## CONTENTS

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 Contents</td>
<td>3</td>
</tr>
<tr>
<td>1.1 Usage</td>
<td>3</td>
</tr>
<tr>
<td>1.2 Illustrative example</td>
<td>11</td>
</tr>
<tr>
<td>1.3 API Reference</td>
<td>28</td>
</tr>
<tr>
<td>Python Module Index</td>
<td>43</td>
</tr>
<tr>
<td>Index</td>
<td>45</td>
</tr>
</tbody>
</table>
objective-weighting is Python 3 library dedicated to multi-criteria decision analysis with criteria weights determined by objective weighting methods. This library includes:

- The VIKOR method VIKOR
- 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)
- Correlation coefficients:
  - spearman (Spearman rank correlation coefficient)
  - weighted_spearman (Weighted Spearman rank correlation coefficient)
  - pearson_coeff (Pearson correlation 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)
- 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 `objective_weighting` package, first install it using pip:

```
pip install objective-weighting
```

1.1.2 Usage examples

The VIKOR method

The VIKOR method provided in this library can be used with single weight vector and with multiple weight vectors, like in the Stochastic Multicriteria Acceptability Analysis (SMAA) method.

Using the VIKOR method with single weight vector:

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

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

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

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

# Create the VIKOR method object providing `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.
```
vikor = VIKOR(v = 0.625)

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

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

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

Output

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

The VIKOR method provided in the objective-weighting library can also be used with multiple weight vectors provided in the matrix. This matrix includes weight vectors in rows. The number of rows is equal to the vectors number, and the number of columns is equal to the criteria number. In this case, the VIKOR method returns a matrix with preference values. Vectors with preference values for each weight vector are contained in each column. The number of rows of the matrix with preference values is equal to the number of alternatives, and the number of columns is equal to the number of weight vectors. This functionality is useful for Stochastic Multicriteria Acceptability Analysis (SMAA) methods. Here is demonstrated how it works using the VIKOR method with multiple weight vectors.

```python
import numpy as np
from objective_weighting.additions import rank_preferences
from objective_weighting.mcda_methods import VIKOR, VIKOR_SMAA

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]])

n = matrix.shape[1]
iterations = 10

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

vikor_smaa = VIKOR_SMAA()
weight_vectors = vikor_smaa._generate_weights(n, iterations)

vikor = VIKOR()
pref = vikor(matrix, weight_vectors, types)
print(pref)
```
Matrix with preference values includes subsequent vectors with preference values in columns. We can rank preferences in this matrix using the `rank_preferences` method in following way:

```python
rank = np.zeros((pref.shape))
for i in range(pref.shape[1]):
    rank[:, i] = rank_preferences(pref[:, i], reverse = False)
print('Rankings: ', rank)
```

Now each column of the above matrix contains a ranking generated for each weight vector.

### Correlation coefficients

Spearman correlation coefficient

```python
import numpy as np
from objective_weighting 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))
```

Now each column of the above matrix contains a ranking generated for each weight vector.

### Correlation coefficients

Spearman correlation coefficient

```python
import numpy as np
from objective_weighting 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

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

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

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

Output

```
Weighted Spearman coeff: 0.8833
```

Pearson correlation coefficient

```python
import numpy as np
from objective_weighting 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

```python
import numpy as np
from objective_weighting 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

```python
import numpy as np
from objective_weighting 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

```python
import numpy as np
from objective_weighting 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 objective_weighting 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

```
```

1.1. Usage
Equal weights: [0.333 0.333 0.333]

Gini coefficient-based weighting method

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

matrix = np.array([[29.4, 83, 47, 114, 12, 30, 120, 170, 90, 1717.75],
                   [38.1, 124.7, 117, 16, 60, 60, 60, 93, 70, 2389],
                   [29.28, 41.13, 58, 16, 30, 60, 120, 170, 78, 239.99],
                   [33.6, 71, 55, 23.6, 60, 240, 240, 132, 140, 2099],
                   [21, 41, 66, 16, 24, 60, 120, 170, 70, 439],
                   [35, 65, 134, 12, 60, 240, 240, 145, 60, 1087],
                   [47, 79, 158, 19, 60, 120, 120, 360, 72, 2499],
                   [28.3, 62.3, 44.9, 116, 12, 60, 60, 130, 90, 999.99],
                   [36.9, 28.6, 121.6, 130, 12, 60, 120, 120, 80, 80, 1099],
                   [32, 41, 60, 16, 30, 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, 360, 72, 2099],
                   [20.2, 64, 80, 70, 8, 24, 60, 120, 166, 480, 699.99],
                   [33, 60, 44, 59, 12, 30, 60, 120, 170, 90, 388],
                   [29, 59, 41, 55, 16, 30, 60, 120, 170, 120, 299],
                   [29, 59, 41, 182, 12, 30, 30, 60, 94, 140, 249],
                   [29.8, 59.2, 41, 65, 16, 30, 60, 120, 160, 90, 219.99],
                   [28.8, 62.5, 41, 70, 12, 60, 120, 120, 170, 138, 1399.99],
                   [24, 50, 59, 60, 12, 10, 30, 30, 140, 78, 269.99],
                   [30, 60, 45, 201, 16, 30, 30, 170, 90, 199.99]])

