Votelib

Jan Šimbera

Jun 11, 2022

CONTENTS:

Votelib is a package to evaluate results of elections under various systems. It aims to provide reliable implementations of many voting systems so that they may be evaluated as they are used in real-world conditions and compared. The primary focus is on political decision making, but the library is not limited to evaluating political decisions only.

More can be learned by looking at the examples of usage below or at the API reference.

CHAPTER

ONE

VOTELIB API REFERENCE

Votelib - a library for evaluating election results.

Votelib objects provide the means to evaluate elections under many known election systems, be it simple or complicated, obscure or ubiquitous.

An election system usually specifies the following:

- Who can stand for the election (what is a valid nomination). This can be checked using the nominators from the `candidate` module.
- What forms of votes are valid. This can be checked using the vote validators from the `vote` module for different vote types from ranked to range votes.
- How to determine who is elected. This is the task of the `evaluate` subpackage and its many modules, which contains evaluators for most of the world's known systems. Many of those considered standalone evaluation methods are actually reducible variants that can use an another evaluator; for these cases, the `convert` module provides converters for votes and results to create more complex systems.

The evaluator can be combined with the validator and nominator by the machinery in the `evaluate` subpackage. The *`VotingSystem`* object can then wrap it into a formalized and named election system.

class votelib.**VotingSystem**(*name*, *evaluator*)

A named voting system. Wraps an election evaluator.

Parameters

- **name** (`str`) – Name of the system; usually mainly includes the body or position to be elected.
- **evaluator** (*`Evaluator`*) – Evaluator representing the system. Can be combined with the validator and nominator by machinery in the `evaluate` subpackage.

1.1 Evaluators API

Evaluate the results of the elections.

There are two basic election types - selections and distributions. In selections, each candidate (usually a physical person) is either elected or not elected. In distributions, some candidates (usually parties) are allocated a positive number of seats (seats are distributied among the parties), while other parties get none and do not appear in the result.

Selection evaluators return a list of candidates. If some of them are tied, a *`core.Tie`* object will appear in the list. The tie object will contain all candidates tied for the seats and will be repeated the number of times equal to the number of tied seats.

Distribution evaluators return a dictionary mapping candidates to the number of seats (or other tokens they should be allocated). In case of a tie, the *`core.Tie`* object will appear as one of the keys, with the number of tied seats as its associated value.

There are some other election concepts, for example *asset voting*, where the candidates bargain with each other with the votes they have received; these understandably cannot be implemented within the current scope of Votelib.[1]

None of the evaluators validate vote correctness; use the tools in the `vote` module for that, potentially wrapped in `convert.InvalidVoteEliminator`.

1.1.1 Plurality evaluator

class `votelib.evaluate.core.`**`Plurality`**

Plurality voting / maximum score evaluator. Elects a list of candidates.

This encompasses the following voting systems:

- *First-past-the-post* (simple plurality, FPTP) for a single seat and one vote per person.
- *Single non-transferable vote* (SNTV) for multiple seats and one vote per voter.
- *Plurality-at-large* (multiple non-transferable vote) for multiple seats and number of votes equal the number of seats.
- *Limited voting* for multiple seats and number of votes per voter fixed and less than the number of seats.
- *Cumulative voting*, a very similar variant where the number of available votes per voter is not tied to the number of candidates or seats.
- *Approval voting* (AV) for number of available votes per voter equal to the number of candidates (but not the more advanced variants of approval voting such as proportional approval voting - see the `approval` for that). A multiwinner variant of approval voting is sometimes called block approval voting and exhibits very different properties. Use *`votelib.convert.ApprovalToSimpleVotes`* to convert approval votes to simple votes accepted by this evaluator.
- *Satisfaction approval voting* (SAV), with one arbitrarily splittable vote per voter. Use *`votelib.convert.ApprovalToSimpleVotes`* to convert approval votes to simple votes accepted by this evaluator.
- *Score voting* after the votes are aggregated by a score vote aggregator such as *`votelib.convert.ScoreToSimpleVotes`* (`votelib.evaluate.cardinal.ScoreVoting` can be used instead).

These voting systems all use this evaluator together with some vote conversion or validation criteria.

`evaluate`(*votes*, *n_seats=1*)

Select candidates by plurality voting.

In case of a tie, returns *`Tie`* objects at the end of the list of elected candidates. There will be as many ties as the number of tied seats; each tie will group the tied candidates.

Parameters

- **`votes`** (`Dict[Union[str, CandidateObject], Number]`) – Simple votes (mapping the candidates to a quantity, the more the better).
- **`n_seats`** (`int`) – Number of candidates to select.

Return type

`List[Union[str, CandidateObject]]`

Returns

A list of elected candidates, sorted in descending order by the input votes.

[1] "Asset voting", Range voting.org. https://rangevoting.org/Asset.html

1.1.2 Proportional distribution evaluators

Distribution evaluators that are usually called proportional.

This contains the most common distribution evaluators used in party-list elections that at least aim to be proportional (although some parameter setups might make the result very nonproportional, such as the Imperiali divisor) and also some similar, simpler distribution evaluators that aim for strict proportionality while relaxing some of the constraints of a distribution evaluator.

class `votelib.evaluate.proportional.`**`BiproportionalEvaluator`**(*divisor_function='d_hondt'*, *apportioner=None*, *signpost_q=None*)

Allocate seats biproportionally to parties and constituencies.

Biproportional apportionment is a method to provide proportional election results in two dimensions - constituencies and parties (candidates). It works by computing the proportional result using a highest averages method along one dimension (here: parties) and then iteratively updating it until proportionality is also reached for constituencies, in a process that somehow resembles iterative proportional fitting (IPF) but for integer values.

There are two main biproportional apportionment algorithms - alternate scaling (AS), which is known to be faster in initial stages but also to stall in some corner cases, and tie-and-transfer (TT), which is slower but robust. Here, tie-and-transfer (TT) is implemented as described in[2].

So far, the only studied variant of biproportional apportionment uses highest averages methods, but usually in an alternative specification by rounding rules (also called signpost sequences). These rounding rules have only been found for D'Hondt and Sainte-Laguë divisors; for other divisors, the implementation is missing yet.

Parameters

- **divisor_function** (`Union[str, Callable[[int], Number]]`) – A callable producing the divisor from the number of seats awarded to the contestant so far, in the same form accepted by *`HighestAverages`*.
- **apportioner** (`Union[`*`Distributor`*`, Dict[`*`Constituency`*`, int], int, None]`) – An optional distribution evaluator to allocate seats to constituencies according to the total numbers of votes cast in each. Can also be an integer stating the uniformly valid number of seats for each constituency, or a dictionary giving the numbers per constituency. If None, the number of seats must be specified to *`evaluate()`*, or the highest averages evaluator defined by *divisor_function* is used.
- **signpost_q** (`Union[int, Fraction, None]`) – The signpost function subtraction constant, which defines the rounding. For D'Hondt and Sainte-Laguë divisors, this is automatically determined; for other divisors, it must be specified manually.

`evaluate`(*votes*, *n_seats*)

Distribute seats biproportionally.

Parameters

- **votes** (`Dict[`*`Constituency`*`, Dict[Union[str, CandidateObject], int]]`) – Simple votes per constituency to be evaluated.
- **n_seats** (`Union[int, Dict[`*`Constituency`*`, int]]`) – Number of seats to be filled, either in total or by constituency.

Return type

`Dict[`*`Constituency`*`, Dict[Union[str, CandidateObject], int]]`

[2] "Chapter 15. Double-Proportional Divisor Methods: Technicalities", F. Pukelsheim. In: *Proportional Representation*, DOI `10.1007/978-3-319-64707-4_15`.

class votelib.evaluate.proportional.**HighestAverages**(*divisor_function='d_hondt'*)

Distribute seats proportionally by ordering divided vote counts.

Divides the vote count for each party by an increasing sequence of divisors (usually small integers), sorts these quotients and awards a seat for each of the first n_seats quotients.

This includes some popular proportional party-list systems like D'Hondt or Sainte-Laguë/Webster. The result is usually quite close to proportionality and avoids the Alabama paradox of largest remainder systems. However, it usually favors either large or smaller parties, depending on the choice of the divisor function.

Parameters

divisor_function (Union[str, Callable[[int], Number]]) – A callable producing the divisor from the number of seats awarded to the contestant so far. For example, the D'Hondt divisor (which uses the natural numbers sequence) would always return the number of currently held seats raised by one. The common divisor functions can be referenced by string name from the *votelib.component.divisor* module.

evaluate(*votes*, *n_seats*, *prev_gains={}*, *max_seats={}*)

Distribute seats proportionally by highest averages.

Parameters

- **votes** (Dict[Union[str, CandidateObject], int]) – Simple votes to be evaluated.
- **n_seats** (int) – Number of seats to be filled.
- **prev_gains** (Dict[Union[str, CandidateObject], int]) – Seats gained by the candidate/party in previous election rounds, to determine the starting divisor.
- **max_seats** (Dict[Union[str, CandidateObject], int]) – Maximum number of seats that the given candidate/party can obtain in total (including previous gains).

Return type

Dict[Union[str, CandidateObject], int]

class votelib.evaluate.proportional.**LargestRemainder**(*quota_function*, ***kwargs*)

Distribute seats proportionally, rounding by largest remainder.

Each contestant is awarded the number of seats according to the number of times their votes fill the provided quota, just like *QuotaDistributor*. In addition to that, the parties that have the largest remainder of votes after the quotas are subtracted get an extra seat until the number of seats is exhausted.

This includes some popular proportional party-list systems like Hare, Droop or Hagenbach-Bischoff. The result is usually very close to proportionality (closer than in highest averages systems) but might suffer from an Alabama paradox where adding proportionally distributed votes might cause one of the contestants to lose seats.

Parameters

quota_function (Union[str, Callable[[int, int], Number]]) – A callable producing the quota threshold from the total number of votes and number of seats. The common quota functions can be referenced by string name from the *votelib.component.quota* module.

All additional keyword arguments have the same meaning as in *QuotaDistributor*.

evaluate(*votes*, *n_seats*, *prev_gains={}*, *max_seats={}*)

Distribute seats proportionally, rounding by largest remainder.

