## CONTENTS

1 Contents .......................................................... 3
  1.1 Usage ...................................................... 3
  1.2 Illustrative examples .................................. 20
  1.3 API Reference ........................................... 42

**Python Module Index** ........................................... 75

**Index** .......................................................... 77
pyrepo-mcda is Python 3 library for Multi-Criteria Decision Analysis. This library includes:

- **MCDA methods:**
  - TOPSIS
  - CODAS
  - MABAC
  - MULTIMOORA
  - MOORA
  - VIKOR
  - WASPAS
  - EDAS
  - SPOTIS

- **Distance metrics:**
  - euclidean (Euclidean distance)
  - manhattan (Manhattan distance)
  - hausdorff (Hausdorff distance)
  - correlation (Correlation distance)
  - chebyshev (Chebyshev distance)
  - std_euclidean (Standardized Euclidean distance)
  - cosine (Cosine distance)
  - csm (Cosine similarity measure)
  - squared_euclidean (Squared Euclidean distance)
  - bray_curtis (Sorensen or Bray-Curtis distance)
  - canberra (Canberra distance)
  - lorentzian (Lorentzian distance)
  - jaccard (Jaccard distance)
  - dice (Dice distance)
  - bhattacharyya (Bhattacharyya distance)
  - hellinger (Hellinger distance)
  - matusita (Matusita distance)
  - squared_chord (Squared-chord distance)
  - pearson_chi_square (Pearson chi square distance)
  - squared_chi_square (Squared chi square distance)

- **Correlation coefficients:**
  - spearman (Spearman rank correlation coefficient)
  - weighted_spearman (Weighted Spearman rank correlation coefficient)
  - pearson_coeff (Pearson correlation coefficient)
- **WS_coeff** (Similarity rank coefficient - WS coefficient)

**Methods for normalization of decision matrix:**
- `linear_normalization` (Linear normalization)
- `minmax_normalization` (Minimum-Maximum normalization)
- `max_normalization` (Maximum normalization)
- `sum_normalization` (Sum normalization)
- `vector_normalization` (Vector normalization)
- `multimoora_normalization` (Normalization method dedicated for the MULTIMOORA method)

**Methods for determination of criteria weights (weighting methods):**
- `entropy_weighting` (Entropy weighting method)
- `std_weighting` (Standard Deviation weighting method)
- `critic_weighting` (CRITIC weighting method)

**Methods for determination of compromise rankings based on several rankings obtained with different MCDA methods:**
- `copeland` (the Copeland method for compromise ranking)
- `dominance_directed_graph` (Dominance Directed Graph for compromise ranking)
- `rank_position_method` (Rank Position Method for compromise ranking)
- `improved_borda_rule` (Improved Borda Rule method for compromise for MULTIMOORA method)

**Methods for sensitivity analysis:**
- `Sensitivity_analysis_weights_percentages` (Method for sensitivity analysis considering percentage modification of criteria weights)
- `Sensitivity_analysis_weights_values` (Method for sensitivity analysis considering setting different values as chosen criterion weight)

**Additions:**
- `rank_preferences` (Method for ordering alternatives according to their preference values obtained with MCDA methods)

Check out the *Usage* section for further information, including how to *Installation* the project.

**Note:** This project is under active development.
1.1 Usage

1.1.1 Installation

To use pyrepo-mcda, first install it using pip:

```
    pip install pyrepo-mcda
```

1.1.2 Usage examples

The TOPSIS method

The TOPSIS method is used to calculate the preference of evaluated alternatives. When creating the object of the TOPSIS method, you have to provide `normalization_method` (it is `minmax_normalization` by default) and `distance_metric` (it is `euclidean` by default). The TOPSIS method requires providing the decision matrix `matrix`, vector with criteria weights `weights`, and vector with criteria types `types`. The TOPSIS method returns a vector with preference values `pref`. To generate the TOPSIS ranking of alternatives, `pref` has to be sorted in descending order. The ranking is generated by `rank_preferences`, providing `pref` as argument and setting parameter `reverse` as `True` because we need to sort preferences descendingly.

```python
import numpy as np
from pyrepo_mcda.mcda_methods import TOPSIS
from pyrepo_mcda import normalizations as norms
from pyrepo_mcda import distance_metrics as dists
from pyrepo_mcda.additions import rank_preferences

# provide decision matrix in array numpy.darray
matrix = np.array([[256, 8, 41, 1.6, 1.77, 7347.16],
                   [256, 8, 32, 1.0, 1.8, 6919.99],
                   [256, 8, 53, 1.6, 1.9, 8400],
                   [256, 8, 41, 1.0, 1.75, 6808.9],
                   [512, 8, 35, 1.6, 1.7, 8479.99],
                   [256, 4, 35, 1.6, 1.7, 7499.99]])

# provide criteria weights in array numpy.darray. All weights must sum to 1.
weights = np.array([0.405, 0.221, 0.134, 0.199, 0.007, 0.034])

# provide criteria types in array numpy.darray. Profit criteria are represented by 1 and cost criteria by -1.
```

(continues on next page)
types = np.array([1, 1, 1, 1, -1, -1])

# Create the TOPSIS method object providing normalization method and distance metric.
tOPSIS = TOPSIS(normalization_method = norms.minmax_normalization, distance_metric =
               dists.euclidean)

# Calculate the TOPSIS preference values of alternatives
pref = topsis(matrix, weights, types)

# Generate ranking of alternatives by sorting alternatives descendingly according to the
# TOPSIS algorithm (reverse = True means sorting in descending order) according to
# preference values
rank = rank_preferences(pref, reverse = True)

print('Preference values: ', np.round(pref, 4))
print('Ranking: ', rank)

Output
Preference values: [0.4242 0.3217 0.4453 0.3353 0.8076 0.2971]
Ranking: [3 5 2 4 1 6]

The VIKOR method

The VIKOR method is used to calculate the preference of evaluated alternatives. When creating the object of the VIKOR method, you have to provide `normalization_method` (it is `None` by default) and `v` parameter. The VIKOR method requires providing the decision matrix `matrix`, vector with criteria weights `weights`, and vector with criteria types `types`. The VIKOR method returns a vector with preference values `pref`. To generate the VIKOR ranking of alternatives, `pref` has to be sorted in ascending order. The ranking is generated by `rank_preferences`, providing `pref` as argument and setting parameter `reverse` as `False` because we need to sort preferences ascendingly.

```python
import numpy as np
from pyrepo_mcda.mcda_methods import VIKOR
from pyrepo_mcda.additions import rank_preferences

# provide decision matrix in array numpy.darray
matrix = np.array([[8, 7, 2, 1],
                   [5, 3, 7, 5],
                   [7, 5, 6, 4],
                   [9, 9, 7, 3],
                   [11, 10, 3, 7],
                   [6, 9, 5, 4]])

# provide criteria weights in array numpy.darray. All weights must sum to 1.
weights = np.array([0.4, 0.3, 0.1, 0.2])

# provide criteria types in array numpy.darray. Profit criteria are represented by 1 and
# cost criteria by -1.
types = np.array([1, 1, 1, 1])

# Create the VIKOR method object providing chosen normalization method `normalization_`
# method` (if you don't want to use normalization set `normalization_method` as None, it
# is default), and
```
The default v parameter is set to 0.5, so if you do not provide it, v will be equal to 0.5.

```python
vikor = VIKOR(normalization_method = None, v = 0.625)
```

# Calculate the VIKOR preference values of alternatives
```
pref = vikor(matrix, weights, types)
```

# Generate ranking of alternatives by sorting alternatives ascendingly according to the VIKOR algorithm (reverse = False means sorting in ascending order) according to preference values
```
rank = rank_preferences(pref, reverse = False)
```

```python
print('Preference values: ', np.round(pref, 4))
print('Ranking: ', rank)
```

Output

```
Preference values: [0.6399 1. 0.6929 0.2714 0. 0.6939]
Ranking: [3 6 4 2 1 5]
```

**The SPOTIS method**

The SPOTIS method is used to calculate the preference of evaluated alternatives. The SPOTIS method requires providing the decision matrix `matrix`, vector with criteria weights `weights`, and vector with criteria types `types` and minimum and maximum bounds of alternatives performance values for particular criteria. The SPOTIS method returns a vector with preference values `pref`. To generate the SPOTIS ranking of alternatives, `pref` has to be sorted in ascending order. The ranking is generated by `rank_preferences`, providing `pref` as argument and setting parameter `reverse` as False because we need to sort preferences ascendingly.

```python
import numpy as np
from pyrepo_mcda.mcda_methods import SPOTIS
from pyrepo_mcda.additions import rank_preferences

# provide decision matrix in array numpy.darray
matrix = np.array([[15000, 4.3, 99, 42, 737],
                   [15290, 5.0, 116, 42, 892],
                   [15350, 5.0, 114, 45, 952],
                   [15490, 5.3, 123, 45, 1120]])

# provide criteria weights in array numpy.darray. All weights must sum to 1.
weights = np.array([0.2941, 0.2353, 0.2353, 0.0588, 0.1765])

# provide criteria types in array numpy.darray. Profit criteria are represented by 1 and cost criteria by -1.
types = np.array([-1, -1, -1, 1, 1])

# Determine minimum bounds of performance values for each criterion in decision matrix
bounds_min = np.array([14000, 3, 80, 35, 650])

# Determine maximum bounds of performance values for each criterion in decision matrix
bounds_max = np.array([16000, 8, 140, 60, 1300])
```

1.1. Usage
# Stack minimum and maximum bounds vertically using vstack. You will get a matrix that
# has two rows and a number of columns equal to the number of criteria
bounds = np.vstack((bounds_min, bounds_max))

# Create the SPOTIS method object
spotis = SPOTIS()

# Calculate the SPOTIS preference values of alternatives
pref = spotis(matrix, weights, types, bounds)

# Generate ranking of alternatives by sorting alternatives ascendingly according to the
# SPOTIS algorithm (reverse = False means sorting in ascending order) according to
# preference values
rank = rank_preferences(pref, reverse = False)

print('Preference values: ', np.round(pref, 4))
print('Ranking: ', rank)

Output
Preference values: [0.4778 0.5781 0.5557 0.5801]
Ranking: [1 3 2 4]

The CODAS method

The CODAS method is used to calculate the preference of evaluated alternatives. When creating the object of the CODAS method, you have to provide normalization_method (it is linear_normalization by default) and distance_metric (it is euclidean by default). The CODAS method requires providing the decision matrix matrix, vector with criteria weights weights, and vector with criteria types types. The CODAS method returns a vector with preference values pref. To generate the CODAS ranking of alternatives, pref has to be sorted in descending order. The ranking is generated by rank_preferences method, providing pref as argument and setting parameter reverse as True because we need to sort preferences descendingly.

```python
import numpy as np
from pyrepo_mcda.mcda_methods import CODAS
from pyrepo_mcda import normalizations as norms
from pyrepo_mcda import distance_metrics as dists
from pyrepo_mcda.additions import rank_preferences

# provide decision matrix in array numpy.darray
matrix = np.array([[45, 3600, 45, 0.9],
[25, 3800, 60, 0.8],
[23, 3100, 35, 0.9],
[14, 3400, 50, 0.7],
[15, 3300, 40, 0.8],
[28, 3000, 30, 0.6]])

# provide criteria weights in array numpy.darray. All weights must sum to 1.
weights = np.array([0.2857, 0.3036, 0.2321, 0.1786])
```

(continues on next page)
# provide criteria types in array numpy.darray. Profit criteria are represented by 1 and cost criteria by -1.
types = np.array([1, -1, 1, 1])

# Create the CODAS method object providing normalization method (in CODAS it is ``linear_normalization`` by default), distance metric, and tau parameter, which is equal to 0.02. tau must be in the range from 0.01 to 0.05.
codas = CODAS(normalization_method = norms.linear_normalization, distance_metric = dists.euclidean, tau = 0.02)

# Calculate the CODAS preference values of alternatives
pref = codas(matrix, weights, types)

# Generate ranking of alternatives by sorting alternatives descendingly according to the CODAS algorithm (reverse = True means sorting in descending order) according to preference values
rank = rank_preferences(pref, reverse = True)

print('Preference values: ', np.round(pref, 4))
print('Ranking: ', rank)

Output

```
Preference values:  [ 1.3914  0.3411 -0.217  -0.5381  -0.7292  -0.2481]
Ranking:  [1 2 3 5 6 4]
```

## The WASPAS method

The WASPAS method is used to calculate the preference of evaluated alternatives. When creating the object of the WASPAS method, you have to provide `normalization_method` (it is `linear_normalization` by default) and `lambda_param` (it is equal to 0.5 by default). The WASPAS method requires providing the decision matrix `matrix`, vector with criteria weights `weights`, and vector with criteria types `types`. The WASPAS method returns a vector with preference values `pref`. To generate the WASPAS ranking of alternatives, `pref` has to be sorted in descending order. The ranking is generated by `rank_preferences` method, providing `pref` as argument and setting parameter `reverse` as `True` because we need to sort preferences descendingly.

```python
import numpy as np
from pyrepo_mcda.mcda_methods import WASPAS
from pyrepo_mcda import normalizations as norms
from pyrepo_mcda.additions import rank_preferences

# provide decision matrix in array numpy.darray
matrix = np.array([[5000, 3, 3, 4, 3, 2],
                   [680, 5, 3, 2, 2, 1],
                   [2000, 3, 2, 3, 4, 3],
                   [600, 4, 3, 1, 2, 2],
                   [800, 2, 4, 3, 3, 4]])

# provide criteria weights in array numpy.darray. All weights must sum to 1.
weights = np.array([0.157, 0.249, 0.168, 0.121, 0.154, 0.151])
```
# provide criteria types in array numpy.darray. Profit criteria are represented by 1 and, cost criteria by -1.
types = np.array([-1, 1, 1, 1, 1, 1])

# Create the WASPAS method object providing normalization method (in WASAPS it is linear normalization by default), and lambda parameter, which is equal to 0.5 default. tau must be in the range from 0 to 1.
waspas = WASPAS(normalization_method=norms.linear_normalization, lambda_param=0.5)

# Calculate the WASPAS preference values of alternatives
pref = waspas(matrix, weights, types)

# Generate ranking of alternatives by sorting alternatives descendingly according to the WASPAS algorithm (reverse = True means sorting in descending order) according to preference values
rank = rank_preferences(pref, reverse = True)

print('Preference values: ', np.round(pref, 4))
print('Ranking: ', rank)

Output
Preference values: [0.5622 0.6575 0.6192 0.6409 0.7228]
Ranking: [5 2 4 3 1]

