Scikit-Criteria Documentation

Release 0.6

Juan BC

Feb 21, 2022

-TUTORIALS

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

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

Our Google Groups mailing list is here.

You can contact me at: jbcabral@unc.edu.ar (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]:
```
       autonomy  comfort  price
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:

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

Plotting

The Data structure support some basic routines for plotting.

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

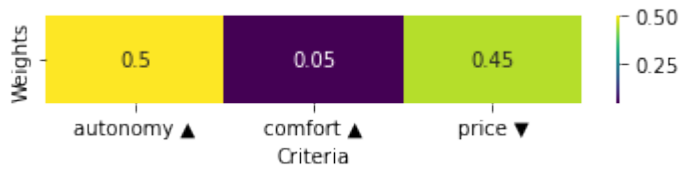
[12]:
```
dm.plot()
```

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

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

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

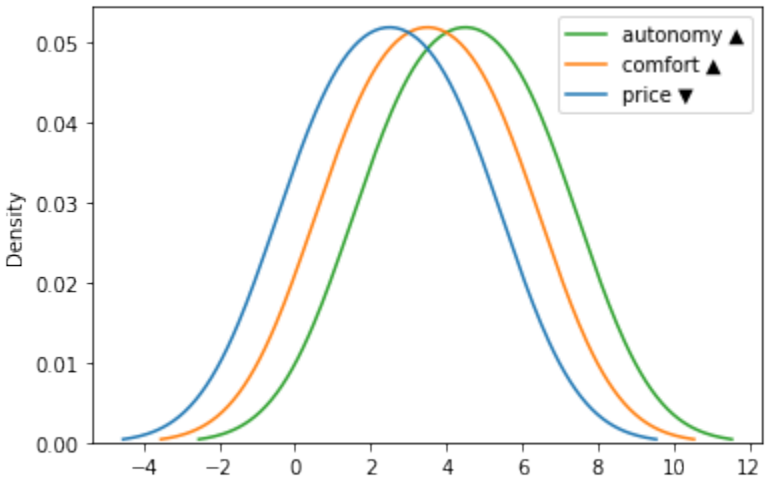
[13]:
```
<AxesSubplot:xlabel='Criteria'>
```



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")
```

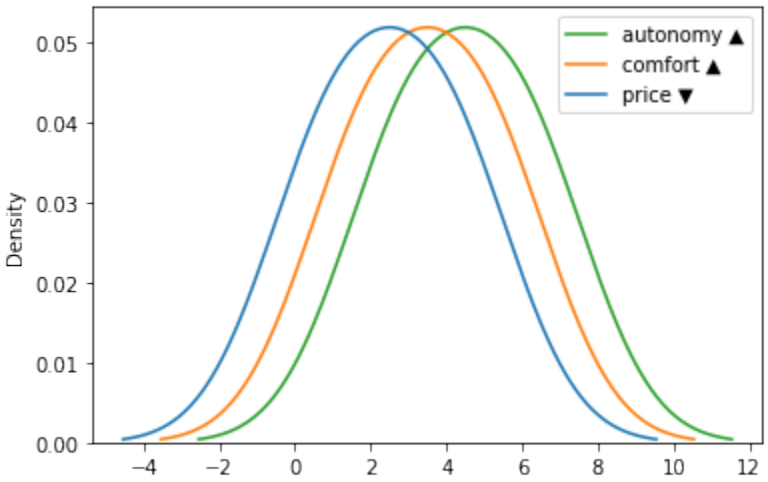
[14]:
```
<AxesSubplot:ylabel='Density'>
```



