Embedded Voting Documentation

Release 0.1.7

Anonymous Authors

Feb 14, 2023

Contents:

CHAPTER 1

Embedded Voting

This contains the code for our work on embedded voting.

- Free software: GNU General Public License v3
- Documentation: https://embedded-voting.readthedocs.io.

1.1 Features

- Create a voting profile in which voters are associated to embeddings.
- Run elections on these profiles with different rules, using the geometrical aspects of the embeddings.
- The rules are defined for cardinal preferences, but some of them are adapted for the case of ordinal preferences.
- There are rules for single-winner elections and multi-winner elections.
- Classes to analyse the evolution of the score when the embeddings of one voter are changing.
- Classes to analyse the manipulability of the rules.
- Classes for algorithm aggregation.
- A lot of tutorials.

1.2 Credits

This package was created with Cookiecutter and the francois-durand/package_helper project template.

CHAPTER 2

Installation

2.1 Stable release

To install Embedded Voting, run this command in your terminal:

```
$ pip install embedded_voting
```

This is the preferred method to install Embedded Voting, as it will always install the most recent stable release.

If you don't have pip installed, this Python installation guide can guide you through the process.

2.2 From sources

The sources for Embedded Voting can be downloaded from the Github repo.

You can either clone the public repository:

```
$ git clone git://github.com/TheoDlmz/embedded_voting
```

Or download the tarball:

```
$ curl -OJL https://github.com/TheoDlmz/embedded_voting/tarball/master
```

Once you have a copy of the source, you can install it with:

```
$ python setup.py install
```

CHAPTER 3

Usage

To use Embedded Voting in a project:

```
import embedded_voting as ev
```

The following notebook will help you to get started with the library.

- Fast tutorial

For more details, the following series of notebooks will guide you through the different aspects of the library:

- 1. My first Profile

This notebook cover the creation of a profile of voters with embeddings, and details the different functions you can use to build and display this profile.

- 2. Run an election
- 3. Analysis of the voting rules

In these notebooks, you will learn how to run a single-winner election on a profile and what are the different scoring rules you can use.

- 4. Ordinal preferences

This notebook shows how you can convert a cardinal voting profile into an ordinal voting profile by combining ordinal voting rule like Plurality and Borda with our rules.

- 5. Manipulability analysis

This notebook show how you can explore the question of the manipulability of the different rules and their ordinal extensions.

- 6. Multi-winner elections

In this notebook, you will learn how to run a multi-winner election on a profile of voters.

- 7. Algorithms aggregation

Finally, this notebook show how profile with embedded voters can be used for the aggregation of decision algorithms.

CHAPTER 4

Tutorials

4.1 Fast Tutorial

This notebook explains how to use the *embedded_voting* package in the context of epistemic social choice and algorithms aggregations.

In general algorithm aggregation rules (Average, Median, Likelihood maximization), you need diversity among the different algorithms. However, in the real world, it is not rare to have a large group of very correlated algorithms, which are trained on the same datasets, or which have the same structure, and give very similar answers. This can biais the results.

With this method, you don't suffer from this correlations between algorithms. This notebook simply explains how to use this method.

First of all, you need to import the package:

[1]:
```
import embedded_voting as ev
```

4.1.1 Generator to simulate algorithm results

Then, if you want to aggregate algorithms' outputs, you need to know the outputs of these algorithms. In this notebook, we will use a score generator that simulates a set of algorithms with dependencies.

In the following cell, we create a set of algorithms with 25 algorithms in the first group, 7 in the second group and 3 isolated algorithms.

[2]:
```
groups_sizes = [25, 7, 1, 1, 1]
features = [[1, 0, 0, 1], [0, 1, 0, 0], [1, 0, 1, 0], [0, 1, 0, 1], [0, 0, 1, 0]]

generator = ev.RatingsGeneratorEpistemicGroupsMix(groups_sizes,
                                                  features,
                                                  group_noise=8,
                                                  independent_noise=.5)
```

(continues on next page)

(continued from previous page)

```
ratings = generator(n_candidates=20)
true_ratings = generator.ground_truth_
print(ratings.shape)
```

```
(35, 20)
```

The last command generates a matrix of scores that contain the outputs given by the algorithms to 20 inputs. If you use this method, you can provide the score matrix by putting your algorithms' results in a matrix of shape $n_{voters} \times n_{candidates}$.

4.1.2 Find the best alternative

Now, we can simply **create an *Aggregator* object** with the following line:

```
[3]: aggregator = ev.Aggregator()
```

The following cell show how to run a "election":

```
[4]: results = aggregator(ratings)
```

Then we can obtain the results like this:

```
[5]: print("Ranking :", results.ranking_)
     print("Winner :", results.winner_)
```

```
Ranking : [2, 11, 5, 13, 16, 7, 18, 0, 3, 6, 12, 14, 1, 19, 9, 10, 15, 8, 17, 4]
Winner : 2
```

You will probably keep using the same *Aggregator* for other elections with the same algorithms, like in the following cell:

```
[6]: for i in range(10):
         ratings = generator(20)
         print(f'Winner {i+1} : {aggregator(ratings).winner_}')
```

```
Winner 1 : 11
Winner 2 : 1
Winner 3 : 0
Winner 4 : 19
Winner 5 : 0
Winner 6 : 18
Winner 7 : 19
Winner 8 : 18
Winner 9 : 1
Winner 10 : 18
```

During each election, the *Aggregator* saves the scores given by the algorithms to know them better. However, it does not compute anything with this new data if it is not asked to do it.

Every now and then, you can retrain your *Aggregator* with these newest data. We advise to do it often where there is not a lot of training data and once you have done enough elections (typically, when you have shown as many candidates than you have algorithms), you don't need to do it a lot.

To train your *Aggregator* on the newest data, do the following:

[7]:
```
aggregator.train()
```

[7]:
```
<embedded_voting.aggregation.aggregator.Aggregator at 0x2b789c15518>
```

You can also train it before an election using the data from the election by doing this:

[8]:
```
results = aggregator(ratings, train=True)
```

For the first election of your aggregator, you do not need to specify that *train* is **True** because the aggregator always do a training step when it is created.

4.1.3 Fine-tune the aggregation rule

If you want to go further, you can change some aspects of the aggregation rule.

The first thing that you may want to change is the aggregation rule itself. The default one is *FastNash*, but you can try *FastLog*, *FastSum* or *FastMin*, which can give different results.

We advise to use *FastNash*, which shows stronger theoretical and experimental results.

[9]:
```
aggregator_log = ev.Aggregator(rule=ev.RuleFastLog())
aggregator_sum = ev.Aggregator(rule=ev.RuleFastSum())
aggregator_min = ev.Aggregator(rule=ev.RuleFastMin())
print("FastNash:", aggregator(ratings).ranking_)
print("FastLog:", aggregator_log(ratings).ranking_)
print("FastSum:", aggregator_sum(ratings).ranking_)
print("FastMin:", aggregator_min(ratings).ranking_)
```

```
FastNash: [18, 1, 0, 13, 15, 14, 2, 10, 11, 9, 19, 7, 3, 5, 12, 4, 8, 17, 6, 16]
FastLog: [18, 1, 0, 13, 15, 14, 10, 2, 11, 9, 19, 7, 3, 5, 12, 4, 8, 17, 6, 16]
FastSum: [18, 15, 1, 0, 13, 14, 11, 10, 9, 2, 7, 19, 3, 12, 5, 4, 17, 8, 6, 16]
FastMin: [18, 1, 0, 15, 13, 14, 11, 2, 10, 9, 19, 7, 3, 12, 5, 17, 8, 4, 6, 16]
```

You can also use the average rule:

[10]:
```
aggregator_avg = ev.Aggregator(rule=ev.RuleSumRatings())
results = aggregator_avg(ratings)
print(aggregator_avg(ratings).ranking_)
```

```
[18, 15, 1, 0, 13, 14, 11, 10, 9, 7, 2, 19, 3, 12, 17, 4, 5, 6, 16, 8]
```

You can also change the transformation of scores. The default one is the following :

$$f(s) = \sqrt{\frac{s}{||s||}}$$

But you can put any rule you want, like the identity function $f(s) = s$ if you want. In general, if you use a coherent score transformation, it will not change a lot the results.

[11]:
```
aggregator_id = ev.Aggregator(rule=ev.RuleFastNash(f=lambda x,y,z:x))
print(aggregator_id(ratings).ranking_)
```

```
[18, 1, 13, 0, 15, 14, 10, 2, 11, 9, 19, 7, 3, 5, 4, 12, 8, 17, 6, 16]
```

4.2 1. My first Profile

In this Notebook, I will explain how to create a **profile of voters with embeddings**.

[1]:
```
import embedded_voting as ev
import numpy as np
import matplotlib.pyplot as plt
```

4.2.1 Build a profile

Let's first create **a simple profile of ratings**, with $m = 5$ candidates and $n = 100$ voters:

[2]:
```
n_candidates = 5
n_voters = 100
profile = ev.Ratings(np.random.rand(n_voters,n_candidates))
profile.voter_ratings(0)
```

[2]:
```
array([0.9818839 , 0.50767343, 0.35742337, 0.4115904 , 0.51233348])
```

Here we created a profile with random ratings between 0 and 1. We could have used the *impartial culture* model for this :

[3]:
```
profile = ev.RatingsGeneratorUniform(n_voters)(n_candidates)
profile.voter_ratings(0)
```

[3]:
```
array([0.43593987, 0.95181078, 0.60167015, 0.42875782, 0.78548049])
```

We can also change the ratings afterwards, for instance by saying that the last 50 voters do not like the first 2 candidates :

[4]:
```
profile[50:,:2] = 0.1
profile.voter_ratings(50)
```

[4]:
```
array([0.1       , 0.1       , 0.97879522, 0.85108355, 0.82567096])
```

Now, we want to create embeddings for our voters. To do so, we create an **Embeddings** object:

[5]:
```
embs = ev.Embeddings(([[.9,0,.1],
                        [.8,.1,0],
                        [.1,.1,.9],
                        [0,.2,.8],
                        [0,1,0],
                        [.2,.3,.2],
                        [.5,.1,.9]]), norm=False)

embs.voter_embeddings(0)
```

[5]:
```
array([0.9, 0. , 0.1])
```

We can normalize the embeddings, so that each vector have norm 1:

[6]:
```
embs = embs.normalized()
embs.voter_embeddings(0)
```

[6]:
```
array([0.99388373, 0.        , 0.11043153])
```

You can also use an *Embedder* to generate embeddings from the ratings. The simplest one is the one generating the uniform distribution of embeddings :

[7]:
```
embedder = ev.EmbeddingsFromRatingsRandom(3)
embeddings = embedder(profile)
embeddings.voter_embeddings(0)
```

[7]:
```
array([0.76396325, 0.43197473, 0.47933077])
```

Let’s now create more complex embeddings for our profile

[8]:
```
positions = [[.8,.2,.2] + np.random.randn(3)*0.05 for _ in range(33)]
positions += [[.2,.8,.2] + np.random.randn(3)*0.05 for _ in range(33)]
positions += [[.2,.2,.8] + np.random.randn(3)*0.05 for _ in range(34)]
embs = ev.Embeddings(np.array(positions), norm=False)
```

There are several way to create embeddings, some of them using the ratings of the voters, but we will see it in another notebook.

4.2.2 Visualize the profile

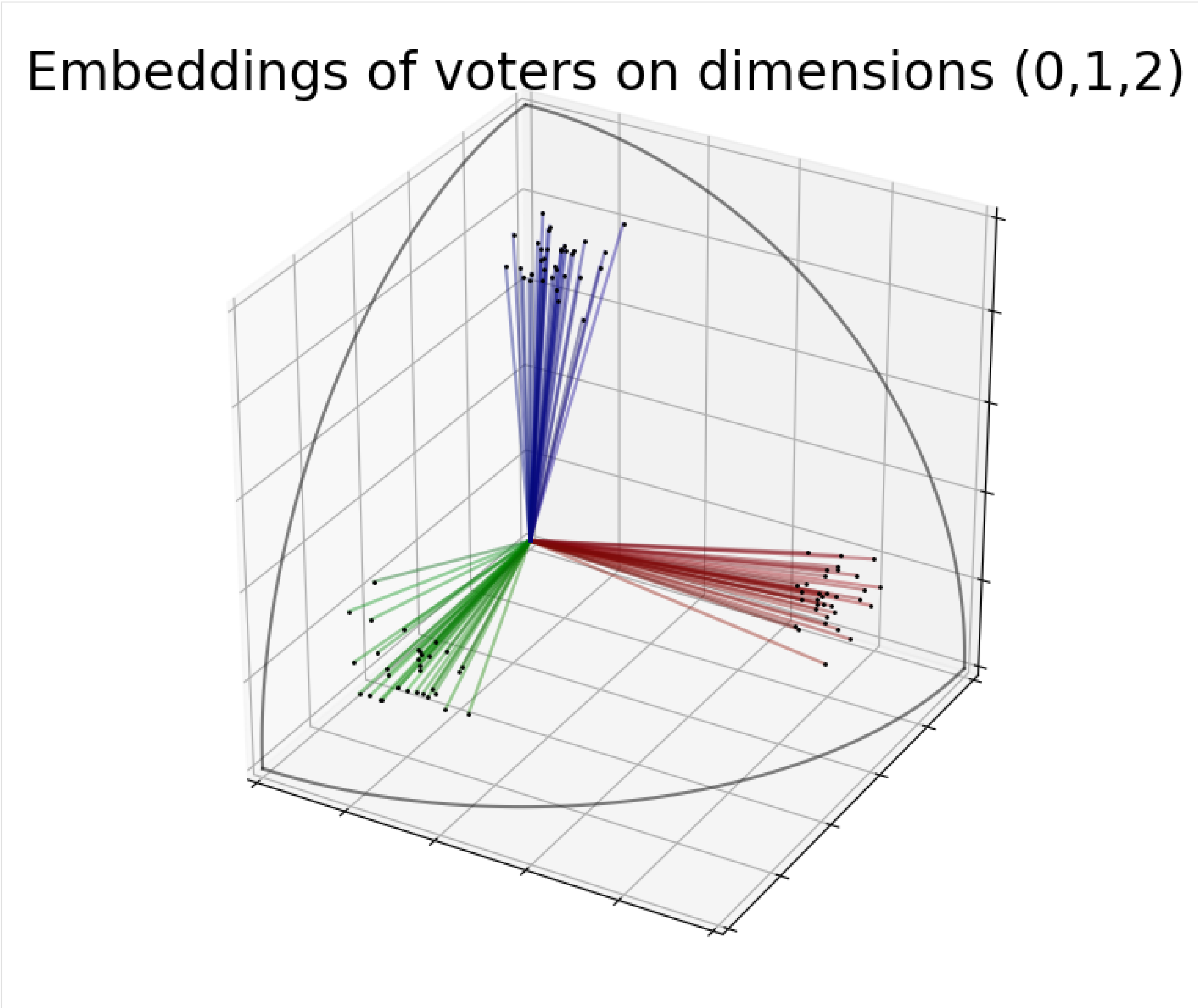
Now that we have a profile, we want to visualize it. Since the number of embeddings dimensions is only 3 in our profile, we can easily plot it on a figure.

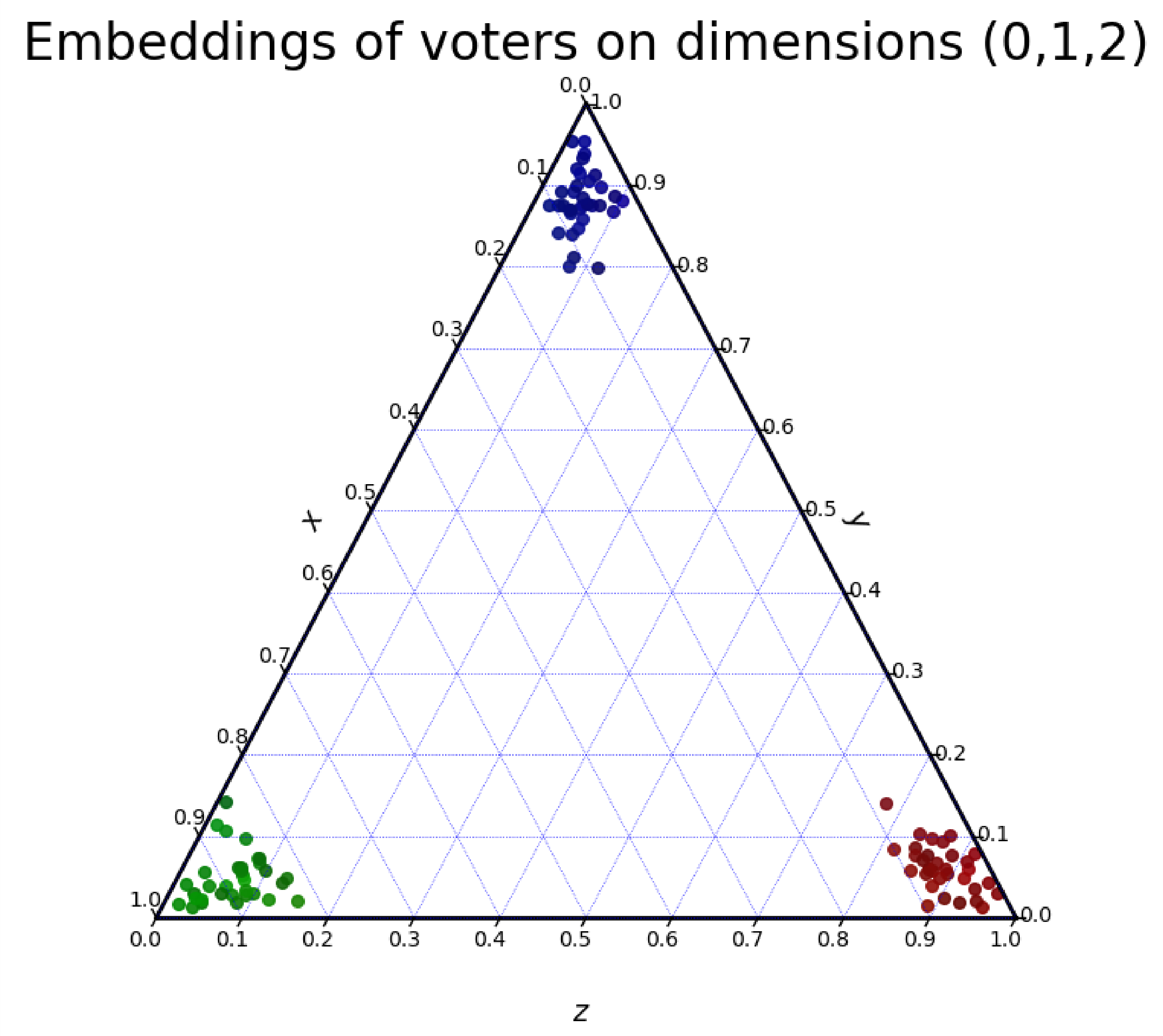
There are two ways of plotting your profile, using a *3D* plot or a *ternary* plot :

- On the **3D plot**, each voter is represented by a line from the origin to its position on the unit sphere.
- On the **ternary plot**, the surface of the unit sphere is represented as a 2D space and each voter is represented by a dot.

On the following figures we can see the **red group of voters**, which corresponds to the 25 voters with similar embeddings I added in *the fourth cell*.

[9]:
```
embs.plot("3D")
embs.plot("ternary")
```

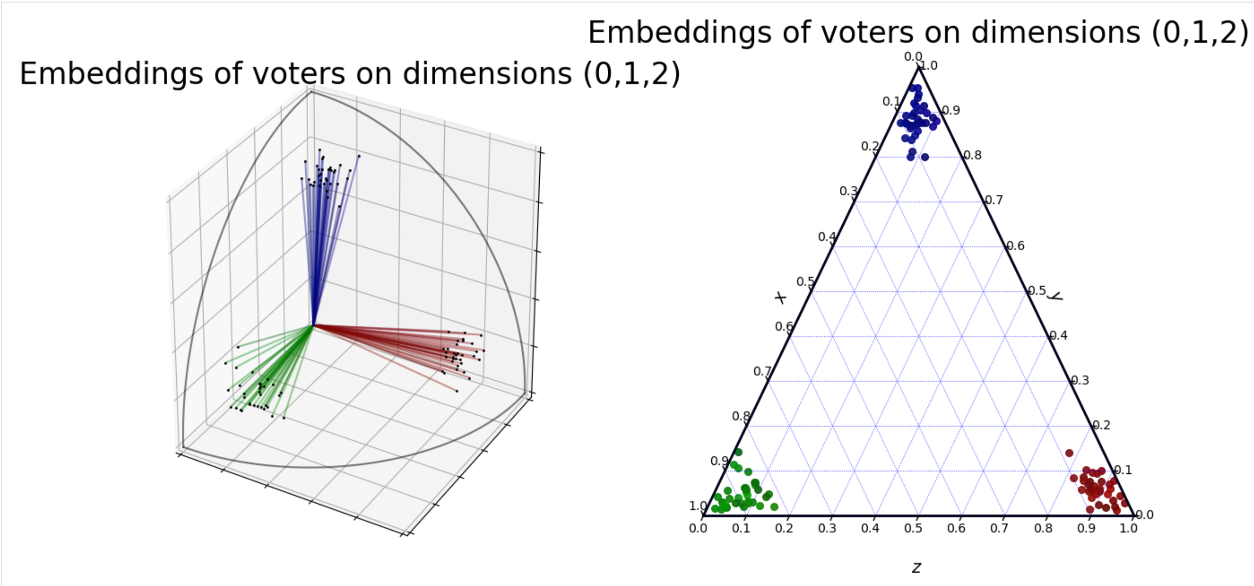

Embeddings of voters on dimensions (0,1,2)



[9]: TernaryAxesSubplot: -9223371919785834117

You can also plot the two figures **side by side** :

```
[10]: fig = plt.figure(figsize=(15,7.5))
embs.plot("3D", fig=fig, plot_position=[1,2,1], show=False)
embs.plot("ternary", fig=fig, plot_position=[1,2,2], show=False)
plt.show()
```



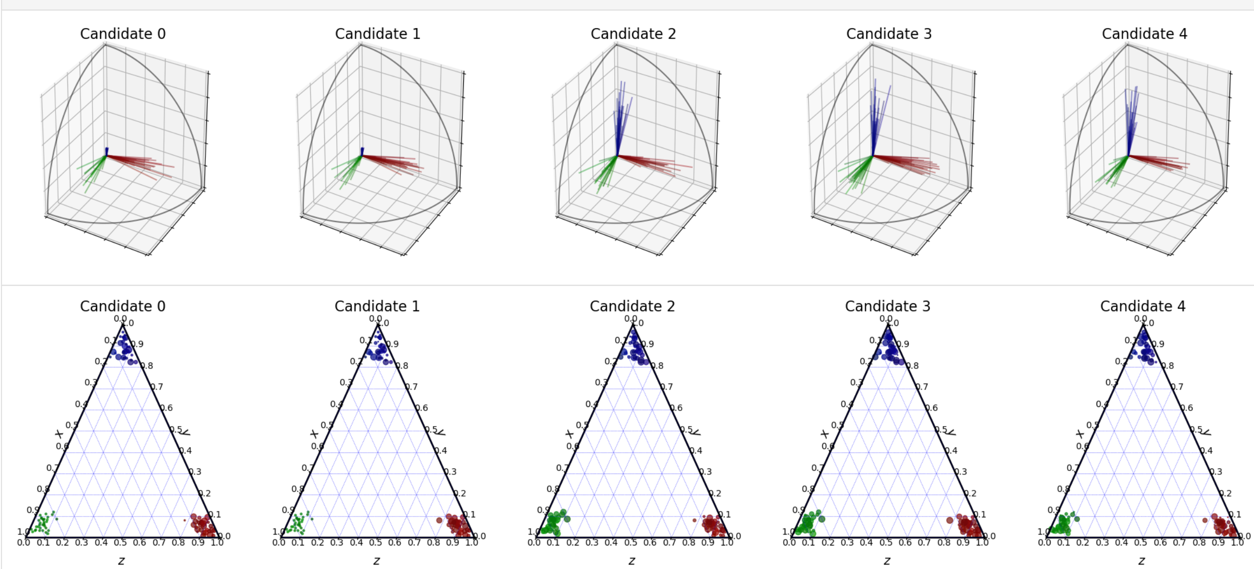
4.2.3 Visualize the candidates

With the same idea, you can visualize the **candidates**.

- On a **3D plot**, the score given by a voter to a candidate is represented by the **size of its vector**.
- On a **ternary plot**, the score given by a voter to a candidate is represented by the **size of the dot**.

Use plot_candidate to plot only **one** candidate and plot_candidates to plot **all** the candidates. In the following plots, we can see that the **blue group** don't like the first two candidates.

[11]:
```
embs.plot_candidates(profile, "3D")
embs.plot_candidates(profile, "ternary")
```



4.2.4 Beyond 3 dimensions

What if the profile has **more** than 3 dimensions?

We still want to visualize the profile and the candidates.

In the following cell, we create a profile with **4 dimensions**.

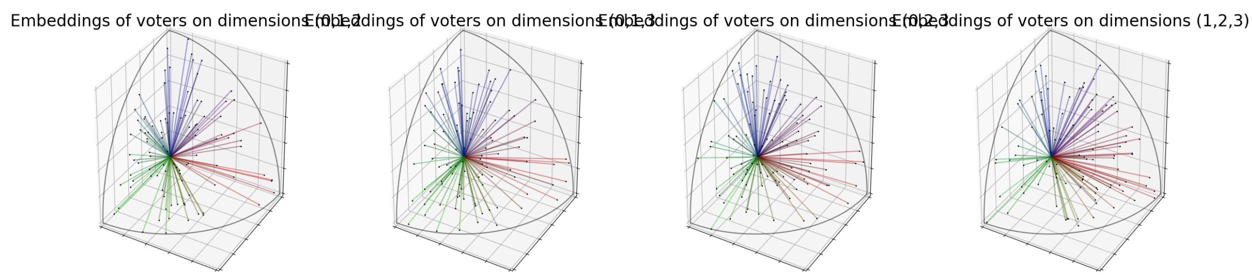
[12]:
```
embs = ev.EmbeddingsFromRatingsRandom(4)(profile).normalized()
```

We use the functions described above and specify **which dimensions** to use on the plots (we need exactly 3 dimensions).

By default, the function uses the first three dimensions.

In the following cell, we show the distribution of voters with different subsets of the 4 possible dimensions.

[13]:
```
fig = plt.figure(figsize=(30,7.5))
embs.plot("3D", dim=[0,1,2], fig=fig, plot_position=[1,4,1], show=False)
embs.plot("3D", dim=[0,1,3], fig=fig, plot_position=[1,4,2], show=False)
embs.plot("3D", dim=[0,2,3], fig=fig, plot_position=[1,4,3], show=False)
embs.plot("3D", dim=[1,2,3], fig=fig, plot_position=[1,4,4], show=False)
plt.show()
```



4.2.5 Recenter and dilate a profile

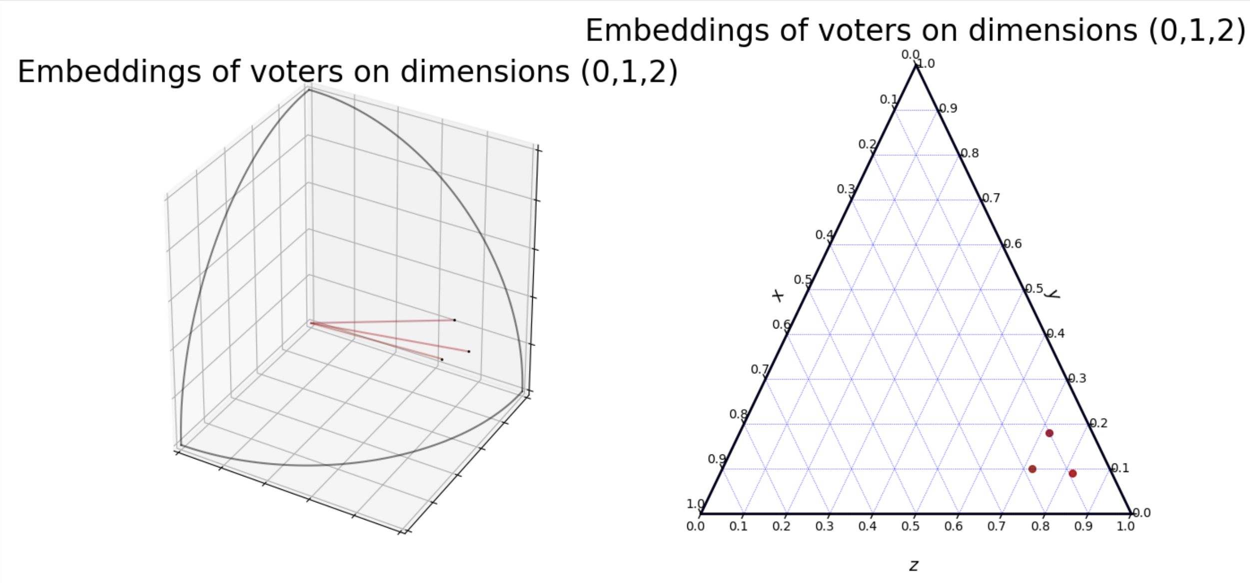
Sometimes the voters' embeddings are really close one to another and it is hard to do anything with the profile, because it looks like every voter is the same.

For instance, we can create three groups of voters with **very similar embeddings** :

[14]:
```
embeddings = ev.Embeddings([[.9,.3,.3],[.8,.4,.3],[.8,.3,.4]], norm=True)
```

If I plot this profile, the three voters are really close to each other:

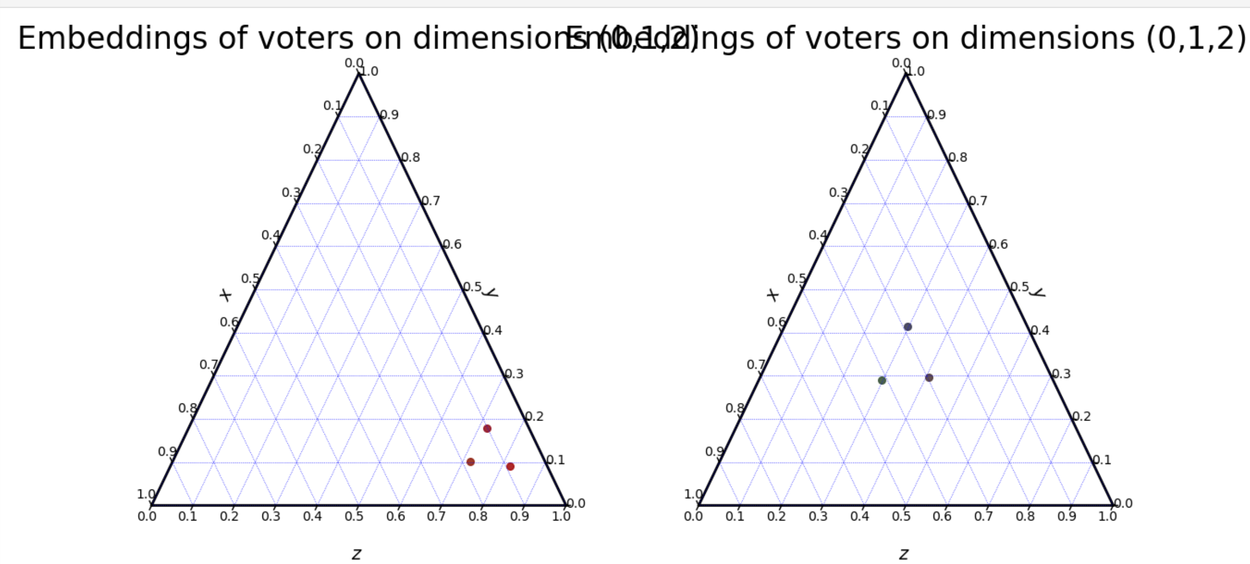
[15]:
```
fig = plt.figure(figsize=(15,7.5))
embeddings.plot("3D", fig=fig, plot_position=[1,2,1], show=False)
embeddings.plot("ternary", fig=fig, plot_position=[1,2,2], show=False)
plt.show()
```



The first thing we can do is **to recenter** the population of voters:

[16]:
```
embeddings_optimized = embeddings.recentered(False)
```

[17]:
```
fig = plt.figure(figsize=(14,7))
embeddings.plot("ternary", fig=fig, plot_position=[1,2,1], show=False)
embeddings_optimized.plot("ternary", fig=fig, plot_position=[1,2,2], show=False)
plt.show()
```



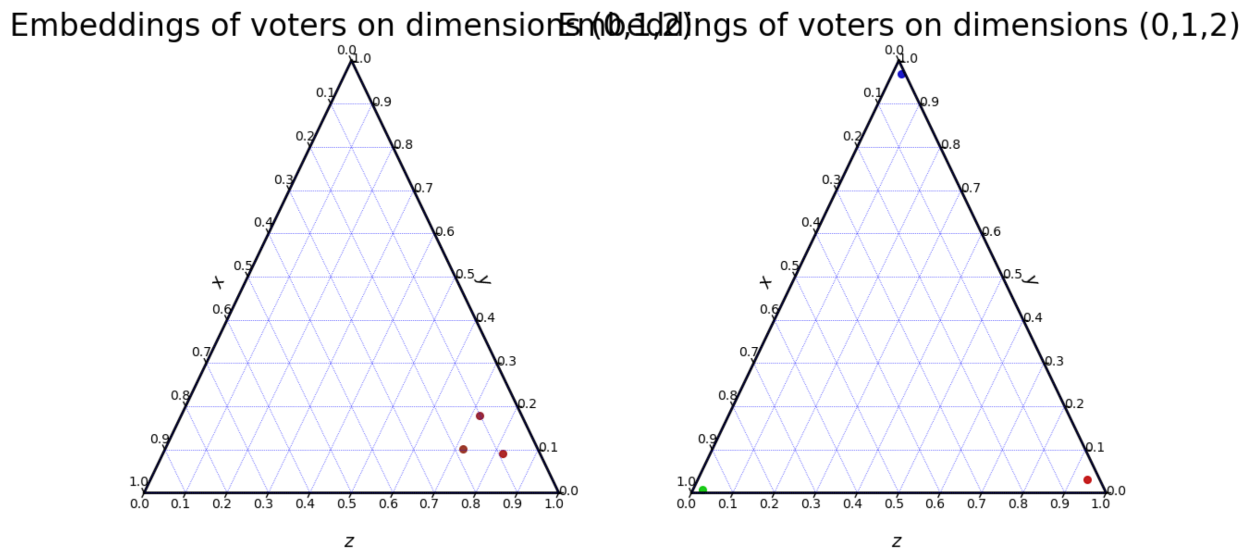
Now, we can **dilate** the profile in such a way that the **relative distance** between each pair of voters remains the same, but they **take all the space they can** on the non-negative orthant.

To do so, we use the funtion dilated.

[18]:
```
embeddings_optimized = embeddings_optimized.dilated(approx=False)
```

As you can see on the second plot, voters are **pushed to the extreme positions** of the non-negative orthant.

```
[19]: fig = plt.figure(figsize=(14,7))
embeddings.plot("ternary", fig=fig, plot_position=[1,2,1], show=False)
embeddings_optimized.plot("ternary", fig=fig, plot_position=[1,2,2], show=False)
plt.show()
```



4.2.6 Introduction to parametric profile generator

Our package also proposes an easy way to build a profile with *"groups"* of voters **who have similar embeddings and preferences**.

To do so, we need to specify :

- The number of **candidates**, **dimensions**, and **voters** in the profile.
- The matrix M of **the scores of each "group"**. $M(i, j)$ is the score given by the group j to the candidate i.
- The **proportion** of the voters in each group.

For instance, in the following cell, I am building a profile of 100 voters in 3 dimensions, with 5 candidates. There are 3 groups in this profile :

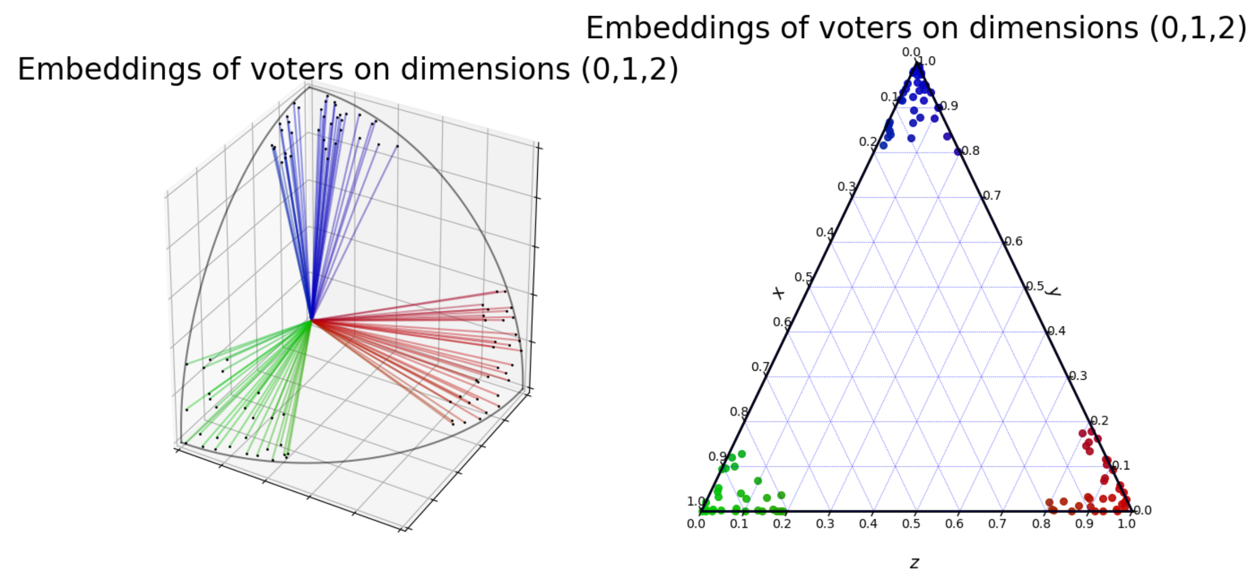
- The **red group**, with 50% of the voters. Voters from this group have preferences close to $c_0 > c_1 > c_2 > c_3 > c_4$.
- The **green group**, with 30% of the voters. Voters from this group have preferences close to $c_1 \sim c_3 > c_0 \sim c_2 \sim c_4$.
- The **blue group**, with 20% of the voters. Voters from this group have preferences close to $c_4 > c_3 > c_2 > c_1 > c_0$.

```
[20]: scores_matrix = np.array([[1, .7, .5, .3, 0], [.2, .8, .2, .8, .2], [0, .3, .5, .7,␣
↪1]])
proba = [.5, .3, .2]
n_voters = 100
n_dimensions, n_candidates = np.array(scores_matrix).shape
embeddingsGenerator = ev.EmbeddingsGeneratorPolarized(n_voters, n_dimensions, proba)
ratingsGenerator = ev.RatingsFromEmbeddingsCorrelated(0, scores_matrix, n_dimensions,␣
↪n_candidates)
```

Then, we need to specify the **level of polarisation** of the profile.

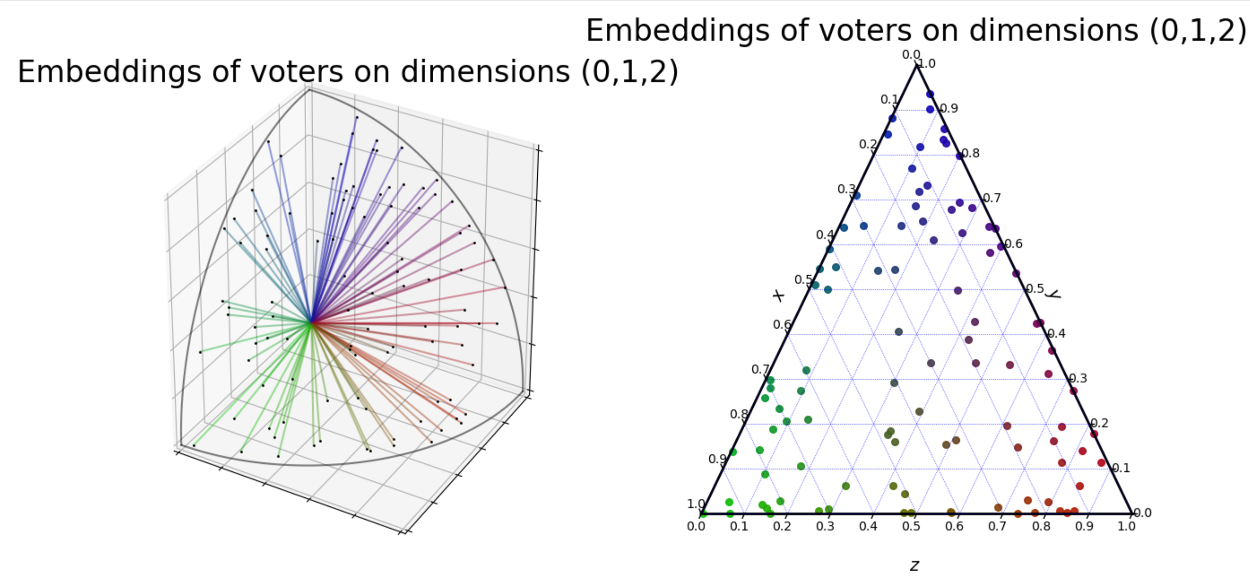
A *high* level of polarisation (> 0.5) means that voters in the different groups are aligned with the dimension of each group. Therefore, there embeddings are really similar.

[21]:
```
embeddings = embeddingsGenerator(polarisation=0.7)

fig = plt.figure(figsize=(15,7.5))
embeddings.plot("3D", fig=fig, plot_position=[1,2,1], show=False)
embeddings.plot("ternary", fig=fig, plot_position=[1,2,2], show=False)
plt.show()
```



On the opposite, if the level of polarisation is *low* (< 0.5), then voters' embeddings are more random.

[22]:
```
embeddings = embeddingsGenerator(polarisation=0.2)

fig = plt.figure(figsize=(15,7.5))
embeddings.plot("3D", fig=fig, plot_position=[1,2,1], show=False)
embeddings.plot("ternary", fig=fig, plot_position=[1,2,2], show=False)
plt.show()
```

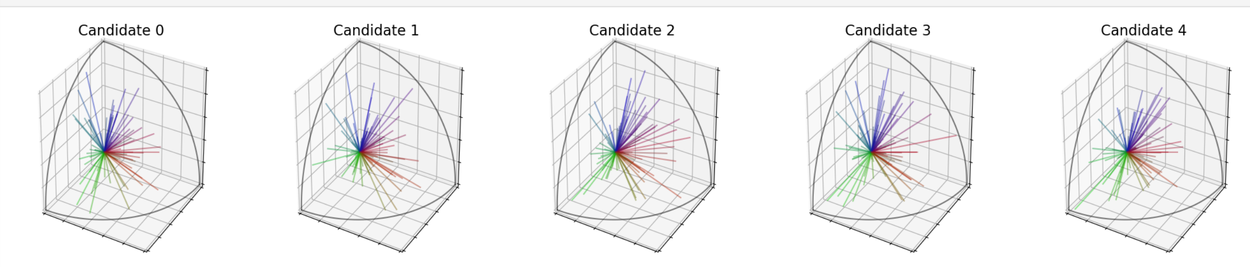


The second important parameter is **coherence**.

The coherence parameter characterizes the **correlation** between the embeddings of the voters and the score they give to the candidates. If this parameter is set to 1, then the scores of a group **dictate** the scores of the voters in this group.

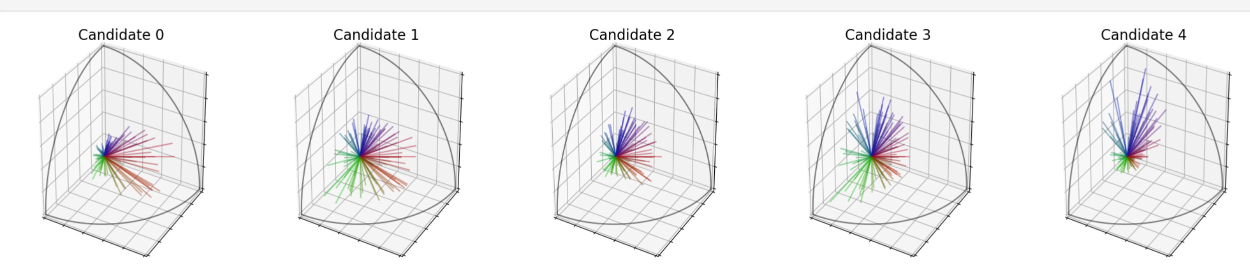
By default, it is set to 0, which means that the scores are **totally random** and there is no correlation between the embeddings and the scores.

[23]:
```
profile = ratingsGenerator(embeddings)
embeddings.plot_candidates(profile)
```



In the following cell, we can see that a *high* coherence implies that embeddings and scores are very correlated.

[24]:
```
ratingsGenerator = ev.RatingsFromEmbeddingsCorrelated(0.8, scores_matrix, n_
↪dimensions, n_candidates)
profile = ratingsGenerator(embeddings)
embeddings.plot_candidates(profile)
```



4.3 2. Run an election

In this notebook, I will explain **how to run a single-winner election** on a voting profile with embedded voters.

[1]:
```
import numpy as np
import embedded_voting as ev
import matplotlib.pyplot as plt

np.random.seed(42)
```

4.3.1 Creating the profile

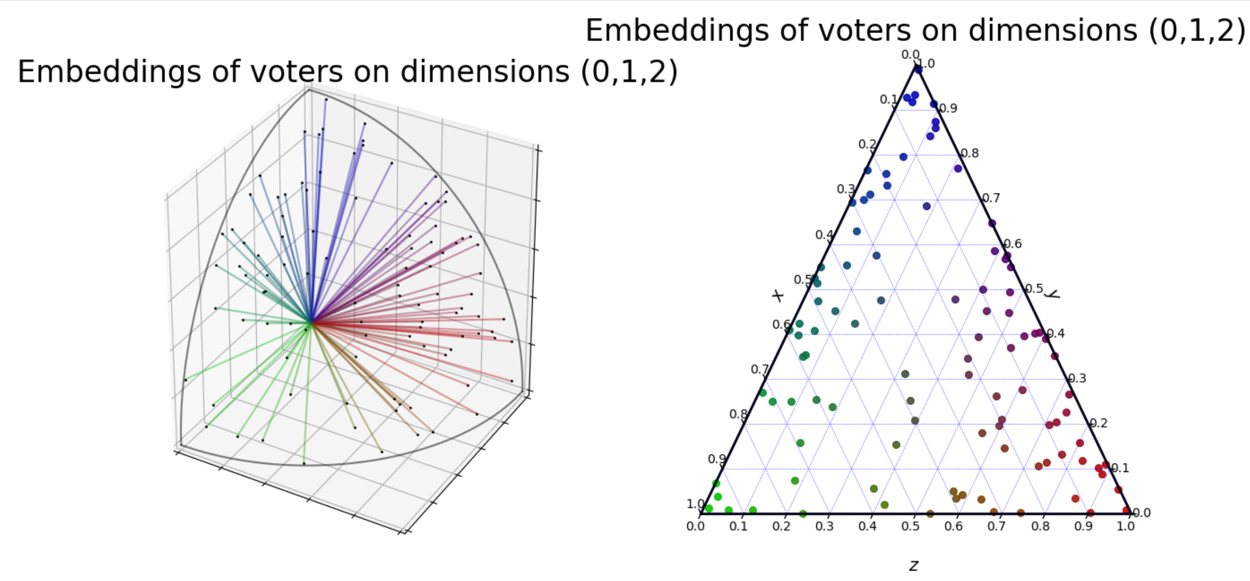
Let's say we have **5 candidates** and **3 groups** of voters:

- The **red group** contains **50%** of the voters, and the average scores of candidates given by this group are $[0.9, 0.3, 0.5, 0.2, 0.2]$.
- The **green group** contains **25%** of the voters, and the average scores of candidates given by this group are $[0.2, 0.6, 0.5, 0.5, 0.8]$.
- The **blue group** contains **25%** of the voters, and the average scores of candidates given by this group are $[0.2, 0.6, 0.5, 0.8, 0.5]$.

[2]:
```
scores_matrix = np.array([[.9, .3, .5, .3, .2], [.2, .6, .5, .5, .8], [.2, .6, .5, .8,
↪ .5]])
proba = [.5, .25, .25]

n_voters = 100
n_dimensions, n_candidates = np.array(scores_matrix).shape
embeddingsGen = ev.EmbeddingsGeneratorPolarized(n_voters, n_dimensions, proba)
ratingsGen = ev.RatingsFromEmbeddingsCorrelated(0.8,  scores_matrix, n_dimensions, n_
↪candidates)

embeddings = embeddingsGen(polarisation=0.4)
profile = ratingsGen(embeddings)
```

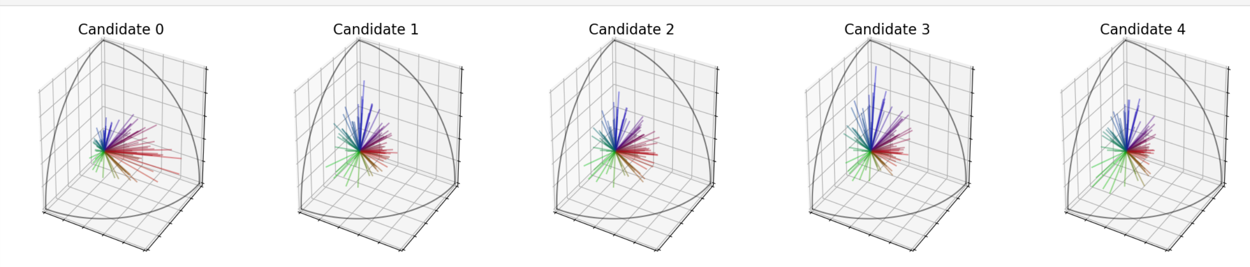
We can **visualize** this profile, as explained in the previous notebook:

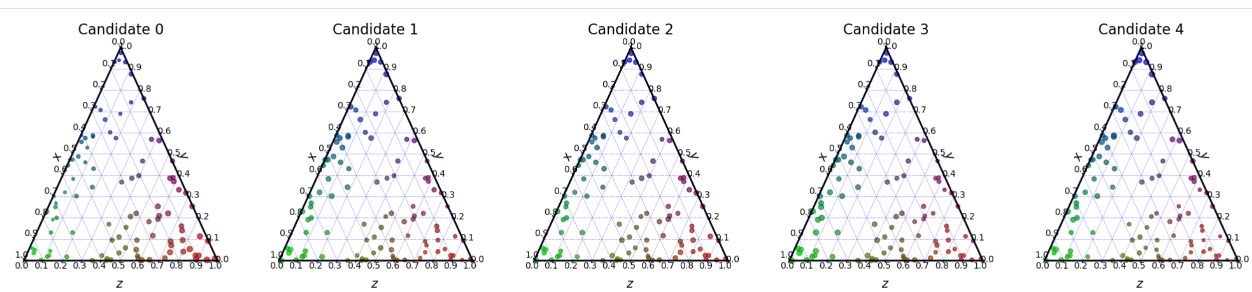
[3]:
```
fig = plt.figure(figsize=(15,7.5))
embeddings.plot("3D", fig=fig, plot_position=[1,2,1], show=False)
embeddings.plot("ternary", fig=fig, plot_position=[1,2,2], show=False)
plt.show()
```



We can also **visualize the candidates**. Each voter is represented by a line and the length of the line represents the score the voter gives to the candidate.

[4]:
```
embeddings.plot_candidates(profile, "3D")
embeddings.plot_candidates(profile, "ternary")
```





Now, we want to determine the **best candidate**. Is it the *Candidate 0*, which is loved by the majority group? Or is it the *Candidate 2*, which is not hated by any group?

To decide that, we can use a whole set of voting rules. First, there is the **simple rules**, which are not based on the embeddings. These rules are *Range voting* (we take the average score) and *Nash voting* (we take the product of the score, or the average log score).

4.3.2 Notations

In the next parts of this notebook, I will use some notations:

Notation	Meaning	Function
v_i	The i^{th} voter	
c_j	The j^{th} candidate	
$s_i(c_j)$	The score given by the voter v_i to the candidate c_j	Profile.scores[i,j]
$S(c_j)$	The score of the candidate c_j after the aggregation	ScoringRule.scores_[j]
$w(c_j)$	The welfare of the candidate c_j	ScoringRule.welfare_[j]
M	The embeddings matrix, such that M_i are the embeddings of v_i	Profile.embeddings
$M(c_j)$	The candidate matrix, such that $M(c_j)_i = s_i(c_j) \times M_i$	Profile.scored_embeddings(j)
$s^*(c_j)$	The vector of score of one candidate, such that $s^*(c_j)_i = s_i(c_j)$	Profile.scores[:,j]

4.3.3 Simple rules

Average score *(Range Voting)*

This is the **most intuitive rule** when we need to aggregate the score of the different voters to establish a ranking of the candidate. We simply take the sum of every vote given to this candidate:

$$\forall j, S(c_j) = \sum_i s_i(c_j)$$

We create the election in the following cell.

```
[5]: election = ev.RuleSumRatings()
```

We then **run** the election like this

```
[6]: election(profile, embeddings)
```

```
[6]: <embedded_voting.rules.singlewinner_rules.rule_sum_ratings.RuleSumRatings at␣
↪0x1e2e5614710>
```

Then, we can compute the **score** of every candidate, the **ranking**, and of course the **winner** of the election:

```
[7]: print('Scores : ', election.scores_)
print('Ranking : ', election.ranking_)
print('Winner : ', election.winner_)
```

```
Scores :  [45.77999664207862, 48.88823373691184, 49.96870542210909, 52.26932370998465,
↪ 47.86098615045205]
Ranking :  [3, 2, 1, 4, 0]
Winner :  3
```

We can also compute **the welfare** of each candidate, where the welfare is defined as:

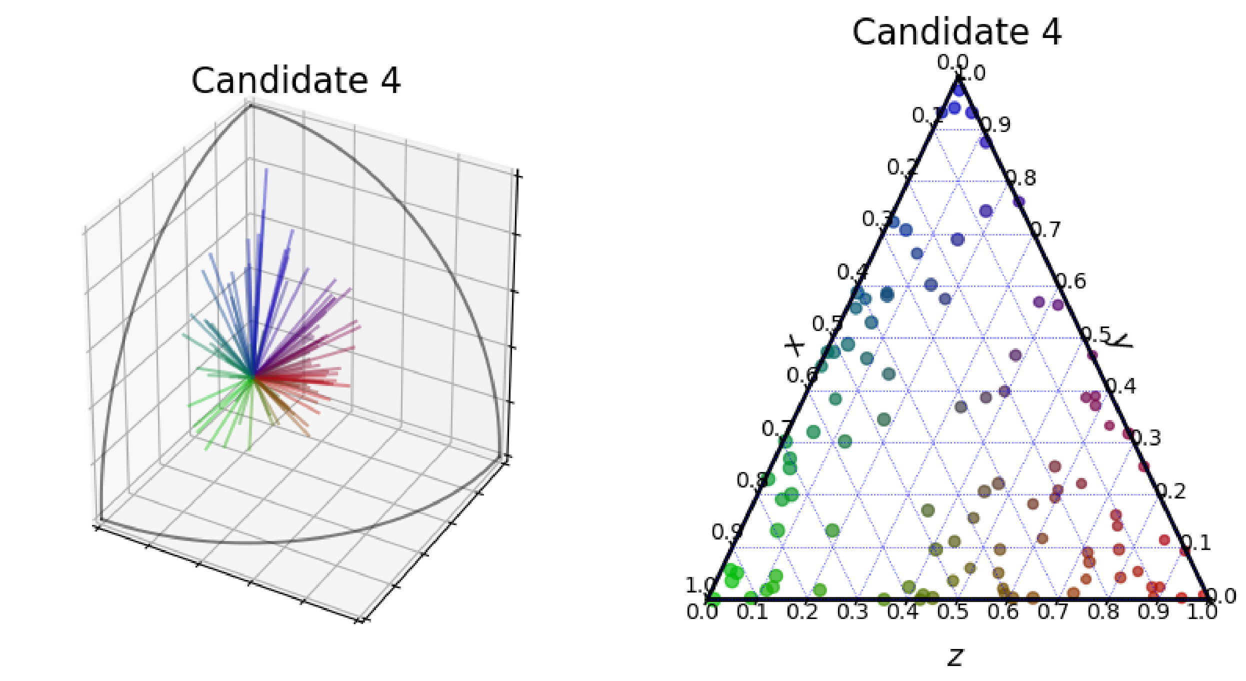
$$w(c_j) = \frac{S(c_j) - \min_c S(c)}{\max_c S(c) - \min_c S(c)}$$

```
[8]: print('Welfare : ', election.welfare_)
```

```
Welfare :  [0.0, 0.4789767971790928, 0.6454766012251681, 1.0, 0.3206787832694216]
```

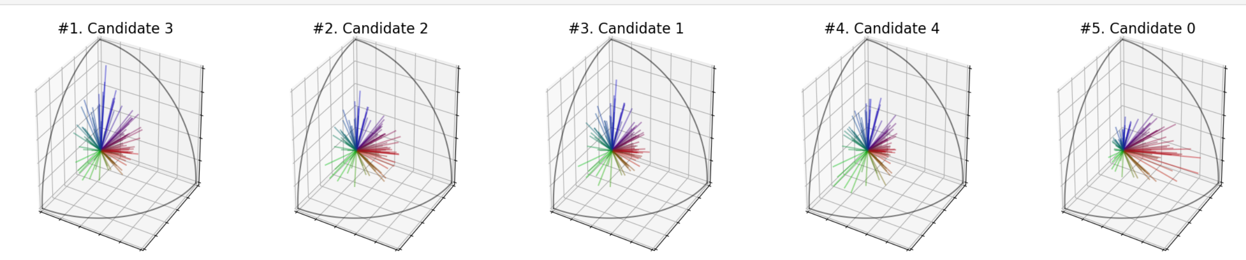
We can **plot the winner** of the election using the function *plot_winner()*.

