Scikit-Criteria Documentation

Release 0.2.10

Juan BC

Jun 22, 2018

Contents

Scikit-Criteria is a collection of Multiple-criteria decision analysis (MCDA) methods integrated into scientific python stack. Is Open source and commercially usable.

CHAPTER 1

Help & discussion mailing list

Our Google Groups mailing list is here.

You can contact me at: jbc.develop@gmail.com (if you have a support question, try the mailing list first)

CHAPTER 2

Code Repository & Issues

https://github.com/leliel12/scikit-criteria

CHAPTER 3

License

Scikit-Criteria is under The 3-Clause BSD License

This license allows unlimited redistribution for any purpose as long as its copyright notices and the license's disclaimers of warranty are maintained.

CHAPTER 4

Citation

If you are using Scikit-Criteria in your research, please cite:

If you use scikit-criteria in a scientific publication, we would appreciate citations to the following paper:

> Cabral, Juan B., Nadia Ayelen Luczywo, and José Luis Zanazzi 2016 Scikit-Criteria: Colección de Métodos de Análisis Multi-Criterio Integrado Al Stack Científico de Python. In XLV Jornadas Argentinas de Informática E Investigación Operativa (45JAIIO)-XIV Simposio Argentino de Investigación Operativa (SIO) (Buenos Aires, 2016) Pp. 59–66. http://45jaiio.sadio.org.ar/sites/default/files/Sio-23.pdf.

Bibtex entry:

```
@inproceedings{scikit-criteria,
    author={
        Juan B Cabral and Nadia Ayelen Luczywo and Jos\'{e} Luis Zanazzi},
    title={
        Scikit-Criteria: Colecci\'{o}n de m\'{e}todos de an\'{a}lisis
        multi-criterio integrado al stack cient\'{i}fico de {P}ython},
    booktitle = {
        XLV Jornadas Argentinas de Inform{\'a}tica
        e Investigaci{\'o}n Operativa (45JAIIO)-
        XIV Simposio Argentino de Investigaci\'{o}n Operativa (SIO)
        (Buenos Aires, 2016)},
    year={2016},
    pages = {59--66},
    url={http://45jaiio.sadio.org.ar/sites/default/files/Sio-23.pdf}
}
```

Full Publication: http://sedici.unlp.edu.ar/handle/10915/58577

CHAPTER 5

Contents

5.1 Installing scikit-criteria

The easiest way to install scikit-criteria is using `pip`

```
pip install -U scikit-criteria
```

If you have not installed NumPy or SciPy yet, you can also install these using conda or pip. When using pip, please ensure that *binary wheels* are used, and NumPy and SciPy are not recompiled from source, which can happen when using particular configurations of operating system and hardware (such as Linux on a Raspberry Pi). Building numpy and scipy from source can be complex (especially on Windows) and requires careful configuration to ensure that they link against an optimized implementation of linear algebra routines. Instead, use a third-party distribution as described below.

5.1.1 Third-party Distributions

If you don't already have a python installation with numpy and scipy, we recommend to install either via your package manager or via a python bundle. These come with numpy, scipy, matplotlib and many other helpful scientific and data processing libraries.

Available options are:

Canopy and Anaconda for all supported platforms

Canopy and Anaconda both ship a recent version of Python, in addition to a large set of scientific python library for Windows, Mac OSX and Linux.

5.2 Tutorial

This section contains a step-by-step by example tutorial of how to use scikit-criteria

Contents:

5.2.1 Quick Start

This tutorial show how to create a scikit-criteria `Data` structure, and how to feed them inside different multicriteria decisions algorithms.

Conceptual Overview

The multicriteria data are really complex thing; mostly because you need at least 2 totally disconected vectors to decribe your problem: A alternative matrix (`mtx`) and a vector that indicated the optimal sense of every criteria (`criteria`); also maybe you want to add weights to your criteria

The `skcteria.Data` object need at least the first two to be created and also accepts the weights, the names of the criteria and the names of alternatives as optional parametes.

Your First Data object

First we need to import the `Data` structure and the `MIN`, `MAX` contants from scikit-criteria:

```
In [2]: from skcriteria import Data, MIN, MAX
```

Then we need to create the `mtx` and `criteria` vectors.

The `mtx` must be a **2D array-like** where every column is a criteria, and every row is an alternative

```
In [3]: # 2 alternatives by 3 criteria
        mtx = [
            [1, 2, 3],  # alternative 1
            [4, 5, 6],  # alternative 2
        ]
        mtx
```

```
Out[3]: [[1, 2, 3], [4, 5, 6]]
```

The `criteria` vector must be a **1D array-like** whit the same number of elements than columns has the alternative mattrix (`mtx`) where every component represent the optimal sense of every criteria.

```
In [4]: # let's says the first two alternatives are
        # for maximization and the last one for minimization
        criteria = [MAX, MAX, MIN]
        criteria
```

```
Out[4]: [1, 1, -1]
```

as you see the `MAX` and `MIN` constants are only aliases for the numbers `-1` (minimization) and `1` (maximization). As you can see the contantes usage makes the code more readable. Also you can use as aliases of minimization and maximization the built-in function `min`, `max`, the numpy function `np.min`, `np.max`, `np.amin`, `np.amax`, `np.nanmin`, `np.nanmax` and the strings `min`, `minimization`, `max` and `maximization`.

Now we can combine this two vectors in our scikit-criteria data.

```
In [5]: # we use the built-in function as aliases
        data = Data(mtx, [min, max, min])
        data
```

```
Out[5]: ALT./CRIT.    C0 (min)    C1 (max)    C2 (min)
        ------------  ----------  ----------  ----------
             A0            1           2           3
             A1            4           5           6
```

As you can see the otput of the `Data` structure is much more friendly as the plain python lists.

To change the generic names of the alternatives (A0 and A1) and the criteria (C0, C1 and C2); let's asume that our Data is about cars (*car 0* and *car 1*) and their characteristics of evaluation are *autonomy* (`MAX`), *confort* (`MAX`) and *price* (`MIN`).

To feed this information to our `Data` structure we have the params: `anames` that accept the names of alternatives (must be the same number as row the `mtx` has), and `cnames` the criteria names (with the same number of elements as columns has the `mtx`)

```
In [6]: data = Data(mtx, criteria,
                    anames=["car 0", "car 1"],
                    cnames=["autonomy", "confort", "price"])
        data
```

```
Out[6]: ALT./CRIT.    autonomy (max)    confort (max)    price (min)
        ------------  ----------------  ---------------  -------------
           car 0              1                 2               3
           car 1              4                 5               6
```

In our final step let's asume we know in our case, that the importance of the autonomy is the *50%*, the confort only a *5%* and the price is *45%*. The param to feed this to the structure is called `weights` and must be a vector with the same elemts as criterias has your alternative matrix (number of columns)

```
In [7]: data = Data(mtx, criteria,
                    weights=[.5, .05, .45],
                    anames=["car 0", "car 1"],
                    cnames=["autonomy", "confort", "price"])
        data
```

```
Out[7]: ALT./CRIT.    autonomy (max) W.0.5    confort (max) W.0.05    price (min) W.0.45
        ------------  ----------------------  ----------------------  --------------------
           car 0                1                       2                      3
           car 1                4                       5                      6
```

Manipulating the Data

The data object are inmutable, if you want to modify it you need create a new one. All the numerical data (mtx, criteria, and weights) are stored as numpy arrays, and the alternative and criteria names as python tuples.

You can acces to the different parts of your data, simply by typing `data.<your-parameter-name>` for example:

```
In [8]: data.mtx
```

```
Out[8]: array([[1, 2, 3],
               [4, 5, 6]])
```

```
In [9]: data.criteria
```

```
Out[9]: array([ 1,  1, -1])
```

```
In [10]: data.weights
```

```
Out[10]: array([0.5 , 0.05, 0.45])
```

```
In [11]: data.anames, data.cnames
```

```
Out[11]: (('car 0', 'car 1'), ('autonomy', 'confort', 'price'))
```

If you want (for example) change the names of the cars from `car 0` and `car 1`; to `VW` and `Ford` you must copy from your original Data

```
In [12]: data = Data(data.mtx, data.criteria,
                     weights=data.weights,
                     anames=["VW", "Ford"],
```

```
                cnames=data.cnames)
data
```

Out[12]:

ALT./CRIT.	autonomy (max) W.0.5	confort (max) W.0.05	price (min) W.0.45
VW	1	2	3
Ford	4	5	6

Note: A more flexible data manipulation API will relased in future versions.

