Scikit-Criteria Documentation

Release 0.9

Juan BC

Aug 03, 2025

TUTORIALS

Ver. 0.9

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 (if you have a support question, try the mailing list first)

CHAPTER

ONE

SUPPORT

This project is completely free of charge and open source. If you find it useful in your work or simply want to support us, you can buy us a coffee:

CHAPTER

TWO

CODE REPOSITORY & ISSUES

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

CHAPTER

THREE

LICENSE

Scikit-Criteria is under The 3-Clause BSD License

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

CHAPTER

FOUR

CITATION

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

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

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

Bibtex entry:

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

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

CHAPTER

FIVE

CONTENTS

5.1 Installation

5.1.1 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.

5.1.2 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/quatrope/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.

5.1.3 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.

5.2 Tutorials

This section contains a step-by-step by example tutorial of how to use Scikit-Criteria

Contents:

5.2.1 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`/w_j 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"],
)
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.

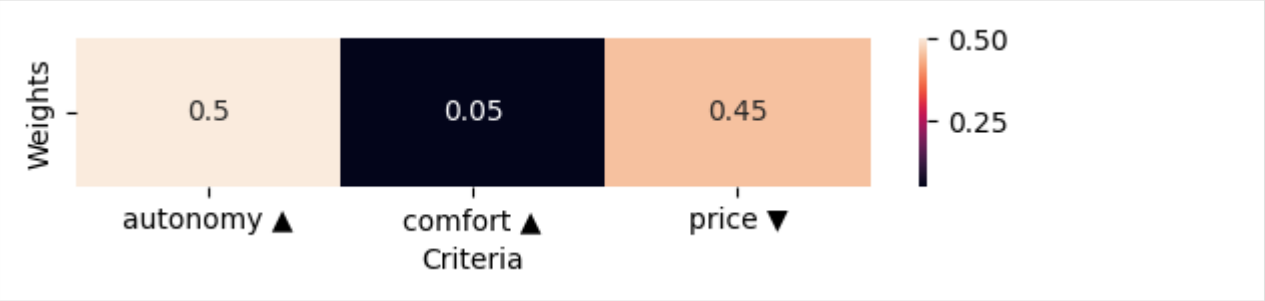
[12]:
```
dm.plot()
```

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

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

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

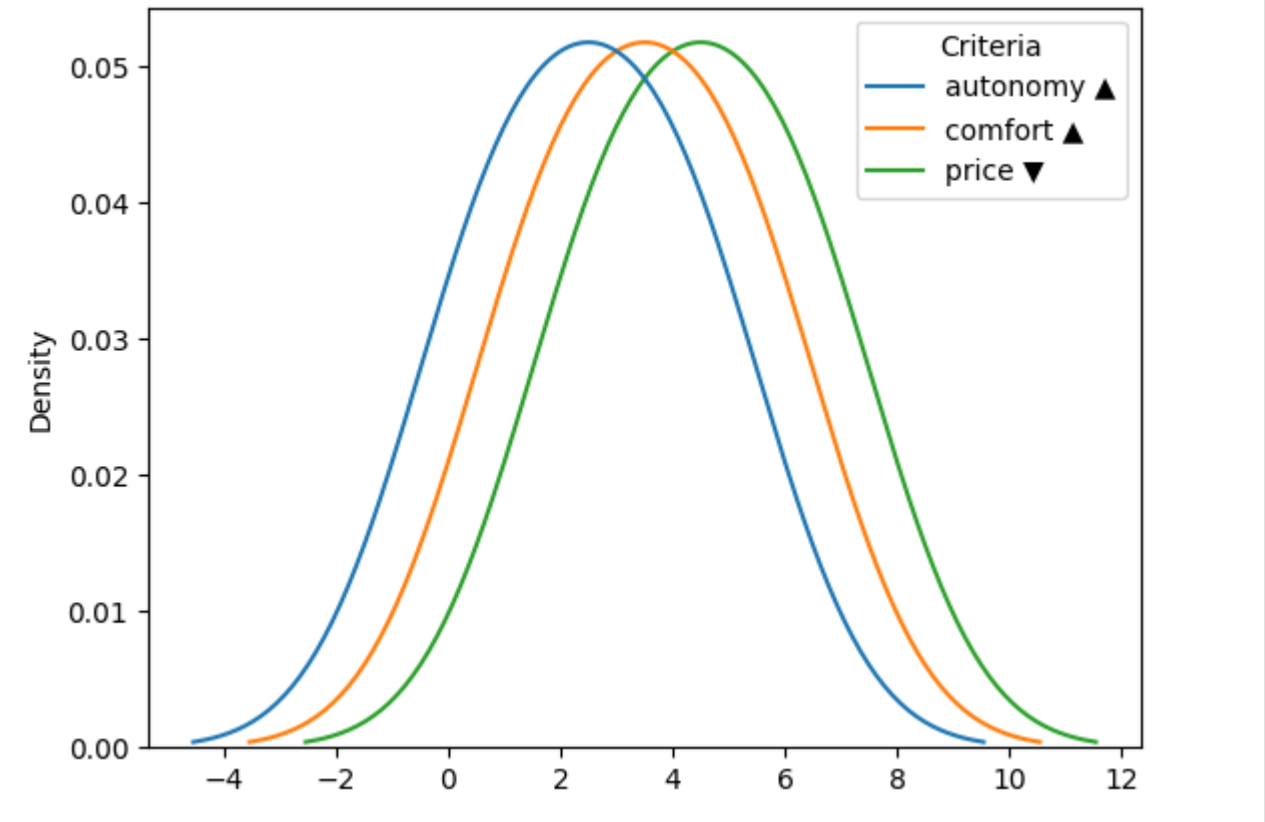
[13]:
```
<AxesSubplot:xlabel='Criteria'>
```



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")
```

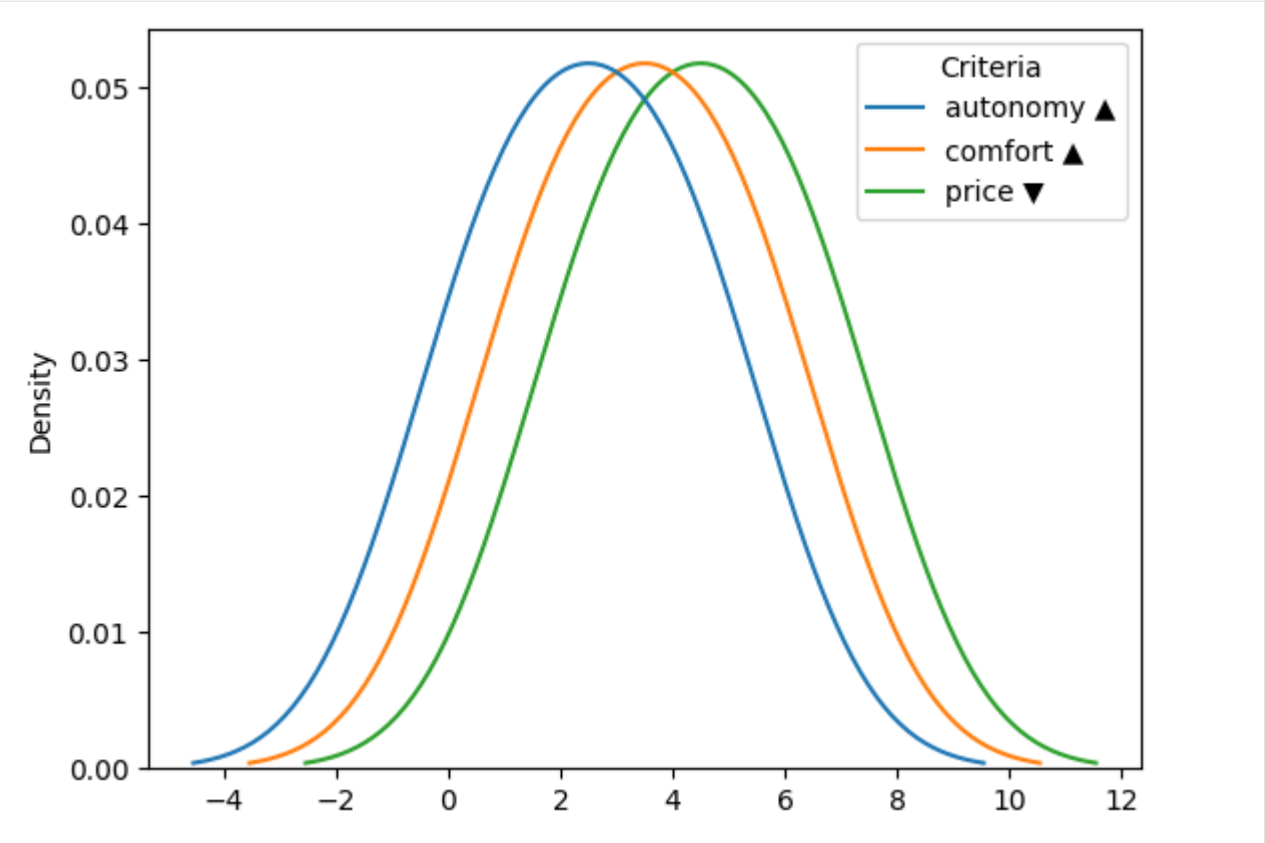
[14]:
```
<AxesSubplot:ylabel='Density'>
```



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

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

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

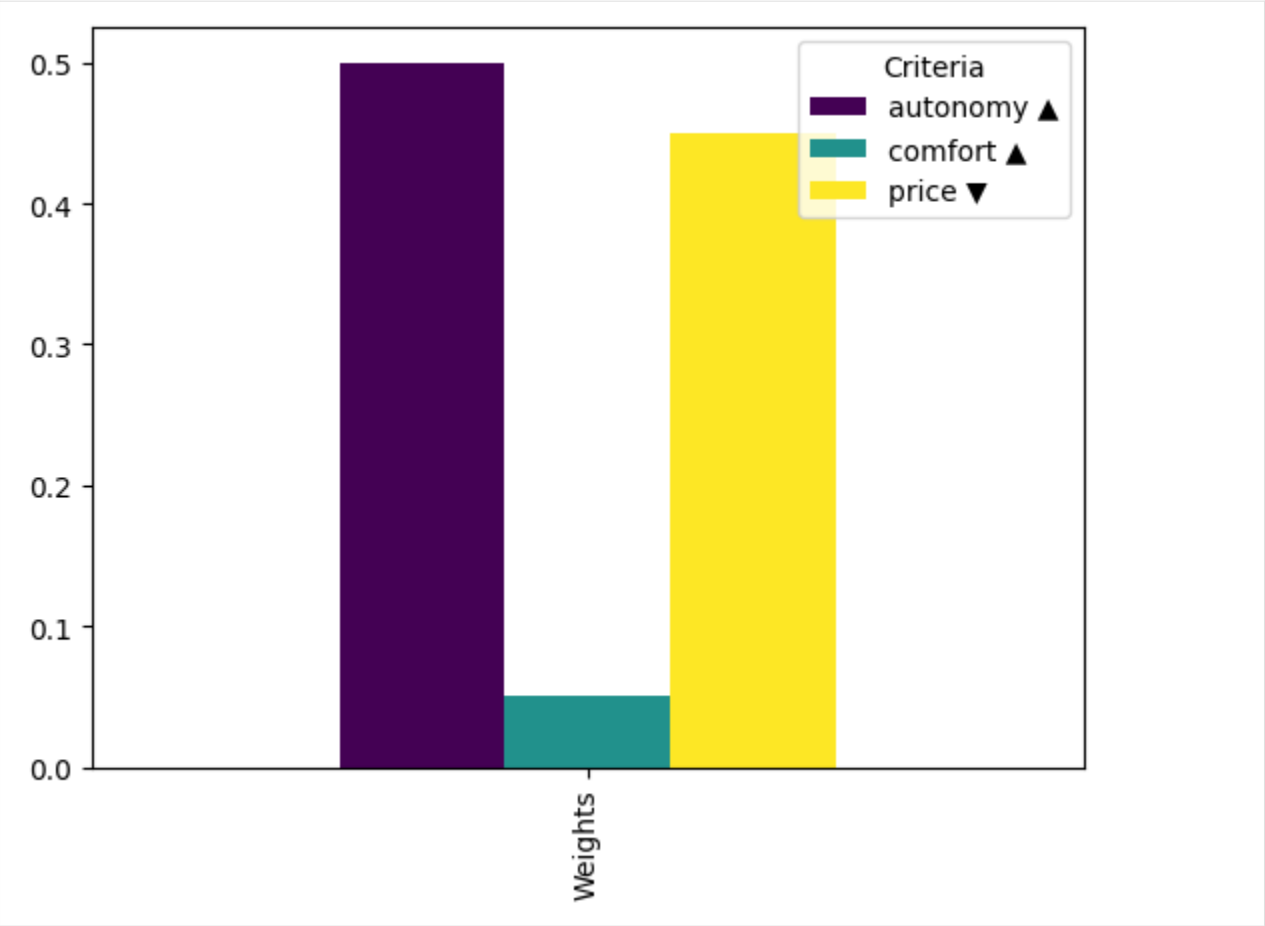


Every plot has their own set of parameters, defined by the subjacent function

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
```

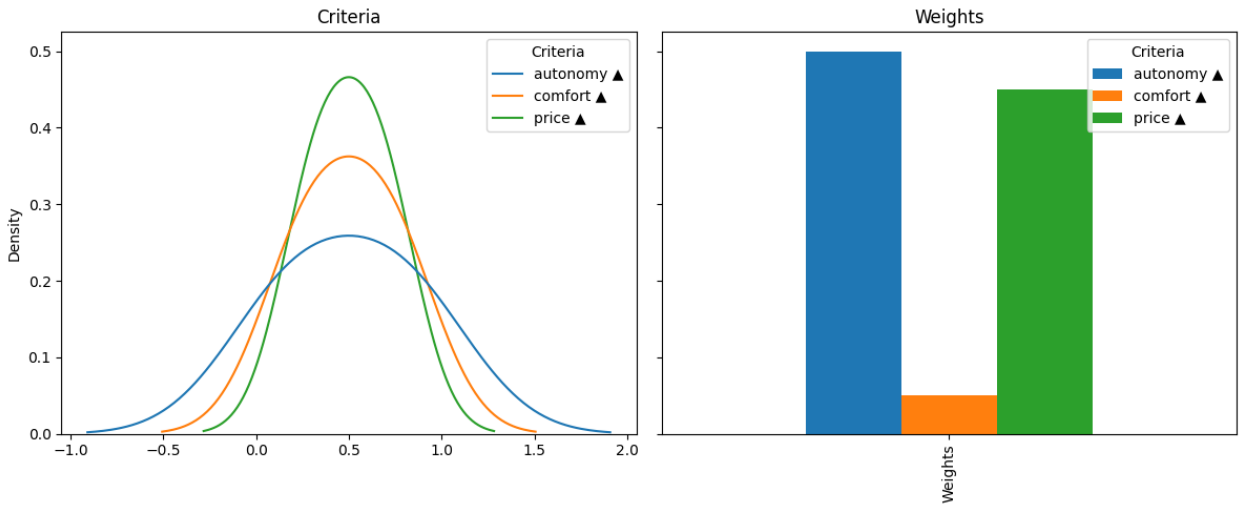
[19]:
```
      autonomy[ 0.50]  comfort[ 0.05]  price[ 0.45]
VW                0.2         0.285714       0.666667
Ford              0.8         0.714286       0.333333
[2 Alternatives x 3 Criteria]
```

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
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.agg 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-1* method later on.

[22]:
```
dec = simple.WeightedSumModel()
rank = dec.evaluate(dmt)  # we use the tansformed 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 acces 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 the 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`.

[ ]:
```
from skcriteria.agg import topsis # 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.

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

pipe
```

```
<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 normalizated 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.agg 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

---

Generated by nbsphinx from a Jupyter notebook. 2025-08-03T00:34:25.057878

5.2.2 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 $ROE >= 2$

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

[ ]:
```
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 >= 2.

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

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

```
<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

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

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

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

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

```
    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 $ROE >= 2$.

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*.

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

```
    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 ($>=$).

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

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

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

- `filters.FilterLE`: Filter Less or Equal than ($<=$).

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

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

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

- `filters.FilterNE`: Filter Not-Equal than ($! =$).

```
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 dominate A_1 ($A_0 \succeq$ ‘A_1). :math:‘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

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

```
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:

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

```
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()`.

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

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

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

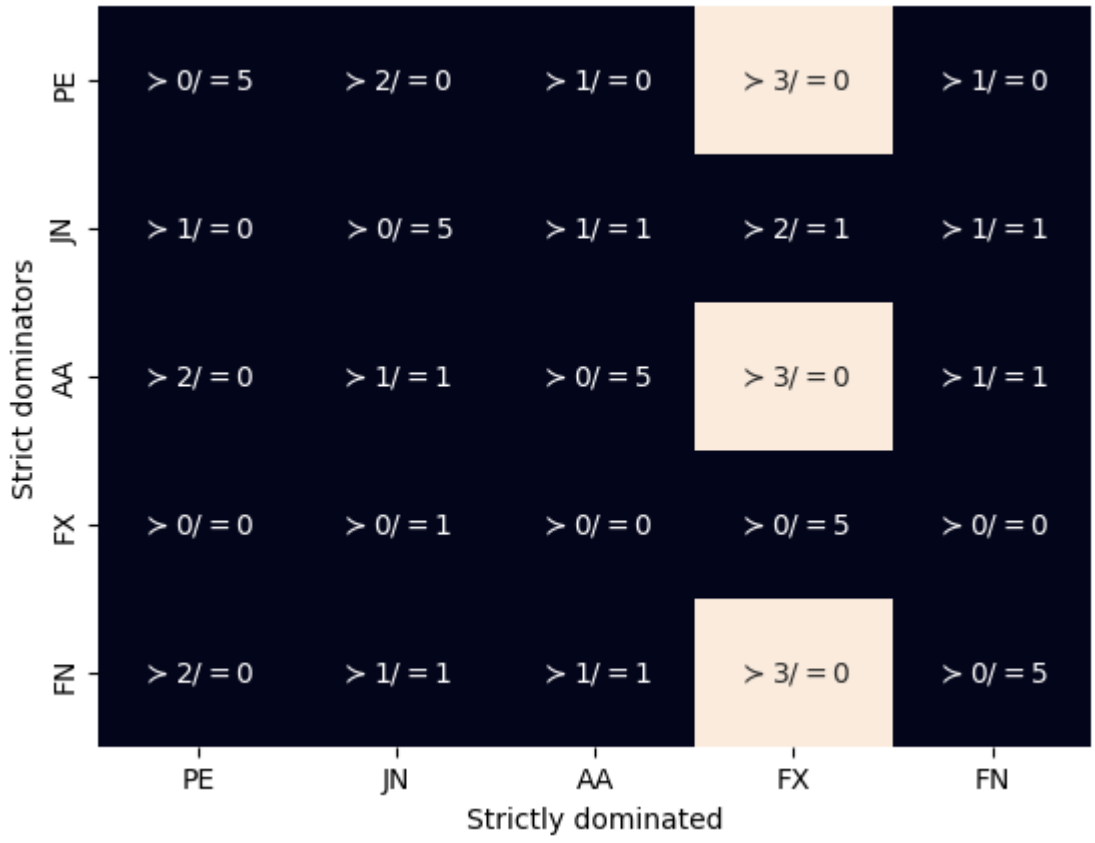
[ ]:
```
dmf.dominance(strict=True)  # equivalent to dmf.dominance.dominance()
```

```
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

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



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

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

```
                Criteria              Performance
                     ROE    CAP     RI
Alternatives 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.

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

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

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

```
    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 expermient

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 >= $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.

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

```
<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

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

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

Generated by nbsphinx from a Jupyter notebook. 2025-08-03T00:34:25.057878

5.2.3 Rankings comparison

This tutorial provides an overview of the use of the Scikit-Criteria ranking comparison tools.

Motivation

It is interesting to note that there are many different aggregation functions (TOPSIS, WeightedSum, MOORA, etc.), which summarize multiple criteria with quite different heuristics to a single analysis dimension; if we add to this the different preprocessing (scaling, weight calculation, optimality sense transformation, etc), the approaches to compute rankings are numerous.

The question then arises:

> What is the best approach MCDM is the best?

Or:

> What defines that an approach is the best?

We can think of some desirable characteristics for all decision algorithms:

- Be easy to understand.
- That the representation of the problem is consistent.
- That at the minimum change of weights everything does not change abruptly.
- That any new alternative that is incorporated does not distort the ranking too much.

This ends up defining a Paradox in which

> The choice of the best multi-criteria method is a multi-criteria problem.

To solve this problem we can use three different options

- Compare rankings manually.
- Exploiting the concept of inversion of the rankings.
- Sensitivity analysis.

In this tutorial we will focus on the first option.

Tools to compare rankings manually

As of *Scikit-Criteria* 0.8 there is a class and a function named `cmp.RanksComparator` and `mkrank_cmp`, which consume multiple rankings and provide tools for analysis and visualization for correlation, regression and direct comparison of results.

To use them we must import them from the `cmp` module.

[1]:
```
from skcriteria.cmp import RanksComparator, mkrank_cmp
```

Experiment setup

First we need a dataset, lets use the decision-matrix extracted from from historical time series cryptocurrencies with `windows_size=7`.

[2]:
```
import skcriteria as skc

dm = skc.datasets.load_van2021evaluation(windows_size=7)
dm
```

[2]:
```
      xRV[ 1.0]  sRV[ 1.0]    xVV[ 1.0]    sVV[ 1.0]  xR2[ 1.0]  \
ADA         0.029       0.156  8.144000e+09  1.586000e+10       0.312
BNB         0.033       0.167  6.141000e+09  1.118000e+10       0.396
BTC         0.015       0.097  2.095000e+11  1.388000e+11       0.281
DOGE        0.057       0.399  8.287000e+09  2.726000e+10       0.327
ETH         0.023       0.127  1.000000e+11  8.054000e+10       0.313
LINK        0.040       0.179  6.707000e+09  1.665000e+10       0.319
LTC         0.015       0.134  2.513000e+10  1.731000e+10       0.320
XLM         0.013       0.176  4.157000e+09  5.469000e+09       0.321
XRP         0.014       0.164  2.308000e+10  2.924000e+10       0.322

         xm[ 1.0]
ADA   1.821000e-11
BNB   9.167000e-09
BTC   1.254000e-08
DOGE  1.459000e-12
ETH   1.737000e-09
LINK  1.582000e-09
LTC   1.816000e-09
XLM   1.876000e-11
XRP   7.996000e-12
[9 Alternatives x 6 Criteria]
```

Now let's create three different options to evaluate our alternatives: One based on `WeightedSumModel`, another one based on `WeightedProductModel` and a final one using `TOPSIS`.

[3]:
```
from skcriteria.pipeline import mkpipe
from skcriteria.preprocessing.invert_objectives import (
    InvertMinimize,
    NegateMinimize,
)
from skcriteria.preprocessing.filters import FilterNonDominated
from skcriteria.preprocessing.scalers import SumScaler, VectorScaler
from skcriteria.agg.simple import WeightedProductModel, WeightedSumModel
from skcriteria.agg.similarity import TOPSIS

ws_pipe = mkpipe(
    InvertMinimize(),
    FilterNonDominated(),
    SumScaler(target="weights"),
    VectorScaler(target="matrix"),
    WeightedSumModel(),
)
```

(continues on next page)

(continued from previous page)