The EDAS method

The EDAS method is used to calculate the preference of evaluated alternatives. The EDAS method requires providing the decision matrix `matrix`, vector with criteria weights `weights`, and vector with criteria types `types`. The EDAS method returns a vector with preference values `pref`. To generate the EDAS ranking of alternatives, `pref` has to be sorted in descending order. The ranking is generated by `rank_preferences` method, providing `pref` as argument and setting parameter `reverse` as `True` because we need to sort preferences descendingly.

```python
import numpy as np
from pyrepo_mcda.mcda_methods import EDAS
from pyrepo_mcda.additions import rank_preferences

# provide decision matrix in array numpy.darray
matrix = np.array([[256, 8, 41, 1.6, 1.77, 7347.16],
                   [256, 8, 32, 1.0, 1.8, 6919.99],
                   [256, 8, 53, 1.6, 1.9, 8400],
                   [256, 8, 41, 1.0, 1.75, 6808],
                   [256, 4, 35, 1.6, 1.7, 8479.99],
                   [256, 4, 35, 1.6, 1.7, 7499.99]])

# provide criteria weights in array numpy.darray. All weights must sum to 1.
weights = np.array([0.405, 0.221, 0.134, 0.199, 0.007, 0.034])

# provide criteria types in array numpy.darray. Profit criteria are represented by 1 and, cost criteria by -1.
types = np.array([1, 1, 1, -1, -1])
```
# Create the EDAS method object.
edas = EDAS()

# Calculate the EDAS preference values of alternatives
pref = edas(matrix, weights, types)

# Generate ranking of alternatives by sorting alternatives descendingly according to the EDAS algorithm (reverse = True means sorting in descending order) according to preference values
rank = rank_preferences(pref, reverse = True)

print('Preference values: ', np.round(pref, 4))
print('Ranking: ', rank)

Output

Preference values: [ 0.4141  0.1300  0.4607  0.2120  0.9443  0.0430 ]
Ranking: [3 5 2 4 1 6]

The MABAC method

The MABAC method is used to calculate the preference of evaluated alternatives. When creating the object of the MABAC method, you have to provide normalization_method (it is minmax_normalization by default). The MABAC method requires providing the decision matrix matrix, vector with criteria weights weights, and vector with criteria types types. The WASPAS method returns a vector with preference values pref. To generate the MABAC ranking of alternatives, pref has to be sorted in descending order. The ranking is generated by rank_preferences method, providing pref as argument and setting parameter reverse as True because we need to sort preferences descendingly.

```python
import numpy as np
from pyrepo_mcda.mcda_methods import MABAC
from pyrepo_mcda import normalizations as norms
from pyrepo_mcda.additions import rank_preferences

# provide decision matrix in array numpy.darray
matrix = np.array([[2.937588, 2.762986, 3.233723, 2.881315, 3.015289, 3.313491],
                   [2.978555, 3.012820, 2.929487, 3.096154, 3.012820, 3.593939],
                   [3.286673, 3.464600, 3.746009, 3.715632, 3.703427, 4.133620],
                   [3.322037, 3.098638, 3.262154, 3.147851, 3.206675, 3.798684],
                   [3.354866, 3.270945, 3.221880, 3.213207, 3.670508, 3.785941],
                   [2.796570, 2.983000, 2.744904, 2.692550, 2.787563, 2.878851],
                   [2.846491, 2.729618, 2.789990, 2.955624, 3.123323, 3.646595],
                   [3.253458, 3.208902, 3.678499, 3.580044, 3.505663, 3.954262],
                   [2.580718, 2.906903, 3.176497, 3.073653, 3.264727, 3.681887],
                   [2.789011, 3.000000, 3.101099, 3.139194, 2.985348, 3.139194],
                   [3.418681, 3.261905, 3.187912, 3.052381, 3.266667, 3.695238]])

# provide criteria weights in array numpy.darray. All weights must sum to 1.
weights = np.array([0.171761, 0.105975, 0.191793, 0.168824, 0.161768, 0.199880])
```
# provide criteria types in array numpy.darray. Profit criteria are represented by 1 and cost criteria by -1.
types = np.array([1, 1, 1, 1, 1])

# Create the MABAC method object providing normalization method. In MABAC it is minmax normalization by default.
mabac = MABAC(normalization_method=norms.minmax_normalization)

# Calculate the MABAC preference values of alternatives
pref = mabac(matrix, weights, types)

# Generate ranking of alternatives by sorting alternatives descendingly according to the MABAC algorithm (reverse = True means sorting in descending order) according to preference values
rank = rank_preferences(pref, reverse = True)

print('Preference values: ', np.round(pref, 4))
print('Ranking: ', rank)

Output

Preference values: [-0.1553 -0.0895  0.5054  0.1324  0.2469 -0.3868 -0.1794  0.3629 -0.
  -0.0842 -0.1675  0.1399]
Ranking: [ 8  7  1  5  3 11 10  2  6  9  4]

The MULTIMOORA method

The MULTIMOORA method is used to calculate ranking of alternatives. When creating the object of the MULTIMOORA method, you have to provide compromise_rank_method (it is dominance_directed_graph by default) because the MULTIMOORA creates ranking based on three subordinate rankings generated by three approaches: Ratio System (RS), Reference Point (RP) and Full Multiplicative Form (FMF). The MULTIMOORA method requires providing the decision matrix matrix, vector with criteria weights weights, and vector with criteria types types. The MULTIMOORA method returns a vector with ranking rank.

```python
import numpy as np
from pyrepo_mcda.mcda_methods import MULTIMOORA
from pyrepo_mcda.additions import rank_preferences
from pyrepo_mcda import compromise_rankings as compromises

# provide decision matrix in array numpy.darray
matrix = np.array([[[4, 3, 3, 4, 3, 2, 4],
            [3, 3, 4, 3, 5, 4, 4],
            [5, 4, 4, 5, 5, 5, 4]]])

# provide criteria weights in array numpy.darray. All weights must sum to 1.
weights = np.array([0.215, 0.215, 0.159, 0.133, 0.102, 0.102, 0.073])

# provide criteria types in array numpy.darray. Profit criteria are represented by 1 and cost criteria by -1.
types = np.array([1, 1, 1, 1, 1])
```
# Create the MULTIMOORA method object providing compromise_rank_method. In MULTIMOORA it is dominance_directed_graph by default.
```
multimoora = MULTIMOORA(compromise_rank_method = compromises.dominance_directed_graph)
```

# Calculate the MULTIMOORA ranking of alternatives
```
rank = multimoora(matrix, weights, types)
```

print('Ranking: ', rank)

Output
```
Ranking: [3 2 1]
```

The MOORA method

The MOORA method is used to obtain preference values of alternatives. Then alternatives have to be sorted according to preference values in descending order. The MOORA method can be applied using MULTIMOORA_RS from multimoora. This method requires providing decision matrix matrix, vector with criteria weights weights (all weights must sum to 1) and vector with criteria types types which are equal to 1 for profit criteria and -1 for cost criteria.

```
import numpy as np
from pyrepo_mcda.mcda_methods import MULTIMOORA_RS as MOORA

matrix = np.array([[4, 3, 3, 4, 3, 2, 4],
                    [3, 3, 4, 3, 5, 4, 4],
                    [5, 4, 4, 5, 5, 5, 4]])

weights = np.array([0.215, 0.215, 0.159, 0.133, 0.102, 0.102, 0.073])

weights = np.array([1, 1, 1, 1, 1, 1, 1])

moora = MOORA()

pref = moora(matrix, weights, types)

rank = rank_preferences(pref, reverse = True)

print('Preference values: ', np.round(pref, 4))
print('Ranking: ', rank)
```

Output
```
Preference values: [0.241 0.1702 0.1431 0.1068 0.1027 0.13 ]
Ranking: [1 2 3 5 6 4]
```
Methods for determining compromise rankings

The Copeland Method for compromise ranking

This method is used to generate compromise ranking based on several rankings provided by different MCDA methods. The copeland method requires providing two-dimensional matrix matrix with different rankings in particular columns. copeland returns vector with compromise ranking.

```python
import numpy as np
from pyrepo_mcda import compromise_rankings as compromises

# Provide matrix with different rankings given by different MCDA methods in columns
matrix = np.array([[7, 8, 7, 6, 7, 7],
                   [4, 7, 5, 7, 5, 4],
                   [8, 9, 8, 9, 8, 9],
                   [1, 4, 1, 1, 1, 1],
                   [2, 2, 4, 3, 2],
                   [3, 1, 4, 3, 2, 3],
                   [10, 5, 10, 9, 8, 10],
                   [6, 3, 6, 4, 6],
                   [9, 10, 9, 10, 10, 9],
                   [5, 6, 3, 2, 6, 5]])

# Calculate the compromise ranking using `copeland` method
result = compromises.copeland(matrix)

print('Copeland compromise ranking: ', result)
```

Output

Copeland compromise ranking: [7 6 8 1 2 3 9 5 10 4]

The Dominance Directed Graph

This method is used to generate compromise ranking based on several rankings provided by different MCDA methods. The dominance_directed_graph method requires providing two-dimensional matrix matrix with different rankings in particular columns. dominance_directed_graph returns vector with compromise ranking.

```python
import numpy as np
from pyrepo_mcda import compromise_rankings as compromises

# Provide matrix with different rankings given by different MCDA methods in columns
matrix = np.array([[3, 2, 3],
                   [2, 3, 2],
                   [1, 1, 1]])

# Calculate the compromise ranking using `dominance_directed_graph` method
result = compromises.dominance_directed_graph(matrix)

print('Dominance directed graph compromise ranking: ', result)
```

Output

Dominance directed graph compromise ranking: [3 2 1]

The Rank Position compromise ranking method
This method is used to generate compromise ranking based on several rankings provided by different MCDA methods. The `rank_position_method` method requires providing two-dimensional matrix `matrix` with different rankings in particular columns. `rank_position_method` returns vector with compromise ranking.

```
import numpy as np
from pyrepo_mcda import compromise_rankings as compromises

# Provide matrix with different rankings given by different MCDA methods in columns
matrix = np.array([[3, 2, 3],
                   [2, 3, 2],
                   [1, 1, 1]])

# Calculate the compromise ranking using `rank_position_method` method
result = compromises.rank_position_method(matrix)

print('Rank position compromise ranking: ', result)
```

Output

```
Rank position compromise ranking: [3 2 1]
```

The Improved Borda Rule compromise ranking method for MULTIMOORA

This method is used to generate compromise ranking based on three rankings provided by particular approaches (RS, RP and FMF) of MULTIMOORA method. The `improved_borda_rule` method requires providing two-dimensional matrix `matrix` with three rankings in particular columns. `improved_borda_rule` returns vector with compromise ranking.

```
import numpy as np
from pyrepo_mcda import compromise_rankings as compromises

# Provide matrix with different preference values given by different MCDA methods in columns
prefs = np.array([[4.94364901e-01, 4.56157867e-02, 3.85006756e-09],
                  [5.26950959e-01, 6.08111832e-02, 9.62516889e-09],
                  [6.77457681e-01, 0.00000000e+00, 4.45609671e-08]])

# Provide matrix with different rankings given by different MCDA methods in columns
ranks = np.array([[3, 2, 3],
                  [2, 3, 2],
                  [1, 1, 1]])

# Calculate the compromise ranking using `improved_borda_rule` method
result = compromises.improved_borda_rule(prefs, ranks)

print('Improved Borda Rule compromise ranking: ', result)
```

Output

```
Improved Borda Rule compromise ranking: [2 3 1]
```
Correlation coefficients

Spearman correlation coefficient

This method is used to calculate correlation between two different rankings. It requires two vectors \( R \) and \( Q \) with rankings of the same size. It returns value of correlation.

```python
import numpy as np
from pyrepo_mcda import correlations as corrs

# Provide two vectors with rankings obtained with different MCDA methods
R = np.array([1, 2, 3, 4, 5])
Q = np.array([1, 3, 2, 4, 5])

# Calculate the correlation using `spearman` coefficient
coeff = corrs.spearman(R, Q)
print('Spearman coeff: ', np.round(coeff, 4))
```

Output

```
Spearman coeff:  0.9
```

Weighted Spearman correlation coefficient

This method is used to calculate correlation between two different rankings. It requires two vectors \( R \) and \( Q \) with rankings of the same size. It returns value of correlation.

```python
import numpy as np
from pyrepo_mcda import correlations as corrs

# Provide two vectors with rankings obtained with different MCDA methods
R = np.array([1, 2, 3, 4, 5])
Q = np.array([1, 3, 2, 4, 5])

# Calculate the correlation using `weighted_spearman` coefficient
coeff = corrs.weighted_spearman(R, Q)
print('Weighted Spearman coeff: ', np.round(coeff, 4))
```

Output

```
Weighted Spearman coeff:  0.8833
```

Similarity rank coefficient WS

This method is used to calculate similarity between two different rankings. It requires two vectors \( R \) and \( Q \) with rankings of the same size. It returns value of similarity.

```python
import numpy as np
from pyrepo_mcda import correlations as corrs

# Provide two vectors with rankings obtained with different MCDA methods
R = np.array([1, 2, 3, 4, 5])
Q = np.array([1, 3, 2, 4, 5])

# Calculate the similarity using `WS_coeff` coefficient
```
coeff = corrs.WS_coeff(R, Q)
print('WS coeff: ', np.round(coeff, 4))

Output

| WS coeff: 0.8542 |

Pearson correlation coefficient

This method is used to calculate correlation between two different rankings. It requires two vectors \( R \) and \( Q \) with rankings of the same size. It returns value of correlation.

```python
import numpy as np
from pyrepo_mcda import correlations as corrs

# Provide two vectors with rankings obtained with different MCDA methods
R = np.array([1, 2, 3, 4, 5])
Q = np.array([1, 3, 2, 4, 5])

# Calculate the correlation using `pearson_coeff` coefficient
coeff = corrs.pearson_coeff(R, Q)
print('Pearson coeff: ', np.round(coeff, 4))
```

Output

| Pearson coeff: 0.9 |

**Methods for criteria weights determination**

**Entropy weighting method**

This method is used to calculate criteria weights based on alternatives performance values provided in decision matrix. This method requires providing two-dimensional decision matrix \( \text{matrix} \) with performance values of alternatives in rows considering criteria in columns. It returns vector with criteria weights. All values in vector \( \text{weights} \) must sum to 1.

```python
import numpy as np
from pyrepo_mcda import weighting_methods as mcda_weights

matrix = np.array([[30, 30, 38, 29],
                    [19, 54, 86, 29],
                    [19, 15, 85, 28.9],
                    [68, 70, 60, 29]])

weights = mcda_weights.entropy_weighting(matrix)
print('Entropy weights: ', np.round(weights, 4))
```

Output

| Entropy weights: [0.463 0.3992 0.1378 0. ] |

**CRITIC weighting method**

1.1. Usage
This method is used to calculate criteria weights based on alternatives performance values provided in decision matrix. This method requires providing two-dimensional decision matrix \( \text{matrix} \) with performance values of alternatives in rows considering criteria in columns. It returns vector with criteria weights. All values in vector \( \text{weights} \) must sum to 1.

```python
import numpy as np
from pyrepo_mcda import weighting_methods as mcda_weights

matrix = np.array([[5000, 3, 4, 3, 2],
                    [680, 5, 3, 2, 1],
                    [2000, 3, 2, 3, 4],
                    [600, 4, 3, 1, 2],
                    [800, 2, 4, 3, 4]])

weights = mcda_weights.critic_weighting(matrix)
print('CRITIC weights: ', np.round(weights, 4))
```

Output

```
CRITIC weights:  [0.157 0.2495 0.1677 0.1211 0.1541 0.1506]
```

Standard deviation weighting method

This method is used to calculate criteria weights based on alternatives performance values provided in decision matrix. This method requires providing two-dimensional decision matrix \( \text{matrix} \) with performance values of alternatives in rows considering criteria in columns. It returns vector with criteria weights. All values in vector \( \text{weights} \) must sum to 1.

```python
import numpy as np
from pyrepo_mcda import weighting_methods as mcda_weights

matrix = np.array([[0.619, 0.449, 0.447],
                    [0.862, 0.466, 0.006],
                    [0.458, 0.698, 0.771],
                    [0.777, 0.631, 0.491],
                    [0.567, 0.992, 0.968]])

weights = mcda_weights.std_weighting(matrix)
print('Standard deviation weights: ', np.round(weights, 4))
```

Output

```
Standard deviation weights:  [0.2173 0.2945 0.4882]
```
Distance metrics

Here are two examples of using distance metrics for Euclidean distance euclidean and Manhattan distance manhattan. Usage of other distance metrics provided in module distance metrics is analogous.

