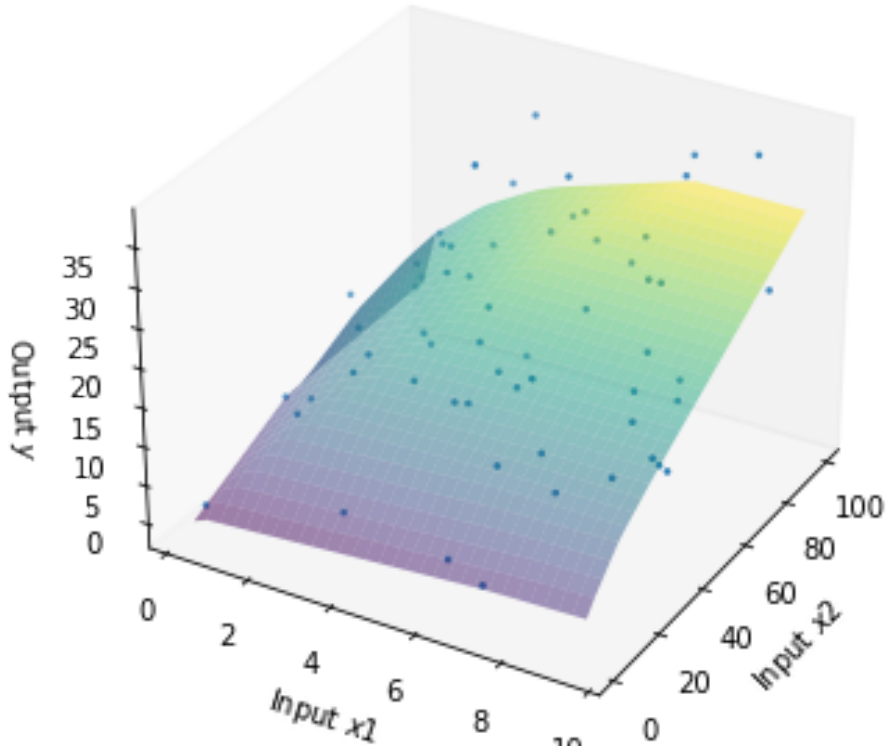




2.8.2 Plot of estimated function/frontier: two-input and one-output

Example: 3D plot with CNLS [.ipynb]

```
# import packages
from pystoned import CNLS
from pystoned.plot import plot3d
from pystoned.constant import CET_ADDI, FUN_PROD, OPT_LOCAL, RTS_VRS
from pystoned.dataset import load_Tim_Coelli_frontier

# import the data from Tim Coelli Frontier 4.1
data = load_Tim_Coelli_frontier(x_select=['capital', 'labour'],
                                y_select=['output'])

# define and solve the CNLS model
CNLS_model = CNLS.CNLS(y=data.y, x=data.x, z=None, cet = CET_ADDI, fun = FUN_PROD, rts =␣
↪RTS_VRS)
CNLS_model.optimize(OPT_LOCAL)

# Plot the estimated function
plot3d(CNLS_model,x_select_1=0, x_select_2=1, fig_name="CNLS_3d")
```



2.9 Data Envelopment Analysis

2.9.1 Radial model: Input orientation

A set of $j = 1, 2, \cdots, n$ observed `DMUs` transform a vector of $i = 1, 2, \cdots, m$ inputs $x \in R^m_{++}$ into a vector of $i = 1, 2, \cdots, s$ outputs $y \in R^s_{++}$ using the technology represented by the following **CRS** production possibility set: $P_{crs} = \{(x, y)|x \geq X\lambda, y \leq Y\lambda, \lambda \geq 0\}$, where $X = (x)_j \in R^{s \times n}$, $Y = (y)_j \in R^{m \times n}$ and $\lambda = (\lambda_1, ..., \lambda_n)^T$ is a intensity vector.

Based on the data matrix (X, Y), we measure the input oriented efficiency of `each observation o` by solving `n` times the following linear programming problems:

$$\begin{aligned} \min_{\phi,\lambda} \quad & \phi \\ \text{s.t.} \quad & \phi x_o \geq X\lambda \\ & Y\lambda \geq y_o \\ & \lambda \geq 0 \end{aligned}$$

The measurement of technical efficiency assuming **VRS** considers the following production possibility set $P_{vrs} = \{(x, y)|x \geq X\lambda, y \leq Y\lambda, e\lambda = 1, \lambda \geq 0.\}$. Thus, the only difference with the CRS model is the adjunction of the

condition $\sum_{j=1}^{n} \lambda_j = 1$.

$$\begin{aligned} \underset{\phi,\lambda}{min} \quad & \phi \\ \text{s.t.} \quad & \phi x_o \geq X\lambda \\ & Y\lambda \geq y_o \\ & \sum_{j=1}^{n} \lambda_j = 1 \\ & \lambda \geq 0 \end{aligned}$$

Example: Intput oriented DEA [.ipynb]

In the following code, we calculate the VRS radial model with pyStoNED.

```
# import packages
from pystoned import DEA
from pystoned import dataset as dataset
from pystoned.constant import RTS_VRS, ORIENT_IO, OPT_LOCAL

# import the data provided with Tim Coelli's Frontier 4.1
data = dataset.load_Tim_Coelli_frontier()

# define and solve the DEA radial model
model = DEA.DEA(data.y, data.x, rts=RTS_VRS, orient=ORIENT_IO, yref=None, xref=None)
model.optimize(OPT_LOCAL)

# display the technical efficiency
model.display_theta()

# display the intensity variables
model.display_lamda()
```

2.9.2 Radial model: Output orientation

A set of $j = 1, 2, \cdots, n$ observed `DMUs` transform a vector of $i = 1, 2, \cdots, m$ inputs $x \in R^m_{++}$ into a vector of $i = 1, 2, \cdots, s$ outputs $y \in R^s_{++}$ using the technology represented by the following **CRS** production possibility set: $P_{crs} = \{(x, y)|x \geq X\lambda, y \leq Y\lambda, \lambda \geq 0\}$, where $X = (x)_j \in R^{s \times n}$, $Y = (y)_j \in R^{m \times n}$ and $\lambda = (\lambda_1, ..., \lambda_n)^T$ is a intensity vector.

Based on the data matrix (X, Y), we measure the input oriented efficiency of `each observation o` by solving `n` times the following linear programming problems:

$$\begin{aligned} \underset{\theta,\lambda}{max} \quad & \theta \\ \text{s.t.} \quad & X\lambda \leq x_o \\ & \theta y_o \leq Y\lambda \\ & \lambda \geq 0 \end{aligned}$$

The measurement of technical efficiency assuming **VRS** considers the following production possibility set $P_{vrs} = \{(x, y)|x \geq X\lambda, y \leq Y\lambda, e\lambda = 1, \lambda \geq 0.\}$. Thus, the only difference with the CRS model is the adjunction of the

condition $\sum_{j=1}^{n} \lambda_j = 1$.

$$\begin{aligned} \underset{\theta,\lambda}{max} \quad & \theta \\ \text{s.t.} \quad & X\lambda \leq x_o \\ & \theta y_o \leq Y\lambda \\ & \sum_{j=1}^{n} \lambda_j = 1 \\ & \lambda \geq 0 \end{aligned}$$

Example: Output oriented DEA [.ipynb]

In the following code, we calculate the VRS radial model with pyStoNED.

```
# import packages
from pystoned import DEA
from pystoned import dataset as dataset
from pystoned.constant import RTS_VRS, ORIENT_OO, OPT_LOCAL

# import the data provided with Tim Coelli's Frontier 4.1
data = dataset.load_Tim_Coelli_frontier()

# define and solve the DEA radial model
model = DEA.DEA(data.y, data.x, rts=RTS_VRS, orient=ORIENT_OO, yref=None, xref=None)
model.optimize(OPT_LOCAL)

# display the technical efficiency
model.display_theta()

# display the intensity variables
model.display_lamda()
```

2.9.3 DEA: Distance function

Chambers et al. (1996) introduced the directional distance function (DDF) into efficiency measurement, and the inefficient DMUs can be projected to the frontier using direction $g = (g_x, g_y) \neq 0_{m+s}$, where $g_x \in R^m$ and $g_y \in R^s$.

The VRA and CRS models are presented as follows

1. CRS

$$\begin{aligned} \underset{\theta,\lambda}{max} \quad & \theta \\ \text{s.t.} \quad & X\lambda \leq x_o - \theta g_x \\ & Y\lambda \geq y_o + \theta g_y \\ & \lambda \geq 0 \end{aligned}$$

2. VRS

$$
\begin{aligned}
& \underset{\theta,\lambda}{max} && \theta \\
\text{s.t.} \quad & && X\lambda \leq x_o - \theta g_x \\
& && Y\lambda \geq y_o + \theta g_y \\
& && \sum_{j=1}^{n} \lambda_j = 1 \\
& && \lambda \geq 0
\end{aligned}
$$

Example: DEA with DDF [.ipynb]

```
# import packages
from pystoned import DEA
from pystoned import dataset as dataset
from pystoned.constant import RTS_VRS, OPT_LOCAL

# import the data provided with Tim Coelli's Frontier 4.1
data = dataset.load_Tim_Coelli_frontier()

# define and solve the DEA DDF model
model = DEA.DDF(y=data.y, x=data.x, b=None, gy=[1], gx=[0.0, 0.0], gb=None, rts=RTS_VRS,␣
↪yref=None, xref=None, bref=None)
model.optimize(OPT_LOCAL)

# display the technical efficiency
model.display_theta()

# display the intensity variables
model.display_lamda()
```

2.9.4 DEA: Undesirable Output (DDF)

With help of DDF, we can resort to DEA to evulate the DMUs with undesirable output. The corresponding CRS and VRS models are presented as follows

1. CRS

$$
\begin{aligned}
& \underset{\theta,\lambda}{max} && \theta \\
\text{s.t.} \quad & && X\lambda \leq x_o - \theta g_x \\
& && B\lambda = b_o - \theta g_b \\
& && Y\lambda \geq y_o + \theta g_y \\
& && \lambda \geq 0
\end{aligned}
$$

2. VRS

$$
\begin{aligned}
\underset{\theta,\lambda}{max} \quad & \theta \\
\text{s.t.} \quad & X\lambda \leq x_o - \theta g_x \\
& B\lambda = b_o - \theta g_b \\
& Y\lambda \geq y_o + \theta g_y \\
& \sum_{j=1}^{n} \lambda_j = 1 \\
& \lambda \geq 0
\end{aligned}
$$

Example: DEA-DDF with bad outputs [.ipynb]

```
# import packages
from pystoned import DEA
from pystoned import dataset as dataset
from pystoned.constant import RTS_VRS, OPT_LOCAL

# import the GHG example data
data = dataset.load_GHG_abatement_cost()

# define and solve the DEA DDF model
model = DEA.DDF(y=data.y, x=data.x, b=data.b, gy=[1], gx=[0.0, 0.0], gb=[-1], rts=RTS_
↪VRS, yref=None, xref=None, bref=None)
model.optimize(OPT_LOCAL)

# display the technical efficiency
model.display_theta()

# display the intensity variables
model.display_lamda()
```

2.9.5 DEA: Multiplier Model

The dual linear programming problems to the envelopment models are called multiplier models. See the following Table:

Frontier Type	Input-Oriented	Output-Oriented
	$\max \sum_{r=1}^{s} \mu_r y_{ro} + \mu$ subject to $\sum_{r=1}^{s} \mu_r y_{rj} - \sum_{i=1}^{m} \nu_i x_{ij} + \mu \leq 0$ $\sum_{i=1}^{m} \nu_i x_{io} = 1$ $\mu_r, \nu_i \geq 0 (\varepsilon)$	$\min \sum_{i=1}^{m} \nu_i x_{io} + \nu$ subject to $\sum_{i=1}^{m} \nu_i x_{ij} - \sum_{r=1}^{s} \mu_r y_{rj} + \nu \geq 0$ $\sum_{r=1}^{s} \mu_r y_{ro} = 1$ $\mu_r, \nu_i \geq 0 (\varepsilon)$
CRS	where $\mu = 0$	where $\nu = 0$
VRS	where μ free	where ν free
NIRS	where $\mu \leq 0$	where $\nu \geq 0$
NDRS	where $\mu \geq 0$	where $\nu \leq 0$

(*Source: Zhu, J., 2009. Multiplier and slack-based models. In Quantitative Models for Performance Evaluation and Benchmarking. Springer, Boston, MA.*)

Example: DEA dual models [.ipynb]

```
# import packages
from pystoned import DEA
from pystoned import dataset as dataset
from pystoned.constant import RTS_VRS, OPT_LOCAL, ORIENT_IO

# import the data provided with Tim Coelli's Frontier 4.1
data = dataset.load_Tim_Coelli_frontier()

# define and solve the DEA multiplier model
model = DEA.DUAL(data.y, data.x, rts=RTS_VRS, orient=ORIENT_IO, yref=None, xref=None)
model.optimize(OPT_LOCAL)

# display the multiplier y
model.display_mu()

# display the multiplier x
model.display_nu()

# print the technical efficiency
print(model.get_efficiency())
```

2.9.6 DEA: Change reference set

Sometimes we need to draw a distinction between the set of **reference** DMUs that characterize the frontier and the set of **evaluated** DMUs used as in the DEA problem. However, by default, all DMUs serve both as benchmarks and evaluated units.

Example: DEA with reference DMUs [.ipynb]

In the following code, we present how to evaluate the DMUs based on the any given reference set:

```
# import packages
from pystoned import DEA
from pystoned.constant import RTS_VRS, ORIENT_OO, OPT_LOCAL

# evaluated DMUs
x = np.array([100,200,300,500,100,200,600,400,550,600])
y = np.array([75,100,300,400,25,50,400, 260, 180, 240])

# reference DMUs
xref = np.array([100,300,500,100,600])
yref = np.array([75,300,400,25,400])

# define and solve the DEA radial model
model = DEA.DEA(y, x, rts=RTS_VRS, orient=ORIENT_OO, yref=yref, xref=xref)
model.optimize(OPT_LOCAL)

# display the technical efficiency
model.display_theta()

# display the intensity variables
model.display_lamda()
```

2.10 Free Disposal Hull

2.10.1 FDH: Input orientation

The FDH estimator was first proposed by Deprins et al. (1984), and the mixed integer linear program (MILP) formulation of FDH was introduced by Tulkens (1993).

Given the input variables $X = [x_1, x_2, \cdots, x_n]$ and output variables $Y = [y_1, y_2, \cdots, y_n]$, we measure the input oriented efficiency for observation i by solving the following MILP problem:

$$\begin{aligned} \min_{\phi,\lambda} \quad & \phi_i \\ \text{s.t.} \quad & X\lambda \leq \phi_i x_i \\ & Y\lambda \geq y_i \\ & \sum \lambda = 1 \\ & \lambda_j \in \{0,1\}, \forall j \end{aligned}$$

where $\lambda = [\lambda_1, \lambda_2, \cdots, \lambda_n]$ is the vector of intensity weights. The efficiency of observation i is ϕ_i^*. The corresponding calculation processes are as follow:

Example: Intput oriented FDH [.ipynb]

```
# import packages
from pystoned import FDH
from pystoned import dataset as dataset
from pystoned.constant import ORIENT_IO, OPT_LOCAL
```

(continues on next page)

(continued from previous page)

```
# import the data provided with Tim Coelli's Frontier 4.1
data = dataset.load_Tim_Coelli_frontier()

# define and solve the FDH model
model = FDH.FDH(data.y, data.x, orient=ORIENT_IO, yref=None, xref=None)
model.optimize(OPT_LOCAL)

# display the technical efficiency
model.display_theta()

# display the intensity variables
model.display_lamda()
```

2.10.2 FDH: Output orientation

The FDH estimator was first proposed by Deprins et al. (1984), and the mixed integer linear program (MILP) formulation of FDH was introduced by Tulkens (1993).