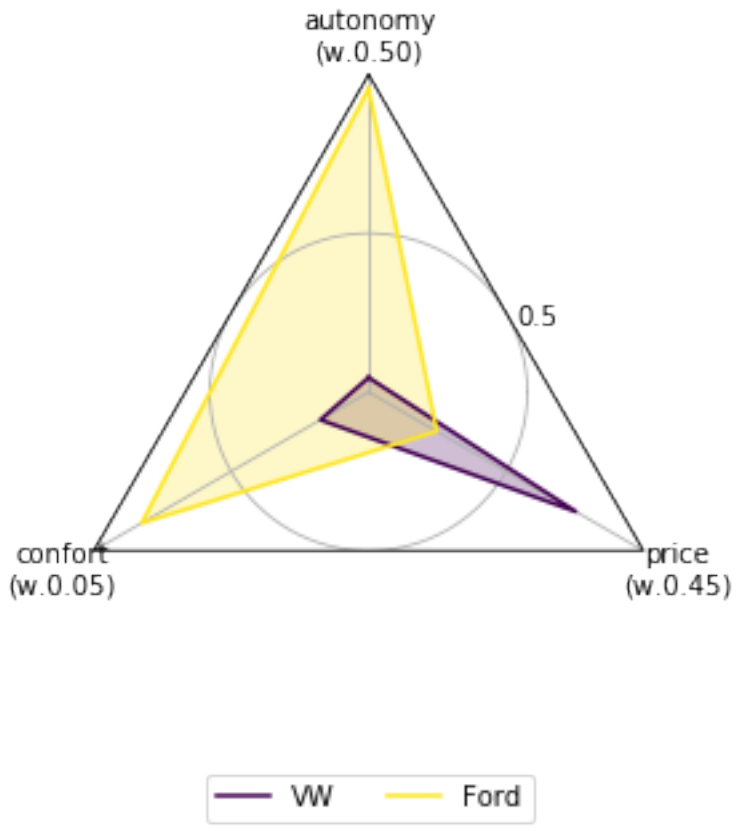
Plotting

The Data structure suport some basic rutines for ploting. Actually 5 types of plots are supported:

- Radar Plot (`radar`).
- Histogram (`hist`).
- Violin Plot (`violin`).
- Box Plot (`box`).
- Scatter Matrix (`scatter`).

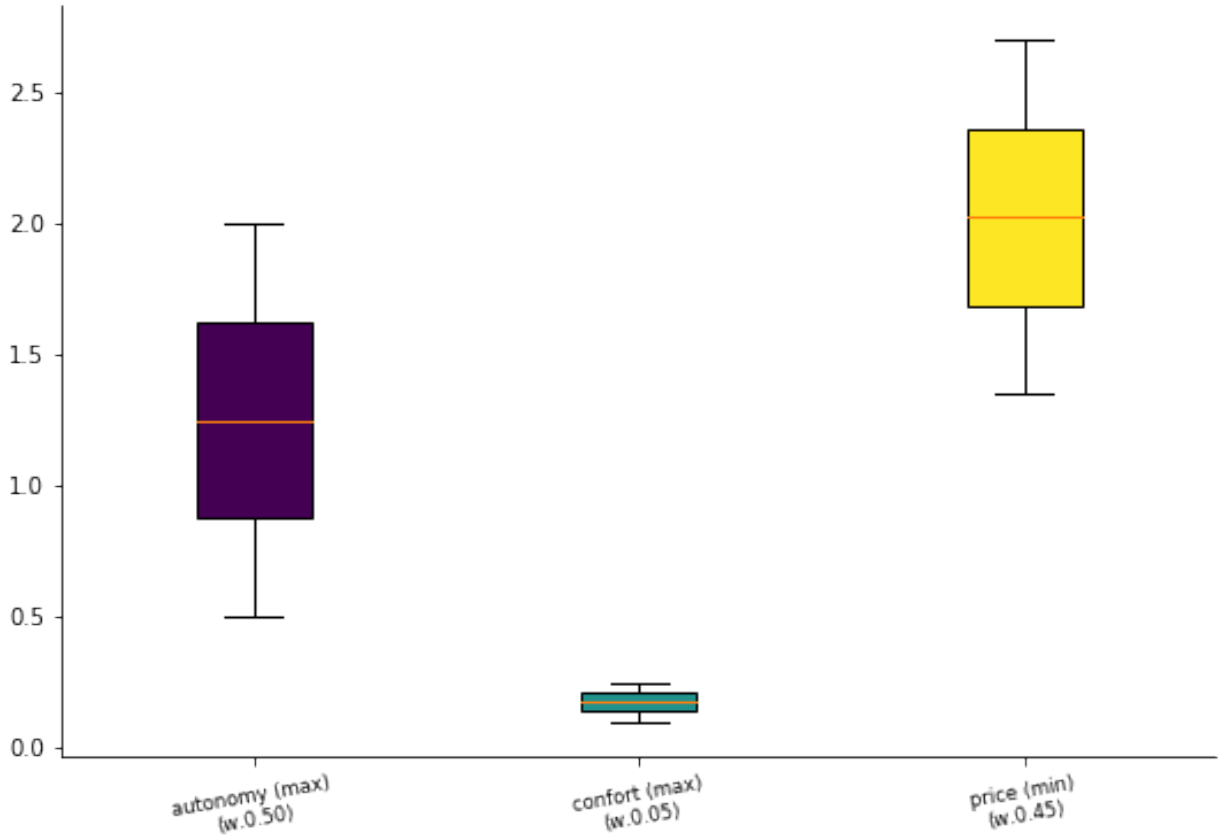
The default scikit criteria uses the Radar Plot to visualize all the data. Take in account that the radar plot by default convert all the minimization criteria to maximization and push all the values to be greater than 1 (obviously all this options can be overided).

```
In [13]: data.plot();
```



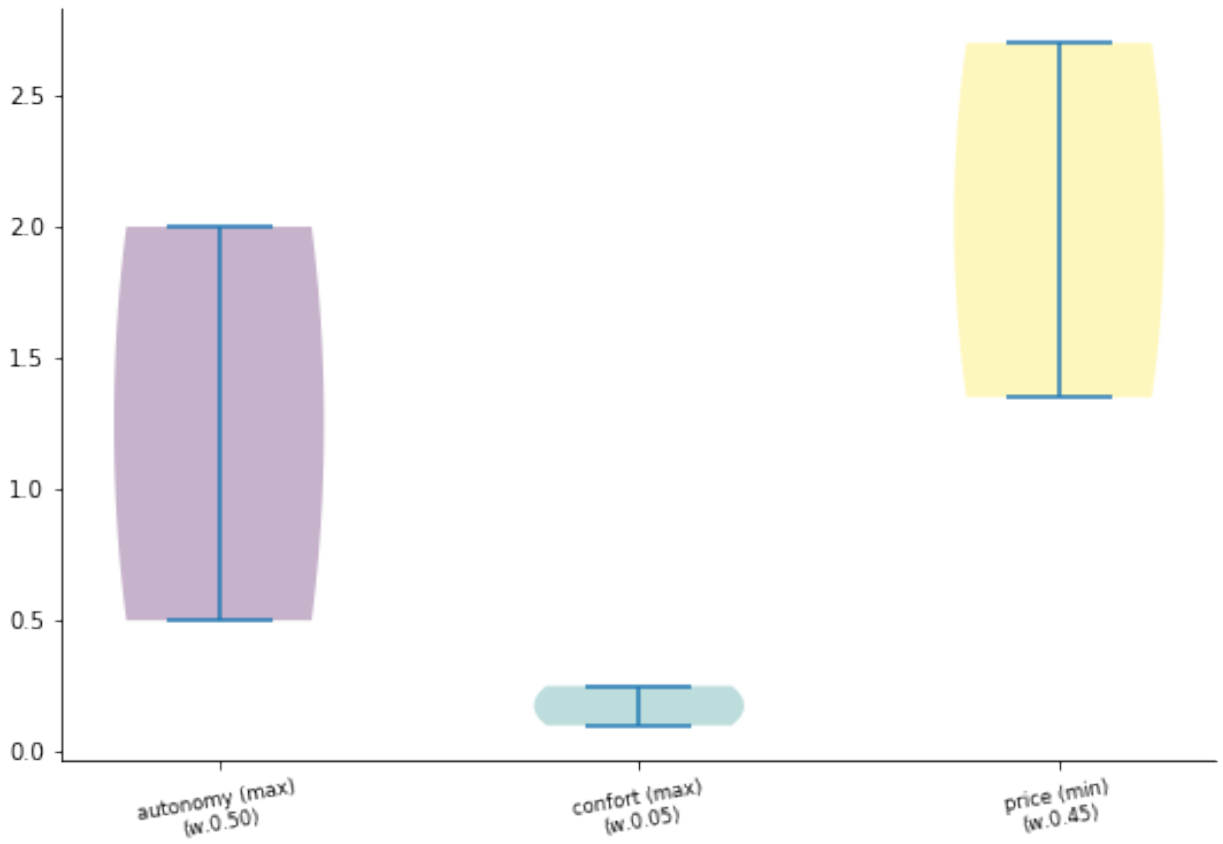
You can accesing the different plot by passing as first parameter the name of the plot

```
In [14]: data.plot("box");
```



or by using the name as method call inside the `plot` attribute

```
In [15]: data.plot.violin();
```

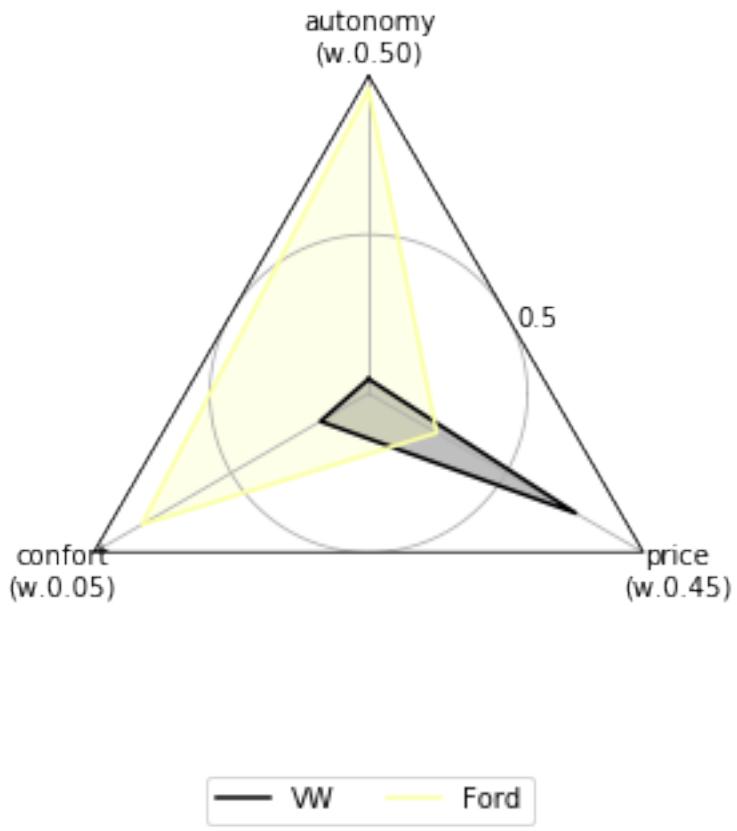


Every plot has their own set of parameters, but at last every one can receive:

- `ax`: The plot axis.
- `cmap`: The color map (More info).
- `mnorm`: The normalization method for the alternative matrix as string (Default: `"none"`).
- `wnorm`: The normalization method for the criteria array as string (Default: `"none"`).
- `weighted`: If you want to weight the criteria (Default: `True`).
- `show_criteria`: Show or not the criteria in the plot (Default: `True` in all except radar).
- `min2max`: Convert the minimization criteria into maximization one (Default: `False` in all except radar).
- `push_negatives`: If a criteria has values lesser than 0, add the minimun value to all the criteria (Default: `False` in all except radar).
- `addepsto0`: If a criteria has values equal to 0, add an ϵ value to all the criteria (Default: `False` in all except radar).

Let's change the colors of the radar plot and show their criteria optimization sense:

```
In [16]: data.plot.radar(cmap="inferno", show_criteria=False);
```



Using this data to feed some MCDA methods

Let's rank our toy data by Weighted Sum Model, Weighted Product Model and TOPSIS

```
In [17]: from skcriteria.madm import closeness, simple
```

First you need to create the decission maker.

Most of methods accepts as hyper parameters (parameters of the to configure the method), the method of normalization of the alternative matrix (divided by the `sum` in Weighted Sum and Weighted Product, and the `vector` normalization on `Topsis`) and the method to normalize the weight array (normally `sum`); But complex methods has more.

Weighted Sum Model:

```
In [18]: # first create the decision maker
         # (with the default hiper parameters)
         dm = simple.WeightedSum()
         dm

Out[18]: <WeightedSum (mnorm=sum, wnorm=sum)>

In [19]: # Now lets decide the ranking
         dec = dm.decide(data)
         dec

Out[19]: WeightedSum (mnorm=sum, wnorm=sum) - Solution:
          ALT./CRIT.    autonomy (max) W.0.5    confort (max) W.0.05    price (min) W.0.45    Rank
         ------------  ----------------------  ----------------------  --------------------  ------
```

```
          VW          1          2          3          1
        Ford          4          5          6          2
```

The result says that the **VW** is better than the **FORD**, lets make the maths:

Note: The last criteria is for minimization and because the WeightedSumModel only accepts maximization criteria by default, scikit-criteria invert all the values to convert the criteria to maximization

---

```
In [20]: print("VW:", 0.5 * 1/5. + 0.05 * 2/7. + 0.45 * 1 / (3/9.))
         print("FORD:", 0.5 * 4/5. + 0.05 * 5/7. + 0.45 * 1 / (6/9.))
```

```
VW: 1.4642857142857144
FORD: 1.1107142857142858
```

If you want to acces this points, the `Decision` object stores all the particular information of every method in a attribute called `e_`

```
In [21]: print(dec.e_)
         dec.e_.points
```

```
Extra(points)
```

```
Out[21]: array([1.46428571, 1.11071429])
```

Also you can acces the type of the solution

```
In [22]: print("Generate a ranking of alternatives?", dec.alpha_solution_)
         print("Generate a kernel of best alternatives?", dec.beta_solution_)
         print("Choose the best alternative?", dec.gamma_solution_)
```

```
Generate a ranking of alternatives? True
Generate a kernel of best alternatives? False
Choose the best alternative? True
```

The rank as numpy array (if this decision is a α-solution)

```
In [23]: dec.rank_
```

```
Out[23]: array([1, 2])
```

The index of the row of the best alternative (if this decision is a γ-solution)

```
In [24]: dec.best_alternative_, data.anames[dec.best_alternative_]
```

```
Out[24]: (0, 'VW')
```

And the kernel of the non supered alternatives (if this decision is a β-solution)

```
In [25]: # this return None because this
         # decision is not a beta-solution
         print(dec.kernel_)
```

```
None
```

Weighted Product Model

```
In [26]: dm = simple.WeightedProduct()
         dm
```

```
Out[26]: <WeightedProduct (mnorm=sum, wnorm=sum)>
```

```
In [27]: dec = dm.decide(data)
         dec
```

```
Out[27]: WeightedProduct (mnorm=sum, wnorm=sum) - Solution:
          ALT./CRIT.    autonomy (max) W.0.5    confort (max) W.0.05    price (min) W.0.45    Rank
          ------------  ----------------------  ----------------------  --------------------  ------
               VW                 1                       2                      3              2
              Ford                4                       5                      6              1
```

As before let's do the math (remember the weights are now exponets)

```
In [28]: print("VW:", ((1/5.) ** 0.5) * ((2/7.) ** 0.05) + ((1 / (3/9.)) ** 0.45))
         print("FORD:", ((4/5.) ** 0.5) * ((5/7.) ** 0.05) + ((1 / (6/9.)) ** 0.45))
```

```
VW: 2.059534375567646
FORD: 2.07967086650222
```

As wee expected the **Ford** are little better than the **VW**. Now lets theck the `e_` object

```
In [29]: print(dec.e_)
         dec.e_.points
```

```
Extra(points)
```

```
Out[29]: array([-0.16198384,  0.02347966])
```

As you note the points are differents, this is because internally to avoid undeflows Scikit-Criteria uses a sums of logarithms instead products. So let's check

```
In [30]: import numpy as np
         print("VW:", 0.5 * np.log10(1/5.) + 0.05 * np.log10(2/7.) + 0.45 * np.log10(1 / (3/9.)))
         print("FORD:", 0.5 * np.log10(4/5.) + 0.05 * np.log10(5/7.) + 0.45 * np.log10(1 / (6/9.)))
```

```
VW: -0.16198383976167505
FORD: 0.023479658287116456
```

TOPSIS

```
In [31]: dm = closeness.TOPSIS()
         dm
```

```
Out[31]: <TOPSIS (mnorm=vector, wnorm=sum)>
```

```
In [32]: dec = dm.decide(data)
         dec
```

```
Out[32]: TOPSIS (mnorm=vector, wnorm=sum) - Solution:
          ALT./CRIT.    autonomy (max) W.0.5    confort (max) W.0.05    price (min) W.0.45    Rank
          ------------  ----------------------  ----------------------  --------------------  ------
               VW                 1                       2                      3              2
              Ford                4                       5                      6              1
```

The TOPSIS add more information into the decision object.

```
In [33]: print(dec.e_)
         print("Ideal:", dec.e_.ideal)
         print("Anti-Ideal:", dec.e_.anti_ideal)
         print("Closeness:", dec.e_.closeness)
```

```
Extra(ideal, anti_ideal, closeness)
Ideal: [0.48507125 0.04642383 0.20124612]
Anti-Ideal: [0.12126781 0.01856953 0.40249224]
Closeness: [0.35548671 0.64451329]
```

Where the `ideal` and `anti_ideal` are the normalizated sintetic better and worst altenatives created by TOPSIS, and the `closeness` is how far from the *anti-ideal* and how closer to the *ideal* are the real alternatives

Finally we can change the normalization criteria of the alternative matric to `sum` (divide every value by the sum opf their criteria) and check the result:

```
In [34]: dm = closeness.TOPSIS(mnorm="sum")
         dm
```

```
Out[34]: <TOPSIS (mnorm=sum, wnorm=sum)>
```

```
In [35]: dm.decide(data)
```

```
Out[35]: TOPSIS (mnorm=sum, wnorm=sum) - Solution:
```

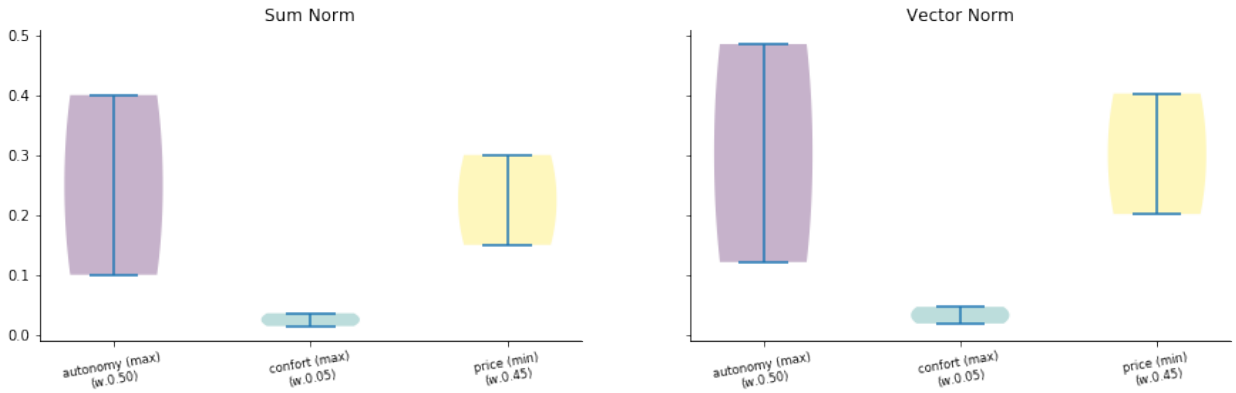
ALT./CRIT.	autonomy (max) W.0.5	confort (max) W.0.05	price (min) W.0.45	Rank
VW	1	2	3	2
Ford	4	5	6	1