```
wp_pipe = mkpipe(
    InvertMinimize(),
    FilterNonDominated(),
    SumScaler(target="weights"),
    VectorScaler(target="matrix"),
    WeightedProductModel(),
)

tp_pipe = mkpipe(
    NegateMinimize(),
    FilterNonDominated(),
    SumScaler(target="weights"),
    VectorScaler(target="matrix"),
    TOPSIS(),
)
```

Now let's run the three options and visualize the rankings

[4]:
```
wsum_result = ws_pipe.evaluate(dm)
wprod_result = wp_pipe.evaluate(dm)
tp_result = tp_pipe.evaluate(dm)

display(wsum_result, wprod_result, tp_result)
```

```
Alternatives  ADA  BNB  BTC  DOGE  ETH  LINK  LTC  XLM  XRP
Rank            8    2    1     7    3     5    6    4    9
[Method: WeightedSumModel]
```

```
Alternatives  ADA  BNB  BTC  DOGE  ETH  LINK  LTC  XLM  XRP
Rank            6    2    1     9    3     5    4    7    8
[Method: WeightedProductModel]
```

```
Alternatives  ADA  BNB  BTC  DOGE  ETH  LINK  LTC  XLM  XRP
Rank            6    2    1     8    5     3    4    7    9
[Method: TOPSIS]
```

Creating a `RanksComparator` instance

There are two ways to create the ranks comparators:

1. Rither we use the `RanksComparator` class giving a sequence `[("name0", rank0), ("name1", rank1), ..., ("nameN", rankN)]` and an extra, referring to extra information about the comparator.

[5]:
```
RanksComparator([("ts", tp_result), ("ws", wsum_result), ("wp", wprod_result)], extra={})
```

[5]:
```
<RanksComparator [ranks=['ts', 'ws', 'wp']]>
```

2. we let the names be inferred from the methods with the `mkrank_cmp()` function. In this case, the parameter "extra" wouldn't be necessary

[6]:
```
rcmp = mkrank_cmp(tp_result, wsum_result, wprod_result)
rcmp
```

[6]:
```
<RanksComparator [ranks=['TOPSIS', 'WeightedSumModel', 'WeightedProductModel']]>
```

`RankComparator` utilities

A set of useful statistics is provided to compare correlations, trends and covariances between the different rankings.

We can start by looking at the correlations

[7]:
```
rcmp.corr()  # by default the pearson correlation is used
```

[7]:
```
Method                 TOPSIS  WeightedSumModel  WeightedProductModel
Method
TOPSIS                1.000000          0.783333              0.916667
WeightedSumModel      0.783333          1.000000              0.816667
WeightedProductModel  0.916667          0.816667              1.000000
```

[8]:
```
rcmp.corr(method="kendall")  # or we can us the kendal correlation
```

[8]:
```
Method                 TOPSIS  WeightedSumModel  WeightedProductModel
Method
TOPSIS                1.000000          0.666667              0.777778
WeightedSumModel      0.666667          1.000000              0.666667
WeightedProductModel  0.777778          0.666667              1.000000
```

Covariances are also available

[9]:
```
rcmp.cov()
```

[9]:
```
Method                TOPSIS  WeightedSumModel  WeightedProductModel
Method
TOPSIS                 7.500             5.875                 6.875
WeightedSumModel       5.875             7.500                 6.125
WeightedProductModel   6.875             6.125                 7.500
```

And the R^2 score (the same as the linear regression) between rankings

[10]:
```
rcmp.r2_score()
```

[10]:
```
Method                 TOPSIS  WeightedSumModel  WeightedProductModel
Method
TOPSIS                1.000000          0.566667              0.833333
WeightedSumModel      0.566667          1.000000              0.633333
WeightedProductModel  0.833333          0.633333              1.000000
```

Another thing available is to analyze how far one ranking is from the other.

By default the Hamming distance is used, but any of the available `scipy.spatial.distance.pdist() <https://docs.scipy.org/doc/scipy/reference/generated/scipy.spatial.distance.pdist.html#scipy.spatial.distance.pdist>`__ functions can be used.

[11]:
```
rcmp.distance()
```

[11]:
```
Method                 TOPSIS  WeightedSumModel  WeightedProductModel
Method
TOPSIS                0.000000          0.666667              0.444444
WeightedSumModel      0.666667          0.000000              0.555556
WeightedProductModel  0.444444          0.555556              0.000000
```

[12]:
```
rcmp.distance(metric="cityblock")
```

[12]:
```
Method                TOPSIS  WeightedSumModel  WeightedProductModel
Method
TOPSIS                   0.0              12.0                   6.0
WeightedSumModel        12.0               0.0                  10.0
WeightedProductModel     6.0              10.0                   0.0
```

A distance function can also be provided

[13]:
```
def my_distance(u,v,w=None):
    return 42

rcmp.distance(metric=my_distance)
```

[13]:
```
Method                TOPSIS  WeightedSumModel  WeightedProductModel
Method
TOPSIS                   0.0              42.0                  42.0
WeightedSumModel        42.0               0.0                  42.0
WeightedProductModel    42.0              42.0                   0.0
```

Finally, if all this is insufficient, we can turn the comparator into a `pandasDataFrame`

[14]:
```
rcmp.to_dataframe()
```

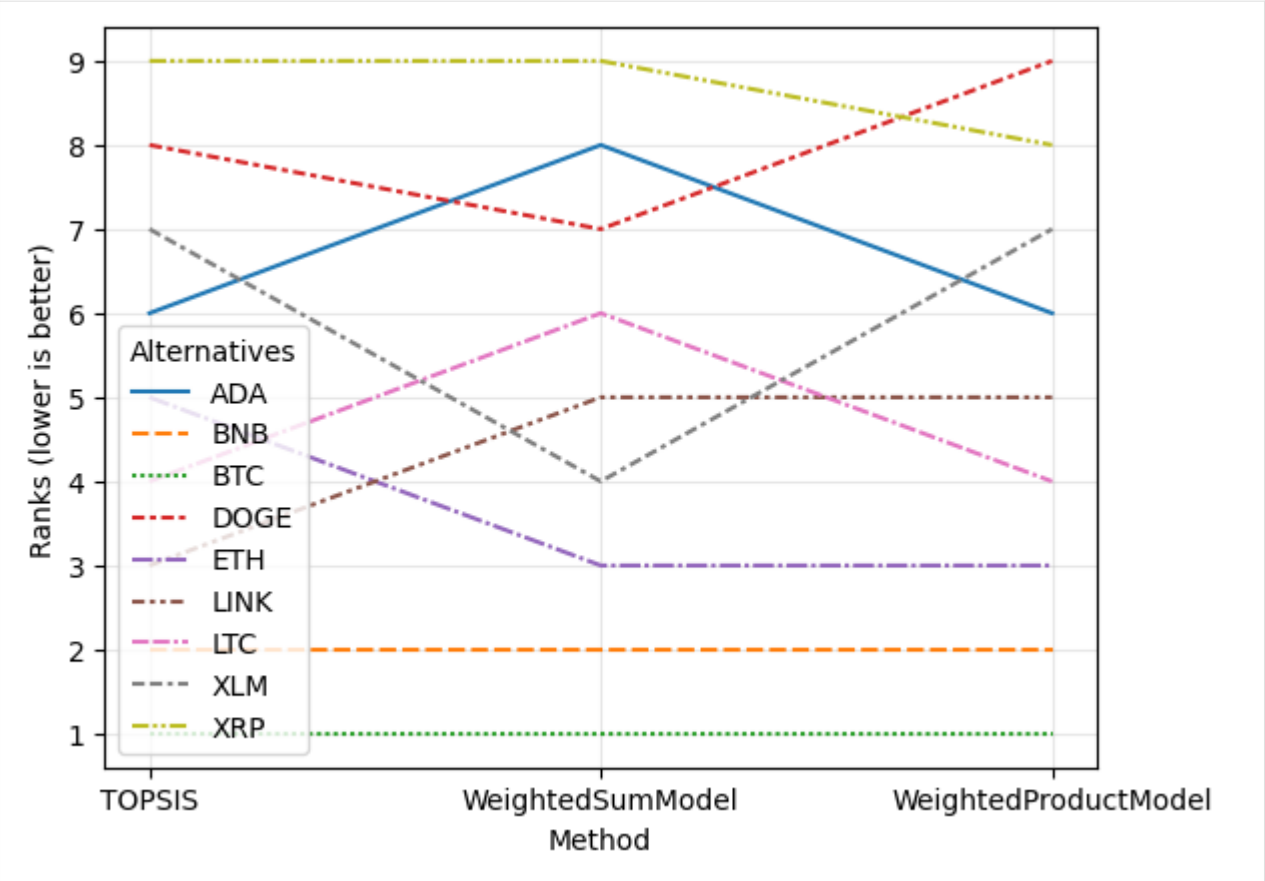
[14]:
```
Method        TOPSIS  WeightedSumModel  WeightedProductModel
Alternatives
ADA                6                 8                     6
BNB                2                 2                     2
BTC                1                 1                     1
DOGE               8                 7                     9
ETH                5                 3                     3
LINK               3                 5                     5
LTC                4                 6                     4
XLM                7                 4                     7
XRP                9                 9                     8
```

`RankComparator` Plots!

The other set of analysis tools are obviously the visualization tools.

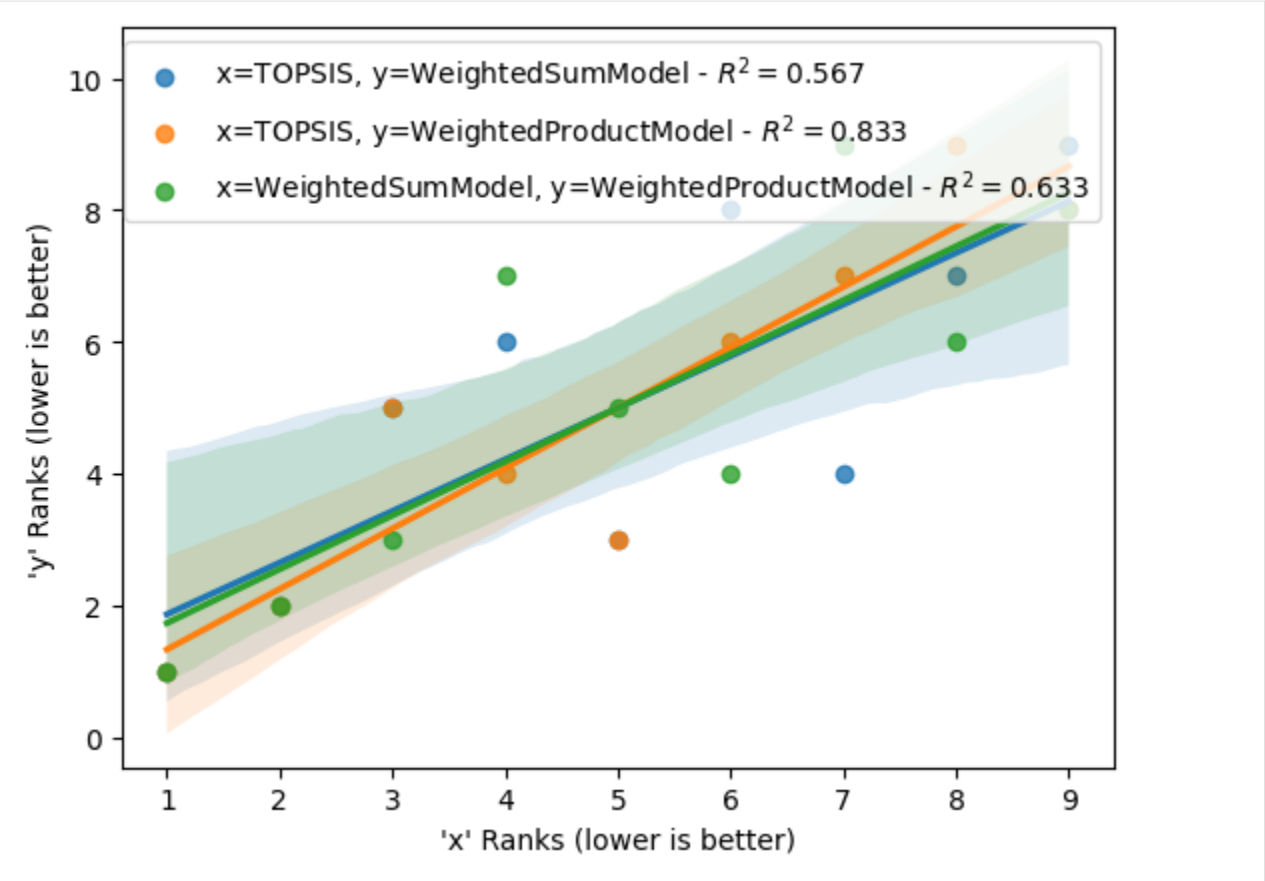
A classic in the area *Ranking flows*

[15]:
```
rcmp.plot.flow();
```



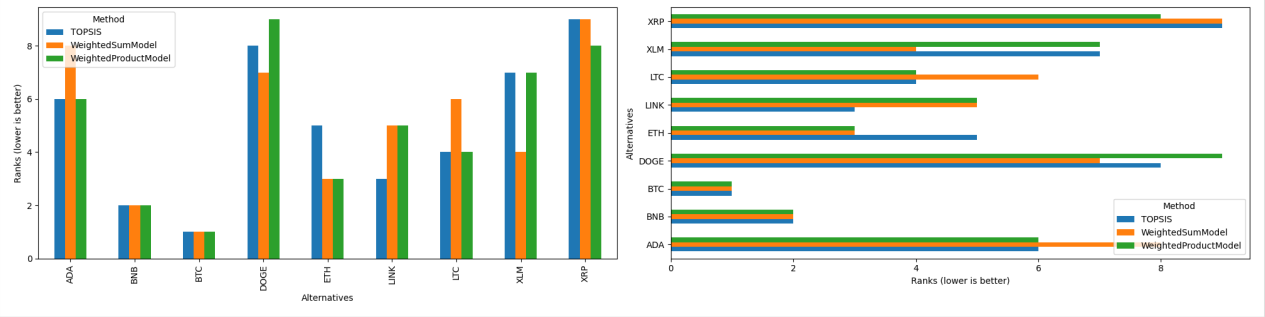
We can also run regressions on all combinations of different rankings.

[16]:
```
rcmp.plot.reg(r2=True, r2_fmt=".3f");
```



There are also bar plots

[17]:
```
import matplotlib.pyplot as plt

fig, axs = plt.subplots(1, 2, figsize=(20, 5))

rcmp.plot.bar(ax=axs[0])
rcmp.plot.barh(ax=axs[1])

fig.tight_layout();
```



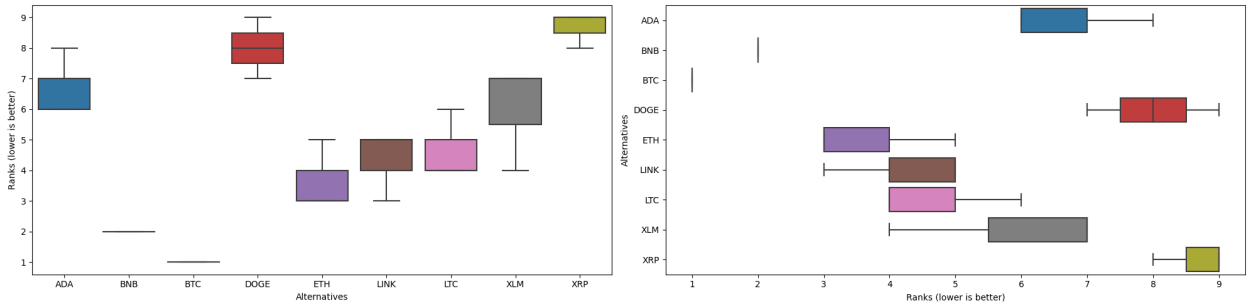
and boxplots

[18]:
```
fig, axs = plt.subplots(1, 2, figsize=(20, 5))
```

(continues on next page)

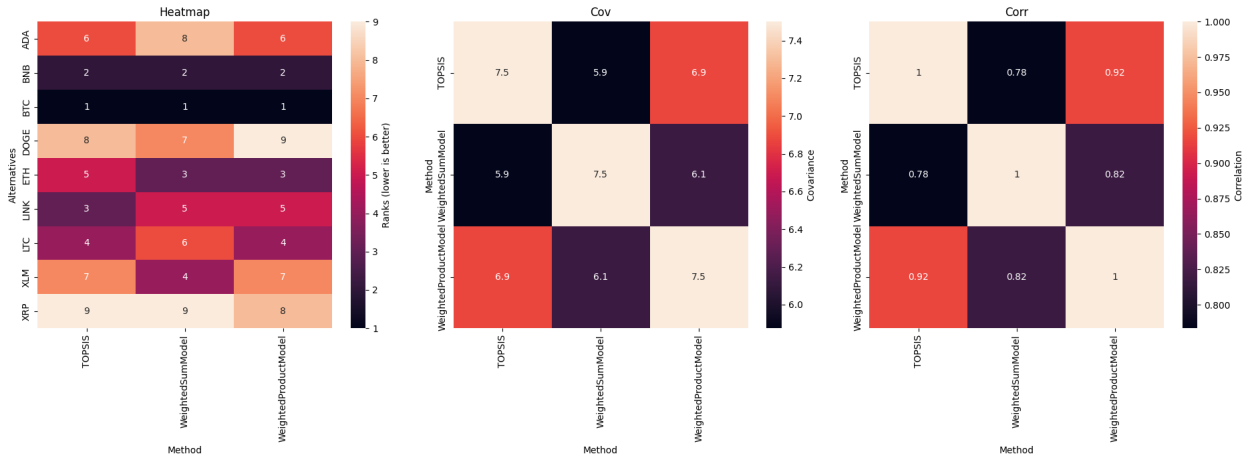
(continued from previous page)

```
rcmp.plot.box(ax=axs[0])
rcmp.plot.box(ax=axs[1], orient="h")

fig.tight_layout();
```

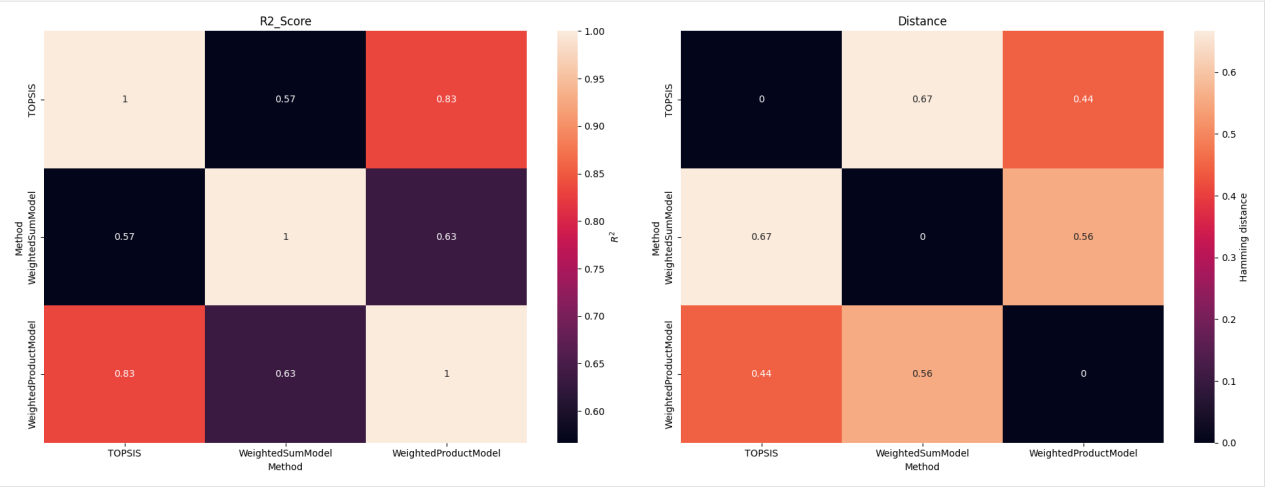


And you can visualize all matrices with statistical heatmap-like plots.

[19]:
```
fig, axs = plt.subplots(1, 3, figsize=(19, 7))

for kind, ax in zip(["heatmap", "cov", "corr"], axs):
    rcmp.plot(kind, ax=ax)
    ax.set_title(kind.title())


fig.tight_layout()
```



[20]:
```
fig, axs = plt.subplots(1, 2, figsize=(19, 7))

for kind, ax in zip(["r2_score", "distance"], axs):
    rcmp.plot(kind, ax=ax)
    ax.set_title(kind.title())

fig.tight_layout()
```



Generated by nbsphinx from a Jupyter notebook. 2025-08-03T00:34:25.057878

5.2.4 Rank reversals

This tutorial provides an overview of the use of the scikit-criteria ranking comparison tools.

Context and motivation

Multi-criteria decision methods may present irregularities in their rankings that compromise the reliability of their results. Depending on how the alternatives are defined and evaluated in a problem, rank reversals are classified in five types. To evaluate the robustness of these methods, Wang & Triantaphyllou (2006) proposed three test criteria that cover all five kinds of rank reversals.

1. **Rank invariance**: The rank of an optimal alternative must remain invariant when a sub-optimal alternative is worsened.
2. **Transitivity**: when a decision problem is partitioned in smaller sub-problems, the relative transitivity of alternatives must be retained.
3. **Recomposition consistency**: a new ranking constructed from a set of smaller sub-problems must be equivalent to the ranking obtained from running the original method.

Rank reversal checking in scikit-criteria

In scikit-criteria we implement rank reversal checking via the classes `RankInvariantChecker` and `TransitivityChecker`, which are available in the `skcriteria.ranksrev` module. Rank reversal checkers take an MCDM as a parameter, along with other relevant information, and implement an `evaluate()` method against a specific decision matrix.

In the following sections we'll review the usage of each checker to verify the validity of some example MCDMs.

Test criterion 1 - Rank Invariance

Test criterion 1 evaluates the stability of an MCDM method's top-ranked alternative under minor degradations of non-optimal alternatives, which roughly attempts to detect rank reversals due to irrelevant changes.

In its most basic form, the `RankInvariantChecker` works by worsening each sub-optimal alternative by a chosen amount `repeat` times, and stores every result in a `RanksComparator`.

Example

We will apply this test to the Van 2021 Evaluation dataset, which ranks cryptocurrencies using a sliding window of 7 days.

[12]:
```
import skcriteria as skc
from skcriteria.pipeline import mkpipe
from skcriteria.preprocessing.scalers import SumScaler, VectorScaler
from skcriteria.preprocessing.invert_objectives import InvertMinimize
from skcriteria.agg.similarity import TOPSIS
from skcriteria.ranksrev import RankInvariantChecker
```

[13]:
```
# Load Van 2021 Evaluation Dataset of cryptocurrencies
dm = skc.datasets.load_van2021evaluation(windows_size=7)