Given the input variables $X = [x_1, x_2, \cdots, x_n]$ and output variables $Y = [y_1, y_2, \cdots, y_n]$, we measure the output oriented efficiency for observation i by solving the following MILP problem:

$$\begin{aligned} \max_{\phi,\lambda} \quad & \phi_i \\ \text{s.t.} \quad & X\lambda \leq x_i \\ & Y\lambda \geq y_i\phi_i \\ & \sum \lambda = 1 \\ & \lambda_j \in \{0,1\}, \forall j \end{aligned}$$

where $\lambda = [\lambda_1, \lambda_2, \cdots, \lambda_n]$ is the vector of intensity weights. The efficiency of observation i is ϕ_i^*. The corresponding calculation processes are as follow:

Example: Output oriented FDH [.ipynb]

```
# import packages
from pystoned import FDH
from pystoned import dataset as dataset
from pystoned.constant import ORIENT_OO, OPT_LOCAL

# import the data provided with Tim Coelli's Frontier 4.1
data = dataset.load_Tim_Coelli_frontier()

# define and solve the FDH model
model = FDH.FDH(data.y, data.x, orient=ORIENT_OO, yref=None, xref=None)
model.optimize(OPT_LOCAL)

# display the technical efficiency
model.display_theta()

# display the intensity variables
model.display_lamda()
```

CHAPTER

THREE

DATASETS

In this section, the package provides four example datasets: First two are used in large number of CNLS/StoNED liturature; the others are commonly used in the SFA liturature. In the Examples, our tutorials will resort to these example data.

3.1 Regulation of Finnish electricity distribution firms

data from Kuosmanen T. (2012)

This dataset contains the following variables:

- the controllable operational expenditure: `OPEX`
- the total capital expenditure: `CAPEX`
- the total expenditure: `TOTEX`
- the weighted amount of energy transmitted through the network GWh: `Energy`
- the total length of the network km: `Length`
- the total number of customers connected to the network: `Customers`
- Underground cabling: `PerUndGr`

References:

[1] Kuosmanen, T. (2012) Stochastic semi-nonparametric frontier estimation of electricity distribution networks: Application of the StoNED method in the Finnish regulatory model, Energy Economics. 34, pp. 2189–2199.

[2] Kuosmanen, T., Saastamoinen, A. and Sipiläinen, T. (2013) What is the best practice for benchmark regulation of electricity distribution? Comparison of DEA, SFA and StoNED methods, Energy Policy. 61, pp. 740–750.

3.2 GHG abatement cost of OECD countries

data from Kuosmanen T. et al. (2020)

This dataset contains the following variables:

- net capital stock (billion euro^2010): `CPNK`
- hours worked by total engaged (billion hours): `HRSN`
- value added (billion euro^2010): `VALK`
- total GHG emissions(million tons of CO_2 equivalents): `GHG`

References:

[1] Kuosmanen, T., Zhou, X. and Dai, S. (2020). How much climate policy has cost for OECD countries? World Development, 125, p. 104681.

3.3 Data provided with Tim Coelli's Frontier 4.1

data from Coelli et al. (1996)

This dataset contains the following variables:

- firm ID: `firm`
- output quantity (value added): `output`
- capital input quantity (quantity index): `capital`
- labour input quantity (quantity index): `labour`

References:

[1] Coelli, T. (1996) A Guide to FRONTIER Version 4.1: A Computer Program for Stochastic Frontier Production and Cost Function Estimation, CEPA Working Paper 96/08, University of New England.

3.4 Rice Production in the Philippines

data from Coelli et al. (2005)

This dataset contains the following variables:

- Time period (1= 1990, . . . , 8 = 1997): `YEARDUM`
- Farmer code (1, . . . , 43): `FMERCODE`
- Output (tonnes of freshly threshed rice): `PROD`
- Area planted (hectares): `AREA`
- Labour used (man-days of family and hired labour): `LABOR`
- Fertiliser used (kg of active ingredients): `NPK`
- Other inputs used (Laspeyres index = 100 for Firm 17 in 1991): `OTHER`
- Output price (pesos per kg): `PRICE`
- Rental price of land (pesos per hectare): `AREAP`
- Labour price (pesos per hired man-day: `LABORP`
- Fertiliser price (pesos per kg of active ingredient): `NPKP`
- Price of other inputs (implicit price index): `OTHERP`
- Age of the household head (years): `AGE`
- Education of the household head (years): `EDYRS`
- Household size: `HHSIZE`
- Number of adults in the household: `NADULT`
- Percentage of area classified as bantog (upland) fields: `BANRAT`

References:

[1] Coelli, T. J., Rao, D. S. P., O'Donnell, C. J., and Battese, G. E. (2005) An Introduction to Efficiency and Productivity Analysis, Springer, New York.

3.5 Import internal data

- Finnish electricity firm data

```
# import dataset module
from pystoned.dataset import load_Finnish_electricity_firm

# import all data (including the contextual varibale)
data = load_Finnish_electricity_firm(x_select=['Energy', 'Length', 'Customers'],
                                     y_select=['TOTEX'],
                                     z_select=['PerUndGr'])
x, y, z = data.x, data.y, data.z

# print data
print(x)
print(y)
print(z)

# (OR) import data (only inputs and output)
data = load_Finnish_electricity_firm(x_select=['Energy', 'Length', 'Customers'],
                                     y_select=['TOTEX'])
x, y = data.x, data.y

# print data
print(x)
print(y)
```

- import OECD GHG emissions data

```
# import dataset module
from pystoned.dataset import load_GHG_abatement_cost

# import all data
data = load_GHG_abatement_cost(x_select=['HRSN', 'CPNK'],
                               y_select=['VALK'],
                               b_select=['GHG'])
x, y, b = data.x, data.y, data.b

# print data
print(x)
print(y)
print(b)
```

- import Tim Coelli's Frontier 4.1 data

```
# import dataset module
from pystoned.dataset import load_Tim_Coelli_frontier

# import all data
```

(continues on next page)

(continued from previous page)

```
data = load_Tim_Coelli_frontier(x_select=['capital', 'labour'],
                                y_select=['output'])
x, y = data.x, data.y

# print data
print(x)
print(y)
```

- import rice production data

```
# import dataset module
from pystoned.dataset import load_Philipines_rice_production

# import all data
data = load_Philipines_rice_production(x_select=['AREA', 'LABOR', 'NPK', 'OTHER', 'AREAP
↪', 'LABORP', 'NPKP', 'OTHERP'],
                                       y_select=['PROD', 'PRICE'])
x, y = data.x, data.y

# print data
print(x)
print(y)

# (OR) import partial data (two input-one output)
data = load_Philipines_rice_production(x_select=['LABOR', 'NPK'],
                                       y_select=['PROD'])
x, y = data.x, data.y

# print data
print(x)
print(y)
```

3.6 Import external data

Assuming that we have a dataset like the following example in *Book1.xlsx*, we then use the Panda to read the Excel file and organize the data using the Numpy.

ID	output	input1	input2	input3	z_var
i1	120	10	55	103	0.8
i2	80	30	49	120	0.6
i3	90	25	72	150	0.3
i4	110	16	39	100	0.5
…	…	…	…	…	…

```
# import basic modules
import numpy as np
import pandas as pd

# import Excel data
```

(continues on next page)

(continued from previous page)

```
df = pd.read_excel("Book1.xlsx")

# output: y
y = df['output']

# inputs: X
x1 = df['input1']
x1 = np.asmatrix(x1).T
x2 = df['input2']
x2 = np.asmatrix(x2).T
x3 = df['input3']
x3 = np.asmatrix(x3).T
x  = np.concatenate((x1, x2, x3), axis=1)

# contextual Variable: z
z = df['z_var']
```

CHAPTER

FOUR

API DOCUMENTATION

The API Documentation including 3 parts: Formulations classes, Residual decomposition and Peripheral methods.

4.1 Formulations Classes

4.1.1 CNLS

class pystoned.CNLS.**CNLS**(*y*, *x*, *z=None*, *cet='addi'*, *fun='prod'*, *rts='vrs'*)

Convex Nonparametric Least Square (CNLS)

__init__(*y*, *x*, *z=None*, *cet='addi'*, *fun='prod'*, *rts='vrs'*)

CNLS model

Parameters

- **y** (*float*) – output variable.
- **x** (*float*) – input variables.
- **z** (*float, optional*) – Contextual variable(s). Defaults to None.
- **cet** (*String, optional*) – CET_ADDI (additive composite error term) or CET_MULT (multiplicative composite error term). Defaults to CET_ADDI.
- **fun** (*String, optional*) – FUN_PROD (production frontier) or FUN_COST (cost frontier). Defaults to FUN_PROD.
- **rts** (*String, optional*) – RTS_VRS (variable returns to scale) or RTS_CRS (constant returns to scale). Defaults to RTS_VRS.

display_alpha()

Display alpha value

display_beta()

Display beta value

display_lamda()

Display lamda value

display_residual()

Dispaly residual value

display_status()

Display the status of problem

get_adjusted_alpha()
Return the shifted constatnt(alpha) term by CCNLS

get_adjusted_residual()
Return the shifted residuals(epsilon) tern by CCNLS

get_alpha()
Return alpha value by array

get_beta()
Return beta value by array

get_frontier()
Return estimated frontier value by array

get_lamda()
Return lamda value by array

get_predict(*x_test*)
Return the estimated function in testing sample

get_residual()
Return residual value by array

get_status()
Return status

optimize(*email='local'*, *solver=None*)
Optimize the function by requested method

Parameters

- **email** (*string*) – The email address for remote optimization. It will optimize locally if OPT_LOCAL is given.
- **solver** (*string*) – The solver chosen for optimization. It will optimize with default solver if OPT_DEFAULT is given.

4.1.2 CNLSDDF

class pystoned.CNLSDDF.**CNLSDDF**(*y*, *x*, *b=None*, *gy=[1]*, *gx=[1]*, *gb=None*, *fun='prod'*)
Convex Nonparametric Least Square with directional distance function

__init__(*y*, *x*, *b=None*, *gy=[1]*, *gx=[1]*, *gb=None*, *fun='prod'*)
CNLS DDF model

Parameters

- **y** (*float*) – output variable.
- **x** (*float*) – input variables.
- **b** (*float), optional*) – undesirable output variables. Defaults to None.
- **gy** (*list, optional*) – output directional vector. Defaults to [1].
- **gx** (*list, optional*) – input directional vector. Defaults to [1].
- **gb** (*list, optional*) – undesirable output directional vector. Defaults to None.
- **fun** (*String, optional*) – FUN_PROD (production frontier) or FUN_COST (cost frontier). Defaults to FUN_PROD.

display_alpha()
Display alpha value

display_beta()
Display beta value

display_delta()
Display delta value

display_gamma()
Display gamma value

display_lamda()
Display lamda value

display_residual()
Dispaly residual value

display_status()
Display the status of problem

get_adjusted_alpha()
Return the shifted constatnt(alpha) term by CCNLS

get_adjusted_residual()
Return the shifted residuals(epsilon) tern by CCNLS

get_alpha()
Return alpha value by array

get_beta()
Return beta value by array

get_delta()
Return delta value by array

get_frontier()
Return estimated frontier value by array

get_gamma()
Return gamma value by array

get_lamda()
Return lamda value by array

get_predict(*x_test*)
Return the estimated function in testing sample

get_residual()
Return residual value by array

get_status()
Return status

optimize(*email='local'*, *solver=None*)
Optimize the function by requested method

Parameters

- **email** (*string*) – The email address for remote optimization. It will optimize locally if OPT_LOCAL is given.
- **solver** (*string*) – The solver chosen for optimization. It will optimize with default solver if OPT_DEFAULT is given.

4.1.3 CNLSG

class pystoned.CNLSG.**CNLSG**(*y*, *x*, *z=None*, *cet='addi'*, *fun='prod'*, *rts='vrs'*)

Convex Nonparametric Least Square (CNLS) with Genetic algorithm

__init__(*y*, *x*, *z=None*, *cet='addi'*, *fun='prod'*, *rts='vrs'*)

CNLSG model

Parameters

- **y** (*float*) – output variable.
- **x** (*float*) – input variables.
- **z** (*float, optional*) – Contextual variable(s). Defaults to None.
- **cet** (*String, optional*) – CET_ADDI (additive composite error term) or CET_MULT (multiplicative composite error term). Defaults to CET_ADDI.
- **fun** (*String, optional*) – FUN_PROD (production frontier) or FUN_COST (cost frontier). Defaults to FUN_PROD.
- **rts** (*String, optional*) – RTS_VRS (variable returns to scale) or RTS_CRS (constant returns to scale). Defaults to RTS_VRS.

display_alpha()

Display alpha value

display_beta()

Display beta value

display_lamda()

Display lamda value

display_residual()

Dispaly residual value

display_status()

Display the status of problem

get_alpha()

Return alpha value by array

get_beta()

Return beta value by array

get_blocks()

Return the number of blocks

get_frontier()

Return estimated frontier value by array

get_lamda()

Return beta value by array

get_predict(*x_test*)

Return the estimated function in testing sample

get_residual()

Return residual value by array

get_runningtime()

Return the running time

get_status()

Return status

get_totalconstr()

Return the number of total constraints

optimize(*email='local'*, *solver=None*)

Optimize the function by requested method

4.1.4 CNLSRDF

class pystoned.CNLSRDF.**CNLSRDF**(*y*, *x*, *z=None*, *rdf='DI'*, *rts='vrs'*)

Convex Nonparametric Least Square with radial distance function

__init__(*y*, *x*, *z=None*, *rdf='DI'*, *rts='vrs'*)

CNLSRDF model :param y: output variable. :type y: float :param x: input variables. :type x: float :param z: control variables. Defaults to None. :type z: float, optional :param cet: CET_ADDI (additive composite error term) or CET_MULT (multiplicative composite error term). Defaults to CET_ADDI. :type cet: String, optional :param rdf: RDF_DI (input distance function) or RDF_DO (output distance function). Defaults to RDF_DI. :type rdf: String, optional :param rts: RTS_VRS (variable returns to scale) or RTS_CRS (constant returns to scale). Defaults to RTS_VRS. :type rts: String, optional

display_alpha()

Display alpha value

display_beta()

Display beta value

display_gamma()

Display gamma value

display_kappa()

Display kappa value

display_lamda()

Display lamda value

display_residual()

Dispaly residual value

display_status()

Display the status of problem

get_adjusted_alpha()

Return the shifted constatnt(alpha) term by CCNLS

get_adjusted_residual()

Return the shifted residuals(epsilon) tern by CCNLS

get_alpha()

Return alpha value by array

get_beta()

Return beta value by array

get_chi()

Return estimated radial distance function by array

get_frontier()

Return estimated frontier value by array

get_gamma()

Return gamma value by array

get_kappa()

Return kappa value by array

get_lamda()

Return lamda value by array

get_predict(*x_test*)

Return the estimated function in testing sample

get_residual()

Return residual value by array

get_status()

Return status

optimize(*email='local'*, *solver=None*)

Optimize the function by requested method :param email: The email address for remote optimization. It will optimize locally if OPT_LOCAL is given. :type email: string :param solver: The solver chosen for optimization. It will optimize with default solver if OPT_DEFAULT is given. :type solver: string

4.1.5 CQER

class pystoned.CQER.**CER**(*y*, *x*, *tau*, *z=None*, *cet='addi'*, *fun='prod'*, *rts='vrs'*)

Convex expectile regression (CER)

__init__(*y*, *x*, *tau*, *z=None*, *cet='addi'*, *fun='prod'*, *rts='vrs'*)

CER model

Parameters

- **y** (*float*) – output variable.
- **x** (*float*) – input variables.
- **z** (*float, optional*) – Contextual variable(s). Defaults to None.
- **tau** (*float*) – expectile.
- **cet** (*String, optional*) – CET_ADDI (additive composite error term) or CET_MULT (multiplicative composite error term). Defaults to CET_ADDI.
- **fun** (*String, optional*) – FUN_PROD (production frontier) or FUN_COST (cost frontier). Defaults to FUN_PROD.
- **rts** (*String, optional*) – RTS_VRS (variable returns to scale) or RTS_CRS (constant returns to scale). Defaults to RTS_VRS.