Euclidean distance

This method is used to calculate the Euclidean distance between two vectors A and B containing real values. The size od A and B must be the same. This method returns value of Euclidean distance between vectors A and B.

```python
import numpy as np
from pyrepo_mcda import distance_metrics as dists

A = np.array([0.165, 0.113, 0.015, 0.019])
B = np.array([0.227, 0.161, 0.053, 0.130])

dist = dists.euclidean(A, B)
print('Distance: ', np.round(dist, 4))
```

Output

Distance: 0.1411

Manhattan distance

This method is used to calculate the Manhattan distance between two vectors A and B containing real values. The size od A and B must be the same. This method returns value of Manhattan distance between vectors A and B.

```python
import numpy as np
from pyrepo_mcda import distance_metrics as dists

A = np.array([0.165, 0.113, 0.015, 0.019])
B = np.array([0.227, 0.161, 0.053, 0.130])

dist = dists.manhattan(A, B)
print('Distance: ', np.round(dist, 4))
```

Output

Distance: 0.259

Normalization methods

Here is an example of vector normalization usage. Other normalizations provided in module normalizations, namely minmax_normalization, max_normalization, sum_normalization, linear_normalization, multimoora_normalization are used in analogous way.

Vector normalization

This method is used to normalize decision matrix matrix. It requires providing decision matrix matrix with performance values of alternatives in rows considering criteria in columns and vector with criteria types types. This method returns normalized matrix.

```python
import numpy as np
from pyrepo_mcda import normalizations as norms

# (continues on next page)
```
matrix = np.array([[8, 7, 2, 1],
[5, 3, 7, 5],
[7, 5, 6, 4],
[9, 9, 7, 3],
[11, 10, 3, 7],
[6, 9, 5, 4]])

types = np.array([1, 1, 1, 1])
norm_matrix = norms.vector_normalization(matrix, types)
print('Normalized matrix: ', np.round(norm_matrix, 4))

Output
Normalized matrix:  [[0.4126 0.3769 0.1525 0.0928]
[0.2579 0.1615 0.5337 0.4642]
[0.361 0.2692 0.4575 0.3714]
[0.4641 0.4845 0.5337 0.2785]
[0.5673 0.5384 0.2287 0.6499]
[0.3094 0.4845 0.3812 0.3714]]

Methods for sensitivity analysis considering criteria weights modification

Sensitivity_analysis_weights_percentages

This method is used to perform the procedure of sensitivity analysis considering percentage modification the weight value of chosen criterion. This method requires providing two-dimensional decision matrix matrix, vector with criteria weights weights, vector with criteria types types, vector with real values of weight modification in percentages percentages (provided in range from 0 to 1), initialized object of chosen MCDA method method, index of column in decision matrix for chosen criterion j and list with directions of weight modification dir. dir can be set in three ways: when you want only increase weight value: [1], when you want only decrease weight value: [-1], when you want decrease and increase weight value: [-1, 1]. dir is set as [1] by default.

import numpy as np
from pyrepo_mcda.sensitivity_analysis_weights_percentages import Sensitivity_analysis_weights_percentages

import numpy as np
from pyrepo_mcda.mcda_methods import CODAS

# provide decision matrix in array numpy.darray
matrix = np.array([[45, 3600, 45, 0.9],
[25, 3800, 60, 0.8],
[23, 3100, 35, 0.9],
[14, 3400, 50, 0.7],
[15, 3300, 40, 0.8],
[28, 3000, 30, 0.6]])

# provide criteria weights in array numpy.darray. All weights must sum to 1.
weights = np.array([0.2857, 0.3036, 0.2321, 0.1786])

(continues on next page)
# provide criteria types in array numpy.darray. Profit criteria are represented by 1 and cost criteria by -1.
types = np.array([1, -1, 1, 1])

# provide vector with percentage values of chosen criterion weight modification
percentages = np.arange(0.05, 0.5, 0.1)

#create the chosen MCDA object
method = TOPSIS(normalization_method=norms.minmax_normalization, distance_metric=dists.euclidean)

# provide index of j-th chosen criterion whose weight will be modified in sensitivity analysis, for example j = 1 for criterion in the second column
j = 1

# Create the Sensitivity_analysis_weights_percentages object
sensitivity_analysis = Sensitivity_analysis_weights_percentages()

# Generate DataFrame with rankings for different modification of weight of chosen criterion
# Provide decision matrix `matrix`, vector with criteria weights `weights`, criteria types `types`, initialized object of chosen MCDA method `method`, and index of column in decision matrix for chosen criterion `j`
data_sens = sensitivity_analysis(matrix, weights, types, percentages, method, j, [1])

Sensitivity_analysis_weights_values
This method is used to perform the procedure of sensitivity analysis considering setting chosen value as the weight of selected criterion. This method requires providing two-dimensional decision matrix, vector with values to be set as selected criterion weight, vector with criteria types, initialized object of chosen MCDA method, and index of column in decision matrix for chosen criterion j

```python
import numpy as np
from pyrepo_mcda.sensitivity_analysis_weights_values import Sensitivity_analysis_weights_values

import numpy as np
from pyrepo_mcda.mcda_methods import CODAS

# provide decision matrix in array numpy.darray
matrix = np.array([[45, 3600, 45, 0.9],
                   [25, 3800, 60, 0.8],
                   [23, 3100, 35, 0.9],
                   [14, 3400, 50, 0.7],
                   [15, 3300, 40, 0.8],
                   [28, 3000, 30, 0.6]])

# provide criteria weights in array numpy.darray. All weights must sum to 1.
weights = np.array([0.2857, 0.3036, 0.2321, 0.1786])

# provide criteria types in array numpy.darray. Profit criteria are represented by 1 and cost criteria by -1.
types = np.array([1, -1, 1, 1])
```

(continues on next page)
```python
types = np.array([1, -1, 1, 1])

# provide vector with values to be set as weight of selected criterion.
weight_values = np.arange(0.05, 0.95, 0.1)

#create the chosen MCDA object
method = TOPSIS(normalization_method=norms.minmax_normalization, distance_metric=dists.euclidean)

# provide index of j-th chosen criterion whose weight will be modified in sensitivity analysis, for example j = 1 for criterion in the second column
j = 1

# Create the Sensitivity_analysis_weights_values object
sensitivity_analysis = Sensitivity_analysis_weights_values()

# Generate DataFrame with rankings for different modification of weight of chosen criterion
# Provide decision matrix `matrix`, vector with values `weight_values` to be set as weight of selected criterion, criteria types `types`, initialized object of chosen MCDA
# method `method` and index of chosen criterion whose weight will be modified.
data_sens = sensitivity_analysis_weights_values(matrix, weight_values, types, method, j)
```

1.2 Illustrative examples

1.2.1 Import necessary packages

Import of the necessary Python packages necessary for running codes provided in examples.

```python
[1]: import copy
    import numpy as np
    import pandas as pd
    import matplotlib
    from tabulate import tabulate

    Import the necessary Python modules from pyrepo-mcda package.

[2]: from pyrepo_mcda.mcda_methods import CODAS, TOPSIS, WASPAS, VIKOR, SPOTIS, EDAS, MABAC, MULTIMOORA
    from pyrepo_mcda import distance_metrics as dists
    from pyrepo_mcda import correlations as corrs
    from pyrepo_mcda import normalizations as norms
    from pyrepo_mcda import weighting_methods as mcda_weights
    from pyrepo_mcda import compromise_rankings as compromises
    from pyrepo_mcda.additions import rank_preferences
    from pyrepo_mcda.sensitivity_analysis_weights_percentages import Sensitivity_analysis_weights_percentages
    from pyrepo_mcda.sensitivity_analysis_weights_values import Sensitivity_analysis_weights_values
```

1.2.2 Examples of supporting functions

Import visualization methods from visualizations.py, including exemplary techniques for displaying charts provided in the visualizations.py file available in examples.

```python
from visualizations import plot_barplot, draw_heatmap, plot_boxplot, plot_lineplot,
    plot_barplot_sensitivity, plot_boxplot_simulation
```

Sample class for creating a dictionary. It will help collect correlation values between different rankings.

```python
# Create dictionary class
class Create_dictionary(dict):
    # __init__ function
    def __init__(self):
        self = dict()

    # Function to add key:value
    def add(self, key, value):
        self[key] = value
```

1.2.3 Loading data from CSV file

You can provide a decision matrix, criteria weights, and types directly in your code in NumPy arrays. You can also load input data from a CSV file. In the provided exemplary CSV file data.csv, there are alternatives in the rows ($A_i$) and criteria in the columns ($C_j$). The last row (types) of the data frame contains criteria types (with values 1 for profit and -1 for cost criteria), and the second-to-last row (weights) contains criteria weights (their sum must be equal to 1)

Create a dataframe with a decision matrix `df_data` and NumPy arrays of weights `weights` and criteria types `types`

```python
data = pd.read_csv('data.csv', index_col = 'Ai')
```

# loading input data from a file using the pandas package into a dataframe
`df_data` = data.iloc[:len(data) - 2, :]
`weights` = data.iloc[len(data) - 2, :].to_numpy()
`types` = data.iloc[len(data) - 1, :].to_numpy()

Display the decision matrix as a dataframe. In this case, we used a nice-looking way to print the dataframe, which is performed using `tabulate`.

```python
header = [df_data.index.name]
header = header + list(df_data.columns)
print(tabulate(df_data, headers = header, tablefmt='github'))
```

<table>
<thead>
<tr>
<th>Ai</th>
<th>C1</th>
<th>C2</th>
<th>C3</th>
<th>C4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A1</td>
<td>45</td>
<td>3600</td>
<td>45</td>
<td>0.9</td>
</tr>
<tr>
<td>A2</td>
<td>25</td>
<td>3800</td>
<td>60</td>
<td>0.8</td>
</tr>
<tr>
<td>A3</td>
<td>23</td>
<td>3100</td>
<td>35</td>
<td>0.9</td>
</tr>
</tbody>
</table>

(continues on next page)
| A4 | 14 | 3400 | 50 | 0.7 |
| A5 | 15 | 3300 | 40 | 0.8 |
| A6 | 28 | 3000 | 30 | 0.6 |

[7]: print(weights)

```
[0.2857 0.3036 0.2321 0.1786]
```

[8]: print(types)

```
[ 1. -1. 1. 1.]
```

Converting a decision matrix data frame df_data to a NumPy array matrix to use the NumPy package for computation.

[9]: matrix = df_data.to_numpy()

Create an index list list_alt_names and data frame for collecting results rank_results indexed by 'Ai'.

[10]: list_alt_names = [r'$A_{'+ str(i) + '}$' for i in range(1, df_data.shape[0] + 1)]

```
list_crit_names = [r'$C_{'+ str(i) + '}$' for i in range(1, df_data.shape[1] + 1)]
```

[11]: rank_results = pd.DataFrame()

```
rank_results['Ai'] = list(list_alt_names)
```

1.2.4 TOPSIS

Create the TOPSIS method object. The default normalization type for this method is Min-Max normalization minmax_normalization, and the default distance metric for determining the distance to ideal (PIS) and non-ideal (NIS) solutions is Euclidean distance. However, other normalization methods from the normalizations module and other distance metrics from the distance_metrics module for investigations and simulations can be applied. In the TOPSIS method, the ranking of the alternatives is created by sorting the preferences in descending order (the best alternative has the highest preference value). Therefore the reverse = True parameter is used in the rank_preferences function to call descending ordering.

[12]: # TOPSIS
    # TOPSIS preference values (preferences) must be sorted in descending order
    topsis = TOPSIS(normalization_method = norms.minmax_normalization, distance_metric =
                      ...dists.euclidean)
    pref = topsis(matrix, weights, types)
    rank = rank_preferences(pref, reverse = True)
    rank_results['TOPSIS'] = rank
1.2.5 CODAS

Create the CODAS method object. The default and recommended normalization type for this method is linear normalization `linear_normalization`, and the default distance metric for determining the distance to a non-ideal (NIS) solution is Euclidean distance. However, other distance metrics from the `distance_metrics` module for investigations and simulations can be applied. In the CODAS method, the ranking of the alternatives is created by sorting the preferences in descending order (the best option has the highest preference value). Therefore the `reverse = True` parameter is used in the `rank_preferences` function to call descending ordering.

13: # CODAS
# CODAS preference values (preferences) must be sorted descending order
codas = CODAS(normalization_method = norms.linear_normalization, distance_metric = dists.euclidean, tau = 0.02)
pref = codas(matrix, weights, types)
rank = rank_preferences(pref, reverse = True)
rank_results['CODAS'] = rank

1.2.6 VIKOR

Create the VIKOR method object. VIKOR can be executed without normalization (parameter `normalization_method = None`), but any normalization technique from the `normalizations` package can also be used. This method does not use distance metrics because its algorithm individually computes the distance from reference solutions. In the VIKOR method, the ranking of the alternatives is created by sorting the preferences in ascending order (the best alternative has the lowest preference value). Therefore the `reverse = False` parameter is used in the `rank_preferences` function to call ascending ordering.

14: # VIKOR
# VIKOR preferences must be sorted in ascending order
vikor = VIKOR(normalization_method = norms.minmax_normalization)
pref = vikor(matrix, weights, types)
rank = rank_preferences(pref, reverse = False)
rank_results['VIKOR'] = rank

1.2.7 SPOTIS

Create a SPOTIS method object. A unique algorithm performs normalization in the SPOTIS method, so the parameter of normalization type is not given. SPOTIS requires bounds, where `bounds_min` denotes the minimum, and `bounds_max` denotes the maximum values of the criteria considered in the problem being solved. Bounds are needed to determine the ideal ISP solution, consisting of maximum bounds values for profit criteria and minimum bounds values for cost criteria. In the SPOTIS method, the ranking of the alternatives is created by sorting the preferences in ascending order (the best option has the lowest preference value). Therefore the `reverse = False` parameter is used in the `rank_preferences` function to call ascending ordering.