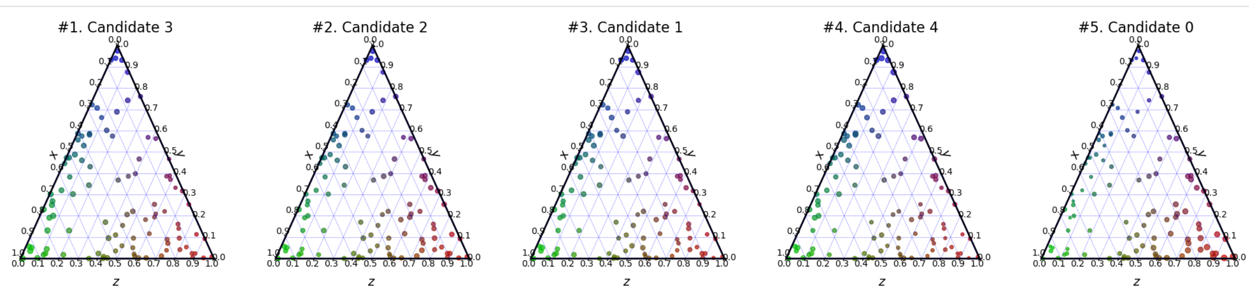
[9]:
```
fig = plt.figure(figsize=(10,5))
election.plot_winner("3D", fig=fig, plot_position=[1,2,1], show=False)
election.plot_winner("ternary", fig=fig, plot_position=[1,2,2], show=False)
plt.show()
```



We can **plot the ranking** of the election using the function *plot_ranking()*.

[10]:
```
election.plot_ranking("3D")
election.plot_ranking("ternary")
```





Product of scores (Nash)

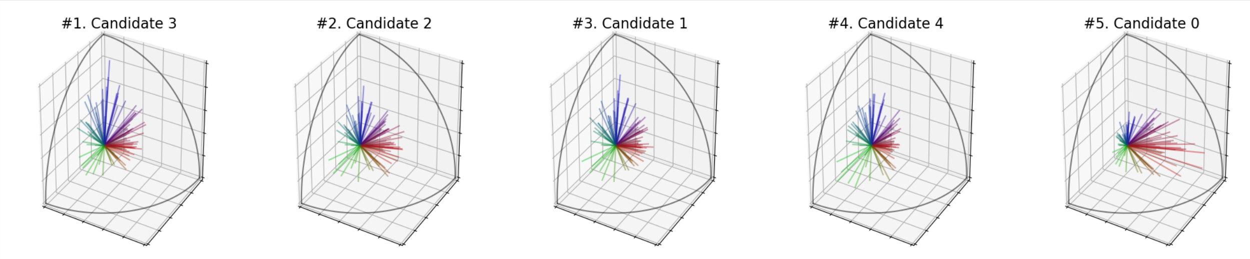
The second intuitive rule is **the product of the scores**, also called *Nash welfare*. It is equivalent to the sum of the log of the scores.

We have

$$S(c_j) = \prod_i s_i(c_j) = e^{\sum_i \log(s_i(c_j))}$$

[12]:
```
election = ev.RuleShiftProduct()
election(profile, embeddings)
print('Scores : ', election.scores_)
print('Ranking : ', election.ranking_)
print('Winner : ', election.winner_)
election.plot_ranking("3D")
```

```
Scores :  [8.951032542639148e+38, 3.715630226275884e+39, 5.989493011777882e+39, 1.
↪3845840911371833e+40, 2.3053960275427698e+39]
Ranking :  [3, 2, 1, 4, 0]
Winner :  3
```



You probably noticed that scores are **composed of two elements** (e.g *(100, 5.393919173647501e-37)*). In this particular case, the first element is the number of non-zero individual scores and the second one is the product of the non-zero scores. Indeed, if some voter gives a score of 0 to every candidate, the product of scores will be 0 for every candidate and we cannot establish a ranking.

We use similar ideas for some of the rules that will come later.

In these cases, if we have $S(c_j) = (S(c_j)_1, S(c_j)_2)$, the score used in the welfare is :

$$S'(c_j) = \begin{cases} S(c_j)_2 & \text{if } S(c_j)_1 = \max_c S(c)_1 \\ 0 & \text{Otherwise} \end{cases}$$

In other word, the welfare is > 0 if and only if the first component of the score is at the maximum.

[13]:
```
print("Welfare : ",election.welfare_)
```

```
Welfare :  [0.0, 0.2177889049017948, 0.3933667635308749, 1.0, 0.1088967138875542]
```

4.3.4 Geometrical rules

All the rules that I will describe now are **using the embeddings** of the voters. Some of them are purely **geometrical**, other are more **algebraic**. Let's start with the geometrical ones.

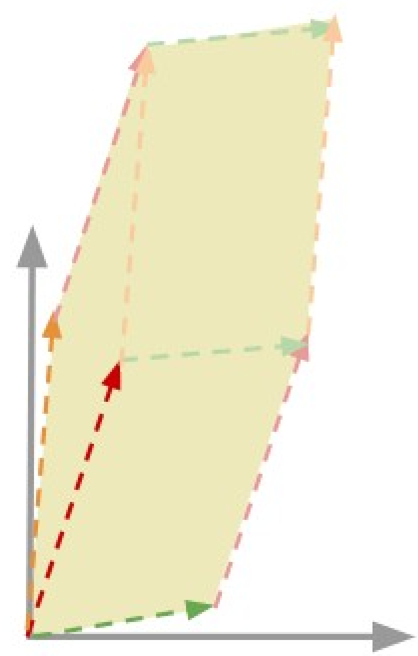
All of the rules presented here **do not depend on the basis** used for the embeddings. Indeed, the result will not change if you change the vector basis of the embeddings (for instance, by doing a **rotation**)

Zonotope

The zonotope of a set of vectors $V = \{\vec{v_1}, \ldots, \vec{v_n}\}$ is the geometrical object defined as $\mathcal{Z}(V) = \{\sum_i t_i \vec{v_i} | \forall i, t_i \in [0, 1]\}$.

For a matrix M, we have $\mathcal{Z}(M) = \mathcal{Z}(\{M_1, \ldots, M_n\})$.

The following figure illustrate the zonotope in 2D for a set of 3 vectors.



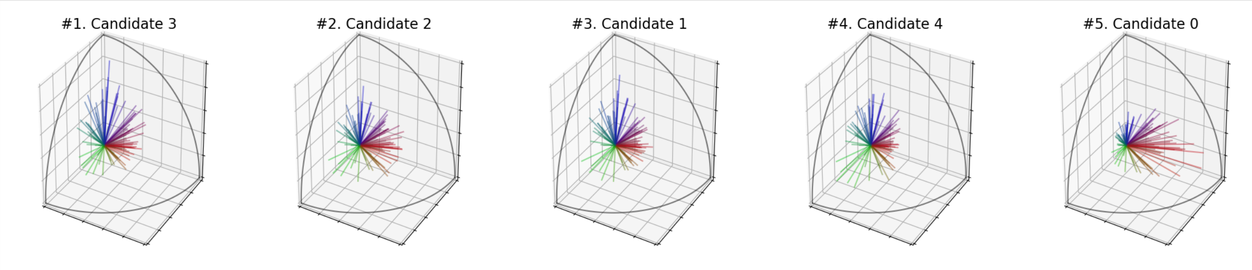
In the case of an election, the score of the candidate c_j is defined as the volume of the Zonotope defined by the rows of the matrix $M(c_j)$:

$$S(c_j) = \text{vol}(\mathcal{Z}(M(c_j)))$$

There is a simple formula to compute this volume, but it is **exponential in the number of dimensions**.

[14]:
```
election = ev.RuleZonotope()
election(profile, embeddings)
print('Scores : ', election.scores_)
print('Ranking : ', election.ranking_)
print('Winner : ', election.winner_)
print('Welfare : ', election.welfare_)
election.plot_ranking("3D")
```

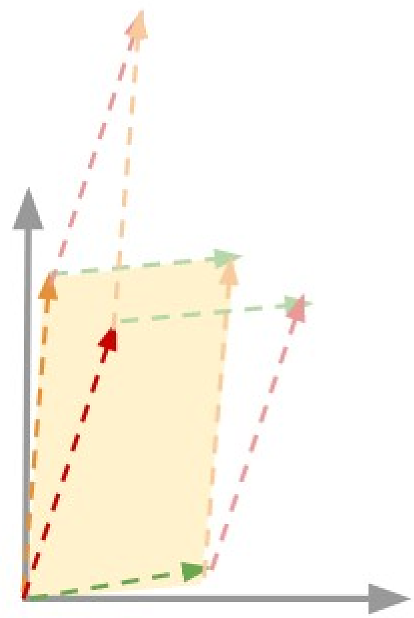
```
Scores :  [(3, 2512.3852946433435), (3, 3478.5303669376613), (3, 3724.291234894805),
↪(3, 4197.7832046975145), (3, 3317.721702753313)]
Ranking :  [3, 2, 1, 4, 0]
Winner :  3
Welfare :  [0.0, 0.5732444940929495, 0.7190622066290026, 1.0, 0.47783161667981705]
```



Maximum Parallelepiped

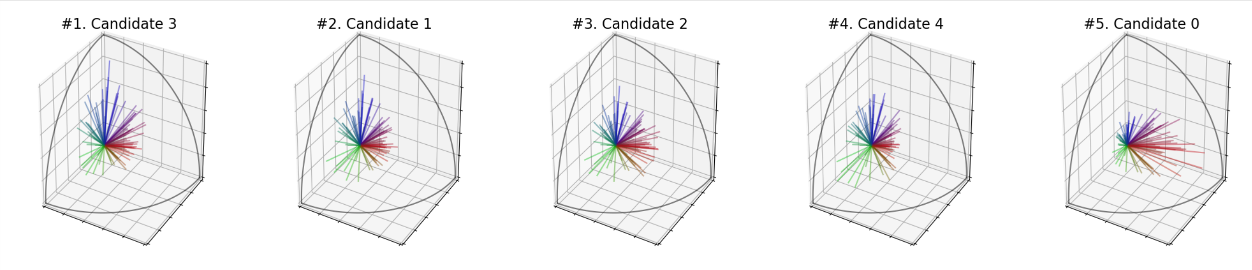
The maxcube rule is also very geometrical. It computes **the maximum volume** spanned by a linearly independent subset of rows of the matrix $M(c_j)$.

The figure below shows an example of how it works.



[15]:
```
election = ev.RuleMaxParallelepiped()
election(profile, embeddings)
print('Scores : ', election.scores_)
print('Ranking : ', election.ranking_)
print('Winner : ', election.winner_)
print('Welfare : ', election.welfare_)
election.plot_ranking("3D")
```

```
Scores :  [(3, 0.1288637458094931), (3, 0.1665379237215778), (3, 0.15408795064264366),
↪ (3, 0.1825593712490694), (3, 0.15152969628703225)]
Ranking :  [3, 1, 2, 4, 0]
Winner :  3
Welfare :  [0.0, 0.701624715303474, 0.46976275304839044, 1.0, 0.42211912594341866]
```



SVD Based rules

Here are some of the most interesting rules you can use for embedded voters. They are based on the **Singular Values Decomposition (SVD)** of the matrices $M(c_j)$.

Indeed, if we denote $(\sigma_1(c_j), \dots, \sigma_n(c_j))$ the **singular values** of the matrix $M(c_j)$, then the *SVDRule()* while simply apply the *aggregation_rule* passed as parameter to them.

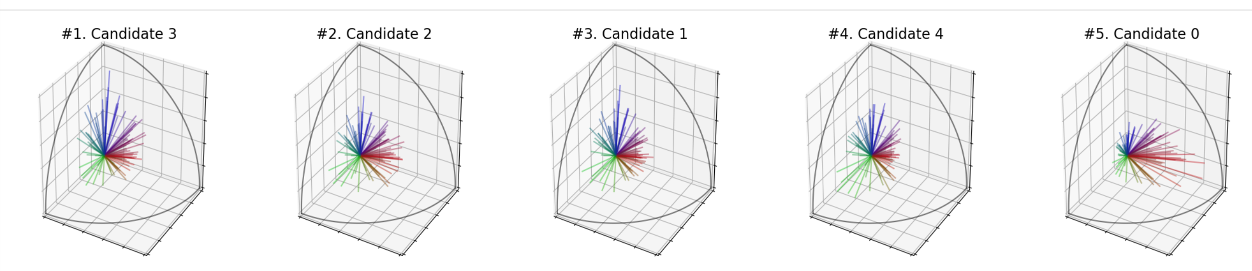
Singular values are very interesting in this context because each σ_k represent one group of voter.

In the following cell, I use the product function, that means that we compute the score with the following formula :

$$S(c_j) = \prod_k \sigma_k(c_j)$$

[16]:
```
election = ev.RuleSVD(aggregation_rule=np.prod, use_rank=False, square_root=True)
election(profile, embeddings)
print('Scores : ', election.scores_)
print('Ranking : ', election.ranking_)
print('Winner : ', election.winner_)
print('Welfare : ', election.welfare_)
election.plot_ranking("3D")
```

```
Scores :  [26.741050420162296, 32.74789296371305, 33.93409961876206, 35.
↪796076724146936, 32.132208898968436]
Ranking :  [3, 2, 1, 4, 0]
Winner :  3
Welfare :  [0.0, 0.6633710761179632, 0.7943708783523339, 1.0, 0.5953774509118517]
```



However, if you want to take the **product** of the singular values, you can directly use SVDNash(), as in the following cell.

This rule is a great rule because the score is equal to what is known as **the volume of the matrix** $M(c_j)$.

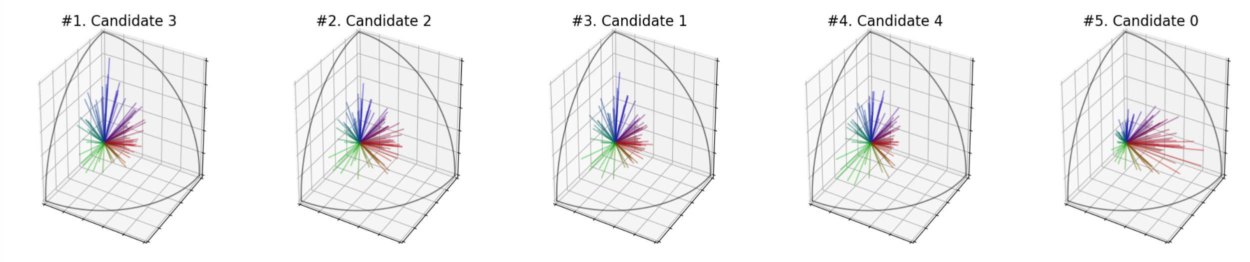
Indeed, we have :

$$S(c_j) = \prod_k \sigma_k(c_j) = \det(M(c_j)^t M(c_j)) = \det(M(c_j) M(c_j)^t)$$

which is often described as the volume of a matrix.

[17]:
```
election = ev.RuleSVDNash(use_rank=False)
election(profile, embeddings)
print('Scores : ', election.scores_)
print('Ranking : ', election.ranking_)
print('Winner : ', election.winner_)
print('Welfare : ', election.welfare_)
election.plot_ranking("3D")
```

```
Scores :  [26.741050420162296, 32.74789296371305, 33.93409961876206, 35.
↪796076724146936, 32.132208898968436]
Ranking :  [3, 2, 1, 4, 0]
Winner :  3
Welfare :  [0.0, 0.6633710761179632, 0.7943708783523339, 1.0, 0.5953774509118517]
```



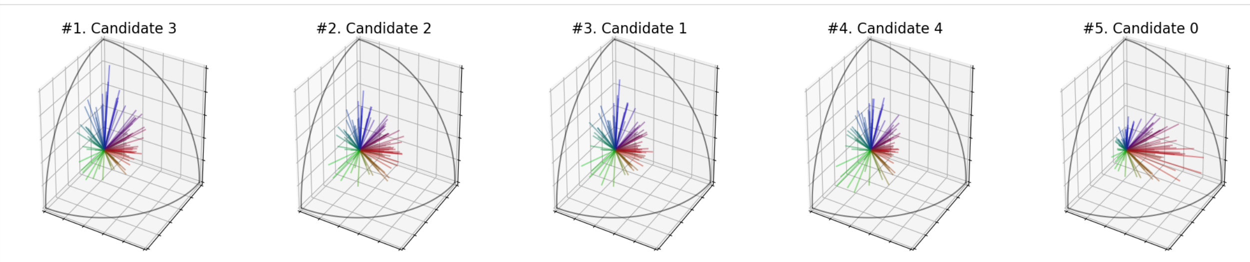
You can take the **sum** of the singular values with *SVDSum()* :

$$S(c_j) = \sum_k \sigma_k(c_j)$$

This corresponds to a **utilitarian** approach of the election.

[18]:
```
election = ev.RuleSVDSum(use_rank=False)
election(profile, embeddings)
print('Scores : ', election.scores_)
print('Ranking : ', election.ranking_)
print('Winner : ', election.winner_)
print('Welfare : ', election.welfare_)
election.plot_ranking("3D")
```

```
Scores :  [10.281197693329844, 10.791272853780711, 10.916846435776982, 11.
↪138673498000554, 10.697528579564]
Ranking :  [3, 2, 1, 4, 0]
Winner :  3
Welfare :  [0.0, 0.5948566218107435, 0.7413022489786075, 1.0, 0.48553076829268343]
```



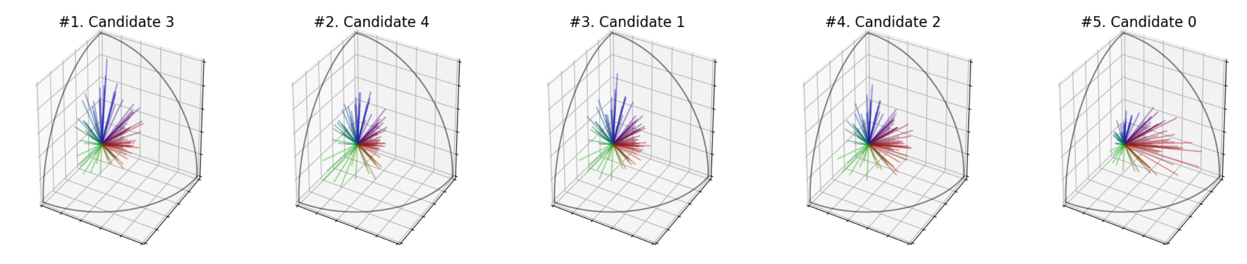
You can take the **minimum** of the singular values with *SVDMin()* :

$$S(c_j) = \min_k \sigma_k(c_j)$$

This corresponds to an **egalitarian** approach of the election.

[19]:
```
election = ev.RuleSVDMin(use_rank=False)
election(profile, embeddings)
print('Scores : ', election.scores_)
print('Ranking : ', election.ranking_)
print('Winner : ', election.winner_)
print('Welfare : ', election.welfare_)
election.plot_ranking("3D")
```

```
Scores :  [1.9464873750164875, 2.213040976979755, 2.1945439239678493, 2.
↪272105557536364, 2.2490434464796065]
Ranking :  [3, 4, 1, 2, 0]
Winner :  3
Welfare :  [0.0, 0.8186078550665599, 0.7618018964165792, 1.0, 0.9291743757111914]
```



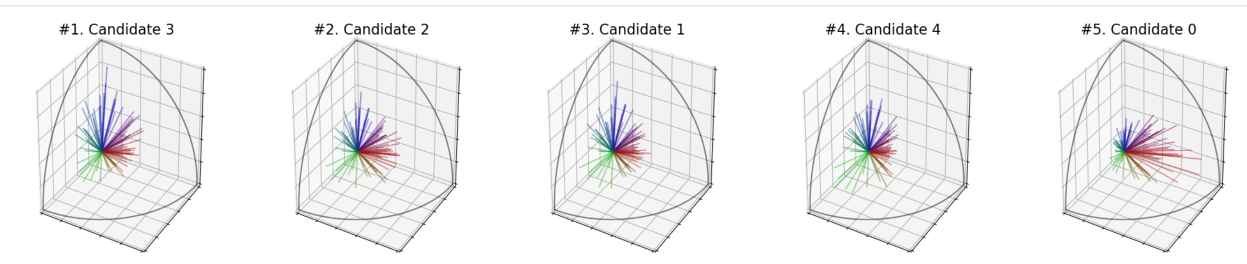
You can take the **maximum** of the singular values with *SVDMax()*:

$$S(c_j) = \max_k \sigma_k(c_j)$$

For single winner voting, this rule seems not very suited, because it will only maximize the satisfaction of **one group**. But it can be very useful **for multi-winner voting** (see the dedicated notebook).

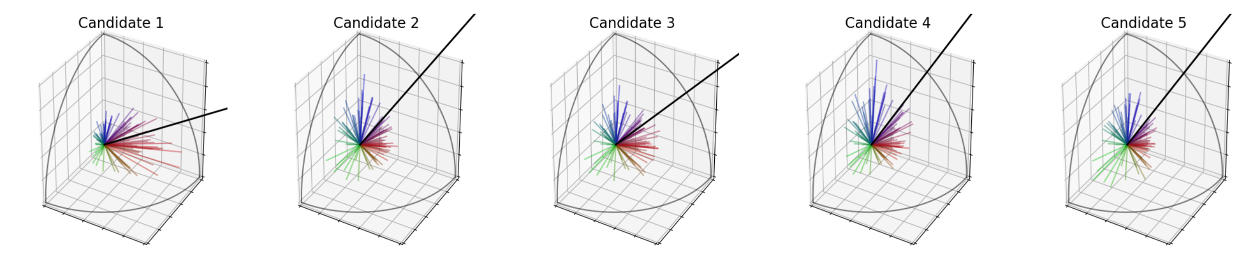
[20]:
```
election = ev.RuleSVDMax(use_rank=False)
election(profile, embeddings)
print('Scores : ', election.scores_)
print('Ranking : ', election.ranking_)
print('Winner : ', election.winner_)
print('Welfare : ', election.welfare_)
election.plot_ranking("3D")
```

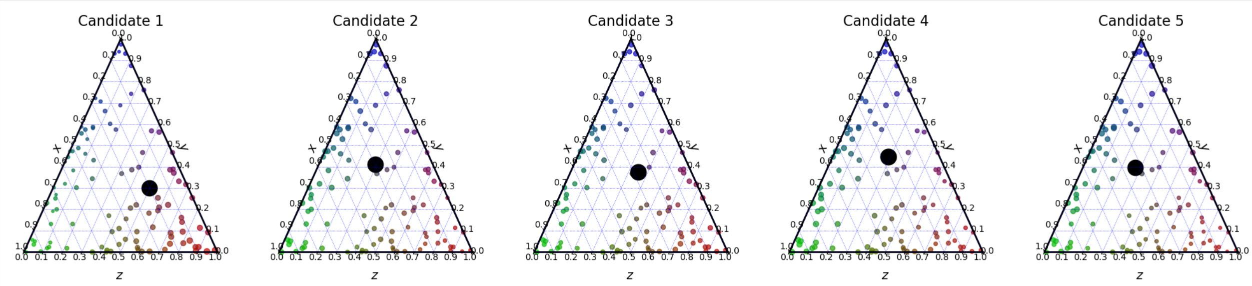
```
Scores :  [6.072281075810715, 6.186173173309101, 6.247073024447592, 6.407979709322462,
↪ 6.110288589883775]
Ranking :  [3, 2, 1, 4, 0]
Winner :  3
Welfare :  [0.0, 0.33926887430836317, 0.5206811443001008, 1.0, 0.1132191503893329]
```



For this rule in particular, we can plot the *"features"* of the candidates, which actually corresponds to **the position of the most important singular vector** from the singular value decomposition.

[21]:
```
election.plot_features("3D")
election.plot_features("ternary")
```





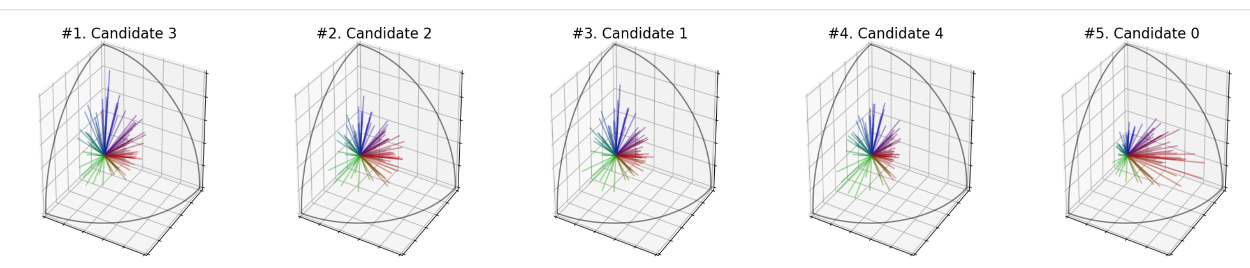
Finally, there is the *SVDLog()* rule, which is a bit more exotic. It corresponds to the following equation :

$$S(c_j) = \sum_k \log\left(1 + \frac{\sigma_k(c_j)}{C}\right)$$

where C is a constant passed as parameter (its default value is 1).

[22]:
```
election = ev.RuleSVDLog(const=2, use_rank=False)
election(profile, embeddings)
print('Scores : ', election.scores_)
print('Ranking : ', election.ranking_)
print('Winner : ', election.winner_)
print('Welfare : ', election.welfare_)
election.plot_ranking("3D")
```

```
Scores :  [2.831659715452525, 2.940987770371308, 2.9627590583541927, 2.
↪996678664966983, 2.927844515160104]
Ranking :  [3, 2, 1, 4, 0]
Winner :  3
Welfare :  [0.0, 0.6625181849748972, 0.7944502330635768, 1.0, 0.582871239882373]
```



The following table summarize the different rules based on the **SVD** :

Name	Equation	Interpretation
SVDNash	$S(c_j) = \prod_k \sigma_k(c_j)$	Nash Welfare
SVDSum	$S(c_j) = \sum_k \sigma_k(c_j)$	Utilitarian
SVDMin	$S(c_j) = \min_k \sigma_k(c_j)$	Egalitarian
SVDMax	$S(c_j) = \max_k \sigma_k(c_j)$	Dictature of majority
SVDLog	$S(c_j) = \sum_k \log\left(1 + \frac{\sigma_k(c_j)}{C}\right)$	Between Nash and Utilitarian

Features based rule

The next rule is based on what is often called *features* in machine learning.

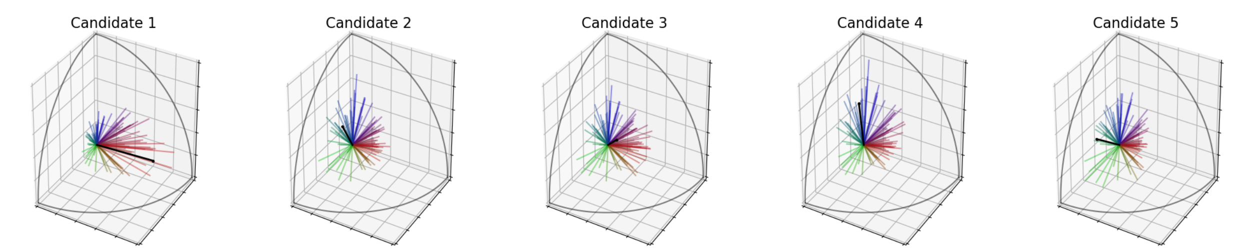
Indeed, it consists in **solving the linear regression** on $MX_j = s^*(c_j)$ for every candidate c_j. We want the vector X_j such that

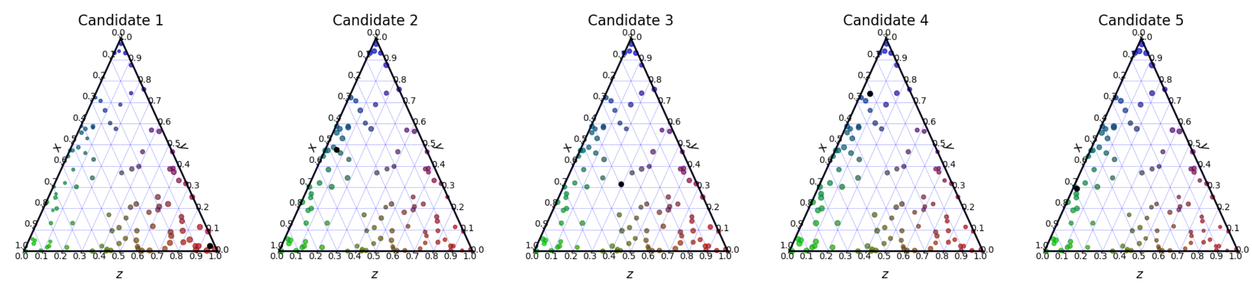
$$X_j = \min_X ||MX - s^*(c_j)||_2^2$$

It corresponds to $X_j = (M^t M)^{-1} M s^*(c_j)$. This is the classic feature vector for candidate c_j.

In the following cell, the **features** of every candidate are shown in black on the 3D plots.

[23]:
```
election = ev.RuleFeatures()
election(profile, embeddings)
election.plot_features("3D")
election.plot_features("ternary")
```



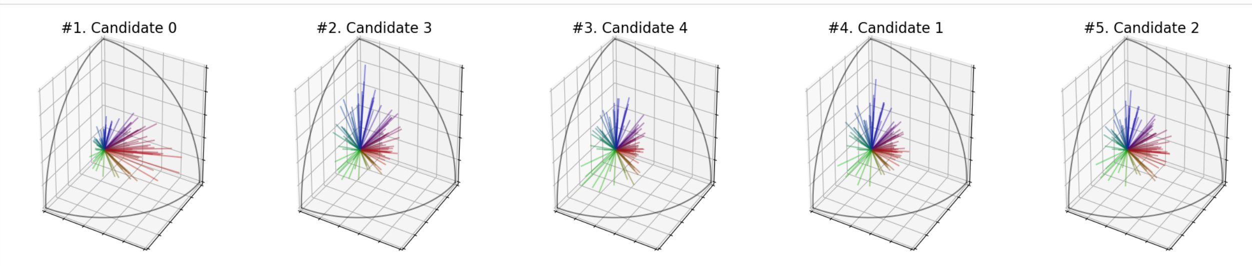


Then, we define the score of candidate c_j as :

$$S(c_j) = ||X_j||_2^2$$

[24]:
```
print('Scores : ', election.scores_)
print('Ranking : ', election.ranking_)
print('Winner : ', election.winner_)
print('Welfare : ', election.welfare_)
election.plot_ranking("3D")
```

```
Scores :  [0.679363163960879, 0.596773936003501, 0.5674211903311394, 0.
↪6579158241916001, 0.6380881542815943]
Ranking :  [0, 3, 4, 1, 2]
Winner :  0
Welfare :  [1.0, 0.2622139374587864, 0.0, 0.8084066318124928, 0.631282097850031]
```



4.4 3. Analysis of the voting rules

To explore the rules in **more details**, we created a class *MovingVoterProfile*, which enables to see the evolution of the candidates' scores depending on the embeddings of one particular voter, which are changing.

[1]:
```
import embedded_voting as ev
import numpy as np
```

[2]:
```
moving_profile = ev.MovingVoter()
```

4.4.1 Description of the profile

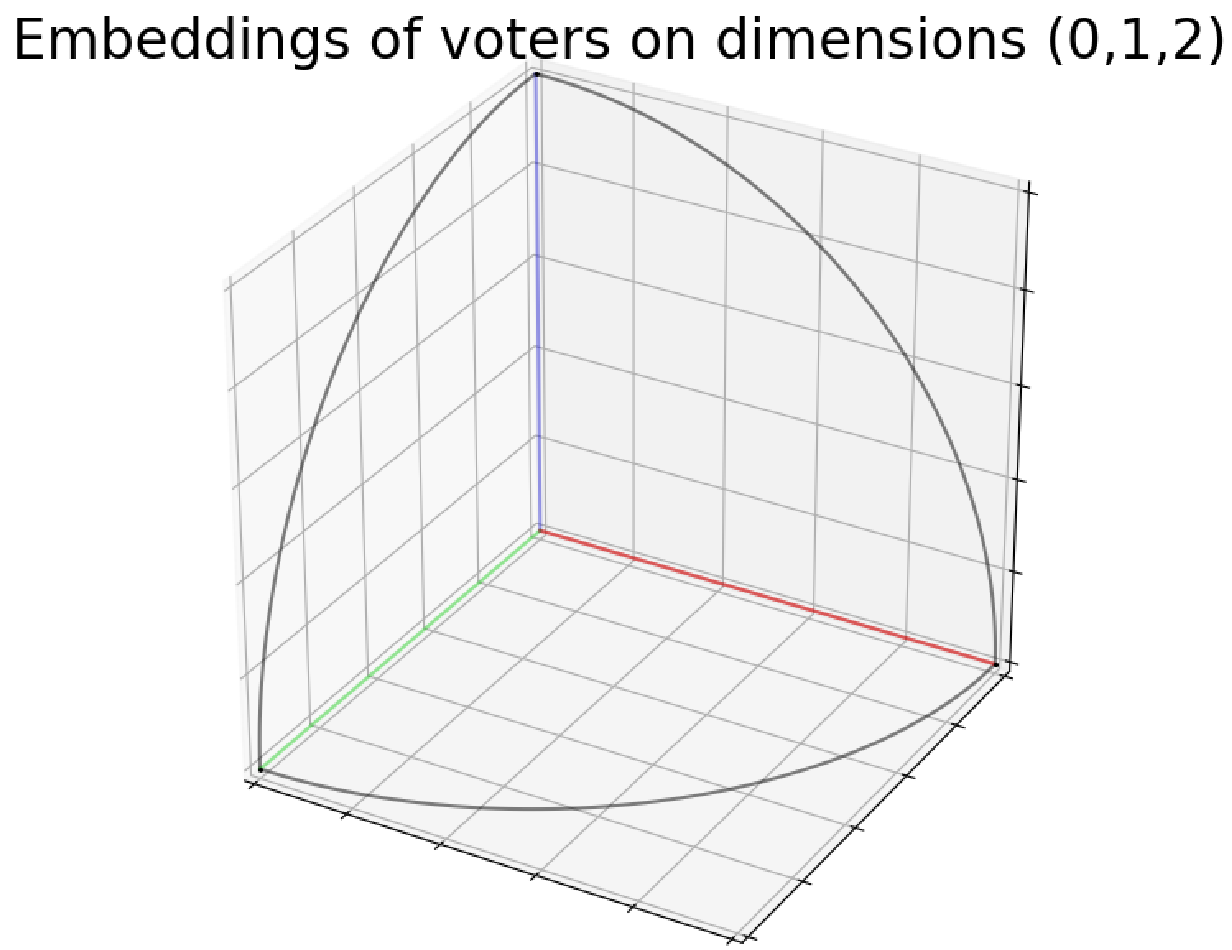
The basic version of the profile contains **4 candidates** and **4 voters** :

- **Voter 0** is the moving voter. His initial position is the same than the **Voter 1**, and he gives a score of 0.8 to every candidate, except for **Candidate 4** which receive a score of 0.5.

- **Voter 1, 2, and 3** respectively supports **Candidate 1, 2 and 3** with a score of 1 and gives a score of 0.1 to every other candidate, except **Candidate 4** which receive a score of 0.5 from every voter.

The following figure shows the initial configuration of the profile.

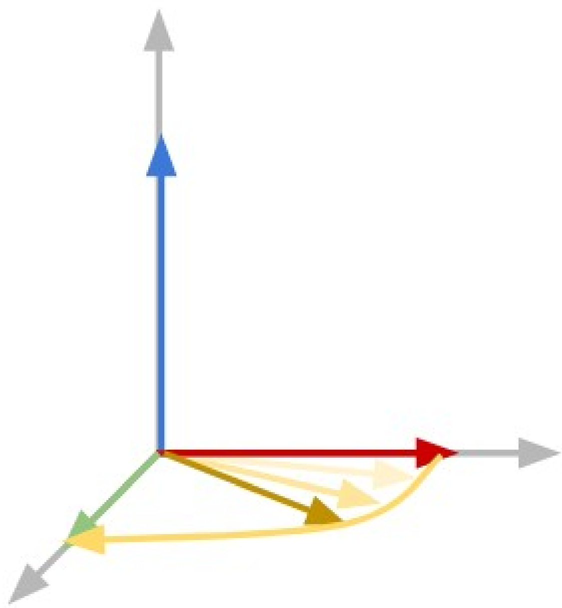
[3]:
```
moving_profile.embeddings.plot()
```



[3]:
```
<matplotlib.axes._subplots.Axes3DSubplot at 0x2531c6f3b38>
```

4.4.2 The evolution of the scores

Then we want to track **the evolution** of the scores of the different candidates depending on the embeddings of the **Voter 0**. These embeddings are changing as detailed on the following figure:



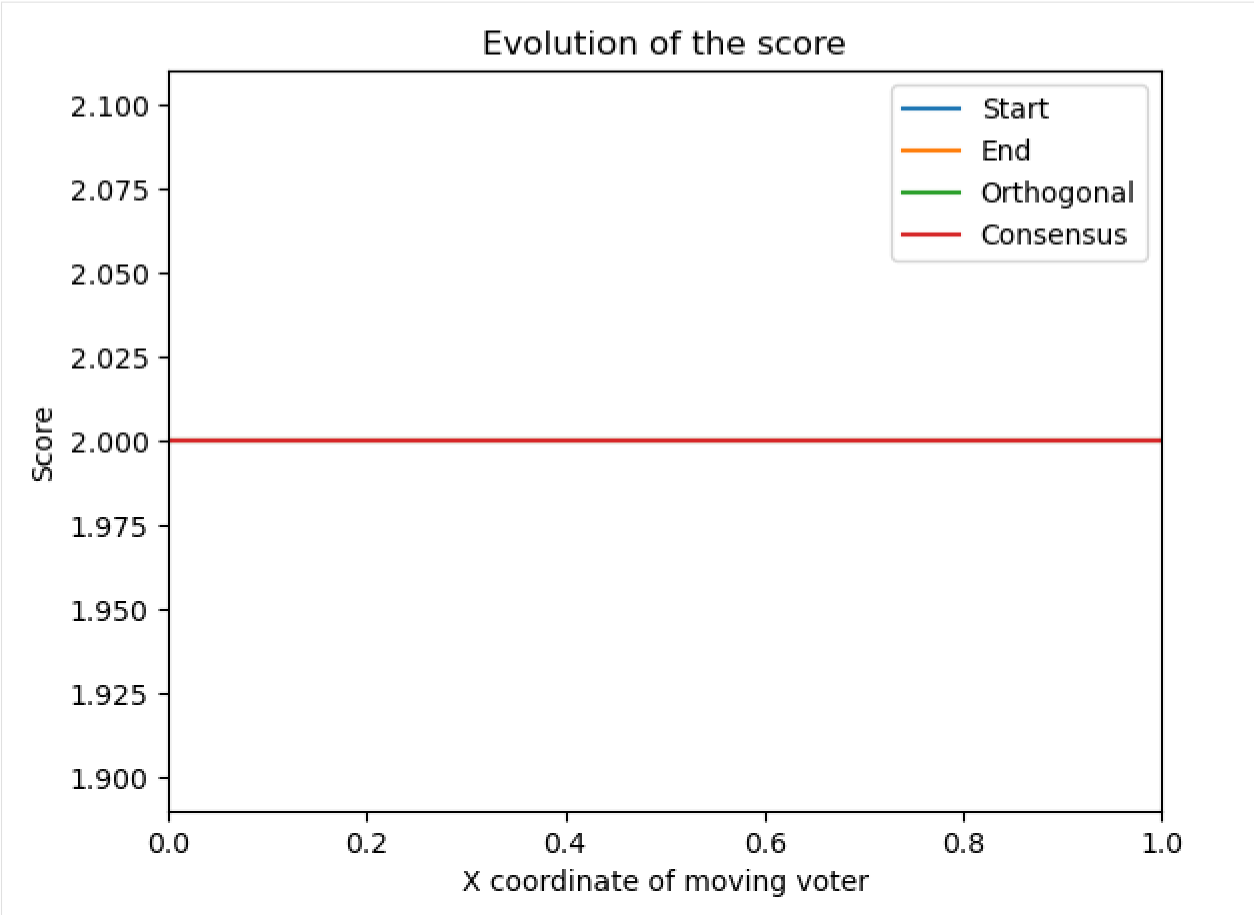
As you can see, the voter starts in the **red** area and ends in the **green** area but always remains orthogonal to the **blue** voter.

Using this, we can see what happens to the different scores **depending on the voting rule** used.

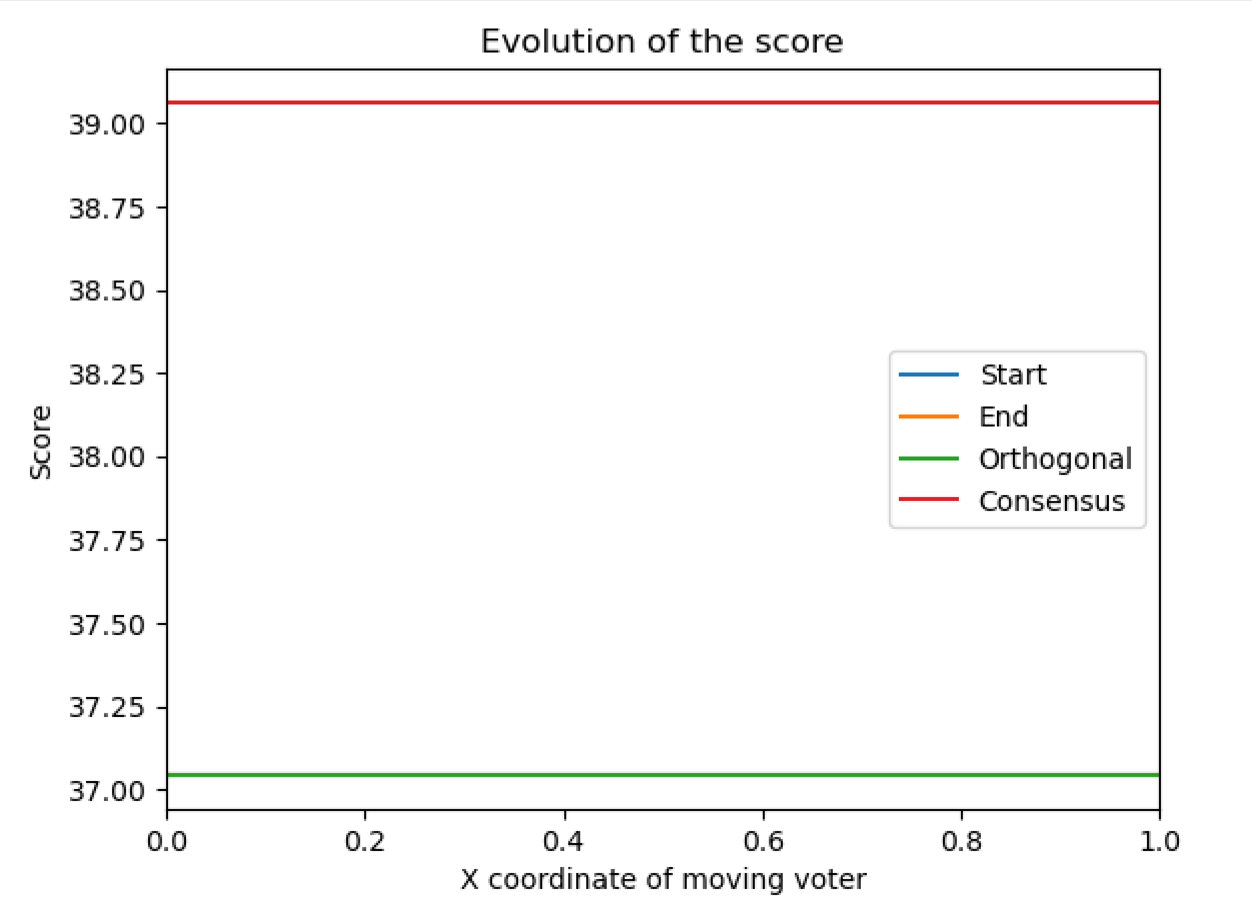
Without any surprise, it **does not change anything** for rules which do not depend on the embeddings :

- When we do **the sum of the scores**, every candidate has the same final score.
- When we do **the product of the scores**, only the **Candidate 4** (consensus) has a good score.

[4]:
```
rule = ev.RuleSumRatings()
moving_profile(rule).plot_scores_evolution()
```



[5]:
```
rule = ev.RuleShiftProduct()
moving_profile(rule).plot_scores_evolution()
```

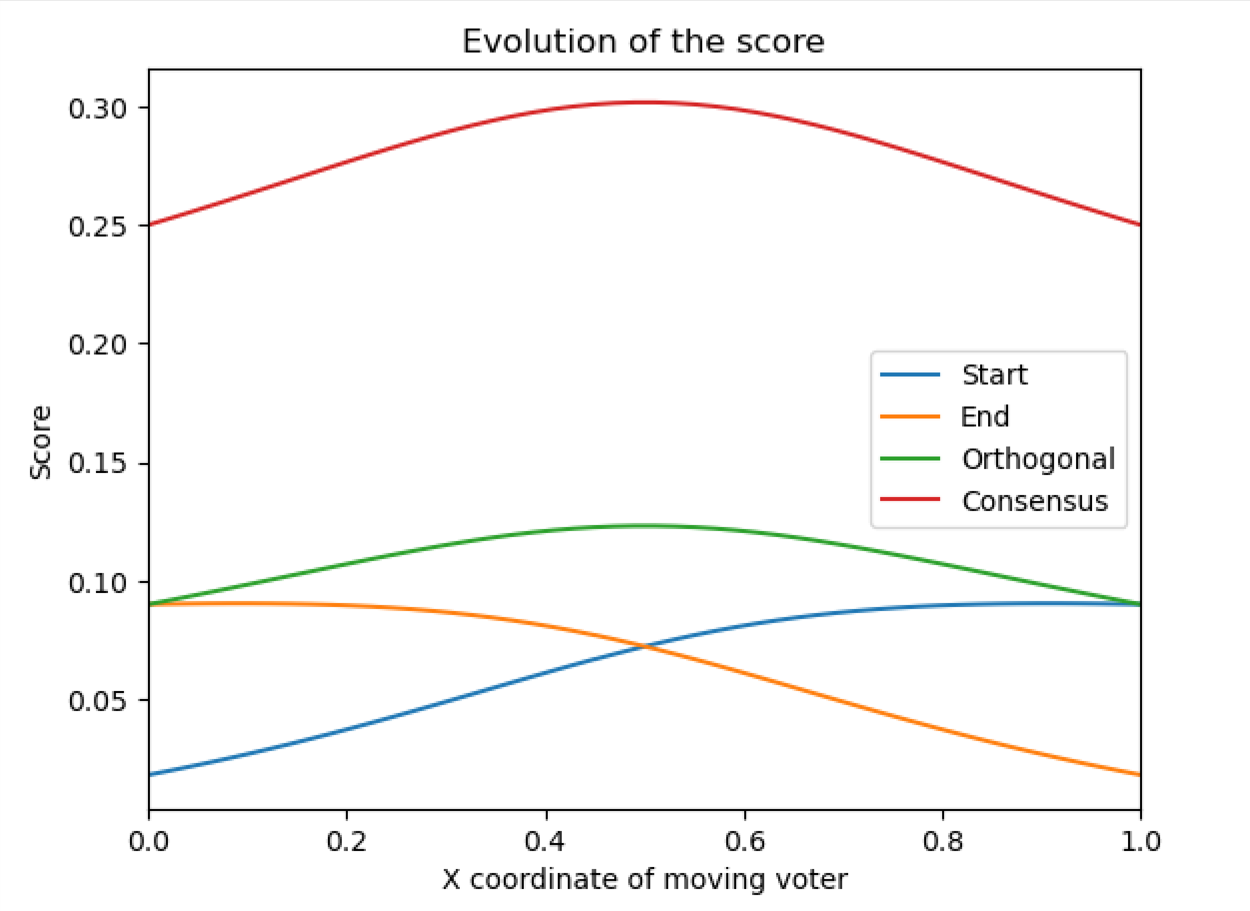


It becomes interesting when we look at **geometrical rules**. What happens for the **Zonotope** and **MaxCube** rules?

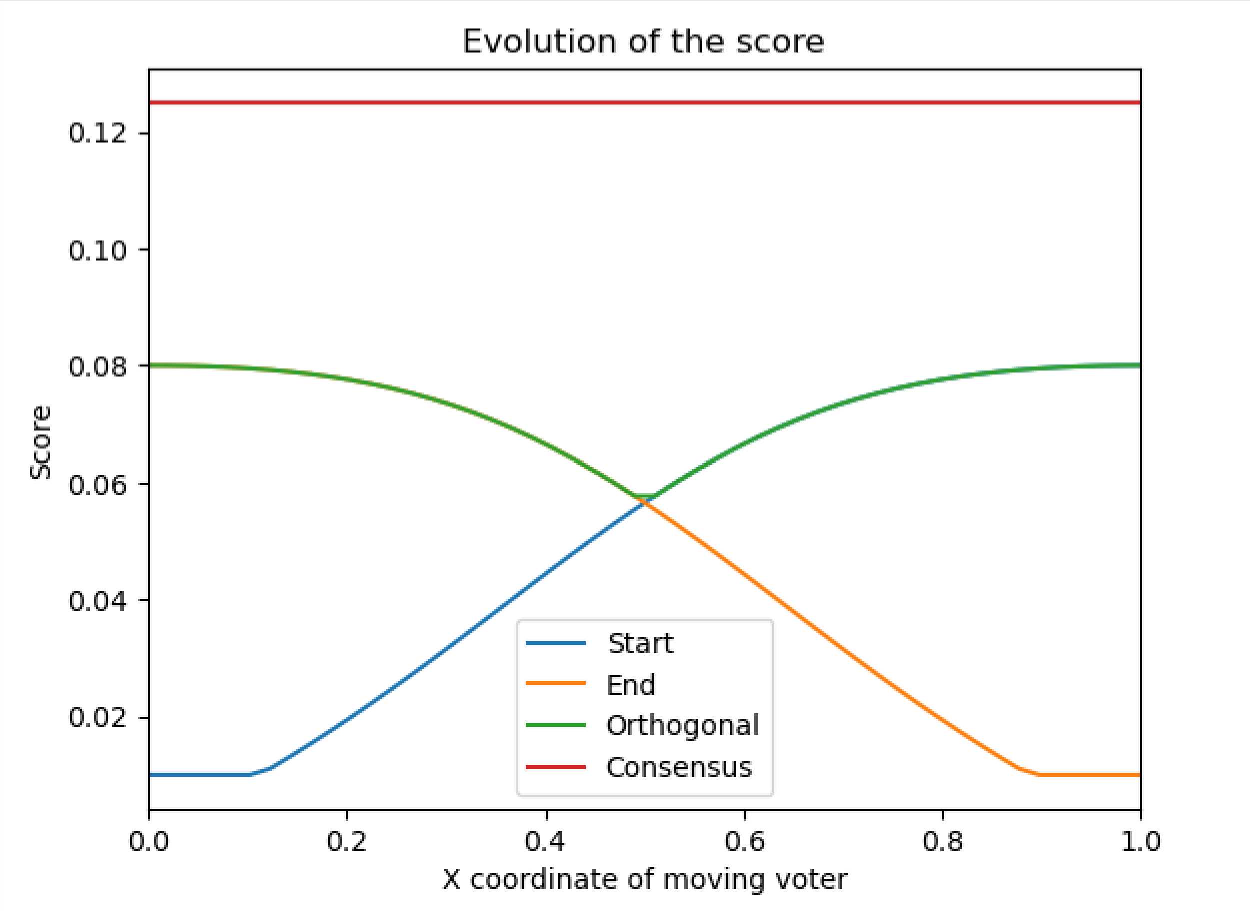
- The **Consensus candidate** gets the best score.
- The second best candidate is the one supported by the **Orthogonal vector**. Indeed, he is supported by the **moving voter** and a another one which is **orthogonal** to the first one, and *orthogonality maximizes the volume*.
- For the same reason, the candidate supported by the **voter of the start** gets a better score at the end, and the candidate supported by the **voter of the end** get the better score at the beginning.

However, you can notice that the score of some candidate is greater when the moving voter is between the two positions, and there is no intuitive interpretation of this observation.