display_alpha()
Display alpha value

display_beta()
Display beta value

display_lamda()
Display lamda value

display_negative_residual()
Dispaly negative residual value

display_positive_residual()
Dispaly positive residual value

display_residual()
Dispaly residual value

display_status()
Display the status of problem

get_alpha()
Return alpha value by array

get_beta()
Return beta value by array

get_frontier()
Return estimated frontier value by array

get_lamda()
Return beta value by array

get_negative_residual()
Return negative residual value by array

get_positive_residual()
Return positive residual value by array

get_predict(*x_test*)
Return the estimated function in testing sample

get_residual()
Return residual value by array

get_status()
Return status

optimize(*email='local'*, *solver=None*)
Optimize the function by requested method

Parameters

- **email** (*string*) – The email address for remote optimization. It will optimize locally if OPT_LOCAL is given.
- **solver** (*string*) – The solver chosen for optimization. It will optimize with default solver if OPT_DEFAULT is given.

class pystoned.CQER.CQR(*y*, *x*, *tau*, *z=None*, *cet='addi'*, *fun='prod'*, *rts='vrs'*)

Convex quantile regression (CQR)

__init__(*y*, *x*, *tau*, *z=None*, *cet='addi'*, *fun='prod'*, *rts='vrs'*)

CQR model

Parameters

- **y** (*float*) – output variable.
- **x** (*float*) – input variables.
- **z** (*float, optional*) – Contextual variable(s). Defaults to None.
- **tau** (*float*) – quantile.
- **cet** (*String, optional*) – CET_ADDI (additive composite error term) or CET_MULT (multiplicative composite error term). Defaults to CET_ADDI.
- **fun** (*String, optional*) – FUN_PROD (production frontier) or FUN_COST (cost frontier). Defaults to FUN_PROD.
- **rts** (*String, optional*) – RTS_VRS (variable returns to scale) or RTS_CRS (constant returns to scale). Defaults to RTS_VRS.

display_alpha()

Display alpha value

display_beta()

Display beta value

display_lamda()

Display lamda value

display_negative_residual()

Dispaly negative residual value

display_positive_residual()

Dispaly positive residual value

display_residual()

Dispaly residual value

display_status()

Display the status of problem

get_alpha()

Return alpha value by array

get_beta()

Return beta value by array

get_frontier()

Return estimated frontier value by array

get_lamda()

Return beta value by array

get_negative_residual()

Return negative residual value by array

get_positive_residual()

Return positive residual value by array

get_predict(*x_test*)

Return the estimated function in testing sample

get_residual()

Return residual value by array

get_status()

Return status

optimize(*email='local'*, *solver=None*)

Optimize the function by requested method

Parameters

- **email** (*string*) – The email address for remote optimization. It will optimize locally if OPT_LOCAL is given.
- **solver** (*string*) – The solver chosen for optimization. It will optimize with default solver if OPT_DEFAULT is given.

4.1.6 CQERDDF

class pystoned.CQERDDF.**CERDDF**(*y*, *x*, *b=None*, *gy=[1]*, *gx=[1]*, *gb=None*, *fun='prod'*, *tau=0.5*)

Convex expectile regression with DDF formulation

__init__(*y*, *x*, *b=None*, *gy=[1]*, *gx=[1]*, *gb=None*, *fun='prod'*, *tau=0.5*)

CER DDF

Parameters

- **y** (*float*) – output variable.
- **x** (*float*) – input variables.
- **b** (*float), optional*) – undesirable output variables. Defaults to None.
- **gy** (*list, optional*) – output directional vector. Defaults to [1].
- **gx** (*list, optional*) – input directional vector. Defaults to [1].
- **gb** (*list, optional*) – undesirable output directional vector. Defaults to None.
- **fun** (*String, optional*) – FUN_PROD (production frontier) or FUN_COST (cost frontier). Defaults to FUN_PROD.
- **tau** (*float, optional*) – expectile. Defaults to 0.5.

display_alpha()

Display alpha value

display_beta()

Display beta value

display_delta()

Display delta value

display_gamma()

Display gamma value

display_lamda()
Display lamda value

display_negative_residual()
Dispaly negative residual value

display_positive_residual()
Dispaly positive residual value

display_residual()
Dispaly residual value

display_status()
Display the status of problem

get_adjusted_alpha()
Return the shifted constatnt(alpha) term by CCNLS

get_adjusted_residual()
Return the shifted residuals(epsilon) tern by CCNLS

get_alpha()
Return alpha value by array

get_beta()
Return beta value by array

get_delta()
Return delta value by array

get_frontier()
Return estimated frontier value by array

get_gamma()
Return gamma value by array

get_lamda()
Return lamda value by array

get_negative_residual()
Return negative residual value by array

get_positive_residual()
Return positive residual value by array

get_predict(*x_test*)
Return the estimated function in testing sample

get_residual()
Return residual value by array

get_status()
Return status

optimize(*email='local'*, *solver=None*)
Optimize the function by requested method

Parameters

- **email** (*string*) – The email address for remote optimization. It will optimize locally if OPT_LOCAL is given.
- **solver** (*string*) – The solver chosen for optimization. It will optimize with default solver if OPT_DEFAULT is given.

class pystoned.CQERDDF.**CQRDDF**(*y*, *x*, *b=None*, *gy=[1]*, *gx=[1]*, *gb=None*, *fun='prod'*, *tau=0.5*)

Convex quantile regression with directional distance function

__init__(*y*, *x*, *b=None*, *gy=[1]*, *gx=[1]*, *gb=None*, *fun='prod'*, *tau=0.5*)

CQR DDF

Parameters

- **y** (*float*) – output variable.
- **x** (*float*) – input variables.
- **b** (*float), optional*) – undesirable output variables. Defaults to None.
- **gy** (*list, optional*) – output directional vector. Defaults to [1].
- **gx** (*list, optional*) – input directional vector. Defaults to [1].
- **gb** (*list, optional*) – undesirable output directional vector. Defaults to None.
- **fun** (*String, optional*) – FUN_PROD (production frontier) or FUN_COST (cost frontier). Defaults to FUN_PROD.
- **tau** (*float, optional*) – quantile. Defaults to 0.5.

display_alpha()

Display alpha value

display_beta()

Display beta value

display_delta()

Display delta value

display_gamma()

Display gamma value

display_lamda()

Display lamda value

display_negative_residual()

Dispaly negative residual value

display_positive_residual()

Dispaly positive residual value

display_residual()

Dispaly residual value

display_status()

Display the status of problem

get_adjusted_alpha()

Return the shifted constatnt(alpha) term by CCNLS

get_adjusted_residual()

Return the shifted residuals(epsilon) tern by CCNLS

get_alpha()

Return alpha value by array

get_beta()

Return beta value by array

get_delta()

Return delta value by array

get_frontier()

Return estimated frontier value by array

get_gamma()

Return gamma value by array

get_lamda()

Return lamda value by array

get_negative_residual()

Return negative residual value by array

get_positive_residual()

Return positive residual value by array

get_predict(*x_test*)

Return the estimated function in testing sample

get_residual()

Return residual value by array

get_status()

Return status

optimize(*email='local'*, *solver=None*)

Optimize the function by requested method

Parameters

- **email** (*string*) – The email address for remote optimization. It will optimize locally if OPT_LOCAL is given.
- **solver** (*string*) – The solver chosen for optimization. It will optimize with default solver if OPT_DEFAULT is given.

4.1.7 CQERG

class pystoned.CQERG.**CERG**(*y*, *x*, *tau*, *z=None*, *cet='addi'*, *fun='prod'*, *rts='vrs'*)

Convex expectile regression (CER) with Genetic algorithm

__init__(*y*, *x*, *tau*, *z=None*, *cet='addi'*, *fun='prod'*, *rts='vrs'*)

CERG model

Parameters

- **y** (*float*) – output variable.
- **x** (*float*) – input variables.

- **tau** (*float*) – quantile.
- **z** (*float, optional*) – Contextual variable(s). Defaults to None.
- **cet** (*String, optional*) – CET_ADDI (additive composite error term) or CET_MULT (multiplicative composite error term). Defaults to CET_ADDI.
- **fun** (*String, optional*) – FUN_PROD (production frontier) or FUN_COST (cost frontier). Defaults to FUN_PROD.
- **rts** (*String, optional*) – RTS_VRS (variable returns to scale) or RTS_CRS (constant returns to scale). Defaults to RTS_VRS.

display_alpha()

Display alpha value

display_beta()

Display beta value

display_lamda()

Display lamda value

display_negative_residual()

Dispaly negative residual value

display_positive_residual()

Dispaly positive residual value

display_residual()

Dispaly residual value

display_status()

Display the status of problem

get_alpha()

Return alpha value by array

get_beta()

Return beta value by array

get_blocks()

Return the number of blocks

get_frontier()

Return estimated frontier value by array

get_lamda()

Return beta value by array

get_negative_residual()

Return negative residual value by array

get_positive_residual()

Return positive residual value by array

get_predict(*x_test*)

Return the estimated function in testing sample

get_residual()

Return residual value by array

get_runningtime()

Return the running time

get_status()

Return status

get_totalconstr()

Return the number of total constraints

optimize(*email='local'*, *solver=None*)

Optimize the function by requested method

class pystoned.CQERG.**CQRG**(*y*, *x*, *tau*, *z=None*, *cet='addi'*, *fun='prod'*, *rts='vrs'*)

Convex quantile regression (CQR) with Genetic algorithm

__init__(*y*, *x*, *tau*, *z=None*, *cet='addi'*, *fun='prod'*, *rts='vrs'*)

CQRG model

Parameters

- **y** (*float*) – output variable.
- **x** (*float*) – input variables.
- **tau** (*float*) – quantile.
- **z** (*float, optional*) – Contextual variable(s). Defaults to None.
- **cet** (*String, optional*) – CET_ADDI (additive composite error term) or CET_MULT (multiplicative composite error term). Defaults to CET_ADDI.
- **fun** (*String, optional*) – FUN_PROD (production frontier) or FUN_COST (cost frontier). Defaults to FUN_PROD.
- **rts** (*String, optional*) – RTS_VRS (variable returns to scale) or RTS_CRS (constant returns to scale). Defaults to RTS_VRS.

display_alpha()

Display alpha value

display_beta()

Display beta value

display_lamda()

Display lamda value

display_negative_residual()

Dispaly negative residual value

display_positive_residual()

Dispaly positive residual value

display_residual()

Dispaly residual value

display_status()

Display the status of problem

get_alpha()

Return alpha value by array

get_beta()
Return beta value by array

get_blocks()
Return the number of blocks

get_frontier()
Return estimated frontier value by array

get_lamda()
Return beta value by array

get_negative_residual()
Return negative residual value by array

get_positive_residual()
Return positive residual value by array

get_predict(*x_test*)
Return the estimated function in testing sample

get_residual()
Return residual value by array

get_runningtime()
Return the running time

get_status()
Return status

get_totalconstr()
Return the number of total constraints

optimize(*email='local'*, *solver=None*)
Optimize the function by requested method

4.1.8 CSVR

class pystoned.CSVR.**CSVR**(*y*, *x*, *fun='prod'*, *epsilon=0.01*, *C=2*)
Convex Support Vector Regression (CSVR)

__init__(*y*, *x*, *fun='prod'*, *epsilon=0.01*, *C=2*)
CSVR model

Parameters

- **y** (*float*) – output variable.
- **x** (*float*) – input variables.
- **epsilon** (*float*) – epsilon-loss function.
- **C** (*float*) – Regularization parameter.
- **fun** (*String, optional*) – FUN_PROD (production frontier) or FUN_COST (cost frontier). Defaults to FUN_PROD.

display_alpha()
Display alpha value

display_beta()

Display beta value

display_status()

Display the status of problem

get_alpha()

Return alpha value by array

get_beta()

Return beta value by array

get_predict(*x_test*)

Return the estimated function in testing sample

get_status()

Return status

optimize(*email='local'*, *solver=None*)

Optimize the function by requested method

Parameters

- **email** (*string*) – The email address for remote optimization. It will optimize locally if OPT_LOCAL is given.
- **solver** (*string*) – The solver chosen for optimization. It will optimize with default solver if OPT_DEFAULT is given.

4.1.9 DEA

class pystoned.DEA.**DDF**(*y*, *x*, *b=None*, *gy=[1]*, *gx=[1]*, *gb=None*, *rts='vrs'*, *yref=None*, *xref=None*, *bref=None*)

__init__(*y*, *x*, *b=None*, *gy=[1]*, *gx=[1]*, *gb=None*, *rts='vrs'*, *yref=None*, *xref=None*, *bref=None*)

DEA: Directional distance function

Parameters

- **y** (*float*) – output variable.
- **x** (*float*) – input variables.
- **b** (*float), optional*) – undesirable output variables. Defaults to None.
- **gy** (*list, optional*) – output directional vector. Defaults to [1].
- **gx** (*list, optional*) – input directional vector. Defaults to [1].
- **gb** (*list, optional*) – undesirable output directional vector. Defaults to None.
- **rts** (*String*) – RTS_VRS (variable returns to scale) or RTS_CRS (constant returns to scale)
- **yref** (*String, optional*) – reference output. Defaults to None.
- **xref** (*String, optional*) – reference inputs. Defaults to None.
- **bref** (*String, optional*) – reference undesirable output. Defaults to None.

class pystoned.DEA.**DEA**(*y*, *x*, *orient*, *rts*, *yref=None*, *xref=None*)

Data Envelopment Analysis (DEA)

__init__(*y*, *x*, *orient*, *rts*, *yref=None*, *xref=None*)

DEA: Envelopment problem

Parameters

- **y** (*float*) – output variable.
- **x** (*float*) – input variables.
- **orient** (*String*) – ORIENT_IO (input orientation) or ORIENT_OO (output orientation)
- **rts** (*String*) – RTS_VRS (variable returns to scale) or RTS_CRS (constant returns to scale)
- **yref** (*String, optional*) – reference output. Defaults to None.
- **xref** (*String, optional*) – reference inputs. Defaults to None.

display_lamda()

Display lamda value

display_status()

Display the status of problem

display_theta()

Display theta value

get_lamda()

Return lamda value by array

get_status()

Return status

get_theta()

Return theta value by array

optimize(*email='local'*, *solver=None*)

Optimize the function by requested method

Parameters

- **email** (*string*) – The email address for remote optimization. It will optimize locally if OPT_LOCAL is given.
- **solver** (*string*) – The solver chosen for optimization. It will optimize with default solver if OPT_DEFAULT is given.

class pystoned.DEA.**DUAL**(*y*, *x*, *orient*, *rts*, *yref=None*, *xref=None*)

__init__(*y*, *x*, *orient*, *rts*, *yref=None*, *xref=None*)

DEA: Multiplier problem

Parameters

- **y** (*float*) – output variable.
- **x** (*float*) – input variables.
- **orient** (*String*) – ORIENT_IO (input orientation) or ORIENT_OO (output orientation)
- **rts** (*String*) – RTS_VRS (variable returns to scale) or RTS_CRS (constant returns to scale)
- **yref** (*String, optional*) – reference output. Defaults to None.

- **xref** (*String, optional*) – reference inputs. Defaults to None.

display_mu()
Display mu value

display_nu()
Display nu value

display_omega()
Display omega value

get_efficiency()
Return efficiency value by array

get_mu()
Return mu value by array

get_nu()
Return nu value by array

get_omega()
Return omega value by array

4.1.10 FDH

class pystoned.FDH.**FDH**(*y*, *x*, *orient*, *yref=None*, *xref=None*)
Free Disposal Hull (FDH)

__init__(*y*, *x*, *orient*, *yref=None*, *xref=None*)
FDH model

Parameters

- **y** (*float*) – output variable.
- **x** (*float*) – input variables.
- **orient** (*String*) – ORIENT_IO (input orientation) or ORIENT_OO (output orientation)
- **yref** (*String, optional*) – reference output. Defaults to None.
- **xref** (*String, optional*) – reference inputs. Defaults to None.

display_lamda()
Display lamda value

display_status()
Display the status of problem

display_theta()
Display theta value

get_lamda()
Return lamda value by array

get_status()
Return status

get_theta()
Return theta value by array

optimize(*email='local'*, *solver=None*)

Optimize the function by requested method

Parameters

- **email** (*string*) – The email address for remote optimization. It will optimize locally if OPT_LOCAL is given.
- **solver** (*string*) – The solver chosen for optimization. It will optimize with default solver if OPT_DEFAULT is given.