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 objective_weighting 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 objective_weighting 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 objective_weighting import weighting_methods as mcda_weights

matrix = np.array([[3, 100, 10, 7],
                   [2.500, 80, 8, 5],
                   [1.800, 50, 20, 11],
                   [2.200, 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 0.22 0.196 0.25 ]

IDOCRIW weighting method

```python
import numpy as np
from objective_weighting 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 objective_weighting 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: [0.415 0.3612 0.2227 0.0012]
```

Coefficient of variation weighting method

```python
import numpy as np
from objective_weighting 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)

```python
from objective_weighting.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)
```

(continues on next page)
# Calculate Rank acceptability index, Central weight vector and final ranking based on SMAA method combined with VIKOR

```python
rank_acceptability_index, central_weight_vector, rank_scores = vikor_smaa(matrix, weight_vectors, types)
```

## 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 are used in analogous way.

Vector normalization

```python
import numpy as np
from objective_weighting 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])

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]]
```

## 1.2 Illustrative example

This example explains the usage of the library package objective_weighting 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.

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

Import the necessary modules and methods from package objective_weighting.

[2]: from objective_weighting.mcda_methods import VIKOR
from objective_weighting.mcda_methods import VIKOR_SMAA
from objective_weighting.additions import rank_preferences
from objective_weighting import correlations as corrs
from objective_weighting import normalizations as norm_methods
from objective_weighting import weighting_methods as mcda_weights

Functions for results visualization.

[3]: # 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`
    """
    # (continues on next page)
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()
```

1.2. Illustrative example
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')
plt.ylabel('Alternatives')
plt.title('Correlation coefficient: ' + title)
plt.tight_layout()
plt.savefig('results/heatmap_weights.pdf')
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')
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.

### 1.2. Illustrative example
**objective-weighting, Release 0.1**

```python
[4]: criteria_presentation = pd.read_csv('criteria_electric_cars.csv', index_col = 'Cj')
criteria_presentation

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

```python
[5]: data_presentation = pd.read_csv('electric_cars_2021.csv', index_col = 'Ai')
data_presentation

<table>
<thead>
<tr>
<th>Name</th>
<th>C1 Max speed [mph]</th>
<th>C2 Battery [kWh]</th>
</tr>
</thead>
<tbody>
<tr>
<td>A1 Tesla Model Y</td>
<td>155.3</td>
<td>74.0</td>
</tr>
<tr>
<td>A2 Tesla Model 3</td>
<td>162.2</td>
<td>79.5</td>
</tr>
<tr>
<td>A3 Ford Mustang Mach-E</td>
<td>112.5</td>
<td>68.0</td>
</tr>
<tr>
<td>A4 Chevrolet Bolt EV and EUV</td>
<td>90.1</td>
<td>66.0</td>
</tr>
<tr>
<td>A5 Volkswagen ID.4</td>
<td>99.4</td>
<td>77.0</td>
</tr>
<tr>
<td>A6 Nissan Leaf</td>
<td>89.5</td>
<td>40.0</td>
</tr>
<tr>
<td>A7 Audi e-tron and e-tron Sportback</td>
<td>124.3</td>
<td>95.0</td>
</tr>
<tr>
<td>A8 Porsche Taycan</td>
<td>155.3</td>
<td>79.2</td>
</tr>
<tr>
<td>A9 Tesla Model S</td>
<td>162.2</td>
<td>100.0</td>
</tr>
<tr>
<td>A10 Hyundai Kona Electric</td>
<td>96.3</td>
<td>39.2</td>
</tr>
<tr>
<td>A11 Tesla Model X</td>
<td>162.2</td>
<td>100.0</td>
</tr>
<tr>
<td>A12 Hyundai Ioniq Electric</td>
<td>102.5</td>
<td>38.3</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Type</th>
<th>C3 Electric motor [kW] Front</th>
<th>C4 Torque [Nm] Front</th>
</tr>
</thead>
<tbody>
<tr>
<td>A1</td>
<td>340</td>
<td>673</td>
</tr>
<tr>
<td>A2</td>
<td>247</td>
<td>639</td>
</tr>
<tr>
<td>A3</td>
<td>198</td>
<td>430</td>
</tr>
<tr>
<td>A4</td>
<td>150</td>
<td>360</td>
</tr>
<tr>
<td>A5</td>
<td>150</td>
<td>310</td>
</tr>
<tr>
<td>A6</td>
<td>110</td>
<td>320</td>
</tr>
<tr>
<td>A7</td>
<td>125</td>
<td>247</td>
</tr>
<tr>
<td>A8</td>
<td>160</td>
<td>300</td>
</tr>
<tr>
<td>A9</td>
<td>205</td>
<td>420</td>
</tr>
<tr>
<td>A10</td>
<td>100</td>
<td>395</td>
</tr>
<tr>
<td>A11</td>
<td>205</td>
<td>420</td>
</tr>
<tr>
<td>A12</td>
<td>101</td>
<td>295</td>
</tr>
<tr>
<td>Type</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Type</th>
<th>C5 Mechanical horsepower [hp]</th>
<th>C6 EPA Fuel Economy Combined [MPGe]</th>
</tr>
</thead>
</table>

