

---

# Scikit-Criteria Documentation

*Release 0.8.3*

**Juan BC**

**Jan 28, 2023**



## TUTORIALS

<b>1</b>	<b>Code Repository &amp; Issues</b>	<b>3</b>
<b>2</b>	<b>License</b>	<b>5</b>
<b>3</b>	<b>Citation</b>	<b>7</b>
<b>4</b>	<b>Contents</b>	<b>9</b>
	<b>Bibliography</b>	<b>101</b>
	<b>Python Module Index</b>	<b>103</b>
	<b>Index</b>	<b>105</b>





---

Ver. 0.8.3

---

**Warning:** If you are using Scikit-Criteria version  $\leq 0.2.11$  check the documentation here: <https://scikit-criteria.readthedocs.io/en/0.2/>

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

Our Google Groups mailing list is [here](#).

**You can contact me at:** [jbcabral@unc.edu.ar](mailto:jbcabral@unc.edu.ar) (if you have a support question, try the mailing list first)



## CODE REPOSITORY & ISSUES

<https://github.com/quatropo/scikit-criteria>





## 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.



## 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>



## CONTENTS

### 4.1 Installation

#### 4.1.1 Using conda

The easiest and fastest way to get the package up and running is to install `scikit-criteria` using `conda`:

```
$ conda install -c conda-forge scikit-criteria
```

or, better yet, using `mamba`, which is a super fast replacement for `conda`:

```
$ conda install -c conda-forge mamba  
$ mamba install -c conda-forge scikit-criteria
```

---

**Note:** We encourage users to use `conda` or `mamba` and the `conda-forge` packages for convenience, especially when developing on Windows. It is recommended to create a new environment.

---

If the installation fails for any reason, please open an issue in the [issue tracker](#).

#### 4.1.2 Alternative installation methods

You can also install `scikit-criteria` from PyPI using `pip`:

```
$ pip install scikit-criteria
```

Finally, you can also install the latest development version of `scikit-criteria` [directly from GitHub](#):

```
$ pip install git+https://github.com/quatropo/scikit-criteria/
```

This is useful if there is some feature that you want to try, but we did not release it yet as a stable version. Although you might find some unpolished details, these development installations should work without problems. If you find any, please open an issue in the [issue tracker](#).

**Warning:** It is recommended that you **never ever use sudo** with `distutils`, `pip`, `setuptools` and friends in Linux because you might seriously break your system [1] [2] [3] [4]. Use [virtual environments](#) instead.

### 4.1.3 If you don't have Python

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.

[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.

## 4.2 Tutorials

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

Contents:

### 4.2.1 Quick Start

This tutorial aims to explain in a simple way, how to create decision matrices, how to analyze them and how to evaluate them with some multi-criteria analysis methods (MCDA).

#### Conceptual overview

Multi-criteria data are complex. This is because at least two syntactically disconnected vectors are needed to describe a problem.

1. `matrix/A` choice set.
2. And the vector of criteria optimality sense `objectives/C`.

Additionally it can be accompanied by a vector `w/wj` with the weighting of the criteria.

To summarize all these data (and some extra ones), *Scikit-Criteria* provides a `DecisionMatrix` object along with a `mkdm()` utility function to facilitate the creation and validation of the data.

#### Your first `DecisionMatrix` object

First we need to import the the *Scikit-Criteria* module.

Then we need to create the `matrix` and `objectives` vectors.

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

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

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

The `objectives` vector must be a **1D array-like** with number of elements same as number of columns in the alternative matrix (`matrix`). Every component of the `objectives` vector represent the optimal sense of each criteria.

```
[3]: # let's says the first two alternatives are
# for maximization and the last one for minimization
objectives = [max, max, min]
objectives
```

```
[3]: [<function max>, <function max>, <function min>]
```

as you see the max and min are the built-in function for find max and mins in collections in python.

As you can see the function usage makes the code more readable. Also you can use as aliases of minimization and maximization the numpy function `np.min`, `np.max`, `np.amin`, `np.amax`, `np.nanmin`, `np.nanmax`; the strings "min", "minimize", "max", "maximize", ">", "<", "+", "-"; and the values -1 (minimize) and 1 (maximize).

Now we can combine this two vectors in our *Scikit-Criteria* decision matrix.

```
[4]: # we use the built-in function as aliases
dm = skc.mkdm(matrix, [min, max, min])
dm
```

```
[4]:      C0[ 1.0]  C1[ 1.0]  C2[ 1.0]
A0           1           2           3
A1           4           5           6
[2 Alternatives x 3 Criteria]
```

As you can see the output of the `DecisionMatrix` object is much more friendly than the plain python lists.

To change the generic names of the alternatives (A0 and A1) and the criteria (C0, C1 and C2); let's assume that our data is about cars (*car 0* and *car 1*) and their characteristics of evaluation are *autonomy* (*maximize*), *comfort* (*maximize*) and *price* (*minimize*).

To feed this information to our `DecisionMatrix` object we have the parameters: `alternatives` that accept the names of alternatives (must be the same number as the rows that `matrix` has), and `criteria` the criteria names (must have same number of elements with the columns that `matrix` has)

```
[5]: dm = skc.mkdm(
      matrix,
      objectives,
      alternatives=["car 0", "car 1"],
      criteria=["autonomy", "comfort", "price"],
      )
dm
```

```
[5]:      autonomy[ 1.0]  comfort[ 1.0]  price[ 1.0]
car 0                 1                 2                 3
car 1                 4                 5                 6
[2 Alternatives x 3 Criteria]
```

In our final step let's assume we know in our case, that the importance of the autonomy is the 50%, the comfort 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 elements as criterias on your alternative matrix (number of columns).

```
[6]: dm = skc.mkdm(
      matrix,
      objectives,
      weights=[0.5, 0.05, 0.45],
      alternatives=["car 0", "car 1"],
      criteria=["autonomy", "comfort", "price"],
```

(continues on next page)

(continued from previous page)

```
)
dm
[6]:      autonomy[ 0.50]  comfort[ 0.05]  price[ 0.45]
car 0           1           2           3
car 1           4           5           6
[2 Alternatives x 3 Criteria]
```

## Manipulating the Data

The data object are immutable, if you want to modify it you need create a new one. All the data are stored as [pandas dataframes](#) and [numpy arrays](#)

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

```
[7]: dm.matrix # note how this data ignores the objectives and the weights
[7]: Criteria      autonomy  comfort  price
Alternatives
car 0              1         2       3
car 1              4         5       6
```

```
[8]: dm.objectives
[8]: autonomy      MAX
comfort          MAX
price            MIN
Name: Objectives, dtype: object
```

```
[9]: dm.weights
[9]: autonomy      0.50
comfort          0.05
price            0.45
Name: Weights, dtype: float64
```

```
[10]: dm.alternatives, dm.criteria
[10]: (_ACArray(['car 0', 'car 1'], dtype=object),
      _ACArray(['autonomy', 'comfort', 'price'], dtype=object))
```

If you want (for example) change the names of the cars from `car 0` and `car 1`; to `VW` and `Ford` you must the copy method and provide the new names:

```
[11]: dm = dm.copy(alternatives=["VW", "Ford"])
dm
[11]:      autonomy[ 0.50]  comfort[ 0.05]  price[ 0.45]
VW           1           2           3
Ford         4           5           6
[2 Alternatives x 3 Criteria]
```

**Note:**



For more complex matiluations you can use the `dm.iloc[x]`, `dm.loc[x]` and `dm[x]` interface.

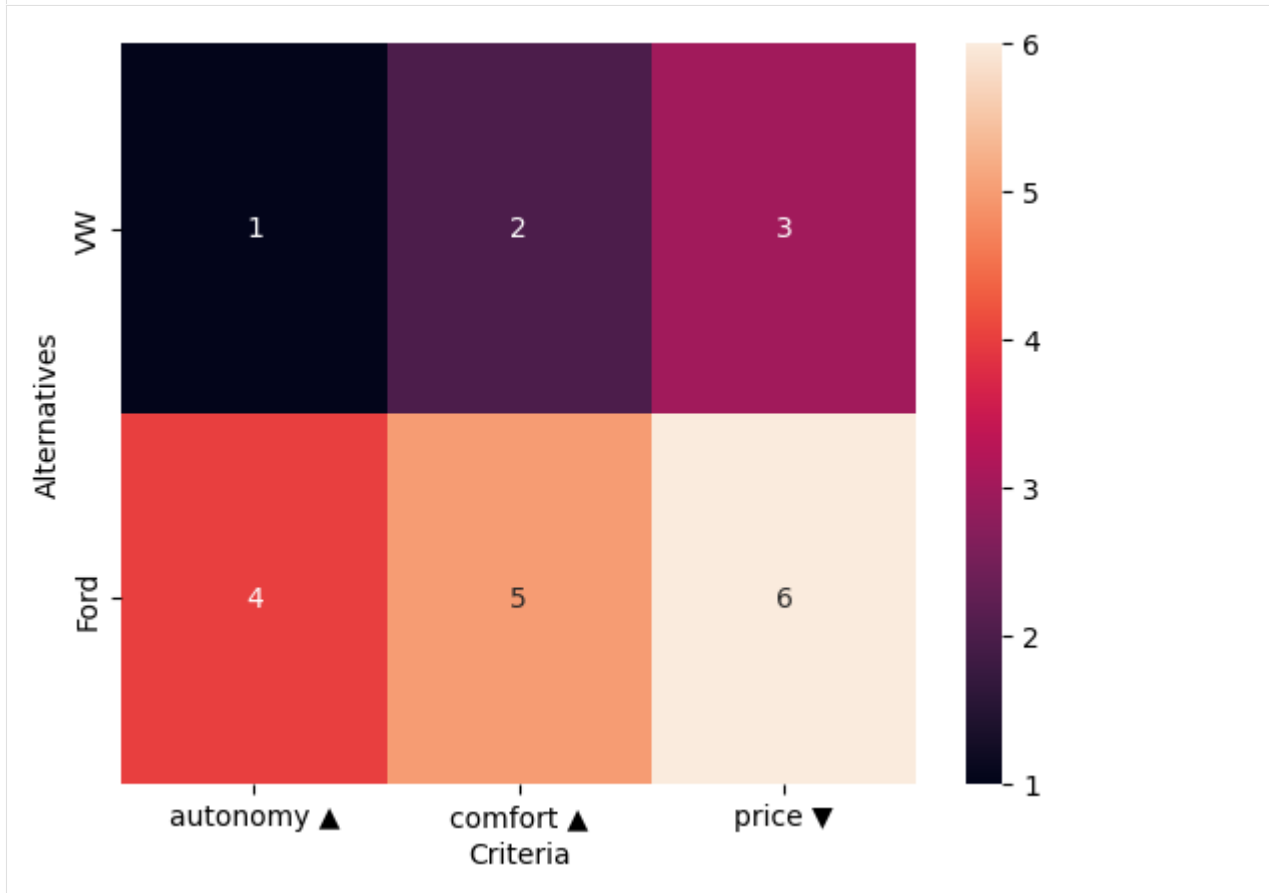
## Plotting

The Data structure support some basic routines for plotting.

The default scikit criteria uses the Heatmap plot to visualize all the data.

```
[12]: dm.plot()
```

```
[12]: <AxesSubplot:xlabel='Criteria', ylabel='Alternatives'>
```



In the same fashion you can plot the weights of the criteria

```
[13]: dm.plot.wheatmap()
```

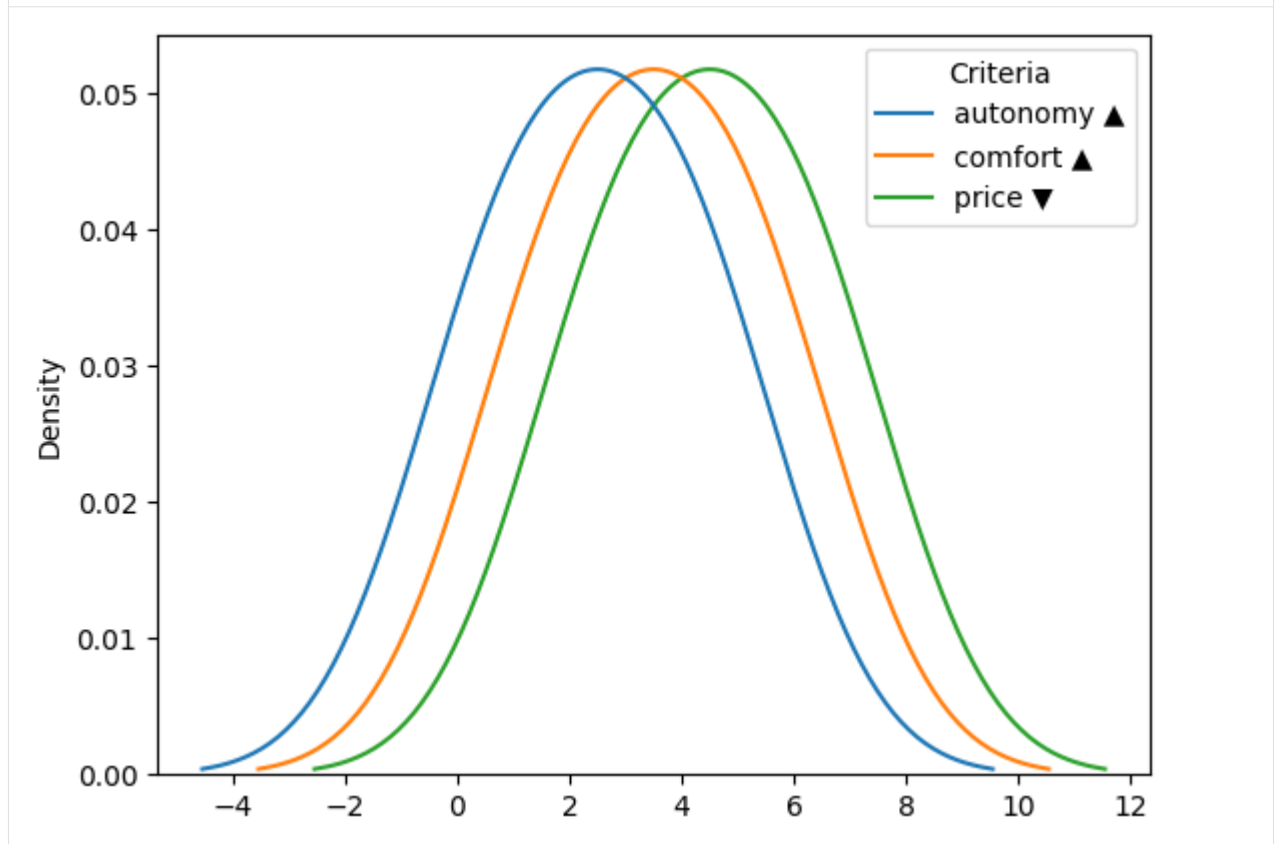
```
[13]: <AxesSubplot:xlabel='Criteria'>
```



You can accessing the different kind of plot by passing the name of the plot as first parameter of the method

```
[14]: dm.plot("kde")
```

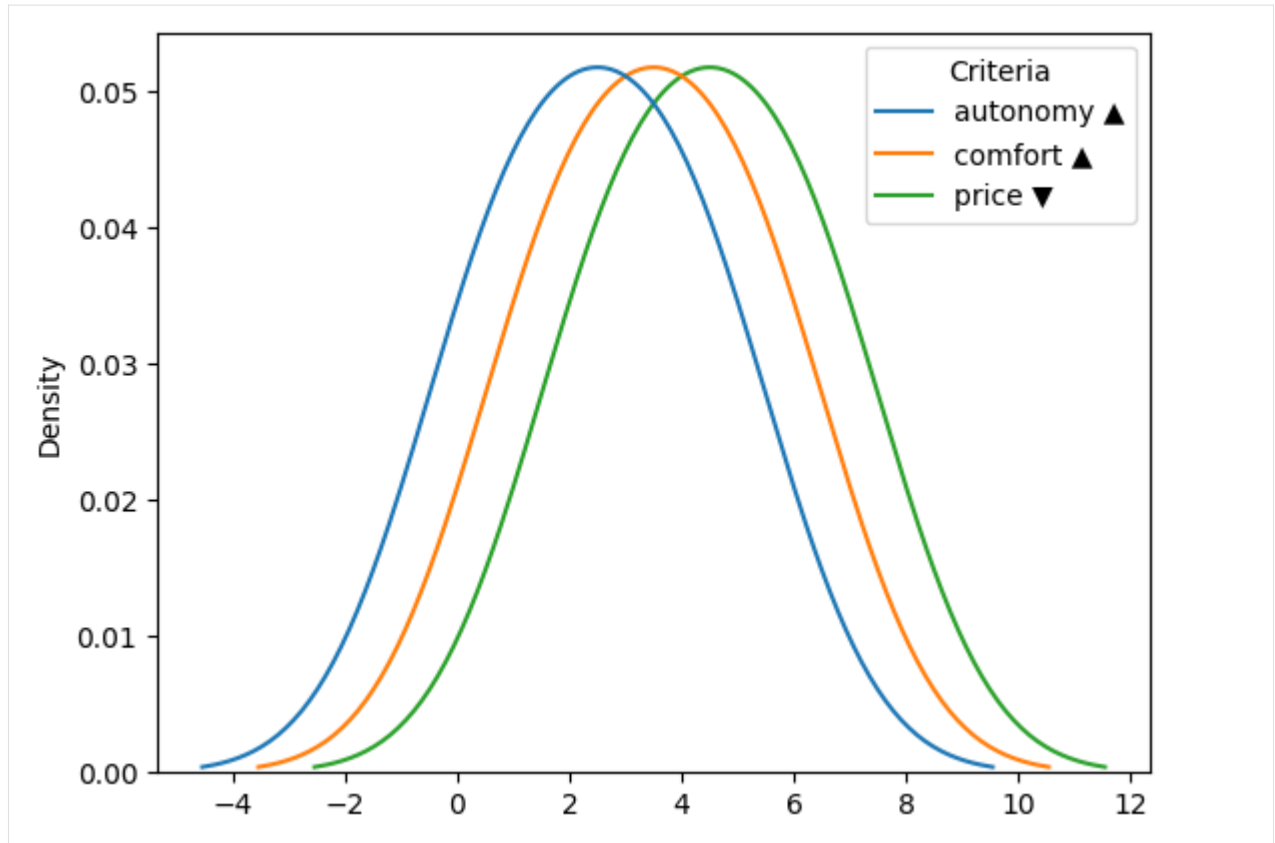
```
[14]: <AxesSubplot:ylabel='Density'>
```



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

```
[15]: dm.plot.kde()
```

```
[15]: <AxesSubplot:ylabel='Density'>
```

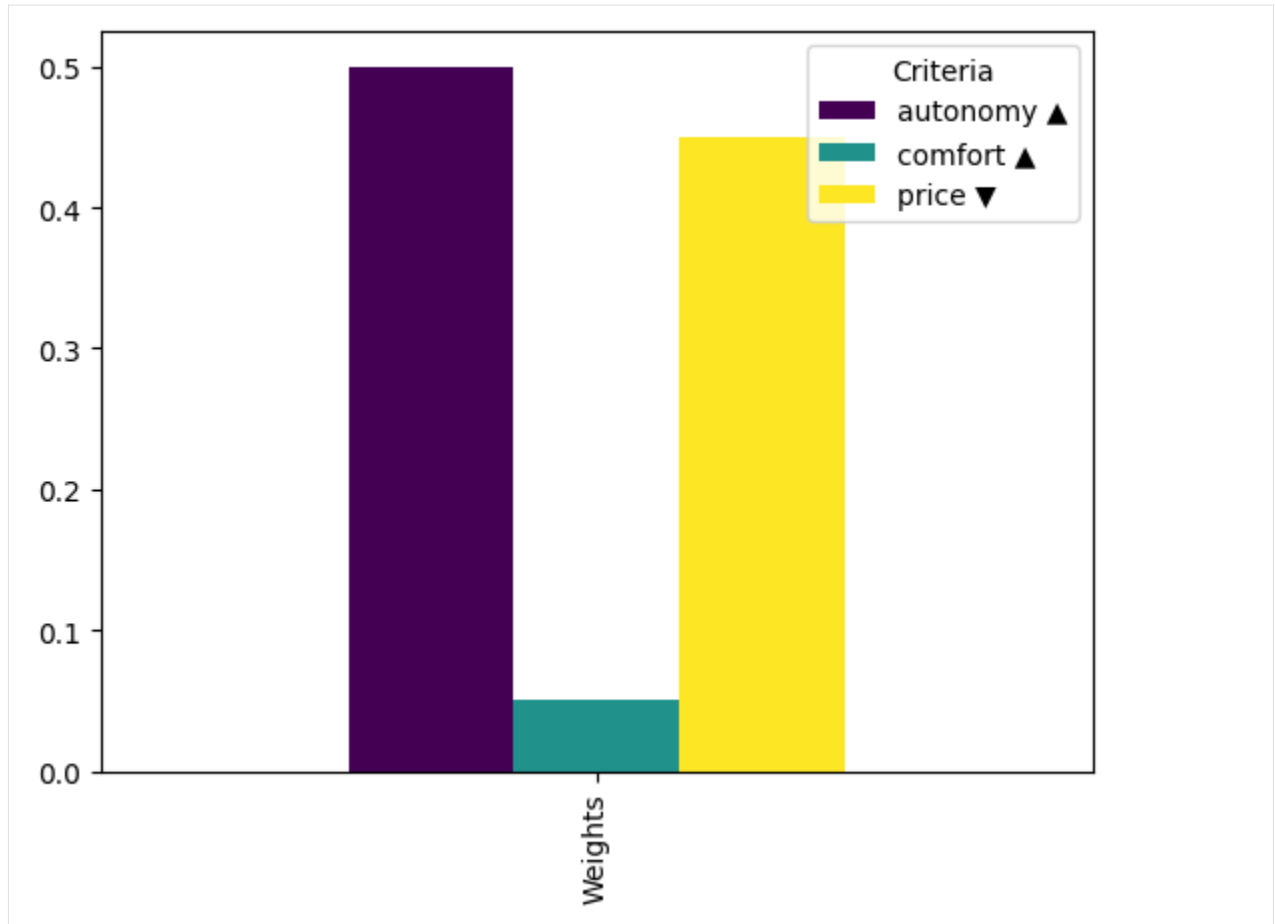


Every plot has their own set of parameters, defined by the subjacent function

Let's change the colors of the weight bar plot and show:

```
[16]: dm.plot.wbar(cmap="viridis")
```

```
[16]: <AxesSubplot:>
```



## Data transformation

Data in its current form is difficult to understand and analyze. On one hand they are out of scale, and on the other they have both minimizing and maximizing criteria.

---

Note: Scikit-Criteria objective preference

For a design decision *Scikit-Criteria* always prefers **Maximize** objectives. There are some functionalities that trigger warnings against **Minimize** criteria, and others that directly and others directly fail.

---

To solve these problems, we will use two processors:

- First `InvertMinimize` which inverts the minimizing objectives. by dividing out the inverse of each criterion value ( $1/C_j$ ).
- Second, `SumScaler` which will divide each criterion value by the total sum of the criteria, taking all of them into the range  $[0, 1]$ .

First we start by importing the two necessary modules.

```
[17]: from skcriteria.preprocessing import invert_objectives, scalers
```

Data in its current form is difficult to understand and analyze. The first thing we must do now is to reverse the maximization criteria.

This involves:

1. Create the transformer and store it in the `inverter` variable.
2. Apply the transformation by calling the `transform` method of the transformer and passing it as parameter our decision matrix `dm`.
3. Save the transformed decision matrix in a new variable `dmt`.

In code:

```
[18]: inverter = invert_objectives.InvertMinimize()
      dmt = inverter.transform(dm)
      dmt
```

```
[18]:      autonomy[ 0.50]  comfort[ 0.05]  price[ 0.45]
      VW                1                2         0.333333
      Ford              4                5         0.166667
      [2 Alternatives x 3 Criteria]
```

The next step is to scale the values between  $[0, 1]$  using the `SumScaler`.

For this step we need

1. Create the transformer and store it in the `inverter` variable. In this case the *scalers* support a parameter called `target` which can have three different values:
  - `target="matrix"` The matrix  $A$  is normalized.
  - `target="weights"` normalizes the weights  $w$ .
  - `target="both"` normalizes matrix  $A$  and weights  $w$ .

In our case we are going to ask the scaler to scale both components of the decision matrix (`target="both"`)

2. Apply the transformation by calling the `transform` method of the transformer and passing it as parameter our decision matrix `dmt`.
3. Save the transformed decision by overwriting the variable `dmt`.

```
[19]: scaler = scalers.SumScaler(target="both")
      dmt = scaler.transform(dmt)
      dmt
```

```
[19]:      autonomy[ 0.50]  comfort[ 0.05]  price[ 0.45]
      VW                0.2            0.285714  0.666667
      Ford              0.8            0.714286  0.333333
      [2 Alternatives x 3 Criteria]
```

Now we can analyze if the matrix graphically by creating a graph for the matrix, and another for the weights.