4.1.11 ICNLS

class pystoned.ICNLS.**ICNLS**(*y*, *x*, *z=None*, *cet='addi'*, *fun='prod'*, *rts='vrs'*)

Isotonic Convex Nonparametric Least Square (ICNLS)

__init__(*y*, *x*, *z=None*, *cet='addi'*, *fun='prod'*, *rts='vrs'*)

ICNLS model

Parameters

- **y** (*float*) – output variable.
- **x** (*float*) – input variables.
- **z** (*float, optional*) – Contextual variable(s). Defaults to None.
- **cet** (*String, optional*) – CET_ADDI (additive composite error term) or CET_MULT (multiplicative composite error term). Defaults to CET_ADDI.
- **fun** (*String, optional*) – FUN_PROD (production frontier) or FUN_COST (cost frontier). Defaults to FUN_PROD.
- **rts** (*String, optional*) – RTS_VRS (variable returns to scale) or RTS_CRS (constant returns to scale). Defaults to RTS_VRS.

display_alpha()

Display alpha value

display_beta()

Display beta value

display_lamda()

Display lamda value

display_residual()

Dispaly residual value

display_status()

Display the status of problem

get_adjusted_alpha()

Return the shifted constatnt(alpha) term by CCNLS

get_adjusted_residual()

Return the shifted residuals(epsilon) tern by CCNLS

get_alpha()

Return alpha value by array

get_beta()

Return beta value by array

get_frontier()

Return estimated frontier value by array

get_lamda()

Return lamda value by array

get_predict(*x_test*)

Return the estimated function in testing sample

get_residual()

Return residual value by array

get_status()

Return status

optimize(*email='local'*, *solver=None*)

Optimize the function by requested method

Parameters

- **email** (*string*) – The email address for remote optimization. It will optimize locally if OPT_LOCAL is given.
- **solver** (*string*) – The solver chosen for optimization. It will optimize with default solver if OPT_DEFAULT is given.

4.1.12 ICQER

class pystoned.ICQER.**ICER**(*y*, *x*, *tau*, *z=None*, *cet='addi'*, *fun='prod'*, *rts='vrs'*)

Isotonic convex expectile regression (ICER)

__init__(*y*, *x*, *tau*, *z=None*, *cet='addi'*, *fun='prod'*, *rts='vrs'*)

ICER model

Parameters

- **y** (*float*) – output variable.
- **x** (*float*) – input variables.
- **tau** (*float*) – expectile.
- **z** (*float, optional*) – Contextual variable(s). Defaults to None.
- **cet** (*String, optional*) – CET_ADDI (additive composite error term) or CET_MULT (multiplicative composite error term). Defaults to CET_ADDI.
- **fun** (*String, optional*) – FUN_PROD (production frontier) or FUN_COST (cost frontier). Defaults to FUN_PROD.
- **rts** (*String, optional*) – RTS_VRS (variable returns to scale) or RTS_CRS (constant returns to scale). Defaults to RTS_VRS.

display_alpha()

Display alpha value

display_beta()

Display beta value

display_lamda()

Display lamda value

display_negative_residual()

Dispaly negative residual value

display_positive_residual()

Dispaly positive residual value

display_residual()

Dispaly residual value

display_status()

Display the status of problem

get_adjusted_alpha()

Return the shifted constatnt(alpha) term by CCNLS

get_adjusted_residual()

Return the shifted residuals(epsilon) tern by CCNLS

get_alpha()

Return alpha value by array

get_beta()

Return beta value by array

get_frontier()

Return estimated frontier value by array

get_lamda()

Return lamda value by array

get_negative_residual()

Return negative residual value by array

get_positive_residual()

Return positive residual value by array

get_predict(*x_test*)

Return the estimated function in testing sample

get_residual()

Return residual value by array

get_status()

Return status

optimize(*email='local'*, *solver=None*)

Optimize the function by requested method

Parameters

- **email** (*string*) – The email address for remote optimization. It will optimize locally if OPT_LOCAL is given.
- **solver** (*string*) – The solver chosen for optimization. It will optimize with default solver if OPT_DEFAULT is given.

class pystoned.ICQER.**ICQR**(*y*, *x*, *tau*, *z=None*, *cet='addi'*, *fun='prod'*, *rts='vrs'*)

Isotonic convex quantile regression (ICQR)

__init__(*y*, *x*, *tau*, *z=None*, *cet='addi'*, *fun='prod'*, *rts='vrs'*)

ICQR model

Parameters

- **y** (*float*) – output variable.
- **x** (*float*) – input variables.
- **tau** (*float*) – quantile.
- **z** (*float, optional*) – Contextual variable(s). Defaults to None.
- **cet** (*String, optional*) – CET_ADDI (additive composite error term) or CET_MULT (multiplicative composite error term). Defaults to CET_ADDI.
- **fun** (*String, optional*) – FUN_PROD (production frontier) or FUN_COST (cost frontier). Defaults to FUN_PROD.
- **rts** (*String, optional*) – RTS_VRS (variable returns to scale) or RTS_CRS (constant returns to scale). Defaults to RTS_VRS.

display_alpha()

Display alpha value

display_beta()

Display beta value

display_lamda()

Display lamda value

display_negative_residual()

Dispaly negative residual value

display_positive_residual()

Dispaly positive residual value

display_residual()

Dispaly residual value

display_status()

Display the status of problem

get_adjusted_alpha()

Return the shifted constatnt(alpha) term by CCNLS

get_adjusted_residual()

Return the shifted residuals(epsilon) tern by CCNLS

get_alpha()

Return alpha value by array

get_beta()

Return beta value by array

get_frontier()

Return estimated frontier value by array

get_lamda()

Return lamda value by array

get_negative_residual()

Return negative residual value by array

get_positive_residual()

Return positive residual value by array

get_predict(*x_test*)

Return the estimated function in testing sample

get_residual()

Return residual value by array

get_status()

Return status

optimize(*email='local'*, *solver=None*)

Optimize the function by requested method

Parameters

- **email** (*string*) – The email address for remote optimization. It will optimize locally if OPT_LOCAL is given.
- **solver** (*string*) – The solver chosen for optimization. It will optimize with default solver if OPT_DEFAULT is given.

4.1.13 pCNLS

class pystoned.pCNLS.**pCNLS**(*y*, *x*, *eta*, *z=None*, *cet='addi'*, *fun='prod'*, *rts='vrs'*, *penalty=1*)

penalized Convex Nonparametric Least Square (pCNLS)

__init__(*y*, *x*, *eta*, *z=None*, *cet='addi'*, *fun='prod'*, *rts='vrs'*, *penalty=1*)

pCNLS model

Parameters

- **y** (*float*) – output variable.
- **x** (*float*) – input variables.
- **eta** (*float*) – regularization parameter.
- **z** (*float, optional*) – Contextual variable(s). Defaults to None.
- **cet** (*String, optional*) – CET_ADDI (additive composite error term) or CET_MULT (multiplicative composite error term). Defaults to CET_ADDI.
- **fun** (*String, optional*) – FUN_PROD (production frontier) or FUN_COST (cost frontier). Defaults to FUN_PROD.
- **rts** (*String, optional*) – RTS_VRS (variable returns to scale) or RTS_CRS (constant returns to scale). Defaults to RTS_VRS.
- **penalty** (*int, optional*) – penalty=1 (L1 norm), penalty=2 (L2 norm), and penalty=3 (Lipschitz norm). Defaults to 1.

display_alpha()

Display alpha value

display_beta()

Display beta value

display_lamda()

Display lamda value

display_residual()

Dispaly residual value

display_status()

Display the status of problem

get_adjusted_alpha()

Return the shifted constatnt(alpha) term by CCNLS

get_adjusted_residual()

Return the shifted residuals(epsilon) tern by CCNLS

get_alpha()

Return alpha value by array

get_beta()

Return beta value by array

get_frontier()

Return estimated frontier value by array

get_lamda()

Return lamda value by array

get_predict(*x_test*)

Return the estimated function in testing sample

get_residual()

Return residual value by array

get_status()

Return status

optimize(*email='local'*, *solver=None*)

Optimize the function by requested method

Parameters

- **email** (*string*) – The email address for remote optimization. It will optimize locally if OPT_LOCAL is given.
- **solver** (*string*) – The solver chosen for optimization. It will optimize with default solver if OPT_DEFAULT is given.

4.1.14 pCQER

class pystoned.pCQER.**pCER**(*y*, *x*, *tau*, *eta*, *z=None*, *cet='addi'*, *fun='prod'*, *rts='vrs'*, *penalty=1*)

penalized Convex Expectile Regression (pCER)

__init__(*y*, *x*, *tau*, *eta*, *z=None*, *cet='addi'*, *fun='prod'*, *rts='vrs'*, *penalty=1*)

pCER model

Parameters

- **y** (*float*) – output variable.
- **x** (*float*) – input variables.
- **z** (*float, optional*) – Contextual variable(s). Defaults to None.
- **tau** (*float*) – expectile.
- **eta** (*float*) – tuning parameter.
- **cet** (*String, optional*) – CET_ADDI (additive composite error term) or CET_MULT (multiplicative composite error term). Defaults to CET_ADDI.
- **fun** (*String, optional*) – FUN_PROD (production frontier) or FUN_COST (cost frontier). Defaults to FUN_PROD.
- **rts** (*String, optional*) – RTS_VRS (variable returns to scale) or RTS_CRS (constant returns to scale). Defaults to RTS_VRS.
- **penalty** (*int, optional*) – penalty=1 (L1 norm), penalty=2 (L2 norm), and penalty=3 (Lipschitz norm). Defaults to 1.

display_alpha()

Display alpha value

display_beta()

Display beta value

display_lamda()

Display lamda value

display_negative_residual()

Dispaly negative residual value

display_positive_residual()

Dispaly positive residual value

display_residual()

Dispaly residual value

display_status()

Display the status of problem

get_alpha()

Return alpha value by array

get_beta()

Return beta value by array

get_frontier()

Return estimated frontier value by array

get_lamda()

Return beta value by array

get_negative_residual()

Return negative residual value by array

get_positive_residual()

Return positive residual value by array

get_predict(*x_test*)

Return the estimated function in testing sample

get_residual()

Return residual value by array

get_status()

Return status

optimize(*email='local'*, *solver=None*)

Optimize the function by requested method

Parameters

- **email** (*string*) – The email address for remote optimization. It will optimize locally if OPT_LOCAL is given.
- **solver** (*string*) – The solver chosen for optimization. It will optimize with default solver if OPT_DEFAULT is given.

class pystoned.pCQER.**pCQR**(*y*, *x*, *tau*, *eta*, *z=None*, *cet='addi'*, *fun='prod'*, *rts='vrs'*, *penalty=1*)

penalized convex quantile regression (pCQR)

__init__(*y*, *x*, *tau*, *eta*, *z=None*, *cet='addi'*, *fun='prod'*, *rts='vrs'*, *penalty=1*)

pCQR model

Parameters

- **y** (*float*) – output variable.
- **x** (*float*) – input variables.
- **z** (*float, optional*) – Contextual variable(s). Defaults to None.
- **tau** (*float*) – quantile.
- **eta** (*float*) – tuning parameter.
- **cet** (*String, optional*) – CET_ADDI (additive composite error term) or CET_MULT (multiplicative composite error term). Defaults to CET_ADDI.
- **fun** (*String, optional*) – FUN_PROD (production frontier) or FUN_COST (cost frontier). Defaults to FUN_PROD.
- **rts** (*String, optional*) – RTS_VRS (variable returns to scale) or RTS_CRS (constant returns to scale). Defaults to RTS_VRS.
- **penalty** (*int, optional*) – penalty=1 (L1 norm), penalty=2 (L2 norm), and penalty=3 (Lipschitz norm). Defaults to 1.

display_alpha()

Display alpha value

display_beta()

Display beta value

display_lamda()

Display lamda value

display_negative_residual()

Dispaly negative residual value

display_positive_residual()

Dispaly positive residual value

display_residual()

Dispaly residual value

display_status()

Display the status of problem

get_alpha()

Return alpha value by array

get_beta()

Return beta value by array

get_frontier()

Return estimated frontier value by array

get_lamda()

Return beta value by array

get_negative_residual()

Return negative residual value by array

get_positive_residual()

Return positive residual value by array

get_predict(*x_test*)

Return the estimated function in testing sample

get_residual()

Return residual value by array

get_status()

Return status

optimize(*email='local'*, *solver=None*)

Optimize the function by requested method

Parameters

- **email** (*string*) – The email address for remote optimization. It will optimize locally if OPT_LOCAL is given.
- **solver** (*string*) – The solver chosen for optimization. It will optimize with default solver if OPT_DEFAULT is given.

4.1.15 pICQER

class pystoned.pICQER.**pICER**(*y*, *x*, *tau*, *eta*, *z=None*, *cet='addi'*, *fun='prod'*, *rts='vrs'*, *penalty=1*)

penalized isotonic convex expectile regression (pICER)

__init__(*y*, *x*, *tau*, *eta*, *z=None*, *cet='addi'*, *fun='prod'*, *rts='vrs'*, *penalty=1*)

pICER model

Parameters

- **y** (*float*) – output variable.
- **x** (*float*) – input variables.

- **tau** (*float*) – expectile.
- **eta** (*float*) – penalty parameter.
- **z** (*float, optional*) – Contextual variable(s). Defaults to None.
- **cet** (*String, optional*) – CET_ADDI (additive composite error term) or CET_MULT (multiplicative composite error term). Defaults to CET_ADDI.
- **fun** (*String, optional*) – FUN_PROD (production frontier) or FUN_COST (cost frontier). Defaults to FUN_PROD.
- **rts** (*String, optional*) – RTS_VRS (variable returns to scale) or RTS_CRS (constant returns to scale). Defaults to RTS_VRS.
- **penalty** (*int, optional*) – penalty=1 (L1 norm), penalty=2 (L2 norm), and penalty=3 (Lipschitz norm). Defaults to 1.

display_alpha()

Display alpha value

display_beta()

Display beta value

display_lamda()

Display lamda value

display_negative_residual()

Dispaly negative residual value

display_positive_residual()

Dispaly positive residual value

display_residual()

Dispaly residual value

display_status()

Display the status of problem

get_adjusted_alpha()

Return the shifted constatnt(alpha) term by CCNLS

get_adjusted_residual()

Return the shifted residuals(epsilon) tern by CCNLS

get_alpha()

Return alpha value by array

get_beta()

Return beta value by array

get_frontier()

Return estimated frontier value by array

get_lamda()

Return lamda value by array

get_negative_residual()

Return negative residual value by array

get_positive_residual()

Return positive residual value by array

get_predict(*x_test*)

Return the estimated function in testing sample

get_residual()

Return residual value by array

get_status()

Return status

optimize(*email='local'*, *solver=None*)

Optimize the function by requested method

Parameters

- **email** (*string*) – The email address for remote optimization. It will optimize locally if OPT_LOCAL is given.
- **solver** (*string*) – The solver chosen for optimization. It will optimize with default solver if OPT_DEFAULT is given.

class pystoned.pICQER.**pICQR**(*y*, *x*, *tau*, *eta*, *z=None*, *cet='addi'*, *fun='prod'*, *rts='vrs'*, *penalty=1*)

Penalized isotonic convex quantile regression (pICQR)

__init__(*y*, *x*, *tau*, *eta*, *z=None*, *cet='addi'*, *fun='prod'*, *rts='vrs'*, *penalty=1*)

pICQR model

Parameters

- **y** (*float*) – output variable.
- **x** (*float*) – input variables.
- **tau** (*float*) – quantile.
- **eta** (*float*) – penalty parameter.
- **z** (*float, optional*) – Contextual variable(s). Defaults to None.
- **cet** (*String, optional*) – CET_ADDI (additive composite error term) or CET_MULT (multiplicative composite error term). Defaults to CET_ADDI.
- **fun** (*String, optional*) – FUN_PROD (production frontier) or FUN_COST (cost frontier). Defaults to FUN_PROD.
- **rts** (*String, optional*) – RTS_VRS (variable returns to scale) or RTS_CRS (constant returns to scale). Defaults to RTS_VRS.
- **penalty** (*int, optional*) – penalty=1 (L1 norm), penalty=2 (L2 norm), and penalty=3 (Lipschitz norm). Defaults to 1.

display_alpha()

Display alpha value

display_beta()

Display beta value

display_lamda()

Display lamda value

display_negative_residual()

Dispaly negative residual value

display_positive_residual()

Dispaly positive residual value

display_residual()

Dispaly residual value

display_status()

Display the status of problem

get_adjusted_alpha()

Return the shifted constatnt(alpha) term by CCNLS

get_adjusted_residual()

Return the shifted residuals(epsilon) tern by CCNLS

get_alpha()

Return alpha value by array

get_beta()

Return beta value by array

get_frontier()

Return estimated frontier value by array

get_lamda()

Return lamda value by array

get_negative_residual()

Return negative residual value by array

get_positive_residual()

Return positive residual value by array

get_predict(*x_test*)

Return the estimated function in testing sample

get_residual()

Return residual value by array

get_status()

Return status

optimize(*email='local'*, *solver=None*)

Optimize the function by requested method

Parameters

- **email** (*string*) – The email address for remote optimization. It will optimize locally if OPT_LOCAL is given.
- **solver** (*string*) – The solver chosen for optimization. It will optimize with default solver if OPT_DEFAULT is given.