15: # SPOTIS
# SPOTIS preferences must be sorted in ascending order
bounds_min = np.amin(matrix, axis = 0)
bounds_max = np.amax(matrix, axis = 0)
bounds = np.vstack((bounds_min, bounds_max))
spotis = SPOTIS()
pref = spotis(matrix, weights, types, bounds)

(continues on next page)
1.2.8 EDAS

Create the EDAS method object. A unique algorithm performs normalization in the EDAS method, so the parameter of normalization type is not given. In the EDAS method, the ranking of the alternatives is created by sorting the preferences in descending order (the best option has the highest preference value). Therefore the `reverse = True` parameter is used in the `rank_preferences` function to call descending ordering.

```py
# EDAS
# EDAS preferences must be sorted in descending order
edas = EDAS()
pref = edas(matrix, weights, types)
rank = rank_preferences(pref, reverse = True)
rank_results['EDAS'] = rank
```

1.2.9 MABAC

Create the MABAC method object. The default and recommended normalization type for this method is Minimum-Maximum normalization. In the MABAC method, the ranking of the alternatives is created by sorting the preferences in descending order (the best option has the highest preference value). Therefore the `reverse = True` parameter is used in the `rank_preferences` function to call descending ordering.

```py
# MABAC
# MABAC preferences must be sorted in descending order
mabac = MABAC(normalization_method = norms.minmax_normalization)
pref = mabac(matrix, weights, types)
rank = rank_preferences(pref, reverse = True)
rank_results['MABAC'] = rank
```

1.2.10 MULTIMOORA

Create the MULTIMOORA method object. This method involves determining by majority voting a ranking from the three rankings provided by the three integrated approaches: - Ratio System Approach (MULTIMOORA_RS in multimoora module, preferences sorted in descending order) - Reference Point Approach (MULTIMOORA_RP in multimoora module, preferences sorted in ascending order) - Full Multiplicative Form (MULTIMOORA_FMF in multimoora module, preferences sorted in descending order)

In these approaches, an individual type of normalization is performed. It is vector normalization, performed for all criteria, like for the profit criteria. This normalization technique, especially for MOORA and MULTIMOORA methods, is provided in normalizations and named multimoora_normalization. The MULTIMOORA method provides a ready ranking of alternatives.

```py
# MULTIMOORA
# MULTIMOORA method returns rank
multimoora = MULTIMOORA()
rank = multimoora(matrix, weights, types)
rank_results['MMOORA'] = rank
```
1.2.11 WASPAS

Create the WASPAS method object. This method’s default and recommended normalization type is linear normalization `linear_normalization`. This method requires an additional lambda parameter `lambda_param` in the range 0 to 1, (0, 0.1, 0.2, ..., 1).

```python
# WASPAS
# WASPAS preferences must be sorted in descending order
waspas = WASPAS(normalization_method = norms.linear_normalization, lambda_param = 0.5)
pref = waspas(matrix, weights, types)
rank = rank_preferences(pref, reverse = True)
rank_results['WASPAS'] = rank
```

Display a data frame with the rankings provided by each method.

```python
rank_results = rank_results.set_index('Ai')
header = [rank_results.index.name]
header = header + list(rank_results.columns)
print(tabulate(rank_results, headers = header, tablefmt='orgtbl'))
```

1.2.12 Compromise ranking generated with the Copeland method

In the following stage, one compromise ranking is created based on the received rankings by method `copeland` provided in `compromise_rankings` module.

```python
compromise_ranking = compromises.copeland(rank_results)
rank_results_final = copy.deepcopy(rank_results)
rank_results_final['Compromise'] = compromise_ranking
header = [rank_results_final.index.name]
header = header + list(rank_results_final.columns)
print(tabulate(rank_results_final, headers = header, tablefmt='github'))
```

(continues on next page)
Display rankings provided by different MCDA methods using an exemplary method named *plot_barplot* for visualization.

![Barplot](image)

### 1.2.13 Sensitivity analysis

**Sensitivity analysis with percentage modification of chosen criterion weights**

Sensitivity analysis is a valuable procedure for determining the sensitivity of alternatives to changes in input data. In this case, the effect of modifying the values of the criteria weights by given percentage values is investigated. First, the percentages by which the weights of each criterion are decreased or increased are input into the vector named `percentages`. The remaining weights are then changed equivalently by the appropriate value so that the sum of all weights still equals 1. Sequentially, the sensitivity analysis is performed in the `_sensitivity_analysis_weights_percentages` function provided in `sensitivity_analysis_weights_percentages` module in class `Sensitivity_analysis_weights_percentages`. Sensitivity analysis is performed for the chosen criterion for the selected MCDA method’s initialized object. Finally, the results are visualized as a dataframe, column, and line chart. Visualization is performed using an exemplary method named *plot_lineplot_sensitivity* provided in examples in `visualizations.py` file.
# sensitivity analysis

# load input vector with percentage values of chosen criterion weights modification for sensitivity analysis
percentages = np.arange(0.25, 0.55, 0.1)
percentages = np.arange(0.05, 0.55, 0.1)

# create the chosen MCDA object
method = TOPSIS(normalization_method=norms.minmax_normalization, distance_metric=dists.euclidean)

# Create the sensitivity analysis method object
sensitivity_analysis = Sensitivity_analysis_weights_percentages()

# Perform sensitivity analysis with weights modification for chosen criteria
for j in [0, 1, 2, 3]:
    data_sens = sensitivity_analysis(matrix, weights, types, percentages, method, j, [-1, 1])

    header = [data_sens.index.name]
    header = header + list(data_sens.columns)
    print(f'Sensitivity analysis for C{j + 1}

<table>
<thead>
<tr>
<th>Ai</th>
<th>-45%</th>
<th>-35%</th>
<th>-25%</th>
<th>-15%</th>
<th>-5%</th>
<th>5%</th>
<th>15%</th>
<th>25%</th>
<th>35%</th>
<th>45%</th>
</tr>
</thead>
<tbody>
<tr>
<td>$A_1$</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>$A_2$</td>
<td>6</td>
<td>6</td>
<td>5</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>$A_3$</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>$A_4$</td>
<td>5</td>
<td>5</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>$A_5$</td>
<td>3</td>
<td>4</td>
<td>4</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>$A_6$</td>
<td>4</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>2</td>
<td>2</td>
</tr>
</tbody>
</table>

1.2. Illustrative examples
Sensitivity analysis for C2

<table>
<thead>
<tr>
<th>Ai</th>
<th>-45%</th>
<th>-35%</th>
<th>-25%</th>
<th>-15%</th>
<th>-5%</th>
<th>5%</th>
<th>15%</th>
<th>25%</th>
<th>35%</th>
</tr>
</thead>
<tbody>
<tr>
<td>$A_1$</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$A_2$</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>4</td>
<td>6</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td></td>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$A_3$</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$A_4$</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>5</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$A_5$</td>
<td>6</td>
<td>6</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$A_6$</td>
<td>5</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

(continues on next page)
### Sensitivity analysis for $C_4$

<table>
<thead>
<tr>
<th>$A_i$</th>
<th>-45%</th>
<th>-35%</th>
<th>-25%</th>
<th>-15%</th>
<th>-5%</th>
<th>5%</th>
<th>15%</th>
<th>25%</th>
<th>35%</th>
<th>---</th>
</tr>
</thead>
<tbody>
<tr>
<td>$A_1$</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>$A_2$</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>$A_3$</td>
<td>3</td>
<td>3</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>$A_4$</td>
<td>5</td>
<td>5</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>$A_5$</td>
<td>6</td>
<td>6</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>$A_6$</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
</tbody>
</table>

#### TOPSIS, modification of $C_3$ weight

![TOPSIS diagram](image-url)
Sensitivity analysis with setting chosen values as selected criterion weight

The second sensitivity analysis method is provided in module `sensitivity_analysis_weights_values` in class `Sensitivity_analysis_weights_values`. It investigates the effect of setting different values from 0 to 1 as chosen criterion weight.

```python
[24] # Perform sensitivity analysis with setting chosen weight value to selected criterion
# other criteria have equal weight values and all criteria weights sum to 1
sensitivity_analysis_weights_values = Sensitivity_analysis_weights_values()
weight_values = np.arange(0.05, 0.95, 0.1)
for j in [0, 1, 2, 3]:
    data_sens = sensitivity_analysis_weights_values(matrix, weight_values, types, method, j)
    header = [data_sens.index.name]
    header = header + list(data_sens.columns)
    print('Sensitivity analysis for C' + str(j + 1))
    print(tabulate(data_sens, headers=header, tablefmt='github'))
    plot_lineplot_sensitivity(data_sens, method.__class__.__name__, list_crit_names[j],
                              "Weight value", "value")
```

Sensitivity analysis for C1

<table>
<thead>
<tr>
<th>Ai</th>
<th>0.05</th>
<th>0.15</th>
<th>0.25</th>
<th>0.35</th>
<th>0.45</th>
<th>0.55</th>
<th>0.65</th>
<th>0.75</th>
<th>0.85</th>
</tr>
</thead>
<tbody>
<tr>
<td>$A_1$</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>$A_2$</td>
<td>4</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>$A_3$</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>$A_4$</td>
<td>5</td>
<td>5</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td></td>
</tr>
<tr>
<td>$A_5$</td>
<td>3</td>
<td>4</td>
<td>4</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>$A_6$</td>
<td>6</td>
<td>6</td>
<td>5</td>
<td>4</td>
<td>4</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

![TOPSIS, modification of C4 weight](image)
Sensitivity analysis for C2
<table>
<thead>
<tr>
<th>Ai</th>
<th>0.05</th>
<th>0.15</th>
<th>0.25</th>
<th>0.35</th>
<th>0.45</th>
<th>0.55</th>
<th>0.65</th>
<th>0.75</th>
<th>0.85</th>
</tr>
</thead>
<tbody>
<tr>
<td>$A_1$</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>4</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>4</td>
</tr>
<tr>
<td>$A_2$</td>
<td>2</td>
<td>2</td>
<td>3</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>4</td>
</tr>
<tr>
<td>$A_3$</td>
<td>3</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>$A_4$</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>5</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>$A_5$</td>
<td>5</td>
<td>5</td>
<td>6</td>
<td>5</td>
<td>5</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>$A_6$</td>
<td>6</td>
<td>6</td>
<td>5</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
</tbody>
</table>

Sensitivity analysis for C3
<table>
<thead>
<tr>
<th>Ai</th>
<th>0.05</th>
<th>0.15</th>
<th>0.25</th>
<th>0.35</th>
<th>0.45</th>
<th>0.55</th>
<th>0.65</th>
<th>0.75</th>
<th>0.85</th>
</tr>
</thead>
<tbody>
<tr>
<td>$A_1$</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>$A_2$</td>
<td>2</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>$A_3$</td>
<td>3</td>
<td>3</td>
<td>2</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
</tbody>
</table>

(continues on next page)
| $A_{2}$ | 5 | 5 | 3 | 2 | 1 | 1 | 1 | 1 |
| $A_{3}$ | 2 | 2 | 2 | 3 | 4 | 5 | 5 | 5 |
| $A_{4}$ | 6 | 6 | 6 | 4 | 3 | 2 | 2 | 2 |
| $A_{5}$ | 4 | 4 | 4 | 5 | 5 | 4 | 4 | 4 |
| $A_{6}$ | 3 | 3 | 5 | 6 | 6 | 6 | 6 | 6 |

Sensitivity analysis for $C_{3}$ weight:

| $A_{1}$ | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| $A_{2}$ | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 |
| $A_{3}$ | 4 | 2 | 2 | 2 | 2 | 2 | 2 | 2 |
| $A_{4}$ | 5 | 5 | 6 | 5 | 5 | 5 | 5 | 5 |
| $A_{5}$ | 6 | 6 | 4 | 4 | 4 | 4 | 4 | 4 |
| $A_{6}$ | 2 | 4 | 5 | 6 | 6 | 6 | 6 | 6 |

Sensitivity analysis for $C_{4}$

| $A_{1}$ | 0.05 | 0.15 | 0.25 | 0.35 | 0.45 | 0.55 | 0.65 | 0.75 | 0.85 |
| $A_{2}$ | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| $A_{3}$ | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 |
| $A_{4}$ | 4 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 |
| $A_{5}$ | 5 | 5 | 6 | 5 | 5 | 5 | 5 | 5 | 5 |
| $A_{6}$ | 6 | 6 | 4 | 4 | 4 | 4 | 4 | 4 | 4 |
An example of another exemplary kind of chart for sensitivity analysis results visualization is the Bar chart `plot_barplot_sensitivity` provided in `visualizations.py` in examples.

```
[25]: plot_barplot_sensitivity(data_sens, method.__class__.__name__, list_crit_names[j], ...
   \"weight_values_bar\")
```

Example of using another exemplary kind of chart for sensitivity analysis results visualization: Radar chart `plot_radar` provided in `visualizations.py` in examples.

```
[26]: plot_radar(data_sens, list_crit_names[j] + \u2018weight modification\u2019, j)
```
1.2.14 Comparative analysis of distance metrics

Results of TOPSIS method using different chosen distance metrics

```
[27]: # Create a list with distance metrics chosen from module `distance metrics`
    distance_metrics = [
        dists.euclidean,
        dists.manhattan,
        dists.hausdorff,
        dists.chebyshev,
        dists.bray_curtis,
        dists.canberra,
        dists.lorentzian,
        dists.jaccard
    ]

    # Create dataframes for preference function values and rankings determined using distance metrics
    df_preferences = pd.DataFrame(index = list_alt_names)
    df_rankings = pd.DataFrame(index = list_alt_names)

    for distance_metric in distance_metrics:
        # Create the TOPSIS method object
        topsis = TOPSIS(normalization_method = norms.minmax_normalization, distance_metric = distance_metric)
        pref = topsis(matrix, weights, types)
        rank = rank_preferences(pref, reverse = True)
        df_preferences[distance_metric.__name__.capitalize().replace('_', ' ')] = pref
        df_rankings[distance_metric.__name__.capitalize().replace('_', ' ')] = rank

    df_rankings
```
### Illustrative examples

#### Distance metrics

<table>
<thead>
<tr>
<th></th>
<th>Euclidean</th>
<th>Manhattan</th>
<th>Hausdorff</th>
<th>Chebyshev</th>
<th>Bray curtis</th>
<th>Canberra</th>
</tr>
</thead>
<tbody>
<tr>
<td>$A_1$</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>$A_2$</td>
<td>4</td>
<td>3</td>
<td>4</td>
<td>4</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>$A_3$</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>$A_4$</td>
<td>6</td>
<td>6</td>
<td>5</td>
<td>6</td>
<td>6</td>
<td>5</td>
</tr>
<tr>
<td>$A_5$</td>
<td>5</td>
<td>5</td>
<td>3</td>
<td>5</td>
<td>5</td>
<td>3</td>
</tr>
<tr>
<td>$A_6$</td>
<td>3</td>
<td>4</td>
<td>4</td>
<td>3</td>
<td>4</td>
<td>6</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>Lorentzian</th>
<th>Jaccard</th>
</tr>
</thead>
<tbody>
<tr>
<td>$A_1$</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>$A_2$</td>
<td>3</td>
<td>5</td>
</tr>
<tr>
<td>$A_3$</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>$A_4$</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>$A_5$</td>
<td>5</td>
<td>4</td>
</tr>
<tr>
<td>$A_6$</td>
<td>4</td>
<td>3</td>
</tr>
</tbody>
</table>

#### Plot box chart of alternatives preference values

```python
# plot box chart of alternatives preference values
plot_barplot(df_rankings, 'Distance metrics')
```

![Box plot of alternatives preference values for different distance metrics](image)

#### TOPSIS, preference values distribution

```python
# plot box chart of alternatives preference values
plot_boxplot(df_preferences.T, 'TOPSIS, preference values distribution')
```
1.2.15 Robustness analysis

Robustness analysis is performed to indicate the intervals of the performance values of the alternatives concerning the selected criteria for which the alternatives hold particular ranking positions. Finally, results are displayed using the exemplary visualization method named `plot_boxplot_simulation` provided in `visualizations.py` in examples.