---

Note: Advanced plots with Matplotlib

If you need more information on how to make graphs using *Matplotlib* please che this tutorial <https://matplotlib.org/stable/tutorials/index>

---

```
[20]: # we are going to user matplotlib capabilities of creat multiple figures
      import matplotlib.pyplot as plt

      # we create 2 axis with the same y axis
```

(continues on next page)

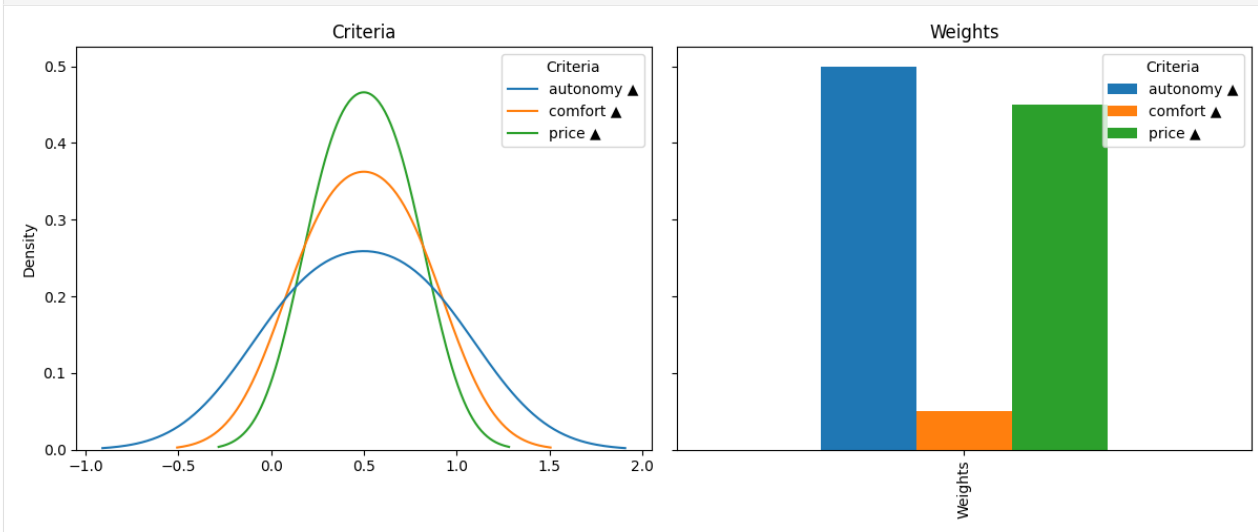
(continued from previous page)

```
fig, axs = plt.subplots(1, 2, figsize=(12, 5), sharey=True)

# in the first axis we plot the criteria KDE
dmt.plot.kde(ax=axs[0])
axs[0].set_title("Criteria")

# in the second axis we plot the weights as bars
dmt.plot.wbar(ax=axs[1])
axs[1].set_title("Weights")

# adjust the layout of the figute based on the content
fig.tight_layout()
```



## Using this data to feed some MCDA methods

### Weighted Sum Model

Let's rank our dummy data by `Weighted Sum Model`

First we need to import the required module

```
[21]: from skcriteria.madm import simple
```

To use the methods of MCDA structure we proceed in the same way as when using transformers:

1. We create the decision maker and store it in some variable (dec in our case).
2. Execute the `evaluate()` method inside the decision maker to create the result.
3. We store the result in some variable (rank in our case).

---

Note: Hyper-parameters

Some multi-criteria methods support “*hyper parameters*”, which are provided at the time of creation of the decision maker.

We will see an example with the *ELECTRE-I* method later on.

```
[22]: dec = simple.WeightedSumModel()
rank = dec.evaluate(dmt) # we use the transformed version of the data
rank
```

```
[22]: Alternatives  VW  Ford
Rank             2    1
[Method: WeightedSumModel]
```

We can see that `WeightedSumModel` prefers the alternative *Ford* over the *VW*.

We can access the intermediate calculators of the method through the `e_` attribute of the result object., which (in the case of `WeightedSumModel`) contains the resulting scores

```
[23]: rank.e_
```

```
[23]: <extra {'score'}>
```

```
[24]: rank.e_.score
```

```
[24]: array([0.41428571, 0.58571429])
```

Obviously you can access all the parts of the ranking as attributes of result object

```
[25]: rank.rank_
```

```
[25]: array([2, 1])
```

```
[26]: rank.alternatives
```

```
[26]: array(['VW', 'Ford'], dtype=object)
```

```
[27]: rank.method
```

```
[27]: 'WeightedSumModel'
```

### Technique for Order of Preference by Similarity to Ideal Solution (TOPSIS)

The following example will be approached with the `TOPSIS`. This method was chosen because of its popularity and because it uses another scaling technique (`VectorScaler`).

So the first thing one would intuitively do is to invert the original matrix criteria (`dm`) and then apply the normalization; but if we have several matrices or several methods this solution becomes cumbersome.

The proposed solution of *Scikit-Criteria* is to offer `pipelines`. The pipelines combine one or several transformers and one decision-maker to facilitate the execution of the experiments.

So, let's import the necessary modules for `TOPSIS` and the `pipelines`:

```
Distances and InvertMinimize
```

```
Since TOPSIS uses distances as a comparison metric, it is not recommended to use the InvertMinimize transformer. Instead we use NegateMinimize.
```

```
[28]: from skcriteria.madm import similarity # here lives TOPSIS
      from skcriteria.pipeline import mkpipe # this function is for create pipelines
```

The trick is that the weights still need to be scaled with SumScaler so be careful to assign the *targets* correctly in each transformer.

```
[29]: pipe = mkpipe(
        invert_objectives.NegateMinimize(),
        scalers.VectorScaler(target="matrix"), # this scaler transform the matrix
        scalers.SumScaler(target="weights"), # and this transform the weights
        similarity.TOPSIS(),
    )

pipe
```

```
[29]: <SKCPipeline [steps=[('negateminimize', <NegateMinimize []>), ('vectorscaler',
  ↳<VectorScaler [target='matrix']>), ('sumscaler', <SumScaler [target='weights']>), (
  ↳'topsis', <TOPSIS [metric='euclidean']>)]]>
```

Now we can directly call the pipeline `evaluate()` method with the original decision-matrix (`dm`).

This method sequentially executes the three transformers and finally the evaluator to obtain a result

```
[30]: rank = pipe.evaluate(dm)
rank
```

```
[30]: Alternatives  VW  Ford
Rank             2    1
[Method: TOPSIS]
```

```
[31]: print(rank.e_)
      print("Ideal:", rank.e_.ideal)
      print("Anti-Ideal:", rank.e_.anti_ideal)
      print("Similarity index:", rank.e_.similarity)

<extra {'similarity', 'ideal', 'anti_ideal'}>
Ideal: [ 0.48507125  0.04642383 -0.20124612]
Anti-Ideal: [ 0.12126781  0.01856953 -0.40249224]
Similarity index: [0.35548671 0.64451329]
```

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

## Élimination et Choix Traduisant la REALITÉ (ELECTRE)

For our final example, we are going to use the method `ELECTRE-I` which has two particularities:

1. It does not return a ranking but a kernel.
2. It supports two hyper-parameters: a concordance threshold  $p$  and a discordance threshold  $q$ .

Let's test the default threshold ( $p=0.65$ ,  $q=0.35$ ) but with two normalizations for different matrix: `VectorScaler` and `SumScaler`.

For this we will make two pipelines



```
[32]: from skcriteria.madm import electre

pipe_vector = mkpipe(
    invert_objectives.InvertMinimize(),
    scalers.VectorScaler(target="matrix"), # this scaler transform the matrix
    scalers.SumScaler(target="weights"), # and this transform the weights
    electre.ELECTRE1(p=0.65, q=0.35),
)

pipe_sum = mkpipe(
    invert_objectives.InvertMinimize(),
    scalers.SumScaler(target="weights"), # transform the matrix and weights
    electre.ELECTRE1(p=0.65, q=0.35),
)
```

```
[33]: kernel_vector = pipe_vector.evaluate(dm)
kernel_vector
```

```
[33]: Alternatives    VW  Ford
Kernel              True  True
[Method: ELECTRE1]
```

```
[34]: kernel_sum = pipe_sum.evaluate(dm)
kernel_sum
```

```
[34]: Alternatives    VW  Ford
Kernel              True  True
[Method: ELECTRE1]
```

As can be seen for this case both scalings give the same results

```
[35]: import datetime as dt
import skcriteria

print("Scikit-Criteria version:", skcriteria.VERSION)
print("Running datetime:", dt.datetime.now())

Scikit-Criteria version: 0.8
Running datetime: 2022-10-26 12:40:19.379650
```

## 4.2.2 Dominance and satisfaction analysis (AKA filters)

This tutorial provides a practical overview of how to use scikit-criteria for satisfaction and dominance analysis, as well as the creation of filters for data cleaning.

## Case

In order to decide to purchase a series of bonds, a company studied five candidate investments: *PE*, *JN*, *AA*, *FX*, *MM* and *GN*.

The finance department decides to consider the following criteria for selection. selection:

1. **ROE**: Return percentage. Sense of optimality, *Maximize*.
2. **CAP**: Market capitalization. Sense of optimality, *Maximize*.
3. **RI**: Risk. Sense of optimality, *Minimize*.

The full decision matrix

```
[1]: import skcriteria as skc

dm = skc.mkdm(
    matrix=[
        [7, 5, 35],
        [5, 4, 26],
        [5, 6, 28],
        [3, 4, 36],
        [1, 7, 30],
        [5, 8, 30],
    ],
    objectives=[max, max, min],
    alternatives=["PE", "JN", "AA", "FX", "MM", "FN"],
    criteria=["ROE", "CAP", "RI"],
)

dm
```

```
[1]:
```

	ROE[ 1.0]	CAP[ 1.0]	RI[ 1.0]
PE	7	5	35
JN	5	4	26
AA	5	6	28
FX	3	4	36
MM	1	7	30
FN	5	8	30

[6 Alternatives x 3 Criteria]

## Satisfaction analysis

It is reasonable to think that any decision-maker would want to set “satisfaction thresholds” for each criterion, in such a way that alternatives that do not exceed the thresholds in any criterion are eliminated.

The basic idea was proposed in the work of “*A Behavioral Model of Rational Choice*”

[Simon, 1955] and presents the definition of “*aspiration levels*” and are set a priori by the decision maker.

For our example we will assume that the decision-maker only accepts alternatives with  $ROE \geq 2$

For this analysis we will need the `skcriteria.preprocessing.filters` module.

```
[2]: from skcriteria.preprocessing import filters
```

The filters are *transformers* and works as follows:

- At the moment of construction they are provided with a dict that as a key has the name of a criterion, and as a value the condition to be satisfied.
- Optionally it receives a parameter `ignore_missing_criteria` which if it is set to `False` (default value) fails any attempt to transform an decision matrix that does not have any of the criteria.
- For an alternative not to be eliminated the alternative has to pass all filter conditions.

The simplest filter consists of instances of the class `filters.Filters`, which as a value of the configuration dict, accepts functions that are applied to the corresponding criteria and returns a mask where the `True` values denote the alternatives that we want to keep.

To write the function that filters the alternatives where  $ROE \geq 2$ .

```
[3]: def roe_filter(v):
      return v >= 2 # criteria are numpy.ndarray

flt = filters.Filter({"ROE": roe_filter})
flt
```

```
[3]: <Filter [criteria_filters={'ROE': <function roe_filter at 0x7fb3f922aa70>}, ignore_
      ↪missing_criteria=False]>
```

However, `scikit-criteria` offers a simpler collection of filters that implements the most common operations of equality, inequality and inclusion a set.

In our case we are interested in the `FilterGE` class, where `GE` stands for *Greater or Equal*.

So the filter would be defined as

```
[4]: flt = filters.FilterGE({"ROE": 2})
flt
```

```
[4]: <FilterGE [criteria_filters={'ROE': 2}, ignore_missing_criteria=False]>
```

The way to apply the filter to a `DecisionMatrix`, is like any other transformer:

```
[5]: dmf = flt.transform(dm)
dmf
```

```
[5]:   ROE[ 1.0]  CAP[ 1.0]  RI[ 1.0]
PE         7         5         35
JN         5         4         26
AA         5         6         28
FX         3         4         36
FN         5         8         30
[5 Alternatives x 3 Criteria]
```

As can be seen, we eliminated the alternative `MM` which did not comply with an  $ROE \geq 2$ .

If on the other hand (to give an example) we would like to filter out the alternatives  $ROE > 3$  and  $CAP > 4$  (using the original matrix), we can use the filter `FilterGT` where `GT` is *Greater Than*.

```
[6]: filters.FilterGT({"ROE": 3, "CAP": 4}).transform(dm)
```

```
[6]:      ROE[ 1.0]  CAP[ 1.0]  RI[ 1.0]
PE          7          5          35
AA          5          6          28
FN          5          8          30
[3 Alternatives x 3 Criteria]
```

---

**Note:**

If it is necessary to filter the alternatives by two separate conditions, a pipeline can be used. An example of this can be seen below, where we combine a satisficing and a dominance filter

---

The complete list of filters implemented by Scikit-Criteria is:

- `filters.Filter`: Filter alternatives according to the value of a criterion using arbitrary functions.

```
filters.Filter({"criterion": lambda v: v > 1})
```

- `filters.FilterGT`: Filter Greater Than ( $>$ ).

```
filters.FilterGT({"criterion": 1})
```

- `filters.FilterGE`: Filter Greater or Equal than ( $\geq$ ).

```
filters.FilterGE({"criterion": 2})
```

- `filters.FilterLT`: Filter Less Than ( $<$ ).

```
filters.FilterLT({"criterion": 1})
```

- `filters.FilterLE`: Filter Less or Equal than ( $\leq$ ).

```
filters.FilterLE({"criterion": 2})
```

- `filters.FilterEQ`: Filter Equal ( $=$ ).

```
filters.FilterEQ({"criterion": 1})
```

- `filters.FilterNE`: Filter Not-Equal than ( $\neq$ ).

```
filters.FilterNE({"criterion": 2})
```

- `filters.FilterIn`: Filter if the values is in a set ( $\in$ ).

```
filters.FilterIn({"criterion": [1, 2, 3]})
```

- `filters.FilterNotIn`: Filter if the values is not in a set ( $\notin$ ).

```
filters.FilterNotIn({"criterion": [1, 2, 3]})
```

## Dominance

An alternative  $A_0$  is said to dominate an alternative  $A_1$  ( $A_0 \succeq A_1$ ), if  $A_0$  is equal in all criteria and better in at least one criterion. On the other hand,  $A_0$  strictly dominates  $A_1$  ( $A_0 \succ A_1$ ), if  $A_0$  is better on all criteria than  $A_1$ .

Under this same train of thought, an alternative that dominates all others is called a “*dominant alternative*”. If there is a dominant alternative, it is undoubtedly the best choice, as long as a full ranking is not required.

On the other hand, an *alternative is dominated* if there exists at least one other alternative that dominates it. If a dominated alternative exists and a consigned ordering is not desired, it must be removed from the set of decision alternatives.

Generally only the non-dominated or efficient alternatives are the interested ones.

## Scikit-Criteria dominance analysis

Scikit-criteria, contains a number of tools within the attribute, `DecisionMatrix.dominance`, useful for the evaluation of dominant and dominated alternatives.

For example, we can access all the dominated alternatives by using the `dominated` method

```
[7]: dmf.dominance.dominated()
```

```
[7]: Alternatives
PE    False
JN    False
AA    False
FX     True
FN    False
Name: Dominated, dtype: bool
```

It can be seen with this, that `FX` is an dominated alternative. In addition if we want to know which are the *strictly dominated* alternatives we need to provide the `strict` parameter to the method:

```
[8]: dmf.dominance.dominated(strict=True)
```

```
[8]: Alternatives
PE    False
JN    False
AA    False
FX     True
FN    False
Name: Strictly dominated, dtype: bool
```

It can be seen that `FX` is strictly dominated by at least one other alternative.

If we wanted to find out which are the dominant alternatives of `FX`, we can opt for two paths:

1. List all the dominant/strictly dominated alternatives of `FX` using `dominator_of()`.

```
[9]: dmf.dominance.dominators_of("FX", strict=True)
```

```
[9]: array(['PE', 'AA', 'FN'], dtype=object)
```

2. Use `dominance()/dominance.dominance()` to see the full relationship between all alternatives.

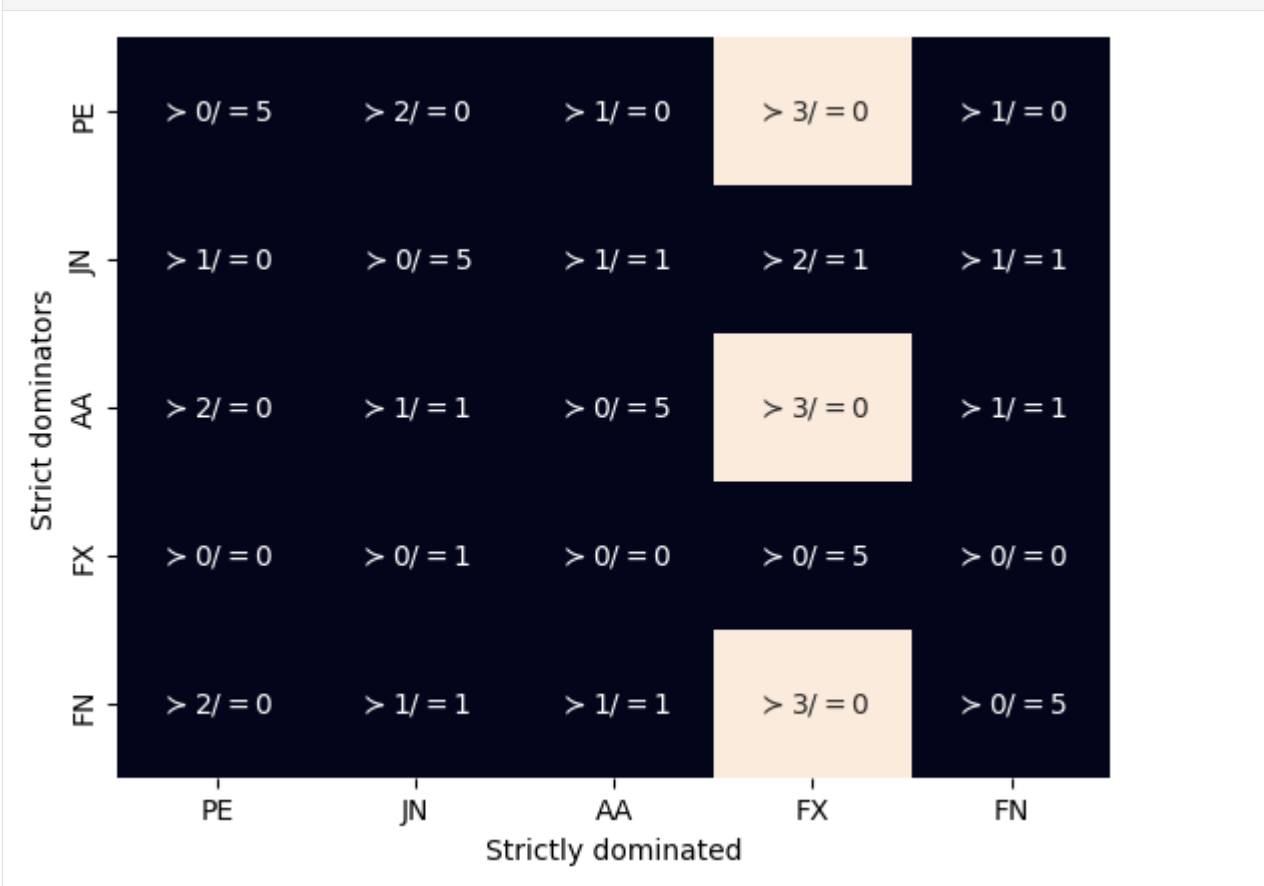
```
[10]: dmf.dominance(strict=True) # equivalent to dmf.dominance.dominance()
```

```
[10]: Strictly dominated    PE    JN    AA    FX    FN
Strict dominators
PE                False False False  True False
JN                False False False False False
AA                False False False  True False
FX                False False False False False
FN                False False False  True False
```

the result of the method is a `DataFrame` that in each cell has a `True` value if the *row alternative* dominates the *column alternative*.

If this matrix is very large: we can, for example, visualize it

```
[11]: dmf.plot.dominance(strict=True);
```



Finally we can see how each of the alternatives relate to each other dominatnes with `FX` using `compare()`.

```
[12]: for dominant in dmf.dominance.dominators_of("FX"):
display(dmf.dominance.compare(dominant, 'FX'))
```

Alternatives	Criteria			Performance
	ROE	CAP	RI	
PE	True	True	True	3
FX	False	False	False	0
Equals	False	False	False	0

		Criteria			Performance
		ROE	CAP	RI	
Alternatives	JN	True	False	True	2
	FX	False	False	False	0
Equals		False	True	False	1

		Criteria			Performance
		ROE	CAP	RI	
Alternatives	AA	True	True	True	3
	FX	False	False	False	0
Equals		False	False	False	0

		Criteria			Performance
		ROE	CAP	RI	
Alternatives	FN	True	True	True	3
	FX	False	False	False	0
Equals		False	False	False	0

### Filter non-dominated alternatives

Finally `skcriteria` offers a way to filter non-dominated alternatives, which it accepts as a parameter if you want to evaluate strict dominance.

```
[13]: flt = filters.FilterNonDominated(strict=True)
      flt
```

```
[13]: <FilterNonDominated [strict=True]>
```

```
[14]: flt.transform(dmf)
```

```
[14]:   ROE[ 1.0]  CAP[ 1.0]  RI[ 1.0]
PE          7          5          35
JN          5          4          26
AA          5          6          28
FN          5          8          30
[4 Alternatives x 3 Criteria]
```

### Full experiment

We can finally create a complete MCDA experiment that takes into account the in satisfaction and dominance analysis.

The complete experiment would have the following steps

1. Eliminate alternatives that do not yield at least 2% ( $\$ROE \geq \$2$ ).
2. Eliminate dominated alternatives.
3. Convert all criteria to maximize.
4. The weights are scaled by the total sum.
5. The matrix is scaled by the vector modulus.
6. Apply **TOPSIS**.

The most convenient way to do this is to use a pipeline.

```
[15]: from skcriteria.preprocessing import scalers, invert_objectives
      from skcriteria.madm.similarity import TOPSIS
      from skcriteria.pipeline import mkpipe

      pipe = mkpipe(
          filters.FilterGE({"ROE": 2}),
          filters.FilterNonDominated(strict=True),
          invert_objectives.NegateMinimize(),
          scalers.SumScaler(target="weights"),
          scalers.VectorScaler(target="matrix"),
          TOPSIS(),
      )

      pipe
```

```
[15]: <SKCPipeline [steps=[('filterge', <FilterGE [criteria_filters={'ROE': 2}, ignore_missing_
↳ criteria=False]>), ('filternondominated', <FilterNonDominated [strict=True]>), (
↳ 'negateminimize', <NegateMinimize []>), ('sumscaler', <SumScaler [target='weights']>),
↳ ('vectorscaler', <VectorScaler [target='matrix']>), ('topsis', <TOPSIS [metric=
↳ 'euclidean']>)]]>
```

We now apply the pipeline to the original data

```
[16]: pipe.evaluate(dm)
```

```
[16]: Alternatives PE JN AA FN
      Rank          3  4  2  1
      [Method: TOPSIS]
```

```
[17]: import datetime as dt
      import skcriteria

      print("Scikit-Criteria version:", skcriteria.VERSION)
      print("Running datetime:", dt.datetime.now())
```

```
Scikit-Criteria version: 0.8
Running datetime: 2022-10-26 12:38:20.917427
```

#### 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](#)



## 4.3 skcriteria package

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

### 4.3.1 skcriteria.core package

Core functionalities and structures of skcriteria.

#### skcriteria.core.data module

Data abstraction layer.

This module defines the `DecisionMatrix` object, which internally encompasses the alternative matrix, weights and objectives (MIN, MAX) of the criteria.

**class** `skcriteria.core.data.DecisionMatrix`(*data\_df*, *objectives*, *weights*)

Bases: `object`

Representation of all data needed in the MCDA analysis.

This object gathers everything necessary to represent a data set used in MCDA:

- An alternative matrix where each row is an alternative and each column is of a different criteria.
- An optimization objective (Minimize, Maximize) for each criterion.
- A weight for each criterion.
- An independent type of data for each criterion

`DecisionMatrix` has two main forms of construction:

1. Use the default constructor of the `DecisionMatrix` class `pandas.DataFrame` where the index is the alternatives and the columns are the criteria; an iterable with the objectives with the same amount of elements that columns/criteria has the dataframe; and an iterable with the weights also with the same amount of elements as criteria.

```
>>> import pandas as pd
>>> from skcriteria import DecisionMatrix, mkdm
```

```
>>> data_df = pd.DataFrame(
...     [[1, 2, 3], [4, 5, 6]],
...     index=["A0", "A1"],
...     columns=["C0", "C1", "C2"]
... )
>>> objectives = [min, max, min]
>>> weights = [1, 1, 1]
```

```
>>> dm = DecisionMatrix(data_df, objectives, weights)
>>> dm
  C0[ 1.0] C1[ 1.0] C2[ 1.0]
A0         1         2         3
A1         4         5         6
[2 Alternatives x 3 Criteria]
```

2. Use the classmethod `DecisionMatrix.from_mcda_data` which requests the data in a more natural way for this type of analysis (the weights, the criteria / alternative names, and the data types are optional)

```
>>> DecisionMatrix.from_mcda_data(
...     [[1, 2, 3], [4, 5, 6]],
...     [min, max, min],
...     [1, 1, 1])
C0[ 1.0] C1[ 1.0] C2[ 1.0]
A0      1      2      3
A1      4      5      6
[2 Alternatives x 3 Criteria]
```

For simplicity a function is offered at the module level analogous to `from_mcda_data` called `mkdm` (make decision matrix).

### Parameters

- **data\_df** (`pandas.DataFrame`) – Dataframe where the index is the alternatives and the columns are the criteria.
- **objectives** (`numpy.ndarray`) – An iterable with the targets with sense of optimality of every criteria (You can use any alias defined in `Objective`) the same length as columns/criteria has the `data_df`.
- **weights** (`numpy.ndarray`) – An iterable with the weights also with the same amount of elements as criteria.