4.1.16 pwCNLS

class pystoned.pwCNLS.**pwCNLS**(*y*, *x*, *w*, *eta*, *z=None*, *cet='addi'*, *fun='prod'*, *rts='vrs'*, *penalty=1*)

penalized Weighted Convex Nonparametric Least Square (pwCNLS)

__init__(*y*, *x*, *w*, *eta*, *z=None*, *cet='addi'*, *fun='prod'*, *rts='vrs'*, *penalty=1*)

wCNLS model

Parameters

- **y** (*float*) – output variable.
- **x** (*float*) – input variables.
- **w** (*float*) – weight variable.
- **eta** (*float*) – regularization parameter.
- **z** (*float, optional*) – Contextual variable(s). Defaults to None.
- **cet** (*String, optional*) – CET_ADDI (additive composite error term) or CET_MULT (multiplicative composite error term). Defaults to CET_ADDI.
- **fun** (*String, optional*) – FUN_PROD (production frontier) or FUN_COST (cost frontier). Defaults to FUN_PROD.
- **rts** (*String, optional*) – RTS_VRS (variable returns to scale) or RTS_CRS (constant returns to scale). Defaults to RTS_VRS.
- **penalty** (*int, optional*) – penalty=1 (L1 norm), penalty=2 (L2 norm), and penalty=3 (Lipschitz norm). Defaults to 1.

display_alpha()

Display alpha value

display_beta()

Display beta value

display_lamda()

Display lamda value

display_residual()

Dispaly residual value

display_status()

Display the status of problem

get_adjusted_alpha()

Return the shifted constatnt(alpha) term by CCNLS

get_adjusted_residual()

Return the shifted residuals(epsilon) tern by CCNLS

get_alpha()

Return alpha value by array

get_beta()

Return beta value by array

get_frontier()

Return estimated frontier value by array

get_lamda()

Return lamda value by array

get_predict(*x_test*)

Return the estimated function in testing sample

get_residual()

Return residual value by array

get_status()

Return status

optimize(*email='local'*, *solver=None*)

Optimize the function by requested method

Parameters

- **email** (*string*) – The email address for remote optimization. It will optimize locally if OPT_LOCAL is given.
- **solver** (*string*) – The solver chosen for optimization. It will optimize with default solver if OPT_DEFAULT is given.

4.1.17 pwCQER

class pystoned.pwCQER.**pwCER**(*y*, *x*, *w*, *tau*, *eta*, *z=None*, *cet='addi'*, *fun='prod'*, *rts='vrs'*, *penalty=1*)

penalized Weighted Convex Expectile Regression (pwCER)

__init__(*y*, *x*, *w*, *tau*, *eta*, *z=None*, *cet='addi'*, *fun='prod'*, *rts='vrs'*, *penalty=1*)

pwCER model

Parameters

- **y** (*float*) – output variable.
- **x** (*float*) – input variables.
- **w** (*float*) – weight variable.
- **eta** (*float*) – tuning parameter.
- **z** (*float, optional*) – Contextual variable(s). Defaults to None.
- **tau** (*float*) – expectile.
- **cet** (*String, optional*) – CET_ADDI (additive composite error term) or CET_MULT (multiplicative composite error term). Defaults to CET_ADDI.
- **fun** (*String, optional*) – FUN_PROD (production frontier) or FUN_COST (cost frontier). Defaults to FUN_PROD.
- **rts** (*String, optional*) – RTS_VRS (variable returns to scale) or RTS_CRS (constant returns to scale). Defaults to RTS_VRS.
- **penalty** (*int, optional*) – penalty=1 (L1 norm), penalty=2 (L2 norm), and penalty=3 (Lipschitz norm). Defaults to 1.

display_alpha()

Display alpha value

display_beta()

Display beta value

display_lamda()
Display lamda value

display_negative_residual()
Dispaly negative residual value

display_positive_residual()
Dispaly positive residual value

display_residual()
Dispaly residual value

display_status()
Display the status of problem

get_alpha()
Return alpha value by array

get_beta()
Return beta value by array

get_frontier()
Return estimated frontier value by array

get_lamda()
Return beta value by array

get_negative_residual()
Return negative residual value by array

get_positive_residual()
Return positive residual value by array

get_predict(*x_test*)
Return the estimated function in testing sample

get_residual()
Return residual value by array

get_status()
Return status

optimize(*email='local'*, *solver=None*)
Optimize the function by requested method

Parameters

- **email** (*string*) – The email address for remote optimization. It will optimize locally if OPT_LOCAL is given.
- **solver** (*string*) – The solver chosen for optimization. It will optimize with default solver if OPT_DEFAULT is given.

class pystoned.pwCQER.**pwCQR**(*y*, *x*, *w*, *tau*, *eta*, *z=None*, *cet='addi'*, *fun='prod'*, *rts='vrs'*, *penalty=1*)
penalized Weighted Convex Quantile Regression (pwCQR)

__init__(*y*, *x*, *w*, *tau*, *eta*, *z=None*, *cet='addi'*, *fun='prod'*, *rts='vrs'*, *penalty=1*)
pwCQR model

Parameters

- **y** (*float*) – output variable.
- **x** (*float*) – input variables.
- **w** (*float*) – weight variable.
- **eta** (*float*) – tuning parameter.
- **z** (*float, optional*) – Contextual variable(s). Defaults to None.
- **tau** (*float*) – quantile.
- **cet** (*String, optional*) – CET_ADDI (additive composite error term) or CET_MULT (multiplicative composite error term). Defaults to CET_ADDI.
- **fun** (*String, optional*) – FUN_PROD (production frontier) or FUN_COST (cost frontier). Defaults to FUN_PROD.
- **rts** (*String, optional*) – RTS_VRS (variable returns to scale) or RTS_CRS (constant returns to scale). Defaults to RTS_VRS.
- **penalty** (*int, optional*) – penalty=1 (L1 norm), penalty=2 (L2 norm), and penalty=3 (Lipschitz norm). Defaults to 1.

display_alpha()

Display alpha value

display_beta()

Display beta value

display_lamda()

Display lamda value

display_negative_residual()

Dispaly negative residual value

display_positive_residual()

Dispaly positive residual value

display_residual()

Dispaly residual value

display_status()

Display the status of problem

get_alpha()

Return alpha value by array

get_beta()

Return beta value by array

get_frontier()

Return estimated frontier value by array

get_lamda()

Return beta value by array

get_negative_residual()

Return negative residual value by array

get_positive_residual()

Return positive residual value by array

get_predict(*x_test*)

Return the estimated function in testing sample

get_residual()

Return residual value by array

get_status()

Return status

optimize(*email='local'*, *solver=None*)

Optimize the function by requested method

Parameters

- **email** (*string*) – The email address for remote optimization. It will optimize locally if OPT_LOCAL is given.
- **solver** (*string*) – The solver chosen for optimization. It will optimize with default solver if OPT_DEFAULT is given.

4.1.18 sCQER

class pystoned.sCQER.**sCER**(*y*, *x*, *tau*, *C*)

Simultaneous estimation of CER

__init__(*y*, *x*, *tau*, *C*)

sCER model

Parameters

- **y** (*float*) – output variable.
- **x** (*float*) – input variables.
- **tau** – vector of expectile.
- **C** – interval (small positive value)

get_alpha()

Return alpha value by array

get_beta()

Return beta value by array

get_frontier()

Return estimated frontier value by array

get_negative_residual()

Return negative residual value by array

get_positive_residual()

Return positive residual value by array

optimize(*email='local'*, *solver=None*)

Optimize the function by requested method

Parameters

- **email** (*string*) – The email address for remote optimization. It will optimize locally if OPT_LOCAL is given.
- **solver** (*string*) – The solver chosen for optimization. It will optimize with default solver if OPT_DEFAULT is given.

class pystoned.sCQER.**sCQR**(*y*, *x*, *tau*, *C*)

Simultaneous estimation of CQR

__init__(*y*, *x*, *tau*, *C*)

sCQR model

Parameters

- **y** (*float*) – output variable.
- **x** (*float*) – input variables.
- **tau** – vector of quantile.
- **C** – interval (small positive value)

get_alpha()

Return alpha value by array

get_beta()

Return beta value by array

get_frontier()

Return estimated frontier value by array

get_negative_residual()

Return negative residual value by array

get_positive_residual()

Return positive residual value by array

optimize(*email='local'*, *solver=None*)

Optimize the function by requested method

Parameters

- **email** (*string*) – The email address for remote optimization. It will optimize locally if OPT_LOCAL is given.
- **solver** (*string*) – The solver chosen for optimization. It will optimize with default solver if OPT_DEFAULT is given.

4.1.19 wCNLS

class pystoned.wCNLS.**wCNLS**(*y*, *x*, *w*, *z=None*, *cet='addi'*, *fun='prod'*, *rts='vrs'*)

Weighted Convex Nonparametric Least Square (wCNLS)

__init__(*y*, *x*, *w*, *z=None*, *cet='addi'*, *fun='prod'*, *rts='vrs'*)

wCNLS model

Parameters

- **y** (*float*) – output variable.
- **x** (*float*) – input variables.
- **w** (*float*) – weight variable.

- **z** (*float, optional*) – Contextual variable(s). Defaults to None.
- **cet** (*String, optional*) – CET_ADDI (additive composite error term) or CET_MULT (multiplicative composite error term). Defaults to CET_ADDI.
- **fun** (*String, optional*) – FUN_PROD (production frontier) or FUN_COST (cost frontier). Defaults to FUN_PROD.
- **rts** (*String, optional*) – RTS_VRS (variable returns to scale) or RTS_CRS (constant returns to scale). Defaults to RTS_VRS.

display_alpha()

Display alpha value

display_beta()

Display beta value

display_lamda()

Display lamda value

display_residual()

Dispaly residual value

display_status()

Display the status of problem

get_adjusted_alpha()

Return the shifted constatnt(alpha) term by CCNLS

get_adjusted_residual()

Return the shifted residuals(epsilon) tern by CCNLS

get_alpha()

Return alpha value by array

get_beta()

Return beta value by array

get_frontier()

Return estimated frontier value by array

get_lamda()

Return lamda value by array

get_predict(*x_test*)

Return the estimated function in testing sample

get_residual()

Return residual value by array

get_status()

Return status

optimize(*email='local'*, *solver=None*)

Optimize the function by requested method

Parameters

- **email** (*string*) – The email address for remote optimization. It will optimize locally if OPT_LOCAL is given.

- **solver** (*string*) – The solver chosen for optimization. It will optimize with default solver if OPT_DEFAULT is given.

4.1.20 wCQER

class pystoned.wCQER.**wCER**(*y*, *x*, *w*, *tau*, *z=None*, *cet='addi'*, *fun='prod'*, *rts='vrs'*)

Weighted Convex Expectile Regression (wCER)

__init__(*y*, *x*, *w*, *tau*, *z=None*, *cet='addi'*, *fun='prod'*, *rts='vrs'*)

wCER model

Parameters

- **y** (*float*) – output variable.
- **x** (*float*) – input variables.
- **w** (*float*) – weight variable.
- **z** (*float, optional*) – Contextual variable(s). Defaults to None.
- **tau** (*float*) – expectile.
- **cet** (*String, optional*) – CET_ADDI (additive composite error term) or CET_MULT (multiplicative composite error term). Defaults to CET_ADDI.
- **fun** (*String, optional*) – FUN_PROD (production frontier) or FUN_COST (cost frontier). Defaults to FUN_PROD.
- **rts** (*String, optional*) – RTS_VRS (variable returns to scale) or RTS_CRS (constant returns to scale). Defaults to RTS_VRS.

display_alpha()

Display alpha value

display_beta()

Display beta value

display_lamda()

Display lamda value

display_negative_residual()

Dispaly negative residual value

display_positive_residual()

Dispaly positive residual value

display_residual()

Dispaly residual value

display_status()

Display the status of problem

get_alpha()

Return alpha value by array

get_beta()

Return beta value by array

get_frontier()

Return estimated frontier value by array

get_lamda()

Return beta value by array

get_negative_residual()

Return negative residual value by array

get_positive_residual()

Return positive residual value by array

get_predict(*x_test*)

Return the estimated function in testing sample

get_residual()

Return residual value by array

get_status()

Return status

optimize(*email='local'*, *solver=None*)

Optimize the function by requested method

Parameters

- **email** (*string*) – The email address for remote optimization. It will optimize locally if OPT_LOCAL is given.
- **solver** (*string*) – The solver chosen for optimization. It will optimize with default solver if OPT_DEFAULT is given.

class pystoned.wCQER.**wCQR**(*y*, *x*, *w*, *tau*, *z=None*, *cet='addi'*, *fun='prod'*, *rts='vrs'*)

Weighted Convex Quantile Regression (wCQR)

__init__(*y*, *x*, *w*, *tau*, *z=None*, *cet='addi'*, *fun='prod'*, *rts='vrs'*)

wCQR model

Parameters

- **y** (*float*) – output variable.
- **x** (*float*) – input variables.
- **w** (*float*) – weight variable.
- **z** (*float, optional*) – Contextual variable(s). Defaults to None.
- **tau** (*float*) – quantile.
- **cet** (*String, optional*) – CET_ADDI (additive composite error term) or CET_MULT (multiplicative composite error term). Defaults to CET_ADDI.
- **fun** (*String, optional*) – FUN_PROD (production frontier) or FUN_COST (cost frontier). Defaults to FUN_PROD.
- **rts** (*String, optional*) – RTS_VRS (variable returns to scale) or RTS_CRS (constant returns to scale). Defaults to RTS_VRS.

display_alpha()

Display alpha value

display_beta()

Display beta value

display_lamda()

Display lamda value

display_negative_residual()

Dispaly negative residual value

display_positive_residual()

Dispaly positive residual value

display_residual()

Dispaly residual value

display_status()

Display the status of problem

get_alpha()

Return alpha value by array

get_beta()

Return beta value by array

get_frontier()

Return estimated frontier value by array

get_lamda()

Return beta value by array

get_negative_residual()

Return negative residual value by array

get_positive_residual()

Return positive residual value by array

get_predict(*x_test*)

Return the estimated function in testing sample

get_residual()

Return residual value by array

get_status()

Return status

optimize(*email='local'*, *solver=None*)

Optimize the function by requested method

Parameters

- **email** (*string*) – The email address for remote optimization. It will optimize locally if OPT_LOCAL is given.
- **solver** (*string*) – The solver chosen for optimization. It will optimize with default solver if OPT_DEFAULT is given.