![TOPSIS, preference values distribution](image)

```python
# Robustness analysis
# Create object of chosen MCDA method
tOPSIS = TOPSIS(normalization_method=norms.minmax_normalization, distance_metric=dists.euclidean)

# Create minimum bounds of criteria performance
bounds_min = np.amin(matrix, axis = 0)
# Create maximum bounds of criteria performance
bounds_max = np.amax(matrix, axis = 0)
bounds = np.vstack((bounds_min, bounds_max))

# Create ideal Solution `isp`
isp = np.zeros(matrix.shape[1])
isp[types == 1] = bounds[1, types == 1]
isp[types == -1] = bounds[0, types == -1]

# Create anti-Ideal Solution `asp`
asp = np.zeros(matrix.shape[1])
asp[types == 1] = bounds[0, types == 1]
asp[types == -1] = bounds[1, types == -1]

# Create dictionary with values of stepwise particular criteria performance change
indexes = {
    0 : 1,
    1 : 10,
    2 : 5,
    3 : 0.1
}
```

(continues on next page)
# Perform simulation for each criterion
# Iterate by all criteria
for j in range(matrix.shape[1]):
    change_val = indexes[j]
    # dictionary for collecting variability in TOPSIS preferences after weights change using different distance metrics
    dict_results_sim = {
        'Rank' : [],
        'Performance' : [],
        'Alternative' : []
    }
    # Iterate by all Alternatives
    for i in range(matrix.shape[0]):
        vec = np.arange(asp[j], isp[j] + types[j] * change_val, types[j] * change_val)
        for v in vec:
            new_matrix = copy.deepcopy(matrix)
            new_matrix[i, j] = v
            pref = topsis(new_matrix, weights, types)
            rank = rank_preferences(pref, reverse = True)
            dict_results_sim['Rank'].append(rank[i])
            dict_results_sim['Performance'].append(v)
            dict_results_sim['Alternative'].append(list_alt_names[i])

df_results_sim = pd.DataFrame(dict_results_sim)

plot_boxplot_simulation(df_results_sim, 'Alternative', 'Performance', 'Rank', 'Alternative', 'Performance', 'TOPSIS, Criterion ' + list_crit_names[j] + ' performance change', 'robustness_' + str(j + 1))
1.2.16 Results correlations

Determination of the correlation between the rankings provided by each MCDA method using three correlation coefficients provided in the module named correlations: \( r_s \) Spearman rank correlation coefficient \( \text{spearman} \) - \( r_w \) Weighted Spearman correlation coefficient \( \text{weighted_spearman} \) - \( WS \) Similarity rank correlation coefficient \( \text{WS_coeff} \) - Pearson correlation coefficient \( \text{pearson_coeff} \)

First, a dataframe containing the correlation values between each pair of MCDA methods calculated using the coefficients available in the correlations module is created, and then it is visualized using the exemplary draw_heatmap visualization method provided in visualizations.py

heat maps of correlations

```python
[31]: data = copy.deepcopy(rank_results_final)
method_types = list(data.columns)

dict_new_heatmap_rw = Create_dictionary()

for el in method_types:
    dict_new_heatmap_rw.add(el, [])

dict_new_heatmap_ws = copy.deepcopy(dict_new_heatmap_rw)
dict_new_heatmap_pearson = copy.deepcopy(dict_new_heatmap_rw)
dict_new_heatmap_spearman = copy.deepcopy(dict_new_heatmap_rw)

# heatmaps for correlations coefficients
for i, j in [(i, j) for i in method_types[::-1] for j in method_types]:
    dict_new_heatmap_rw[j].append(corrs.weighted_spearman(data[i], data[j]))
```

(continues on next page)
dict_new_heatmap_ws[j].append(corrs.WS_coeff(data[i], data[j]))

dict_new_heatmap_pearson[j].append(corrs.pearson_coeff(data[i], data[j]))

dict_new_heatmap_spearman[j].append(corrs.spearman(data[i], data[j]))

df_new_heatmap_rw = pd.DataFrame(dict_new_heatmap_rw, index = method_types[::-1])
df_new_heatmap_rw.columns = method_types

df_new_heatmap_ws = pd.DataFrame(dict_new_heatmap_ws, index = method_types[::-1])
df_new_heatmap_ws.columns = method_types

df_new_heatmap_pearson = pd.DataFrame(dict_new_heatmap_pearson, index = method_types[::-1])
df_new_heatmap_pearson.columns = method_types

df_new_heatmap_spearman = pd.DataFrame(dict_new_heatmap_spearman, index = method_types[::-1])
df_new_heatmap_spearman.columns = method_types

# correlation matrix with rw coefficient
draw_heatmap(df_new_heatmap_rw, r'$r_w$')

# correlation matrix with WS coefficient
draw_heatmap(df_new_heatmap_ws, r'$WS$')
[34]: 
# correlation matrix with Pearson coefficient

draw_heatmap(df_new_heatmap_pearson, r"$Pearson$")

[35]: 
# correlation matrix with Spearman coefficient

draw_heatmap(df_new_heatmap_spearman, r"$r_s$")

1.2. Illustrative examples
### 1.3 API Reference

This page contains auto-generated API reference documentation\(^1\).

#### 1.3.1 pyrepo_mcda

**Subpackages**

pyrepo_mcda.mcda_methods

**Submodules**

pyrepo_mcda.mcda_methods.codas

**Module Contents**

**Classes**

<table>
<thead>
<tr>
<th>CODAS</th>
<th>Helper class that provides a standard way to create an ABC using</th>
</tr>
</thead>
</table>

\(^1\) Created with sphinx-autoapi
class pyrepo_mcda.mcda_methods.codas.CODAS(normalization_method=linear_normalization, distance_metric=euclidean, tau=0.02)

Bases: pyrepo_mcda.mcda_methods.mcda_method.MCDA_method

Helper class that provides a standard way to create an ABC using inheritance.

__call__(self, matrix, weights, types)

Score alternatives provided in decision matrix matrix with m alternatives and n criteria using criteria weights and criteria types.

Parameters

- **matrix** (ndarray) – Decision matrix with m alternatives in rows and n criteria in columns.
- **weights** (ndarray) – Vector with criteria weights. Sum of weights must be equal to 1.
- **types** (ndarray) – Vector with criteria types. Profit criteria are represented by 1 and cost by -1.

Returns

Vector with preference values of each alternative. The best alternative has the highest preference value.

Return type

ndarray

Examples

```python
>>> codas = CODAS(normalization_method = linear_normalization, distance_metric = euclidean, tau = 0.02)
>>> pref = codas(matrix, weights, types)
>>> rank = rank_preferences(pref, reverse = True)
```

_psi(self, x)

static _codas(self, matrix, weights, types, normalization_method, distance_metric)

Module Contents

Classes

EDAS Helper class that provides a standard way to create an ABC using inheritance.

class pyrepo_mcda.mcda_methods.edas.EDAS

Bases: pyrepo_mcda.mcda_methods.mcda_method.MCDA_method

Helper class that provides a standard way to create an ABC using inheritance.

__call__(self, matrix, weights, types)

Score alternatives provided in decision matrix matrix using criteria weights and criteria types.

Parameters

- **matrix** (ndarray) – Decision matrix with m alternatives in rows and n criteria in columns.
• **weights** (*ndarray*) – Vector with criteria weights. Sum of weights must be equal to 1.

• **types** (*ndarray*) – Vector with criteria types. Profit criteria are represented by 1 and cost by -1.

**Returns** Vector with preference values of each alternative. The best alternative has the highest preference value.

**Return type** *ndarray*

**Examples**

```python
>>> edas = EDAS()
>>> pref = edas(matrix, weights, types)
>>> rank = rank_preferences(pref, reverse = True)
```

**_edas_(matrix, weights, types)**

**pyrepo_mcda.mcda_methods.mabac**

**Module Contents**

**Classes**

**MABAC**

```python
class pyrepo_mcda.mcda_methods.mabac.MABAC(normalization_method=minmax_normalization)
Bases: pyrepo_mcda.mcda_methods.mcda_method.MCDA_method

__call__(self, matrix, weights, types)
Score alternatives provided in decision matrix *matrix* using criteria *weights* and criteria *types*.

**Parameters**

• **matrix** (*ndarray*) – Decision matrix with m alternatives in rows and n criteria in columns.

• **weights** (*ndarray*) – Vector with criteria weights. Sum of weights must be equal to 1.

• **types** (*ndarray*) – Vector with criteria types. Profit criteria are represented by 1 and cost by -1.

**Returns** Vector with preference values of each alternative. The best alternative has the highest preference value.

**Return type** *ndarray*
```
Examples

```python
>>> mabac = MABAC(normalization_method = minmax_normalization)
>>> pref = mabac(matrix, weights, types)
>>> rank = rank_preferences(pref, reverse = True)
```

```python
_mabac(matrix, weights, types, normalization_method)
```

### pyrepo_mcda.mcda_methods.mcda_method

#### Module Contents

#### Classes

```python
 MCDA_method
 Helper class that provides a standard way to create an ABC using
```

```python
class pyrepo_mcda.mcda_methods.mcda_method.MCDA_method
    Bases: abc.ABC
    Helper class that provides a standard way to create an ABC using inheritance.

    __call__(self, matrix, weights, types)
    Score alternatives from decision matrix `matrix` using criteria weights `weights` and criteria types `types`

    Parameters
    ----------
    * `matrix` *(ndarray)* – decision matrix with performance values for m alternatives in rows and n criteria in columns
    * `weights` *(ndarray)* – vector with criteria weights with size equal to number of columns n of `matrix`
    * `types` *(ndarray)* – vector with criteria types containing values of 1 for profit criteria and -1 for cost criteria with size equal to number of columns n of `matrix`
```

```python
static _verify_input_data(matrix, weights, types)
```

### pyrepo_mcda.mcda_methods.multimoora

#### Module Contents

#### Classes

```python
 MULTIMOORA_RS
 MULTIMOORA_RP
 MULTIMOORA_FMF
 MULTIMOORA
```

1.3. API Reference 45
class pyrepo_mcda.mcda_methods.multimoora.MULTIMOORA_RS
    Bases: pyrepo_mcda.mcda_methods.mcda_method.MCDA_method

    __call__(self, matrix, weights, types)
    Score alternatives provided in decision matrix matrix using vector with criteria weights weights and vector with criteria types types.

    Parameters
    • matrix (ndarray) – Decision matrix with m alternatives in rows and n criteria in columns.
    • weights (ndarray) – Criteria weights. Sum of weights must be equal to 1.
    • types (ndarray) – Criteria types. Profit criteria are represented by 1 and cost by -1.

    Returns Preference values of each alternative. The best alternative has the highest preference value.

    Return type ndarray

Examples

>>> multimoora_rs = MULTIMOORA_RS()
>>> pref = multimoora_rs(matrix, weights, types)
>>> rank = rank_preferences(pref, reverse = True)

static _multimoora_rs(matrix, weights, types)

class pyrepo_mcda.mcda_methods.multimoora.MULTIMOORA_RP
    Bases: pyrepo_mcda.mcda_methods.mcda_method.MCDA_method

    __call__(self, matrix, weights, types)
    Score alternatives provided in decision matrix matrix using vector with criteria weights weights and vector with criteria types types.

    Parameters
    • matrix (ndarray) – Decision matrix with m alternatives in rows and n criteria in columns.
    • weights (ndarray) – Criteria weights. Sum of weights must be equal to 1.
    • types (ndarray) – Criteria types. Profit criteria are represented by 1 and cost by -1.

    Returns Preference values of each alternative. The best alternative has the lowest preference value.

    Return type ndarray

Examples

>>> multimoora_rp = MULTIMOORA_RP()
>>> pref = multimoora_rp(matrix, weights, types)
>>> rank = rank_preferences(pref, reverse = False)

static _multimoora_rp(matrix, weights, types)

class pyrepo_mcda.mcda_methods.multimoora.MULTIMOORA_FMF
    Bases: pyrepo_mcda.mcda_methods.mcda_method.MCDA_method
__call__(self, matrix, weights, types)

Score alternatives provided in decision matrix matrix using vector with criteria weights weights and vector with criteria types types.

Parameters

- matrix (ndarray) – Decision matrix with m alternatives in rows and n criteria in columns.
- weights (ndarray) – Criteria weights. Sum of weights must be equal to 1.
- types (ndarray) – Criteria types. Profit criteria are represented by 1 and cost by -1.

Returns Preference values of each alternative. The best alternative has the highest preference value.

Return type ndarray

Examples

```python
>>> multimoora_fmf = MULTIMOORA_FMF()
>>> pref = multimoora_fmf(matrix, weights, types)
>>> rank = rank_preferences(pref, reverse = True)
```

static _multimoora_fmf(matrix, weights, types)

class pyrepo_mcda.mcda_methods.multimoora.MULTIMOORA(compromise_rank_method=dominance_directed_graph)

Bases: pyrepo_mcda.mcda_methods.mcda_method.MCDA_method

__call__(self, matrix, weights, types)

Score alternatives provided in decision matrix matrix using vector with criteria weights weights and vector with criteria types types.

Parameters

- matrix (ndarray) – Decision matrix with m alternatives in rows and n criteria in columns.
- weights (ndarray) – Criteria weights. Sum of weights must be equal to 1.
- types (ndarray) – Criteria types. Profit criteria are represented by 1 and cost by -1.

Returns Preference values of each alternative. The best alternative has the highest preference value.

Return type ndarray

Examples

```python
>>> multimoora = MULTIMOORA()
>>> rank = multimoora(matrix, weights, types)
```

_multimoora(matrix, weights, types, compromise_rank_method)
class pyrepo_mcda.mcda_methods.spotis.SPOTIS

Bases: pyrepo_mcda.mcda_methods.mcda_method.MCDA_method

__call__(self, matrix, weights, types, bounds)

Score alternatives provided in decision matrix matrix using criteria weights and criteria types.