# Create the MCDA pipeline
dmaker = mkpipe(
    InvertMinimize(),
    SumScaler(target="weights"),
    VectorScaler(target="matrix"),
    TOPSIS()
)

original_result = dmaker.evaluate(dm)
display(original_result)
```

```
Alternatives  ADA  BNB  BTC  DOGE  ETH  LINK  LTC  XLM  XRP
Rank            8    2    1     5    4     6    7    3    9
[Method: TOPSIS]
```

Now we wrap our MCDA method with the `RankInvariantChecker`, which will:

- Apply small degradations to each suboptimal alternative,
- Recompute the ranking each time,
- Collect and compare all rankings.

We'll repeat the mutation twice per alternative.

[14]:
```
# Create the stability evaluator
rrt1 = RankInvariantChecker(
    dmaker=dmaker,
    repeat=2,
    allow_missing_alternatives=True
)
```

(continues on next page)

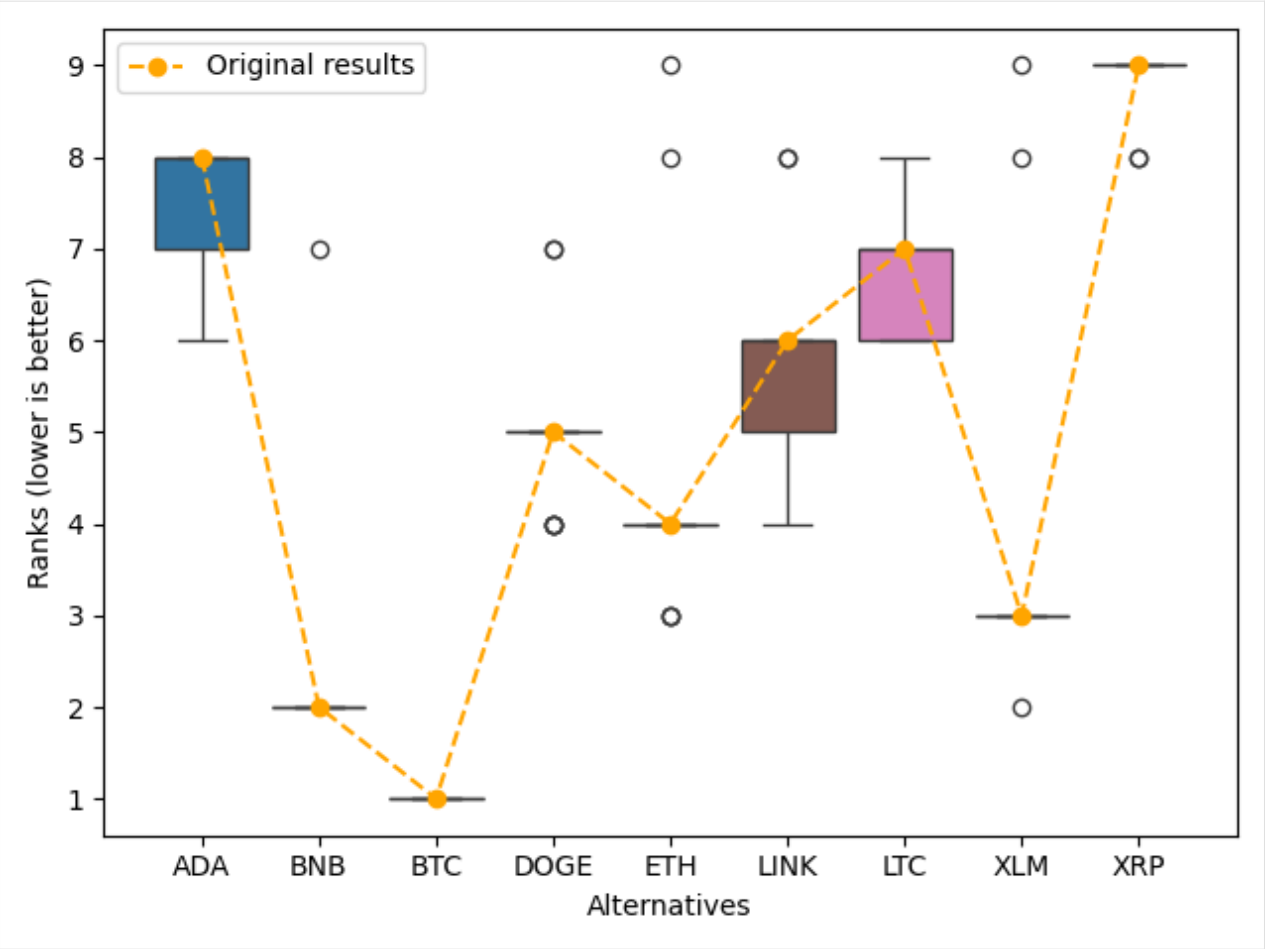
(continued from previous page)

```
# Execute the RRT1 test
comparison = rrt1.evaluate(dm)
```

We can plot a boxplot of the ranks received by each alternative across all RRT1 mutations.
This allows us to observe:

- How stable each alternative's position is,
- Whether the top-ranked option remains consistent,
- How sensitive each alternative is to perturbations.

[15]:
```
import matplotlib.pyplot as plt

ax = comparison.plot()
ax.plot(
    original_result,
    marker="o",
    linestyle="--",
    color="orange",
    label="Original results",
    zorder=3,
)
ax.legend()
plt.tight_layout();
```



Test criterions 2 & 3 - Transitivity and Recomposition Consistency

Test criterions 2 and 3 are closely connected in their goal of evaluating the internal consistency and robustness MCDMs via problem decomposition.

- **Test Criterion 2 (Transitivity):** Ensures that transitivity holds across pairwise comparisons that is, if alternative $A_1 \succ A_2$, and $A_2 \succ A_3$, then $A_1 \succ A_3$.
- **Test Criterion 3 (Recomposition Consistency):** Builds a total ordering from pairwise relationships to recompose them into a complete global ranking for comparison against the original ranking.

Because of this similarity, we opted to unify both criteria into a single class `TransitivityChecker`, that runs both tests secuentially and yields all the relevant details about the test criterions that were encountered in the process.

Example failing both test cases

For this example we'll use a dataset proposed by Poh & Ang (1999) in search of alternative fuels for land transportation in Singapore.

[16]:
```
import skcriteria as skc
from skcriteria.pipeline import mkpipe
from skcriteria.preprocessing.scalers import SumScaler, VectorScaler
from skcriteria.preprocessing.invert_objectives import InvertMinimize
from skcriteria.preprocessing.filters import FilterNonDominated
from skcriteria.agg.similarity import TOPSIS
from skcriteria.ranksrev.rank_transitivity_check import RankTransitivityChecker
```

[17]:
```
# Create a decision matrix
dm = skc.mkdm(
    matrix=[
        [4.57, 4.64, 62.2, 43.8, 8.49, 65.7],
        [65.7, 67.3, 6.03, 43.8, 4.25, 9.42],
        [20.3, 14.0, 20.2, 6.25, 58.3, 20.3],
        [9.42, 14.0, 11.5, 6.25, 29.0, 4.57],
    ],
    objectives=[max, max, max, max, max, max],
    alternatives=["status quo", "oil & ev", "oil & ngv", "methanol"],
    criteria=["supply", "emission", "tech", "safety", "cost", "consumer preference"],
)
```

To pick our best choice, we'll use a TOPSIS based method and print the resulting ranking.

[18]:
```
# Define TOPSIS pipeline
dmaker = mkpipe(
    SumScaler(target="weights"),
    VectorScaler(target="matrix"),
    TOPSIS()
)

# Get original ranking
original_result = dmaker.evaluate(dm)
display(original_result)
```

```
Alternatives  status quo  oil & ev  oil & ngv  methanol
Rank                  2         1          3         4
[Method: TOPSIS]
```

To verify that we made the right choice picking TOPSIS, we'll be using the `TransitivityChecker`. As we stated before, for this we'll need our original pipeline, but we also add two extra parameters:

- `allow_missing_alternatives`: desicion maker pipelines can sometimes return rankings with fewer alternatives than the original ones (using a pipeline that implements a filter, for example), which could cause issues in some partitions. This parameter allows for missing alternatives in a ranking to be added with a value of the maximum value of the ranking obtained + 1.
- `max_ranks`: limits the amount of rankings that can be constructed during the recomposition phase.

[19]:
```
checker = RankTransitivityChecker(
    dmaker,
```

(continues on next page)

(continued from previous page)

```
    allow_missing_alternatives=True,
    max_ranks=10
)
```

[20]:
```
results = checker.evaluate(dm=dm)

display("Transitivity Analysis Results:")
display(f"Test Criterion 2 (Transitivity): {results.e_.test_criterion_2}")
display(f"Test Criterion 3 (Recomposition Consistency): {results.e_.test_criterion_3}")
display(f"Transitivity Break Rate: {results.e_.transitivity_break_rate:.4f}")
```

```
'Transitivity Analysis Results:'
```

```
'Test Criterion 2 (Transitivity): False'
```

```
'Test Criterion 3 (Recomposition Consistency): False'
```

```
'Transitivity Break Rate: 0.5000'
```

The `TransitivityChecker` evaluation provides several key pieces of information:

1. **Test Criterion 2 Result**: Indicates whether the method maintains transitivity across all pairwise comparisons
2. **Test Criterion 3 Result**: Shows whether the recomposed ranking matches the original ranking
3. **Transitivity Break Rate**: A normalized measure of how many transitivity violations exist relative to the theoretical maximum.

Given that our pipeline failed the test, we should be able to see many different rankings of our original desicion matrix.

[21]:
```
display("Ranking Comparison")

items = list(results.named_ranks.items())
for rank_name, ranking in items[:4]:
    display(ranking)
```

```
'Ranking Comparison'
```

```
Alternatives  status quo  oil & ev  oil & ngv  methanol
Rank                   2         1          3         4
[Method: TOPSIS]
```

```
Alternatives  status quo  oil & ev  oil & ngv  methanol
Rank                   3         1          2         4
[Method: TOPSIS + RRT3 RECOMPOSITION_1]
```

```
Alternatives  status quo  oil & ev  oil & ngv  methanol
Rank                   1         2          3         4
[Method: TOPSIS + RRT3 RECOMPOSITION_2]
```

```
Alternatives  status quo  oil & ev  oil & ngv  methanol
Rank                   2         3          1         4
[Method: TOPSIS + RRT3 RECOMPOSITION_3]
```

Example that passes both tests

This example uses a simple stock selection decision matrix and applies a TOPSIS-based MCDA pipeline. The pipeline includes preprocessing steps such as inverting objectives (for maximization) and filtering non-dominated alternatives.

We then check whether the resulting rankings satisfy the transitivity and recomposition consistency properties, in such case, Both criteria should return True, and the transitivity break rate should be 0, indicating robust decision-making consistency.

[22]:
```
# Create a decision matrix for energy alternatives
dm = skc.datasets.load_simple_stock_selection()

# Define TOPSIS pipeline
dmaker = mkpipe(InvertMinimize(), FilterNonDominated(), TOPSIS())

# Get original ranking
original_result = dmaker.evaluate(dm)
display(original_result)

# Create the transitivity checker
checker = RankTransitivityChecker(dmaker, allow_missing_alternatives=True, max_
↪ranks=1000)

# Run the transitivity evaluation
results = checker.evaluate(dm=dm)
display(results["Original"])
display(results["Recomposition1"])

display("Transitivity Analysis Results:")
display(f"Test Criterion 2 (Transitivity): {results.e_.test_criterion_2}")
display(f"Test Criterion 3 (Recomposition Consistency): {results.e_.test_criterion_3}")
display(f"Transitivity Break Rate: {results.e_.transitivity_break_rate:.4f}")
```

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

```
Alternatives PE JN AA GN FX MM
Rank          3  4  2  1  5  5
[Method: TOPSIS]
```

```
Alternatives PE JN AA GN FX MM
Rank          3  4  2  1  5  5
[Method: TOPSIS + RRT3 RECOMPOSITION_1]
```

```
'Transitivity Analysis Results:'
```

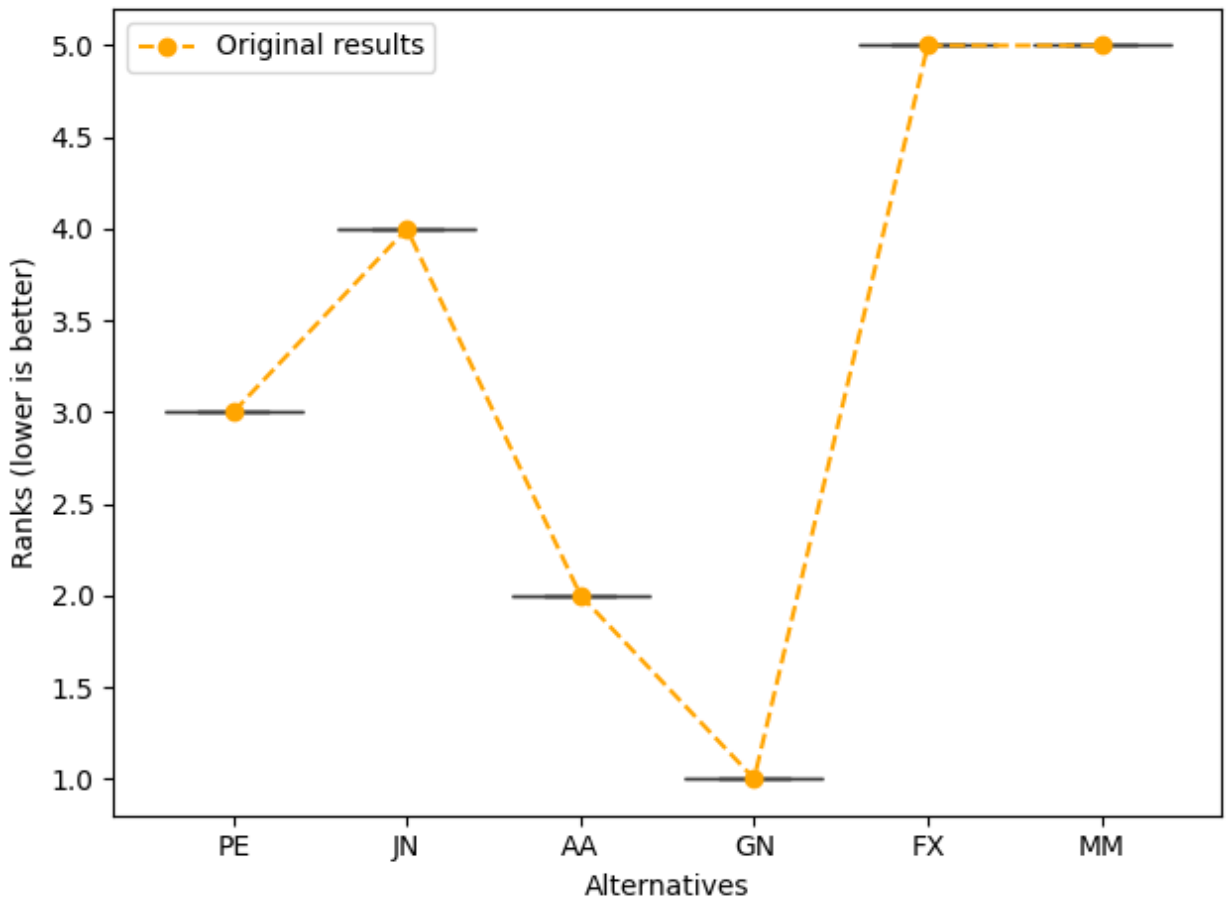
```
'Test Criterion 2 (Transitivity): True'
```

```
'Test Criterion 3 (Recomposition Consistency): True'
```

```
'Transitivity Break Rate: 0.0000'
```

The plot below shows the ranking results for each alternative, with confidence intervals indicating the stability of rankings across different evaluation scenarios. The dashed line connects the mean ranks to visualize potential transitivity issues in the decision matrix.

[23]:
```
import matplotlib.pyplot as plt

ax = results.plot()
ax.plot(
    results[0],
    marker="o",
    linestyle="--",
    color="orange",
    label="Original results",
    zorder=3,
)
ax.legend()
plt.tight_layout();
```



Untying and Recomposition strategies

When transitivity violations are detected in the dominance graph (i.e., cycles), the algorithm applies cycle-breaking strategies to restore consistency and generate valid rankings.

The strategy used is controlled via the `make_transitive_strategy` parameter:

- `random` (default): removes edges uniformly at random from cycles.
- `weighted`: removes edges with higher participation in cycles (based on frequency), prioritizing structurally disruptive edges. Custom strategies can also be provided as callables: `func(cycle, edge_freq, rng)` → `edge_tuple`.

Multiple acyclic graphs may be generated depending on the number of cycle-breaking configurations (`max_ranks`), each yielding a potentially different recomposed ranking.

Ties during pairwise evaluation are resolved through dominance analysis: if one alternative dominates the other in more criteria, it's considered superior even if the scores are tied.

To illustrate these strategies in practice, we revisit the decision matrix used in the *Example failing both test cases* section. We apply the `TransitivityChecker` to the original TOPSIS ranking and generate alternative rankings using two different cycle-breaking strategies: `random` and `weighted`.

This example shows how different strategies for restoring transitivity can affect the final ranking, even when based on the same input data.

[24]:
```
# Create a decision matrix
dm = skc.mkdm(
    matrix=[
        [4.57, 4.64, 62.2, 43.8, 8.49, 65.7],
        [65.7, 67.3, 6.03, 43.8, 4.25, 9.42],
        [20.3, 14.0, 20.2, 6.25, 58.3, 20.3],
        [9.42, 14.0, 11.5, 6.25, 29.0, 4.57],
    ],
    objectives=[max, max, max, max, max, max],
    alternatives=["status quo", "oil & ev", "oil & ngv", "methanol"],
    criteria=["supply", "emission", "tech", "safety", "cost", "consumer preference"
    ],
)

# Define TOPSIS pipeline
dmaker = mkpipe(
    SumScaler(target="weights"), VectorScaler(target="matrix"), TOPSIS()
)
original_result = dmaker.evaluate(dm)
```

[25]:
```
# Recomposition using different cycle-breaking strategies

# Use the default strategy (random)
checker_random = RankTransitivityChecker(
    dmaker,
    cycle_removal_strategy="random",
    max_ranks=1
)
res_random = checker_random.evaluate(dm=dm)
```

(continues on next page)

(continued from previous page)

```
# Use the 'weighted' strategy
checker_weighted = RankTransitivityChecker(
    dmaker,
    cycle_removal_strategy="weighted",
    max_ranks=1
)
res_weighted = checker_weighted.evaluate(dm=dm)

# Access recomposed rankings
rrandom = res_random["Recomposition1"]
rweighted = res_weighted["Recomposition1"]

display(original_result, rrandom, rweighted)
```

```
Alternatives  status quo  oil & ev  oil & ngv  methanol
Rank                   2         1          3         4
[Method: TOPSIS]
```

```
Alternatives  status quo  oil & ev  oil & ngv  methanol
Rank                   3         1          2         4
[Method: TOPSIS + RRT3 RECOMPOSITION_1]
```

```
Alternatives  status quo  oil & ev  oil & ngv  methanol
Rank                   1         2          3         4
[Method: TOPSIS + RRT3 RECOMPOSITION_1]
```

References

- Wang & Triantaphyllou (2006). Ranking Irregularities When Evaluating Alternatives by Using Some Multi-Criteria Decision Analysis Methods.
- Poh & Ang (1999). Transportation fuels and policy for Singapore: an AHP planning approach.

Generated by nbsphinx from a Jupyter notebook. 2025-08-03T00:34:25.057878

5.2.5 Extending Aggregation and Transformation Functions

This tutorial serves as a guide for utilizing the extension tools for aggregation and transformer functions in Scikit-Criteria. After going through this tutorial, you will be able to implement your own multi-criteria decision models compatible with the data types and tools provided by the library.

1. Introduction

In Scikit-Criteria, leveraging the provided decorators (`@extend.mkagg` and `@extend.mktransformer`) for extending aggregation and transformation functions provides a powerful means to customize decision-making models allowing the creation of custom functions, enabling domain-specific logic implementation for diverse use cases.

Decorators simplify the process of converting functions into model classes, promoting flexibility in model creation without complex class hierarchies. This facilitates quick prototyping and experimentation by allowing direct modification of functions. Additionally, the decorators handle hyperparameter initialization, encapsulating them within models, promoting clean, organized code and reducing the chances of errors related to parameter handling.

Example Usage:

[1]:
```
# Import the decorators from the module
from skcriteria.extend import mkagg, mktransformer

# Define custom aggregation and transformation functions
@mkagg
def CustomAggregation(**kwargs):
    # Implement aggregation logic
    pass

@mktransformer
def CustomTransformation(**kwargs):
    # Implement transformation logic
    pass
```

While this code is syntactically valid, attempting to use it may not work as intended since it doesn't return the required values.

2. A New Aggregation Model

To create a custom aggregation model, follow these steps:

1. Declare a function with the name of your model using the CapWords/UpperCamelCase/PascalCase convention. While this is not mandatory, not adhering to this convention will trigger a warning message from scikit-criteria, notifying that the model name does not follow the Scikit-Criteria standard.

[2]:
```
@mkagg
def bad_model_name(**kwargs):
    pass
```

```
/home/juanbc/proyectos/skcriteria/src/skcriteria/extend.py:211: NonStandardNameWarning:␣
↪Models names should normally use the 'CapWords' convention.try change 'bad_model_name'␣
↪to 'Bad_Model_Name' or 'BAD_MODEL_NAME'
  return _agg_maker if maybe_func is None else _agg_maker(maybe_func)
```

[3]:
```
@mkagg
def GodModelName(**kwargs):
    pass
```

2. The function should take parameters representing the decomposed decision matrix after calling the `DecisionMatrix.to_dict()` method, and a parameter `hparams`, which will be explained later and contains the hyper-parameters of the model.

- `hparams`: Model Hyperparameters.

- `matrix`: Alternatives matrix as numpy array.
- `objectives`: numpy array of objectives for criteria as integers: $maximize = 1$ and $minimize = -1$.
- `weights`: Weights of the criteria as numpy array.
- `dtypes`: Data types of the criteria as numpy array.
- `alternatives`: Names of the alternatives as numpy array.
- `criteria`: Names of the criteria as numpy array.

Additionally, if you do not want to use any of those parameters of the matrix, you can declare the function with Variable Keyword Arguments (**kwargs).

If any parameter is forgotten and `**kwargs` is not present, a TypeError is raised.

So this next two functions are a valid Aggregation functions

[4]:
```
@mkagg
def AllParameters(hparams, matrix, objectives, weights, dtypes, alternatives, criteria):
    pass

@mkagg
def OnlyTwoWithKwargs(matrix, weights, **kwargs):
    pass
```

3. Utilizing the received parameters, the function should return two objects:
    1. A `numpy.array/list/tuple` or any kind of sequence containing a valid ranking. Where the `i`-th position in the returned sequence has the ranking value for the `i`-th alternative in the array of alternatives received as a parameter.
    2. A `dict` with extra values from the ranking (intermediate results or other useful data for decision-making analysis).

Note: Understanding the Rankings

A valid ranking has the following conditions:

1. **Length:** It should have the same length as the number of alternatives received by the function.
2. **Ascending and Consecutive Order:** The values must be in ascending order and consecutive. This means that values should start from 1 and increase by increments of 1 without skips For example, `[1, 2, 3, 4]` and `[1, 2, 1]` is valid, but `[4, 2, 4, 1]` is not valid because the value 3 is missing.
3. **Integers Only:** Values must be integers. Fractional or other types of values are not allowed.

So if we have the alternatives `["banana", "apple", "orange"]` and the ranking `[1, 2, 1]`

The meaning of the ranking in relation to the alternatives is as follows:

1. The first position in the ranking is 1, indicating that the alternative in the first position is the most preferred or the best choice.
2. The second position in the ranking is 2, suggesting that the alternative in the second position is the second-best choice.
3. The third position in the ranking is also 1, implying that the alternative in the third position is equally preferred to the alternative in the first position.

Therefore, the ranking `[1, 2, 1]` could be interpreted as stating that "Banana" and "Orange" are equally preferred, and "Apple" is the second preferred choice. It's important to note that the ranking must adhere to the specific conditions mentioned in the definitions, such as the correct length, ascending and consecutive order, and integer values.

With all of this, a complete and valid aggregation function would be:

[ ]:
```
import numpy as np

@mkagg
def AllAlternativesAreFirst(alternatives, **kwargs):
    # Assign a rank of 1 to each alternative
    rank = [1] * len(alternatives)

    # Define extra information (example: some important value)
    extra = {"some_important_value": "the_important_value"}

    # Return the rank and extra information
    return rank, extra
```

Let's test the new aggregation with a dataset.

[6]:
```
import skcriteria as skc
dm = skc.datasets.load_simple_stock_selection() # load the dataset
dm
```

[6]:
```
    ROE[ 2.0]  CAP[ 4.0]  RI[ 1.0]
PE          7          5         35
JN          5          4         26
AA          5          6         28
FX          3          4         36
MM          1          7         30
GN          5          8         30
[6 Alternatives x 3 Criteria]
```

[7]:
```
# Instantiate the new aggregation
agg = AllAlternativesAreFirst()
agg
```

[7]:
```
<AllAlternativesAreFirst []>
```

[8]:
```
# evaluate
rank = agg.evaluate(dm)
rank
```

[8]:
```
Alternatives  PE  JN  AA  FX  MM  GN
Rank           1   1   1   1   1   1
[Method: AllAlternativesAreFirst]
```

[9]:
```
rank.e_.some_important_value
```

[9]:
```
'the_important_value'
```

3. Hyperparameters

The Hyper-parameters (in the context of machine learning) are parameters that allow you to specify details on how the function will carry out its aggregation. In this sense, they are more similar to Free-Parameters as they cannot be predicted or constrained by the model.

In Scikit-Criteria, we define the concept of Hyper-parameters similar to the Hyper-parameters in Scikit-Learn: Parameters received by the model's (Aggregation function class) constructor and **always** should have some default value.

For example, in the case of Scikit-Criteria's implementation of TOPSIS, it has a hyper-parameter for the metric it will use, and by default, it is set to `"euclidean"`.

[ ]:
```
from skcriteria.agg import topsis
topsis.TOPSIS()
```

```
<TOPSIS [metric='euclidean']>
```

[ ]:
```
topsis.TOPSIS(metric="cityblock")
```

```
<TOPSIS [metric='cityblock']>
```

The hyper-parameters can be provided as named parameters to the `@mkagg` decorator, and their values can be accessed using the `hparams` parameter.

> **Note: Regarding the nature of hparams**
>
> If you are familiar with how methods work in Python classes, `hparams` is essentially the `self` of the model.

Now, for example, if we want to create a model named `MaybeWSM`, which is a weighted-sum-model that uses weights only when the `use_weight` hyperparameter is set to `True`, and the default value is indeed `True`.

[ ]:
```
import numpy as np

from skcriteria.utils import rank

@mkagg(use_weights=True)
def MaybeWSM(hparams, matrix, objectives, weights, **kwargs):
    """The Maybe-Weighted Sum Model (WSM) to rank alternatives.