4.1.21 weakCNLS

class pystoned.weakCNLS.**weakCNLS**(*y*, *x*, *b*, *z=None*, *cet='addi'*, *fun='prod'*, *rts='vrs'*)

Convex Nonparametric Least Square with weak disposability (weakCNLS)

__init__(*y*, *x*, *b*, *z=None*, *cet='addi'*, *fun='prod'*, *rts='vrs'*)

weakCNLS model

Parameters

- **y** (*float*) – output variable.
- **x** (*float*) – input variables.
- **b** (*float*) – undersiable variables.
- **z** (*float, optional*) – Contextual variable(s). Defaults to None.
- **cet** (*String, optional*) – CET_ADDI (additive composite error term) or CET_MULT (multiplicative composite error term). Defaults to CET_ADDI.
- **fun** (*String, optional*) – FUN_PROD (production frontier) or FUN_COST (cost frontier). Defaults to FUN_PROD.
- **rts** (*String, optional*) – RTS_VRS (variable returns to scale) or RTS_CRS (constant returns to scale). Defaults to RTS_VRS.

display_alpha()

Display alpha value

display_beta()

Display beta value

display_delta()

Display delta value

display_lamda()

Display lamda value

display_residual()

Dispaly residual value

display_status()

Display the status of problem

get_adjusted_alpha()

Return the shifted constatnt(alpha) term by CCNLS

get_adjusted_residual()

Return the shifted residuals(epsilon) tern by CCNLS

get_alpha()

Return alpha value by array

get_beta()

Return beta value by array

get_delta()

Return delta value by array

get_frontier()

Return estimated frontier value by array

get_lamda()

Return lamda value by array

get_predict(*x_test*)

Return the estimated function in testing sample

get_residual()

Return residual value by array

get_status()

Return status

optimize(*email='local'*, *solver=None*)

Optimize the function by requested method

Parameters

- **email** (*string*) – The email address for remote optimization. It will optimize locally if OPT_LOCAL is given.
- **solver** (*string*) – The solver chosen for optimization. It will optimize with default solver if OPT_DEFAULT is given.

4.2 Residual Decomposition

4.2.1 StoNED

class pystoned.StoNED.**StoNED**(*model*)

Stochastic nonparametric envelopment of data (StoNED)

__init__(*model*)

StoNED model: The input model for residual decomposition

get_stoned(*method='MOM'*)

Parameters

method (*String, optional*) – RED_MOM (Method of moments) or RED_QLE (Quassi-likelihood estimation). Defaults to RED_MOM.

Calculate the StoNED frontier

get_technical_inefficiency(*method='MOM'*)

Parameters

method (*String, optional*) – RED_MOM (Method of moments) or RED_QLE (Quassi-likelihood estimation). Defaults to RED_MOM.

calculate sigma_u, sigma_v, mu, and epsilon value

get_unconditional_expected_inefficiency(*method='MOM'*)

Parameters

method (*String, optional*) – RED_MOM (Method of moments) or RED_QLE (Quassi-likelihood estimation) or RED_KDE (Kernel deconvolution estimation). Defaults to RED_MOM.

4.3 Peripheral Classes

4.3.1 plot

pystoned.plot.**plot2d**(*model*, *x_select=0*, *label_name='estimated function'*, *fig_name=None*, *method='MOM'*)

Plot 2d estimated function/frontier

Parameters

- **model** – The input model for plotting.
- **x_select** (*Integer*) – The selected x for plotting.
- **label_name** (*String*) – the estimator name.
- **fig_name** (*String, optional*) – The name of figure to save. Defaults to None.

pystoned.plot.**plot3d**(*model*, *x_select_1=0*, *x_select_2=1*, *fig_name=None*, *line_transparent=False*, *pane_transparent=False*)

Plot 3d estimated function/frontier

Parameters

- **model** – The input model for plotting.
- **x_select_1** (*Integer*) – The selected x for plotting.
- **x_select_2** (*Integer*) – The selected x for plotting.
- **fun** (*String, optional*) – FUN_PROD (production frontier) or FUN_COST (cost frontier). Defaults to FUN_PROD.
- **fig_name** (*String, optional*) – The name of figure to save. Defaults to None.
- **line_transparent** (*bool, optional*) – control the transparency of the lines. Defaults to False.
- **pane_transparent** (*bool, optional*) – control the transparency of the pane. Defaults to False.

4.3.2 dataset

pystoned.dataset.**load_Finnish_electricity_firm**(*x_select=['Energy', 'Length', 'Customers']*, *y_select=['OPEX', 'CAPEX', 'TOTEX']*, *z_select=['PerUndGr']*)

Loading Finnish electricity firm data

Parameters

- **x_select** (*list, optional*) – input variables. Defaults to ['Energy', 'Length', 'Customers'].
- **y_select** (*list, optional*) – output variable. Defaults to ['OPEX', 'CAPEX', 'TOTEX'].
- **z_select** (*list, optional*) – contextual variable. Defaults to ['PerUndGr'].

Returns
selected input-output

Return type
Numbers

pystoned.dataset.**load_GHG_abatement_cost**(*year=None*, *x_select=['HRSN', 'CPNK']*, *y_select=['VALK']*, *b_select=['GHG']*)

Loading OECD GHG emissions data

Parameters

- **year** (*Numbers, optional*) – years. Defaults to None.
- **x_select** (*list, optional*) – input variables. Defaults to ['HRSN', 'CPNK'].
- **y_select** (*list, optional*) – output variable. Defaults to ['VALK'].
- **b_select** (*list, optional*) – bad output variable. Defaults to ['GHG'].

Returns
selected input-output

Return type
Numbers

pystoned.dataset.**load_Philipines_rice_production**(*year=None*, *x_select=['AREA', 'LABOR', 'NPK', 'OTHER', 'AREAP', 'LABORP', 'NPKP', 'OTHERP']*, *y_select=['PROD', 'PRICE']*)

Loading Philipines rice data

Parameters

- **year** (*Numbers, optional*) – years. Defaults to None.
- **x_select** (*list, optional*) – input variables. Defaults to ['AREA', 'LABOR', 'NPK', 'OTHER', 'AREAP', 'LABORP', 'NPKP', 'OTHERP'].
- **y_select** (*list, optional*) – output variable. Defaults to ['PROD', 'PRICE'].

Returns
selected input-output

Return type
Numbers

pystoned.dataset.**load_Tim_Coelli_frontier**(*x_select=['capital', 'labour']*, *y_select=['output']*)

Loading Tim Coelli 4.1 data

Parameters

- **x_select** (*list, optional*) – input variables. Defaults to ['capital', 'labour'].
- **y_select** (*list, optional*) – output variable. Defaults to ['output'].

Returns
selected input-output

Return type
Numbers

class pystoned.dataset.**production_data**(*dmu*, *x*, *y*, *b=None*, *z=None*)

Example datasets provided by the pyStoNED

__init__(*dmu*, *x*, *y*, *b=None*, *z=None*)

General data structure

Parameters

- **dmu** (*String*) – decision making unit.
- **x** (*Numbers*) – input variables.
- **y** (*Numbers*) – output variables.
- **b** (*Numbers, optional*) – bad output variables. Defaults to None.
- **z** (*Numbers, optional*) – contextual variables. Defaults to None.

4.3.3 constant

pystoned.constant.**CET_ADDI = 'addi'**

Additive composite error term.

Type
CET_ADDI

pystoned.constant.**CET_MULT = 'mult'**

Multiplicative composite error term.

Type
CET_MULT

pystoned.constant.**FUN_COST = 'cost'**

Cost frontier.

Type
FUN_COST

pystoned.constant.**FUN_PROD = 'prod'**

Production frontier.

Type
FUN_PROD

pystoned.constant.**ORIENT_IO = 'io'**

Input orientation.

Type
ORIENT_IO

pystoned.constant.**ORIENT_OO = 'oo'**

Output orientation.

Type
ORIENT_OO

pystoned.constant.**RDF_DI = 'DI'**

Input distance function.

Type
RDF_DI

pystoned.constant.**RDF_DO = 'DO'**

Output distance function.

Type
RDF_DO

pystoned.constant.**RED_KDE = 'KDE'**

Kernel deconvolution estimation.

Type
RED_KDE

pystoned.constant.**RED_MOM = 'MOM'**

Method of moments.

Type
RED_MOM

pystoned.constant.**RED_QLE = 'QLE'**

Quassi-likelihood estimation.

Type
RED_QLE

pystoned.constant.**RTS_CRS = 'crs'**

Constant returns to scale.

Type
RTS_CRS

pystoned.constant.**RTS_VRS = 'vrs'**

Variable returns to scale.

Type
RTS_VRS

4.3.4 Inner Functions

Genetic algorithms related models

CNLSG1

class pystoned.utils.CNLSG1.**CNLSG1**(*y*, *x*, *cutactive*, *cet='addi'*, *fun='prod'*, *rts='vrs'*)

initial Group-VC-added CNLS (CNLS+G) model

__init__(*y*, *x*, *cutactive*, *cet='addi'*, *fun='prod'*, *rts='vrs'*)

CNLS+G model 1

Parameters

- **y** (*float*) – output variable.
- **x** (*float*) – input variables.
- **cutactive** (*float*) – active concavity constraint.
- **cet** (*String, optional*) – CET_ADDI (additive composite error term) or CET_MULT (multiplicative composite error term). Defaults to CET_ADDI.
- **fun** (*String, optional*) – FUN_PROD (production frontier) or FUN_COST (cost frontier). Defaults to FUN_PROD.
- **rts** (*String, optional*) – RTS_VRS (variable returns to scale) or RTS_CRS (constant returns to scale). Defaults to RTS_VRS.

get_alpha()

Return alpha value by array

get_beta()

Return beta value by array

optimize(*email='local'*, *solver=None*)

Optimize the function by requested method

Parameters

- **email** (*string*) – The email address for remote optimization. It will optimize locally if OPT_LOCAL is given.
- **solver** (*string*) – The solver chosen for optimization. It will optimize with default solver if OPT_DEFAULT is given.

CNLSG2

class pystoned.utils.CNLSG2.**CNLSG2**(*y*, *x*, *cutactive*, *active*, *cet='addi'*, *fun='prod'*, *rts='vrs'*)

CNLS+G in iterative loop

__init__(*y*, *x*, *cutactive*, *active*, *cet='addi'*, *fun='prod'*, *rts='vrs'*)

CNLS+G model

Parameters

- **y** (*float*) – output variable.
- **x** (*float*) – input variables.
- **cutactive** (*float*) – active concavity constraint.
- **active** (*float*) – violated concavity constraint.
- **cet** (*String, optional*) – CET_ADDI (additive composite error term) or CET_MULT (multiplicative composite error term). Defaults to CET_ADDI.
- **fun** (*String, optional*) – FUN_PROD (production frontier) or FUN_COST (cost frontier). Defaults to FUN_PROD.
- **rts** (*String, optional*) – RTS_VRS (variable returns to scale) or RTS_CRS (constant returns to scale). Defaults to RTS_VRS.

get_alpha()

Return alpha value by array

get_beta()

Return beta value by array

optimize(*email='local'*, *solver=None*)

Optimize the function by requested method

Parameters

- **email** (*string*) – The email address for remote optimization. It will optimize locally if OPT_LOCAL is given.
- **solver** (*string*) – The solver chosen for optimization. It will optimize with default solver if OPT_DEFAULT is given.

CNLSZG1

class pystoned.utils.CNLSZG1.**CNLSZG1**(*y*, *x*, *z*, *cutactive*, *cet='addi'*, *fun='prod'*, *rts='vrs'*)

initial Group-VC-added CNLSZ (CNLSZ+G) model

__init__(*y*, *x*, *z*, *cutactive*, *cet='addi'*, *fun='prod'*, *rts='vrs'*)

CNLSZ+G model

Parameters

- **y** (*float*) – output variable.
- **x** (*float*) – input variables.
- **z** (*float, optional*) – Contextual variable(s). Defaults to None.
- **cutactive** (*float*) – active concavity constraint.
- **cet** (*String, optional*) – CET_ADDI (additive composite error term) or CET_MULT (multiplicative composite error term). Defaults to CET_ADDI.

- **fun** (*String, optional*) – FUN_PROD (production frontier) or FUN_COST (cost frontier). Defaults to FUN_PROD.
- **rts** (*String, optional*) – RTS_VRS (variable returns to scale) or RTS_CRS (constant returns to scale). Defaults to RTS_VRS.

get_alpha()

Return alpha value by array

get_beta()

Return beta value by array

optimize(*email='local'*, *solver=None*)

Optimize the function by requested method

Parameters

- **email** (*string*) – The email address for remote optimization. It will optimize locally if OPT_LOCAL is given.
- **solver** (*string*) – The solver chosen for optimization. It will optimize with default solver if OPT_DEFAULT is given.

CNLSZG2

class pystoned.utils.CNLSZG2.**CNLSZG2**(*y*, *x*, *z*, *cutactive*, *active*, *cet='addi'*, *fun='prod'*, *rts='vrs'*)

CNLSZ+G in iterative loop

__init__(*y*, *x*, *z*, *cutactive*, *active*, *cet='addi'*, *fun='prod'*, *rts='vrs'*)

CNLSZ+G model

Parameters

- **y** (*float*) – output variable.
- **x** (*float*) – input variables.
- **z** (*float, optional*) – Contextual variable(s). Defaults to None.
- **cutactive** (*float*) – active concavity constraint.
- **active** (*float*) – violated concavity constraint.
- **cet** (*String, optional*) – CET_ADDI (additive composite error term) or CET_MULT (multiplicative composite error term). Defaults to CET_ADDI.
- **fun** (*String, optional*) – FUN_PROD (production frontier) or FUN_COST (cost frontier). Defaults to FUN_PROD.
- **rts** (*String, optional*) – RTS_VRS (variable returns to scale) or RTS_CRS (constant returns to scale). Defaults to RTS_VRS.

get_alpha()

Return alpha value by array

get_beta()

Return beta value by array

optimize(*email='local'*, *solver=None*)

Optimize the function by requested method

Parameters

- **email** (*string*) – The email address for remote optimization. It will optimize locally if OPT_LOCAL is given.
- **solver** (*string*) – The solver chosen for optimization. It will optimize with default solver if OPT_DEFAULT is given.

CQERG1

class pystoned.utils.CQERG1.**CERG1**(*y*, *x*, *tau*, *cutactive*, *cet='addi'*, *fun='prod'*, *rts='vrs'*)

initial Group-VC-added CER (CER+G) model

__init__(*y*, *x*, *tau*, *cutactive*, *cet='addi'*, *fun='prod'*, *rts='vrs'*)

CER+G model

Parameters

- **y** (*float*) – output variable.
- **x** (*float*) – input variables.
- **tau** (*float*) – expectile.
- **cutactive** (*float*) – active concavity constraint.
- **cet** (*String, optional*) – CET_ADDI (additive composite error term) or CET_MULT (multiplicative composite error term). Defaults to CET_ADDI.
- **fun** (*String, optional*) – FUN_PROD (production frontier) or FUN_COST (cost frontier). Defaults to FUN_PROD.
- **rts** (*String, optional*) – RTS_VRS (variable returns to scale) or RTS_CRS (constant returns to scale). Defaults to RTS_VRS.

class pystoned.utils.CQERG1.**CQRG1**(*y*, *x*, *tau*, *cutactive*, *cet='addi'*, *fun='prod'*, *rts='vrs'*)

initial Group-VC-added CQR (CQR+G) model

__init__(*y*, *x*, *tau*, *cutactive*, *cet='addi'*, *fun='prod'*, *rts='vrs'*)

CQR+G model

Parameters

- **y** (*float*) – output variable.
- **x** (*float*) – input variables.
- **tau** (*float*) – quantile.
- **cutactive** (*float*) – active concavity constraint.
- **cet** (*String, optional*) – CET_ADDI (additive composite error term) or CET_MULT (multiplicative composite error term). Defaults to CET_ADDI.
- **fun** (*String, optional*) – FUN_PROD (production frontier) or FUN_COST (cost frontier). Defaults to FUN_PROD.
- **rts** (*String, optional*) – RTS_VRS (variable returns to scale) or RTS_CRS (constant returns to scale). Defaults to RTS_VRS.

get_alpha()

Return alpha value by array

get_beta()

Return beta value by array

optimize(*email='local'*, *solver=None*)

Optimize the function by requested method

Parameters

- **email** (*string*) – The email address for remote optimization. It will optimize locally if OPT_LOCAL is given.
- **solver** (*string*) – The solver chosen for optimization. It will optimize with default solver if OPT_DEFAULT is given.