Parameters

- **matrix** (*ndarray*) – Decision matrix with m alternatives in rows and n criteria in columns.
- **weights** (*ndarray*) – Vector with criteria weights. Sum of weights must be equal to 1.
- **types** (*ndarray*) – Vector with criteria types. Profit criteria are represented by 1 and cost by -1.
- **bounds** (*ndarray*) – Bounds is ndarray with 2 rows and number of columns equal to criteria number. Bounds contain minimum values in the first row and maximum values in the second row for each criterion. Minimum and maximum values for the same criterion cannot be the same.

Returns Vector with preference values of each alternative. The best alternative has the lowest preference value.

Return type *ndarray*

Examples

```python
>>> bounds_min = np.amin(matrix, axis = 0)
>>> bounds_max = np.amax(matrix, axis = 0)
>>> bounds = np.vstack((bounds_min, bounds_max))
>>> spotis = SPOTIS()
>>> pref = spotis(matrix, weights, types, bounds)
>>> rank = rank_preferences(pref, reverse = False)
```

static _spotis(matrix, weights, types, bounds)
Module Contents

Classes

**TOPSIS**

Helper class that provides a standard way to create an ABC using inheritance.

```python
class pyrepo_mcda.mcda_methods.topsis.TOPSIS(normalization_method=minmax_normalization, distance_metric=euclidean):
    Bases: pyrepo_mcda.mcda_methods.mcda_method.MCDA_method

    Helper class that provides a standard way to create an ABC using inheritance.

    __call__(self, matrix, weights, types)

    Score alternatives provided in decision matrix `matrix` with m alternatives in rows and n criteria in columns using criteria `weights` and criteria `types`.

    Parameters

    - `matrix` *(ndarray)* – Decision matrix with m alternatives in rows and n criteria in columns.
    - `weights` *(ndarray)* – Vector with criteria weights. Sum of weights must be equal to 1.
    - `types` *(ndarray)* – Vector with criteria types. Profit criteria are represented by 1 and cost by -1.

    Returns

    Vector with preference values of each alternative. The best alternative has the highest preference value.

    Return type `ndarray`

    Examples

    ```python
    >>> topsis = TOPSIS(normalization_method = minmax_normalization, distance_metric = euclidean)
    >>> pref = topsis(matrix, weights, types)
    >>> rank = rank_preferences(pref, reverse = True)
    ```
```

**VIKOR**

Module Contents

Classes

```
class pyrepo_mcda.mcda_methods.vikor.VIKOR(normalization_method=None, v=0.5)
    Bases: pyrepo_mcda.mcda_methods.mcda_method.MCDA_method
    __call__(self, matrix, weights, types)
    Score alternatives provided in decision matrix matrix using criteria weights and criteria types.

    Parameters
    • matrix (ndarray) – Decision matrix with m alternatives in rows and n criteria in columns.
    • weights (ndarray) – Vector with criteria weights. Sum of weights must be equal to 1.
    • types (ndarray) – Vector with criteria types. Profit criteria are represented by 1 and cost by -1.

    Returns Vector with preference values of each alternative. The best alternative has the lowest preference value.

    Return type ndarray

Examples

>>> vikor = VIKOR(normalization_method = minmax_normalization)
>>> pref = vikor(matrix, weights, types)
>>> rank = rank_preferences(pref, reverse = False)

static _vikor(matrix, weights, types, normalization_method, v)

pyrepo_mcda.mcda_methods.waspas

Module Contents

Classes

WASPAS Helper class that provides a standard way to create an ABC using inheritance.

class pyrepo_mcda.mcda_methods.waspas.WASPAS(normalization_method=linear_normalization, lambda_param=0.5)
    Bases: pyrepo_mcda.mcda_methods.mcda_method.MCDA_method
    Helper class that provides a standard way to create an ABC using inheritance.
    __call__(self, matrix, weights, types)
    Score alternatives provided in decision matrix matrix with m alternatives and n criteria using criteria weights and criteria types.

    Parameters
    • matrix (ndarray) – Decision matrix with m alternatives in rows and n criteria in columns.
    • weights (ndarray) – Vector with criteria weights. Sum of weights must be equal to 1.
    • types (ndarray) – Vector with criteria types. Profit criteria are represented by 1 and cost by -1.
**Returns**  Vector with preference values of each alternative. The best alternative has the highest preference value.

**Return type**  ndarray

**Examples**

```python
>>> waspas = WASPAS(normalization_method = linear_normalization, lambda_param = 0.5)
>>> pref = waspas(matrix, weights, types)
>>> rank = rank_preferences(pref, reverse = True)
```

```python
static _waspas(matrix, weights, types, normalization_method, lambda_param)
```

**Package Contents**

**Classes**

- **CODAS**  Helper class that provides a standard way to create an ABC using
- **EDAS**  Helper class that provides a standard way to create an ABC using
- **MABAC**
- **MULTIMOORA**
- **MULTIMOORA_RS**
- **SPOTIS**
- **TOPSIS**  Helper class that provides a standard way to create an ABC using
- **VIKOR**
- **WASPAS**  Helper class that provides a standard way to create an ABC using

**class**  `pyrepo_mcda.mcda_methods.CODAS(normalization_method=linear_normalization, distance_metric=euclidean, tau=0.02)`

**Bases:**  `pyrepo_mcda.mcda_methods.mcda_method.MCDA_method`

Helper class that provides a standard way to create an ABC using inheritance.

```python
__call__(self, matrix, weights, types)
```

Score alternatives provided in decision matrix `matrix` with m alternatives and n criteria using criteria `weights` and criteria `types`.

**Parameters**

- **matrix (ndarray)** – Decision matrix with m alternatives in rows and n criteria in columns.
- **weights (ndarray)** – Vector with criteria weights. Sum of weights must be equal to 1.
• **types** (*ndarray*) – Vector with criteria types. Profit criteria are represented by 1 and cost by -1.

**Returns** Vector with preference values of each alternative. The best alternative has the highest preference value.

**Return type** *ndarray*

**Examples**

```python
>>> codas = CODAS(normalization_method = linear_normalization, distance_metric = euclidean, tau = 0.02)
>>> pref = codas(matrix, weights, types)
>>> rank = rank_preferences(pref, reverse = True)
```

```python
_defas(matrix, weights, types)
```

**class** `pyrepo_mcda.mcda_methods.EDAS`

**Bases:** `pyrepo_mcda.mcda_methods.mcda_method.MCDA_method`

Helper class that provides a standard way to create an ABC using inheritance.

**__call__** *(self, matrix, weights, types)*

Score alternatives provided in decision matrix *matrix* using criteria *weights* and criteria *types*.

**Parameters**

- **matrix** (*ndarray*) – Decision matrix with *m* alternatives in rows and *n* criteria in columns.
- **weights** (*ndarray*) – Vector with criteria weights. Sum of weights must be equal to 1.
- **types** (*ndarray*) – Vector with criteria types. Profit criteria are represented by 1 and cost by -1.

**Returns** Vector with preference values of each alternative. The best alternative has the highest preference value.

**Return type** *ndarray*

**Examples**

```python
>>> edas = EDAS()
>>> pref = edas(matrix, weights, types)
>>> rank = rank_preferences(pref, reverse = True)
```

```python
_defas(matrix, weights, types)
```

**class** `pyrepo_mcda.mcda_methods.MABAC` *(normalization_method=minmax_normalization)*

**Bases:** `pyrepo_mcda.mcda_methods.mcda_method.MCDA_method`

**__call__** *(self, matrix, weights, types)*

Score alternatives provided in decision matrix *matrix* using criteria *weights* and criteria *types*.

**Parameters**

- **matrix** (*ndarray*) – Decision matrix with *m* alternatives in rows and *n* criteria in columns.
• **weights** (**ndarray**) – Vector with criteria weights. Sum of weights must be equal to 1.

• **types** (**ndarray**) – Vector with criteria types. Profit criteria are represented by 1 and cost by -1.

**Returns** Vector with preference values of each alternative. The best alternative has the highest preference value.

**Return type**  **ndrarray**

**Examples**

```python
>>> mabac = MABAC(normalization_method = minmax_normalization)
>>> pref = mabac(matrix, weights, types)
>>> rank = rank_preferences(pref, reverse = True)
```

_mabac_(**matrix**, **weights**, **types**, **normalization_method**)

**class** pyrepo.mcda.mcda_methods.MULTIMOORA(**compromise_rank_method**=dominance_directed_graph)

**Bases:** pyrepo.mcda.mcda_methods.mcda_method.MCDA_method

**__call__**(self, **matrix**, **weights**, **types**)

Score alternatives provided in decision matrix **matrix** using vector with criteria weights **weights** and vector with criteria types **types**.

**Parameters**

• **matrix** (**ndarray**) – Decision matrix with m alternatives in rows and n criteria in columns.

• **weights** (**ndarray**) – Criteria weights. Sum of weights must be equal to 1.

• **types** (**ndarray**) – Criteria types. Profit criteria are represented by 1 and cost by -1.

**Returns** Preference values of each alternative. The best alternative has the highest preference value.

**Return type**  **ndrarray**

**Examples**

```python
>>> multimoora = MULTIMOORA()
>>> rank = multimoora(matrix, weights, types)
```

_multimoora_(**matrix**, **weights**, **types**, **compromise_rank_method**)

**class** pyrepo.mcda.mcda_methods.MULTIMOORA_RS

**Bases:** pyrepo.mcda.mcda_methods.mcda_method.MCDA_method

**__call__**(self, **matrix**, **weights**, **types**)

Score alternatives provided in decision matrix **matrix** using vector with criteria weights **weights** and vector with criteria types **types**.

**Parameters**

• **matrix** (**ndarray**) – Decision matrix with m alternatives in rows and n criteria in columns.

• **weights** (**ndarray**) – Criteria weights. Sum of weights must be equal to 1.

• **types** (**ndarray**) – Criteria types. Profit criteria are represented by 1 and cost by -1.
Returns Preference values of each alternative. The best alternative has the highest preference value.

Return type ndarray

Examples

```python
g>>> multimoora_rs = MULTIMOORA_RS()
g>>> pref = multimoora_rs(matrix, weights, types)
g>>> rank = rank_preferences(pref, reverse = True)
```
__call__(self, matrix, weights, types)
Score alternatives provided in decision matrix matrix with m alternatives in rows and n criteria in columns
using criteria weights and criteria types.

Parameters

- **matrix** (ndarray) – Decision matrix with m alternatives in rows and n criteria in columns.
- **weights** (ndarray) – Vector with criteria weights. Sum of weights must be equal to 1.
- **types** (ndarray) – Vector with criteria types. Profit criteria are represented by 1 and cost by -1.

Returns Vector with preference values of each alternative. The best alternative has the highest preference value.

Return type ndarrray

Examples

```python
>>> topsis = TOPSIS(normalization_method = minmax_normalization, distance_metric = euclidean)
>>> pref = topsis(matrix, weights, types)
>>> rank = rank_preferences(pref, reverse = True)
```

static _topsis(matrix, weights, types, normalization_method, distance_metric)

class pyrepo_mcda.mcda_methods.VIKOR(normalization_method=None, v=0.5)
Bases: pyrepo_mcda.mcda_methods.mcda_method.MCDA_method

__call__(self, matrix, weights, types)
Score alternatives provided in decision matrix matrix using criteria weights and criteria types.

Parameters

- **matrix** (ndarray) – Decision matrix with m alternatives in rows and n criteria in columns.
- **weights** (ndarray) – Vector with criteria weights. Sum of weights must be equal to 1.
- **types** (ndarray) – Vector with criteria types. Profit criteria are represented by 1 and cost by -1.

Returns Vector with preference values of each alternative. The best alternative has the lowest preference value.

Return type ndarrray

Examples

```python
>>> vikor = VIKOR(normalization_method = minmax_normalization)
>>> pref = vikor(matrix, weights, types)
>>> rank = rank_preferences(pref, reverse = False)
```

static _vikor(matrix, weights, types, normalization_method, v)
class pyrepo_mcda.mcda_methods.WASPAS(
    normalization_method=linear_normalization,
    lambda_param=0.5)

Bases: pyrepo_mcda.mcda_methods.mcda_method.MCDA_method

Helper class that provides a standard way to create an ABC using inheritance.

__call__(self, matrix, weights, types)

Score alternatives provided in decision matrix matrix with m alternatives and n criteria using criteria weights and criteria types.

Parameters

- **matrix** (ndarray) – Decision matrix with m alternatives in rows and n criteria in columns.
- **weights** (ndarray) – Vector with criteria weights. Sum of weights must be equal to 1.
- **types** (ndarray) – Vector with criteria types. Profit criteria are represented by 1 and cost by -1.

Returns Vector with preference values of each alternative. The best alternative has the highest preference value.

Return type ndarray

Examples

```python
>>> waspas = WASPAS(normalization_method = linear_normalization, lambda_param = 0.5)
>>> pref = waspas(matrix, weights, types)
>>> rank = rank_preferences(pref, reverse = True)
```

**static _waspas(matrix, weights, types, normalization_method, lambda_param)**

Submodules

pyrepo_mcda.additions

Module Contents

Functions

**rank_preferences(pref, reverse=True)**

Rank alternatives according to MCDA preference function values. If more than one alternative have the same preference function value, they will be given the same rank value (tie).

Parameters

- **pref** (ndarray) – Vector with MCDA preference function values for alternatives
- **reverse** (bool) – The boolean variable is True for MCDA methods that rank alternatives in descending order (for example, TOPSIS, CODAS) and False for MCDA methods that rank alternatives in ascending order (for example, VIKOR, SPOTIS)
**Returns**  Vector with alternatives ranking. Alternative with 1 value is the best and has the first position in the ranking.

**Return type**  ndarray

---

**Examples**

```python
>>> rank = rank_preferences(pref, reverse = True)
```

---

**pyrepo_mcda.compromise_rankings**

**Module Contents**

**Functions**

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>copeland(matrix)</code></td>
<td>Calculate the compromise ranking considering several rankings obtained using different methods using the Copeland compromise ranking methodology.</td>
</tr>
<tr>
<td><code>dominance_directed_graph(matrix)</code></td>
<td>Calculate the compromise ranking considering several rankings obtained using different methods using Dominance Directed Graph methodology.</td>
</tr>
<tr>
<td><code>rank_position_method(matrix)</code></td>
<td>Calculate the compromise ranking considering several rankings obtained using different methods using the Rank Position Method.</td>
</tr>
<tr>
<td><code>improved_borda_rule(prefs, ranks)</code></td>
<td>Calculate the compromise ranking considering several rankings obtained using different methods using the Improved Borda Rule.</td>
</tr>
</tbody>
</table>

---

**pyrepo_mcda.compromise_rankings.copeland(matrix)**

Calculate the compromise ranking considering several rankings obtained using different methods using the Copeland compromise ranking methodology.

**Parameters**

- `matrix (ndarray)`: Two-dimensional matrix containing different rankings in columns.

**Returns**  Vector including compromise ranking.

**Return type**  ndarray

**Examples**

