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
  - AHP
  - ARAS
  - COPRAS
  - CRADIS
  - MARCOS
  - PROMETHEE II
  - PROSA C
  - SAW

- 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)

- Objective weighting methods for determining criteria weights required by Multi-Criteria Decision Analysis (MCDA) methods:
  - equal_weighting (Equal weighting method)
  - entropy_weighting (Entropy weighting method)
  - std_weighting (Standard deviation weighting method)
  - critic_weighting (CRITIC weighting method)
  - gini_weighting (Gini coefficient-based weighting method)
  - merec_weighting (MEREC weighting method)
  - stat_var_weighting (Statistical variance weighting method)
  - cilos_weighting (CILOS weighting method)
  - idocriw_weighting (IDOCRIW weighting method)
  - angle_weighting (Angle weighting method)
  - coeff_var_weighting (Coefficient of variation weighting method)

- Stochastic Multicriteria Acceptability Analysis Method - SMAA combined with VIKOR (VIKOR_SMAA)

- 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
```

(continues on next page)
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.
# 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 v parameter. 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

```python
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

```python
rank = rank_preferences(pref, reverse = False)
```

```python
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])
```

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

# 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:  [0.478  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.
# 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])

# 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.
→ default. 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 -0.2170 -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.

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],
                   [600, 5, 3, 2, 2, 1],
                   [2000, 3, 2, 3, 4, 3],
                   [600, 4, 3, 1, 2, 2],
                   [800, 2, 4, 3, 3, 4]])

(continues on next page)
# 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, 6808.9],
                   [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.
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 0.13 0.4607 0.212 0.9443 0.043]
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],
                   [2.796570, 2.983000, 2.744904, 2.692550, 2.787563, 2.878851],
                   [2.796570, 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, 3.052381, 3.695238]])
```

(continues on next page)
# 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, 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.
# 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, 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.

```python
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, 5, 5, 5, 5, 4]])

weights = np.array([0.215, 0.215, 0.159, 0.133, 0.102, 0.102, 0.073])
types = 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]
The ARAS method

The ARAS method is used to obtain utility function values for alternatives. Then alternatives have to be ranked according to utility function values in descending order. There is a possibility to select the normalization method of the decision matrix during the ARAS method object initialization. The default normalization for ARAS is \( \text{sum}\_\text{normalization} \). If you do not provide a normalization method, it will be set automatically to \( \text{sum}\_\text{normalization} \). The ranking is generated using \text{rank}\_\text{preferences} method from \text{additions} submodule providing utility function values \( \text{pref} \) as argument and setting parameter \text{reverse} as True because we need to sort preferences descendingly.

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

# provide decision matrix in array numpy.darray
matrix = np.array([[80, 16, 2, 5],
                   [110, 32, 2, 9],
                   [130, 64, 4, 9],
                   [185, 64, 4, 1],
                   [135, 64, 3, 4],
                   [140, 32, 3, 5],
                   [185, 64, 6, 7],
                   [110, 16, 3, 3],
                   [120, 16, 4, 3],
                   [340, 128, 6, 5]])

# provide criteria weights in array numpy.darray. All weights must sum to 1.
weights = np.array([0.60338, 0.13639, 0.19567, 0.06456])

# 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 ARAS method object providing the normalization method. In ARAS, it is \`\text{sum}\_\text{normalization}\` by default, so if you do not provide a normalization method, it will be set as \`\text{sum}\_\text{normalization}`.
aras = ARAS(normalization_method=norms.sum_normalization)

# Calculate the ARAS preference values of alternatives
pref = aras(matrix, weights, types)

# Generate ranking of alternatives by sorting alternatives descendingly according to the ARAS 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:  [0.6891 0.5852 0.4667 0.5492 0.498 0.5696 0.5495 0.5451 0.5355]
```

(continues on next page)
The COPRAS method

The COPRAS method is used to obtain utility function values for alternatives. Then alternatives have to be ranked according to utility function values in descending order. There is a possibility to select the normalization method of the decision matrix during the COPRAS method object initialization. In the COPRAS method, each normalization is performed automatically as for profit criteria according to the algorithm of this method. The default normalization for COPRAS is sum_normalization. If you do not provide a normalization method, it will be set automatically to sum_normalization. The ranking is generated using rank_preferences method from additions submodule providing utility function values 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 COPRAS
from pyrepo_mcda import normalizations as norms
from pyrepo_mcda.additions import rank_preferences

# provide decision matrix in array numpy.darray
matrix = np.array([[80, 16, 2, 5],
                   [110, 32, 2, 9],
                   [130, 64, 4, 9],
                   [185, 64, 4, 1],
                   [135, 64, 3, 4],
                   [140, 32, 3, 5],
                   [185, 64, 6, 7],
                   [110, 16, 3, 3],
                   [120, 16, 4, 3],
                   [340, 128, 6, 5]])

# provide criteria weights in array numpy.darray. All weights must sum to 1.
weights = np.array([0.60338, 0.13639, 0.19567, 0.06456])

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

# Create the COPRAS method object providing the normalization method. In COPRAS, it is sum_normalization by default, so if you do not provide a normalization method, it will be set as sum_normalization.
copras = COPRAS(normalization_method=norms.sum_normalization)

# Calculate the COPRAS preference values of alternatives
pref = copras(matrix, weights, types)

# Generate ranking of alternatives by sorting alternatives descendingly according to the COPRAS 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))
```

1.1. Usage
print('Ranking: ', rank)

Output

<table>
<thead>
<tr>
<th>Preference values:</th>
<th>1.0</th>
<th>0.8526</th>
<th>0.9193</th>
<th>0.6852</th>
<th>0.8052</th>
<th>0.7259</th>
<th>0.8344</th>
<th>0.7976</th>
<th>0.791</th>
<th>0.7953</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ranking:</td>
<td>1</td>
<td>3</td>
<td>2</td>
<td>10</td>
<td>5</td>
<td>9</td>
<td>4</td>
<td>6</td>
<td>8</td>
<td>7</td>
</tr>
</tbody>
</table>

The MARCOS method

The MARCOS method is used to obtain utility function values for alternatives. Then alternatives have to be ranked according to utility function values in descending order. MARCOS does not require the decision-maker to select the normalization method because it has its own decision matrix normalization method. The ranking is generated using `rank_preferences` method from `additions` submodule providing utility function values `pref` as argument and setting parameter `reverse` as `True` because we need to sort preferences descendingly.

```python
import numpy as np
from pyrepo_mcda.mcdm_methods import MARCOS
from pyrepo_mcda.normalizations as norms
from pyrepo_mcda.additions import rank_preferences

# provide decision matrix in array numpy.darray

# provide criteria weights in array numpy.darray. All weights must sum to 1.
weights = np.array([0.127, 0.159, 0.060, 0.075, 0.043, 0.051, 0.075, 0.061, 0.053, 0.020, 0.039, 0.022, 0.017, 0.027, 0.022, 0.039, 0.017, 0.035, 0.015, 0.024, 0.016])

# 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, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1])

# Create the MARCOS method object. MARCOS does not require the decision-maker to select the normalization method because it has its own decision matrix normalization method.
marcos = MARCOS()
```
# Calculate the MARCOS preference values of alternatives

```python
pref = marcos(matrix, weights, types)
```

# Generate ranking of alternatives by sorting alternatives descendingly according to the MARCOS algorithm (reverse = True means sorting in descending order) according to preference values

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

Output

```
Preference values:  [0.5244  0.8457  0.7963  0.8432  0.4990  0.7220  0.2906]
Ranking:  [6 1 5 3 2 7 4 8]
```

### The CRADIS method

The CRADIS method is used to obtain utility function values for alternatives. Then alternatives have to be ranked according to utility function values in descending order. There is a possibility to select the normalization method of the decision matrix during the CRADIS method object initialization. The default normalization for CRADIS is `linear_normalization`. If you do not provide a normalization method, it will be set automatically to `linear_normalization`. The ranking is generated using `rank_preferences` method from `additions` submodule providing utility function values `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 CRADIS
from pyrepo_mcda import normalizations as norms
from pyrepo_mcda.additions import rank_preferences

# provide decision matrix in array numpy.darray
matrix = np.array([[1.82, 1.59, 2.62, 2.62, 4.31, 3.3, 2.29, 3.3, 4.31, 5.31, 2.29, 1.26, 0.36, 30, 10, 5.02],
                   [1.82, 1.59, 2.62, 2.62, 3.63, 3.3, 2.29, 3.3, 4.31, 6., 2.29, 1.26, 0.54, 40., 11.5, 6.26],
                   [2.88, 2.62, 3.3, 3., 4.64, 3.91, 2.52, 3.91, 3.3, 6., 3.3, 1.44, 0.75, 50., 12.5, 8.97],
                   [1.82, 1.59, 2.62, 3.17, 3.63, 3.3, 2.29, 3.3, 4.31, 6., 3.3, 2., 0.57, 65., 17.5, 8.79],
                   [3.11, 3., 3.91, 4., 5., 4.58, 3.3, 4., 2.29, 5., 3.91, 2.88, 1.35, 100., 16.5, 11.68],
                   [2.88, 2.29, 3.63, 3.63, 5., 4.31, 3.3, 4.31, 2.88, 6., 4.31, 2.29, 1.2, 100., 15.5, 12.9]])

# provide criteria weights in array numpy.darray. All weights must sum to 1.
weights = np.array([0.07, 0.05, 0.05, 0.06, 0.09, 0.06, 0.06, 0.06, 0.05, 0.07, 0.05, 0.05, 0.09, 0.06, 0.07, 0.06])

# 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., -1., -1., -1., -1., -1., -1., -1., -1., -1., -1.])
```
# Create the CRADIS method object. The default normalization for CRADIS is ``linear_normalization`` but you can select others.

```python
cradis = CRADIS(normalization_method=norms.linear_normalization)
```

# Calculate the CRADIS preference values of alternatives

```python
pref = cradis(matrix, weights, types)
```

# Generate ranking of alternatives by sorting alternatives descendingly according to the CRADIS algorithm (reverse = True means sorting in descending order) according to preference values

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

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

Output

```
Preference values: [0.7943 0.8213 0.6423 0.7375 0.5803 0.6125]
Ranking: [2 1 4 3 6 5]
```

The PROMETHEE II method

The PROMETHEE II method does not require providing normalization method by decision-maker. It requires providing list with preference functions for each criterion selected from six available preference functions: Type 1 _usual_function, Type 2 _ushape_function, Type 3 _vshape_function, Type 4 _level_function, Type 5 _linear_function, Type 6 _gaussian_function.

If the decision-maker does not provide a list with preference functions `preference_functions`, this list is generated automatically, and preference functions in this list are set to default preference function Type 1 _usual_function.

Depending on chosen preference function PROMETHEE II requires providing p, q or both p and q parameters. The Type 1 _usual_function does not require any parameter from p, q. Type 2 _ushape_function requires q parameter. Type 3 _vshape_function requires p parameter. Type 4 _level_function, Type 5 _linear_function and Type 6 _gaussian_function require p and q parameters.

p is a vector with values of the threshold of the absolute preference for each criterion. q is a vector with values of the threshold of indifference for each criterion.

If the decision-maker does not provide p or q parameters, they are set automatically.

The ranking is generated using `rank_preferences` method from `additions` submodule providing utility function values `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 PROMETHEE_II
from pyrepo_mcda import normalizations as norms
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]],
                   dtype=float)
```

(continues on next page)
[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 PROMETHEE II method object. PROMETHEE II does not require normalization method.
promethee_II = PROMETHEE_II()

# provide preference functions selected from six preference functions available for PROMETHEE II for each criterion
preference_functions = [promethee_II._linear_function for pf in range(len(weights))]

# provide p or q or both p and q parameters depending on chosen preference function
p = 2 * np.ones(len(weights))
q = 1 * np.ones(len(weights))

# Calculate the PROMETHEE II preference values of alternatives
pref = promethee_II(matrix, weights, types, preference_functions = preference_functions, p=p, q=q)

# Generate ranking of alternatives by sorting alternatives descendingly according to the PROMETHEE II 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: [-0.26 -0.52 -0.22 0.36 0.7 -0.06]
Ranking: [5 6 4 2 1 3]

Usage examples for other preference functions with matrix, weights, types, p, and q provided above for PROMETHEE II

Usual

promethee_II = PROMETHEE_II()
preference_functions = [promethee_II._usual_function for pf in range(len(weights))]
pref = promethee_II(matrix, weights, types, preference_functions = preference_functions)
rank = rank_preferences(pref, reverse=True)

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

Output
Preference values: [-0.28, -0.5, -0.24, 0.32, 0.84, -0.14]  
Ranking: [5, 6, 4, 2, 1, 3]

**U-shape**

```python
promethee_II = PROMETHEE_II()
pref = promethee_II(matrix, weights, types, preference_functions = preference_functions, q = q)  
rank = rank_preferences(pref, reverse=True)
print('Preference values: ', np.round(pref, 4))
print('Ranking: ', rank)
```

Output

Preference values: [-0.26, -0.52, -0.22, 0.36, 0.7, -0.06]  
Ranking: [5, 6, 4, 2, 1, 3]

**V-shape**

```python
promethee_II = PROMETHEE_II()
pref = promethee_II(matrix, weights, types, preference_functions = preference_functions, p = p)
rank = rank_preferences(pref, reverse=True)
print('Preference values: ', np.round(pref, 4))
print('Ranking: ', rank)
```

Output

Preference values: [-0.27, -0.51, -0.23, 0.34, 0.77, -0.1]  
Ranking: [5, 6, 4, 2, 1, 3]

**Level**

```python
promethee_II = PROMETHEE_II()
pref = promethee_II(matrix, weights, types, preference_functions = preference_functions, p = p, q = q)
rank = rank_preferences(pref, reverse=True)
print('Preference values: ', np.round(pref, 4))
print('Ranking: ', rank)
```

Output

Preference values: [-0.25, -0.46, -0.22, 0.32, 0.65, -0.04]  
Ranking: [5, 6, 4, 2, 1, 3]

**Linear**
```python
promethee_II = PROMETHEE_II()
preference_functions = [promethee_II._linear_function for pf in range(len(weights))]
pref = promethee_II(matrix, weights, types, preference_functions = preference_functions,␣
       p = p, q = q)
rank = rank_preferences(pref, reverse=True)
print('Preference values: ', np.round(pref, 4))
print('Ranking: ', rank)
```

Output

```
Preference values: [-0.26 -0.52 -0.22  0.36  0.7  -0.06]
Ranking:  [5 6 4 2 1 3]
```

**Gaussian**

```python
promethee_II = PROMETHEE_II()
preference_functions = [promethee_II._gaussian_function for pf in range(len(weights))]
pref = promethee_II(matrix, weights, types, preference_functions = preference_functions,␣
       p = p, q = q)
rank = rank_preferences(pref, reverse=True)
print('Preference values: ', np.round(pref, 4))
print('Ranking: ', rank)
```

Output

```
Preference values: [-0.2339 -0.4536 -0.2213  0.3048  0.6569 -0.0528]
Ranking:  [5 6 4 2 1 3]
```

### The PROSA-C method

The PROSA-C method is based on the PROMETHEE II method. Thus it is similar. It does not require providing normalization method by decision-maker. It requires providing list with preference functions for each criterion selected from six available preference functions: Type 1 _usual_function, Type 2 _ushape_function, Type 3 _vshape_function, Type 4 _level_function, Type 5 _linear_function, Type 6 _gaussian_function.

If the decision-maker does not provide a list with preference functions preference_functions, this list is generated automatically, and preference functions in this list are set to default preference function Type 1 _usual_function.

Depending on chosen preference function PROSA-C requires providing p, q or both p and q parameters. The Type 1 _usual_function does not require any parameter from p, q. Type 2 _ushape_function requires q parameter. Type 3 _vshape_function requires p parameter. Type 4 _level_function, Type 5 _linear_function and Type 6 _gaussian_function require p and q parameters.

p is a vector with values of the threshold of the absolute preference for each criterion. q is a vector with values of the threshold of indifference for each criterion.

If the decision-maker does not provide p or q parameters, they are set automatically.

PROSA-C enables to reduce criteria compensation using sustainability coefficient s, which is an additional parameter of this method compared to PROMETHEE II.

The s parameter is a vector with sustainability coefficient values for each criterion. It is recommended to set the value of s in the range from 0 to 0.5. If decision-maker does not provide s parameter, it is set automatically to a default value equal to 0.3 for each criterion.

1.1. Usage
The ranking is generated using `rank_preferences` method from `additions` submodule providing utility function values `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 PROSA_C
from pyrepo_mcda import normalizations as norms
from pyrepo_mcda.additions import rank_preferences

# provide decision matrix in array numpy.darray
matrix = np.array([[38723, 34913, 25596, 34842, 22570, 39773, 19500, 34525, 16486],
[33207, 32085, 2123, 32095, 1445, 17485, 868, 16958, 958],
[0, 0.2, 0.2, 0.2, 99, 99, 99, 99, 99],
[3375, 3127, 3547, 3115, 4135, 3160, 4295, 3653],
[11.36, 12.78, 12.78, 12.86, 17, 12.86, 17, 12.86],
[-320.9, -148.4, -148.4, -9.9, -9.9, 0, -9.9, 0, -9.9],
[203.7, 463, 356.2, 552.5, 295, 383, 264, 352, 264],
[0, 11.7, 44.8, 11.7, 95.9, 95.9, 116.8, 116.8, 164.9],
[0, 4.9, 10.7, 5.4, 11.2, 11.2, 11.2, 11.2, 11.2],
[1, 1, 1, 3.5, 4, 4, 4, 4, 4],
[21.5, 47.9, 27.7, 39.7, 1.5, 22.7, 2.7, 23.9, 1],
[0, 3.7, 4.5, 10.3, 95.9, 95.9, 116.8, 116.8, 164.9]])

matrix = matrix.T

# provide criteria weights in array numpy.darray. All weights must sum to 1.
weights = np.array([0.3333, 0.1667, 0.1667, 0.3333, 0.25, 0.75, 1, 1, 0.4, 0.20, 0.40, 1])
weights = weights / np.sum(weights)

# 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, 1, -1, 1, 1, -1, -1])

# Create the PROSA-C method object. PROSA-C does not require normalization method.
prosa_c = PROSA_C()

# provide preference functions selected from six preference functions available for
# PROSA-C for each criterion
preference_functions = [prosa_c._linear_function for pf in range(len(weights))]

# provide p or q or both p and q parameters depending on chosen preference function and
# s parameter
p = np.array([2100, 5000, 50, 200, 5, 20, 100, 80, 4, 23, 3])
q = np.array([420, 1000, 10, 40, 1, 7, 50, 20, 1, 1, 4.6, 1])
s = np.array([0.4, 0.5, 0.3, 0.4, 0.3, 0.4, 0.3, 0.3, 0.2, 0.4, 0.4, 0.2])

# Calculate the PROSA-C preference values of alternatives
pref = prosa_c(matrix, weights, types, preference_functions = preference_functions, p = p, q = q, s = s)

# Generate ranking of alternatives by sorting alternatives descendingly according to the
# PROSA-C 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: [-0.5921 -0.6014 -0.324 -0.4381 0.2791 -0.0703 0.3739 0.0451 0.3592]
Ranking: [8 9 6 7 3 5 1 4 2]

The SAW method

The SAW method is used to obtain utility function values for alternatives. Then alternatives have to be ranked according to utility function values in descending order. There is a possibility to select the normalization method of the decision matrix during the SAW method object initialization. The default normalization for SAW is linear_normalization. If you do not provide a normalization method, it will be set automatically to linear_normalization. The ranking is generated using rank_preferences method from additions submodule providing utility function values 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 SAW
from pyrepo_mcda import normalizations as norms
from pyrepo_mcda.additions import rank_preferences

# provide decision matrix in array numpy.darray
matrix = np.array([[0.75, 0.50, 0.75, 0, 0, 0, 1],
                   [0.75, 1, 0.75, 0, 0, 0, 0.75],
                   [0.75, 0.75, 0.75, 0, 0.50, 0.25, 1],
                   [0.50, 0.50, 0.75, 1, 0.50, 0, 0.75]])

# provide criteria weights in array numpy.darray. All weights must sum to 1.
weights = np.array([0.1, 0.1, 0.1, 0.15, 0.2, 0.25, 0.1])

# 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, 1])

# Create the SAW method object. The default normalization for SAW is `linear_normalization` but you can select others.
saw = SAW(normalization_method=norms.linear_normalization)

# Calculate the SAW preference values of alternatives
pref = saw(matrix, weights, types)

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

(continues on next page)
print('Preference values: ', np.round(pref, 4))
print('Ranking: ', rank)

Output

Preference values: [0.35  0.375  0.825  0.6417]
Ranking:  [4 3 1 2]

The AHP method

The first step of the classical AHP method application requires a matrix of significance comparisons of criteria. Next, check the consistency of the matrix with criteria comparisons. Criteria weights are calculated based on the criteria comparison matrix using one of three methods: _calculate_eigenvector, _normalized_column_sum or _geometric_mean. Then provide a comparison matrix of alternatives for each criterion. Utility function values of AHP are calculated by _classic_ahp function. The ranking is generated using the rank_preferences method from the additions submodule, providing utility function values pref as argument and setting parameter reverse as True because we need to sort preferences descendingly.