CQERG2

class pystoned.utils.CQERG2.**CERG2**(*y*, *x*, *tau*, *cutactive*, *active*, *cet='addi'*, *fun='prod'*, *rts='vrs'*)

CER+G in iterative loop

__init__(*y*, *x*, *tau*, *cutactive*, *active*, *cet='addi'*, *fun='prod'*, *rts='vrs'*)

CER+G model

Parameters

- **y** (*float*) – output variable.
- **x** (*float*) – input variables.
- **tau** (*float*) – expectile.
- **cutactive** (*float*) – active concavity constraint.
- **active** (*float*) – violated concavity constraint.
- **cet** (*String, optional*) – CET_ADDI (additive composite error term) or CET_MULT (multiplicative composite error term). Defaults to CET_ADDI.
- **fun** (*String, optional*) – FUN_PROD (production frontier) or FUN_COST (cost frontier). Defaults to FUN_PROD.
- **rts** (*String, optional*) – RTS_VRS (variable returns to scale) or RTS_CRS (constant returns to scale). Defaults to RTS_VRS.

class pystoned.utils.CQERG2.**CQRG2**(*y*, *x*, *tau*, *cutactive*, *active*, *cet='addi'*, *fun='prod'*, *rts='vrs'*)

CQR+G in iterative loop

__init__(*y*, *x*, *tau*, *cutactive*, *active*, *cet='addi'*, *fun='prod'*, *rts='vrs'*)

CQR+G model

Parameters

- **y** (*float*) – output variable.
- **x** (*float*) – input variables.
- **tau** (*float*) – quantile.
- **cutactive** (*float*) – active concavity constraint.
- **active** (*float*) – violated concavity constraint.
- **cet** (*String, optional*) – CET_ADDI (additive composite error term) or CET_MULT (multiplicative composite error term). Defaults to CET_ADDI.
- **fun** (*String, optional*) – FUN_PROD (production frontier) or FUN_COST (cost frontier). Defaults to FUN_PROD.

- **rts** (*String, optional*) – RTS_VRS (variable returns to scale) or RTS_CRS (constant returns to scale). Defaults to RTS_VRS.

get_alpha()

Return alpha value by array

get_beta()

Return beta value by array

optimize(*email='local'*, *solver=None*)

Optimize the function by requested method

Parameters

- **email** (*string*) – The email address for remote optimization. It will optimize locally if OPT_LOCAL is given.
- **solver** (*string*) – The solver chosen for optimization. It will optimize with default solver if OPT_DEFAULT is given.

CQERZG1

class pystoned.utils.CQERZG1.**CERZG1**(*y*, *x*, *z*, *tau*, *cutactive*, *cet='addi'*, *fun='prod'*, *rts='vrs'*)

initial Group-VC-added CERZ (CERZ+G) model

__init__(*y*, *x*, *z*, *tau*, *cutactive*, *cet='addi'*, *fun='prod'*, *rts='vrs'*)

CERZ+G model

Parameters

- **y** (*float*) – output variable.
- **x** (*float*) – input variables.
- **z** (*float, optional*) – Contextual variable(s). Defaults to None.
- **tau** (*float*) – expectile.
- **cutactive** (*float*) – active concavity constraint.
- **cet** (*String, optional*) – CET_ADDI (additive composite error term) or CET_MULT (multiplicative composite error term). Defaults to CET_ADDI.
- **fun** (*String, optional*) – FUN_PROD (production frontier) or FUN_COST (cost frontier). Defaults to FUN_PROD.
- **rts** (*String, optional*) – RTS_VRS (variable returns to scale) or RTS_CRS (constant returns to scale). Defaults to RTS_VRS.

class pystoned.utils.CQERZG1.**CQRZG1**(*y*, *x*, *z*, *tau*, *cutactive*, *cet='addi'*, *fun='prod'*, *rts='vrs'*)

initial Group-VC-added CQRZ (CQRZ+G) model

__init__(*y*, *x*, *z*, *tau*, *cutactive*, *cet='addi'*, *fun='prod'*, *rts='vrs'*)

CQRZ+G model

Parameters

- **y** (*float*) – output variable.
- **x** (*float*) – input variables.
- **z** (*float, optional*) – Contextual variable(s). Defaults to None.
- **tau** (*float*) – quantile.

- **cutactive** (*float*) – active concavity constraint.
- **cet** (*String, optional*) – CET_ADDI (additive composite error term) or CET_MULT (multiplicative composite error term). Defaults to CET_ADDI.
- **fun** (*String, optional*) – FUN_PROD (production frontier) or FUN_COST (cost frontier). Defaults to FUN_PROD.
- **rts** (*String, optional*) – RTS_VRS (variable returns to scale) or RTS_CRS (constant returns to scale). Defaults to RTS_VRS.

get_alpha()

Return alpha value by array

get_beta()

Return beta value by array

optimize(*email='local'*, *solver=None*)

Optimize the function by requested method

Parameters

- **email** (*string*) – The email address for remote optimization. It will optimize locally if OPT_LOCAL is given.
- **solver** (*string*) – The solver chosen for optimization. It will optimize with default solver if OPT_DEFAULT is given.

CQERZG2

class pystoned.utils.CQERZG2.**CERZG2**(*y*, *x*, *z*, *tau*, *cutactive*, *active*, *cet='addi'*, *fun='prod'*, *rts='vrs'*)

CERZ+G in iterative loop

__init__(*y*, *x*, *z*, *tau*, *cutactive*, *active*, *cet='addi'*, *fun='prod'*, *rts='vrs'*)

CERZ+G model

Parameters

- **y** (*float*) – output variable.
- **x** (*float*) – input variables.
- **z** (*float, optional*) – Contextual variable(s). Defaults to None.
- **tau** (*float*) – expectile.
- **cutactive** (*float*) – active concavity constraint.
- **active** (*float*) – violated concavity constraint.
- **cet** (*String, optional*) – CET_ADDI (additive composite error term) or CET_MULT (multiplicative composite error term). Defaults to CET_ADDI.
- **fun** (*String, optional*) – FUN_PROD (production frontier) or FUN_COST (cost frontier). Defaults to FUN_PROD.
- **rts** (*String, optional*) – RTS_VRS (variable returns to scale) or RTS_CRS (constant returns to scale). Defaults to RTS_VRS.

class pystoned.utils.CQERZG2.**CQRZG2**(*y*, *x*, *z*, *tau*, *cutactive*, *active*, *cet='addi'*, *fun='prod'*, *rts='vrs'*)

CQRZ+G in iterative loop

__init__(*y*, *x*, *z*, *tau*, *cutactive*, *active*, *cet='addi'*, *fun='prod'*, *rts='vrs'*)

CQRZ+G model

Parameters

- **y** (*float*) – output variable.
- **x** (*float*) – input variables.
- **z** (*float, optional*) – Contextual variable(s). Defaults to None.
- **tau** (*float*) – quantile.
- **cutactive** (*float*) – active concavity constraint.
- **active** (*float*) – violated concavity constraint.
- **cet** (*String, optional*) – CET_ADDI (additive composite error term) or CET_MULT (multiplicative composite error term). Defaults to CET_ADDI.
- **fun** (*String, optional*) – FUN_PROD (production frontier) or FUN_COST (cost frontier). Defaults to FUN_PROD.
- **rts** (*String, optional*) – RTS_VRS (variable returns to scale) or RTS_CRS (constant returns to scale). Defaults to RTS_VRS.

get_alpha()

Return alpha value by array

get_beta()

Return beta value by array

optimize(*email='local'*, *solver=None*)

Optimize the function by requested method

Parameters

- **email** (*string*) – The email address for remote optimization. It will optimize locally if OPT_LOCAL is given.
- **solver** (*string*) – The solver chosen for optimization. It will optimize with default solver if OPT_DEFAULT is given.

Sweet spot function/Interpolation

interpolation

pystoned.utils.interpolation.**interpolation**(*alpha*, *beta*, *x*, *fun='prod'*)

Interpolate estimated function/frontier

Parameters

- **alpha** (*float*) – estimated alpha.
- **beta** (*float*) – estimated beta.
- **x** (*float*) – input variables.
- **fun** (*String, optional*) – FUN_PROD (production frontier) or FUN_COST (cost frontier). Defaults to FUN_PROD.

Returns

interpolated frontier

Return type
yat

sweet

pystoned.utils.sweet.**sweet**(*x*)

Sweet spot approach

Parameters
x (*float*) – input variables.

Returns
active concavity constraint.

Return type
list

Tools

tools

pystoned.utils.tools.**set_neos_email**(*address*)

pass email address to NEOS server

Parameters
address (*String*) – your own vaild email address.

CHAPTER

FIVE

ADVANCED TOPICS

5.1 Removing the constraint on Beta

We can remove the constrinat on Beta by adding `model.__model__.beta.setlb(None)` before the model optimization (i.e., `model.optimize()`). Now the estimated Beta can take any value in Reals. For example, the constraint Beta >= 0 is removed in CQR:

```
# import packages
from pystoned import CQER
from pystoned.constant import CET_ADDI, FUN_PROD, OPT_LOCAL, RTS_VRS
from pystoned import dataset as dataset

# import the GHG example data
data = dataset.load_GHG_abatement_cost(x_select=['HRSN', 'CPNK', 'GHG'], y_select=['VALK
↪'])

# calculate the quantile model
model = CQER.CQR(y=data.y, x=data.x, tau=0.5, z=None, cet=CET_ADDI, fun=FUN_PROD,␣
↪rts=RTS_VRS)
# remove the constraint beta >= 0
model.__model__.beta.setlb(None)
model.optimize(OPT_LOCAL)

# display estimated beta
model.display_beta()
```

5.2 Adding an additional constraint

The extral constraint, e.g., 0 <= Beta <= 1, can be added using

```
from pyomo.environ import Constraint

def constraint_rule(model, i, j):
    upperbound = [1.0, 1.0, 1.0]
    return model.beta[i, j] <= upperbound[j]

res.__model__.beta_constraint_rule = Constraint(res.__model__.I,
                                                res.__model__.J,
                                                rule=constraint_rule,
                                                doc='beta constraint')
```

A simple CQR example:

```
# import packages
from pystoned import CQER
from pystoned.constant import CET_ADDI, FUN_PROD, OPT_LOCAL, RTS_VRS
from pystoned import dataset as dataset
from pyomo.environ import Constraint

# import the GHG example data
data = dataset.load_GHG_abatement_cost(x_select=['HRSN', 'CPNK', 'GHG'], y_select=['VALK
↪'])

# calculate the quantile model
res = CQER.CQR(y=data.y, x=data.x, tau=0.5, z=None, cet=CET_ADDI, fun=FUN_PROD, rts=RTS_
↪VRS)
# add an extral constraint on beta
def constraint_rule(model, i, j):
    upperbound = [1.0, 1.0, 1.0]
    return model.beta[i, j] <= upperbound[j]

res.__model__.beta_constraint_rule = Constraint(res.__model__.I,
                                                res.__model__.J,
                                                rule=constraint_rule,
                                                doc='beta constraint')
res.optimize(OPT_LOCAL)

# display estimated beta
res.display_beta()
```

5.3 Miscellaneous

5.3.1 gams2python

pyomo provides a good coding environment that can help us smoothly transfer from GAMS to Python. Thus, we prepare a short tutorial to help GAMSers to understand how to rewrite the CNLS models, even other complicated models on Python.

A short comparison:

```
### We first present the GAMS code,
#sets
#     i        "DMU's"  /1*10/
#     j        'outputs' /Energy, Length, Customers/;

### then the Python code is added to compare.
from pyomo.environ import *
model = ConcreteModel(name = "CNLS")
model.i = Set(initialize=['i1', 'i2', 'i3', 'i4', 'i5', 'i6','i7', 'i8', 'i9','i10'],␣
↪doc='DMUS', ordered=True)
model.j = Set(initialize=['Energy', 'Length', 'Customers'], doc='outputs')

# alias(i,h);
```

(continues on next page)

(continued from previous page)

```
model.h = SetOf(model.i)

#Table data(i,j)
#$Include energy.txt;

dtab = {
        ('i1',  'Energy')   : 75,
        ('i1',  'Length')   : 878,
        ('i1',  'Customers'): 4933,
        ('i2',  'Energy')   : 62,
        ('i2',  'Length')   : 964,
        ('i2',  'Customers'): 6149,
        ('i3',  'Energy')   : 78,
        ('i3',  'Length')   : 676,
        ('i3',  'Customers'): 6098,
        ('i4',  'Energy')   : 683,
        ('i4',  'Length')   : 12522,
        ('i4',  'Customers'): 55226,
        ('i5',  'Energy')   : 27,
        ('i5',  'Length')   : 697,
        ('i5',  'Customers'): 1670,
        ('i6',  'Energy')   : 295,
        ('i6',  'Length')   : 953,
        ('i6',  'Customers'): 22949,
        ('i7',  'Energy')   : 44,
        ('i7',  'Length')   : 917,
        ('i7',  'Customers'): 3599,
        ('i8',  'Energy')   : 171,
        ('i8',  'Length')   : 1580,
        ('i8',  'Customers'): 11081,
        ('i9',  'Energy')   : 98,
        ('i9',  'Length')   : 116,
        ('i9',  'Customers'): 377,
        ('i10', 'Energy')   : 203,
        ('i10', 'Length')   : 740,
        ('i10', 'Customers'): 10134,
        }
model.d = Param(model.i, model.j, initialize=dtab, doc='output data')

#PARAMETERS
#c(i)         "Total cost of firm i"
#y(i,j)       "Output j of firm i";
model.c = Param(model.i, initialize={'i1': 1612,
                                     'i2': 1659,
                                     'i3': 1708,
                                     'i4': 18918,
                                     'i5': 1167,
                                     'i6': 3395,
                                     'i7': 1333,
                                     'i8': 3518,
                                     'i9': 1415,
                                     'i10':2469,
```

(continues on next page)

(continued from previous page)

```
                                    },
                        doc='Cost data')

def y_init(model, i, j):
    return  model.d[i, j]
model.y = Param(model.i, model.j, initialize=y_init, doc='output data')

#VARIABLES
#E(i)            "Composite error term (v + u)"
#SSE             "Sum of squares of residuals";
#POSITIVE VARIABLES
#b(i,j)    "Beta-coefficients (positivity ensures monotonicity)"
#Chat(i)  ;

model.b = Var(model.i, model.j, bounds=(0.0,None), doc='beta-coeff')
model.e = Var(model.i, doc='res')
model.f = Var(model.i, bounds=(0.0,None), doc='frontier')

#Equations
#QSSE                  objective function = sum of squares of residuals
#QREGRESSION(i)        log-transformed regression equation
#Qlog(i)               supporting hyperplanes of the nonparametric cost function
#QCONC(i,h)            concavity constraint (Afriat inequalities);

#QSSE..                SSE=e=sum(i,E(i)*E(i)) ;
#QREGRESSION(i)..      log(C(i)) =e= log(Chat(i) + 1) + E(i);
#Qlog(i)..             Chat(i) =e= sum(j, b(i,j)*Y(i,j)) - 1;
#QCONC(i,h)..          sum(j, b(i,j)*Y(i,j)) =g= sum(j, b(h,j)*Y(i,j));

def objective_rule(model):
    return sum(model.e[i]*model.e[i] for i in model.i)
model.objective = Objective(rule=objective_rule, sense=minimize, doc='objective function
↪')

def qreg_rule(model, i):
    return log(model.c[i]) == log(model.f[i] + 1) + model.e[i]
model.qreg = Constraint(model.i, rule=qreg_rule, doc='log-transformed regression')

def qlog_rule(model, i):
    return model.f[i] == sum(model.b[i, j]*model.y[i, j] for j in model.j) - 1
model.qlog = Constraint(model.i, rule=qlog_rule, doc='cost function')

def qconvex_rule(model, i, h):
    return sum(model.b[i,j]*model.y[i,j] for j in model.j) >= sum(model.b[h,j]*model.y[i,
↪j] for j in model.j)
model.qconvex = Constraint(model.i, model.h, rule=qconvex_rule, doc='convexity constraint
↪')

# Execute the model
#MODEL StoNED /all/;
#SOLVE StoNED using NLP Minimizing SSE;
```

(continues on next page)

(continued from previous page)

```
from pyomo.opt import SolverFactory
import pyomo.environ
from os import environ
solver_manager = SolverManagerFactory('neos')
environ['NEOS_EMAIL'] = 'email@address'
results = solver_manager.solve(model, opt='minos')

#display E.l, b.l;

model.e.display()
model.b.display()
```

5.3.2 CNLS_ConcreteModel

We also prepare a concrete model that does not need to call any of our developed functions in the pyStoNED package. In this concrete model, one can define the parameter specification by reading data from an Excel file.

