Scikit-Criteria Documentation

Release 0.8.7.dev0

Juan BC

Feb 09, 2024

TUTORIALS

Ver. 0.8.7.dev0

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

CODE REPOSITORY & ISSUES

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

CHAPTER

TWO

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

THREE

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

FOUR

CONTENTS

4.1 Installation

4.1.1 Using conda

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

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

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

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

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

If the installation fails for any reason, please open an issue in the issue tracker.

4.1.2 Alternative installation methods

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

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

Finally, you can also install the latest development version of scikit-criteria directly from GitHub:

```
$ pip install git+https://github.com/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.

4.1.3 If you don't have Python

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

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

4.2 Tutorials

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

Contents:

4.2.1 Quick Start

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

Conceptual overview

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

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

Additionally it can be accompanied by a vector `w`/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"],
```

(continues on next page)

(continued from previous page)

```
)
dm
```

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

Manipulating the Data

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

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

[7]:
```
dm.matrix  # note how this data ignores the objectives and the weights
```

[7]:
```
Criteria      autonomy  comfort  price
Alternatives
car 0                1        2      3
car 1                4        5      6
```

[8]:
```
dm.objectives
```

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

[9]:
```
dm.weights
```

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

[10]:
```
dm.alternatives, dm.criteria
```

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

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

[11]:
```
dm = dm.copy(alternatives=["VW", "Ford"])
dm
```

[11]:
```
      autonomy[ 0.50]  comfort[ 0.05]  price[ 0.45]
VW                  1               2             3
Ford                4               5             6
[2 Alternatives x 3 Criteria]
```

Note:

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

Plotting

The Data structure support some basic routines for plotting.

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

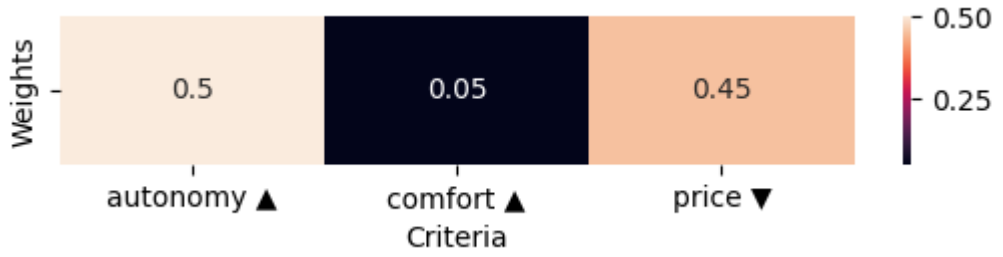
[12]:
```
dm.plot()
```

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

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

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

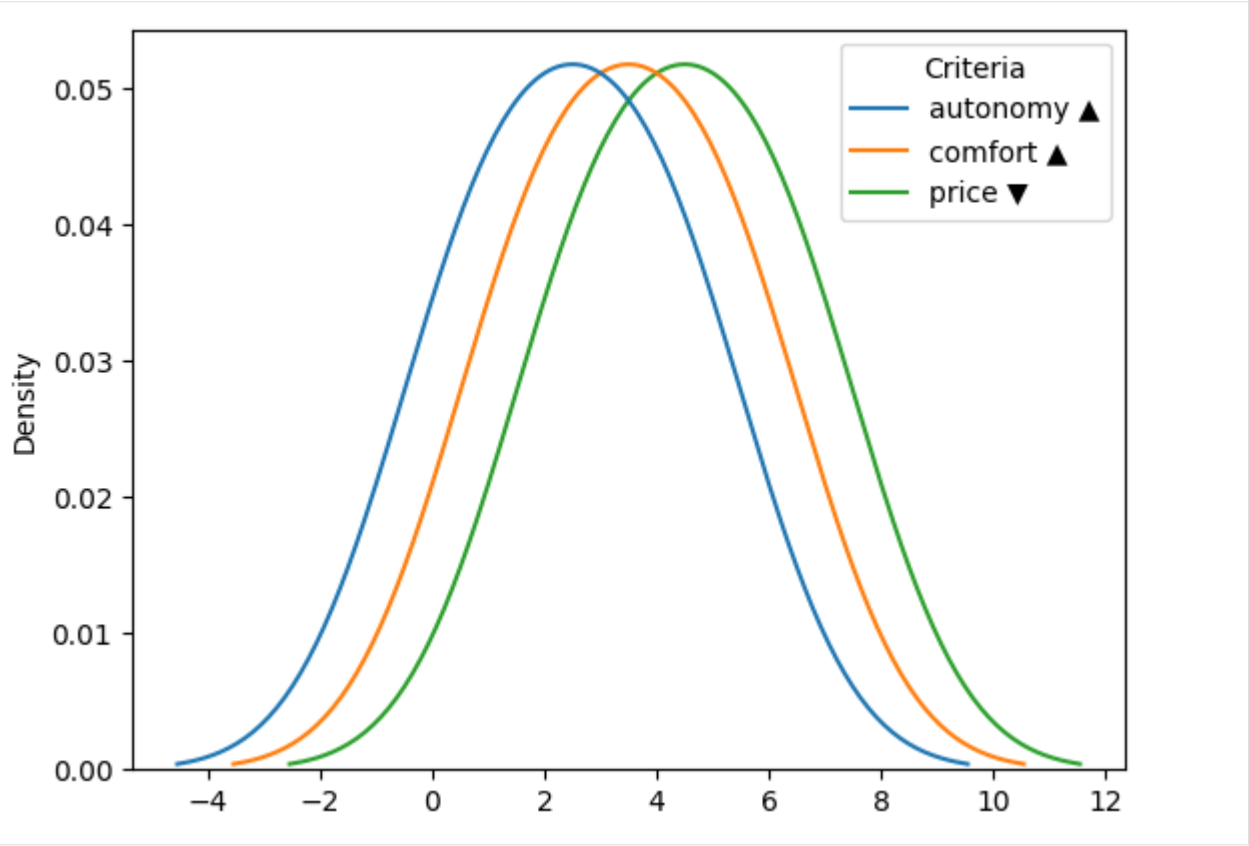
[13]:
```
<AxesSubplot:xlabel='Criteria'>
```