Parameters

- **votes** (Dict[Union[str, CandidateObject], int]) – Simple votes to be evaluated.
- **n_seats** (int) – Number of seats to be filled.

- **prev_gains** (`Dict[Union[str, CandidateObject], int]`) – Seats gained by the candidate/party in previous election rounds to be subtracted from the proportional result awarded here.
- **max_seats** (`Dict[Union[str, CandidateObject], int]`) – Maximum number of seats that the given candidate/party can obtain in total (including previous gains).

Return type
`Dict[Union[str, CandidateObject], int]`

`class votelib.evaluate.proportional.`**`PureProportionality`**

Distribute seats among candidates strictly proportionally (no rounding).

This evaluator is mostly auxiliary since it gives fractional seat counts.

`evaluate`(*votes*, *n_seats*, *prev_gains={}*, *max_seats={}*)

Distribute seats exactly proportionally, giving fractional seats.

Parameters

- **votes** (`Dict[Union[str, CandidateObject], int]`) – Simple votes to be evaluated.
- **n_seats** (`int`) – Number of seats to be filled.
- **prev_gains** (`Dict[Union[str, CandidateObject], int]`) – Seats gained by the candidate/party in previous election rounds to be subtracted from the proportional result awarded here.
- **max_seats** (`Dict[Union[str, CandidateObject], int]`) – Maximum number of seats that the given candidate/party can obtain in total (including previous gains).

Return type
`Dict[Union[str, CandidateObject], Fraction]`

`class votelib.evaluate.proportional.`**`QuotaDistributor`**(*quota_function='droop'*, *accept_equal=True*, *on_overaward='error'*)

Distribute seats proportionally, according to multiples of quota filled.

Each contestant is awarded the number of seats according to the number of times their votes fill the provided quota. (This essentially means the numbers of votes are divided by the quota and rounded down to get the numbers of seats.)

This is usually not a self-standing evaluator because it (except for very rare cases) does not award the full number of seats; it is, however, used in initial stages of some proportional systems, such as the Czech Chamber of Deputies election.

Parameters

- **quota_function** (`Union[str, Callable[[int, int], Number]]`) – A callable producing the quota threshold from the total number of votes and number of seats. The common quota functions can be referenced by string name from the `votelib.component.quota` module.
- **accept_equal** (`bool`) – Whether to consider the candidate elected when their votes exactly equal the quota.
- **on_overaward** (`str`) – Some ill-conditioned but still used quotas, such as Imperiali, may distribute more seats than the allocated total since they are low. In that case, do the following:
  - `'ignore'` - return the result regardless,
  - `'error'` - raise a VotingSystemError,

– `'subtract'` - subtract seats from the candidates with the smallest remainder after the quota division (those that exceed the quota by a lowest margin) until the total seat count is met.

evaluate(*votes*, *n_seats*, *prev_gains={}*, *max_seats={}*)

Distribute seats proportionally by multiples of quota filled.

Parameters

- **votes** (`Dict[Union[str, CandidateObject], int]`) – Simple votes to be evaluated.
- **n_seats** (`int`) – Number of seats to be filled.
- **prev_gains** (`Dict[Union[str, CandidateObject], int]`) – Seats gained by the candidate/party in previous election rounds to be subtracted from the proportional result awarded here.
- **max_seats** (`Dict[Union[str, CandidateObject], int]`) – Maximum number of seats that the given candidate/party can obtain in total (including previous gains).

Return type

`Dict[Union[str, CandidateObject], int]`

class votelib.evaluate.proportional.**VotesPerSeat**(*votes_per_seat*, *rounding='ROUND_DOWN'*, *accept_equal=True*)

Award seats for each N votes cast for each candidate.

This is an old and simple system that was used e.g. in pre-war Germany[3]. It divides the number of votes by a pre-specified constant and rounds to give the appropriate number of seats. It is also used as an auxiliary evaluator in some other systems with fixed quota.

Parameters

- **votes_per_seat** (`int`) – Number of votes required for the candidate to obtain a single seat.
- **rounding** (`str`) – A rounding mode from the *decimal* Python library. The default option is to round down, which is the most frequent case, but this evaluator allows to specify a different method as well.
- **accept_equal** (`bool`) – If False, whenever the number of votes is exactly divisible by *votes_per_seat*, award one less seat.

1.1.3 Condorcet selection evaluators

Condorcet selection evaluators.

These evaluators work by examining pairwise orderings between candidates (how many voters prefer one candidate to another), which is also the form of votes they take in; if you have ranked vote input, use `votelib.convert.RankedToCondorcetVotes` to convert it first and account for possible shared rankings and unranked candidates.

All of the methods in this module reliably select a Condorcet winner when there is one in the input.

These evaluators are considered more advanced in terms of satisfied criteria than transferable vote systems but still cannot defy some impossibility theorems such as Arrow's or Gibbard's.

These evaluators only take few parameters; therefore, a dictionary of their instances with different setups is provided in the `EVALUATORS` module variable.

[3] "Reichstag (Weimarer Republik): Wahlsystem", Wikipedia. https://de.wikipedia.org/wiki/Reichstag_(Weimarer_Republik)#Wahlsystem

class votelib.evaluate.condorcet.**CondorcetWinner**

Condorcet winner selector.

Selects a candidate that pairwise beats all other candidates, if there is one, or an empty list otherwise.

evaluate(*votes*)

Select the Condorcet winner.

Parameters

votes (Dict[Tuple[Union[str, CandidateObject], Union[str, CandidateObject]], int]) – Condorcet votes (counts of candidate pairs as they appear in the voter rankings); use *votelib.convert.RankedToCondorcetVotes* to produce them from ranked votes.

Return type

List[Union[str, CandidateObject]]

class votelib.evaluate.condorcet.**Copeland**(*second_order=True*)

Copeland (count of pairwise wins) Condorcet selection evaluator.

Calculates the pairwise wins, constructs the Copeland score by taking `number_of_pairwise_wins - number_of_pairwise_losses` for each candidate, and uses this score to rank candidates. This often produces ties, which can be used by second order tiebreaking - by preferring the candidates who have pairwise beaten the candidates with the highest total Copeland score.

Parameters

second_order (bool) – Whether to use second-order Copeland tiebreaking.

evaluate(*votes*, *n_seats=1*)

Select candidates using the Copeland method.

Parameters

- **votes** (Dict[Tuple[Union[str, CandidateObject], Union[str, CandidateObject]], int]) – Condorcet votes (counts of candidate pairs as they appear in the voter rankings); use *votelib.convert.RankedToCondorcetVotes* to produce them from ranked votes.
- **n_seats** (int) – Number of candidates to select.

Return type

List[Union[str, CandidateObject]]

class votelib.evaluate.condorcet.**KemenyYoung**

Kemeny-Young Condorcet selection evaluator.

Kemeny-Young orders the candidates based on their pairwise comparison by constructing an objective function that measures the quality of any ordering, evaluating it for all permutations of the candidate set, and selecting the ordering with the maximum score.

The objective function is given as the number of satisfied pairwise orderings of candidates as given by the voters.

WARNING: Due to the enumeration of all candidate set permutations, this method is highly computationally expensive (`O(n!)` in the number of candidates) and infeasible on common machines for more than a handful of candidates.

evaluate(*votes*, *n_seats=1*)

Select candidates by the Kemeny-Young Condorcet method.

Parameters

- **votes** (Dict[Tuple[Union[str, CandidateObject], Union[str, CandidateObject]], int]) – Condorcet votes (counts of candidate pairs as they appear in the voter rankings); use *votelib.convert.RankedToCondorcetVotes* to produce them from ranked votes.

- **n_seats** (int) – Number of candidates to select.

Return type
List[Union[str, CandidateObject]]

static score(*variant*, *votes*)

Compute the Kemeny-Young ordering score (objective function value).

Parameters

- **variant** (Collection[Union[str, CandidateObject]]) – The ordering of candidates to evaluate.
- **votes** (Dict[Tuple[Union[str, CandidateObject], Union[str, CandidateObject]], int]) – Condorcet votes (counts of candidate pairs as they appear in the voter rankings); use *votelib.convert.RankedToCondorcetVotes* to produce them from ranked votes.

Return type
int

class votelib.evaluate.condorcet.**MinimaxCondorcet**(*pairwin_scoring='winning_votes'*)

Minimax Condorcet selection evaluator.

Also known as successive reversal or Simpson-Kramer method. Selects as the winner the candidate whose greatest pairwise defeat is smaller than the greatest pairwise defeat of any other candidate.

The magnitude of the pairwise defeat can be measured in different ways according to the pairwise win scorer provided.

Parameters
pairwin_scoring (Union[str, Callable]) – A pairwise win scorer callable. Most common variants are found in the *votelib.component.pairwin_scorer* module and can be referred to by their names.

evaluate(*votes*, *n_seats=1*)

Select candidates by the Minimax Condorcet method.

Parameters

- **votes** (Dict[Tuple[Union[str, CandidateObject], Union[str, CandidateObject]], int]) – Condorcet votes (counts of candidate pairs as they appear in the voter rankings); use *votelib.convert.RankedToCondorcetVotes* to produce them from ranked votes.
- **n_seats** (int) – Number of candidates to select.

Return type
List[Union[str, CandidateObject]]

class votelib.evaluate.condorcet.**RankedPairs**(*pairwin_scoring='winning_votes'*)

Tideman's ranked pairs Condorcet selection evaluator.

Ranks pairwise wins by their magnitude and sequentially locks pairs of who beats whom in descending order into a ranking, discarding pairs that would contradict previously established rankings (i.e. create a cycle).

The magnitude of the pairwise win can be measured in different ways according to the pairwise win scorer provided.

Parameters

pairwin_scoring (Union[str, Callable]) – A pairwise win scorer callable. Most common variants are found in the *votelib.component.pairwin_scorer* module and can be referred to by their names.

evaluate(*votes*, *n_seats=1*)

Select candidates by the ranked pairs method.

Parameters

- **votes** (Dict[Tuple[Union[str, CandidateObject], Union[str, CandidateObject]], int]) – Condorcet votes (counts of candidate pairs as they appear in the voter rankings); use *votelib.convert.RankedToCondorcetVotes* to produce them from ranked votes.
- **n_seats** (int) – Number of candidates to select.