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

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

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

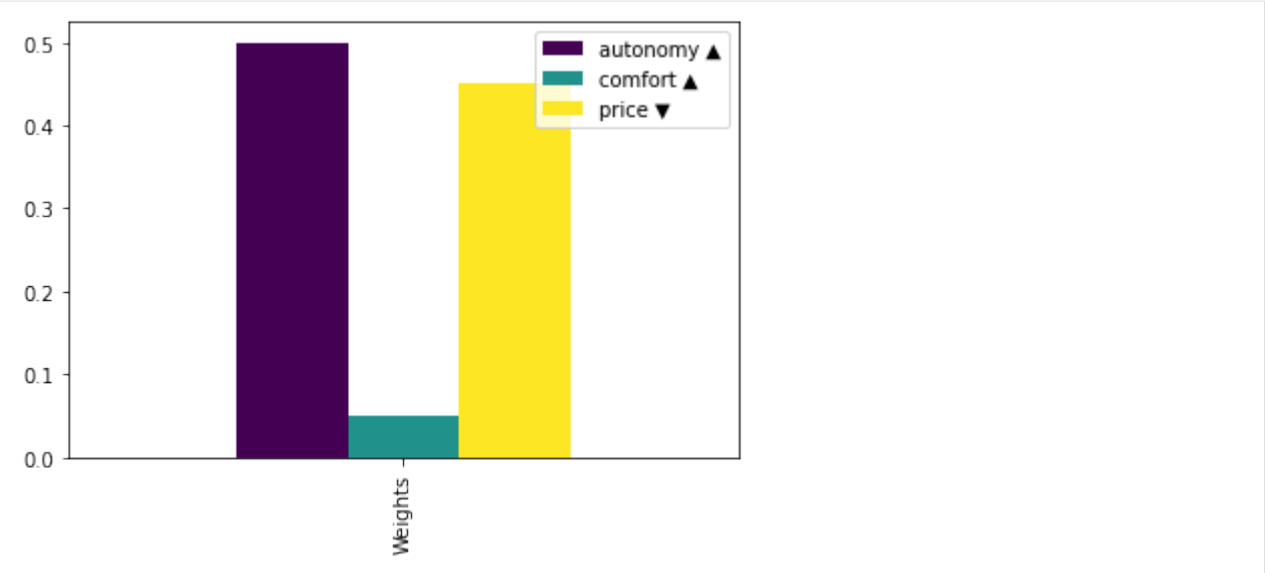


Every plot has their own set of parameters, defined by the subjacent function

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 `MinimizeToMaximize` 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.MinimizeToMaximize()
dmt = inverter.transform(dm)
dmt
```

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

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

For this step we need

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

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

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

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

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

Note: Advanced plots with Matplotlib

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

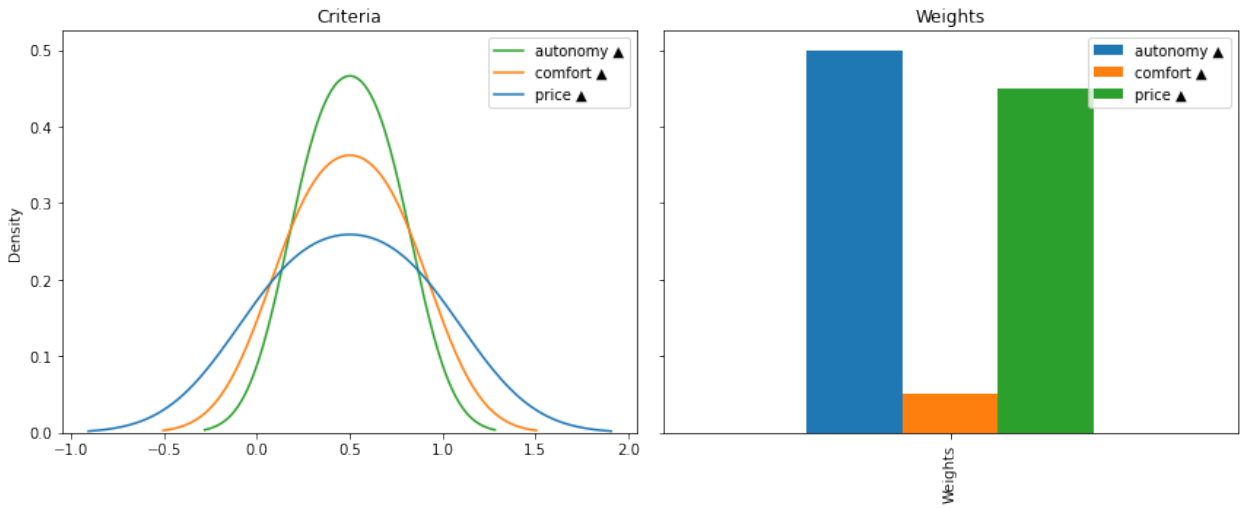
---

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

# we create 2 axis with the same y axis
fig, axs = plt.subplots(1, 2, figsize=(12, 5), sharey=True)

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


# in the second axis we plot the weights as bars
```

(continues on next page)