```python
>>> rank = copeland(matrix)
```

---

**pyrepo_mcda.compromise_rankings.dominance_directed_graph(matrix)**

Calculate the compromise ranking considering several rankings obtained using different methods using Dominance Directed Graph methodology.

**Parameters**

- `matrix (ndarray)`: Two-dimensional matrix containing different rankings in columns.

**Returns**  Vector including compromise ranking.

**Return type**  ndarray
Examples

```python
>>> rank = dominance_directed_graph(matrix)
```

pyrepo_mcda.compromise_rankings.rank_position_method(matrix)
Calculate the compromise ranking considering several rankings obtained using different methods using Rank Position Method

Parameters

- **matrix (ndarray)** – Two-dimensional matrix containing different rankings in columns.

Returns

- Vector including compromise ranking.

Return type

- ndarray

Examples

```python
>>> rank = rank_position_method(matrix)
```

pyrepo_mcda.compromise_rankings.improved_borda_rule(prefs, ranks)
Calculate the compromise ranking considering several rankings obtained using different methods using Improved Borda rule methodology

Parameters

- **prefs (ndarray)** – Two-dimensional matrix containing preferences calculated by different methods in columns.
- **ranks (ndarray)** – Two-dimensional matrix containing rankings determined by different methods in columns.

Returns

- Vector including compromise ranking.

Return type

- ndarray

Examples

```python
>>> rank = improved_borda_rule(prefs, ranks)
```

pyrepo_mcda.correlations

Module Contents

Functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>spearman(R, Q)</td>
<td>Calculate Spearman rank correlation coefficient between two vectors</td>
</tr>
<tr>
<td>weighted_spearman(R, Q)</td>
<td>Calculate Weighted Spearman rank correlation coefficient between two vectors</td>
</tr>
<tr>
<td>pearson_coeff(R, Q)</td>
<td>Calculate Pearson correlation coefficient between two vectors</td>
</tr>
<tr>
<td>WS_coeff(R, Q)</td>
<td>Calculate Rank similarity coefficient between two vectors</td>
</tr>
</tbody>
</table>
Calculate Spearman rank correlation coefficient between two vectors

Parameters

• \textit{R} (\texttt{ndarray}) – First vector containing values
• \textit{Q} (\texttt{ndarray}) – Second vector containing values

Returns Value of correlation coefficient between two vectors

Return type float

Examples

```python
>>> rS = spearman(R, Q)
```

Calculate Weighted Spearman rank correlation coefficient between two vectors

Parameters

• \textit{R} (\texttt{ndarray}) – First vector containing values
• \textit{Q} (\texttt{ndarray}) – Second vector containing values

Returns Value of correlation coefficient between two vectors

Return type float

Examples

```python
>>> rW = weighted_spearman(R, Q)
```

Calculate Pearson correlation coefficient between two vectors

Parameters

• \textit{R} (\texttt{ndarray}) – First vector containing values
• \textit{Q} (\texttt{ndarray}) – Second vector containing values

Returns Value of correlation coefficient between two vectors

Return type float

Examples

```python
>>> corr = pearson_coeff(R, Q)
```

Calculate Rank similarity coefficient between two vectors

Parameters

• \textit{R} (\texttt{ndarray}) – First vector containing values
• \textit{Q} (\texttt{ndarray}) – Second vector containing values
Returns Value of similarity coefficient between two vectors

Return type float

Examples

```python
>>> ws = WS_coeff(R, Q)
```

`pyrepo_mcda.distance_metrics`

Module Contents
Functions

**euclidean**(A, B) Calculate Euclidean distance between two vectors A and B.

**manhattan**(A, B) Calculate Manhattan (Taxicab) distance between two vectors A and B.

**hausdorff_distance**(A, B) Calculate Hausdorff distance between two vectors A and B.

**correlation**(A, B) Calculate Correlation distance between two vectors A and B.

**chebyshev**(A, B) Calculate Chebyshev distance between two vectors A and B.

**std_euclidean**(A, B) Calculate Standardized Euclidean distance between two vectors A and B.

**cosine**(A, B) Calculate Cosine distance between two vectors A and B.

**csm**(A, B) Calculate Cosine similarity measure of distance between two vectors A and B.

**squared_euclidean**(A, B) Calculate Squared Euclidean distance between two vectors A and B.

**bray_curtis**(A, B) Calculate Bray-Curtis distance between two vectors A and B.

**canberra**(A, B) Calculate Canberra distance between two vectors A and B.

**lorentzian**(A, B) Calculate Lorentzian distance between two vectors A and B.

**jaccard**(A, B) Calculate Jaccard distance between two vectors A and B.

**dice**(A, B) Calculate Dice distance between two vectors A and B.

**bhattacharyya**(A, B) Calculate Bhattacharyya distance between two vectors A and B.

**hellinger**(A, B) Calculate Hellinger distance between two vectors A and B.

**matusita**(A, B) Calculate Matusita distance between two vectors A and B.

**squared_chord**(A, B) Calculate Squared-Chord distance between two vectors A and B.

**pearson_chi_square**(A, B) Calculate Pearson Chi Square distance between two vectors A and B.

**squared_chi_square**(A, B) Calculate Squared Chi Square distance between two vectors A and B.

**euclidean**(A, B)
Calculate Euclidean distance between two vectors A and B.

**Parameters**

- **A** (*ndarray*) – First vector containing values
- **B** (*ndarray*) – Second vector containing values

**Returns**
distance value between two vectors

**Return type** float
Examples

```python
>>> distance = euclidean(A, B)
```

`pyrepo_mcda.distance_metrics.manhattan(A, B)`

Calculate Manhattan (Taxicab) distance between two vectors \( A \) and \( B \).

**Parameters**
- \( A \) (*ndarray*) – First vector containing values
- \( B \) (*ndarray*) – Second vector containing values

**Returns** distance value between two vectors

**Return type** float

Examples

```python
>>> distance = manhattan(A, B)
```

`pyrepo_mcda.distance_metrics.hausdorff_distance(A, B)`

`pyrepo_mcda.distance_metrics.hausdorff(A, B)`

Calculate Hausdorff distance between two vectors \( A \) and \( B \).

**Parameters**
- \( A \) (*ndarray*) – First vector containing values
- \( B \) (*ndarray*) – Second vector containing values

**Returns** distance value between two vectors

**Return type** float

Examples

```python
>>> distance = hausdorff(A, B)
```

`pyrepo_mcda.distance_metrics.correlation(A, B)`

Calculate Correlation distance between two vectors \( A \) and \( B \).

**Parameters**
- \( A \) (*ndarray*) – First vector containing values
- \( B \) (*ndarray*) – Second vector containing values

**Returns** distance value between two vectors

**Return type** float
Examples

```python
distance = correlation(A, B)
```

pyrepo_mcda.distance_metrics.chebyshev(A, B)
Calculate Chebyshev distance between two vectors $A$ and $B$.

Parameters
- A (ndarray) – First vector containing values
- B (ndarray) – Second vector containing values

Returns distance value between two vectors
Return type float

Examples

```python
distance = chebyshev(A, B)
```

pyrepo_mcda.distance_metrics.std_euclidean(A, B)
Calculate Standardized Euclidean distance between two vectors $A$ and $B$.

Parameters
- A (ndarray) – First vector containing values
- B (ndarray) – Second vector containing values

Returns distance value between two vectors
Return type float

Examples

```python
distance = std_euclidean(A, B)
```

pyrepo_mcda.distance_metrics.cosine(A, B)
Calculate Cosine distance between two vectors $A$ and $B$.

Parameters
- A (ndarray) – First vector containing values
- B (ndarray) – Second vector containing values

Returns distance value between two vectors
Return type float
Examples

```python
>>> distance = cosine(A, B)
```

```
pyrepo_mcda.distance_metrics.csm(A, B)
```

Calculate Cosine similarity measure of distance between two vectors A and B.

**Parameters**

- **A** (*ndarray*) – First vector containing values
- **B** (*ndarray*) – Second vector containing values

**Returns**

distance value between two vectors

**Return type**

*float*

Examples

```python
>>> distance = csm(A, B)
```

```
pyrepo_mcda.distance_metrics.squared_euclidean(A, B)
```

Calculate Squared Euclidean distance between two vectors A and B.

**Parameters**

- **A** (*ndarray*) – First vector containing values
- **B** (*ndarray*) – Second vector containing values

**Returns**

distance value between two vectors

**Return type**

*float*

Examples

```python
>>> distance = squared_euclidean(A, B)
```

```
pyrepo_mcda.distance_metrics.bray_curtis(A, B)
```

Calculate Bray-Curtis distance between two vectors A and B.

**Parameters**

- **A** (*ndarray*) – First vector containing values
- **B** (*ndarray*) – Second vector containing values

**Returns**

distance value between two vectors

**Return type**

*float*
Examples

```python
>>> distance = bray_curtis(A, B)
```

`pyrepo_mcda.distance_metrics.canberra(A, B)`
Calculate Canberra distance between two vectors A and B.

**Parameters**

- A (ndarray) – First vector containing values
- B (ndarray) – Second vector containing values

**Returns** distance value between two vectors

**Return type** float

Examples

```python
>>> distance = canberra(A, B)
```

`pyrepo_mcda.distance_metrics.lorentzian(A, B)`
Calculate Lorentzian distance between two vectors A and B.

**Parameters**

- A (ndarray) – First vector containing values
- B (ndarray) – Second vector containing values

**Returns** distance value between two vectors

**Return type** float

Examples

```python
>>> distance = lorentzian(A, B)
```

`pyrepo_mcda.distance_metrics.jaccard(A, B)`
Calculate Jaccard distance between two vectors A and B.

**Parameters**

- A (ndarray) – First vector containing values
- B (ndarray) – Second vector containing values

**Returns** distance value between two vectors

**Return type** float
Examples

```python
>>> distance = jaccard(A, B)
```

`pyrepo_mcda.distance_metrics.dice(A, B)`
Calculate Dice distance between two vectors `A` and `B`.

**Parameters**
- `A (ndarray)` – First vector containing values
- `B (ndarray)` – Second vector containing values

**Returns** distance value between two vectors

**Return type** `float`

Examples

```python
>>> distance = dice(A, B)
```

`pyrepo_mcda.distance_metrics.bhattacharyya(A, B)`
Calculate Bhattacharyya distance between two vectors `A` and `B`.

**Parameters**
- `A (ndarray)` – First vector containing values
- `B (ndarray)` – Second vector containing values

**Returns** distance value between two vectors

**Return type** `float`

Examples

```python
>>> distance = bhattacharyya(A, B)
```

`pyrepo_mcda.distance_metrics.hellinger(A, B)`
Calculate Hellinger distance between two vectors `A` and `B`.

**Parameters**
- `A (ndarray)` – First vector containing values
- `B (ndarray)` – Second vector containing values

**Returns** distance value between two vectors

**Return type** `float`
Examples

```python
>>> distance = hellinger(A, B)
```

pyrepo_mcda.distance_metrics.matusita(A, B)

Calculate Matusita distance between two vectors A and B.

Parameters

• A (ndarray) – First vector containing values
• B (ndarray) – Second vector containing values

Returns distance value between two vectors

Return type float

Examples

```python
>>> distance = matusita(A, B)
```

pyrepo_mcda.distance_metrics.squared_chord(A, B)

Calculate Squared-Chord distance between two vectors A and B.

Parameters

• A (ndarray) – First vector containing values
• B (ndarray) – Second vector containing values

Returns distance value between two vectors

Return type float

Examples

```python
>>> distance = squared_chord(A, B)
```

pyrepo_mcda.distance_metrics.pearson_chi_square(A, B)

Calculate Pearson Chi Square distance between two vectors A and B.

Parameters

• A (ndarray) – First vector containing values
• B (ndarray) – Second vector containing values

Returns distance value between two vectors

Return type float
Examples

```python
>>> distance = pearson_chi_square(A, B)
```

```python
pyrepo_mcda.distance_metrics.squared_chi_square(A, B)
```

Calculate Squared Chi Square distance between two vectors \( A \) and \( B \).

**Parameters**
- \( A \) (ndarray) – First vector containing values
- \( B \) (ndarray) – Second vector containing values

**Returns**
- distance value between two vectors

**Return type**
- float

Examples

```python
>>> distance = squared_chi_square(A, B)
```

**Module Contents**

**(Functions)**

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>linear_normalization</code></td>
<td>Normalize decision matrix using linear normalization method.</td>
</tr>
<tr>
<td><code>minmax_normalization</code></td>
<td>Normalize decision matrix using minimum-maximum normalization method.</td>
</tr>
<tr>
<td><code>max_normalization</code></td>
<td>Normalize decision matrix using maximum normalization method.</td>
</tr>
<tr>
<td><code>sum_normalization</code></td>
<td>Normalize decision matrix using sum normalization method.</td>
</tr>
<tr>
<td><code>vector_normalization</code></td>
<td>Normalize decision matrix using vector normalization method.</td>
</tr>
<tr>
<td><code>multimoora_normalization</code></td>
<td>Normalize decision matrix using vector normalization method as for profit criteria.</td>
</tr>
</tbody>
</table>

**pyrepo_mcda.normalizations.linear_normalization(matrix, types)**

Normalize decision matrix using linear normalization method.

**Parameters**
- matrix (ndarray) – Decision matrix with \( m \) alternatives in rows and \( n \) criteria in columns
- types (ndarray) – Criteria types. Profit criteria are represented by 1 and cost by -1.

**Returns**
- Normalized decision matrix

**Return type**
- ndarray
Examples

```python
>>> nmatrix = linear_normalization(matrix, types)
```

`pyrepo_mcda.normalizations.minmax_normalization(matrix, types)`

Normalize decision matrix using minimum-maximum normalization method.

Parameters

- `matrix` (*ndarray*) – Decision matrix with m alternatives in rows and n criteria in columns
- `types` (*ndarray*) – Criteria types. Profit criteria are represented by 1 and cost by -1.

Returns

Normalized decision matrix

Return type `ndarray`

Examples

```python
>>> nmatrix = minmax_normalization(matrix, types)
```

`pyrepo_mcda.normalizations.max_normalization(matrix, types)`

Normalize decision matrix using maximum normalization method.

Parameters

- `matrix` (*ndarray*) – Decision matrix with m alternatives in rows and n criteria in columns
- `types` (*ndarray*) – Criteria types. Profit criteria are represented by 1 and cost by -1.

Returns

Normalized decision matrix

Return type `ndarray`

Examples

```python
>>> nmatrix = max_normalization(matrix, types)
```

`pyrepo_mcda.normalizations.sum_normalization(matrix, types)`

Normalize decision matrix using sum normalization method.

Parameters

- `matrix` (*ndarray*) – Decision matrix with m alternatives in rows and n criteria in columns
- `types` (*ndarray*) – Criteria types. Profit criteria are represented by 1 and cost by -1.

Returns

Normalized decision matrix

Return type `ndarray`
Examples