Classical AHP

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

# Step 1 - provide matrix for criteria comparisons
PCcriteria = np.array([[1, 1, 5, 3],
                       [1, 1, 5, 3],
                       [1/5, 1/5, 1, 1/3],
                       [1/3, 1/3, 3, 1]])

# Create the object of the AHP method
ahp = AHP()

# Step 2 - check consistency of matrix with criteria comparison
ahp._check_consistency(PCcriteria)

# Step 3 - compute priority vector of criteria (weights)
weights = ahp._calculate_eigenvector(PCcriteria)

# Step 4 - provide pairwise comparison matrices of the alternatives for each criterion
PCM1 = np.array([[1, 5, 1, 1, 1/3, 3],
                 [1/5, 1, 1/3, 1/5, 1/7, 1],
                 [1, 3, 1, 1/3, 1/5, 1],
                 [1, 5, 3, 1, 1/3, 3],
                 [3, 7, 5, 3, 1, 7],
                 [1/3, 1, 1, 1/3, 1/7, 1]])

PCM2 = np.array([[1, 7, 3, 1/3, 1/3, 1/3],
                 [1/7, 1, 1/3, 1/7, 1/9, 1/7],
                 [1/3, 3, 1, 1/5, 1/5, 1/5],
                 [3, 7, 5, 1, 1, 1],
                 [3, 9, 5, 1, 1, 1],
                 [3, 7, 5, 1, 1, 1]])

PCM3 = np.array([[1, 1/9, 1/7, 1/9, 1, 1/5],
                 [9, 1, 1, 1, 5, 3],
                 [1/9, 1/7, 1/9, 1, 1/5, 3],
                 [1, 9, 1, 1, 5, 3],
                 [1/9, 1/7, 1/9, 1, 1/5, 3],
                 [1, 9, 1, 1, 5, 3]])
```

(continues on next page)
Another usage of AHP - for numerical values of performances and weights

If you have a decision matrix with numerical performance values, a vector with numerical criteria weights, and determined criteria types (profit or cost), you can use the AHP method like other MCDA methods (for example, SAW):

```python
# provide decision matrix in array numpy.darray
matrix = np.array([[0.75, 0.50, 0.75, 0, 0, 0, 1],
                   [0.75, 1, 0.75, 0, 0, 0, 0.75],
                   [0.75, 0.75, 0.75, 0, 0.50, 0.25, 1],
                   [0.50, 0.50, 0.75, 1, 0.50, 0, 0.75]])

# provide criteria weights in array numpy.darray. All weights must sum to 1.
weights = np.array([0.1, 0.1, 0.1, 0.15, 0.2, 0.25, 0.1])
```

1.1. Usage
# 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, 1])

# Create the AHP method object. The default normalization for SAW is \texttt{sum_normalization} but you can select others.
ahp = AHP(normalization_method=norms.sum_normalization)

# Calculate the AHP preference values of alternatives
pref = ahp(matrix, weights, types)

# Generate ranking of alternatives by sorting alternatives descendingly according to the AHP 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:  [0.099 0.1101 0.4581 0.3328]
Ranking:  [4 3 1 2]

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],
                    [8, 9, 8, 9, 8, 8],
                    [1, 4, 1, 1, 1, 1],
                    [4, 1, 4, 3, 2],
                    [3, 1, 4, 3, 2],
                    [10, 5, 10, 9, 10],
                    [6, 3, 6, 5, 4, 6],
                    [9, 10, 10, 10, 9],
                    [5, 6, 3, 2, 6, 5]]])

# Calculate the compromise ranking using \texttt{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.

```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 `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.

1.1. Usage
```python
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
```

(continues on next page)
# 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 matrix with performance values of alternatives in rows considering criteria in columns. It returns vector with criteria weights. All values in vector 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

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 matrix with performance values of alternatives in rows considering criteria in columns. It returns vector with criteria weights. All values in vector weights must sum to 1.

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

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]])

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 matrix with performance values of alternatives in rows considering criteria in columns. It returns vector with criteria weights. All values in vector 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]

Equal weighting method

```}

```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.equal_weighting(matrix)
print('Equal weights: ', np.round(weights, 3))

Output
Equal weights: [0.333 0.333 0.333]

Gini coefficient-based weighting method

```}

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

matrix = np.array([[29.4, 83, 47, 114, 12, 30, 120, 240, 170, 90, 1717.75],
                   [30, 38.1, 124.7, 117, 16, 60, 60, 60, 93, 70, 2389],
                   [29.28, 59.27, 41.13, 58, 16, 30, 120, 170, 78, 239.99],
                   [33.6, 71, 55, 159, 23.6, 60, 240, 240, 132, 140, 2099],
                   [21, 59, 41, 66, 16, 24, 60, 120, 170, 70, 439],
                   [35, 65, 42, 134, 12, 60, 240, 240, 145, 60, 1087],
                   [47, 79, 54, 158, 19, 60, 120, 120, 360, 72, 2499],
                   [28.3, 62.3, 44.9, 116, 12, 30, 60, 60, 130, 90, 999.99],
                   [36.9, 28.6, 121.6, 130, 12, 60, 120, 120, 80, 80, 1099],
                   [32, 59, 41, 60, 16, 30, 120, 120, 170, 60, 302.96],
                   [28.4, 66.3, 48.6, 126, 12, 60, 240, 240, 132, 135, 1629],
                   [29.8, 46, 113, 47, 18, 50, 50, 50, 360, 72, 2099],
                   [20.2, 64, 80, 70, 8, 24, 60, 120, 166, 480, 699.99],
                   [20.2, 64, 80, 70, 8, 24, 60, 120, 166, 480, 699.99],
                   [20.2, 64, 80, 70, 8, 24, 60, 120, 166, 480, 699.99],
                   [20.2, 64, 80, 70, 8, 24, 60, 120, 166, 480, 699.99],
                   [20.2, 64, 80, 70, 8, 24, 60, 120, 166, 480, 699.99],
                   [20.2, 64, 80, 70, 8, 24, 60, 120, 166, 480, 699.99],
                   [20.2, 64, 80, 70, 8, 24, 60, 120, 166, 480, 699.99],
                   [20.2, 64, 80, 70, 8, 24, 60, 120, 166, 480, 699.99]])

(continues on next page)
weights = mcda_weights.gini_weighting(matrix)
print('Gini coefficient-based weights: ', np.round(weights, 4))

Output
Gini coefficient-based weights: [0.0362 0.0437 0.0848 0.0984 0.048 0.0842 0.1379 0.→1125 0.0745 0.1107 0.169 ]

MEREC weighting method

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

matrix = np.array([[450, 8000, 54, 145],
[10, 9100, 2, 160],
[100, 8200, 31, 153],
[220, 9300, 1, 162],
[5, 8400, 23, 158]])

types = np.array([1, 1, -1, -1])

weights = mcda_weights.merec_weighting(matrix, types)
print('MEREC weights: ', np.round(weights, 4))

Output
MEREC weights: [0.5752 0.0141 0.4016 0.0091]
```

Statistical variance weighting method

```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.stat_var_weighting(matrix)
print('Statistical variance weights: ', np.round(weights, 4))

Output
Statistical variance weights: [0.3441 0.3497 0.3062]
```
CILOS weighting method

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

matrix = np.array([[3, 100, 10, 7],
                   [2.5, 80, 8, 5],
                   [1.8, 50, 20, 11],
                   [2.2, 70, 12, 9]])

types = np.array([-1, 1, -1, 1])

weights = mcda_weights.cilos_weighting(matrix, types)
print('CILOS weights:', np.round(weights, 3))
```

Output

```
CILOS weights: [0.334 0.220 0.196 0.250]
```

IDOCRIW weighting method

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

matrix = np.array([[3.0, 100, 10, 7],
                   [2.5, 80, 8, 5],
                   [1.8, 50, 20, 11],
                   [2.2, 70, 12, 9]])

types = np.array([-1, 1, -1, 1])

weights = mcda_weights.idocriw_weighting(matrix, types)
print('IDOCRIW weights:', np.round(weights, 3))
```

Output

```
IDOCRIW weights: [0.166 0.189 0.355 0.291]
```

Angle weighting method

```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]])

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

weights = mcda_weights.angle_weighting(matrix, types)
print('Angle weights:', np.round(weights, 4))
```

Output

```
Angle weights: [0.187 0.187 0.355 0.291]
```
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.coeff_var_weighting(matrix)
print('Coefficient of variation weights: ', np.round(weights, 4))

Output
Coefficient of variation weights:  [0.4258 0.361  0.2121 0.0011]

Stochastic Multicriteria Acceptability Analysis Method - SMAA (VIKOR_SMAA)

from pyrepo_mcda.mcda_methods import VIKOR_SMAA

# Criteria number
n = matrix.shape[1]
# Number of weight vectors to generate for SMAA
iterations = 10000

# Create the object of the `VIKOR_SMAA` method
vikor_smaa = VIKOR_SMAA()
# Generate weight vectors for SMAA. Number of weight vectors is equal to `iterations` = number. Vectors include `n` values.
weight_vectors = vikor_smaa._generate_weights(n, iterations)

# Calculate Rank acceptability index, Central weight vector and final ranking based on SMAA method combined with VIKOR
rank_acceptability_index, central_weight_vector, rank_scores = vikor_smaa(matrix, weight_vectors, types)

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 of A and B must be the same. This method returns value of Euclidean distance between vectors A and B.

import numpy as np
from pyrepo_mcda import distance_metrics as dists

(continues on next page)
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 of A and B must be the same. This method returns value of Manhattan distance between vectors A and B.

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.

import numpy as np
from pyrepo_mcda import normalizations as norms

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])

(continues on next page)
```python
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.

```python
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])

# 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, and index of column in decision matrix for chosen criterion j

import numpy as np
from pyrepo_mcda.sensitivity_analysis_weights_values import Sensitivity_analysis_weights_values

import numpy as np
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])

# 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
(continues on next page)
\[ \text{analysis, for example } j = 1 \text{ for criterion in the second column} \]

\[ j = 1 \]

# 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 of MCDA methods: TOPSIS, CODAS, VIKOR, SPOTIS, EDAS, MABAC, MULTIMOORA, WASPAS, Compromise rankings, Sensitivity analysis

1.2.1 Import necessary packages

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

[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_
sensitivity, plot_barplot_sensitivity, plot_radar, 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>
<tr>
<td>A4</td>
<td>14</td>
<td>3400</td>
<td>50</td>
<td>0.7</td>
</tr>
<tr>
<td>A5</td>
<td>15</td>
<td>3300</td>
<td>40</td>
<td>0.8</td>
</tr>
<tr>
<td>A6</td>
<td>28</td>
<td>3000</td>
<td>30</td>
<td>0.6</td>
</tr>
</tbody>
</table>
```python
print(weights)
[0.2857 0.3036 0.2321 0.1786]

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:

```python
matrix = df_data.to_numpy()
```

Create an index list `list_alt_names` and data frame for collecting results `rank_results` indexed by ‘Ai’:

```python
list_alt_names = ['A_{'+ str(i) + '}' for i in range(1, df_data.shape[0] + 1)]
list_crit_names = ['C_{'+ str(i) + '}' for i in range(1, df_data.shape[1] + 1)]
```

```python
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.

```python
# 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.

```python
# CODAS
# CODAS preference values (preferences) must be sorted descending order
codas = CODAS(normalization_method = norms.linear_normalization, distance_metric = dists.euclidean, tau = 0.02)
```

(continues on next page)
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.

```python
# 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.

```python
# 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)
rank = rank_preferences(pref, reverse = False)
rank_results['SPOTIS'] = rank
```
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.

```python
[16]: # 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.

```python
[17]: # 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.

```python
[18]: # 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
[19]: # 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
[20]: 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'))
```

<table>
<thead>
<tr>
<th>Ai</th>
<th>TOPSIS</th>
<th>CODAS</th>
<th>VIKOR</th>
<th>SPOTIS</th>
<th>EDAS</th>
<th>MABAC</th>
<th>MMOORA</th>
<th>WASPAS</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>␣</td>
</tr>
<tr>
<td>$A_{2}$</td>
<td>4</td>
<td>2</td>
<td>5</td>
<td>3</td>
<td>2</td>
<td>3</td>
<td>2</td>
<td>␣</td>
</tr>
<tr>
<td>$A_{3}$</td>
<td>2</td>
<td>3</td>
<td>2</td>
<td>2</td>
<td>3</td>
<td>2</td>
<td>3</td>
<td>␣</td>
</tr>
<tr>
<td>$A_{4}$</td>
<td>6</td>
<td>5</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>5</td>
<td>␣</td>
</tr>
<tr>
<td>$A_{5}$</td>
<td>5</td>
<td>6</td>
<td>4</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>4</td>
<td>␣</td>
</tr>
<tr>
<td>$A_{6}$</td>
<td>3</td>
<td>4</td>
<td>3</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>3</td>
<td>␣</td>
</tr>
</tbody>
</table>

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
[21]: 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'))
```

<table>
<thead>
<tr>
<th>Ai</th>
<th>TOPSIS</th>
<th>CODAS</th>
<th>VIKOR</th>
<th>SPOTIS</th>
<th>EDAS</th>
<th>MABAC</th>
<th>MMOORA</th>
<th>WASPAS</th>
<th>Compromise</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>
</tr>
<tr>
<td>$A_{2}$</td>
<td>4</td>
<td>2</td>
<td>5</td>
<td>3</td>
<td>2</td>
<td>3</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>$A_{3}$</td>
<td>2</td>
<td>3</td>
<td>2</td>
<td>2</td>
<td>3</td>
<td>2</td>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>$A_{4}$</td>
<td>6</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>$A_{5}$</td>
<td>5</td>
<td>6</td>
<td>4</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>$A_{6}$</td>
<td>3</td>
<td>4</td>
<td>3</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
</tbody>
</table>
Display rankings provided by different MCDA methods using an exemplary method named `plot_barplot` for visualization.

```
[22]: df_plot = copy.deepcopy(rank_results)
plot_barplot(df_plot, 'MCDA methods')
```

The PostScript backend does not support transparency; partially transparent artists will be rendered opaque.

### 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.

```python
percentages = np.arange(0.25, 0.55, 0.1)
```

Create the chosen MCDA object.

```python
method = TOPSIS(normalization_method=norms.minmax_normalization, distance_metric=dists.euclidean)
```

Create the sensitivity analysis method object.

```python
sensitivity_analysis = Sensitivity_analysis_weights_percentages()
```

Perform sensitivity analysis with weights modification for chosen criteria.

```python
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('Sensitivity analysis for C' + str(j + 1))
    print(tabulate(data_sens, headers = header, tablefmt='github'))
```

**Sensitivity analysis for C1**

<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>0</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>2</td>
<td>0</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>3</td>
<td>2</td>
</tr>
</tbody>
</table>
```

### 1.2. Illustrative examples of MCDA methods: TOPSIS, CODAS, VIKOR, SPOTIS, EDAS, MABAC, MULTIMOORA, WASPAS, Compromise rankings, Sensitivity analysis
### 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></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>
<td>3</td>
</tr>
<tr>
<td>$A_1$</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>
<td>3</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>$A_3$</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>$A_4$</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>$A_5$</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>$A_6$</td>
<td>6</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>3</td>
<td>2</td>
</tr>
</tbody>
</table>

### Sensitivity analysis for C3

<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></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>$A_1$</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>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>3</td>
<td>2</td>
</tr>
</tbody>
</table>

(continues on next page)
1.2. Illustrative examples of MCDA methods: TOPSIS, CODAS, VIKOR, SPOTIS, EDAS, MABAC, MULTIMOORA, WASPAS, Compromise rankings, Sensitivity analysis

### Sensitivity analysis for C4

**Ai** | -45% | -35% | -25% | -15% | -5% | 5% | 15% | 25% | 35% | 45%
--- | --- | --- | --- | --- | --- | --- | --- | --- | --- | ---
| $A_{1}$ | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| $A_{2}$ | 4 | 4 | 4 | 4 | 4 | 4 | 4 | 4 | 4 | 4 |
| $A_{3}$ | 3 | 3 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 |
| $A_{4}$ | 5 | 5 | 6 | 6 | 6 | 6 | 6 | 6 | 6 | 6 |
| $A_{5}$ | 6 | 6 | 5 | 5 | 5 | 5 | 5 | 5 | 5 | 5 |
| $A_{6}$ | 2 | 2 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 |

### TOPSIS, modification of C3 weight

![TOPSIS, modification of C3 weight](chart3.png)

### TOPSIS, modification of C4 weight

![TOPSIS, modification of C4 weight](chart4.png)
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>1</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>3</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>4</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>
</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>
</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>2</td>
</tr>
</tbody>
</table>

### Sensitivity analysis for C2

(continues on next page)
### 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>3</td>
<td>5</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>$A_3$</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>$A_4$</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>4</td>
<td>3</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>$A_5$</td>
<td>4</td>
<td>4</td>
<td>4</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>3</td>
<td>3</td>
<td>5</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>6</td>
</tr>
</tbody>
</table>

#### Illustrative examples of MCDA methods

- TOPSIS
- CODAS
- VIKOR
- SPOTIS
- EDAS
- MABAC
- MULTIMOORA
- WASPAS
- Compromise rankings
- Sensitivity analysis

---

**TOPSIS, modification of $C_2$ weight**

*Graph showing rank changes across different weight values.*

---

1.2. Illustrative examples of MCDA methods: TOPSIS, CODAS, VIKOR, SPOTIS, EDAS, MABAC, MULTIMOORA, WASPAS, Compromise rankings, Sensitivity analysis
Sensitivity analysis for $C_4$

<table>
<thead>
<tr>
<th>$A_i$</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>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>$A_{2}$</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>$A_{3}$</td>
<td>4</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>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>$A_{5}$</td>
<td>6</td>
<td>6</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>$A_{6}$</td>
<td>2</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>6</td>
</tr>
</tbody>
</table>

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")

The PostScript backend does not support transparency; partially transparent artists will be rendered opaque.
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] + ' weight modification', j)

The PostScript backend does not support transparency; partially transparent artists will be rendered opaque.