    If the use_weights parameter in hparams is set to True, the
    function applies weights to the decision matrix. This is done
    by taking the inner product of the matrix and the weights vector.

    """
    # Check if objectives contain -1 (minimize objectives)
    if -1 in objectives:
        raise ValueError("'MaybeWSM' cant operate with minimize objectives")

    # If use_weights is True, apply weights to the matrix
    if hparams.use_weights:
        matrix = matrix * weights

    # Calculate the scores by row/alternative
    score = np.sum(matrix, axis=1)
```

(continues on next page)

(continued from previous page)

```
    # rank_values calculates the ranking based on the scores.
    # `reverse = True` indicates that higher scores are closer to the 1st place.
    # Additionally, we will return the calculated 'score' as extra information.
    return rank.rank_values(score, reverse=True), {"score": score}
```

Let's use our MaybeWSM model.

First, let's see what happens if we create a `MaybeWSM` with the default (`use_weights=True`) and try to evaluate the available decision matrix (`dm`).

[13]:
```
with_useweight = MaybeWSM()
with_useweight
```

[13]:
```
<MaybeWSM [use_weights=True]>
```

If we use `dm` as it is right now, we will get an exception: `'MaybeWSM' can't operate with minimize objectives` because, indeed, `dm` has some criteria to minimize.

[14]:
```
dm.minwhere  # the critetia to minimize
```

[14]:
```
ROE    False
CAP    False
RI      True
Name: minwhere, dtype: bool
```

For this reason, first, we will use the `InvertMinimize` transformer to eliminate criteria to minimize.

[15]:
```
from skcriteria.preprocessing import invert_objectives

dm = invert_objectives.InvertMinimize().transform(dm)
dm.minwhere
```

[15]:
```
ROE    False
CAP    False
RI     False
Name: minwhere, dtype: bool
```

[16]:
```
rank_with_uw = with_useweight.evaluate(dm)
rank_with_uw
```

[16]:
```
Alternatives  PE  JN  AA  FX  MM  GN
Rank           3   5   2   6   4   1
[Method: MaybeWSM]
```

Now, let's try `use_weights=False`.

[17]:
```
without_useweight = MaybeWSM(use_weights=False)
without_useweight
```

[17]:
```
<MaybeWSM [use_weights=False]>
```

[18]:
```
rank_without_uw = without_useweight.evaluate(dm)
rank_without_uw
```

[18]:
```
Alternatives  PE  JN  AA  FX  MM  GN
Rank           2   4   3   6   5   1
[Method: MaybeWSM]
```

It can be seen that depending on the configuration of the hyperparameter `use_weights`, the results are different.

In addition to this, the `score` is available within `extra_`.

[19]:
```
rank_with_uw.e_.score, rank_without_uw.e_.score
```

[19]:
```
(array([34.02857143, 26.03846154, 34.03571429, 22.02777778, 30.03333333,
        42.03333333]),
 array([12.02857143,  9.03846154, 11.03571429,  7.02777778,  8.03333333,
        13.03333333]))
```

3. A New Transformer

The only difference between creating a new aggregator and a transformer lies in the type of data returned by the decorated function. Everything else is exactly the same (received parameters, function names, and functionality of hyperparameters).

The decorated function must return a dictionary that can have the same keys as the parameters received by the function except for `hparam`: `matrix`, `objectives`, `weights`, `dtypes`, `alternatives`, or `criteria`; and whose values must be the new values with which to replace the original ones in the transformation matrix.

It is not necessary to return all values; only the ones that you want to change.

For example, if we want to create a transformer `StrFormat` that converts the text of the names of each criterion and alternative using the methods of `str`, and by default, it converts texts to lowercase.

[ ]:
```
@mktransformer(operation=str.lower)
def StrFormat(alternatives, criteria, hparams, **kwargs):
    """Applies a string formatting operation (lowercasing by default) to alternatives␣
↪and criteria."""
    # Apply the string formatting operation to each alternative
    new_alternatives = [hparams.operation(a) for a in alternatives]

    # Apply the string formatting operation to each criterion
    new_criteria = [hparams.operation(c) for c in criteria]

    # Return the transformed alternatives and criteria in a dictionary
    return {"alternatives": new_alternatives, "criteria": new_criteria}

trans = StrFormat()
trans
```

```
<StrFormat [operation=<method 'lower' of 'str' objects>]>
```

[21]:
```
trans.transform(dm)
```

[21]:
```
    roe[ 2.0]  cap[ 4.0]  ri[ 1.0]
pe          7          5   0.028571
jn          5          4   0.038462
aa          5          6   0.035714
fx          3          4   0.027778
```

(continues on next page)

(continued from previous page)

```
mm           1           7   0.033333
gn           5           8   0.033333
[6 Alternatives x 3 Criteria]
```

We can use any function provided by `str`.

[22]:
```
trans = StrFormat(operation=str.capitalize)
trans
```

[22]:
```
<StrFormat [operation=<method 'capitalize' of 'str' objects>]>
```

[23]:
```
trans.transform(dm)
```

[23]:
```
    Roe[ 2.0]  Cap[ 4.0]  Ri[ 1.0]
Pe           7           5   0.028571
Jn           5           4   0.038462
Aa           5           6   0.035714
Fx           3           4   0.027778
Mm           1           7   0.033333
Gn           5           8   0.033333
[6 Alternatives x 3 Criteria]
```

In fact, given our implementation, any arbitrary function that converts text can be used. For example, if we want to create our own function that adds exclamation marks to the end of each criterion and alternative.

[24]:
```
def add_exclamation(text):
    return text + " !! "

trans = StrFormat(operation=add_exclamation)
trans
```

[24]:
```
<StrFormat [operation=<function add_exclamation at 0x79a56af36f80>]>
```

[25]:
```
trans.transform(dm)
```

[25]:
```
        ROE !! [ 2.0]  CAP !! [ 4.0]  RI !! [ 1.0]
PE !!               7               5       0.028571
JN !!               5               4       0.038462
AA !!               5               6       0.035714
FX !!               3               4       0.027778
MM !!               1               7       0.033333
GN !!               5               8       0.033333
[6 Alternatives x 3 Criteria]
```

3.1 Special considerations regarding `dtypes`

By design decision, scikitcriteria always attempts to **always** preserve the original data types, unless it needs to infer them again.

This may not seem important to a user at first glance, so let's use an example of a transformer affected by this characteristic.

First, let's reload the original decision matrix, where the values of all criteria are `int`.

[26]:
```
dm = skc.datasets.load_simple_stock_selection()
dm
```

[26]:
```
    ROE[ 2.0]  CAP[ 4.0]  RI[ 1.0]
PE          7          5         35
JN          5          4         26
AA          5          6         28
FX          3          4         36
MM          1          7         30
GN          5          8         30
[6 Alternatives x 3 Criteria]
```

Now, let's create a transformer that converts all criteria to the `float` type.

[ ]:
```
@mktransformer
def AsFloat(matrix, **kwargs):
    """Converts the elements of a decision-matrix to floating-point numbers."""
    # Convert the elements of the matrix to floating-point numbers
    new_matrix = matrix.astype(float)

    # Return the transformed matrix in a dictionary
    return {"matrix": new_matrix}

trans = AsFloat()
trans
```

```
<AsFloat []>
```

Now, let's test its functionality.

[28]:
```
trans.transform(dm)
```

[28]:
```
    ROE[ 2.0]  CAP[ 4.0]  RI[ 1.0]
PE          7          5         35
JN          5          4         26
AA          5          6         28
FX          3          4         36
MM          1          7         30
GN          5          8         30
[6 Alternatives x 3 Criteria]
```

As can be seen, the numbers are still integers. This is because the `dtypes` parameter of the matrix indicates that those columns are indeed integers.

[29]:
```
dm.dtypes  # check the dtypes
```

[29]:
```
ROE     int64
CAP     int64
RI      int64
dtype: object
```

The simplest solution would be to ensure that the dtypes are inferred again based on the values of the new matrix. This is achieved by assigning the dtype values to None.

[ ]:
```
@mktransformer
def AsFloat(matrix, **kwargs):
    """Converts the elements of a decision-matrix to floating-point numbers."""
    # Convert the elements of the matrix to floating-point numbers
    new_matrix = matrix.astype(float)

    # Return the transformed matrix in a dictionary
    # and assign the dtypes as None
    return {"matrix": new_matrix, "dtypes": None}

trans = AsFloat()
trans.transform(dm)
```

```
    ROE[ 2.0]  CAP[ 4.0]  RI[ 1.0]
PE         7.0         5.0       35.0
JN         5.0         4.0       26.0
AA         5.0         6.0       28.0
FX         3.0         4.0       36.0
MM         1.0         7.0       30.0
GN         5.0         8.0       30.0
[6 Alternatives x 3 Criteria]
```

While this may seem somewhat inconvenient, it gives the user complete control over the data types of the matrix without assuming default behaviors that may be undesirable.

It's essential to consider that the original `dtypes` are also received by the transformer (in our case, they are inside `**kwargs`) and can be used to determine the new types.

Generated by nbsphinx from a Jupyter notebook. 2025-08-03T00:34:25.057878

5.2.6 Extra tutorials

This section is a collection of articles, blog-posts and other curated materials, written outside of core developers.

5.2.7 Scientific articles

Scientific articles or paper is an academic work that is usually published in an academic journal. It contains original research results or reviews existing results. Such a paper, also called an article, will only be considered valid if it undergoes a process of peer review by one or more referees who check that the content of the paper is suitable for publication in the journal [Wikipedia contributors, 2023].

Several bibliographic databases organize digital collections of references to published literature, including journal and newspaper articles and conference proceedings. The following links contain publications that cite the Scikit-Criteria paper [Cabral et al., 2016], and present novel applications of multi-criteria models to different scientific areas.

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're already familiar with Python and eager to explore the scientific stack further, be sure to check out the Scipy Lecture Notes

5.3 skcriteria package

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

5.3.1 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: *DiffEqualityMixin*

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.DatFrame`) – Dataframe where the index is the alternatives and the columns are the criteria.
- **objectives** (`numpy.ndarray`) – Aan 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, al the alternatives are names "A[n]" where n is the number of the row of *matrix* statring at 0.
- **criteria** (Iterable o None (default None)) – Optional names of the criteria. If is None, al the alternatives are names "C[m]" where m is the number of the columns of *matrix* statring 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.

constant_criteria(*std_kws=None*, *isclose_kws=None*)

Identifies criteria with constant values based on std deviation.

This method calculates the standard deviation of each column and determines which are effectively constant (standard deviation ~ 0) using numerical comparison with tolerance.

Parameters

- **std_kws** (*dict, optional*) – Additional keyword arguments for pandas.DataFrame.std(). Default: {}
- **isclose_kws** (*dict, optional*) – Additional keyword arguments for numpy.isclose(). Default: {}

Returns

Boolean series where True indicates the column is constant. Index corresponds to DataFrame column names. Series name is 'ConstantsCriteria'.

Return type

pandas.Series

copy(***kwargs*)

Create a copy of the current DecisionMatrix instance.

Deprecated since version 0.9: Using kwargs with copy() is deprecated. Use DecisionMatrix.replace() instead.

Parameters

****kwargs** (*dict, optional (deprecated)*) – Keyword arguments to modify attributes in the copied instance. This parameter is deprecated.

Returns

A new DecisionMatrix instance with the same data as the original.

Return type
DecisionMatrix

See also

replace
Preferred method to create a copy with modifications.

`replace`(***kwargs*)

Create a new DecisionMatrix instance with updated attributes.

Creates a copy of the current DecisionMatrix and updates it with the provided keyword arguments.

Parameters
****kwargs** (*dict*) – Keyword arguments specifying attributes to modify in the new instance. Any valid DecisionMatrix attribute can be updated.

Returns
A new DecisionMatrix instance with the updated attributes.

Return type
DecisionMatrix

Examples

```
>>> dm = DecisionMatrix(...)
>>> new_dm = dm.replace(weights=[0.3, 0.7])
```

`to_dataframe`()

Convert the entire DecisionMatrix into a dataframe.

The objectives and weights ara 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
```

(continues on next page)

(continued from previous page)

```
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.

to_latex(*bold_columns=True*, ***kwargs*)

Generate LaTeX table.

Parameters

- **bold_columns** (*bool, default=True*) – If True, bold the columns.
- **pandas.DataFrame.to_latex().** (*Same parameters as*)

Returns
LaTeX table.

Return type
str

Notes

By default, this method uses `bold_rows=True`.

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.

Deprecated since version 0.6: Use `DecisionMatrix.stats()`, `DecisionMatrix.stats('describe)` or `DecisionMatrix.stats.describe()` instead.

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

Returns
Summary statistics of DecisionMatrix provided.

Return type
`pandas.DataFrame`

to_dmsy(*filepath_or_buffer=None*)

Save a DecisionMatrix to a DMSY format file or buffer.

Parameters
filepath_or_buffer (*str or file-like object or None*) – Path where to save the DMSY file or a file-like object to write to. if None, return the DMSY data as a string

Returns
DMSY data as a string if filepath_or_buffer is None else None

Return type
str

Examples

```
>>> import skcriteria as skc
>>> dm = skc.mkdm([[1, 2], [3, 4]], [max, min])
>>> skc.io.to_dmsy(dm, "output.dmsy")
```

property shape

Return a tuple with (number_of_alternatives, number_of_criteria).

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

diff(*other*, *rtol=1e-05*, *atol=1e-08*, *equal_nan=True*, *check_dtypes=False*)

Return the difference between two objects within a tolerance.

This method should be implemented by subclasses to define how differences between objects are calculated.

The tolerance parameters rtol and atol, equal_nan, and check_dtypes are provided to be used by the numpy and pandas equality functions. These parameters allow you to customize the behavior of the equality comparison, such as setting the relative and absolute tolerance for numeric comparisons, considering NaN values as equal, and checking for the data type of the objects being compared.

Parameters

- **other** (*object*) – The object to compare to.
- **rtol** (*float, optional*) – The relative tolerance parameter. Default is 1e-05.
- **atol** (*float, optional*) – The absolute tolerance parameter. Default is 1e-08.
- **equal_nan** (*bool, optional*) – Whether to consider NaN values as equal. Default is True.
- **check_dtypes** (*bool, optional*) – Whether to check the data type of the objects. Default is False.

Returns
The difference between the current and the other object.

Return type
the_diff

See also

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

Notes

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.

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, *ìloc* ` 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, *ìloc* ` 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`, al the alternatives are names "A[n]" where n is the number of the row of *matrix* statring at 0.
- **criteria** (Iterable o None (default `None`)) – Optional names of the criteria. If is `None`, al the alternatives are names "C[m]" where m is the number of the columns of *matrix* statring 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.

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 is True, it indicates that this alternative is dominated by at least one other alternative.

Return type
pandas.Series

dominators_of = <methodtools._LruCacheWire object>

has_loops(*, *strict=False*)

Returns True if there is a loop in the dominance graph.

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

This method uses the networkx library to compute the dominance graph and check if it is a DAG.

skcriteria.core.methods module

Core functionalities of scikit-criteria.

class skcriteria.core.methods.**SKCMethodABC**

Bases: `object`

Base class for all class in scikit-criteria.

Notes

All subclasses should specify:

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

If the class is *abstract* all validations are turned off.

get_method_name()

Return the name of the method as string.

get_parameters()

Return the parameters of the method as dictionary.

copy(***kwargs*)

Create a copy of the current SKCMethodABC instance.

Deprecated since version 0.9: Using kwargs with copy() is deprecated. Use SKCMethodABC.replace() instead.

Parameters
****kwargs** (*dict, optional*) – Keyword arguments to modify attributes in the copied instance. This parameter is deprecated.

Returns
A new instance with the same data as the original.

Return type
SKCMethodABC

See also

replace
Preferred method to create a copy with modifications.

Examples

```
>>> method = SKCMethodABC(...)
>>> method_copy = method.copy()
```

replace(***kwargs*)

Create a new instance with updated parameters.

This method creates a new instance of the class with the same parameters as the current instance, but with the option to override specific parameters through kwargs.

Parameters
****kwargs** (*dict*) – Keyword arguments to override in the new instance. Any parameter that is valid for the class constructor can be specified.

Returns
A new instance with updated parameters.

Return type
SKCMethodABC

Examples

```
>>> method = SKCMethodABC(...)
>>> new_method = method.replace(parameter=new_value)
```

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 an objective instance 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.

Deprecated since version 0.8: Use `MAX/MIN.to_symbol()` instead.

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.

5.3.2 skcriteria.io package

Input/Output utilities for scikit-criteria.

This module provides functions for reading and writing DecisionMatrix objects.

skcriteria.io.dmsy module

DMSY (Decision Matrix Simple YAML) format support.

This module provides functions to read and write DecisionMatrix objects in DMSY format, a simple YAML-based format designed for easy human readability and editing of multi-criteria decision analysis data.

The DMSY format generates YAML files optimized for human readability through intelligent use of flow and block styles. One-dimensional arrays are represented in compact flow style (e.g., [1, 2, 3]), while multi-dimensional structures use block style for the outer dimension with flow style for inner dimensions (e.g., - [1, 2, 3] on separate lines). This combination allows complex decision matrices to maintain a clear and navigable structure, facilitating both manual inspection and direct file editing, without sacrificing compactness when appropriate.

skcriteria.io.dmsy.**DEFAULT_DM_TYPE = 'discrete-dm'**

Default decision matrix type for DMSY format. Currently only supports discrete multi-criteria decision matrices.

skcriteria.io.dmsy.**DEFAULT_DMSY_VERSION = 1**

Default DMSY format version number. Version 1 is the current stable format specification.

class skcriteria.io.dmsy.**CustomYAMLDumper**(*stream*, *default_style=None*, *default_flow_style=False*, *canonical=None*, *indent=None*, *width=None*, *allow_unicode=None*, *line_break=None*, *encoding=None*, *explicit_start=None*, *explicit_end=None*, *version=None*, *tags=None*, *sort_keys=True*)

Bases: SafeDumper

Custom YAML Dumper that handles numpy arrays and uses flow style for lists.

This dumper automatically converts numpy arrays to Python lists and formats all lists using flow style (e.g., [1, 2, 3] instead of block style).

yaml_multi_representers = {<class 'numpy.generic'>: <function _numpy_scalar_representer>}

yaml_representers = {<class 'NoneType'>: <function SafeRepresenter.represent_none>, <class 'bool'>: <function SafeRepresenter.represent_bool>, <class 'bytes'>: <function SafeRepresenter.represent_binary>, <class 'datetime.date'>: <function SafeRepresenter.represent_date>, <class 'datetime.datetime'>: <function SafeRepresenter.represent_datetime>, <class 'dict'>: <function SafeRepresenter.represent_dict>, <class 'float'>: <function SafeRepresenter.represent_float>, <class 'frozenset'>: <function _iterable_not_ndarray_representer>, <class 'int'>: <function SafeRepresenter.represent_int>, <class 'list'>: <function _iterable_not_ndarray_representer>, <class 'numpy.ndarray'>: <function _numpy_array_representer>, <class 'set'>: <function _iterable_not_ndarray_representer>, <class 'str'>: <function SafeRepresenter.represent_str>, <class 'tuple'>: <function _iterable_not_ndarray_representer>, None: <function SafeRepresenter.represent_undefined>}

class skcriteria.io.dmsy.**DMSYDiscreteHandlerV1**

Bases: object

Handler for discrete decision matrices in DMSY format version 1.

Parameters

- **dm_type** (*str*) – The decision matrix type this handler supports.
- **version** (*int*) – The DMSY format version this handler supports.

dm_type = 'discrete-dm'

version = 1

to_dm(*dm_data*)

Convert DMSY data to DecisionMatrix object.

Parameters
dm_data (*dict*) – Dictionary containing the decision matrix data from DMSY format.

Returns
The constructed DecisionMatrix object.

Return type
DecisionMatrix

to_yml(*dm*)

Convert DecisionMatrix object to DMSY-compatible data.

Parameters
dm (DecisionMatrix) – The DecisionMatrix object to convert.

Returns
Dictionary containing the data in DMSY format.

Return type
dict

skcriteria.io.dmsy.**read_dmsy**(*filepath_or_buffer*)

Load a DecisionMatrix from a DMSY format file or buffer.

Parameters
filepath_or_buffer (*str or file-like object*) – Path to the DMSY file or a file-like object containing DMSY data.

Returns
The loaded DecisionMatrix object.

Return type
DecisionMatrix

Examples

```
>>> import skcriteria as skc
>>> dm = skc.io.read_dmsy("dataset.dmsy")
```

skcriteria.io.dmsy.**to_dmsy**(*dm*, *filepath_or_buffer*)

Save a DecisionMatrix to a DMSY format file or buffer.

Parameters

- **dm** (DecisionMatrix) – The DecisionMatrix object to save.
- **filepath_or_buffer** (*str or file-like object*) – Path where to save the DMSY file or a file-like object to write to.

Examples