[6]:
```
rule = ev.RuleZonotope()
moving_profile(rule).plot_scores_evolution()
```



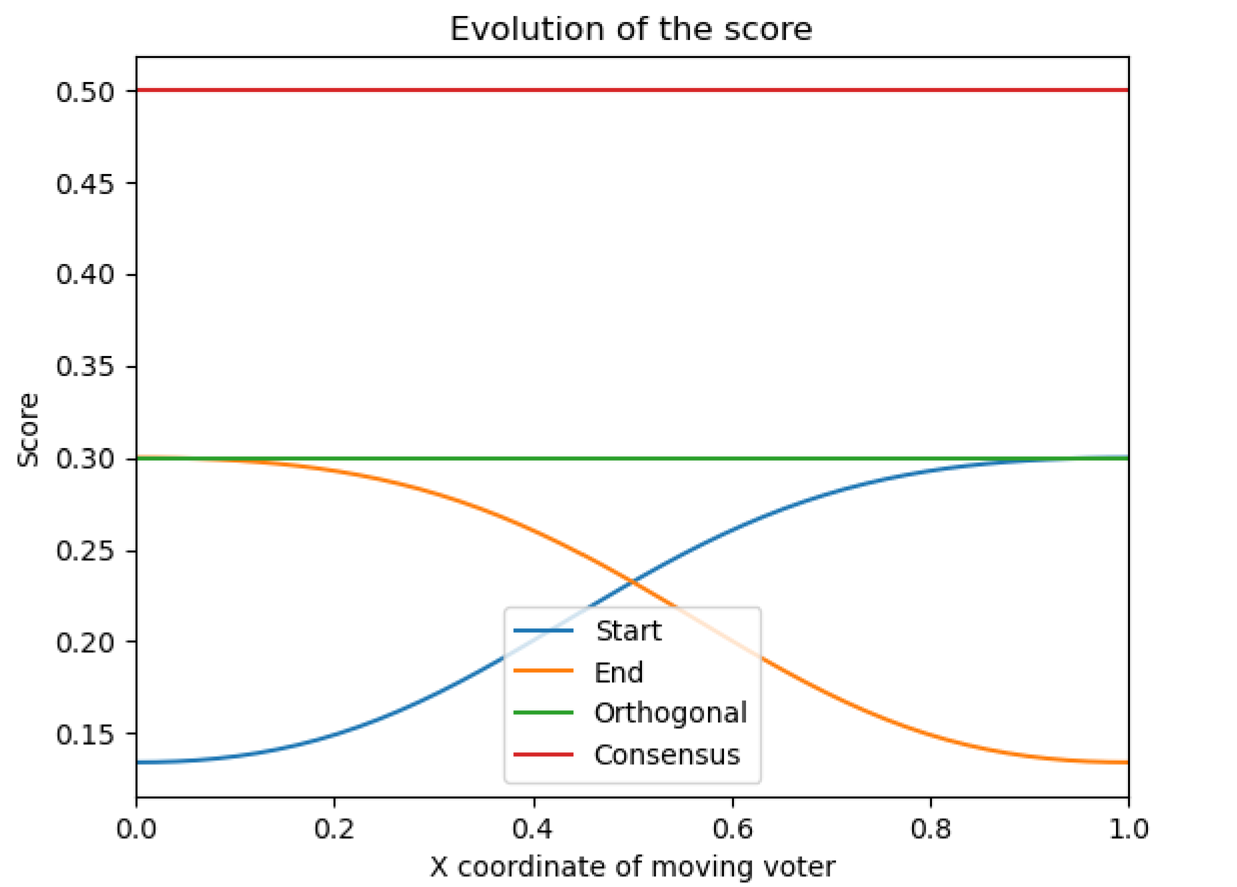
[7]:
```
rule = ev.RuleMaxParallelepiped()
moving_profile(rule).plot_scores_evolution()
```



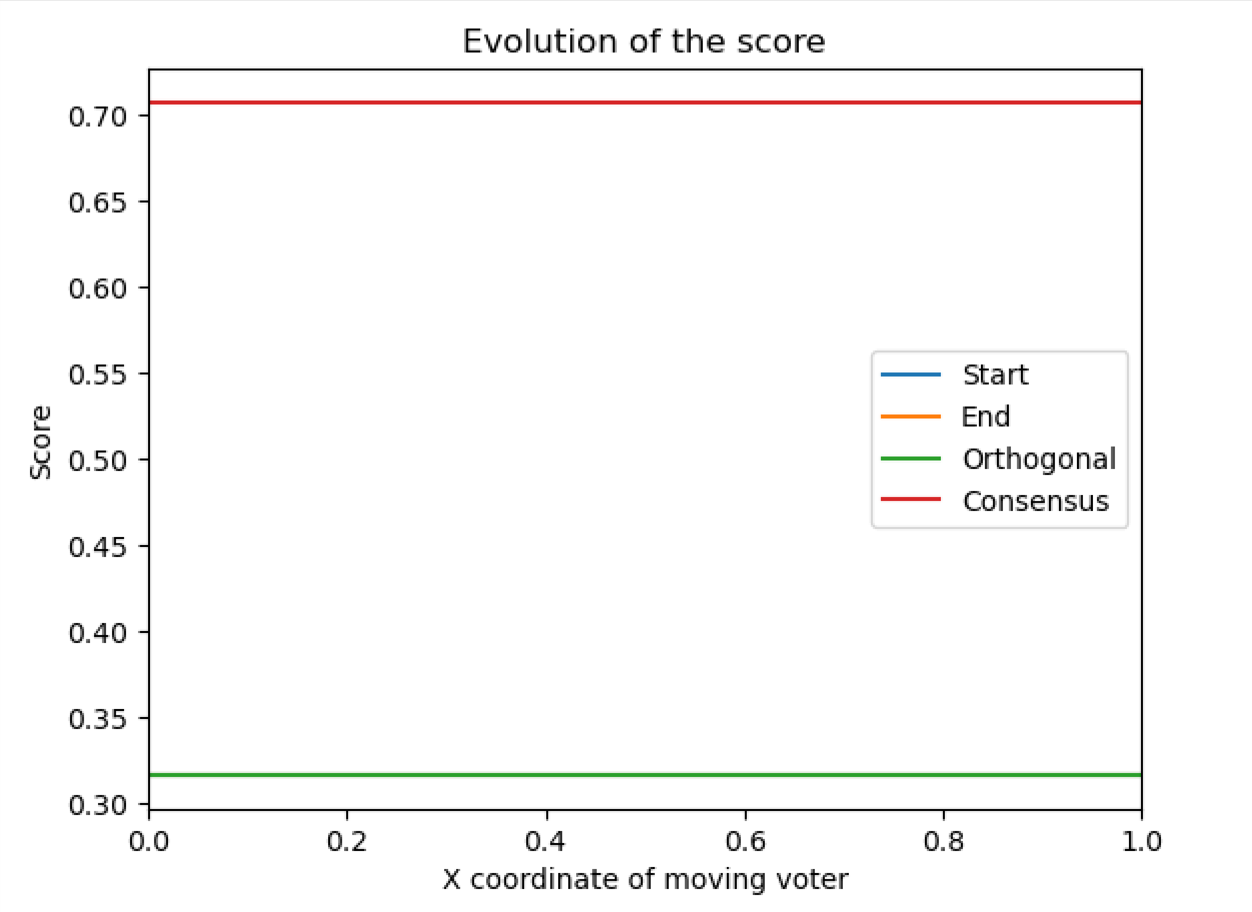
What happens with **SVD Rules**?

- **SVDNash**, **SVDLog** and **SVDSum** work a bit like the **Zonotope** and **MaxCube** rules, but the scores of the candidates are always between their scores at the **beginning** and at the **end**.
- **SVDMin** is not very interesting: nothing really change between the beginning and the end.
- **SVDMax** is the opposite of the other rules : the **Consensus candidate** and the **Orthogonal candidate** receive the worst scores, but the candidate supported by the voter from the **start** get the best score at the beginning and the candidate supported by the voter from the **end** get the best score at the end.

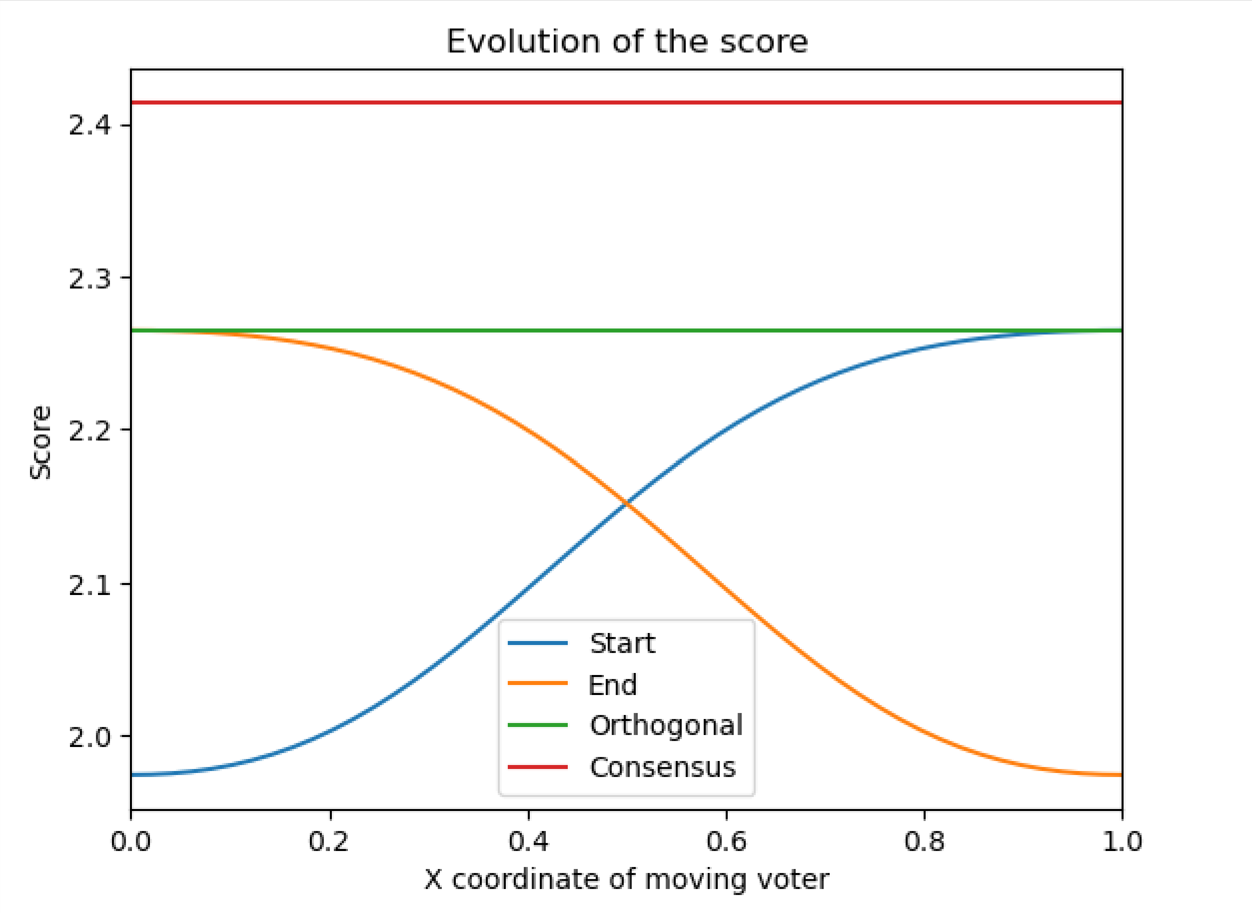
[8]:
```
rule = ev.RuleSVDNash()
moving_profile(rule).plot_scores_evolution()
```



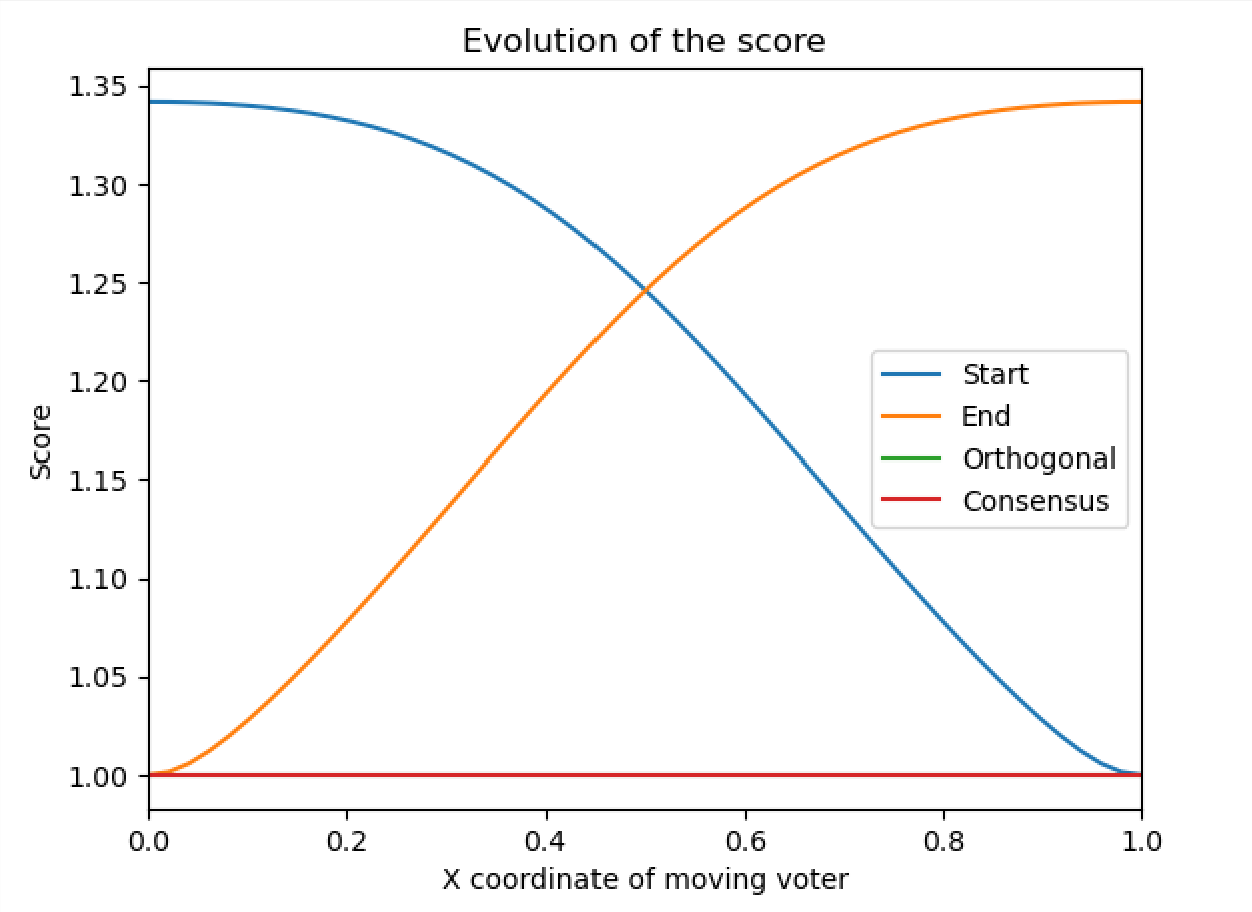
[9]:
```
rule = ev.RuleSVDMin()
moving_profile(rule).plot_scores_evolution()
```



[10]:
```
rule = ev.RuleSVDSum()
moving_profile(rule).plot_scores_evolution()
```

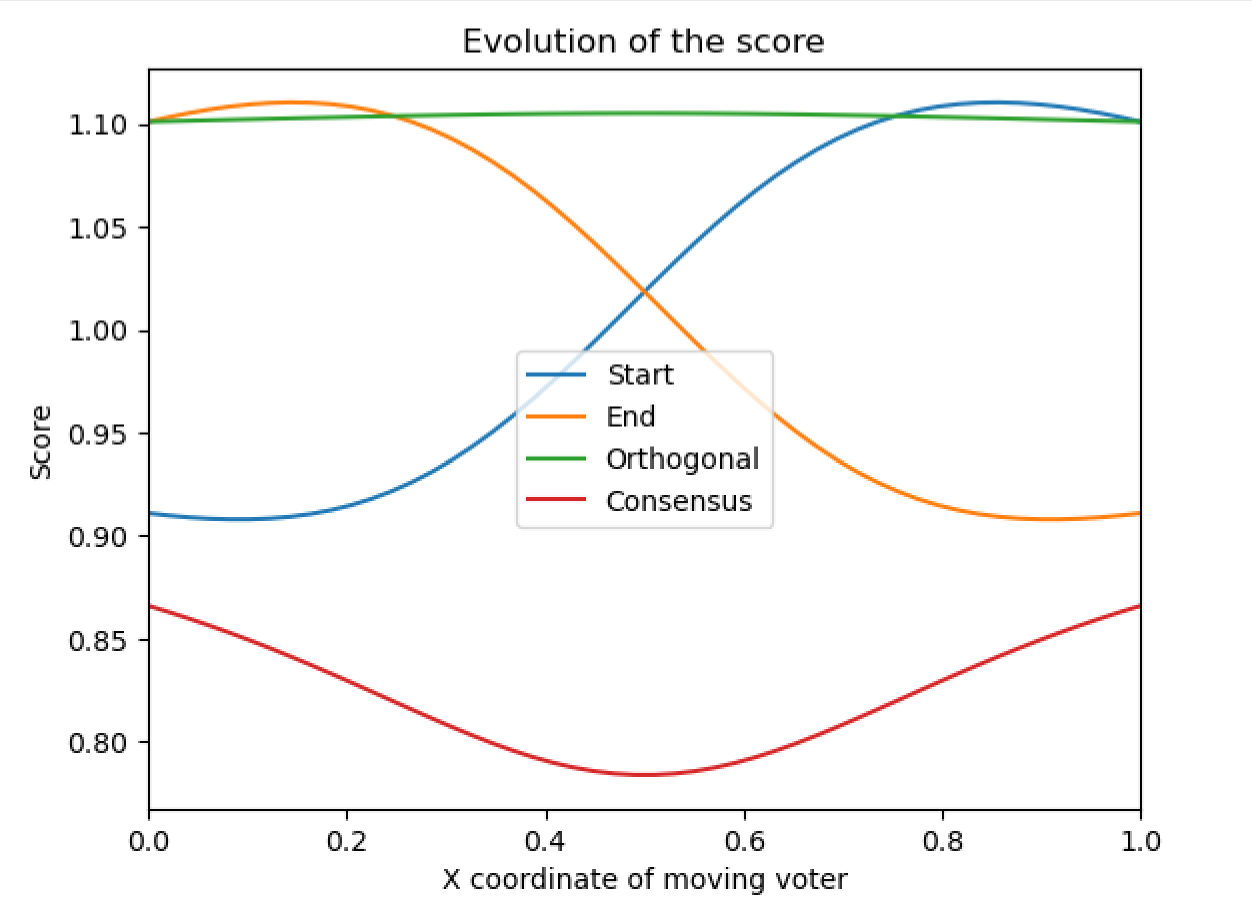


[11]:
```
rule = ev.RuleSVDMax()
moving_profile(rule).plot_scores_evolution()
```



Finally, we obtain a beautiful figure with the **Features rule**, even if it is a bit strange.

[12]:
```
rule = ev.RuleFeatures()
moving_profile(rule).plot_scores_evolution()
```

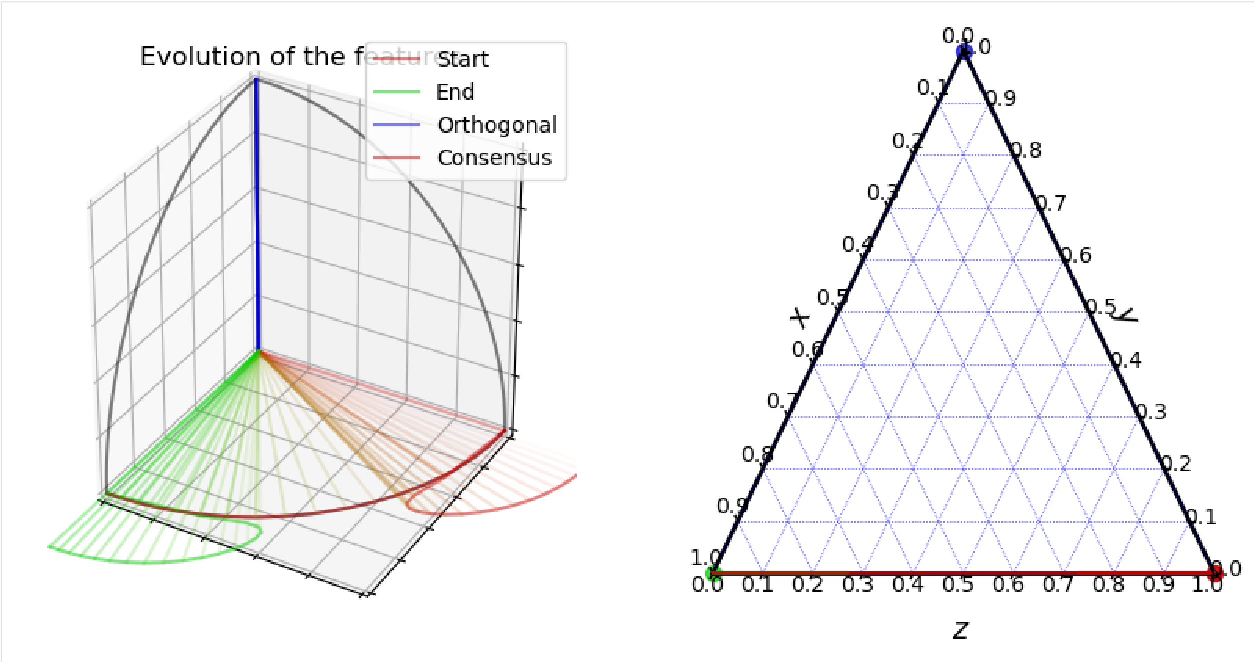


4.4.3 The evolutions of the features

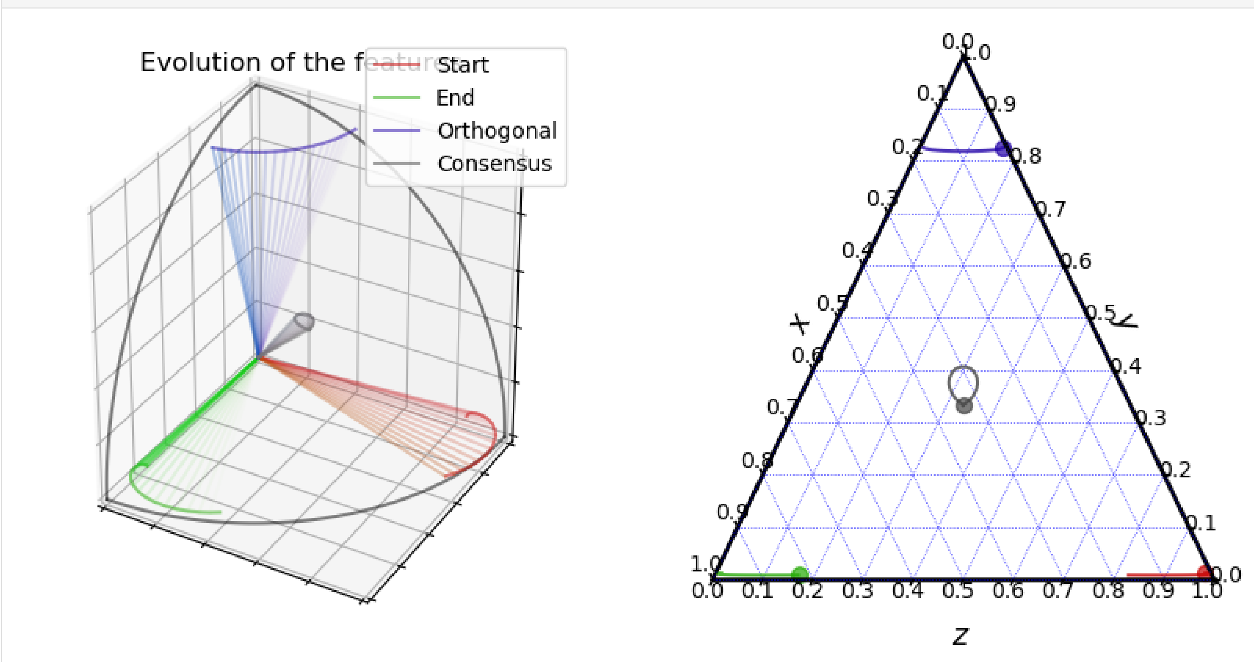
Some rules associate **features vectors** to every candidate. That is the case of the **SVDMax** and the **Features** rules. We can show the evolution of these vectors using the same class.

You can see that there are **major differences** between the features of the two rules. For instance, the features of the **Consensus candidate** follow the moving voter for the **SVDMax** rule, and they are on the center of the simplex for the **Features** rule.

[13]:
```
rule = ev.RuleSVDMax()
moving_profile(rule).plot_features_evolution()
```




```
[14]: rule = ev.RuleFeatures()
      moving_profile(rule).plot_features_evolution()
```

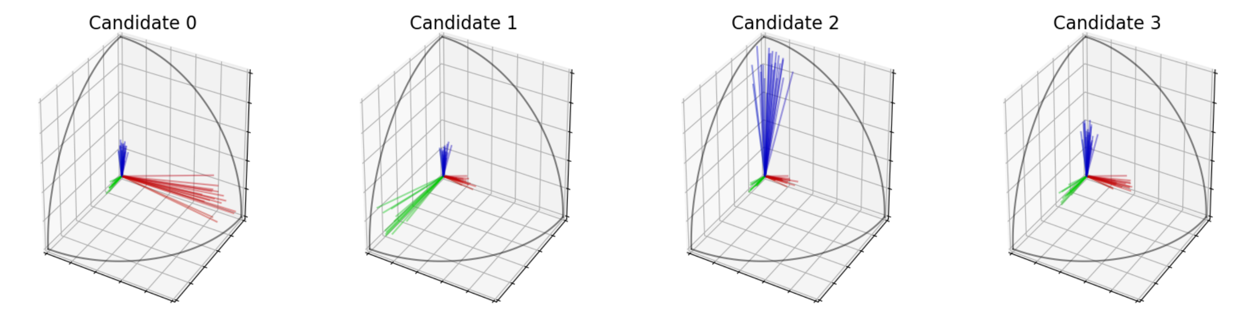


4.4.4 More complex profiles

Of course, you can play with **more complex profiles**, and even change the index of the moving voter.

```
[15]: scores = np.array([[1, .1, .1, .3], [.1, 1, .1, .3], [.1, .1, 1, .3]])
      embs = ev.EmbeddingsGeneratorPolarized(50, 3)(polarisation=.8)
      profile = ev.RatingsFromEmbeddingsCorrelated(.8, scores, 3, 4)(embs)
```

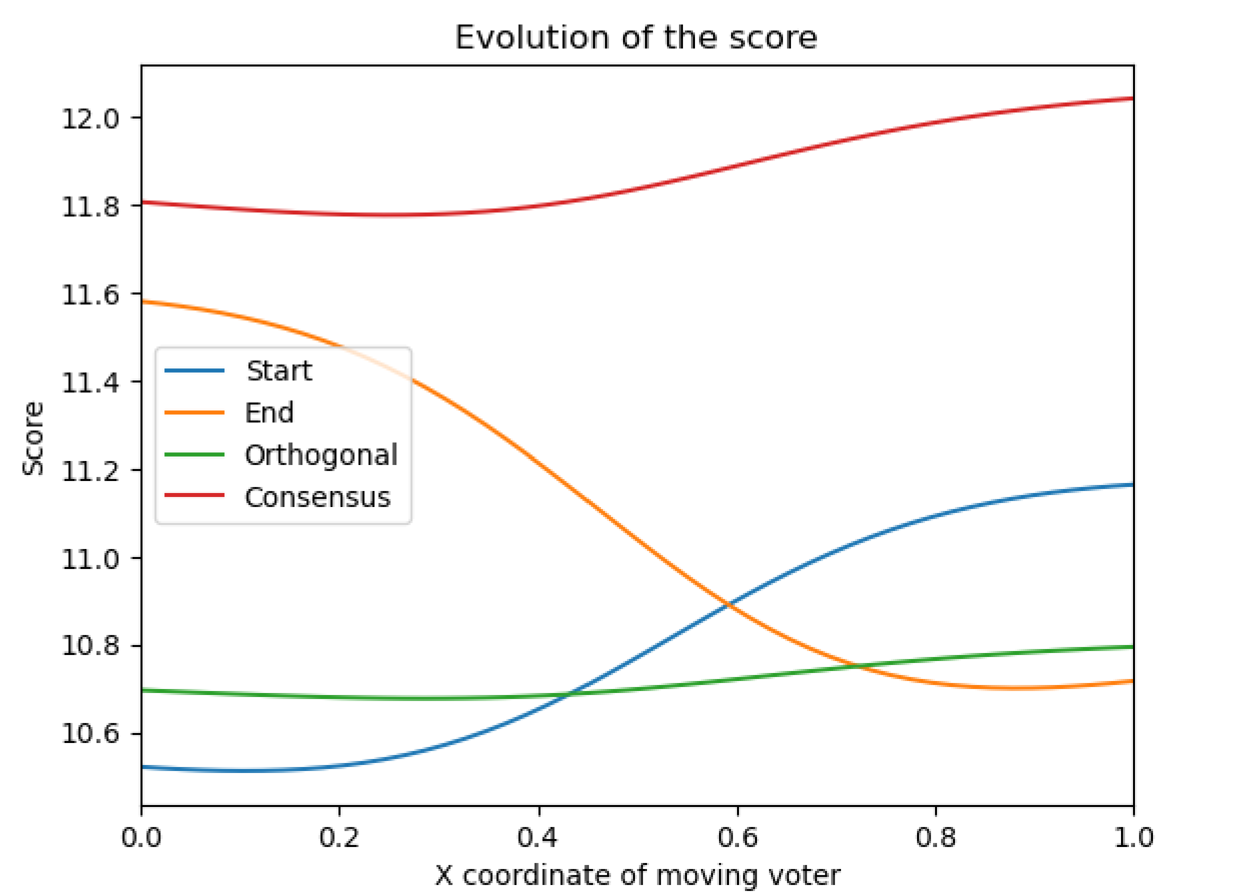
[16]:
```
embs.plot_candidates(profile)
```



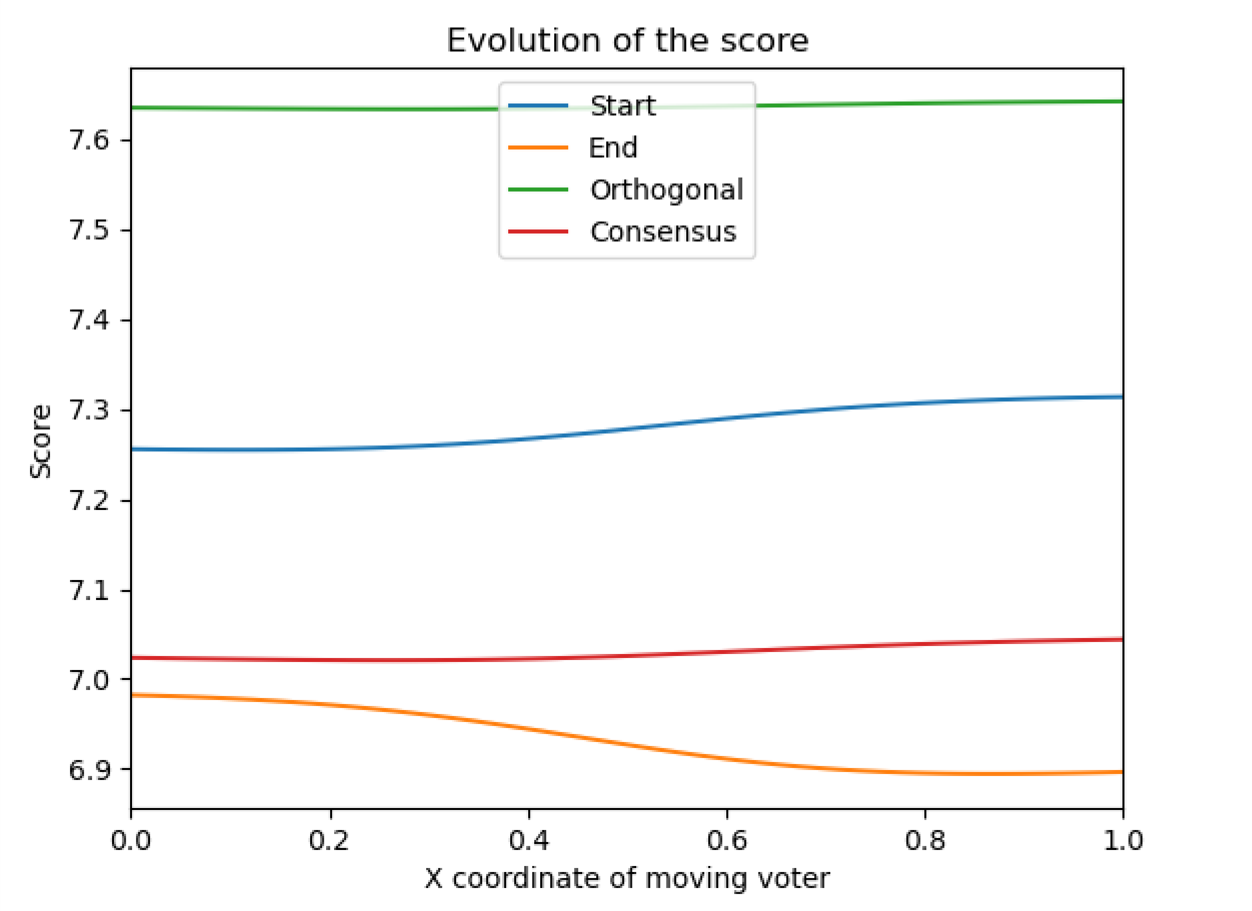
[17]:
```
moving_profile = ev.MovingVoter(embs)
```

We now obtain very funny plots for the **SVD Rules**:

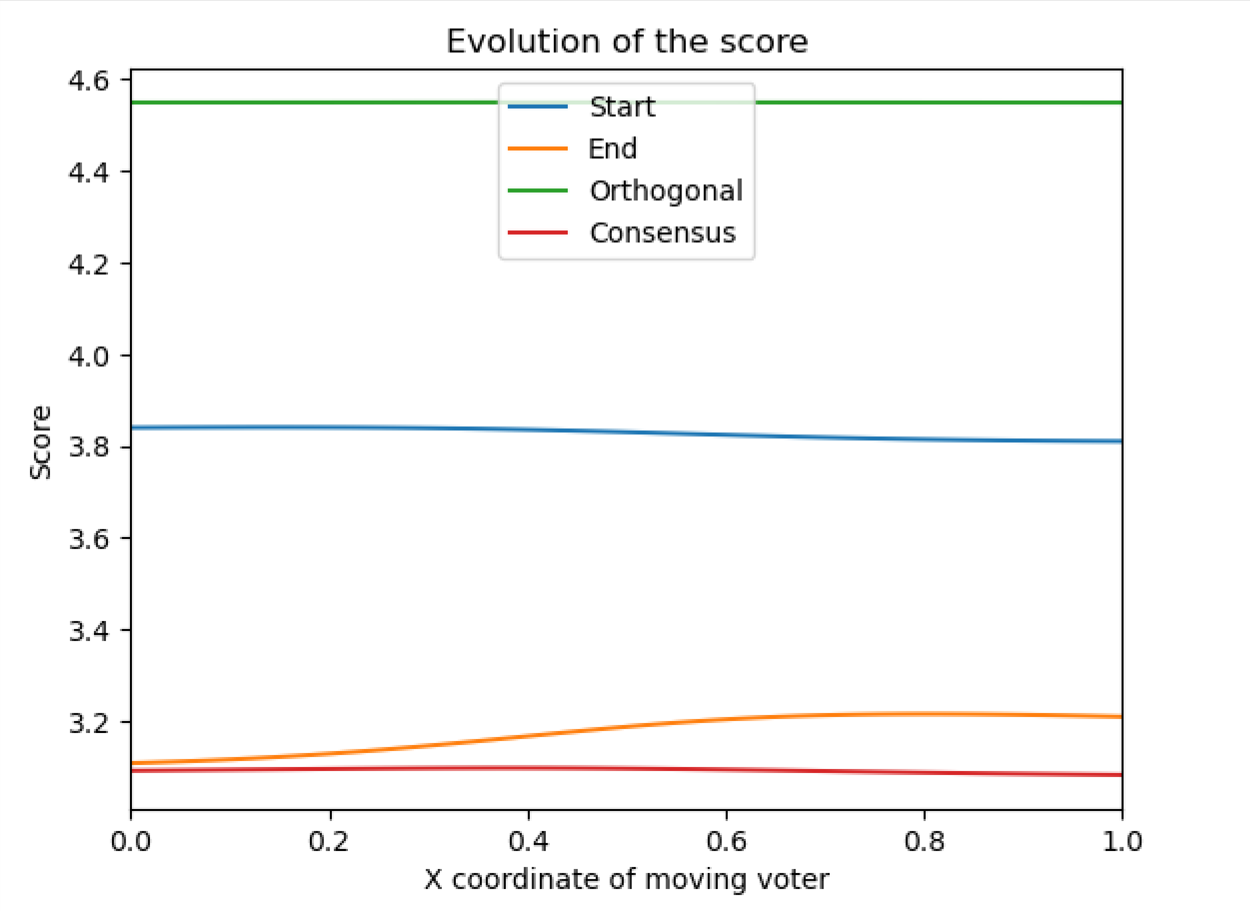
[18]:
```
rule = ev.RuleSVDNash()
moving_profile(rule, profile).plot_scores_evolution()
```



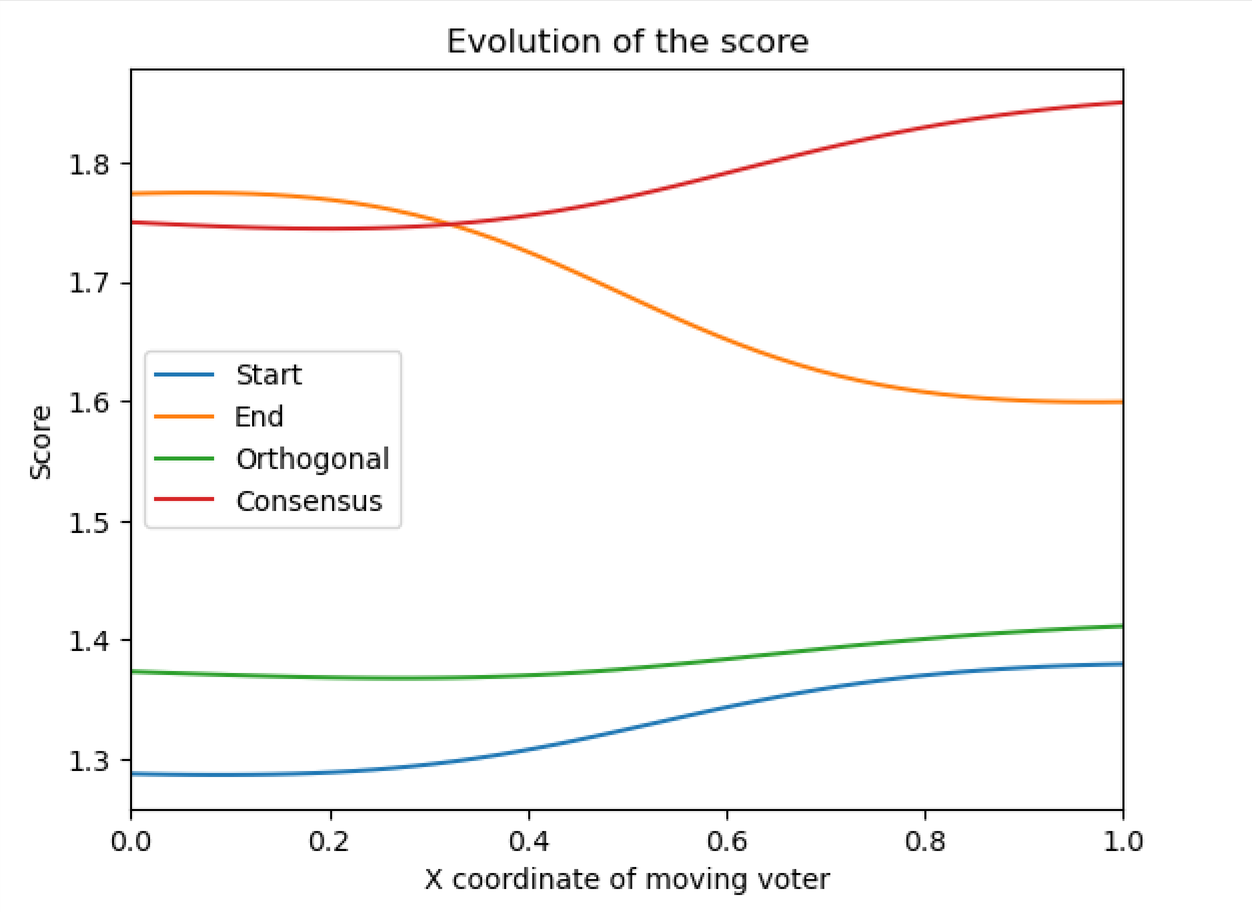
[19]:
```
rule = ev.RuleSVDSum()
moving_profile(rule, profile).plot_scores_evolution()
```



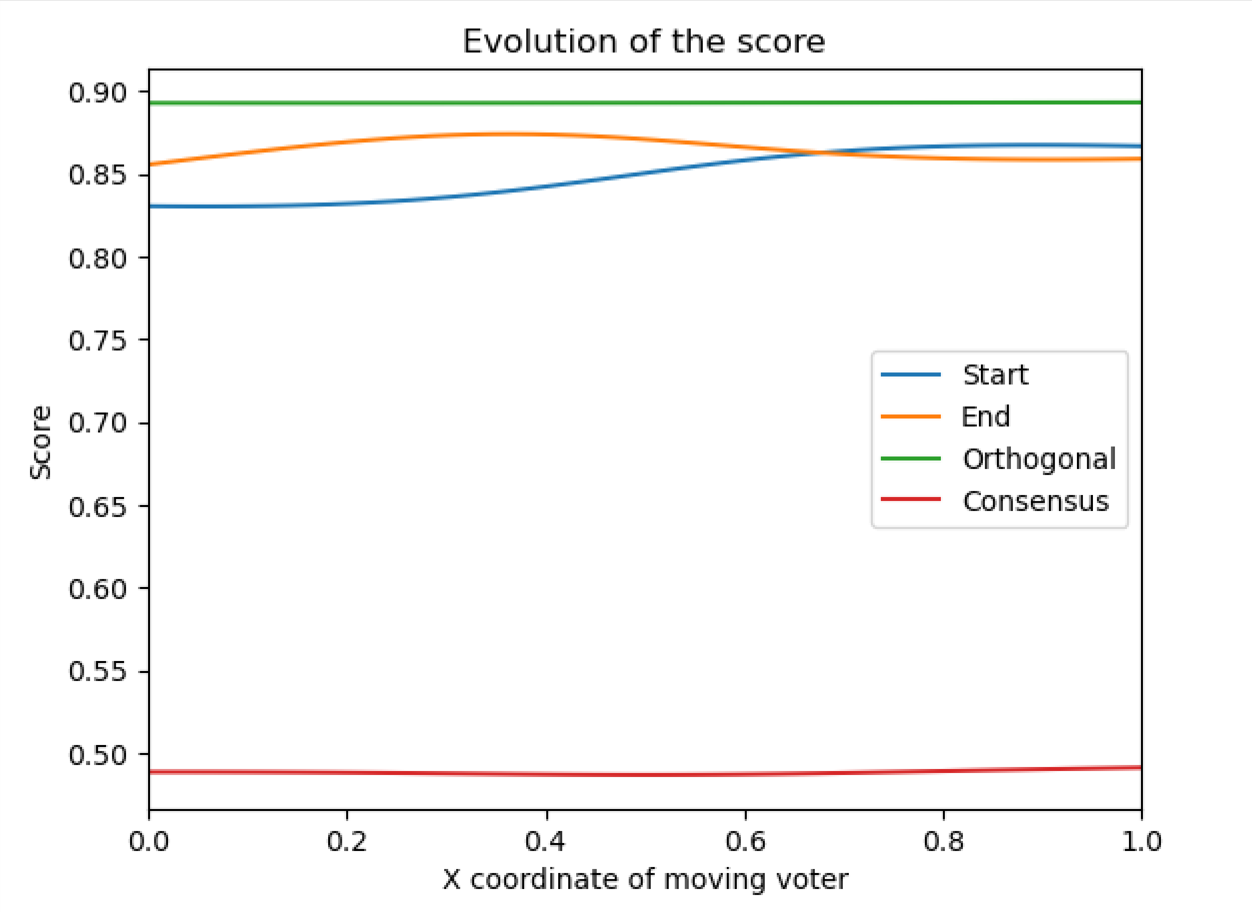
[20]:
```
rule = ev.RuleSVDMax()
moving_profile(rule, profile).plot_scores_evolution()
```



[21]:
```
rule = ev.RuleSVDMin()
moving_profile(rule, profile).plot_scores_evolution()
```

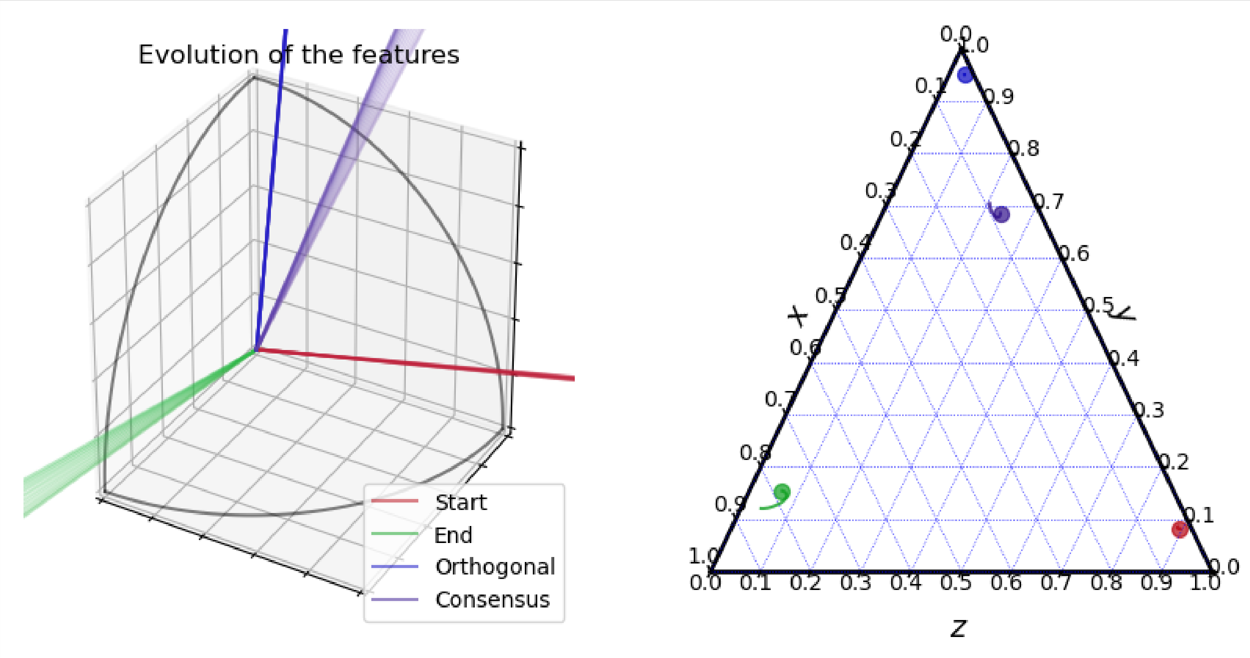


[22]:
```
rule = ev.RuleFeatures()
moving_profile(rule, profile).plot_scores_evolution()
```

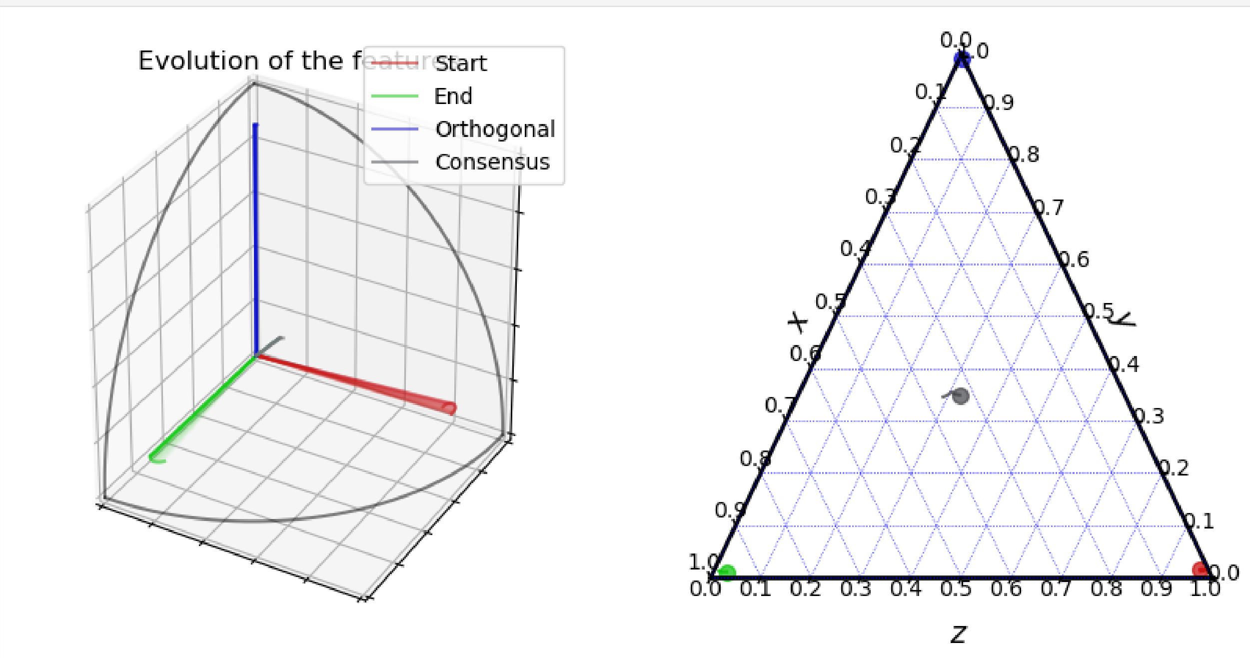


The features between **SVDMax** and **Features** rules are now far more similar:

[23]:
```
rule = ev.RuleSVDMax()
moving_profile(rule, profile).plot_features_evolution()
```



[24]:
```
rule = ev.RuleFeatures()
moving_profile(rule, profile).plot_features_evolution()
```



4.5 4. Ordinal preferences

In this notebook, we are going to see how to implement an election with **ordinal preferences** in our model of voters with embeddings.

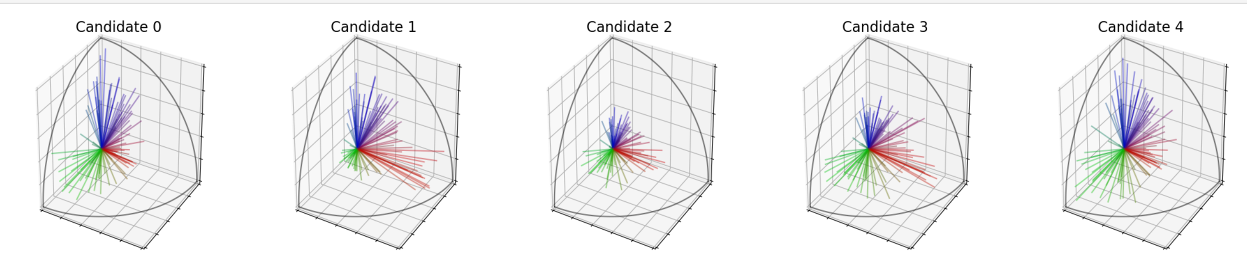
An election with **ordinal preferences** corresponds to an election in which each voter gives a **ranking** of the candidates instead of giving a different **score** to each candidate. It has been studied a lot and many rules exists for this model (*Plurality, Borda, k-approval, Condorcet, Instand Runoff, Maximin, etc.*).

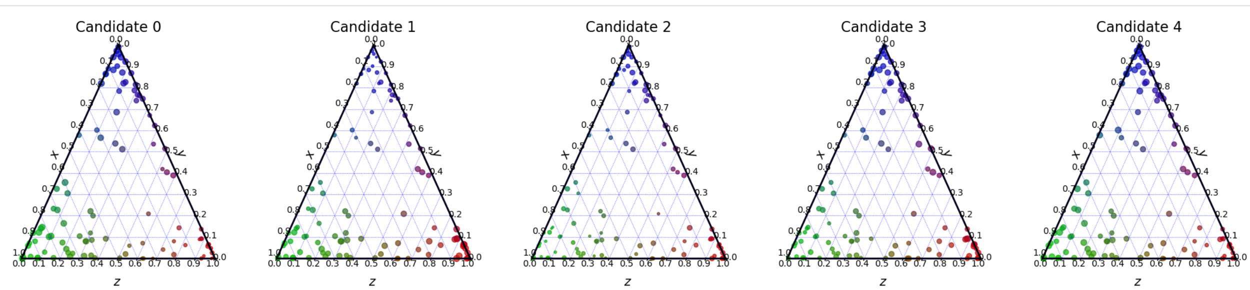
[1]:
```
import embedded_voting as ev
import numpy as np
import matplotlib.pyplot as plt
np.random.seed(442)
```

4.5.1 Classic election

Let's **run an election** with **5 candidates** and **100 voters**. We obtain the following profile:

[2]:
```
n_voters = 100
n_candidates = 5
n_dimensions = 3

embeddingsGen = ev.EmbeddingsGeneratorPolarized(n_voters, n_dimensions)
ratingsGen = ev.RatingsFromEmbeddingsCorrelated(coherence=0.6, n_dim=n_dimensions, n_
↪candidates=n_candidates)

embeddings = embeddingsGen(polarisation=0.5)
profile = ratingsGen(embeddings)

embeddings.plot_candidates(profile, "3D")
embeddings.plot_candidates(profile, "ternary")
```





[3]:
```
election = ev.RuleSVDNash()
election(profile, embeddings)
```

[3]:
```
<embedded_voting.rules.singlewinner_rules.rule_svd_nash.RuleSVDNash at 0x19b7b143240>
```

We can also print **all the information** about the results of this rule:

[4]:
```
print('Scores : ', election.scores_)
print('Ranking : ', election.ranking_)
print('Winner : ', election.winner_)
print('Welfare : ', election.welfare_)
```

```
Scores :  [53.97439136824531, 47.0012710723997, 27.897020264969875, 54.46431821175544,
↪ 63.55019310806972]
Ranking :  [4, 3, 0, 1, 2]
Winner :  4
Welfare :  [0.7314179643431743, 0.5358359238181288, 0.0, 0.7451594298129144, 1.0]
```

4.5.2 Positional scoring rules

Now, let's assume that instead of asking a **score vector** to each voter, we ask for a **ranking** of the candidate, and apply some rule with all the rankings.

A broad family of rule are **positional scoring rule**. A positional scoring rule is characterized by a vector $p = (p_1, \dots, p_m)$ such that each voter v_i gives p_j points to the voters with rank j. The winner is the candidate with the maximum total score.

We can adapt this idea to scores between 0 and 1 by setting the score given by the voter v_i to candidate c_j as $\frac{p_k}{p_n}$ if the candidate c_j is ranked at position k in the ranking of v_i.