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

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

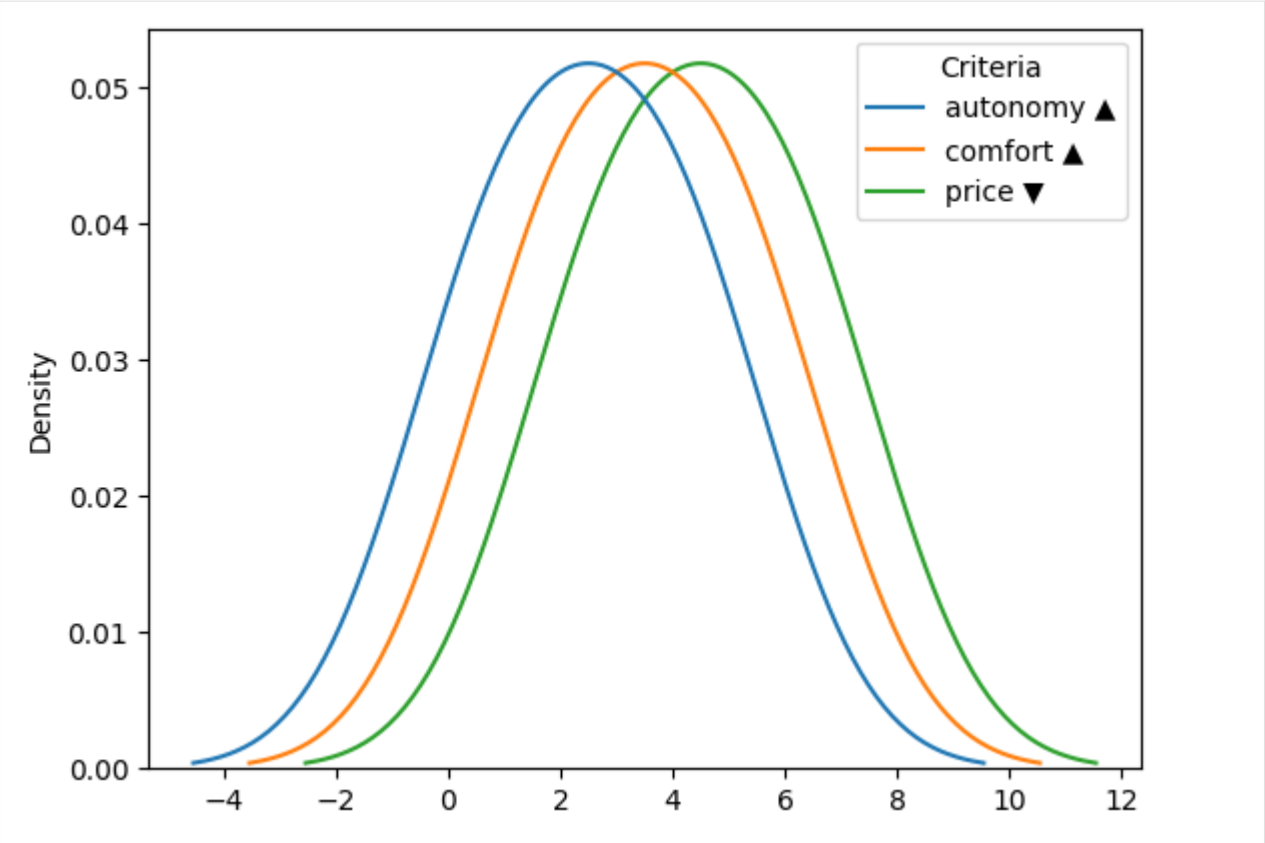
[14]:
```
<AxesSubplot:ylabel='Density'>
```



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

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

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

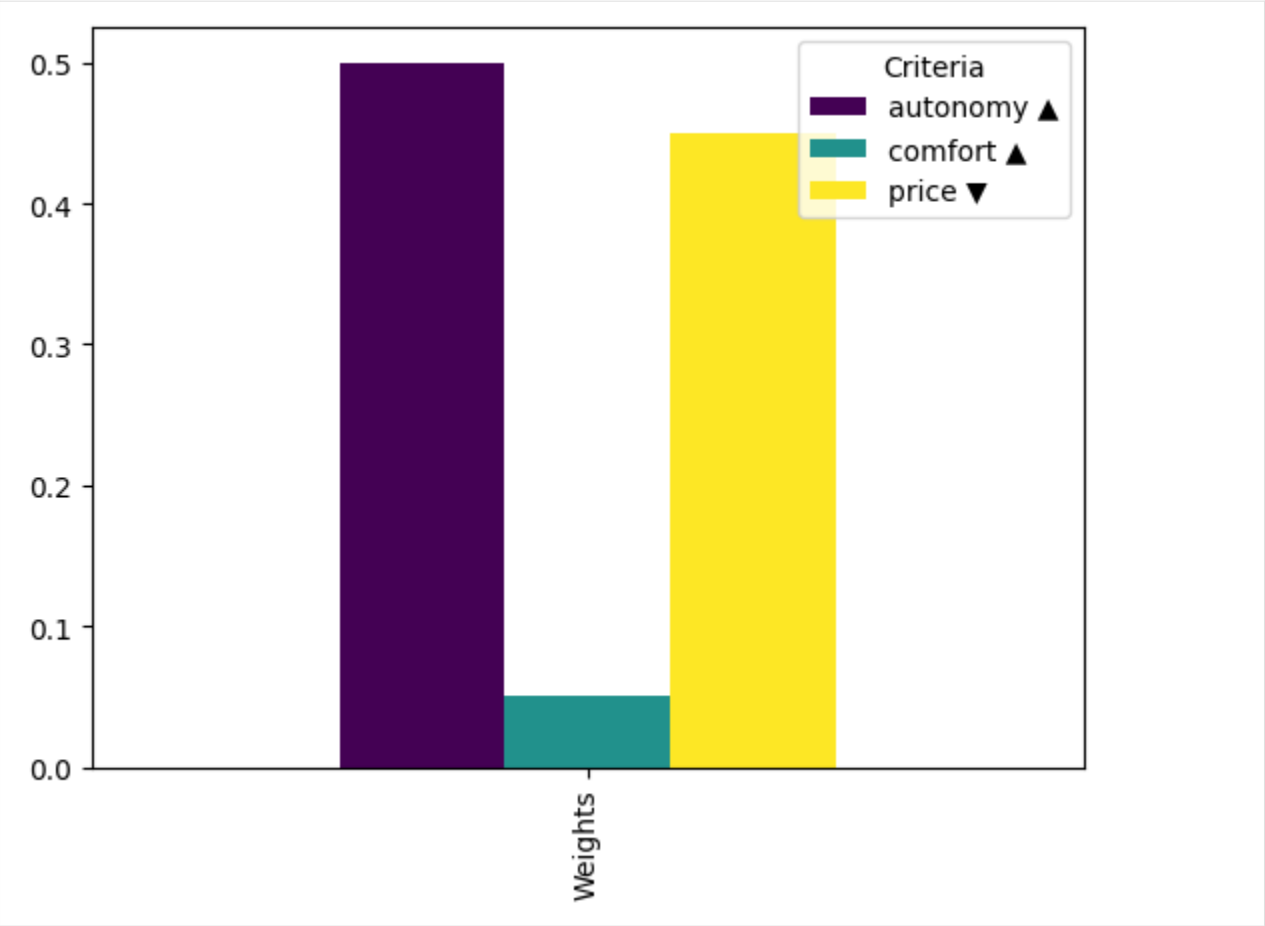


Every plot has their own set of parameters, defined by the subjacent function

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

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

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



Data transformation

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

Note: Scikit-Criteria objective preference

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

To solve these problems, we will use two processors:

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

First we start by importing the two necessary modules.

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

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

This involves:

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

In code:

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

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

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

For this step we need

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

   In our case we are going to ask the scaler to scale both components of the decision matrix (`target="both"`)
2. Apply the transformation by calling the `transform` method of the transformer and passing it as parameter our decision matrix `dmt`.
3. Save the transformed decision by overwriting the variable `dmt`.