The rankin has changed so, we can compare the two normalization by plotting

```
In [36]: import matplotlib.pyplot as plt

         f, (ax1, ax2) = plt.subplots(1, 2, sharey=True)
         ax1.set_title("Sum Norm")
         data.plot.violin(mnorm="sum", ax=ax1);

         ax2.set_title("Vector Norm")
         data.plot.violin(mnorm="vector", ax=ax2);
         f.set_figwidth(15)
```




```
In [37]: import datetime as dt
         import skcriteria
         print("Scikit-Criteria version:", skcriteria.VERSION)
         print("Running datetime:", dt.datetime.now())
```

```
Scikit-Criteria version: 0.2.10
Running datetime: 2018-06-22 00:59:20.974570
```

5.2.2 The SIMUS tutorial

SIMUS (*Sequential Interactive Model for Urban Systems*)

Is a tool to aid decision-making problems with multiple objectives. The method solves successive scenarios formulated as linear programs. For each scenario, the decision-maker must choose the criterion to be considered objective while the remaining restrictions constitute the constrains system that the projects are subject to. In each case, if there is a feasible solution that is optimum, it is recorded in a matrix of efficient results. Then, from this matrix two rankings allow the decision maker to compare results obtained by different procedures. The first ranking is obtained through a linear weighting of each column by a factor - equivalent of establishing a weight - and that measures the participation

of the corresponding project. In the second ranking, the method uses dominance and subordinate relationships between projects, concepts from the French school of MCDM.

The Case: Land rehabilitation

An important port city has been affected by the change in the modality of maritime transport, since the start of containers transport in the mid-20th century. The city was left with 39 hectares of empty docks, warehouses and a railway terminal.

Three projects was developed to decide what to do with this places

- **Project 1:** Corporate towers - Hotels - Navy Base - Small park
- **Project 2:** Habitational towers - Comercial Center in the old Railway terminal.
- **Project 3:** Convention center - Big park and recreational area.

The criteria for the analysis of proposals are:

1. New jobs positions (**jobs**).

- Green spaces (**green**)
- Financial feasibility (**fin**)
- Environmental impact (**env**)

Only for the 2nd criteria a maximun limit pf 500 are provided. El Decisor considera a los cuatro criterios como objetivos, por lo que se deberán resolver cuatro programas lineales con tres restricciones cada uno. The data are provided in the next table:

Criteria	Project 1	Project 2	Project 3	Right side value	Optimal Sense
jobs	250	130	350	•	Maximize
green	120	200	340	500	Maximize
fin	20	40	15	•	Maximize
env	800	1000	600	•	Maximize

Data input

We can create a `skcriteria.Data` object with all this information (except the limits):

Note: SIMUS uses the alternatives as columns and the criteria as rows; but in *scikit-criteria* is the oposite, so expect to see the previous table transposed.

```
In [1]: # first lets import the DATA class
        from skcriteria import Data

        data = Data(
            # the alternative matrix
            mtx=[[250, 120, 20, 800],
                 [130, 200, 40, 1000],
```

```
                [350, 340, 15, 600]],

            # optimal sense
            criteria=[max, max, min, max],

            # names of alternatives and criteria
            anames=["Prj 1", "Prj 2", "Prj 3"],
            cnames=["jobs", "green", "fin", "env"])

        # show the data object
        data
```

Out[1]:

ALT./CRIT.	jobs (max)	green (max)	fin (min)	env (max)
Prj 1	250	120	20	800
Prj 2	130	200	40	1000
Prj 3	350	340	15	600

Create the model

```
In [2]: # import the class
        from skcriteria.madm.simus import SIMUS

        # create the new simus and
        dm = SIMUS()
```

By default the call `SIMUS()` create a solver that internally uses the PuLP solver to solve the linear programs. Other availables solvers are:

- `SUMUS(solver='glpk')` for the GNU Linear programming toolkit
- `SUMUS(solver='gurobi')` to use Gurobi Optimizer
- `SUMUS(solver='cplex')` for IBM ILOG CPLEX Optimization Studio

Note: The check the full list of available optimizers are stored in `skcriteria.utils.lp.SOLVERS`.

Also the `njobs` parameters determines how many cores the user want to use to run the linear programs. For example `SIMUS(njobs=2)` uses up to two cores. (By default all CPUs are used).

Also the last (and most important) parameter is `rank_by` (default is 1): determines which of the two ranks methods executed by SIMUS is the one that determines the final ranking. If the experiment is consistent, the two methos *must* detemines the *same* ranking (Please check the paper for more details).

Solve the problem

This is achived by calling the method `decide()` of the decision maker object (`dm`)

```
In [3]: # store the decision inside the dec variable
        dec = dm.decide(data, b=[None, 500, None, None])

        # let's see the decision
        dec
```

Out[3]: SIMUS (mnorm=none, wnorm=none) - Solution:

ALT./CRIT.	jobs (max)	green (max)	fin (min)	env (max)	Rank

```
          Prj 1          250           120           20           800        3
          Prj 2          130           200           40          1000        2
          Prj 3          350           340           15           600        1
```

If you check the last column the raking is:

1. Project 3

- Project 2
- Project 1

Analysis

Most of the "intermediate" data of the SIMUS method are stored in the `e_` field of the decision object `dec`.

```
In [4]: dec.e_
```

```
Out[4]: Extra(rank_by, solver, stages, stage_results, points1, points2, tita_j_p, tita_j_d, doms, dor
```

for example the attribute `stages` stores all the Linear programs executed by SIMUS:

```
In [5]: dec._e.stages
```

```
Out[5]: [no-name:
         MAXIMIZE
         250*x0 + 130*x1 + 350*x2 + 0
         SUBJECT TO
         _C1: 120 x0 + 200 x1 + 340 x2 <= 500

         _C2: 20 x0 + 40 x1 + 15 x2 >= 15

         _C3: 800 x0 + 1000 x1 + 600 x2 <= 1000

         VARIABLES
         x0 Continuous
         x1 Continuous
         x2 Continuous, no-name:
         MAXIMIZE
         120*x0 + 200*x1 + 340*x2 + 0
         SUBJECT TO
         _C1: 250 x0 + 130 x1 + 350 x2 <= 350

         _C2: 20 x0 + 40 x1 + 15 x2 >= 15

         _C3: 800 x0 + 1000 x1 + 600 x2 <= 1000

         VARIABLES
         x0 Continuous
         x1 Continuous
         x2 Continuous, no-name:
         MINIMIZE
         20*x0 + 40*x1 + 15*x2 + 0
         SUBJECT TO
         _C1: 250 x0 + 130 x1 + 350 x2 <= 350

         _C2: 120 x0 + 200 x1 + 340 x2 <= 500

         _C3: 800 x0 + 1000 x1 + 600 x2 <= 1000

         VARIABLES
```

```
x0 Continuous
x1 Continuous
x2 Continuous, no-name:
MAXIMIZE
800*x0 + 1000*x1 + 600*x2 + 0
SUBJECT TO
_C1: 250 x0 + 130 x1 + 350 x2 <= 350

_C2: 120 x0 + 200 x1 + 340 x2 <= 500

_C3: 20 x0 + 40 x1 + 15 x2 >= 15

VARIABLES
x0 Continuous
x1 Continuous
x2 Continuous]
```

The attribute `stages_results` stores the *eficients restults normalized matrix*

```
In [6]: dec.e_.stage_results
```

```
Out[6]: array([[0.125     , 0.        , 0.875     ],
               [0.        , 0.38888889, 0.61111111],
               [0.        , 0.        , 0.        ],
               [0.05681818, 0.94318182, 0.        ]])
```

References

Munier, N., Carignano, C., & Alberto, C. UN MÉTODO DE PROGRAMACIÓN MULTIOBJETIVO. Revista de la Escuela de Perfeccionamiento en Investigación Operativa, 24(39).

See also:

If you're new to Python, you might want to start by getting an idea of what the language is like. Scikit-criteria is 100% Python, so if you've got minimal comfort with Python you'll probably get a lot more out of our project.

If you're new to programming entirely, you might want to start with this list of Python resources for non-programmers

If you already know a few other languages and want to get up to speed with Python quickly, we recommend Dive Into Python. If that's not quite your style, there are many other books about Python.

At last if you already know Python but check the Scipy Lecture Notes

5.3 API

Scikit-Criteria is a collections of algorithms, methods and techniques for multiple-criteria decision analysis.

5.3.1 `skcriteria.base` module

Module containing the basic functionality for the data representation used inside Scikit-Criteria.

class skcriteria.base.**Data**(*mtx*, *criteria*, *weights=None*, *anames=None*, *cnames=None*, *meta=None*)

Bases: `object`

Multi-Criteria data representation.

This make easy to manipulate:

- The matrix of alternatives. (`mtx`)
- The array with the sense of optimality of every criteria (`criteria`).
- Optional weights of the criteria (`weights`)
- Optional names of the alternatives (`anames`) and the criteria (`cnames`)
- Optional metadata (`meta`)

Attributes

anames	Names of the alternatives as tuple of string.
cnames	Names of the criteria as tuple of string.
criteria	Sense of optimality of every criteria
meta	Dict-like metadata
mtx	Alternative matrix as 2d numpy.ndarray.
weights	Relative importance of the criteria or None if all the same

Methods

plot	alias of *skcriteria.plot.DataPlotMethods*
raw()	Return a (mtx, criteria, weights, anames, cnames) tuple
to_str(**params)	String representation of the Data object.

anames
Names of the alternatives as tuple of string.

cnames
Names of the criteria as tuple of string.

criteria
Sense of optimality of every criteria

meta
Dict-like metadata

mtx
Alternative matrix as 2d numpy.ndarray.

plot
alias of *skcriteria.plot.DataPlotMethods*

raw()
Return a (mtx, criteria, weights, anames, cnames) tuple

to_str(**params*)
String representation of the Data object.