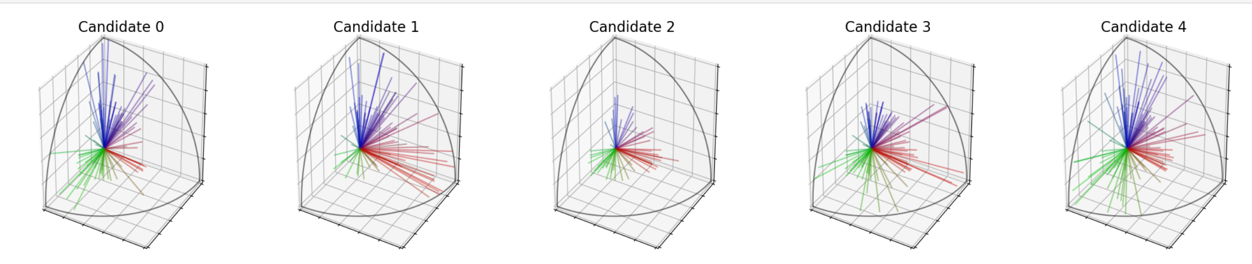
For instance, if the positional scoring rule is $(2, 1, 1, 1, 0)$, each voter gives a score of 1 to her favorite candidate, 0 to her least favorite candidate and $\frac{1}{2}$ to every other candidate:

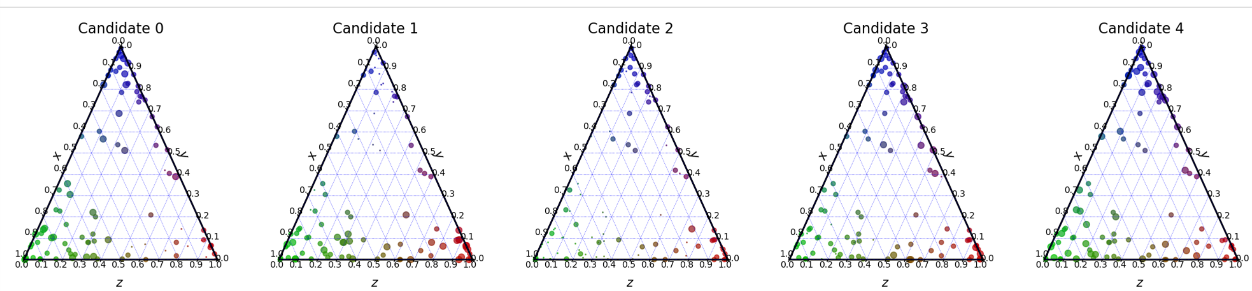
```
[5]: ordinal_election = ev.RulePositional([2, 1, 1, 1, 0], rule=ev.RuleSVDNash())
     ordinal_election(profile, embeddings)
```

```
[5]: <embedded_voting.rules.singlewinner_rules.rule_positional.RulePositional at␣
     ↪0x19b78b52710>
```

If we plot the profile of the candidates now, it is very different than before:

```
[6]: ordinal_election.plot_fake_ratings("3D")
     ordinal_election.plot_fake_ratings("ternary")
```





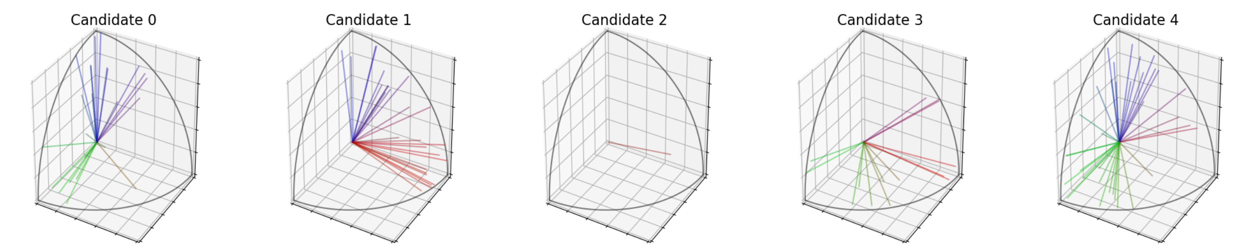

```
[7]: print('Scores : ', ordinal_election.score_(1))
     print('Ranking : ', ordinal_election.ranking_)
     print('Winner : ', ordinal_election.winner_)
```

```
Scores :  36.58919819094699
Ranking :  [4, 3, 0, 1, 2]
Winner :  4
```

Plurality

Plurality is the positional scoring rule defined by the scoring vector $(1, 0, \dots, 0)$. It is equivalent to saying that each voter only vote for his favorite candidate. We can see that in that case, almost nobody voted for candidate c_4:

[8]:
```
plurality_election = ev.RulePositionalPlurality(n_candidates, rule=ev.RuleSVDNash())
plurality_election(profile, embeddings)
plurality_election.plot_fake_ratings("3D")
```



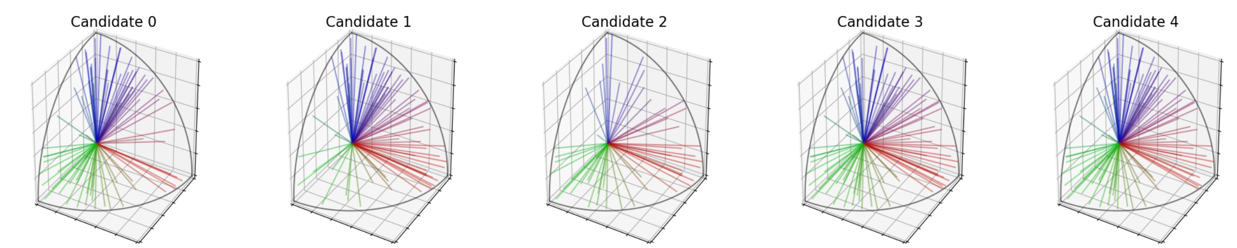
[9]:
```
print('Scores : ', plurality_election.scores_)
print('Ranking : ', plurality_election.ranking_)
print('Winner : ', plurality_election.winner_)
```

```
Scores :  [9.491165112035668, 6.254785145245596, 0j, 5.271663297309905, 19.
↪956241701329688]
Ranking :  [4, 0, 1, 3, 2]
Winner :  4
```

Veto

The **Veto** is the opposite of Plurality. In this rule, every voter votes for all candidates **but one**. That is why it looks like every candidate is liked by a lot of voters:

[10]:
```
veto_election = ev.RulePositionalVeto(n_candidates, rule=ev.RuleSVDNash())
veto_election(profile, embeddings)
veto_election.plot_fake_ratings("3D")
```



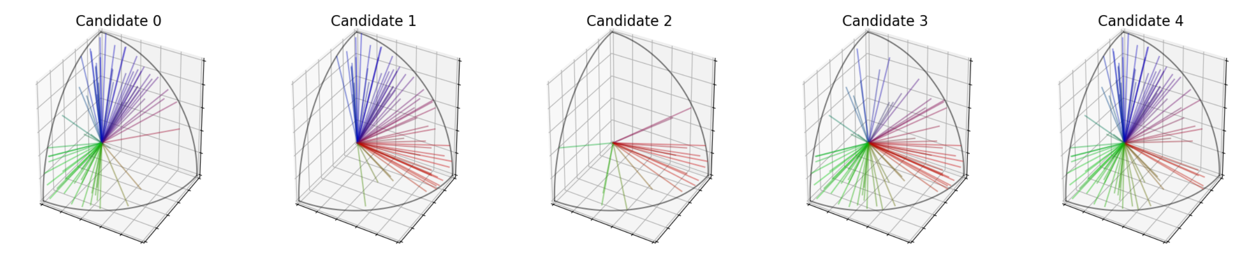
[11]:
```
print('Scores : ', veto_election.scores_)
print('Ranking : ', veto_election.ranking_)
print('Winner : ', veto_election.winner_)
```

```
Scores :  [89.53416233306291, 71.28984292198523, 45.10431048514931, 107.
↪41444287970856, 115.7630932057704]
Ranking :  [4, 3, 0, 1, 2]
Winner :  4
```

k-Approval

K-approval is the rule in between Plurality and Veto. Each voter votes for his **k** favorite candidates only. For instance, with $k = 3$:

```
[12]: kapp_election = ev.RulePositionalKApproval(n_candidates, k=3, rule=ev.RuleSVDNash())
kapp_election(profile, embeddings)
kapp_election.plot_fake_ratings("3D")
```



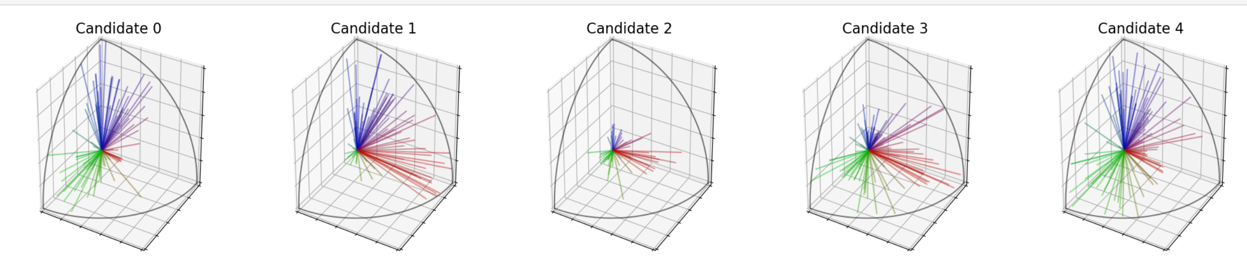

```
[13]: print('Scores : ', kapp_election.scores_)
print('Ranking : ', kapp_election.ranking_)
print('Winner : ', kapp_election.winner_)
```

```
Scores :  [54.598541290954344, 34.53096196174671, 4.654730263274076, 59.
↪83731740897562, 87.9835379096844]
Ranking :  [4, 3, 0, 1, 2]
Winner :  4
```

Borda

Borda use the scoring vector $(m - 1, m - 2, \ldots, 1, 0)$ where m is the total number of candidates.

```
[14]: borda_election = ev.RulePositionalBorda(n_candidates, rule=ev.RuleSVDNash())
borda_election(profile, embeddings)
borda_election.plot_fake_ratings("3D")
```




```
[15]: print('Scores : ', borda_election.scores_)
print('Ranking : ', borda_election.ranking_)
print('Winner : ', borda_election.winner_)
```

```
Scores :  [43.259760423282884, 32.05532869328766, 8.87628566982566, 46.
↪946353853379726, 66.4341676382639]
Ranking :  [4, 3, 0, 1, 2]
Winner :  4
```

4.5.3 Instant Runoff Voting (IRV)

Finally, we implemented **Instant Runoff Voting** which is not a positional scoring rule.

In this voting system, at each step, every voter votes for his favorite candidate, and the candidate with the lowest score **is eliminated**. Consequently, we perform $m - 1$ elections before we can find the winner. The ranking obtained is the inverse of the order in which the candidates are eliminated.

```
[16]: irv_election = ev.RuleInstantRunoff(rule=ev.RuleSVDNash())
      irv_election(profile, embeddings)
```

```
[16]: <embedded_voting.rules.singlewinner_rules.rule_instant_runoff.RuleInstantRunoff at␣
      ↪0x19b7b762748>
```

```
[17]: print('Ranking : ', irv_election.ranking_)
      print('Winner : ', irv_election.winner_)
```

```
Ranking :  [4, 0, 1, 3, 2]
Winner :  4
```

You can see that we can obtain different rankings depending on the ordinal voting rule that we use.

4.6 5. Manipulability analysis

For this project, we also looked at the manipulability of the voting rules we introduced in the previous notebook. More precisely, we wanted to see if **using ordinal extensions** with our voting rules would lower the degree of manipulability.

That's what we are going to see in this notebook, using the case of one of my favorite rules : **SVDNash**.

We analysed **two kinds** of manipulation :

- **Single-voter manipulation**
- **Coalition trivial manipulation**

I will explain these different manipulations in their respective sections.

```
[1]: import numpy as np
     import embedded_voting as ev
     import matplotlib.pyplot as plt
     np.random.seed(420)
```

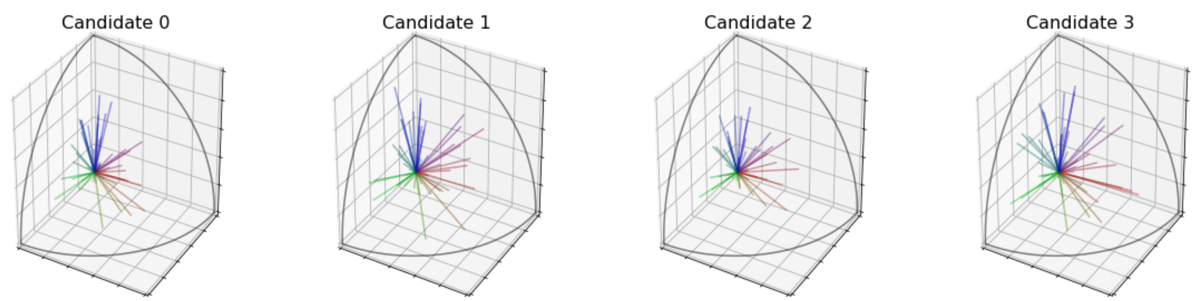
First of all, we **create a random profile** with three groups of voter of the same size.

```
[2]: n_voters = 50
     n_candidates = 4
     n_dim = 3

     embeddingsGenerator = ev.EmbeddingsGeneratorPolarized(n_voters, n_dim)
     ratingsGenerator = ev.RatingsFromEmbeddingsCorrelated(coherence=0.2,  n_dim = n_dim,␣
     ↪n_candidates = n_candidates)
```

(continues on next page)

(continued from previous page)

```
embeddings = embeddingsGenerator(polarisation=0.4)
ratings = ratingsGenerator(embeddings)

embeddings.plot_candidates(ratings)
```



4.6.1 Single-voter manipulation

The **single-voter manipulation** is easy to understand.

Let's say the winner of an election is candidate c_w and some voter v_i prefers candidate c_j to c_w (i.e. $c_j >_i c_w$). Then, v_i can **manipulate the election** by putting c_j first (even if it's not his favorite candidate) and c_w last. More generally, if v_i can change his preferences so that c_j becomes the winner instead of c_w, then v_i can manipulate the election for c_j.

The questions we want to ask :

- **What proportion** of the population can manipulate the election?
- What is the **average welfare** obtained after manipulation of the election by a voter?
- What is the **worst welfare** obtained after manipulation of the election by a voter?

No extensions

Let's create an election using the rule **SVDNash**. The winner is candidate c_4.

```
[3]: election = ev.RuleSVDNash()(ratings, embeddings)
print("Winner : ", election.winner_)
print("Ranking : ", election.ranking_)
print("Welfare : ", election.welfare_)
```

```
Winner :  3
Ranking :  [3, 1, 0, 2]
Welfare :  [0.24843681867948134, 0.7504122119950717, 0.0, 1.0]
```

With the class *SingleVoterManipulation*, I can answer the different questions about **the manipulability**.

To do so, when v_i manipulates as explained above, we only set the score of the candidate c_j to 1 and every other score is set to 0. It will work for **every monotonic rule** (which is the case of every rule we introduced).

For instance, in our election, a lot of the voters **can** manipulate the election, and the **worst welfare** that can be attained is the welfare of candidate c_1, which is **ranked second**.

[4]:
```
manipulation = ev.Manipulation(ratings, embeddings, election)
print("Is manipulable : ", manipulation.is_manipulable_)
print("Proportion of manipulators : ", manipulation.prop_manipulator_)
print("Average welfare after manipulation : ", manipulation.avg_welfare_)
print("Worst welfare after manipulation : ", manipulation.worst_welfare_)
```

```
Is manipulable :  False
Proportion of manipulators :  0.0
Average welfare after manipulation :  1.0
Worst welfare after manipulation :  1.0
```

Extensions

For **ordinal extension** (using *rankings*), we cannot use the above class, because if we set the score of every candidate to 0, we cannot rank the candidates anymore (they have the same score).

In the general case, we need to **test every possible ranking** for each voter. However, for some extension (*borda, k-approval, instant runoff*), we implemented **faster algorithms** for this.

For instance, with the **Borda extension** :

[5]:
```
ordinal_election = ev.RulePositionalBorda(n_candidates, rule=ev.RuleSVDNash())
ordinal_election(ratings, embeddings)
print("Winner : ", ordinal_election.winner_)
print("Ranking : ", ordinal_election.ranking_)
print("Welfare : ", ordinal_election.welfare_)
```

```
Winner :  3
Ranking :  [3, 1, 0, 2]
Welfare :  [0.33549316772419624, 0.7033412728596413, 0.0, 1.0]
```

Now, if we test the manipulability with *SingleVoterManipulationExtension* class, we reduce the number of manipulators

[6]:
```
ordinal_manipulation = ev.ManipulationOrdinal(ratings,
                                              embeddings,
                                              rule_positional=ordinal_election,
                                              rule=election)
print("Is manipulable : ", ordinal_manipulation.is_manipulable_)
print("Proportion of manipulators : ", ordinal_manipulation.prop_manipulator_)
print("Average welfare after manipulation : ", ordinal_manipulation.avg_welfare_)
print("Worst welfare after manipulation : ", ordinal_manipulation.worst_welfare_)
```

```
Is manipulable :  False
Proportion of manipulators :  0.0
Average welfare after manipulation :  1.0
Worst welfare after manipulation :  1.0
```

However, the above cell takes a lot of time (around 15 seconds). Using the specific class *SingleVoterManipulationBorda*, this computation time can be reduced to 0.5 seconds.

[7]:
```
borda_manipulation = ev.ManipulationOrdinalBorda(ratings,
                                                 embeddings,
                                                 rule=election)

print(borda_manipulation.extended_rule)
print("Is manipulable : ", borda_manipulation.is_manipulable_)
```

(continues on next page)

(continued from previous page)

```
print("Proportion of manipulators : ", borda_manipulation.prop_manipulator_)
print("Average welfare after manipulation : ", borda_manipulation.avg_welfare_)
print("Worst welfare after manipulation : ", borda_manipulation.worst_welfare_)
```

```
<embedded_voting.rules.singlewinner_rules.rule_positional_borda.RulePositionalBorda␣
↪object at 0x0000023B6FCD2E80>
Is manipulable :  False
Proportion of manipulators :  0.0
Average welfare after manipulation :  1.0
Worst welfare after manipulation :  1.0
```

Using **3-Approval**, we obtain less manipulators

```
[8]: approval_manipulation = ev.ManipulationOrdinalKApproval(ratings, embeddings, k=3,␣
↪rule=election)
print("Is manipulable : ", approval_manipulation.is_manipulable_)
print("Proportion of manipulators : ", approval_manipulation.prop_manipulator_)
print("Average welfare after manipulation : ", approval_manipulation.avg_welfare_)
print("Worst welfare after manipulation : ", approval_manipulation.worst_welfare_)
```

```
Is manipulable :  False
Proportion of manipulators :  0.0
Average welfare after manipulation :  1.0
Worst welfare after manipulation :  1.0
```

Using **Instant Runoff**, the profile is not manipulable

```
[9]: irv_manipulation = ev.ManipulationOrdinalIRV(ratings, embeddings, rule=election)
print("Is manipulable : ", irv_manipulation.is_manipulable_)
print("Proportion of manipulators : ", irv_manipulation.prop_manipulator_)
print("Average welfare after manipulation : ", irv_manipulation.avg_welfare_)
print("Worst welfare after manipulation : ", irv_manipulation.worst_welfare_)
```

```
Is manipulable :  False
Proportion of manipulators :  0.0
Average welfare after manipulation :  1.0
Worst welfare after manipulation :  1.0
```

4.6.2 Coalition manipulation

The second kind of manipulation that is easy to compute and represent is the **coalition manipulation**. More specifically, is there a **trivial manipulation by a coalition**?

Let's say that the winner of the election is the candidate c_w and let's name $V(j)$ the group of voters that prefer some candidate c_j to the winner c_w:

$$V(j) = \{v_i | c_j >_i c_w\}$$

Let's say now that all these voters set c_j first and c_w last. Is c_j **the new winner** of the election ? If the answer is **yes**, then the profile is manipulable by a trivial coalition.

Obviously, if the profile is **manipulable by a single voter**, then it is also **manipulable by a coalition**.

No extensions

When we don't use any extension, the profile is **very manipulable**. Indeed, every candidate can be elected after a trivial manipulation.

Consequently, the **worst Nash welfare** attainable is 0.

[10]:
```
manipulation = ev.ManipulationCoalition(ratings, embeddings, election)
print("Is manipulable : ", manipulation.is_manipulable_)
print("Worst welfare after manipulation : ", manipulation.worst_welfare_)
```

```
Is manipulable :  False
Worst welfare after manipulation :  1.0
```

Extensions

However, it **is a bit better** when we use an ordinal extension. You can use the general class *ManipulationCoalitionExtension* or the specific classes for *Borda*, *k-Approval* and *Instant runoff*. However, they use the same algorithm.

Using **Borda** extension

[11]:
```
borda_extension = ev.RulePositionalBorda(n_candidates, rule=ev.RuleSVDNash())
borda_manipulation = ev.ManipulationCoalitionOrdinal(ratings, embeddings, borda_
↪extension, election)
# manipulation = ev.ManipulationCoalitionBorda(profile, election)
print("Is manipulable : ", borda_manipulation.is_manipulable_)
print("Worst welfare after manipulation : ", borda_manipulation.worst_welfare_)
```

```
Is manipulable :  True
Worst welfare after manipulation :  0.7504122119950696
```

Using **3-Approval**

[12]:
```
kapp_manipulation = ev.ManipulationCoalitionOrdinalKApproval(ratings, embeddings, k=3,
↪ rule=election)
print("Is manipulable : ", kapp_manipulation.is_manipulable_)
print("Worst welfare after manipulation : ", kapp_manipulation.worst_welfare_)
```

```
Is manipulable :  False
Worst welfare after manipulation :  1.0
```

Finally, with **Instant Runoff voting**

[13]:
```
irv_manipulation = ev.ManipulationCoalitionOrdinalIRV(ratings, embeddings,␣
↪rule=election)
print("Is manipulable : ", irv_manipulation.is_manipulable_)
print("Worst welfare after manipulation : ", irv_manipulation.worst_welfare_)
```

```
Is manipulable :  False
Worst welfare after manipulation :  1.0
```

4.6.3 Manipulation maps

However, we cannot really judge a rule or an extension on one example. That's why we propose functions to show **manipulation maps** for some rule.

A map consists of an image of size $s \times s$ such that each pixel represents **one test**. A dark pixel represents a 0 and a yellow pixel represents a 1.

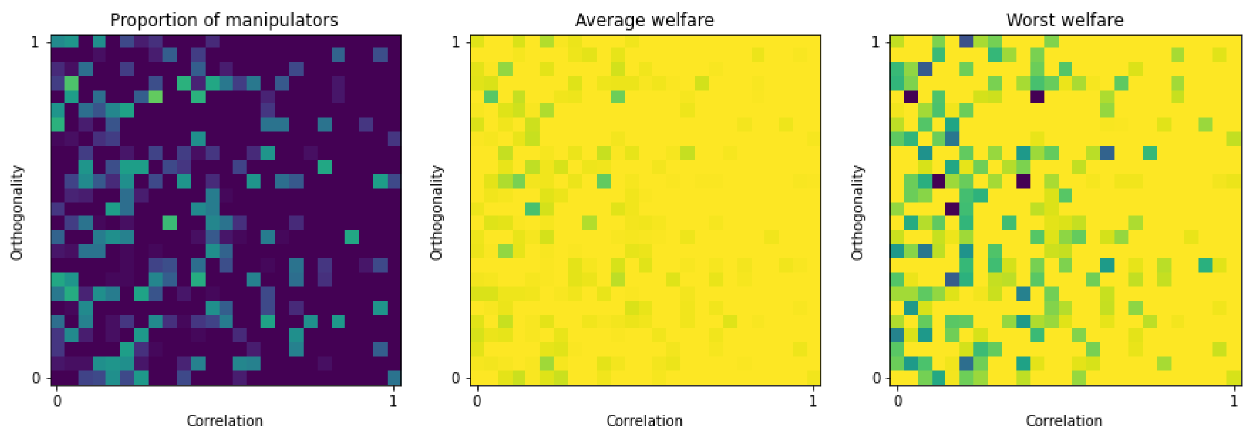
Moreover, we use a **parametric profile** for each test and we vary the *orthogonality* and the *correlation* of the parametric profiles for each test: The more the pixel is on the right, the higher the *correlation*, and the more the pixel is on the top, the higher the *orthogonality*.

For each test, a new *scores_matrix* is randomly generated for the parametric profile.

No extensions

For instance, if we do not use extensions, we can see that the profiles are not very manipulable by **single-voters**, and when this is the case, the **worst Nash welfare** is high.

[14]:
```
manipulation = ev.Manipulation(ratings, embeddings, election)
res = manipulation.manipulation_map(map_size=25)
```



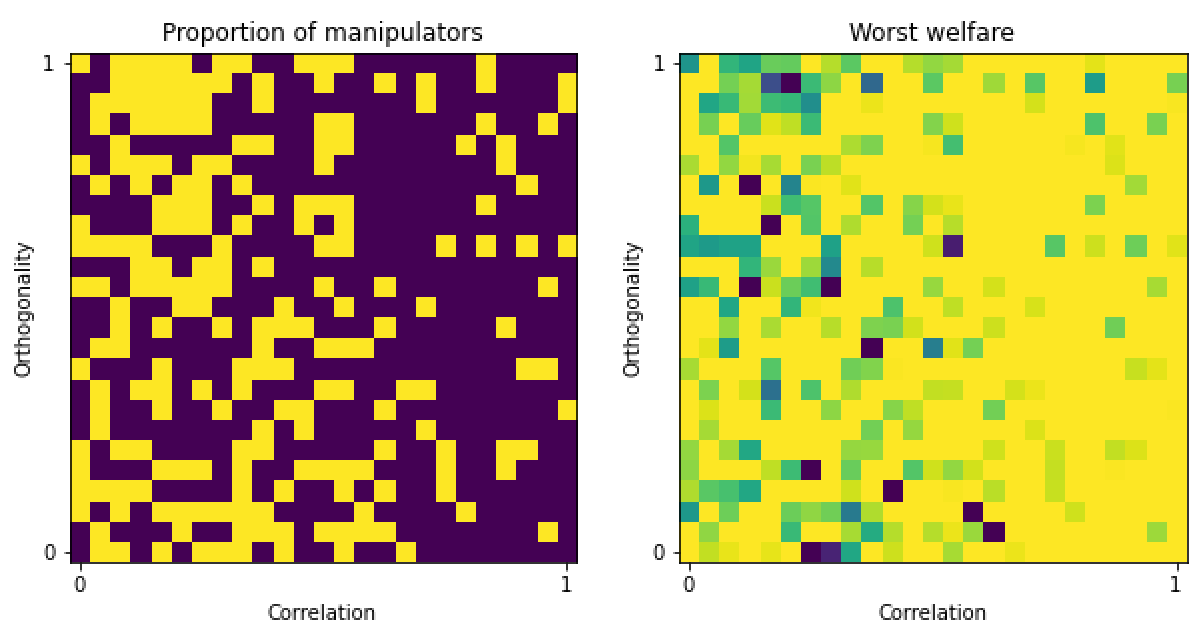
You can find **the data** used in the manipulation maps in the *output* of the function *manipulation_map()*.

[15]:
```
res.keys()
```

[15]:
```
dict_keys(['manipulator', 'worst_welfare', 'avg_welfare'])
```

However, almost every profile is manipulable **by trivial coalitions**, and often the **worst Nash welfare** is 0:

[16]:
```
manipulation = ev.ManipulationCoalition(ratings, embeddings, election)
res = manipulation.manipulation_map(map_size=25)
```



Again, the output of the function contains the data of the manipulation maps.

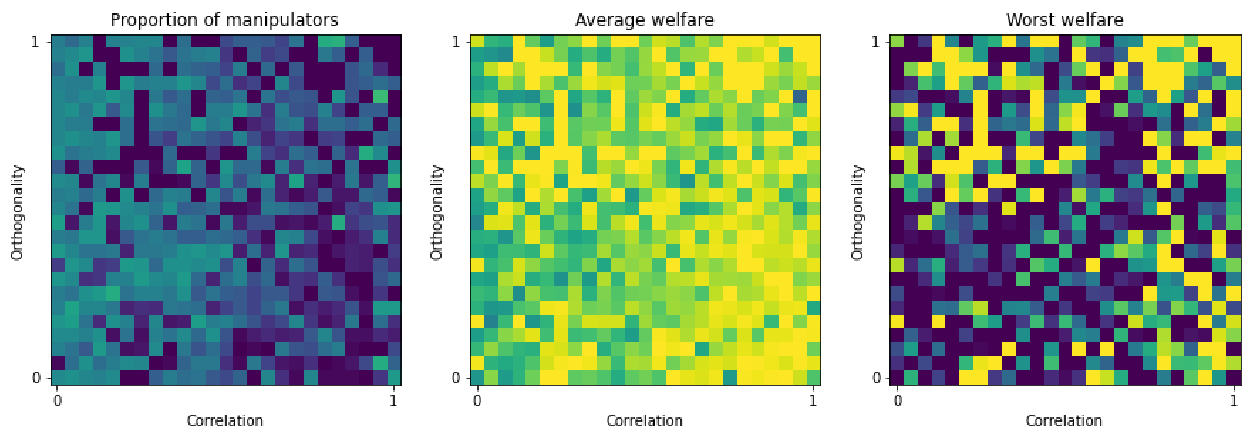
```
[17]: res.keys()
```

```
[17]: dict_keys(['manipulator', 'worst_welfare'])
```

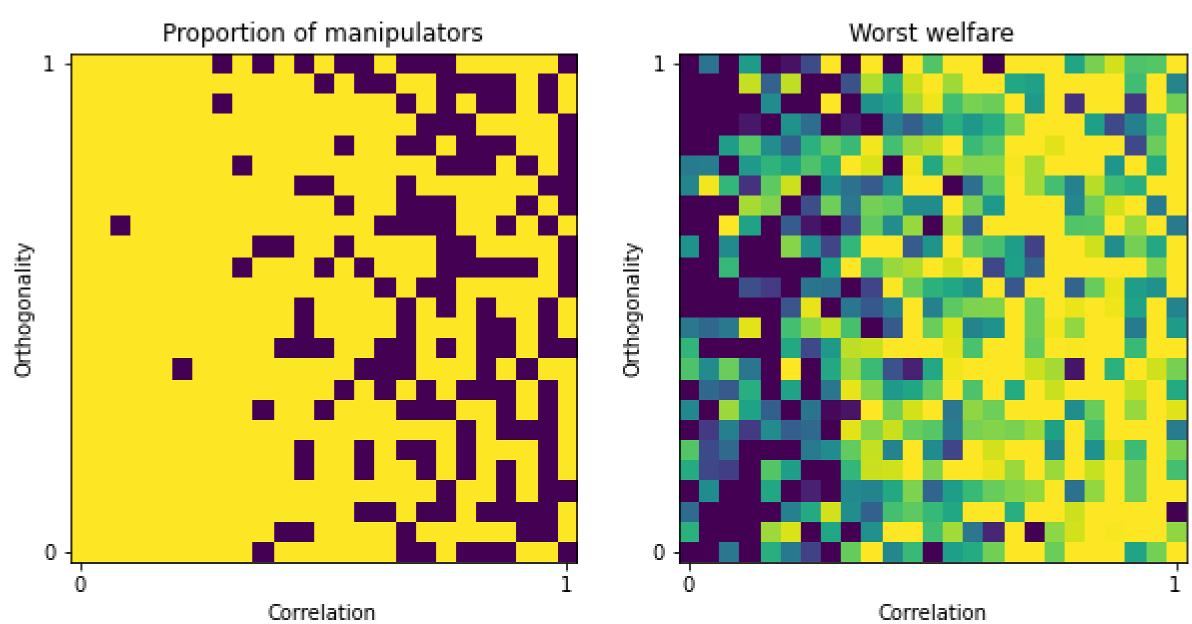
Borda

With **Borda**, we improve a little bit the resistance to **manipulation by coalition**. However, we decrease the resistance to **manipulation by a single-voter**.

```
[18]: borda_manipulation = ev.ManipulationOrdinalBorda(ratings, embeddings, rule=election)
res = borda_manipulation.manipulation_map(map_size=25)
```



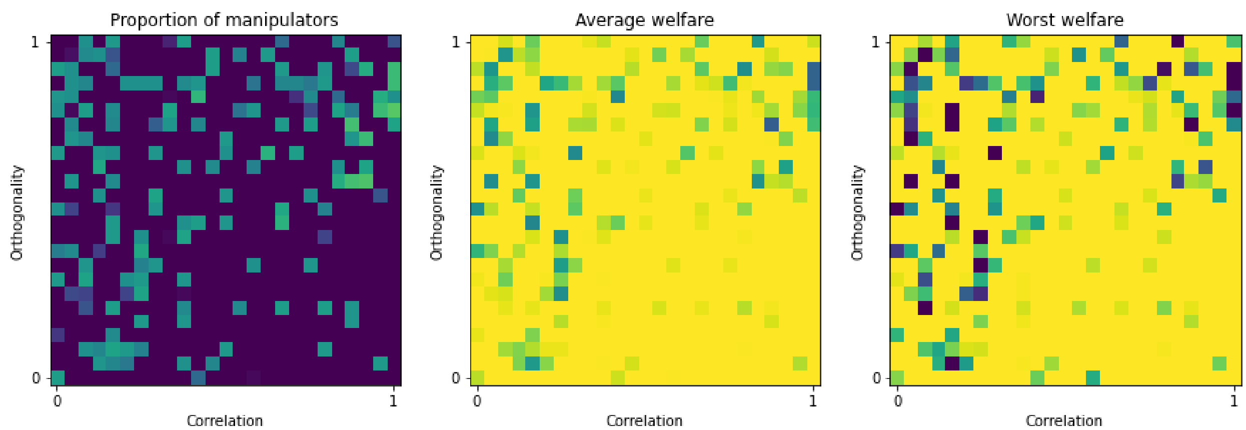

```
[19]: borda_manipulation = ev.ManipulationCoalitionOrdinalBorda(ratings, embeddings,␣
↪election)
res = borda_manipulation.manipulation_map(map_size=25)
```



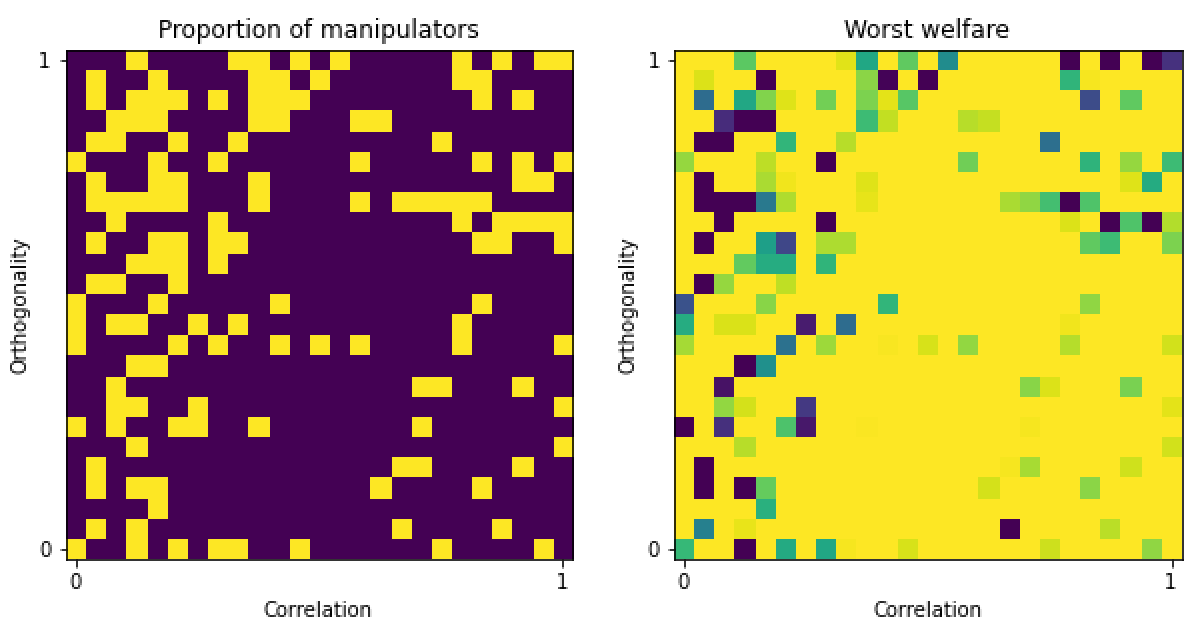
IRV

With **Instant Runoff**, we increase by **a lot** the resistance to **manipulation by coalition** without altering the resistance to **manipulation by a single voter**.

[20]:
```
irv_manipulation = ev.ManipulationOrdinalIRV(ratings, embeddings, rule=election)
res = irv_manipulation.manipulation_map(map_size=25)
```



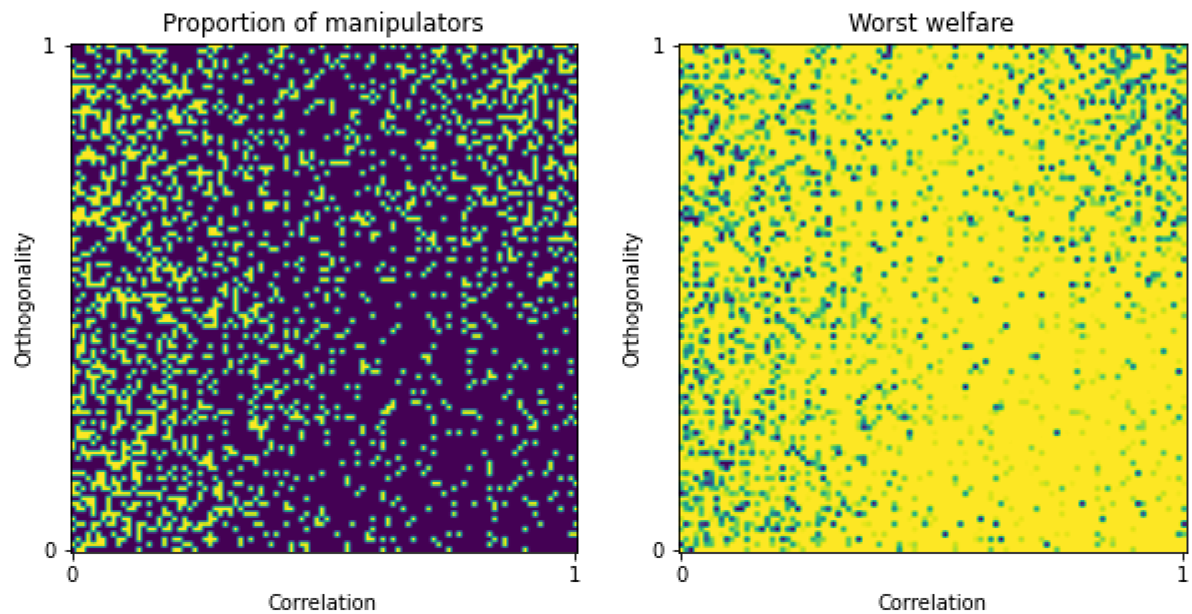
[21]:
```
irv_manipulation = ev.ManipulationCoalitionOrdinalIRV(ratings, embeddings, election)
res = irv_manipulation.manipulation_map(map_size=25)
```



change map size

You can also plot **more detailed** manipulation maps by changing the map_size parameter.

[22]:
```
irv_manipulation = ev.ManipulationCoalitionOrdinalIRV(ratings, embeddings, election)
res = irv_manipulation.manipulation_map(map_size=100)
```

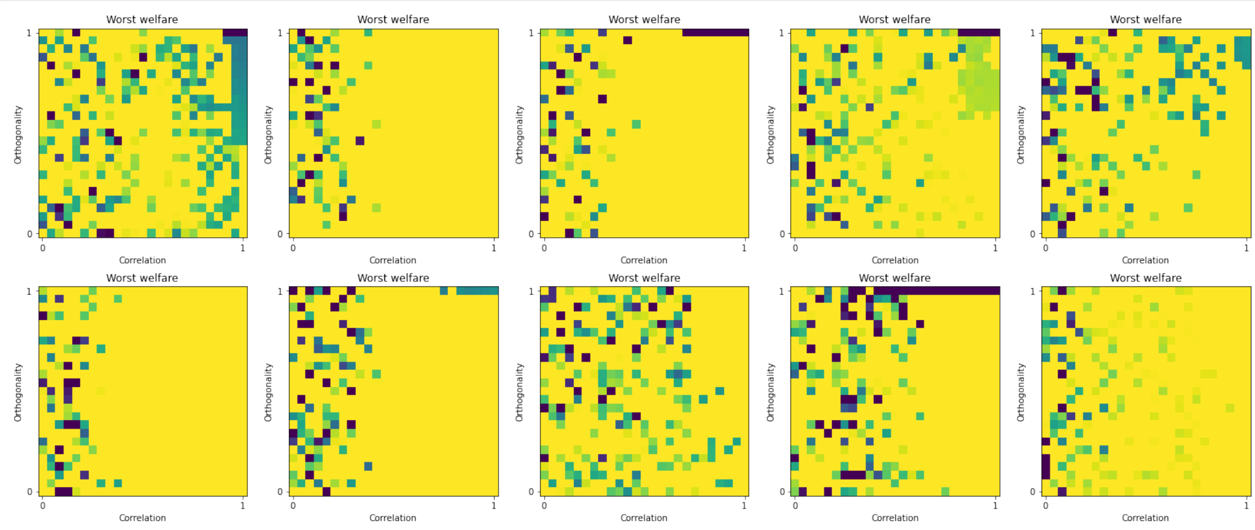


With particular scores matrix

In the previous plots, we changed the *scores_matrix* for each test (consequently, for each dot). You can modify that by specifying a matrix in the parameters of the function. It will use it for every test.

For instance, for **Instant Runoff**, you can see that the manipulation map heavily rely on this matrix: For some of them, there is a **dark spot** in the upper right corner (high *correlation* and high *orthogonality*).

[23]:
```
irv_manipulation = ev.ManipulationCoalitionOrdinalIRV(ratings, embeddings, election)
fig = plt.figure(figsize=(25, 10))
map_size = 25
for i in range(10):
    res = irv_manipulation.manipulation_map(map_size=map_size, ratings_dim_
↪candidate=np.random.rand(3, 4), show=False)
    image = res['worst_welfare']
    ev.create_map_plot(fig, image, [2, 5, i+1], "Worst welfare")
```



4.7 6. Multi-winner elections

We've already seen how to run a **single-winner election** with embedded voters. Now, I will explain how you can simulate **multi-winner election** on this framework.

I will explain in detail what you can do with the multi-winner rules *IterSVD()* and *IterFeatures()*. If you are interested and want to implement your own multi-winner rule, you can check the doc and the code for the class *IterRule()*.

[1]:
```
import embedded_voting as ev
import numpy as np
import matplotlib.pyplot as plt
np.random.seed(42)
```

4.7.1 Create the profile

You should start to be familiar with this part of the notebook. As always, we are creating **a profile of embedded voters**. This time, we have 20 candidates instead of 5, because it enables us to see what our multi-winner rules really do.

To do so, I simply use the candidates from the profile of the **Notebook 2**, and duplicate them 3 times. The profile is as follows :

- The **red group** contains **50%** of the voters, and the average scores of candidates given by this group are $[0.9, 0.3, 0.5, 0.2, 0.2]$.
- The **green group** contains **30%** of the voters, and the average scores of candidates given by this group are $[0.2, 0.6, 0.5, 0.3, 0.9]$.
- The **blue group** contains **20%** of the voters, and the average scores of candidates given by this group are $[0.2, 0.6, 0.5, 0.9, 0.3]$.

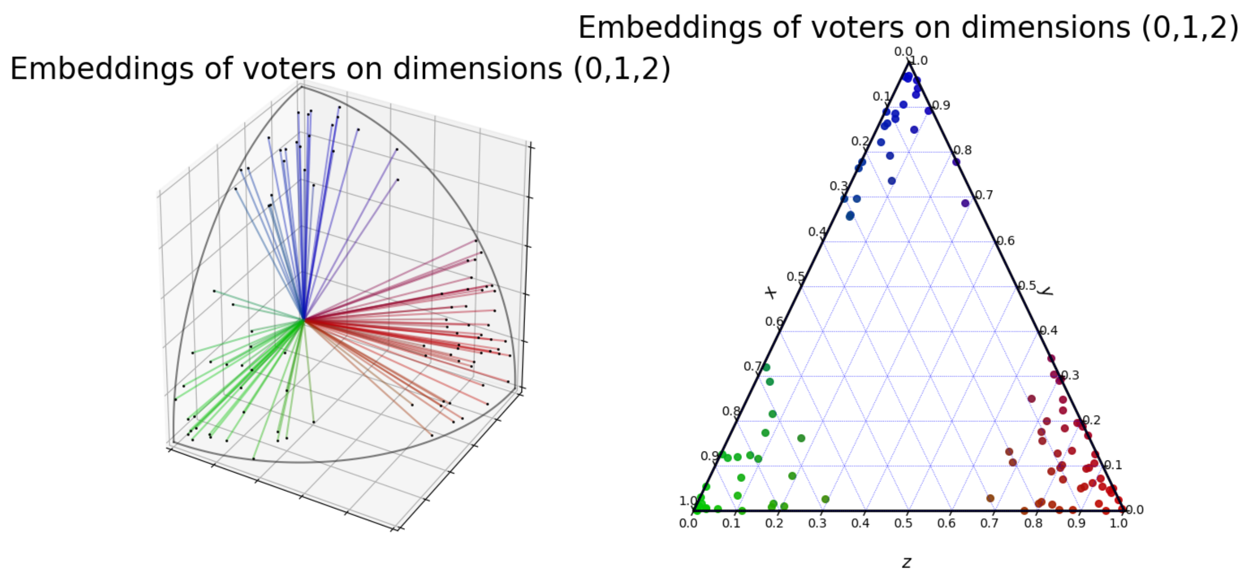
In that way, candidates $(0, 5, 10, 15)$ are candidates of the **red group**, candidates $(4, 9, 14, 19)$ are candidates of the **green group**, candidates $(3, 8, 13, 18)$ are candidates of the **blue group**, and all other candidates are more *"consensual"*.

In the following cell, I create my profile using *ParametricProfile()*:

```
[2]: scores_matrix_bloc = np.array([[.9, .3, .5, .3, .2], [.2, .6, .5, .3, .9], [.2, .6, .
     ↪5, .9, .3]])
     scores_matrix = np.concatenate([scores_matrix_bloc]*4, 1)

     proba = [.5, .3, .2]

     n_voters = 100
     n_dimensions, n_candidates = np.array(scores_matrix).shape
     embeddingsGen = ev.EmbeddingsGeneratorPolarized(n_voters, n_dimensions, proba)
     ratingsGen = ev.RatingsFromEmbeddingsCorrelated(0.8, scores_matrix,  n_dimensions,n_
     ↪candidates )

     embeddings = embeddingsGen(polarisation=0.6)
     profile = ratingsGen(embeddings)
```

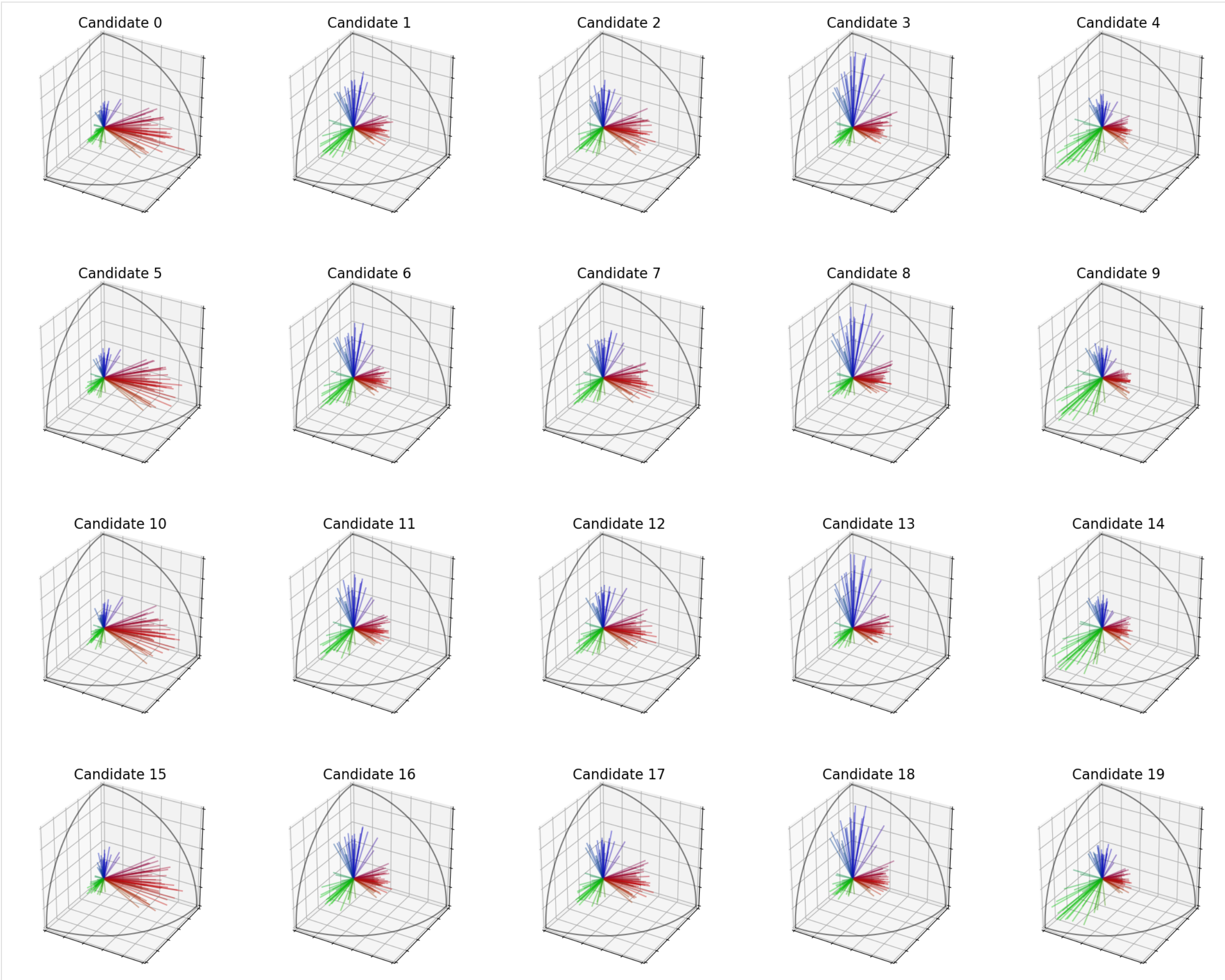
The following cell **displays the profile distribution**:

```
[3]: fig = plt.figure(figsize=(15,7.5))
     embeddings.plot("3D", fig=fig, plot_position=[1,2,1], show=False)
     embeddings.plot("ternary", fig=fig, plot_position=[1,2,2], show=False)
     plt.show()
```



This cell displays **the candidates**. You can see that each column contains very similar candidates.

```
[4]: embeddings.plot_candidates(profile, "3D")
```



4.7.2 How IterRule work (a bit of theory)

Our goal was to elaborate rules that respects the proportionality with respect to both **the scores and the embeddings** of the voters. That means that if a group of voter with similar embeddings represent 25% of the population, 25% of the winning committee should be composed of their **favorite candidates**

To do so, we implemented an adaptation of *Single Transferable Vote (STV)* to profiles with embedded voters.

First, **some notations** :

Notations	Meaning
v_i	The i^{th} voter
c_j	The j^{th} candidate
$s_i(c_j)$	The score given by the voter v_i to the candidate c_j
M	The embeddings matrix, such that M_i are the embeddings of v_i
k	The size of the winning committee
$0 \leq t < k$	The iteration number
$w_i(t)$	The weight of the voter v_i at time t
$W(t)$	The t^{th} candidate of the winning committee
$sat_i(t)$	The satisfaction of voter v_i with candidate $W(t)$

The rules we are using in this notebook are following this algorithm:

- At the initialization, the weight of all voters is equal to $w_i(0) = 1$.
- At each step $t \in [0, k-1[$:
  - Apply a **voting rule** on the profile defined by the scores $(w_i \times s_i)$ and the embeddings matrix M. The voting rule should return a score and a feature vector for each candidate. Select the candidate c not yet in the committee with the maximum score. This will be the winner $W(t)$. Let's denote $v(t)$ the feature vector of this candidate.
  - We compute **the satisfaction** of every voter with the new candidate, which is defined as

$$s_i(W(t)) \times \cos(v(t), M_i)$$

  where cos is the **cosine similarity**. Therefore, a voter with embeddings close to the candidate's features will be more satisfied than a candidate orthogonal to the features of the candidate.

  - **Update the weights** of the voters according to their level of satisfaction. The sum of all the removed weights should be equal to **a quota** of voter, for instance $\frac{n}{k}$.

At the end, the weights of all voters should be close to 0.

In this notebook, I will present two rules based on this algorithm : **IterSVD** and **IterFeatures**.

- **IterSVD** uses a *SVD based* rule presented on the notebook 2 to determine the scores of the candidates. The feature vector is the vector of the *SVD* associated to the **largest singular value**. A very well-suited aggregation function to achieve proportionnality is the **maximum** function.
- **IterFeatures** is based on the *Features* rule presented on the notebook 2. The notion of feature vector in that case is straightforward.

Now, let's see how it works!

4.7.3 Run an election

The following cell shows how you **instantiate a multi-winner election**.

Here we want a committee of size $k = 5$ and we are using the *classic* method of quota (see next section).

[5]:
```
election = ev.MultiwinnerRuleIterSVD(k=5, quota="classic")
election(profile, embeddings)
```

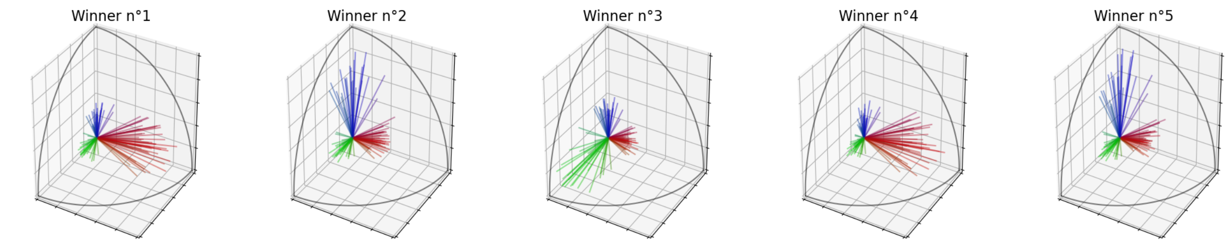
[5]:
```
<embedded_voting.rules.multiwinner_rules.multiwinner_rule_iter_svd.
↪MultiwinnerRuleIterSVD at 0x20aa08c0f28>
```

You can immediately **print** and **plot** the winning committee.

In our case, the committee contains 3 candidates of the **red group**, 2 candidates of the **green group** and 0 of the **blue group**. This gives us proportion of $(40\%, 40\%, 20\%)$ instead of the correct proportionality $(60\%, 40\%, 0\%)$.

[6]:
```
print("Winners : ",election.winners_)
election.plot_winners()
```

```
Winners :  [5, 18, 19, 15, 3]
```

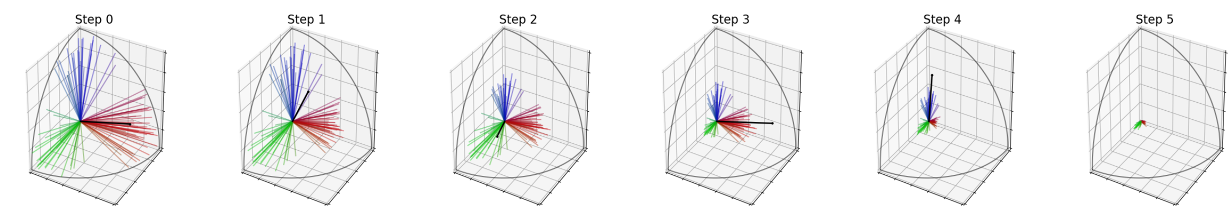


The following cell shows how to plot **the evolution of the voters' weights** with time. The black vector represents the **feature vector** of the candidate selected at this step.

For instance, the first candidate is liked by the **red group**, so its vector is very similar to the vectors of voters in this group, and you can see that the weight of every voter of this group is reduced during step 2.

[7]:
```
election.plot_weights(row_size=6)
```

```
Weight / remaining candidate :  [20.0, 20.0, 20.000000000000004, 20.000000000000004,␣
↪20.000000000000014]
```



4.7.4 Changing some parameters

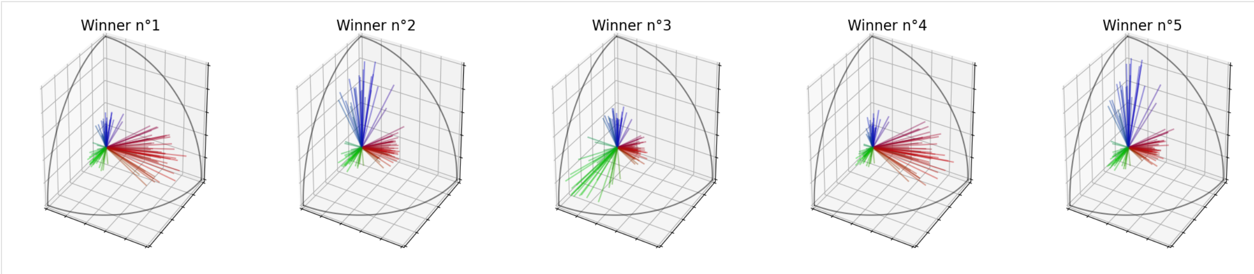
You can change the method of quota using the function *set_quota()*. There are two possible quotas:

- *Classic* quota $Q = \frac{n}{k}$.
- *Droop* quota $Q = \frac{n}{k+1} + 1$.