(continues on next page)
<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Ai</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A1</td>
<td>456.0</td>
<td></td>
<td>111</td>
<td></td>
</tr>
<tr>
<td>A2</td>
<td>283.0</td>
<td></td>
<td>113</td>
<td></td>
</tr>
<tr>
<td>A3</td>
<td>266.0</td>
<td></td>
<td>98</td>
<td></td>
</tr>
<tr>
<td>A4</td>
<td>201.2</td>
<td></td>
<td>120</td>
<td></td>
</tr>
<tr>
<td>A5</td>
<td>201.2</td>
<td></td>
<td>97</td>
<td></td>
</tr>
<tr>
<td>A6</td>
<td>147.5</td>
<td></td>
<td>111</td>
<td></td>
</tr>
<tr>
<td>A7</td>
<td>187.7</td>
<td></td>
<td>78</td>
<td></td>
</tr>
<tr>
<td>A8</td>
<td>214.6</td>
<td></td>
<td>79</td>
<td></td>
</tr>
<tr>
<td>A9</td>
<td>502.9</td>
<td></td>
<td>120</td>
<td></td>
</tr>
<tr>
<td>A10</td>
<td>134.1</td>
<td></td>
<td>120</td>
<td></td>
</tr>
<tr>
<td>A11</td>
<td>502.9</td>
<td></td>
<td>98</td>
<td></td>
</tr>
<tr>
<td>A12</td>
<td>136.1</td>
<td></td>
<td>133</td>
<td></td>
</tr>
<tr>
<td>Type</td>
<td>1.0</td>
<td></td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

C7 EPA Fuel Economy City [MPGe]  C8 EPA Fuel Economy Highway [MPGe]  

<table>
<thead>
<tr>
<th>Ai</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>A1</td>
<td>115</td>
<td>106</td>
</tr>
<tr>
<td>A2</td>
<td>118</td>
<td>107</td>
</tr>
<tr>
<td>A3</td>
<td>105</td>
<td>91</td>
</tr>
<tr>
<td>A4</td>
<td>131</td>
<td>109</td>
</tr>
<tr>
<td>A5</td>
<td>102</td>
<td>90</td>
</tr>
<tr>
<td>A6</td>
<td>123</td>
<td>99</td>
</tr>
<tr>
<td>A7</td>
<td>78</td>
<td>77</td>
</tr>
<tr>
<td>A8</td>
<td>79</td>
<td>80</td>
</tr>
<tr>
<td>A9</td>
<td>124</td>
<td>115</td>
</tr>
<tr>
<td>A10</td>
<td>132</td>
<td>108</td>
</tr>
<tr>
<td>A11</td>
<td>103</td>
<td>93</td>
</tr>
<tr>
<td>A12</td>
<td>145</td>
<td>121</td>
</tr>
<tr>
<td>Type</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

C9 EPA range [miles]  

<table>
<thead>
<tr>
<th>Ai</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>A1</td>
<td>244</td>
</tr>
<tr>
<td>A2</td>
<td>263</td>
</tr>
<tr>
<td>A3</td>
<td>230</td>
</tr>
<tr>
<td>A4</td>
<td>259</td>
</tr>
<tr>
<td>A5</td>
<td>260</td>
</tr>
<tr>
<td>A6</td>
<td>226</td>
</tr>
<tr>
<td>A7</td>
<td>222</td>
</tr>
<tr>
<td>A8</td>
<td>227</td>
</tr>
<tr>
<td>A9</td>
<td>402</td>
</tr>
<tr>
<td>A10</td>
<td>258</td>
</tr>
<tr>
<td>A11</td>
<td>371</td>
</tr>
<tr>
<td>A12</td>
<td>170</td>
</tr>
<tr>
<td>Type</td>
<td>1</td>
</tr>
</tbody>
</table>

C10 Turning Diameter / Radius, curb to curb [feet]  C11 Base price [$]  

<table>
<thead>
<tr>
<th>Ai</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>A1</td>
<td>39.8</td>
<td>65440</td>
</tr>
<tr>
<td>A2</td>
<td>38.8</td>
<td>60440</td>
</tr>
<tr>
<td>A3</td>
<td>38.1</td>
<td>56575</td>
</tr>
</tbody>
</table>

(continues on next page)
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
# 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()

# Symbols for alternatives Ai
list_alt_names = [r'$A_{i + str(i)}$' for i in range(1, df_data.shape[0] + 1)]
# Symbols for columns Cj
cols = [r'$C_{j + str(j)}$' for j in range(1, data.shape[1] + 1)]
print('Decision matrix')
df_data

Decision matrix

<table>
<thead>
<tr>
<th></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>Ai</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<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>34.8</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>