A concreteModel for CNLS estimation:

```
# Import packages
import pd as pd
import numpy as np
import sys
from pyomo.environ import *

# Sets
i = np.array(['i{}'.format(i) for i in range(1, 90)])
model.i = Set(initialize=i, doc='DMUs', ordered=True )
model.j = Set(initialize=['j1', 'j2', 'j3'], doc='inputs and outputs')

# alias
model.h = SetOf(model.i)

# Parameters
# output (y1,y2,y3), avaiable at https://raw.githubusercontent.com/ds2010/pyStoNED/
↪master/pystoned/data/electricityFirms.csv.
df1 = pd.read_excel('y.xlsx', header=0, index_col=0)
Dict1 = dict()
for i in df1.index:
    for j in df1.columns:
        Dict1[i, j] = float(df1[j][i])
model.y = Param(model.i, model.j, initialize = Dict1)

# input (cost)
df2 = pd.read_excel('x.xlsx', header=0, index_col=0)
Dict2 = dict()
for i in df2.index:
    Dict2[i] = float(df2['TOTEX'][i])
model.c = Param(model.i, initialize = Dict2)

#Variables
model.b = Var(model.i, model.j, bounds=(0.0,None), doc='beta-coeff')
```

(continues on next page)

(continued from previous page)

```
model.e = Var(model.i, doc='res')
model.f = Var(model.i, bounds=(0.0,None), doc='frontier')

# Constraints and objective f.
def qreg_rule(model, i):
    return log(model.c[i]) == log(model.f[i] + 1) + model.e[i]
model.qreg = Constraint(model.i, rule=qreg_rule, doc='log-transformed regression')

def qlog_rule(model, i):
    return model.f[i] == sum(model.b[i, j]*model.y[i, j] for j in model.j) - 1
model.qlog = Constraint(model.i, rule=qlog_rule, doc='cost function')

def qconcav_rule(model, i, h):
    return sum(model.b[i,j]*model.y[i,j] for j in model.j) >= sum(model.b[h,j]*model.y[i,
↪j] for j in model.j)
model.qconcav = Constraint(model.i, model.h, rule=qconcav_rule, doc='concavity constraint
↪')

def objective_rule(model):
    return sum(model.e[i]*model.e[i] for i in model.i)
model.objective = Objective(rule=objective_rule, sense=minimize, doc='Define objective␣
↪function')

#Solve the model
from os import environ
solver_manager = SolverManagerFactory('neos')
environ['NEOS_EMAIL'] = 'email@address'
results = solver_manager.solve(model, opt='knitro', tee=True)

#retrive the beta
ind = list(model.b)
val = list(model.b[:,:].value)
df_b = [ i + tuple([j]) for i, j in zip(ind, val)]
df_b = np.asarray(df_b)
# display beta coefficients
df_b

df_b = pd.DataFrame(df_b,columns = ['Name', 'Key', 'Value'])
df_b = df_b.pivot(index='Name', columns='Key', values='Value')
# retrive the residuals
ind = list(model.e)
val = list(model.e[:].value)
df_e = np.asarray(val)
#display residuals
df_e
```

5.3.3 DEA_ConcreteModel

We also prepare a concrete model that can be used to calculate the input oriented VRS model.

A ConcreteModel for DEA estimation:

```
# import PYOMO package
from pyomo.environ import *
# creat a concrete model
model = ConcreteModel()

# Sets
model.i = Set(initialize=['i1', 'i2','i3', 'i4','i5', 'i6','i7', 'i8','i9', 'i10'], doc=
↪'DMUs', ordered=True)
model.j = Set(initialize=['OPEX', 'CAPEX'])   # inputs
model.k = Set(initialize=['Energy', 'Length', 'Customers'])   #outputs

# Alias
model.io = SetOf(model.i)

# Parameters (define and import the input-output data)
inputdata = {
                ('OPEX', 'i1') :   681,
                ('OPEX', 'i2') :   559,
                ('OPEX', 'i3') :   836,
                ('OPEX', 'i4') :   7559,
                ('OPEX', 'i5') :   424,
                ('OPEX', 'i6') :   1483,
                ('OPEX', 'i7') :   658,
                ('OPEX', 'i8') :   1433,
                ('OPEX', 'i9') :   850,
                ('OPEX', 'i10') :  1155,
                ('CAPEX', 'i1') :  729,
                ('CAPEX', 'i2') :  673,
                ('CAPEX', 'i3') :  851,
                ('CAPEX', 'i4') :  8384,
                ('CAPEX', 'i5') :  562,
                ('CAPEX', 'i6') :  1587,
                ('CAPEX', 'i7') :  570,
                ('CAPEX', 'i8') :  1311,
                ('CAPEX', 'i9') :  564,
                ('CAPEX', 'i10'):  1108,
                }
model.x = Param(model.j, model.i, initialize=inputdata)

outputdata = {
                ('Energy', 'i1'):   75,
                ('Energy', 'i2'):   62,
                ('Energy', 'i3'):   78,
                ('Energy', 'i4'):   683,
                ('Energy', 'i5'):   27,
                ('Energy', 'i6'):   295,
                ('Energy', 'i7'):   44,
                ('Energy', 'i8'):   171,
                ('Energy', 'i9'):   98,
                ('Energy', 'i10'):  203,
                ('Length', 'i1'):   878,
                ('Length', 'i2'):   964,
                ('Length', 'i3'):   676,
```

(continues on next page)

(continued from previous page)

```
                ('Length', 'i4'):   12522,
                ('Length', 'i5'):   697,
                ('Length', 'i6'):   953,
                ('Length', 'i7'):   917,
                ('Length', 'i8'):   1580,
                ('Length', 'i9'):   116,
                ('Length', 'i10'):  740,
                ('Customers', 'i1'):4933,
                ('Customers', 'i2'):6149,
                ('Customers', 'i3'):6098,
                ('Customers', 'i4'):55226,
                ('Customers', 'i5'):1670,
                ('Customers', 'i6'):22949,
                ('Customers', 'i7'):3599,
                ('Customers', 'i8'):11081,
                ('Customers', 'i9'):377,
                ('Customers', 'i10'):10134,
                }
model.y = Param(model.k, model.i, initialize=outputdata)

# Variables
model.lamda = Var(model.io, model.i, bounds=(0.0, None), doc='envelopment efficiency')
model.theta = Var(model.io, doc='intensity variable')

# objective
def objective_rule(model):
    return sum(model.theta[io] for io in model.io)
model.objective = Objective(rule=objective_rule, sense=minimize, doc='Define objective␣
↪function')
# Constraints
def output_rule(model, io, k):
    return sum(model.lamda[io, i] * model.y[k, i] for i in model.i) >= model.y[k, io]
model.output = Constraint(model.io, model.k, rule=output_rule, doc='output constraints')
def input_rule(model, io, j):
    return (model.theta[io] * model.x[j, io]) >= sum(
              model.lamda[io, i] * model.x[j, i] for i in model.i)
model.input = Constraint(model.io, model.j, rule=input_rule, doc='input constraints')
def vrs_rule(model, io):
    return sum(model.lamda[io, i] for i in model.i) == 1
model.vrs = Constraint(model.io, rule=vrs_rule, doc='VRS constraints')

# calculate the DEA model
from pyomo.opt import SolverFactory
from os import environ
solver_manager = SolverManagerFactory('neos')
environ['NEOS_EMAIL'] = 'email@address'
results = solver_manager.solve(model, opt='cplex')

# display the estimates
# efficiency
model.theta.display()
# intensity
```

(continues on next page)

(continued from previous page)

```
model.lamda.display()
```

CHAPTER

SIX

CONTRIBUTING

Thanks for your contribution to the pyStoNED. Please first discuss the change with the developers of this repository by creating an issue before making a change.

6.1 Style

For the consistency of the source code, we follow PEP8 python style guide.

Using `yapf -i` to format the codes:

```
$ yapf -i pystoned/example.py
```

Using `pylint` to check the other requirements:

```
$ pylint pystoned/example.py
```

6.2 Commit message

Example:

```
refactor(pyStoNED): Rename the variables in biMat.py

For fitting the requirement of PEP8 style,
I renamed the variables in biMat.py:
* ... to ...
* ... to ...
* ... to ...

Fixes #0000
```

Please write 1 commit message for one modification. Do not contain multiple changes in 1 commit. If the commit fixes any issue, please add `Fixes #xxxx` to the end of the commit message.

The first line should not be longer than 70 characters, the second line is always blank and other lines should be wrapped at 80 characters. The type should always be lowercase as shown below:

- feat: (new feature for the user, not a new feature for build script)
- fix: (bug fix for the user, not a fix to a build script)
- docs: (changes to the documentation)
- style: (formatting, missing semi colons, etc; no production code change)

- refactor: (refactoring code, eg. renaming a variable)
- test: (adding tests, refactoring tests; no production code change)

Reference:

- https://seesparkbox.com/foundry/semantic_commit_messages
- http://karma-runner.github.io/1.0/dev/git-commit-msg.html

6.3 Pull Request Process

1. Create a Fork of the project to your own repository.
2. Create a Branch in your Fork.
3. Make your contributions.
4. Make a Test that checks the change you did.
5. Format your code by `yapf` to `PEP8` style.
6. Create a Pull Request (PR) and Provide your test in it.
7. We will review your PR and discuss with you.

Thanks for your contributions!

CHAPTER

SEVEN

CITING PYSTONED

If you use pyStoNED for published work, we encourage you to cite our pyStoNED paper and other related theoretical papers. Please use the following BibTeX entries:

- pyStoNED paper

```
@techreport{Dai2021,
    title    = {{pyStoNED: A Python package for convex regression and frontier␣
↪estimation}},
    author   = {Dai, Sheng and Fang, Yu Hsueh and Lee, Chia Yen and Kuosmanen, Timo},
    year     = 2021,
    type     = {{arXiv preprint arXiv:2109.12962}}
}
```

- Theoretical papers

```
@article{Kuosmanen2008,
    title    = {{Representation theorem for convex nonparametric least squares}},
    year     = {2008},
    journal  = {Econometrics Journal},
    author   = {Kuosmanen, Timo},
    pages    = {308--325},
    doi      = {10.1111/j.1368-423X.2008.00239.x}
}
```

```
@article{Kuosmanen2010,
    title    = {{Data envelopment analysis as nonparametric least-squares regression}},
    year     = {2010},
    journal  = {Operations Research},
    author   = {Kuosmanen, Timo and Johnson, Andrew L.},
    pages    = {149--160},
    doi      = {10.1287/opre.1090.0722}
}
```

```
@article{Kuosmanen2012,
    title.   = {{Stochastic non-smooth envelopment of data: Semi-parametric frontier␣
↪estimation subject to shape constraints}},
    year     = {2012},
    journal  = {Journal of Productivity Analysis},
    author   = {Kuosmanen, Timo and Kortelainen, Mika},
    pages    = {11--28},
```

(continues on next page)

(continued from previous page)

```
    doi       = {10.1007/s11123-010-0201-3}
}
```

```
@article{Lee2013,
    title     = {{A more efficient algorithm for Convex Nonparametric Least Squares}},
    year      = {2013},
    journal   = {European Journal of Operational Research},
    author    = {Lee, Chia Yen and Johnson, Andrew L. and Moreno-Centeno, Erick and␣
↪Kuosmanen, Timo},
    pages     = {391--400},
    doi       = {10.1016/j.ejor.2012.11.054},
}
```

```
@incollection{Ray2015,
    title     = {{An Introduction to CNLS and StoNED Methods for Efficiency Analysis:␣
↪Economic Insights and Computational Aspects}},
    year      = {2015},
    booktitle= {Benchmarking for Performance Evaluation: A Production Frontier Approach},
    author    = {Johnson, Andrew L. and Kuosmanen, Timo},
    editor    = {Ray, Subhash C. and Kumbhakar, Subal C. and Dua, Pami},
    chapter   = {3},
    pages     = {117--186},
    publisher= {Springer}
}
```

```
@incollection{Kuosmanen2015,
    title     = {{Stochastic Nonparametric Approach to Efficiency Analysis: A unified␣
↪Framework}},
    year      = {2015},
    booktitle= {Data Envelopment Analysis},
    author    = {Kuosmanen, Timo and Johnson, Andrew and Saastamoinen, Antti},
    editor    = {Zhu, Joe},
    chapter   = {7},
    pages     = {191-244},
    publisher= {Springer}
}
```

CHAPTER

EIGHT

FREE ONLINE COURSE

Users of the pyStoNED package may benefit of the free online course "Productivity and Efficiency Analysis"(30E00300) organized at Aalto University of Business, Spring 2020, including video lessons, empirical examples and supporting materials. The course materials are available on the YouTube channel.

Table of contents:

- Lesson 1a: What is productivity and why it matters?
- Lesson 1b: Taxonomy of frontier estimation methods
- Lesson 1c: Productivity analysis in action: Incentive regulation of electricity distribution networks
- Lesson 2a: Application of DEA
- Lesson 2b: DEA formulations
- Lesson 2c: Production theory
- Lesson 2d: Statistical approach to DEA
- Lesson 2e*: Pre-history of DEA in economics
- Lesson 3a: Parametric approach
- Lesson 3b: Basics of SFA
- Lesson 3c: Application of SFA
- Lesson 4a: Application of StoNED
- Lesson 4b: StoNED estimation
- Lesson 4c: Convex regression
- Lesson 4d: Decomposing the residual
- Lesson 4e*: Quantiles and expectiles
- Lesson 5a: What are z-variables and where to put them?
- Lesson 5b: SFA modeling of z-variables
- Lesson 5c: Contextual variables in DEA
- Lesson 5d: Semi-nonparametric approach
- Lesson 6a: Synergies of joint production
- Lesson 6b: Distance functions and DEA
- Lesson 6c: Bad outputs
- Lesson 6d: Parametric distance functions

- Lesson 6e: StoNED with multiple outputs
- Lesson 7a: Growth accounting and index numbers
- Lesson 7b: Malmquist index
- Lesson 7c: Application of Malmquist index
- Lesson 7d: Case: Green TFP growth
- Lesson 8a: Entry & exit
- Lesson 8b: Share weights
- Lesson 8c: Application to Finnish agriculture

CHAPTER

NINE

LIST OF ACRONYMS

For sake of demonstration, we use the following list of acronyms in this project.

COLS - Corrected Ordinary Least Squares

C2NLS - Corrected Convex Nonparametric Least Squares

CER - Convex Expectile Regression

CNLS - Convex Nonparametric Least Squares

CQR - Convex Quantile Regression

CRS - Constant Returns to Scale

DDF - Directional Distance Function

DEA - Data Envelopment Analysis

DMU - Decision-Making Unit

FDH - Free Disposal Hull

GAMS - General Algebraic Modeling System

MOM - Method of Moments

QLE - Quasi-likelihood Estimation

ICNLS - Isotonic Convex Nonparametric Least Square

ICER - Isotonic Convex Quantile Regression

ICQR - Isotonic Convex Expectile Regression

StoNED - Stochastic Nonparametric Envelopment of Data

StoNEZD - Stochastic Semi-nonparametric Envelopment of Z Variables Data

KDE - Kernel Density Estimation

VRS - Variable Returns to Scale

PYTHON MODULE INDEX

p

INDEX

Symbols

C

D

F

G

I

L

M

O

P

R

S

W