Parameters kwargs :

Parameters to configure tabulate

weights
Relative importance of the criteria or None if all the same

5.3.2 skcriteria.validate module

This module core functionalities for validate the data used inside scikit criteria.

- Constants that represent minimization and mazimization criteria.
- Scikit-Criteria Criteria ndarray creation.
- Scikit-Criteria Data validation.

skcriteria.validate.**MIN = -1**
Int: Minimization criteria

skcriteria.validate.**MAX = 1**
Int: Maximization criteria

exception skcriteria.validate.**DataValidationError**
Bases: exceptions.ValueError

Raised when some part of the multicriteria data (alternative matrix, criteria array or weights array) are not compatible with another part.

skcriteria.validate.**criteriarr**(*criteria*)
Validate if the iterable only contains MIN (or any alias) and MAX (or any alias) values. And also always returns an ndarray representation of the iterable.

Parameters **criteria** : Array-like

Iterable containing all the values to be validated by the function.

Returns numpy.ndarray :

Criteria array.

Raises **DataValidationError :**

if some value of the criteria array are not MIN (-1) or MAX (1)

skcriteria.validate.**validate_data**(*mtx*, *criteria*, *weights=None*)
Validate if the main components of the Data in scikit-criteria are compatible.

The function tests:

- The matrix (mtx) must be 2-dimensional.
- The criteria array must be a criteria array (criteriarr function).
- The number of criteria must be the same number of columns in mtx.
- The weight array must be None or an iterable with the same length of the criteria.

Parameters **mtx** : 2D array-like

2D alternative matrix, where every column (axis 0) are a criteria, and every row (axis 1) is an alternative.

criteria : Array-like

The sense of optimality of every criteria. Must has only MIN (-1) and MAX (1) values. Must has the same elements as columns has mtx

weights : array like or None

The importance of every criteria. Must has the same elements as columns has `mtx` or None.

Returns **mtx** : numpy.ndarray

mtx representations as 2d numpy.ndarray.

criteria : numpy.ndarray

A criteria as numpy.ndarray.

weights : numpy.ndarray or None

A weights as numpy.ndarray or None (if weights is None).

Raises **DataValidationError :**

If the data are incompatible.

5.3.3 `skcriteria.plot` package

Plotting utilities

class skcriteria.plot.**DataPlotMethods**(*data*)

Bases: object

Data plotting accessor and method

Examples

```
>>> data.plot()
>>> data.plot.hist()
>>> data.plot.scatter('x', 'y')
>>> data.plot.radar()
```

These plotting methods can also be accessed by calling the accessor as a method with the `kind` argument: `data.plot(kind='violin')` is equivalent to `data.plot.violin()`

Methods

`__call__`([kind])	Make plots of Data using matplotlib / pylab.
`bars`(**kwargs)	
`box`(**kwargs)	
`hist`(**kwargs)	
`plot`(func[, mnorm, wnorm, anames, cnames, ...])	Preprocess the data and send to the plot function *func*.
`preprocess`(data, mnorm, wnorm, anames, ...)	Preprocess the data to be plotted.
`radar`(**kwargs)	Creates a radar chart, also known as a spider or star chart [R25].
`scatter`(**kwargs)	
`to_str`()	
`violin`(**kwargs)	

plot (*func*, *mnorm=u'none'*, *wnorm=u'none'*, *anames=None*, *cnames=None*, *cmap=None*, *weighted=True*, *show_criteria=True*, *min2max=False*, *push_negatives=False*, *addepsto0=False*, ***kwargs*)

Preprocess the data and send to the plot function *func*.

Parameters **func** : callable

The function that make the plot. The return value of func are the recutn value of this method.

mnorm: string, callable, optional (default="none")

Normalization method for the alternative matrix.

wnorm : string, callable, optional (default="none")

Normalization method for the weights array.

anames : list of str or None, optional (default=None)

The list of alternative names to be render in the plot. If is None then the alternative names of data are used.

cnames : list of str or None, optional (default=None)

The list of criteria names to be render in the plot. If is None then the criteria names of data are used.

cmap : string or None, optional (default=None)

Name of the color map to be used *[R20]*

weighted : bool, optional (default=True)

If the data must be weighted before redering.

show_criteria : bool, optional (default=True)

I the sense of optimality must be rendered in the plot.

min2max : bool, optional (default=False)

If true all the data of the minimization criteria are inverted before render.

push_negatives : bool, optional (default=False)

If True all the criterias with some value < 0 are incremented to be at least 0 in the minimun value.

addepsto0 : bool, optional (default=False)

If true add an small value to all the zeros inside the data.

kwargs :

Arguments to send to *func*

Returns The return value of *func*.

Notes

All the plot methods of Scikit-Criteria returns a matplotlib axis.

References

[R20]

preprocess(*data*, *mnorm*, *wnorm*, *anames*, *cnames*, *cmap*, *weighted*, *show_criteria*, *min2max*, *push_negatives*, *addepsto0*)

Preprocess the data to be plotted.

Parameters **data** : skcritria.core.Data

The data to be preprocessed.

mnorm: string, callable

Normalization method for the alternative matrix.

wnorm : string, callable

Normalization method for the weights array.

anames : list of str or None

The list of alternative names to be render in the plot. If is None then the alternative names of data are used.

cnames : list of str or None

The list of criteria names to be render in the plot. If is None then the criteria names of data are used.

cmap : string or None

Name of the color map to be used *[R21]*

weighted : bool

If the data must be weighted before redering.

show_criteria : bool

I the sense of optimality must be rendered in the plot.

min2max : bool

If true all the data of the minimization criteria are inverted before render.

push_negatives : bool

If True all the criterias with some value < 0 are incremented to be at least 0 in the minimun value.

addepsto0 : bool

If true add an small value to all the zeros inside the data.

Returns **preprocessed_data** : dict

All the data ready to be sended to a plot function

References

[R21]

radar (***kwargs*)

Creates a radar chart, also known as a spider or star chart *[R22]*.

A radar chart is a graphical method of displaying multivariate data in the form of a two-dimensional chart of three or more quantitative variables represented on axes starting from the same point. The relative position and angle of the axes is typically uninformative.

Parameters **frame** : {"polygon", "circle"}

Shape of frame surrounding axes.

ax : None or PolarAxes, optional (default=None)

Axis where the radar must be redered. Is is None a new axis are created.

legendcol : int, optional (default=5)

How many columns must has the legend.

subplots_kwargs : dict or None, optional (default=None)

Argument to send to `matplotlib.pyplot.subplots` if axis is None. If axis is not None, subplots_kwargs are ignored.

Returns **ax** : matplotlib.projections.polar.PolarAxes

Axis where the radar are rendered

See also:

DataPlotMethods.plot To check all the available parameters

Notes

All the parameters in `plot()` are supported; but by default this method override some default values:

- `show_criteria=False`
- `min2max=True`
- `push_negatices=True`
- `addepsto0=True`

References

[R22]

5.3.4 `skcriteria.madm` package

This package contains several implementations of Multi criteria decision analysys methods (MADM) methods.

Conflicting criteria are typical in evaluating options: cost or price is usually one of the main criteria, and some measure of quality is typically another criterion, easily in conflict with the cost. In purchasing a car, cost, comfort, safety, and fuel economy may be some of the main criteria we consider – it is unusual that the cheapest car is the most comfortable and the safest one. In portfolio management, we are interested in getting high returns but at the same time reducing our risks, but the stocks that have the potential of bringing high returns typically also carry high risks of losing money. In a service industry, customer satisfaction and the cost of providing service are fundamental conflicting criteria.

Modules:

skcriteria.madm.simple module

Simplests method of multi-criteria

class skcriteria.madm.simple.**WeightedSum**(*mnorm=u'sum'*, *wnorm=u'sum'*)

Bases: skcriteria.madm._dmaker.DecisionMaker

The weighted sum model (WSM) is the best known and simplest multi-criteria decision analysis for evaluating a number of alternatives in terms of a number of decision criteria. It is very important to state here that it is applicable only when all the data are expressed in exactly the same unit. If this is not the case, then the final result is equivalent to "adding apples and oranges." To avoid this problem a previous normalization step is necesary.

In general, suppose that a given MCDA problem is defined on m alternatives and n decision criteria. Furthermore, let us assume that all the criteria are benefit criteria, that is, the higher the values are, the better it is. Next suppose that w_j denotes the relative weight of importance of the criterion C_j and a_{ij} is the performance value of alternative A_i when it is evaluated in terms of criterion C_j. Then, the total (i.e., when all the criteria are considered simultaneously) importance of alternative A_i, denoted as $A_i^{WSM-score}$, is defined as follows:

$$A_i^{WSM-score} = \sum_{j=1}^{n} w_j a_{ij}, \ for\ i = 1, 2, 3, ..., m$$

For the maximization case, the best alternative is the one that yields the maximum total performance value.

Parameters **mnorm** : string, callable, optional (default="sum")

Normalization method for the alternative matrix.

wnorm : string, callable, optional (default="sum")

Normalization method for the weights array.