```
>>> import skcriteria as skc
>>> dm = skc.mkdm([[1, 2], [3, 4]], [max, min])
>>> skc.io.to_dmsy(dm, "output.dmsy")
```

5.3.3 skcriteria.agg package

MCDA aggregation methods and internal machinery.

skcriteria.agg._agg_base module

Core functionalities to create madm decision-maker classes.

class skcriteria.agg._agg_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.agg._agg_base.**ResultABC**(*method*, *alternatives*, *values*, *extra*)

Bases: *DiffEqualityMixin*

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)

diff(*other*, *rtol=1e-05*, *atol=1e-08*, *equal_nan=False*, *check_dtypes=False*)

Return the difference between two objects within a tolerance.

This method should be implemented by subclasses to define how differences between objects are calculated.

The tolerance parameters rtol and atol, equal_nan, and check_dtypes are provided to be used by the numpy and pandas equality functions. These parameters allow you to customize the behavior of the equality comparison, such as setting the relative and absolute tolerance for numeric comparisons, considering NaN values as equal, and checking for the data type of the objects being compared.

Parameters

- **other** (*object*) – The object to compare to.
- **rtol** (*float, optional*) – The relative tolerance parameter. Default is 1e-05.
- **atol** (*float, optional*) – The absolute tolerance parameter. Default is 1e-08.
- **equal_nan** (*bool, optional*) – Whether to consider NaN values as equal. Default is True.
- **check_dtypes** (*bool, optional*) – Whether to check the data type of the objects. Default is False.

Returns

The difference between the current and the other object.

Return type

the_diff

> **See also**
>
> equals, aequals, numpy.isclose(), numpy.all(), numpy.any(), numpy.equal(), numpy.allclose()

Notes

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.

values_equals(*other*)

Check if the alternatives and values are the same.

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

class skcriteria.agg._agg_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.agg._agg_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.agg.aras module

Implements the Additive Ratio Assessment (Balezentiene & Kusta, 2012).

skcriteria.agg.aras.**aras**(*matrix*, *weights*, *ideal*)

Execute the ARAS method without any validation.

This function assumes that the ideal alternative has already been extracted from the decision matrix and provided separately.

Parameters

- **matrix** (*ndarray of shape (n_alternatives, n_criteria)*) – The decision matrix excluding the ideal alternative.
- **weights** (*ndarray of shape (n_criteria,)*) – The weight of each criterion.
- **ideal** (*ndarray of shape (n_criteria,)*) – The ideal alternative, provided separately.

class skcriteria.agg.aras.**ARAS**

Bases: *SKCDecisionMakerABC*

Additive Ratio Assessment (ARAS) method.

ARAS is a multi-criteria decision-making method that ranks alternatives based on the ratio of each alternative's weighted score to that of an ideal alternative.

Each alternative's score is computed by summing its weighted values across all criteria. The utility of an alternative is then defined as the ratio of its score to the score of the ideal alternative. A higher utility value indicates a better alternative.

The ideal alternative is provided through the *ideal* parameter in the *evaluate* method, not at instantiation time.

Warning

UserWarning
If the provided ideal alternative is not greater than or equal to the maximum value for each criterion, some utility scores may exceed 1.

References

[Zavadskas & Turskis, 2010]

evaluate(*dm*, *, *ideal=None*)

Evaluate the alternatives in the given decision matrix using ARAS.

If no ideal is provided when calling *evaluate*, the method automatically assumes the ideal as the maximum value in each criterion column of the decision matrix and emits a warning.

Parameters

- **dm** (skcriteria.data.DecisionMatrix) – Decision matrix on which the ranking will be calculated.
- **ideal** (*ndarray of shape (n_criteria,), optional*) – The ideal alternative, if the ideal is None, the method will automatically calculate as the maximum value in each criterion.

Raises

- **ValueError** – If any objective in the decision matrix is set to minimization.
- **ValueError** – If the number of criteria in the ideal alternative does not match the number in the decision matrix.

Warning

UserWarning
If no ideal alternative is provided, the method will use the column-wise maxima of the decision matrix as the default ideal.

Returns
Ranking.

Return type
skcriteria.data.RankResult

skcriteria.agg.cocoso module

Implementation of a Combined Compromise Solution (CoCoSo) method.

skcriteria.agg.cocoso.**cocoso**(*matrix*, *weights*, *lambda_value*)

Execute CoCoSo without any validation.

class skcriteria.agg.cocoso.**CoCoSo**(*lambda_value=0.5*)

Bases: *SKCDecisionMakerABC*

Combined Compromise Solution (CoCoSo) method.

The CoCoSo method combines the Weighted Sum Model (WSM) and Weighted Product Model (WPM) approaches to provide a comprehensive ranking solution for multi-criteria decision-making problems. It uses three different aggregation strategies to balance the advantages of both WSM and WPM methods.

The method calculates three compromise scores:

- **k_a**: Arithmetic mean of normalized WSM and WPM scores
- **k_b**: Relative scores compared to the best alternatives
- **k_c**: Balanced compromise using the lambda parameter

The final score combines these three measures using both geometric and arithmetic means to provide a robust ranking.

Parameters

lambda_value (*float, optional (default=0.5)*) – Aggregation parameter in [0, 1] that balances WSM and WPM. When lambda_value = 0, the method relies more on WPM; When lambda_value = 1, the method relies more on WSM; When lambda_value = 0.5, both methods have equal influence.

References

[Yazdani et al., 2019]

property lambda_value

Balance parameter.

skcriteria.agg.codas module

Combinative Distance-Based Assessment - CODAS.

The CODAS method evaluates alternatives using two distance metrics. It first calculates both Euclidean and Taxicab distances from a negative-ideal solution, which represents the worst performance across all criteria.

The method constructs a relative assessment matrix based on these distances, where the Euclidean distance serves as the primary measure, and the Taxicab distance acts as a tiebreaker when alternatives are very similar according to the first. The final ranking is determined by summing the values in the relative assessment matrix for each alternative, with higher scores indicating better performance.

skcriteria.agg.codas.**codas**(*matrix*, *weights*, *tau*)

Execute CODAS without any validation and assuming tau value.

class skcriteria.agg.codas.**CODAS**(*tau=0.02*)

Bases: *SKCDecisionMakerABC*

Rank alternatives using CODAS method.

COmbinative Distance-based ASsessment (CODAS) is an MCDM method that ranks alternatives by comparing how far they are from the worst possible solution (anti-ideal), simultaneously using Euclidean distance as the primary measure and Taxicab (Manhattan) distance as a tiebreaker when alternatives are very similar according to the first.

Parameters

tau (*float, optional (default=0.02)*) – tau is the threshold parameter that can be set by the decision-maker. Used to construct the relative assessment matrix.

Raises

- **ValueError:** – If the objectives contain a minimize objective. If the decision matrix is not normalized.
- **UserWarning:** – If tau is not set between 0.01 and 0.05.

References

[GHORABAEE et al., 2016]

property tau

Which tau value will be used.

skcriteria.agg.copras module

COPRAS (Complex Proportional Assessment) method.

skcriteria.agg.copras.**sum_indexes**(*matrix: ndarray*, *objectives: ndarray*)

Determine the sums of the minimizing and maximizing indexes.

Each column represents a criterion. This function separates those to be maximized from those to be minimized, and sums the values accordingly for each alternative.

skcriteria.agg.copras.**determine_significances**(*s_max*, *s_min: ndarray*)

Determine the significances of the compared alternatives.

This reflects the combined advantages and disadvantages of each alternative, using the COPRAS method formulas.

skcriteria.agg.copras.**copras**(*matrix*, *weights*, *objectives*)

Execute the COPRAS method without any validation.

Steps:

1. Compute the weighted normalized decision-making matrix.
2. Calculate sums describing the alternatives.
3. Determine significances of alternatives.
4. Calculate the utility degree of each alternative.
5. Rank the alternatives based on utility.

class skcriteria.agg.copras.**COPRAS**

Bases: *SKCDecisionMakerABC*

The COPRAS method.

The COmplex PRoportional ASsessment (COPRAS) method, introduced by Zavadskas and Kaklauskas, is used to evaluate the superiority of one alternative over another. It supports comparison of alternatives based on maximizing and minimizing index values.

Raises

ValueError – If any matrix value is < 0, if there are no criteria to minimize or if an alternative has all 0s for values in all minimizing criteria.

References

[Zavadskas et al., 1994]

skcriteria.agg.edas module

Evaluation based on Distance from Average Solution - EDAS.

The EDAS method evaluates alternatives by comparing them to an average solution benchmark. It calculates two key metrics: Positive Distance from Average (PDA) for performance exceeding the average, and Negative Distance from Average (NDA) for performance below average. These measures capture how each alternative deviates from the mean performance across all criteria.

The final appraisal combines these deviations through a weighted, normalized scoring process. After computing weighted sums of PDA and NDA for each alternative, the method normalizes these values and averages them to produce a comprehensive evaluation score.

skcriteria.agg.edas.**edas**(*matrix*, *weights*, *objectives*)

Execute EDAS without any validation.

class skcriteria.agg.edas.**EDAS**

Bases: *SKCDecisionMakerABC*

Rank alternatives using EDAS method.

The Evaluation based on Distance from Average Solution (EDAS) method ranks alternatives by comparing their performance to the average solution across all criteria. For each alternative, it calculates Positive (PDA) and Negative (NDA) distances from average values, which are then weighted, normalized, and combined into a final appraisal score.

References

[KeshavarzGhorabaee et al., 2015]

skcriteria.agg.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.agg.electre.**concordance**(*matrix*, *objectives*, *weights*)

Calculate the concordance matrix.

skcriteria.agg.electre.**discordance**(*matrix*, *objectives*)

Calculate the discordance matrix.

skcriteria.agg.electre.**electre1**(*matrix*, *objectives*, *weights*, *p=0.65*, *q=0.35*)

Execute ELECTRE1 without any validation.

class skcriteria.agg.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.agg.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.

Deprecated since version 0.9: The ELECTRE II implementation now handle this internally. The method will be removed in v1.0

Notes

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

skcriteria.agg.electre.**electre2_gomez2004tomada**(*matrix*, *objectives*, *weights*, *p0=0.65*, *p1=0.5*, *p2=0.35*, *q0=0.65*, *q1=0.35*)

Execute ELECTRE2 from "Tomada de decisões em cenários complexos" [Gomes et al., 2004] without any validation.

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

Execute ELECTRE2 from "Tomada de decisões em cenários complexos" [Gomes et al., 2004] without any validation.

Deprecated since version 0.9: Use `electre2_gomez2004tomada` instead.

class skcriteria.agg.electre.**ELECTRE2**(*, *p0=0.65*, *p1=0.5*, *p2=0.35*, *q0=0.65*, *q1=0.35*)

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.

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 >= p0 >= p1 >= p2 >= 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 >= p0 >= p1 >= p2 >= 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 >= p0 >= p1 >= p2 >= 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 >= q0 >= q1 >= 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 >= q0 >= q1 >= 0".

Notes

There are many 'Electre II' methods in the literature with different amount of parameters [Rogers et al., 2000] and different ways of defining discordance [Liu & Ma, 2021].

This implementation is based on the one presented in the books "Tomada de decisões em cenários complexos" [Gomes et al., 2004] and "ELECTRE and Decision Support"[Rogers et al., 2000].

In this version 5 parameters are chosen, concordance thresholds $1 >= p_0 >= p_1 >= p_2 >= 0$ and discordance thresholds $1 >= q_0 >= q_1 >= 0$, after calculating Concordance and Discordance Matrices (C and D recpectively) the two type of outranking relations between alternatives are defined as:

If $C(a, b) > C(b, a)$ then

- Alternative a strongly outranks alternative b if ($C(a, b) > p_0$ and $D(a, b) < q_0$) or ($C(a, b) > p_1$ and $D(a, b) < q_1$)
- Alternative a weakly outranks b if $C(a, b) > p_2$ and $D(a, b) < q_0$

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.agg.ervd module

Implementation of ERVD method.

skcriteria.agg.ervd.**ervd**(*matrix*, *objectives*, *weights*, *reference_points*, *alpha*, *lambd*, *metric*, *w_metric*, ***kwargs*)

Execute ERVD without any validation.

class skcriteria.agg.ervd.**ERVD**(***, *lambda_value=2.25*, *alpha_value=0.88*, *metric=functools.partial(<function minkowski>, p=1)*, *w_metric=True*)

Bases: *SKCDecisionMakerABC*

Election based on Relative Value Distances (ERVD) decision-making method.

This method integrates an s-shape value function, departing from the traditional expected utility function, to more accurately capture risk-averse and risk-seeking behaviors. ERVD builds upon the foundational principles of the TOPSIS method, extending its capabilities by incorporating concepts from prospect theory to refine the assessment of alternatives based on their relative distances from ideal and anti-ideal solutions.

Parameters

- **lambda_value** (*float, default=2.25*) – Represents the attenuation factor of the losse.
- **alpha_value** (*float, default=0.88*) – Diminishing sensitivity parameters.
- **metric** (*str or callable, default='minkowski'*) – The distance metric to be used for calculating distances between alternatives and ideal/anti-ideal points. It can be a string representing a metric name from *scipy.spatial.distance* or a custom callable function that computes distances.
- **w_metric** (*bool, default=True*) – Whether to use weights in the distance metric calculation. If True, the weights will be applied to the alternatives when calculating distances. If False, the distances will be calculated without weights.

References

[Shyur et al., 2015]

property alpha_value

Diminishing sensitivity parameter.

property lambda_value

Attenuation factor of the losses.

property metric

Which distance metric will be used.

property w_metric

Whether to use weights in the metric.

evaluate(*dm*, *, *reference_points=None*)

Validate the dm and calculate and evaluate the alternatives.

Parameters

- **dm** (skcriteria.data.DecisionMatrix) – Decision matrix on which the ranking will be calculated.
- **reference_points** (*array-like, optional*) – Reference points for each criterion.

Returns

Ranking.

Return type

skcriteria.data.RankResult

skcriteria.agg.mabac module

Implementation of Multi-Attributive Border Approximation Area Comparison (MABAC) method.

skcriteria.agg.mabac.**mabac**(*matrix*, *weights*)

Execute MABAC without any validation.

class skcriteria.agg.mabac.**MABAC**

Bases: *SKCDecisionMakerABC*

Multi-Attributive Border Approximation Area Comparison (MABAC) method.

MABAC is a multi-criteria decision-making method that determines the distance of each alternative from the border approximation area. The method is based on the concept of border approximation area (BAA), which is calculated as the geometric mean of the weighted normalized decision matrix.

The method consists of the following steps:

```
1. Normalization of the decision matrix
2. Calculation of the weighted normalized decision matrix
3. Determination of the border approximation area (BAA)
4. Calculation of the distance from BAA
5. Calculation of the final score
```

References

[Pamucar & Cirovic, 2015]

skcriteria.agg.moora module

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

skcriteria.agg.moora.**ratio**(*matrix*, *objectives*, *weights*)

Execute ratio MOORA without any validation.

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

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

References

[Brauers & Zavadskas, 2006]

skcriteria.agg.moora.**refpoint**(*matrix*, *objectives*, *weights*)

Execute reference point MOORA without any validation.

class skcriteria.agg.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:

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

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

[Brauers & Zavadskas, 2012]

skcriteria.agg.moora.**fmf**(*matrix*, *objectives*, *weights*)

Execute Full Multiplicative Form without any validation.

class skcriteria.agg.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; $n - i$ = 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 <= 0.

References

[Brauers & Zavadskas, 2012]

skcriteria.agg.moora.**multimoora**(*matrix*, *objectives*, *weights*)

Execute weighted product model without any validation.

class skcriteria.agg.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 <= 0.

References

[Brauers & Zavadskas, 2012]

skcriteria.agg.ocra module

Implementation of OCRA (Operational Competitiveness Rating) method for general MCDM purposes.

skcriteria.agg.ocra.**ocra_performance**(*matrix*, *objectives*, *weights*)

Compute the overall performance of each alternative.

class skcriteria.agg.ocra.**OCRA**

Bases: *SKCDecisionMakerABC*

OCRA (Operational Competitiveness Rating) method.

OCRA was initially intended (Parkan, 1994) to maximize the efficiency of a Production Unit (PU), seen as a set of activities that consume resources (inputs) and generate rewards (outputs), thus leading to a higher operational competitiveness. OCRA is thought of as an improvement of Data Envelopment Analysis (DEA): more efficient, robust, and properly sensitive to changes in inputs or outputs.

In a general-purpose sense, PUs are the alternatives to be compared, and a quantity and value of each type of input/output is instead given as a criteria value and corresponding weight. Inputs are non-beneficial criteria that should be minimized, and Outputs are beneficial criteria that should be maximized; thus, the entire Decision Matrix is used.

The performance of beneficial and non-beneficial criteria is computed separately and aggregated for each alternative, and then I/O criteria are summed to yield a final performance ranking. The Min value of each criteria is used in both cases (rather that Max - Min), following the implementation from Işık, A. T. (2016) and Madić, M. (2015). This means values are not scaled as 0-1 (possible future extension); however, they are floored to the Min value twice (first separately, then overall), such that the worst performance is always zero.

$$I_i = \sum_{j=1}^{g} w_j \frac{\max x_{ij} - x_{ij}}{\min x_{ij}} \; O_i = \sum_{j=g+1}^{n} w_j \frac{x_{ij} - \min x_{ij}}{\min x_{ij}} \; for \; i = 1, 2, 3, ..., m$$

with: $j = 1, 2, ..., g$ for the objectives to be minimized, $j = g+1, g+2, ..., n$ for the objectives to be maximized. w_j is the relative importance (weight) of each criteria.

References

[Parkan, 1994] [Isk & Adal, 2016] [Madic et al., 2015]

skcriteria.agg.probid module

Implementation of PROBID and SimplifiedPROBID.

PROBID (Preference Ranking On the Basis of Ideal-Average Distance) and SimplifiedPROBID (simple variation of PROBID).

class skcriteria.agg.probid.**BasePROBID**(*, *metric='euclidean'*)

Bases: *SKCDecisionMakerABC*

Base abstract class for PROBID variants.

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`, `sqeuclidean`, `wminkowski`, `yule`.

Warning

UserWarning:
If some objective is to minimize.

References

[Wang et al., 2021]

property metric

Which distance metric will be used.

skcriteria.agg.probid.**probid**(*matrix*, *objectives*, *weights*, *metric='euclidean'*, ***kwargs*)

Execute PROBID without any validation.

class skcriteria.agg.probid.**PROBID**(*, *metric='euclidean'*)

Bases: *BasePROBID*

Executes the PROBID method.

The PROBID method considers a spectrum of ideal solutions and the average solution to determine the performance score of each optimal solution.

skcriteria.agg.probid.**simplifiedprobid**(*matrix*, *objectives*, *weights*, *metric='euclidean'*, ***kwargs*)

Execute SimplifiedPROBID without any validation.

class skcriteria.agg.probid.**SimplifiedPROBID**(*, *metric='euclidean'*)

Bases: *BasePROBID*

Executes the SimplifiedPROBID method.

The SimplifiedPROBID method simplifies PROBID method by using only the top and bottom quartiles of ideal solutions.

skcriteria.agg.ram module

RAM (Root Assessment Method) decision-making method.

skcriteria.agg.ram.**ram**(*matrix*, *objectives*, *weights*)

Execute RAM without any validation.

class skcriteria.agg.ram.**RAM**

Bases: *SKCDecisionMakerABC*

Root Assessment Method (RAM).

RAM calculates the utility value of each alternative by aggregating their scores over the criteria, treating beneficial and non-beneficial criteria differently. The method uses an aggregation function that combines compensatory and partially compensatory characteristics.

The aggregation function used is:

$$RI_i = \sqrt[2+S_{-i}]{2 + S_{+i}}$$

where S_{+i} is the weighted sum of beneficial criteria for alternative i, S_{-i} is the weighted sum of non-beneficial criteria for alternative i. RI_i is the score of alternative i based on the RAM method. The ranking is done based on the score, where higher scores indicate better alternatives.

To use this method, the decision matrix should be normalized using the following formula:

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

References

[Sotoudeh-Anvari, 2023]

skcriteria.agg.rim module

RIM reference ideal method.

class skcriteria.agg.rim.**RIM**

Bases: *SKCDecisionMakerABC*

Reference Ideal Method (RIM) for multi-criteria decision analysis.

RIM ranks alternatives based on their similarity to a user-defined *reference ideal region* instead of relying on classical ideal and anti-ideal points. This method considers intervals as ideals, allowing more flexible and realistic preference modeling.

The method normalizes the decision matrix values with respect to the ideal intervals and the valid ranges of each criterion. Alternatives closer to the ideal intervals receive higher scores.

Parameters

- **ref_ideals** (*list of tuple*) – Specifies the ideal reference intervals for each criterion. Each tuple should be of the form (ideal_min, ideal_max). If not provided, the default ideal value for the criteria depends on the desired objectives; if it is to be maximized, the highest value within the matrix of that criterion will be set as the ideal value, and for the criteria to be minimized, the minimum value will be used (which generates intervals of length zero).
- **ranges** (*list of tuple*) – List of tuples specifying the min and max bounds of each criterion Each tuple should be of the form (range_min, range_max). If not provided, they are calculated from the maximum and minimum values of the decision matrix per criterion.

References

[Cables et al., 2016]

evaluate(*dm*, ***, *ref_ideals=None*, *ranges=None*)

Validate the decision matrix and calculate a ranking.

If *ref_ideals* or *ranges* are not provided, default values will be automatically inferred:

- If *ref_ideals* is None, an interval of length zero is created using the column-wise maximum (for MAX objectives) or minimum (for MIN objectives) from the decision matrix.

- If *ranges* is None, the valid range for each criterion is set to the minimum and maximum values of the corresponding column.

Parameters

- **dm** (DecisionMatrix) – Decision matrix to evaluate.
- **ref_ideals** (*array-like of tuple, optional*) – Reference ideal intervals (per criterion), where each tuple defines (ideal_min, ideal_max). If None, a degenerate interval is used based on the objectives.
- **ranges** (*array-like of tuple, optional*) – Ranges (min, max) for each criterion. If None, calculated from column-wise min and max values.

Warning

UserWarning
If *ref_ideals* or *ranges* are not provided, default values are inferred from the decision matrix.

Returns
Ranking.

Return type
skcriteria.data.RankResult

skcriteria.agg.similarity module (DEPRECATED)

Deprecated since version 0.9: The skcriteria.agg.similarity module is deprecated and will be removed in a future version. Please use *skcriteria.agg.topsis* instead. The module 'skcriteria.agg.similarity' is deprecated since v0.9 and will be removed in v1.0. Please use 'skcriteria.agg.topsis' instead.

class skcriteria.agg.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, sqeuclidean, 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.agg.similarity.**topsis**(*matrix*, *objectives*, *weights*, *metric='euclidean'*, ***kwargs*)

Execute TOPSIS without any validation.

skcriteria.agg.simple module

Some simple and compensatory methods.

skcriteria.agg.simple.**wsm**(*matrix*, *weights*)

Execute weighted sum model without any validation.

class skcriteria.agg.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:

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

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.agg.simple.**wpm**(*matrix*, *weights*)

Execute weighted product model without any validation.

class skcriteria.agg.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:

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

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

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

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

Raises

ValueError: – If some objective is for minimization or some value in the matrix is <= 0.

References

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

skcriteria.agg.simus module

SIMUS (Sequential Interactive Model for Urban Systems) Method.

skcriteria.agg.simus.**simus**(*matrix*, *objectives*, *b=None*, *rank_by=1*, *solver='pulp'*)

Execute SIMUS without any validation.

class skcriteria.agg.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 constrains system that the projects are subject to. In each case, if there is a feasible solution that is optimum, it is recorded in a matrix of efficient results. Then, from this matrix two rankings allow the decision maker to compare results obtained by different procedures. The first ranking is obtained through a linear weighting of each column by a factor - equivalent of establishing a weight - and that measures the participation of the corresponding project. In the second ranking, the method uses dominance and subordinate relationships between projects, concepts from the French school of MCDM.

Parameters

- **rank_by** (*1 or 2 (default=1)*) – Witch 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 <=; and if the column minimizes the constraint is >=. 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 <= or >= 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

skcriteria.agg.spotis module

SPOTIS method.

skcriteria.agg.spotis.**spotis**(*matrix*, *weights*, *bounds*, *isp*)

Execute SPOTIS method.

class skcriteria.agg.spotis.**SPOTIS**

Bases: *SKCDecisionMakerABC*

The Stable Preference Ordering Towards Ideal Solution (SPOTIS) method.

The SPOTIS method is a multi-criteria decision analysis method that is exempt of rank reversal. The method is rank reversal free because the preference ordering established from the score matrix of the MCDM problem does not require relative comparisons between the alternatives, but only comparisons with respect to the ideal solution (ISP) chosen by the MCDM designer.

References

[Dezert et al., 2020]

evaluate(*dm*, ***, *bounds=None*, *isp=None*)

Validate the decision matrix and calculate a ranking.

Parameters

- **dm** (skcriteria.data.DecisionMatrix) – Decision matrix on which the ranking will be calculated.
- **bounds** (*array-like, optional*) – The bounds of the problem. If not provided, they will be calculated from the matrix.
- **isp** (*array-like, optional*) – The ideal solution point (ISP), if not provided, it will be calculated from the bounds.

Raises

ValueError: –

- If bounds are provided and the matrix has values out of the bounds. - If ISP is provided and the ISP has values out of the bounds (either given or calculated from the matrix). - If bounds or ISP have an invalid shape.

Returns

Ranking.

Return type

skcriteria.data.RankResult

skcriteria.agg.topsis module

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

skcriteria.agg.topsis.**topsis**(*matrix*, *objectives*, *weights*, *metric='euclidean'*, ***kwargs*)

Execute TOPSIS without any validation.

class skcriteria.agg.topsis.**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, sqeuclidean, 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.agg.vikor module

VIKOR method.

class skcriteria.agg.vikor.**VIKOR**(*, *v=0.5*, *use_compromise_set=True*)

Bases: *SKCDecisionMakerABC*

The VIKOR Method for Multi-Criteria Decision Making.

VIKOR (VIseKriterijumska Optimizacija I Kompromisno Resenje) introduces the concept of a compromise solution, which is a feasible solution that is closest to the ideal, and represents a balance between the majority rule (group utility) and the individual regret of the opponent.

The method evaluates alternatives by converting an n-criteria decision problem into a bi-criteria one using the Manhattan distance (S_k, or group utility) and the Chebyshev distance (R_k, or individual regret). These are then combined into a single aggregated score (Q_k) using a weight factor v that reflects the decision-making strategy: emphasis on group utility (high v) or individual regret (low v).

VIKOR allows the identification of a single compromise solution if the following two conditions are met:

- **Acceptable advantage:**
  The best-ranked alternative is sufficiently better than the second.
- **Acceptable stability:**
  The best-ranked alternative must also be the best in at least one of the original distance metrics.

Otherwise, it identifies a set of compromise solutions.

Parameters

- **v** (*float, optional, default=0.5*) – The strategy weight that reflects the decision-making tendency. $v = 0$ gives full weight to the Chebyshev distance (individual regret), $v = 1$ gives full weight to the Manhattan distance (group utility), and $v = 0.5$ balances both. Must satisfy $0 <= v <= 1$.
- **use_compromise_set** (*bool, optional, default=True*) – If True, all alternatives within the identified compromise set are ranked equally at the top position (rank 1). If False, only the best Q_k remains at the top rank, and it is up to the user to examine the compromise set afterwards.

> **Warning**

UserWarning:
Division by zero may occur during scaling if any criterion has identical values across all alternatives, or there is identical group utility or individual regret across all alternatives.

References

[Opricovic & Tzeng, 2004]

property v

The strategy weight for VIKOR.

property use_compromise_set

Whether to use the compromise set in ranking.

skcriteria.agg.waspas module

WASPAS method.

skcriteria.agg.waspas.**waspas**(*matrix*, *weights*, *lambda_value*)

Execute WASPAS without any validation.

class skcriteria.agg.waspas.**WASPAS**(*, *lambda_value=0.5*)

Bases: *SKCDecisionMakerABC*

The Weighted Aggregated Sum Product ASsessment method.

WASPAS is a multicriteria decision analysis method that combines the Weighted Sum Model (WSM) and the Weighted Product Model (WPM) using an aggregation parameter $\lambda \in [0, 1]$.

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. Let w_j denote the weight of criterion C_j, and a_{ij} be the performance value of alternative A_i with respect to criterion C_j.

The WASPAS score of alternative A_i is defined as:

$$A_i^{WASPAS} = \lambda \cdot \sum_{j=1}^{n} w_j a_{ij} + (1 - \lambda) \cdot \prod_{j=1}^{n} a_{ij}^{w_j}$$

By default, $\lambda = 0.5$.

Raises

ValueError: – If some objective is for minimization, or some value in the matrix is <= 0, or if the parameter *lambda_value* is not in the range [0, 1].

References

[Zavadskas et al., 2012]

property lambda_value

Aggregation parameter in [0, 1] that balances WSM and WPM.

5.3.4 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], where 1 corresponds to the ideal value.

The result score x_{aj} expresses the degree to which the alternative a 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], where 1 corresponds to the ideal value.

The result score x_{aj} expresses the degree to which the alternative a 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.**SKCByCriteriaFilterABC**(*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: *SKCByCriteriaFilterABC*

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.**SKCArithmeticFilterABC**(*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"],
... )
```