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,
] (continues on next page)
# 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

[27]: Euclidean  Manhattan  Hausdorff  Chebyshev  Bray curtis  Canberra  
    $A_1$  1  1  2  1  1  1
    $A_2$  4  3  4  4  3  4
    $A_3$  2  2  1  2  2  2
    $A_4$  6  6  5  6  6  5
    $A_5$  5  5  3  5  5  3
    $A_6$  3  4  4  3  4  6

Lorentzian  Jaccard
    $A_1$  1  1
    $A_2$  3  5
    $A_3$  2  2
    $A_4$  6  6
    $A_5$  5  4
    $A_6$  4  3

[28]: # plot box chart of alternatives preference values
    plot_barplot(df_rankings, 'Distance metrics')

    The PostScript backend does not support transparency; partially transparent artists will be rendered opaque.
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.

```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,
}
```

(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))

The PostScript backend does not support transparency; partially transparent artists will...

be rendered opaque.

![Boxplot Simulation](image.png)
The PostScript backend does not support transparency; partially transparent artists will be rendered opaque.

1.2. Illustrative examples of MCDA methods: TOPSIS, CODAS, VIKOR, SPOTIS, EDAS, MABAC, MULTIMOORA, WASPAS, Compromise rankings, Sensitivity analysis
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 `spearman` - $r_w$ Weighted Spearman correlation coefficient `weighted_spearman` - $W_S$ Similarity rank correlation coefficient `WS_coeff` - Pearson correlation coefficient `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(corr.weighted_spearman(data[i], data[j]))
    dict_new_heatmap_ws[j].append(corr.WS_coeff(data[i], data[j]))
    dict_new_heatmap_pearson[j].append(corr.pearson_coeff(data[i], data[j]))
    dict_new_heatmap_spearman[j].append(corr.spearman(data[i], data[j]))

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

(continues on next page)
```python
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
```

[32]: # correlation matrix with rw coefficient
draw_heatmap(df_new_heatmap_rw, r'$r_w$')

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

1.2. Illustrative examples of MCDA methods: TOPSIS, CODAS, VIKOR, SPOTIS, EDAS, MABAC, MULTIMOORA, WASPAS, Compromise rankings, Sensitivity analysis
### Correlation: \( r_s \)

<table>
<thead>
<tr>
<th></th>
<th>TOPSIS</th>
<th>CODAS</th>
<th>VIKOR</th>
<th>SPOTIS</th>
<th>EDAS</th>
<th>MABAC</th>
<th>MMOORA</th>
<th>WASPAS</th>
<th>Compromise</th>
</tr>
</thead>
<tbody>
<tr>
<td>TOPSIS</td>
<td>1.00</td>
<td>0.84</td>
<td>0.97</td>
<td>0.94</td>
<td>0.85</td>
<td>0.94</td>
<td>0.88</td>
<td>0.84</td>
<td>0.94</td>
</tr>
<tr>
<td>CODAS</td>
<td>1.00</td>
<td>0.74</td>
<td>0.90</td>
<td>0.90</td>
<td>0.88</td>
<td>0.97</td>
<td>1.00</td>
<td>0.88</td>
<td>1.00</td>
</tr>
<tr>
<td>VIKOR</td>
<td>0.83</td>
<td>0.66</td>
<td>0.94</td>
<td>0.94</td>
<td>0.91</td>
<td>0.91</td>
<td>1.00</td>
<td>0.91</td>
<td>1.00</td>
</tr>
<tr>
<td>SPOTIS</td>
<td>0.88</td>
<td>0.94</td>
<td>0.94</td>
<td>0.94</td>
<td>0.91</td>
<td>0.91</td>
<td>1.00</td>
<td>0.91</td>
<td>1.00</td>
</tr>
<tr>
<td>EDAS</td>
<td>0.88</td>
<td>0.94</td>
<td>0.94</td>
<td>0.94</td>
<td>0.91</td>
<td>0.91</td>
<td>1.00</td>
<td>0.91</td>
<td>1.00</td>
</tr>
<tr>
<td>MABAC</td>
<td>0.89</td>
<td>0.94</td>
<td>0.94</td>
<td>0.94</td>
<td>0.91</td>
<td>0.91</td>
<td>1.00</td>
<td>0.91</td>
<td>1.00</td>
</tr>
<tr>
<td>MMOORA</td>
<td>0.89</td>
<td>0.94</td>
<td>0.94</td>
<td>0.94</td>
<td>0.91</td>
<td>0.91</td>
<td>1.00</td>
<td>0.91</td>
<td>1.00</td>
</tr>
<tr>
<td>WASPAS</td>
<td>0.83</td>
<td>0.66</td>
<td>0.94</td>
<td>0.94</td>
<td>0.91</td>
<td>0.91</td>
<td>1.00</td>
<td>0.91</td>
<td>1.00</td>
</tr>
<tr>
<td>Compromise</td>
<td>1.00</td>
<td>0.84</td>
<td>0.97</td>
<td>0.94</td>
<td>0.85</td>
<td>0.94</td>
<td>0.88</td>
<td>0.84</td>
<td>0.94</td>
</tr>
</tbody>
</table>

### Correlation: Pearson

<table>
<thead>
<tr>
<th></th>
<th>TOPSIS</th>
<th>CODAS</th>
<th>VIKOR</th>
<th>SPOTIS</th>
<th>EDAS</th>
<th>MABAC</th>
<th>MMOORA</th>
<th>WASPAS</th>
<th>Compromise</th>
</tr>
</thead>
<tbody>
<tr>
<td>TOPSIS</td>
<td>1.00</td>
<td>0.77</td>
<td>0.83</td>
<td>1.00</td>
<td>0.94</td>
<td>1.00</td>
<td>0.91</td>
<td>0.89</td>
<td>1.00</td>
</tr>
<tr>
<td>CODAS</td>
<td>1.00</td>
<td>0.54</td>
<td>0.66</td>
<td>0.94</td>
<td>1.00</td>
<td>0.94</td>
<td>0.91</td>
<td>0.91</td>
<td>1.00</td>
</tr>
<tr>
<td>VIKOR</td>
<td>0.83</td>
<td>0.54</td>
<td>1.00</td>
<td>0.94</td>
<td>0.91</td>
<td>0.91</td>
<td>1.00</td>
<td>0.91</td>
<td>1.00</td>
</tr>
<tr>
<td>SPOTIS</td>
<td>0.88</td>
<td>0.77</td>
<td>0.66</td>
<td>0.94</td>
<td>0.94</td>
<td>0.94</td>
<td>0.94</td>
<td>0.94</td>
<td>0.94</td>
</tr>
<tr>
<td>EDAS</td>
<td>0.88</td>
<td>0.77</td>
<td>0.66</td>
<td>0.94</td>
<td>0.94</td>
<td>0.94</td>
<td>0.94</td>
<td>0.94</td>
<td>0.94</td>
</tr>
<tr>
<td>MABAC</td>
<td>0.88</td>
<td>0.77</td>
<td>0.66</td>
<td>0.94</td>
<td>0.94</td>
<td>0.94</td>
<td>0.94</td>
<td>0.94</td>
<td>0.94</td>
</tr>
<tr>
<td>MMOORA</td>
<td>0.88</td>
<td>0.77</td>
<td>0.66</td>
<td>0.94</td>
<td>0.94</td>
<td>0.94</td>
<td>0.94</td>
<td>0.94</td>
<td>0.94</td>
</tr>
<tr>
<td>WASPAS</td>
<td>0.88</td>
<td>0.77</td>
<td>0.66</td>
<td>0.94</td>
<td>0.94</td>
<td>0.94</td>
<td>0.94</td>
<td>0.94</td>
<td>0.94</td>
</tr>
<tr>
<td>Compromise</td>
<td>1.00</td>
<td>0.77</td>
<td>0.94</td>
<td>0.94</td>
<td>0.83</td>
<td>0.94</td>
<td>0.83</td>
<td>0.77</td>
<td>0.94</td>
</tr>
</tbody>
</table>

**Notes:**

[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'(s)')
1.3 Illustrative examples of MCDA methods: CRADIS, COPRAS, ARAS, MARCOS, PROMETHEE II, PROSA-C, SAW, AHP

1.3.1 Import necessary packages

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

[1]:
```python
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
import pandas as pd
import matplotlib
```

Import the necessary Python modules from pyrepo-mcda package.

[2]:
```python
from pyrepo_mcda.mcda_methods import CRADIS, AHP, MARCOS, SAW, ARAS, COPRAS, PROMETHEE_II, PROSA_C
from pyrepo_mcda.mcda_methods import TOPSIS, VIKOR, MABAC, EDAS, SPOTIS, WASPAS
from pyrepo_mcda import normalizations as norms
from pyrepo_mcda.additions import rank_preferences
from pyrepo_mcda import correlations as corrs
from pyrepo_mcda import weighting_methods as mcda_weights
```

Supporting function for running provided examples including visualization. Here are functions called plot_barplot for displaying bar chart showing rankings and draw_heatmap for displaying heat map with rankings correlations. You can copy and customize their codes to your case. Class Create_dictionary helps create a correlation matrix to show rankings correlations.
```python
# bar (column) chart
def plot_barplot(df_plot, legend_title, num):
    ""
    Visualization method to display column chart of alternatives rankings obtained with
different methods.

    Parameters
    ----------
    df_plot : DataFrame
        DataFrame containing rankings of alternatives obtained with different methods.
    title : str
        Title of the legend (Name of group of explored methods, for example MCDA methods).

    Examples
    --------
    >>> plot_barplot(df_plot, legend_title='MCDA methods')
    ""
    step = 1
    list_rank = np.arange(1, len(df_plot) + 1, step)
    ax = df_plot.plot(kind='bar', width = 0.8, stacked=False, edgecolor = 'black',
    figsize = (9,4))
    ax.set_xlabel('Alternatives', fontsize = 12)
    ax.set_ylabel('Rank', fontsize = 12)
    ax.set_yticks(list_rank)
    ax.set_xticklabels(df_plot.index, rotation = 'horizontal')
    ax.tick_params(axis = 'both', labelsize = 12)
    plt.legend(bbox_to_anchor=(0., 1.02, 1., .102), loc='lower left',
    ncol=4, mode="expand", borderaxespad=0., edgecolor = 'black', title = legend_title,
    fontsize = 12)
    ax.grid(True, linestyle = ':')
    ax.set_axisbelow(True)
    plt.tight_layout()

    # heat maps with correlations
def draw_heatmap(df_new_heatmap, title, num):
    ""
    Visualization method to display heatmap with correlations of compared rankings
    generated using different methods.
```
Parameters
----------
data : DataFrame
   DataFrame with correlation values between compared rankings
title : str
   title of chart containing name of used correlation coefficient

Examples
--------
>>> draw_heatmap(df_new_heatmap, title)
"""
plt.figure(figsize = (8, 5))
sns.set(font_scale = 1.2)
heatmap = sns.heatmap(df_new_heatmap, annot=True, fmt=".3f", cmap="RdYlGn",
linewidth=0.5, linecolor='w')
plt.yticks(va="center")
plt.xlabel("MCDA methods")
plt.title("Correlation: ' + title)
plt.tight_layout()
title = title.replace($", ")
plt.savefig('./results/correlations_' + title + str(num) + '.eps')
plt.savefig('./results/correlations_' + title + str(num) + '.png')
plt.show()

# 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

An example decision problem involves selecting the best alternative for the location of an offshore wind farm based on Ziemba, P., Wątróbski, J., Ziolo, M., & Karczmarczyk, A. (2017). Using the PROSA method in offshore wind farm location problems. Energies, 10(11), 1755. DOI: https://doi.org/10.3390/en10111755. The problem includes four alternatives evaluated against twelve criteria. The evaluation criteria are displayed below. Preference direction ($\text{Max}$ denotes profit criteria with maximization aim, and $\text{Min}$ represents cost criteria with minimalization aim.

In this case, the preference function $V$-shape for PROMETHEE II and PROSA-C was selected, which means that there is required a vector $p$ (Preference Threshold) with values of the threshold of absolute preference, above which there is a total preference for one of the two actions and assigning the preference degree the value of 1.

More detailed information about preference functions, preference and indifference thresholds are available in reference paper:


[4]: crits = pd.read_csv('example1criteria.csv', index_col = 'Cj')
## Crits

<table>
<thead>
<tr>
<th>Criterion Name</th>
<th>Unit of Measurement</th>
<th>Preference Direction</th>
</tr>
</thead>
<tbody>
<tr>
<td>Investment cost</td>
<td>mln PLN</td>
<td>Min</td>
</tr>
<tr>
<td>Payback period</td>
<td>years</td>
<td>Min</td>
</tr>
<tr>
<td>Distance from power stations</td>
<td>km</td>
<td>Min</td>
</tr>
<tr>
<td>Mean sea water depth</td>
<td>m</td>
<td>Min</td>
</tr>
<tr>
<td>Undersea geological condition</td>
<td>points</td>
<td>Min</td>
</tr>
<tr>
<td>Employment</td>
<td>number</td>
<td>Max</td>
</tr>
<tr>
<td>Conflict with fisheries</td>
<td>points</td>
<td>Min</td>
</tr>
<tr>
<td>Density of shipping traffic</td>
<td>points</td>
<td>Min</td>
</tr>
<tr>
<td>Distance from shore</td>
<td>km</td>
<td>Max</td>
</tr>
<tr>
<td>Influence on protected areas</td>
<td>points</td>
<td>Min</td>
</tr>
<tr>
<td>CO2 reduction</td>
<td>tonnes</td>
<td>Max</td>
</tr>
<tr>
<td>SO2 reduction</td>
<td>tonnes</td>
<td>Max</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Weight of Criterion Preference function for PROMETHEE II and PROSA-C</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cj</td>
</tr>
<tr>
<td>-------------------------</td>
</tr>
<tr>
<td>C1</td>
</tr>
<tr>
<td>C2</td>
</tr>
<tr>
<td>C3</td>
</tr>
<tr>
<td>C4</td>
</tr>
<tr>
<td>C5</td>
</tr>
<tr>
<td>C6</td>
</tr>
<tr>
<td>C7</td>
</tr>
<tr>
<td>C8</td>
</tr>
<tr>
<td>C9</td>
</tr>
<tr>
<td>C10</td>
</tr>
<tr>
<td>C11</td>
</tr>
<tr>
<td>C12</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Preference Threshold (p) for PROMETHEE II and PROSA-C</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cj</td>
</tr>
<tr>
<td>-------------------------</td>
</tr>
<tr>
<td>C1</td>
</tr>
<tr>
<td>C2</td>
</tr>
<tr>
<td>C3</td>
</tr>
<tr>
<td>C4</td>
</tr>
<tr>
<td>C5</td>
</tr>
<tr>
<td>C6</td>
</tr>
<tr>
<td>C7</td>
</tr>
<tr>
<td>C8</td>
</tr>
<tr>
<td>C9</td>
</tr>
<tr>
<td>C10</td>
</tr>
<tr>
<td>C11</td>
</tr>
<tr>
<td>C12</td>
</tr>
</tbody>
</table>

Load a decision matrix containing the performance values of an example decision problem from a CSV file as a DataFrame.

```python
[6]: data = pd.read_csv('./results/dataset1.csv', index_col = 'Ai')
```

```python
[7]: data
```
Convert a DataFrame with decision matrix data to a NumPy array matrix to use the NumPy package for computation.

```python
[8]: matrix = data.to_numpy()
```

Provide criteria weights. The weights must be normalized using sum normalization and give a value of 1 after being summed.

```python
[9]: weights = np.array([20, 5, 5, 1.67, 1.67, 11.67, 11.67, 5, 5, 16.67, 8.33, 8.33])
weights = weights / np.sum(weights)
```

Provide criteria types. Assign 1 to profit criteria and -1 to cost criteria. Other values are not allowed.

```python
[11]: types = np.array([-1, -1, -1, -1, -1, 1, -1, -1, 1, -1, 1, 1])
```

Create the DataFrame for results rankings, including rankings provided by investigated MCDA methods.

```python
[12]: alt_names = [r'$A_{'+ str(el) + '}$' for el in range(1, matrix.shape[0] + 1)]
rank_results = pd.DataFrame(index=alt_names)
```

All MCDA methods investigated here sort alternatives according to preference values in descending order. It means that the best alternative has the highest preference value. Thus, we rank alternatives using the `rank_preferences` function with parameter `reverse=True`.

### 1.3.2 PROMETHEE II

In this case, you will use the V-shape preference function, so provide vector `p`.

```python
[13]: p = np.array([7280, 4, 13.4, 7.4, 3, 1662, 3, 3, 13.8, 3, 766240, 17820])
```

```python
[14]: # Create the PROMETHEE II method object
    promethee_II = PROMETHEE_II()
    # Assign preference functions to each criterion
    preference_functions = [promethee_II._vshape_function for pf in range(len(weights))]
```

(continues on next page)
# Calculate the utility function value (the preference value) for each alternative
pref = promethee_II(matrix, weights, types, preference_functions, p = p)
# Determine the ranking based on `pref` by sorting it in descending order (reverse = True)
rank = rank_preferences(pref, reverse=True)
# Save the PROMETHEE II ranking in the DataFrame
rank_results['PROMETHEE II'] = rank

PROMETHEE II utility function values

```
[15]: pref
array([-0.04449699, 0.18838474, -0.09537616, -0.04851159])
```

The best scored is alternative $A_2$ because it has the highest preference value.

PROMETHEE II ranking

```
[16]: rank
array([2, 1, 4, 3])
```

If you would like to use the linear preference function (V-shape with indifference) or Level function, Gaussian function, using PROMETHEE II is as follows: `pref = promethee_II(matrix, weights, types, preference_functions, p = p, q = q)` with additional q providing. U-shape function requires q, and the Usual function does not require any parameter.

Here is a summary of the required parameters for the preference function for PROMETHEE II and PROSA-C:

- **Usual**: No parameters
- **U-shape**: q
- **V-shape**: p
- **Level**: p, q
- **Linear (V-shape with indifference)**: p, q
- **Gaussian**: p, q

### 1.3.3 PROSA-C

Using PROSA-C is similar to PROMETHEE II, but it requires providing an additional argument, a vector $s$ including sustainability coefficients for each criterion. The sustainability coefficient determines the reduction of criteria compensation. If you do not specify the arguments $p$, $q$, and $s$ for the PROMETHEE II and PROSA methods, they will be set automatically, based on:

Ziemba, P. (2020). Multi-criteria stochastic selection of electric vehicles for the sustainable development of local government and state administration units in Poland. Energies, 13(23), 6299. DOI: https://doi.org/10.3390/en13236299

```python
u = np.sqrt(np.sum(np.square(np.mean(matrix, axis = 0) - matrix), axis = 0) / matrix.shape[0])
p = 2 * u
q = 0.5 * u
s = np.repeat(0.3, len(weights))
```
ARAS

In contrast to PROMETHEE II and PROSA-C, ARAS requires normalization of the decision matrix. In the ARAS method you can choose one technique for normalization decision matrix from five available: linear_normalization, minmax_normalization, max_normalization, sum_normalization and vector_normalization. The default normalization for ARAS is sum_normalization. The normalization method is chosen in method object initialization in the constructor of the method. In the case of other MCDA methods presented below, normalization method selection is analogous.

COPRAS

The COPRAS method also requires the normalization of the decision matrix. The default normalization method for COPRAS is the Sum normalization used like for profit criteria because in COPRAS decision matrix is normalized before dividing criteria into profit and cost. There is also an option to select another normalization method from normalizations submodule. Chosen normalization method is performed automatically like for profit criteria, according to the COPRAS algorithm.

CRADIS

The CRADIS method enables choosing a normalization technique. The default is linear_normalization.