Return type

List[Union[str, CandidateObject]]

class votelib.evaluate.condorcet.**Schulze**

Schulze (beatpath) Condorcet selection evaluator.

Also called Schwartz Sequential dropping or path voting. Finds paths between pairs of candidates in which each candidate pairwise beats the next and then selects the candidates with strongest such paths.

evaluate(*votes*, *n_seats=1*)

Select candidates using the Schulze method.

Parameters

- **votes** (Dict[Tuple[Union[str, CandidateObject], Union[str, CandidateObject]], int]) – Condorcet votes (counts of candidate pairs as they appear in the voter rankings); use *votelib.convert.RankedToCondorcetVotes* to produce them from ranked votes.
- **n_seats** (int) – Number of candidates to select.

Return type

List[Union[str, CandidateObject]]

class votelib.evaluate.condorcet.**SchwartzSet**

Schwartz set selector.

The Schwartz set is the smallest possible non-empty set whose candidates are pairwise unbeaten by all other candidates.

evaluate(*votes*)

Select the Schwartz set.

Parameters

votes (Dict[Tuple[Union[str, CandidateObject], Union[str, CandidateObject]], int]) – Condorcet votes (counts of candidate pairs as they appear in the voter rankings); use *votelib.convert.RankedToCondorcetVotes* to produce them from ranked votes.

Return type

List[Union[str, CandidateObject]]

class votelib.evaluate.condorcet.**SmithSet**

Smith set selector.

The Smith set is the smallest possible non-empty set whose candidates beat all other candidates in pairwise preference comparisons.

`evaluate`(*votes*)

Select the Smith set.

Parameters

`votes` (`Dict[Tuple[Union[str, CandidateObject], Union[str, CandidateObject]], int]`) – Condorcet votes (counts of candidate pairs as they appear in the voter rankings); use *`votelib.convert.RankedToCondorcetVotes`* to produce them from ranked votes.

Return type

`List[Union[str, CandidateObject]]`

`votelib.evaluate.condorcet.`**`beat_counts`**(*votes*)

Count the number of candidates a given candidate beats pairwise.

Parameters

`votes` (`Dict[Tuple[Union[str, CandidateObject], Union[str, CandidateObject]], int]`) – Condorcet votes (counts of candidate pairs as they appear in the voter rankings); use *`votelib.convert.RankedToCondorcetVotes`* to produce them from ranked votes.

Return type

`Dict[Union[str, CandidateObject], int]`

`votelib.evaluate.condorcet.`**`pairwise_wins`**(*votes*, *include_ties=False*)

Select pairs of candidates where the first is preferred to the second.

Parameters

- `votes` (`Dict[Tuple[Union[str, CandidateObject], Union[str, CandidateObject]], int]`) – Condorcet votes (counts of candidate pairs as they appear in the voter rankings); use *`votelib.convert.RankedToCondorcetVotes`* to produce them from ranked votes.
- `include_ties` (`bool`) – Whether to include pairs of candidates that are tied. Such a pair will be included in both directions.

Return type

`List[Tuple[Union[str, CandidateObject], Union[str, CandidateObject]]]`

Returns

Ordered pairs from the input that are generally preferred to the opposite ranking (i.e. listed in this order by more voters).

1.1.4 Sequential (vote addition) selection evaluators

Evaluators that operate sequentially on ranked votes.

This hosts mainly the transferable vote evaluator (`TransferableVoteSelector`) but also its less known relative, *`PreferenceAddition`*.

`class votelib.evaluate.sequential.`**`Baldwin`**(*converter=None*)

Baldwin's sequential Borda elimination method.

`class votelib.evaluate.sequential.`**`Benham`**

Benham sequential Condorcet selector.[5]

[5] Green-Armytage, James. "Four Condorcet-Hare Hybrid Methods for Single-Winner Elections", Voting Matters. http://www.votingmatters.org.uk/ISSUE29/I29P1.pdf

Selects a Condorcet winner. If one does not exist, eliminates one candidate using transferable vote (instant-runoff) and re-runs.

`class votelib.evaluate.sequential.`**`PreferenceAddition`**`(`*coefficients=[1]*, *split_equal_rankings=True*`)`

Evaluates ranked votes by stepwise addition of lower preferences.

Each candidate starts with their first-preference votes and lower-preference votes are added to them until a sufficient amount of candidates achieve a majority. This includes the following voting systems:

- *Bucklin voting* where the lower preferences are added without change.
- *Oklahoma system* where the lower preferences are added divided by the order of preference, thus yielding the row of coefficients 1, 1/2, 1/3, 1/4… (Use `Fraction` to maintain exact values.)

Parameters

- **`coefficients`** (`Union[List[Number], Callable[[int], Number]]`) – Coefficients to multiply the lower preferences with before they are added to the vote totals (the default adds them simply, which creates Bucklin voting).
- **`split_equal_rankings`** (`bool`) – Whether to split votes having multiple alternatives at the same rank, or to add the whole amount to each of these alternatives.

`evaluate``(`*votes*, *n_seats=1*`)`

Select candidates by sequential preference addition.

Parameters

- **`votes`** (`Dict[Tuple[Union[str, CandidateObject, FrozenSet[Union[str, CandidateObject]]], ...], int]`) – Ranked votes.
- **`n_seats`** (`int`) – Number of candidates to be elected.

Return type

`List[Union[str, CandidateObject]]`

`class votelib.evaluate.sequential.`**`TidemanAlternative`**`(`*set_selector=<votelib.evaluate.condorcet.SmithSet object>*`)`

Tideman alternative selector (Alternative Smith/Schwartz).[4]

Uses a Smith or Schwartz set selector; if the set has multiple members, eliminates one candidate by transferable vote (instant-runoff) and reruns. For election of multiple candidates, it runs in multiple tiers where previous winners are eliminated from all following tiers.[Page 12, 5]

Parameters

`set_selector` (`SeatlessSelector`) – The Smith/Schwartz set selector. Any seatless selector that accepts the Condorcet (pairwise preference) vote format.

`class votelib.evaluate.sequential.`**`TransferableVoteDistributor`**`(`*transferer=<votelib.component.transfer.Gregory object>*, *retainer=None*, *eliminate_step=-1*, *quota_function='droop'*, *accept_quota_equal=True*, *mandatory_quota=False*`)`

Select candidates by eliminating and transfering votes among them.

This is the evaluator for transferable vote selection (since the evaluator does not concern itself with restrictions on allowed votes, it covers not only single transferable vote systems - STV -, but also potential multiple transferable vote systems). Its single-winner variant is also called instant-runoff voting (IRV).

[4] "Tideman alternative method", Wikipedia. https://en.wikipedia.org/wiki/Tideman_alternative_method

First, this evaluator looks at first-preference votes. If any of the candidates has at least a specified quota of votes, they are awarded the number of seats according to how many times the quota fits into their number of votes. If they hit their maximum achievable number of seats (if there is one), their votes over the quota are redistributed (reallocated) to other candidates according to the next stated preference.

If no candidate has the quota, the candidate with the fewest currently allocated votes is eliminated and their votes transferred according to their next stated preference. Votes that have no further stated preferences are called exhausted and are removed from consideration.

The current version does not use elimination breakpoints.

Parameters

- **`transferer`** (`Union[str, VoteTransferer]`) – An instance determining how much votes for eliminated candidates to transfer and to whom. It must provide the `votelib.component.transfer.VoteTransferer` interface (experimental, stability not guaranteed). The basic vote transfer variants are implemented in the *`votelib.component.transfer`* module and can be referred to by their names as strings.
- **`retainer`** (`Optional[`*`Selector`*`]`) – A selector determining which candidates to retain when elimination is to be performed (it may accept a number of seats, which will correspond to the number of candidates to retain). If not given, the candidates with the lowest amounts of currently allocated votes will be eliminated one by one.
- **`eliminate_step`** (`Optional[int]`) – Determines how many candidates to eliminate whenever elimination is to be performed. If a negative integer, determines how many candidates to eliminate at each step (the default eliminates one candidate at a time). If a positive integer, determines how many candidates to retain after elimination - this essentially allows only a single elimination and might cause an infinite loop if not used properly.
- **`quota_function`** (`Union[str, Callable[[int, int], Number], None]`) – A callable producing the quota threshold from the total number of votes and number of seats. The common quota functions can be referenced by string name from the *`votelib.component.quota`* module. If None, no election by quota takes place - the candidates are just eliminated until the desired number remains. (Not specifying the quota only works when the maximum numbers of seats per candidate are specified.)
- **`accept_quota_equal`** (`bool`) – Whether to use non-strict comparison for the number of votes against the quota.
- **`mandatory_quota`** (`bool`) – If True, the candidates must reach the quota to be elected, i.e. they cannot be merely the last to be eliminated. If False, as soon as the number of unallocated seats equals the number of remaining candidates, these are all elected regardless of reaching the quota.

`evaluate`(*votes*, *n_seats*, *prev_gains={}*, *max_seats={}*)

Allocate seats to candidates by transferable vote.

Parameters

- **`votes`** (`Dict[Tuple[Union[str, CandidateObject, FrozenSet[Union[str, CandidateObject]]], ...], int]`) – Ranked votes. Equal rankings are allowed.
- **`n_seats`** (`int`) – Number of seats to allocate to candidates.
- **`prev_gains`** (`Dict[Union[str, CandidateObject], int]`) – Seats gained by the candidate/party in previous election rounds to be subtracted from the result awarded here.
- **`max_seats`** (`Dict[Union[str, CandidateObject], int]`) – Maximum number of seats that the given candidate/party can obtain in total (including previous gains).

Return type
`Dict[Union[str, CandidateObject], int]`

next_count(*allocation*, *n_seats*, *total_n_votes*, *prev_gains={}*, *max_seats={}*)
Advance the transferable voting process by one iteration (count).