However, there is **not a big difference** between the results if we use on quota or another. For instance, with $k = 5$, we obtain the **same committee** as before:

[8]:
```
election.set_quota("droop")
print("Winners : ",election.winners_)
election.plot_winners()
```

```
Winners :  [5, 18, 19, 15, 3]
```

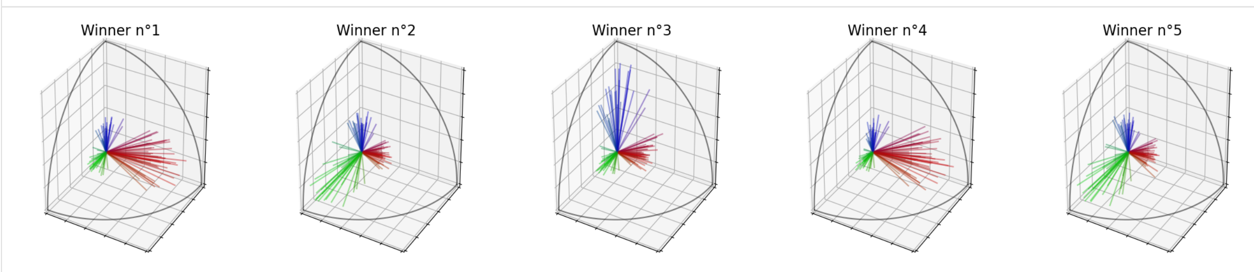


You can also change parameters related to the *SVD*, for instance the **square_root** parameter, which can influence the results.

Indeed, as you can see on the following cell, when we are not using the square root of the score, we **give more opportunities to small groups** (like the group of blue voters).

[9]:
```
election = ev.MultiwinnerRuleIterSVD(k=5, quota="classic", square_root=False)
election(profile, embeddings)
print("Winners : ",election.winners_)
election.plot_winners()
```

```
Winners :  [5, 19, 3, 15, 14]
```



4.7.5 Changing the size of the committee

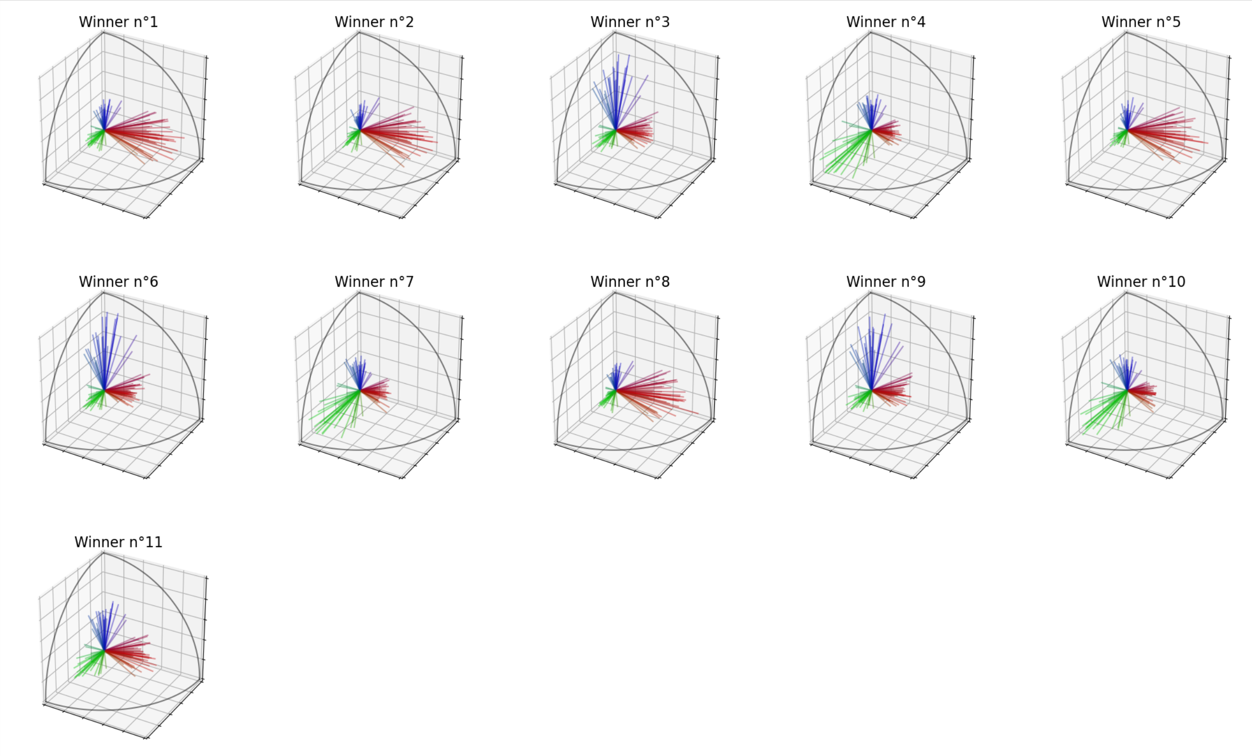
You can **change the size** of the winning committee, by calling the function *set_k()*.

As you can see, if we set the size of the committee to $k = 11$ candidates, There are 4 candidates of the **red group** (that is the maximum possible), there are also 4 candidates of the **green group**, and 2 candidates of the **blue group**. The last candidate *(actually Winner 10)* is a *consensual* candidate.

The **proportions obtained** are close to the real proportions $(50\%, 30\%, 20\%)$.

[10]:
```
election = ev.MultiwinnerRuleIterSVD()
election(profile, embeddings)
election.set_k(11)
print("Winners : ",election.winners_)
election.plot_winners()
```

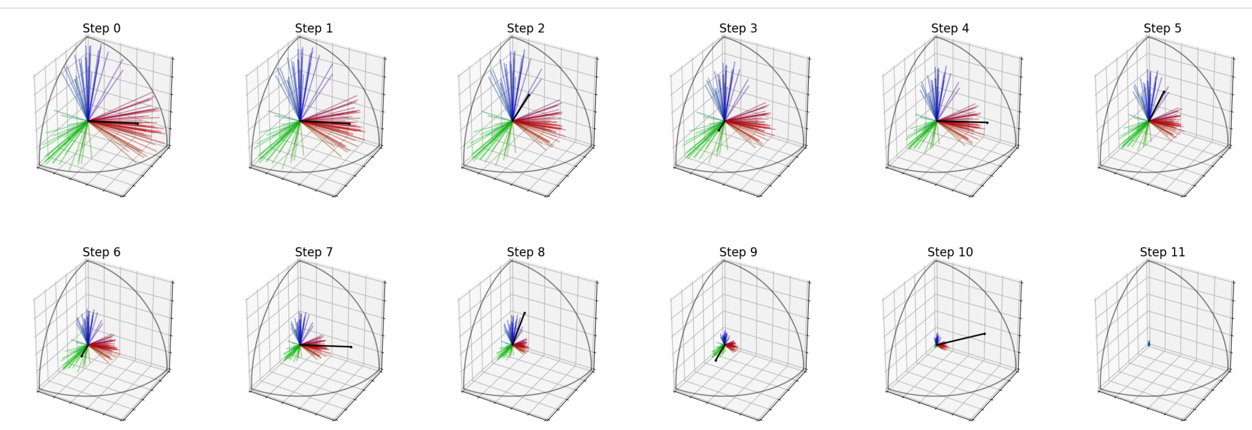
```
Winners :  [5, 10, 18, 19, 15, 3, 14, 0, 8, 9, 7]
```



As before, we can display the evolutions of **the weights** of the voters, with the feature of the selected candidate at each step in **black**. In the end, all weights are almost 0.

[11]:
```
election.plot_weights(row_size=6)
```

```
Weight / remaining candidate :  [9.090909090909092, 9.09090909090909, 9.
↪090909090909092, 9.09090909090909, 9.090909090909092, 9.090909090909092, 9.
↪09090909090909, 9.09090909090909, 9.090909090909088, 9.090909090909088, 9.
↪090909090909085]
```

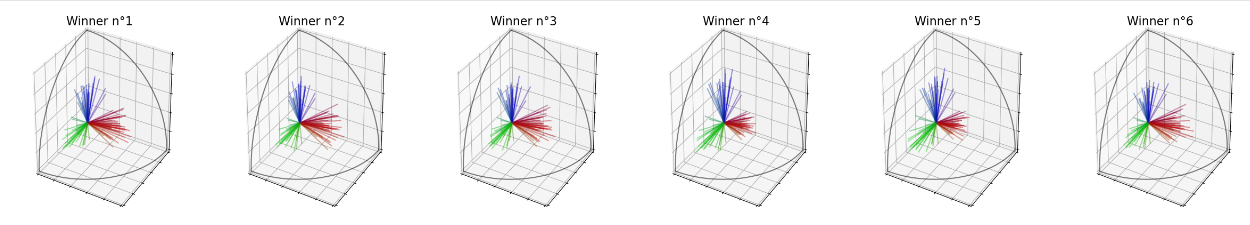


4.7.6 Using other SVD Rules

You can use another rule than **the maximum** for the *IterSVD()* function. However, the maximum is well suited for proportionality, which is not the case for other aggregation functions. For instance, with **the product**, we only obtain *consensual* candidates, as it is shown in the following cell:

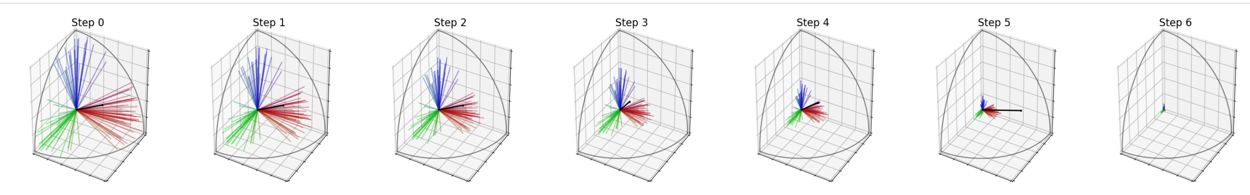
[12]:
```
election = ev.MultiwinnerRuleIterSVD(k=6, quota="classic", aggregation_rule=np.prod)
election(profile, embeddings)
print("Winners : ",election.winners_)
election.plot_winners(row_size=6)
```

```
Winners :  [7, 2, 17, 6, 1, 12]
```



[13]:
```
election.plot_weights(row_size=7)
```

```
Weight / remaining candidate :  [16.666666666666668, 16.666666666666664, 16.
↪666666666666664, 16.666666666666664, 16.666666666666664, 16.666666666666654]
```



4.7.7 IterSVD versus IterFeatures

Everything I explained earlier for **IterSVD** also works for **IterFeatures**. Let's see how the two rules compare on some examples:

[14]:
```
election_svd = ev.MultiwinnerRuleIterSVD(k=5)
election_features = ev.MultiwinnerRuleIterFeatures(k=5)

election_svd(profile, embeddings)
election_features(profile, embeddings)
```

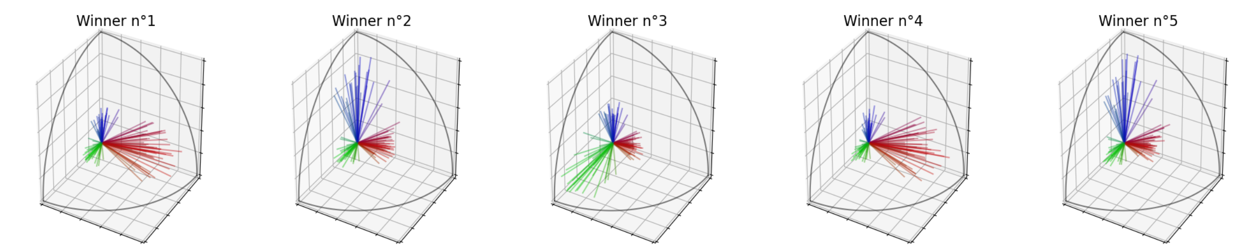
[14]:
```
<embedded_voting.rules.multiwinner_rules.multiwinner_rule_iter_features.
↪MultiwinnerRuleIterFeatures at 0x20a9c67e860>
```

With $k = 5$, the proportions achieved by **IterFeatures** are $(40\%, 40\%, 20\%)$, which is more egalitarian than the $(60\%, 40\%, 0\%)$ achieved by **IterSVD**.

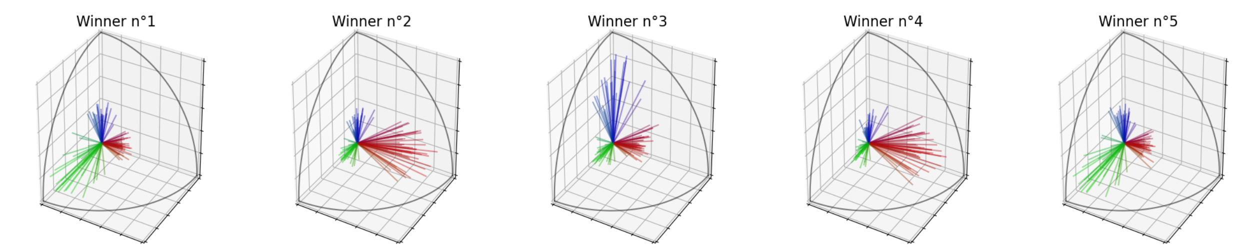
It is due to the fact that **IterSVD** takes into account the size of each each group to choose a winner but not **IterFeatures**. For the latter, the size of the group only appears when we update the weights of the voters.

[15]:
```
print("Winners (SVD) : ",election_svd.winners_)
election_svd.plot_winners()
print("Winners (Features) : ",election_features.winners_)
election_features.plot_winners()
```

```
Winners (SVD) :  [5, 18, 19, 15, 3]
```




```
Winners (Features) :  [19, 5, 3, 10, 14]
```



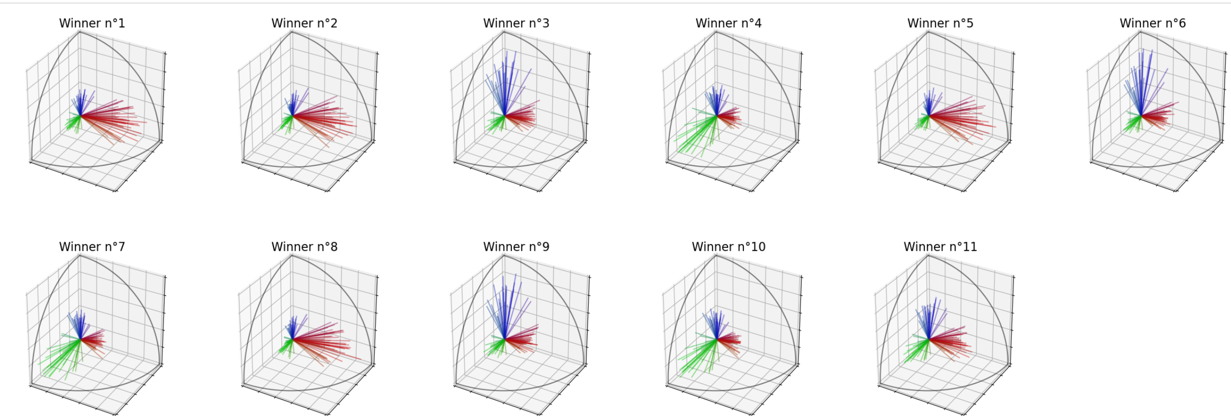
It is even clearer with $k = 11$. As you can see in the following cells, the first two candidates selected by **IterFeatures** are a green candidate and a blue candidate, even if the red group is the biggest.

[16]:
```
election_svd.set_k(11)
election_features.set_k(11)
```

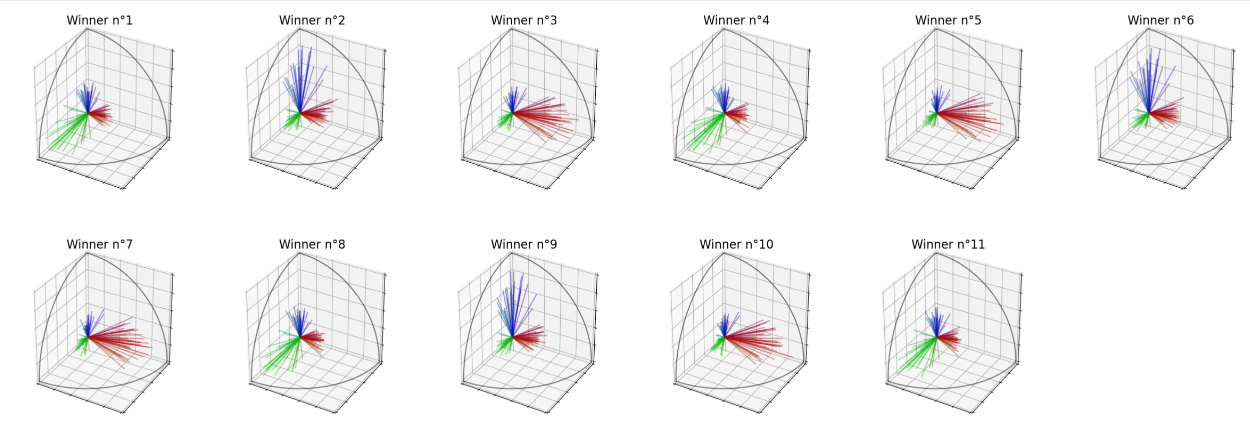
[16]:
```
<embedded_voting.rules.multiwinner_rules.multiwinner_rule_iter_features.
↪MultiwinnerRuleIterFeatures at 0x20a9c67e860>
```

[17]:
```
print("Winners (SVD) : ",election_svd.winners_)
election_svd.plot_winners(row_size=6)
print("Winners (Features) : ",election_features.winners_)
election_features.plot_winners(row_size=6)
```

```
Winners (SVD) :  [5, 10, 18, 19, 15, 3, 14, 0, 8, 9, 7]
```

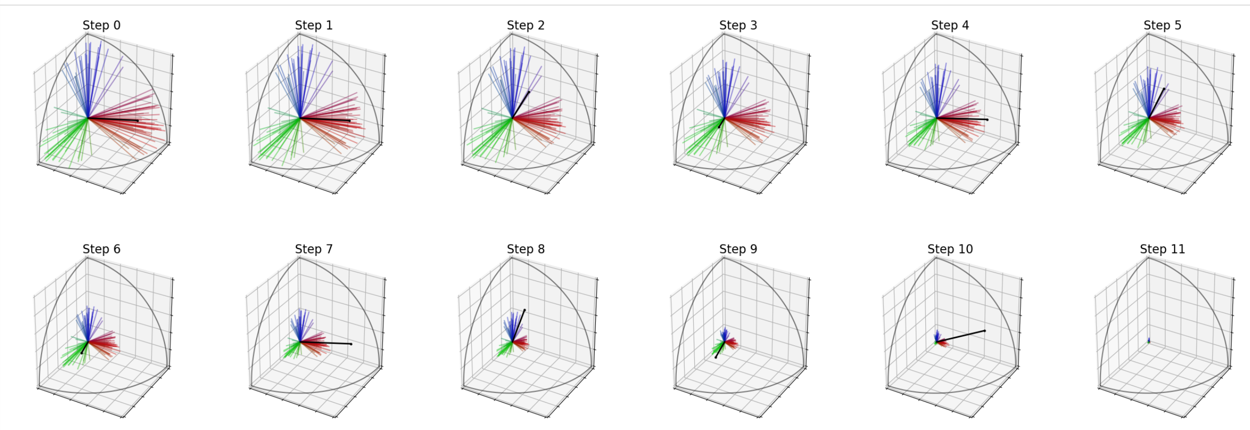



```
Winners (Features) :  [19, 3, 5, 14, 15, 18, 10, 9, 13, 0, 4]
```

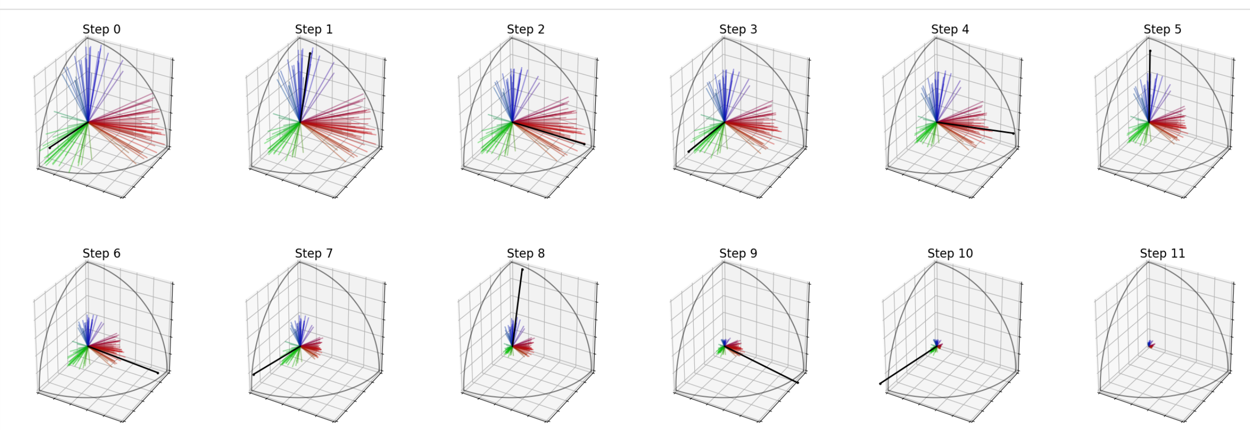


[18]:
```
print("IterSVD")
election_svd.plot_weights(row_size=6, verbose=False)
print("IterFeatures")
election_features.plot_weights(row_size=6, verbose=False)
```

```
IterSVD
```




```
IterFeatures
```



4.8 7. Algorithms aggregation

In this notebook we will compare the different voting rules on an online learning scenario. We have different aggregators with different scoring rules, and each aggregator start with 0 training data. Then each aggregator use the data from the successive aggregations to train the embeddings.

[1]:
```
import numpy as np
import embedded_voting as ev
import matplotlib.pyplot as plt
from tqdm import tqdm
np.random.seed(42)
```

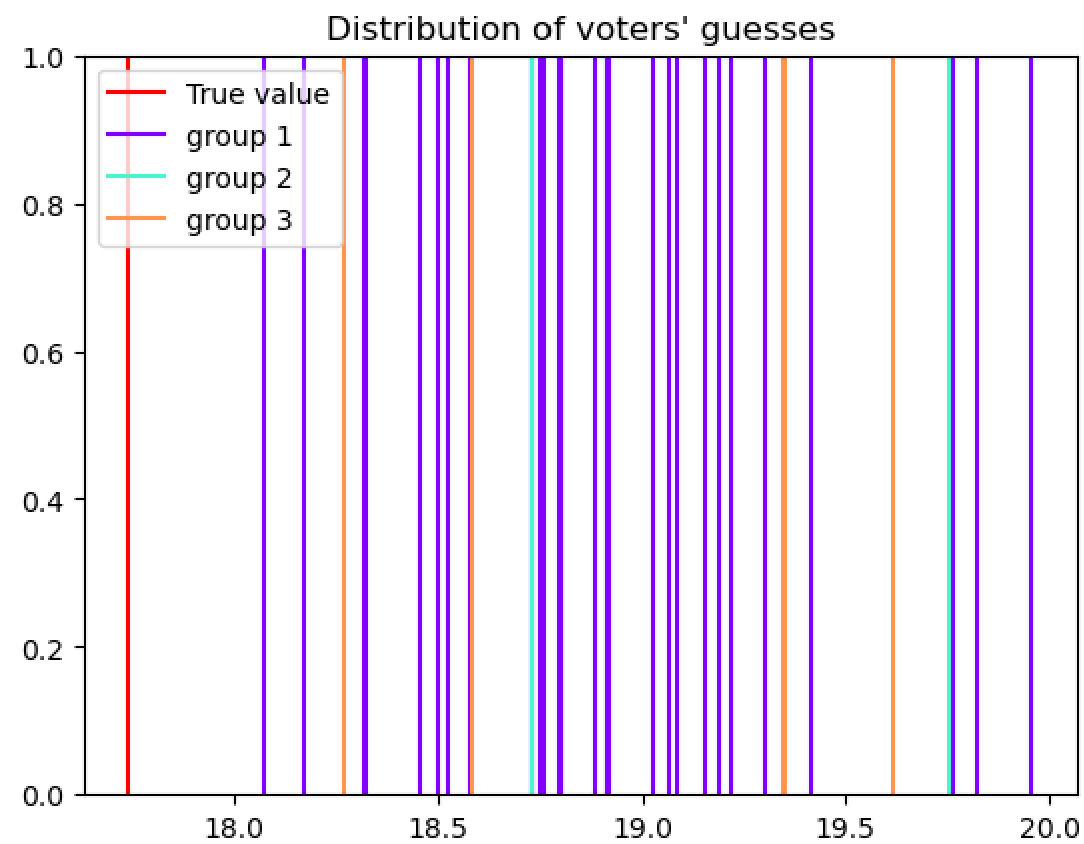
We will comapre 5 rules : *FastNash*, *FastSum*, *SumScores*, *ProductScores*, *MLEGaussian*

[2]:
```
def create_f(A):
    def f(ratings_v, history_mean, history_std):
        return np.sqrt(np.maximum(0, A + (ratings_v - history_mean) / history_std))
    return f
```

[3]:
```
list_agg = [ev.Aggregator(rule=ev.RuleFastNash(f=create_f(0)),name="RuleFastNash"),
            ev.AggregatorFastSum(),
            ev.AggregatorSumRatings(),
            ev.AggregatorProductRatings(),
            ev.AggregatorMLEGaussian()]
```

For the generator, we use a model with 30 algorithms in the same group G_1, 2 algorithms in agroup G_2 and 5 algorithms between the two (but closer to G_2)

[4]:
```
groups_sizes = [30, 2, 5]
features = [[1, 0], [0, 1], [0.3,0.7]]

generator = ev.RatingsGeneratorEpistemicGroupsMix(groups_sizes, features, group_
↪noise=8, independent_noise=0.5)
generator.plot_ratings()
```



[5]:
```
onlineLearning = ev.OnlineLearning(list_agg, generator)
```

Each election contains 20 alternatives, we run 50 successive elections for each experiment and run this 1000 times.

[6]:
```
n_candidates = 20
n_steps = 50
n_try = 1000
onlineLearning(n_candidates, n_steps, n_try)
```

```
100%|| 1000/1000 [57:38<00:00,  3.46s/it]
```

Finally, we can display the result of the experiment

[7]:
```
onlineLearning.plot()
```

CHAPTER 5

IJCAI

You can find here the notebooks used to produce the experiments of the paper submitted to IJCAI.

5.1 Reference Scenario

The following packages are required to run this notebook. If you miss one, please install them, e.g. with `pip`.

[1]:
```
import numpy as np
import dill as pickle
import matplotlib.pyplot as plt
from tqdm import tqdm
import tikzplotlib
from multiprocess.pool import Pool
```

Our own module (use `pip install -e .` from the source folder to install it).

[2]:
```
import embedded_voting as ev # Our own module
```

Direct load of some useful variables and functions.

[3]:
```
from embedded_voting.experiments.aggregation import make_generator, make_aggs, f_max,␣
↪f_renorm
from embedded_voting.experiments.aggregation import handles, colors, evaluate,␣
↪default_order
```

5.1.1 Building Data

We first create the data for the reference scenario and the first experiments showed in the paper. In particular, we consider more agents and candidates per election than required.

In details:

- We create data for 10 000 simulations of decision.

- Each decision problem (a.k.a. election) involves 50 candidates each.
- Each election has 50 estimators (voters): 30 in one correlated group and 20 independent estimators.

For the reference scenario, only the first 20 candidates are considered, and only 24 estimators indexed from 10 to 34, giving 20 in the same group and 4 independents (the rest of the data is used in variations of this scenario).

The feature noise is a normal noise of standard deviation 1, while the distinct noise has standard deviation 0.1. The score generator (for the true score of the candidate) also follows a normal law of standard deviation 1.

The parameters:

```
[4]: n_tries = 10000 # Number of simulations
n_training = 1000 # Number of training samples for trained rules
n_c = 50
groups = [30]+[1]*20
```

The generator of estimations, from the *generator* class of our package:

```
[5]: generator = make_generator(groups=groups)
```

We create the datasets

```
[6]: data = {
    'training': generator(n_training),
    'testing': generator(n_tries*n_c).reshape(sum(groups), n_tries, n_c),
    'truth': generator.ground_truth_.reshape(n_tries, n_c)
}
```

We save them for further use later.

```
[7]: with open('base_case_data.pkl', 'wb') as f:
    pickle.dump(data, f)
```

5.1.2 Computation

We extract what we need from the dataset: 24 estimators (20+4x1) and 20 candidates.

```
[8]: n_c = 20
groups = [20] + [1]*4
n_v = sum(groups)
voters = slice(30-groups[0], 30+len(groups)-1)
training = data['training'][voters, :]
testing = data['testing'][voters, :, :n_c]
truth = data['truth'][:, :n_c]
```

We define the list of rules we want to compare. For the reference scenario we add the *Random* rule.

```
[9]: list_agg = make_aggs(groups, order=default_order+['Rand'])
```

We run the aggregation methods on all the 10 000 simulations, compute the average, and save the results.

```
[10]: with Pool() as p:
    res = evaluate(list_agg=list_agg, truth=truth, testing=testing, training=training,
↪ pool=p)
```

```
100%|| 10000/10000 [00:49<00:00, 201.93it/s]
```

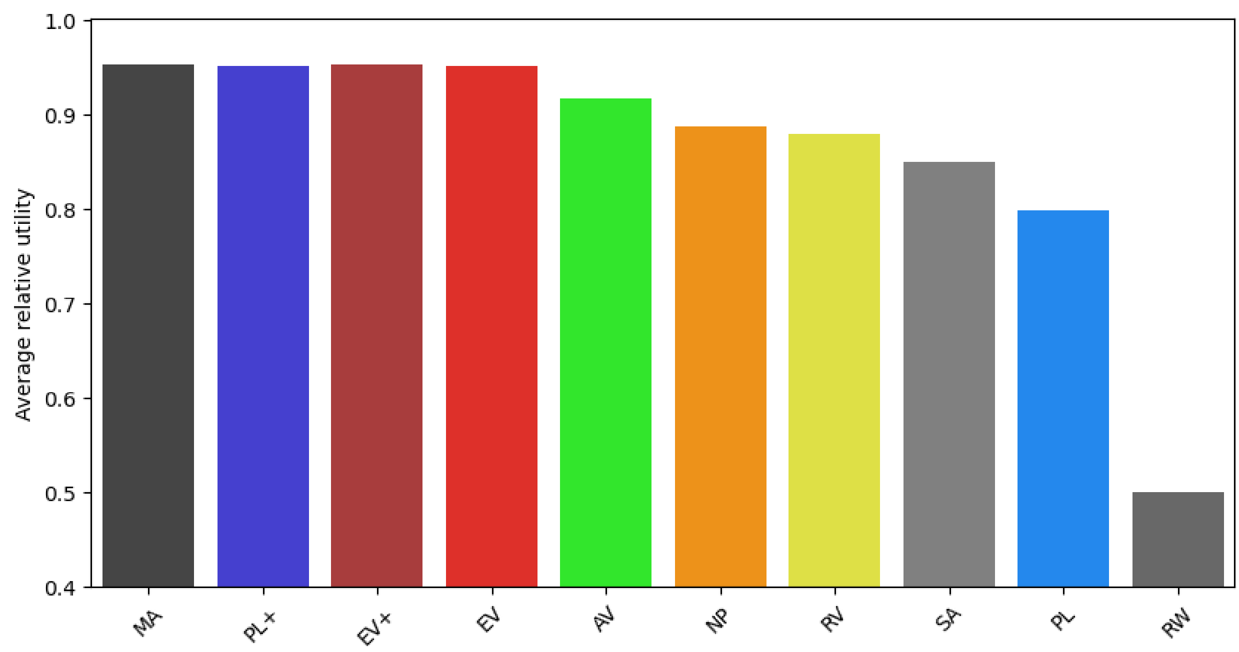
We save the results.

[11]:
```
with open('base_case_results.pkl', 'wb') as f:
    pickle.dump(res, f)
```

5.1.3 Display

We create a figure and export it.

[12]:
```
n_agg = len(res)

plt.figure(figsize=(10,5))
for i in range(n_agg):
    name = list_agg[i].name
    plt.bar(i, res[i],color=colors[name])

plt.xticks(range(n_agg), [handles[agg.name] for agg in list_agg], rotation=45)
plt.xlim(-0.5,n_agg-0.5)
plt.ylabel("Average relative utility")
plt.ylim(0.4)
tikzplotlib.save("basecase.tex")

# save figure
plt.savefig("basecase.png", dpi=300, bbox_inches='tight')
plt.show()
```



5.2 Impact of Numerical Parameters

This notebook investigates how the reference scenario evolves if we change: - The number of correlated agents. - The number of independent agents. - The number of candidates. - The number of training samples for trained aggregators.

The common point between the four studies above is that we use the same drawings of utilities and estimates. For example, an experiment with 40 candidates will share the exact same 20 first candidates than an experiment with 20 candidates only.

First we load some packages and the dataset built in the *reference scenario* notebook, which contains all inputs required for the analysis presented in this notebook.

```
[1]: import numpy as np
     import dill as pickle
     import matplotlib.pyplot as plt
     from tqdm import tqdm
     import tikzplotlib
     from multiprocess.pool import Pool
```

```
[2]: import embedded_voting as ev # Our own module
```

Direct load of some useful variables and functions.

```
[3]: from embedded_voting.experiments.aggregation import make_generator, make_aggs, f_max,␣
     ↪f_renorm
     from embedded_voting.experiments.aggregation import handles, colors, evaluate,␣
     ↪default_order
```

```
[4]: with open('base_case_data.pkl', 'rb') as f:
         data = pickle.load(f)
```

5.2.1 Correlated Agents

Computation

```
[5]: n_c = 20
     cor_size = [1] + [i for i in range(2, 31, 2)]

     res = np.zeros((9,len(cor_size)))
     with Pool() as p:
         for j, s in enumerate(cor_size):
             groups = [s] + [1]*4
             voters = slice(30-groups[0], 30+len(groups)-1)
             training = data['training'][voters, :]
             testing = data['testing'][voters, :, :n_c]
             truth = data['truth'][:, :n_c]

             list_agg = make_aggs(groups)
             res[:, j] = evaluate(list_agg=list_agg, truth=truth, testing=testing,␣
     ↪training=training, pool=p)
```

```
100%|| 10000/10000 [00:11<00:00, 885.54it/s]
100%|| 10000/10000 [00:08<00:00, 1202.57it/s]
100%|| 10000/10000 [00:08<00:00, 1143.43it/s]
100%|| 10000/10000 [00:09<00:00, 1066.39it/s]
100%|| 10000/10000 [00:10<00:00, 994.11it/s]
100%|| 10000/10000 [00:10<00:00, 957.09it/s]
100%|| 10000/10000 [00:11<00:00, 880.80it/s]
100%|| 10000/10000 [00:12<00:00, 800.92it/s]
100%|| 10000/10000 [00:12<00:00, 801.31it/s]
100%|| 10000/10000 [00:13<00:00, 764.85it/s]
100%|| 10000/10000 [00:13<00:00, 715.41it/s]
```

(continues on next page)

(continued from previous page)

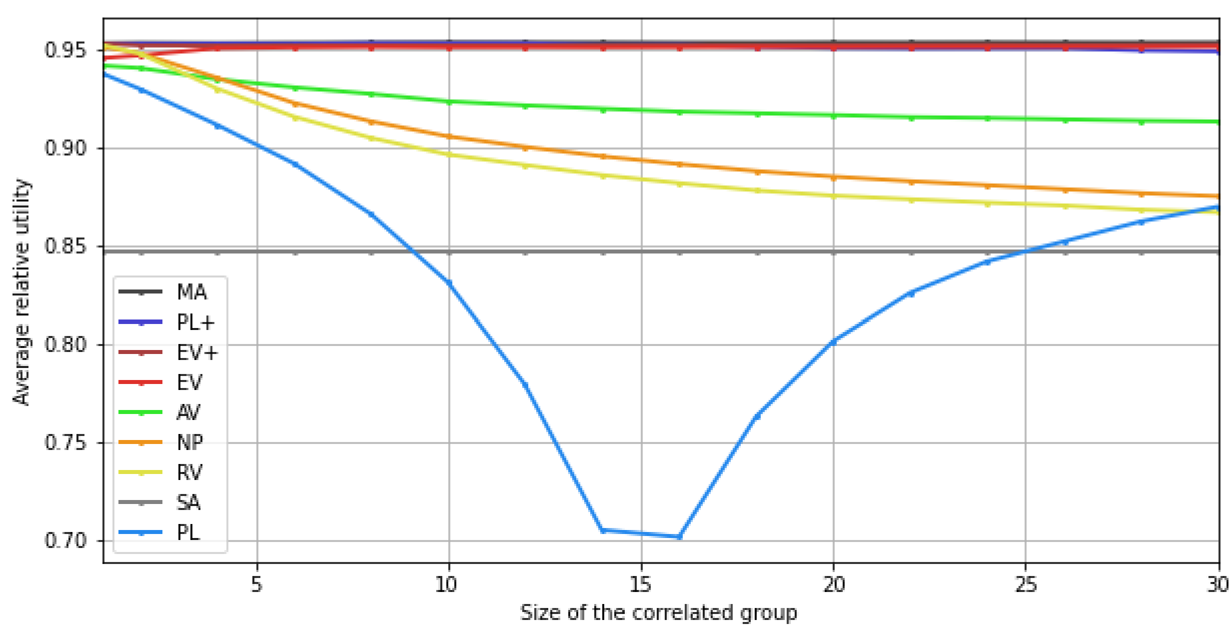
```
100%|| 10000/10000 [00:14<00:00, 672.15it/s]
100%|| 10000/10000 [00:15<00:00, 655.62it/s]
100%|| 10000/10000 [00:16<00:00, 613.33it/s]
100%|| 10000/10000 [00:17<00:00, 579.84it/s]
100%|| 10000/10000 [00:18<00:00, 536.30it/s]
```

We save the results.

```
[6]: with open('correlated_agents.pkl', 'wb') as f:
         pickle.dump(res, f)
```

Display

```
[7]: plt.figure(figsize=(10,5))
     for i, agg in enumerate(list_agg):
         plt.plot(cor_size, res[i], "o-", color=colors[agg.name], label=handles[agg.name],
                  linewidth=2, markersize=2)

     plt.legend()
     plt.xlabel("Size of the correlated group")
     plt.ylabel("Average relative utility")
     plt.xticks(range(5,31,5), range(5,31,5))
     plt.xlim(1,30)
     plt.grid()
     tikzplotlib.save("correlated_agents.tex", axis_height ='6cm', axis_width ='8cm')
     # save figure
     plt.savefig("correlated_agents.png", dpi=300)
     plt.show()
```



5.2.2 Independent Agents

Computation

[8]:
```
ind_size = [0, 1] + [i for i in range(2, 21, 2)]
res = np.zeros((9,len(ind_size)))
```

[9]:
```
with Pool() as p:
    for j, indep in enumerate(ind_size):
        groups = [20] + [1]*indep
        voters = slice(30-groups[0], 30+len(groups)-1)
        training = data['training'][voters, :]
        testing = data['testing'][voters, :, :n_c]
        truth = data['truth'][:, :n_c]

        list_agg = make_aggs(groups)
        res[:, j] = evaluate(list_agg=list_agg, truth=truth, testing=testing,␣
↪training=training, pool=p)
```

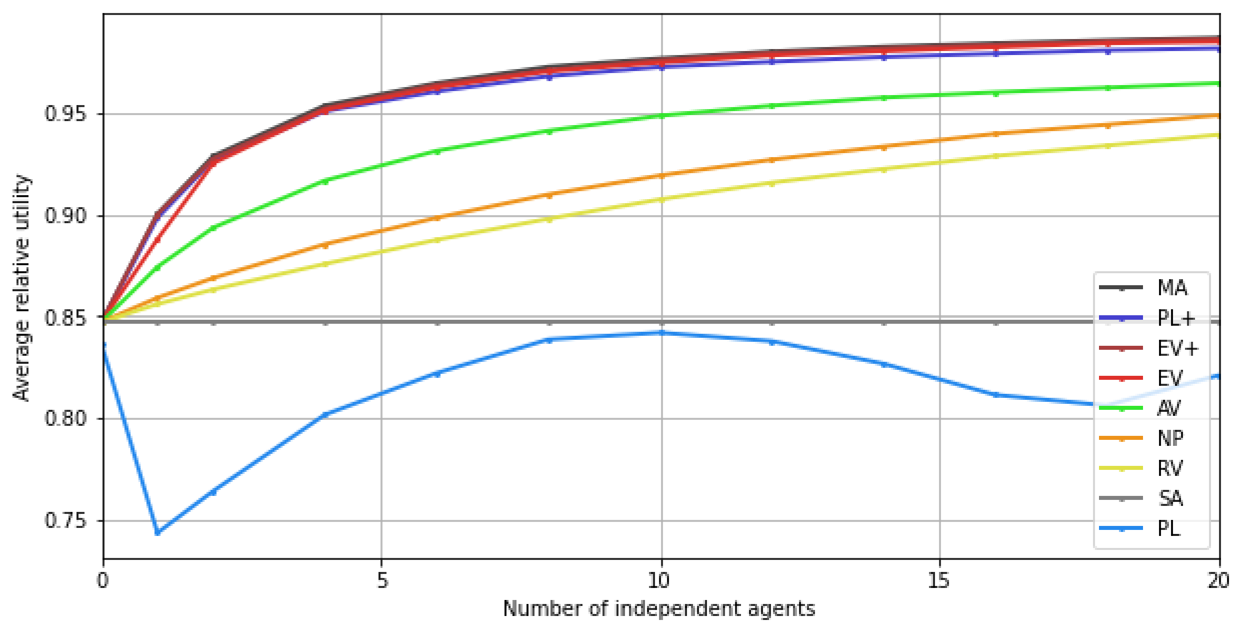
```
100%|| 10000/10000 [00:15<00:00, 663.77it/s]
100%|| 10000/10000 [00:12<00:00, 806.14it/s]
100%|| 10000/10000 [00:12<00:00, 788.17it/s]
100%|| 10000/10000 [00:13<00:00, 740.03it/s]
100%|| 10000/10000 [00:14<00:00, 685.59it/s]
100%|| 10000/10000 [00:15<00:00, 666.25it/s]
100%|| 10000/10000 [00:15<00:00, 627.06it/s]
100%|| 10000/10000 [00:16<00:00, 595.24it/s]
100%|| 10000/10000 [00:17<00:00, 558.29it/s]
100%|| 10000/10000 [00:18<00:00, 532.86it/s]
100%|| 10000/10000 [00:19<00:00, 511.57it/s]
100%|| 10000/10000 [00:20<00:00, 491.89it/s]
```

We save the results.

[10]:
```
with open('independent_agents.pkl', 'wb') as f:
    pickle.dump(res, f)
```

Display

[11]:
```
plt.figure(figsize=(10,5))
for i, agg in enumerate(list_agg):
    plt.plot(ind_size, res[i], "o-", color=colors[agg.name], label=handles[agg.name],
             linewidth=2, markersize=2)

plt.legend()
plt.xlabel("Number of independent agents")
plt.ylabel("Average relative utility")
plt.xticks(range(0,21,5), range(0,21,5))
plt.xlim(0,20)
# plt.ylim(0.5)
plt.grid()
tikzplotlib.save("independents_agents.tex", axis_height ='6cm', axis_width ='8cm')
# save figure
plt.savefig("independents_agents.png", dpi=300)
plt.show()
```



5.2.3 Candidates

Computation

[12]:
```
cand_size = [2,3,4,5,10,15,20,25,30,35,40,45,50]
res = np.zeros((9,len(cand_size)))
```

[13]:
```
groups = [20] + [1]*4
voters = slice(30-groups[0], 30+len(groups)-1)
with Pool() as p:
    for j, n_c in enumerate(cand_size):
        training = data['training'][voters, :]
        testing = data['testing'][voters, :, :n_c]
        truth = data['truth'][:, :n_c]
        list_agg = make_aggs()
        res[:, j] = evaluate(list_agg=list_agg, truth=truth, testing=testing,␣
↪training=training, pool=p)
```

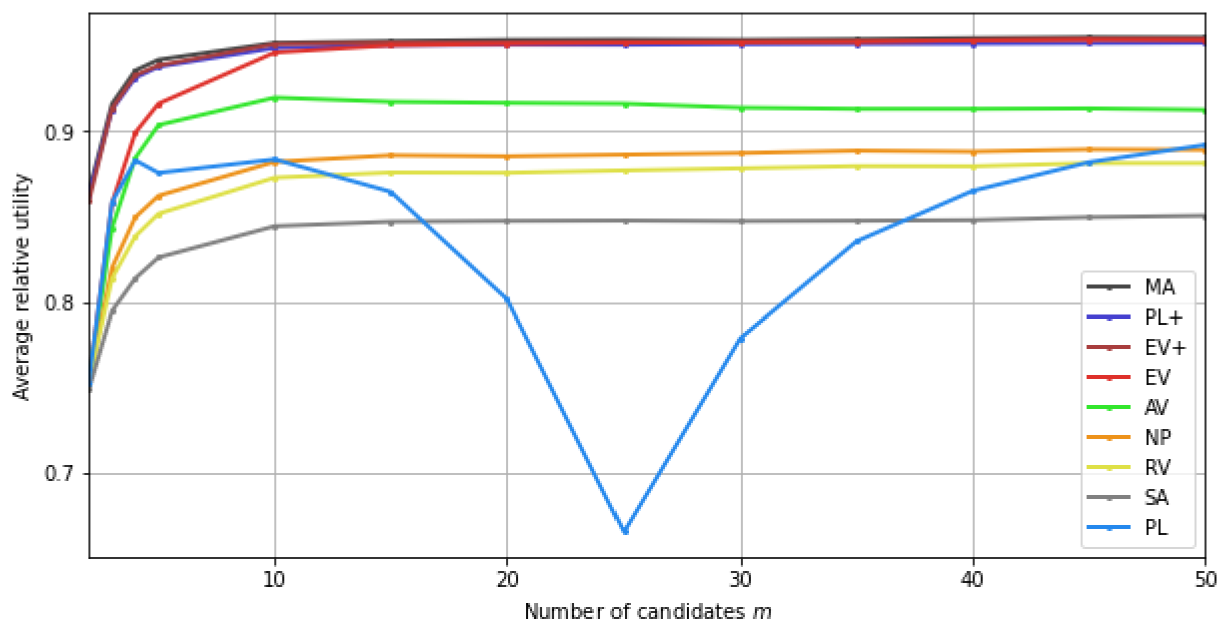
```
100%|| 10000/10000 [00:12<00:00, 774.52it/s]
100%|| 10000/10000 [00:10<00:00, 976.28it/s]
100%|| 10000/10000 [00:10<00:00, 960.23it/s]
100%|| 10000/10000 [00:10<00:00, 958.54it/s]
100%|| 10000/10000 [00:11<00:00, 886.43it/s]
100%|| 10000/10000 [00:12<00:00, 805.59it/s]
100%|| 10000/10000 [00:13<00:00, 740.61it/s]
100%|| 10000/10000 [00:14<00:00, 678.46it/s]
100%|| 10000/10000 [00:15<00:00, 636.49it/s]
100%|| 10000/10000 [00:18<00:00, 552.78it/s]
100%|| 10000/10000 [00:17<00:00, 556.79it/s]
100%|| 10000/10000 [00:19<00:00, 525.67it/s]
100%|| 10000/10000 [01:10<00:00, 141.72it/s]
```

We save the results.

```
[14]: with open('candidates.pkl', 'wb') as f:
          pickle.dump(res, f)
```

Display

```
[15]: plt.figure(figsize=(10,5))
      for i, agg in enumerate(list_agg):
          plt.plot(cand_size, res[i], "o-", color=colors[agg.name], label=handles[agg.name],
                   linewidth=2, markersize=2)

      plt.legend()
      plt.xlabel("Number of candidates $m$")
      plt.ylabel("Average relative utility")
      plt.xticks(range(0,51,10), range(0,51,10))
      plt.yticks([0.7,0.8,0.9], [0.7,0.8,0.9])
      plt.xlim(2,50)
      # plt.ylim(0.6)
      plt.grid()
      tikzplotlib.save("candidates.tex", axis_height ='6cm', axis_width ='8cm')
      # save figure
      plt.savefig("candidates.png", dpi=300)
      plt.show()
```



5.2.4 Training Size

Note: due to space constraints, the following analysis, which studies the convergence speed of training for EV+ and PL+, is not reported in the paper.

Computation

[16]:
```
from copy import copy
train_size = [0,20,40,60,100,140,300,620,1260]
results = np.zeros((2,len(train_size)))
n_c = 20

for j, train in enumerate(train_size):
    training = data['training'][voters, :train]
    testing = data['testing'][voters, :, :n_c]
    truth = data['truth'][:, :n_c]
    n_tries = testing.shape[1]

    list_agg =  [ev.Aggregator(rule=ev.RuleRatingsHistory(rule=ev.RuleMLEGaussian(),␣
↪f=f_renorm),
                              name="PL+"),
                ev.Aggregator(rule=ev.RuleFastNash(), name="EV+")]


    if training.shape[1]:
        for i in range(2):
            _ = list_agg[i](training).winner_

    sa = groups[0]-1 # index of the last agent from the group
    # We run the simulations
    for index_try in tqdm(range(n_tries)):
        ratings_candidates = testing[:, index_try, :]
        # Welfare
        welfare = ev.RuleSumRatings()(ev.Ratings([truth[index_try, :]])).welfare_
        # We run the aggregators, and we look at the welfare of the winner
        for k,agg in enumerate(list_agg):
            agg2 = copy(agg)
            w = agg2(ratings_candidates).winner_
            results[k, j] += welfare[w]
res = results/n_tries
```

```
100%|| 10000/10000 [00:31<00:00, 319.79it/s]
100%|| 10000/10000 [00:32<00:00, 310.74it/s]
100%|| 10000/10000 [00:39<00:00, 250.68it/s]
100%|| 10000/10000 [00:40<00:00, 245.43it/s]
100%|| 10000/10000 [00:42<00:00, 235.20it/s]
100%|| 10000/10000 [00:51<00:00, 192.86it/s]
100%|| 10000/10000 [01:11<00:00, 140.22it/s]
100%|| 10000/10000 [03:10<00:00, 52.43it/s]
100%|| 10000/10000 [06:01<00:00, 27.63it/s]
```

We save the results.

[17]:
```
with open('training.pkl', 'wb') as f:
    pickle.dump(res, f)
```

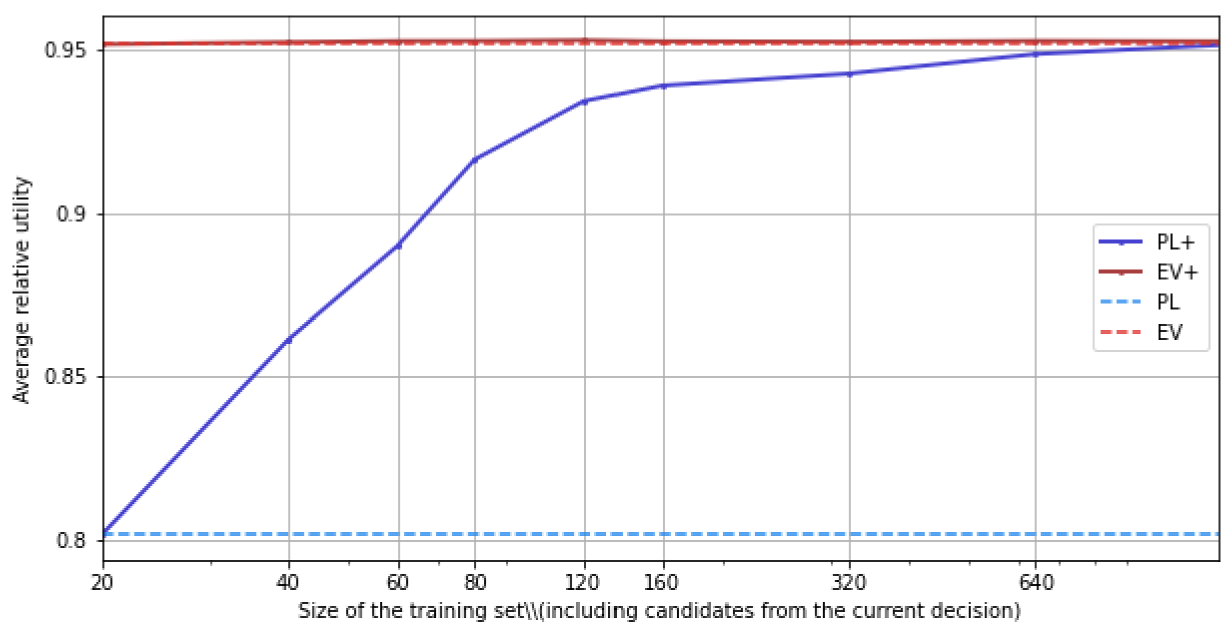
Display

[18]:
```
plt.figure(figsize=(10,5))
for i, agg in enumerate(list_agg):
    plt.plot([x+20 for x in train_size], res[i], "o-", color=colors[agg.name],
```

(continues on next page)

(continued from previous page)

```
             label=handles[agg.name], linewidth=2, markersize=2)

for i, agg in enumerate(list_agg):
    plt.plot([x+20 for x in train_size], [res[i][0]]*len(train_size), "--",
             color=colors[agg.name[:2]], label=handles[agg.name[:2]])


plt.legend()
plt.xlabel("Size of the training set\\\\(including candidates from the current␣
↪decision)")
plt.ylabel("Average relative utility")
plt.xscale("log")
plt.xticks([x+20 for x in train_size], [x+20 for x in train_size])
plt.yticks([0.8,0.85,0.9,0.95], [0.8,0.85,0.9,0.95])
plt.xlim(20,1260)
# plt.ylim(0.5)
plt.grid()
tikzplotlib.save("training.tex")
# save figure
plt.savefig("training.png", dpi=300)
plt.show()
```



We can see that, at least in the reference scenario, EV+ doesn't actually need to be trained. PL+ does.

5.3 Changing Noises

This notebook investigates how the reference scenario evolves if we change: - The intensities of feature and distinct noises. - The shape of the distribution used to draw candidate utilities, feature noises, and distinct noises.