1.3.4 ARAS

```python
s = np.repeat(0.3, len(weights))
prosa_c = PROSA_C()
pref = prosa_c(matrix, weights, types, preference_functions, p = p, s = s)
rank = rank_preferences(pref, reverse=True)
rank_results['PROSA C'] = rank
```

1.3.5 COPRAS

```python
aras = ARAS(normalization_method=norms.linear_normalization)
pref = aras(matrix, weights, types)
rank = rank_preferences(pref, reverse=True)
rank_results['ARAS'] = rank
```

1.3.6 CRADIS

```python
cradis = CRADIS()
pref = cradis(matrix, weights, types)
rank = rank_preferences(pref, reverse=True)
rank_results['CRADIS'] = rank
```
1.3.7 MARCOS

The MARCOS method does not enable the selection of the normalization technique because this method normalizes the decision matrix using a specific method involving ideal and anti-ideal solutions.

```python
[21]: marcos = MARCOS()
pref = marcos(matrix, weights, types)
rank = rank_preferences(pref, reverse=True)
rank_results['MARCOS'] = rank
```

1.3.8 SAW

The SAW method enables choosing a normalization technique. The default is linear_normalization.

```python
[22]: saw = SAW()
pref = saw(matrix, weights, types)
rank = rank_preferences(pref, reverse=True)
rank_results['SAW'] = rank
```

Display rankings determined by each investigated MCDA method.

```python
[23]: rank_results

<table>
<thead>
<tr>
<th></th>
<th>PROMETHEE II</th>
<th>PROSA</th>
<th>ARAS</th>
<th>COPRAS</th>
<th>CRADIS</th>
<th>MARCOS</th>
<th>SAW</th>
</tr>
</thead>
<tbody>
<tr>
<td>$A_{1}$</td>
<td>2</td>
<td>3</td>
<td>3</td>
<td>2</td>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>$A_{2}$</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>$A_{3}$</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>$A_{4}$</td>
<td>3</td>
<td>2</td>
<td>2</td>
<td>3</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
</tbody>
</table>
```

```python
[24]: plot_barplot(rank_results, legend_title='MCDA methods', num = 1)
```

The PostScript backend does not support transparency; partially transparent artists will be rendered opaque.

Calculate correlations between rankings using Weighted Spearman correlation coefficient.

```python
[25]: method_types = list(rank_results.columns)
dict_new_heatmap_rw = Create_dictionary()
for el in method_types:
    dict_new_heatmap_rw.add(el, [])
```

(continues on next page)
# 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(rank_results[i], rank_results[j]))

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

Display the heat map of correlations.

[26]: # correlation matrix with rw coefficient

<table>
<thead>
<tr>
<th></th>
<th>SAW</th>
<th>MARCOS</th>
<th>CRADIS</th>
<th>COPRAS</th>
<th>ARAS</th>
<th>PROSA-C</th>
<th>PROMETHEE II</th>
</tr>
</thead>
<tbody>
<tr>
<td>SAW</td>
<td>1.000</td>
<td>1.000</td>
<td>0.800</td>
<td>1.000</td>
<td>1.000</td>
<td>1.000</td>
<td>1.000</td>
</tr>
<tr>
<td>MARCOS</td>
<td>1.000</td>
<td>1.000</td>
<td>1.000</td>
<td>0.800</td>
<td>1.000</td>
<td>1.000</td>
<td>1.000</td>
</tr>
<tr>
<td>CRADIS</td>
<td>0.800</td>
<td>1.000</td>
<td>1.000</td>
<td>0.800</td>
<td>1.000</td>
<td>1.000</td>
<td>1.000</td>
</tr>
<tr>
<td>COPRAS</td>
<td>1.000</td>
<td>0.800</td>
<td>0.800</td>
<td>1.000</td>
<td>0.800</td>
<td>0.800</td>
<td>0.800</td>
</tr>
<tr>
<td>ARAS</td>
<td>0.800</td>
<td>1.000</td>
<td>1.000</td>
<td>0.800</td>
<td>1.000</td>
<td>1.000</td>
<td>1.000</td>
</tr>
<tr>
<td>PROSA-C</td>
<td>0.800</td>
<td>1.000</td>
<td>1.000</td>
<td>0.800</td>
<td>1.000</td>
<td>1.000</td>
<td>1.000</td>
</tr>
<tr>
<td>PROMETHEE II</td>
<td>1.000</td>
<td>0.800</td>
<td>0.800</td>
<td>1.000</td>
<td>0.800</td>
<td>0.800</td>
<td>0.800</td>
</tr>
</tbody>
</table>

MCDA methods

**1.3.9 AHP**

**Classical usage of AHP**

The classical application of AHP using pairwise comparisons of criteria and alternatives by decision-maker based on:

Papathanasiou, J., & Ploskas, N. (2018). AHP. In Multiple Criteria Decision Aid (pp. 109-129). Springer, Cham. DOI: https://doi.org/10.1007/978-3-319-91648-4_5

Step 1a. Provide matrix with pairwise comparison values of criteria

[27]:

```python
PCcriteria = np.array([[1, 1, 5, 3],
                      [1, 1, 5, 3],
                      [1/5, 1/5, 1, 1/3],
                      [1/3, 1/3, 3, 1]])
```

[28]:

```python
PCcriteria
```

array([[1., 1., 5., 3.],
       [1., 1., 5., 3.],
       [1., 1., 5., 3.],
       [1., 1., 5., 3.]])

(continues on next page)
Step 1b. Provide matrices with pairwise comparison values for alternatives

Step 2. Create the AHP method object and check the consistency of matrix with criteria comparison. Values equal and below 0.1 (≤ 0.1) denote that the matrix is consistent and it is correct.

Step 3. Calculate the priority vector of criteria (criteria weights). The AHP method can be used only for criteria weights determination incorporating a decision-maker.

Step 4. Form pairwise comparison matrices of the alternatives for each criterion.
alt_matrices.append(PCM2)
alt_matrices.append(PCM3)
alt_matrices.append(PCM4)

Step 5. This step checks the consistency of pairwise comparison matrices of the alternatives. Values equal and below 0.1 denote that the matrix is consistent and it is correct.

```python
[35]: calculate_priority_vector_method = ahp._calculate_eigenvector
pref = ahp._classic_ahp(alt_matrices, weights, calculate_priority_vector_method)
```

Inconsistency index:
- 0.03349113123332669
- 0.026437064499443722
- 0.022144721786307425
- 0.039785112041738534

AHP utility (preference) function values:

```python
[36]: pref
```

```python
[37]: rank = rank_preferences(pref, reverse = True)
```

AHP ranking

```python
[38]: rank
```

Another usage of AHP for ranking generation with numerical performance values, weights and criteria types

If you have a decision matrix with numerical performance values, a vector with numerical criteria weights, and determined criteria types (profit or cost), you can use the AHP method like other MCDA methods (for example, SAW):

```python
[39]: matrix = np.array([[0.75, 0.50, 0.75, 0, 0, 0, 1],
                      [0.75, 1, 0.75, 0, 0, 0, 0.75],
                      [0.75, 0.75, 0.75, 0, 0.50, 0.25, 1],
                      [0.50, 0.50, 0.75, 1, 0.50, 0, 0.75]])
weights = np.array([0.1, 0.1, 0.1, 0.15, 0.2, 0.25, 0.1])
types = np.array([1, 1, 1, 1, 1, 1, 1])
ahp = AHP(normalization_method=norms.linear_normalization)
pref = ahp(matrix, weights, types)
```

```python
[40]: pref
```

```python
[40]: array([0.35 , 0.375 , 0.825 , 0.64166667])
```

1.3. Illustrative examples of MCDA methods: CRADIS, COPRAS, ARAS, MARCOS, PROMETHEE II, PROSA-C, SAW, AHP
1.4 Illustrative example for weighting methods

This example explains the usage of the Python 3 library package pyrepo that provides methods for multi-criteria decision analysis using objective weighting methods. This library contains module `weighting_methods` with the following weighting methods:

1. Equal `equal_weighting`
2. Entropy `entropy_weighting`
3. Standard deviation `std_weighting`
4. CRITIC `critic_weighting`
5. Gini coefficient-based `gini_weighting`
6. MEREC `merec_weighting`
7. Statistical variance `stat_var_weighting`
8. CILOS `cilos_weighting`
9. IDOCRIW `idocriw_weighting`
10. Angle `angle_weighting`
11. Coefficient of variance `coeff_var_weighting`

In addition to the weighting methods, the library also provides other methods necessary for multi-criteria decision analysis, which are as follows:

The VIKOR method for multi-criteria decision analysis VIKOR in module `mcda_methods`,

Normalization techniques:

1. Linear `linear_normalization`
2. Minimum-Maximum `minmax_normalization`
3. Maximum `max_normalization`
4. Sum `sum_normalization`
5. Vector `vector_normalization`

Correlation coefficients:

1. Spearman rank correlation coefficient `rs_spearman`
2. Weighted Spearman rank correlation coefficient `rw_weighted_spearman`
3. Pearson coefficient `pearson_coeff`

Import other necessary Python modules.
Import the necessary modules and methods from package pyrepo.

from pyrepo_mcda.mcda_methods import VIKOR
from pyrepo_mcda.mcda_methods import VIKOR_SMAA
from pyrepo_mcda.additions import rank_preferences
from pyrepo_mcda import correlations as corrs
from pyrepo_mcda import normalizations as norm_methods
from pyrepo_mcda import weighting_methods as mcda_weights

Functions for results visualization.

# Functions for visualizations

def plot_barplot(df_plot, x_name, y_name, title):
    ""
    Display stacked column chart of weights for criteria for \`x_name == Weighting\`
    methods
    and column chart of ranks for alternatives \`x_name == Alternatives\`
    Parameters
    ----------
    df_plot : dataframe
        dataframe with criteria weights calculated different weighting methods
        or with alternatives rankings for different weighting methods
    x_name : str
        name of x axis, Alternatives or Weighting methods
    y_name : str
        name of y axis, Ranks or Weight values
    title : str
        name of chart title, Weighting methods or Criteria
    Examples
    --------
    >>> plot_barplot(df_plot, x_name, y_name, title)
    """

    list_rank = np.arange(1, len(df_plot) + 1, 1)
    stacked = True
    width = 0.5
    if x_name == 'Alternatives':
        stacked = False
        width = 0.8
    elif x_name == 'Alternative':
        pass
    else:
        df_plot = df_plot.T
ax = df_plot.plot(kind='bar', width = width, stacked=stacked, edgecolor = 'black',
figsize = (9,4))
    ax.set_xlabel(x_name, fontsize = 12)
    ax.set_ylabel(y_name, fontsize = 12)

    if x_name == 'Alternatives':
        ax.set_yticks(list_rank)
    ax.set_xticklabels(df_plot.index, rotation = 'horizontal')
    ax.tick_params(axis = 'both', labelsize = 12)

plt.legend(bbox_to_anchor=(0., 1.02, 1., .102), loc='lower left',
ncol=4, mode='expand', borderaxespad=0., edgecolor = 'black', title = title,
fontsize = 11)
    ax.grid(True, linestyle = '--')
    ax.set_axisbelow(True)
plt.tight_layout()
plt.savefig('results/bar_chart_weights_' + x_name + '.pdf')
plt.savefig('results/bar_chart_weights_' + x_name + '.eps')
plt.show()

def draw_heatmap(data, title):
    """
    Display heatmap with correlations of compared rankings generated using different methods
    Parameters
    ----------
    data : dataframe
        dataframe with correlation values between compared rankings
    title : str
        title of chart containing name of used correlation coefficient
    Examples
    --------
    >>> draw_heatmap(data, title)
    """

    plt.figure(figsize = (6, 4))
    sns.set(font_scale=1.0)
    heatmap = sns.heatmap(data, annot=True, fmt='.2f', cmap="RdYlBu",
                         linewidth=0.5, linecolor='w')
    plt.yticks(va="center")
    plt.xlabel('Weighting methods')
    title = title.replace('$', '')
    title = title.replace('{', '')
    title = title.replace('}', '')
    plt.title('Correlation coefficient: ' + title)
    plt.tight_layout()
    plt.savefig('results/heatmap_weights.pdf')

(continues on next page)
```python
plt.savefig('results/heatmap_weights.eps')
plt.show()

def draw_heatmap_smaa(data, title):
    """
    Display heatmap with correlations of compared rankings generated using different methods
    
    Parameters
    ----------
    data : dataframe
        dataframe with correlation values between compared rankings
    title : str
        title of chart containing name of used correlation coefficient
    
    Examples
    --------
    >>> draw_heatmap(data, title)
    """
    sns.set(font_scale=1.0)
    heatmap = sns.heatmap(data, annot=True, fmt='.2f', cmap='RdYlBu_r',
                          linewidth=0.05, linecolor='w')
    plt.yticks(rotation=0)
    plt.ylabel('Alternatives')
    plt.tick_params(labelbottom=False, labeltop=True)
    plt.title(title)
    plt.tight_layout()
    plt.savefig('results/heatmap_smaa.pdf')
    plt.savefig('results/heatmap_smaa.eps')
    plt.show()

def plot_boxplot(data):
    """
    Display boxplot showing distribution of criteria weights determined with different methods.
    
    Parameters
    ----------
    data : dataframe
        dataframe with correlation values between compared rankings
    
    Examples
    --------
    >>> plot_boxplot(data)
    """
    df_melted = pd.melt(data)
    plt.figure(figsize = (7, 4))
```

(continues on next page)
ax = sns.boxplot(x = 'variable', y = 'value', data = df_melted, width = 0.6)
ax.grid(True, linestyle = '--')
ax.set_axisbelow(True)
ax.set_xlabel('Criterion', fontsize = 12)
ax.set_ylabel('Different weights distribution', fontsize = 12)
plt.tight_layout()
plt.savefig('results/boxplot_weights.pdf')
plt.savefig('results/boxplot_weights.eps')
plt.show()

# 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

As an illustrative example, a dataset will be used containing performances of the twelve best-selling electric cars in 2021 according to a ranking available at https://www.caranddriver.com/features/g36278968/best-selling-evs-of-2021/ The dataset is displayed below. $A_1$-$A_{12}$ are the individual alternatives in rows, columns $C_1$-$C_{11}$ denote the criteria, and the Type row contains the criteria type, where 1 indicates a profit criterion (stimulant) and -1 a cost criterion (destimulant). The following are the evaluation criteria for the electric cars evaluated in this research.

<table>
<thead>
<tr>
<th>Cj</th>
<th>Name</th>
<th>Unit</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>C1</td>
<td>Max speed</td>
<td>mph</td>
<td>1</td>
</tr>
<tr>
<td>C2</td>
<td>Battery capacity</td>
<td>kWh</td>
<td>1</td>
</tr>
<tr>
<td>C3</td>
<td>Electric motor</td>
<td>kW</td>
<td>1</td>
</tr>
<tr>
<td>C4</td>
<td>Maximum torque</td>
<td>Nm</td>
<td>1</td>
</tr>
<tr>
<td>C5</td>
<td>Horsepower</td>
<td>hp</td>
<td>1</td>
</tr>
<tr>
<td>C6</td>
<td>EPA Fuel Economy Combined</td>
<td>MPGe</td>
<td>1</td>
</tr>
<tr>
<td>C7</td>
<td>EPA Fuel Economy City</td>
<td>MPGe</td>
<td>1</td>
</tr>
<tr>
<td>C8</td>
<td>EPA Fuel Economy Highway</td>
<td>MPGe</td>
<td>1</td>
</tr>
<tr>
<td>C9</td>
<td>EPA range</td>
<td>miles</td>
<td>1</td>
</tr>
<tr>
<td>C10</td>
<td>Turning Diameter / Radius, curb to curb</td>
<td>feet</td>
<td>-1</td>
</tr>
<tr>
<td>C11</td>
<td>Base price</td>
<td>USD</td>
<td>-1</td>
</tr>
</tbody>
</table>

As an illustrative example, a dataset will be used containing performances of the twelve best-selling electric cars in 2021 according to a ranking available at https://www.caranddriver.com/features/g36278968/best-selling-evs-of-2021/ The dataset is displayed below. $A_1$-$A_{12}$ are the individual alternatives in rows, columns $C_1$-$C_{11}$ denote the criteria, and the Type row contains the criteria type, where 1 indicates a profit criterion (stimulant) and -1 a cost criterion (destimulant). The following are the evaluation criteria for the electric cars evaluated in this research.

<table>
<thead>
<tr>
<th>Ai</th>
<th>Name</th>
<th>C1 Max speed [mph]</th>
<th>C2 Battery [kWh]</th>
</tr>
</thead>
<tbody>
<tr>
<td>A1</td>
<td>Tesla Model Y</td>
<td>155.3</td>
<td>74.0</td>
</tr>
<tr>
<td>A2</td>
<td>Tesla Model 3</td>
<td>162.2</td>
<td>79.5</td>
</tr>
<tr>
<td>A3</td>
<td>Ford Mustang Mach-E</td>
<td>112.5</td>
<td>68.0</td>
</tr>
</tbody>
</table>

(continues on next page)
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>340</td>
<td>673</td>
<td>456.0</td>
<td>111</td>
<td>115</td>
<td>106</td>
</tr>
<tr>
<td>2</td>
<td>247</td>
<td>639</td>
<td>283.0</td>
<td>113</td>
<td>118</td>
<td>107</td>
</tr>
<tr>
<td>3</td>
<td>198</td>
<td>430</td>
<td>266.0</td>
<td>98</td>
<td>105</td>
<td>91</td>
</tr>
<tr>
<td>4</td>
<td>150</td>
<td>360</td>
<td>201.2</td>
<td>120</td>
<td>131</td>
<td>109</td>
</tr>
<tr>
<td>5</td>
<td>150</td>
<td>310</td>
<td>201.2</td>
<td>97</td>
<td>102</td>
<td>90</td>
</tr>
<tr>
<td>6</td>
<td>110</td>
<td>320</td>
<td>147.5</td>
<td>111</td>
<td>134.1</td>
<td>120</td>
</tr>
<tr>
<td>7</td>
<td>125</td>
<td>247</td>
<td>187.7</td>
<td>78</td>
<td>150</td>
<td>98</td>
</tr>
<tr>
<td>8</td>
<td>160</td>
<td>300</td>
<td>214.6</td>
<td>79</td>
<td>502.9</td>
<td>120</td>
</tr>
<tr>
<td>9</td>
<td>205</td>
<td>420</td>
<td>502.9</td>
<td>120</td>
<td>150</td>
<td>98</td>
</tr>
<tr>
<td>10</td>
<td>100</td>
<td>395</td>
<td>134.1</td>
<td>120</td>
<td>502.9</td>
<td>133</td>
</tr>
<tr>
<td>11</td>
<td>205</td>
<td>420</td>
<td>150</td>
<td>111</td>
<td>136.1</td>
<td>133</td>
</tr>
<tr>
<td>12</td>
<td>101</td>
<td>295</td>
<td>1.0</td>
<td>1</td>
<td>1.0</td>
<td>1</td>
</tr>
</tbody>
</table>
Load a decision matrix containing only the performance values of the alternatives against the criteria and the criteria type in the last row, as shown below. Transform the decision matrix and criteria type from dataframe to NumPy array.

```python
[6]: # Load data from CSV
    filename = 'dataset_cars.csv'
    data = pd.read_csv(filename, index_col = 'Ai')
    # Load decision matrix from CSV
    df_data = data.iloc[:len(data) - 1, :]
    # Criteria types are in the last row of CSV
    types = data.iloc[len(data) - 1, :].to_numpy()
    # Convert decision matrix from dataframe to numpy ndarray type for faster calculations.
    matrix = df_data.to_numpy()
```

(continues on next page)
# Symbols for alternatives Ai
list_alt_names = [r'$A_{' + str(i) + '}$' for i in range(1, df_data.shape[0] + 1)]