Criteria types

print('Criteria types')
types

Criteria types
```
1.2.1 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)
```

```
<p>| | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>$C_{1}$</td>
<td>$C_{2}$</td>
<td>$C_{3}$</td>
<td>$C_{4}$</td>
<td>$C_{5}$</td>
<td>$C_{6}$</td>
<td>$C_{7}$</td>
<td>Weights</td>
</tr>
<tr>
<td>0.057741</td>
<td>0.099843</td>
<td>0.142673</td>
<td>0.096488</td>
<td>0.236087</td>
<td>0.024544</td>
<td>0.032432</td>
<td></td>
</tr>
<tr>
<td>$C_{8}$</td>
<td>$C_{9}$</td>
<td>$C_{10}$</td>
<td>$C_{11}$</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0.018126</td>
<td>0.053958</td>
<td>0.003863</td>
<td>0.234244</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

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)

# Calculate alternatives preference function values with VIKOR method
pref = vikor(matrix, weights, types)

# rank alternatives according to preference values
rank = rank_preferences(pref, reverse = False)

df_results = pd.DataFrame(index = list_alt_names)
df_results['Pref'] = pref
df_results['Rank'] = rank
df_results
```

```
<p>| | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Pref</td>
<td>Rank</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$A_{1}$</td>
<td>0.000000</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$A_{2}$</td>
<td>0.325154</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$A_{3}$</td>
<td>0.531050</td>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$A_{4}$</td>
<td>0.682258</td>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$A_{5}$</td>
<td>0.734162</td>
<td>7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$A_{6}$</td>
<td>0.922091</td>
<td>10</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$A_{7}$</td>
<td>0.884828</td>
<td>9</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$A_{8}$</td>
<td>0.821773</td>
<td>8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$A_{9}$</td>
<td>0.332600</td>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$A_{10}$</td>
<td>0.940460</td>
<td>11</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$A_{11}$</td>
<td>0.696434</td>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$A_{12}$</td>
<td>0.954832</td>
<td>12</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

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 = [
(continues on next page)
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.

```python
[11]: df_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
```

[12]: df_weights

```csv
C_1 0.057741 0.093960 0.080882 0.067363 0.143855 0.089362
    C_2 0.099843 0.099277 0.103800 0.125195 0.103976 0.076405
    C_3 0.142673 0.066132 0.128202 0.103489 0.067308 0.094271
    C_4 0.096488 0.075874 0.103200 0.093050 0.076665 0.079572
    C_5 0.236887 0.071195 0.163531 0.124581 0.112880 0.154235
    C_6 0.024544 0.112865 0.052308 0.064886 0.074361 0.071876
    C_7 0.032432 0.120602 0.060388 0.077107 0.073925 0.076822
    C_8 0.018126 0.103536 0.046188 0.053708 0.076150 0.069418
```

### Objective-Weighting

(continued from previous page)

<table>
<thead>
<tr>
<th>C_9</th>
<th>0.053958</th>
<th>0.065514</th>
<th>0.073099</th>
<th>0.087109</th>
<th>0.060565</th>
<th>0.039702</th>
</tr>
</thead>
<tbody>
<tr>
<td>C_10</td>
<td>0.003863</td>
<td>0.098432</td>
<td>0.021151</td>
<td>0.018566</td>
<td>0.126025</td>
<td>0.017062</td>
</tr>
<tr>
<td>C_11</td>
<td>0.234244</td>
<td>0.092612</td>
<td>0.167270</td>
<td>0.184947</td>
<td>0.084289</td>
<td>0.231276</td>
</tr>
</tbody>
</table>

#### ANGLE

<table>
<thead>
<tr>
<th>C_1</th>
<th>0.081732</th>
<th>0.079378</th>
</tr>
</thead>
<tbody>
<tr>
<td>C_2</td>
<td>0.103002</td>
<td>0.101129</td>
</tr>
<tr>
<td>C_3</td>
<td>0.129702</td>
<td>0.129595</td>
</tr>
<tr>
<td>C_4</td>
<td>0.108379</td>
<td>0.106746</td>
</tr>
<tr>
<td>C_5</td>
<td>0.162354</td>
<td>0.166788</td>
</tr>
<tr>
<td>C_6</td>
<td>0.053145</td>
<td>0.051074</td>
</tr>
<tr>
<td>C_7</td>
<td>0.060739</td>
<td>0.058510</td>
</tr>
<tr>
<td>C_8</td>
<td>0.046061</td>
<td>0.044183</td>
</tr>
<tr>
<td>C_9</td>
<td>0.081691</td>
<td>0.079337</td>
</tr>
<tr>
<td>C_10</td>
<td>0.021711</td>
<td>0.020518</td>
</tr>
<tr>
<td>C_11</td>
<td>0.151484</td>
<td>0.162742</td>
</tr>
</tbody>
</table>

#### COEFF

<table>
<thead>
<tr>
<th>C_1</th>
<th>0.081732</th>
<th>0.079378</th>
</tr>
</thead>
<tbody>
<tr>
<td>C_2</td>
<td>0.103002</td>
<td>0.101129</td>
</tr>
<tr>
<td>C_3</td>
<td>0.129702</td>
<td>0.129595</td>
</tr>
<tr>
<td>C_4</td>
<td>0.108379</td>
<td>0.106746</td>
</tr>
<tr>
<td>C_5</td>
<td>0.162354</td>
<td>0.166788</td>
</tr>
<tr>
<td>C_6</td>
<td>0.053145</td>
<td>0.051074</td>
</tr>
<tr>
<td>C_7</td>
<td>0.060739</td>
<td>0.058510</td>
</tr>
<tr>
<td>C_8</td>
<td>0.046061</td>
<td>0.044183</td>
</tr>
<tr>
<td>C_9</td>
<td>0.081691</td>
<td>0.079337</td>
</tr>
<tr>
<td>C_10</td>
<td>0.021711</td>
<td>0.020518</td>
</tr>
<tr>
<td>C_11</td>
<td>0.151484</td>
<td>0.162742</td>
</tr>
</tbody>
</table>

### Illustrative Example

#### df_preferences

<table>
<thead>
<tr>
<th>A_1</th>
<th>0.000000</th>
<th>0.193324</th>
<th>0.000000</th>
<th>0.000000</th>
<th>0.210477</th>
<th>0.000000</th>
</tr>
</thead>
<tbody>
<tr>
<td>A_2</td>
<td>0.325154</td>
<td>0.053863</td>
<td>0.267784</td>
<td>0.096602</td>
<td>0.062729</td>
<td>0.100057</td>
</tr>
<tr>
<td>A_3</td>
<td>0.531050</td>
<td>0.351973</td>
<td>0.351973</td>
<td>0.351973</td>
<td>0.351973</td>
<td>0.351973</td>
</tr>
<tr>
<td>A_4</td>
<td>0.682258</td>
<td>0.384420</td>
<td>0.629619</td>
<td>0.376115</td>
<td>0.705929</td>
<td>0.353278</td>
</tr>
<tr>
<td>A_5</td>
<td>0.734162</td>
<td>0.449121</td>
<td>0.713059</td>
<td>0.485436</td>
<td>0.680768</td>
<td>0.473333</td>
</tr>
<tr>
<td>A_6</td>
<td>0.922991</td>
<td>0.558323</td>
<td>0.879933</td>
<td>0.619888</td>
<td>0.856815</td>
<td>0.549559</td>
</tr>
<tr>
<td>A_7</td>
<td>0.884828</td>
<td>1.000000</td>
<td>0.869011</td>
<td>0.662208</td>
<td>0.710609</td>
<td>0.657640</td>
</tr>
<tr>
<td>A_8</td>
<td>0.821773</td>
<td>0.920743</td>
<td>0.786866</td>
<td>0.580755</td>
<td>0.677000</td>
<td>0.528558</td>
</tr>
<tr>
<td>A_9</td>
<td>0.332600</td>
<td>0.223787</td>
<td>0.289556</td>
<td>0.255499</td>
<td>0.263261</td>
<td>0.301515</td>
</tr>
<tr>
<td>A_10</td>
<td>0.940460</td>
<td>0.490234</td>
<td>0.868050</td>
<td>0.580755</td>
<td>0.677000</td>
<td>0.528558</td>
</tr>
<tr>
<td>A_11</td>
<td>0.696434</td>
<td>0.493401</td>
<td>0.676774</td>
<td>0.682902</td>
<td>0.506772</td>
<td>0.732254</td>
</tr>
<tr>
<td>A_12</td>
<td>0.954832</td>
<td>0.453666</td>
<td>0.869930</td>
<td>0.585575</td>
<td>0.544842</td>
<td>0.495033</td>
</tr>
</tbody>
</table>

#### df_rankings

<table>
<thead>
<tr>
<th>A_1</th>
<th>1</th>
<th>2</th>
<th>1</th>
<th>1</th>
<th>2</th>
<th>1</th>
<th>1</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>A_2</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>3</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)
The PostScript backend does not support transparency; partially transparent artists will be rendered opaque.

1.2. Illustrative example