Parameters

- **allocation** (`Dict[Union[str, CandidateObject, None], Dict[Tuple[Union[str, CandidateObject, FrozenSet[Union[str, CandidateObject]]], ...], Number]]`) – Current allocation of ranked votes to candidates still contesting the remaining seats. Eliminated candidates are not present in the dictionary keys. Exhausted votes are keyed by None.
- **n_seats** (`int`) – Total number of seats to award.
- **total_n_votes** (`int`) – Total number of votes. Used to compute the election quota.
- **prev_gains** (`Dict[Union[str, CandidateObject], int]`) – Numbers of seats the candidates gained in the previous counts of the election.
- **max_seats** (`Dict[Union[str, CandidateObject], int]`) – Maximum number of seats that the given candidate/party can obtain in total (including previous gains).

Return type
`Tuple[Dict[Union[str, CandidateObject, None], Dict[Tuple[Union[str, CandidateObject, FrozenSet[Union[str, CandidateObject]]], ...], Number]], Dict[Union[str, CandidateObject], int]]`

Returns
A 2-tuple containing the new allocation of votes and a mapping of candidates to newly assigned seats (might be empty if no seats were awarded on this count).

nth_count(*votes*, *n_seats=1*, *count_number=1*, *prev_gains={}*, *max_seats={}*)
Get the intermediate counting state at a given iteration (count).

Parameters

- **votes** (`Dict[Tuple[Union[str, CandidateObject, FrozenSet[Union[str, CandidateObject]]], ...], Number]`) – Ranked votes. Equal rankings are allowed.
- **n_seats** (`int`) – Number of candidates to select.
- **count_number** (`int`) – 1-indexed count number.

Return type
`Tuple[Dict[Union[str, CandidateObject, None], Dict[Tuple[Union[str, CandidateObject, FrozenSet[Union[str, CandidateObject]]], ...], Number]], Dict[Union[str, CandidateObject], int]]`

Returns
A 2-tuple containing the allocation of votes after the given count and a list of elected candidates so far (might be empty).

`votelib.evaluate.sequential.`**initial_allocation**(*votes*, *transferer=<votelib.component.transfer.Gregory object>*)

Allocate votes by first preference.

Parameters

- **votes** (`Dict[Tuple[Union[str, CandidateObject, FrozenSet[Union[str, CandidateObject]]], ...], Number]`) – Ranked votes.

- **transferer** (`VoteTransferer`) – A `votelib.component.transfer.VoteTransferer` to transfer votes from shared first ranks among their candidates.

Return type
`Dict[Union[str, CandidateObject, None], Dict[Tuple[Union[str, CandidateObject, FrozenSet[Union[str, CandidateObject]]], ...], Number]]`

Returns
The votes dictionary separated into subdictionaries keyed by candidate to whom the votes are allocated. A candidate with no first preference votes will be assigned to an empty dictionary.

1.1.5 Approval voting selection evaluators

Advanced approval voting methods.

This module contains approval voting evaluators that cannot be reduced to a plurality evaluation by aggregating the scores. Use *`votelib.convert.ApprovalToSimpleVotes`* in conjunction with `votelib.evaluate.Plurality` to evaluate simple approval voting (AV) or satisfaction approval voting (SAV).

`class votelib.evaluate.approval.`**`ProportionalApproval`**

Proportional Approval Voting (PAV) evaluator.[6]

This method uses approval votes (voters select one or more permissible candidates) and evaluates the satisfaction of voters with each of the combinations of elected candidates. The satisfaction for each voter is given as the sum of reciprocals from 1 to N, where N is the number of elected candidates that the voter approved of.

WARNING: Due to the enumeration of all candidate set combinations, this method is highly computationally expensive (`O(n!)` in the number of candidates) and infeasible on common machines for more than a handful of candidates.

Tie breaking not implemented - the method itself does not provide a way to do it, a dedicated tie breaker will probably be necessary.

`evaluate`(*votes*, *n_seats*)

Select candidates by proportional approval.

Parameters

- **votes** (`Dict[FrozenSet[Union[str, CandidateObject]], int]`) – Approval votes.
- **n_seats** (`int`) – Number of candidates to be elected.

Return type
`List[Union[str, CandidateObject]]`

Returns
Selected candidates in decreasing order measured by drop in satisfaction when the given candidate is excluded from the selected set.

`class votelib.evaluate.approval.`**`QuotaSelector`**(*quota_function='droop'*, *accept_equal=True*, *on_more_over_quota='error'*)

Quota threshold (plurality) selector.

Elects candidates with more (or also equally many, depending on *accept_equal*) votes than the specified quota. This often gives fewer candidates than the number of seats, and thus usually needs to be accompanied by an another evaluation step. In very rare cases, it might select more candidates than the number of seats.

This is a component in the following systems:

[6] "Proportional approval voting", Wikipedia. https://en.wikipedia.org/wiki/Proportional_approval_voting

- *Two-round runoff* (usually with the Droop quota and a single seat) where it gives the first-round winner if they have a majority of votes, and no one otherwise.

It can also serve as a threshold evaluator (eliminator) in proportional systems that restrict the first party seat from being a remainder seat, or a kickstart for Huntington-Hill related methods that are not defined for zero-seat parties.

Parameters

- `quota_function` (`Union[str, Callable[[int, int], Number]]`) – A callable producing the quota threshold from the total number of votes and number of seats.
- `accept_equal` (`bool`) – Whether to elect candidates that only just reach the quota threshold (this is known to produce some instabilities).
- `on_more_over_quota` (`str`) – How to handle the case when more candidates fulfill the quota that there is seats:
  - `'error'`: raise a *`votelib.evaluate.core.VotingSystemError`*,
  - `'select'`: select the candidates with the most votes (possibly producing ties when they are equal).

`class votelib.evaluate.approval.`**`SequentialProportionalApproval`**

Sequential Proportional Approval Voting (SPAV) evaluator.[7]

This method uses approval votes (voters select one or more permissible candidates) but evaluates them iteratively, unlike proportional approval voting. In each iteration, the best candidate is selected and all ballots that approve of them are reduced in value to `1/n`, where `n` is the number of the candidates on that ballot already elected plus one (the value of those votes thus decreases to a half after one of the marked candidates is elected, to a third if a second one is elected, and so on).

Tie breaking not yet implemented.

`evaluate`(*votes*, *n_seats*)

Select candidates by sequential proportional approval.

Parameters

- `votes` (`Dict[FrozenSet[Union[str, CandidateObject]], int]`) – Approval votes.
- `n_seats` (`int`) – Number of candidates to be elected.

Return type

`List[Union[str, CandidateObject]]`

Returns

Selected candidates ordered as they were selected in the successive iterations.

1.1.6 Auxiliary evaluators

Open list evaluators

Open party list selection evaluators.

Many party-list proportional systems feature open lists - that is, voters get to cast preferential votes for the candidates on that list, and the order of those candidates might then be altered by these preferences. These evaluators serve to select the candidates from the party list that actually get one of the seats allocated to it.

[7] "Sequential proportional approval voting", Wikipedia. https://en.wikipedia.org/wiki/Sequential_proportional_approval_voting

The evaluators in this module accept an extra argument to their `evaluate()` methods, namely the original ordering of the candidates on the party list. Many methods only alter this ordering (allow *jumps*) if the lower candidates have significantly more votes than those above them; almost all of them rely on this ordering to break ties.

For the most common open list evaluation case, *`ThresholdOpenList`* (allowing jumps when the candidate gets at least a given fraction of the total votes for the list or at least a quota of the votes based on the number of seats) is provided. *`ListOrderTieBreaker`* provides a wrapper for any selection evaluator and uses the list ordering to break ties.

`class votelib.evaluate.openlist.`**`ListOrderTieBreaker`**`(`*evaluator*`)`

A wrapper for any selector for open-list candidate selection.

The candidates that the provided evaluator returns with the given votes are provided as the result of the open list. If there is a tie, it is broken by the ordering on the party list.

Parameters
`evaluator` (*`Selector`*) – Any selection evaluator.

`evaluate``(`*votes*, *n_seats*, *candidate_list*`)`

Select candidates from an open party list.

Parameters

- **`votes`** (`Dict[Any, Number]`) – Preferential votes for candidates on the list. Must be of the type that is accepted by the underlying evaluator.
- **`n_seats`** (`int`) – Number of seats to be awarded to the list - the number of candidates to elect.
- **`candidate_list`** (`List[Union[str, CandidateObject]]`) – The original (party-determined) list ordering.

Return type
`List[Union[str, CandidateObject]]`

`class votelib.evaluate.openlist.`**`ThresholdOpenList`**`(`*jump_fraction=None*, *quota_function=None*, *quota_fraction=1*, *take_higher=False*, *accept_equal=False*, *list_precedence=False*`)`

A threshold-based open list evaluator.

Allows candidates that got at least a given number of votes for the list as preferential votes to jump over all other candidates. These jumping candidates are selected first, ordered by their number of votes; the rest of the seats is filled by other candidates on the list in its original order.

The number of votes required for the jump is determined relative to the total number of votes received by the list (`jump_fraction`) or to a quota computed from this total and the number of seats to award to the list (`quota_function` and `quota_fraction`).

Parameters

- **`jump_fraction`** (`Optional[Fraction]`) – The fraction of total votes for the list that the candidate must obtain to jump ahead in the rankings.
- **`quota_function`** (`Union[str, Callable[[int, int], Number], None]`) – A callable producing the quota threshold (number of votes required to jump) from the total number of votes and number of seats. The common quota functions can be referenced by string name from the *`votelib.component.quota`* module. The Hare quota (`'hare'`) is used most frequently.
- **`quota_fraction`** (`Fraction`) – A number to multiply the calculated quota with, to allow e.g. all candidates that got at least half of the quota to jump.

- **take_higher** (bool) – If both the jump fraction and quota are specified, this determines whether to take the higher of the thresholds (amounting to an AND function) or the lower one (amounting to an OR function, the default).
- **accept_equal** (bool) – Whether to use non-strict comparison for the number of votes against the quota or jump fraction.
- **list_precedence** (bool) – Whether the original list ordering takes precedence when more candidates are allowed to jump than the number of seats. If True, the candidates lowest on the list are eliminated; if False, the candidates with the least votes are eliminated.

evaluate(*votes*, *n_seats*, *candidate_list*)

Select candidates from an open party list.