(continued from previous page)

```
dmt.plot.wbar(ax=axs[1])
axs[1].set_title("Weights")

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



Using this data to feed some MCDA methods

Weighted Sum Model

Let's rank our dummy data by Weighted Sum Model

First we need to import the required module

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

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

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

Note: Hyper-parameters

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

We will see an example with the *ELECTRE-1* method later on.

---

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

[22]:
```
      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*:

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

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

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

(continues on next page)

(continued from previous page)

```
)

pipe
```

```
[29]: SKCPipeline(steps=[('minimizetomaximize', MinimizeToMaximize()), ('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]:       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({'anti_ideal', 'ideal', 'similarity'})
Ideal: [0.48507125 0.04642383 0.40249224]
Anti-Ideal: [0.12126781 0.01856953 0.20124612]
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.madm import electre


      pipe_vector = mkpipe(
          invert_objectives.MinimizeToMaximize(),
          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),
      )
```

(continues on next page)

(continued from previous page)

```
pipe_sum = mkpipe(
    invert_objectives.MinimizeToMaximize(),
    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]:
```
          VW  Ford
Kernel  True  True
[Method: ELECTRE1]
```

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

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

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

---

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

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

```
Scikit-Criteria version: 0.6
Running datetime: 2022-02-21 19:53:03.925968
```

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 .

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

The filters are *transformers* and works as follows:

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

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

To write the function that filters the alternatives where $ROE >= 2.

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

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

[3]:
```
Filter(criteria_filters={'ROE': <function roe_filter at 0x7f439c9d0430>}, ignore_missing_
↪criteria=False)
```

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

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

So the filter would be defined as

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

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

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

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

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

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

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

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

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

Note:

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

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

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

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

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

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

- `filters.FilterGE`: Filter Greater or Equal than (>=).

```
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

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

[7]:
```
PE    False
JN    False
AA    False
FX     True
FN    False
dtype: bool
```

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

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

[8]:
```
PE    False
JN    False
AA    False
FX     True
FN    False
dtype: bool
```

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

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

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

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

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

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

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

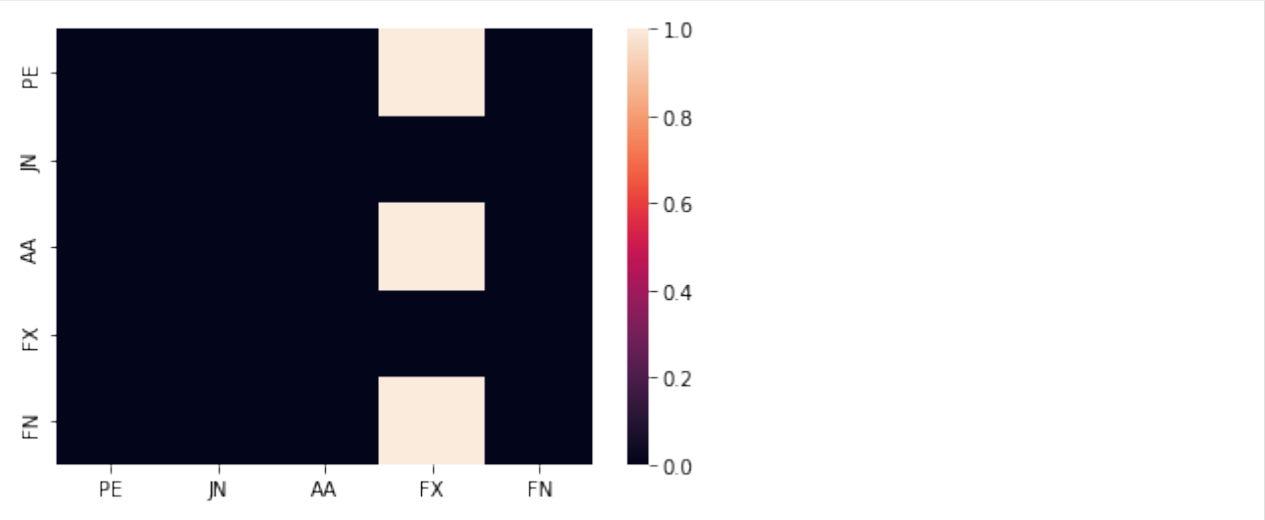
[10]:
```
       PE     JN     AA     FX     FN
PE  False  False  False   True  False
JN  False  False  False  False  False
AA  False  False  False   True  False
FX  False  False  False  False  False
FN  False  False  False   True  False
```

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 in the form of a *heatmap* using the library seaborn

[11]:
```
import seaborn as sns
sns.heatmap(dmf.dominance.dominance(strict=True))
```

[11]:
```
<AxesSubplot:>
```



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

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

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

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

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

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

Filter non-dominated alternatives

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

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

[13]:
```
FilterNonDominated(strict=True)
```

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

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

Full 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.