```python
>>> nmatrix = sum_normalization(matrix, types)
```

`pyrepo_mcda.normalizations.sum_normalization(matrix, types)`

Normalize decision matrix using vector normalization method.

**Parameters**

- `matrix` *(ndarray)* – Decision matrix with m alternatives in rows and n criteria in columns
- `types` *(ndarray)* – Criteria types. Profit criteria are represented by 1 and cost by -1.

**Returns** Normalized decision matrix

**Return type** ndarray

Examples

```python
>>> nmatrix = vector_normalization(matrix, types)
```

`pyrepo_mcda.normalizations.vector_normalization(matrix, types)`

Normalize decision matrix using vector normalization method.

**Parameters**

- `matrix` *(ndarray)* – Decision matrix with m alternatives in rows and n criteria in columns
- `types` *(ndarray)* – Criteria types. Profit criteria are represented by 1 and cost by -1.

**Returns** Normalized decision matrix

**Return type** ndarray

Examples

```python
>>> nmatrix = multimoora_normalization(matrix)
```

`pyrepo_mcda.normalizations.multimoora_normalization(matrix)`

Normalize decision matrix using vector normalization method as for profit criteria.

**Parameters**

- `matrix` *(ndarray)* – Decision matrix with m alternatives in rows and n criteria in columns

Examples

```python
>>> nmatrix = multimoora_normalization(matrix)
```

`pyrepo_mcda.sensitivity_analysis_weights_percentages`
• **weights** (*ndarray*) – Vector with criteria weights. All weights in this vector must sum to 1.

• **types** (*ndarray*) – Vector with criteria types. Types can be equal to 1 for profit criteria and -1 for cost criteria.

• **percentages** (*ndarray*) – Vector with percentage values of given criteria weight modification in range from 0 to 1.

• **method** (*class*) – Initialized object of class of chosen MCDA method

• **j** (*int*) – Index of column in decision matrix `matrix` that indicates for which criterion the weight is modified.

• **dir_list** (*list*) – List with directions (signs of value) of criterion weight modification. 1 denotes increasing, and -1 denotes decreasing weight value. You can provide [-1, 1] for increasing and decreasing, [-1] for only decreasing, or [1] for only increasing chosen criterion weight.

**Returns** `data_sens` – dataframe with rankings calculated for subsequent modifications of criterion `j` weight

**Return type** DataFrame

**Examples**

```python
>>> sensitivity_analysis = Sensitivity_analysis_weights_percentages()
>>> df_sens = sensitivity_analysis(matrix, weights, types, percentages, method, ...
˓→j, [-1, 1])
```

**_change_weights**(self, j, weights, change_val)

Method for criteria weights modification in sensitivity analysis procedure.

**Parameters**

• **j** (*int*) – Index of column in decision matrix `matrix` that indicates for which criterion the weight is modified.

• **weights** (*ndarray*) – Vector of criteria weights

• **change_val** (*float*) – Percentage value of criterion weight modification in range from 0 to 1

**Returns** `weights_copy` – Vector with criteria weights after modification their values for sensitivity analysis

**Return type** *ndarray*

**_static_sensitivity_analysis_weights_percentages**(self, matrix, weights, types, percentages, method, list_alt_names, j, dir_list)
Module Contents

Classes

*Sensitivity_analysis_weights_values*

class pyrepo_mcda.sensitivity_analysis_weights_values.Sensitivity_analysis_weights_values

__call__ (self, matrix, weight_values, types, method, j)
Method for sensitivity analysis. This method determines rankings of alternatives using chosen MCDA method name `mcda_name` for the value of criterion \( j \) weight set as chosen `weight_value`.

Parameters

• \textbf{matrix (ndarray)} – Decision matrix with performance values of alternatives. This matrix includes data on alternatives in rows considering criteria in columns.

• \textbf{weight_values (ndarray)} – Vector with values to be set as the weight of chosen criterion in the sensitivity analysis procedure in range from 0 to 1.

• \textbf{types (ndarray)} – Vector with criteria types. Types must be equal to 1 for profit criteria and -1 for cost criteria.

• \textbf{method (class)} – Initialized object of class of chosen MCDA method

• \textbf{j (int)} – Index of the column in decision matrix `matrix` that indicates for which criterion the weight is set with chosen value.

Returns \textbf{data_sens} – dataframe with rankings calculated for subsequent changes of criterion \( j \) weight. Particular rankings for different weight values of criterion \( j \) are included in subsequent columns of the dataframe.

Return type DataFrame

Examples

```python
>>> sensitivity_analysis = Sensitivity_analysis_weights_values()
>>> df_sens = sensitivity_analysis(matrix, weight_values, types, method, j)
```

__change_weights__ (self, matrix, weight_value, j)
Method for criteria weights changing in sensitivity analysis procedure.

Parameters

• \textbf{matrix (ndarray)} – Decision matrix with performance values of alternatives. This matrix includes data on alternatives in rows considering criteria in columns

• \textbf{weight_value (float)} – Value in range from 0 to 1 to be set as the weight of chosen criterion with index \( j \).

• \textbf{j (int)} – Index of the column in decision matrix `matrix` indicating for which criterion the weight value is changed.

```python
```
Returns **new_weights** – Vector with criteria weights after changing their values for sensitivity analysis

**Return type**.ndarray

```python
static _sensitivity_analysis_weights_values(self, matrix, weight_values, types, method, list_alt_names, j)
```

### pyrepo_mcda.weighting_methods

**Module Contents**

**Functions**

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>equal_weighting(matrix)</code></td>
<td>Calculate criteria weights using objective Equal weighting method</td>
</tr>
<tr>
<td><code>entropy_weighting(matrix)</code></td>
<td>Calculate criteria weights using objective Entropy weighting method</td>
</tr>
<tr>
<td><code>std_weighting(matrix)</code></td>
<td>Calculate criteria weights using objective Standard deviation weighting method</td>
</tr>
<tr>
<td><code>critic_weighting(matrix)</code></td>
<td>Calculate criteria weights using objective CRITIC weighting method</td>
</tr>
</tbody>
</table>

**Examples**

```python
>>> weights = equal_weighting(matrix)
```

**pyrepo_mcda.weighting_methods.entropy_weighting(matrix)**

Calculate criteria weights using objective Entropy weighting method

**Parameters**

- `matrix` (ndarray) – Decision matrix with performance values of m alternatives and n criteria

**Returns**

- vector of criteria weights

**Return type**.

```python
>>> weights = entropy_weighting(matrix)
```
Examples

```python
>>> weights = entropy_weighting(matrix)
```

`pyrepo_mcda.weighting_methods.std_weighting(matrix)`

Calculate criteria weights using objective Standard deviation weighting method

**Parameters**
- `matrix` (ndarray) – Decision matrix with performance values of m alternatives and n criteria

**Returns**
- vector of criteria weights

**Return type**
- ndarray

Examples

```python
>>> weights = std_weighting(matrix)
```

`pyrepo_mcda.weighting_methods.critic_weighting(matrix)`

Calculate criteria weights using objective CRITIC weighting method

**Parameters**
- `matrix` (ndarray) – Decision matrix with performance values of m alternatives and n criteria

**Returns**
- vector of criteria weights

**Return type**
- ndarray

Examples

```python
>>> weights = critic_weighting(matrix)
```
p
pyrepo_mcda, 42
pyrepo_mcda.additions, 56
pyrepo_mcda.compromise_rankings, 57
pyrepo_mcda.correlations, 58
pyrepo_mcda.distance_metrics, 60
pyrepo_mcda.mcda_methods, 42
pyrepo_mcda.mcda_methods.codas, 42
pyrepo_mcda.mcda_methods.edas, 43
pyrepo_mcda.mcda_methods.mabac, 44
pyrepo_mcda.mcda_methods.mcda_method, 45
pyrepo_mcda.mcda_methods.multimoora, 45
pyrepo_mcda.mcda_methods.topsis, 49
pyrepo_mcda.mcda_methods.vikor, 49
pyrepo_mcda.mcda_methods.waspas, 50
pyrepo_mcda.normalizations, 68
pyrepo_mcda.sensitivity_analysis_weights_percentages, 70
pyrepo_mcda.sensitivity_analysis_weights_values, 72
pyrepo_mcda.weighting_methods, 73
Symbols

__call__(pyrepo_mcda.mcda_methods.CODAS method), 51
__call__(pyrepo_mcda.mcda_methods.EDAS method), 52
__call__(pyrepo_mcda.mcda_methods.MABAC method), 52
__call__(pyrepo_mcda.mcda_methods.MULTIMOORA method), 53
__call__(pyrepo_mcda.mcda_methods.MULTIMOORA_FMF method), 47
__call__(pyrepo_mcda.mcda_methods.MULTIMOORA_RP method), 46
__call__(pyrepo_mcda.mcda_methods.MULTIMOORA_RS method), 46
__call__(pyrepo_mcda.mcda_methods.SPOTIS method), 48
__call__(pyrepo_mcda.mcda_methods.TOPSIS method), 49
__call__(pyrepo_mcda.mcda_methods.VIKOR method), 50
__call__(pyrepo_mcda.mcda_methods.WASPAS method), 50
__call__(pyrepo_mcda.sensitivity_analysis_weights_percentages.Sensitivity_analysis_weights_percentages method), 70
__call__(pyrepo_mcda.sensitivity_analysis_weights_values.Sensitivity_analysis_weights_values method), 72
_change_weights(pyrepo_mcda.sensitivity_analysis_weights_percentages.Sensitivity_analysis_weights_percentages method), 71
_change_weights(pyrepo_mcda.sensitivity_analysis_weights_values.Sensitivity_analysis_weights_values method), 72
_codas(pyrepo_mcda.mcda_methods.CODAS static method), 52
_codas(pyrepo_mcda.mcda_methods.codas.CODAS static method), 43
_edas(pyrepo_mcda.mcda_methods.EDAS method), 52
_edas(pyrepo_mcda.mcda_methods.edas.EDAS method), 44
_mabac(pyrepo_mcda.mcda_methods.MABAC method), 53
_mabac(pyrepo_mcda.mcda_methods.mabac.MABAC method), 45
_multimoora(pyrepo_mcda.mcda_methods.MULTIMOORA method), 53
_multimoora(pyrepo_mcda.mcda_methods.multimoora.MULTIMOORA method), 47
_multimoora_fmf(pyrepo_mcda.mcda_methods.multimoora.MULTIMOORA_FMF static method), 47
_multimoora_rp(pyrepo_mcda.mcda_methods.multimoora.MULTIMOORA_RP static method), 46
_multimoora_rs(pyrepo_mcda.mcda_methods.multimoora.MULTIMOORA_RS static method), 46
_psi(pyrepo_mcda.mcda_methods.codas.CODAS method), 52
_psi(pyrepo_mcda.mcda_methods.codas.CODAS method), 43
_sensitivity_analysis_weights_percentages(pyrepo_mcda.sensitivity_analysis_weights_percentages.Sensitivity_analysis_weights_percentages method), 71
_sensitivity_analysis_weights_values(pyrepo_mcda.sensitivity_analysis_weights_values.Sensitivity_analysis_weights_values method), 77
pyrepo_mcda.mcda_methods.spotis, 48
pyrepo_mcda.mcda_methods.topsis, 49
pyrepo_mcda.mcda_methods.vikor, 49
pyrepo_mcda.mcda_methods.waspas, 50
pyrepo_mcda.normalizations, 68
pyrepo_mcda.sensitivity_analysis_weights_percentages, 70
pyrepo_mcda.sensitivity_analysis_weights_values, 72
pyrepo_mcda.weighting_methods, 73
MULTIMOORA (class in pyrepo_mcda.mcda_methods), 53
MULTIMOORA (class in pyrepo_mcda.mcda_methods.multimoora), 47
MULTIMOORA_FMFM (class in pyrepo_mcda.mcda_methods.multimoora), 46
multimoora_normalization() (in module pyrepo_mcda.normalizations), 70
MULTIMOORA_RP (class in pyrepo_mcda.mcda_methods.multimoora), 46
MULTIMOORA_RS (class in pyrepo_mcda.mcda_methods), 53
MULTIMOORA_RS (class in pyrepo_mcda.mcda_methods.multimoora), 46

P
pearson_chi_square() (in module pyrepo_mcda.distance_metrics), 67
pearson_coeff() (in module pyrepo_mcda.correlations), 59
pyrepo_mcda module, 42
pyrepo_mcda.additions module, 56
pyrepo_mcda.compromise_rankings module, 58
pyrepo_mcda.correlations module, 57
pyrepo_mcda.distance_metrics module, 60
pyrepo_mcda.mcda_methods module, 42
pyrepo_mcda.mcda_methods.codas module, 42
pyrepo_mcda.mcda_methods.edas module, 43
pyrepo_mcda.mcda_methods.mabac module, 44
pyrepo_mcda.mcda_methods.mcda_method module, 45
pyrepo_mcda.mcda_methods.multimoora module, 45
pyrepo_mcda.mcda_methods.spotis module, 48
pyrepo_mcda.mcda_methods.topsis module, 49
pyrepo_mcda.mcda_methods.vikor module, 49
pyrepo_mcda.mcda_methods.waspas module, 50
pyrepo_mcda.normalizations module, 68
pyrepo_mcda.sensitivity_analysis_weights_percentages module, 70
pyrepo_mcda.sensitivity_analysis_weights_values module, 72
pyrepo_mcda.weighting_methods module, 73
R
rank_position_method() (in module pyrepo_mcda.compromise_rankings), 58
rank_preferences() (in module pyrepo_mcda.additions), 56
S
Sensitivity_analysis_weights_percentages (class in pyrepo_mcda.sensitivity_analysis_weights_percentages), 70
Sensitivity_analysis_weights_values (class in pyrepo_mcda.sensitivity_analysis_weights_values), 72
spearman() (in module pyrepo_mcda.correlations), 58
SPOTIS (class in pyrepo_mcda.mcda_methods), 54
SPOTIS (class in pyrepo_mcda.mcda_methods.spotis), 48
squared_chi_square() (in module pyrepo_mcda.distance_metrics), 68
squared_chord() (in module pyrepo_mcda.distance_metrics), 67
squared_euclidean() (in module pyrepo_mcda.distance_metrics), 64
std_euclidean() (in module pyrepo_mcda.distance_metrics), 63
std_weighting() (in module pyrepo_mcda.weighting_methods), 74
sum_normalization() (in module pyrepo_mcda.normalizations), 69
T
TOPSIS (class in pyrepo_mcda.mcda_methods), 54
TOPSIS (class in pyrepo_mcda.mcda_methods.topsis), 49
V
vector_normalization() (in module pyrepo_mcda.normalizations), 70
VIKOR (class in pyrepo_mcda.mcda_methods), 55
VIKOR (class in pyrepo_mcda.mcda_methods.vikor), 49

W

WASPAS (class in pyrepo_mcda.mcda_methods), 55
WASPAS (class in pyrepo_mcda.mcda_methods.waspas), 50
weighted_spearman() (in module pyrepo_mcda.correlations), 59
WS_coeff() (in module pyrepo_mcda.correlations), 59