Parameters

- **votes** (Dict[Union[str, CandidateObject], Number]) – Preferential votes (simple) for candidates on the list.
- **n_seats** (int) – Number of seats to be awarded to the list - the number of candidates to elect.
- **candidate_list** (List[Union[str, CandidateObject]]) – The original (party-determined) list ordering.

Return type

List[Union[str, CandidateObject]]

Electoral threshold evaluators

Electoral threshold evaluators and other seatless selectors.

These evaluators mainly serve as auxiliary components (usually preconditions) in proportional or similar voting systems. They are seatless selectors, which means they return a list of elected candidates without being given the number of them to select; that number is determined by other means.

class votelib.evaluate.threshold.**AbsoluteThreshold**(*threshold*, *accept_equal=True*)

Absolute threshold seatless selector.

Selects all candidates with more (or equally many) votes than the specified absolute number. Does not accept a number of seats as an argument to the `evaluate()` method.

Parameters

- **threshold** (Number) – The absolute threshold as a number of votes.
- **accept_equal** (bool) – Whether to elect candidates that only just reach the threshold.

evaluate(*votes*)

Select candidates by a given absolute threshold of votes.

Parameters

votes (Dict[Union[str, CandidateObject], Number]) – Simple votes.

Return type

List[Union[str, CandidateObject]]

class votelib.evaluate.threshold.**AlternativeThresholds**(*partials*)

An OR function for threshold evaluators.

Wraps multiple seatless selectors and selects a candidate that is selected by any single one of them.

Parameters
`partials` (List[*SeatlessSelector*]) – The selectors to wrap.

`evaluate`(*votes*, *prev_gains={}*)

Select candidates by dispatching to partial selectors and ORing.

Parameters
`votes` (Dict[Union[str, CandidateObject], Number]) – Simple votes.

Return type
List[Union[str, CandidateObject]]

Returns
A list of candidates that were selected by any single one of the internal partial selectors. They are ordered by mean rank in those partial selections.

`class votelib.evaluate.threshold.CoalitionMemberBracketer`(*evaluators*, *default*)

Dispatch to different seatless selectors for coalitions.

In many proportional systems the threshold to exclude smaller parties is raised for coalitions depending on the number of their members, to prevent the fragmentation of the resulting elected body.

Parameters

- `evaluators` (Dict[int, *SeatlessSelector*]) – A dictionary mapping the number of coalition members to the corresponding seatless selector such as *RelativeThreshold*. Atomic parties (non-coalitions) will be dispatched to 1.
- `default` (*SeatlessSelector*) – A default seatless selector for coalitions that do not have the corresponding count in evaluators. This is useful for clauses like “four and more party coalitions”.

`evaluate`(*votes*)

Select parties by dispatching to partial selectors.

Parameters
`votes` (Dict[*ElectionParty*, Number]) – Simple votes for parties. The keys in the dictionary must provide an `is_coalition` property and if its value is truthy, a `get_n_coalition_members()` method that returns the number of coalition members as an integer.

Return type
List[*ElectionParty*]

`class votelib.evaluate.threshold.PreviousGainThreshold`(*selector*)

A threshold on gained seats in previous election rounds.

In some multi-round systems (especially mixed-member proportional ones), an alternative to clearing a national-level vote fraction threshold is to gain a specific minimum number of seats in the first round (e.g. in Germany a party may qualify for list seats if they gain three or more district seats).

This evaluator passes the `prev_gains` argument of ordinary distribution evaluators to the `votes` argument of a given seatless selector, allowing one to specify a threshold based on previously gained seats.

Parameters
`selector` (*SeatlessSelector*) – The selector to evaluate the threshold on previous gains; *AbsoluteThreshold* would be the most typical choice.

`evaluate`(*votes*, *prev_gains*)

Select candidates by previous gain according to the inner selector.

Parameters

- **votes** (Dict[Any, int]) – Will be disregarded.
- **prev_gains** (Dict[Union[str, CandidateObject], int]) – Seats gained by the candidate/party in previous election rounds. Will be passed as votes to the inner selector.

Return type
List[Union[str, CandidateObject]]

`class votelib.evaluate.threshold.`**`PropertyBracketer`**`(`*property*, *evaluators*, *default=None*`)`

Dispatch to different seatless selectors for some types of parties.

In many proportional systems the threshold to exclude smaller parties is lowered or nonexistent for parties with a special designation, such as parties of regional minorities. This wrapper class allows to provide special seatless selectors for such special cases.

Parameters

- **property** (str) – The property (attribute) of the candidate to get as the key to distinguish which evaluator to use. To get the property, if the candidate object has a *properties* attribute, it is used, otherwise, `getattr()` is used on the candidate object directly.
- **evaluators** (Dict[Any, Optional[*SeatlessSelector*]]) – A dictionary mapping the values of the property to the corresponding seatless selector such as *RelativeThreshold*.
- **default** (Optional[*SeatlessSelector*]) – A default seatless selector for candidates that do not define the specified property or whose property value is outside the set of keys of the evaluators dictionary.

`evaluate``(`*votes*`)`

Select candidates by dispatching to partial selectors.

Parameters
votes (Dict[Union[str, CandidateObject], Number]) – Simple votes. The keys in the dictionary (i.e. candidate objects) should provide the property name specified at the setup of this elector; if they do not, the default evaluator is used for them.

Return type
List[Union[str, CandidateObject]]

`class votelib.evaluate.threshold.`**`RelativeThreshold`**`(`*threshold*, *accept_equal=True*`)`

Relative threshold seatless selector.

Selects all candidates with more (or equally many) votes than the specified fraction of total votes. Does not accept a number of seats as an argument to the `evaluate()` method.

This is a common component in proportional systems that excludes very small parties to increase stability of the resulting elected body.

Parameters

- **threshold** (Number) – The relative threshold as a fraction of total votes.
- **accept_equal** (bool) – Whether to elect candidates that only just reach the threshold.

`evaluate``(`*votes*`)`

Select candidates by a given threshold of fraction of total votes.

Parameters
votes (Dict[Union[str, CandidateObject], Number]) – Simple votes.

Return type
List[Union[str, CandidateObject]]

Other auxiliary evaluators

Evaluators for special partial purposes, especially tiebreaking.

These evaluators should not be used as the main component of an election system (except for obscure ones). Many of them choose the winners randomly, so they are useful for tiebreaking, but that is about it.

You can make the random evaluators outputs stable if you give them a seed for the random generator, but be careful with that in a real-world setting.

`class votelib.evaluate.auxiliary.`**`CandidateNumberRanker`**

Select first N candidates with lowest candidate number.

This is useful for tiebreaking with an externally determined sort order.

`evaluate``(`*votes*, *n_seats=1*`)`

Select the first candidates that appear in the votes dictionary.

Parameters

- **`votes`** (`Dict[Union[str, CandidateObject], Any]`) – Simple votes. The quantities of votes are disregarded.
- **`n_seats`** (`int`) – Number of candidates to be selected.

Return type

`List[Union[str, CandidateObject]]`

`class votelib.evaluate.auxiliary.`**`InputOrderSelector`**

Select first N candidates as they appear in the vote counts.

This is useful for tiebreaking with an externally determined sort order, e.g. by ballot numbers or pre-generated random numbers. It takes advantage of dictionaries in Python 3.7+ maintaining insertion order.

`evaluate``(`*votes*, *n_seats=1*`)`

Select the first candidates that appear in the votes dictionary.

Parameters

- **`votes`** (`Dict[Union[str, CandidateObject], Any]`) – Simple votes. The quantities of votes are disregarded.
- **`n_seats`** (`int`) – Number of candidates to be selected.

Return type

`List[Union[str, CandidateObject]]`

`class votelib.evaluate.auxiliary.`**`RFC3797Selector`**`(`*sources*`)`

Select candidates randomly by the algorithm from RFC 3797.

This is a well-defined random selection method using external sources of randomness, that are to be provided as numbers or lists thereof. Once the sources of randomness are fixed in the constructor, the selection is deterministic with regard to the input order of candidates (candidate names or votes are disregarded).

Parameters

`sources` (`List[Union[Number, Collection[Number]]]`) – Sources of randomness (seeds). For detailed recommendations on where to take them from, see the RFC. The list should contain an item per source. The list items may contain a number, a string (which will be stripped of accents and anything besides ASCII alphanumeric characters and uppercased) or a list of numbers. Strings and floats are STRONGLY not recommended.

`evaluate`(*votes*, *n_seats=1*)

Select the candidates by the algorithm from RFC 3797.

Parameters

- **votes** (`Dict[Union[str, CandidateObject], Any]`) – Simple votes. The quantities of votes are disregarded.
- **n_seats** (`int`) – Number of candidates to be selected.

Return type

`List[Union[str, CandidateObject]]`

`classmethod source_bytestring`(*sources*)

Create the base randomness string from given randomness sources.

Follows the procedure as given by Section 4 of the RFC.

Return type

`bytes`

`class votelib.evaluate.auxiliary.RandomUnrankedBallotSelector`(*seed=None*)

Select candidates by drawing random simple ballots from the tally.

Useful for tiebreaking. Can also be used to evaluate *random approval voting* if the approval votes are converted to simple first.

Parameters

seed (`Optional[int]`) – Seed for the random generator that performs the sampling.

`evaluate`(*votes*, *n_seats=1*)

Select candidates by drawing random ballots.

Parameters

- **votes** (`Dict[Union[str, CandidateObject], Number]`) – Simple votes.
- **n_seats** (`int`) – Number of candidates (ballots) to be selected.

Return type

`List[Union[str, CandidateObject]]`

`class votelib.evaluate.auxiliary.Sortitor`(*seed=None*)

Perform sortition (random sampling) among the candidates.

This selects the candidates purely randomly, with equal probabilities for each of them. Useful for tiebreaking and some obscure protocols such as Venetian doge election.

Parameters

seed (`Optional[int]`) – Seed for the random generator that performs the sampling.

`evaluate`(*votes*, *n_seats=1*)

Select candidates randomly.

Parameters

- **votes** (`Dict[Union[str, CandidateObject], Any]`) – Simple votes. The quantities of votes are disregarded.
- **n_seats** (`int`) – Number of candidates to be selected.

Return type

`List[Union[str, CandidateObject]]`

1.1.7 Base classes and composition objects

General voting system evaluator machinery.