[1]:
```
import numpy as np
import dill as pickle
import matplotlib.pyplot as plt
```

(continues on next page)

(continued from previous page)

```
from tqdm import tqdm
import tikzplotlib
from multiprocess.pool import Pool
```

```
[2]: import embedded_voting as ev # Our own module
```

Direct load of some useful variables and functions.

```
[3]: from embedded_voting.experiments.aggregation import make_generator, make_aggs, f_max,␣
↪f_renorm
from embedded_voting.experiments.aggregation import handles, colors, evaluate,␣
↪default_order
```

```
[4]: n_tries = 10000 # Number of simulations
n_training = 1000 # Number of training samples for trained rules
n_c = 20
```

5.3.1 Changing Noise Intensities

Computation

```
[5]: res = np.zeros( (9, 3, 3) )
with Pool() as p:
    for j, distinct_noise in enumerate([.1, 1, 10]):
        for k, group_noise in enumerate([.1, 1, 10]):
            generator = make_generator(feat_noise=group_noise, dist_noise=distinct_
↪noise)
            training = generator(n_training)
            testing = generator(n_tries*n_c).reshape(generator.n_voters, n_tries, n_c)
            truth = generator.ground_truth_.reshape(n_tries, n_c)

            list_agg = make_aggs(distinct_noise=distinct_noise, group_noise=group_
↪noise)
            res[:, j, k] = evaluate(list_agg=list_agg, truth=truth,
                                    testing=testing, training=training, pool=p)
```

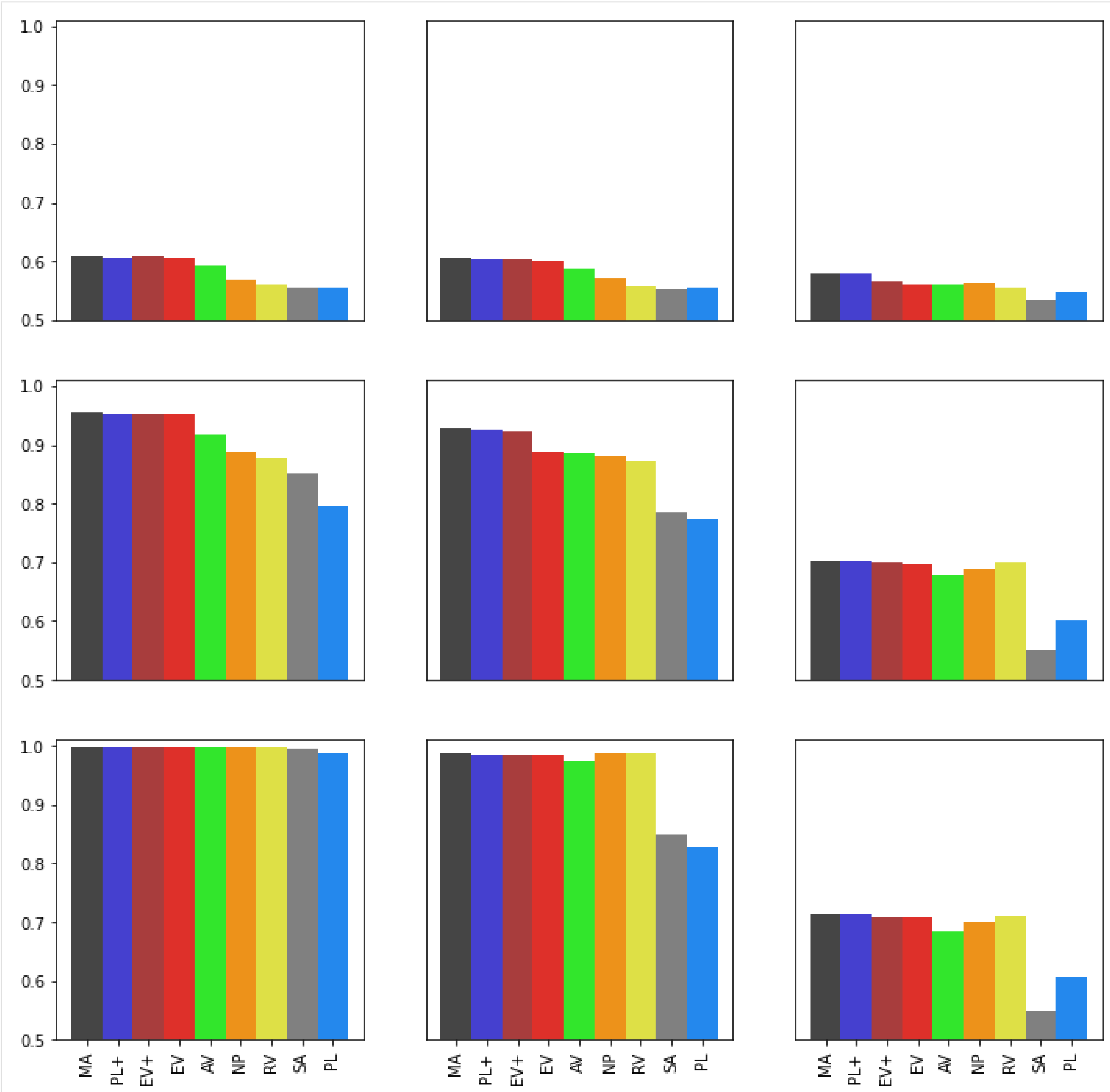
```
100%|| 10000/10000 [00:19<00:00, 523.39it/s]
100%|| 10000/10000 [00:17<00:00, 563.79it/s]
100%|| 10000/10000 [00:15<00:00, 644.25it/s]
100%|| 10000/10000 [00:15<00:00, 650.89it/s]
100%|| 10000/10000 [00:15<00:00, 629.96it/s]
100%|| 10000/10000 [00:15<00:00, 653.53it/s]
100%|| 10000/10000 [00:15<00:00, 649.56it/s]
100%|| 10000/10000 [00:16<00:00, 613.23it/s]
100%|| 10000/10000 [00:15<00:00, 656.05it/s]
```

We save the results.

```
[6]: with open('noises_intensity.pkl', 'wb') as f:
    pickle.dump(res, f)
```

Display

[7]:
```
fig, ax = plt.subplots(3,3, figsize=(12,12))

n_agg = len(list_agg)
for j in range(3):
    for k in range(3):
        ax[j,k].bar(np.arange(n_agg), res[:, k, 2-j], color=[colors[agg.name] for agg␣
↪in list_agg], width=1)
        ax[j,k].set_ylim(0.5,1.01)
        ax[j,k].set_xlim(-1, n_agg)
        if k != 0:
            ax[j,k].get_yaxis().set_visible(False)
            ax[j,k].set_yticks([])
        if j == 2:
            ax[j,k].set_xticks(np.arange(n_agg))
            ax[j,k].set_xticklabels([handles[agg.name] for agg in list_agg],␣
↪rotation=90)
        else:
            ax[j,k].get_xaxis().set_visible(False)
            ax[j,k].set_xticks([])

plt.ylabel("Average relative utility")
tikzplotlib.save("noises_intensity.tex")

# Save figure:
plt.savefig("noises_intensity.png")

# Show the graph
plt.show()
```



5.3.2 Changing Noise Distributions

Finally, we try different noise functions to deviate from a normal distribution. First we try wider and thiner distributions, using *gennorm*, and then we try skewed distributions, using *skewnorm*. In all cases, mean and standard deviations stay the same, only the higher moments are altered.

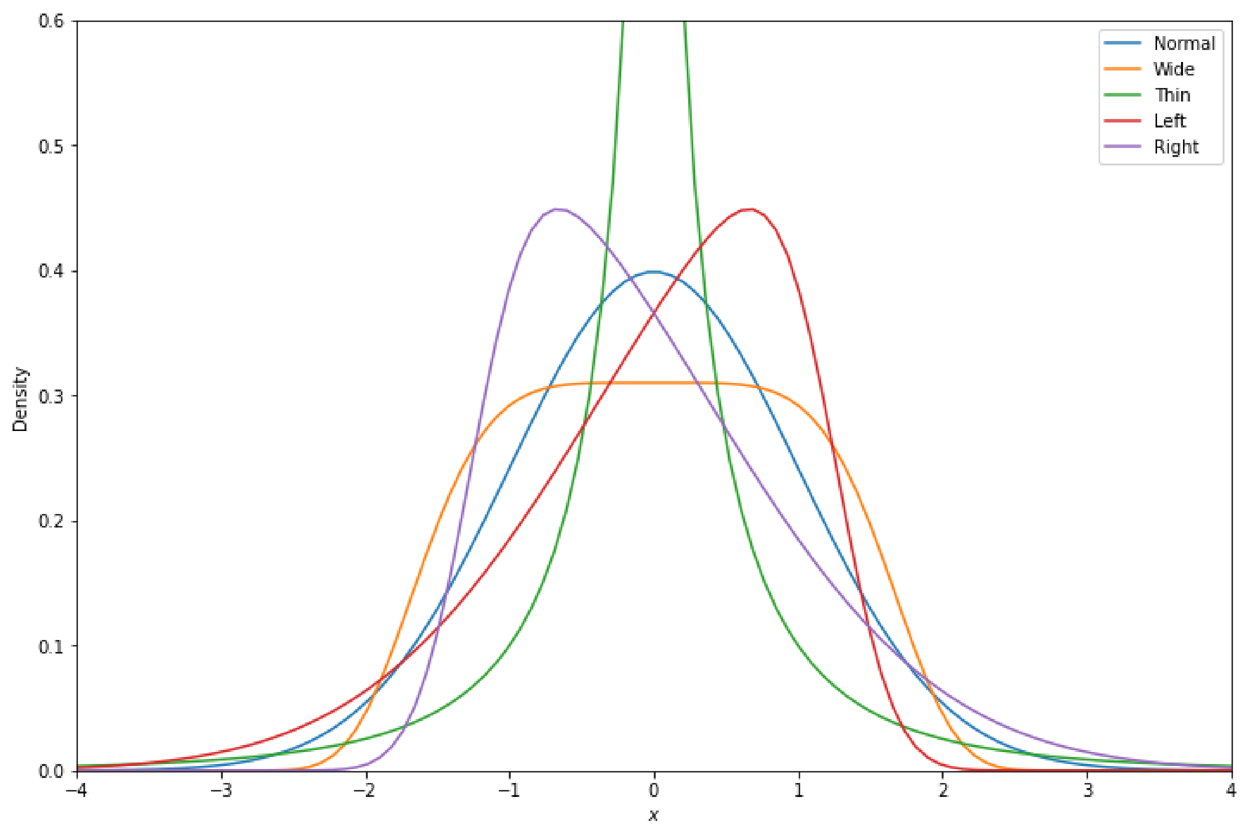
This analysis is not shown in the paper due to space constraints.

Considered Distributions

For illustration, we display the density of each considered distribution, normalized with 0 mean and unit standard deviation.

[8]:
```
from scipy.stats import gennorm, skewnorm, norm
def normalize(rv, a, x):
    _, var = rv.stats(a, moments='mv')
    scale = 1/np.sqrt(var)
    mean, _ = rv.stats(a, scale=scale, moments='mv')
    return rv.pdf(x, a, loc=-mean, scale=scale)
```

[9]:
```
plt.figure(figsize=(12, 8))
x = np.linspace(-4, 4, 100)
plt.plot(x, norm.pdf(x), label='Normal')
plt.plot(x, normalize(gennorm, 5, x), label='Wide')
plt.plot(x, normalize(gennorm, .5, x), label='Thin')
plt.plot(x, normalize(skewnorm, -5, x), label='Left')
plt.plot(x, normalize(skewnorm, 5, x), label='Right')
plt.legend()
plt.xlabel('$x$')
plt.ylabel('Density')
plt.xlim([-4, 4])
plt.ylim([0, .6])
plt.show()
```



Computation

[10]:
```
def create_gennorm(beta=1):
    std = gennorm.rvs(beta, scale=1, size=100000).std()
    scale = 1/std
```

(continues on next page)

(continued from previous page)

```
    def f1(size):
        return gennorm.rvs(beta, scale=scale, size=size)
    return f1


def create_skewnorm(beta=1):
    std = skewnorm.rvs(beta, scale=1, size=100000).std()
    scale = 1/std
    def f1(size):
        return skewnorm.rvs(beta, scale=scale, size=size)
    return f1
```

```
[11]: list_noise = [None, create_gennorm(5), create_gennorm(0.5), create_skewnorm(-5),␣
      ↪create_skewnorm(5)]
      dist_names = ["N", "W", "T",  "L", "R"]
```

```
[12]: res = np.zeros((9, 5))
      # No need to recompute base case
      with open('base_case_results.pkl', 'rb') as f:
          res[:, 0] = pickle.load(f)[:-1]
      list_agg = make_aggs()
```

```
[13]: with Pool() as p:
          for i in range(1, 5):
              generator = make_generator(truth=ev.TruthGeneratorGeneral(list_noise[i]),
                                         feat_f=list_noise[i],
                                         dist_f=list_noise[i])
              training = generator(n_training)
              testing = generator(n_tries*n_c).reshape(generator.n_voters, n_tries, n_c)
              truth = generator.ground_truth_.reshape(n_tries, n_c)

              res[:, i] = evaluate(list_agg=list_agg, truth=truth,
                                   testing=testing, training=training, pool=p)
```

```
100%|| 10000/10000 [00:19<00:00, 505.82it/s]
100%|| 10000/10000 [00:16<00:00, 624.14it/s]
100%|| 10000/10000 [00:20<00:00, 478.08it/s]
100%|| 10000/10000 [00:13<00:00, 756.62it/s]
```

We save the results.

```
[14]: with open('noises_function.pkl', 'wb') as f:
          pickle.dump(res, f)
```

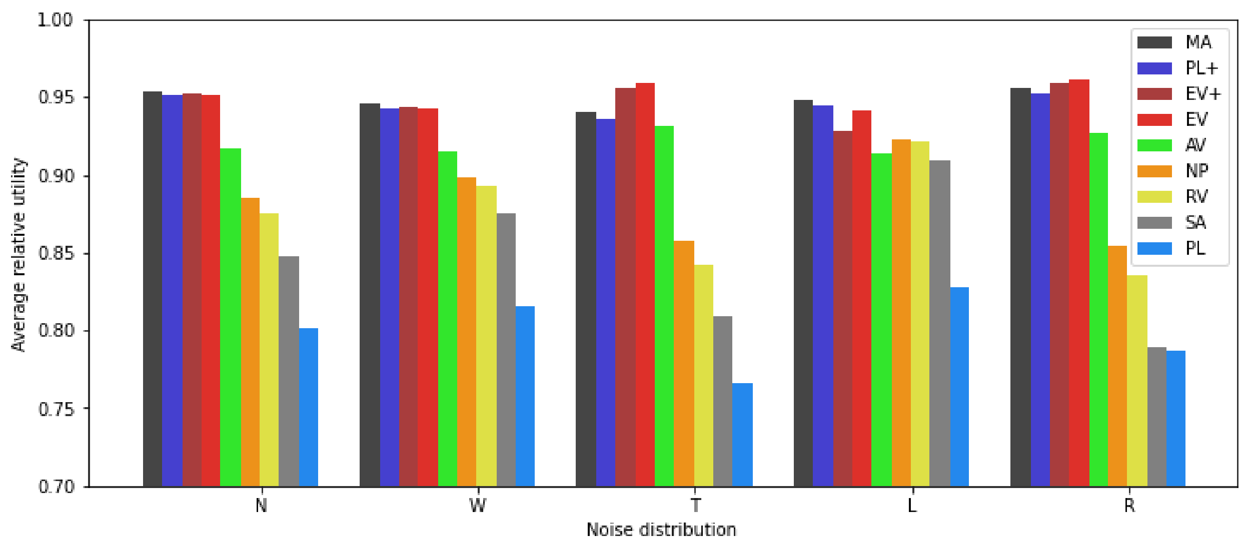
Display

```
[15]: plt.figure(figsize=(12,5))

      for i, agg in enumerate(list_agg):
          plt.bar([j-0.5+0.09*i for j in range(5)], res[i,:],  color=colors[agg.name],
                  label=handles[agg.name], width=0.09)

      plt.legend()
      plt.xticks([i*1 for i in range(5)], dist_names)
```

(continues on next page)

(continued from previous page)

```
plt.xlabel("Noise distribution")
plt.ylabel("Average relative utility")
plt.ylim(0.7,1)
plt.xlim(-0.8,4.5)

tikzplotlib.save("noise_function.tex")
# save figure:
plt.savefig("noise_function.png")
plt.show()
```



We note that MA is no longer a maximum-likelihood estimator as it assumes an incorrect underlying model. The same holds for its approximations PL and PL+. And indeed MA is no longer a *de facto* upper bound of performance: it is outperformed in case R, and even more in case T, by EV/EV+.

5.4 Soft Partition of the Agents

This notebook investigates how the reference scenario evolves if: - The internal cohesion of the large group weakens. - The influence of the large group over the independent agents growths (absorption phenomenon).

[1]:
```
import numpy as np
import dill as pickle
import matplotlib.pyplot as plt
from tqdm import tqdm
import tikzplotlib
from multiprocess.pool import Pool
```

[2]:
```
import embedded_voting as ev # Our own module
```

Direct load of some useful variables and functions.

[3]:
```
from embedded_voting.experiments.aggregation import make_generator, make_aggs, f_max,␣
↪f_renorm
from embedded_voting.experiments.aggregation import handles, colors, evaluate,␣
↪default_order
```

```
[4]: n_tries = 10000 # Number of simulations
n_training = 1000 # Number of training samples for trained rules
n_c = 20
```

5.4.1 Cohesion

Computation

We define the correlation matrix (the Euclidian row-normalization is performed by the generator).

```
[5]: def mymatrix_homogeneity(alpha):
    M = np.eye(24)
    for i in range(20):
        M[i] = [alpha**(np.abs(j-i)) for j in range(20)]+[0]*4
    return M
```

```
[6]: res = np.zeros((9, 11))
list_alpha = [0.1*i for i in range(11)]
with Pool() as p:
    for i, alpha in enumerate(list_alpha[:-1]):
        groups = [1]*24
        features = mymatrix_homogeneity(alpha)
        generator = make_generator(groups=groups, features=features)
        training = generator(n_training)
        testing = generator(n_tries*n_c).reshape(generator.n_voters, n_tries, n_c)
        truth = generator.ground_truth_.reshape(n_tries, n_c)

        list_agg = make_aggs(groups=groups, features=features)
        res[:, i] = evaluate(list_agg=list_agg, truth=truth, testing=testing,
↪training=training, pool=p)
```

```
100%|| 10000/10000 [00:20<00:00, 495.47it/s]
100%|| 10000/10000 [00:16<00:00, 615.94it/s]
100%|| 10000/10000 [00:15<00:00, 632.79it/s]
100%|| 10000/10000 [00:15<00:00, 631.09it/s]
100%|| 10000/10000 [00:15<00:00, 636.35it/s]
100%|| 10000/10000 [00:15<00:00, 659.55it/s]
100%|| 10000/10000 [00:15<00:00, 640.37it/s]
100%|| 10000/10000 [00:15<00:00, 641.47it/s]
100%|| 10000/10000 [00:15<00:00, 665.18it/s]
100%|| 10000/10000 [00:14<00:00, 700.92it/s]
```

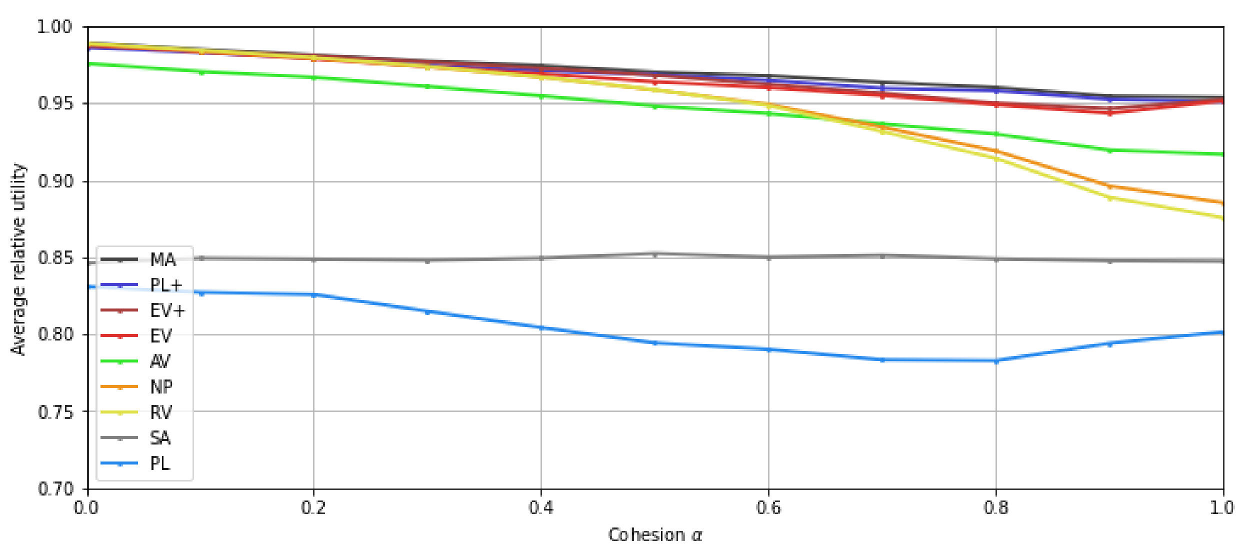
```
[7]: # No need to recompute base case
with open('base_case_results.pkl', 'rb') as f:
    ref_res = pickle.load(f)[:-1]
res[:, -1] = ref_res
```

We save the results.

```
[8]: with open('cohesion.pkl', 'wb') as f:
    pickle.dump(res, f)
```

Display

[9]:
```
plt.figure(figsize=(12,5))

for i, agg in enumerate(list_agg):
    plt.plot(list_alpha, res[i,:],  'o-', color=colors[agg.name], label=handles[agg.
↪name],
             linewidth=2, markersize=2)

plt.legend()
plt.xlabel("Cohesion $\\alpha$")
plt.ylabel("Average relative utility")
# plt.title("Alpha")
plt.ylim(0.7,1)
plt.xlim(0,1)
plt.grid()

tikzplotlib.save("cohesion.tex", axis_height ='6cm', axis_width ='8cm')
# save figure:
plt.savefig("cohesion.png")
plt.show()
```



5.4.2 Absorption

Computation

[10]:
```
def mymatrix_absorption(beta):
    M = np.eye(5)
    for i in range(4):
        M[i+1,0] = beta
        M[i+1,i+1] = 1-beta
    return M
```

[11]:
```
res = np.zeros((9, 11))
res[:, 0] = ref_res
list_beta = [0.1*i for i in range(11)]
with Pool() as p:
    for i, beta in enumerate(list_beta[1:]):
        groups = [20]+[1]*4
        features = mymatrix_absorption(beta)
        generator = make_generator(groups=groups, features=features)
        training = generator(n_training)
        testing = generator(n_tries*n_c).reshape(generator.n_voters, n_tries, n_c)
        truth = generator.ground_truth_.reshape(n_tries, n_c)

        list_agg = make_aggs(groups=groups, features=features)
        res[:, i+1] = evaluate(list_agg=list_agg, truth=truth, testing=testing,␣
↪training=training, pool=p)
```

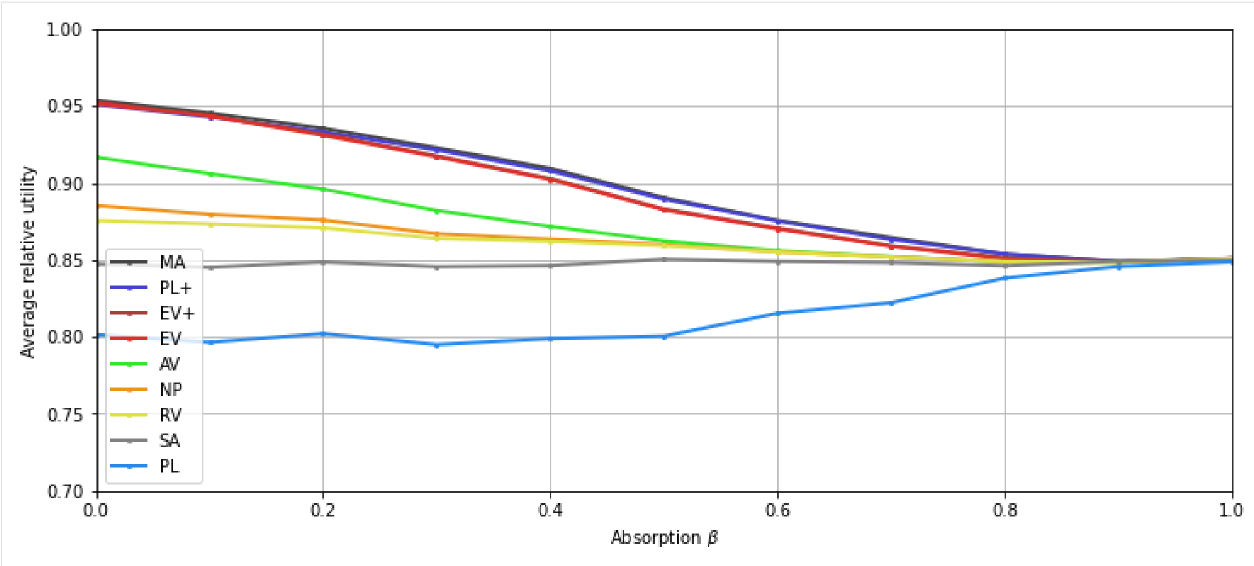
```
100%|| 10000/10000 [00:17<00:00, 577.24it/s]
100%|| 10000/10000 [00:14<00:00, 692.22it/s]
100%|| 10000/10000 [00:17<00:00, 568.29it/s]
100%|| 10000/10000 [00:14<00:00, 702.97it/s]
100%|| 10000/10000 [00:14<00:00, 690.11it/s]
100%|| 10000/10000 [00:14<00:00, 695.19it/s]
100%|| 10000/10000 [00:14<00:00, 700.98it/s]
100%|| 10000/10000 [00:18<00:00, 546.87it/s]
100%|| 10000/10000 [00:14<00:00, 686.72it/s]
100%|| 10000/10000 [00:14<00:00, 698.80it/s]
```

We save the results.

[12]:
```
with open('absorption.pkl', 'wb') as f:
    pickle.dump(res, f)
```

Display

[13]:
```
plt.figure(figsize=(12,5))

for i, agg in enumerate(list_agg):
    plt.plot(list_beta, res[i,:],  'o-', color=colors[agg.name], label=handles[agg.
↪name],
             linewidth=2, markersize=2)

plt.legend()
plt.xlabel("Absorption $\\beta$")
plt.ylabel("Average relative utility")
plt.ylim(0.7,1)
plt.xlim(0,1)
plt.grid()

tikzplotlib.save("absorption.tex", axis_height ='6cm', axis_width ='8cm')
# save figure:
plt.savefig("absorption.png")
plt.show()
```



1.00
0.95
0.90
0.85
0.80
0.75
0.70
Average relative utility
0.0
0.2
0.4
0.6
0.8
1.0
Absorption β
MA
PL+
EV+
EV
AV
NP
RV
SA
PL

CHAPTER 6

Reference

6.1 Truth Generators

6.1.1 Truth Generator

class embedded_voting.**TruthGenerator**

A generator for the ground truth ("true value") of each candidate.

6.1.2 Truth Generator General

class embedded_voting.**TruthGeneratorGeneral**(*function=None*)

A general generator for the ground truth ("true value") of each candidate.

The true value of each candidate is independent and follow a probability distribution defined by the function *function*.

Parameters **function** (*None -> np.ndarray float*) – The function that defines the probability distribution of the true value of each candidate. If *None*, the normal distribution is used.

Examples

```
>>> np.random.seed(42)
>>> truth_generator = TruthGeneratorGeneral()
>>> truth_generator(n_candidates=3)
array([ 0.49671415, -0.1382643 ,  0.64768854])
```

6.1.3 Truth Generator with particular distribution

Uniform distribution

class embedded_voting.**TruthGeneratorUniform**(*minimum_value=10*, *maximum_value=20*, *seed=42*)

A uniform generator for the ground truth ("true value") of each candidate.

The true value of each candidate is independent and uniform in [*minimum_value*, *maximum_value*].

Parameters

- **minimum_value** (*Number*) – The minimum true value of a candidate.
- **maximum_value** (*Number*) – The maximum true value of a candidate.

Examples

```
>>> np.random.seed(42)
>>> truth_generator = TruthGeneratorUniform(minimum_value=10, maximum_value=20)
>>> truth_generator(n_candidates=3)
array([17.73956049, 14.3887844 , 18.5859792 ])
```

Normal distribution

class embedded_voting.**TruthGeneratorNormal**(*center=15*, *noise=5*)

A normal generator for the ground truth ("true value") of each candidate.

The true value of each candidate is independent and follow a Gaussian distribution with mean *center* and standard deviation *noise*.

Parameters

- **center** (*float*) – The mean of the Gaussian distribution.
- **noise** (*float*) – The standard deviation of the Gaussian distribution.

Examples

```
>>> np.random.seed(42)
>>> truth_generator = TruthGeneratorNormal(center=15, noise=5)
>>> truth_generator(n_candidates=3)
array([17.48357077, 14.30867849, 18.23844269])
```

6.2 Ratings classes

6.2.1 Ratings

class embedded_voting.**Ratings**

Ratings of the voters in a given election.

Parameters **ratings** (*list, np.ndarray or Ratings*) – The ratings given by each voter to each candidate.

n_voters

The number of voters.

Type int

n_candidates
The number of candidates.

Type int

Examples

```
>>> ratings = Ratings([[1, .8, .5], [.3, .5, .9]])
>>> ratings
Ratings([[1. , 0.8, 0.5],
         [0.3, 0.5, 0.9]])
>>> ratings.n_voters
2
>>> ratings.n_candidates
3
>>> ratings.voter_ratings(0)
array([1. , 0.8, 0.5])
>>> ratings.candidate_ratings(0)
array([1. , 0.3])
```

6.2.2 Ratings Generator

class embedded_voting.**RatingsGenerator**(*n_voters*)
This abstract class creates *Ratings* from scratch using some function.

Parameters **n_voters** (*int*) – Number of voters in the embeddings.

Uniform Ratings

class embedded_voting.**RatingsGeneratorUniform**(*n_voters*, *minimum_rating=0*, *maximum_rating=1*)
Generate uniform random ratings.

Examples

```
>>> np.random.seed(42)
>>> generator = RatingsGeneratorUniform(n_voters=5)
>>> generator(n_candidates=4)
Ratings([[0.37454012, 0.95071431, 0.73199394, 0.59865848],
         [0.15601864, 0.15599452, 0.05808361, 0.86617615],
         [0.60111501, 0.70807258, 0.02058449, 0.96990985],
         [0.83244264, 0.21233911, 0.18182497, 0.18340451],
         [0.30424224, 0.52475643, 0.43194502, 0.29122914]])
```

6.2.3 Ratings Generator Epistemic

class embedded_voting.**RatingsGeneratorEpistemic**(*n_voters=None*, *truth_generator=None*)
A generator of ratings based on a ground truth ("true value") for each candidate.

Parameters

- **n_voters** (*int*) – The number of voters in the generator.
- **truth_generator** (TruthGenerator) – The truth generator used to generate to true values of each candidate. Default: *TruthGeneratorUniform(10, 20)*.

ground_truth_

The ground truth ("true value") for each candidate, corresponding to the last ratings generated.

Type np.ndarray

plot_ratings (*show=True*)

This function plots the true value of a candidate and the ratings given by each voter for a candidate with new random values and ratings.

Parameters **show** (*bool*) – If True, displays the plot at the end of the function.

Grouped Mean

class embedded_voting.**RatingsGeneratorEpistemicGroupsMean**(*groups_sizes*, *group_noise=1*, *independent_noise=0*, *truth_generator=None*)

A generator of ratings such that voters are separated into different groups and the noise of an voter on a candidate is equal to the noise of his group plus his own independent noise.

This is a particular case of *RatingsGeneratorEpistemicGroupsMix* when *groups_features* is the identity matrix, i.e. each group has its own exclusive feature.

As a result, for each candidate *i*:

- For each group, a *sigma_group* is drawn (absolute part of a normal variable, scaled by *group_noise*). Then a *noise_group* is drawn (normal variable scaled by *sigma_group*).
- For each voter, *noise_dependent* is equal to the *noise_group* of her group.
- For each voter, *noise_independent* is drawn (normal variable scaled by *independent_noise*).
- For each voter of each group, the rating is computed as *ground_truth[i]* + *noise_dependent* + *noise_independent*.

Parameters

- **groups_sizes** (*list or np.ndarray*) – The number of voters in each groups. The sum is equal to n_voters.
- **group_noise** (*float*) – The variance used to sample the noise of each group.
- **independent_noise** (*float*) – The variance used to sample the independent noise of each voter.
- **truth_generator** (TruthGenerator) – The truth generator used to generate to true values of each candidate. Default: *TruthGeneratorUniform(10, 20)*.

ground_truth_

The ground truth ("true value") for each candidate, corresponding to the last ratings generated.

Type np.ndarray

Examples

```
>>> np.random.seed(44)
>>> generator = RatingsGeneratorEpistemicGroupsMean([2, 2])
>>> generator()  # doctest: +ELLIPSIS
Ratings([[16.3490...],
         [16.3490...],
         [19.16928...],
         [19.16928...]])
>>> generator.ground_truth_  # doctest: +ELLIPSIS
array([17.739...])
```

Grouped Noise

class embedded_voting.**RatingsGeneratorEpistemicGroupsNoise**(*groups_sizes*, *group_noise=1*, *truth_generator=None*)

A generator of ratings such that voters are separated into different groups and for each candidate the variance of each voter of the same group is the same.

For each candidate *i*:

- For each group, a *sigma_group* is drawn (absolute part of a normal variable, scaled by *group_noise*).
- For each voter, her *sigma_voter* is equal to the *sigma_group* of her group. Her *noise_voter* is drawn (normal variable scaled by *sigma_voter*).
- For each voter, the rating is computed as *ground_truth[i]* + *noise_voter*.

Parameters

- **groups_sizes** (*list or np.ndarray*) – The number of voters in each groups. The sum is equal to `n_voters`.
- **group_noise** (*float*) – The variance used to sample the variances of each group.
- **truth_generator** (`TruthGenerator`) – The truth generator used to generate to true values of each candidate. Default: *TruthGeneratorUniform(10, 20)*.

ground_truth_

The ground truth ("true value") for each candidate, corresponding to the last ratings generated.

Type np.ndarray

Examples

```
>>> np.random.seed(42)
>>> generator = RatingsGeneratorEpistemicGroupsNoise([2, 2])
>>> generator()  # doctest: +ELLIPSIS
Ratings([[18.196...],
         [18.812...],
         [17.652...],
         [17.652...]])
>>> generator.ground_truth_  # doctest: +ELLIPSIS
array([17.739...])
```

Grouped Mix

class embedded_voting.**RatingsGeneratorEpistemicGroupsMix**(*groups_sizes*, *groups_features*, *group_noise=1*, *independent_noise=0*, *truth_generator=None*)

A generator of ratings such that voters are separated into different groups and the noise of an voter on a candidate is equal to the noise of his group plus his own independent noise. The noise of different groups can be correlated due to the group features.

For each candidate *i*:

- For each feature, a *sigma_feature* is drawn (absolute part of a normal variable, scaled by *group_noise*). Then a *noise_feature* is drawn (normal variable scaled by *sigma_feature*).
- For each group, *noise_group* is the barycenter of the values of *noise_feature*, with the weights for each feature given by *groups_features*.
- For each voter, *noise_dependent* is equal to the *noise_group* of her group.
- For each voter, *noise_independent* is drawn (normal variable scaled by *independent_noise*).
- For each voter of each group, the rating is computed as *ground_truth[i]* + *noise_dependent* + *noise_independent*.

Parameters

- **groups_sizes** (*list or np.ndarray*) – The number of voters in each groups. The sum is equal to `n_voters`.
- **groups_features** (*list or np.ndarray*) – The features of each group of voters. Should be of the same length than `group_sizes`. Each row of this matrix correspond to the features of a group.
- **group_noise** (*float*) – The variance used to sample the noise of each group.
- **independent_noise** (*float*) – The variance used to sample the independent noise of each voter.
- **truth_generator** (TruthGenerator) – The truth generator used to generate to true values of each candidate. Default: *TruthGeneratorUniform(10, 20)*.

ground_truth_

The ground truth ("true value") for each candidate, corresponding to the last ratings generated.

Type np.ndarray

Examples

```
>>> np.random.seed(42)
>>> features = [[1, 0], [0, 1], [1, 1]]
>>> generator = RatingsGeneratorEpistemicGroupsMix([2, 2, 2], features)
>>> generator()  # doctest: +ELLIPSIS
Ratings([[18.1960...],
         [18.1960...],
         [18.3058...],
         [18.3058...],
         [18.2509...],
```

(continues on next page)

(continued from previous page)

```
        [18.2509...]])
>>> generator.ground_truth_  # doctest: +ELLIPSIS
array([17.7395...])
```

```
>>> np.random.seed(42)
>>> features = [[1, 0, 1, 1], [0, 1, 0, 1], [1, 1, 0, 0]]
>>> generator = RatingsGeneratorEpistemicGroupsMix([2, 2, 2], features)
>>> generator() # doctest: +ELLIPSIS
Ratings([[17.951...],
         [17.951...],
         [17.737...],
         [17.737...],
         [18.438...],
         [18.438...]])
```

Multivariate

class embedded_voting.**RatingsGeneratorEpistemicMultivariate**(*covariance_matrix*, *independent_noise=0*, *truth_generator=None*)

A generator of ratings based on a covariance matrix.

Parameters

- **covariance_matrix** (*np.ndarray*) – The covariance matrix of the voters. Should be of shape `n_voters, n_voters`.
- **independent_noise** (*float*) – The variance of the independent noise.
- **truth_generator** (`TruthGenerator`) – The truth generator used to generate to true values of each candidate. Default: *TruthGeneratorUniform(10, 20)*.

ground_truth_

The ground truth ("true value") for each candidate, corresponding to the last ratings generated.

Type np.ndarray

Examples

```
>>> np.random.seed(42)
>>> generator = RatingsGeneratorEpistemicMultivariate(np.ones((5, 5)))
>>> generator()  # doctest: +ELLIPSIS
Ratings([[17.2428...],
         [17.2428...],
         [17.2428...],
         [17.2428...],
         [17.2428...]])
>>> generator.independent_noise = 0.5
>>> generator()  # doctest: +ELLIPSIS
Ratings([[14.5710...],
         [14.3457...],
         [15.0093...],
         [14.3981...],
         [14.1460...]])
```

Grouped Mix Free

class embedded_voting.**RatingsGeneratorEpistemicGroupsMixFree**(*groups_sizes*, *groups_features*, *group_noise=1*, *independent_noise=0*, *truth_generator=None*, *group_noise_f=None*, *independent_noise_f=None*)

A generator of ratings such that voters are separated into different groups and the noise of an voter on a candidate is equal to the noise of his group plus his own independent noise. The noise of different groups can be correlated due to the group features.

For each candidate *i*:

- For each feature, a *sigma_feature* is drawn (absolute part of a normal variable, scaled by *group_noise*). Then a *noise_feature* is drawn according to *group_noise_f* (scaled by *group_noise*).
- For each group, *noise_group* is the barycenter of the values of *noise_feature*, with the weights for each feature given by *groups_features*.
- For each voter, *noise_dependent* is equal to the *noise_group* of her group.
- For each voter, *noise_independent* is drawn according to *independent_noise_f* (scaled by *independent_noise*).
- For each voter of each group, the rating is computed as *ground_truth[i]* + *noise_dependent* + *noise_independent*.

Parameters

- **groups_sizes** (*list or np.ndarray*) – The number of voters in each groups. The sum is equal to `n_voters`.
- **groups_features** (*list or np.ndarray*) – The features of each group of voters. Should be of the same length than `group_sizes`. Each row of this matrix correspond to the features of a group.
- **group_noise** (*float*) – The variance used to sample the noise of each group.
- **independent_noise** (*float*) – The variance used to sample the independent noise of each voter.
- **truth_generator** (*TruthGenerator*) – The truth generator used to generate to true values of each candidate. Default: *TruthGeneratorUniform(10, 20)*.
- **group_noise_f** (*function*) – The function used to sample the noise of each group. Default: *np.random.normal*.
- **independent_noise_f** (*function*) – The function used to sample the independent noise of each voter. Default: *np.random.normal*.

ground_truth_

The ground truth ("true value") for each candidate, corresponding to the last ratings generated.

Type np.ndarray

Examples

```
>>> np.random.seed(42)
>>> features = [[1, 0], [0, 1], [1, 1]]
>>> generator = RatingsGeneratorEpistemicGroupsMixFree([2, 2, 2], features, group_
↪noise_f=np.random.normal, independent_noise_f=np.random.normal)
>>> generator()  # doctest: +ELLIPSIS
Ratings([[18.23627...],
         [18.23627...],
         [17.60129...],
         [17.60129...],
         [17.99302...],
         [17.99302...]])
>>> generator.ground_truth_  # doctest: +ELLIPSIS
array([17.73956...])
```

6.3 Embeddings

6.3.1 Embeddings

class embedded_voting.**Embeddings**

Embeddings of the voters.

Parameters

- **positions** (*np.ndarray or list or* Embeddings) – The embeddings of the voters. Its dimensions are *n_voters*, *n_dim*.
- **norm** (*bool*) – If True, normalize the embeddings.

n_voters

The number of voters in the ratings.

Type int

n_dim

The number of dimensions of the voters' embeddings.

Type int

Examples

```
>>> embeddings = Embeddings([[1, 0], [0, 1], [0.5, 0.5]], norm=True)
>>> embeddings.n_voters
3
>>> embeddings.n_dim
2
>>> embeddings.voter_embeddings(0)
array([1., 0.])
```

copy (*order='C'*)

Return a copy of the array.

Parameters order (*{'C', 'F', 'A', 'K'}, optional*) – Controls the memory layout of the copy. 'C' means C-order, 'F' means F-order, 'A' means 'F' if *a* is Fortran contiguous, 'C' otherwise. 'K' means match the layout of *a* as closely as possible. (Note that this function and `numpy.copy()` are very similar but have different default values for their order= arguments, and this function always passes sub-classes through.)

See also:

`numpy.copy()` Similar function with different default behavior

`numpy.copyto()`

Notes

This function is the preferred method for creating an array copy. The function `numpy.copy()` is similar, but it defaults to using order 'K', and will not pass sub-classes through by default.

Examples

```
>>> x = np.array([[1,2,3],[4,5,6]], order='F')
```

```
>>> y = x.copy()
```

```
>>> x.fill(0)
```

```
>>> x
array([[0, 0, 0],
       [0, 0, 0]])
```

```
>>> y
array([[1, 2, 3],
       [4, 5, 6]])
```

```
>>> y.flags['C_CONTIGUOUS']
True
```

`dilated`(*approx=True*)

Dilate the embeddings of the voters so that they take more space.

The *center* is computed with *`get_center()`*. The angular dilatation factor is such that after transformation, the maximum angle between the center and an embedding vector will be pi / 4.

Parameters **`approx`** (*bool*) – Passed to *`get_center()`* in order to compute the center of the voters' embeddings.

Returns A new Embeddings object with the dilated embeddings.

Return type *Embeddings*

Examples

```
>>> embeddings = Embeddings(np.array([[.5,.4,.4],[.4,.4,.5],[.4,.5,.4]]),␣
↪norm=True)
>>> embeddings
Embeddings([[0.66226618, 0.52981294, 0.52981294],
            [0.52981294, 0.52981294, 0.66226618],
```

(continues on next page)

(continued from previous page)

```
              [0.52981294, 0.66226618, 0.52981294]])
>>> embeddings.dilated()
Embeddings([[0.98559856, 0.11957316, 0.11957316],
              [0.11957316, 0.11957316, 0.98559856],
              [0.11957316, 0.98559856, 0.11957316]])
```

Note that the resulting embedding may not be in the positive orthant, even if the original embedding is:

```
>>> embeddings = Embeddings([[1, 0], [.7, .7]], norm=True)
>>> embeddings.dilated()
Embeddings([[ 0.92387953, -0.38268343],
              [ 0.38268343,  0.92387953]])
```

```
>>> Embeddings([[1, 0]], norm=True).dilated()
Embeddings([[1., 0.]])
```

dilated_aux(*center*, *k*)

Dilate the embeddings of the voters.

For each *vector* of the embedding, we apply a "spherical dilatation" that moves *vector* by multiplying the angle between *center* and *vector* by a given dilatation factor.

More formally, for each *vector* of the embedding, there exists a unit vector *unit_orthogonal* and an angle *theta in [0, pi/2]* such that *vector = norm(vector) * (cos(theta) * center + sin(theta) * unit_orthogonal)*. Then the image of *vector* is *norm(vector) * (cos(k * theta) * center + sin(k * theta) * unit_orthogonal)*.

Parameters

- **center** (*np.ndarray*) – Unit vector: center of the dilatation.
- **k** (*float*) – Angular dilatation factor.

Returns A new Embeddings object with the dilated embeddings.

Return type *Embeddings*

Examples

```
>>> embeddings = Embeddings([[1, 0], [1, 1]], norm=True)
>>> dilated_embeddings = embeddings.dilated_aux(center=np.array([1, 0]), k=2)
>>> np.round(dilated_embeddings, 4)
array([[1., 0.],
       [0., 1.]])
```

```
>>> embeddings = Embeddings([[1, 0], [1, 1]], norm=False)
>>> dilated_embeddings = embeddings.dilated_aux(center=np.array([1, 0]), k=2)
>>> np.abs(np.round(dilated_embeddings, 4)) # Abs for rounding errors
array([[1.    , 0.    ],
       [0.    , 1.4142]])
```

dilated_new(*approx=True*)

Dilate the embeddings of the voters so that they take more space in the positive orthant.

The *center* is computed with *get_center()*. The angular dilatation factor the largest possible so that all vectors stay in the positive orthant. Cf. *max_angular_dilatation_factor()*.

Parameters **approx** (*bool*) – Passed to *get_center()* in order to compute the center of the voters' embeddings.

Returns A new Embeddings object with the dilated embeddings.

Return type *Embeddings*

Examples

```
>>> embeddings = Embeddings(np.array([[.5,.4,.4],[.4,.4,.5],[.4,.5,.4]]),␣
↪norm=True)
>>> embeddings
Embeddings([[0.66226618, 0.52981294, 0.52981294],
            [0.52981294, 0.52981294, 0.66226618],
            [0.52981294, 0.66226618, 0.52981294]])
>>> dilated_embeddings = embeddings.dilated_new()
>>> np.abs(np.round(dilated_embeddings, 4))
array([[1., 0., 0.],
       [0., 0., 1.],
       [0., 1., 0.]])
```

```
>>> embeddings = Embeddings([[1, 0], [.7, .7]], norm=True)
>>> dilated_embeddings = embeddings.dilated_new()
>>> np.abs(np.round(dilated_embeddings, 4))
array([[1.    , 0.    ],
       [0.7071, 0.7071]])
```

```
>>> embeddings = Embeddings([[2, 1], [100, 200]], norm=False)
>>> dilated_embeddings = embeddings.dilated_new()
>>> np.round(dilated_embeddings, 4)
array([[  2.2361,   0.    ],
       [  0.    , 223.6068]])
```

get_center(*approx=True*)

Return the center direction of the embeddings.

For this method, we work on the normalized embeddings. Cf. `normalized()`.

With *approx* set to False, we use an exponential algorithm in *n_dim*. If *r* is the rank of the embedding matrix, we first find the *r* voters with maximal determinant (in absolute value), i.e. whose associated parallelepiped has the maximal volume (e.g. in two dimensions, it means finding the two vectors with maximal angle). Then the result is the mean of the embeddings of these voters, normalized in the sense of the Euclidean norm.

With *approx* set to True, we use a polynomial algorithm: we simply take the mean of the embeddings of all the voters, normalized in the sense of the Euclidean norm.

Parameters **approx** (`bool`) – Whether the computation is approximate.

Returns The normalized position of the center vector. Size: `n_dim`.

Return type np.ndarray

Examples

```
>>> embeddings = Embeddings([[1, 0], [0, 1], [.5, .5], [.7, .3]], norm=True)
>>> embeddings.get_center(approx=False)
array([0.70710678, 0.70710678])
```

```
>>> embeddings = Embeddings([[1, 0], [0, 1], [.5, .5], [.7, .3]], norm=False)
>>> embeddings.get_center(approx=False)
array([0.70710678, 0.70710678])
```

```
>>> embeddings = Embeddings([[1, 0], [0, 1], [.5, .5], [.7, .3]], norm=True)
>>> embeddings.get_center(approx=True)
array([0.78086524, 0.62469951])
```

mixed_with(*other*, *intensity*)

Mix this embedding with another one.

Parameters

- **other** (Embeddings) – Another embedding with the name number of voters and same number of dimensions.
- **intensity** (*float*) – Must be in [0, 1].

Returns A new Embeddings object with the mixed embeddings.

Return type *Embeddings*

Examples

For a given voter, the direction of the final embedding is an “angular barycenter” between the original direction and the direction in *other*, with mixing parameter *intensity*:

```
>>> embeddings = Embeddings([[1, 0]], norm=True)
>>> other_embeddings = Embeddings([[0, 1]], norm=True)
>>> embeddings.mixed_with(other_embeddings, intensity=1/3)
Embeddings([[0.8660254, 0.5      ]])
```

For a given voter, the norm of the final embedding is a barycenter between the original norm and the norm in *other*, with mixing parameter *intensity*:

```
>>> embeddings = Embeddings([[1, 0]], norm=False)
>>> other_embeddings = Embeddings([[5, 0]], norm=False)
>>> embeddings.mixed_with(other_embeddings, intensity=1/4)
Embeddings([[2., 0.]])
```

normalized()

Normalize the embeddings of the voters so the Euclidean norm of every embedding is 1.

Returns A new Embeddings object with the normalized embeddings.

Return type *Embeddings*

Examples

```
>>> embeddings = Embeddings(-np.array([[.5,.9,.4],[.4,.7,.5],[.4,.2,.4]]),
↪norm=False)
>>> embeddings
Embeddings([[-0.5, -0.9, -0.4],
            [-0.4, -0.7, -0.5],
            [-0.4, -0.2, -0.4]])
>>> embeddings.normalized()
```

(continues on next page)

(continued from previous page)

```
Embeddings([[-0.45267873, -0.81482171, -0.36214298],
            [-0.42163702, -0.73786479, -0.52704628],
            [-0.66666667, -0.33333333, -0.66666667]])
```

plot (*plot_kind='3D'*, *dim: list = None*, *fig=None*, *plot_position=None*, *show=True*)

Plot the embeddings of the voters, either on a 3D plot, or on a ternary plot. Only three dimensions can be represented.

Parameters

- **plot_kind** (*str*) – The kind of plot we want to show. Can be `'3D'` or `'ternary'`.
- **dim** (*list*) – A list of length 3 containing the three dimensions of the embeddings we want to plot. All elements of this list should be lower than `n_dim`. By default, it is set to `[0, 1, 2]`.
- **fig** (*matplotlib figure*) – The figure on which we add the plot. The default figure is a *8 x 8* matplotlib figure.
- **plot_position** (*list*) – List of length 3 containing the matplotlib position `[n_rows, n_columns, position]`. By default, it is set to `[1, 1, 1]`.
- **show** (*bool*) – If True, display the figure at the end of the function.

Returns The matplotlib ax with the figure, if you want to add something to it.

Return type matplotlib ax

plot_candidate (*ratings*, *candidate*, *plot_kind='3D'*, *dim: list = None*, *fig=None*, *plot_position=None*, *show=True*)

Plot the matrix associated to a candidate.

The embedding of each voter is multiplied by the rating she assigned to the candidate.

Parameters

- **ratings** (*np.ndarray*) – Matrix of ratings given by voters to the candidates.
- **candidate** (*int*) – The candidate for which we want to show the ratings. Should be lower than `n_candidates` of ratings.
- **plot_kind** (*str*) – The kind of plot we want to show. Can be `'3D'` or `'ternary'`.
- **fig** (*matplotlib figure*) – The figure on which we add the plot.
- **plot_position** (*list*) – The position of the plot on the figure. Should be of the form `[n_rows, n_columns, position]`.
- **dim** (*list*) – The 3 dimensions we are using for our plot. By default, it is set to `[0, 1, 2]`.
- **show** (*bool*) – If True, display the figure at the end of the function.

Returns The matplotlib ax with the figure, if you want to add something to it.

Return type matplotlib ax

plot_candidates (*ratings*, *plot_kind='3D'*, *dim: list = None*, *list_candidates=None*, *list_titles=None*, *row_size=5*, *show=True*)

Plot the matrix associated to a candidate for every candidate in a list of candidates.

Parameters

- **ratings** (`Ratings`) – Ratings given by voters to candidates.

- **plot_kind** (*str*) – The kind of plot we want to show. Can be `'3D'` or `'ternary'`.
- **dim** (*list*) – The 3 dimensions we are using for our plot. By default, it is set to `[0, 1, 2]`.
- **list_candidates** (*int list*) – The list of candidates we want to plot. Should contains integer lower than `n_candidates`. By default, we plot every candidates.
- **list_titles** (*str list*) – Contains the title of the plots. Should be the same length than *list_candidates*.
- **row_size** (*int*) – Number of subplots by row. By default, it is set to 5 plots by rows.
- **show** (*bool*) – If True, display the figure at the end of the function.

plot_ratings_candidate(*ratings_candidate*, *title=''*, *plot_kind='3D'*, *dim: list = None*, *fig=None*, *plot_position=None*, *show=True*)

Plot the matrix associated to a candidate.

The embedding of each voter is multiplied by the rating she assigned to the candidate.

Parameters

- **ratings_candidate** (*np.ndarray*) – The rating each voters assigned to the given candidate. Should be of length *n_voters*.
- **title** (*str*) – Title of the figure.
- **plot_kind** (*str*) – The kind of plot we want to show. Can be `'3D'` or `'ternary'`.
- **fig** (*matplotlib figure*) – The figure on which we add the plot.
- **plot_position** (*list*) – The position of the plot on the figure. Should be of the form `[n_rows, n_columns, position]`.
- **dim** (*list*) – The 3 dimensions we are using for our plot. By default, it is set to `[0, 1, 2]`.
- **show** (*bool*) – If True, display the figure at the end of the function.

Returns The matplotlib ax with the figure, if you want to add something to it.

Return type matplotlib ax

recentered(*approx=True*)

Recenter the embeddings so that their new center is [1, . . . , 1].

Parameters **approx** (*bool*) – Passed to `get_center()` in order to compute the center of the voters' embeddings.

Returns A new Embeddings object with the recentered embeddings.

Return type *Embeddings*

Examples

```
>>> embeddings = Embeddings(-np.array([[.5,.9,.4],[.4,.7,.5],[.4,.2,.4]]),
↪norm=True)
>>> embeddings
Embeddings([[-0.45267873, -0.81482171, -0.36214298],
            [-0.42163702, -0.73786479, -0.52704628],
            [-0.66666667, -0.33333333, -0.66666667]])
>>> embeddings.recentered()
```

(continues on next page)

(continued from previous page)

```
Embeddings([[0.40215359, 0.75125134, 0.52334875],
            [0.56352875, 0.6747875 , 0.47654713],
            [0.70288844, 0.24253193, 0.66867489]])
```

```
>>> embeddings = Embeddings([[1, 0], [np.sqrt(3)/2, 1/2], [1/2, np.sqrt(3)/
→2]], norm=True)
>>> embeddings
Embeddings([[1.        , 0.        ],
            [0.8660254, 0.5       ],
            [0.5      , 0.8660254]])
>>> embeddings.recentered(approx=False)
Embeddings([[0.96592583, 0.25881905],
            [0.70710678, 0.70710678],
            [0.25881905, 0.96592583]])
```

recentered_and_dilated(*approx=True*)

Recenter and dilate.

This is just a shortcut for the (common) operation `recentered()`, then `dilated_new()`.

Parameters **approx** (*bool*) – Passed to `get_center()` in order to compute the center of the voters' embeddings.

Returns A new Embeddings object with the recentered and dilated embeddings.

Return type *Embeddings*

Examples

```
>>> embeddings = Embeddings([[1, 0], [np.sqrt(3)/2, 1/2], [1/2, np.sqrt(3)/
→2]], norm=True)
>>> embeddings
Embeddings([[1.        , 0.        ],
            [0.8660254, 0.5       ],
            [0.5      , 0.8660254]])
>>> new_embeddings = embeddings.recentered_and_dilated(approx=False)
>>> np.abs(np.round(new_embeddings, 4))
array([[1.    , 0.    ],
       [0.7071, 0.7071],
       [0.    , 1.    ]])
```

times_ratings_candidate(*ratings_candidate*)

This method computes the embeddings multiplied by the ratings given by the voters to a given candidate. For each voter, its embeddings are multiplied by the given rating.

Parameters **ratings_candidate** (*np.ndarray*) – The vector of ratings given by the voters to a given candidate.

Returns A new Embeddings object, where the embedding of each voter is multiplied by the rating she assigned to the candidate.

Return type Embddings

Examples

```
>>> embeddings = Embeddings(np.array([[1, 0], [0, 1], [0.5, 0.5]]),␣
↪norm=False)
>>> embeddings.times_ratings_candidate(np.array([.8, .5, .4]))
Embeddings([[0.8, 0. ],
            [0. , 0.5],
            [0.2, 0.2]])
```

6.3.2 Embeddings generator

class embedded_voting.**EmbeddingsGenerator**(*n_voters*, *n_dim*)

This abstract class creates Embeddings from scratch using some function.

Parameters

- **n_voters** (*int*) – Number of voters in the embeddings.
- **n_dim** (*int*) – Number of dimensions for the embeddings.

Random Embeddings

class embedded_voting.**EmbeddingsGeneratorUniform**(*n_voters*, *n_dim*)

Create random embeddings uniformly on the non-negative orthant.

The embedding of each voter is a unit vector that is uniformly drawn on the intersection of the unit sphere with the non-negative orthant.

Examples

```
>>> np.random.seed(42)
>>> generator = EmbeddingsGeneratorUniform(10, 2)
>>> generator()
Embeddings([[0.96337365, 0.26816265],
            [0.39134578, 0.92024371],
            [0.70713157, 0.70708199],
            [0.89942118, 0.43708299],
            [0.65433791, 0.75620229],
            [0.70534506, 0.70886413],
            [0.1254653 , 0.99209801],
            [0.95076   , 0.30992809],
            [0.95508537, 0.29633078],
            [0.54080587, 0.84114744]])
```

From correlations

class embedded_voting.**EmbeddingsCorrelation**

Embeddings based on correlation, dedicated to *RuleFast*.