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

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

pipe
```

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

(continues on next page)

(continued from previous page)

We now apply the pipeline to the original data

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

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

---

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

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

```
Scikit-Criteria version: 0.6
Running datetime: 2022-02-21 19:53:14.631410
```

See also:

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

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

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

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

4.3 skcriteria package

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

4.3.1 skcriteria.core package

Core functionalities and structures of skcriteria.

skcriteria.core.data module

Data abstraction layer.

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

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

Bases: enum.Enum

Representation of criteria objectives (Minimize, Maximize).

MIN = -1

Internal representation of minimize criteria

MAX = 1
Internal representation of maximize criteria

classmethod construct_from_alias(*alias*)
Return the alias internal representation of the objective.

to_string()
Return the printable representation of the objective.

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

Representation of all data needed in the MCDA analysis.

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

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

DecisionMatrix has two main forms of construction:

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

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

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

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

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

```
>>> DecisionMatrix.from_mcda_data(
...     [[1, 2, 3], [4, 5, 6]],
...     [min, max, min],
...     [1, 1, 1])
```

(continues on next page)

(continued from previous page)

```
   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.

property criteria
Names of the criteria.

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 objetvies 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)

equals(*other*)

Return True if the decision matrix are equal.

This method calls *DecisionMatrix.aquals* whitout tolerance.

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

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

Return type bool:py:class:

See also:

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

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

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

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

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

The proceeds as follows:

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

Parameters

- **other** (skcriteria.DecisionMatrix) – Other instance to compare.
- **rtol** (*float*) – The relative tolerance parameter (see Notes in numpy.allclose()).
- **atol** (*float*) – The absolute tolerance parameter (see Notes in numpy.allclose()).
- **equal_nan** (*bool*) – Whether to compare NaN's as equal. If True, NaN's in dm will be considered equal to NaN's in *other* in the output array.

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

Return type bool:py:class:

See also:

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

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: *skcriteria.utils.accabc.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(*a*, *, *strict=False*)

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

Parameters

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

Returns List of alternatives that dominate `a`.

Return type numpy.ndarray

has_loops(*, *strict=False*)

Retorna True si la matriz contiene loops de dominacia.

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

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

Returns If True a loop exists.

Return type bool

Notes

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

skcriteria.core.methods module

Core functionalities of scikit-criteria.

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

Bases: object

Base class for all class in scikit-criteria.

Notes

All estimators should specify:

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

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

get_parameters()

Return the parameters of the method as dictionary.

copy(***kwargs*)

Return a deep copy of the current Object..

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

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

Return type A new object.

class skcriteria.core.methods.**SKCTransformerABC**

Bases: *skcriteria.core.methods.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.core.methods.**SKCMatrixAndWeightTransformerABC**(*target*)
Bases: *skcriteria.core.methods.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.core.plot module

Plot helper for the DecisionMatrix object.

class skcriteria.core.plot.**DecisionMatrixPlotter**(*dm*)
Bases: *skcriteria.utils.accabc.AccessorABC*

Make plots of DecisionMatrix.

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.

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

skcriteria.core.stats module

Stats helper for the DecisionMatrix object.

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

Bases: *skcriteria.utils.accabc.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.

4.3.2 skcriteria.madm package

MCDA methods.

skcriteria.madm._base module

Core functionalities to create madm decision-maker classes.

class skcriteria.madm._base.**SKCDecisionMakerABC**
Bases: *skcriteria.core.methods.SKCMethodABC*

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

evaluate(*dm*)
Validate the dm and calculate and evaluate the alternatives.

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

Returns Ranking.

Return type skcriteria.data.RankResult

class skcriteria.madm._base.**ResultABC**(*method*, *alternatives*, *values*, *extra*)
Bases: object

Base class to implement different types of results.

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

Parameters

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

property values
Values assigned to each alternative by the method.