Returns **Decision** : skcriteria.madm.Decision

With values:

- **kernel_**: None
- **rank_**: A ranking (start at 1) where the i-nth element represent the position of the i-nth alternative.
- **best_alternative_**: The index of the best alternative.
- **alpha_solution_**: True
- **beta_solution_**: False
- **gamma_solution_**: True
- **e_**: Particular data created by this method.
  - **e_.points**: Array where the i-nth element represent the importance of the i-nth alternative.

Notes

If some criteria is for minimization, this implementation calculates the inverse.

References

[R12], *[R13]*, *[R14]*

Attributes

mnorm	Normalization function for the alternative matrix.
wnorm	Normalization function for the weights vector.

Methods

as_dict()	Create a simply dict representation of the object.
decide(data[, criteria, weights])	Execute the Solver over the given data.
make_result(data, kernel, rank, extra)	Create a new skcriteria.madm.Decision
preprocess(data)	Normalize the alternative matrix and weight vector.
solve(**kwargs)	Execute the multi-criteria method.

solve (***kwargs*)
Execute the multi-criteria method.

Parameters **data** : skcriteria.Data

Preprocessed Data.

Returns object

object or tuple of objects with the raw result data.

class skcriteria.madm.simple.**WeightedProduct** (*mnorm=u'sum'*, *wnorm=u'sum'*)
Bases: skcriteria.madm._dmaker.DecisionMaker

The weighted product model (WPM) is a popular multi-criteria decision analysis method. It is similar to the weighted sum model. The main difference is that instead of addition in the main mathematical operation now there is multiplication.

In general, suppose that a given MCDA problem is defined on m alternatives and n decision criteria. Furthermore, let us assume that all the criteria are benefit criteria, that is, the higher the values are, the better it is. Next suppose that w_j denotes the relative weight of importance of the criterion C_j and a_{ij} is the performance value of alternative A_i when it is evaluated in terms of criterion C_j. Then, the total (i.e., when all the criteria are considered simultaneously) importance of alternative A_i, denoted as $A_i^{WPM-score}$, is defined as follows:

$$A_i^{WPM-score} = \prod_{j=1}^{n} a_{ij}^{w_j}, \ for\ i = 1, 2, 3, ..., m$$

To avoid underflow, instead the multiplication of the values we add the logarithms of the values; so $A_i^{WPM-score}$, is finally defined as:

$$A_i^{WPM-score} = \sum_{j=1}^{n} w_j \log(a_{ij}), for\ i = 1, 2, 3, ..., m$$

For the maximization case, the best alternative is the one that yields the maximum total performance value.

Parameters **mnorm** : string, callable, optional (default="sum")

Normalization method for the alternative matrix.

wnorm : string, callable, optional (default="sum")

Normalization method for the weights array.

Returns **Decision** : skcriteria.madm.Decision

With values:

- **kernel_**: None
- **rank_**: A ranking (start at 1) where the i-nth element represent the position of the i-nth alternative.
- **best_alternative_**: The index of the best alternative.
- **alpha_solution_**: True
- **beta_solution_**: False
- **gamma_solution_**: True
- **e_**: Particular data created by this method.
  - **e_.points**: Array where the i-nth element represent the importance of the i-nth alternative.

Notes

The implementation works as follow:

- If we have some values of any criteria < 0 in the alternative-matrix we add the minimimun value of this criteria to all the criteria.
- If we have some 0 in some criteria all the criteria is incremented by 1.
- If some criteria is for minimization, this implementation calculates the inverse.
- Instead the multiplication of the values we add the logarithms of the values to avoid underflow.

References

[R15], *[R16]*, *[R17]*

Attributes

`mnorm`	Normalization function for the alternative matrix.
`wnorm`	Normalization function for the weights vector.

Methods

`as_dict`()	Create a simply `dict` representation of the object.
`decide`(data[, criteria, weights])	Execute the Solver over the given data.
`make_result`(data, kernel, rank, extra)	Create a new `skcriteria.madm.Decision`
`preprocess`(**kwargs)	Normalize the alternative matrix and weight vector.
`solve`(**kwargs)	Execute the multi-criteria method.

preprocess(***kwargs*)

Normalize the alternative matrix and weight vector.

Creates a new instance of data by aplying the normalization function to the alternative matrix and the weights vector containded inside the given data.

Parameters **data** : `skcriteria.Data`

A data to be Preprocessed

Returns `skcriteria.Data`

A new instance of data with the `mtx` attributes normalized with `mnorm` and `weights` normalized with wnorm. `anames` and `cnames` are preseved

solve(***kwargs*)

Execute the multi-criteria method.

Parameters **data** : `skcriteria.Data`

Preprocessed Data.

Returns `object`

object or tuple of objects with the raw result data.

skcriteria.madm.closeness module

Methods based on an aggregating function representing "closeness to the ideal".

class skcriteria.madm.closeness.**TOPSIS**(*mnorm=u'vector'*, *wnorm=u'sum'*)

Bases: `skcriteria.madm._dmaker.DecisionMaker`

TOPSIS is based on the concept that the chosen alternative should have the shortest geometric distance from the ideal solution and the longest euclidean distance from the worst solution.

An assumption of TOPSIS is that the criteria are monotonically increasing or decreasing, and also allow trade-offs between criteria, where a poor result in one criterion can be negated by a good result in another criterion.

Parameters **mnorm** : string, callable, optional (default="vector")

Normalization method for the alternative matrix.

wnorm : string, callable, optional (default="sum")

Normalization method for the weights array.

Returns **Decision** : `skcriteria.madm.Decision`

With values:

- **kernel_**: None
- **rank_**: A ranking (start at 1) where the i-nth element represent the position of the i-nth alternative.
- **best_alternative_**: The index of the best alternative.
- **alpha_solution_**: True
- **beta_solution_**: False
- **gamma_solution_**: True
- **e_**: Particular data created by this method.
  - **e_.closeness**: Array where the i-nth element represent the closenees of the i-nth alternative to ideal and worst solution.

References

[R1], *[R2]*, *[R3]*

Attributes

`mnorm`	Normalization function for the alternative matrix.
`wnorm`	Normalization function for the weights vector.

Methods

`as_dict`()	Create a simply `dict` representation of the object.
`decide`(data[, criteria, weights])	Execute the Solver over the given data.
`make_result`(data, kernel, rank, extra)	Create a new `skcriteria.madm.Decision`
`preprocess`(data)	Normalize the alternative matrix and weight vector.
`solve`(**kwargs)	Execute the multi-criteria method.

solve (***kwargs*)

Execute the multi-criteria method.

Parameters **data** : `skcriteria.Data`

Preprocessed Data.

Returns `object`

object or tuple of objects with the raw result data.

skcriteria.madm.moora module

Implementation of a family of Multi-objective optimization on the basis of ratio analysis (MOORA) methods.

class skcriteria.madm.moora.**RatioMOORA**(*wnorm=u'sum'*)

Bases: `skcriteria.madm._dmaker.DecisionMaker`

The method refers to a matrix of responses of alternatives to objectives, to which ratios are applied.

In MOORA the set of ratios (by default) has the square roots of the sum of squared responses as denominators.

$$\overline{X}_{ij} = \frac{X_{ij}}{\sqrt{\sum_{j=1}^{m} X_{ij}^2}}$$

These ratios, as dimensionless, seem to be the best choice among different ratios. These dimensionless ratios, situated between zero and one, are added in the case of maximization or subtracted in case of minimization:

$$Ny_i = \sum_{i=1}^{g} Nx_{ij} - \sum_{i=1}^{g+1} Nx_{ij}$$

with: $i = 1, 2, ..., g$ for the objectives to be maximized, $i = g+1, g+2, ..., n$ for the objectives to be minimized.

Finally, all alternatives are ranked, according to the obtained ratios.

Parameters **wnorm** : string, callable, optional (default="sum")

Normalization method for the weights array.

Returns **Decision** : `skcriteria.madm.Decision`

With values:

- **kernel_**: None
- **rank_**: A ranking (start at 1) where the i-nth element represent the position of the i-nth alternative.
- **best_alternative_**: The index of the best alternative.
- **alpha_solution_**: True
- **beta_solution_**: False
- **gamma_solution_**: True
- **e_**: Particular data created by this method.
  - **e_.points**: Array where the i-nth element represent the importance of the i-nth alternative.

References

[R7]

Attributes

`mnorm`	Normalization function for the alternative matrix.
`wnorm`	Normalization function for the weights vector.

Methods

`as_dict`(**kwargs)	Create a simply `dict` representation of the object.
`decide`(data[, criteria, weights])	Execute the Solver over the given data.
`make_result`(data, kernel, rank, extra)	Create a new `skcriteria.madm.Decision`
`preprocess`(data)	Normalize the alternative matrix and weight vector.
`solve`(**kwargs)	Execute the multi-criteria method.

as_dict (***kwargs*)
Create a simply `dict` representation of the object.

Notes

```
x.as_dict != dict(x)
```

solve (***kwargs*)
Execute the multi-criteria method.

Parameters **data** : `skcriteria.Data`

Preprocessed Data.

Returns `object`

object or tuple of objects with the raw result data.

class `skcriteria.madm.moora.`**RefPointMOORA**(*wnorm=u'sum'*)

Bases: `skcriteria.madm._dmaker.DecisionMaker`

Rank the alternatives from a reference point selected with the Min-Max Metric of Tchebycheff.

$$\min_j\{\max_i |r_i - x^*_{ij}|\}$$

This reference point theory starts from the already normalized ratios as defined in the MOORA method, namely formula:

$$\overline{X}_{ij} = \frac{X_{ij}}{\sqrt{\sum_{j=1}^{m} X_{ij}^2}}$$

Preference is given to a reference point possessing as co-ordinates the dominating co-ordinates per attribute of the candidate alternatives and which is designated as the *Maximal Objective Reference Point*. This approach is called realistic and non-subjective as the co-ordinates, which are selected for the reference point, are realized in one of the candidate alternatives.