**classmethod** `from_mcda_data`(*matrix, objectives, weights=None, alternatives=None, criteria=None, dtypes=None*)

Create a new `DecisionMatrix` object.

This method receives the parts of the matrix, in what conceptually the matrix of alternatives is usually divided

### Parameters

- **matrix** (*Iterable*) – The matrix of alternatives. Where every row is an alternative and every column is a criteria.
- **objectives** (*Iterable*) – The array with the sense of optimality of every criteria. You can use any alias provided by the objective class.
- **weights** (*Iterable o None (default None)*) – Optional weights of the criteria. If is `None` all the criteria are weighted with 1.
- **alternatives** (*Iterable o None (default None)*) – Optional names of the alternatives. If is `None`, all the alternatives are names “A[n]” where n is the number of the row of *matrix* starting at 0.
- **criteria** (*Iterable o None (default None)*) – Optional names of the criteria. If is `None`, all the alternatives are names “C[m]” where m is the number of the columns of *matrix* starting at 0.
- **dtypes** (*Iterable o None (default None)*) – Optional types of the criteria. If is `None`, the type is inferred automatically by pandas.

### Returns

A new decision matrix.

**Return type***DecisionMatrix***Example**

```
>>> DecisionMatrix.from_mcda_data(
...     [[1, 2, 3], [4, 5, 6]],
...     [min, max, min],
...     [1, 1, 1])
   C0[ 1.0] C1[ 1.0] C2[ 1.0]
A0         1         2         3
A1         4         5         6
[2 Alternatives x 3 Criteria]
```

For simplicity a function is offered at the module level analogous to `from_mcda_data` called `mkdm` (make decision matrix).

**Notes**

This functionality generates more sensitive defaults than using the constructor of the `DecisionMatrix` class but is slower.

**property alternatives**

Names of the alternatives.

From this array you can also access the values of the alternatives as `pandas.Series`.

**property criteria**

Names of the criteria.

From this array you can also access the values of the criteria as `pandas.Series`.

**property weights**

Weights of the criteria.

**property objectives**

Objectives of the criteria as `Objective` instances.

**property minwhere**

Mask with value `True` if the criterion is to be minimized.

**property maxwhere**

Mask with value `True` if the criterion is to be maximized.

**property iobjectives**

Objectives of the criteria as `int`.

- Minimize = `Objective.MIN.value`
- Maximize = `Objective.MAX.value`

**property matrix**

Alternatives matrix as `pandas.DataFrame`.

The matrix excludes weights and objectives.

If you want to create a `DataFrame` with objectives and weights, use `DecisionMatrix.to_dataframe()`

**property dtypes**

Dtypes of the criteria.

**property plot**

Plot accessor.

**property stats**

Descriptive statistics accessor.

**property dominance**

Dominance information accessor.

**copy(\*\*kwargs)**

Return a deep copy of the current DecisionMatrix.

This method is also useful for manually modifying the values of the DecisionMatrix object.

**Parameters**

**kwargs** – The same parameters supported by `from_mcda_data()`. The values provided replace the existing ones in the object to be copied.

**Returns**

A new decision matrix.

**Return type**

*DecisionMatrix*

**to\_dataframe()**

Convert the entire DecisionMatrix into a dataframe.

The objectives and weights are added as rows before the alternatives.

**Returns**

A Decision matrix as pandas DataFrame.

**Return type**

pd.DataFrame

**Example**

```
>>> dm = DecisionMatrix.from_mcda_data(
>>> dm
...     [[1, 2, 3], [4, 5, 6]],
...     [min, max, min],
...     [1, 1, 1])
...     C0[ 1.0] C1[ 1.0] C2[ 1.0]
A0         1         2         3
A1         4         5         6

>>> dm.to_dataframe()
           C0  C1  C2
objectives MIN  MAX  MIN
weights     1.0  1.0  1.0
A0          1   2   3
A1          4   5   6
```

**to\_dict()**

Return a dict representation of the data.

All the values are represented as numpy array.

**describe(\*\*kwargs)**

Generate descriptive statistics.

Descriptive statistics include those that summarize the central tendency, dispersion and shape of a dataset's distribution, excluding NaN values.

**Parameters**

**pandas.DataFrame.describe()**. (Same parameters as) –

**Returns**

Summary statistics of DecisionMatrix provided.

**Return type**

pandas.DataFrame

**property shape**

Return a tuple with (number\_of\_alternatives, number\_of\_criteria).

dm.shape <==> np.shape(dm)

**equals(other)**

Return True if the decision matrix are equal.

This method calls *DecisionMatrix.aquals* without tolerance.

**Parameters**

**other** (skcriteria.DecisionMatrix) – Other instance to compare.

**Returns**

**equals** – Returns True if the two dm are equals.

**Return type**

bool:py:class:

**See also:**

[aequals](#), [numpy.isclose\(\)](#), [numpy.all\(\)](#), [numpy.any\(\)](#), [numpy.equal\(\)](#), [numpy.allclose\(\)](#)

**aequals(other, rtol=1e-05, atol=1e-08, equal\_nan=False)**

Return True if the decision matrix are equal within a tolerance.

The tolerance values are positive, typically very small numbers. The relative difference ( $rtol * abs(b)$ ) and the absolute difference *atol* are added together to compare against the absolute difference between *a* and *b*.

NaNs are treated as equal if they are in the same place and if `equal_nan=True`. Infs are treated as equal if they are in the same place and of the same sign in both arrays.

The proceeds as follows:

- If `other` is the same object return True.
- If `other` is not instance of 'DecisionMatrix', has different shape 'criteria', 'alternatives' or 'objectives' returns False.
- Next check the 'weights' and the matrix itself using the provided tolerance.

**Parameters**

- **other** (skcriteria.DecisionMatrix) – Other instance to compare.

- **rtol** (*float*) – The relative tolerance parameter (see Notes in `numpy.allclose()`).
- **atol** (*float*) – The absolute tolerance parameter (see Notes in `numpy.allclose()`).
- **equal\_nan** (*bool*) – Whether to compare NaN’s as equal. If True, NaN’s in `dm` will be considered equal to NaN’s in *other* in the output array.

**Returns**

**aequals** – Returns True if the two `dm` are equal within the given tolerance; False otherwise.

**Return type**

`bool:py:class:`

**See also:**

`equals`, `numpy.isclose()`, `numpy.all()`, `numpy.any()`, `numpy.equal()`, `numpy.allclose()`

**property loc**

Access a group of alternatives and criteria by label(s) or a boolean array.

`.loc[]` is primarily alternative label based, but may also be used with a boolean array.

Unlike `DataFrames`, `iloc`` of `DecisionMatrix` always returns an instance of `DecisionMatrix`.

**property iloc**

Purely integer-location based indexing for selection by position.

`.iloc[]` is primarily integer position based (from 0 to `length-1` of the axis), but may also be used with a boolean array.

Unlike `DataFrames`, `iloc`` of `DecisionMatrix` always returns an instance of `DecisionMatrix`.

`skcriteria.core.data.mkdm(matrix, objectives, weights=None, alternatives=None, criteria=None, dtypes=None)`

Create a new `DecisionMatrix` object.

This method receives the parts of the matrix, in what conceptually the matrix of alternatives is usually divided

**Parameters**

- **matrix** (*Iterable*) – The matrix of alternatives. Where every row is an alternative and every column is a criteria.
- **objectives** (*Iterable*) – The array with the sense of optimality of every criteria. You can use any alias provided by the objective class.
- **weights** (*Iterable* o *None* (default *None*)) – Optional weights of the criteria. If is *None* all the criteria are weighted with 1.
- **alternatives** (*Iterable* o *None* (default *None*)) – Optional names of the alternatives. If is *None*, all the alternatives are names “A[n]” where n is the number of the row of *matrix* starting at 0.
- **criteria** (*Iterable* o *None* (default *None*)) – Optional names of the criteria. If is *None*, all the alternatives are names “C[m]” where m is the number of the columns of *matrix* starting at 0.
- **dtypes** (*Iterable* o *None* (default *None*)) – Optional types of the criteria. If is *None*, the type is inferred automatically by pandas.

**Returns**

A new decision matrix.

**Return type***DecisionMatrix***Example**

```
>>> DecisionMatrix.from_mcd_data(
...     [[1, 2, 3], [4, 5, 6]],
...     [min, max, min],
...     [1, 1, 1])
  C0[ 1.0] C1[ 1.0] C2[ 1.0]
A0         1         2         3
A1         4         5         6
[2 Alternatives x 3 Criteria]
```

For simplicity a function is offered at the module level analogous to `from_mcd_data` called `mkdm` (make decision matrix).

**Notes**

This functionality generates more sensitive defaults than using the constructor of the `DecisionMatrix` class but is slower.

**skcriteria.core.dominance module**

Dominance helper for the `DecisionMatrix` object.

**class** `skcriteria.core.dominance.DecisionMatrixDominanceAccessor`(*dm*)

Bases: *AccessorABC*

Calculate basic statistics of the decision matrix.

**bt()**

Compare on how many criteria one alternative is better than another.

*bt* = better-than.

**Returns**

Where the value of each cell identifies on how many criteria the row alternative is better than the column alternative.

**Return type**

`pandas.DataFrame`

**eq()**

Compare on how many criteria two alternatives are equal.

**Returns**

Where the value of each cell identifies how many criteria the row and column alternatives are equal.

**Return type**

`pandas.DataFrame`

**dominance**(\*, *strict=False*)

Compare if one alternative dominates or strictly dominates another alternative.

In order to evaluate the dominance of an alternative *a0* over an alternative *a1*, the algorithm evaluates that *a0* is better in at least one criterion and that *a1* is not better in any criterion than *a0*. In the case that `strict = True` it also evaluates that there are no equal criteria.

**Parameters**

**strict** (bool, default `False`) – If True, strict dominance is evaluated.

**Returns**

Where the value of each cell is True if the row alternative dominates the column alternative.

**Return type**

`pandas.DataFrame`

**compare**(*a0*, *a1*)

Compare two alternatives.

It creates a summary data frame containing the comparison of the two alternatives on a per-criteria basis, indicating which of the two is the best value, or if they are equal. In addition, it presents a “Performance” column with the count for each case.

**Parameters**

- **a0** (*str*) – Names of the alternatives to compare.
- **a1** (*str*) – Names of the alternatives to compare.

**Returns**

Comparison of the two alternatives by criteria.

**Return type**

`pandas.DataFrame`

**dominated**(\*, *strict=False*)

Which alternative is dominated or strictly dominated by at least one other alternative.

**Parameters**

**strict** (bool, default `False`) – If True, strict dominance is evaluated.

**Returns**

Where the index indicates the name of the alternative, and if the value is `True`, it indicates that this alternative is dominated by at least one other alternative.

**Return type**

`pandas.Series`

**dominators\_of**(*a*, \*, *strict=False*)

Array of alternatives that dominate or strictly-dominate the alternative provided by parameters.

**Parameters**

- **a** (*str*) – On what alternative to look for the dominators.
- **strict** (bool, default `False`) – If True, strict dominance is evaluated.

**Returns**

List of alternatives that dominate a.

**Return type**

`numpy.ndarray`



**has\_loops**(\*, *strict=False*)

Retorna True si la matriz contiene loops de dominacia.

A loop is defined as if there are alternatives  $a0, a1$  and 'a2' such that "a0 a1 a2 a0" if *strict=True*, or "a0 a1 a2 a0" if *strict=False*

**Parameters**

**strict** (bool, default False) – If True, strict dominance is evaluated.

**Returns**

If True a loop exists.

**Return type**

bool

**Notes**

If the result of this method is True, the `dominators_of()` method raises a `RecursionError` for at least one alternative.

**skcriteria.core.methods module**

Core functionalities of scikit-criteria.

**class** `skcriteria.core.methods.SKMethodABC`

Bases: `object`

Base class for all class in scikit-criteria.

**Notes**

All estimators should specify:

- `_skcriteria_dm_type`: The type of the decision maker.
- `_skcriteria_parameters`: Available parameters.
- `_skcriteria_abstract_class`: If the class is abstract.

If the class is *abstract* the user can ignore the other two attributes.

**get\_parameters()**

Return the parameters of the method as dictionary.

**copy**(\*\**kwargs*)

Return a deep copy of the current Object..

This method is also useful for manually modifying the values of the object.

**Parameters**

**kwargs** – The same parameters supported by object constructor. The values provided replace the existing ones in the object to be copied.

**Return type**

A new object.

### skcriteria.core.objectives module

Definition of the objectives (MIN, MAX) for the criteria.

**class** skcriteria.core.objectives.**Objective**(*value*)

Bases: [Enum](#)

Representation of criteria objectives (Minimize, Maximize).

**MIN** = -1

Internal representation of minimize criteria

**MAX** = 1

Internal representation of maximize criteria

**classmethod** **from\_alias**(*alias*)

Return a n objective instase based on some given alias.

**to\_symbol**()

Return the printable symbol representation of the objective.

**classmethod** **construct\_from\_alias**(*alias*)

Return an objective instance based on some given alias.

Deprecated since version 0.8: Use `Objective.from_alias()` instead.

**to\_string**()

Return the printable representation of the objective.

### skcriteria.core.plot module

Plot helper for the DecisionMatrix object.

**class** skcriteria.core.plot.**DecisionMatrixPlotter**(*dm*)

Bases: [AccessorABC](#)

DecisionMatrix plot utilities.

Kind of plot to produce:

- 'heatmap' : criteria heat-map (default).
- 'wheatmap' : weights heat-map.
- 'bar' : criteria vertical bar plot.
- 'wbar' : weights vertical bar plot.
- 'barh' : criteria horizontal bar plot.
- 'wbarh' : weights horizontal bar plot.
- 'hist' : criteria histogram.
- 'whist' : weights histogram.
- 'box' : criteria boxplot.
- 'wbox' : weights boxplot.
- 'kde' : criteria Kernel Density Estimation plot.
- 'wkde' : weights Kernel Density Estimation plot.

- ‘ogive’ : criteria empirical cumulative distribution plot.
- ‘wogive’ : weights empirical cumulative distribution plot.
- ‘area’ : criteria area plot.
- ‘dominance’ : the dominance matrix as a heatmap.
- ‘frontier’ : criteria pair-wise Pareto-Frontier.

**heatmap**(\*\*kwargs)

Plot the alternative matrix as a color-encoded matrix.

**Parameters**

**\*\*kwargs** – Additional keyword arguments are passed and are documented in `seaborn.heatmap`.

**Return type**

`matplotlib.axes.Axes` or `numpy.ndarray` of them

**wheatmap**(\*\*kwargs)

Plot weights as a color-encoded matrix.

**Parameters**

**\*\*kwargs** – Additional keyword arguments are passed and are documented in `seaborn.heatmap`.

**Return type**

`matplotlib.axes.Axes` or `numpy.ndarray` of them

**bar**(\*\*kwargs)

Criteria vertical bar plot.

A bar plot is a plot that presents categorical data with rectangular bars with lengths proportional to the values that they represent. A bar plot shows comparisons among discrete categories. One axis of the plot shows the specific categories being compared, and the other axis represents a measured value.

**Parameters**

**\*\*kwargs** – Additional keyword arguments are passed and are documented in `DataFrame.plot.bar`.

**Return type**

`matplotlib.axes.Axes` or `numpy.ndarray` of them

**wbar**(\*\*kwargs)

Weights vertical bar plot.

A bar plot is a plot that presents categorical data with rectangular bars with lengths proportional to the values that they represent. A bar plot shows comparisons among discrete categories. One axis of the plot shows the specific categories being compared, and the other axis represents a measured value.

**Parameters**

**\*\*kwargs** – Additional keyword arguments are passed and are documented in `DataFrame.plot.bar`.

**Return type**

`matplotlib.axes.Axes` or `numpy.ndarray` of them

**barh**(\*\*kwargs)

Criteria horizontal bar plot.

A bar plot is a plot that presents categorical data with rectangular bars with lengths proportional to the values that they represent. A bar plot shows comparisons among discrete categories. One axis of the plot shows the specific categories being compared, and the other axis represents a measured value.

**Parameters**

**\*\*kwargs** – Additional keyword arguments are passed and are documented in `DataFrame.plot.barh`.

**Return type**

`matplotlib.axes.Axes` or `numpy.ndarray` of them

**wbarh(\*\*kwargs)**

Weights horizontal bar plot.

A bar plot is a plot that presents categorical data with rectangular bars with lengths proportional to the values that they represent. A bar plot shows comparisons among discrete categories. One axis of the plot shows the specific categories being compared, and the other axis represents a measured value.

**Parameters**

**\*\*kwargs** – Additional keyword arguments are passed and are documented in `DataFrame.plot.barh`.

**Return type**

`matplotlib.axes.Axes` or `numpy.ndarray` of them

**hist(\*\*kwargs)**

Draw one histogram of the criteria.

A histogram is a representation of the distribution of data. This function groups the values of all given Series in the DataFrame into bins and draws all bins in one `matplotlib.axes.Axes`.

**Parameters**

**\*\*kwargs** – Additional keyword arguments are passed and are documented in `seaborn.histplot`.

**Return type**

`matplotlib.axes.Axes` or `numpy.ndarray` of them

**whist(\*\*kwargs)**

Draw one histogram of the weights.

A histogram is a representation of the distribution of data. This function groups the values of all given Series in the DataFrame into bins and draws all bins in one `matplotlib.axes.Axes`.

**Parameters**

**\*\*kwargs** – Additional keyword arguments are passed and are documented in `seaborn.histplot`.

**Return type**

`matplotlib.axes.Axes` or `numpy.ndarray` of them

**box(\*\*kwargs)**

Make a box plot of the criteria.

A box plot is a method for graphically depicting groups of numerical data through their quartiles.

For further details see Wikipedia's entry for [boxplot](#).

**Parameters**

**\*\*kwargs** – Additional keyword arguments are passed and are documented in `seaborn.boxplot`.

**Return type**

matplotlib.axes.Axes or numpy.ndarray of them

**wbox**(\*\*kwargs)

Make a box plot of the weights.

A box plot is a method for graphically depicting groups of numerical data through their quartiles.

For further details see Wikipedia's entry for [boxplot](#).

**Parameters**

**\*\*kwargs** – Additional keyword arguments are passed and are documented in `seaborn.boxplot`.

**Return type**

matplotlib.axes.Axes or numpy.ndarray of them

**kde**(\*\*kwargs)

Criteria kernel density plot using Gaussian kernels.

In statistics, [kernel density estimation](#) (KDE) is a non-parametric way to estimate the probability density function (PDF) of a random variable. This function uses Gaussian kernels and includes automatic bandwidth determination.

**Parameters**

**\*\*kwargs** – Additional keyword arguments are passed and are documented in `seaborn.kdeplot`.

**Return type**

matplotlib.axes.Axes or numpy.ndarray of them

**wkde**(\*\*kwargs)

Weights kernel density plot using Gaussian kernels.

In statistics, [kernel density estimation](#) (KDE) is a non-parametric way to estimate the probability density function (PDF) of a random variable. This function uses Gaussian kernels and includes automatic bandwidth determination.

**Parameters**

**\*\*kwargs** – Additional keyword arguments are passed and are documented in `seaborn.kdeplot`.

**Return type**

matplotlib.axes.Axes or numpy.ndarray of them

**ogive**(\*\*kwargs)

Criteria empirical cumulative distribution plot.

In statistics, an empirical distribution function (eCDF) is the distribution function associated with the empirical measure of a sample. This cumulative distribution function is a step function that jumps up by  $1/n$  at each of the  $n$  data points. Its value at any specified value of the measured variable is the fraction of observations of the measured variable that are less than or equal to the specified value.

**Parameters**

**\*\*kwargs** – Additional keyword arguments are passed and are documented in `seaborn.ecdfplot`.

**Return type**

matplotlib.axes.Axes or numpy.ndarray of them

**wogive**(\*\*kwargs)

Weights empirical cumulative distribution plot.

In statistics, an empirical distribution function (eCDF) is the distribution function associated with the empirical measure of a sample. This cumulative distribution function is a step function that jumps up by  $1/n$  at each of the  $n$  data points. Its value at any specified value of the measured variable is the fraction of observations of the measured variable that are less than or equal to the specified value.

**Parameters**

**\*\*kwargs** – Additional keyword arguments are passed and are documented in `seaborn.ecdfplot`.

**Return type**

`matplotlib.axes.Axes` or `numpy.ndarray` of them

**area**(\*\*kwargs)

Draw an criteria stacked area plot.

An area plot displays quantitative data visually. This function wraps the `matplotlib` area function.

**Parameters**

**\*\*kwargs** – Additional keyword arguments are passed and are documented in `DataFrame.plot.area()`.

**Returns**

Area plot, or array of area plots if `subplots` is `True`.

**Return type**

`matplotlib.axes.Axes` or `numpy.ndarray`

**dominance**(\*, *strict=False*, \*\*kwargs)

Plot dominance as a color-encoded matrix.

In order to evaluate the dominance of an alternative  $a0$  over an alternative  $a1$ , the algorithm evaluates that  $a0$  is better in at least one criterion and that  $a1$  is not better in any criterion than  $a0$ . In the case that `strict = True` it also evaluates that there are no equal criteria.

**Parameters**

- **strict** (bool, default `False`) – If `True`, strict dominance is evaluated.
- **\*\*kwargs** – Additional keyword arguments are passed and are documented in `seaborn.heatmap`.

**Return type**

`matplotlib.axes.Axes` or `numpy.ndarray` of them

**frontier**(*x*, *y*, \*, *strict=False*, *ax=None*, *legend=True*, *scatter\_kws=None*, *line\_kws=None*)

Pareto frontier on two arbitrarily selected criteria.

A selection of an alternative of an  $A_o$  is a pareto-optimal solution when there is no other solution that selects an alternative that does not belong to  $A_o$  such that it improves on one objective without worsening at least one of the others.

From this point of view, the concept is used to analyze the possible optimal options of a solution given a variety of objectives or desires and one or more evaluation criteria.

Given a “universe” of alternatives, one seeks to determine the set that are Pareto efficient (i.e., those alternatives that satisfy the condition of not being able to better satisfy one of those desires or objectives without worsening some other). That set of optimal alternatives establishes a “Pareto set” or the “Pareto Frontier”.

The study of the solutions in the frontier allows designers to analyze the possible alternatives within the established parameters, without having to analyze the totality of possible solutions.

### Parameters

- **x** (*str*) – Criteria names. Variables that specify positions on the x and y axes.
- **y** (*str*) – Criteria names. Variables that specify positions on the x and y axes.
- **weighted** (bool, default False) – If its True the domination analysis is performed over the weighted matrix.
- **strict** (bool, default False) – If True, strict dominance is evaluated.
- **weighted** – If True, the weighted matrix is evaluated.
- **ax** (matplotlib.axes.Axes) – Pre-existing axes for the plot. Otherwise, call matplotlib.pyplot.gca internally.
- **legend** (bool, default True) – If False, no legend data is added and no legend is drawn.
- **scatter\_kws** (dict, default None) – Additional parameters passed to seaborn.scatterplot.
- **scatter\_kws** – Additional parameters passed to seaborn.lineplot, except for estimator and sort.

### Return type

matplotlib.axes.Axes or numpy.ndarray of them

### References

[Wikipedia contributors, 2022b] [Wikipedia contributors, 2022a]

## skcriteria.core.stats module

Stats helper for the DecisionMatrix object.

**class** skcriteria.core.stats.**DecisionMatrixStatsAccessor**(*dm*)

Bases: *AccessorABC*

Calculate basic statistics of the decision matrix.

Kind of statistic to produce:

- ‘corr’ : Compute pairwise correlation of columns, excluding NA/null values.
- ‘cov’ : Compute pairwise covariance of columns, excluding NA/null values.
- ‘describe’ : Generate descriptive statistics.
- ‘kurtosis’ : Return unbiased kurtosis over requested axis.
- ‘mad’ : Return the mean absolute deviation of the values over the requested axis.
- ‘max’ : Return the maximum of the values over the requested axis.
- ‘mean’ : Return the mean of the values over the requested axis.
- ‘median’ : Return the median of the values over the requested axis.
- ‘min’ : Return the minimum of the values over the requested axis.
- ‘pct\_change’ : Percentage change between the current and a prior element.
- ‘quantile’ : Return values at the given quantile over requested axis.
- ‘sem’ : Return unbiased standard error of the mean over requested axis.

- ‘skew’ : Return unbiased skew over requested axis.
- ‘std’ : Return sample standard deviation over requested axis.
- ‘var’ : Return unbiased variance over requested axis.

**mad**(*axis=0, skipna=True*)

Return the mean absolute deviation of the values over a given axis.

**Parameters**

- **axis** (*int*) – Axis for the function to be applied on.
- **skipna** (*bool, default True*) – Exclude NA/null values when computing the result.

### 4.3.2 `skcriteria.madm` package

MCDA methods.

#### `skcriteria.madm._madm_base` module

Core functionalities to create madm decision-maker classes.

**class** `skcriteria.madm._madm_base.SKCDecisionMakerABC`

Bases: `SKCMethodABC`

Abstract class for all decisor based methods in scikit-criteria.

**evaluate**(*dm*)

Validate the dm and calculate and evaluate the alternatives.

**Parameters**

**dm** (`skcriteria.data.DecisionMatrix`) – Decision matrix on which the ranking will be calculated.

**Returns**

Ranking.

**Return type**

`skcriteria.data.RankResult`

**class** `skcriteria.madm._madm_base.ResultABC`(*method, alternatives, values, extra*)

Bases: `object`

Base class to implement different types of results.