Parameters

- **positions** (*np.ndarray or list or Embeddings*) – The embeddings of the voters. Its dimensions are n_voters, n_dim.

- **n_sing_val** (*int*) – "Effective" number of singular values.
- **ratings_means** (*np.ndarray*) – Mean rating for each voter.
- **ratings_stds** (*np.ndarray*) – Standard deviation of the ratings for each voter.
- **norm** (*bool*) – If True, normalize the embeddings.

Examples

```
>>> embeddings = EmbeddingsCorrelation([[1, 2], [3, 4]], n_sing_val=2, ratings_
↪means=[.1, .2],
...                                     ratings_stds=[.3, .4], norm=True)
>>> embeddings
EmbeddingsCorrelation([[0.4472136 , 0.89442719],
                       [0.6       , 0.8       ]])
>>> embeddings.n_sing_val
2
>>> embeddings.ratings_means
[0.1, 0.2]
```

```
>>> embeddings2 = embeddings.copy()
>>> embeddings2.n_sing_val
2
```

6.3.3 Polarized Embeddings

class embedded_voting.**EmbeddingsGeneratorPolarized**(*n_voters*, *n_dim*, *prob=None*)

Generates parametrized embeddings with `n_dim` groups of voters. This class creates two embeddings: one according to uniform distribution, the other one fully polarized (with groups of voters on the canonical basis), and we can parametrize the embeddings to get one distribution between these two extremes.

Parameters

- **n_voters** (*int*) – Number of voters in the embeddings.
- **n_dim** (*int*) – Number of dimensions for the embeddings.
- **prob** (*list*) – The probabilities for each voter to be in each group. Default is uniform distribution.

Examples

```
>>> np.random.seed(42)
>>> generator = EmbeddingsGeneratorPolarized(10, 2)
>>> generator(polarisation=1)
Embeddings([[1., 0.],
            [0., 1.],
            [1., 0.],
            [0., 1.],
            [0., 1.],
            [1., 0.],
            [0., 1.],
            [1., 0.],
            [1., 0.],
```

(continues on next page)

(continued from previous page)

```
             [0., 1.]])
>>> generator(polarisation=0)
Embeddings([[0.96337365, 0.26816265],
            [0.39134578, 0.92024371],
            [0.70713157, 0.70708199],
            [0.89942118, 0.43708299],
            [0.65433791, 0.75620229],
            [0.70534506, 0.70886413],
            [0.1254653 , 0.99209801],
            [0.95076   , 0.30992809],
            [0.95508537, 0.29633078],
            [0.54080587, 0.84114744]])
>>> generator(polarisation=0.5)
Embeddings([[0.9908011 , 0.13532618],
            [0.19969513, 0.97985808],
            [0.92388624, 0.38266724],
            [0.53052663, 0.84766827],
            [0.34914017, 0.93707051],
            [0.92340269, 0.38383261],
            [0.06285695, 0.99802255],
            [0.98761328, 0.15690762],
            [0.98870758, 0.14985764],
            [0.28182668, 0.95946533]])
```

class embedded_voting.**EmbeddingsGeneratorFullyPolarized**(*n_voters*, *n_dim*, *prob=None*)

Create embeddings that are random vectors of the canonical basis.

Parameters

- **n_voters** (*int*) – Number of voters in the embeddings.
- **n_dim** (*int*) – Number of dimensions for the embeddings.
- **prob** (*list*) – The probabilities for each voter to be in each group. Default is uniform distribution.

Examples

```
>>> np.random.seed(42)
>>> generator = EmbeddingsGeneratorFullyPolarized(10, 5)
>>> generator()
Embeddings([[0., 1., 0., 0., 0.],
            [0., 0., 0., 0., 1.],
            [0., 0., 0., 1., 0.],
            [0., 0., 1., 0., 0.],
            [1., 0., 0., 0., 0.],
            [1., 0., 0., 0., 0.],
            [1., 0., 0., 0., 0.],
            [0., 0., 0., 0., 1.],
            [0., 0., 0., 1., 0.],
            [0., 0., 0., 1., 0.]])
```

6.4 Linking Ratings and Embeddings

6.4.1 Ratings From Embeddings

class embedded_voting.**RatingsFromEmbeddings**(*n_candidates*)

This abstract class is used to generate ratings from embeddings.

Parameters n_candidates (*int*) – The number of candidates wanted in the ratings.

6.4.2 Embeddings From Ratings Correlation

class embedded_voting.**EmbeddingsFromRatingsCorrelation**(*preprocess_ratings=None*, *svd_factor=0.95*)

Use the correlation with each voter as the embeddings.

Morally, we have two levels of embedding.

- First, *v_i = preprocess_ratings(ratings_voter_i)* for each voter *i*, which is used as a computation step but not recorded.
- Second, *M = v @ v.T*, which is recorded as the final embeddings.

Other attributes are computed and recorded:

- *n_sing_val*: the number of relevant singular values when we compute the SVD. This is based on the Principal Component Analysis (PCA).
- *ratings_means*: the mean rating for each voter (without preprocessing).
- *ratings_stds*: the standard deviation of the ratings for each voter (without preprocessing).

Examples

```
>>> np.random.seed(42)
>>> ratings = np.ones((5, 3))
>>> generator = EmbeddingsFromRatingsCorrelation(preprocess_ratings=normalize)
>>> embeddings = generator(ratings)
>>> embeddings
EmbeddingsCorrelation([[1., 1., 1., 1., 1.],
                       [1., 1., 1., 1., 1.],
                       [1., 1., 1., 1., 1.],
                       [1., 1., 1., 1., 1.],
                       [1., 1., 1., 1., 1.]])
>>> embeddings.n_sing_val
1
```

In fact, the typical usage is with *center_and_normalize*:

```
>>> generator = EmbeddingsFromRatingsCorrelation(preprocess_ratings=center_and_
↪normalize)
>>> embeddings = generator(ratings)
>>> embeddings
EmbeddingsCorrelation([[0., 0., 0., 0., 0.],
                       [0., 0., 0., 0., 0.],
                       [0., 0., 0., 0., 0.],
                       [0., 0., 0., 0., 0.],
```

(continues on next page)

(continued from previous page)

```
                        [0., 0., 0., 0., 0.]])
>>> embeddings.n_sing_val
0
```

6.4.3 Embeddings From Ratings

class embedded_voting.**EmbeddingsFromRatings**

An abstract class that convert ratings into embeddings using some function.

Random

class embedded_voting.**EmbeddingsFromRatingsRandom**(*n_dim=0*)

Generates random normalized embeddings for the voters.

The embeddings of the voters are drawn uniformly at random on the part of the sphere where all coordinates are positive. These embeddings actually does not take the ratings into account.

Examples

```
>>> np.random.seed(42)
>>> ratings = np.array([[1, 0], [1, 1], [0, 1]])
>>> embeddings_from_ratings = EmbeddingsFromRatingsRandom(n_dim=5)
>>> embeddings_from_ratings(ratings)
Embeddings([[0.28396232, 0.07904315, 0.37027159, 0.87068807, 0.13386116],
            [0.12251149, 0.82631858, 0.40155802, 0.24565113, 0.28389299],
            [0.17359769, 0.1744638 , 0.09063981, 0.71672067, 0.64615953]])
```

Identity

class embedded_voting.**EmbeddingsFromRatingsIdentity**

Use the identity matrix as the embeddings for the voters.

Intuitively, each voter is alone in her group. These embeddings actually does not take the ratings into account.

Examples

```
>>> ratings = np.array([[.4, .6], [.1, .9], [.7, .5]])
>>> embeddings_from_ratings = EmbeddingsFromRatingsIdentity()
>>> embeddings_from_ratings(ratings)
Embeddings([[1., 0., 0.],
            [0., 1., 0.],
            [0., 0., 1.]])
```

Self

class embedded_voting.**EmbeddingsFromRatingsSelf**(*norm*)

Use the normalized ratings as the embeddings for the voters.

Parameters **norm** (*bool*) – Whether the embeddings should be normalized.

Examples

```
>>> ratings = np.array([[1, 0], [1, 1], [0, 1]])
>>> embeddings_from_ratings = EmbeddingsFromRatingsSelf(norm=True)
>>> embeddings_from_ratings(ratings)
Embeddings([[1.        , 0.        ],
            [0.70710678, 0.70710678],
            [0.        , 1.        ]])
```

6.4.4 Correlated Ratings From Embeddings

class embedded_voting.**RatingsFromEmbeddingsCorrelated**(*coherence=0*, *ratings_dim_candidate=None*, *n_dim=None*, *n_candidates=None*, *minimum_random_rating=0*, *maximum_random_rating=1*, *clip=False*)

Generate ratings from embeddings and from a matrix where each embedding dimension gives a rating to each candidate.

ratings_automatic[voter, candidate] is computed as the average of *ratings_dim_candidate[:, candidate]*, weighted by the squares of *emdeddings[voter, :]*. In particular, for each *voter* belonging to group *i* (in the sense that their embedding is the i-th vector of the canonical basis), then *ratings_automatic[voter, candidate]* is equal to *ratings_dim_candidate[i, candidate]*.

ratings_random[voter, candidate] is computed as a uniform random number between *minimum_random_rating* and *maximum_random_rating*.

Finally, *ratings* is the barycenter: *coherence * ratings_automatic + (1 - coherence) * ratings_random*.

Parameters

- **coherence** (*float*) – Between 0 and 1, indicates the degree of coherence between voters having similar embeddings. If 0, the ratings are purely random. If 1, the ratings are automatically deduced from *embeddings* and *ratings_dim_candidate*.
- **ratings_dim_candidate** (*np.ndarray or list*) – An array with shape `n_dim`, `n_candidates`. The coefficient *ratings_dim_candidate[dim, candidate]* is the score given by the group represented by the dimension *dim* to the *candidate*. By default, it is set at random with a uniform distribution in the interval [*minimum_random_rating*, *maximum_random_rating*].
- **n_dim** (*int*) – The number of dimension of the embeddings. Used to generate *ratings_dim_candidate* if it is not specified.
- **n_candidates** (*int*) – The number of candidates. Used to generate *ratings_dim_candidate* if it is not specified.
- **minimum_random_rating** (*float*) – Minimum rating for the random part.
- **maximum_random_rating** (*float*) – Maximum rating for the random part.
- **clip** (*bool*) – If true, the final ratings are clipped in the interval [*minimum_random_rating*, *maximum_random_rating*].

Examples

```
>>> np.random.seed(42)
>>> embeddings = Embeddings(np.array([[0, 1], [1, 0], [1, 1]]), norm=True)
>>> generator = RatingsFromEmbeddingsCorrelated(coherence=.5, ratings_dim_
↪candidate=np.array([[.8,.4],[.1,.7]]))
>>> generator(embeddings)
Ratings([[0.23727006, 0.82535715],
         [0.76599697, 0.49932924],
         [0.30300932, 0.35299726]])
```

6.5 Voting Rules

6.5.1 Single-Winner voting rules

General Class

class embedded_voting.**Rule**(*score_components=1*, *embeddings_from_ratings=None*)

The general class of functions for scoring rules. These rules aggregate the scores of every voter to create a ranking of the candidates and select a winner.

Parameters

- **score_components** (*int*) – The number of components in the aggregated score of every candidate. If > *1*, we perform a lexical sort to obtain the ranking.
- **embeddings_from_ratings** (EmbeddingsFromRatings) – If no embeddings are specified in the call, this *EmbeddingsFromRatings* object is use to generate the embeddings from the ratings. Default: *EmbeddingsFromRatingsIdentity()*.

ratings_

The ratings of voters on which we run the election.

Type *Ratings*

embeddings_

The embeddings of the voters on which we run the election.

Type *Embeddings*

plot_ranking(*plot_kind='3D'*, *dim=None*, *row_size=5*, *show=True*)

Plot the matrix associated to each candidate, in the same order than the ranking of the election.

Parameters

- **plot_kind** (*str*) – The kind of plot we want to show. Can be `'3D'` or `'ternary'`.
- **dim** (*list*) – The 3 dimensions we are using for our plot. By default, it is set to `[0, 1, 2]`.
- **row_size** (*int*) – Number of subplots by row. By default, it is set to 5 by rows.
- **show** (*bool*) – If True, displays the figure at the end of the function.

plot_winner(*plot_kind='3D'*, *dim=None*, *fig=None*, *plot_position=None*, *show=True*)

Plot the matrix associated to the winner of the election.

Cf. *Embeddings.plot_candidate()*.

Parameters

- **plot_kind** (*str*) – The kind of plot we want to show. Can be `'3D'` or `'ternary'`.
- **dim** (*list*) – The 3 dimensions we are using for our plot. By default, it is set to `[0, 1, 2]`.
- **fig** (*matplotlib figure*) – The figure on which we add the plot.
- **plot_position** (*list*) – The position of the plot on the figure. Should be of the form `[n_rows, n_columns, position]`.
- **show** (*bool*) – If True, displays the figure at the end of the function.

Returns The ax with the plot.

Return type matplotlib ax

ranking_

Return the ranking of the candidates based on their aggregated scores.

Returns The ranking of the candidates. In case of tie, candidates with lower indices are favored.

Return type list of int

score_(*candidate*)

Return the aggregated score of a given candidate.

Parameters **candidate** (*int*) – Index of the candidate for whom we want the score.

Returns if `score_components = 1`, return a float, otherwise a tuple of length `score_components`.

Return type float or tuple

scores_

Return the aggregated scores of all candidates.

Returns The scores of all candidates. The score of each candidate is a float if `score_components` = 1 and a tuple of length `score_components` otherwise.

Return type list

scores_focus_on_last_

Return the last score component of each candidate, but only if the other score components are maximal.

If `score_components` is 1, return *scores_*. Otherwise, for each candidate:

- Return the last score component if all other components are maximal.
- Return 0 otherwise.

Note that if the last score component is defined as non-negative, and if it is always positive for the winner, then *scores_focus_on_last_* is enough to determine which candidate has the best score by lexicographical order.

Returns The scores of every candidates.

Return type float list

Examples

Cf. *RuleMaxParallelepiped*.

welfare_

Return the welfare of all candidates, where the welfare is defined as *(score - score_min)/(score_max - score_min)*.

If scores are tuple, then *scores_focus_on_last_* is used.

If *score_max* = *score_min*, then by convention, all candidates have a welfare of 1.

Returns Welfare of all candidates.

Return type list of float

winner_

Return the winner of the election.

Returns The index of the winner of the election. In case of tie, candidates with lower indices are favored.

Return type int

Trivial Rules

Sum of scores (Range Voting)

class embedded_voting.**RuleSumRatings**(*score_components=1*, *embeddings_from_ratings=None*)

Voting rule in which the score of a candidate is the sum of her ratings.

No embeddings are used for this rule.

Parameters

- **score_components** (*int*) – The number of components in the aggregated score of every candidate. If > *1*, we perform a lexical sort to obtain the ranking.
- **embeddings_from_ratings** (EmbeddingsFromRatings) – If no embeddings are specified in the call, this *EmbeddingsFromRatings* object is use to generate the embeddings from the ratings. Default: *EmbeddingsFromRatingsIdentity()*.

Examples

```
>>> ratings = Ratings(np.array([[.5, .6, .3], [.7, 0, .2], [.2, 1, .8]]))
>>> election = RuleSumRatings()(ratings)
>>> election.scores_
[1.4, 1.6, 1.3]
>>> election.ranking_
[1, 0, 2]
>>> election.winner_
1
>>> election.welfare_
[0.3333333333333328, 1.0, 0.0]
```

Product of scores (Nash)

class embedded_voting.**RuleShiftProduct**(*score_components=1*, *embeddings_from_ratings=None*)

Voting rule in which the score of a candidate is the product of her ratings, shifted by 2, and clamped at 0.1.

No embeddings are used for this rule.

Parameters

- **score_components** (*int*) – The number of components in the aggregated score of every candidate. If > *1*, we perform a lexical sort to obtain the ranking.
- **embeddings_from_ratings** (EmbeddingsFromRatings) – If no embeddings are specified in the call, this *EmbeddingsFromRatings* object is use to generate the embeddings from the ratings. Default: *EmbeddingsFromRatingsIdentity()*.

Examples

```
>>> ratings = Ratings(np.array([[.5, .6, .3], [.7, 0, .2], [.2, 1, .8]]))
>>> election = RuleShiftProduct()(ratings)
>>> election.scores_
[14.85..., 15.60..., 14.16...]
>>> election.ranking_
[1, 0, 2]
>>> election.winner_
1
```

Approval Rules

class embedded_voting.**RuleApprovalProduct**(*embeddings_from_ratings=None*)

Voting rule in which the score of a candidate is the number of approval (vote greater than 0) that it gets. Ties are broken by the product of the positive ratings.

More precisely, her score is a tuple whose components are:

- The number of her nonzero ratings.
- The product of her nonzero ratings.

Note that this rule is well suited only if ratings are nonnegative.

No embeddings are used for this rule.

Parameters embeddings_from_ratings (EmbeddingsFromRatings) – If no embeddings are specified in the call, this *EmbeddingsFromRatings* object is use to generate the embeddings from the ratings. Default: *EmbeddingsFromRatingsIdentity()*.

Examples

```
>>> ratings = Ratings(np.array([[.5, .6, .3], [.7, 0, .2], [.2, 1, .8]]))
>>> election = RuleApprovalProduct()(ratings)
>>> election.scores_
[(3, 0.06999999999999999), (2, 0.6), (3, 0.048)]
>>> election.ranking_
[0, 2, 1]
>>> election.winner_
0
>>> election.welfare_
[1.0, 0.0, 0.6857142857142858]
```

class embedded_voting.**RuleApprovalSum**(*embeddings_from_ratings=None*)

Voting rule in which the score of a candidate is the number of approval (vote greater than 0) that it gets. Ties are broken by sum of score (range voting).

More precisely, her score is a tuple whose components are:

- The number of her nonzero ratings.
- The sum of her ratings.

No embeddings are used for this rule.

Parameters **embeddings_from_ratings** (EmbeddingsFromRatings) – If no embeddings are specified in the call, this *EmbeddingsFromRatings* object is use to generate the embeddings from the ratings. Default: *EmbeddingsFromRatingsIdentity()*.

Examples

```
>>> ratings = Ratings(np.array([[.5, .6, .3], [.7, 0, .2], [.2, 1, .8]]))
>>> election = RuleApprovalSum()(ratings)
>>> election.ranking_
[0, 2, 1]
>>> election.scores_
[(3, 1.4), (2, 1.6), (3, 1.3)]
>>> election.winner_
0
>>> election.welfare_
[1.0, 0.0, 0.9285714285714287]
```

class embedded_voting.**RuleApprovalRandom**(*embeddings_from_ratings=None*)

Voting rule in which the score of a candidate is the number of approval (vote greater than 0) that it gets. Ties are broken at random.

More precisely, her score is a tuple whose components are:

- The number of her nonzero ratings.
- A random value.

No embeddings are used for this rule.

Parameters **embeddings_from_ratings** (EmbeddingsFromRatings) – If no embeddings are specified in the call, this *EmbeddingsFromRatings* object is use to generate the embeddings from the ratings. Default: *EmbeddingsFromRatingsIdentity()*.

Examples

```
>>> np.random.seed(42)
>>> ratings = Ratings(np.array([[.5, .6, .3], [.7, 0, .2], [.2, 1, .8]]))
>>> election = RuleApprovalRandom()(ratings)
>>> election.ranking_
[2, 0, 1]
>>> election.scores_
[(3, 0.3745401188473625), (2, 0.9507143064099162), (3, 0.7319939418114051)]
>>> election.winner_
2
>>> election.welfare_
[0.5116710637256354, 0.0, 1.0]
```

Geometric Rules

Zonotope

class embedded_voting.**RuleZonotope**(*embeddings_from_ratings=None*)

Voting rule in which the aggregated score of a candidate is the volume of the zonotope described by his embedding matrix *M* such that *M[i] = score[i, candidate] * embeddings[i]*. (cf `times_ratings_candidate()`).

For each candidate, the rank *r* of her associated matrix is computed. The volume of the zonotope is the sum of the volumes of all the parallelepipeds associated to a submatrix keeping only *r* voters (cf. `volume_parallelepiped()`). The score of the candidate is then *(r, volume)*.

Parameters **embeddings_from_ratings** (`EmbeddingsFromRatings`) – If no embeddings are specified in the call, this *EmbeddingsFromRatings* object is use to generate the embeddings from the ratings. Default: *EmbeddingsFromRatingsIdentity()*.

Examples

```
>>> ratings = Ratings([[1], [1]])
>>> embeddings = Embeddings([[1, 0, 0], [-.5, 1, 0]], norm=False)
>>> election = RuleZonotope()(ratings, embeddings)
>>> election.scores_
[(2, 1.0)]
```

```
>>> ratings = Ratings(np.array([[.5, .6, .3], [.7, 0, .2], [.2, 1, .8]]))
>>> embeddings = Embeddings(np.array([[1, 1], [1, 0], [0, 1]]), norm=True)
>>> election = RuleZonotope()(ratings, embeddings)
>>> election.scores_  # doctest: +ELLIPSIS
[(2, 0.458...), (2, 0.424...), (2, 0.372...)]
>>> election.ranking_
[0, 1, 2]
>>> election.winner_
0
>>> election.welfare_  # doctest: +ELLIPSIS
[1.0, 0.605..., 0.0]
```

Max Parallelepiped

class embedded_voting.**RuleMaxParallelepiped**(*embeddings_from_ratings=None*)

Voting rule in which the aggregated score of a candidate is the volume of a parallelepiped described by `n_dim` rows of the candidate embedding matrix *M* such that *M[i] = score[i, candidate] * embeddings[i]*. (cf `times_ratings_candidate()`).

For each candidate, the rank *r* of her associated matrix is computed. Then we choose *r* voters in order to maximize the volume of the parallelepiped associated to the submatrix keeping only these voters (cf. `volume_parallelepiped()`). The score of the candidate is then *(r, volume)*.

Parameters **embeddings_from_ratings** (`EmbeddingsFromRatings`) – If no embeddings are specified in the call, this *EmbeddingsFromRatings* object is use to generate the embeddings from the ratings. Default: *EmbeddingsFromRatingsIdentity()*.

Examples

```
>>> ratings = Ratings(np.array([[.5, .6, .3], [.7, 0, .2], [.2, 1, .8]]))
>>> embeddings = Embeddings(np.array([[1, 1], [1, 0], [0, 1]]), norm=True)
>>> election = RuleMaxParallelepiped()(ratings, embeddings)
>>> election.scores_  # doctest: +ELLIPSIS
[(2, 0.24...), (2, 0.42...), (2, 0.16...)]
>>> election.ranking_
[1, 0, 2]
>>> election.winner_
1
>>> election.welfare_  # doctest: +ELLIPSIS
[0.305..., 1.0, 0.0]
```

```
>>> ratings = Ratings([[1, 10], [1, 10], [1, 0]])
>>> embeddings = Embeddings([[1, 0, 0], [0, 1, 0], [0, 0, 1]], norm=False)
>>> election = RuleMaxParallelepiped()(ratings, embeddings)
>>> election.scores_  # doctest: +ELLIPSIS
[(3, 1.0), (2, 100.0...)]
>>> election.scores_focus_on_last_
[1.0, 0]
```

SVD Rules

General SVD

class embedded_voting.**RuleSVD**(*aggregation_rule=<function prod>, square_root=True, use_rank=False, embedded_from_ratings=None*)

Voting rule in which the aggregated score of a candidate is based on singular values of his embedding matrix (cf `times_ratings_candidate()`).

Implicitly, ratings are assumed to be nonnegative.

Parameters

- **aggregation_rule** (*callable*) – The aggregation rule for the singular values. Input : float list. Output : float. By default, it is the product of the singular values.
- **square_root** (*boolean*) – If True, use the square root of ratings in the matrix. By default, it is True.
- **use_rank** (*boolean*) – If True, consider the rank of the matrix when doing the ranking. By default, it is False.
- **embedded_from_ratings** (EmbeddingsFromRatings) – If no embeddings are specified in the call, this *EmbeddingsFromRatings* object is use to generate the embeddings from the ratings. Default: *EmbeddingsFromRatingsIdentity()*.

Examples

```
>>> ratings = Ratings(np.array([[.5, .6, .3], [.7, 0, .2], [.2, 1, .8]]))
>>> embeddings = Embeddings(np.array([[1, 1], [1, 0], [0, 1]]), norm=True)
>>> election = RuleSVD()(ratings, embeddings)
>>> election.scores_  # DOCTEST: +ELLIPSIS
[0.6041522986797..., 0.547722557505..., 0.5567764362830...]
>>> election.ranking_
[0, 2, 1]
```

(continues on next page)

(continued from previous page)

```
>>> election.winner_
0
>>> election.welfare_  # DOCTEST: +ELLIPSIS
[1.0, 0.0, 0.16044515869439...]
```

Special cases

class embedded_voting.**RuleSVDSum**(*square_root=True*, *use_rank=False*, *embedded_from_ratings=None*)

Voting rule in which the aggregated score of a candidate is the sum of the singular values of his embedding matrix (cf `times_ratings_candidate()`).

Parameters

- **square_root** (*boolean*) – If True, use the square root of score in the matrix. By default, it is True.
- **use_rank** (*boolean*) – If True, consider the rank of the matrix when doing the ranking. By default, it is False.
- **embedded_from_ratings** (`EmbeddingsFromRatings`) – If no embeddings are specified in the call, this *EmbeddingsFromRatings* object is use to generate the embeddings from the ratings. Default: *EmbeddingsFromRatingsIdentity()*.

Examples

```
>>> ratings = Ratings(np.array([[.5, .6, .3], [.7, 0, .2], [.2, 1, .8]]))
>>> embeddings = Embeddings(np.array([[1, 1], [1, 0], [0, 1]]), norm=True)
>>> election = RuleSVDSum()(ratings, embeddings)
>>> election.scores_  # DOCTEST: +ELLIPSIS
[1.6150246429573..., 1.6417810801109..., 1.5535613514007...]
>>> election.ranking_
[1, 0, 2]
>>> election.winner_
1
>>> election.welfare_  # DOCTEST: +ELLIPSIS
[0.6967068756070..., 1.0, 0.0]
```

class embedded_voting.**RuleSVDNash**(*square_root=True*, *use_rank=False*, *embedded_from_ratings=None*)

Voting rule in which the aggregated score of a candidate is the product of the singular values of his embedding matrix (cf `times_ratings_candidate()`).

Parameters

- **square_root** (*boolean*) – If True, use the square root of score in the matrix. By default, it is True.
- **use_rank** (*boolean*) – If True, consider the rank of the matrix when doing the ranking. By default, it is False.
- **embedded_from_ratings** (`EmbeddingsFromRatings`) – If no embeddings are specified in the call, this *EmbeddingsFromRatings* object is use to generate the embeddings from the ratings. Default: *EmbeddingsFromRatingsIdentity()*.

Examples

```
>>> ratings = Ratings(np.array([[.5, .6, .3], [.7, 0, .2], [.2, 1, .8]]))
>>> embeddings = Embeddings(np.array([[1, 1], [1, 0], [0, 1]]), norm=True)
>>> election = RuleSVDNash()(ratings, embeddings)
>>> election.scores_  # DOCTEST: +ELLIPSIS
[0.6041522986797..., 0.547722557505..., 0.5567764362830...]
>>> election.ranking_
[0, 2, 1]
>>> election.winner_
0
>>> election.welfare_  # DOCTEST: +ELLIPSIS
[1.0, 0.0, 0.16044515869439...]
```

class embedded_voting.**RuleSVDMin**(*square_root=True*, *use_rank=False*, *embedded_from_ratings=None*)

Voting rule in which the aggregated score of a candidate is the minimum singular value of his embedding matrix (cf `times_ratings_candidate()`).

Parameters

- **square_root** (*boolean*) – If True, use the square root of score in the matrix. By default, it is True.
- **use_rank** (*boolean*) – If True, consider the rank of the matrix when doing the ranking. By default, it is False.
- **embedded_from_ratings** (`EmbeddingsFromRatings`) – If no embeddings are specified in the call, this *EmbeddingsFromRatings* object is use to generate the embeddings from the ratings. Default: *EmbeddingsFromRatingsIdentity()*.

Examples

```
>>> ratings = Ratings(np.array([[.5, .6, .3], [.7, 0, .2], [.2, 1, .8]]))
>>> embeddings = Embeddings(np.array([[1, 1], [1, 0], [0, 1]]), norm=True)
>>> election = RuleSVDMin()(ratings, embeddings)
>>> election.scores_
[0.5885971537535042, 0.4657304054015261, 0.5608830567730065]
>>> election.ranking_
[0, 2, 1]
>>> election.winner_
0
>>> election.welfare_
[1.0, 0.0, 0.7744377762720253]
```

class embedded_voting.**RuleSVDMax**(*square_root=True*, *use_rank=False*, *embedded_from_ratings=None*)

Voting rule in which the aggregated score of a candidate is the maximum singular value of his embedding matrix (cf `times_ratings_candidate()`).

Parameters

- **square_root** (*boolean*) – If True, use the square root of score in the matrix. By default, it is True.
- **use_rank** (*boolean*) – If True, consider the rank of the matrix when doing the ranking. By default, it is False.

- **embedded_from_ratings** (EmbeddingsFromRatings) – If no embeddings are specified in the call, this *EmbeddingsFromRatings* object is use to generate the embeddings from the ratings. Default: *EmbeddingsFromRatingsIdentity()*.

Examples

```
>>> ratings = Ratings(np.array([[.5, .6, .3], [.7, 0, .2], [.2, 1, .8]]))
>>> embeddings = Embeddings(np.array([[1, 1], [1, 0], [0, 1]]), norm=True)
>>> election = RuleSVDMax()(ratings, embeddings)
>>> election.scores_  # DOCTEST: +ELLIPSIS
[1.0264274892038..., 1.1760506747094..., 0.9926782946277...]
>>> election.ranking_
[1, 0, 2]
>>> election.winner_
1
>>> election.welfare_  # DOCTEST: +ELLIPSIS
[0.184047317055..., 1.0, 0.0]
```

features_

A function to get the feature vectors of all the candidates. The feature vector is defined as the singular vector associated to the maximal singular value.

Returns The feature vectors of all the candidates, of shape *n_candidates*, *n_dim*.

Return type np.ndarray

Examples

```
>>> ratings = Ratings(np.array([[.5, .6, .3], [.7, 0, .2], [.2, 1, .8]]))
>>> embeddings = Embeddings(np.array([[1, 1], [1, 0], [0, 1]]), norm=True)
>>> election = RuleSVDMax()(ratings, embeddings)
>>> election.features_
array([[0.94829535, 0.39279679],
       [0.31392742, 1.13337759],
       [0.22807074, 0.96612315]])
```

plot_features (*plot_kind='3D'*, *dim=None*, *row_size=5*, *show=True*)

This function plot the features vector of every candidates in the given dimensions.

Parameters

- **plot_kind** (*str*) – The kind of plot we want to show. Can be `'3D'` or `'ternary'`.
- **dim** (*list*) – The 3 dimensions we are using for our plot. By default, it is set to `'[0, 1, 2]'`.
- **row_size** (*int*) – Number of subplots by row. By default, it is set to 5 by rows.
- **show** (*bool*) – If True, displays the figure at the end of the function.

class embedded_voting.**RuleSVDLog** (*const=1*, *square_root=True*, *use_rank=False*, *embedded_from_ratings=None*)

Voting rule in which the aggregated score of a candidate is the sum of *log(1 + sigma/const)* where sigma are the singular values of his embedding matrix and *const* is a constant.

Parameters

- **const** (*float*) – The constant by which we divide the singular values in the log.

- **square_root** (*boolean*) – If True, use the square root of score in the matrix. By default, it is True.
- **use_rank** (*boolean*) – If True, consider the rank of the matrix when doing the ranking. By default, it is False.
- **embedded_from_ratings** (EmbeddingsFromRatings) – If no embeddings are specified in the call, this *EmbeddingsFromRatings* object is use to generate the embeddings from the ratings. Default: *EmbeddingsFromRatingsIdentity()*.

Examples

```
>>> ratings = Ratings(np.array([[.5, .6, .3], [.7, 0, .2], [.2, 1, .8]]))
>>> embeddings = Embeddings(np.array([[1, 1], [1, 0], [0, 1]]), norm=True)
>>> election = RuleSVDLog()(ratings, embeddings)
>>> election.scores_
[1.169125718695728, 1.1598653051965206, 1.1347313336962574]
>>> election.ranking_
[0, 1, 2]
>>> election.winner_
0
>>> election.welfare_
[1.0, 0.7307579856610341, 0.0]
```

Features Rule

class embedded_voting.**RuleFeatures**(*score_components=1*, *embeddings_from_ratings=None*)

Voting rule in which the aggregated score of a candidate is the norm of the feature vector of this candidate.

Intuitively, for each candidate, her feature on embedding dimension *d* is the ideal rating that a voter of group *d* should put to that candidate. In this model, the actual rating of a voter for this candidate would be a mean of the features, weighted by the voter's embedding: *embeddings[voter, :] @ features[candidate, :]*. Considering all the voters and all the candidates, we then obtain *ratings = embeddings @ features.T*, i.e. *features = (inv(embeddings) @ ratings).T*.

Since *embeddings* is not always invertible, we consider in practice *features = (pinv(embeddings) @ ratings).T*. This can be seen as a least-square approximation of the inital model.

Finally, the score of a candidate is the Euclidean norm of her vector of features.

Examples

```
>>> ratings = Ratings(np.array([[.5, .6, .3], [.7, 0, .2], [.2, 1, .8]]))
>>> embeddings = Embeddings(np.array([[1, 1], [1, 0], [0, 1]]), norm=True)
>>> election = RuleFeatures()(ratings, embeddings)
>>> election.scores_
[0.669..., 0.962..., 0.658...]
>>> election.ranking_
[1, 0, 2]
>>> election.winner_
1
>>> election.welfare_
[0.0353..., 1.0, 0.0]
```

features_

This function return the feature vector of all candidates.

Returns The matrix of features. Its shape is `n_candidates`, `n_dim`.

Return type np.ndarray

plot_features (*plot_kind='3D'*, *dim: list = None*, *row_size=5*, *show=True*)

This function plot the features vector of all candidates in the given dimensions.

Parameters

- **plot_kind** (*str*) – The kind of plot we want to show. Can be `'3D'` or `'ternary'`.
- **dim** (*list*) – The 3 dimensions we are using for our plot. By default, it is set to `[0, 1, 2]`.
- **row_size** (*int*) – The number of subplots by row. By default, it is set to 5 plots by row.
- **show** (*bool*) – If True, plot the figure at the end of the function.

Fast Rules

Fast

class embedded_voting.**RuleFast** (*embeddings_from_ratings=None*, *f=None*, *aggregation_rule=<function prod>*)

Voting rule in which the aggregated score of a candidate is based on singular values of his score matrix.

Parameters

- **embeddings_from_ratings** (`EmbeddingsFromRatingsCorrelation`) – If no embeddings are specified in the call, this *EmbeddingsFromRatings* object is use to generate the embeddings from the ratings. Default: *EmbeddingsFromRatingsCorrelation(preprocess_ratings=center_and_normalize)*.
- **f** (*callable*) – The transformation for the ratings given by each voter. Input : (ratings_v: np.ndarray, history_mean: Number, history_std: Number). Output : modified_ratings_v: np.ndarray.
- **aggregation_rule** (*callable*) – The aggregation rule for the singular values. Input : list of float. Output : float. By default, it is the product of the singular values.

Examples

```
>>> ratings = np.array([[.5, .6, .3], [.7, 0, .2], [.2, 1, .8]])
>>> election = RuleFast()(ratings)
>>> election.ranking_
[1, 0, 2]
>>> election.winner_
1
```

modified_ratings_

Modified ratings. For each voter, *f* is applied to her original ratings.

Type *Ratings*

Variants

class embedded_voting.**RuleFastNash**(*embeddings_from_ratings=None*, *f=None*)
Voting rule in which the aggregated score of a candidate is the product of the important singular values of his score matrix.

Parameters

- **embeddings_from_ratings** (EmbeddingsFromRatingsCorrelation) – If no embeddings are specified in the call, this *EmbeddingsFromRatings* object is use to generate the embeddings from the ratings. Default: *EmbeddingsFromRatingsCorrelation(preprocess_ratings=center_and_normalize)*.
- **f** (*callable*) – The transformation for the ratings given by each voter. Input : (ratings_v: np.ndarray, history_mean: Number, history_std: Number). Output : modified_ratings_v: np.ndarray.

Examples

```
>>> ratings = np.array([[.5, .6, .3], [.7, 0, .2], [.2, 1, .8]])
>>> election = RuleFastNash()(ratings)
>>> election.ranking_
[1, 0, 2]
>>> election.winner_
1
```

class embedded_voting.**RuleFastSum**(*embeddings_from_ratings=None*, *f=None*)
Voting rule in which the aggregated score of a candidate is the sum of the important singular values of his score matrix.

Parameters

- **embeddings_from_ratings** (EmbeddingsFromRatingsCorrelation) – If no embeddings are specified in the call, this *EmbeddingsFromRatings* object is use to generate the embeddings from the ratings. Default: *EmbeddingsFromRatingsCorrelation(preprocess_ratings=center_and_normalize)*.
- **f** (*callable*) – The transformation for the ratings given by each voter. Input : (ratings_v: np.ndarray, history_mean: Number, history_std: Number). Output : modified_ratings_v: np.ndarray.

Examples

```
>>> ratings = np.array([[.5, .6, .3], [.7, 0, .2], [.2, 1, .8]])
>>> election = RuleFastSum()(ratings)
>>> election.ranking_
[1, 0, 2]
>>> election.winner_
1
```

class embedded_voting.**RuleFastMin**(*embeddings_from_ratings=None*, *f=None*)
Voting rule in which the aggregated score of a candidate is the minimum of the important singular values of his score matrix.

Parameters

- **embeddings_from_ratings** (`EmbeddingsFromRatingsCorrelation`) – If no embeddings are specified in the call, this *EmbeddingsFromRatings* object is use to generate the embeddings from the ratings. Default: *EmbeddingsFromRatingsCorrelation(preprocess_ratings=center_and_normalize)*.
- **f** (*callable*) – The transformation for the ratings given by each voter. Input : (ratings_v: np.ndarray, history_mean: Number, history_std: Number). Output : modified_ratings_v: np.ndarray.

Examples

```
>>> ratings = np.array([[.5, .6, .3], [.7, 0, .2], [.2, 1, .8]])
>>> election = RuleFastMin()(ratings)
>>> election.ranking_
[1, 0, 2]
>>> election.winner_
1
```

class embedded_voting.**RuleFastLog**(*embeddings_from_ratings=None*, *f=None*)

Voting rule in which the aggregated score of a candidate is the log sum of the important singular values of his score matrix.

Parameters

- **embeddings_from_ratings** (`EmbeddingsFromRatingsCorrelation`) – If no embeddings are specified in the call, this *EmbeddingsFromRatings* object is use to generate the embeddings from the ratings. Default: *EmbeddingsFromRatingsCorrelation(preprocess_ratings=center_and_normalize)*.
- **f** (*callable*) – The transformation for the ratings given by each voter. Input : (ratings_v: np.ndarray, history_mean: Number, history_std: Number). Output : modified_ratings_v: np.ndarray.

Examples

```
>>> ratings = np.array([[.5, .6, .3], [.7, 0, .2], [.2, 1, .8]])
>>> election = RuleFastLog()(ratings)
>>> election.ranking_
[1, 0, 2]
>>> election.winner_
1
```

Maximum Likelihood

MLE Gaussian

class embedded_voting.**RuleMLEGaussian**(*embeddings_from_ratings=None*, *tol=1e-06*)

A rule that computes the scores of the candidates, assuming that the embeddings of the voters correspond to a covariance matrix.

For this rule, the embeddings must be a matrix *n_voters* * *n_voters*.

Examples

Consider a generating epistemic model, where the true value of each candidate is uniformly drawn in a given interval, and where the voters add a noise which is multivariate Gaussian.

```
>>> np.random.seed(42)
>>> covariance_matrix = np.array([
...     [2.02, 1.96, 0.86, 0.81, 1.67],
...     [1.96, 3.01, 1.46, 0.69, 1.59],
...     [0.86, 1.46, 0.94, 0.39, 0.7 ],
...     [0.81, 0.69, 0.39, 0.51, 0.9 ],
...     [1.67, 1.59, 0.7 , 0.9 , 1.78]
... ])
>>> ratings_generator = RatingsGeneratorEpistemicMultivariate(covariance_
↪matrix=covariance_matrix)
>>> ratings = ratings_generator(n_candidates=2)
>>> ratings_generator.ground_truth_
array([17.73956049, 14.3887844 ])
>>> ratings
Ratings([[17.56232759, 14.51592899],
         [16.82544972, 15.78818081],
         [17.51952581, 14.44449175],
         [17.34964888, 14.4010885 ],
         [16.69480298, 14.9281998 ]])
```

If we know the covariance matrix of the noises, then *RuleMLEGaussian* is the maximum likelihood estimator of the ground truth:

```
>>> election = RuleMLEGaussian()(ratings, embeddings=covariance_matrix)
>>> election.scores_ # doctest: +ELLIPSIS
[268.6683142..., 221.5083075...]
```

Model Aware

class embedded_voting.**RuleModelAware**(*groups_sizes*, *groups_features*, *group_noise=1*, *independent_noise=0*)

A rule that is know the noise parameters of the model and use the maximum likelihood to select the best candidate.

Parameters

- **groups_sizes** (*list of int*) – The number of voters in each group.
- **groups_features** (*np.ndarray of shape (n_groups, n_features)*) – The features of each group.
- **group_noise** (*float*) – The value of the feature noise.
- **independent_noise** (*float*) – The value of the distinct noise.

Examples

```
>>> ratings = Ratings(np.array([[.5, .6, .3], [.7, 0, .2], [.2, 1, .8]]))
>>> election = RuleModelAware([2, 1], [[1, 0], [0, 1]], group_noise=1,␣
↪independent_noise=1)(ratings)
>>> election.ranking_
```

(continues on next page)

(continued from previous page)

```
[1, 2, 0]
>>> election.scores_
[0.5, 0.7, 0.5666666...]
>>> election.winner_
1
```

Extensions to ordinal votes

Positional scoring rules

General class

class embedded_voting.**RulePositional**(*points*, *rule=None*)

This class enables to extend a voting rule to an ordinal input with a positional scoring rule.

Parameters

- **points** (*list*) – The vector of the positional scoring rule. Should be of the same length than the number of candidates. In each ranking, candidate ranked at position *i* get *points[i]* points.
- **rule** (Rule) – The aggregation rule used to determine the aggregated scores of the candidates.

fake_ratings_

The modified ratings of voters (with ordinal scores) on which we run the election.

Type ratings

points

The vector of the positional scoring rule. Should be of the same length than the number of candidates. In each ranking, candidate ranked at position *i* get *points[i]* points.

Type np.ndarray

base_rule

The aggregation rule used to determine the aggregated scores of the candidates.

Type *Rule*

_rule

The aggregation rule instantiated with the `fake_ratings`.

Type *Rule*

_score_components

The number of components in the score of every candidate. If > *1*, we perform a lexical sort to obtain the ranking.

Examples

```
>>> ratings = np.array([[.1, .2, .8, 1], [.7, .9, .8, .6], [1, .6, .1, .3]])
>>> embeddings = Embeddings([[1, 0], [1, 1], [0, 1]], norm=True)
>>> election = RuleSVDNash()(ratings, embeddings)
>>> election.ranking_
[3, 0, 1, 2]
```

(continues on next page)

(continued from previous page)

```
>>> election_bis = RulePositional([2, 1, 1, 0])(ratings, embeddings)
>>> election_bis.fake_ratings_
Ratings([[0. , 0.5, 0.5, 1. ],
         [0.5, 1. , 0.5, 0. ],
         [1. , 0.5, 0. , 0.5]])
>>> election_bis.set_rule(RuleSVDNash())(ratings, embeddings).ranking_
[1, 3, 0, 2]
```

plot_fake_ratings(*plot_kind='3D'*, *dim=None*, *list_candidates=None*, *list_titles=None*, *row_size=5*, *show=True*)

This function plot the candidates in the fake ratings, obtained using the scoring vector `points`.

Parameters

- **plot_kind** (*str*) – The kind of plot we want to show. Can be `'3D'` or `'ternary'`.
- **dim** (*list*) – The 3 dimensions we are using for our plot. By default, it is set to `[0, 1, 2]`.
- **list_candidates** (*int list*) – The list of candidates we want to plot. Should contains integers lower than `n_candidates`. By default, we plot all candidates.
- **list_titles** (*str list*) – Contains the title of the plots. Should be the same length than *list_candidates*.
- **row_size** (*int*) – Number of subplots by row. By default, it is set to 5 plots by rows.
- **show** (*bool*) – If True, displays the figure at the end of the function.

set_rule(*rule*)

This function updates the `base_rule` used for the election.

Parameters **rule** (`Rule`) – The new rule to use.

Returns The object itself.

Return type *RulePositional*

Particular cases

class embedded_voting.**RulePositionalPlurality**(*n_candidates*, *rule=None*)

This class enables to extend a voting rule to an ordinal input with Plurality rule (vector `[1, 0, ..., 0]`).

Parameters **rule** (`Rule`) – The aggregation rule used to determine the aggregated scores of the candidates.

Examples

```
>>> ratings = np.array([[.1, .2, .8, 1], [.7, .9, .8, .6], [1, .6, .1, .3]])
>>> embeddings = Embeddings(np.array([[1, 0], [1, 1], [0, 1]]), norm=True)
>>> election = RulePositionalPlurality(4, rule=RuleSVDNash(use_
↪rank=True))(ratings, embeddings)
>>> election.fake_ratings_
Ratings([[0., 0., 0., 1.],
         [0., 1., 0., 0.],
         [1., 0., 0., 0.]])
>>> election.ranking_
[0, 1, 3, 2]
```

class embedded_voting.**RulePositionalVeto**(*n_candidates*, *rule=None*)

This class enables to extend a voting rule to an ordinal input with Veto rule (vector `[1, ..., 1, 0]`).

Parameters **rule** (Rule) – The aggregation rule used to determine the aggregated scores of the candidates.

Examples

```
>>> ratings = np.array([[.1, .2, .8, 1], [.7, .9, .8, .6], [1, .6, .1, .3]])
>>> embeddings = Embeddings(np.array([[1, 0], [1, 1], [0, 1]]), norm=True)
>>> election = RulePositionalVeto(n_candidates=4, rule=RuleSVDNash())(ratings,␣
↪embeddings)
>>> election.fake_ratings_
Ratings([[0., 1., 1., 1.],
         [1., 1., 1., 0.],
         [1., 1., 0., 1.]])
>>> election.ranking_
[1, 3, 0, 2]
```

class embedded_voting.**RulePositionalKApproval**(*n_candidates*, *k=2*, *rule=None*)

This class enables to extend a voting rule to an ordinal input with k-Approval rule (vector `[1, 1, ..., 0]` with *k* ones).

Parameters

- **k** (*int*) – The k parameter of the k-approval. By default, it is set to 2.
- **rule** (Rule) – The aggregation rule used to determine the aggregated scores of the candidates.

Examples

```
>>> ratings = np.array([[.1, .2, .8, 1], [.7, .9, .8, .6], [1, .6, .1, .3]])
>>> embeddings = Embeddings(np.array([[1, 0], [1, 1], [0, 1]]), norm=True)
>>> election = RulePositionalKApproval(n_candidates=4, k=2, rule=RuleSVDNash(use_
↪rank=True))(
...     ratings, embeddings)
>>> election.fake_ratings_
Ratings([[0., 0., 1., 1.],
         [0., 1., 1., 0.],
         [1., 1., 0., 0.]])
>>> election.ranking_
[1, 2, 0, 3]
```

class embedded_voting.**RulePositionalBorda**(*n_candidates*, *rule=None*)

This class enables to extend a voting rule to an ordinal input with Borda rule (vector `[m-1, m-2, ..., 1, 0]`).

Parameters **rule** (Rule) – The aggregation rule used to determine the aggregated scores of the candidates.

Examples

```
>>> ratings = np.array([[.1, .2, .8, 1], [.7, .9, .8, .6], [1, .6, .1, .3]])
>>> embeddings = Embeddings(np.array([[1, 0], [1, 1], [0, 1]]), norm=True)
>>> election = RulePositionalBorda(n_candidates=4, rule=RuleSVDNash())(ratings,␣
↪embeddings)
>>> election.fake_ratings_
Ratings([[0.        , 0.33333333, 0.66666667, 1.        ],
         [0.33333333, 1.        , 0.66666667, 0.        ],
         [1.        , 0.66666667, 0.        , 0.33333333]])
>>> election.ranking_
[1, 3, 2, 0]
```

Instant Runoff voting

class embedded_voting.**RuleInstantRunoff**(*rule=None*)

This class enables to extend a voting rule to an ordinal input with Instant Runoff ranking. You cannot access to the `scores_` because IRV only compute the ranking of the candidates.

Parameters **rule** (`Rule`) – The aggregation rule used to determine the aggregated scores of the candidates.

Examples

```
>>> ratings = np.array([[.1, .2, .8, 1], [.7, .9, .8, .6], [1, .6, .1, .3]])
>>> embeddings = Embeddings(np.array([[1, 0], [1, 1], [0, 1]]), norm=True)
>>> election = RuleInstantRunoff(RuleSVDNash())(ratings, embeddings)
>>> election.ranking_
[1, 0, 2, 3]
```

Taking historical data into account

class embedded_voting.**RuleRatingsHistory**(*rule*, *embeddings_from_ratings=None*, *f=None*)

Rule that use the ratings history to improve the embeddings, in particular the quality of the mean and deviation of ratings for every voter. The original rule is then applied to the modified ratings.

Parameters

- **rule** (`Rule`) – The rule to apply to the modified ratings.
- **embeddings_from_ratings** (`EmbeddingsFromRatings`) – The function to convert ratings to embeddings.
- **f** (*callable*) – The function to apply to the ratings. It takes as input the ratings, the mean and the standard deviation of the ratings in the historic. It returns the modified ratings. By default, it is set to *f(ratings_v, history_mean, history_std) = np.sqrt(np.maximum(0, (ratings_v - history_mean) / history_std))*.

modified_ratings_

Modified ratings. For each voter, *f* is applied to her original ratings.

Type *Ratings*

6.5.2 Multi-Winner voting rules

General Class

class embedded_voting.**MultiwinnerRule**(*k=None*)

A class for multiwinner rules, in other words aggregation rules that elect a committee of candidates of size *k_*, given a ratings of voters with embeddings.

Parameters **k** (*int*) – The size of the committee.

ratings

The ratings given by voters to candidates

Type np.ndarray

embeddings

The embeddings of the voters

Type *Embeddings*

k_

The size of the committee.

Type int

set_k(*k*)

A function to update the size *k_* of the winning committee

Parameters **k** (*int*) – The new size of the committee.

Returns The object itself.

Return type *MultiwinnerRule*

winners_

A function that returns the winners, i.e. the members of the elected committee.

Returns The indexes of the elected candidates.

Return type int list

Iterative Rules

General Class

class embedded_voting.**MultiwinnerRuleIter**(*k=None*, *quota='classic'*, *take_min=False*)

A class for multi-winner rules that are adaptations of STV to the embeddings ratings model.

Parameters

- **k** (*int*) – The size of the committee.
- **quota** (*str*) – The quota used for the re-weighing step. Either `'droop'` quota *(n/(k+1) +1)* or `'classic'` quota *(n/k)*.
- **take_min** (*bool*) – If True, when the total satisfaction is less than the *quota*, we replace the quota by the total satisfaction. By default, it is set to False.

quota

The quota used for the re-weighing step. Either `'droop'` quota *(n/(k+1) +1)* or `'classic'` quota *(n/k)*.

Type str

take_min

If True, when the total satisfaction is less than the *quota*, we replace the quota by the total satisfaction. By default, it is set to False.

Type bool

weights

Current weight of every voter

Type np.ndarray

features_vectors

This function return the features vectors associated to the candidates in the winning committee.

Returns The list of the features vectors of each candidate. Each vector is of length `n_dim`.

Return type list

plot_weights(*plot_kind='3D'*, *dim=None*, *row_size=5*, *verbose=True*, *show=True*)

This function plot the evolution of the voters' weights after each step of the rule.

Parameters

- **plot_kind** (*str*) – The kind of plot we want to show. Can be `3D` or `ternary`.
- **dim** (*list*) – The 3 dimensions we are using for our plot. By default, it is set to `[0, 1, 2]`.
- **row_size** (*int*) – Number of subplots by row. By default, it is set to 5.
- **verbose** (*bool*) – If True, print the total weight divided by the number of remaining candidates at the end of each step.
- **show** (*bool*) – If True, displays the figure at the end of the function.

plot_winners(*plot_kind='3D'*, *dim=None*, *row_size=5*, *show=True*)

This function plot the winners of the election.

Parameters

- **plot_kind** (*str*) – The kind of plot we want to show. Can be `3D` or `ternary`.
- **dim** (*list*) – The 3 dimensions we are using for our plot. By default, it is set to `[0, 1, 2]`.
- **row_size** (*int*) – Number of subplots by row. By default, it is set to 5.
- **show** (*bool*) – If True, displays the figure at the end of the function.

set_quota(*quota*)

A function to update the *quota* of the rule.

Parameters **quota** (*str*) – The new quota, should be either `'droop'` or `'classic'`.

Returns The object itself.

Return type *MultiwinnerRule*

winners_

This function return the winning committee.

Returns The winning committee.

Return type int list

IterRule + SVD

class embedded_voting.**MultiwinnerRuleIterSVD**(*k=None*, *aggregation_rule=<function amax>*, *square_root=True*, *quota='classic'*, *take_min=False*)

Iterative multiwinner rule based on a SVD aggregation rule.

Parameters

- **k** (*int*) – The size of the committee.
- **aggregation_rule** (*callable*) – The aggregation rule for the singular values. By default, it is the maximum.
- **square_root** (*bool*) – If True, we take the square root of the scores instead of the scores for the `scored_embeddings()`.
- **quota** (*str*) – The quota used for the re-weighing step. Either `'droop'` quota *(n/(k+1) +1)* or `'classic'` quota *(n/k)*.
- **take_min** (*bool*) – If True, when the total satisfaction is less than the `quota`, we replace the quota by the total satisfaction. By default, it is set to False.

Examples

```
>>> np.random.seed(42)
>>> ratings_dim_candidate = np.array([[1, 0.8, 0.5, 0, 0, 0], [0, 0, 0, 0.5, 0.8,␣
↪1]])
>>> probability = [3/4, 1/4]
>>> embeddings = EmbeddingsGeneratorPolarized(100, 2, probability)(1)
>>> ratings = RatingsFromEmbeddingsCorrelated(coherence=1, ratings_dim_
↪candidate=ratings_dim_candidate)(embeddings)
>>> election = MultiwinnerRuleIterSVD(3)(ratings, embeddings)
>>> election.winners_
[0, 1, 5]
>>> _ = election.set_k(4)
>>> election.winners_
[0, 1, 5, 2]
>>> election.plot_weights(dim=[0, 0, 0], show=False)
Weight / remaining candidate :  [25.0, 24.99999999999999, 24.999999999999996, 30.
↪999999999999993]
>>> election.features_vectors
Embeddings([[1., 0.],
            [1., 0.],
            [0., 1.],
            [1., 0.]])
```

IterRule + Features

class embedded_voting.**MultiwinnerRuleIterFeatures**(*k=None*, *quota='classic'*, *take_min=False*)

Iterative multiwinner rule based on the `RuleFeatures` aggregation rule.

Parameters

- **k** (*int*) – The size of the committee.

- **quota** (*str*) – The quota used for the re-weighing step. Either 'droop' quota *(n/(k+1) +1)* or 'classic' quota *(n/k)*.
- **take_min** (*bool*) – If True, when the total satisfaction is less than the quota, we replace the quota by the total satisfaction. By default, it is set to False.