Parameters **wnorm** : string, callable, optional (default="sum")

Normalization method for the weights array.

Returns **Decision** : `skcriteria.madm.Decision`

With values:

- **kernel_**: None
- **rank_**: A ranking (start at 1) where the i-nth element represent the position of the i-nth alternative.
- **best_alternative_**: The index of the best alternative.
- **alpha_solution_**: True
- **beta_solution_**: False
- **gamma_solution_**: True
- **e_**: Particular data created by this method.
  - **e_.points**: array where the i-nth element represent the closenees of the i-nth alternative to a reference point based on the *Min-Max Metric of Tchebycheff*.

References

[R8], *[R9]*

Attributes

`mnorm`	Normalization function for the alternative matrix.
`wnorm`	Normalization function for the weights vector.

Methods

as_dict(**kwargs)	Create a simply dict representation of the object.
decide(data[, criteria, weights])	Execute the Solver over the given data.
make_result(data, kernel, rank, extra)	Create a new skcriteria.madm.Decision
preprocess(data)	Normalize the alternative matrix and weight vector.
solve(**kwargs)	Execute the multi-criteria method.

as_dict (***kwargs*)
Create a simply dict representation of the object.

Notes

```
x.as_dict != dict(x)
```

solve (***kwargs*)
Execute the multi-criteria method.

Parameters **data** : skcriteria.Data

Preprocessed Data.

Returns object

object or tuple of objects with the raw result data.

class skcriteria.madm.moora.**FMFMOORA**(*wnorm=u'sum'*)
Bases: skcriteria.madm._dmaker.DecisionMaker

Full Multiplicative Form, a method that is non-linear, non-additive, does not use weights and does not require normalization.

To combine a minimization and maximization of different criteria in the same problem all the method uses the formula:

$$U'_j = \frac{\prod_{g=1}^{i} x_{gi}}{\prod_{k=i+1}^{n} x_{kj}}$$

Where j = the number of alternatives; i = the number of objectives to be maximized; ni = the number of objectives to be minimize; and U'_j: the utility of alternative j with objectives to be maximized and objectives to be minimized.

To avoid underflow, instead the multiplication of the values we add the logarithms of the values; so U'_j:, is finally defined as:

$$U'_j = \sum_{g=1}^{i} \log(x_{gi}) - \sum_{k=i+1}^{n} \log(x_{kj})$$

Parameters **wnorm** : string, callable, optional (default="sum")

Normalization method for the weights array.

Returns **Decision** : skcriteria.madm.Decision

With values:

- **kernel_**: None

- **rank_**: A ranking (start at 1) where the i-nth element represent the position of the i-nth alternative.
- **best_alternative_**: The index of the best alternative.
- **alpha_solution_**: True
- **beta_solution_**: False
- **gamma_solution_**: True
- **e_**: Particular data created by this method.
  - **e_.points**: Array where the i-nth element represent the importance of the i-nth alternative.

Notes

The implementation works as follow:

- Before determine U_j the values are normalized by the ratio sugested by MOORA.

$$\overline{X}_{ij} = \frac{X_{ij}}{\sqrt{\sum_{j=1}^{m} X_{ij}^2}}$$

- If we have some values of any criteria < 0 in the alternative-matrix we add the minimimun value of this criteria to all the criteria.
- If we have some 0 in some criteria all the criteria is incremented by 1.
- If some criteria is for minimization, this implementation calculates the inverse.
- Instead the multiplication of the values we add the logarithms of the values to avoid underflow.

References

[R10]

Attributes

`mnorm`	Normalization function for the alternative matrix.
`wnorm`	Normalization function for the weights vector.

Methods

`as_dict`(**kwargs)	Create a simply `dict` representation of the object.
`decide`(data[, criteria, weights])	Execute the Solver over the given data.
`make_result`(data, kernel, rank, extra)	Create a new `skcriteria.madm.Decision`
`preprocess`(**kwargs)	Normalize the alternative matrix and weight vector.
`solve`(**kwargs)	Execute the multi-criteria method.

as_dict(***kwargs*)
Create a simply `dict` representation of the object.

Notes

```
x.as_dict != dict(x)
```

preprocess(***kwargs*)
Normalize the alternative matrix and weight vector.

Creates a new instance of data by aplying the normalization function to the alternative matrix and the weights vector containded inside the given data.

Parameters **data** : `skcriteria.Data`

A data to be Preprocessed

Returns `skcriteria.Data`

A new instance of data with the `mtx` attributes normalized with `mnorm` and `weights` normalized with wnorm. `anames` and `cnames` are preseved

solve(***kwargs*)
Execute the multi-criteria method.

Parameters **data** : `skcriteria.Data`

Preprocessed Data.

Returns `object`

object or tuple of objects with the raw result data.

class skcriteria.madm.moora.**MultiMOORA**
Bases: `skcriteria.madm._dmaker.DecisionMaker`

MULTIMOORA is compose the ranking resulting of aplyting the methods, RatioMOORA, RefPointMOORA and FMFMOORA.

These three methods represent all possible methods with dimensionless measures in multi-objective optimization and one can not argue that one method is better than or is of more importance than the others; so for determining the final ranking the implementation maximizes how many times an alternative *i* dominates and alternative *j*.

Returns **Decision** : `skcriteria.madm.Decision`

With values:

- **kernel_**: None
- **rank_**: A ranking (start at 1) where the i-nth element represent the position of the i-nth alternative.
- **best_alternative_**: The index of the best alternative.
- **alpha_solution_**: True
- **beta_solution_**: False
- **gamma_solution_**: True
- **e_**: Particular data created by this method.
  - **e_.rank_mtx**: 2x3 Array where the first column is the RatioMOORA ranking, the second one the RefPointMOORA ranking and the last the FMFMOORA ranking.

Notes

The implementation works as follow:

- Before determine U_j the values are normalized by the ratio sugested by MOORA.

$$\overline{X}_{ij} = \frac{X_{ij}}{\sqrt{\sum_{j=1}^{m} X_{ij}^2}}$$

- If we have some values of any criteria < 0 in the alternative-matrix we add the minimimun value of this criteria to all the criteria.
- If we have some 0 in some criteria all the criteria is incremented by 1.
- If some criteria is for minimization, this implementation calculates the inverse.
- Instead the multiplication of the values we add the logarithms of the values to avoid underflow.
- For determining the final ranking the implementation maximizes how many times an alternative *i* dominates and alternative *j*.

References

[R11]

Attributes

`mnorm`	Normalization function for the alternative matrix.
`wnorm`	Normalization function for the weights vector.

Methods

`as_dict`(**kwargs)	Create a simply `dict` representation of the object.
`decide`(data[, criteria, weights])	Execute the Solver over the given data.
`make_result`(data, kernel, rank, extra)	Create a new `skcriteria.madm.Decision`
`preprocess`(**kwargs)	Normalize the alternative matrix and weight vector.
`solve`(**kwargs)	Execute the multi-criteria method.

as_dict(***kwargs*)

Create a simply `dict` representation of the object.

Notes

```
x.as_dict != dict(x)
```

preprocess(***kwargs*)

Normalize the alternative matrix and weight vector.

Creates a new instance of data by aplying the normalization function to the alternative matrix and the weights vector containded inside the given data.

Parameters **data** : `skcriteria.Data`

A data to be Preprocessed

Returns `skcriteria.Data`

A new instance of data with the `mtx` attributes normalized with `mnorm` and `weights` normalized with wnorm. `anames` and `cnames` are preseved

solve(***kwargs*)

Execute the multi-criteria method.

Parameters **data** : `skcriteria.Data`

Preprocessed Data.

Returns `object`

object or tuple of objects with the raw result data.

skcriteria.madm.electre.py module

ELECTRE is a family of multi-criteria decision analysis methods that originated in Europe in the mid-1960s. The acronym ELECTRE stands for: ELimination Et Choix Traduisant la REalité (ELimination and Choice Expressing REality).

Usually the Electre Methods are used to discard some alternatives to the problem, which are unacceptable. After that we can use another MCDA to select the best one. The Advantage of using the Electre Methods before is that we can apply another MCDA with a restricted set of alternatives saving much time.

class skcriteria.madm.electre.**ELECTRE1**(*p=0.65*, *q=0.35*, *mnorm=u'sum'*, *wnorm=u'sum'*, *njobs=None*)

Bases: `skcriteria.madm._dmaker.DecisionMaker`

The ELECTRE I model find the kernel solution in a situation where true criteria and restricted outranking relations are given.

That is, ELECTRE I cannot derive the ranking of alternatives but the kernel set. In ELECTRE I, two indices called the concordance index and the discordance index are used to measure the relations between objects.

Parameters **p** : float, optional (default=0.65)

Concordance threshold. Threshold of how much one alternative is at least as good as another to be significative.

q : float, optional (default=0.35)

Discordance threshold. Threshold of how much the degree one alternative is strictly preferred to another to be significative.

mnorm : string, callable, optional (default="sum")

Normalization method for the alternative matrix.

wnorm : string, callable, optional (default="sum")

Normalization method for the weights array.

njobs : int, default=None

How many cores to use to solve the linear programs and the second method. By default all the availables cores are used.