```python
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]))
```

(continues on next page)
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'$r_w$')

1.2.2 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.

<table>
<thead>
<tr>
<th>Alternative</th>
<th>Rank 1</th>
<th>Rank 2</th>
<th>Rank 3</th>
<th>Rank 4</th>
<th>Rank 5</th>
<th>Rank 6</th>
<th>Rank 7</th>
<th>Rank 8</th>
</tr>
</thead>
<tbody>
<tr>
<td>A_1</td>
<td>0.2832</td>
<td>0.2568</td>
<td>0.1835</td>
<td>0.1387</td>
<td>0.0527</td>
<td>0.0515</td>
<td>0.0225</td>
<td>0.0418</td>
</tr>
<tr>
<td>A_2</td>
<td>0.2275</td>
<td>0.3477</td>
<td>0.2215</td>
<td>0.1245</td>
<td>0.0426</td>
<td>0.0295</td>
<td>0.0067</td>
<td>0.0000</td>
</tr>
<tr>
<td>A_3</td>
<td>0.0003</td>
<td>0.0103</td>
<td>0.0245</td>
<td>0.0766</td>
<td>0.2942</td>
<td>0.1366</td>
<td>0.1508</td>
<td>0.1312</td>
</tr>
<tr>
<td>A_4</td>
<td>0.1000</td>
<td>0.0661</td>
<td>0.0732</td>
<td>0.1385</td>
<td>0.1657</td>
<td>0.2421</td>
<td>0.0772</td>
<td>0.0375</td>
</tr>
<tr>
<td>A_5</td>
<td>0.0003</td>
<td>0.0106</td>
<td>0.0125</td>
<td>0.0195</td>
<td>0.0732</td>
<td>0.1075</td>
<td>0.2602</td>
<td>0.1469</td>
</tr>
<tr>
<td>A_6</td>
<td>0.0000</td>
<td>0.0003</td>
<td>0.0086</td>
<td>0.0437</td>
<td>0.0216</td>
<td>0.0350</td>
<td>0.1309</td>
<td>0.1130</td>
</tr>
<tr>
<td>A_7</td>
<td>0.0000</td>
<td>0.0000</td>
<td>0.0008</td>
<td>0.0013</td>
<td>0.0063</td>
<td>0.0342</td>
<td>0.0294</td>
<td>0.0343</td>
</tr>
<tr>
<td>A_8</td>
<td>0.0000</td>
<td>0.0018</td>
<td>0.0014</td>
<td>0.0663</td>
<td>0.0650</td>
<td>0.0458</td>
<td>0.0575</td>
<td>0.1444</td>
</tr>
<tr>
<td>A_9</td>
<td>0.3883</td>
<td>0.1099</td>
<td>0.2878</td>
<td>0.0365</td>
<td>0.0311</td>
<td>0.0269</td>
<td>0.0204</td>
<td>0.0157</td>
</tr>
<tr>
<td>A_10</td>
<td>0.0096</td>
<td>0.0398</td>
<td>0.0651</td>
<td>0.0694</td>
<td>0.0870</td>
<td>0.1581</td>
<td>0.0947</td>
<td>0.0754</td>
</tr>
<tr>
<td>A_11</td>
<td>0.0000</td>
<td>0.1106</td>
<td>0.0861</td>
<td>0.2885</td>
<td>0.0747</td>
<td>0.0533</td>
<td>0.0587</td>
<td>0.0949</td>
</tr>
<tr>
<td>A_12</td>
<td>0.0378</td>
<td>0.0461</td>
<td>0.0350</td>
<td>0.0565</td>
<td>0.0859</td>
<td>0.0795</td>
<td>0.0910</td>
<td>0.1649</td>
</tr>
</tbody>
</table>

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