# Symbols for columns Cj
cols = [r'$C_{' + str(j) + '}$' for j in range(1, data.shape[1] + 1)]

print('Decision matrix')

def_data

Decision matrix

<table>
<thead>
<tr>
<th>Ai</th>
<th>C1</th>
<th>C2</th>
<th>C3</th>
<th>C4</th>
<th>C5</th>
<th>C6</th>
<th>C7</th>
<th>C8</th>
<th>C9</th>
<th>C10</th>
<th>C11</th>
</tr>
</thead>
<tbody>
<tr>
<td>A1</td>
<td>155.3</td>
<td>74.0</td>
<td>340</td>
<td>673</td>
<td>456.0</td>
<td>111</td>
<td>115</td>
<td>106</td>
<td>244</td>
<td>39.8</td>
<td>65440</td>
</tr>
<tr>
<td>A2</td>
<td>162.2</td>
<td>79.5</td>
<td>247</td>
<td>639</td>
<td>283.0</td>
<td>113</td>
<td>118</td>
<td>107</td>
<td>263</td>
<td>38.8</td>
<td>60440</td>
</tr>
<tr>
<td>A3</td>
<td>112.5</td>
<td>68.0</td>
<td>198</td>
<td>430</td>
<td>266.0</td>
<td>98</td>
<td>105</td>
<td>91</td>
<td>230</td>
<td>38.1</td>
<td>56575</td>
</tr>
<tr>
<td>A4</td>
<td>90.1</td>
<td>66.0</td>
<td>150</td>
<td>360</td>
<td>201.2</td>
<td>120</td>
<td>131</td>
<td>109</td>
<td>259</td>
<td>34.8</td>
<td>32495</td>
</tr>
<tr>
<td>A5</td>
<td>99.4</td>
<td>77.0</td>
<td>150</td>
<td>310</td>
<td>201.2</td>
<td>97</td>
<td>102</td>
<td>90</td>
<td>260</td>
<td>36.4</td>
<td>45635</td>
</tr>
<tr>
<td>A6</td>
<td>89.5</td>
<td>40.0</td>
<td>110</td>
<td>320</td>
<td>147.5</td>
<td>111</td>
<td>123</td>
<td>99</td>
<td>226</td>
<td>34.8</td>
<td>28425</td>
</tr>
<tr>
<td>A7</td>
<td>124.3</td>
<td>95.0</td>
<td>125</td>
<td>247</td>
<td>187.7</td>
<td>78</td>
<td>78</td>
<td>77</td>
<td>222</td>
<td>40.0</td>
<td>84595</td>
</tr>
<tr>
<td>A8</td>
<td>155.3</td>
<td>79.2</td>
<td>160</td>
<td>300</td>
<td>214.6</td>
<td>79</td>
<td>79</td>
<td>80</td>
<td>227</td>
<td>38.4</td>
<td>105150</td>
</tr>
<tr>
<td>A9</td>
<td>162.2</td>
<td>100.0</td>
<td>205</td>
<td>420</td>
<td>502.9</td>
<td>120</td>
<td>124</td>
<td>115</td>
<td>402</td>
<td>40.3</td>
<td>96440</td>
</tr>
<tr>
<td>A10</td>
<td>96.3</td>
<td>39.2</td>
<td>100</td>
<td>395</td>
<td>134.1</td>
<td>120</td>
<td>132</td>
<td>108</td>
<td>258</td>
<td>34.8</td>
<td>35245</td>
</tr>
<tr>
<td>A11</td>
<td>162.2</td>
<td>100.0</td>
<td>205</td>
<td>420</td>
<td>502.9</td>
<td>98</td>
<td>103</td>
<td>93</td>
<td>371</td>
<td>40.8</td>
<td>127940</td>
</tr>
<tr>
<td>A12</td>
<td>102.5</td>
<td>38.3</td>
<td>101</td>
<td>295</td>
<td>136.1</td>
<td>133</td>
<td>145</td>
<td>121</td>
<td>170</td>
<td>34.8</td>
<td>34250</td>
</tr>
</tbody>
</table>

print('Criteria types')
types

Criteria types

array([ 1., 1., 1., 1., 1., 1., 1., -1., -1., -1.])

1.4.1 Objective weighting methods

Calculate the weights with the selected weighing method. In this case, the Entropy weighting method (entropy_weighting) is selected.

weights = mcda_weights.entropy_weighting(matrix)
df_weights = pd.DataFrame(weights.reshape(1, -1), index = ['Weights'], columns = cols)
df_weights

Weights 0.057741 0.099843 0.142673 0.096488 0.236087 0.024544 0.032432

Weights 0.142673 0.099843 0.057741 0.236087 0.032432 0.096488 0.234244

Use the VIKOR method to determine the value of the preference function (pref) and the ranking of alternatives (rank). The VIKOR method ranks alternatives ascendingly according to preference function values, so the reverse parameter in the rank_preferences method is set to False.

# Create the VIKOR method object
vikor = VIKOR(normalization_method=norm_methods.minmax_normalization)

(continues on next page)
# Calculate alternatives preference function values with VIKOR method

def vikor(matrix, weights, types):
    pref = vikor(matrix, weights, types)

# rank alternatives according to preference values

def rank_preferences(pref, reverse = False):
    rank = rank_preferences(pref, reverse = False)
    df_results = pd.DataFrame(index = list_alt_names)
    df_results['Pref'] = pref
    df_results['Rank'] = rank
    df_results

The second part of the manual contains codes for benchmarking against several different criteria weighting methods. List the weighting methods you wish to explore.

# Create a list with weighting methods that you want to explore
weighting_methods_set = [
    mcda_weights.equal_weighting,
    mcda_weights.entropy_weighting,
    mcda_weights.critic_weighting,
    mcda_weights.gini_weighting,
    mcda_weights.merec_weighting,
    mcda_weights.stat_var_weighting,
    mcda_weights.idocriw_weighting,
    mcda_weights.angle_weighting,
    mcda_weights.coeff_var_weighting
]

Below is a loop with code to collect results for each weighting technique. Then display the results, namely weights, preference function values and rankings.

# Create dataframes for weights, preference function values and rankings determined using different weighting methods

def_weights = pd.DataFrame(index = cols)
df_preferences = pd.DataFrame(index = list_alt_names)
df_rankings = pd.DataFrame(index = list_alt_names)

# Create dataframes for weights, preference function values and rankings determined using different weighting methods
df_weights = pd.DataFrame(index = cols)
df_preferences = pd.DataFrame(index = list_alt_names)
df_rankings = pd.DataFrame(index = list_alt_names)

# Create the VIKOR method object
vikor = VIKOR()
for weight_type in weighting_methods_set:
    if weight_type.__name__ in ['cilos_weighting', 'idocriw_weighting', 'angle_weighting', 'merec_weighting']:
        weights = weight_type(matrix, types)
    else:
        weights = weight_type(matrix)
    df_weights[weight_type.__name__[:-10].upper().replace('_', ' ')] = weights
    pref = vikor(matrix, weights, types)
    rank = rank_preferences(pref, reverse = False)
    df_preferences[weight_type.__name__[:-10].upper().replace('_', ' ')] = pref
    df_rankings[weight_type.__name__[:-10].upper().replace('_', ' ')] = rank

### df_weights

| C_1  | 0.090909 | 0.057741 | 0.093960 | 0.080882 | 0.067363 | 0.143855 |
| C_2  | 0.090909 | 0.099843 | 0.099277 | 0.103800 | 0.125195 | 0.103976 |
| C_3  | 0.090909 | 0.142673 | 0.066132 | 0.128202 | 0.103489 | 0.067308 |
| C_4  | 0.090909 | 0.096488 | 0.075874 | 0.103200 | 0.093050 | 0.076665 |
| C_5  | 0.090909 | 0.236087 | 0.071195 | 0.163513 | 0.124581 | 0.112880 |
| C_6  | 0.090909 | 0.024544 | 0.112865 | 0.052308 | 0.064886 | 0.074361 |
| C_7  | 0.090909 | 0.032432 | 0.120602 | 0.060388 | 0.077107 | 0.073925 |
| C_8  | 0.090909 | 0.018126 | 0.103536 | 0.046188 | 0.053708 | 0.076150 |
| C_9  | 0.090909 | 0.053958 | 0.065514 | 0.073099 | 0.087109 | 0.060565 |
| C_10 | 0.090909 | 0.083863 | 0.098432 | 0.021151 | 0.018566 | 0.126025 |
| C_11 | 0.090909 | 0.234244 | 0.092612 | 0.167270 | 0.184947 | 0.084289 |

### IDOCRIW ANGLE COEFF VAR

| C_1  | 0.089362 | 0.081732 | 0.079378 |
| C_2  | 0.074065 | 0.103002 | 0.101129 |
| C_3  | 0.094271 | 0.129702 | 0.129595 |
| C_4  | 0.079572 | 0.108379 | 0.106746 |
| C_5  | 0.154235 | 0.162354 | 0.166788 |
| C_6  | 0.071876 | 0.053145 | 0.051074 |
| C_7  | 0.076822 | 0.060739 | 0.058510 |
| C_8  | 0.069418 | 0.046061 | 0.044183 |
| C_9  | 0.037902 | 0.081691 | 0.079337 |
| C_10 | 0.017062 | 0.021711 | 0.020518 |
| C_11 | 0.231276 | 0.151484 | 0.162742 |

### df_preferences

| A_1  | 0.276946 | 0.000000 | 0.193324 | 0.000000 | 0.000000 | 0.210477 |
| A_2  | 0.061114 | 0.325154 | 0.053863 | 0.267784 | 0.06602 | 0.062729 |

1.4. Illustrative example for weighting methods
### df_rankings

<table>
<thead>
<tr>
<th>IDOCRIW</th>
<th>ANGLE</th>
<th>COEFF VAR</th>
</tr>
</thead>
<tbody>
<tr>
<td>$A_{1}$</td>
<td>1</td>
<td>0.000000</td>
</tr>
<tr>
<td>$A_{2}$</td>
<td>2</td>
<td>0.100057</td>
</tr>
<tr>
<td>$A_{3}$</td>
<td>4</td>
<td>0.332813</td>
</tr>
<tr>
<td>$A_{4}$</td>
<td>6</td>
<td>0.353278</td>
</tr>
<tr>
<td>$A_{5}$</td>
<td>5</td>
<td>0.473333</td>
</tr>
<tr>
<td>$A_{6}$</td>
<td>6</td>
<td>0.549559</td>
</tr>
<tr>
<td>$A_{7}$</td>
<td>7</td>
<td>0.657640</td>
</tr>
<tr>
<td>$A_{8}$</td>
<td>8</td>
<td>0.798193</td>
</tr>
<tr>
<td>$A_{9}$</td>
<td>9</td>
<td>0.301515</td>
</tr>
<tr>
<td>$A_{10}$</td>
<td>10</td>
<td>0.528558</td>
</tr>
<tr>
<td>$A_{11}$</td>
<td>11</td>
<td>0.732254</td>
</tr>
<tr>
<td>$A_{12}$</td>
<td>12</td>
<td>0.495033</td>
</tr>
</tbody>
</table>

### COEFF VAR

<table>
<thead>
<tr>
<th>$A_{1}$</th>
<th>$A_{2}$</th>
<th>$A_{3}$</th>
<th>$A_{4}$</th>
<th>$A_{5}$</th>
<th>$A_{6}$</th>
<th>$A_{7}$</th>
<th>$A_{8}$</th>
<th>$A_{9}$</th>
<th>$A_{10}$</th>
<th>$A_{11}$</th>
<th>$A_{12}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>7</td>
<td>12</td>
<td>9</td>
<td>8</td>
<td>2</td>
<td>10</td>
<td>6</td>
<td>11</td>
</tr>
</tbody>
</table>

(continues on next page)
Visualize the results as column graphs of weights, rankings, and correlations.

```
[15]: plot_barplot(df_weights, 'Weighting methods', 'Weight value', 'Criteria')
```

The PostScript backend does not support transparency; partially transparent artists will be rendered opaque.

```
[16]: plot_boxplot(df_weights.T)
```

```
[17]: plot_barplot(df_rankings, 'Alternatives', 'Rank', 'Weighting methods')
```

The PostScript backend does not support transparency; partially transparent artists will be rendered opaque.
results = copy.deepcopy(df_rankings)
method_types = list(results.columns)
dict_new_heatmap_rw = Create_dictionary()

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

# 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(results[i], results[j]))

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

# correlation matrix with rw coefficient
draw_heatmap(df_new_heatmap_rw, r'$r_w$')
1.4.2 Stochastic Multicriteria Acceptability Analysis Method (SMAA)

[19]: cols_ai = [str(el) for el in range(1, matrix.shape[0] + 1)]

[20]: # criteria number
n = matrix.shape[1]
# number of SMAA iterations
iterations = 10000

[21]: # create the VIKOR_SMAA method object
vikor_smaa = VIKOR_SMAA()
# generate multiple weight vectors in matrix
weight_vectors = vikor_smaa._generate_weights(n, iterations)

[22]: # Calculate the rank acceptability index, central weight vector and final ranking
rank_acceptability_index, central_weight_vector, rank_scores = vikor_smaa(matrix, weight_‐
˓→vectors, types)

[23]: acc_in_df = pd.DataFrame(rank_acceptability_index, index = list_alt_names, columns =
˓→cols_ai)
acc_in_df.to_csv('results_smaa/ai.csv')

Rank acceptability indexes

This is dataframe with rank acceptability indexes for each alternative in relation to ranks. Rank acceptability index shows the share of different scores placing an alternative in a given rank.

[24]: acc_in_df

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>A_{1}</td>
<td>0.2361</td>
<td>0.2458</td>
<td>0.1879</td>
<td>0.1354</td>
<td>0.0507</td>
<td>0.0546</td>
<td>0.0227</td>
<td>0.0480</td>
</tr>
<tr>
<td>A_{2}</td>
<td>0.2208</td>
<td>0.3555</td>
<td>0.2194</td>
<td>0.1165</td>
<td>0.0455</td>
<td>0.0345</td>
<td>0.0078</td>
<td>0.0000</td>
</tr>
<tr>
<td>A_{3}</td>
<td>0.0001</td>
<td>0.0111</td>
<td>0.0229</td>
<td>0.0725</td>
<td>0.2870</td>
<td>0.1485</td>
<td>0.1467</td>
<td>0.1366</td>
</tr>
<tr>
<td>A_{4}</td>
<td>0.1136</td>
<td>0.0670</td>
<td>0.0717</td>
<td>0.1356</td>
<td>0.1719</td>
<td>0.2304</td>
<td>0.0778</td>
<td>0.0375</td>
</tr>
<tr>
<td>A_{5}</td>
<td>0.0003</td>
<td>0.0123</td>
<td>0.0129</td>
<td>0.0217</td>
<td>0.0780</td>
<td>0.0999</td>
<td>0.2542</td>
<td>0.1427</td>
</tr>
<tr>
<td>A_{6}</td>
<td>0.0000</td>
<td>0.0007</td>
<td>0.0070</td>
<td>0.0511</td>
<td>0.0251</td>
<td>0.0369</td>
<td>0.1353</td>
<td>0.1146</td>
</tr>
<tr>
<td>A_{7}</td>
<td>0.0000</td>
<td>0.0000</td>
<td>0.0011</td>
<td>0.0012</td>
<td>0.0062</td>
<td>0.0298</td>
<td>0.0306</td>
<td>0.0327</td>
</tr>
<tr>
<td>A_{8}</td>
<td>0.0000</td>
<td>0.0011</td>
<td>0.0025</td>
<td>0.0050</td>
<td>0.0626</td>
<td>0.0392</td>
<td>0.0569</td>
<td>0.1442</td>
</tr>
<tr>
<td>A_{9}</td>
<td>0.3802</td>
<td>0.1025</td>
<td>0.2888</td>
<td>0.0389</td>
<td>0.0271</td>
<td>0.0282</td>
<td>0.0239</td>
<td>0.0177</td>
</tr>
<tr>
<td>A_{10}</td>
<td>0.0106</td>
<td>0.0425</td>
<td>0.0703</td>
<td>0.0684</td>
<td>0.0860</td>
<td>0.1690</td>
<td>0.0987</td>
<td>0.0715</td>
</tr>
<tr>
<td>A_{11}</td>
<td>0.0009</td>
<td>0.1083</td>
<td>0.0779</td>
<td>0.2967</td>
<td>0.0668</td>
<td>0.0490</td>
<td>0.0606</td>
<td>0.0944</td>
</tr>
<tr>
<td>A_{12}</td>
<td>0.0383</td>
<td>0.0532</td>
<td>0.0376</td>
<td>0.0570</td>
<td>0.0931</td>
<td>0.0800</td>
<td>0.0848</td>
<td>0.1601</td>
</tr>
</tbody>
</table>

(continues on next page)
$A_{8}$ 0.0743 0.1210 0.4544 0.0388
$A_{9}$ 0.0680 0.0247 0.0000 0.0000
$A_{10}$ 0.1419 0.1403 0.0911 0.0097
$A_{11}$ 0.0243 0.0794 0.1010 0.0416
$A_{12}$ 0.1131 0.1181 0.0814 0.0833

Rank acceptability indexes displayed in the form of stacked bar chart.

matplotlib.rcdefaults()
plot_barplot(acc_in_df, 'Alternative', 'Rank acceptability index', 'Rank')

The PostScript backend does not support transparency; partially transparent artists will be rendered opaque.