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

property method
Name of the method that generated the result.

property alternatives
Names of the alternatives evaluated.

property extra_
Additional information about the result.

Note: e_ is an alias for this property

property e_
Additional information about the result.

Note: e_ is an alias for this property

property shape
Tuple with (number_of_alternatives, number_of_alternatives).

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

equals(*other*)
Check if the alternatives and ranking are the same.

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

class skcriteria.madm._base.**RankResult**(*method*, *alternatives*, *values*, *extra*)
Bases: *skcriteria.madm._base.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 rank_
Alias for values.

class skcriteria.madm._base.**KernelResult**(*method*, *alternatives*, *values*, *extra*)
Bases: *skcriteria.madm._base.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 kernelwhere_
Indexes of the alternatives that are part of the kernel.

skcriteria.madm.electre module

ELimination Et Choix Traduisant la REalité - ELECTRE.

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

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

skcriteria.madm.electre.**concordance**(*matrix*, *objectives*, *weights*)
Calculate the concordance matrix.

skcriteria.madm.electre.**discordance**(*matrix*, *objectives*)
Calculate the discordance matrix.

skcriteria.madm.electre.**electre1**(*matrix*, *objectives*, *weights*, *p=0.65*, *q=0.35*)
Execute ELECTRE1 without any validation.

class skcriteria.madm.electre.**ELECTRE1**(*p=0.65*, *q=0.35*)
Bases: *skcriteria.madm._base.SKCDecisionMakerABC*

Find a the kernel solution through ELECTRE-1.

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

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

Parameters

- **p** (*float, optional (default=0.65)*) – Concordance threshold. Threshold of how much one alternative is at least as good as another to be significative.
- **q** (*float, optional (default=0.35)*) – Discordance threshold. Threshold of how much the degree one alternative is strictly preferred to another to be significative.

References

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

property p
Concordance threshold.

property q
Discordance threshold.

skcriteria.madm.moora module

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

skcriteria.madm.moora.**ratio**(*matrix*, *objectives*, *weights*)
Execute ratio MOORA without any validation.

class skcriteria.madm.moora.**RatioMOORA**
Bases: *skcriteria.madm._base.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.madm.moora.**refpoint**(*matrix*, *objectives*, *weights*)
Execute reference point MOORA without any validation.

class skcriteria.madm.moora.**ReferencePointMOORA**
Bases: *skcriteria.madm._base.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.madm.moora.**fmf**(*matrix*, *objectives*, *weights*)
Execute Full Multiplicative Form without any validation.

class skcriteria.madm.moora.**FullMultiplicativeForm**
Bases: *skcriteria.madm._base.SKCDecisionMakerABC*

Non-linear, non-additive ranking method method.

Full Multiplicative Form does not use weights and does not require normalization.

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

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

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

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

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

Notes

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

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

References

[Brauers & Zavadskas, 2012]

skcriteria.madm.moora.**multimoora**(*matrix*, *objectives*, *weights*)
Execute weighted product model without any validation.

class skcriteria.madm.moora.**MultiMOORA**
Bases: *skcriteria.madm._base.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.madm.similarity module

Methods based on a similarity between alternatives.

skcriteria.madm.similarity.**topsis**(*matrix*, *objectives*, *weights*, *metric='euclidean'*, ***kwargs*)
Execute TOPSIS without any validation.

class skcriteria.madm.similarity.**TOPSIS**(***, *metric='euclidean'*)
Bases: *skcriteria.madm._base.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.madm.simple module

Some simple and compensatory methods.

skcriteria.madm.simple.**wsm**(*matrix*, *weights*)
Execute weighted sum model without any validation.

class skcriteria.madm.simple.**WeightedSumModel**
Bases: *skcriteria.madm._base.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.madm.simple.`**`wpm`**(*matrix*, *weights*)
Execute weighted product model without any validation.

class `skcriteria.madm.simple.`**`WeightedProductModel`**
Bases: *skcriteria.madm._base.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.madm.simus module

SIMUS (Sequential Interactive Model for Urban Systems) Method.

skcriteria.madm.simus.**simus**(*matrix*, *objectives*, *b=None*, *rank_by=1*, *solver='pulp'*)
Execute SIMUS without any validation.

class skcriteria.madm.simus.**SIMUS**(*, *rank_by=1*, *solver='pulp'*)
Bases: *skcriteria.madm._base.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.distance module

Normalization through the distance to distance function.