Examples

```
>>> np.random.seed(42)
>>> ratings_dim_candidate = np.array([[1, 0.8, 0.5, 0, 0, 0], [0, 0, 0, 0.5, 0.8,␣
↪1]])
>>> probability = [3/4, 1/4]
>>> embeddings = EmbeddingsGeneratorPolarized(100, 2, probability)(1)
>>> ratings = RatingsFromEmbeddingsCorrelated(coherence=1, ratings_dim_
↪candidate=ratings_dim_candidate)(embeddings)
>>> election = MultiwinnerRuleIterFeatures(3)(ratings, embeddings)
>>> election.winners_
[0, 5, 1]
>>> _ = election.set_k(4)
>>> election.winners_
[0, 5, 1, 2]
>>> election.plot_weights(dim=[0, 0, 0], show=False)
Weight / remaining candidate :  [25.0, 24.999999999999986, 27.999999999999993, 30.
↪999999999999986]
>>> election.features_vectors
Embeddings([[1., 0.],
            [0., 1.],
            [1., 0.],
            [1., 0.]])
```

static compute_features(*embeddings*, *scores*)
A function to compute features for some embeddings and scores.

Parameters

- **embeddings** (*np.ndarray*) – The embeddings of the voters. Should be of shape *n_voters*, n_dim.
- **scores** (*np.ndarray*) – The scores given by the voters to the candidates. Should be of shape *n_voters*, *n_candidates*.

Returns The features of every candidates. Of shape *n_candidates*, *n_dim*.

Return type np.ndarray

6.6 Analysis Tools

6.6.1 Simulations

Tools for benchmarking aggregation rules

class embedded_voting.experiments.aggregation.**RandomWinner**
Returns a random winner. Mimics a *Rule*. .. rubric:: Examples

```
>>> np.random.seed(42)
>>> generator = make_generator()
>>> ratings = generator(7)
>>> rule = RandomWinner()
>>> rule(ratings).winner_
4
>>> rule(ratings).winner_
3
```

class embedded_voting.experiments.aggregation.**SingleEstimator**(*i*)

Returns the best estimation of one given agent. Mimics a *Rule*. :param i: Index of the selected agents. :type i: int

Examples

```
>>> np.random.seed(42)
>>> generator = make_generator()
>>> ratings = generator(7)
>>> rule = SingleEstimator(10)
>>> ratings[10, :]
Ratings([ 1.2709017 ,  0.03209107,  1.98196138,  1.12347711, -1.55465272,
         -0.72448238,  0.63366952])
>>> rule(ratings).winner_
2
```

embedded_voting.experiments.aggregation.**evaluate**(*list_agg*, *truth*, *testing*, *training*, *pool=None*)

Run a sim. :param list_agg: Rules to test. :type list_agg: list :param truth: Ground truth of testing values (n_tries X n_candidates). :type truth: ndarray :param testing: Estimated scores (n_agents X n_tries X n_candidates). :type testing: ndarray :param training: Training scores (n_agents X training_size). :type training: ndarray :param pool: Use parallelism. :type pool: Pool, optional.

Returns Efficiency of each algorithm.

Return type ndarray

Examples

```
>>> np.random.seed(42)
>>> n_training = 10
>>> n_tries = 100
>>> n_c = 20
>>> generator = make_generator()
>>> training = generator(n_training)
>>> testing = generator(n_tries*n_c).reshape(generator.n_voters, n_tries, n_c)
>>> truth = generator.ground_truth_.reshape(n_tries, n_c)
>>> list_agg = make_aggs(order=default_order+['Rand'])
>>> with Pool() as p:
...     res = evaluate(list_agg=list_agg[:-1], truth=truth, testing=testing,
↪training=training, pool=p)
>>> ', '.join( f"{a.name}: {r:.2f}" for a, r in zip(list_agg, res) )
'MA: 0.94, PL+: 0.89, EV+: 0.95, EV: 0.94, AV: 0.90, PV: 0.86, RV: 0.85, Single:
↪0.82, PL: 0.78'
>>> res = evaluate(list_agg=list_agg, truth=truth, testing=testing,
↪training=training)
```

(continues on next page)

(continued from previous page)

```
>>> ', '.join( f"{a.name}: {r:.2f}" for a, r in zip(list_agg, res) )
'MA: 0.94, PL+: 0.89, EV+: 0.95, EV: 0.94, AV: 0.90, PV: 0.86, RV: 0.85, Single:␣
↪0.82, PL: 0.78, Rand: 0.49'
```

embedded_voting.experiments.aggregation.**f_max**(*ratings_v*, *history_mean*, *history_std*)

Parameters

- **ratings_v** (ndarray) – Score vector.
- **history_mean** (float) – Observed mean.
- **history_std** (float) – Observed standard deviation

Returns The positive part of the normalized scores.

Return type ndarray

Examples

```
>>> f_max(10, 5, 2)
2.5
>>> f_max(10, 20, 10)
0.0
```

embedded_voting.experiments.aggregation.**f_renorm**(*ratings_v*, *history_mean*, *history_std*)

Parameters

- **ratings_v** (ndarray) – Score vector.
- **history_mean** (float) – Observed mean.
- **history_std** (float) – Observed standard deviation

Returns The scores with mean and std normalized.

Return type ndarray

Examples

```
>>> f_renorm(10, 5, 2)
2.5
>>> f_renorm(10, 20, 10)
-1.0
```

embedded_voting.experiments.aggregation.**make_aggs**(*groups=None*, *order=None*, *features=None*, *group_noise=1*, *distinct_noise=0.1*)

Crafts a list of aggregator rules. :param groups: Sizes of each group (for the Model-Aware rule). :type groups: list of *int* :param order: Short names of the aggregators to return. :type order: list, optional :param features: Features correlations (for the Model-Aware rule). Default to independent groups. :type features: ndarray, optional :param group_noise: Feature noise intensity. :type group_noise: float, default=1.0 :param distinct_noise: Distinct noise intensity. :type distinct_noise: float, default=0.1

Returns Aggregators.

Return type list

Examples

```
>>> list_agg = make_aggs()
>>> [agg.name for agg in list_agg]
['MA', 'PL+', 'EV+', 'EV', 'AV', 'PV', 'RV', 'Single', 'PL']
```

embedded_voting.experiments.aggregation.**make_generator**(*groups=None*, *truth=None*, *features=None*, *feat_noise=1*, *feat_f=None*, *dist_noise=0.1*, *dist_f=None*)

Parameters

- **groups** (list of *int*) – Sizes of each group.
- **truth** (TruthGenerator, default=N(0, 1)) – Ground truth generator.
- **features** (ndarray) – Features correlations.
- **feat_noise** (float, default=1.0) – Feature noise intensity.
- **feat_f** (*method*, default to normal law) – Feature noise distribution.
- **dist_noise** (float, default=0.1) – Distinct noise intensity.
- **dist_f** (*method*, default to normal law) – Distinct noise distribution.

Returns Provides grounds truth and estimates.

Return type Generator

Examples

```
>>> np.random.seed(42)
>>> generator = make_generator()
>>> ratings = generator(2)
>>> truth = generator.ground_truth_
>>> truth[0]
0.4967141530112327
>>> ratings[:, 0]
Ratings([1.22114616, 1.09745525, 1.1986587 , 1.09806092, 1.09782972,
         1.16859892, 0.95307467, 0.97191091, 1.08817394, 1.04311958,
         1.17582742, 1.05360028, 1.00317232, 1.29096757, 1.12182506,
         1.15115551, 1.00192787, 1.08996442, 1.15549495, 1.02930333,
         2.05731381, 0.20249691, 0.23340782, 2.01575631])
>>> truth[1]
-0.13826430117118466
>>> ratings[:, 1]
Ratings([ 1.73490024,  1.51804687,  1.58119528,  1.73370001,  1.78786054,
          1.73115071,  1.70244906,  1.68390351,  1.56616168,  1.64202946,
          1.66795001,  1.81972611,  1.74837571,  1.53770987,  1.74642228,
          1.67550566,  1.64632168,  1.77518151,  1.81711384,  1.8071419 ,
         -0.23568328, -1.22689647,  0.71740695, -1.26155344])
```

6.6.2 Moving Voter Analysis

class embedded_voting.**MovingVoter**(*embeddings=None*, *moving_voter=0*)

This subclass of *Embeddings* can be used to see what happen to the scores of the different candidates when a voter moves from a group to another.

There is 4 candidates and 3 groups: Each group strongly support one of the candidate and dislike the other candidates, except the last candidate which is fine for every group.

The moving voter is a voter that do not have any preference between the candidates (she gives a score of *0.8* to every candidate, except *0.5* for the last one), but her embeddings move from one position to another.

Parameters

- **embeddings** (Embeddings) – The embeddings of the voters. If none is specified, embeddings are the identity matrix.
- **moving_voter** (*int*) – The index of the voter that is moving

rule

The rule we are using in the election.

Type *Rule*

moving_voter

The index of the voter that is moving.

Type int

ratings_

The ratings given by the voters to the candidates

Type np.ndarray

Examples

```
>>> moving_profile = MovingVoter()
>>> moving_profile(RuleSumRatings())  # DOCTEST: +ELLIPSIS
<embedded_voting.experiments.moving_voter.MovingVoter object at ...>
>>> moving_profile.moving_voter
0
>>> moving_profile.embeddings
Embeddings([[1., 0., 0.],
            [0., 0., 1.],
            [0., 1., 0.],
            [1., 0., 0.]])
>>> moving_profile.ratings_
Ratings([[0.8, 0.8, 0.8, 0.5],
         [0.1, 0.1, 1. , 0.5],
         [0.1, 1. , 0.1, 0.5],
         [1. , 0.1, 0.1, 0.5]])
```

plot_features_evolution(*show=True*)

This function plot the evolution of the features of the candidates when the moving voters' embeddings are changing. Only works for *RuleSVDMax* and *RuleFeatures*.

Parameters **show** (*bool*) – If True, displays the figure at the end of the function.

Examples

```
>>> p = MovingVoter()(RuleSVDMax())
>>> p.plot_features_evolution(show=False)
```

plot_scores_evolution(*show=True*)

This function plot the evolution of the scores of the candidates when the moving voters' embeddings are changing.

Parameters **show** (*bool*) – If True, displays the figure at the end of the function.

Examples

```
>>> p = MovingVoter()(RuleSVDNash())
>>> p.plot_scores_evolution(show=False)
```

6.6.3 Manipulation Analysis

Single-Voter Manipulation

General class

class embedded_voting.**Manipulation**(*ratings, embeddings, rule=None*)

This general class is used for the analysis of the manipulability of some *Rule* by a single voter.

For instance, what proportion of voters can change the result of the rule (to their advantage) by giving false preferences ?

Parameters

- **ratings** (*Ratings or np.ndarray*) – The ratings of voters to candidates
- **embeddings** (*Embeddings*) – The embeddings of the voters
- **rule** (*Rule*) – The aggregation rule we want to analysis.

ratings

The ratings of voters on which we do the analysis.

Type Profile

rule

The aggregation rule we want to analysis.

Type *Rule*

winner_

The index of the winner of the election without manipulation.

Type int

scores_

The scores of the candidates without manipulation.

Type float list

welfare_

The welfares of the candidates without manipulation.

Type float list

Examples

```
>>> np.random.seed(42)
>>> ratings_dim_candidate = [[1, .2, 0], [.5, .6, .9], [.1, .8, .3]]
>>> embeddings = EmbeddingsGeneratorPolarized(10, 3)(.8)
>>> ratings = RatingsFromEmbeddingsCorrelated(coherence=.8, ratings_dim_
↪candidate=ratings_dim_candidate)(embeddings)
>>> manipulation = Manipulation(ratings, embeddings, RuleSVDNash())
>>> manipulation.winner_
1
>>> manipulation.welfare_
[0.6651173304239..., 1.0, 0.0]
```

avg_welfare_

The function computes the average welfare of the winning candidate after a voter manipulation.

Returns The average welfare.

Return type float

Examples

```
>>> np.random.seed(42)
>>> ratings_dim_candidate = [[1, .2, 0], [.5, .6, .9], [.1, .8, .3]]
>>> embeddings = EmbeddingsGeneratorPolarized(10, 3)(.8)
>>> ratings = RatingsFromEmbeddingsCorrelated(coherence=.8, ratings_dim_
↪candidate=ratings_dim_candidate)(embeddings)
>>> manipulation = Manipulation(ratings, embeddings, RuleSVDNash())
>>> manipulation.avg_welfare_
0.933...
```

is_manipulable_

This function quickly computes if the ratings is manipulable or not.

Returns If True, the ratings is manipulable by a single voter.

Return type bool

Examples

```
>>> np.random.seed(42)
>>> ratings_dim_candidate = [[1, .2, 0], [.5, .6, .9], [.1, .8, .3]]
>>> embeddings = EmbeddingsGeneratorPolarized(10, 3)(.8)
>>> ratings = RatingsFromEmbeddingsCorrelated(coherence=.8, ratings_dim_
↪candidate=ratings_dim_candidate)(embeddings)
>>> manipulation = Manipulation(ratings, embeddings, RuleSVDNash())
>>> manipulation.is_manipulable_
True
```

manipulation_global_

This function applies the function *manipulation_voter()* to every voter.

Returns The list of the best candidates that can be turned into the winner for each voter.

Return type int list

Examples

```
>>> np.random.seed(42)
>>> ratings_dim_candidate = [[1, .2, 0], [.5, .6, .9], [.1, .8, .3]]
>>> embeddings = EmbeddingsGeneratorPolarized(10, 3)(.8)
>>> ratings = RatingsFromEmbeddingsCorrelated(coherence=.8, ratings_dim_
↪candidate=ratings_dim_candidate)(embeddings)
>>> manipulation = Manipulation(ratings, embeddings, RuleSVDNash())
>>> manipulation.manipulation_global_
[1, 1, 0, 1, 1, 1, 1, 1, 0, 1]
```

manipulation_map (*map_size=20*, *ratings_dim_candidate=None*, *show=True*)

A function to plot the manipulability of the ratings when the `polarisation` and the `coherence` of the `ParametricProfile` vary. The number of voters, dimensions, and candidates are those of the `profile_`.

Parameters

- **map_size** (*int*) – The number of different `coherence` and `polarisation` parameters tested. The total number of test is *map_size* ^2.
- **ratings_dim_candidate** (*np.ndarray*) – Matrix of shape `n_dim`, `n_candidates` containing the scores given by each group. More precisely, *ratings_dim_candidate[i,j]* is the score given by the group represented by the dimension *i* to the candidate *j*. If None specified, a new matrix is generated for each test.
- **show** (*bool*) – If True, display the manipulation maps at the end of the function.

Returns The manipulation maps : `manipulator` for the proportion of manipulator, `worst_welfare` and `avg_welfare` for the welfare maps.

Return type dict

Examples

```
>>> np.random.seed(42)
>>> emb = EmbeddingsGeneratorPolarized(100, 3)(0)
>>> rat = RatingsFromEmbeddingsCorrelated(n_dim=3, n_candidates=5)(emb)
>>> manipulation = Manipulation(rat, emb, rule=RuleSVDNash())
>>> maps = manipulation.manipulation_map(map_size=5, show=False)
>>> maps['manipulator']
array([[0.33, 0.  , 0.  , 0.  , 0.  ],
       [0.34, 0.22, 0.  , 0.  , 0.  ],
       [0.01, 0.  , 0.  , 0.  , 0.  ],
       [0.  , 0.19, 0.  , 0.  , 0.  ],
       [0.57, 0.22, 0.  , 0.  , 0.  ]])
```

manipulation_voter (*i*)

This function return, for the *i^th* voter, its favorite candidate that he can turn to a winner by manipulating the election.

Parameters **i** (*int*) – The index of the voter.

Returns The index of the best candidate that can be elected by manipulation.

Return type int

prop_manipulator_

This function computes the proportion of voters that can manipulate the election.

Returns The proportion of voters that can manipulate the election.

Return type float

Examples

```
>>> np.random.seed(42)
>>> ratings_dim_candidate = [[1, .2, 0], [.5, .6, .9], [.1, .8, .3]]
>>> embeddings = EmbeddingsGeneratorPolarized(10, 3)(.8)
>>> ratings = RatingsFromEmbeddingsCorrelated(coherence=.8, ratings_dim_
↪candidate=ratings_dim_candidate)(embeddings)
>>> manipulation = Manipulation(ratings, embeddings, RuleSVDNash())
>>> manipulation.prop_manipulator_
0.2
```

set_profile(*ratings*, *embeddings=None*)

This function update the ratings of voters on which we do the analysis.

Parameters

- **ratings** (Ratings *or np.ndarray*) –
- **embeddings** (Embeddings) –

Returns The object itself.

Return type *Manipulation*

worst_welfare_

This function computes the worst possible welfare achievable by single voter manipulation.

Returns The worst welfare.

Return type float

Examples

```
>>> np.random.seed(42)
>>> ratings_dim_candidate = [[1, .2, 0], [.5, .6, .9], [.1, .8, .3]]
>>> embeddings = EmbeddingsGeneratorPolarized(10, 3)(.8)
>>> ratings = RatingsFromEmbeddingsCorrelated(coherence=.8, ratings_dim_
↪candidate=ratings_dim_candidate)(embeddings)
>>> manipulation = Manipulation(ratings, embeddings, RuleSVDNash())
>>> manipulation.worst_welfare_
0.665...
```

For ordinal extensions

class embedded_voting.**ManipulationOrdinal**(*ratings*, *embeddings*, *rule_positional*, *rule=None*)

This class extends the *Manipulation* class to ordinal rule_positional (irv, borda, plurality, etc.).

Parameters

- **ratings** (*Profile*) – The ratings of voters on which we do the analysis.
- **embeddings** (Embeddings) – The embeddings of the voters.
- **rule_positional** (RulePositional) – The ordinal rule_positional used.

- **rule** (Rule) – The aggregation rule we want to analysis.

rule
The aggregation rule we want to analysis.

Type *Rule*

winner_
The index of the winner of the election without manipulation.

Type int

welfare_
The welfares of the candidates without manipulation.

Type float list

extended_rule
The rule we are analysing

Type *Rule*

rule_positional
The rule_positional used.

Type *RulePositional*

Examples

```
>>> np.random.seed(42)
>>> ratings_dim_candidate = [[1, .2, 0], [.5, .6, .9], [.1, .8, .3]]
>>> embeddings = EmbeddingsGeneratorPolarized(10, 3)(.8)
>>> ratings = RatingsFromEmbeddingsCorrelated(coherence=.8, ratings_dim_
↪candidate=ratings_dim_candidate)(embeddings)
>>> rule_positional = RulePositionalBorda(3)
>>> manipulation = ManipulationOrdinal(ratings, embeddings, rule_positional,␣
↪RuleSVDNash())
>>> manipulation.prop_manipulator_
0.0
>>> manipulation.manipulation_global_
[1, 1, 1, 1, 1, 1, 1, 1, 1, 1]
>>> manipulation.avg_welfare_
1.0
```

manipulation_voter(*i*)
This function return, for the *i^th* voter, its favorite candidate that he can turn to a winner by manipulating the election.

Parameters **i** (*int*) – The index of the voter.

Returns The index of the best candidate that can be elected by manipulation.

Return type int

Particular cases

Borda

class embedded_voting.**ManipulationOrdinalBorda**(*ratings*, *embeddings*, *rule=None*)

This class do the single voter manipulation analysis for the *RulePositionalBorda* rule_positional. It is faster than the general class class:*ManipulationOrdinal*.

Parameters

- **ratings** (Ratings *or np.ndarray*) – The ratings of voters to candidates
- **embeddings** (Embeddings) – The embeddings of the voters
- **rule** (Rule) – The aggregation rule we want to analysis.

Examples

```
>>> np.random.seed(42)
>>> ratings_dim_candidate = [[1, .2, 0], [.5, .6, .9], [.1, .8, .3]]
>>> embeddings = EmbeddingsGeneratorPolarized(10, 3)(.8)
>>> ratings = RatingsFromEmbeddingsCorrelated(coherence=.8, ratings_dim_
↪candidate=ratings_dim_candidate)(embeddings)
>>> manipulation = ManipulationOrdinalBorda(ratings, embeddings, RuleSVDNash())
>>> manipulation.prop_manipulator_
0.0
>>> manipulation.avg_welfare_
1.0
>>> manipulation.worst_welfare_
1.0
>>> manipulation.manipulation_global_
[1, 1, 1, 1, 1, 1, 1, 1, 1, 1]
```

manipulation_voter(*i*)

This function return, for the *i^th* voter, its favorite candidate that he can turn to a winner by manipulating the election.

Parameters **i** (*int*) – The index of the voter.

Returns The index of the best candidate that can be elected by manipulation.

Return type int

k-Approval

class embedded_voting.**ManipulationOrdinalKApproval**(*ratings*, *embeddings*, *k=2*, *rule=None*)

This class do the single voter manipulation analysis for the *RulePositionalKApproval* rule_positional. It is faster than the general class class:*ManipulationOrdinal*.

Parameters

- **ratings** (*Profile*) – The ratings of voters on which we do the analysis.
- **embeddings** (Embeddings) – The embeddings of the voters.
- **k** (*int*) – The k parameter for the k-approval rule.
- **rule** (Rule) – The aggregation rule we want to analysis.

Examples

```
>>> np.random.seed(42)
>>> ratings_dim_candidate = [[1, .2, 0], [.5, .6, .9], [.1, .8, .3]]
>>> embeddings = EmbeddingsGeneratorPolarized(10, 3)(.8)
>>> ratings = RatingsFromEmbeddingsCorrelated(coherence=.8, ratings_dim_
↪candidate=ratings_dim_candidate)(embeddings)
>>> manipulation = ManipulationOrdinalKApproval(ratings, embeddings, 2,␣
↪RuleSVDNash())
>>> manipulation.prop_manipulator_
0.0
>>> manipulation.avg_welfare_
1.0
>>> manipulation.worst_welfare_
1.0
>>> manipulation.manipulation_global_
[1, 1, 1, 1, 1, 1, 1, 1, 1, 1]
```

manipulation_voter (*i*)

This function return, for the *i^th* voter, its favorite candidate that he can turn to a winner by manipulating the election.

Parameters **i** (*int*) – The index of the voter.

Returns The index of the best candidate that can be elected by manipulation.

Return type int

Instant Runoff

class embedded_voting.**ManipulationOrdinalIRV** (*ratings*, *embeddings*, *rule=None*)

This class do the single voter manipulation analysis for the *RuleInstantRunoff* rule_positional. It is faster than the general class class:*ManipulationOrdinal*.

Parameters

- **ratings** (Ratings *or np.ndarray*) – The ratings of voters to candidates
- **embeddings** (Embeddings) – The embeddings of the voters
- **rule** (Rule) – The aggregation rule we want to analysis.

Examples

```
>>> np.random.seed(42)
>>> ratings_dim_candidate = [[1, .2, 0], [.5, .6, .9], [.1, .8, .3]]
>>> embeddings = EmbeddingsGeneratorPolarized(10, 3)(.8)
>>> ratings = RatingsFromEmbeddingsCorrelated(coherence=.8, ratings_dim_
↪candidate=ratings_dim_candidate)(embeddings)
>>> manipulation = ManipulationOrdinalIRV(ratings, embeddings, RuleSVDNash())
>>> manipulation.prop_manipulator_
0.4
>>> manipulation.avg_welfare_
0.4
>>> manipulation.worst_welfare_
0.0
```

(continues on next page)

(continued from previous page)

```
>>> manipulation.manipulation_global_
[2, 2, 1, 2, 1, 2, 1, 2, 1, 2]
```

manipulation_voter(*i*)
This function return, for the *i^th* voter, its favorite candidate that he can turn to a winner by manipulating the election.

Parameters **i** (*int*) – The index of the voter.

Returns The index of the best candidate that can be elected by manipulation.

Return type int

Trivial Manipulations by Coalitions

General class

class embedded_voting.**ManipulationCoalition**(*ratings*, *embeddings*, *rule=None*)
This general class is used for the analysis of the manipulability of the rule by a coalition of voter.

It only look if there is a trivial manipulation by a coalition of voter. That means, for some candidate *c* different than the current winner *w*, gather every voter who prefers *c* to *w*, and ask them to put *c* first and *w* last. If *c* is the new winner, then the ratings can be manipulated.

Parameters

- **ratings** (*Ratings or np.ndarray*) – The ratings of voters to candidates
- **embeddings** (*Embeddings*) – The embeddings of the voters
- **rule** (*Rule*) – The aggregation rule we want to analysis.

ratings
The ratings of voter on which we do the analysis.

Type Profile

rule
The aggregation rule we want to analysis.

Type *Rule*

winner_
The index of the winner of the election without manipulation.

Type int

scores_
The scores of the candidates without manipulation.

Type float list

welfare_
The welfare of the candidates without manipulation.

Type float list

Examples

```
>>> np.random.seed(42)
>>> ratings_dim_candidate = [[1, .2, 0], [.5, .6, .9], [.1, .8, .3]]
>>> embeddings = EmbeddingsGeneratorPolarized(10, 3)(.8)
>>> ratings = RatingsFromEmbeddingsCorrelated(coherence=.8, ratings_dim_
↪candidate=ratings_dim_candidate)(embeddings)
>>> manipulation = ManipulationCoalition(ratings, embeddings, RuleSVDNash())
>>> manipulation.winner_
1
>>> manipulation.welfare_
[0.6651173304239..., 1.0, 0.0]
```

is_manipulable_

A function that quickly computes if the ratings is manipulable.

Returns If True, the ratings is manipulable for some candidate.

Return type bool

Examples

```
>>> np.random.seed(42)
>>> ratings_dim_candidate = [[1, .2, 0], [.5, .6, .9], [.1, .8, .3]]
>>> embeddings = EmbeddingsGeneratorPolarized(10, 3)(.8)
>>> ratings = RatingsFromEmbeddingsCorrelated(coherence=.8, ratings_dim_
↪candidate=ratings_dim_candidate)(embeddings)
>>> manipulation = ManipulationCoalition(ratings, embeddings, RuleSVDNash())
>>> manipulation.is_manipulable_
True
```

manipulation_map(*map_size=20*, *ratings_dim_candidate=None*, *show=True*)

A function to plot the manipulability of the ratings when the `polarisation` and the `coherence` vary.

Parameters

- **map_size** (*int*) – The number of different coherence and polarisation parameters tested. The total number of test is *map_size* ^2.
- **ratings_dim_candidate** (*np.ndarray*) – Matrix of shape `n_dim`, `n_candidates` containing the scores given by each group. More precisely, *ratings_dim_candidate[i,j]* is the score given by the group represented by the dimension *i* to the candidate *j*. If not specified, a new matrix is generated for each test.
- **show** (*bool*) – If True, displays the manipulation maps at the end of the function.

Returns The manipulation maps : `manipulator` for the proportion of manipulator, `worst_welfare` and `avg_welfare` for the welfare maps.

Return type dict

Examples

```
>>> np.random.seed(42)
>>> emb = EmbeddingsGeneratorPolarized(100, 3)(0)
>>> rat = RatingsFromEmbeddingsCorrelated(n_dim=3, n_candidates=5)(emb)
>>> manipulation = ManipulationCoalition(rat, emb, RuleSVDNash())
```

(continues on next page)

(continued from previous page)

```
>>> maps = manipulation.manipulation_map(map_size=5, show=False)
>>> maps['worst_welfare']
array([[0.91880682, 1.        , 1.        , 1.        , 0.93714861],
       [0.9354928 , 0.75627811, 1.        , 1.        , 1.        ],
       [0.6484071 , 1.        , 1.        , 1.        , 1.        ],
       [0.68626628, 0.9024018 , 1.        , 1.        , 1.        ],
       [0.91491621, 0.9265847 , 1.        , 1.        , 1.        ]])
```

trivial_manipulation(*candidate*, *verbose=False*)

This function computes if a trivial manipulation is possible for the candidate passed as parameter.

Parameters

- **candidate** (*int*) – The index of the candidate for which we manipulate.
- **verbose** (*bool*) – Verbose mode. By default, is set to False.

Returns If True, the ratings is manipulable for this candidate.

Return type bool

Examples

```
>>> np.random.seed(42)
>>> ratings_dim_candidate = [[1, .2, 0], [.5, .6, .9], [.1, .8, .3]]
>>> embeddings = EmbeddingsGeneratorPolarized(10, 3)(.8)
>>> ratings = RatingsFromEmbeddingsCorrelated(coherence=.8, ratings_dim_
↪candidate=ratings_dim_candidate)(embeddings)
>>> manipulation = ManipulationCoalition(ratings, embeddings, RuleSVDNash())
>>> manipulation.trivial_manipulation(0, verbose=True)
1 voters interested to elect 0 instead of 1
Winner is 0
True
```

worst_welfare_

A function that compute the worst welfare attainable by coalition manipulation.

Returns The worst welfare.

Return type float

Examples

```
>>> np.random.seed(42)
>>> ratings_dim_candidate = [[1, .2, 0], [.5, .6, .9], [.1, .8, .3]]
>>> embeddings = EmbeddingsGeneratorPolarized(10, 3)(.8)
>>> ratings = RatingsFromEmbeddingsCorrelated(coherence=.8, ratings_dim_
↪candidate=ratings_dim_candidate)(embeddings)
>>> manipulation = ManipulationCoalition(ratings, embeddings, RuleSVDNash())
>>> manipulation.worst_welfare_
0.6651173304239...
```

For ordinal extensions

class embedded_voting.**ManipulationCoalitionOrdinal**(*ratings*, *embeddings*, *rule_positional=None*, *rule=None*)

This class extends the *ManipulationCoalition* class to ordinal rules (irv, borda, plurality, etc.), because the *ManipulationCoalition* cannot be used for ordinal preferences.

Parameters

- **ratings** (Ratings *or* *np.ndarray*) – The ratings of voters to candidates
- **embeddings** (Embeddings) – The embeddings of the voters
- **rule_positional** (RulePositional) – The ordinal rule used.
- **rule** (Rule) – The aggregation rule we want to analysis.

rule

The aggregation rule we want to analysis.

Type *Rule*

winner_

The index of the winner of the election without manipulation.

Type int

welfare_

The welfares of the candidates without manipulation.

Type float list

extended_rule

The rule we are analysing

Type *Rule*

rule_positional

The positional rule used.

Type *RulePositional*

Examples

```
>>> np.random.seed(42)
>>> ratings_dim_candidate = [[1, .2, 0], [.5, .6, .9], [.1, .8, .3]]
>>> embeddings = EmbeddingsGeneratorPolarized(10, 3)(.8)
>>> ratings = RatingsFromEmbeddingsCorrelated(coherence=0.8, ratings_dim_
↪candidate=ratings_dim_candidate)(embeddings)
>>> rule_positional = RuleInstantRunoff()
>>> manipulation = ManipulationCoalitionOrdinal(ratings, embeddings, rule_
↪positional, RuleSVDNash())
>>> manipulation.winner_
2
>>> manipulation.is_manipulable_
True
>>> manipulation.worst_welfare_
0.0
```

trivial_manipulation(*candidate*, *verbose=False*)

This function computes if a trivial manipulation is possible for the candidate passed as parameter.

Parameters

- **candidate** (*int*) – The index of the candidate for which we manipulate.
- **verbose** (*bool*) – Verbose mode. By default, is set to False.

Returns If True, the ratings is manipulable for this candidate.

Return type bool

Examples

```
>>> np.random.seed(42)
>>> ratings_dim_candidate = [[1, .2, 0], [.5, .6, .9], [.1, .8, .3]]
>>> embeddings = EmbeddingsGeneratorPolarized(10, 3)(.8)
>>> ratings = RatingsFromEmbeddingsCorrelated(coherence=.8, ratings_dim_
↪candidate=ratings_dim_candidate)(embeddings)
>>> manipulation = ManipulationCoalition(ratings, embeddings, RuleSVDNash())
>>> manipulation.trivial_manipulation(0, verbose=True)
1 voters interested to elect 0 instead of 1
Winner is 0
True
```

Particular cases

Borda

class embedded_voting.**ManipulationCoalitionOrdinalBorda**(*ratings*, *embeddings*, *rule=None*)

This class do the coalition manipulation analysis for the `RulePositionalBorda` rule_positional.

Parameters

- **ratings** (`Ratings` *or np.ndarray*) – The ratings of voters to candidates
- **embeddings** (`Embeddings`) – The embeddings of the voters
- **rule** (`Rule`) – The aggregation rule we want to analysis.

Examples

```
>>> np.random.seed(42)
>>> ratings_dim_candidate = [[1, .2, 0], [.5, .6, .9], [.1, .8, .3]]
>>> embeddings = EmbeddingsGeneratorPolarized(10, 3)(.8)
>>> ratings = RatingsFromEmbeddingsCorrelated(coherence=0.8, ratings_dim_
↪candidate=ratings_dim_candidate)(embeddings)
>>> manipulation = ManipulationCoalitionOrdinalBorda(ratings, embeddings,␣
↪RuleSVDNash())
>>> manipulation.winner_
1
>>> manipulation.is_manipulable_
False
>>> manipulation.worst_welfare_
1.0
```

k-Approval

class embedded_voting.**ManipulationCoalitionOrdinalKApproval**(*ratings*, *embeddings*, *k=2*, *rule=None*)

This class do the coalition manipulation analysis for the `RulePositionalKApproval` rule_positional.

Parameters

- **ratings** (`Ratings or np.ndarray`) – The ratings of voters to candidates
- **embeddings** (`Embeddings`) – The embeddings of the voters
- **k** (`int`) – The parameter of the k-approval rule.
- **rule** (`Rule`) – The aggregation rule we want to analysis.

Examples

```
>>> np.random.seed(42)
>>> ratings_dim_candidate = [[1, .2, 0], [.5, .6, .9], [.1, .8, .3]]
>>> embeddings = EmbeddingsGeneratorPolarized(10, 3)(.8)
>>> ratings = RatingsFromEmbeddingsCorrelated(coherence=0.8, ratings_dim_
↪candidate=ratings_dim_candidate)(embeddings)
>>> manipulation = ManipulationCoalitionOrdinalKApproval(ratings, embeddings, k=2,
↪ rule=RuleSVDNash())
>>> manipulation.winner_
1
>>> manipulation.is_manipulable_
False
>>> manipulation.worst_welfare_
1.0
```

Instant Runoff

class embedded_voting.**ManipulationCoalitionOrdinalIRV**(*ratings*, *embeddings*, *rule=None*)

This class do the coalition manipulation analysis for the `RuleInstantRunoff` rule_positional.

Parameters

- **ratings** (`Ratings or np.ndarray`) – The ratings of voters to candidates
- **embeddings** (`Embeddings`) – The embeddings of the voters
- **rule** (`Rule`) – The aggregation rule we want to analysis.

Examples

```
>>> np.random.seed(42)
>>> ratings_dim_candidate = [[1, .2, 0], [.5, .6, .9], [.1, .8, .3]]
>>> embeddings = EmbeddingsGeneratorPolarized(10, 3)(.8)
>>> ratings = RatingsFromEmbeddingsCorrelated(coherence=0.8, ratings_dim_
↪candidate=ratings_dim_candidate)(embeddings)
>>> manipulation = ManipulationCoalitionOrdinalIRV(ratings, embeddings,␣
↪RuleSVDNash())
>>> manipulation.winner_
```

(continues on next page)

(continued from previous page)

```
2
>>> manipulation.is_manipulable_
True
>>> manipulation.worst_welfare_
0.0
```

6.6.4 Algorithms Aggregation

Online Learning Analysis

class embedded_voting.**OnlineLearning**(*list_agg*, *generator=None*)

Class to compare the performance of different aggregators on a given generator.

Parameters

- **list_agg** (*list of Aggregator*) – List of aggregators to compare.
- **generator** (TruthGenerator) – Generator to use for the true ratings of the candidates

6.7 Aggregator

class embedded_voting.**Aggregator**(*rule=None*, *embeddings_from_ratings=None*, *default_train=True*, *name='aggregator'*, *default_add=True*)

A class for an election generator with memory.

You can run an election by calling it with the matrix of ratings.

Parameters

- **rule** (Rule) – The aggregation rule you want to use in your elections. Default is *RuleFastNash*
- **embeddings_from_ratings** (EmbeddingsFromRatings) – If no embeddings are specified in the call, this *EmbeddingsFromRatings* object is use to generate the embeddings from the ratings. Default: *EmbeddingsFromRatingsCorrelation(preprocess_ratings=center_and_normalize)*.
- **default_train** (*bool*) – If True, then by default, train the embeddings at each election.
- **name** (*str, optional*) – Name of the aggregator.
- **default_add** (*bool*) – If True, then by default, add the ratings to the history.

ratings_history

The history of all ratings given by the voters.

Type np.ndarray

embeddings

The current embeddings of the voters.

Type *Embeddings*

Examples

```
>>> aggregator = Aggregator()
>>> results = aggregator([[7, 5, 9, 5, 1, 8], [7, 5, 9, 5, 2, 7], [6, 4, 2, 4, 4,␣
↪6], [3, 8, 1, 3, 7, 8]])
>>> results.embeddings_
Embeddings([[ 1.        ,  0.98602958,  0.01549503, -0.43839669],
            [ 0.98602958,  1.        , -0.09219821, -0.54916602],
            [ 0.01549503, -0.09219821,  1.        ,  0.43796787],
            [-0.43839669, -0.54916602,  0.43796787,  1.        ]])
>>> results.ranking_
[5, 0, 1, 3, 4, 2]
>>> results.winner_
5
>>> results = aggregator([[2, 4, 8], [9, 2, 1], [0, 2, 5], [4, 5, 3]])
>>> results.ranking_
[2, 1, 0]
```

reset()

Reset the variables *ratings_history* and *embeddings*.

Returns The object itself.

Return type *Aggregator*

train()

Update the variable *embeddings*, based on *ratings_history*.

Returns The object itself.

Return type *Aggregator*

6.8 Utils

6.8.1 Utilities functions for plots

This file is part of Embedded Voting.

embedded_voting.utils.plots.**create_3d_plot**(*fig*, *position=None*)

Create the background for a 3D plot on the non-negative orthant.

Parameters

- **fig** – The matplotlib figure on which we are drawing.
- **position** – The position of the subplot on which we are drawing.

Returns

Return type matplotlib ax

embedded_voting.utils.plots.**create_map_plot**(*fig*, *image*, *position*, *title=''*)

Create the background for a map plot.

Parameters

- **fig** (*matplotlib figure*) – The matplotlib figure on which we are drawing.
- **image** (*np.ndarray*) – The image to plot. Should be of size *map_size*, *map_size*.
- **position** (*list*) – The position of the subplot on which we are drawing.

- **title** (*str*) – Title of the plot.

Returns

Return type matplotlib ax

embedded_voting.utils.plots.**create_ternary_plot**(*fig*, *position=None*)

Create the background for a 2D ternary plot of the non-negative orthant.

Returns

Return type matplotlib ax

6.8.2 Miscellaneous utility functions

This file is part of Embedded Voting.

embedded_voting.utils.miscellaneous.**center_and_normalize**(*x*)

Center and normalize the input vector.

Parameters **x** (*np.ndarray or list*) –

Returns *x* minus its mean. Then the result is normalized (divided by its norm).

Return type np.ndarray

Examples

```
>>> my_vector = [0, 1, 2]
>>> center_and_normalize(my_vector)
array([-0.70710678,  0.        ,  0.70710678])
```

embedded_voting.utils.miscellaneous.**clean_zeros**(*matrix*, *tol=1e-10*)

Replace in-place all small values of a matrix by 0.

Parameters

- **matrix** (ndarray) – Matrix to clean.
- **tol** (float, optional) – Threshold. All entries with absolute value lower than *tol* are put to zero.

Returns

Return type None

Examples

```
>>> import numpy as np
>>> mat = np.array([[1e-12, -.3], [.8, -1e-13]])
>>> clean_zeros(mat)
>>> mat # doctest: +NORMALIZE_WHITESPACE
array([[ 0. , -0.3],
       [ 0.8,  0. ]])
```

embedded_voting.utils.miscellaneous.**max_angular_dilatation_factor**(*vector*, *center*)

Maximum angular dilatation factor to stay in the positive orthant.

Consider *center* and *vector* two unit vectors of the positive orthant. Consider a "spherical dilatation" that moves *vector* by multiplying the angle between *center* and *vector* by a given dilatation factor. The question is: what is the maximal value of this dilatation factor so that the result still is in the positive orthant?

More formally, there exists a unit vector *unit_orthogonal* and an angle *theta in [0, pi/2]* such that *vector = cos(theta) * center + sin(theta) * unit_orthogonal*. Then there exists a maximal angle *theta_max in [0, pi/2]* such that *cos(theta_max) * center + sin(theta_max) * unit_orthogonal* is still in the positive orthant. We define the maximal angular dilatation factor as *theta_max / theta*.

Parameters

- **vector** (*np.ndarray*) – A unit vector in the positive orthant.
- **center** (*np.ndarray*) – A unit vector in the positive orthant.

Returns The maximal angular dilatation factor. If *vector* is equal to *center*, then *np.inf* is returned.

Return type float

Examples

```
>>> max_angular_dilatation_factor(
...     vector=np.array([1, 0]),
...     center=np.array([1, 1]) * np.sqrt(1/2)
... )
1.0
>>> max_angular_dilatation_factor(
...     vector=np.array([1, 1]) * np.sqrt(1/2),
...     center=np.array([1, 0])
... )
2.0
```

```
>>> my_center = np.array([1., 1., 1.]) * np.sqrt(1/3)
>>> my_unit_orthogonal = np.array([1, -1, 0]) * np.sqrt(1/2)
>>> my_theta = np.pi / 9
>>> my_vector = np.cos(my_theta) * my_center + np.sin(my_theta) * my_unit_
↪orthogonal
>>> k = max_angular_dilatation_factor(vector=my_vector, center=my_center)
>>> k
1.961576024...
>>> dilated_vector = np.cos(k * my_theta) * my_center + np.sin(k * my_theta) * my_
↪unit_orthogonal
>>> np.round(dilated_vector, 4)
array([0.8944, 0.    , 0.4472])
```

```
>>> max_angular_dilatation_factor(
...     np.array([np.sqrt(1/2), np.sqrt(1/2)]),
...     np.array([np.sqrt(1/2), np.sqrt(1/2)])
... )
inf
```

embedded_voting.utils.miscellaneous.**normalize**(*x*)

Normalize the input vector.

Parameters **x** (*np.ndarray or list*) –

Returns *x* divided by its Euclidean norm.

Return type np.ndarray

Examples

```
>>> my_vector = np.arange(3)
>>> normalize(my_vector)
array([0.        , 0.4472136 , 0.89442719])
```

```
>>> my_vector = [0, 1, 2]
>>> normalize(my_vector)
array([0.        , 0.4472136 , 0.89442719])
```

If x is null, then x is returned (only case where the result has not a norm of 1):

```
>>> my_vector = [0, 0, 0]
>>> normalize(my_vector)
array([0, 0, 0])
```

`embedded_voting.utils.miscellaneous.`**`pseudo_inverse_scalar`**`(`*x*`)`

Parameters **x** (`float`) –

Returns Inverse of x if it is not 0.

Return type `float`

Examples

```
>>> pseudo_inverse_scalar(2.0)
0.5
>>> pseudo_inverse_scalar(0)
0.0
```

`embedded_voting.utils.miscellaneous.`**`ranking_from_scores`**`(`*scores*`)`

Deduce ranking over the candidates from their scores.

Parameters **scores** (*list*) – List of floats, or list of tuple.

Returns **ranking** – The indices of the candidates, so candidate *ranking[0]* has the best score, etc. If scores are floats, higher scores are better. If scores are tuples, a lexicographic order is used. In case of tie, candidates with lower indices are favored.

Return type list

Examples

```
>>> my_scores = [4, 1, 3, 4, 0, 2, 1, 0, 1, 0]
>>> ranking_from_scores(my_scores)
[0, 3, 2, 5, 1, 6, 8, 4, 7, 9]
```

```
>>> my_scores = [(1, 0, 3), (2, 1, 5), (0, 1, 1), (2, 1, 4)]
>>> ranking_from_scores(my_scores)
[1, 3, 0, 2]
```

`embedded_voting.utils.miscellaneous.`**`singular_values_short`**`(`*matrix*`)`

Singular values of a matrix (short version).

Parameters **matrix** (*np.ndarray*) –

Returns Singular values of the matrix. In order to have a "short" version (and limit computation), we consider the square matrix of smallest dimensions among *matrix @ matrix.T* and *matrix.T @ matrix*, and then output the square roots of its eigenvalues.

Return type np.ndarray

Examples

```
>>> my_matrix = np.array([
...     [0.2 , 0.5 , 0.7 , 0.9 , 0.4 ],
...     [0.1 , 0.  , 1.  , 0.8 , 0.8 ],
...     [0.17, 0.4 , 0.66, 0.8 , 0.4 ]
... ])
>>> singular_values = singular_values_short(my_matrix)
>>> np.round(singular_values, 4)
array([2.2747, 0.5387, 0.    ])
```

`embedded_voting.utils.miscellaneous.`**`volume_parallelepiped`**(*matrix*)

Volume of the parallelepiped defined by the rows of a matrix.

Parameters **matrix** (*np.ndarray*) – The matrix.

Returns The volume of the parallelepiped defined by the rows of a matrix (in the *r*-dimensional space defined by its *r* rows). If the rank of the matrix is less than its number of rows, then the result is 0.

Return type float

Examples

```
>>> volume_parallelepiped(matrix=np.array([[10, 0, 0, 0], [0, 42, 0, 0]]))  #␣
↪doctest: +ELLIPSIS
420.0...
```

```
>>> volume_parallelepiped(matrix=np.array([[10, 0, 0, 0], [42, 0, 0, 0]]))
0.0
```

```
>>> volume_parallelepiped(matrix=np.array([[10, 0, 0, 0]]))  # doctest: +ELLIPSIS
10.0...
```

`embedded_voting.utils.miscellaneous.`**`winner_from_scores`**(*scores*)

Deduce the best of candidates from their scores.

Parameters **scores** (*list*) – List of floats, or list of tuple.

Returns **winner** – The index of the winning candidate. If scores are floats, higher scores are better. If scores are tuples, a lexicographic order is used. In case of tie, candidates with lower indices are favored.

Return type int

Examples

```
>>> my_scores = [4, 1, 3, 4, 0, 2, 1, 0, 1, 0]
>>> winner_from_scores(my_scores)
0
```

```
>>> my_scores = [(1, 0, 3), (2, 1, 5), (0, 1, 1), (2, 1, 4)]
>>> winner_from_scores(my_scores)
1
```

CHAPTER 7

Contributing

Contributions are welcome, and they are greatly appreciated! Every little bit helps, and credit will always be given.

You can contribute in many ways:

7.1 Types of Contributions

7.1.1 Report Bugs

Report bugs at https://github.com/TheoDlmz/embedded_voting/issues.

If you are reporting a bug, please include:

- Your operating system name and version.
- Any details about your local setup that might be helpful in troubleshooting.
- Detailed steps to reproduce the bug.

7.1.2 Fix Bugs

Look through the GitHub issues for bugs. Anything tagged with "bug" and "help wanted" is open to whoever wants to implement it.

7.1.3 Implement Features

Look through the GitHub issues for features. Anything tagged with "enhancement" and "help wanted" is open to whoever wants to implement it.

7.1.4 Write Documentation

Embedded Voting could always use more documentation, whether as part of the official Embedded Voting docs, in docstrings, or even on the web in blog posts, articles, and such.

7.1.5 Submit Feedback

The best way to send feedback is to file an issue at https://github.com/TheoDlmz/embedded_voting/issues.

If you are proposing a feature:

- Explain in detail how it would work.
- Keep the scope as narrow as possible, to make it easier to implement.
- Remember that this is a volunteer-driven project, and that contributions are welcome :)

7.2 Get Started!

Ready to contribute? Here's how to set up *embedded_voting* for local development.

1. Fork the *embedded_voting* repo on GitHub.
2. Clone your fork locally:

```
$ git clone git@github.com:your_name_here/embedded_voting.git
```

3. Install your local copy into a virtualenv. Assuming you have virtualenvwrapper installed, this is how you set up your fork for local development:

```
$ mkvirtualenv embedded_voting
$ cd embedded_voting/
$ python setup.py develop
```

4. Create a branch for local development:

```
$ git checkout -b name-of-your-bugfix-or-feature
```

   Now you can make your changes locally.

5. When you're done making changes, check that your changes pass flake8 and the tests, including testing other Python versions with tox:

```
$ flake8 embedded_voting tests
$ python setup.py test or pytest
$ tox
```

   To get flake8 and tox, just pip install them into your virtualenv.

6. Commit your changes and push your branch to GitHub:

```
$ git add .
$ git commit -m "Your detailed description of your changes."
$ git push origin name-of-your-bugfix-or-feature
```

7. Submit a pull request through the GitHub website.

7.3 Pull Request Guidelines

Before you submit a pull request, check that it meets these guidelines:

1. The pull request should include tests.
2. If the pull request adds functionality, the docs should be updated. Put your new functionality into a function with a docstring, and add the feature to the list in README.rst.
3. The pull request should work for Python 3.6, 3.7 and 3.8, and for PyPy. Check https://travis-ci.org/TheoDlmz/embedded_voting/pull_requests and make sure that the tests pass for all supported Python versions.

7.4 Tips

To run a subset of tests:

```
$ pytest tests.test_embedded_voting
```

7.5 Deploying

A reminder for the maintainers on how to deploy. Make sure all your changes are committed (including an entry in HISTORY.rst). Then run:

```
$ bump2version patch # possible: major / minor / patch
$ git push
$ git push --tags
```

Travis will then deploy to PyPI if tests pass.

CHAPTER 8

Credits

8.1 Development Lead

- Théo Delemazure <theo.delemazure@ens.fr>
- François Durand <fradurand@gmail.com>
- Fabien Mathieu <fabien.mathieu@normalesup.org>

8.2 Contributors

None yet. Why not be the first?

CHAPTER 9

History

9.1 0.1.7 (2023-02-14)

- New API for aggregation simulations: *evaluate*, *make_generator*, *f_max*, *f_renorm*, *SingleEstimator*, *RandomWinner*, *make_aggs*.
- Notebooks for IJCAI-23 paper submission

9.2 0.1.6 (2023-01-23)

- *Aggregators*: * Possibility to add or not the current ratings to the training set.
- *Embeddings*:
  - The parameter *norm* has no default value (instead of *True*).
  - Fix a bug: when *norm=False*, the values of the attributes *n_voter* and *n_dim* were swapped by mistake.
  - Rename method *scored* to *times_ratings_candidate*.
  - Rename method *_get_center* to *get_center*, so that it is now part of the API.
  - Rename method *normalize* to *normalized*, *recenter* to *recentered*, *dilate* to *dilated* because they return a new *Embeddings* object (not modify the object in place).
  - Fix a bug in method *get_center*.
  - Methods *get_center*, *recentered* and *dilated* now also work with non-normalized embeddings.
  - Document that *dilated* can output embeddings that are not in the positive orthant.
  - Add *dilated_new*: new dilatation method whose output is in the positive orthant.
  - Add *recentered_and_dilated*: recenter and dilate the embeddings (using *dilated_new*).
  - Add *mixed_with*: mix the given *Embeddings* object with another one.
  - Rename *plot_scores* to *plot_ratings_candidate*.

- Embeddings generators:
  - Rename *EmbeddingsGeneratorRandom* to *EmbeddingsGeneratorUniform*.
  - Add *EmbeddingsGeneratorFullyPolarized*: create embeddings that are random vectors of the canonical basis.
  - *EmbeddingsGeneratorPolarized* now relies on *EmbeddingsGeneratorUniform*, *EmbeddingsGeneratorFullyPolarized* and the method *Embeddings.mixed_with*.
  - Move *EmbeddingCorrelation* and renamed it.
  - Rewrote the *EmbeddingsFromRatingsCorrelation* and how it compute the number of singular values to take.
- Epistemic ratings generators:
  - Add *TruthGenerator*: a generator for the ground truth ("true value") of each candidate.
  - Add *TruthGeneratorUniform*: a uniform generator for the ground truth ("true value") of each candidate.
  - *RatingsGeneratorEpistemic* and its subclasses now take a *TruthGenerator* as parameter.
  - Add *RatingsGeneratorEpistemicGroups* as an intermediate class between the parent class *RatingsGeneratorEpistemic* and the child classes using groups of voters.
  - *RatingsGeneratorEpistemic* now do not take groups sizes as parameter: only *RatingsGeneratorEpistemicGroups* and its subclasses do.
  - Rename *RatingsGeneratorEpistemicGroupedMean* to *RatingsGeneratorEpistemicGroupsMean*, *RatingsGeneratorEpistemicGroupedMix* to *RatingsGeneratorEpistemicGroupsMix* *RatingsGeneratorEpistemicGroupedNoise* to *RatingsGeneratorEpistemicGroupsNoise*.
  - Remove method *RatingsGeneratorEpistemic.generate_true_values*: the same result can be obtained with *RatingsGeneratorEpistemic.truth_generator*.
  - Add *RatingsGeneratorEpistemicGroupedMixFree* and *RatingsGeneratorEpistemicGroupsMixScale*.
- Ratings generators:
  - *RatingsGenerator* and subclasses: remove **args* in call because it was not used.
  - *RatingsGeneratorUniform*: add optional parameters *minimum_rating* and *maximum_rating*.
  - Possibility to save scores in a csv file
- *RatingsFromEmbeddingsCorrelated*:
  - Move parameter *coherence* from *__call__* to *__init__*.
  - Rename parameter *scores_matrix* to *ratings_dim_candidate*.
  - Parameters *n_dim* and *n_candidates* are optional if *ratings_dim_candidate* is specified.
  - Add optional parameters *minimum_random_rating*, *maximum_random_rating* and *clip*.
  - Parameter *clip* now defaults to *False* (the former version behaved as if *clip* was always True).
- Single-winner rules:
  - Rename *ScoringRule* to *Rule*.
  - Rename all subclasses accordingly. For example, rename *FastNash* to *RuleFastNash*.
  - Rename *SumScores* to *RuleSumRatings* and *ProductScores* to *RuleProductRatings*.
  - Rename *RulePositionalExtension* to *RulePositional* and rename subclasses accordingly.
  - Rename *RuleInstantRunoffExtension* to *RuleInstantRunoff*.

  - Add *RuleApprovalSum*, *RuleApprovalProduct*, *RuleApprovalRandom*.
  - Changed the default renormalization function in *RuleFast*.
  - Change the method in *RuleMLEGaussian*.
  - Add *RuleModelAware*.
  - Add *RuleRatingsHistory*.
  - Add *RuleShiftProduct* which replace *RuleProductRatings*.
- Multiwinner rules: rename all rules with prefix *MultiwinnerRule*. For example, rename *IterFeatures* to *MultiwinnerRuleIterFeatures*.
- Manipulation:
  - Rename *SingleVoterManipulation* to *Manipulation* and rename subclasses accordingly.
  - Rename *SingleVoterManipulationExtension* to *ManipulationOrdinal* and rename subclasses accordingly.
  - Rename *ManipulationCoalitionExtension* to *ManipulationCoalitionOrdinal* and rename subclasses accordingly.
- Rename *AggregatorSum* to *AggregatorSumRatings* and *AggregatorProduct* to *AggregatorProductRatings*.
- Add *max_angular_dilatation_factor*: maximum angular dilatation factor to stay in the positive orthant.
- Rename *create_3D_plot* to *create_3d_plot*.
- Moved function to the utils module.
- Reorganize the file structure of the project.

9.3 0.1.5 (2022-01-04)

- Aggregator functions.
- Online learning.
- Refactoring Truth epistemic generators.
- Rule taking history into account.

9.4 0.1.4 (2021-12-06)

- New version with new structure for Ratings and Embeddings

9.5 0.1.3 (2021-10-27)

- New version with new internal structure for the library

9.6 0.1.2 (2021-07-05)

- New version with handy way to use the library for algorithm aggregation and epistemic social choice

9.7 0.1.1 (2021-04-02)

- Minor bugs.

9.8 0.1.0 (2021-03-31)

- End of the internship, first release on PyPI.

CHAPTER 10

Indices and tables

- genindex
- modindex
- search

Python Module Index

e

Index

Symbols

A

B

C

D

E

N

O

P

Q

R

S

T

V

W