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

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

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

Note: Advanced plots with Matplotlib

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

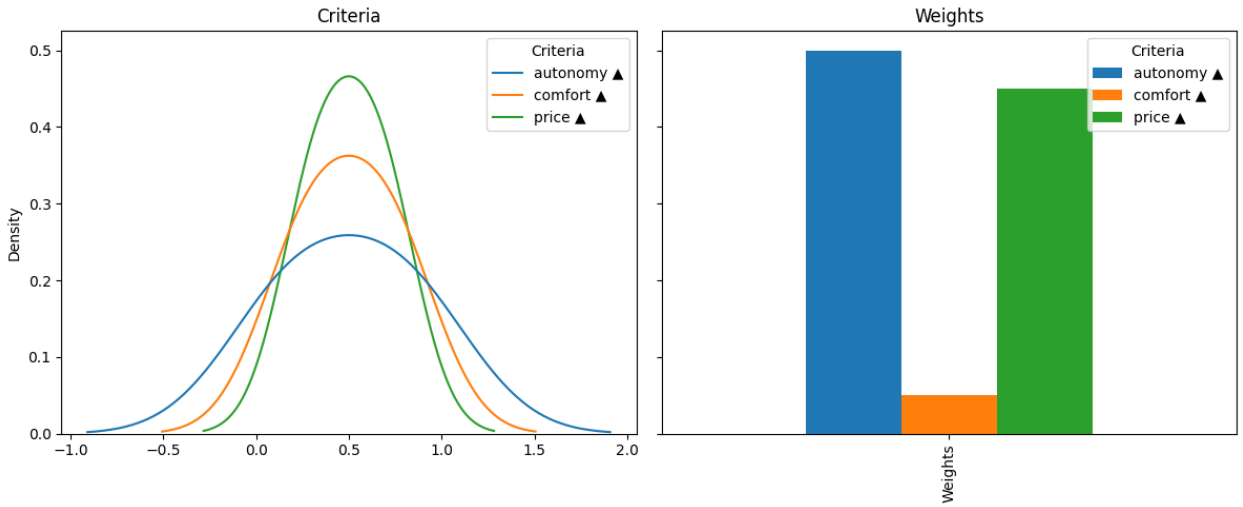
---

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

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

(continues on next page)

(continued from previous page)

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

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


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

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



Using this data to feed some MCDA methods

Weighted Sum Model

Let's rank our dummy data by Weighted Sum Model

First we need to import the required module

[21]:
```
from skcriteria.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`.

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

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

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

pipe
```

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

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

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

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

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

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

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

Where the `ideal` and `anti_ideal` are the 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. 2024-02-09T19:34:46.636173

4.2.2 Dominance and satisfaction analysis (AKA filters)

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

Case

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

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

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

The full decision matrix

[1]:
```
import skcriteria as skc

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

dm
```

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

Satisfaction analysis

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

The basic idea was proposed in the work of "*A Behavioral Model of Rational Choice*" [Simon, 1955] and presents the definition of "*aspiration levels*" and are set a priori by the decision maker.

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

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.

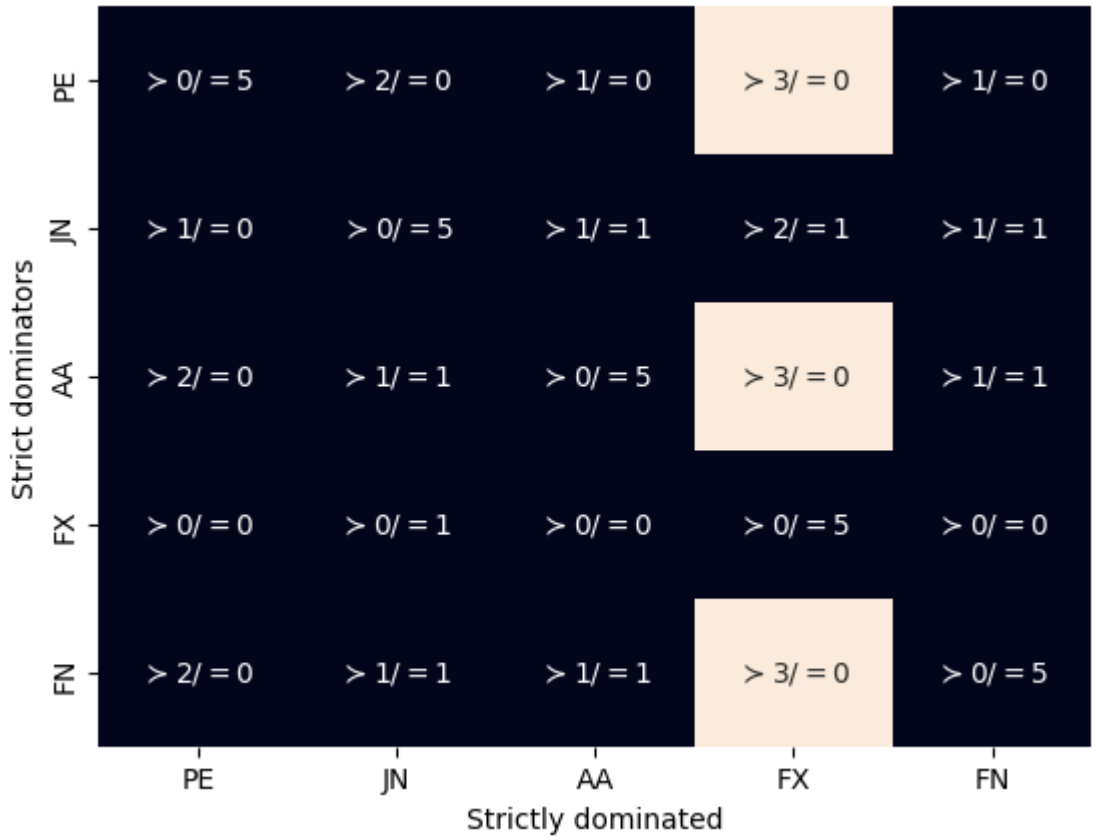
[ ]:
```
dmf.dominance(strict=True)  # equivalent to dmf.dominance.dominance()
```

```
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
```

(continues on next page)

(continued from previous page)

```
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(),
```

(continues on next page)

(continued from previous page)

```
)

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. 2024-02-09T19:34:46.636173

4.2.3 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 (
```

(continues on next page)