```python
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

[26]: `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.

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

```
[28]:

central_weights_df

<table>
<thead>
<tr>
<th>$C_{1}$</th>
<th>$C_{2}$</th>
<th>$C_{3}$</th>
<th>$C_{4}$</th>
<th>$C_{5}$</th>
<th>$C_{6}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>A_{1}</td>
<td>0.081767</td>
<td>0.067123</td>
<td>0.168040</td>
<td>0.123759</td>
<td>0.122593</td>
</tr>
<tr>
<td>A_{2}</td>
<td>0.120370</td>
<td>0.088817</td>
<td>0.074862</td>
<td>0.126533</td>
<td>0.055946</td>
</tr>
<tr>
<td>A_{3}</td>
<td>0.011378</td>
<td>0.031920</td>
<td>0.221420</td>
<td>0.093060</td>
<td>0.137050</td>
</tr>
<tr>
<td>A_{4}</td>
<td>0.042460</td>
<td>0.080267</td>
<td>0.066423</td>
<td>0.054914</td>
<td>0.067923</td>
</tr>
<tr>
<td>A_{5}</td>
<td>0.043732</td>
<td>0.293897</td>
<td>0.031455</td>
<td>0.027025</td>
<td>0.062786</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>
</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>
</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>
</tr>
<tr>
<td>A_{9}</td>
<td>0.096682</td>
<td>0.116171</td>
<td>0.065802</td>
<td>0.062445</td>
<td>0.105615</td>
</tr>
<tr>
<td>A_{10}</td>
<td>0.052157</td>
<td>0.033426</td>
<td>0.037122</td>
<td>0.148589</td>
<td>0.036476</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>
</tr>
<tr>
<td>A_{12}</td>
<td>0.078808</td>
<td>0.036380</td>
<td>0.040406</td>
<td>0.048161</td>
<td>0.042576</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>$C_{7}$</th>
<th>$C_{8}$</th>
<th>$C_{9}$</th>
<th>$C_{10}$</th>
<th>$C_{11}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>A_{1}</td>
<td>0.073561</td>
<td>0.076675</td>
<td>0.058107</td>
<td>0.054456</td>
</tr>
<tr>
<td>A_{2}</td>
<td>0.078061</td>
<td>0.081630</td>
<td>0.058107</td>
<td>0.054456</td>
</tr>
<tr>
<td>A_{3}</td>
<td>0.078061</td>
<td>0.081630</td>
<td>0.058107</td>
<td>0.054456</td>
</tr>
<tr>
<td>A_{4}</td>
<td>0.092394</td>
<td>0.080736</td>
<td>0.086954</td>
<td>0.207156</td>
</tr>
<tr>
<td>A_{5}</td>
<td>0.042495</td>
<td>0.016120</td>
<td>0.046958</td>
<td>0.240973</td>
</tr>
<tr>
<td>A_{6}</td>
<td>0.000000</td>
<td>0.000000</td>
<td>0.000000</td>
<td>0.000000</td>
</tr>
<tr>
<td>A_{7}</td>
<td>0.000000</td>
<td>0.000000</td>
<td>0.000000</td>
<td>0.000000</td>
</tr>
<tr>
<td>A_{8}</td>
<td>0.000000</td>
<td>0.000000</td>
<td>0.000000</td>
<td>0.000000</td>
</tr>
<tr>
<td>A_{9}</td>
<td>0.099716</td>
<td>0.016120</td>
<td>0.046958</td>
<td>0.240973</td>
</tr>
<tr>
<td>A_{10}</td>
<td>0.098032</td>
<td>0.082048</td>
<td>0.071769</td>
<td>0.236257</td>
</tr>
<tr>
<td>A_{11}</td>
<td>0.000000</td>
<td>0.000000</td>
<td>0.000000</td>
<td>0.000000</td>
</tr>
<tr>
<td>A_{12}</td>
<td>0.155527</td>
<td>0.139289</td>
<td>0.039885</td>
<td>0.159784</td>
</tr>
</tbody>
</table>
```