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

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

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

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

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

class skcriteria.preprocessing.distance.**CenitDistance**

Bases: *skcriteria.core.methods.SKCTransformerABC*

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

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

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

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

References

[Diakoulaki et al., 1995]

skcriteria.preprocessing.filters module

Normalization through the distance to distance function.

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

Bases: *skcriteria.core.methods.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: *skcriteria.preprocessing.filters.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],
```

(continues on next page)

(continued from previous page)

```
...         [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: *skcriteria.preprocessing.filters.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: *skcriteria.preprocessing.filters.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: *skcriteria.preprocessing.filters.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: *skcriteria.preprocessing.filters.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],
```

(continues on next page)

(continued from previous page)

```
...         [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: *skcriteria.preprocessing.filters.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: *skcriteria.preprocessing.filters.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: *skcriteria.preprocessing.filters.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],
...     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: *skcriteria.preprocessing.filters.SKCByCriteriaFilterABC*

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

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 comparision 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: *skcriteria.preprocessing.filters.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],
...     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: *skcriteria.preprocessing.filters.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],
```

(continues on next page)

(continued from previous page)

```
...     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: *skcriteria.core.methods.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.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: *skcriteria.core.methods.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 inverting minimization criteria and converting them into maximization ones.

In addition to the main functionality, an agnostic MCDA function is offered that inverts columns of a matrix based on a mask.

skcriteria.preprocessing.invert_objectives.**invert**(*matrix*, *mask*)

Inverts all the columns selected by the mask.

Parameters

- **matrix** (numpy.ndarray like.) – 2D array.
- **mask** (numpy.ndarray like.) – Boolean array like with the same elements as columns has the matrix.

Returns New matrix with the selected columns inverted. The result matrix dtype float.

Return type numpy.ndarray

Examples

```
>>> from skcriteria import invert
>>> invert([
...     [1, 2, 3],
...     [4, 5, 6]
... ],
... [True, False, True])
array([[1.        , 2.        , 0.33333333],
       [0.25      , 5.        , 0.16666667]])

>>> invert([
...     [1, 2, 3],
...     [4, 5, 6]
... ],
... [False, True, False])
array([[1.        , 2.        , 0.33333333],
       [0.25      , 5.        , 0.16666667]])
array([[1. , 0.5, 3. ],
       [4. , 0.2, 6. ]]
```

class skcriteria.preprocessing.invert_objectives.**MinimizeToMaximize**

Bases: *skcriteria.core.methods.SKCTransformerABC*

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`.

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]])

# 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: *skcriteria.core.methods.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 differrent strategies.

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

skcriteria.preprocessing.scalers.**scale_by_stdscore**(*arr*, *axis=None*)

Standardize the values by removing the mean and divided by the std-dev.

The standard score of a sample x is calculated as:

$$z = (x - \mu)/\sigma$$

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_stdscore
>>> mtx = [[1, 2], [3, 4]]

# ratios with the max value of the array
>>> scale_by_stdscore(mtx)
array([[-1.34164079, -0.4472136 ],
       [ 0.4472136 ,  1.34164079]])

# ratios with the max value of the arr by column
>>> scale_by_stdscore(mtx, axis=0)
array([[-1., -1.],
       [ 1.,  1.]])

# ratios with the max value of the array by row
>>> scale_by_stdscore(mtx, axis=1)
array([[-1.,  1.],
       [-1.,  1.]])
```

class skcriteria.preprocessing.scalers.**StandarScaler**(*target*)

Bases: *skcriteria.core.methods.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*.

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: *skcriteria.core.methods.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_minmax**(*arr*, *axis=None*)

Fraction of the range normalizer.

Subtracts to each value of the array the minimum and then divides it by the total range.

$$\overline{X}_{ij} = \frac{X_{ij} - \min X_{ij}}{\max_{X_{ij}} - \min_{X_{ij}}}$$

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_minmax
>>> mtx = [[1, 2], [3, 4]]

# ratios with the range of the array
>>> scale_by_minmax(mtx)
array([[0.        , 0.33333333],
       [0.66666667, 1.        ]])

# ratios with the range by column
>>> scale_by_minmax(mtx, axis=0)
array([[0., 0.],
       [1., 1.]])

# ratios with the range by row
>>> scale_by_minmax(mtx, axis=1)
array([[0., 1.],
      [0., 1.]])
```

class skcriteria.preprocessing.scalers.**MinMaxScaler**(*target*)

Bases: *skcriteria.core.methods.SKCMatrixAndWeightTransformerABC*

Scaler based on the range.

$$\overline{X}_{ij} = \frac{X_{ij} - \min X_{ij}}{\max_{X_{ij}} - \min_{X_{ij}}}$$

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.