Rank acceptability indexes displayed in the form of heatmap

draw_heatmap_smaa(acc_in_df, 'Rank acceptability indexes')
Central weight vector

The central weight vector describes the preferences of a typical decision-maker, supporting this alternative with the assumed preference model. It allows the decision-maker to see what criteria preferences result in the best evaluation of given alternatives. Rows containing only zeroes mean that a given alternative never becomes a leader.

```python
[27]: central_weights_df = pd.DataFrame(central_weight_vector, index = list_alt_names, columns = cols)
central_weights_df.to_csv('results_smaa/cw.csv')
```

```python
[28]:
<table>
<thead>
<tr>
<th></th>
<th>$C_{1}$</th>
<th>$C_{2}$</th>
<th>$C_{3}$</th>
<th>$C_{4}$</th>
<th>$C_{5}$</th>
<th>$C_{6}$</th>
<th>$C_{7}$</th>
<th>$C_{8}$</th>
<th>$C_{9}$</th>
<th>$C_{10}$</th>
<th>$C_{11}$</th>
<th>$C_{12}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$A_{1}$</td>
<td>0.080044</td>
<td>0.065913</td>
<td>0.166905</td>
<td>0.126321</td>
<td>0.122206</td>
<td>0.077438</td>
<td>0.071596</td>
<td>0.080231</td>
<td>0.056639</td>
<td>0.054395</td>
<td>0.098312</td>
<td></td>
</tr>
<tr>
<td>$A_{2}$</td>
<td>0.117195</td>
<td>0.089724</td>
<td>0.076283</td>
<td>0.128039</td>
<td>0.056262</td>
<td>0.080681</td>
<td>0.078959</td>
<td>0.081181</td>
<td>0.071432</td>
<td>0.110424</td>
<td>0.109820</td>
<td></td>
</tr>
<tr>
<td>$A_{3}$</td>
<td>0.003336</td>
<td>0.023275</td>
<td>0.030771</td>
<td>0.102729</td>
<td>0.283001</td>
<td>0.043135</td>
<td>0.090514</td>
<td>0.078441</td>
<td>0.080131</td>
<td>0.214907</td>
<td>0.127288</td>
<td></td>
</tr>
<tr>
<td>$A_{4}$</td>
<td>0.044721</td>
<td>0.084670</td>
<td>0.065501</td>
<td>0.020685</td>
<td>0.040810</td>
<td>0.035974</td>
<td>0.048625</td>
<td>0.025734</td>
<td>0.096084</td>
<td>0.077371</td>
<td>0.000000</td>
<td></td>
</tr>
<tr>
<td>$A_{5}$</td>
<td>0.054954</td>
<td>0.277008</td>
<td>0.068419</td>
<td>0.020685</td>
<td>0.040810</td>
<td>0.035974</td>
<td>0.048625</td>
<td>0.025734</td>
<td>0.096084</td>
<td>0.077371</td>
<td>0.000000</td>
<td></td>
</tr>
<tr>
<td>$A_{6}$</td>
<td>0.000000</td>
<td>0.000000</td>
<td>0.000000</td>
<td>0.000000</td>
<td>0.000000</td>
<td>0.000000</td>
<td>0.000000</td>
<td>0.000000</td>
<td>0.000000</td>
<td>0.000000</td>
<td>0.000000</td>
<td></td>
</tr>
<tr>
<td>$A_{7}$</td>
<td>0.000000</td>
<td>0.000000</td>
<td>0.000000</td>
<td>0.000000</td>
<td>0.000000</td>
<td>0.000000</td>
<td>0.000000</td>
<td>0.000000</td>
<td>0.000000</td>
<td>0.000000</td>
<td>0.000000</td>
<td></td>
</tr>
<tr>
<td>$A_{8}$</td>
<td>0.000000</td>
<td>0.000000</td>
<td>0.000000</td>
<td>0.000000</td>
<td>0.000000</td>
<td>0.000000</td>
<td>0.000000</td>
<td>0.000000</td>
<td>0.000000</td>
<td>0.000000</td>
<td>0.000000</td>
<td></td>
</tr>
<tr>
<td>$A_{9}$</td>
<td>0.000000</td>
<td>0.000000</td>
<td>0.000000</td>
<td>0.000000</td>
<td>0.000000</td>
<td>0.000000</td>
<td>0.000000</td>
<td>0.000000</td>
<td>0.000000</td>
<td>0.000000</td>
<td>0.000000</td>
<td></td>
</tr>
<tr>
<td>$A_{10}$</td>
<td>0.000000</td>
<td>0.000000</td>
<td>0.000000</td>
<td>0.000000</td>
<td>0.000000</td>
<td>0.000000</td>
<td>0.000000</td>
<td>0.000000</td>
<td>0.000000</td>
<td>0.000000</td>
<td>0.000000</td>
<td></td>
</tr>
<tr>
<td>$A_{11}$</td>
<td>0.000000</td>
<td>0.000000</td>
<td>0.000000</td>
<td>0.000000</td>
<td>0.000000</td>
<td>0.000000</td>
<td>0.000000</td>
<td>0.000000</td>
<td>0.000000</td>
<td>0.000000</td>
<td>0.000000</td>
<td></td>
</tr>
<tr>
<td>$A_{12}$</td>
<td>0.000000</td>
<td>0.000000</td>
<td>0.000000</td>
<td>0.000000</td>
<td>0.000000</td>
<td>0.000000</td>
<td>0.000000</td>
<td>0.000000</td>
<td>0.000000</td>
<td>0.000000</td>
<td>0.000000</td>
<td></td>
</tr>
</tbody>
</table>

Rank scores

```python
[29]: rank_scores_df = pd.DataFrame(rank_scores, index = list_alt_names, columns = ['Rank'])
rank_scores_df.to_csv('results_smaa/fr.csv')
```

```python
[30]:
<table>
<thead>
<tr>
<th>Rank</th>
</tr>
</thead>
<tbody>
<tr>
<td>$A_{1}$</td>
</tr>
<tr>
<td>$A_{2}$</td>
</tr>
<tr>
<td>$A_{3}$</td>
</tr>
<tr>
<td>$A_{4}$</td>
</tr>
</tbody>
</table>
```

(continues on next page)
1.5 API Reference

This page contains auto-generated API reference documentation.

1.5.1 pyrepo_mcda

Subpackages

pyrepo_mcda.mcda_methods

Submodules

pyrepo_mcda.mcda_methods.ahp

Module Contents

Classes

AHP

class pyrepo_mcda.mcda_methods.ahp.AHP(normalization_method=minmax_normalization)

Bases: pyrepo_mcda.mcda_methods.mcda_method.MCDA_method

__call__(matrix, weights, types)

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

Parameters

- matrix (ndarray) – Decision matrix with numerical performance values of alternatives. Decision matrix includes m alternatives in rows and n criteria in columns.
- weights (ndarray) – Vector with criteria weights given in numerical values. The sum of weights must be equal to 1.
- types (ndarray) – Criteria types. Profit criteria are represented by 1 and cost by -1.

1 Created with sphinx-autoapi
• Results

• --------

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

Examples

```python
>>> ah = AHP()
>>> pref = ah(matrix, weights, types)
>>> rank = rank_preferences(pref, reverse = True)
```

_check_consistency(X)
Consistency Check on the Pairwise Comparison Matrix of the Criteria or alternatives

Parameters

- X (ndarray) – matrix of pairwise comparisons

Examples

```python
>>> PCcriteria = np.array([[1, 1, 5, 3], [1, 1, 5, 3], [1/5, 1/5, 1, 1/3], [1/3, 1/3, 1, 1/3]])
>>> ah = AHP()
>>> ah._check_consistency(PCcriteria)
```

_calculate_eigenvector(X)
Compute the Priority Vector of Criteria (weights) or alternatives using Eigenvector method

Parameters

- X (ndarray) – matrix of pairwise comparisons

Returns

- Eigenvector

Return type

- ndarray

Examples

```python
>>> PCM1 = np.array([[1, 5, 1, 1, 1/3, 3],
                   [1/5, 1, 1/3, 1/5, 1/7, 1],
                   [1, 3, 1, 1/3, 1/5, 1],
                   [1, 5, 3, 1, 1/3, 3],
                   [3, 7, 5, 3, 1, 7],
                   [1/3, 1, 1, 1/3, 1/7, 1]])
>>> ah = AHP()
>>> S = ah._calculate_eigenvector(PCM1)
```

_normalized_column_sum(X)
Compute the Priority Vector of Criteria (weights) or alternatives using The normalized column sum method
**Parameters**

- **X (ndarray)** – matrix of pairwise comparisons

**Returns**

- Vector with weights calculated with the normalized column sum method

**Return type**

- ndarray

**Examples**

```python
>>> PCM1 = np.array([[1, 5, 1, 1, 1/3, 3],
                   [1/5, 1, 1/3, 1/5, 1/7, 1],
                   [1, 3, 1, 1/3, 1/5, 1],
                   [1, 5, 3, 1, 1/3, 3],
                   [3, 7, 5, 3, 1, 7],
                   [1/3, 1, 1, 1/3, 1/7, 1]])
>>> ahp = AHP()
>>> S = ahp._normalized_column_sum(PCM1)
```

**_geometric_mean(X)**

Compute the Priority Vector of Criteria (weights) or alternatives using The geometric mean method

**Parameters**

- **X (ndarray)** – matrix of pairwise comparisons

**Returns**

- Vector with weights calculated with the geometric mean method

**Return type**

- ndarray

**Examples**

```python
>>> PCM1 = np.array([[1, 5, 1, 1, 1/3, 3],
                   [1/5, 1, 1/3, 1/5, 1/7, 1],
                   [1, 3, 1, 1/3, 1/5, 1],
                   [1, 5, 3, 1, 1/3, 3],
                   [3, 7, 5, 3, 1, 7],
                   [1/3, 1, 1, 1/3, 1/7, 1]])
>>> ahp = AHP()
>>> S = ahp._geometric_mean(PCM1)
```

**_classic_ahp(alt_matrices, weights, calculate_priority_vector_method=None)**

Calculate the global alternative priorities. This is a method for classic AHP where you provide matrices with values of pairwise comparisons of alternatives and weights in the form of a priority vector.

**Parameters**

- **alt_matrices (list)** – list with matrices including values of pairwise comparisons of alternatives
- **weights (ndarray)** – priority vector of criteria (weights)
- **calculate_priority_vector_method (function)** – Method for calculation of the priority vector. It can be chosen from three available methods: _calculate_eigenvector, _calculate_eigenvector,
_normalized_column_sum and _geometric_mean if the user does not provide calculate_priority_vector_method, it is automatically set as the default _calculate_eigenvector

**Returns**

vector with the global alternative priorities

**Return type**

ndarray

**Examples**

```python
>>> PCcriteria = np.array([[1, 1, 5, 3], [1, 1, 5, 3], [1, 3, 1, 1, 3, 1]])
>>> PCM1 = np.array([[1, 5, 1, 1, 1/3, 3], [1/5, 1, 1/3, 1/5, 1/7, 1], [1, 3, 1, 1/3, 1/5, 1], [1, 5/3, 1, 1/3, 3], [3, 7, 5, 3, 1, 7], [1/3, 1, 1, 1/3, 1/7, 1]])
>>> PCM2 = np.array([[1, 7, 3, 1/3, 1/3, 1/3], [1/7, 1, 1/3, 1/7, 1/9, 1/7], [1/3, 3, 1, 1/5, 1/5, 1/5], [3, 7, 5, 1, 1, 1], [3, 9, 5, 1, 1, 1], [3, 7, 5, 1, 1, 1]])
>>> PCM3 = np.array([[1, 1/9, 1/7, 1/9, 1, 1/5], [9, 1, 1, 1, 5, 3], [7, 1, 1, 1, 5, 1], [9, 1, 1, 1, 7, 3], [1, 1/5, 1/5, 1/7, 1, 1/3], [5, 1/3, 1, 1/3, 3, 1]])
>>> PCM4 = np.array([[1, 1/5, 1/5, 1/3, 1/7, 1/5], [5, 1, 1, 3, 1/3, 1], [5, 1, 1, 1, 1/3, 1], [3, 1/3, 1, 1, 1/7, 1], [7, 3, 3, 7, 1, 5], [5, 1, 1, 1, 1/5, 1]])
```

```python
>>> ahp = AHP()
>>> ahp._check_consistency(PCcriteria)
>>> weights = ahp._calculate_eigenvector(PCcriteria)
>>> alt_matrices = []
>>> alt_matrices.append(PCM1)
>>> alt_matrices.append(PCM2)
>>> alt_matrices.append(PCM3)
>>> alt_matrices.append(PCM4)
>>> calculate_priority_vector_method = ahp._calculate_eigenvector
>>> pref = ahp._classic_ahp(alt_matrices, weights, calculate_priority_vector_method)
>>> rank = rank_preferences(pref, reverse = True)
```
Score alternatives provided in decision matrix matrix using criteria weights and criteria types.

Parameters

- **matrix** (*ndarray*) – Decision matrix with numerical performance values of alternatives. The decision matrix includes m alternatives in rows and n criteria in columns.
- **weights** (*ndarray*) – Vector with criteria weights given in numerical values. The sum of weights must be equal to 1.
- **types** (*ndarray*) – Criteria types. Profit criteria are represented by 1 and cost by -1.

Results –

- ndarray

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

Examples

```python
>>> ahp = AHP()
>>> pref = ahp(matrix, weights, types)
>>> rank = rank_preferences(pref, reverse = True)
```

### pyrepo_mcda.mcda_methods aras

#### Module Contents

#### Classes

**ARAS**

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

```python
class pyrepo_mcda.mcda_methods aras.ARAS(normalization_method=sum_normalization):
    Bases: pyrepo_mcda.mcda_methods.mcda_method.MCDA_method
    Helper class that provides a standard way to create an ABC using inheritance.
    __call__(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*) – 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
>>> aras = ARAS()
>>> pref = aras(matrix, weights, types)
>>> rank = rank_preferences(pref, reverse = True)
```

```python
static _aras(matrix, weights, types, normalization_method)
```

```python
pyrepo_mcda.mcda_methods.codas
```

Module Contents

Classes

```python
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.

```python
__call__(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
static _codas(self, matrix, weights, types, normalization_method, distance_metric)
```

1.5. API Reference 91
pyrepo-mcda, Release 0.1

pyrepo_mcda.mcda_methods.copras

Module Contents

Classes

class pyrepo_mcda.mcda_methods.copras.COPRAS(normalization_method=sum_normalization)
    Bases: pyrepo_mcda.mcda_methods.mcda_method.MCDA_method
    Helper class that provides a standard way to create an ABC using inheritance.
    __call__(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) – 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

>>> copras = COPRAS(normalization_method = sum_normalization)
>>> pref = copras(matrix, weights, types)
>>> rank = rank_preferences(pref, reverse = True)

static _copras(matrix, weights, types, normalization_method)

pyrepo_mcda.mcda_methods.cradis

Module Contents

Classes

class pyrepo_mcda.mcda_methods.cradis.CRADIS(normalization_method=linear_normalization)
    Bases: pyrepo_mcda.mcda_methods.mcda_method.MCDA_method
__call__(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) – 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
>>> cradis = CRADIS()
>>> pref = cradis(matrix, weights, types)
>>> rank = rank_preferences(pref, reverse = True)

static _cradis(matrix, weights, types, normalization_method)

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__(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
_module_ (weights, types)
```

**pyrepo_mcda.mcda_methods.mabac**

**Module Contents**

**Classes**

```python
class MABAC
```

```python
__call__ (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 (weights, types, normalization_method)
```
```python
class pyrepo_mcda.mcda_methods.marcos.MARCOS
    Bases: pyrepo_mcda.mcda_methods.mcda_method.MCDA_method

    __call__(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) – 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
>>> marcos = MARCOS()
>>> pref = marcos(matrix, weights, types)
>>> rank = rank_preferences(pref, reverse = True)
```

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

```python
pyrepo_mcda.mcda_methods.mcda_method

Module Contents

Classes
```

```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__(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) – matrix with criteria weights vectors with number of columns 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

static _verify_input_data(matrix, weights, types)

pyrepo_mcda.mcda_methods.multimoora

Module Contents

Classes

MULTIMOORA_RS
MULTIMOORA_RP
MULTIMOORA_FMF
MULTIMOORA

class pyrepo_mcda.mcda_methods.multimoora.MULTIMOORA_RS
Bases: pyrepo_mcda.mcda_methods.mcda_method.MCDA_method
__call__(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_rs = MULTIMOORA_RS()
>>> pref = multimoora_rs(matrix, weights, types)
>>> rank = rank_preferences(pref, reverse = True)
```

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

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

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

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

class pyrepo_mcda.mcda_methods.multimoora.MULTIMOORA_FMF
Bases: pyrepo_mcda.mcda_methods.mcda_method.MCDA_method
__call__(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)
```

```python
class pyrepo_mcda.mcda_methods.multimoora.MULTIMOORA(compromise_rank_method=dominance_directed_graph)
Bases: pyrepo_mcda.mcda_methods.mcda_method.MCDA_method
__call__(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)
```

```python
Multimoora(matrix, weights, compromise_rank_method)
```

```python
pyrepo_mcda.mcda_methods.promethee

Module Contents

Classes

PROMETHEE_II

class pyrepo_mcda.mcda_methods.promethee.PROMETHEE_II
Bases: pyrepo_mcda.mcda_methods.mcda_method.MCDA_method
__call__(matrix, weights, types, preference_functions=None, p=None, q=None)
_usual_function(d, p, q)
_ushape_function(d, p, q)
```
_vshape_function(d, p, q)
_level_function(d, p, q)
_linear_function(d, p, q)
_gaussian_function(d, p, q)

def __promethee_II(self, matrix, weights, types, preference_functions, p, q):
    Score alternatives provided in the 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) – Criteria weights. The sum of weights must be equal to 1.
    • `types` (ndarray) – Criteria types. Profit criteria are represented by 1 and cost by -1.
    • `preference_functions` (list) – List with methods containing preference functions for calculating the preference degree for each criterion.
    • `p` (ndarray) – Vector with values representing the threshold of absolute preference.
    • `q` (ndarray) – Vector with values representing the threshold of indifference.

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

    Return type
    ndarray

    Examples

    >>> promethee_II = PROMETHEE_II()
    >>> preference_functions = [promethee_II._linear_function for pf in range(len(weights))]
    >>> u = np.sqrt(np.sum(np.square(np.mean(matrix, axis = 0) - matrix), axis = 0) / matrix.shape[0])
    >>> p = 2 * u
    >>> q = 0.5 * u
    >>> pref = promethee_II(matrix, weights, types, preference_functions, p = p, q = q)
    >>> rank = rank_preferences(pref, reverse = True)
class pyrepo_mcda.mcda_methods.prosa_c.PROSA_C

Bases: pyrepo_mcda.mcda_methods.promethee.PROMETHEE_II

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

__call__(matrix, weights, types, preference_functions=None, p=None, q=None, s=None)

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) – matrix with criteria weights vectors with number of columns 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.

static _prosa_c(self, matrix, weights, types, preference_functions, p, q, s)

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

Parameters

- **matrix** (ndarray) – Decision matrix with m alternatives in rows and n criteria in columns.
- **weights** (ndarray) – Criteria weights. The sum of weights must be equal to 1.
- **types** (ndarray) – Criteria types. Profit criteria are represented by 1 and cost by -1.
- **preference_functions** (list) – List with methods containing preference functions for calculating the preference degree for each criterion.
- **p** (ndarray) – Vector with values representing the threshold of absolute preference.
- **q** (ndarray) – Vector with values representing the threshold of indifference.
- **s** (ndarray) – Vector with values of the coefficient sj for the criteria.

Returns

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

Return type

ndarray

Examples

```python
>>> prosa_c = PROSA_C()
>>> preference_functions = [prosa_c._linear_function for pf in
˓� range(len(weights))]
>>> u = np.sqrt(np.sum(np.square(np.mean(matrix, axis = 0) - matrix), axis = 0) / matrix.shape[0])
>>> p = 2 * u
>>> q = 0.5 * u
>>> s = np.repeat(0.3, len(weights))
>>> pref = promethee_II(matrix, weights, types, preference_functions, p = p, q = q, s = s)
>>> rank = rank_preferences(pref, reverse = True)
```
```python
class pyrepo_mcda.mcda_methods.saw.SAW(normalization_method=linear_normalization):
    Bases: pyrepo_mcda.mcda_methods.mcda_method.MCDA_method

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

    __call__(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

>>> saw = SAW(normalization_method = minmax_normalization)
>>> pref = saw(matrix, weights, types)
>>> rank = rank_preferences(pref, reverse = True)
```

```python
static _saw(matrix, weights, types, normalization_method)
```
class pyrepo_mcda.mcda_methods.spotis.SPOTIS
    Bases: pyrepo_mcda.mcda_methods.mcda_method.MCDA_method

    __call__(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

>>> 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)

pyrepo_mcda.mcda_methods.topsis

Module Contents

Classes

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

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.
Score alternatives provided in decision matrix \texttt{matrix} with \( m \) alternatives in rows and \( n \) criteria in columns using criteria \texttt{weights} and criteria \texttt{types}.

**Parameters**

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

\texttt{ndarray}

**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)**
Return type
ndarray

Examples

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

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

pyrepo_mcda.mcda_methods.vikor_smaa

Module Contents

Classes

```python
VIKOR_SMAA
```

class pyrepo_mcda.mcda_methods.vikor_smaa.VIKOR_SMAA(normalization_method=None, v=0.5)

__call__(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)* – Matrix with i vectors in rows of n weights in columns. i means number of iterations of SMAA
- **types** *(ndarray)* – Vector with criteria types. Profit criteria are represented by 1 and cost by -1.