Any evaluation of the `DecisionMatrix` is expected to result in an object that extends the functionalities of this class.

**Parameters**

- **method** (*str*) – Name of the method that generated the result.
- **alternatives** (*array-like*) – Names of the alternatives evaluated.
- **values** (*array-like*) – Values assigned to each alternative by the method, where the *i*-th value refers to the valuation of the *i*-th. alternative.
- **extra** (*dict-like*) – Extra information provided by the method regarding the evaluation of the alternatives.



**property values**

Values assigned to each alternative by the method.

The  $i$ -th value refers to the valuation of the  $i$ -th. alternative.

**property method**

Name of the method that generated the result.

**property alternatives**

Names of the alternatives evaluated.

**property extra\_**

Additional information about the result.

---

**Note:** `e_` is an alias for this property

---

**property e\_**

Additional information about the result.

---

**Note:** `e_` is an alias for this property

---

**to\_series()**

The result as *pandas.Series*.

**property shape**

Tuple with (number\_of\_alternatives, ).

`rank.shape <==> np.shape(rank)`

**values\_equals(*other*)**

Check if the alternatives and ranking are the same.

The method doesn't check the method or the extra parameters.

**aequals(*other*, *rtol*=1e-05, *atol*=1e-08, *equal\_nan*=False)**

Return True if the result are equal within a tolerance.

The tolerance values are positive, typically very small numbers. The relative difference ( $rtol * abs(b)$ ) and the absolute difference *atol* are added together to compare against the absolute difference between *a* and *b*.

NaNs are treated as equal if they are in the same place and if `equal_nan=True`. Infs are treated as equal if they are in the same place and of the same sign in both arrays.

The proceeds as follows:

- If `other` is the same object return True.
- If `other` is not instance of 'DecisionMatrix', has different shape 'criteria', 'alternatives' or 'objectives' returns False.
- Next check the 'weights' and the matrix itself using the provided tolerance.

**Parameters**

- **other** (*Result*) – Other result to compare.
- **rtol** (*float*) – The relative tolerance parameter (see Notes in `numpy.allclose()`).
- **atol** (*float*) – The absolute tolerance parameter (see Notes in `numpy.allclose()`).

- **equal\_nan** (*bool*) – Whether to compare NaN’s as equal. If True, NaN’s in *dm* will be considered equal to NaN’s in *other* in the output array.

**Returns**

**aequals** – Returns True if the two result are equal within the given tolerance; False otherwise.

**Return type**

bool:py:class:

**See also:**

[equals](#), [numpy.isclose\(\)](#), [numpy.all\(\)](#), [numpy.any\(\)](#), [numpy.equal\(\)](#), [numpy.allclose\(\)](#)

**equals**(*other*)

Return True if the results are equal.

This method calls *aquals* without tolerance.

**Parameters**

**other** ([skcriteria.DecisionMatrix](#)) – Other instance to compare.

**Returns**

**equals** – Returns True if the two results are equals.

**Return type**

bool:py:class:

**See also:**

[aequals](#), [numpy.isclose\(\)](#), [numpy.all\(\)](#), [numpy.any\(\)](#), [numpy.equal\(\)](#), [numpy.allclose\(\)](#)

**class** [skcriteria.madm.\\_madm\\_base.RankResult](#)(*method, alternatives, values, extra*)

Bases: [ResultABC](#)

Ranking of alternatives.

This type of results is used by methods that generate a ranking of alternatives.

**Parameters**

- **method** (*str*) – Name of the method that generated the result.
- **alternatives** (*array-like*) – Names of the alternatives evaluated.
- **values** (*array-like*) – Values assigned to each alternative by the method, where the *i*-th value refers to the valuation of the *i*-th. alternative.
- **extra** (*dict-like*) – Extra information provided by the method regarding the evaluation of the alternatives.

**property** [has\\_ties\\_](#)

Return True if two alternatives shares the same ranking.

**property** [ties\\_](#)

Counter object that counts how many times each value appears.

**property** [rank\\_](#)

Alias for [values](#).

**property** [untied\\_rank\\_](#)

Ranking whitout ties.

if the ranking has ties this property assigns unique and consecutive values in the ranking. This method only assigns the values using the command `numpy.argsort(rank_) + 1`.

`to_series(*, untied=False)`

The result as `pandas.Series`.

`class skcriteria.madm._madm_base.KernelResult(method, alternatives, values, extra)`

Bases: `ResultABC`

Separates the alternatives between good (kernel) and bad.

This type of results is used by methods that select which alternatives are good and bad. The good alternatives are called “kernel”

#### Parameters

- **method** (*str*) – Name of the method that generated the result.
- **alternatives** (*array-like*) – Names of the alternatives evaluated.
- **values** (*array-like*) – Values assigned to each alternative by the method, where the *i*-th value refers to the valuation of the *i*-th. alternative.
- **extra** (*dict-like*) – Extra information provided by the method regarding the evaluation of the alternatives.

**property kernel\_**

Alias for values.

**property kernel\_size\_**

How many alternatives has the kernel.

**property kernel\_where\_**

Indexes of the alternatives that are part of the kernel.

**property kernelwhere\_**

Indexes of the alternatives that are part of the kernel.

Deprecated since version 0.7: Use `kernel_where_` instead

**property kernel\_alternatives\_**

Return the names of alternatives in the kernel.

### `skcriteria.madm.electre` module

ELimination Et Choix Traduisant la REalité - ELECTRE.

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.

`skcriteria.madm.electre.concordance(matrix, objectives, weights)`

Calculate the concordance matrix.

`skcriteria.madm.electre.discordance(matrix, objectives)`

Calculate the discordance matrix.

`skcriteria.madm.electre.electre1(matrix, objectives, weights, p=0.65, q=0.35)`

Execute ELECTRE1 without any validation.

`class skcriteria.madm.electre.ELECTRE1(*, p=0.65, q=0.35)`

Bases: *SKCDecisionMakerABC*

Find a kernel of alternatives through ELECTRE-1.

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.

#### References

[Roy, 1990] [Roy, 1968] [Tzeng & Huang, 2011]

#### property p

Concordance threshold.

#### property q

Discordance threshold.

`skcriteria.madm.electre.weights_outrank(matrix, weights, objectives)`

Calculate a matrix of comparison of alternatives where the value of each cell determines how many times the value of the criteria weights of the row alternative exceeds those of the column alternative.

#### Notes

For more information about this matrix please check “Tomada de decisões em cenários complexos” [Gomes et al., 2004], p. 100

`skcriteria.madm.electre.electre2(matrix, objectives, weights, p0=0.65, p1=0.5, p2=0.35, q0=0.65, q1=0.35)`

Execute ELECTRE2 without any validation.

Deprecated since version 0.8: `electre2` implementation will change in version after 0.8

`class skcriteria.madm.electre.ELECTRE2(*args, **kwargs)`

Bases: *SKCDecisionMakerABC*

Find the ranking solution through ELECTRE-2.

ELECTRE II was proposed by Roy and Bertier (1971-1973) to overcome ELECTRE I’s inability to produce a ranking of alternatives. Instead of simply finding the kernel set, ELECTRE II can order alternatives by introducing the strong and the weak outranking relations.

Deprecated since version 0.8: `ELECTRE2` implementation will change in version after 0.8

## Notes

This implementation is based on the one presented in the book “Tomada de decisões em cenários complexos” [Gomes et al., 2004].

### Parameters

- **p0** (*float, optional (default=0.65, 0.5, 0.35)*) – Matching thresholds. These are the thresholds that indicate the extent to which an alternative can be considered equivalent, good or very good with respect to another alternative.

These thresholds must meet the condition “ $1 \geq p0 \geq p1 \geq p2 \geq 0$ ”.

- **p1** (*float, optional (default=0.65, 0.5, 0.35)*) – Matching thresholds. These are the thresholds that indicate the extent to which an alternative can be considered equivalent, good or very good with respect to another alternative.

These thresholds must meet the condition “ $1 \geq p0 \geq p1 \geq p2 \geq 0$ ”.

- **p2** (*float, optional (default=0.65, 0.5, 0.35)*) – Matching thresholds. These are the thresholds that indicate the extent to which an alternative can be considered equivalent, good or very good with respect to another alternative.

These thresholds must meet the condition “ $1 \geq p0 \geq p1 \geq p2 \geq 0$ ”.

- **q0** (*float, optional (default=0.65, 0.35)*) – Discordance threshold. Threshold of the degree to which an alternative is equivalent, preferred or strictly preferred to another alternative.

These thresholds must meet the condition “ $1 \geq q0 \geq q1 \geq 0$ ”.

- **q1** (*float, optional (default=0.65, 0.35)*) – Discordance threshold. Threshold of the degree to which an alternative is equivalent, preferred or strictly preferred to another alternative.

These thresholds must meet the condition “ $1 \geq q0 \geq q1 \geq 0$ ”.

## References

[Gomes et al., 2004] [Roy & Bertier, 1971] [Roy & Bertier, 1973]

### property p0

Concordance threshold 0.

### property p1

Concordance threshold 1.

### property p2

Concordance threshold 2.

### property q0

Discordance threshold 0.

### property q1

Discordance threshold 1.

**skcriteria.madm.moora module**

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

`skcriteria.madm.moora.ratio`(*matrix*, *objectives*, *weights*)

Execute ratio MOORA without any validation.

**class** `skcriteria.madm.moora.RatioMOORA`

Bases: `SKCDecisionMakerABC`

Ratio based MOORA method.

In MOORA the set of ratios are suggested to be normalized as the square roots of the sum of squared responses as denominators, but you can use any scaler.

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

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

**References**

[Brauers & Zavadskas, 2006]

`skcriteria.madm.moora.refpoint`(*matrix*, *objectives*, *weights*)

Execute reference point MOORA without any validation.

**class** `skcriteria.madm.moora.ReferencePointMOORA`

Bases: `SKCDecisionMakerABC`

Rank the alternatives by distance to a reference point.

The reference point is 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 suggested in the MOORA method, namely formula:

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

Preference is given to a reference point possessing as coordinates the dominating coordinates 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 coordinates, which are selected for the reference point, are realized in one of the candidate alternatives.

## References

[Brauwers & Zavadskas, 2012]

`skcriteria.madm.moora.fmf(matrix, objectives, weights)`

Execute Full Multiplicative Form without any validation.

**class** `skcriteria.madm.moora.FullMultiplicativeForm`

Bases: `SKCDecisionMakerABC`

Non-linear, non-additive ranking method method.

Full Multiplicative Form 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:

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

## Notes

The implementation works Instead the multiplication of the values we add the logarithms of the values to avoid underflow.

### Raises

**ValueError:** – If some objective is for minimization or some value in the matrix is  $\leq 0$ .

## References

[Brauwers & Zavadskas, 2012]

`skcriteria.madm.moora.multimoora(matrix, objectives, weights)`

Execute weighted product model without any validation.

**class** `skcriteria.madm.moora.MultiMOORA`

Bases: `SKCDecisionMakerABC`

Combination of RatioMOORA, RefPointMOORA and FullMultiplicativeForm.

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  $j$ .

### Raises

**ValueError:** – If some objective is for minimization or some value in the matrix is  $\leq 0$ .

## References

[Brauwers & Zavadskas, 2012]

### `skcriteria.madm.similarity` module

Methods based on a similarity between alternatives.

`skcriteria.madm.similarity.topsis`(*matrix, objectives, weights, metric='euclidean', \*\*kwargs*)

Execute TOPSIS without any validation.

**class** `skcriteria.madm.similarity.TOPSIS`(\**, metric='euclidean'*)

Bases: *SKCDecisionMakerABC*

The Technique for Order of Preference by Similarity to Ideal Solution.

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

**metric** (*str* or *callable*, *optional*) – The distance metric to use. If a string, the distance function can be *braycurtis*, *canberra*, *chebyshev*, *cityblock*, *correlation*, *cosine*, *dice*, *euclidean*, *hamming*, *jaccard*, *jensenshannon*, *kulsinski*, *mahalanobis*, *matching*, *minkowski*, *rogerstanimoto*, *russellrao*, *seuclidean*, *sokalmichener*, *sokalsneath*, *squeuclidean*, *wminkowski*, *yule*.

#### Warning:

#### UserWarning:

If some objective is to minimize.

## References

[Hwang & Yoon, 1981] [Wikipedia contributors, 2021a] [Tzeng & Huang, 2011]

### `property metric`

Which distance metric will be used.

### `skcriteria.madm.simple` module

Some simple and compensatory methods.

`skcriteria.madm.simple.wsm`(*matrix, weights*)

Execute weighted sum model without any validation.

**class** `skcriteria.madm.simple.WeightedSumModel`

Bases: *SKCDecisionMakerABC*

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 necessary.

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:

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

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

#### Raises

**ValueError:** – If some objective is for minimization.

#### References

[Fishburn, 1967], [Wikipedia contributors, 2021b], [Tzeng & Huang, 2011]

`skcriteria.madm.simple.wpm(matrix, weights)`

Execute weighted product model without any validation.

**class** `skcriteria.madm.simple.WeightedProductModel`

Bases: `SKCDecisionMakerABC`

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:

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

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:

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

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

#### Raises

**ValueError:** – If some objective is for minimization or some value in the matrix is  $\leq 0$ .

## References

[Bridgman, 1922] [Miller & others, 1963]

### `skcriteria.madm.simus` module

SIMUS (Sequential Interactive Model for Urban Systems) Method.

`skcriteria.madm.simus.simus`(*matrix*, *objectives*, *b=None*, *rank\_by=1*, *solver='pulp'*)

Execute SIMUS without any validation.

**class** `skcriteria.madm.simus.SIMUS`(\**, rank\_by=1, solver='pulp'*)

Bases: *SKCDecisionMakerABC*

SIMUS (Sequential Interactive Model for Urban Systems).

SIMUS 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 constraints 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

- **rank\_by** (*1 or 2 (default=1)*) – Which 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 underlying linear programs. The full list are available in *pulp.listSolvers(True)*. “pulp” or None used the default solver selected by “PuLP”.

#### Warning:

#### UserWarning:

If the method detect different weights by criteria.

#### Raises

- **ValueError:** – If the length of *b* does not match the number of criteria.
- **See** –
- **---** –
- **PuLP Documentation** <<https://coin-or.github.io/pulp/>>` –

#### property `solver`

Solver used by PuLP.

#### property `rank_by`

Which of the two ranking provided by SIMUS is used.

**evaluate**(*dm*, \*, *b=None*)

Validate the decision matrix and calculate a ranking.

**Parameters**

- **dm** (`skcriteria.data.DecisionMatrix`) – Decision matrix on which the ranking will be calculated.
- **b** (`numpy.ndarray`) – Right-side-value of the LP problem,  
SIMUS automatically assigns the vector of the right side (b) in the constraints of linear programs.

If the criteria are to maximize, then the constraint is  $\leq$ ; and if the column minimizes the constraint is  $\geq$ . The b/right side value limits of the constraint are chosen automatically based on the minimum or maximum value of the criteria/column if the constraint is  $\leq$  or  $\geq$  respectively.

The user provides “b” in some criteria and lets SIMUS choose automatically others. For example, if you want to limit the two constraints of the dm with 4 criteria by the value 100, b must be `[None, 100, 100, None]` where None will be chosen automatically by SIMUS.

**Returns**

Ranking.

**Return type**

`skcriteria.data.RankResult`

### 4.3.3 `skcriteria.preprocessing` package

Multiple data transformation routines.

#### `skcriteria.preprocessing._preprocessing_base` module

Core functionalities to create transformers.

**class** `skcriteria.preprocessing._preprocessing_base.SKCTransformerABC`

Bases: `SKCMethodABC`

Abstract class for all transformer in scikit-criteria.

**transform**(*dm*)

Perform transformation on *dm*.

**Parameters**

**dm** (`skcriteria.data.DecisionMatrix`) – The decision matrix to transform.

**Returns**

Transformed decision matrix.

**Return type**

`skcriteria.data.DecisionMatrix`

**class** `skcriteria.preprocessing._preprocessing_base.SKCMatrixAndWeightTransformerABC(target)`

Bases: `SKCTransformerABC`

Transform weights and matrix together or independently.

The Transformer that implements this abstract class can be configured to transform *weights*, *matrix* or *both* so only that part of the DecisionMatrix is altered.

This abstract class require to redefine `_transform_weights` and `_transform_matrix`, instead of `_transform_data`.

**property target**

Determine which part of the DecisionMatrix will be transformed.

**skcriteria.preprocessing.distance module**

**Warning:** This module is deprecated.

Normalization through the distance to distance function.

This entire module is deprecated.

`skcriteria.preprocessing.distance.cenit_distance(matrix, objectives)`

Calculate a scores with respect to an ideal and anti-ideal alternative.

For every criterion  $f$  of this multicriteria problem we define a membership function  $x_j$  mapping the values of  $f_j$  to the interval  $[0, 1]$ .

The result score  $x_{aj}$  is close to the ideal value  $f_j^*$ , which is the best performance in criterion , and far from the anti-ideal value  $f_j^*$ , which is the worst performance in criterion  $j$ . Both ideal and anti-ideal, are achieved by at least one of the alternatives under consideration.

$$x_{aj} = \frac{f_j(a) - f_j^*}{f_j^* - f_j^*}$$

Deprecated since version 0.8: Use `skcriteria.preprocessing.scalers.matrix_scale_by_cenit_distance` instead

**class** `skcriteria.preprocessing.distance.CenitDistance(*args, **kwargs)`

Bases: `CenitDistanceMatrixScaler`

Relative scores with respect to an ideal and anti-ideal alternative.

For every criterion  $f$  of this multicriteria problem we define a membership function  $x_j$  mapping the values of  $f_j$  to the interval  $[0, 1]$ .

The result score  $x_{aj}$  is close to the ideal value  $f_j^*$ , which is the best performance in criterion , and far from the anti-ideal value  $f_j^*$ , which is the worst performance in criterion  $j$ . Both ideal and anti-ideal, are achieved by at least one of the alternatives under consideration.

$$x_{aj} = \frac{f_j(a) - f_j^*}{f_j^* - f_j^*}$$

Deprecated since version 0.8: Use `skcriteria.preprocessing.scalers.CenitDistanceMatrixScaler` instead

## References

[Diakoulaki et al., 1995]

### skcriteria.preprocessing.filters module

Normalization through the distance to distance function.

```
class skcriteria.preprocessing.filters.SKByCriteriaFilterABC(criteria_filters, *,
                                                         ignore_missing_criteria=False)
```

Bases: *SKCTransformerABC*

Abstract class capable of filtering alternatives based on criteria values.

This abstract class require to redefine `_coerce_filters` and `_make_mask`, instead of `_transform_data`.

#### Parameters

- **criteria\_filters** (*dict*) – It is a dictionary in which the key is the name of a criterion, and the value is the filter condition.
- **ignore\_missing\_criteria** (*bool*, *default: False*) – If True, it is ignored if a decision matrix does not have any particular criteria that should be filtered.

#### property criteria\_filters

Conditions on which the alternatives will be evaluated.

It is a dictionary in which the key is the name of a criterion, and the value is the filter condition.

#### property ignore\_missing\_criteria

If the value is True the filter ignores the lack of a required criterion.

If the value is False, the lack of a criterion causes the filter to fail.

```
class skcriteria.preprocessing.filters.Filter(criteria_filters, *, ignore_missing_criteria=False)
```

Bases: *SKByCriteriaFilterABC*

Function based filter.

This class accepts as a filter any arbitrary function that receives as a parameter a as a parameter a criterion and returns a mask of the same size as the number of the number of alternatives in the decision matrix.

#### Parameters

- **criteria\_filters** (*dict*) – It is a dictionary in which the key is the name of a criterion, and the value is the filter condition.
- **ignore\_missing\_criteria** (*bool*, *default: False*) – If True, it is ignored if a decision matrix does not have any particular criteria that should be filtered.

## Examples

```

>>> from skcriteria.preprocess import filters

>>> dm = skc.mkdm(
...     matrix=[
...         [7, 5, 35],
...         [5, 4, 26],
...         [5, 6, 28],
...         [1, 7, 30],
...         [5, 8, 30]
...     ],
...     objectives=[max, max, min],
...     alternatives=["PE", "JN", "AA", "MM", "FN"],
...     criteria=["ROE", "CAP", "RI"],
... )

>>> tfm = filters.Filter({
...     "ROE": lambda e: e > 1,
...     "RI": lambda e: e >= 28,
... })
>>> tfm.transform(dm)
ROE[ 2.0] CAP[ 4.0] RI[ 1.0]
PE         7         5         35
AA         5         6         28
FN         5         8         30
[3 Alternatives x 3 Criteria]

```

```

class skcriteria.preprocessing.filters.SKArithmeticFilterABC(criteria_filters, *,
                                                            ignore_missing_criteria=False)

```

Bases: *SKCByCriteriaFilterABC*

Provide a common behavior to make filters based on the same comparator.

This abstract class require to redefine `_filter` method, and this will apply to each criteria separately.

This class is designed to implement in general arithmetic comparisons of “==”, “!=”, “>”, “>=”, “<”, “<=” taking advantage of the functions provided by numpy (e.g. `np.greater_equal()`).

#### Parameters

- **criteria\_filters** (*dict*) – It is a dictionary in which the key is the name of a criterion, and the value is the filter condition.
- **ignore\_missing\_criteria** (*bool*, *default: False*) – If True, it is ignored if a decision matrix does not have any particular criteria that should be filtered.

## Notes

The filter implemented with this class are slightly faster than function-based filters.

**class** `skcriteria.preprocessing.filters.FilterGT(criteria_filters, *, ignore_missing_criteria=False)`

Bases: `SKCArithmeticFilterABC`

Keeps the alternatives for which the criteria value are greater than a value.

### Parameters

- **criteria\_filters** (*dict*) – It is a dictionary in which the key is the name of a criterion, and the value is the filter condition.
- **ignore\_missing\_criteria** (*bool*, *default: False*) – If True, it is ignored if a decision matrix does not have any particular criteria that should be filtered.

## Notes

The filter implemented with this class are slightly faster than function-based filters.

## Examples

```
>>> from skcriteria.preprocess import filters

>>> dm = skc.mkdm(
...     matrix=[
...         [7, 5, 35],
...         [5, 4, 26],
...         [5, 6, 28],
...         [1, 7, 30],
...         [5, 8, 30]
...     ],
...     objectives=[max, max, min],
...     alternatives=["PE", "JN", "AA", "MM", "FN"],
...     criteria=["ROE", "CAP", "RI"],
... )

>>> tfm = filters.FilterGT({"ROE": 1, "RI": 27})
>>> tfm.transform(dm)
  ROE[ 2.0] CAP[ 4.0] RI[ 1.0]
PE         7         5         35
AA         5         6         28
FN         5         8         30
[3 Alternatives x 3 Criteria]
```

**class** `skcriteria.preprocessing.filters.FilterGE(criteria_filters, *, ignore_missing_criteria=False)`

Bases: `SKCArithmeticFilterABC`

Keeps the alternatives for which the criteria value are greater or equal than a value.

### Parameters

- **criteria\_filters** (*dict*) – It is a dictionary in which the key is the name of a criterion, and the value is the filter condition.

- **ignore\_missing\_criteria** (*bool*, *default: False*) – If True, it is ignored if a decision matrix does not have any particular criteria that should be filtered.

## Notes

The filter implemented with this class are slightly faster than function-based filters.

## Examples

```
>>> from skcriteria.preprocess import filters

>>> dm = skc.mkdm(
...     matrix=[
...         [7, 5, 35],
...         [5, 4, 26],
...         [5, 6, 28],
...         [1, 7, 30],
...         [5, 8, 30]
...     ],
...     objectives=[max, max, min],
...     alternatives=["PE", "JN", "AA", "MM", "FN"],
...     criteria=["ROE", "CAP", "RI"],
... )

>>> tfm = filters.FilterGE({"ROE": 1, "RI": 27})
>>> tfm.transform(dm)
ROE[ 2.0] CAP[ 4.0] RI[ 1.0]
PE         7         5         35
AA         5         6         28
MM         1         7         30
FN         5         8         30
[4 Alternatives x 3 Criteria]
```

**class** `skcriteria.preprocessing.filters.FilterLT(criteria_filters, *, ignore_missing_criteria=False)`

Bases: `SKArithmeticFilterABC`

Keeps the alternatives for which the criteria value are less than a value.

### Parameters

- **criteria\_filters** (*dict*) – It is a dictionary in which the key is the name of a criterion, and the value is the filter condition.
- **ignore\_missing\_criteria** (*bool*, *default: False*) – If True, it is ignored if a decision matrix does not have any particular criteria that should be filtered.



## Notes

The filter implemented with this class are slightly faster than function-based filters.

## Examples

```
>>> from skcriteria.preprocess import filters

>>> dm = skc.mkdm(
...     matrix=[
...         [7, 5, 35],
...         [5, 4, 26],
...         [5, 6, 28],
...         [1, 7, 30],
...         [5, 8, 30]
...     ],
...     objectives=[max, max, min],
...     alternatives=["PE", "JN", "AA", "MM", "FN"],
...     criteria=["ROE", "CAP", "RI"],
... )

>>> tfm = filters.FilterLT({"RI": 28})
>>> tfm.transform(dm)
ROE[ 2.0] CAP[ 4.0] RI[ 1.0]
JN          5          4          26
[1 Alternatives x 3 Criteria]
```

**class** `skcriteria.preprocessing.filters.FilterLE`(*criteria\_filters*, \*, *ignore\_missing\_criteria=False*)

Bases: `SKCArithmeticFilterABC`

Keeps the alternatives for which the criteria value are less or equal than a value.