(continued from previous page)

```
    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(),
)

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)]`

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

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

2. we let the names be inferred from the methods with the `mkrank_cmp()` function

[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 `pandas.DataFrame`

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

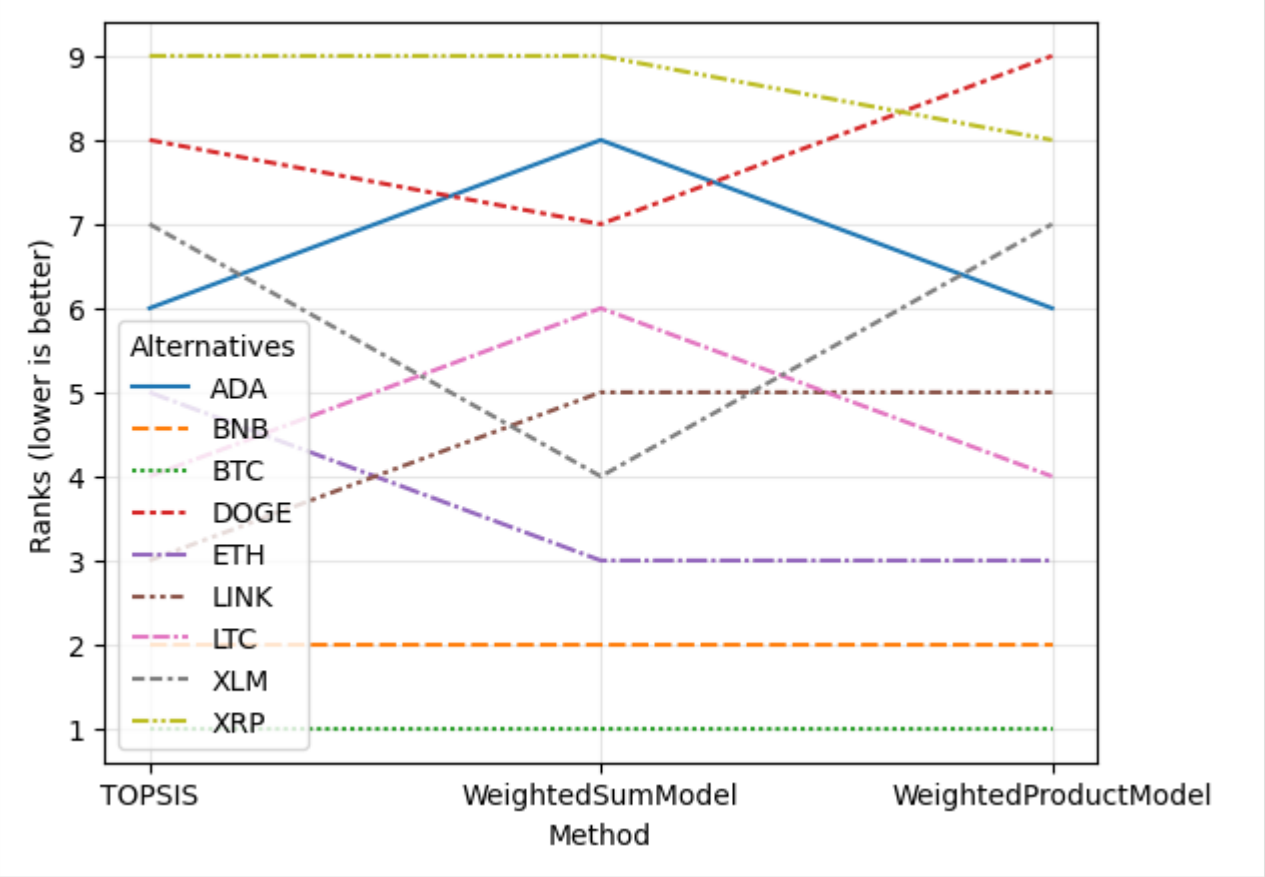
[14]:
```
Method       TOPSIS  WeightedSumModel  WeightedProductModel
Alternatives
ADA               6                 8                     6
BNB               2                 2                     2
BTC               1                 1                     1
DOGE              8                 7                     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.

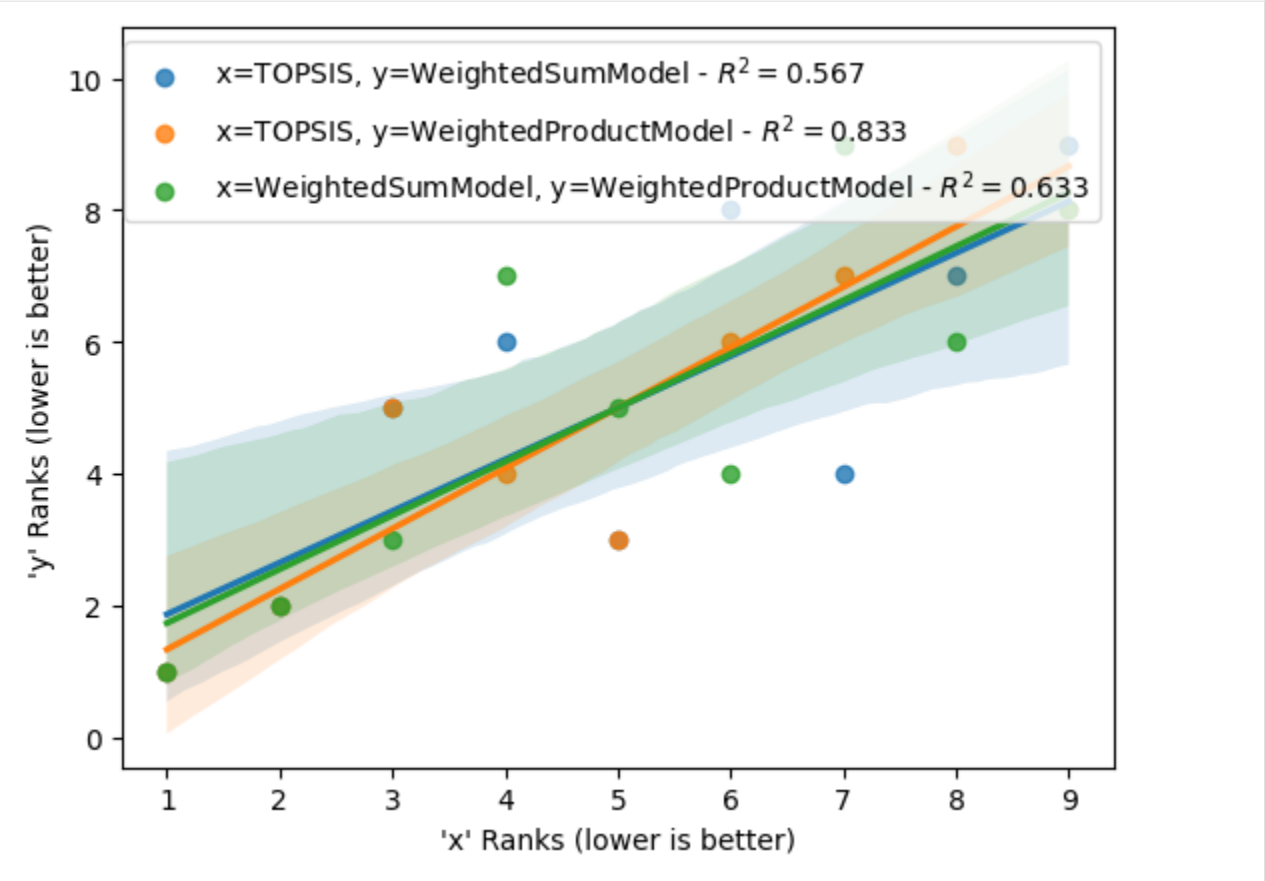
A classic in the area *Ranking flows*

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



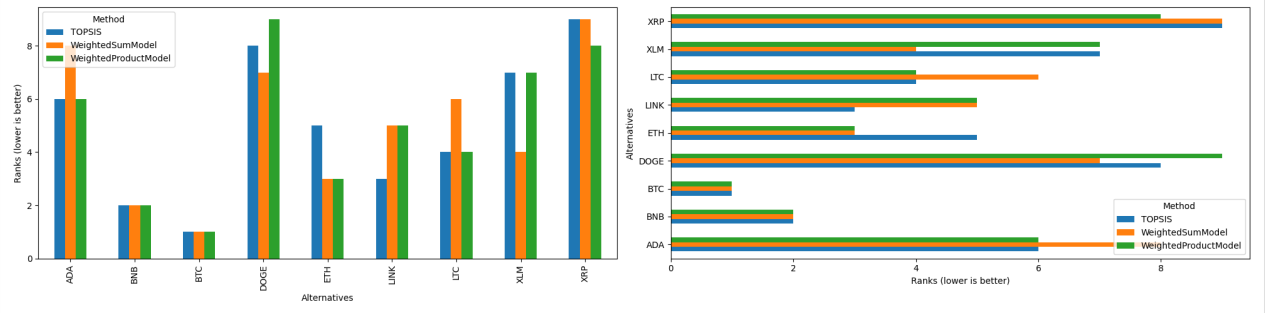
We can also run regressions on all combinations of different rankings.