Returns
Matrix with acceptability indexes values for each alternative in rows in relation to each rank in columns, Matrix with central weight vectors for each alternative in rows Matrix with final ranking of alternatives

Return type
ndrarray, ndrarray, ndrarray
Examples

```python
>>> vikor_smaa = VIKOR_SMAA(normalization_method = minmax_normalization)
>>> rank_acceptability_index, central_weight_vector, rank_scores = vikor_smaa(matrix, weights, types)
```

**_generate_weights(n, iterations)_**

Function to generate multiple weight vectors

**Parameters**

- `n` *(int)* – Number of criteria
- `iterations` *(int)* – Number of weight vector to generate

**Returns**

Matrix containing in rows vectors with weights for n criteria

**Return type**

`ndarray`

```python
def _vikor_smaa(self, 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__(matrix, weights, types)**

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

- **AHP**
- **ARAS**
  Helper class that provides a standard way to create an ABC using
- **CODAS**
  Helper class that provides a standard way to create an ABC using
- **COPRAS**
  Helper class that provides a standard way to create an ABC using
- **CRADIS**
- **EDAS**
  Helper class that provides a standard way to create an ABC using
- **MABAC**
- **MARCOS**
- **MULTIMOORA**
- **MULTIMOORA_RS**
- **PROMETHEE_II**
- **PROSA_C**
  Helper class that provides a standard way to create an ABC using
- **SAW**
  Helper class that provides a standard way to create an ABC using
- **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
- **VIKOR_SMAA**
class pyrepo_mcda.mcda_methods.AHP(normalization_method=minmax_normalization)
    Bases: pyrepo_mcda.mcda_methods.mcda_method.MCDA_method

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

    Parameters
    • matrix (ndarray) – Decision matrix with numerical performance values of alternatives.
      Decision matrix includes m alternatives in rows and n criteria in columns.
    • weights (ndarray) – Vector with criteria weights given in numerical values. The sum of
      weights must be equal to 1.
    • types (ndarray) – Criteria types. Profit criteria are represented by 1 and cost by -1.
    • Results –
    •  
      ndarray
      Preference values of each alternative. The best alternative has the highest preference
      value.

    Examples

    >>> ahp = AHP()
    >>> pref = ahp(matrix, weights, types)
    >>> rank = rank_preferences(pref, reverse = True)

    _check_consistency(X)
    Consistency Check on the Pairwise Comparison Matrix of the Criteria or alternatives

    Parameters
    • X (ndarray) – matrix of pairwise comparisons

    Examples

    >>> PCcriteria = np.array([[1, 1, 5, 3], [1, 1, 5, 3], [1/5, 1/5, 1, 1/3], [1/3, 
      → 1/3, 3, 1]])
    >>> ahp = AHP()
    >>> ahp._check_consistency(PCcriteria)

    _calculate_eigenvector(X)
    Compute the Priority Vector of Criteria (weights) or alternatives using Eigenvector method

    Parameters
    • X (ndarray) – matrix of pairwise comparisons

    Returns
    Eigenvector

    Return type
    ndarray
Examples

```python
>>> PCM1 = np.array([[1, 5, 1, 1, 1/3, 3],
                   [1/5, 1, 1/3, 1/5, 1/7, 1],
                   [1, 3, 1, 1/3, 1/5, 1],
                   [1, 5, 3, 1, 1/3, 3],
                   [3, 7, 5, 3, 1, 7],
                   [1/3, 1, 1, 1/3, 1/7, 1]])
>>> ahp = AHP()
>>> S = ahp._calculate_eigenvector(PCM1)

_normalized_column_sum(X)
Compute the Priority Vector of Criteria (weights) or alternatives using The normalized column sum method

Parameters
X (ndarray) – matrix of pairwise comparisons

Returns
Vector with weights calculated with The normalized column sum method

Return type
ndarray

Examples

```python
>>> PCM1 = np.array([[1, 5, 1, 1, 1/3, 3],
                   [1/5, 1, 1/3, 1/5, 1/7, 1],
                   [1, 3, 1, 1/3, 1/5, 1],
                   [1, 5, 3, 1, 1/3, 3],
                   [3, 7, 5, 3, 1, 7],
                   [1/3, 1, 1, 1/3, 1/7, 1]])
>>> ahp = AHP()
>>> S = ahp._normalized_column_sum(PCM1)

_geometric_mean(X)
Compute the Priority Vector of Criteria (weights) or alternatives using The geometric mean method

Parameters
X (ndarray) – matrix of pairwise comparisons

Returns
Vector with weights calculated with The geometric mean method

Return type
ndarray
Examples

```python
>>> PCM1 = np.array([[1, 5, 1, 1/3, 3],
[1/5, 1, 1/3, 1/5, 1/7, 1],
[1, 3, 1, 1/3, 1/5, 1],
[1, 5, 3, 1, 1/3, 3],
[3, 7, 5, 3, 1, 7],
[1/3, 1, 1, 1/3, 1/7, 1]])
>>> ahp = AHP()
>>> S = ahp._geometric_mean(PCM1)

_classic_ahp(alt_matrices, weights, calculate_priority_vector_method=None)

Calculate the global alternative priorities. This is a method for classic AHP where you provide matrices
with values of pairwise comparisons of alternatives and weights in the form of a priority vector.

Parameters

- **alt_matrices** (list) – list with matrices including values of pairwise comparisons of alternatives
- **weights** (ndarray) – priority vector of criteria (weights)
- **calculate_priority_vector_method** (function) – Method for calculation of the priority vector. It can be chosen from three available methods: _calculate_eigenvector, _normalized_column_sum and _geometric_mean if the user does not provide calculate_priority_vector_method, it is automatically set as the default _calculate_eigenvector

Returns

vector with the global alternative priorities

Return type

ndarray

Examples

```python
>>> PCcriteria = np.array([[1, 1, 5, 3], [1, 1, 5, 3],
[1/5, 1/5, 1, 1/3], [1/3, 1/3, 3, 1]])
>>> PCM1 = np.array([[1, 5, 1, 1/3, 3],
[1/5, 1, 1/3, 1/5, 1/7, 1],
[1, 3, 1, 1/3, 1/5, 1],
[1, 5, 3, 1, 1/3, 3],
[3, 7, 5, 3, 1, 7],
[1/3, 1, 1, 1/3, 1/7, 1]])
>>> PCM2 = np.array([[1, 7, 3, 1/3, 1/3, 1/3],
[1/7, 1, 1/3, 1/7, 1/9, 1/7],
[1/3, 3, 1, 1/5, 1/5, 1/5],
[3, 7, 5, 1, 1, 1],
[3, 9, 5, 1, 1, 1],
[3, 7, 5, 1, 1, 1]])
>>> PCM3 = np.array([[1, 1/9, 1/7, 1/9, 1, 1/5],
[9, 1, 1, 1, 5, 3],
[7, 1, 1, 1, 5, 1],
[9, 1, 1, 1, 7, 3],
[1, 1/5, 1/5, 1/7, 1, 1/3],
[5, 1/3, 1, 1/3, 3, 1]])
```
>>> PCM4 = np.array([[1, 1/5, 1/5, 1/3, 1/7, 1/5],
[5, 1, 1, 3, 1/3, 1],
[5, 1, 1, 1, 1/3, 1],
[3, 1/3, 1, 1, 1/7, 1],
[7, 3, 3, 7, 1, 5],
[5, 1, 1, 1, 1/5, 1]])

>>> ahp = AHP()
>>> ahp._check_consistency(PCcriteria)
>>> weights = ahp._calculate_eigenvector(PCcriteria)
>>> alt_matrices = []
>>> alt_matrices.append(PCM1)
>>> alt_matrices.append(PCM2)
>>> alt_matrices.append(PCM3)
>>> alt_matrices.append(PCM4)

>>> calculate_priority_vector_method = ahp._calculate_eigenvector
>>> pref = ahp._classic_ahp(alt_matrices, weights, calculate_priority_vector_method)
>>> rank = rank_preferences(pref, reverse = True)

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

Parameters

- **matrix** (ndarray) – Decision matrix with numerical performance values of alternatives. The decision matrix includes m alternatives in rows and n criteria in columns.
- **weights** (ndarray) – Vector with criteria weights given in numerical values. The sum of weights must be equal to 1.
- **types** (ndarray) – Criteria types. Profit criteria are represented by 1 and cost by -1.
- **Results**
- " "
- ndarray
Preference values of each alternative. The best alternative has the highest preference value.

Examples

>>> ahp = AHP()
>>> pref = ahp(matrix, weights, types)
>>> rank = rank_preferences(pref, reverse = True)

class pyrepo_mcda.mcda_methods.ARAS(normalization_method=sum_normalization)
Bases: pyrepo_mcda.mcda_methods.mcda_method.MCDA_method
Helper class that provides a standard way to create an ABC using inheritance.

__call__(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*) – 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
globals()['aras'] = ARAS()
globals()['pref'] = aras(matrix, weights, types)
globals()['rank'] = rank_preferences(pref, reverse = True)
```

```
static _aras(matrix, weights, types, normalization_method)

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.

```
__call__(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
globals()['codas'] = CODAS(normalization_method = linear_normalization, distance_metric=euclidean, tau = 0.02)
globals()['pref'] = codas(matrix, weights, types)
globals()['rank'] = rank_preferences(pref, reverse = True)
```
```python
class pyrepo_mcda.mcda_methods.COPRAS(normalization_method=sum_normalization):
    Bases: pyrepo_mcda.mcda_methods.mcda_method.MCDA_method
    Helper class that provides a standard way to create an ABC using inheritance.

    __call__(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) – 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
```

```python
class pyrepo_mcda.mcda_methods.CRADIS(normalization_method=linear_normalization):
    Bases: pyrepo_mcda.mcda_methods.mcda_method.MCDA_method

    __call__(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) – 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
>>> cradis = CRADIS()
>>> pref = cradis(matrix, weights, types)
>>> rank = rank_preferences(pref, reverse = True)
```

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

_mabac(weights, types, normalization_method)

class pyrepo_mcda.mcda_methods.MAROCSBases: pyrepo_mcda.mcda_methods.mcda_method.MCDA_method__call__(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*) – 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
>>> marcos = MARCOS()
>>> pref = marcos(matrix, weights, types)
>>> rank = rank_preferences(pref, reverse = True)
```

_static__marcos(matrix, weights, types)

class pyrepo_mcda.mcda_methods.MULTIMOORACompromiseRankMethod = dominance_directed_graph

Bases: pyrepo_mcda.mcda_methods.mcda_method.MCDA_method__call__(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)
```

_class_ `pyrepo_mcda.mcda_methods.MULTIMOORA_RS`

Bases: `pyrepo_mcda.mcda_methods.mcda_method.MCDA_method`

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

_class_ `pyrepo_mcda.mcda_methods.PROMETHEE_II`

Bases: `pyrepo_mcda.mcda_methods.mcda_method.MCDA_method`

__call__(matrix, weights, types, preference_functions=None, p=None, q=None)

_usual_function(d, p, q)

_ushape_function(d, p, q)

_vshape_function(d, p, q)

_level_function(d, p, q)

_linear_function(d, p, q)

_gaussian_function(d, p, q)

static _promethee_II(self, matrix, weights, types, preference_functions, p, q)

Score alternatives provided in the 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*) – Criteria weights. The sum of weights must be equal to 1.

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

• **preference_functions** (*list*) – List with methods containing preference functions for calculating the preference degree for each criterion.

• **p** (*ndarray*) – Vector with values representing the threshold of absolute preference.

• **q** (*ndarray*) – Vector with values representing the threshold of indifference.

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

**Return type**
*ndarray*

**Examples**

```python
>>> promethee_II = PROMETHEE_II()
>>> preference_functions = [promethee_II._linear_function for pf in
range(len(weights))]
>>> u = np.sqrt(np.sum(np.square(np.mean(matrix, axis = 0) - matrix), axis = 0, / matrix.shape[0]))
>>> p = 2 * u
>>> q = 0.5 * u
>>> pref = promethee_II(matrix, weights, types, preference_functions, p = p, q = q)
>>> rank = rank_preferences(pref, reverse = True)
```

---

**class** pyrepo_mcda.mcda_methods.PROSA_C

**Bases:** pyrepo_mcda.mcda_methods.promethee.PROMETHEE_II

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

**__call__** (*matrix*, *weights*, *types*, *preference_functions=None*, *p=None*, *q=None*, *s=None*)

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*) – Matrix with criteria weights vectors with number of columns 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*.

**static _prosa_c** (*self*, *matrix*, *weights*, *types*, *preference_functions*, *p*, *q*, *s*)

Score alternatives provided in the 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*) – Criteria weights. The sum of weights must be equal to 1.

• **types** (*ndarray*) – Criteria types. Profit criteria are represented by 1 and cost by -1.
• **preference_functions** (*list*) – List with methods containing preference functions for calculating the preference degree for each criterion.

• **p** (*ndarray*) – Vector with values representing the threshold of absolute preference.

• **q** (*ndarray*) – Vector with values representing the threshold of indifference.

• **s** (*ndarray*) – Vector with values of the coefficient \( s_j \) for the criteria

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

**Return type**
*ndarray*

**Examples**

```python
>>> prosa_c = PROSA_C()
>>> preference_functions = [prosa_c._linear_function for pf in

˓→range(len(weights))]
>>> u = np.sqrt(np.sum(np.square(np.mean(matrix, axis = 0) - matrix), axis = 0) / matrix.shape[0])
>>> p = 2 * u
>>> q = 0.5 * u
>>> s = np.repeat(0.3, len(weights))
>>> pref = promethee_II(matrix, weights, types, preference_functions, p = p, q

˓→= q, s = s)
>>> rank = rank_preferences(pref, reverse = True)
```

**class** pyrepo_mcda.mcda_methods.SAW(*normalization_method=linear_normalization*)

Bases: pyrepo_mcda.mcda_methods.mcda_method.MCDA_method

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

**__call__**(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
>>> saw = SAW(normalization_method = minmax_normalization)
>>> pref = saw(matrix, weights, types)
>>> rank = rank_preferences(pref, reverse = True)
```

```python
static _saw(matrix, weights, types, normalization_method)
```

class pyrepo_mcda.mcda_methods.SPOTIS
    Bases: pyrepo_mcda.mcda_methods.mcda_method.MCDA_method

    __call__(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)
```

```python
static _spotis(matrix, weights, types, bounds)
```

class pyrepo_mcda.mcda_methods.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__(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)
```

**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__(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**

```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__(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
>>> 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)
class pyrepo_mdca.mcda_methods.VIKOR_SMAA(normalization_method=None, v=0.5)
__call__(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) – Matrix with i vectors in rows of n weights in columns. i means number of iterations of SMAA
- types (ndarray) – Vector with criteria types. Profit criteria are represented by 1 and cost by -1.

Returns
Matrix with acceptability indexes values for each alternative in rows in relation to each rank in columns. Matrix with central weight vectors for each alternative in rows Matrix with final ranking of alternatives

Return type
ndarray, ndarray, ndarray
Examples

```python
>>> vikor_smaa = VIKOR_SMAA(normalization_method = minmax_normalization)
>>> rank_acceptability_index, central_weight_vector, rank_scores = vikor_smaa(matrix, weights, types)
```

`_generate_weights(n, iterations)`

Function to generate multiple weight vectors

Parameters

- `n (int)` – Number of criteria
- `iterations (int)` – Number of weight vector to generate

Returns

Matrix containing in rows vectors with weights for `n` criteria

Return type

`ndarray`

```python
static _vikor_smaa(self, matrix, weights, types, normalization_method, v)
```

Submodules

`pyrepo_mcda.additions`

Module Contents

Functions

```python
rank_preferences(pref[, reverse])
```

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

```python
pyrepo_mcda.correlations.spearman(R, Q)
```

Calculate Spearman rank correlation coefficient between two vectors

**Parameters**

- `R (ndarray)` – First vector containing values
- `Q (ndarray)` – Second vector containing values

**Returns**

Value of correlation coefficient between two vectors

**Return type**

`float`

**Examples**

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

```python
pyrepo_mcda.correlations.weighted_spearman(R, Q)
```

Calculate Weighted Spearman rank correlation coefficient between two vectors

**Parameters**

- `R (ndarray)` – First vector containing values
- `Q (ndarray)` – Second vector containing values

**Returns**

Value of correlation coefficient between two vectors

**Return type**

`float`

**Examples**

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

```python
pyrepo_mcda.correlations.pearson_coeff(R, Q)
```

Calculate Pearson correlation coefficient between two vectors

**Parameters**

- `R (ndarray)` – First vector containing values
• Q (ndarray) – Second vector containing values

**Returns**
Value of correlation coefficient between two vectors

**Return type**
float

**Examples**

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

### pyrepo_mcda.correlations.WS_coeff(R, Q)

Calculate Rank similarity coefficient between two vectors

**Parameters**

• R (ndarray) – First vector containing values
• Q (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

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>euclidean(A, B)</code></td>
<td>Calculate Euclidean distance between two vectors A and B.</td>
</tr>
<tr>
<td><code>manhattan(A, B)</code></td>
<td>Calculate Manhattan (Taxicab) distance between two vectors A and B.</td>
</tr>
<tr>
<td><code>hausdorff_distance(A, B)</code></td>
<td>Calculate Hausdorff distance between two vectors A and B.</td>
</tr>
<tr>
<td><code>correlation(A, B)</code></td>
<td>Calculate Correlation distance between two vectors A and B.</td>
</tr>
<tr>
<td><code>chebyshev(A, B)</code></td>
<td>Calculate Chebyshev distance between two vectors A and B.</td>
</tr>
<tr>
<td><code>std_euclidean(A, B)</code></td>
<td>Calculate Standardized Euclidean distance between two vectors A and B.</td>
</tr>
<tr>
<td><code>cosine(A, B)</code></td>
<td>Calculate Cosine distance between two vectors A and B.</td>
</tr>
<tr>
<td><code>csm(A, B)</code></td>
<td>Calculate Cosine similarity measure of distance between two vectors A and B.</td>
</tr>
<tr>
<td><code>squared_euclidean(A, B)</code></td>
<td>Calculate Squared Euclidean distance between two vectors A and B.</td>
</tr>
<tr>
<td><code>bray_curtis(A, B)</code></td>
<td>Calculate Bray-Curtis distance between two vectors A and B.</td>
</tr>
<tr>
<td><code>canberra(A, B)</code></td>
<td>Calculate Canberra distance between two vectors A and B.</td>
</tr>
<tr>
<td><code>lorentzian(A, B)</code></td>
<td>Calculate Lorentzian distance between two vectors A and B.</td>
</tr>
<tr>
<td><code>jaccard(A, B)</code></td>
<td>Calculate Jaccard distance between two vectors A and B.</td>
</tr>
<tr>
<td><code>dice(A, B)</code></td>
<td>Calculate Dice distance between two vectors A and B.</td>
</tr>
<tr>
<td><code>bhattacharyya(A, B)</code></td>
<td>Calculate Bhattacharyya distance between two vectors A and B.</td>
</tr>
<tr>
<td><code>hellinger(A, B)</code></td>
<td>Calculate Hellinger distance between two vectors A and B.</td>
</tr>
<tr>
<td><code>matusita(A, B)</code></td>
<td>Calculate Matusita distance between two vectors A and B.</td>
</tr>
<tr>
<td><code>squared_chord(A, B)</code></td>
<td>Calculate Squared-Chord distance between two vectors A and B.</td>
</tr>
<tr>
<td><code>pearson_chi_square(A, B)</code></td>
<td>Calculate Pearson Chi Square distance between two vectors A and B.</td>
</tr>
<tr>
<td><code>squared_chi_square(A, B)</code></td>
<td>Calculate Squared Chi Square distance between two vectors A and B.</td>
</tr>
</tbody>
</table>

### pyrepo_mcda.distance_metrics.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)
```

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)
```

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)
```

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` *(ndarray)* – distance value between two vectors

**Return type**
- float

Examples

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

pyrepo_mcda.distance_metrics.squared_chisquare(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` *(ndarray)* – distance value between two vectors

**Return type**
- float

Examples

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

pyrepo_mcda.normalizations

**Module Contents**
Functions

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

**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.\textbf{max\_normalization}(\textit{matrix}, \textit{types})

Normalize decision matrix using maximum normalization method.

Parameters

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

Returns

Normalized decision matrix

Return type

\textit{ndarray}

Examples

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

pyrepo_mcda.normalizations.\textbf{sum\_normalization}(\textit{matrix}, \textit{types})

Normalize decision matrix using sum normalization method.

Parameters

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

Returns

Normalized decision matrix

Return type

\textit{ndarray}

Examples

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

pyrepo_mcda.normalizations.\textbf{vector\_normalization}(\textit{matrix}, \textit{types})

Normalize decision matrix using vector normalization method.

Parameters

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

Returns

Normalized decision matrix

Return type

\textit{ndarray}
Examples

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

**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**

**Module Contents**

**Classes**

```python
class pyrepo_mcda.sensitivity_analysis_weights_percentages.
Sensitivity_analysis_weights_percentages
    __call__ (matrix, weights, types, percentages, method, j, dir_list)
    Method for sensitivity analysis. This method determines rankings of alternatives using chosen MCDA method name `mcda_name` for different modifications of criterion `j` weight.
    