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: *skcriteria.core.methods.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.**scale_by_max**(*arr*, *axis=None*)

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

$$\overline{X}_{ij} = \frac{X_{ij}}{\max_{X_{ij}}}$$

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_max
>>> mtx = [[1, 2], [3, 4]]

# ratios with the max value of the array
>>> scale_by_max(mtx)
array([[ 0.25,  0.5 ],
       [ 0.75,  1.  ]])

# ratios with the max value of the arr by column
>>> scale_by_max(mtx, axis=0)
array([[ 0.33333334,  0.5],
       [ 1.        ,  1. ]])

# ratios with the max value of the array by row
>>> scale_by_max(mtx, axis=1)
array([[ 0.5 ,  1.],
       [ 0.75,  1.]])
```

class skcriteria.preprocessing.scalers.**MaxScaler**(*target*)

Bases: *skcriteria.core.methods.SKCMatrixAndWeightTransformerABC*

Scaler based on the maximum values.

$$\overline{X}_{ij} = \frac{X_{ij}}{\max_{X_{ij}}}$$

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.

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: *skcriteria.core.methods.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: *skcriteria.preprocessing.weighters.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.

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

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

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

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: *skcriteria.preprocessing.weighters.SKCWeighterABC*

Set as weight the normalized standard deviation of each criterion.

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

Calculate the weights as 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.

This routine will normalize the criteria if they don't sum to 1.

See also:

scipy.stats.entropy Calculate the entropy of a distribution for given probability values.

Examples

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

```
>>> entropy_weights(mtx)
array([0.46906241, 0.53093759])
```

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

Bases: *skcriteria.preprocessing.weighters.SKCWeighterABC*

Assigns 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.

This transformer will normalize the criteria if they don't sum 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`.

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`.

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=<function pearson_correlation>*, *scale=True*)

Execute the CRITIC method without any validation.

class skcriteria.preprocessing.weighters.**Critic**(*correlation='pearson'*, *scale=True*)

Bases: *skcriteria.preprocessing.weighters.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" or "spearman"] or callable. (default "pearson")*) – 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 `kendall` correlation is also available implemented. It is also possible to provide a function that receives as a single parameter, the matrix of alternatives, and returns the correlation matrix.
- **scale** (bool (default `True`)) – True if it is necessary to scale the data with `skcriteria.preprocesisng.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': <function pearson_correlation>, 'spearman': <function spearman_correlation>}

property scale

Return if it is necessary to scale the data.

property correlation

Correlation function.

4.3.4 skcriteria.pipeline module

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

class skcriteria.pipeline.**SKCPipeline**(*steps*)

Bases: *skcriteria.core.methods.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. A step's estimator may be replaced entirely by setting the parameter with its name to another dmaker or a transformer removed by setting it to *'passthrough'* or *None*.

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-learn *SKCPipeline* object.

Return type *SKCPipeline*

4.3.5 skcriteria.utils package

Utilities for skcriteria.

skcriteria.utils.accabc module

Accessor base class.

class skcriteria.utils.accabc.**AccessorABC**
Bases: abc.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: collections.abc.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.decorators module

Multiple decorator to use inside scikit-criteria.

skcriteria.utils.decorators.**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*>__

exception skcriteria.utils.decorators.**SKCriteriaDeprecationWarning**
Bases: DeprecationWarning

Skcriteria deprecation warning.

skcriteria.utils.decorators.**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.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: skcriteria.utils.lp._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: skcriteria.utils.lp._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: skcriteria.utils.lp._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: skcriteria.utils.lp._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: skcriteria.utils.lp._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.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

4.4 Changelog

4.4.1 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.

4.4.2 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

4.4.3 Version 0.2

First OO stable version.

4.4.4 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.

[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.

[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*.

[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.

[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.

[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].

PYTHON MODULE INDEX

S

INDEX

F

G

H

I

K

M

N

O

P

Q

R

S

T

V

W