(continues on next page)

(continued from previous page)

```
>>> 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: *SKCArithmeticFilterABC*

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.**SKCImputerABC**

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*, *keep_empty_criteria=False*)

Bases: *SKCImputerABC*

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.
- **keep_empty_criteria** (*bool, default=False*) – If True, criteria that consist exclusively of missing values when *fit* is called are returned in results when *transform* is called. The imputed value is always *0* except when *strategy="constant"* in which case *fill_value* will be used instead.

  Added in version 0.8.5.

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".

property keep_empty_criteria

If True, criteria that consist exclusively of missing values when *fit* is called are returned in results when *transform* is called.

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*, *keep_empty_criteria=False*, *fill_value=None*)

Bases: *SKCImputerABC*

Multivariate imputer that estimates each criteria from all the others.

A strategy for imputing missing values by modeling each criteria with missing values as a function of other criteria 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(abs(X_t - X_{t-1}))/max(abs(X[known_vals])) < 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.
- **keep_empty_criteria** (*bool, default=False*) – If True, criteria that consist exclusively of missing values when *fit* is called are returned in results when *transform* is called. The imputed value is always *0* except when *strategy="constant"* in which case *fill_value* will be used instead.

  Added in version 0.8.5.
- **fill_value** (*str or numerical value, default=None*) – When strategy="constant", fill_value is used to replace all occurrences of missing_values. For string or object data types, fill_value must be a string. If None, fill_value will be 0 when imputing numerical data and "missing_value" for strings or object data types.

  Added in version 0.8.5.

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.

property keep_empty_criteria

If True, criteria that consist exclusively of missing values when *fit* is called are returned in results when *transform* is called.

property fill_value

Used to replace all occurrences of missing_values When strategy="constant".

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

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.
- **keep_empty_criteria** (*bool, default=False*) – If True, criteria that consist exclusively of missing values when *fit* is called are returned in results when *transform* is called. The imputed value is always *0* except when *strategy="constant"* in which case *fill_value* will be used instead.

  Added in version 0.8.5.

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.

property keep_empty_criteria

If True, criteria that consist exclusively of missing values when *fit* is called are returned in results when *transform* is called.

skcriteria.preprocessing.increment module

Functionalities to add an 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.

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

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 >= 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.

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

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 thar 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 thar are converted to `numpy.float64`.

class skcriteria.preprocessing.invert_objectives.**MinMaxInverter**(*constant_criteria_kws=None*)

Bases: *SKCObjectivesInverterABC*

Normalize and invert minimization criteria using min-max scaling.

This class implements a dual-purpose transformation that simultaneously normalizes all criteria to the [0,1] range and inverts minimization criteria to maximization criteria. This is particularly useful for MCDA methods that require all criteria to have the same optimization direction and comparable scales.

The transformation preserves the relative order of alternatives while ensuring mathematical consistency across different criterion types and eliminates scale differences between criteria.

For minimization criteria, values are inverted by normalizing with:

(x - max) / (min - max)

which converts the minimization problem into a maximization one.

For maximization criteria, values are normalized with:

(x - min) / (max - min)

where the original maximum value becomes 1 (best) and the original minimum value becomes 0 (worst).

After transformation, all criteria become maximization criteria with values in the [0,1] range, where higher values are always better.

Parameters

constant_criteria_kws (*dict*, *optional*) – Keyword arguments passed to the `DecisionMatrix.constant_criteria()` method to identify constant criteria. Default is `None`, which uses default detection parameters.

Notes

- This transformation is idempotent for already normalized data with the same min-max bounds.
- Constant criteria (where all alternatives have the same value) will result in NaN values after transformation due to division by zero in the normalization formula. In such cases, the constant criteria will be transformed to 0.
- The transformation maintains the preference order within each criterion: better alternatives before transformation remain better after transformation.

property constant_criteria_kws

Get the constant criteria keyword arguments.

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

class skcriteria.preprocessing.invert_objectives.**BenefitCostInverter**

Bases: *SKCObjectivesInverterABC*

Inverts using ratios based on criterion type.

The matrix transformation is given by:

For each criterion j, the normalized value is calculated as:

if j is a benefit criteria:
n_{ij} = frac{x_{ij}}{max_i x_{ij}}

if j is a cost criteria:
n_{ij} = frac{min_i x_{ij}}{x_{ij}}

Raises
ValueError: – If the decision matrix contains negative values.

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 valuer are sean >= 0.

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

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

Deprecated since version 0.9: Superseeded by private methods inside the PushNegatives class.

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 >= 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 >= 0.

If the matrix/weights has negative values this function increment the values proportionally to made all the matrix/weights >= 0

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

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.**StandarScaler**(*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.StandarScaler`.

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.

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

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..

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

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.

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

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*

Scalerbased on the total sum of values.

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

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], where 1 corresponds to the ideal value.

The result score x_{aj} expresses the degree to which the alternative a 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], where 1 corresponds to the ideal value.

The result score x_{aj} expresses the degree to which the alternative a 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 os 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]

skcriteria.preprocessing.weighters.**merec_weights**(*matrix*, *objectives*)

Execute the MEREC method without any validation.

class skcriteria.preprocessing.weighters.**MEREC**

Bases: *SKCWeighterABC*

MEREC: Method based on the Removal Effects of Criteria.

The MEREC method computes objective weights for each criterion based on its impact on the overall performance of alternatives when removed. The idea is that the more a criterion affects the total evaluation when excluded, the more important it is.

This implementation includes a simple linear normalization.

Reference

[Keshavarz-Ghorabaee et al., 2021]

skcriteria.preprocessing.weighters.**gini_weights**(*matrix*)

Calculates weights using the Gini coefficient.

Computes the weights for each criterion (column) of the input matrix by calculating the Gini coefficient of each column, then normalizing those values to sum to 1.

The columns are sorted to use the more efficient formula for the Gini coefficient:

$$G = \frac{1}{n}\left(n + 1 - 2 \cdot \frac{\sum_{i=1}^{n}\left(\sum_{j=1}^{i} x_j\right)}{\sum_{i=1}^{n} x_i}\right)$$

class skcriteria.preprocessing.weighters.**GiniWeighter**

Bases: *SKCWeighterABC*

Calculates the weights with the Gini coefficient.

The method aims at the determination of objective weights of relative importance in MCDM problems. It uses the Gini coefficient of the data of each criterion to assign the weights, giving a higher weight to a more unequal distribution. It takes the decision matrix as a parameter.

References

[Li & Chi, 2009]

skcriteria.preprocessing.weighters.**rancom_weights**(*weights*)

RANCOM (RANking COMparison) weighting method.

The RANCOM method is designed to handle expert inaccuracies in multi-criteria decision making by transforming initial weight values through ranking comparison. The method builds a Matrix of Ranking Comparison (MAC) where all weights are compared pairwise, then calculates Summed Criteria Weights (SWC) to derive final normalized weights.

The method operates under the following assumptions:

- The sum of input weights equals 1

- Lower weight values correspond to higher importance
- Ties between criteria are allowed

Algorithm Steps:

1. Convert weights to rankings (lower weight = higher rank/importance)
2. Build MAC (Matrix of Ranking Comparison): An nxn matrix where rankings are compared pairwise with values:
   - aij = 1 if rank_i < rank_j (criteria i is more important than j)
   - aij = 0.5 if rank_i = rank_j (criteria i and j have equal importance)
   - aij = 0 if rank_i > rank_j (criteria i is less important than j)
3. Calculate SWC (Summed Criteria Weights): Sum each row of the MAC matrix
4. Normalize final weights: wi = SWCi / sum(SWC)

Parameters
weights (*array-like*) – Input weights. Lower values correspond to higher importance.

Notes

- RANCOM is particularly useful when dealing with subjective weight assignments from experts where small inaccuracies in weight specification can significantly impact results.
- The method provides a systematic way to handle ranking inconsistencies.
- Unlike other weighting methods, RANCOM transforms existing weights rather than deriving weights from the decision matrix.

Examples

```
>>> from skcriteria.preprocessing import rancom_weights
>>> weights = [0.4, 0.2, 0.25, 0.05]
>>> rancom_weights(weights)
array([0.4375, 0.1875, 0.3125, 0.0625])
```

class skcriteria.preprocessing.weighters.**RANCOM**

Bases: *SKCWeighterABC*

Ranking Comparison (RANCOM) method.

The RANCOM method is designed to handle expert inaccuracies in multi-criteria decision making by transforming initial weight values through ranking comparison.

The method builds a Matrix of Ranking Comparison (MAC) where all weights are compared pairwise, then calculates Summed Criteria Weights (SWC) to derive final normalized weights.

RANCOM uses predefined weights provided through the weighting process and does not require additional configuration parameters.

Warning

UserWarning
If there are fewer than five weights. The original paper suggests that RANCOM works better with five or more criteria, though nothing prevents its use with four or fewer criteria.

References

[Wieckowski et al., 2023]

5.3.5 skcriteria.cmp package

Utilities for a-posteriori analysis of experiments.

skcriteria.cmp.ranks_rev package

Rank reversal tools.

Rank reversal is a change in the preferred order of alternatives that occurs when the selection method or available options change. It is a significant issue in decision-making, particularly in multi-criteria decision-making.

One way to test the validity of decision-making methods is to construct special test problems and then study the solutions they derive. If the solutions exhibit some logic contradictions (in the form of undesirable rank reversals of the alternatives), then one may argue that something is wrong with the method that derived them.

The module offers features for automating the execution and assessment of standard tests for rank reversal, primarily focusing on alterations in the available options.

This Deprecated backward compatibility layer around skcriteria.ranksrev.

Deprecated since version 0.9: 'skcriteria.cmp.ranks_rev' package is deprecated, use 'skcriteria.ranksrev' instead

skcriteria.cmp.ranks_rev.ranks_inv_check module

Test Criterion #1 for evaluating the effectiveness MCDA method.

According to this criterion, the best alternative identified by the method should remain unchanged when a non-optimal alternative is replaced by a worse alternative, provided that the relative importance of each decision criterion remains the same.

skcriteria.cmp.ranks_cmp module

Ranking comparison routines.

class skcriteria.cmp.ranks_cmp.**RanksComparator**(*ranks*, *extra*)

Bases: Sequence, *DiffEqualityMixin*

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.agg.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 rannks parameters.

property extra_

Additional information about the comparison.

Note

e_ is an alias for this property.

property e_

Additional information about the comparison.

Note

e_ is an alias for this property.

diff(*other*, *rtol=1e-05*, *atol=1e-08*, *equal_nan=True*, *check_dtypes=False*)

Return the difference between two objects within a tolerance.

This method should be implemented by subclasses to define how differences between objects are calculated.

The tolerance parameters rtol and atol, equal_nan, and check_dtypes are provided to be used by the numpy and pandas equality functions. These parameters allow you to customize the behavior of the equality comparison, such as setting the relative and absolute tolerance for numeric comparisons, considering NaN values as equal, and checking for the data type of the objects being compared.

Parameters

- **other** (*object*) – The object to compare to.
- **rtol** (*float, optional*) – The relative tolerance parameter. Default is 1e-05.
- **atol** (*float, optional*) – The absolute tolerance parameter. Default is 1e-08.
- **equal_nan** (*bool, optional*) – Whether to consider NaN values as equal. Default is True.
- **check_dtypes** (*bool, optional*) – Whether to check the data type of the objects. Default is False.

Returns
The difference between the current and the other object.

Return type
the_diff

See also

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

Notes

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.

`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', 'kulczynski1', 'maha-

lanobis', '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`

extra_get(*key*, *default=None*)

Retrieve a specific key from each rank, returning a dictionary of results.

This method iterates through all ranks and attempts to get the value associated with the specified key. If the key is not found in a rank, the default value is used.

Parameters

- **key** (*hashable*) – The key to look up in each rank.
- **default** (*any, optional*) – The value to return if the key is not found in a rank. Defaults to None.

Returns
A dictionary where each key is the name of a rank, and each value is the result of calling *get(key, default)* on that rank.

Return type
dict

Notes

The returned dictionary will have an entry for every rank, even if the key was not found and the default value was used.

eget(*key*, *default=None*)

Retrieve a specific key from each rank, returning a dictionary of results.

This method iterates through all ranks and attempts to get the value associated with the specified key. If the key is not found in a rank, the default value is used.

Parameters

- **key** (*hashable*) – The key to look up in each rank.
- **default** (*any, optional*) – The value to return if the key is not found in a rank. Defaults to None.

Returns
A dictionary where each key is the name of a rank, and each value is the result of calling *get(key, default)* on that rank.

Return type
dict

Notes

The returned dictionary will have an entry for every rank, even if the key was not found and the default value was used.

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*, *extra=None*)

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 objcects.

Returns
rcmp – Returns a scikit-criteria *RanksComparator* object.

Return type
RanksComparator

5.3.6 skcriteria.ranksrev package

Rank reversal tools.

Rank reversal is a change in the preferred order of alternatives that occurs when the selection method or available options change. It is a significant issue in decision-making, particularly in multi-criteria decision-making.

One way to test the validity of decision-making methods is to construct special test problems and then study the solutions they derive. If the solutions exhibit some logic contradictions (in the form of undesirable rank reversals of the alternatives), then one may argue that something is wrong with the method that derived them.

The module offers features for automating the execution and assessment of standard tests for rank reversal, primarily focusing on alterations in the available options.

skcriteria.ranksrev.rank_invariant_check module

Tools for evaluating the stability of MCDA method's best alternative.

According to this criterion, the best alternative identified by the method should remain unchanged when a non-optimal alternative is replaced by a worse alternative, provided that the relative importance of each decision criterion remains the same.

class skcriteria.ranksrev.rank_invariant_check.**RankInvariantChecker**(*dmaker*, *, *repeat=1*, *allow_missing_alternatives=False*, *last_diff_strategy='median'*, *random_state=None*)

Bases: *SKCMethodABC*

Stability evaluator of MCDA method's best alternative.

According to this criterion, the best alternative identified by the method should remain unchanged when a non-optimal alternative is replaced by a worse alternative, provided that the relative importance of each decision criterion remains the same.

To illustrate, suppose that the MCDA method has ranked a set of alternatives, and one of the alternatives, A_j, is replaced by another alternative, A'_j, which is less desirable than Ak. The MCDA method should still identify the same best alternative when the alternatives are re-ranked using the same method. Furthermore, the relative rankings of the remaining alternatives that were not changed should also remain the same.

The current implementation worsens each non-optimal alternative `repeat` times, and stores each resulting output in a collection for comparison with the reference ranking. In essence, the test is run once for each suboptimal alternative.

This class assumes that there is another suboptimal alternative A_j that is just the next worst alternative to A_k, so that $A_k \succ A_j$. Then it generates a mutation A'_k such that A'_k is worse than A_k but still better than A_j ($A_k \succ A'_k \succ A_j$). In the case that the worst alternative is reached, its degradation is limited by default with respect to the median of all limits of the previous alternatives mutations, in order not to break he distribution of each criterion.

Parameters

- **dmaker** (Decision maker - must implement the `evaluate()` method) – The MCDA method, or pipeline to evaluate.
- **repeat** (*int, default = 1*) – How many times to mutate each suboptimal alternative.

  The total number of rankings returned by this method is given by the number of alternatives in the decision matrix minus one multiplied by `repeat`.
- **allow_missing_alternatives** (*bool, default = False*) – `dmaker` can somehow return rankings with fewer alternatives than the original ones (using a pipeline that implements a filter, for example). By setting this parameter to `True`, the invariance test allows for missing alternatives in a ranking to be added with a value of the maximum value of the ranking obtained + 1.

  On the other hand, if the value is `False`, when a ranking is missing an alternative, the test will fail with a `ValueError`.

  If more than one alternative is removed, all of them are added with the same value
- **last_diff_strategy** (*str or callable (default: "median").*) – True if any mutation is allowed that does not possess all the alternatives of the original decision matrix.
- **random_state** (*int, numpy.random.default_rng or None (default: None)*) – Controls the random state to generate variations in the sub-optimal alternatives.

property dmaker

The MCDA method, or pipeline to evaluate.

property repeat

How many times to mutate each suboptimal alternative.

property allow_missing_alternatives

True if any mutation is allowed that does not possess all the alternatives of the original decision matrix.

property last_diff_strategy

Since the least preferred alternative has no lower bound (since there is nothing immediately below it), this function calculates a limit ceiling based on the bounds of all the other suboptimal alternatives.

property random_state

Controls the random state to generate variations in the sub-optimal alternatives.

evaluate(*dm*)

Executes a the invariance test.

Parameters
dm (DecisionMatrix) – The decision matrix to be evaluated.

Returns
An object containing multiple rankings of the alternatives, with information on any changes made to the original decision matrix in the *extra_* attribute. Specifically, the *extra_* attribute contains a an object in the key *rank_inv_check* that provides information on any changes made to the original decision matrix, including the the noise applied to worsen any sub-optimal alternative.

Return type
RanksComparator

skcriteria.ranksrev.rank_transitivity_check module

Transitivity Checker for MCDM Robustness Evaluation.

This module evaluates the logical consistency and stability of Multi-Criteria Decision Making (MCDM) methods through transitivity analysis. It decomposes decision problems into pairwise comparisons and reconstructs global rankings to assess method robustness.

The module validates whether rankings satisfy the transitivity property (if A B and B C, then A C) and provides mechanisms to handle violations.

Key Features

- Transitivity validation through pairwise decomposition
- Ranking recomposition with cycle-breaking strategies
- Comprehensive diagnostic reporting

class skcriteria.ranksrev.rank_transitivity_check.**RankTransitivityChecker**(*dmaker*, *, *fallback=None*, *random_state=None*, *allow_missing_alternatives=False*, *cycle_removal_strategy='random'*, *max_ranks=50*, *parallel_backend=None*, *n_jobs=None*)

Bases: *SKCMethodABC*

Robustness evaluator for Multi-Criteria Decision Making (MCDM) methods.

This class validates the logical consistency and stability of MCDM method rankings by analyzing transitivity properties through pairwise alternative comparisons. It identifies ranking inconsistencies and provides alternative ranking reconstructions when transitivity violations occur.

The evaluation process is the following:

1. **Pairwise Dominance Analysis**: Evaluates all possible pairs of alternatives using the provided MCDM method to construct a directed dominance graph representing preference relationships.
2. **Transitivity Validation** (Test Criterion 2): Detects cycles in the dominance graph that violate the transitivity property. A transitive ranking requires that if A > B and B > C, then A > C must hold.
3. **Ranking Stability Assessment** (Test Criterion 3): Compares the original ranking with reconstructed rankings to evaluate consistency when the decision problem is decomposed and recomposed.
4. **Ranking Reconstruction**: When transitivity violations exist, applies cycle-breaking strategies to generate alternative valid rankings through graph decomposition techniques.

Parameters

- **dmaker** (*object*) – Decision maker instance that must implement an `evaluate(dm)` method. This represents the MCDM method or pipeline to be evaluated for robustness.
- **fallback** (*object*) – Optional fallback decision maker for tie-breaking in pairwise comparisons. Must also implement an `evaluate(dm)` method. If not provided, lexicographical tie breaking is used.
- **random_state** (*int, numpy.random.Generator, or None, default=None*) – Controls randomization in cycle-breaking strategies and alternative ranking generation. Ensures reproducible results when set to a specific integer.
- **allow_missing_alternatives** (*bool, default=False*) – Whether to allow rankings that don't include all original alternatives (using a pipeline that implements a filter, for example can remove alternatives). When False, raises ValueError if any alternative is missing from results. When True, missing alternatives are assigned the worst ranking + 1.
- **cycle_removal_strategy** (*str or callable, default="random"*) – Strategy for breaking cycles in non-transitive dominance graphs. Available built-in strategies include cycle removal heuristics. Can also accept custom callable functions for specialized approaches.
- **max_ranks** (*int, default=50*) – Maximum number of alternative rankings to generate when breaking cycles. Controls computational complexity by limiting the number of decompositions.

- **parallel_backend** (*str or None, default=None*) – Backend for parallel computation of pairwise evaluations. Options include 'threading', 'multiprocessing', or None for sequential. Improves performance for large numbers of alternatives.
- **n_jobs** (*int or None, default=None*) – Number of parallel jobs for pairwise evaluation. When None, uses all available processors. Set to 1 for sequential processing.

Raises

- **TypeError** – If `dmaker` doesn't implement the required `evaluate()` method.
- **ValueError** – If `cycle_removal_strategy` is not a valid strategy name or callable. If `allow_missing_alternatives=False` and alternatives are missing from results.

Examples

Basic usage with an MCDM method:

```
>>> from skcriteria.preprocessing import invert_objectives
>>> from skcriteria.agg import simple
>>>
>>> # Create a decision maker
>>> dm_method = simple.WeightedSum()
>>>
>>> # Initialize transitivity checker
>>> checker = RankTransitivityChecker(dm_method)
>>>
>>> # Evaluate a decision matrix
>>> result = checker.evaluate(dm=decision_matrix)
>>>
>>> # Check test results
>>> print(f"Test Criterion 2: {result.extra['test_criterion_2']}")
>>> print(f"Test Criterion 3: {result.extra['test_criterion_3']}")
```

Advanced configuration with custom parameters:

```
>>> checker = RankTransitivityChecker(
...     dmaker=dm_method,
...     random_state=42,
...     allow_missing_alternatives=True,
...     cycle_removal_strategy="random",
...     max_ranks=100,
...     parallel_backend="threading",
...     n_jobs=4
... )
```

property dmaker

The MCDA method, or pipeline to evaluate.

property fallback

The MCDA method, or pipeline to evaluate for tie breaking.

property random_state

Controls the random state to generate variations in the suboptimal alternatives.

property allow_missing_alternatives

Whether rankings are allowed that don't contain all original alternatives.

property cycle_removal_strategy

The strategy function used for breaking transitivity cycles.

property max_ranks

Maximum number of rankings to be generated.

property parallel_backend

The parallel backend used to generate all the alternatives.

property n_jobs

The number of parallel jobs used in the pairwise evaluations.

evaluate(*, *dm*)

Execute the complete transitivity test and ranking analysis.

This method performs a comprehensive transitivity analysis, including dominance graph construction, transitivity testing, and ranking recomposition. It provides multiple ranking perspectives when cycles are present and diagnostic information about the decision problem's structure.

Parameters

dm (DecisionMatrix) – The decision matrix to be evaluated, containing alternatives and criteria values for multi-criteria decision analysis.

Returns

A comprehensive result object containing:

- Multiple named rankings (original + recompositions)
- **Diagnostic information in the *extra* attribute:**
  - test_criterion_2: Transitivity consistency test result
  - test_criterion_3: Ranking stability test result
  - pairwise_dominance_graph: The constructed dominance graph
  - transitivity_break: List of transitivity violations
  - transitivity_break_rate: Normalized violation rate

Return type

RanksComparator

5.3.7 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).

The criteria and alternatives in Scikit-Criteria are original to the authors, but the numerical values used were taken from an unknown source that has since been 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**(*windows_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.
- `xR2` - 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]

5.3.8 skcriteria.pipelines package

The Module implements utilities to build a composite decision-maker.

skcriteria.pipelines.combinatorial module

The Module implements utilities to build a combinatorial pipeline.

class skcriteria.pipelines.combinatorial.**SKCCombinatorialPipeline**(*steps*)

Bases: *SKCMethodABC*

Model that encapsulates a pipeline of MCDA methods with alternatives.

This class allows you to define a sequential pipeline of data transformation and aggregation steps, where some steps may have multiple alternative implementations. The `CombinatorialPipeline` will generate all possible pipelines by combining these alternatives and evaluate them.

Parameters

steps (*list of (str, method or list of methods)*) – List of (name, transform) tuples (implementing `fit/transform`) that are chained, in the order in which they are chained. Steps can be a single method or a list of alternative methods.

```
# simple pipeline
steps = [
    ("inverter", invert_objectives.InvertObjectives()),
    ("scaler", scalers.SumScaler(target="matrix")),
    ("agg", simple.WeightedSum())
]

# pipeline with alternatives in the scaler step
steps = [
    ("inverter", invert_objectives.InvertObjectives()),
    (
        "scaler",
        [
            scalers.SumScaler(target="matrix"),
            scalers.VectorScaler(target="matrix"),
        ],
    ),
    ("agg", simple.WeightedSum()),
]
```

property steps

The raw steps provided during initialization.

property named_steps

The raw steps provided during initialization as a dict-like.

property pipelines

List of all generated pipelines.

property named_pipelines

A dict-like of all generated pipelines.

transform(*dm*)

Transform the data, without applying the final evaluator.

This method applies the transformation steps (such as inverters and scalers) of each individual pipeline to the input decision matrix. It does not execute the final aggregation step (the evaluator), behaving analogously to the *transform* method of a standard *SKCPipeline*.

Since multiple pipelines are generated, this method does not return a single transformed decision matrix, but rather a dictionary-like containing all transformed matrices, each associated with the unique name of the pipeline that generated it.

Parameters
dm (skcriteria.core.DecisionMatrix) – The decision matrix to transform.

Returns
A dictionary mapping the name of each pipeline (str) to its corresponding transformed skcriteria.core.DecisionMatrix.

Return type
dict-like

See also

SKCCombinatorialPipeline.evaluate
Evaluate all pipelines and compare their final rankings.

SKCCombinatorialPipeline.pipelines
Access the list of individual generated pipelines.

SKCPipeline.transform
Transforms the data using a single pipeline.

evaluate(*dm*)

Evaluates all generated pipelines with the given DecisionMatrix.

Parameters
dm (skcriteria.core.DecisionMatrix) – The decision matrix to evaluate.

Returns
A comparator object containing the ranks of all alternatives for each generated pipeline.

Return type
skcriteria.cmp.RanksComparator

skcriteria.pipelines.combinatorial.**mkcombinatorial**(**steps*)

Construct a CombinatorialPipeline from the given transformers and decision-maker.

This is a shorthand for the CombinatorialPipeline 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 CombinatorialPipeline object.

Return type
CombinatorialPipeline

skcriteria.pipelines.simple_pipeline module

The Module implements utilities to build a composite decision-maker.

class skcriteria.pipelines.simple_pipeline.**SKCPipeline**(*steps*)

Bases: *SKCMethodABC*

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.pipelines.simple_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

5.3.9 skcriteria.extend module

Functionalities for the user's extension of scikit-criteria.

This module introduces decorators that enable the creation of aggregation and transformation models using only functions.

It is important to note that models created with these decorators are much less flexible than those created using inheritance and lack certain properties of real objects.

exception skcriteria.extend.**NonStandardNameWarning**

Bases: UserWarning

Custom warning class to indicate that a name does not follow a specific standard.

This warning is raised when a given name does not adhere to the specified naming convention.

skcriteria.extend.**mkagg**(*maybe_func=None*, ***hparams*)

Decorator factory function for creating aggregation classes.

Parameters

- **maybe_func** (*callable, optional*) – Optional aggregation function to be wrapped into a class. If provided, the decorator is applied immediately.

  The decorated function should receive the parameters 'matrix', 'objectives', 'weights', 'dtypes', 'alternatives', 'criteria', 'hparams', or kwargs.

  Additionally, it should return an array with rankings for each alternative and an optional dictionary with calculations that you wish to store in the 'extra' attribute of the ranking."

- ****hparams** (*keyword arguments*) – Hyperparameters specific to the aggregation function.

Returns
Agg – Aggregation class decorator or Aggregatio model with added functionality.

Return type
class or decorator

Notes

This decorator is designed for creating aggregation model from aggregation functions. It provides an interface for creating aggregated decision-making models.

Examples

```
>>> @mkagg
>>> def MyAgg(**kwargs):
>>>     # Implementation of the aggregation function
```

The above example will create an aggregation class with the name 'MyAgg' based on the provided aggregation function.

```
>>> @mkagg(foo=1)
>>> def MyAgg(**kwargs):
>>>     # Implementation of the aggregation function
```

The above example will create an aggregation class with the specified hyperparameter 'foo' and the name 'MyAgg'.

skcriteria.extend.**mktransformer**(*maybe_func=None*, ***hparams*)

Decorator factory function for creating transformation classes.

Parameters

- **maybe_func** (*callable, optional*) – Optional transformation function to be wrapped into a class. If provided, the decorator is applied immediately.

  The decorated function should receive the parameters 'matrix', 'objectives', 'weights', 'dtypes', 'alternatives', 'criteria', 'hparams', or kwargs.

  In addition, it must return a dictionary whose keys are some as the received parameters (including the keys in 'kwargs'). These values replace those of the original array. If you return 'hparams,' the transformer will ignore it.

  If you want the transformer to infer the types again, return *dtypes* with value *None*.

  It is the function's responsibility to maintain compatibility.

- ****hparams** (*keyword arguments*) – Hyperparameters specific to the transformation function.

Returns

Trans – Transformation class decorator or Transformation model with added functionality.

Return type

class or decorator

Notes

This decorator is designed for creating transformation models from transformation functions. It provides an interface for creating transformed decision-making models.

Examples

```
>>> @mktrans
>>> def MyTrans(**kwargs):
>>>     # Implementation of the transformation function
>>>     pass
```

The above example will create a transformation class with the name 'MyTrans' based on the provided transformation function.

```
>>> @mktrans(foo=1)
>>> def MyTrans(**kwargs):
>>>     # Implementation of the transformation function
>>>     pass
```

The above example will create a transformation class with the specified hyperparameter 'foo' and the name 'MyTrans'.

5.3.10 skcriteria.testing module

Public testing utility functions.

This module exposes "assert" functions which facilitate the comparison in a testing environment of objects created in skcriteria.

The functionalities are extensions of those present in "pandas.testing" and "numpy.testing".

skcriteria.testing.**assert_dmatrix_equals**(*left*, *right*, ***diff_kws*)

Asserts that two DecisionMatrix objects are equal by comparing their attributes with some tolerance.

Parameters

- **left** (DecisionMatrix) – The first DecisionMatrix object to compare.
- **right** (DecisionMatrix) – The second DecisionMatrix object to compare.
- ****diff_kws** (*dict*) – Additional keyword arguments to pass to the *DecisionMatrix.diff* method.

Raises

AssertionError – If the two DecisionMatrix objects are not equal.

skcriteria.testing.**assert_result_equals**(*left*, *right*, ***diff_kws*)

Asserts that two results objects are equal by comparing their attributes with some tolerance.

Parameters

- **left** (*skcriteria.agg.ResultABC*) – The left result to compare.
- **right** (*skcriteria.agg.ResultABC*) – The right result to compare.
- ****diff_kws** (*dict*) – Optional keyword arguments to pass to the result *diff* method.

Raises

AssertionError if the two results are not equal. –

skcriteria.testing.**assert_rcmp_equals**(*left*, *right*, ***diff_kws*)

Asserts that the left and right RankComparator objects are equal by comparing their attributes with some tolerance.

Parameters

- **left** (RanksComparator) – The left object to compare.
- **right** (*Any*) – The right object to compare.
- ****diff_kws** (*keyword arguments*) – Additional keyword arguments to pass to the *diff* method.

Raises

- **AssertionError** – If the left object is not an instance of RanksComparator.
- **AssertionError** – If the right object is not an instance of RanksComparator.
- **AssertionError** – If the left and right objects have different lengths.
- **AssertionError** – If the ranks at any index of the left and right objects are not equal.

5.3.11 skcriteria.tiebreaker module

Tie breaker estrategies for eliminating ties in rankings.

exception skcriteria.tiebreaker.**TieUnresolvedWarning**

Bases: UserWarning

Warning for when ties remain unresolved after a tie-breaker is applied.

class skcriteria.tiebreaker.**FallbackTieBreaker**(*dmaker*, *untier*, *, *force=True*)

Bases: *SKCMethodABC*

Decision maker that breaks ties in rankings using a fallback decision maker.

This class takes a primary decision maker that may produce tied rankings and uses a fallback decision maker to break those ties. If the fallback decision maker also produces ties and force=True, it uses the untied_rank_ property to ensure a complete ranking without ties.

Parameters

- **dmaker** (*decision maker*) – Primary decision maker that implements the *evaluate()* method. This decision maker may produce rankings with ties.
- **untier** (*decision maker*) – Fallback decision maker used to break ties. It will be applied only to the tied alternatives from the primary decision maker.
- **force** (*bool, default True*) – If True, when the fallback decision maker also produces ties, uses the untied_rank_ property to force a complete ranking without ties. If False, allows the final ranking to have ties if the fallback fails to break them completely.

Examples

```
>>> from skcriteria import mkdm
>>> from skcriteria.agg import simple
>>>
>>> # Create a decision matrix
>>> dm = mkdm([[1, 2], [3, 4]], [1, 1], ["max", "max"])
>>>
>>> # Primary decision maker
>>> primary = simple.WeightedSum()
>>>
>>> # Fallback decision maker for tie breaking
>>> fallback = simple.WeightedProduct()
>>>
>>> # Create fallback tie breaker
>>> tie_breaker = FallbackTieBreaker(primary, fallback)
>>>
>>> # Evaluate
>>> result = tie_breaker.evaluate(dm)
```

property dmaker

Primary decision maker.

Returns
The primary decision maker instance used for initial ranking.

Return type
decision maker

property untier

Fallback decision maker for breaking ties.

Returns
The fallback decision maker instance used to break ties from the primary decision maker.

Return type
decision maker

property force

Whether to force complete untying using `untied_rank_`.

Returns
True if forced untying is enabled, False otherwise.

Return type
bool

evaluate(*dm*)

Evaluate the decision matrix using the fallback tie-breaking approach.

This method first applies the primary decision maker to get an initial ranking. If ties exist, it systematically applies the fallback decision maker to each group of tied alternatives to break the ties.

Parameters
dm (DecisionMatrix) – Decision matrix to evaluate containing alternatives and criteria.

Returns
result – Ranking result with ties broken using the fallback decision maker. If force=True and ties still remain, uses `untied_rank_` to ensure a complete ranking without any ties.

Return type
skcriteria.agg.RankResult

class skcriteria.agg.**RankResult**

5.3.12 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_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
```

get(*key*, *default=None*)

Get item from bunch.

to_dict()

Convert the Bunch object to a dictionary.

This method performs a deep copy of the _data attribute, ensuring that the original data remains unchanged.

Returns

A deep copy of the _data attribute.

Return type

dict

Example

```
>>> bunch = Bunch()
>>> bunch._data = {'key1': 'value1', 'key2': 'value2'}
>>> dict_data = bunch.to_dict()
>>> print(dict_data)
{'key1': 'value1', 'key2': 'value2'}
```

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.

exception skcriteria.utils.cmanagers.**HiddenAlreadyUsedInThisContext**

Bases: RuntimeError

Raised when a context attempts to use the 'hidden' context manager more than once within the same scope.

exception skcriteria.utils.cmanagers.**NonGlobalHidden**

Bases: RuntimeError

Exception raised when the 'hidden' decorator is used in a context that is not the global scope of a module.

This exception indicates that the 'hidden' decorator should only be applied globally, outside of any functions or methods, and an attempt to use it within a local context (e.g., inside a function or method) has been detected.

skcriteria.utils.cmanagers.**hidden**(*, *hide_this=True*, *dry=False*)

A context manager for hiding objects in the global scope.

Parameters

- **hide_this** (*bool, optional*) – Whether to hide the 'hidden' context manager itself and/or the hidden module. Defaults to True.

- **dry** (*bool, optional, default False*) – If is True, the objects are not hide. Useful for testing.

Raises

- *NonGlobalHidden* – If 'hidden' is declared inside a function, class or method.
- *HiddenAlreadyUsedInThisContext* – If the 'hidden' context manager is used more than once in the same context.

Yields

None

Notes

- This context manager is intended to be used globally (outside any functions or methods).
- It hides objects within the global scope for the duration of the context.

Implementation Details

- The context manager retrieves the current frame and ensures it is used globally.
- It captures the state of the global scope before entering the context.
- Objects introduced within the context are hidden in the global scope.
- The '__dir__' attribute of the global scope is customized to include logic to hide the objects introduced within the context.

skcriteria.utils.cycle_removal module

Utility to remove cycles from Networkx graphs.

skcriteria.utils.cycle_removal.**generate_acyclic_graphs**(*graph*, ***, *strategy='random'*, *max_graphs=10*, *seed=None*)

Generate multiple acyclic graphs by removing edges from cycles.

This function creates multiple directed acyclic graphs (DAGs) from a potentially cyclic input graph by strategically removing edges that participate in cycles. Different strategies can be used to select which edges to remove, and multiple attempts generate diverse solutions.

Parameters

- **graph** (*networkx.DiGraph*) – The input directed graph, which may contain cycles.
- **strategy** (*str or callable, default "random"*) – Edge selection strategy for cycle breaking: - "random": Select edges uniformly at random - "weighted": Select edges proportional to their cycle frequency - callable: Custom edge selection function with signature func(cycle, edge_freq, rng) -> edge_tuple
- **max_graphs** (*int, default 10*) – Maximum number of acyclic graphs to generate. The function may return fewer graphs if it cannot generate enough unique solutions.
- **seed** (*int, optional*) – Random seed for reproducible results. If None, results will vary between runs.

Returns

A list of tuples where each tuple contains: - acyclic_graph (networkx.DiGraph): A directed acyclic graph - removed_edges (set): Set of edges that were removed to break cycles

If the input graph is already acyclic, returns a single tuple with the original graph and an empty set of removed edges.

Return type
list of tuple

Raises
`ValueError` – If the strategy parameter is not recognized and not a callable function.

Notes

The algorithm works by:

1. **Cycle Detection**: Find all simple cycles in the input graph
2. **Edge Selection**: For each cycle, select an edge to remove based on the chosen strategy
3. **Graph Modification**: Create a new graph with selected edges removed
4. **Validation**: Check if the resulting graph is acyclic
5. **Iteration**: Repeat with different random choices to generate multiple solutions

The function attempts up to *2 * max_graphs* iterations to generate the requested number of acyclic graphs. This provides robustness against cases where the random selection produces duplicate solutions.

Strategy Details:

- **Random**: Each edge in each cycle has equal probability of removal
- **Weighted: Edges appearing in more cycles are more likely to be** removed

Performance Considerations:

- Time complexity depends on the number of cycles and their lengths
- Memory usage scales with the number of generated graphs
- For large graphs with many cycles, consider reducing *max_graphs*

Examples

```
>>> import networkx as nx
>>>
>>> # Create a cyclic graph
>>> G = nx.DiGraph([(1, 2), (2, 3), (3, 1), (1, 4)])
>>>
>>> # Generate acyclic graphs using random strategy
>>> results = generate_acyclic_graphs(G, strategy="random", max_graphs=5)
>>> len(results)
5
>>>
>>> # Check that results are acyclic
>>> for dag, removed in results:
```

(continues on next page)

(continued from previous page)

```
...     print(f"Acyclic: {nx.is_directed_acyclic_graph(dag)}")
...     print(f"Removed edges: {removed}")
Acyclic: True
Removed edges: {(3, 1)}
Acyclic: True
Removed edges: {(2, 3)}
...
```

```
>>> # Use weighted strategy for more systematic edge removal
>>> results_weighted = generate_acyclic_graphs(
...     G, strategy="weighted", max_graphs=3, seed=42
... )
>>>
>>> # Already acyclic graph
>>> dag = nx.DiGraph([(1, 2), (2, 3)])
>>> results_acyclic = generate_acyclic_graphs(dag)
>>> len(results_acyclic)
1
>>> results_acyclic[0][1]  # No edges removed
set()
```

See also

networkx.simple_cycles
Find all simple cycles in a directed graph

networkx.is_directed_acyclic_graph
Check if a graph is acyclic

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.**add_sphinx_deprecated_directive**(*doc*, ***, *reason*, *version*)

Add the Sphinx deprecation directive to a given doc.

Parameters

- **doc** (*str*) – The original documentation.
- **reason** (*str*) – Reason message which documents the deprecation 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".

skcriteria.utils.deprecate.**warn**(*reason*, *, *category=<class 'skcriteria.utils.deprecate.SKCriteriaDeprecationWarning'>*)

Raises a deprecation warning.

It will result in a warning being emitted immediately

Parameters

- **reason** (*str*) – Reason message which documents the deprecation in your library.
- **category** (*default='SKCriteriaDeprecationWarning'*) – Class of the 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/*>__

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.dict_cmp module

Utilities to compare two dictionaries with numpy arrays.

skcriteria.utils.dict_cmp.**dict_allclose**(*left*, *right*, *rtol=1e-05*, *atol=1e-08*, *equal_nan=False*)

Compares two dictionaries. If values of type "numpy.array" are encountered, the function utilizes "numpy.allclose" for comparison.

Parameters

- **left** (*dict*) – The left dictionary.
- **right** (*dict*) – The right dictionary.
- **rtol** (*float, optional*) – The relative tolerance parameter for *np.allclose*.
- **atol** (*float, optional*) – The absolute tolerance parameter for *np.allclose*.
- **equal_nan** (*bool, optional*) – Whether to consider NaN values as equal.

Returns

True if the dictionaries are equal, False otherwise.

Return type

bool

Notes

This function iteratively compares the values of corresponding keys in the input dictionaries *left* and *right*. It handles various data types, including NumPy arrays, and uses the *np.allclose* function for numeric array comparisons with customizable tolerance levels. The comparison is performed iteratively, and the function returns True if all values are equal based on the specified criteria. If the dictionaries have different lengths or keys, or if the types of corresponding values differ, the function returns False.

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*>__

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 sintax 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
).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 sintax 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
).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.object_diff module

Utilities to calculate the difference between two objects.

skcriteria.utils.object_diff.**MISSING = <MISSING>**

A singleton object used to represent missing values.

skcriteria.utils.object_diff.**diff**(*left*, *right*, ***members*)

Calculates the difference between two objects, *left* and *right*, and returns a *Difference* object.

Parameters

- **left** (*object*) – The first object to compare.
- **right** (*object*) – The second object to compare.
- ****members** (*dict*) – Keyword named arguments representing members to compare. The values of the members is the function to compare the members values

Returns

A *Difference* object representing the differences between the two objects.