    **Parameters**
    
    - `matrix (ndarray)` – Decision matrix with alternatives performances data. This matrix includes values of alternatives performances in rows considering criteria in columns
    
    - `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(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)

pyrepo_mcda.sensitivity_analysis_weights_values

Module Contents

Classes

Sensitivity_analysis_weights_values

class pyrepo_mcda.sensitivity_analysis_weights_values.Sensitivity_analysis_weights_values

__call__(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

- `matrix (ndarray)` – Decision matrix with performance values of alternatives. This matrix includes data on alternatives in rows considering criteria in columns.
• **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.

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

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

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

**Returns**

- **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**(matrix, weight_value, j)

Method for criteria weights changing in sensitivity analysis procedure.

**Parameters**

- **matrix** (*ndarray*) – Decision matrix with performance values of alternatives. This matrix includes data on alternatives in rows considering criteria in columns

- **weight_value** (*float*) – Value in range from 0 to 1 to be set as the weight of chosen criterion with index *j*.

- **j** (*int*) – Index of the column in decision matrix *matrix* indicating for which criterion the weight value is changed.

**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>
<tr>
<td><code>gini_weighting(matrix)</code></td>
<td>Calculate criteria weights using objective Gini coefficient-based weighting method.</td>
</tr>
<tr>
<td><code>merec_weighting(matrix, types)</code></td>
<td>Calculate criteria weights using objective MEREC weighting method.</td>
</tr>
<tr>
<td><code>stat_var_weighting(matrix)</code></td>
<td>Calculate criteria weights using objective Statistical variance weighting method.</td>
</tr>
<tr>
<td><code>cilos_weighting(matrix, types)</code></td>
<td>Calculate criteria weights using objective CILOS weighting method.</td>
</tr>
<tr>
<td><code>idocriw_weighting(matrix, types)</code></td>
<td>Calculate criteria weights using objective IDOCRIW weighting method.</td>
</tr>
<tr>
<td><code>angle_weighting(matrix, types)</code></td>
<td>Calculate criteria weights using objective Angle weighting method.</td>
</tr>
<tr>
<td><code>coeff_var_weighting(matrix)</code></td>
<td>Calculate criteria weights using objective Coefficient of variation weighting method.</td>
</tr>
</tbody>
</table>

### pyrepo_mcda.weighting_methods.equal_weighting(matrix)

Calculate criteria weights using objective Equal 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 = 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**

- ndarray
Examples

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

```python
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)
```

```python
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)
```

```python
pyrepo_mcda.weighting_methods.gini_weighting(matrix)
```

Calculate criteria weights using objective Gini coefficient-based 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 = gini_weighting(matrix)
```

`pyrepo_mcda.weighting_methods.merec_weighting(matrix, types)`

Calculate criteria weights using objective MEREC weighting method.

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

**Returns**
- Vector of criteria weights.

**Return type**
- ndarray

Examples

```python
>>> weights = merec_weighting(matrix, types)
```

`pyrepo_mcda.weighting_methods.stat_var_weighting(matrix)`

Calculate criteria weights using objective Statistical variance 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 = stat_var_weighting(matrix)
```

`pyrepo_mcda.weighting_methods.cilos_weighting(matrix, types)`

Calculate criteria weights using objective CILOS weighting method.

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

**Returns**
- Vector of criteria weights.

**Return type**
- ndarray

Examples

```python
>>> weights = cilos_weighting(matrix, types)
```
Calculate criteria weights using objective IDOCRIW weighting method.

Parameters

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

Returns

Vector of criteria weights.

Return type

`ndarray`

**Examples**

```python
>>> weights = idocriw_weighting(matrix, types)
```

Calculate criteria weights using objective Angle weighting method.

Parameters

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

Returns

Vector of criteria weights.

Return type

`ndarray`

**Examples**

```python
>>> weights = angle_weighting(matrix, types)
```

Calculate criteria weights using objective Coefficient of variation 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 = coeff_var_weighting(matrix)
```
p
pyrepo_mcda, 86
pyrepo_mcda.additions, 121
pyrepo_mcda.compromise_rankings, 122
pyrepo_mcda.correlations, 123
pyrepo_mcda.distance_metrics, 126
pyrepo_mcda.mcda_methods, 86
pyrepo_mcda.mcda_methods.ahp, 86
pyrepo_mcda.mcda_methods aras, 90
pyrepo_mcda.mcda_methods.codas, 91
pyrepo_mcda.mcda_methods.copras, 92
pyrepo_mcda.mcda_methods.cradis, 92
pyrepo_mcda.mcda_methods.edas, 93
pyrepo_mcda.mcda_methods.mabac, 94
pyrepo_mcda.mcda_methods.marcos, 95
pyrepo_mcda.mcda_methods.mcda_method, 95
pyrepo_mcda.mcda_methods.multimoora, 96
pyrepo_mcda.mcda_methods.promethee, 98
pyrepo_mcda.mcda_methods.prosa_c, 99
pyrepo_mcda.mcda_methods.saw, 101
pyrepo_mcda.mcda_methods.spotis, 101
pyrepo_mcda.mcda_methods.topsis, 102
pyrepo_mcda.mcda_methods.vikor, 103
pyrepo_mcda.mcda_methods.vikor_smaa, 104
pyrepo_mcda.mcda_methods.waspas, 105
pyrepo_mcda.normalizations, 133
pyrepo_mcda.sensitivity_analysis_weights_percentages, 136
pyrepo_mcda.sensitivity_analysis_weights_values, 137
pyrepo_mcda.weighting_methods, 138
Symbols

__call__(pyrepo_mcda.mcda_methods.AHP method), 107
__call__(pyrepo_mcda.mcda_methods.ARAS method), 110
__call__(pyrepo_mcda.mcda_methods.CODAS method), 111
__call__(pyrepo_mcda.mcda_methods.COPRAS method), 112
__call__(pyrepo_mcda.mcda_methods.CRADIS method), 112
__call__(pyrepo_mcda.mcda_methods.EDAS method), 113
__call__(pyrepo_mcda.mcda_methods.MABAC method), 113
__call__(pyrepo_mcda.mcda_methods.MARCOS method), 114
__call__(pyrepo_mcda.mcda_methods.MULTIMOORA method), 114
__call__(pyrepo_mcda.mcda_methods.MULTIMOORA_RS method), 115
__call__(pyrepo_mcda.mcda_methods.PROMETHEE II method), 115
__call__(pyrepo_mcda.mcda_methods.PROSA_C method), 116
__call__(pyrepo_mcda.mcda_methods.SAW method), 117
__call__(pyrepo_mcda.mcda_methods.SPOTIS method), 118
__call__(pyrepo_mcda.mcda_methods.TOPSIS method), 118
__call__(pyrepo_mcda.mcda_methods.VIKOR method), 119
__call__(pyrepo_mcda.mcda_methods.VIKOR_SMAA method), 120
__call__(pyrepo_mcda.mcda_methods.WASPAS method), 119
__call__(pyrepo_mcda.mcda_methods.ahp.AHP method), 86
__call__(pyrepo_mcda.mcda_methods.aras.ARAS method), 90
__call__(pyrepo_mcda.mcda_methods.codas.CODAS method), 107
__call__(pyrepo_mcda.mcda_methods.copras.COPRAS method), 92
__call__(pyrepo_mcda.mcda_methods.cradis.CRADIS method), 92
__call__(pyrepo_mcda.mcda_methods.edas.EDAS method), 93
__call__(pyrepo_mcda.mcda_methods.mabac.MABAC method), 94
__call__(pyrepo_mcda.mcda_methods.marcos.MARCOS method), 95
__call__(pyrepo_mcda.mcda_methods.multimoora.MULTIMOORA method), 96
__call__(pyrepo_mcda.mcda_methods.multimoora.MULTIMOORA_R method), 97
__call__(pyrepo_mcda.mcda_methods.multimoora.MULTIMOORA_RF method), 98
__call__(pyrepo_mcda.mcda_methods.multimoora.MULTIMOORA_RS method), 98
__call__(pyrepo_mcda.mcda_methods.multimoora.MULTIMOORA_F method), 97
__call__(pyrepo_mcda.mcda_methods.multimoora.MULTIMOORA_FINAL method), 98
__call__(pyrepo_mcda.mcda_methods.multimoora.MULTIMOORA_FSM method), 97
__call__(pyrepo_mcda.mcda_methods.multimoora.MULTIMOORA_FINAL method), 98
__call__(pyrepo_mcda.mcda_methods.promethee.PROMETHEE II method), 97
__call__(pyrepo_mcda.mcda_methods.promethee.PROMETHEE method), 98
__call__(pyrepo_mcda.mcda_methods.promethee.PROSA method), 100
__call__(pyrepo_mcda.mcda_methods.saw.SAW method), 101
__call__(pyrepo_mcda.mcda_methods.spotis.SPOTIS method), 102
__call__(pyrepo_mcda.mcda_methods.topsis.TOPSIS method), 102
__call__(pyrepo_mcda.mcda_methods.vikor.VIKOR method), 103
__call__(pyrepo_mcda.mcda_methods.vikor_smaa.VIKOR_SMAA method), 104
__call__(pyrepo_mcda.mcda_methods.waspas.WASPAS method), 105
__call__(pyrepo_mcda.sensitivity_analysis_weights_percentages.Sensitivity_analysis_weights_percentages method), 136
__call__(pyrepo_mcda.sensitivity_analysis_weights_values.Sensitivity_analysis_weights_values method), 137
<table>
<thead>
<tr>
<th>Function/Class</th>
<th>Module/Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>hellinger()</td>
<td>pyrepo_mcda.distance_metrics</td>
</tr>
<tr>
<td>idocriw_weighting()</td>
<td>pyrepo_mcda.weighting_methods</td>
</tr>
<tr>
<td>improved_borda_rule()</td>
<td>pyrepo_mcda.compromise_rankings</td>
</tr>
<tr>
<td>jaccard()</td>
<td>pyrepo_mcda.distance_metrics</td>
</tr>
<tr>
<td>linear_normalization()</td>
<td>pyrepo_mcda.normalizations</td>
</tr>
<tr>
<td>lorentzian()</td>
<td>pyrepo_mcda.distance_metrics</td>
</tr>
<tr>
<td>MABAC</td>
<td>pyrepo_mcda.mcda_methods</td>
</tr>
<tr>
<td>MABAC</td>
<td>pyrepo_mcda.mcda_methods.mabac</td>
</tr>
<tr>
<td>manhattan()</td>
<td>pyrepo_mcda.distance_metrics</td>
</tr>
<tr>
<td>MARCOS</td>
<td>pyrepo_mcda.mcda_methods.marcos</td>
</tr>
<tr>
<td>matusita()</td>
<td>pyrepo_mcda.distance_metrics</td>
</tr>
<tr>
<td>max_normalization()</td>
<td>pyrepo_mcda.normalizations</td>
</tr>
<tr>
<td>MCDA_method</td>
<td>pyrepo_mcda.mcda_methods.mcdma_method</td>
</tr>
<tr>
<td>mrec_weighting()</td>
<td>pyrepo_mcda.weighting_methods</td>
</tr>
<tr>
<td>minmax_normalization()</td>
<td>pyrepo_mcda.normalizations</td>
</tr>
<tr>
<td>module</td>
<td>pyrepo_mcda</td>
</tr>
<tr>
<td>pyrepo_mcda.additions</td>
<td>121</td>
</tr>
<tr>
<td>pyrepo_mcda.compromise_rankings</td>
<td>122</td>
</tr>
<tr>
<td>pyrepo_mcda.correlations</td>
<td>123</td>
</tr>
<tr>
<td>pyrepo_mcda.distance_metrics</td>
<td>126</td>
</tr>
<tr>
<td>pyrepo_mcda.mcda_methods</td>
<td>86</td>
</tr>
<tr>
<td>pyrepo_mcda.mcda_methods.ahp</td>
<td>86</td>
</tr>
<tr>
<td>pyrepo_mcda.mcda_methods.aras</td>
<td>90</td>
</tr>
<tr>
<td>pyrepo_mcda.mcda_methods.codas</td>
<td>91</td>
</tr>
<tr>
<td>pyrepo_mcda.mcda_methods.copras</td>
<td>92</td>
</tr>
<tr>
<td>pyrepo_mcda.mcda_methods.cradis</td>
<td>92</td>
</tr>
<tr>
<td>pyrepo_mcda.mcda_methods.edas</td>
<td>93</td>
</tr>
<tr>
<td>pyrepo_mcda.mcda_methods.mabac</td>
<td>94</td>
</tr>
<tr>
<td>pyrepo_mcda.mcda_methods.marcos</td>
<td>95</td>
</tr>
<tr>
<td>pyrepo_mcda.mcda_methods.mcdma_method</td>
<td>pyrepo_mcda.mcda_methods.mcdma_method</td>
</tr>
<tr>
<td>pyrepo_mcda.mcda_methods.multimoora</td>
<td>pyrepo_mcda.mcda_methods.mcdma_method</td>
</tr>
<tr>
<td>pyrepo_mcda.mcda_methods.promethee</td>
<td>pyrepo_mcda.mcda_methods.promethee</td>
</tr>
<tr>
<td>pyrepo_mcda.mcda_methods.prosa_c</td>
<td>pyrepo_mcda.mcda_methods.prosa_c</td>
</tr>
<tr>
<td>pyrepo_mcda.mcda_methods.spawas</td>
<td>105</td>
</tr>
<tr>
<td>pyrepo_mcda.mcda_methods.normalizations</td>
<td>133</td>
</tr>
<tr>
<td>pyrepo_mcda.mcda_methods.sensitivity_analysis_weights_percentages</td>
<td>136</td>
</tr>
<tr>
<td>pyrepo_mcda.mcda_methods.sensitivity_analysis_weights_values</td>
<td>137</td>
</tr>
<tr>
<td>pyrepo_mcda.mcda.mcda_methods.multimoora</td>
<td>pyrepo_mcda.mcda_methods.multimoora</td>
</tr>
<tr>
<td>pyrepo_mcda.mcda.mcda_methods.promethee</td>
<td>pyrepo_mcda.mcda_methods.promethee</td>
</tr>
<tr>
<td>pyrepo_mcda.mcda.mcda_methods.prosa_c</td>
<td>pyrepo_mcda.mcda_methods.prosa_c</td>
</tr>
<tr>
<td>pyrepo_mcda.mcda.mcda_methods.spawas</td>
<td>105</td>
</tr>
<tr>
<td>pyrepo_mcda.mcda.mcda_methods.normalizations</td>
<td>133</td>
</tr>
<tr>
<td>pearson_chi_square()</td>
<td>pyrepo_mcda.distance_metrics</td>
</tr>
<tr>
<td>pearson_coeff()</td>
<td>pyrepo_mcda.correlations</td>
</tr>
<tr>
<td>PROMETHEE_II</td>
<td>pyrepo_mcda.mcda_methods</td>
</tr>
<tr>
<td>PROMETHEE_II</td>
<td>pyrepo_mcda.mcda_methods.promethee</td>
</tr>
<tr>
<td>PROSA_C</td>
<td>pyrepo_mcda.mcda_methods.prosa_c</td>
</tr>
<tr>
<td>PROSA_C</td>
<td>pyrepo_mcda.mcda_methods.prosa_c</td>
</tr>
<tr>
<td>pyrepo_mcda</td>
<td>module, 86</td>
</tr>
<tr>
<td>pyrepo_mcda.additions</td>
<td>module, 121</td>
</tr>
<tr>
<td>pyrepo_mcda.compromise_rankings</td>
<td>module, 121</td>
</tr>
<tr>
<td>pyrepo_mcda.correlations</td>
<td>module, 122</td>
</tr>
</tbody>
</table>

**Index**
pyrepo_mcda.distance_metrics module, 126
pyrepo_mcda.mcda_methods module, 86
pyrepo_mcda.mcda_methods.ahp module, 86
pyrepo_mcda.mcda_methods.aras module, 90
pyrepo_mcda.mcda_methods.codas module, 91
pyrepo_mcda.mcda_methods.copras module, 92
pyrepo_mcda.mcda_methods.cradis module, 92
pyrepo_mcda.mcda_methods.edas module, 93
pyrepo_mcda.mcda_methods.mabac module, 94
pyrepo_mcda.mcda_methods.marcos module, 95
pyrepo_mcda.mcda_methods.mcda_method module, 95
pyrepo_mcda.mcda_methods.multimoora module, 96
pyrepo_mcda.mcda_methods.promethee module, 98
pyrepo_mcda.mcda_methods.prosa_c module, 99
pyrepo_mcda.mcda_methods.saw module, 101
pyrepo_mcda.mcda_methods.spotis module, 101
pyrepo_mcda.mcda_methods.topsis module, 102
pyrepo_mcda.mcda_methods.vikor module, 103
pyrepo_mcda.mcda_methods.vikor_smaa module, 104
pyrepo_mcda.mcda_methods.waspas module, 105
pyrepo mcda.normalizations module, 133
pyrepo mcda.sensitivity_analysis_weights_percentages module, 136
pyrepo mcda.sensitivity_analysis_weights_values module, 137
pyrepo mcda.weighting_methods module, 138
S
SAW (class in pyrepo_mcda.mcda_methods), 117
SAW (class in pyrepo_mcda.mcda_methods.saw), 101
Sensitivity_analysis_weights_percentages (class in pyrepo_mcda.sensitivity_analysis_weights_percentages), 136
Sensitivity_analysis_weights_values (class in pyrepo_mcda.sensitivity_analysis_weights_values), 137
spearman() (in module pyrepo_mcda.correlations), 124
SPOTIS (class in pyrepo_mcda.mcda_methods), 118
SPOTIS (class in pyrepo_mcda.mcda_methods.spotis), 101
squared_chi_square() (in module pyrepo_mcda.distance_metrics), 133
squared_chord() (in module pyrepo_mcda.distance_metrics), 132
squared_euclidean() (in module pyrepo_mcda.distance_metrics), 129
stat_var_weighting() (in module pyrepo_mcda.weighting_methods), 141
std_euclidean() (in module pyrepo_mcda.distance_metrics), 128
std_weighting() (in module pyrepo_mcda.weighting_methods), 140
sum_normalization() (in module pyrepo_mcda.normalizations), 135
T
TOPSIS (class in pyrepo_mcda.mcda_methods), 118
TOPSIS (class in pyrepo_mcda.mcda_methods.topsis), 102
V
vector_normalization() (in module pyrepo_mcda.normalizations), 135
VIKOR (class in pyrepo_mcda.mcda_methods), 119
VIKOR (class in pyrepo_mcda.mcda_methods.vikor), 103
VIKOR_SMAA (class in pyrepo_mcda.mcda_methods), 120
VIKOR_SMAA (class in pyrepo_mcda.mcda_methods.vikor_smaa), 104
W
WASPAS (class in pyrepo_mcda.mcda_methods), 119
WASPAS (class in pyrepo_mcda.mcda_methods.waspas), 105
weighted_spearman() (in module pyrepo_mcda.correlations), 124
WS_coeff() (in module pyrepo_mcda.correlations), 125
R
rank_position_method() (in module pyrepo_mcda.compromise_rankings), 123
rank_preferences() (in module pyrepo_mcda.additions), 121