Rank scores

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

```
[30]:

rank_scores_df

<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)

1.2. Illustrative example 27
1.3 API Reference

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

1.3.1 objective_weighting

Subpackages

objective_weighting.mcda_methods

Submodules

objective_weighting.mcda_methods.mcda_method

Module Contents

Classes

<table>
<thead>
<tr>
<th>Class</th>
<th>objective_weighting.mcda_methods.mcda_method.MCDA_method</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Helper class that provides a standard way to create an ABC using inheritance.</td>
</tr>
</tbody>
</table>

```
class objective_weighting.mcda_methods.mcda_method.MCDA_method
    Bases: abc.ABC
    Helper class that provides a standard way to create an ABC using inheritance.
    __call__(self, matrix, weights, types)
        Score alternatives from decision matrix matrix using criteria weights weights and criteria types types

Parameters

* matrix (ndarray) – decision matrix with performance values for m alternatives in rows and n criteria in columns
* weights (ndarray) – matrix with criteria weights vectors with number of columns equal to number of columns n of matrix
```

1 Created with sphinx-autoapi
• **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 \( \text{matrix} \)

    static _verify_input_data(matrix, weights, types)

    objective_weighting.mcda_methods.vikor

Module Contents

Classes

    VIKOR

**class** objective_weighting.mcda_methods.vikor.VIKOR(normalization_method=None, v=0.5)

**Bases:** objective_weighting.mcda_methods.mcda_method.MCDA_method

    __call__(self, matrix, weights, types)

    Score alternatives provided in decision matrix \( \text{matrix} \) using criteria \( \text{weights} \) and criteria \( \text{types} \).

    **Parameters**

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

    **Returns**

    Matrix with vectors containing preference values of each alternative. The best alternative has the lowest preference value. Vectors are placed in subsequent columns of matrix.

    **Return type**

    ndarray

**Examples**

```python
>>> vikor = VIKOR(normalization_method = minmax_normalization)
>>> pref = vikor(matrix, weights, types)
>>> rank = np.zeros((pref.shape))
>>> for i in range(pref.shape[1]):
...    rank[:, i] = rank_preferences(pref[:, i], reverse = False)
```

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

1.3. API Reference
Module Contents

Classes

VIKOR_SMAA

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

__call__(self, matrix, weights, types)

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

Parameters

- **matrix** (ndarray) – Decision matrix with m alternatives in rows and n criteria in columns.
- **weights** (ndarray) – Matrix with i vectors in rows of n weights in columns. i means
  number of iterations of SMMA
- **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(self, 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

static _vikor_smaa(self, matrix, weights, types, normalization_method, v)
Package Contents

Classes

**VIKOR**

**VIKOR_SMAA**

class objective_weighting.mcda_methods.VIKOR(normalization_method=None, v=0.5)

Bases: objective_weighting.mcda_methods.mcda_method.MCDA_method

__call__(self, matrix, weights, types)

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

Parameters

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

• weights (ndarray) – Matrix containing vectors with criteria weights in subsequent rows. Sum of weights in each vector must be equal to 1.

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

Returns

Matrix with vectors containing preference values of each alternative. The best alternative has the lowest preference value. Vectors are placed in subsequent columns of matrix.

Return type

ndarray

Examples

```python
>>> vikor = VIKOR(normalization_method = minmax_normalization)
>>> pref = vikor(matrix, weights, types)
>>> rank = np.zeros((pref.shape))
>>> for i in range(pref.shape[1]):
...     rank[:, i] = rank_preferences(pref[:, i], reverse = False)
```

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

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

__call__(self, matrix, weights, types)

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

Parameters

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

• weights (ndarray) – 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**(self, 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

**static _vikor_smaa**(self, matrix, weights, types, normalization_method, v)

**Submodules**

**objective_weighting.additions**

**Module Contents**

**Functions**

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

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

Return type
darray

Examples

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

**objective_weighting.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>
</tbody>
</table>

**objective_weighting.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)
```

**objective_weighting.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)
```

`objective_weighting.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)
```

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

**Returns**
Normalized decision matrix

**Return type**
`ndarray`

**Examples**

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

`objective_weighting.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**

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

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

Normalize decision matrix using maximum normalization method.

**Parameters**

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

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

**Returns**
Normalized decision matrix

**Return type**
`ndarray`

**Examples**

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

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

Normalize decision matrix using sum normalization method.