Return type

Difference

Notes

This function compares the values of corresponding members in the *left* and *right* objects. If a member is missing in either object, it is considered a difference. If a member is present in both objects, it is compared using the corresponding comparison function specified in *members*.

Examples

```
>>> obj_a = SomeClass(a=1, b=2)
>>> obj_b = SomeClass(a=1, b=3, c=4)
>>> diff(obj_a, obj_b, a=np.equal, b=np.equal)
<Difference different_types=False members_diff=('b', 'c')>
```

class skcriteria.utils.object_diff.**DiffEqualityMixin**

Bases: ABC

Abstract base class for classes with a diff method.

This class provides methods for comparing objects with a tolerance, allowing for differences within specified limits. It is designed to be used with numpy and pandas equality functions.

Extra methods:

- **aequals**
  almost-equals, Check if the two objects are equal within a tolerance.
- **equals(other)**
  Return True if the objects are equal.
- **__eq__(other)**
  Implement equality comparison.
- **__ne__(other)**
  Implement inequality comparison.

abstract diff(*other*, *rtol=1e-05*, *atol=1e-08*, *equal_nan=True*, *check_dtypes=False*)

Return the difference between two objects within a tolerance.

This method should be implemented by subclasses to define how differences between objects are calculated.

The tolerance parameters rtol and atol, equal_nan, and check_dtypes are provided to be used by the numpy and pandas equality functions. These parameters allow you to customize the behavior of the equality comparison, such as setting the relative and absolute tolerance for numeric comparisons, considering NaN values as equal, and checking for the data type of the objects being compared.

Notes

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.

Parameters

- **other** (*object*) – The object to compare to.
- **rtol** (*float, optional*) – The relative tolerance parameter. Default is 1e-05.
- **atol** (*float, optional*) – The absolute tolerance parameter. Default is 1e-08.
- **equal_nan** (*bool, optional*) – Whether to consider NaN values as equal. Default is True.
- **check_dtypes** (*bool, optional*) – Whether to check the data type of the objects. Default is False.

Returns
The difference between the current and the other object.

Return type
the_diff

See also

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

aequals(*other*, *, *rtol=1e-05*, *atol=1e-08*, *equal_nan=True*, *check_dtypes=False*)

Check if the two objects are equal within a tolerance.

All the parameters ara passed to the *diff* method.

Parameters

- **other** (*object*) – The object to compare to.
- **rtol** (*float, optional*) – The relative tolerance parameter. Default is 1e-05.
- **atol** (*float, optional*) – The absolute tolerance parameter. Default is 1e-08.
- **equal_nan** (*bool, optional*) – Whether to consider NaN values as equal. Default is True.

- **check_dtypes** (*bool, optional*) – Whether to check the data type of the objects. Default is False.

Returns
True if the objects are equal within the specified tolerance, False otherwise.

Return type
bool

equals(*other*)

Return True if the objects are equal.

This method calls *aquals()* without tolerance.

Parameters
other (object) – Other instance to compare.

Returns
equals – Returns True if the two objects are equals.

Return type
bool:py:class:

See also

aequals, diff.

skcriteria.utils.ondemand_import module

On-demand importer of modules.

The on-demand importer is a function that returns a callable object that imports the module when the object is called.

Notes

This ondemand importer is inspired on the one from scikit-learn, but adds a more power to introspection.

skcriteria.utils.ondemand_import.**is_package**(*obj*)

Check if the object is a package.

Parameters
obj (*object*) – The object to check.

Returns
True if the object is a package, False otherwise.

Return type
bool

class skcriteria.utils.ondemand_import.**OnDemandImporter**(*package_name: str, package: ModuleType*)

Bases: object

Enhanced on-demand importer for lazy loading of package modules.

This class implements a mechanism for lazy loading of modules within a package. It postpones the import of a module until it is explicitly requested, allowing for more efficient loading of large packages. Unlike simpler implementations, this version also provides directory listing capabilities.

Parameters

- **package_name** (*str*) – The fully qualified name of the package.
- **package** (*types.ModuleType*) – The package module object.

Raises
ValueError – If the provided package object is not actually a package.

Notes

This implementation uses a frozen dataclass to ensure immutability of the importer's state.

package_name: str

package: ModuleType

property package_context

Get the package's context dictionary.

Returns
Dictionary of the package's variables and modules.

Return type
dict

property package_path

Get the package's search path.

Returns
List of directories where the package's modules can be found.

Return type
list

import_or_get_attribute(*name*)

Dynamically imports or retrieves a module as an attribute.

This function is the core of the lazy-loading mechanism. It either returns an already loaded module from cache or imports it when first requested, then adds it to the parent package namespace.

Parameters
name (*str*) – Module name to import or retrieve (without parent package prefix)

Returns
The cached or newly imported module or subpackage

Return type
module

Raises
AttributeError – If the module doesn't exist or cannot be imported

Notes

The implementation:

- First checks if the module exists in the package_context dictionary cache
- Imports the module if not found in cache
- Sets up recursive lazy-loading for any imported subpackages
- Raises AttributeError specifically for Jedi compatibility

Jedi, the autocompletion engine used in Jupyter and other scientific environments, explores namespaces by calling __getattr__ and only ignores ImportError and AttributeError exceptions during this process. This implementation ensures compatibility with Jedi's behavior.

list_available_modules()

List all available modules in the package.

This method combines the already imported modules with the modules available on disk that have not yet been imported.

Returns
Sorted list of all available module names in the package.

Return type
list

skcriteria.utils.ondemand_import.**mk_ondemand_importer_for**(*package_name*)

Create an on-demand importer for a specific package.

This function creates and returns an instance of _OnDemandImporter for the specified package. The package must already be imported and available in sys.modules.

Parameters
package_name (*str*) – The fully qualified name of the package for which to create an importer.

Returns
An instance of _OnDemandImporter configured for the specified package.

Return type
_OnDemandImporter

Examples

```
>>> # In a package's __init__.py
>>> importer = mk_ondemand_importer_for(__name__)
>>> __getattr__ = importer.import_module
>>> __dir__ = importer.list_available_modules
```

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 array_a[i] > array_b[i] if reverse is False, otherwise array_a[i] array_b[i] if 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 element is the unique name of the second is the named object.

Return type
list of tuples

5.3.13 skcriteria.madm deprecated package

Warning

This package is deprecated, and is simply an alias for the skcriteria.agg package.

Therefore

```
from skcriteria.madm.similarity import TOPSIS
from skcriteria.madm import electre
```

Is equivalent to

```
from skcriteria.agg.similarity import TOPSIS
from skcriteria.agg import electre
```

MCDA aggregation methods and internal machinery.

This Deprecated backward compatibility layer around skcriteria.agg.

Deprecated since version 0.8.5: 'skcriteria.madm' module is deprecated, use 'skcriteria.agg' instead

class skcriteria.madm.**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.

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

Bases: *DiffEqualityMixin*

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)

diff(*other*, *rtol=1e-05*, *atol=1e-08*, *equal_nan=False*, *check_dtypes=False*)

Return the difference between two objects within a tolerance.

This method should be implemented by subclasses to define how differences between objects are calculated.

The tolerance parameters rtol and atol, equal_nan, and check_dtypes are provided to be used by the numpy and pandas equality functions. These parameters allow you to customize the behavior of the equality comparison, such as setting the relative and absolute tolerance for numeric comparisons, considering NaN values as equal, and checking for the data type of the objects being compared.

Parameters

- **other** (*object*) – The object to compare to.
- **rtol** (*float, optional*) – The relative tolerance parameter. Default is 1e-05.
- **atol** (*float, optional*) – The absolute tolerance parameter. Default is 1e-08.
- **equal_nan** (*bool, optional*) – Whether to consider NaN values as equal. Default is True.
- **check_dtypes** (*bool, optional*) – Whether to check the data type of the objects. Default is False.

Returns
The difference between the current and the other object.

Return type
the_diff

See also

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

Notes

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.

values_equals(*other*)

Check if the alternatives and values are the same.

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

class skcriteria.madm.**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.**MABAC**

Bases: *SKCDecisionMakerABC*

Multi-Attributive Border Approximation Area Comparison (MABAC) method.

MABAC is a multi-criteria decision-making method that determines the distance of each alternative from the border approximation area. The method is based on the concept of border approximation area (BAA), which is calculated as the geometric mean of the weighted normalized decision matrix.

The method consists of the following steps:

```
1. Normalization of the decision matrix
2. Calculation of the weighted normalized decision matrix
3. Determination of the border approximation area (BAA)
4. Calculation of the distance from BAA
5. Calculation of the final score
```

References

[Pamucar & Cirovic, 2015]

5.3.14 skcriteria.pipeline module

Deprecated since version 0.9: The skcriteria.pipeline module is deprecated and will be removed in a future version. Please use *skcriteria.pipelines* instead. The 'skcriteria.pipeline' module is deprecated since 0. and will be removed in 1.0 Use 'skcriteria.pipelines' instead.

class skcriteria.pipeline.**SKCPipeline**(*steps*)

Bases: *SKCMethodABC*

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

5.4 Changelog

5.4.1 Version 0.9

New Features

- **New MCDA Methods:** The library has been expanded with a dozen new aggregation methods, offering a wider range of tools for decision analysis:
  - **ARAS:** Additive Ratio Assessment.
  - **COCOSO:** Combined Compromise Solution.
  - **EDAS:** Evaluation based on Distance from Average Solution.
  - **ERVD:** Evaluation based on Reference Vector-Distance.
  - **MABAC:** Multi-Attributive Border Approximation area Comparison.
  - **OCRA:** Operational Competitiveness Rating Analysis.
  - **PROBID:** Probabilistic Omsi-based method for criterion weighting.
  - **RAM:** Range of Value Method.
  - **RIM:** Reference Ideal Method.
  - **SPOTIS:** Stable Preference Ordering Towards Ideal Solution.
  - **WASPAS:** Weighted Aggregated Sum Product Assessment.
- **New Criteria Inverters:** The
  - **MinMaxInverter:** A new inverter that normalizes all criteria to the [0,1] range and inverts minimization criteria to maximization. This is particularly useful for MCDA methods that require all criteria to have the same optimization direction and comparable scales.
  - **BenefitCostInverter:** A new inverter that uses ratios based on the criterion type. For benefit criteria, values are normalized by dividing by the maximum value of the criterion. For cost criteria, the minimum value of the criterion is divided by the value of the criterion
- **New Criteria Weighers:** New methods have been added to objectively calculate criteria weights:
  - **MEREC:** Method based on the Removal Effects of Criteria.
  - **Gini:** A weighter based on the Gini coefficient.
- **Rank Reversal Evaluators:** Two new classes have been introduced to analyze ranking stability:
  - *RankInvariantChecker*: Evaluates ranking consistency when one or more alternatives are removed (Test 1 of: Wang, X., & Triantaphyllou, E. (2008). Ranking irregularities when evaluating alternatives by using some ELECTRE methods. Omega, 36(1), 45-63. https://doi.org/10.1016/j.omega.2005.12.003).
  - *RankTransitivityChecker*: Analyzes ranking transitivity by comparing results with subsets of alternatives (Test 2 and 3 of: Wang, X., & Triantaphyllou, E. (2008). Ranking irregularities when evaluating alternatives by using some ELECTRE methods. Omega, 36(1), 45-63. https://doi.org/10.1016/j.omega.2005.12.003)
- **Combinatorial Pipelines:** The `CombinatorialPipeline` class has been added, a powerful tool that allows for the creation and exhaustive evaluation of all possible combinations of transformers and aggregation methods, facilitating multiple scalers and other kinds of analyses.
- **LaTeX Export:** The `DecisionMatrix` class now features a `to_latex()` method for easily exporting the decision matrix to a LaTeX table format.

- **Dominance Graph Cycle Detection:** A `has_loops()` method has been added to the `DecisionMatrix` class, which uses the `networkx` library to detect inconsistencies (cycles) in dominance relationships.

Improvements and Behavioral Changes

- **``PushNegatives`` Fix:** A bug in the `PushNegatives` transformer has been resolved. It now correctly shifts all matrix values so that the minimum value is zero.
- **``ELECTRE2`` Fix:** An error in the calculation of the "weak kernel" in `ELECTRE2` has been corrected, ensuring the results align with the method's literature.
- **Import Performance Improvement:** An on-demand import mechanism has been implemented, reducing the library's initial loading time.
- **Python 3.12 and 3.13 Support:** Compatibility has been added for the latest Python versions.
- **Dependency Updates:** The required versions of `NumPy` (to 2.0), `NetworkX` (to 3.2), and `scikit-learn` (to 1.6) have been updated.

API Changes (Breaking Changes)

- **New ``replace()`` Method:** A `replace()` method has been introduced in `DecisionMatrix` and all method classes (`SKCMethodABC`) as the recommended way to create copies with modified parameters. Using `copy(**kwargs)` for this purpose is now deprecated.
- **Module Restructuring:**
  - The `skcriteria.agg.similarity` module has been renamed to `skcriteria.agg.topsis`.
  - The `skcriteria.pipeline` module has been restructured into a package (`skcriteria.pipelines`), and the main class is now `skcriteria.pipelines.simple_pipeline.SimplePipeline`.
- **Name Changes for Consistency:**
  - The `Electre2` class has been renamed to `ELECTRE2`.
  - The `TieBreaker` class has been renamed to `FallbackTieBreaker`.

5.4.2 Version 0.8.7

- **New** Added functionality for user extension of scikit-criteria with decorators for creating aggregation and transformation models using functions.

```
>>> from skcriteria.extend import mkagg, mktransformer
>>>
>>> @mkagg
>>> def MyAgg(**kwargs):
>>>     # Implementation of the aggregation function
>>>
>>> @mkagg(foo=1)
>>> def MyAggWithHyperparam(**kwargs):
>>>     # Implementation of the aggregation function with
>>>     # hyperparameter 'foo'
>>>
```

(continues on next page)

(continued from previous page)

```
>>> @mktransformer
>>> def MyTransformer(**kwargs):
>>>     # Implementation of the transformation function
>>>
>>> @mktransformer(bar=2)
>>> def MyTransformerWithHyperparam(**kwargs):
>>>     # Implementation of the transformation function with
>>>     # hyperparameter 'bar'
```

These decorators enable the creation of aggregation and transformation classes based on provided functions, allowing users to define decision-making models with less flexibility than traditional inheritance-based models.

For more information check the tutorial Extending Aggregation and Transformation Functions

- **New Module:** Introduced the `skcriteria.testing` module, exposing utility functions for for comparing objects created in Scikit-Criteria in a testing environment. These functions facilitate the comparison of instances of the `DecisionMatrix`, `ResultABC`, and `RanksComparator` classes.

  The assertion functions utilize pandas and numpy testing utilities for comparing matrices, series, and other attributes.

  Check the Reference for more information.

- **New** The API of the agg, pipeline, preprocessing, and extend modules has been cleaned up to prevent autocompletion with imports from other modules. The imported modules are still present, but they are excluded when attempting to autocomplete. This functionality is achieved thanks to the context manager `skcriteria.utils.cmanagers.hidden()`.

- **New** All methods (agg and transformers) has a new `get_method_name` instance method.

- **Drop** Drop support for Python 3.8

5.4.3 Version 0.8.6

- **New** Rank reversal 1 implemenethed in the `RankInvariantChecker` class

```
>>> import skcriteria as skc
>>> from skcriteria.cmp import RankInvariantChecker
>>> from skcriteria.agg.similarity import TOPSIS

>>> dm = skc.datasets.load_van2021evaluation()
>>> rrt1 = RankInvariantChecker(TOPSIS())
>>> rrt1.evaluate(dm)
<RanksComparator [ranks=['Original', 'M.ETH', 'M.LTC', 'M.XLM', 'M.BNB', 'M.ADA',
↪'M.LINK', 'M.XRP', 'M.DOGE']]>
```

- **New** The module `skcriteria.madm` was deprecated in favor of `skcriteria.agg`
- Add support for Python 3.11.
- Removed Python 3.7. Google collab now work with 3.8.
- Updated Scikit-Learn to 1.3.x.
- Now all cached methods and properties are stored inside the instance. Previously this was stored inside the class generating a memoryleak.

5.4.4 Version 0.8.3

- Fixed a bug detected on the EntropyWeighted, Now works as the literature specifies

5.4.5 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`.

5.4.6 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 folow the Python recomendation
- `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.scalers`.
- 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`

5.4.7 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.

5.4.8 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 througth a common framework (`skcriteria.utils.accabc` module).
- New deprecation mechanism through the
- `skcriteria.utils.decorators.deprecated` decorator.

5.4.9 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

5.4.10 Version 0.2

First OO stable version.

5.4.11 Version 0.1

Only functions.

5.5 Bibliography

5.6 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 multimoora: a method for multi-objective optimization. *Informatica*, *23*(1), 1–25.

[Bridgman, 1922] Bridgman, P. W. (1922). *Dimensional analysis*. Yale university press.

[Cables et al., 2016] Cables, E., Lamata, M. T., & Verdegay, J. L. (2016). Rim-reference ideal method in multicriteria decision making. *Information Sciences*, *337*, 1–10.

[Cabral et al., 2016] Cabral, J. B., Luczywo, N. A., & Zanazzi, J. L. (2016). Scikit-criteria: colección de métodos de análisis multi-criterio integrado al stack científico de Python. *XLV Jornadas Argentinas de Informática e Investigación Operativa (45JAIIO)- XIV Simposio Argentino de Investigación Operativa (SIO) (Buenos Aires, 2016)* (pp. 59–66). URL: http://45jaiio.sadio.org.ar/sites/default/files/Sio-23.pdf

[Dezert et al., 2020] Dezert, J., Tchamova, A., Han, D., & Tacnet, J.-M. (2020). The spotis rank reversal free method for multi-criteria decision-making support. *2020 IEEE 23rd International Conference on Information Fusion (FUSION)*.

[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.

[GHORABAEE et al., 2016] GHORABAEE, M. K., ZAVADSKAS, E. K., TURSKIS, Z., & ANTUCHEVICIENE, J. (2016). A new combinative distance-based assessment (codas) method for multi-criteria decision-making. *ECONOMIC COMPUTATION AND ECONOMIC CYBERNETICS STUDIES AND RESEARCH*, *50*(3), 25–44.

[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.

[Isk & Adal, 2016] Işık, A. T., & Adalı, E. A. (2016). A new integrated decision making approach based on swara and ocra methods for the hotel selection problem. *International Journal of Advanced Operations Management*, *8*(2), 140–151.

[KeshavarzGhorabaee et al., 2015] Keshavarz Ghorabaee, M., Zavadskas, E. K., Olfat, L., & Turskis, Z. (2015). Multi-criteria inventory classification using a new method of evaluation based on distance from average solution (edas). *Informatica*, *26*(3), 435–451.

[Keshavarz-Ghorabaee et al., 2021] Keshavarz-Ghorabaee, M., Amiri, M., Zavadskas, E. K., Turskis, Z., & Antucheviciene, J. (2021). Determination of objective weights using a new method based on the removal effects of criteria (merec). *Symmetry*, *13*(4), 525.

[Li & Chi, 2009] Li, G., & Chi, G. (2009). A new determining objective weights method-gini coefficient weight. *2009 First International Conference on Information Science and Engineering* (pp. 3726–3729).

[Liu & Ma, 2021] Liu, X., & Ma, Y. (2021). A method to analyze the rank reversal problem in the electre ii method. *Omega*, *102*, 102317. URL: https://www.sciencedirect.com/science/article/pii/S030504832030671X, doi:https://doi.org/10.1016/j.omega.2020.102317

[Madic et al., 2015] Madić, M., Petković, D., & Radovanović, M. (2015). Selection of non-conventional machining processes using the ocra method. *Serbian Journal of Management*, *10*(1), 61–73.

[Miller & others, 1963] Miller, D. W., & others. (1963). Executive decisions and operations research. *AGRIS*.

[Opricovic & Tzeng, 2004] Opricovic, S., & Tzeng, G.-H. (2004). Compromise solution by mcdm methods: a comparative analysis of vikor and topsis. *European journal of operational research*, *156*(2), 445–455.

[Pamucar & Cirovic, 2015] Pamučar, D., & Ćirović, G. (2015). The selection of transport and handling resources in logistics centers using multi-attributive border approximation area comparison (mabac). *Expert Systems with Applications*, *42*(6), 3016-3028. URL: https://www.sciencedirect.com/science/article/pii/S0957417414007568, doi:https://doi.org/10.1016/j.eswa.2014.11.057

[Parkan, 1994] Parkan, C. (1994). Operational competitiveness ratings of production units. *Managerial and Decision Economics*, *15*(3), 201–221.

[Rajkumar, 2021] Rajkumar, S. (2021 , Jul). *Cryptocurrency historical prices*.

[Rogers et al., 2000] Rogers, M., Bruen, M., & Maystre, L.-Y. (2000 , 01). *ELECTRE and Decision Support*. Springer New York, NY.

[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 Confèrence internationale de recherché opérationalle*.

[Shyur et al., 2015] Shyur, H.-J., Yin, L., Shih, H.-S., & Cheng, C.-B. (2015). A multiple criteria decision making method based on relative value distances. *Foundations of Computing and Decision Sciences*, *40*(4), 299–315.

[Simon, 1955] Simon, H. A. (1955). A behavioral model of rational choice. *The quarterly journal of economics*, *69*(1), 99–118.

[Sotoudeh-Anvari, 2023] Sotoudeh-Anvari, A. (2023). Root assessment method (ram): a novel multi-criteria decision making method and its applications in sustainability challenges. *Journal of Cleaner Production*, *423*, 138695. URL: https://www.sciencedirect.com/science/article/pii/S0959652623028536, doi:https://doi.org/10.1016/j.jclepro.2023.138695

[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*.

[Wang et al., 2021] Wang, Z., Rangaiah, G. P., & Wang, X. (2021). Preference ranking on the basis of ideal-average distance method for multi-criteria decision-making. *Industrial & Engineering Chemistry Research*, *60*(30), 11216–11230.

[Wieckowski et al., 2023] Więckowski, J., Kizielewicz, B., Shekhovtsov, A., & Sałabun, W. (2023). Rancom: a novel approach to identifying criteria relevance based on inaccuracy expert judgments. *Engineering Applications of Artificial Intelligence*, *122*, 106114. URL: https://www.sciencedirect.com/science/article/pii/S0952197623002981, doi:https://doi.org/10.1016/j.engappai.2023.106114

[Yazdani et al., 2019] Yazdani, M., Zaraté, P., Kazimieras Zavadskas, E., & Turskis, Z. (2019). A combined compromise solution (cocoso) method for multi-criteria decision-making problems. *Management Decision*, *57*(9), 2501–2519.

[Zavadskas et al., 1994] Zavadskas, E., Kaklauskas, A., & Šarka, V. (1994 , 01). The new method of multicriteria complex proportional assessment of projects. *Technological and Economic Development of Economy*, *1*, 131-139.

[Zavadskas & Turskis, 2010] Zavadskas, E. K., & Turskis, Z. (2010). A new additive ratio assessment (aras) method in multicriteria decision-making. *Technological and economic development of economy*, *16*(2), 159–172.

[Zavadskas et al., 2012] Zavadskas, E. K., Turskis, Z., Antucheviciene, J., & Zakarevicius, A. (2012). Optimization of weighted aggregated sum product assessment. *Elektronika ir elektrotechnika*, *122*(6), 3–6.

[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].

[Wikipedia contributors, 2023] Wikipedia contributors (2023). *Academic publishing — Wikipedia, The Free Encyclopedia*. [Online; accessed 28-January-2023].

PYTHON MODULE INDEX

S

INDEX

D

E

F

G

H

I

K

L

M

N

O

P

Q

R

S

T

U

V

W

Y