### Parameters

- **criteria\_filters** (*dict*) – It is a dictionary in which the key is the name of a criterion, and the value is the filter condition.
- **ignore\_missing\_criteria** (*bool*, *default: False*) – If True, it is ignored if a decision matrix does not have any particular criteria that should be filtered.

## Notes

The filter implemented with this class are slightly faster than function-based filters.

## Examples

```
>>> from skcriteria.preprocess import filters

>>> dm = skc.mkdm(
...     matrix=[
...         [7, 5, 35],
...         [5, 4, 26],
...         [5, 6, 28],
...         [1, 7, 30],
...         [5, 8, 30]
...     ],
...     objectives=[max, max, min],
...     alternatives=["PE", "JN", "AA", "MM", "FN"],
...     criteria=["ROE", "CAP", "RI"],
... )

>>> tfm = filters.FilterLE({"RI": 28})
>>> tfm.transform(dm)
ROE[ 2.0] CAP[ 4.0] RI[ 1.0]
JN      5      4      26
AA      5      6      28
[2 Alternatives x 3 Criteria]
```

**class** `skcriteria.preprocessing.filters.FilterEQ(criteria_filters, *, ignore_missing_criteria=False)`

Bases: `SKCArithmeticFilterABC`

Keeps the alternatives for which the criteria value are equal than a value.

### Parameters

- **criteria\_filters** (*dict*) – It is a dictionary in which the key is the name of a criterion, and the value is the filter condition.
- **ignore\_missing\_criteria** (*bool*, *default: False*) – If True, it is ignored if a decision matrix does not have any particular criteria that should be filtered.

## Notes

The filter implemented with this class are slightly faster than function-based filters.

## Examples

```
>>> from skcriteria.preprocess import filters

>>> dm = skc.mkdm(
...     matrix=[
...         [7, 5, 35],
...         [5, 4, 26],
...         [5, 6, 28],
...         [1, 7, 30],
...         [5, 8, 30]
...     ],
...     objectives=[max, max, min],
...     alternatives=["PE", "JN", "AA", "MM", "FN"],
...     criteria=["ROE", "CAP", "RI"],
... )

>>> tfm = filters.FilterEQ({"CAP": 7, "RI": 30})
>>> tfm.transform(dm)
ROE[ 2.0] CAP[ 4.0] RI[ 1.0]
MM          1          7          30
[1 Alternatives x 3 Criteria]
```

**class** `skcriteria.preprocessing.filters.FilterNE(criteria_filters, *, ignore_missing_criteria=False)`  
 Bases: `SKCArithmeticFilterABC`

Keeps the alternatives for which the criteria value are not equal than a value.

### Parameters

- **criteria\_filters** (*dict*) – It is a dictionary in which the key is the name of a criterion, and the value is the filter condition.
- **ignore\_missing\_criteria** (*bool*, *default: False*) – If True, it is ignored if a decision matrix does not have any particular criteria that should be filtered.

## Notes

The filter implemented with this class are slightly faster than function-based filters.

## Examples

```
>>> from skcriteria.preprocess import filters

>>> dm = skc.mkdm(
...     matrix=[
...         [7, 5, 35],
...         [5, 4, 26],
...         [5, 6, 28],
...         [1, 7, 30],
...         [5, 8, 30]
...     ],
...     objectives=[max, max, min],
```

(continues on next page)

(continued from previous page)

```

...     alternatives=["PE", "JN", "AA", "MM", "FN"],
...     criteria=["ROE", "CAP", "RI"],
... )

>>> tfm = filters.FilterNE({"CAP": 7, "RI": 30})
>>> tfm.transform(dm)
   ROE[ 2.0] CAP[ 4.0] RI[ 1.0]
PE         7         5         35
JN         5         4         26
AA         5         6         28
[3 Alternatives x 3 Criteria]

```

```
class skcriteria.preprocessing.filters.SKCSetFilterABC(criteria_filters, *,
                                                    ignore_missing_criteria=False)
```

Bases: *SKCByCriteriaFilterABC*

Provide a common behavior to make filters based on set operations.

This abstract class require to redefine `_set_filter` method, and this will apply to each criteria separately.

This class is designed to implement in general set comparison like “inclusion” and “exclusion”.

#### Parameters

- **criteria\_filters** (*dict*) – It is a dictionary in which the key is the name of a criterion, and the value is the filter condition.
- **ignore\_missing\_criteria** (*bool*, *default: False*) – If True, it is ignored if a decision matrix does not have any particular criteria that should be filtered.

```
class skcriteria.preprocessing.filters.FilterIn(criteria_filters, *, ignore_missing_criteria=False)
```

Bases: *SKCSetFilterABC*

Keeps the alternatives for which the criteria value are included in a set of values.

#### Parameters

- **criteria\_filters** (*dict*) – It is a dictionary in which the key is the name of a criterion, and the value is the filter condition.
- **ignore\_missing\_criteria** (*bool*, *default: False*) – If True, it is ignored if a decision matrix does not have any particular criteria that should be filtered.

## Examples

```

>>> from skcriteria.preprocess import filters

>>> dm = skc.mkdm(
...     matrix=[
...         [7, 5, 35],
...         [5, 4, 26],
...         [5, 6, 28],
...         [1, 7, 30],
...         [5, 8, 30]
...     ],
...     objectives=[max, max, min],

```

(continues on next page)

(continued from previous page)

```

...     alternatives=["PE", "JN", "AA", "MM", "FN"],
...     criteria=["ROE", "CAP", "RI"],
... )

>>> tfm = filters.FilterIn({"ROE": [7, 1], "RI": [30, 35]})
>>> tfm.transform(dm)
   ROE[ 2.0] CAP[ 4.0] RI[ 1.0]
PE         7         5         35
MM         1         7         30
[2 Alternatives x 3 Criteria]

```

**class** `skcriteria.preprocessing.filters.FilterNotIn(criteria_filters, *, ignore_missing_criteria=False)`

Bases: `SKCSetFilterABC`

Keeps the alternatives for which the criteria value are not included in a set of values.

#### Parameters

- **criteria\_filters** (*dict*) – It is a dictionary in which the key is the name of a criterion, and the value is the filter condition.
- **ignore\_missing\_criteria** (*bool*, *default: False*) – If True, it is ignored if a decision matrix does not have any particular criteria that should be filtered.

#### Examples

```

>>> from skcriteria.preprocess import filters

>>> dm = skc.mkdm(
...     matrix=[
...         [7, 5, 35],
...         [5, 4, 26],
...         [5, 6, 28],
...         [1, 7, 30],
...         [5, 8, 30]
...     ],
...     objectives=[max, max, min],
...     alternatives=["PE", "JN", "AA", "MM", "FN"],
...     criteria=["ROE", "CAP", "RI"],
... )

>>> tfm = filters.FilterNotIn({"ROE": [7, 1], "RI": [30, 35]})
>>> tfm.transform(dm)
   ROE[ 2.0] CAP[ 4.0] RI[ 1.0]
JN         5         4         26
AA         5         6         28
[2 Alternatives x 3 Criteria]

```

**class** `skcriteria.preprocessing.filters.FilterNonDominated(*, strict=False)`

Bases: `SKCTransformerABC`

Keeps the non dominated or non strictly-dominated alternatives.

In order to evaluate the dominance of an alternative  $a0$  over an alternative  $a1$ , the algorithm evaluates that  $a0$  is better in at least one criterion and that  $a1$  is not better in any criterion than  $a0$ . In the case that `strict = True` it also evaluates that there are no equal criteria.

#### Parameters

**strict** (bool, default False) – If True, strictly dominated alternatives are removed, otherwise all dominated alternatives are removed.

#### Examples

```
>>> from skcriteria.preprocess import filters

>>> dm = skc.mkdm(
...     matrix=[
...         [7, 5, 35],
...         [5, 4, 26],
...         [5, 6, 28],
...         [1, 7, 30],
...         [5, 8, 30]
...     ],
...     objectives=[max, max, min],
...     alternatives=["PE", "JN", "AA", "MM", "FN"],
...     criteria=["ROE", "CAP", "RI"],
... )

>>> tfm = filters.FilterNonDominated(strict=False)
>>> tfm.transform(dm)
ROE[ 1.0] CAP[ 1.0] RI[ 1.0]
PE         7         5         35
JN         5         4         26
AA         5         6         28
FN         5         8         30
[4 Alternatives x 3 Criteria]
```

#### property strict

If the filter must remove the dominated or strictly-dominated alternatives.

#### transform(dm)

Perform transformation on *dm*.

#### Parameters

- **dm** (skcriteria.data.DecisionMatrix) –
- **transform.** (The decision matrix to) –

#### Returns

Transformed decision matrix.

#### Return type

skcriteria.data.DecisionMatrix

## skcriteria.preprocessing.impute module

Module that provides multiple strategies for missing value imputation.

The classes implemented here are a thin layer on top of the *sklearn.impute* module classes.

**class** skcriteria.preprocessing.impute.SKImputerABC

Bases: *SKCTransformerABC*

Abstract class capable of impute missing values of the matrix.

This abstract class require to redefine `_impute`, instead of `_transform_data`.

**class** skcriteria.preprocessing.impute.SimpleImputer(\*, missing\_values=nan, strategy='mean', fill\_value=None)

Bases: *SKImputerABC*

Imputation transformer for completing missing values.

Internally this class uses the *sklearn.impute.SimpleImputer* class.

### Parameters

- **missing\_values** (*int, float, str, np.nan, None or pandas.NA, default=np.nan*) – The placeholder for the missing values. All occurrences of *missing\_values* will be imputed.
- **strategy** (*str, default='mean'*) – The imputation strategy.
  - If “mean”, then replace missing values using the mean along each column. Can only be used with numeric data.
  - If “median”, then replace missing values using the median along each column. Can only be used with numeric data.
  - If “most\_frequent”, then replace missing using the most frequent value along each column. Can be used with strings or numeric data. If there is more than one such value, only the smallest is returned.
  - If “constant”, then replace missing values with *fill\_value*. Can be used with strings or numeric data.
- **fill\_value** (*str or numerical value, default=None*) – When strategy == “constant”, *fill\_value* is used to replace all occurrences of *missing\_values*. If left to the default, *fill\_value* will be 0.

### property missing\_values

The placeholder for the missing values.

### property strategy

The imputation strategy.

### property fill\_value

Used to replace all occurrences of *missing\_values*, when strategy == “constant”.

**class** skcriteria.preprocessing.impute.IterativeImputer(*estimator=None, \*, missing\_values=nan, sample\_posterior=False, max\_iter=10, tol=0.001, n\_nearest\_criteria=None, initial\_strategy='mean', imputation\_order='ascending', skip\_complete=False, min\_value=-inf, max\_value=inf, verbose=0, random\_state=None*)

Bases: `SKCImputerABC`

Multivariate imputer that estimates each feature from all the others.

A strategy for imputing missing values by modeling each feature with missing values as a function of other features in a round-robin fashion.

Internally this class uses the `sklearn.impute.IterativeImputer` class.

This estimator is still **experimental** for now: the predictions and the API might change without any deprecation cycle. To use it, you need to explicitly import `enable_iterative_imputer`:

```
>>> # explicitly require this experimental feature
>>> from sklearn.experimental import enable_iterative_imputer # noqa
>>> # now you can import normally from sklearn.impute
>>> from skcriteria.preprocess.impute import IterativeImputer
```

### Parameters

- **estimator** (*estimator object, default=BayesianRidge()*) – The estimator to use at each step of the round-robin imputation. If `sample_posterior=True`, the estimator must support `return_std` in its `predict` method.
- **missing\_values** (*int or np.nan, default=np.nan*) – The placeholder for the missing values. All occurrences of `missing_values` will be imputed.
- **sample\_posterior** (*bool, default=False*) – Whether to sample from the (Gaussian) predictive posterior of the fitted estimator for each imputation. Estimator must support `return_std` in its `predict` method if set to `True`. Set to `True` if using `IterativeImputer` for multiple imputations.
- **max\_iter** (*int, default=10*) – Maximum number of imputation rounds to perform before returning the imputations computed during the final round. A round is a single imputation of each criteria with missing values. The stopping criterion is met once  $\max(\text{abs}(X_{t-1} - X_t)) / \max(\text{abs}(X[\text{known\_vals}]]) < \text{tol}$ , where  $X_t$  is  $X$  at iteration  $t$ . Note that early stopping is only applied if `sample_posterior=False`.
- **tol** (*float, default=1e-3*) – Tolerance of the stopping condition.
- **n\_nearest\_criteria** (*int, default=None*) – Number of other criteria to use to estimate the missing values of each criteria column. Nearness between criteria is measured using the absolute correlation coefficient between each criteria pair (after initial imputation). To ensure coverage of criteria throughout the imputation process, the neighbor criteria are not necessarily nearest, but are drawn with probability proportional to correlation for each imputed target criteria. Can provide significant speed-up when the number of criteria is huge. If `None`, all criteria will be used.
- **initial\_strategy** (*{'mean', 'median', 'most\_frequent', 'constant'}, default='mean'*) – Which strategy to use to initialize the missing values. Same as the `strategy` parameter in `SimpleImputer`.
- **imputation\_order** (*{'ascending', 'descending', 'roman', 'arabic', 'random'}, default='ascending'*) – The order in which the criteria will be imputed. Possible values:
  - `'ascending'`: From criteria with fewest missing values to most.
  - `'descending'`: From criteria with most missing values to fewest.
  - `'roman'`: Left to right.
  - `'arabic'`: Right to left.



- *random*: A random order for each round.
- **min\_value** (*float* or array-like of shape  $(n\_criteria,)$ , *default=-np.inf*) – Minimum possible imputed value. Broadcast to shape  $(n\_criteria,)$  if scalar. If array-like, expects shape  $(n\_criteria,)$ , one min value for each criteria. The default is *-np.inf*.
- **max\_value** (*float* or array-like of shape  $(n\_criteria,)$ , *default=np.inf*) – Maximum possible imputed value. Broadcast to shape  $(n\_criteria,)$  if scalar. If array-like, expects shape  $(n\_criteria,)$ , one max value for each criteria. The default is *np.inf*.
- **verbose** (*int*, *default=0*) – Verbosity flag, controls the debug messages that are issued as functions are evaluated. The higher, the more verbose. Can be 0, 1, or 2.
- **random\_state** (*int*, *RandomState* instance or *None*, *default=None*) – The seed of the pseudo random number generator to use. Randomizes selection of estimator criteria if *n\_nearest\_criteria* is not *None*, the *imputation\_order* if *random*, and the sampling from posterior if *sample\_posterior=True*. Use an integer for determinism.

**property estimator**

Used at each step of the round-robin imputation.

**property missing\_values**

The placeholder for the missing values.

**property sample\_posterior**

Whether to sample from the (Gaussian) predictive posterior of the fitted estimator for each imputation.

**property max\_iter**

Maximum number of imputation rounds.

**property tol**

Tolerance of the stopping condition.

**property n\_nearest\_criteria**

Number of other criteria to use to estimate the missing values of each criteria column.

**property initial\_strategy**

Which strategy to use to initialize the missing values.

**property imputation\_order**

The order in which the criteria will be imputed.

**property min\_value**

Minimum possible imputed value.

**property max\_value**

Maximum possible imputed value.

**property verbose**

Verbosity flag, controls the debug messages that are issued as functions are evaluated.

**property random\_state**

The seed of the pseudo random number generator to use.

```
class skcriteria.preprocessing.impute.KNNImputer(*, missing_values=nan, n_neighbors=5,
                                                weights='uniform', metric='nan_euclidean')
```

Bases: *SKCImputerABC*

Imputation for completing missing values using k-Nearest Neighbors.

Internally this class uses the `sklearn.impute.KNNImputer` class.

Each sample's missing values are imputed using the mean value from  $n\_neighbors$  nearest neighbors found in the training set. Two samples are close if the criteria that neither is missing are close.

#### Parameters

- **missing\_values** (*int, float, str, np.nan or None, default=np.nan*) – The placeholder for the missing values. All occurrences of *missing\_values* will be imputed.
- **n\_neighbors** (*int, default=5*) – Number of neighboring samples to use for imputation.
- **weights** (*{'uniform', 'distance'} or callable, default='uniform'*) – Weight function used in prediction. Possible values:
  - *'uniform'*: uniform weights. All points in each neighborhood are weighted equally.
  - *'distance'*: weight points by the inverse of their distance. in this case, closer neighbors of a query point will have a greater influence than neighbors which are further away.
  - *callable*: a user-defined function which accepts an array of distances, and returns an array of the same shape containing the weights.
- **metric** (*{'nan\_euclidean'} or callable, default='nan\_euclidean'*) – Distance metric for searching neighbors. Possible values:
  - *'nan\_euclidean'*
  - *callable* : a user-defined function which conforms to the definition of `_pairwise_callable(X, Y, metric, **kwds)`. The function accepts two arrays, X and Y, and a *missing\_values* keyword in *kwds* and returns a scalar distance value.

#### property missing\_values

The placeholder for the missing values.

#### property n\_neighbors

Number of neighboring samples to use for imputation.

#### property weights

Weight function used in prediction.

#### property metric

Distance metric for searching neighbors.

### skcriteria.preprocessing.increment module

Functionalities to add a value when an array has a zero.

In addition to the main functionality, an MCDA agnostic function is offered to add value to zero on an array along an arbitrary axis.

`skcriteria.preprocessing.increment.add_value_to_zero(arr, value, axis=None)`

Add value if the axis has a value 0.

$$\bar{X}_{ij} = X_{ij} + value$$

#### Parameters

- **arr** (*numpy.ndarray like.*) – A array with values
- **value** (*number*) – Number to add if the axis has a 0.

- **axis** (`int` optional) – Axis along which to operate. By default, flattened input is used.

**Returns**

array with all values  $\geq$  value.

**Return type**

`numpy.ndarray`

**Examples**

```
>>> from skcriteria import add_to_zero

# no zero
>>> mtx = [[1, 2], [3, 4]]
>>> add_to_zero(mtx, value=0.5)
array([[1, 2],
       [3, 4]])

# with zero
>>> mtx = [[0, 1], [2,3]]
>>> add_to_zero(mtx, value=0.5)
array([[ 0.5, 1.5],
       [ 2.5, 3.5]])
```

**class** `skcriteria.preprocessing.increment.AddValueToZero`(*target*, *value=1.0*)

Bases: *SKCMatrixAndWeightTransformerABC*

Add value if the matrix/weight whe has a value 0.

$$\bar{X}_{ij} = X_{ij} + value$$

**property value**

Value to add to the matrix/weight when a zero is found.

**skcriteria.preprocessing.invert\_objectives module**

Implementation of functionalities for convert minimization criteria into maximization ones.

**class** `skcriteria.preprocessing.invert_objectives.SKCOBJECTIVESInverterABC`

Bases: *SKCTransformerABC*

Abstract class capable of invert objectives.

This abstract class require to redefine `_invert`, instead of `_transform_data`.

**class** `skcriteria.preprocessing.invert_objectives.NegateMinimize`

Bases: *SKCOBJECTIVESInverterABC*

Transform all minimization criteria into maximization ones.

The transformations are made by calculating the inverse value of the minimization criteria.  $\min C \equiv \max -C$ .

**class** `skcriteria.preprocessing.invert_objectives.InvertMinimize`

Bases: *SKCObjectivesInverterABC*

Transform all minimization criteria into maximization ones.

The transformations are made by calculating the inverse value of the minimization criteria.  $\min C \equiv \max \frac{1}{C}$

### Notes

All the dtypes of the decision matrix are preserved except the inverted ones that are converted to `numpy.float64`.

**class** `skcriteria.preprocessing.invert_objectives.MinimizeToMaximize(*args, **kwargs)`

Bases: *InvertMinimize*

Transform all minimization criteria into maximization ones.

The transformations are made by calculating the inverse value of the minimization criteria.  $\min C \equiv \max \frac{1}{C}$

Deprecated since version 0.7: Use `skcriteria.preprocessing.invert_objectives.InvertMinimize` instead

### Notes

All the dtypes of the decision matrix are preserved except the inverted ones that are converted to `numpy.float64`.

## `skcriteria.preprocessing.push_negatives` module

Functionalities for remove negatives from criteria.

In addition to the main functionality, an MCDA agnostic function is offered to push negatives values on an array along an arbitrary axis.

`skcriteria.preprocessing.push_negatives.push_negatives(arr, axis)`

Increment the array until all the values are seen  $\geq 0$ .

If an array has negative values this function increment the values proportionally to make all the array positive along an axis.

$$\bar{X}_{ij} = \begin{cases} X_{ij} + \min_{X_{ij}} & \text{if } X_{ij} < 0 \\ X_{ij} & \text{otherwise} \end{cases}$$

### Parameters

- **arr** (`numpy.ndarray` like.) – A array with values
- **axis** (`int` optional) – Axis along which to operate. By default, flattened input is used.

### Returns

array with all values  $\geq 0$ .

### Return type

`numpy.ndarray`

## Examples

```

>>> from skcriteria.preprocess import push_negatives
>>> mtx = [[1, 2], [3, 4]]
>>> mtx_lt0 = [[-1, 2], [3, 4]] # has a negative value

>>> push_negatives(mtx) # array without negatives don't be affected
array([[1, 2],
       [3, 4]])

# all the array is incremented by 1 to eliminate the negative
>>> push_negatives(mtx_lt0)
array([[0, 3],
       [4, 5]])

# by column only the first one (with the negative value) is affected
>>> push_negatives(mtx_lt0, axis=0)
array([[0, 2],
       [4, 4]])

# by row only the first row (with the negative value) is affected
>>> push_negatives(mtx_lt0, axis=1)
array([[0, 3],
       [3, 4]])

```

**class** `skcriteria.preprocessing.push_negatives.PushNegatives(target)`

Bases: `SKCMatrixAndWeightTransformerABC`

Increment the matrix/weights until all the valuer are sean  $\geq 0$ .

If the matrix/weights has negative values this function increment the values proportionally to made all the matrix/weights positive along an axis.

$$\bar{X}_{ij} = \begin{cases} X_{ij} + \min_{X_{ij}} & \text{if } X_{ij} < 0 \\ X_{ij} & \text{otherwise} \end{cases}$$

## `skcriteria.preprocessing.scalers module`

Functionalities for scale values based on different strategies.

In addition to the Transformers, a collection of an MCDA agnostic functions are offered to scale an array along an arbitrary axis.

**class** `skcriteria.preprocessing.scalers.StandardScaler(target, *, with_mean=True, with_std=True)`

Bases: `SKCMatrixAndWeightTransformerABC`

Standardize the dm by removing the mean and scaling to unit variance.

The standard score of a sample  $x$  is calculated as:

$$z = (x - u) / s$$

where  $u$  is the mean of the values, and  $s$  is the standard deviation of the training samples or one if `with_std=False`.

This is a thin wrapper around `sklearn.preprocessing.StandardScaler`.

### Parameters

- **with\_mean** (*bool*, *default=True*) – If True, center the data before scaling.

- **with\_std** (*bool*, *default=True*) – If True, scale the data to unit variance (or equivalently, unit standard deviation).

**property with\_mean**

True if the features will be center before scaling.

**property with\_std**

True if the features will be scaled to the unit variance.

**class** `skcriteria.preprocessing.scalers.MinMaxScaler`(*target*, \*, *clip=False*, *criteria\_range=(0, 1)*)

Bases: *SKCMatrixAndWeightTransformerABC*

Scaler based on the range.

The matrix transformation is given by:

```
X_std = (X - X.min(axis=0)) / (X.max(axis=0) - X.min(axis=0))
X_scaled = X_std * (max - min) + min
```

And the weight transformation:

```
X_std = (X - X.min(axis=None)) / (X.max(axis=None) - X.min(axis=None))
X_scaled = X_std * (max - min) + min
```

If the scaler is configured to work with ‘matrix’ each value of each criteria is divided by the range of that criteria. In other hand if is configure to work with ‘weights’, each value of weight is divided by the range the weights.

This is a thin wrapper around `sklearn.preprocessing.MinMaxScaler`.

**Parameters**

- **criteria\_range** (*tuple* (*min*, *max*), *default=(0, 1)*) – Desired range of transformed data.
- **clip** (*bool*, *default=False*) – Set to True to clip transformed values of held-out data to provided *criteria\_range*.

**property clip**

True if the transformed values will be clipped to held-out the value provided *criteria\_range*.

**property criteria\_range**

Range of transformed data.

**class** `skcriteria.preprocessing.scalers.MaxAbsScaler`(*target*)

Bases: *SKCMatrixAndWeightTransformerABC*

Scaler based on the maximum values.

If the scaler is configured to work with ‘matrix’ each value of each criteria is divided by the maximum value of that criteria. In other hand if is configure to work with ‘weights’, each value of weight is divided by the maximum value the weights.

This estimator scales and translates each criteria individually such that the maximal absolute value of each criteria in the training set will be 1.0. It does not shift/center the data, and thus does not destroy any sparsity.

This is a thin wrapper around `sklearn.preprocessing.MaxAbsScaler`.

**class** `skcriteria.preprocessing.scalers.MaxScaler`(\**args*, \*\**kwargs*)

Bases: *MaxAbsScaler*

Scaler based on the maximum values.

From `skcriteria >= 0.8` this is a thin wrapper around `sklearn.preprocessing.MaxAbsScaler`.

Deprecated since version 0.8: Use `skcriteria.preprocessing.scalers.MaxAbsScaler` instead

`skcriteria.preprocessing.scalers.scale_by_vector(arr, axis=None)`

Divide the array by norm of values defined vector along an axis.

Calculates the set of ratios as the square roots of the sum of squared responses of a given axis as denominators. If *axis* is *None* sum all the array.