**Parameters**

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

• **types** (*ndarray*) – Criteria types. Profit criteria are represented by 1 and cost by -1.
**Objective-Weighting, Release 0.1**

### Returns
Normalized decision matrix

### Return Type
ndarray

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

**objective_weighting.normalizations.vector_normalization(matrix, types)**

Normalize decision matrix using vector normalization method.

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

#### Returns
Normalized decision matrix

#### Return Type
ndarray

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

**objective_weighting.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>

**objective_weighting.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)
```

**objective_weighting.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)
```

**objective_weighting.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)
```

**objective_weighting.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)
```

**objective_weighting.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)
```

**objective_weighting.weighting_methods.merec_weighting** *(matrix, types)*

Calculate criteria weights using objective MERECA 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)
```

**objective_weighting.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)
```

**objective_weighting.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)
```
PYTHON MODULE INDEX

O

objective_weighting, 28
objective_weighting.additions, 32
objective_weighting.correlations, 33
objective_weighting.mcda_methods, 28
objective_weighting.mcda_methods.mcda_method, 28
objective_weighting.mcda_methods.vikor, 29
objective_weighting.mcda_methods.vikor_smaa, 30
objective_weighting.normalizations, 34
objective_weighting.weighting_methods, 36
Symbols

__call__() (objective_weighting.mcda_methods.VIKOR method), 31
__call__() (objective_weighting.mcda_methods.VIKOR_SMAA method), 31
__call__() (objective_weighting.mcda_methods.MCDA_method), 28
__call__() (objective_weighting.mcda_methods.VIKOR method), 29
__call__() (objective_weighting.mcda_methods.VIKOR_SMAA method), 30
__call__() (objective_weighting.mcda_methods.vikor.VIKOR method), 31
__call__() (objective_weighting.mcda_methods.vikor_smaa.VIKOR_SMAA method), 32
__call__() (objective_weighting.mcda_methods.MCDA_method static method), 28
__call__() (objective_weighting.mcda_methods.vikor_smaa.VIKOR_SMAA method), 30
__call__() (objective_weighting.mcda_methods.mcda_method.MCDA_method static method), 29
__call__() (objective_weighting.mcda_methods.vikor_smaa.VIKOR_SMAA method), 32
__call__() (objective_weighting.mcda_methods.vikor_smaa.VIKOR_SMAA method), 30

_verify_input_data() (objective_weighting.mcda_methods.mcda_method.MCDA_method static method), 29
_verify_input_data() (objective_weighting.mcda_methods.vikor_smaa.VIKOR_SMAA method), 30

_verify_input_data() (objective_weighting.mcda_methods.mcda_method.MCDA_method static method), 29

_angle_weighting() (in module objective_weighting.weighting_methods), 40

_Cilos_weighting() (in module objective_weighting.weighting_methods), 39

_coeff_var_weighting() (in module objective_weighting.weighting_methods), 40

critic_weighting() (in module objective_weighting.weighting_methods), 38

Entrophy_weighting() (in module objective_weighting.weighting_methods), 37

_entrophy_weighting() (in module objective_weighting.weighting_methods), 37

_gini_weighting() (in module objective_weighting.weighting_methods), 38

idocriw_weighting() (in module objective_weighting.weighting_methods), 39

Linear_normalization() (in module objective_weighting.normalizations), 34

max_normalization() (in module objective_weighting.normalizations), 35

MCDA_method (class in objective_weighting.mcda_methods.mcda_method, 28

 Mercer_weighting() (in module objective_weighting.weighting_methods), 39

_minmax_normalization() (in module objective_weighting.normalizations), 35

module

objective_weighting, 28
objective_weighting.additions, 32
objective_weighting.correlations, 33
objective_weighting.mcda_methods, 28
objective_weighting.mcda_methods.mcda_method, 28
objective_weighting.mcda_methods.vikor, 28
objective_weighting.mcda_methods.vikor_smaa, 30
objective_weighting.normalizations, 34
objective_weighting.weighting_methods, 36
objective-weighting, Release 0.1

O

objective_weighting
  module, 28
objective_weighting.additions
  module, 32
objective_weighting.correlations
  module, 33
objective_weighting.mcda_methods
  module, 28
objective_weighting.mcda_methods.mcda_method
  module, 28
objective_weighting.mcda_methods.vikor
  module, 29
objective_weighting.mcda_methods.vikor_smaa
  module, 30
objective_weighting.normalizations
  module, 34
objective_weighting.weighting_methods
  module, 36

P

pearson_coeff() (in module objective_weighting.correlations), 34

R

rank_preferences() (in module objective_weighting.additions), 32

S

spearman() (in module objective_weighting.correlations), 33
stat_var_weighting() (in module objective_weighting.weighting_methods), 39
std_weighting() (in module objective_weighting.weighting_methods), 38
sum_normalization() (in module objective_weighting.normalizations), 35

V

vector_normalization() (in module objective_weighting.normalizations), 36
VIKOR (class in objective_weighting.mcda_methods), 31
VIKOR (class in objective_weighting.mcda_methods.vikor), 29
VIKOR_SMAA (class in objective_weighting.mcda_methods), 31
VIKOR_SMAA (class in objective_weighting.mcda_methods.vikor_smaa), 30

W

weighted_spearman() (in module objective_weighting.correlations), 33