[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();
```



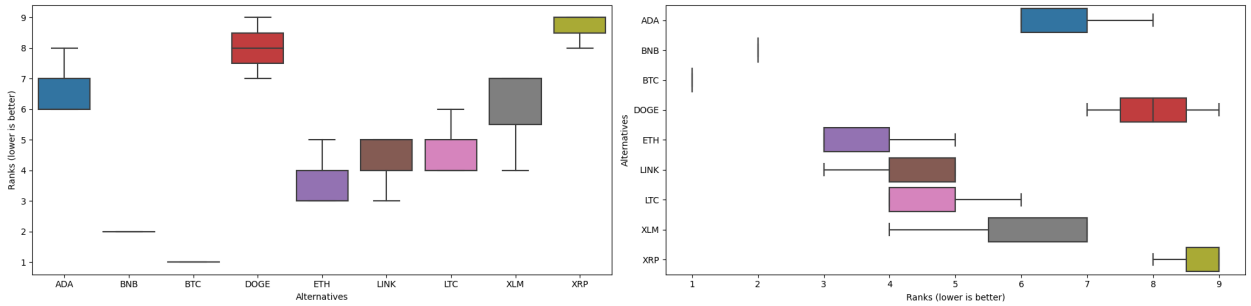
and boxplots

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

(continues on next page)

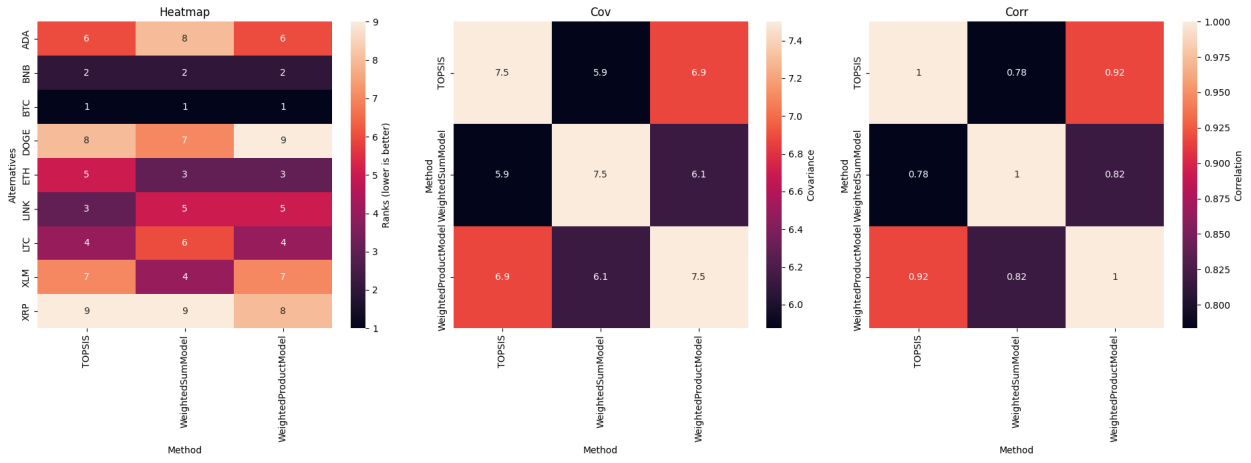
(continued from previous page)

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

fig.tight_layout();
```

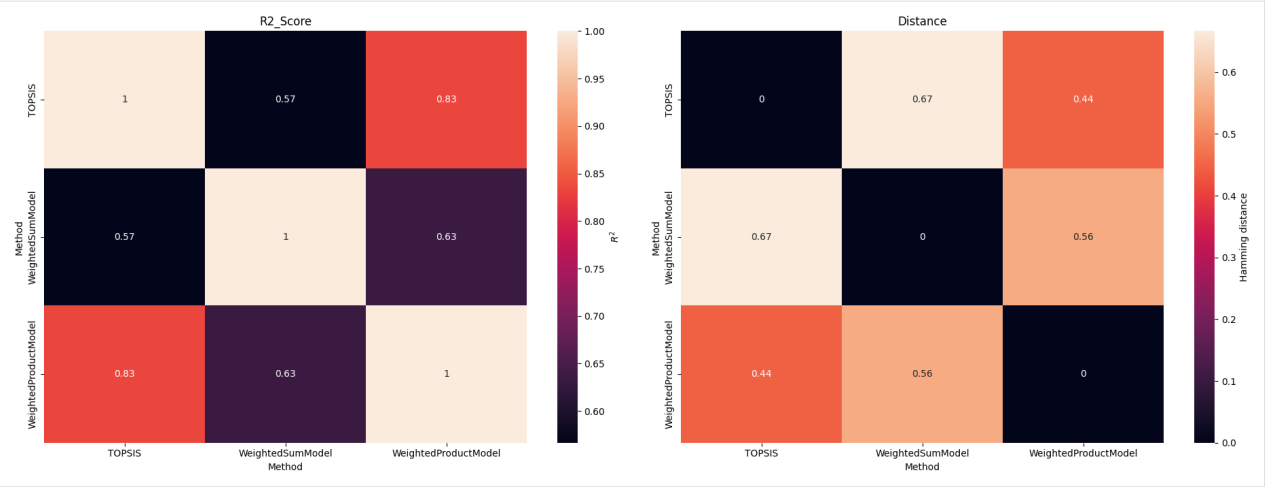


And you can visualize all matrices with statistical heatmap-like plots.

[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. 2024-02-09T19:34:46.636173

4.2.4 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:

[5]:
```
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"`.

[10]:
```
from skcriteria.agg import similarity
similarity.TOPSIS()
```

[10]:
```
<TOPSIS [metric='euclidean']>
```

[11]:
```
similarity.TOPSIS(metric="cityblock")
```

[11]:
```
<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`.

[12]:
```
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)

    # 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.

[20]:
```
@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
```

[20]:
```
<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
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.

[27]:
```
@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
```

[27]:
```
<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.

[30]:
```
@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)
```

[30]:
```
    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. 2024-02-09T19:34:46.636173

4.2.5 Extra tutorials

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

4.2.6 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

4.3 skcriteria package

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

4.3.1 skcriteria.core package

Core functionalities and structures of skcriteria.

skcriteria.core.data module

Data abstraction layer.

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

class skcriteria.core.data.**DecisionMatrix**(*data_df*, *objectives*, *weights*)

Bases: *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.

copy(***kwargs*)

Return a deep copy of the current DecisionMatrix.

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

Parameters

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

Returns
A new decision matrix.

Return type
DecisionMatrix

`to_dataframe()`

Convert the entire DecisionMatrix into a dataframe.