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

#### Parameters

- **arr** (`numpy.ndarray` like.) – A array with values
- **axis** (`int` optional) – Axis along which to operate. By default, flattened input is used.

#### Returns

array of ratios

#### Return type

`numpy.ndarray`

#### Examples

```
>>> from skcriteria.preprocess import scale_by_vector
>>> mtx = [[1, 2], [3, 4]]

# ratios with the vector value of the array
>>> scale_by_vector(mtx)
array([[ 0.18257418,  0.36514837],
       [ 0.54772252,  0.73029673]])

# ratios by column
>>> scale_by_vector(mtx, axis=0)
array([[ 0.31622776,  0.44721359],
       [ 0.94868326,  0.89442718]])

# ratios by row
>>> scale_by_vector(mtx, axis=1)
array([[ 0.44721359,  0.89442718],
       [ 0.60000002,  0.80000001]])
```

**class** `skcriteria.preprocessing.scalers.VectorScaler(target)`

Bases: `SKCMatrixAndWeightTransformerABC`

Scaler based on the norm of the vector..

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

If the scaler is configured to work with ‘matrix’ each value of each criteria is divided by the norm of the vector defined by the values of that criteria. In other hand if is configure to work with ‘weights’, each value of weight is divided by the vector defined by the values of the weights.

`skcriteria.preprocessing.scalers.scale_by_sum(arr, axis=None)`

Divide of every value on the array by sum of values along an axis.

$$\bar{X}_{ij} = \frac{X_{ij}}{\sum_{j=1}^m X_{ij}}$$

#### Parameters

- **arr** (`numpy.ndarray` like.) – A array with values
- **axis** (`int` optional) – Axis along which to operate. By default, flattened input is used.

#### Returns

array of ratios

#### Return type

`numpy.ndarray`

#### Examples

```
>>> from skcriteria.preprocess import scale_by_sum
>>> mtx = [[1, 2], [3, 4]]

>>> scale_by_sum(mtx) # ratios with the sum of the array
array([[ 0.1      ,  0.2      ],
       [ 0.30000001,  0.40000001]])

# ratios with the sum of the array by column
>>> scale_by_sum(mtx, axis=0)
array([[ 0.25      ,  0.33333334],
       [ 0.75      ,  0.66666669]])

# ratios with the sum of the array by row
>>> scale_by_sum(mtx, axis=1)
array([[ 0.33333334,  0.66666669],
       [ 0.42857143,  0.5714286 ]])
```

`class skcriteria.preprocessing.scalers.SumScaler(target)`

Bases: `SKCMatrixAndWeightTransformerABC`

Scaler based on the total sum of values.

$$\bar{X}_{ij} = \frac{X_{ij}}{\sum_{j=1}^m X_{ij}}$$

If the scaler is configured to work with ‘matrix’ each value of each criteria is divided by the total sum of all the values of that criteria. In other hand if is configure to work with ‘weights’, each value of weight is divided by the total sum of all the weights.

`skcriteria.preprocessing.scalers.matrix_scale_by_cenit_distance(matrix, objectives)`

Calculate a scores with respect to an ideal and anti-ideal alternative.

For every criterion  $f$  of this multicriteria problem we define a membership function  $x_j$  mapping the values of  $f_j$  to the interval  $[0, 1]$ .



The result score  $x_{aj}$  is close to the ideal value  $f_j^*$ , which is the best performance in criterion , and far from the anti-ideal value  $f_j^*$ , which is the worst performance in criterion  $j$ . Both ideal and anti-ideal, are achieved by at least one of the alternatives under consideration.

$$x_{aj} = \frac{f_j(a) - f_j^*}{f_j^* - f_j^*}$$

**class** `skcriteria.preprocessing.scalers.CenitDistanceMatrixScaler`

Bases: `SKCTransformerABC`

Relative scores with respect to an ideal and anti-ideal alternative.

For every criterion  $f$  of this multicriteria problem we define a membership function  $x_j$  mapping the values of  $f_j$  to the interval  $[0, 1]$ .

The result score  $x_{aj}$  is close to the ideal value  $f_j^*$ , which is the best performance in criterion , and far from the anti-ideal value  $f_j^*$ , which is the worst performance in criterion  $j$ . Both ideal and anti-ideal, are achieved by at least one of the alternatives under consideration.

$$x_{aj} = \frac{f_j(a) - f_j^*}{f_j^* - f_j^*}$$

## References

[Diakoulaki et al., 1995]

## `skcriteria.preprocessing.weighters` module

Functionalities for weight the criteria.

In addition to the main functionality, an MCDA agnostic function is offered to calculate weights to a matrix along an arbitrary axis.

**class** `skcriteria.preprocessing.weighters.SKCWeighterABC`

Bases: `SKCTransformerABC`

Abstract class capable of determine the weights of the matrix.

This abstract class require to redefine `_weight_matrix`, instead of `_transform_data`.

`skcriteria.preprocessing.weighters.equal_weights(matrix, base_value=1)`

Use the same weights for all criteria.

The result values are normalized by the number of columns.

$$w_j = \frac{base\_value}{m}$$

Where  $m$  is the number os columns/criteria in matrix.

### Parameters

- **matrix** (`numpy.ndarray` like.) – The matrix of alternatives on which to calculate weights.
- **base\_value** (`int` or `float`.) – Value to be normalized by the number of criteria to create the weights.

### Returns

array of weights

### Return type

`numpy.ndarray`

## Examples

```
>>> from skcriteria.preprocess import equal_weights
>>> mtx = [[1, 2], [3, 4]]

>>> equal_weights(mtx)
array([0.5, 0.5])
```

**class** `skcriteria.preprocessing.weighters.EqualWeighter`(*base\_value=1.0*)

Bases: *SKCWeighterABC*

Assigns the same weights to all criteria.

The algorithm calculates the weights as the ratio of *base\_value* by the total criteria.

### property `base_value`

Value to be normalized by the number of criteria.

`skcriteria.preprocessing.weighters.std_weights`(*matrix*)

Calculate weights as the standard deviation of each criterion.

The result is normalized by the number of columns.

$$w_j = \frac{s_j}{m}$$

Where  $m$  is the number of columns/criteria in matrix.

### Parameters

**matrix** (numpy.ndarray like.) – The matrix of alternatives on which to calculate weights.

### Returns

array of weights

### Return type

numpy.ndarray

## Examples

```
>>> from skcriteria.preprocess import std_weights
>>> mtx = [[1, 2], [3, 4]]

>>> std_weights(mtx)
array([0.5, 0.5])
```

**class** `skcriteria.preprocessing.weighters.StdWeighter`

Bases: *SKCWeighterABC*

Set as weight the normalized standard deviation of each criterion.

`skcriteria.preprocessing.weighters.entropy_weights`(*matrix*)

Calculate the weights as the complement of the entropy of each criterion.

It uses the underlying `scipy.stats.entropy` function which assumes that the values of the criteria are probabilities of a distribution.

The logarithmic base to use is the number of rows/alternatives in the matrix.

This routine will normalize the sum of the weights to 1.

**See also:**

**scipy.stats.entropy**

Calculate the entropy of a distribution for given probability values.

**class** skcriteria.preprocessing.weighters.**EntropyWeighter**

Bases: *SKCWeighterABC*

Assigns the complement of the entropy of the criteria as weights.

It uses the underlying `scipy.stats.entropy` function which assumes that the values of the criteria are probabilities of a distribution.

The logarithmic base to use is the number of rows/alternatives in the matrix.

This transformer will normalize the sum of the weights to 1.

**See also:**

**scipy.stats.entropy**

Calculate the entropy of a distribution for given probability values.

**skcriteria.preprocessing.weighters.pearson\_correlation**(*arr*)

Return Pearson product-moment correlation coefficients.

This function is a thin wrapper of `numpy.corrcoef`.

Deprecated since version 0.8: Please use `pd.DataFrame(arr.T).correlation('pearson')`

**Parameters**

**arr** (*array like*) – A 1-D or 2-D array containing multiple variables and observations. Each row of *arr* represents a variable, and each column a single observation of all those variables.

**Returns**

**R** – The correlation coefficient matrix of the variables.

**Return type**

`numpy.ndarray`

**See also:**

**numpy.corrcoef**

Return Pearson product-moment correlation coefficients.

**skcriteria.preprocessing.weighters.spearman\_correlation**(*arr*)

Calculate a Spearman correlation coefficient.

This function is a thin wrapper of `scipy.stats.spearmanr`.

Deprecated since version 0.8: Please use `pd.DataFrame(arr.T).correlation('spearman')`

**Parameters**

**arr** (*array like*) – A 1-D or 2-D array containing multiple variables and observations. Each row of *arr* represents a variable, and each column a single observation of all those variables.

**Returns**

**R** – The correlation coefficient matrix of the variables.

**Return type**

`numpy.ndarray`

**See also:**

**scipy.stats.spearmanr**

Calculate a Spearman correlation coefficient with associated p-value.

```
skcriteria.preprocessing.weighters.critic_weights(matrix, objectives, correlation='pearson',
                                                scale=True)
```

Execute the CRITIC method without any validation.

```
class skcriteria.preprocessing.weighters.CRITIC(correlation='pearson', scale=True)
```

Bases: *SKCWeighterABC*

CRITIC (CRiteria Importance Through Intercriteria Correlation).

The method aims at the determination of objective weights of relative importance in MCDM problems. The weights derived incorporate both contrast intensity and conflict which are contained in the structure of the decision problem.

**Parameters**

- **correlation** (*str* ["pearson", "spearman", "kendall"] or callable.) – This is the correlation function used to evaluate the discordance between two criteria. In other words, what conflict does one criterion a criterion with respect to the decision made by the other criteria. By default the pearson correlation is used, and the spearman and kendall correlation is also available implemented. It is also possible to provide a callable with input two 1d arrays and returning a float. Note that the returned matrix from corr will have 1 along the diagonals and will be symmetric regardless of the callable's behavior
- **scale** (bool (default True)) – True if it is necessary to scale the data with skcriteria.preprocessing.matrix\_scale\_by\_cenit\_distance prior to calculating the correlation

**Warning:****UserWarning:**

If some objective is to minimize. The original paper only suggests using it against maximization criteria, but there is no real mathematical constraint to use it for minimization.

**References**

[Diakoulaki et al., 1995]

```
CORRELATION = ('pearson', 'spearman', 'kendall')
```

**property scale**

Return if it is necessary to scale the data.

**property correlation**

Correlation function.

```
class skcriteria.preprocessing.weighters.Critic(*args, **kwargs)
```

Bases: *CRITIC*

CRITIC (CRiteria Importance Through Intercriteria Correlation).

The method aims at the determination of objective weights of relative importance in MCDM problems. The weights derived incorporate both contrast intensity and conflict which are contained in the structure of the decision problem.

Deprecated since version 0.8: Use skcriteria.preprocessing.weighters.CRITIC instead

**Parameters**

- **correlation** (*str* [`"pearson"`, `"spearman"`, `"kendall"`] or *callable*.) – This is the correlation function used to evaluate the discordance between two criteria. In other words, what conflict does one criterion a criterion with respect to the decision made by the other criteria. By default the `pearson` correlation is used, and the `spearman` and `kendall` correlation is also available implemented. It is also possible to provide a callable with input two 1d arrays and returning a float. Note that the returned matrix from `corr` will have 1 along the diagonals and will be symmetric regardless of the callable's behavior
- **scale** (bool (default `True`)) – True if it is necessary to scale the data with `skcriteria.preprocessing.matrix_scale_by_cenit_distance` prior to calculating the correlation

**Warning:****UserWarning:**

If some objective is to minimize. The original paper only suggests using it against maximization criteria, but there is no real mathematical constraint to use it for minimization.

**References**

[Diakoulaki et al., 1995]

**4.3.4 `skcriteria.cmp` package**

Utilities for a-posteriori analysis of experiments.

**`skcriteria.cmp.ranks_cmp` module**

Ranking comparison routines.

**class** `skcriteria.cmp.ranks_cmp.RanksComparator`(*ranks*)

Bases: `SKMethodABC`

Rankings comparator object.

This class is intended to contain a collection of rankings on which you want to do comparative analysis.

All rankings must have exactly the same alternatives, although their order may vary.

All methods support the `untied` parameter, which serves to untie rankings in case there are results that can assign more than one alternative to the same position (e.g. `ELECTRE2`).

**Parameters**

**ranks** (*list*) – List of (name, ranking) tuples of `skcriteria.madm.RankResult` with the same alternatives.

**See also:**

**`skcriteria.cmp.mkrank_cmp`**

Convenience function for simplified ranks comparator construction.

**property ranks**

List of ranks in the comparator.

**property named\_ranks**

Dictionary-like object, with the following attributes.

Read-only attribute to access any rank parameter by user given name. Keys are ranks names and values are ranks parameters.

**to\_dataframe(\*, untied=False)**

Convert the entire RanksComparator into a dataframe.

The alternatives are the rows, and the different rankings are the columns.

**Parameters**

**untied** (bool, default False) – If it is True and any ranking has ties, the `RankResult.untied_rank_` property is used to assign each alternative a single ranked order. On the other hand, if it is False the rankings are used as they are.

**Returns**

A RanksComparator as pandas DataFrame.

**Return type**

`pd.DataFrame`

**corr(\*, untied=False, \*\*kwargs)**

Compute pairwise correlation of rankings, excluding NA/null values.

By default the pearson correlation coefficient is used.

Please check the full documentation of a `pandas.DataFrame.corr()` method for details about the implementation.

**Parameters**

- **untied** (bool, default False) – If it is True and any ranking has ties, the `RankResult.untied_rank_` property is used to assign each alternative a single ranked order. On the other hand, if it is False the rankings are used as they are.
- **kwargs** – Other keyword arguments are passed to the `pandas.DataFrame.corr()` method.

**Returns**

A DataFrame with the correlation between rankings.

**Return type**

`pd.DataFrame`

**cov(\*, untied=False, \*\*kwargs)**

Compute pairwise covariance of rankings, excluding NA/null values.

Please check the full documentation of a `pandas.DataFrame.cov()` method for details about the implementation.

**Parameters**

- **untied** (bool, default False) – If it is True and any ranking has ties, the `RankResult.untied_rank_` property is used to assign each alternative a single ranked order. On the other hand, if it is False the rankings are used as they are.
- **kwargs** – Other keyword arguments are passed to the `pandas.DataFrame.cov()` method.

**Returns**

A DataFrame with the covariance between rankings.

**Return type**

pd.DataFrame

**r2\_score**(\*, *untied=False*, *\*\*kwargs*)

Compute pairwise coefficient of determination regression score function of rankings, excluding NA/null values.

Best possible score is 1.0 and it can be negative (because the model can be arbitrarily worse).

Please check the full documentation of a `sklearn.metrics.r2_score` function for details about the implementation and the behaviour.

**Parameters**

- **untied** (bool, default `False`) – If it is `True` and any ranking has ties, the `RankResult.untied_rank_` property is used to assign each alternative a single ranked order. On the other hand, if it is `False` the rankings are used as they are.
- **kwargs** – Other keyword arguments are passed to the `sklearn.metrics.r2_score()` function.

**Returns**

A DataFrame with the coefficient of determination between rankings.

**Return type**

pd.DataFrame

**distance**(\*, *untied=False*, *metric='hamming'*, *\*\*kwargs*)

Compute pairwise distance between rankings.

By default the 'hamming' distance is used, which is simply the proportion of disagreeing components in Two rankings.

Please check the full documentation of a `scipy.spatial.distance.pdist` function for details about the implementation and the behaviour.

**Parameters**

- **untied** (bool, default `False`) – If it is `True` and any ranking has ties, the `RankResult.untied_rank_` property is used to assign each alternative a single ranked order. On the other hand, if it is `False` the rankings are used as they are.
- **metric** (str or function, default `"hamming"`) – The distance metric to use. The distance function can be 'braycurtis', 'canberra', 'chebyshev', 'cityblock', 'correlation', 'cosine', 'dice', 'euclidean', 'hamming', 'jaccard', 'jensenshannon', 'kulczynskil', 'mahalanobis', 'matching', 'minkowski', 'rogerstanimoto', 'russellrao', 'seuclidean', 'sokalmichener', 'sokalsneath', 'sqeuclidean', 'yule'.
- **kwargs** – Other keyword arguments are passed to the `scipy.spatial.distance.pdist()` function.

**Returns**

A DataFrame with the distance between rankings.

**Return type**

pd.DataFrame

**property plot**

Plot accessor.

**class** `skcriteria.cmp.ranks_cmp.RanksComparatorPlotter`(*ranks\_cmp*)

Bases: `AccessorABC`

RanksComparator plot utilities.

Kind of plot to produce:

- ‘flow’ : Changes in the rankings of the alternatives as flow lines (default)
- ‘reg’ : Pairwise rankings data and a linear regression model fit plot.
- ‘heatmap’ : Rankings as a color-encoded matrix.
- ‘corr’ : Pairwise correlation of rankings as a color-encoded matrix.
- ‘cov’ : Pairwise covariance of rankings as a color-encoded matrix.
- ‘r2\_score’ : Pairwise coefficient of determination regression score function of rankings as a color-encoded matrix.
- ‘distance’ : Pairwise distance between rankings as a color-encoded matrix.
- ‘box’ : Box-plot of rankings with respect to alternatives
- ‘bar’ : Ranking of alternatives by method with vertical bars.
- ‘barh’ : Ranking of alternatives by method with horizontal bars.

**flow**(\*, *untied=False*, *grid\_kws=None*, *\*\*kwargs*)

Represents changes in the rankings of the alternatives as lines flowing through the ranking-methods.

#### Parameters

- **untied** (bool, default `False`) – If it is `True` and any ranking has ties, the `RankResult.untied_rank_` property is used to assign each alternative a single ranked order. On the other hand, if it is `False` the rankings are used as they are.
- **grid\_kws** (*dict* or *None*) – Dict with keyword arguments passed to `matplotlib.axes.plt.Axes.grid`
- **kwargs** – Other keyword arguments are passed to the `seaborn.lineplot()` function, except for data, estimator and sort.

#### Return type

`matplotlib.axes.Axes` or `numpy.ndarray` of them

**reg**(\*, *untied=False*, *r2=True*, *palette=None*, *legend=True*, *r2\_fmt='.2g'*, *r2\_kws=None*, *\*\*kwargs*)

Plot a pairwise rankings data and a linear regression model fit.

#### Parameters

- **untied** (bool, default `False`) – If it is `True` and any ranking has ties, the `RankResult.untied_rank_` property is used to assign each alternative a single ranked order. On the other hand, if it is `False` the rankings are used as they are.
- **r2** (bool, default `True`) – If `True`, the coefficient of determination results are added to the regression legend.
- **palette** (`matplotlib/seaborn` color palette, default `None`) – Set of colors for mapping the hue variable.
- **legend** (bool, default `True`) – If `False`, suppress the legend for semantic variables.
- **r2\_fmt** (str, default `"2.g"`) – String formatting code to use when adding the coefficient of determination.



- **r2\_kws** (*dict* or *None*) – Dict with keywords arguments passed to `sklearn.metrics.r2_score()` function.
- **kwargs** – Other keyword arguments are passed to the `seaborn.lineplot()` function.

**Return type**

`matplotlib.axes.Axes` or `numpy.ndarray` of them

**heatmap**(\**, untied=False, \*\*kwargs*)

Plot the rankings as a color-encoded matrix.

**Parameters**

- **untied** (bool, default `False`) – If it is `True` and any ranking has ties, the `RankResult.untied_rank_` property is used to assign each alternative a single ranked order. On the other hand, if it is `False` the rankings are used as they are.
- **kwargs** – Other keyword arguments are passed to the `seaborn.heatmap()` function.

**Return type**

`matplotlib.axes.Axes` or `numpy.ndarray` of them

**corr**(\**, untied=False, corr\_kws=None, \*\*kwargs*)

Plot the pairwise correlation of rankings as a color-encoded matrix.

By default the pearson correlation coefficient is used.

**Parameters**

- **untied** (bool, default `False`) – If it is `True` and any ranking has ties, the `RankResult.untied_rank_` property is used to assign each alternative a single ranked order. On the other hand, if it is `False` the rankings are used as they are.
- **corr\_kws** (*dict* or *None*) – Dict with keywords arguments passed the `pandas.DataFrame.corr()` method.
- **kwargs** – Other keyword arguments are passed to the `seaborn.heatmap()` function.

**Return type**

`matplotlib.axes.Axes` or `numpy.ndarray` of them

**cov**(\**, untied=False, cov\_kws=None, \*\*kwargs*)

Plot the pairwise covariance of rankings as a color-encoded matrix.

**Parameters**

- **untied** (bool, default `False`) – If it is `True` and any ranking has ties, the `RankResult.untied_rank_` property is used to assign each alternative a single ranked order. On the other hand, if it is `False` the rankings are used as they are.
- **cov\_kws** (*dict* or *None*) – Dict with keywords arguments passed the `pandas.DataFrame.cov()` method.
- **kwargs** – Other keyword arguments are passed to the `seaborn.heatmap()` function.

**Return type**

`matplotlib.axes.Axes` or `numpy.ndarray` of them

**r2\_score**(*untied=False, r2\_kws=None, \*\*kwargs*)

Plot the pairwise coefficient of determination regression score function of rankings as a color-encoded matrix.

**Parameters**

- **untied** (bool, default False) – If it is True and any ranking has ties, the `RankResult.untied_rank_` property is used to assign each alternative a single ranked order. On the other hand, if it is False the rankings are used as they are.
- **cov\_kws** (*dict* or None) – Dict with keywords arguments passed the `pandas.DataFrame.cov()` method.
- **kwargs** – Other keyword arguments are passed to the `seaborn.heatmap()` function.

**Return type**

`matplotlib.axes.Axes` or `numpy.ndarray` of them

**distance**(\**, untied=False, metric='hamming', distance\_kws=None, \*\*kwargs*)

Plot the pairwise distance between rankings as a color-encoded matrix.

By default the 'hamming' distance is used, which is simply the proportion of disagreeing components in Two rankings.

**Parameters**

- **untied** (bool, default False) – If it is True and any ranking has ties, the `RankResult.untied_rank_` property is used to assign each alternative a single ranked order. On the other hand, if it is False the rankings are used as they are.
- **metric** (str or function, default "hamming") – The distance metric to use. The distance function can be 'braycurtis', 'canberra', 'chebyshev', 'cityblock', 'correlation', 'cosine', 'dice', 'euclidean', 'hamming', 'jaccard', 'jensenshannon', 'kulczynski1', 'mahalanobis', 'matching', 'minkowski', 'rogerstanimoto', 'russellrao', 'seuclidean', 'sokalmichener', 'sokalsneath', 'sqeuclidean', 'yule'.
- **distance\_kws** (*dict* or None) – Dict with keywords arguments passed the `scipy.spatial.distance.pdist` function
- **kwargs** – Other keyword arguments are passed to the `seaborn.heatmap()` function.

**Return type**

`matplotlib.axes.Axes` or `numpy.ndarray` of them

**box**(\**, untied=False, \*\*kwargs*)

Draw a boxplot to show rankings with respect to alternatives.

**Parameters**

- **untied** (bool, default False) – If it is True and any ranking has ties, the `RankResult.untied_rank_` property is used to assign each alternative a single ranked order. On the other hand, if it is False the rankings are used as they are.
- **kwargs** – Other keyword arguments are passed to the `seaborn.boxplot()` function.

**Return type**

`matplotlib.axes.Axes` or `numpy.ndarray` of them

**bar**(\**, untied=False, \*\*kwargs*)

Draw plot that presents ranking of alternatives by method with vertical bars.

**Parameters**

- **untied** (bool, default False) – If it is True and any ranking has ties, the `RankResult.untied_rank_` property is used to assign each alternative a single ranked order. On the other hand, if it is False the rankings are used as they are.
- **kwargs** – Other keyword arguments are passed to the `pandas.DataFrame.plot.bar()` method.

**Return type**

matplotlib.axes.Axes or numpy.ndarray of them

`barh(*, untied=False, **kwargs)`

Draw plot that presents ranking of alternatives by method with horizontal bars.

**Parameters**

- **untied** (bool, default False) – If it is True and any ranking has ties, the `RankResult.untied_rank_` property is used to assign each alternative a single ranked order. On the other hand, if it is False the rankings are used as they are.
- **kwargs** – Other keyword arguments are passed to the `pandas.DataFrame.plot.barh()` method.

**Return type**

matplotlib.axes.Axes or numpy.ndarray of them

`skcriteria.cmp.ranks_cmp.mkrank_cmp(*ranks)`

Construct a RankComparator from the given rankings.

This is a shorthand for the RankComparator constructor; it does not require, and does not permit, naming the estimators. Instead, their names will be set to the method attribute of the rankings automatically.

**Parameters**

**\*ranks** (*list of RankResult objects*) – List of the scikit-criteria RankResult objects.

**Returns**

**rcmp** – Returns a scikit-criteria *RanksComparator* object.

**Return type**

*RanksComparator*

### 4.3.5 skcriteria.datasets package

The *skcriteria.datasets* module includes utilities to load datasets.

`skcriteria.datasets.load_simple_stock_selection()`

Simple stock selection decision matrix.

This matrix was designed primarily for teaching and evaluating the behavior of an experiment.

Among the data we can find: two maximization criteria (ROE, CAP), one minimization criterion (RI), dominated alternatives (FX), and one alternative with an outlier criterion (ROE, MM = 1).

Although the criteria and alternatives are original from the authors of Scikit-Criteria, the numerical values were extracted at some point from a somewhere which we have forgotten.