Common evaluation objects

class votelib.evaluate.core.**Tie**

Candidates tied for a seat.

This object, a subclass of `frozenset`, is produced by evaluators that do not resolve all evaluation ties - for example, a plurality evaluator when two or more candidates have an equal number of votes and only some of them fit into the number of seats to fill. It can either be presented in the result (such as in the case when the given voting system has no tiebreaking mechanism) or caught by a tiebreaking evaluator - use *TieBreaking* for this.

This object also provides some special static and class methods to handle ties in some basic ways.

static any(*result*)

Return True if there is any tie in the list, False otherwise.

Return type
`bool`

classmethod break_by_list(*elected*, *breaker*)

Break ties in the elected list according to ordering in breaker.

Return type
`List[Union[str, CandidateObject]]`

classmethod reconcile(*elected*)

Reconcile multiply tied candidates.

A placeholder implementation to account for the situation where e.g. partial evaluations gave ties but their multiple occurrence in the overall result means the ties can be unambiguously resolved.

Parameters
elected (`List[Union[str, CandidateObject, Tie]]`) – Selection result - a list of candidates or ties.

Return type
`List[Union[str, CandidateObject]]`

static tie_rankings(*rankings*)

Form a single ranking out of a list of rankings.

Produces ties where the rankings do not agree.

Return type
`List[Union[str, CandidateObject, Tie]]`

class votelib.evaluate.core.**VotingSystemError**

A voting system with a valid setup ended up in an unresolvable state.

Abstract base classes for evaluators

`class votelib.evaluate.core.Evaluator`

Evaluate votes for candidates and allocate seats to them.

A root abstract base class for all evaluators.

`abstract evaluate(votes, *args, **kwargs)`

Evaluate votes for candidates and allocate seats to them.

Return type
`Union[List[Union[str, CandidateObject]], Dict[Union[str, CandidateObject], int]]`

`class votelib.evaluate.core.Selector`

Elect a given number of candidates.

Requires a number of seats to determine the number of candidates to elect.

`abstract evaluate(votes, n_seats, *args, **kwargs)`

Elect n_seats candidates as a list.

Parameters

- **votes** – Votes of any type.
- **n_seats** – Number of candidates to elect.

Return type
`List[Union[str, CandidateObject]]`

Returns
A list of candidates elected, ordered by magnitude of victory (winner first).

`class votelib.evaluate.core.SeatlessSelector`

Elect some candidates.

Does not allow to pass a number of seats to determine the number of candidates to elect; it arises naturally from the votes and other selector settings.

`abstract evaluate(votes, *args, **kwargs)`

Elect a list of candidates without a guaranteed count.

Parameters
votes – Votes of any type.

Return type
`List[Union[str, CandidateObject]]`

Returns
A list of candidates elected, ordered by magnitude of victory (winner first).

`class votelib.evaluate.core.Distributor`

Allocate seats to candidates based on collective preference.

`abstract evaluate(votes, n_seats, prev_gains={}, max_seats={})`

Allocate n_seats to candidates as a dictionary.

Parameters

- **votes** (`Dict[Any, int]`) – Votes of any type.
- **n_seats** (`int`) – Number of seats to allocate to candidates.

- **prev_gains** (Dict[Union[str, CandidateObject], int]) – Seats gained by the candidate/party in previous election rounds.
- **max_seats** (Dict[Union[str, CandidateObject], int]) – Maximum number of seats that the given candidate/party can obtain in total (including previous gains).

Return type
Dict[Union[str, CandidateObject], int]

Returns
Numbers of seats allocated to respective candidates. Candidates with no allocated seats do not appear in the dictionary. The ordering of the dictionary is unspecified.

class votelib.evaluate.core.**SeatCountCalculator**

Composite evaluators

class votelib.evaluate.core.**FixedSeatCount**(*evaluator*, *n_seats*)

An evaluator wrapper that provides a fixed seat count.

Useful when the seat count for a given system is predefined and constant. Then, you do not need to specify it each time you call the evaluator, just provide the votes.

Parameters

- **evaluator** (*Evaluator*) – The evaluator to wrap. Must accept number of seats into its evaluate method.
- **n_seats** (int) – The fixed number of seats to provide to the evaluator at each call.

evaluate(*votes*, ***kwargs*)

Run the inner evaluator with the predefined number of seats.

All other keyword parameters are passed through to the wrapped evaluator.

Parameters
votes (Dict[Any, int]) – Votes of the type accepted by the wrapped evaluator.

Return type
Union[List[Union[str, CandidateObject]], Dict[Union[str, CandidateObject], int]]

class votelib.evaluate.core.**TieBreaking**(*main*, *tiebreaker*, *subsetter=<votelib.vote.SimpleSubsetter object>*)

Break ties from the main evaluation through a dedicated tiebreaker.

Runs the main evaluator on the input, and if ties are present in its output, runs them through a separately specified selector with the original votes subsetted to just those concerning the tied candidates.

Tiebreakers can be nested if there are more tiebreaking methods with different priority. To do so, wrap the core evaluator with the highest priority tiebreaker first, and then supply the result as the main evaluator to another tiebreaker with lower priority.

Parameters

- **main** (*Evaluator*) – The main evaluator; might be a selector or a distributor.
- **tiebreaker** (*Selector*) – A selector to evaluate ties.
- **vote_subsetter** – A subsetter to subset a vote to just concern the tied candidates.

`evaluate`(*votes*, **args*, ***kwargs*)

Evaluate the election, breaking ties if they arise.

Any arguments besides the votes dictionary are passed unchanged to the main evaluator.

Parameters

`votes` – Votes for the election; will be used to feed the main evaluator (and the tiebreaker, after subsetting).

`class votelib.evaluate.core.`**`Conditioned`**(*eliminator*, *evaluator*, *subsetter=None*, *depth=1*)

An evaluator whose votes are pre-selected to exclude some variants.

Before passing the votes to the main evaluator, an eliminator is evaluated first, and only the candidates returned by it are allowed to proceed to the main evaluation. This is useful for implementing vote thresholds in proportional systems or for two-stage Condorcet-like systems, such as Smith//Score.

Parameters

- **`evaluator`** (*`Evaluator`*) – The main evaluator to produce the results.
- **`eliminator`** (*`SeatlessSelector`*) – A selector to determine which variants proceed to the main evaluator - only those that are returned by its evaluate method do so. It can accept previous gain counts but should not need the number of seats (it should be independent of it).
- **`subsetter`** (`Optional[VoteSubsetter]`) – A subsetter to subset a vote to just concern the candidates returned by the eliminator.

`evaluate`(*votes*, *n_seats=None*, *prev_gains={}*, ***kwargs*)

Evaluate the main evaluator for variants that passed the eliminator.

All other keyword arguments are passed through to the main evaluator.

Parameters

- **`votes`** (`Dict[Any, int]`) – Votes to be passed to the main evaluator and eliminator.
- **`n_seats`** (`Optional[int]`) – Number of seats to allocate by the main evaluator.
- **`prev_gains`** (`Dict[Union[str, CandidateObject], int]`) – Numbers of seats previously gained by the candidates (e.g. in previous scrutinia). Will be passed to both the main evaluator and eliminator, if they accept them (as determined by the *accepts_seats* function).

Return type

`Union[List[Union[str, CandidateObject]], Dict[Union[str, CandidateObject], int]]`

`class votelib.evaluate.core.`**`PreConverted`**(*converter*, *evaluator*)

An evaluator whose votes are first run through a converter.

Useful when the evaluator accepts a different type of votes than the actual ballots, where the converter adapts them - e.g. when the votes are by constituency but the evaluation is nationwide, or when ranked votes are given but a part of the system runs on simple votes.

Parameters

- **`evaluator`** – An evaluator to run.
- **`converter`** – A converter to apply on the votes before passing them to the evaluator.

`evaluate`(*votes*, **args*, ***kwargs*)

Convert the votes by and evaluate them through the evaluator.

All other arguments are passed through to the evaluator.

Parameters

- **votes** – Votes to be passed to the converter.
- **n_seats** – Number of seats to allocate by the evaluator.

class votelib.evaluate.core.**PostConverted**(*evaluator*, *converter*)

An evaluator whose results are run through a converter.

Useful when the evaluator is a part of a larger system and its output needs to be adapted to it, e.g. in multi-member proportional systems where first votes are received by candidates in a selection evaluation but the second votes are evaluated by distribution to parties.

Parameters

- **evaluator** – An evaluator to run.
- **converter** – A converter to apply on the results of the evaluator.

evaluate(*votes*, **args*, ***kwargs*)

Run the evaluator and return its result, converted.

All other arguments are passed through to the evaluator.

Parameters

- **votes** – Votes for the evaluator.
- **n_seats** – Number of seats to allocate by the evaluator.

class votelib.evaluate.core.**ByConstituency**(*evaluator*, *apportioner=None*, *preselector=None*, *subsetter=<votelib.vote.SimpleSubsetter object>*)

Perform constituency-wise evaluation of given system.

Evaluates and returns the results of the given system separately per each constituency it is given votes for. If desired, apportions the seats to constituencies according to the total numbers of votes cast in each.

The selector and vote subsetter perform what the `Conditioned` evaluator wrapper would do; however, they are only used after apportionment is performed, because that is often defined before candidates are excluded. If this is not desired, define the selector and vote subsetter on a `Conditioned` wrapper, with the selector wrapped in an additional `PreConverted` with `votelib.convert.VoteTotals`, and leave the selector undefined here.

Parameters

- **evaluator** (`Evaluator`) – Evaluator to use on the constituency level.
- **apportioner** (`Union[Distributor, Dict[Constituency, int], int, None]`) – An optional distribution evaluator to allocate seats to constituencies according to the total numbers of votes cast in each. Can also be an integer stating the uniformly valid number of seats for each constituency, or a dictionary giving the numbers per constituency. If None, the number of seats must be specified to `evaluate()`.
- **preselector** (`Optional[Selector]`) – An optional selection evaluator to select candidates eligible for constituency-wise evaluation. Votes for candidates that do not pass its selection will be removed for all constituencies before evaluation. If not given, no preselection will be applied.
- **subsetter** (`VoteSubsetter`) – A subsetter to subset a vote to just concern the candidates returned by the selector. The default option needs to be modified if the votes are more deeply nested.

evaluate(*votes*, *n_seats=None*, *prev_gains={}*, *max_seats={}*)

Return the election results evaluated by constituency.