The objectives and weights 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
weights     1.0  1.0  1.0
A0            1    2    3
A1            4    5    6
```

`to_dict()`

Return a dict representation of the data.

All the values are represented as numpy array.

`describe(**kwargs)`

Generate descriptive statistics.

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

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

Returns
Summary statistics of DecisionMatrix provided.

Return type
pandas.DataFrame

`property shape`

Return a tuple with (number_of_alternatives, number_of_criteria).

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

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*)

Retorna True si la matriz contiene loops de dominacia.

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

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

Returns
If True a loop exists.

Return type
bool

Notes

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

skcriteria.core.methods module

Core functionalities of scikit-criteria.

class skcriteria.core.methods.**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*)

Return a custom copy of the current decision-maker.

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

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

Return type
A new object.

skcriteria.core.objectives module

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

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

Bases: Enum

Representation of criteria objectives (Minimize, Maximize).

MIN = -1

Internal representation of minimize criteria

MAX = 1

Internal representation of maximize criteria

classmethod from_alias(*alias*)

Return a n objective instase based on some given alias.

to_symbol()

Return the printable symbol representation of the objective.

classmethod construct_from_alias(*alias*)

Return an objective instance based on some given alias.

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

to_string()

Return the printable representation of the objective.

skcriteria.core.plot module

Plot helper for the DecisionMatrix object.

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

Bases: *AccessorABC*

DecisionMatrix plot utilities.

Kind of plot to produce:

- 'heatmap' : criteria heat-map (default).

- 'wheatmap' : weights heat-map.
- 'bar' : criteria vertical bar plot.
- 'wbar' : weights vertical bar plot.
- 'barh' : criteria horizontal bar plot.
- 'wbarh' : weights horizontal bar plot.
- 'hist' : criteria histogram.
- 'whist' : weights histogram.
- 'box' : criteria boxplot.
- 'wbox' : weights boxplot.
- 'kde' : criteria Kernel Density Estimation plot.
- 'wkde' : weights Kernel Density Estimation plot.
- 'ogive' : criteria empirical cumulative distribution plot.
- 'wogive' : weights empirical cumulative distribution plot.
- 'area' : criteria area plot.
- 'dominance': the dominance matrix as a heatmap.
- 'frontier': criteria pair-wise Pareto-Frontier.

heatmap(***kwargs*)

Plot the alternative matrix as a color-encoded matrix.

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

Return type
matplotlib.axes.Axes or numpy.ndarray of them

wheatmap(***kwargs*)

Plot weights as a color-encoded matrix.

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

Return type
matplotlib.axes.Axes or numpy.ndarray of them

bar(***kwargs*)

Criteria vertical bar plot.

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

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

Return type
matplotlib.axes.Axes or numpy.ndarray of them

wbar(***kwargs*)

Weights vertical bar plot.

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

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

Return type
matplotlib.axes.Axes or numpy.ndarray of them

barh(***kwargs*)

Criteria horizontal bar plot.

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

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

Return type
matplotlib.axes.Axes or numpy.ndarray of them

wbarh(***kwargs*)

Weights horizontal bar plot.

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

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

Return type
matplotlib.axes.Axes or numpy.ndarray of them

hist(***kwargs*)

Draw one histogram of the criteria.

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

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

Return type
matplotlib.axes.Axes or numpy.ndarray of them

whist(***kwargs*)

Draw one histogram of the weights.

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

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

Return type
matplotlib.axes.Axes or numpy.ndarray of them

box(***kwargs*)

Make a box plot of the criteria.

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

For further details see Wikipedia's entry for boxplot.

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

Return type
matplotlib.axes.Axes or numpy.ndarray of them

wbox(***kwargs*)

Make a box plot of the weights.

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

For further details see Wikipedia's entry for boxplot.

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

Return type
matplotlib.axes.Axes or numpy.ndarray of them

kde(***kwargs*)

Criteria kernel density plot using Gaussian kernels.

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

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

Return type
matplotlib.axes.Axes or numpy.ndarray of them

wkde(***kwargs*)

Weights kernel density plot using Gaussian kernels.

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

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

Return type
matplotlib.axes.Axes or numpy.ndarray of them

`ogive`(***kwargs*)

Criteria empirical cumulative distribution plot.

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

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

Return type
matplotlib.axes.Axes or numpy.ndarray of them

`wogive`(***kwargs*)

Weights empirical cumulative distribution plot.

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

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

Return type
matplotlib.axes.Axes or numpy.ndarray of them

`area`(***kwargs*)

Draw an criteria stacked area plot.

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

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

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

Return type
matplotlib.axes.Axes or numpy.ndarray

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

Plot dominance as a color-encoded matrix.

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

Parameters

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

Return type
matplotlib.axes.Axes or numpy.ndarray of them

frontier(*x*, *y*, *, *strict=False*, *ax=None*, *legend=True*, *scatter_kws=None*, *line_kws=None*)

Pareto frontier on two arbitrarily selected criteria.

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

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

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

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

Parameters

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

Return type
matplotlib.axes.Axes or numpy.ndarray of them

References

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

skcriteria.core.stats module

Stats helper for the DecisionMatrix object.

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

Bases: *AccessorABC*

Calculate basic statistics of the decision matrix.

Kind of statistic to produce:

- 'corr' : Compute pairwise correlation of columns, excluding NA/null values.
- 'cov' : Compute pairwise covariance of columns, excluding NA/null values.

- 'describe' : Generate descriptive statistics.
- 'kurtosis' : Return unbiased kurtosis over requested axis.
- 'mad' : Return the mean absolute deviation of the values over the requested axis.
- 'max' : Return the maximum of the values over the requested axis.
- 'mean' : Return the mean of the values over the requested axis.
- 'median' : Return the median of the values over the requested axis.
- 'min' : Return the minimum of the values over the requested axis.
- 'pct_change' : Percentage change between the current and a prior element.
- 'quantile' : Return values at the given quantile over requested axis.
- 'sem' : Return unbiased standard error of the mean over requested axis.
- 'skew' : Return unbiased skew over requested axis.
- 'std' : Return sample standard deviation over requested axis.
- 'var' : Return unbiased variance over requested axis.

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

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

Parameters

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

4.3.2 skcriteria.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.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.

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**(*matrix*, *objectives*, *weights*, *p0=0.65*, *p1=0.5*, *p2=0.35*, *q0=0.65*, *q1=0.35*)

Execute ELECTRE2 without any validation.

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

class skcriteria.agg.electre.**ELECTRE2**(**args*, ***kwargs*)

Bases: *SKCDecisionMakerABC*

Find the ranking solution through ELECTRE-2.

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

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

Notes

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

Parameters

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

  These thresholds must meet the condition "1 >= 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".

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.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.similarity module

Methods based on a similarity between alternatives.

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

Execute TOPSIS without any validation.

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.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

4.3.3 skcriteria.preprocessing package

Multiple data transformation routines.

skcriteria.preprocessing._preprocessing_base module

Core functionalities to create transformers.

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

Bases: *SKCMethodABC*

Abstract class for all transformer in scikit-criteria.

transform(*dm*)

Perform transformation on *dm*.

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

Returns
Transformed decision matrix.

Return type
skcriteria.data.DecisionMatrix

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

Bases: *SKCTransformerABC*

Transform weights and matrix together or independently.

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

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

property target

Determine which part of the DecisionMatrix will be transformed.

skcriteria.preprocessing.distance module

Warning: This module is deprecated.

Normalization through the distance to distance function.

This entire module is deprecated.

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

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

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

The result score x_{aj}‘$expressesthedegreetowhichthealternative : math : ‘a$ 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].

The result score x_{aj}‘$expressesthedegreetowhichthealternative : math : ‘a$ 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"],
... )

>>> 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.

  New 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.

  New 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.

  New 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.

  New 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`.

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.

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

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]])
```

(continues on next page)