Description:

In order to decide to buy a series of stocks, a company studied 5 candidate investments: PE, JN, AA, FX, MM and GN. The finance department decides to consider the following criteria for selection:

1. ROE (Max): Return % for each monetary unit invested.
2. CAP (Max): Years of market capitalization.
3. RI (Min): Risk of the stock.

`skcriteria.datasets.load_van2021evaluation(window_size=7)`

Dataset extracted from from historical time series cryptocurrencies.

This dataset is extracted from:

Van Heerden, N., Cabral, J. y Luczywo, N. (2021). Evaluación de la importancia de criterios para la selección de criptomonedas. XXXIV ENDIO - XXXII EPIO Virtual 2021, Argentina.

The nine available alternatives are based on the ranking of the 20 cryptocurrencies with the largest market capitalization calculated on the basis of circulating supply, according to information retrieved from “Cryptocurrency Historical Prices” retrieved on July 21st, 2021, from there only the coins with complete data between October 9th, 2018 to July 6th of 2021, excluding stable-coins, since they maintain a stable price and therefore do not carry associated yields; the alternatives that met these requirements turned out to be: Cardano (ADA), Binance coin (BNB), Bitcoin (BTC), Dogecoin (DOGE), Ethereum (ETH), Chainlink (LINK), Litecoin (LTC), Stellar (XLM) and Ripple (XRP).

Two decision matrices were created for two sizes of overlapping moving windows: 7 and 15 days. Six criteria were defined on these windows that seek to represent returns and risks:

- $xRV$  - average Window return ( $\bar{x}RV$ ) - Maximize: is the average of the differences between the closing price of the cryptocurrency on the last day and the first day of each window, divided by the price on the first day.
- $sRV$  - window return deviation ( $sRV$ ) - Minimize: is the standard deviation of window return. The greater the deviation, the returns within the windows have higher variance and are unstable.
- $xVV$  - average of the volume of the window ( $\bar{x}VV$ ) - Maximize: it is the average of the summations of the transaction amount of the cryptocurrency in dollars in each window, representing a liquidity measure of the asset.
- $sVV$  - window volume deviation ( $sVV$ ) - Minimize: it is the deviation of the window volumes. The greater the deviation, the volumes within the windows have higher variance and are unstable.
- $xR^2$  - mean of the correlation coefficient ( $\bar{x}R^2$ ) - Maximize: it is the mean of the  $R^2$  of the fit of the linear trends with respect to the data. It is a measure that defines how well it explains that linear trend to the data within the window.
- $xm$  - mean of the slope ( $\bar{x}m$ ) - Maximize: it is the mean of the slope of the linear trend between the closing prices in dollars and the volumes traded in dollars of the cryptocurrency within each window.

#### Parameters

**windows\_size** (7 o 15, default 7) – If the decision matrix based on 7 or 15 day overlapping moving windows is desired.

## References

[VanHeerden et al., 2021b] [VanHeerden et al., 2021a] [Rajkumar, 2021]

### 4.3.6 skcriteria.pipeline module

The Module implements utilities to build a composite decision-maker.

**class** `skcriteria.pipeline.SKCPipeline`(steps)

Bases: `SKMethodABC`

Pipeline of transforms with a final decision-maker.

Sequentially apply a list of transforms and a final decisionmaker. Intermediate steps of the pipeline must be ‘transforms’, that is, they must implement *transform* method.

The final decision-maker only needs to implement *evaluate*.

The purpose of the pipeline is to assemble several steps that can be applied together while setting different parameters.

**Parameters**

**steps** (*list*) – List of (name, transform) tuples (implementing evaluate/transform) that are chained, in the order in which they are chained, with the last object an decision-maker.

**See also:**

*skcriteria.pipeline.mkpipe*

Convenience function for simplified pipeline construction.

**property steps**

List of steps of the pipeline.

**property named\_steps**

Dictionary-like object, with the following attributes.

Read-only attribute to access any step parameter by user given name. Keys are step names and values are steps parameters.

**evaluate**(*dm*)

Run the all the transformers and the decision maker.

**Parameters**

**dm** (`skcriteria.data.DecisionMatrix`) – Decision matrix on which the result will be calculated.

**Returns**

**r** – Whatever the last step (decision maker) returns from their evaluate method.

**Return type**

Result

**transform**(*dm*)

Run the all the transformers.

**Parameters**

**dm** (`skcriteria.data.DecisionMatrix`) – Decision matrix on which the transformations will be applied.

**Returns**

**dm** – Transformed decision matrix.

**Return type**

`skcriteria.data.DecisionMatrix`

`skcriteria.pipeline.mkpipe`(\*steps)

Construct a Pipeline from the given transformers and decision-maker.

This is a shorthand for the SKCPipeline constructor; it does not require, and does not permit, naming the estimators. Instead, their names will be set to the lowercase of their types automatically.

**Parameters**

**\*steps** (*list of transformers and decision-maker object*) – List of the scikit-criteria transformers and decision-maker that are chained together.

**Returns**

**p** – Returns a scikit-criteria *SKCPipeline* object.

**Return type***SKCPipeline*

### 4.3.7 `skcriteria.utils` package

Utilities for `skcriteria`.

#### `skcriteria.utils.accabc` module

Accessor base class.

**class** `skcriteria.utils.accabc.AccessorABC`

Bases: `ABC`

Generalization of the accessor idea for use in `scikit-criteria`.

Instances of this class are callable and accept as the first parameter 'kind' the name of a method to be executed followed by all the all the parameters of this method.

If 'kind' is None, the method defined in the class variable '`_default_kind`' is used.

The last two considerations are that 'kind', cannot be a private method and that all subclasses of the method and that all `AccessorABC` subclasses have to redefine '`_default_kind`'.

#### `skcriteria.utils.bunch` module

Container object exposing keys as attributes.

**class** `skcriteria.utils.bunch.Bunch`(*name*, *data*)

Bases: `Mapping`

Container object exposing keys as attributes.

Concept based on the `sklearn.utils.Bunch`.

Bunch objects are sometimes used as an output for functions and methods. They extend dictionaries by enabling values to be accessed by key, `bunch["value_key"]`, or by an attribute, `bunch.value_key`.

#### Examples

```
>>> b = SKCBunch("data", {"a": 1, "b": 2})
>>> b
data({a, b})
>>> b['b']
2
>>> b.b
2
>>> b.a = 3
>>> b['a']
3
>>> b.c = 6
>>> b['c']
6
```

## skcriteria.utils.cmanagers module

Multiple context managers to use inside scikit-criteria.

`skcriteria.utils.cmanagers.df_temporal_header(df, header, name=None)`

Temporarily replaces a DataFrame columns names.

Optionally also assign another name to the columns.

### Parameters

- **header** (*sequence*) – The new names of the columns.
- **name** (*str or None (default None)*) – New name for the index containing the columns in the DataFrame. If ‘None’ the original name of the columns present in the DataFrame is preserved.

## skcriteria.utils.deprecate module

Multiple decorator to use inside scikit-criteria.

**exception** `skcriteria.utils.deprecate.SKCriteriaDeprecationWarning`

Bases: `DeprecationWarning`

Skcriteria deprecation warning.

**exception** `skcriteria.utils.deprecate.SKCriteriaFutureWarning`

Bases: `FutureWarning`

Skcriteria future warning.

`skcriteria.utils.deprecate.deprecated(*, reason, version)`

Mark functions, classes and methods as deprecated.

It will result in a warning being emitted when the object is called, and the “deprecated” directive was added to the docstring.

### Parameters

- **reason** (*str*) – Reason message which documents the deprecation in your library.
- **version** (*str*) – Version of your project which deprecates this feature. If you follow the [Semantic Versioning](#), the version number has the format “MAJOR.MINOR.PATCH”.

## Notes

This decorator is a thin layer over `deprecated.deprecated()`.

Check: [<github https://pypi.org/project/Deprecated/>](https://github.com/https://pypi.org/project/Deprecated/)\_\_

`skcriteria.utils.deprecate.will_change(*, reason, version)`

Mark functions, classes and methods as “to be changed”.

It will result in a warning being emitted when the object is called, and the “deprecated” directive was added to the docstring.

### Parameters

- **reason** (*str*) – Reason message which documents the “to be changed” in your library.
- **version** (*str*) – Version of your project which marks as this feature. If you follow the [Semantic Versioning](#), the version number has the format “MAJOR.MINOR.PATCH”.

## Notes

This decorator is a thin layer over `deprecated.deprecated()`.

Check: <github <https://pypi.org/project/Deprecated/>>\_\_

## `skcriteria.utils.doctools` module

Multiple decorator to use inside scikit-criteria.

`skcriteria.utils.doctools.doc_inherit(parent, warn_class=True)`

Inherit the 'parent' docstring.

Returns a function/method decorator that, given parent, updates the docstring of the decorated function/method based on the *numpy* style and the corresponding attribute of parent.

### Parameters

- **parent** (*Union[str, Any]*) – The docstring, or object of which the docstring is utilized as the parent docstring during the docstring merge.
- **warn\_class** (*bool*) – If it is true, and the decorated is a class, it throws a warning since there are some issues with inheritance of documentation in classes.

## Notes

This decorator is a thin layer over `custom_inherit.doc_inherit decorator()`.

Check: <github [https://github.com/rsokl/custom\\_inherit](https://github.com/rsokl/custom_inherit)>\_\_

## `skcriteria.utils.lp` module

Utilities for linnear programming based on PuLP.

This file contains an abstraction class to manipulate in a more OOP way the underlining PuLP model

`skcriteria.utils.lp.is_solver_available(solver)`

Return True if the solver is available.

**class** `skcriteria.utils.lp.Float(name, low=None, up=None, *args, **kwargs)`

Bases: `_Var`

`pulp.LpVariable` with `pulp.LpContinuous` category.

## Example

This two codes are equivalent.

```
x = pulp.LpVariable("x", cat=pulp.LpContinuous) # pure PuLP
x = lp.Float("x") # skcriteria.utils.lp version
```

```
var_type = 'Continuous'
```

**class** `skcriteria.utils.lp.Int(name, low=None, up=None, *args, **kwargs)`

Bases: `_Var`

`pulp.LpVariable` with `pulp.LpInteger` category.



### Example

This two codes are equivalent.

```
x = pulp.LpVariable("x", cat=pulp.LpInteger) # pure PuLP
x = lp.Int("x") # skcriteria.utils.lp version
```

```
var_type = 'Integer'
```

```
class skcriteria.utils.lp.Bool(name, low=None, up=None, *args, **kwargs)
```

Bases: `_Var`

`pulp.LpVariable` with `pulp.LpBinary` category.

### Example

This two codes are equivalent.

```
x = pulp.LpVariable("x", cat=pulp.LpBinary) # pure PuLP
x = lp.Bool("x") # skcriteria.utils.lp version
```

```
var_type = 'Binary'
```

```
class skcriteria.utils.lp.Minimize(z, name='no-name', solver=None, **solver_kwds)
```

Bases: `_LPBase`

Creates a Minimize LP problem with a way better syntax than PuLP.

#### Parameters

- **z** (`LpAffineExpression`) – A linear combination of `LpVariables`.
- **name** (`str` (`default="no-name"`)) – Name of the problem.
- **solver** (`None`, `str` or any `pulp.LpSolver` instance (`default=None`)) – Solver of the problem. If it's `None`, the default solver is used. `PULP` is an alias of `None`.
- **solver\_kwds** (`dict`) – Dictionary of keyword arguments for the solver.

### Example

```
# variable declaration
x0 = lp.Float("x0", low=0)
x1 = lp.Float("x1", low=0)
x2 = lp.Float("x2", low=0)

# model
model = lp.Maximize( # or lp.Minimize
    z=250 * x0 + 130 * x1 + 350 * x2
)

# constraints
model.subject_to(
    120 * x0 + 200 * x1 + 340 * x2 <= 500,
    -20 * x0 + -40 * x1 + -15 * x2 <= -15,
```

(continues on next page)

(continued from previous page)

```

    800 * x0 + 1000 * x1 + 600 * x2 <= 1000,
)

```

Also you can create the model and the constraints in one “line”.

```

model = lp.Maximize( # or lp.Minimize
    z=250 * x0 + 130 * x1 + 350 * x2, solver=solver
).subject_to(
    120 * x0 + 200 * x1 + 340 * x2 <= 500,
    -20 * x0 + -40 * x1 + -15 * x2 <= -15,
    800 * x0 + 1000 * x1 + 600 * x2 <= 1000,
)

```

```
sense = 1
```

```
class skcriteria.utils.lp.Maximize(z, name='no-name', solver=None, **solver_kwds)
```

Bases: `_LPBase`

Creates a Maximize LP problem with a way better syntax than PuLP.

#### Parameters

- **z** (`LpAffineExpression`) – A linear combination of `LpVariables`.
- **name** (`str` (default="no-name")) – Name of the problem.
- **solver** (`None`, `str` or any `pulp.LpSolver` instance (default=`None`)) – Solver of the problem. If it's `None`, the default solver is used. PULP is an alias os `None`.
- **solver\_kwds** (`dict`) – Dictionary of keyword arguments for the solver.

#### Example

```

# variable declaration
x0 = lp.Float("x0", low=0)
x1 = lp.Float("x1", low=0)
x2 = lp.Float("x2", low=0)

# model
model = lp.Maximize( # or lp.Minimize
    z=250 * x0 + 130 * x1 + 350 * x2
)

# constraints
model.subject_to(
    120 * x0 + 200 * x1 + 340 * x2 <= 500,
    -20 * x0 + -40 * x1 + -15 * x2 <= -15,
    800 * x0 + 1000 * x1 + 600 * x2 <= 1000,
)

```

Also you can create the model and the constraints in one “line”.

```

model = lp.Maximize( # or lp.Minimize
    z=250 * x0 + 130 * x1 + 350 * x2, solver=solver

```

(continues on next page)

(continued from previous page)

```

).subject_to(
    120 * x0 + 200 * x1 + 340 * x2 <= 500,
    -20 * x0 + -40 * x1 + -15 * x2 <= -15,
    800 * x0 + 1000 * x1 + 600 * x2 <= 1000,
)

```

```
sense = -1
```

### skcriteria.utils.rank module

Functions for calculate and compare ranks (ordinal series).

`skcriteria.utils.rank.rank_values(arr, reverse=False)`

Evaluate an array and return a 1 based ranking.

#### Parameters

- **arr** (`numpy.ndarray`, `numpy.ndarray`) – A array with values
- **reverse** (`bool` default `False`) – By default (`False`) the lesser values are ranked first (like in time lapse in a race or Golf scoring) if is `True` the data is highest values are the first.

#### Returns

Array of rankings the i-nth element has the ranking of the i-nth element of the row array.

#### Return type

`numpy.ndarray`

### Examples

```

>>> from skcriteria.util.rank import rank_values
>>> # the fastest (the lowest value) goes first
>>> time_laps = [0.59, 1.2, 0.3]
>>> rank_values(time_laps)
array([2, 3, 1])
>>> # highest is better
>>> scores = [140, 200, 98]
>>> rank_values(scores, reverse=True)
array([2, 1, 3])

```

`skcriteria.utils.rank.dominance(array_a, array_b, reverse=False)`

Calculate the dominance or general dominance between two arrays.

#### Parameters

- **array\_a** – The first array to compare.
- **array\_b** – The second array to compare.
- **reverse** (`bool` (`default=False`)) – `array_a[i] > array_b[i]` if `reverse` is `False`, otherwise `array_a[i] < array_b[i]`. Also `reverse` can be an array of boolean of the same shape as `array_a` and `array_b` to revert every item independently. In other words, `reverse` assume the data is a minimization problem.

**Returns**

**dominance** – Named tuple with 4 parameters:

- **eq**: How many values are equals in both arrays.
- **aDb**: **How many values of array\_a dominate those of the same position in array\_b.**
- **bDa**: **How many values of array\_b dominate those of the same position in array\_a.**
- **eq\_where**: **Where the values of array\_a are equals those of the same position in array\_b.**
- **aDb\_where**: **Where the values of array\_a dominates those of the same position in array\_b.**
- **bDa\_where**: **Where the values of array\_b dominates those of the same position in array\_a.**

**Return type**

`_Dominance`

**skcriteria.utils.unames module**

Utility to achieve unique names for a collection of objects.

`skcriteria.utils.unames.unique_names(*, names, elements)`

Generate names unique name.

**Parameters**

- **elements** (*iterable of size n*) – objects to be named
- **names** (*iterable of size n*) – names candidates

**Returns**

Returns a list where each element is a tuple. Each tuple contains two elements: The first is the unique name of the second.

**Return type**

`list` of tuples

## 4.4 Changelog

### 4.4.1 Version 0.8.3

- Fixed a bug detected on the EntropyWeighted, Now works as the literature specifies

### 4.4.2 Version 0.8.2

- We bring back Python 3.7 because is the version used in google.colab.
- Bugfixes in `plot.frontier` and `dominance.eq`.

### 4.4.3 Version 0.8

- **New** The `skcriteria.cmp` package utilities to compare rankings.
- **New** The new package `skcriteria.datasets` include two datasets (one a toy and one real) to quickly start your experiments.
- **New** `DecisionMatrix` now can be sliced with a syntax similar of the `pandas.DataFrame`.
  - `dm["c0"]` cut the  $c0$  criteria.
  - `dm[["c0", "c2"]]` cut the criteria  $c0$  and  $c2$ .
  - `dm.loc["a0"]` cut the alternative  $a0$ .
  - `dm.loc[["a0", "a1"]]` cut the alternatives  $a0$  and  $a1$ .
  - `dm.iloc[0:3]` cuts from the first to the third alternative.
- **New** imputation methods for replacing missing data with substituted values. These methods are in the module `skcriteria.preprocessing.impute`.
- **New** results object now has a `to_series` method.
- **Changed Behaviour:** The ranks and kernels `equals` are now called `values_equals`. The new `aequals` support tolerances to compare numpy arrays internally stored in `extra_`, and the `equals` method is equivalent to `aequals(rtol=0, atol=0)`.
- We detected a bad behavior in `ELECTRE2`, so we decided to launch a `FutureWarning` when the class is instantiated. In the version after 0.8, a new implementation of `ELECTRE2` will be provided.
- Multiple `__repr__` was improved to follow the [Python recommendation](#)
- `Critic` weighter was renamed to `CRITIC` (all capitals) to be consistent with the literature. The old class is still there but is deprecated.
- All the functions and classes of `skcriteria.preprocessing.distance` was moved to `skcriteria.preprocessing.scalars`.
- The `StdWeighter` now uses the **sample** standar-deviation. From the numerical point of view, this does not generate any change, since the deviations are scaled by the sum. Computationally speaking there may be some difference from the ~5th decimal digit onwards.
- Two method of the `Objective` enum was deprecated and replaced:
  - `Objective.construct_from_alias()` -> `Objective.from_alias()` (*classmethod*)
  - `Objective.to_string()` -> `Objective.to_symbol()`
 The deprecated methods will be removed in version *1.0*.
- Add a dominance plot `DecisionMatrix.plot.dominance()`.
- `WeightedSumModel` raises a `ValueError` when some value  $< 0$ .
- Moved internal modules
  - `skcriteria.core.methods.SKCTransformerABC` -> `skcriteria.preprocessing.SKCTransformerABC`

– `skcriteria.core.methods.SKCMatrixAndWeightTransformerABC` → `skcriteria.preprocessing.SKCMatrixAndWeightTransformerABC`

#### 4.4.4 Version 0.7

- **New method:** ELECTRE2.
- **New preprocessing strategy:** A new way to transform from minimization to maximization criteria: `NegateMinimize()` which reverses the sign of the values of the criteria to be minimized (useful for not breaking distance relations in methods like *TOPSIS*). Additionally the previous we rename the `MinimizeToMaximize()` transformer to `InvertMinimize()`.
- Now the `RankingResult`, support repeated/tied rankings and some methods were implemented to deal with these cases.
  - `RankingResult.has_ties_` to see if there are tied values.
  - `RankingResult.ties_` to see how often values are repeated.
  - **`RankingResult.untied_rank_` to get a ranking with no repeated values.** repeated values.
- `KernelResult` now implements several new properties:
  - `kernel_alternatives_` to know which alternatives are in the kernel.
  - `kernel_size_` to know the number of alternatives in the kernel.
  - `kernel_where_` was replaced by `kernelwhere_` to standardize the api.

#### 4.4.5 Version 0.6

- Support for Python 3.10.
- All the objects of the project are now immutable by design, and can only be mutated troughs the `object.copy()` method.
- Dominance analysis tools (`DecisionMatrix.dominance`).
- The method `DecisionMatrix.describe()` was deprecated and will be removed in version *1.0*.
- New statistics functionalities `DecisionMatrix.stats` accessor.
- The accessors are now cached in the `DecisionMatrix`.
- Tutorial for dominance and satisfaction analysis.
- TOPSIS now support hyper-parameters to select different metrics.
- Generalize the idea of accessors in scikit-criteria through a common framework (`skcriteria.utils.accabc` module).
- New deprecation mechanism through the
- `skcriteria.utils.decorators.deprecated` decorator.

### 4.4.6 Version 0.5

In this version scikit-criteria was rewritten from scratch. Among other things:

- The model implementation API was simplified.
- The Data object was removed in favor of DecisionMatrix which implements many more useful features for MCDA.
- Plots were completely re-implemented using Seaborn.
- Coverage was increased to 100%.
- Pipelines concept was added (Thanks to Scikit-learn).
- New documentation. The quick start is totally rewritten!

**Full Changelog:** <https://github.com/quatrope/scikit-criteria/commits/0.5>

### 4.4.7 Version 0.2

First OO stable version.

### 4.4.8 Version 0.1

Only functions.

## 4.5 Bibliography

## 4.6 Indices and tables

- genindex
- modindex
- search





## BIBLIOGRAPHY

- [Brauers & Zavadskas, 2006] Brauers, W. K., & Zavadskas, E. K. (2006). The moora method and its application to privatization in a transition economy. *Control and cybernetics*, 35, 445–469.
- [Brauers & Zavadskas, 2012] Brauers, W. K. M., & Zavadskas, E. K. (2012). Robustness of multimooora: a method for multi-objective optimization. *Informatika*, 23(1), 1–25.
- [Bridgman, 1922] Bridgman, P. W. (1922). *Dimensional analysis*. Yale university press.
- [Diakoulaki et al., 1995] Diakoulaki, D., Mavrotas, G., & Papayannakis, L. (1995). Determining objective weights in multiple criteria problems: the critic method. *Computers & Operations Research*, 22(7), 763–770.
- [Fishburn, 1967] 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.
- [Gomes et al., 2004] Gomes, L., González-Araya, M., & Carignano, C. (2004 , 11). *Tomada de decisões em cenários complexos*. Thomson.
- [Hwang & Yoon, 1981] Hwang, C.-L., & Yoon, K. (1981). Methods for multiple attribute decision making. *Multiple attribute decision making* (pp. 58–191). Springer.
- [Miller & others, 1963] Miller, D. W., & others. (1963). Executive decisions and operations research. *AGRIS*.
- [Rajkumar, 2021] Rajkumar, S. (2021 , Jul). *Cryptocurrency historical prices*.
- [Roy, 1968] 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.
- [Roy, 1990] Roy, B. (1990). The outranking approach and the foundations of electre methods. *Readings in multiple criteria decision aid* (pp. 155–183). Springer.
- [Roy & Bertier, 1971] Roy, B., & Bertier, P. (1971). La méthode electre ii. *Note de travail*, 142.
- [Roy & Bertier, 1973] Roy, B., & Bertier, P. (1973). La méthode electre ii(une application au média-planning...). *VII ème Conférence internationale de recherché opérationnelle*.
- [Simon, 1955] Simon, H. A. (1955). A behavioral model of rational choice. *The quarterly journal of economics*, 69(1), 99–118.
- [Tzeng & Huang, 2011] Tzeng, G.-H., & Huang, J.-J. (2011). *Multiple attribute decision making: methods and applications*. CRC press.
- [VanHeerden et al., 2021a] Van Heerden, N. A., Cabral, J. B., & Luczywo, N. (2021). Evaluación de la importancia de criterios para la selección de criptomonedas. *XXXIV ENDIO - XXXII EPIO Virtual 2021*.
- [VanHeerden et al., 2021b] Van Heerden, N. A., Cabral, J. B., & Luczywo, N. (2021). Evaluation of the importance of criteria for the selection of cryptocurrencies. *arXiv preprint arXiv:2109.00130*.

- [Wikipedia contributors, 2021a] Wikipedia contributors (2021). *TOPSIS* — *Wikipedia, The Free Encyclopedia*. [Online; accessed 23-November-2021].
- [Wikipedia contributors, 2021b] Wikipedia contributors (2021). *Weighted sum model* — *Wikipedia, The Free Encyclopedia*. [Online; accessed 23-November-2021].
- [Wikipedia contributors, 2022a] Wikipedia contributors (2022). *Pareto efficiency* — *Wikipedia, The Free Encyclopedia*. [Online; accessed 9-October-2022].
- [Wikipedia contributors, 2022b] Wikipedia contributors (2022). *Pareto front* — *Wikipedia, The Free Encyclopedia*. [Online; accessed 9-October-2022].