Parameters

- **`votes`** (`Dict[`*`Constituency`*`, Dict[Any, int]]`) – Votes in the format accepted by the inner evaluator.
- **`n_seats`** (`Union[int, Dict[`*`Constituency`*`, int], None]`) – Number of seats to be allocated:
  - If `None`, the apportioner must be a distributor that does not accept seat count, and will be called with the total numbers of votes cast in each constituency to provide a seat count for each constituency.
  - If an integer and the apportioner is None, regarded as the uniform number of seats for each constituency.
  - If an integer and the apportioner is a distributor, the apportioner will be called with this integer as the total number of seats to distribute it to the constituencies.
  - If a dictionary, it will be regarded as the mapping of constituencies to their numbers of seats.
- **`prev_gains`** (`Dict[`*`Constituency`*`, Dict[Union[str, CandidateObject], int]]`) – Seats gained by the candidate/party in previous election rounds in each constituency, to inform the underlying evaluator.
- **`max_seats`** (`Dict[`*`Constituency`*`, Dict[Union[str, CandidateObject], int]]`) – Maximum number of seats that the given candidate/party can obtain in total per constituency (including previous gains).

Return type
`Union[Dict[`*`Constituency`*`, Dict[Union[str, CandidateObject], int]], Dict[`*`Constituency`*`, List[Union[str, CandidateObject]]]]`

Returns
Results of the evaluation by constituency.

`class votelib.evaluate.core.`**`PreApportioned`**`(`*evaluator*, *apportioner*`)`

Apportion seats to constituencies before the evaluation.

In some systems, apportionment is not fixed but is only done at the time of the election based on the number of votes cast in the constituencies. This component also allows to perform this.

Parameters

- **`evaluator`** (*`Distributor`*) – The per-constituency election evaluator.
- **`apportioner`** (`Union[`*`Distributor`*`, Dict[`*`Constituency`*`, int], int]`) – How to apportion the seats to constituencies:
  - If an integer, each constituency gets this many seats.
  - If a dictionary mapping constituencies to integers, each constituency gets the number of seats specified.
  - If a distribution evaluator, it is called on vote totals in each constituency to produce the number of seats. In this variant, n_seats must be passed to `evaluate()`.

`class votelib.evaluate.core.`**`RemovedApportionment`**`(`*evaluator*`)`

Disregard previously computed apportionment for this evaluator.

In some systems, only a single (usually the “proportionality leveling”) component does not respect an otherwise valid apportionment; this wrapper sums the apportioned seats back up into a single number for it.

Parameters
`evaluator` (*Distributor*) – The wrapped evaluator that expects unapportioned seats (i.e. a single number).

class `votelib.evaluate.core.`**`ByParty`**(*overall_evaluator*, *allocator=None*, *subsetter=<votelib.vote.SimpleSubsetter object>*)

Distribute overall party results among its constituency lists.

This is an inverted variant of the *ByConstituency* evaluator. Here, the total seat counts for each party are determined by an overall evaluator on nationally aggregated votes and an allocation distributor is subsequently used to disaggregate the result of the party to constituencies.

Useful for some mixed-member proportional system such as the German Bundestag one, where this mechanism is used to distribute overhang leveling seats.

Parameters

- **overall_evaluator** (*Distributor*) – Evaluator to use on the central (national) level.
- **allocator** (Optional[*Distributor*]) – An optional distribution evaluator to allocate seats of the party to individual constituencies. If None, the overall evaluator is reused. In either case, the allocator must accept simple votes for constituencies as candidates.
- **subsetter** (VoteSubsetter) – A subsetter according to the vote type used, to extract votes for any single party from the overall votes.

evaluate(*votes*, *n_seats=None*, *prev_gains={}*, *max_seats={}*)

Return constituency-wise election results evaluated by party.

Parameters

- **votes** (Dict[*Constituency*, Dict[Any, int]]) – Votes in the format accepted by the overall evaluator. They will be subsetted and aggregated to simple votes for constituencies for the allocator evaluation.
- **n_seats** (Optional[int]) – Number of seats to be allocated. If None, the overall evaluator must be seatless.
- **prev_gains** (Dict[*Constituency*, Dict[Union[str, CandidateObject], int]]) – Seats gained by the candidate/party in previous election rounds in each constituency, to inform the allocator. Not passed to the overall evaluator.
- **max_seats** (Dict[*Constituency*, Dict[Union[str, CandidateObject], int]]) – Maximum number of seats that the given candidate/party can obtain in total per constituency (including previous gains). Not passed to the overall evaluator.

Return type
Dict[*Constituency*, Dict[Union[str, CandidateObject], int]]

Returns
Results of the evaluation by constituency.

class `votelib.evaluate.core.`**`MultistageDistributor`**(*rounds*, *depth=1*)

A distribution evaluator with several rounds of awarding seats.

Useful for several systems, e.g. multi-member proportional systems where the first round evaluates the constituencies and the second one evaluates the national level.

Parameters

- **rounds** (List[*Distributor*]) – Partial distributors. Will be called one by one, with the subsequent distributors getting the results of the previous steps as previous gains.

- **depth** (int) – Nesting depth of the results. Use numbers larger than 1 when the results from the rounds are nested by constituency levels (2 for one constituency level, etc.)

evaluate(*votes*, *n_seats*, *prev_gains={}*, *max_seats={}*)

Evaluate all rounds of the distribution.

Parameters

- **votes** (Union[Dict[Any, int], List[Dict[Any, int]]]) – Votes to be evaluated. Either a single set to be passed to all rounds, or separate sets of votes for each round.
- **n_seats** (int) – Number of seats to be filled in total.
- **prev_gains** (Dict[Union[str, CandidateObject], int]) – Seats gained by the candidate/party in previous election rounds.
- **max_seats** (Dict[Union[str, CandidateObject], int]) – Maximum number of seats that the given candidate/party can obtain in total (including previous gains).

Return type

Dict[Union[str, CandidateObject], int]

class votelib.evaluate.core.**UnusedVotesDistributor**(*rounds*, *quota_functions=None*, *depth=1*)

Run several rounds of awarding seats, subtracting used votes.

In some elections, candidates who receive a seat have a certain quota of votes (usually the quota used to grant that seat) subtracted from their vote count, which then continues into subsequent rounds of evaluation. An example is the new (after 2021) Czech Parliament lower house election.

Parameters

- **rounds** (List[*Distributor*]) – Partial distributors. Will be called one by one, with the subsequent distributors getting the results of the previous steps as previous gains and the votes progressively lowered according to corresponding subtracted quotas.
- **quota_functions** (Optional[List[Union[str, Callable[[int, int], Number]]]]) – Quota functions to use for used vote subtraction. The length of this list should be one item smaller than the number of rounds (since after the last round, there is no meaningful subtraction).
- **depth** (int) – Nesting depth of the results. Use numbers larger than 1 when the results from the rounds are nested by constituency levels (2 for one constituency level, etc.)

evaluate(*votes*, *n_seats*, *prev_gains={}*, *max_seats={}*)

Evaluate all rounds of the distribution.

Parameters

- **votes** (Dict[Union[str, CandidateObject], int]) – Votes to be evaluated. Either a single set to be passed to all rounds, or separate sets of votes for each round.
- **n_seats** (int) – Number of seats to be filled in total.
- **prev_gains** (Dict[Union[str, CandidateObject], int]) – Seats gained by the candidate/party in previous election rounds.
- **max_seats** (Dict[Union[str, CandidateObject], int]) – Maximum number of seats that the given candidate/party can obtain in total (including previous gains).

Return type

Dict[Union[str, CandidateObject], int]

class votelib.evaluate.core.**AdjustedSeatCount**(*calculator*, *evaluator*)

Distribute an adjusted total number of seats according to votes cast.

Useful in multi-member proportional systems where overhang seats are accounted for, e.g. by leveling.

Parameters

- **calculator** (*SeatCountCalculator*) – A calculator determining how many seats to award. It is called with the evaluation parameters (votes, intended number of seats, previously gained seats…).
- **evaluator** (*Distributor*) – A distribution evaluator producing the actual results with the adjusted number of seats.

evaluate(*votes*, *n_seats*, *prev_gains*, *max_seats={}*)

Distribute an adjusted total number of seats.

Parameters

- **votes** (Dict[Any, int]) – Votes to be evaluated, of any type accepted by the calculator and evaluator.
- **n_seats** (int) – Intended (baseline) number of seats that would arise if no adjustment was necessary.
- **prev_gains** (Dict[Union[str, CandidateObject], int]) – Seats gained by the candidate/party in previous election rounds.
- **max_seats** (Dict[Union[str, CandidateObject], int]) – Maximum number of seats that the given candidate/party can obtain in total (including previous gains).

Return type

Dict[Union[str, CandidateObject], int]

class votelib.evaluate.core.**PartyListEvaluator**(*party_eval*, *list_eval=None*, *list_votes_converter=None*)

Evaluate which candidates are elected from party lists.

Useful to determine the actual representatives elected when the result is first evaluated by party. This wrapper evaluates the party-based result first, then uses a party list evaluator to determine which candidates from the given party lists actually get elected.

Parameters

- **party_eval** (*Distributor*) – A distribution evaluator determining the party-based election result.
- **list_eval** (Optional[OpenListEvaluator]) – An open list evaluator determining which candidates from the party list to elect, according to the preferential votes for the candidates. Look to the *votelib.evaluate.openlist* for some of these. If None, the party lists are considered closed and the elected candidates are taken from the top of each list, without considering the list votes.
- **list_votes_converter** (Optional[Converter]) – A converter to apply to list votes before passing to the list evaluator. The converter should produce a result where the candidate votes are grouped by party (such as votelib.convert.GroupVotesByParty). If not given, the list votes are passed unchanged.

evaluate(*votes*, *n_seats*, ***, *party_lists*, *list_votes=None*, ***kwargs*)

Return lists of candidates elected for each party.