(continued from previous page)

```
# 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 positive along an axis.

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

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].

The result score x_{aj}'$expressesthedegreetowhichthealternative : math : 'a$ 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].

The result score x_{aj}‘$expressesthedegreetowhichthealternative : math : ‘a$ 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]

4.3.4 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.

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.

class skcriteria.cmp.ranks_rev.rank_inv_check.**RankInvariantChecker**(*dmaker*, *, *repeat=1*, *allow_missing_alternatives=False*, *last_diff_strategy='median'*, *random_state=None*)

Bases: *SKCMethodABC*

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.

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 original 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 *rrt1* 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.cmp.ranks_cmp module

Ranking comparison routines.

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

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.

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', 'mahalanobis', 'matching', 'minkowski', 'rogerstanimoto', 'russellrao', 'seuclidean', 'sokalmichener', 'sokalsneath', 'sqeuclidean', 'yule'.
- **kwargs** – Other keyword arguments are passed to the scipy.spatial.distance.pdist() function.

Returns
A DataFrame with the distance between rankings.

Return type
pd.DataFrame

property plot

Plot accessor.

class skcriteria.cmp.ranks_cmp.**RanksComparatorPlotter**(*ranks_cmp*)

Bases: *AccessorABC*

RanksComparator plot utilities.

Kind of plot to produce:

- 'flow' : Changes in the rankings of the alternatives as flow lines (default)
- 'reg' : Pairwise rankings data and a linear regression model fit plot.
- 'heatmap' : Rankings as a color-encoded matrix.
- 'corr' : Pairwise correlation of rankings as a color-encoded matrix.
- 'cov' : Pairwise covariance of rankings as a color-encoded matrix.
- 'r2_score' : Pairwise coefficient of determination regression score function of rankings as a color-encoded matrix.
- 'distance' : Pairwise distance between rankings as a color-encoded matrix.
- 'box' : Box-plot of rankings with respect to alternatives
- 'bar' : Ranking of alternatives by method with vertical bars.
- 'barh' : Ranking of alternatives by method with horizontal bars.

flow(*, *untied=False*, *grid_kws=None*, ***kwargs*)

Represents changes in the rankings of the alternatives as lines flowing through the ranking-methods.

Parameters

- **untied** (bool, default `False`) – If it is `True` and any ranking has ties, the `RankResult.untied_rank_` property is used to assign each alternative a single ranked order. On the other hand, if it is `False` the rankings are used as they are.
- **grid_kws** (*dict or None*) – Dict with keyword arguments passed to `matplotlib.axes.plt.Axes.grid`
- **kwargs** – Other keyword arguments are passed to the `seaborn.lineplot()` function. except for data, estimator and sort.

Return type

matplotlib.axes.Axes or numpy.ndarray of them

reg(*, *untied=False*, *r2=True*, *palette=None*, *legend=True*, *r2_fmt='.2g'*, *r2_kws=None*, ***kwargs*)

Plot a pairwise rankings data and a linear regression model fit.

Parameters

- **untied** (bool, default `False`) – If it is `True` and any ranking has ties, the `RankResult.untied_rank_` property is used to assign each alternative a single ranked order. On the other hand, if it is `False` the rankings are used as they are.
- **r2** (bool, default `True`) – If True, the coefficient of determination results are added to the regression legend.
- **palette** (matplotlib/seaborn color palette, default `None`) – Set of colors for mapping the hue variable.
- **legend** (bool, default `True`) – If False, suppress the legend for semantic variables.
- **r2_fmt** (str, default `"2.g"`) – String formatting code to use when adding the coefficient of determination.
- **r2_kws** (*dict or None*) – Dict with keywords arguments passed to `sklearn.metrics.r2_score()` function.
- **kwargs** – Other keyword arguments are passed to the `seaborn.lineplot()` function.

Return type

matplotlib.axes.Axes or numpy.ndarray of them

heatmap(*, *untied=False*, ***kwargs*)

Plot the rankings as a color-encoded matrix.

Parameters

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

Return type

matplotlib.axes.Axes or numpy.ndarray of them

corr(*, *untied=False*, *corr_kws=None*, ***kwargs*)

Plot the pairwise correlation of rankings as a color-encoded matrix.

By default the pearson correlation coefficient is used.

Parameters

- **untied** (bool, default `False`) – If it is `True` and any ranking has ties, the `RankResult.untied_rank_` property is used to assign each alternative a single ranked order. On the other hand, if it is `False` the rankings are used as they are.
- **corr_kws** (*dict or None*) – Dict with keywords arguments passed the `pandas.DataFrame.corr()` method.
- **kwargs** – Other keyword arguments are passed to the `seaborn.heatmap()` function.

Return type
matplotlib.axes.Axes or numpy.ndarray of them

cov(*, *untied=False*, *cov_kws=None*, ***kwargs*)

Plot the pairwise covariance of rankings as a color-encoded matrix.

Parameters

- **untied** (bool, default `False`) – If it is `True` and any ranking has ties, the `RankResult.untied_rank_` property is used to assign each alternative a single ranked order. On the other hand, if it is `False` the rankings are used as they are.
- **cov_kws** (*dict or None*) – Dict with keywords arguments passed the `pandas.DataFrame.cov()` method.
- **kwargs** – Other keyword arguments are passed to the `seaborn.heatmap()` function.

Return type
matplotlib.axes.Axes or numpy.ndarray of them

r2_score(*untied=False*, *r2_kws=None*, ***kwargs*)

Plot the pairwise coefficient of determination regression score function of rankings as a color-encoded matrix.

Parameters

- **untied** (bool, default `False`) – If it is `True` and any ranking has ties, the `RankResult.untied_rank_` property is used to assign each alternative a single ranked order. On the other hand, if it is `False` the rankings are used as they are.
- **cov_kws** (*dict or None*) – Dict with keywords arguments passed the `pandas.DataFrame.cov()` method.
- **kwargs** – Other keyword arguments are passed to the `seaborn.heatmap()` function.

Return type
matplotlib.axes.Axes or numpy.ndarray of them

distance(*, *untied=False*, *metric='hamming'*, *distance_kws=None*, ***kwargs*)

Plot the pairwise distance between rankings as a color-encoded matrix.

By default the 'hamming' distance is used, which is simply the proportion of disagreeing components in Two rankings.

Parameters

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

- **metric** (str or function, default "hamming") – The distance metric to use. The distance function can be 'braycurtis', 'canberra', 'chebyshev', 'cityblock', 'correlation', 'cosine', 'dice', 'euclidean', 'hamming', 'jaccard', 'jensenshannon', 'kulczynski1', 'mahalanobis', 'matching', 'minkowski', 'rogerstanimoto', 'russellrao', 'seuclidean', 'sokalmichener', 'sokalsneath', 'sqeuclidean', 'yule'.
- **distance_kws** (*dict or None*) – Dict with keywords arguments passed the scipy.spatial.distance.pdist function
- **kwargs** – Other keyword arguments are passed to the seaborn.heatmap() function.

Return type
matplotlib.axes.Axes or numpy.ndarray of them

box(*, *untied=False*, ***kwargs*)

Draw a boxplot to show rankings with respect to alternatives.

Parameters

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

Return type
matplotlib.axes.Axes or numpy.ndarray of them

bar(*, *untied=False*, ***kwargs*)

Draw plot that presents ranking of alternatives by method with vertical bars.

Parameters

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

Return type
matplotlib.axes.Axes or numpy.ndarray of them

barh(*, *untied=False*, ***kwargs*)

Draw plot that presents ranking of alternatives by method with horizontal bars.

Parameters

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

Return type
matplotlib.axes.Axes or numpy.ndarray of them

skcriteria.cmp.ranks_cmp.**mkrank_cmp**(**ranks*)

Construct a RankComparator from the given rankings.

This is a shorthand for the RankComparator constructor; it does not require, and does not permit, naming the estimators. Instead, their names will be set to the method attribute of the rankings automatically.

Parameters
***ranks** (*list of RankResult objects*) – List of the scikit-criteria RankResult objcects.

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

Return type
RanksComparator

4.3.5 skcriteria.datasets package

The *skcriteria.datasets* module includes utilities to load datasets.

skcriteria.datasets.**load_simple_stock_selection**()

Simple stock selection decision matrix.

This matrix was designed primarily for teaching and evaluating the behavior of an experiment.

Among the data we can find: two maximization criteria (ROE, CAP), one minimization criterion (RI), dominated alternatives (FX), and one alternative with an outlier criterion (ROE, MM = 1).

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]

4.3.6 skcriteria.pipeline module

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

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

4.3.7 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'.

4.3.8 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.

4.3.9 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
>>> b.c = 6
>>> b['c']
6
```