Returns **Decision** : `skcriteria.madm.Decision`

With values:

- **kernel_**: Array with the indexes of the alternatives in he kernel.
- **rank_**: None
- **best_alternative_**: None
- **alpha_solution_**: False
- **beta_solution_**: True
- **gamma_solution_**: False
- **e_**: Particular data created by this method.
  - **e_.closeness**: Array where the i-nth element represent the closenees of the i-nth alternative to ideal and worst solution.
  - **e_.outrank**: numpy.ndarray of bool The outranking matrix of superation. If the element[i][j] is True The alternative `i` outrank the alternative `j`.
  - **e_.mtx_concordance**: numpy.ndarray The concordance indexes matrix where the element[i][j] measures how much the alternative `i` is at least as good as `j`.
  - **e_.mtx_discordance**: numpy.ndarray The discordance indexes matrix where the element[i][j] measures the degree to which the alternative `i` is strictly preferred to `j`.
  - **e_.p**: float Concordance index threshold.
  - **e_.q**: float Discordance index threshold.

References

[R4], *[R5]*, *[R6]*

Attributes

`mnorm`	Normalization function for the alternative matrix.
`njobs`	How many cores to use to solve the linear programs and the second method.
`p`	Concordance threshold.
`q`	Discordance threshold.
`wnorm`	Normalization function for the weights vector.

Methods

`as_dict`(**kwargs)	Create a simply `dict` representation of the object.
`decide`(data[, criteria, weights])	Execute the Solver over the given data.
`make_result`(data, kernel, rank, extra)	Create a new `skcriteria.madm.Decision`
`preprocess`(data)	Normalize the alternative matrix and weight vector.
`solve`(**kwargs)	Execute the multi-criteria method.

as_dict(***kwargs*)

Create a simply `dict` representation of the object.

Notes

```
x.as_dict != dict(x)
```

njobs
How many cores to use to solve the linear programs and the second method. By default all the availables cores are used.

p
Concordance threshold. Threshold of how much one alternative is at least as good as another to be significative.

q
Discordance threshold. Threshold of how much the degree one alternative is strictly preferred to another to be significative.

solve(***kwargs*)
Execute the multi-criteria method.

Parameters **data** : `skcriteria.Data`

Preprocessed Data.

Returns `object`

object or tuple of objects with the raw result data.

skcriteria.madm.simus module

SIMUS (Sequential Interactive Model for Urban Systems) Method

class skcriteria.madm.simus.**SIMUS**(*mnorm=u'none'*, *wnorm=u'none'*, *rank_by=1*, *solver=u'pulp'*, *njobs=None*)
Bases: `skcriteria.madm._dmaker.DecisionMaker`

SIMUS (Sequential Interactive Model for Urban Systems) developed by Nolberto Munier (2011) is a tool to aid decision-making problems with multiple objectives. The method solves successive scenarios formulated as linear programs. For each scenario, the decision-maker must choose the criterion to be considered objective while the remaining restrictions constitute the constrains system that the projects are subject to. In each case, if there is a feasible solution that is optimum, it is recorded in a matrix of efficient results. Then, from this matrix two rankings allow the decision maker to compare results obtained by different procedures. The first ranking is obtained through a linear weighting of each column by a factor - equivalent of establishing a weight - and that measures the participation of the corresponding project. In the second ranking, the method uses dominance and subordinate relationships between projects, concepts from the French school of MCDM.

Parameters **mnorm** : string, callable, optional (default="none")

Normalization method for the alternative matrix.

wnorm : string, callable, optional (default="none")

Normalization method for the weights array.

rank_by : 1 or 2 (default=1)

Wich of the two methods are used to calculate the ranking. The two methods are executed always.

solver : str, default="pulp"

Which solver to use to solve the undelying linear programs. The full list are available in *skcriteria.utils.lp.SOLVERS*

njobs : int, default=None

How many cores to use to solve the linear programs and the second method. By default all the availables cores are used.

Returns **Decision** : `skcriteria.madm.Decision`

With values:

- **kernel_**: None
- **rank_**: A ranking (start at 1) where the i-nth element represent the position of the i-nth alternative.
- **best_alternative_**: The index of the best alternative.
- **alpha_solution_**: True
- **beta_solution_**: False
- **gamma_solution_**: True
- **e_**: Particular data created by this method.
  - **rank_by**: 1 or 2. Wich of the two methods are used to calculate the ranking. Esentialy if the rank is calculated with `e_.points1` or `e_points2`
  - **solver**: With solver was used for the underlying linear problems.
  - **stages**: The underlying linear problems.
  - **stage_results**: The values of the variables of the linear problems as a n-dimensional array. When th *n-th* row represent the result values of the variables for the *n-th* stage.
  - **points1**: The points of every alternative obtained by the first method.
  - **points2**: The points of every alternative obtained by the first method.
  - **tita_j_p**: 2nd. method domination.
  - **tita_j_d**: 2nd. method subordination.
  - **doms**: Total dominance matrix of the 2nd. method.
  - **dom_by_crit**: Dominance by criteria of the 2nd method.

References

[R18], *[R19]*

Attributes

`mnorm`	Normalization function for the alternative matrix.
`njobs`	How many cores to use to solve the linear programs and the second method.
`solver`	Which solver to use to solve the undelying linear programs.
`wnorm`	Normalization function for the weights vector.

Methods

as_dict()	Create a simply dict representation of the object.
decide(data[, criteria, weights])	Execute the Solver over the given data.
make_result(data, kernel, rank, extra)	Create a new skcriteria.madm.Decision
preprocess(data)	Normalize the alternative matrix and weight vector.
solve(**kwargs)	Execute the multi-criteria method.

njobs
How many cores to use to solve the linear programs and the second method. By default all the availables cores are used.

solve (***kwargs*)
Execute the multi-criteria method.

Parameters **data** : skcriteria.Data

Preprocessed Data.

Returns object

object or tuple of objects with the raw result data.

solver
Which solver to use to solve the undelying linear programs. The full list are available in *skcriteria.utils.lp.SOLVERS*

5.3.5 skcriteria.weights package

This package contains utilities to make some treatments to weights

Modules:

skcriteria.weights.equal module

skcriteria.weights.divergence module

skcriteria.weights.critic module

5.4 Indices and tables

- genindex
- modindex
- search

Bibliography

[R20] https://matplotlib.org/users/colormaps.html

[R21] https://matplotlib.org/users/colormaps.html

[R22] http://en.wikipedia.org/wiki/Radar_chart

[R12] Fishburn, P. C. (1967). Letter to the editor—additive utilities with incomplete product sets: application to priorities and assignments. Operations Research, 15(3), 537-542.

[R13] Weighted sum model. In Wikipedia, The Free Encyclopedia. Retrieved from https://en.wikipedia.org/wiki/Weighted_sum_model

[R14] Tzeng, G. H., & Huang, J. J. (2011). Multiple attribute decision making: methods and applications. CRC press.

[R15] Bridgman, P.W. (1922). Dimensional Analysis. New Haven, CT, U.S.A.: Yale University Press.

[R16] Miller, D.W.; M.K. Starr (1969). Executive Decisions and Operations Research. Englewood Cliffs, NJ, U.S.A.: Prentice-Hall, Inc.

[R17] Wen, Y. (2007, September 16). Using log-transform to avoid underflow problem in computing posterior probabilities. from http://web.mit.edu/wenyang/www/log_transform_for_underflow.pdf

[R1] Yoon, K., & Hwang, C. L. (1981). Multiple attribute decision making: methods and applications. SPRINGER-VERLAG BERLIN AN.

[R2] TOPSIS. In Wikipedia, The Free Encyclopedia. Retrieved from https://en.wikipedia.org/wiki/TOPSIS

[R3] Tzeng, G. H., & Huang, J. J. (2011). Multiple attribute decision making: methods and applications. CRC press.

[R7] BRAUERS, W. K.; ZAVADSKAS, Edmundas Kazimieras. The MOORA method and its application to privatization in a transition economy. Control and Cybernetics, 2006, vol. 35, p. 445-469.‘

[R8] Brauers, W. K. M., & Zavadskas, E. K. (2012). Robustness of MULTIMOORA: a method for multi-objective optimization. Informatica, 23(1), 1-25.

[R9] Karlin, S., & Studden, W. J. (1966). Tchebycheff systems: With applications in analysis and statistics. New York: Interscience.

[R10] Brauers, W. K. M., & Zavadskas, E. K. (2012). Robustness of MULTIMOORA: a method for multi-objective optimization. Informatica, 23(1), 1-25.

[R11] Brauers, W. K. M., & Zavadskas, E. K. (2012). Robustness of MULTIMOORA: a method for multi-objective optimization. Informatica, 23(1), 1-25.

[R4] Roy, B. (1990). The outranking approach and the foundations of ELECTRE methods. In Readings in multiple criteria decision aid (pp.155-183). Springer, Berlin, Heidelberg.

[R5] Roy, B. (1968). Classement et choix en présence de points de vue multiples. Revue française d'informatique et de recherche opérationnelle, 2(8), 57-75.

[R6] Tzeng, G. H., & Huang, J. J. (2011). Multiple attribute decision making: methods and applications. CRC press.

[R18] Munier, N. (2011). A strategy for using multicriteria analysis in decision-making: a guide for simple and complex environmental projects. Springer Science & Business Media.

[R19] Munier, N., Carignano, C., & Alberto, C. UN MÉTODO DE PROGRAMACIÓN MULTIOBJETIVO. Revista de la Escuela de Perfeccionamiento en Investigación Operativa, 24(39).

Python Module Index

S

Index

T

V

W