## PYTHON MODULE INDEX

### S

- skcriteria, 29
- skcriteria.cmp, 81
- skcriteria.cmp.ranks\_cmp, 81
- skcriteria.core, 29
- skcriteria.core.data, 29
- skcriteria.core.dominance, 35
- skcriteria.core.methods, 37
- skcriteria.core.objectives, 38
- skcriteria.core.plot, 38
- skcriteria.core.stats, 43
- skcriteria.datasets, 87
- skcriteria.madm, 44
- skcriteria.madm.\_madm\_base, 44
- skcriteria.madm.electre, 47
- skcriteria.madm.moora, 50
- skcriteria.madm.similarity, 52
- skcriteria.madm.simple, 52
- skcriteria.madm.simus, 54
- skcriteria.pipeline, 88
- skcriteria.preprocessing, 55
- skcriteria.preprocessing.\_preprocessing\_base, 55
- skcriteria.preprocessing.distance, 56
- skcriteria.preprocessing.filters, 57
- skcriteria.preprocessing.impute, 67
- skcriteria.preprocessing.increment, 70
- skcriteria.preprocessing.invert\_objectives, 71
- skcriteria.preprocessing.push\_negatives, 72
- skcriteria.preprocessing.scalers, 73
- skcriteria.preprocessing.weighters, 77
- skcriteria.utils, 90
- skcriteria.utils.accabc, 90
- skcriteria.utils.bunch, 90
- skcriteria.utils.cmanagers, 91
- skcriteria.utils.deprecate, 91
- skcriteria.utils.doctools, 92
- skcriteria.utils.lp, 92
- skcriteria.utils.rank, 95
- skcriteria.utils.unames, 96



## A

AccessorABC (class in *skcriteria.utils.accabc*), 90

add\_value\_to\_zero() (in module *skcriteria.preprocessing.increment*), 70

AddValueToZero (class in *skcriteria.preprocessing.increment*), 71

aequals() (*skcriteria.core.data.DecisionMatrix* method), 33

aequals() (*skcriteria.madm.\_madm\_base.ResultABC* method), 45

alternatives (*skcriteria.core.data.DecisionMatrix* property), 31

alternatives (*skcriteria.madm.\_madm\_base.ResultABC* property), 45

area() (*skcriteria.core.plot.DecisionMatrixPlotter* method), 42

## B

bar() (*skcriteria.cmp.ranks\_cmp.RanksComparatorPlotter* method), 86

bar() (*skcriteria.core.plot.DecisionMatrixPlotter* method), 39

barh() (*skcriteria.cmp.ranks\_cmp.RanksComparatorPlotter* method), 87

barh() (*skcriteria.core.plot.DecisionMatrixPlotter* method), 39

base\_value (*skcriteria.preprocessing.weighters.EqualWeighter* property), 78

Bool (class in *skcriteria.utils.lp*), 93

box() (*skcriteria.cmp.ranks\_cmp.RanksComparatorPlotter* method), 86

box() (*skcriteria.core.plot.DecisionMatrixPlotter* method), 40

bt() (*skcriteria.core.dominance.DecisionMatrixDominanceAccessor* method), 35

Bunch (class in *skcriteria.utils.bunch*), 90

## C

cenit\_distance() (in module *skcriteria.preprocessing.distance*), 56

CenitDistance (class in *skcriteria.preprocessing.distance*), 56

CenitDistanceMatrixScaler (class in *skcriteria.preprocessing.scalers*), 77

clip (*skcriteria.preprocessing.scalers.MinMaxScaler* property), 74

compare() (*skcriteria.core.dominance.DecisionMatrixDominanceAccessor* method), 36

concordance() (in module *skcriteria.madm.electre*), 47

construct\_from\_alias() (*skcriteria.core.objectives.Objective* class method), 38

copy() (*skcriteria.core.data.DecisionMatrix* method), 32

copy() (*skcriteria.core.methods.SKCMMethodABC* method), 37

corr() (*skcriteria.cmp.ranks\_cmp.RanksComparator* method), 82

corr() (*skcriteria.cmp.ranks\_cmp.RanksComparatorPlotter* method), 85

CORRELATION (*skcriteria.preprocessing.weighters.CRITIC* attribute), 80

correlation (*skcriteria.preprocessing.weighters.CRITIC* property), 80

cov() (*skcriteria.cmp.ranks\_cmp.RanksComparator* method), 82

cov() (*skcriteria.cmp.ranks\_cmp.RanksComparatorPlotter* method), 85

criteria (*skcriteria.core.data.DecisionMatrix* property), 31

criteria\_filters (*skcriteria.preprocessing.filters.SKCBByCriteriaFilterABC* property), 57

criteria\_range (*skcriteria.preprocessing.scalers.MinMaxScaler* property), 74

CRITIC (class in *skcriteria.preprocessing.weighters*), 80

Critic (class in *skcriteria.preprocessing.weighters*), 80

critic\_weights() (in module *skcriteria.preprocessing.weighters*), 80

## D

DecisionMatrix (class in *skcriteria.core.data*), 29

- DecisionMatrixDominanceAccessor (class in *skcriteria.core.dominance*), 35
- DecisionMatrixPlotter (class in *skcriteria.core.plot*), 38
- DecisionMatrixStatsAccessor (class in *skcriteria.core.stats*), 43
- deprecated() (in module *skcriteria.utils.deprecate*), 91
- describe() (*skcriteria.core.data.DecisionMatrix* method), 33
- df\_temporal\_header() (in module *skcriteria.utils.cmanagers*), 91
- discordance() (in module *skcriteria.madm.electre*), 47
- distance() (*skcriteria.cmp.ranks\_cmp.RanksComparator* method), 83
- distance() (*skcriteria.cmp.ranks\_cmp.RanksComparatorPlotter* method), 86
- doc\_inherit() (in module *skcriteria.utils.doctools*), 92
- dominance (*skcriteria.core.data.DecisionMatrix* property), 32
- dominance() (in module *skcriteria.utils.rank*), 95
- dominance() (*skcriteria.core.dominance.DecisionMatrixDominanceAccessor* method), 35
- dominance() (*skcriteria.core.plot.DecisionMatrixPlotter* method), 42
- dominated() (*skcriteria.core.dominance.DecisionMatrixDominanceAccessor* method), 36
- dominators\_of() (*skcriteria.core.dominance.DecisionMatrixDominanceAccessor* method), 36
- dtypes (*skcriteria.core.data.DecisionMatrix* property), 31
- ## E
- e\_ (*skcriteria.madm.\_madm\_base.ResultABC* property), 45
- ELECTRE1 (class in *skcriteria.madm.electre*), 47
- electre1() (in module *skcriteria.madm.electre*), 47
- ELECTRE2 (class in *skcriteria.madm.electre*), 48
- electre2() (in module *skcriteria.madm.electre*), 48
- entropy\_weights() (in module *skcriteria.preprocessing.weighters*), 78
- EntropyWeighter (class in *skcriteria.preprocessing.weighters*), 79
- eq() (*skcriteria.core.dominance.DecisionMatrixDominanceAccessor* method), 35
- equal\_weights() (in module *skcriteria.preprocessing.weighters*), 77
- equals() (*skcriteria.core.data.DecisionMatrix* method), 33
- equals() (*skcriteria.madm.\_madm\_base.ResultABC* method), 46
- EqualWeighter (class in *skcriteria.preprocessing.weighters*), 78
- estimator (*skcriteria.preprocessing.impute.IterativeImputer* property), 69
- evaluate() (*skcriteria.madm.\_madm\_base.SKCDecisionMakerABC* method), 44
- evaluate() (*skcriteria.madm.simus.SIMUS* method), 54
- evaluate() (*skcriteria.pipeline.SKCPipeline* method), 89
- extra\_ (*skcriteria.madm.\_madm\_base.ResultABC* property), 45
- ## F
- fill\_value (*skcriteria.preprocessing.impute.SimpleImputer* property), 67
- Filter (class in *skcriteria.preprocessing.filters*), 57
- FilterEQ (class in *skcriteria.preprocessing.filters*), 62
- FilterGE (class in *skcriteria.preprocessing.filters*), 59
- FilterGT (class in *skcriteria.preprocessing.filters*), 59
- FilterIn (class in *skcriteria.preprocessing.filters*), 64
- FilterLE (class in *skcriteria.preprocessing.filters*), 61
- FilterLT (class in *skcriteria.preprocessing.filters*), 60
- FilterNE (class in *skcriteria.preprocessing.filters*), 63
- FilterNonDominated (class in *skcriteria.preprocessing.filters*), 65
- FilterNotIn (class in *skcriteria.preprocessing.filters*), 65
- Float (class in *skcriteria.utils.lp*), 92
- flow() (*skcriteria.cmp.ranks\_cmp.RanksComparatorPlotter* method), 84
- fmf() (in module *skcriteria.madm.moora*), 51
- from\_alias() (*skcriteria.core.objectives.Objective* class method), 38
- from\_mcda\_data() (*skcriteria.core.data.DecisionMatrix* class method), 30
- frontier() (*skcriteria.core.plot.DecisionMatrixPlotter* method), 42
- FullMultiplicativeForm (class in *skcriteria.madm.moora*), 51
- ## G
- get\_parameters() (*skcriteria.core.methods.SKCMMethodABC* method), 37
- ## H
- has\_loops() (*skcriteria.core.dominance.DecisionMatrixDominanceAccessor* method), 36
- has\_ties\_ (*skcriteria.madm.\_madm\_base.RankResult* property), 46
- heatmap() (*skcriteria.cmp.ranks\_cmp.RanksComparatorPlotter* method), 85
- heatmap() (*skcriteria.core.plot.DecisionMatrixPlotter* method), 39

- `hist()` (*skcriteria.core.plot.DecisionMatrixPlotter* method), 40
- ## I
- `ignore_missing_criteria` (*skcriteria.preprocessing.filters.SKByCriteriaFilterABC* property), 57
- `iloc` (*skcriteria.core.data.DecisionMatrix* property), 34
- `imputation_order` (*skcriteria.preprocessing.impute.IterativeImputer* property), 69
- `initial_strategy` (*skcriteria.preprocessing.impute.IterativeImputer* property), 69
- `Int` (class in *skcriteria.utils.lp*), 92
- `InvertMinimize` (class in *skcriteria.preprocessing.invert\_objectives*), 71
- `iobjectives` (*skcriteria.core.data.DecisionMatrix* property), 31
- `is_solver_available()` (in module *skcriteria.utils.lp*), 92
- `IterativeImputer` (class in *skcriteria.preprocessing.impute*), 67
- ## K
- `kde()` (*skcriteria.core.plot.DecisionMatrixPlotter* method), 41
- `kernel_` (*skcriteria.madm.\_madm\_base.KernelResult* property), 47
- `kernel_alternatives_` (*skcriteria.madm.\_madm\_base.KernelResult* property), 47
- `kernel_size_` (*skcriteria.madm.\_madm\_base.KernelResult* property), 47
- `kernel_where_` (*skcriteria.madm.\_madm\_base.KernelResult* property), 47
- `KernelResult` (class in *skcriteria.madm.\_madm\_base*), 47
- `kernelwhere_` (*skcriteria.madm.\_madm\_base.KernelResult* property), 47
- `KNNImputer` (class in *skcriteria.preprocessing.impute*), 69
- ## L
- `load_simple_stock_selection()` (in module *skcriteria.datasets*), 87
- `load_van2021evaluation()` (in module *skcriteria.datasets*), 87
- `loc` (*skcriteria.core.data.DecisionMatrix* property), 34
- ## M
- `mad()` (*skcriteria.core.stats.DecisionMatrixStatsAccessor* method), 44
- `matrix` (*skcriteria.core.data.DecisionMatrix* property), 31
- `matrix_scale_by_cenit_distance()` (in module *skcriteria.preprocessing.scalers*), 76
- `MAX` (*skcriteria.core.objectives.Objective* attribute), 38
- `max_iter` (*skcriteria.preprocessing.impute.IterativeImputer* property), 69
- `max_value` (*skcriteria.preprocessing.impute.IterativeImputer* property), 69
- `MaxAbsScaler` (class in *skcriteria.preprocessing.scalers*), 74
- `Maximize` (class in *skcriteria.utils.lp*), 94
- `MaxScaler` (class in *skcriteria.preprocessing.scalers*), 74
- `maxwhere` (*skcriteria.core.data.DecisionMatrix* property), 31
- `method` (*skcriteria.madm.\_madm\_base.ResultABC* property), 45
- `metric` (*skcriteria.madm.similarity.TOPSIS* property), 52
- `metric` (*skcriteria.preprocessing.impute.KNNImputer* property), 70
- `MIN` (*skcriteria.core.objectives.Objective* attribute), 38
- `min_value` (*skcriteria.preprocessing.impute.IterativeImputer* property), 69
- `Minimize` (class in *skcriteria.utils.lp*), 93
- `MinimizeToMaximize` (class in *skcriteria.preprocessing.invert\_objectives*), 72
- `MinMaxScaler` (class in *skcriteria.preprocessing.scalers*), 74
- `minwhere` (*skcriteria.core.data.DecisionMatrix* property), 31
- `missing_values` (*skcriteria.preprocessing.impute.IterativeImputer* property), 69
- `missing_values` (*skcriteria.preprocessing.impute.KNNImputer* property), 70
- `missing_values` (*skcriteria.preprocessing.impute.SimpleImputer* property), 67
- `mkdm()` (in module *skcriteria.core.data*), 34
- `mkpipe()` (in module *skcriteria.pipeline*), 89
- `mkrank_cmp()` (in module *skcriteria.cmp.ranks\_cmp*), 87
- module
- skcriteria*, 29
  - skcriteria.cmp*, 81
  - skcriteria.cmp.ranks\_cmp*, 81
  - skcriteria.core*, 29
  - skcriteria.core.data*, 29
  - skcriteria.core.dominance*, 35
  - skcriteria.core.methods*, 37

- skcriteria.core.objectives, 38
  - skcriteria.core.plot, 38
  - skcriteria.core.stats, 43
  - skcriteria.datasets, 87
  - skcriteria.madm, 44
  - skcriteria.madm.\_madm\_base, 44
  - skcriteria.madm.electre, 47
  - skcriteria.madm.mooraa, 50
  - skcriteria.madm.similarity, 52
  - skcriteria.madm.simple, 52
  - skcriteria.madm.simus, 54
  - skcriteria.pipeline, 88
  - skcriteria.preprocessing, 55
  - skcriteria.preprocessing.\_preprocessing\_base, 55
  - skcriteria.preprocessing.distance, 56
  - skcriteria.preprocessing.filters, 57
  - skcriteria.preprocessing.impute, 67
  - skcriteria.preprocessing.increment, 70
  - skcriteria.preprocessing.invert\_objectives, 71
  - skcriteria.preprocessing.push\_negatives, 72
  - skcriteria.preprocessing.scalers, 73
  - skcriteria.preprocessing.weighters, 77
  - skcriteria.utils, 90
  - skcriteria.utils.accabc, 90
  - skcriteria.utils.bunch, 90
  - skcriteria.utils.cmanagers, 91
  - skcriteria.utils.deprecate, 91
  - skcriteria.utils.doctools, 92
  - skcriteria.utils.lp, 92
  - skcriteria.utils.rank, 95
  - skcriteria.utils.unames, 96
  - MultiMOORA (class in skcriteria.madm.mooraa), 51
  - multimoora() (in module skcriteria.madm.mooraa), 51
- ## N
- n\_nearest\_criteria (skcriteria.preprocessing.impute.IterativeImputer property), 69
  - n\_neighbors (skcriteria.preprocessing.impute.KNNImputer property), 70
  - named\_ranks (skcriteria.cmp.ranks\_cmp.RanksComparator property), 82
  - named\_steps (skcriteria.pipeline.SKCPipeline property), 89
  - NegateMinimize (class in skcriteria.preprocessing.invert\_objectives), 71
- ## O
- Objective (class in skcriteria.core.objectives), 38
  - objectives (skcriteria.core.data.DecisionMatrix property), 31
  - ogive() (skcriteria.core.plot.DecisionMatrixPlotter method), 41
- ## P
- p (skcriteria.madm.electre.ELECTRE1 property), 48
  - p0 (skcriteria.madm.electre.ELECTRE2 property), 49
  - p1 (skcriteria.madm.electre.ELECTRE2 property), 49
  - p2 (skcriteria.madm.electre.ELECTRE2 property), 49
  - pearson\_correlation() (in module skcriteria.preprocessing.weighters), 79
  - plot (skcriteria.cmp.ranks\_cmp.RanksComparator property), 83
  - plot (skcriteria.core.data.DecisionMatrix property), 32
  - push\_negatives() (in module skcriteria.preprocessing.push\_negatives), 72
  - PushNegatives (class in skcriteria.preprocessing.push\_negatives), 73
- ## Q
- q (skcriteria.madm.electre.ELECTRE1 property), 48
  - q0 (skcriteria.madm.electre.ELECTRE2 property), 49
  - q1 (skcriteria.madm.electre.ELECTRE2 property), 49
- ## R
- r2\_score() (skcriteria.cmp.ranks\_cmp.RanksComparator method), 83
  - r2\_score() (skcriteria.cmp.ranks\_cmp.RanksComparatorPlotter method), 85
  - random\_state (skcriteria.preprocessing.impute.IterativeImputer property), 69
  - rank\_ (skcriteria.madm.\_madm\_base.RankResult property), 46
  - rank\_by (skcriteria.madm.simus.SIMUS property), 54
  - rank\_values() (in module skcriteria.utils.rank), 95
  - RankResult (class in skcriteria.madm.\_madm\_base), 46
  - ranks (skcriteria.cmp.ranks\_cmp.RanksComparator property), 81
  - RanksComparator (class in skcriteria.cmp.ranks\_cmp), 81
  - RanksComparatorPlotter (class in skcriteria.cmp.ranks\_cmp), 83
  - ratio() (in module skcriteria.madm.mooraa), 50
  - RatioMOORA (class in skcriteria.madm.mooraa), 50
  - ReferencePointMOORA (class in skcriteria.madm.mooraa), 50
  - refpoint() (in module skcriteria.madm.mooraa), 50
  - reg() (skcriteria.cmp.ranks\_cmp.RanksComparatorPlotter method), 84
  - ResultABC (class in skcriteria.madm.\_madm\_base), 44
- ## S
- sample\_posterior (skcriteria.preprocessing.impute.IterativeImputer



- property*), 69
- scale* (*skcriteria.preprocessing.weighters.CRITIC property*), 80
- scale\_by\_sum()* (in module *skcriteria.preprocessing.scalers*), 76
- scale\_by\_vector()* (in module *skcriteria.preprocessing.scalers*), 75
- sense* (*skcriteria.utils.lp.Maximize attribute*), 95
- sense* (*skcriteria.utils.lp.Minimize attribute*), 94
- shape* (*skcriteria.core.data.DecisionMatrix property*), 33
- shape* (*skcriteria.madm.\_madm\_base.ResultABC property*), 45
- SimpleImputer* (class in *skcriteria.preprocessing.impute*), 67
- SIMUS* (class in *skcriteria.madm.simus*), 54
- simus()* (in module *skcriteria.madm.simus*), 54
- SKArithmeticFilterABC* (class in *skcriteria.preprocessing.filters*), 58
- SKByCriteriaFilterABC* (class in *skcriteria.preprocessing.filters*), 57
- SKDecisionMakerABC* (class in *skcriteria.madm.\_madm\_base*), 44
- SKImputerABC* (class in *skcriteria.preprocessing.impute*), 67
- SKMatrixAndWeightTransformerABC* (class in *skcriteria.preprocessing.\_preprocessing\_base*), 55
- SKMethodABC* (class in *skcriteria.core.methods*), 37
- SKObjectivesInverterABC* (class in *skcriteria.preprocessing.invert\_objectives*), 71
- SKPipeline* (class in *skcriteria.pipeline*), 88
- skcriteria* module, 29
- skcriteria.cmp* module, 81
- skcriteria.cmp.ranks\_cmp* module, 81
- skcriteria.core* module, 29
- skcriteria.core.data* module, 29
- skcriteria.core.dominance* module, 35
- skcriteria.core.methods* module, 37
- skcriteria.core.objectives* module, 38
- skcriteria.core.plot* module, 38
- skcriteria.core.stats* module, 43
- skcriteria.datasets* module, 87
- skcriteria.madm* module, 44
- skcriteria.madm.\_madm\_base* module, 44
- skcriteria.madm.electre* module, 47
- skcriteria.madm.moora* module, 50
- skcriteria.madm.similarity* module, 52
- skcriteria.madm.simple* module, 52
- skcriteria.madm.simus* module, 54
- skcriteria.pipeline* module, 88
- skcriteria.preprocessing* module, 55
- skcriteria.preprocessing.\_preprocessing\_base* module, 55
- skcriteria.preprocessing.distance* module, 56
- skcriteria.preprocessing.filters* module, 57
- skcriteria.preprocessing.impute* module, 67
- skcriteria.preprocessing.increment* module, 70
- skcriteria.preprocessing.invert\_objectives* module, 71
- skcriteria.preprocessing.push\_negatives* module, 72
- skcriteria.preprocessing.scalers* module, 73
- skcriteria.preprocessing.weighters* module, 77
- skcriteria.utils* module, 90
- skcriteria.utils.accabc* module, 90
- skcriteria.utils.bunch* module, 90
- skcriteria.utils.cmanagers* module, 91
- skcriteria.utils.deprecate* module, 91
- skcriteria.utils.doctools* module, 92
- skcriteria.utils.lp* module, 92
- skcriteria.utils.rank* module, 95
- skcriteria.utils.unames* module, 96
- SKCriteriaDeprecationWarning*, 91
- SKCriteriaFutureWarning*, 91

SKCSetFilterABC (class in *skcriteria.preprocessing.filters*), 64  
 SKCTransformerABC (class in *skcriteria.preprocessing.\_preprocessing\_base*), 55  
 SKCWeighterABC (class in *skcriteria.preprocessing.weighters*), 77  
 solver (*skcriteria.madm.simus.SIMUS* property), 54  
 spearman\_correlation() (in module *skcriteria.preprocessing.weighters*), 79  
 StandarScaler (class in *skcriteria.preprocessing.scalers*), 73  
 stats (*skcriteria.core.data.DecisionMatrix* property), 32  
 std\_weights() (in module *skcriteria.preprocessing.weighters*), 78  
 StdWeighter (class in *skcriteria.preprocessing.weighters*), 78  
 steps (*skcriteria.pipeline.SKCPipeline* property), 89  
 strategy (*skcriteria.preprocessing.impute.SimpleImputer* property), 67  
 strict (*skcriteria.preprocessing.filters.FilterNonDominated* property), 66  
 SumScaler (class in *skcriteria.preprocessing.scalers*), 76

## T

target (*skcriteria.preprocessing.\_preprocessing\_base.SKCMatrixAndWeightTransformerABC* property), 56  
 ties\_ (*skcriteria.madm.\_madm\_base.RankResult* property), 46  
 to\_dataframe() (*skcriteria.cmp.ranks\_cmp.RanksComparator* method), 82  
 to\_dataframe() (*skcriteria.core.data.DecisionMatrix* method), 32  
 to\_dict() (*skcriteria.core.data.DecisionMatrix* method), 32  
 to\_series() (*skcriteria.madm.\_madm\_base.RankResult* method), 46  
 to\_series() (*skcriteria.madm.\_madm\_base.ResultABC* method), 45  
 to\_string() (*skcriteria.core.objectives.Objective* method), 38  
 to\_symbol() (*skcriteria.core.objectives.Objective* method), 38  
 tol (*skcriteria.preprocessing.impute.IterativeImputer* property), 69  
 TOPSIS (class in *skcriteria.madm.similarity*), 52  
 topsis() (in module *skcriteria.madm.similarity*), 52  
 transform() (*skcriteria.pipeline.SKCPipeline* method), 89  
 transform() (*skcriteria.preprocessing.\_preprocessing\_base.SKCTransformerABC* method), 55  
 transform() (*skcriteria.preprocessing.filters.FilterNonDominated* method), 66

## U

unique\_names() (in module *skcriteria.utils.unames*), 96  
 untied\_rank\_ (*skcriteria.madm.\_madm\_base.RankResult* property), 46

## V

value (*skcriteria.preprocessing.increment.AddValueToZero* property), 71  
 values (*skcriteria.madm.\_madm\_base.ResultABC* property), 44  
 values\_equals() (*skcriteria.madm.\_madm\_base.ResultABC* method), 45  
 var\_type (*skcriteria.utils.lp.Bool* attribute), 93  
 var\_type (*skcriteria.utils.lp.Float* attribute), 92  
 var\_type (*skcriteria.utils.lp.Int* attribute), 93  
 VectorScaler (class in *skcriteria.preprocessing.scalers*), 75  
 verbose (*skcriteria.preprocessing.impute.IterativeImputer* property), 69

## W

wbar() (*skcriteria.core.plot.DecisionMatrixPlotter* method), 39  
 warn() (*skcriteria.core.plot.DecisionMatrixPlotter* method), 40  
 wbox() (*skcriteria.core.plot.DecisionMatrixPlotter* method), 41  
 WeightedProductModel (class in *skcriteria.madm.simple*), 53  
 WeightedSumModel (class in *skcriteria.madm.simple*), 52  
 weights (*skcriteria.core.data.DecisionMatrix* property), 31  
 weights (*skcriteria.preprocessing.impute.KNNImputer* property), 70  
 weights\_outrank() (in module *skcriteria.madm.electre*), 48  
 wheatmap() (*skcriteria.core.plot.DecisionMatrixPlotter* method), 39  
 whist() (*skcriteria.core.plot.DecisionMatrixPlotter* method), 40  
 will\_change() (in module *skcriteria.utils.deprecate*), 91  
 with\_mean (*skcriteria.preprocessing.scalers.StandarScaler* property), 74  
 with\_std (*skcriteria.preprocessing.scalers.StandarScaler* property), 74  
 wkde() (*skcriteria.core.plot.DecisionMatrixPlotter* method), 41  
 wogive() (*skcriteria.core.plot.DecisionMatrixPlotter* method), 41  
 wpm() (in module *skcriteria.madm.simple*), 53

`wsm()` (*in module `skcriteria.madm.simple`*), 52