skcriteria.utils.cmanagers module

Multiple context managers to use inside scikit-criteria.

skcriteria.utils.cmanagers.**df_temporal_header**(*df*, *header*, *name=None*)

Temporarily replaces a DataFrame columns names.

Optionally also assign another name to the columns.

Parameters

- **header** (*sequence*) – The new names of the columns.
- **name** (*str or None (default None)*) – New name for the index containing the columns in the DataFrame. If 'None' the original name of the columns present in the DataFrame is preserved.

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.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*, *version*, *, *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.
- **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".
- **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.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 revese 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

4.3.10 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

4.4 Changelog

4.4.1 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'
>>>
>>> @mktransformer
>>> def MyTransformer(**kwargs):
>>>     # Implementation of the transformation function
>>>
>>> @mktransformer(bar=2)
>>> def MyTransformerWithHyperparam(**kwargs):
```

(continues on next page)

(continued from previous page)

```
>>>     # 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

4.4.2 Version 0.8.6

- **New** Rank reversal 1 implementhed 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.

4.4.3 Version 0.8.3

- Fixed a bug detected on the EntropyWeighted, Now works as the literature specifies

---

4.4.4 Version 0.8.2

- We bring back Python 3.7 because is the version used in google.colab.
- Bugfixes in `plot.frontier` and `dominance.eq`.

4.4.5 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`

4.4.6 Version 0.7

- **New method**: `ELECTRE2`.
- **New preprocessing strategy:** A new way to transform from minimization to maximization criteria: `NegateMinimize()` which reverses the sign of the values of the criteria to be minimized (useful for not breaking distance relations in methods like *TOPSIS*). Additionally the previous we rename the `MinimizeToMaximize()` transformer to `InvertMinimize()`.
- Now the `RankingResult`, support repeated/tied rankings and some methods were implemented to deal with these cases.
  - `RankingResult.has_ties_` to see if there are tied values.
  - `RankingResult.ties_` to see how often values are repeated.
  - **`RankingResult.untied_rank_` to get a ranking with no repeated values.**
    repeated values.
- `KernelResult` now implements several new properties:
  - `kernel_alternatives_` to know which alternatives are in the kernel.
  - `kernel_size_` to know the number of alternatives in the kernel.
  - `kernel_where_` was replaced by `kernelwhere_` to standardize the api.

4.4.7 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.

4.4.8 Version 0.5

In this version scikit-criteria was rewritten from scratch. Among other things:

- The model implementation API was simplified.
- The `Data` object was removed in favor of `DecisionMatrix` which implements many more useful features for MCDA.
- Plots were completely re-implemented using Seaborn.
- Coverage was increased to 100%.
- Pipelines concept was added (Thanks to Scikit-learn).
- New documentation. The quick start is totally rewritten!

Full Changelog: https://github.com/quatrope/scikit-criteria/commits/0.5

4.4.9 Version 0.2

First OO stable version.

4.4.10 Version 0.1

Only functions.

4.5 Bibliography

4.6 Indices and tables

- genindex
- modindex
- search

BIBLIOGRAPHY

[Brauers & Zavadskas, 2006] Brauers, W. K., & Zavadskas, E. K. (2006). The moora method and its application to privatization in a transition economy. *Control and cybernetics*, *35*, 445–469.

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

[Bridgman, 1922] Bridgman, P. W. (1922). *Dimensional analysis*. Yale university press.

[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

[Diakoulaki et al., 1995] Diakoulaki, D., Mavrotas, G., & Papayannakis, L. (1995). Determining objective weights in multiple criteria problems: the critic method. *Computers & Operations Research*, *22*(7), 763–770.

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

[Gomes et al., 2004] Gomes, L., González-Araya, M., & Carignano, C. (2004 , 11). *Tomada de decisões em cenários complexos*. Thomson.

[Hwang & Yoon, 1981] Hwang, C.-L., & Yoon, K. (1981). Methods for multiple attribute decision making. *Multiple attribute decision making* (pp. 58–191). Springer.

[Miller & others, 1963] Miller, D. W., & others. (1963). Executive decisions and operations research. *AGRIS*.

[Rajkumar, 2021] Rajkumar, S. (2021 , Jul). *Cryptocurrency historical prices*.

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

[Roy, 1990] Roy, B. (1990). The outranking approach and the foundations of electre methods. *Readings in multiple criteria decision aid* (pp. 155–183). Springer.

[Roy & Bertier, 1971] Roy, B., & Bertier, P. (1971). La méthode electre ii. *Note de travail*, *142*.

[Roy & Bertier, 1973] Roy, B., & Bertier, P. (1973). La méthode electre ii(une application au média-planning...). *VII ème Confèrence internationale de recherché opérationalle*.

[Simon, 1955] Simon, H. A. (1955). A behavioral model of rational choice. *The quarterly journal of economics*, *69*(1), 99–118.

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

[VanHeerden et al., 2021a] Van Heerden, N. A., Cabral, J. B., & Luczywo, N. (2021). Evaluación de la importancia de criterios para la selección de criptomonedas. *XXXIV ENDIO - XXXII EPIO Virtual 2021*.

[VanHeerden et al., 2021b] Van Heerden, N. A., Cabral, J. B., & Luczywo, N. (2021). Evaluation of the importance of criteria for the selection of cryptocurrencies. *arXiv preprint arXiv:2109.00130*.

[Wikipedia contributors, 2021a] Wikipedia contributors (2021). *TOPSIS — Wikipedia, The Free Encyclopedia*. [Online; accessed 23-November-2021].

[Wikipedia contributors, 2021b] Wikipedia contributors (2021). *Weighted sum model — Wikipedia, The Free Encyclopedia*. [Online; accessed 23-November-2021].

[Wikipedia contributors, 2022a] Wikipedia contributors (2022). *Pareto efficiency — Wikipedia, The Free Encyclopedia*. [Online; accessed 9-October-2022].

[Wikipedia contributors, 2022b] Wikipedia contributors (2022). *Pareto front — Wikipedia, The Free Encyclopedia*. [Online; accessed 9-October-2022].

[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