All keyword arguments are passed to the party evaluator.

Parameters

- **votes** (Dict[Any, Number]) – Votes in the format accepted by the party evaluator.
- **n_seats** (int) – Number of seats to be allocated, to be passed to the party evaluator.
- **party_lists** (Dict[*ElectionParty*, List[Union[str, CandidateObject]]]) – Lists of candidates for each party in the order they were submitted for election.
- **list_votes** (Optional[Dict[*ElectionParty*, Dict[Any, Number]]]) – Votes for individual candidates on the party lists. If not given, the list evaluator must be None and the party lists are considered closed.

Raises

ValueError – If only one of list votes and a list evaluator is given.

Return type

Dict[*ElectionParty*, List[*Person*]]

Seat count adjustment calculators

class votelib.evaluate.core.**AllowOverhang**(*evaluator*)

Increase the total number of seats to allow keeping overhang seats.

Overhang seats arise in multi-member proportional systems when a party gains more seats in the first round (usually local and plurality-based) than its proportional share according to second round (national, proportional party-based) votes. This calculator increases the number of seats so that all parties keep their overhang seats while the number of seats awarded in the second round stays constant. This leads to parties that performed strong in the first round but weak in the second round gaining more representation than would be decided by the second round only.

This is the system used in New Zealand legislative elections.

Parameters

evaluator (*Distributor*) – A distribution evaluator producing the proportional results from the second round votes to determine which seats are overhang. Usually will be the same or very similar to the evaluator used in the second round directly.

calculate(*votes*, *n_seats*, *prev_gains*, *max_seats={}*)

Return an adjustment to the total number of seats to allow overhang.

Parameters

- **votes** (Dict[Any, int]) – Second round votes to produce the proportional result to determine overhang.
- **n_seats** (int) – Intended (baseline) number of seats that would arise if no adjustment was necessary. This is passed to the internal evaluator to determine the proportional result.
- **prev_gains** (Dict[Union[str, CandidateObject], int]) – Seats gained in the first round results.
- **max_seats** (Dict[Union[str, CandidateObject], int]) – Maximum number of seats that the given candidate/party can obtain in total, including first round results.

Return type

int

Returns

Zero if no adjustment is necessary, a positive integer if it is, amounting to the number of overhang seats detected.

class `votelib.evaluate.core.`**`LevelOverhang`**(*evaluator*)

Increase the total number of seats to proportional, keeping overhang.

Overhang seats arise in multi-member proportional systems when a party gains more seats in the first round (usually local and plurality-based) than its proportional share according to second round (national, proportional party-based) votes. This calculator increases the number of seats so that all parties keep their overhang seats while the overall seat counts for parties remain proportional according to the second round votes. This means that even when the first round seats end up dominated by a single party which has a huge overhang, the other parties still get enough *leveling* seats on the national level to maintain a representation proportional to their second round votes.

Parameters

`evaluator` (*`Distributor`*) – A distribution evaluator producing the proportional results from the second round votes to determine which seats are overhang. Usually will be the same or very similar to the evaluator used in the second round directly.

`calculate`(*votes*, *n_seats*, *prev_gains*, *max_seats={}*)

Return an adjustment to the total number of seats to level overhang.

Parameters

- **`votes`** (`Dict[Any, int]`) – Second round votes to produce the proportional result to determine overhang.
- **`n_seats`** (`int`) – Intended (baseline) number of seats that would arise if no adjustment was necessary. This is passed to the internal evaluator to determine the proportional result.
- **`prev_gains`** (`Dict[Union[str, CandidateObject], int]`) – Seats gained in the first round results.
- **`max_seats`** (`Dict[Union[str, CandidateObject], int]`) – Maximum number of seats that the given candidate/party can obtain in total, including first round results.

Return type

`int`

Returns

Zero if no adjustment is necessary, a positive integer if it is, amounting to the sum of overhang seats detected and leveling seats required.

class `votelib.evaluate.core.`**`LevelOverhangByConstituency`**(*constituency_evaluator*, *overall_evaluator=None*)

Increase the total number of seats to proportional in constituencies.

This is a variant *`LevelOverhang`* that works out overhang and leveling seats on a constituency level (such as Länder in Germany) while still maintaining nationwide proportionality. For a detailed description of the overhang leveling process, see that class. This variant detects overhang seats for each constituency separately and progressively increases the number of leveling seats until proportionality is satisfied on a national level.

This is the system used for the election to German Bundestag.

Parameters

- **`constituency_evaluator`** (*`Distributor`*) – Evaluates the total party result per constituency.
- **`overall_evaluator`** (`Optional[`*`Distributor`*`]`) – Evaluates the total party result nationwide; if not given, an aggregate of the constituency result is used instead.

`calculate`(*votes*, *n_seats*, *prev_gains*, *max_seats={}*)

Adjust the seat count to level overhang by constituency.

Parameters

- **votes** (Dict[*Constituency*, Dict[Any, int]]) – Second round votes to produce the proportional result to determine overhang, by constituency.
- **n_seats** (int) – Intended (baseline) number of seats that would arise if no adjustment was necessary. This is passed to the internal evaluator to determine the proportional result.
- **prev_gains** (Dict[*Constituency*, Dict[Union[str, CandidateObject], int]]) – Seats gained in the first round results by parties, by constituency.
- **max_seats** (Dict[*Constituency*, Dict[Union[str, CandidateObject], int]]) – Maximum number of seats that the given candidate/party can obtain in total, including first round results, by constituency.

Return type
int

Returns
Zero if no adjustment is necessary, a positive integer if it is, amounting to the sum of overhang seats detected and leveling seats required for all constituencies.

Auxiliary evaluation functions

votelib.evaluate.core.**get_n_best**(*votes*, *n_seats*)

Return n_seats candidates with the highest number of votes.

Essentially a plurality selection function. Produces ties correctly so is useful as a component in many other systems that use selection by maximum somewhere in their process.

Parameters

- **votes** (Dict[Union[str, CandidateObject], Number]) – Mapping of candidates to the number of votes obtained.
- **n_seats** (int) – Number of seats to be filled.

Return type
List[Union[str, CandidateObject, *Tie*]]

Returns
A list of top n_seats candidates. If there is a tie, the last items will refer to a single Tie object containing the tied candidates.

votelib.evaluate.core.**accepts_seats**(*evaluator*)

Whether evaluator takes seat count as an argument to evaluate().

Return type
bool

votelib.evaluate.core.**accepts_prev_gains**(*evaluator*)

Whether evaluator takes previous gains as an argument to evaluate().

Return type
bool

1.2 Converters API

Converters between vote and result formats.

These objects have a *convert()* method that converts between different formats of votes or election results.

1.2.1 Vote aggregators

`class votelib.convert.`**`ApprovalToSimpleVotes`**`(`*split=False*`)`

Aggregate approval votes to simple votes.

Aggregate votes for candidate sets (approval votes) to separate votes for individual candidates. Useful for example in approval voting.

Parameters

`split` (`bool`) – Whether to split the vote power among all the candidates in the set (as in satisfaction approval voting)[1] or give full vote to each (as in ordinary approval voting)[2].

`convert``(`*votes*`)`

Convert approval votes to simple votes.

Return type

`Dict[Union[str, CandidateObject], Number]`

`class votelib.convert.`**`ScoreToSimpleVotes`**`(`*function='mean'*, *unscored_value=None*, *min_count=0*, *truncation=0*, *bottom_value=0*`)`

Aggregate scores (cardinal votes) to simple votes.

Useful for score voting (range voting) systems. Note that, if the usual central value functions such as mean or median are used, this does not give scores that could be passed to ordinary magnitude-based evaluators since mean/median scores do not scale with the number of voters.

This can be used directly in combination with a simple-vote plurality evaluator but is also a component in several cardinal vote evaluators.

The scores can be arbitrary values as long as the aggregation function can cope with them, but numeric scores are the most common.

Parameters

- **`function`** (`Union[Callable[[List[Any]], Any], str]`) – A function to aggregate all scores given to a candidate to a single score. It can also be specified by name (string); this looks into the exact aggregator register in the utility module (which reimplements some common aggregation functions to use exact arithmetic, such as the mean, which is the default here) and then searches builtin functions and the statistics stdlib module namespaces.
- **`unscored_value`** (`Union[Callable[[List[Any]], Any], Any, None]`) – Score to give to a candidate that was not assigned a score by the voter. None means such ballots will not be considered for the candidate. A function can be specified to determine the value from all assigned scores (such as assigning the minimum). Builtin functions and functions from the statistics stdlib module can be specified by their names.
- **`min_count`** (`int`) – Minimum count of voter scores for the candidate to be considered. Candidates below this threshold will be assigned bottom_value. This is used in some systems to prevent an unknown upset candidate to win by high score mean although they were scored only by a handful of highly dedicated voters.

[1] "Satisfaction approval voting", Wikipedia. https://en.wikipedia.org/wiki/Satisfaction_approval_voting
[2] "Approval voting", Wikipedia. https://en.wikipedia.org/wiki/Approval_voting

- `truncation` (`Number`) – Fraction (if lower than 1) or count (if at least 1) of lowest and highest scores to disregard before aggregating, to stabilize the result. (Both ends are trimmed using this, so the number/fraction of scores disregarded is twice the count/fraction.)
- `bottom_value` (`Any`) – Value to assign to candidates with less voter scores than min_count. This would usually be the lowest possible aggregate score.

`aggregate`(*scores*)

Aggregate corrected score votes to scores per candidate.

Parameters
`scores` (`Dict[Union[str, CandidateObject], Dict[Any, int]]`) – Corrected scores in an intermediate format of a nested dict (candidates are mapped to dictionaries mapping score values to numbers of their occurrences).

Return type
`Dict[Union[str, CandidateObject], Any]`

Returns
A mapping from candidates to their aggregate scores.

`aggregate_one`(*cscores*)

Aggregate corrected scores for a candidate to a single score.

Return type
`Any`

`convert`(*votes*)

Convert score votes to simple votes.

Parameters
`votes` (`Dict[FrozenSet[Tuple[Union[str, CandidateObject], Any]], int]`) – Uncorrected score votes.

Return type
`Dict[Union[str, CandidateObject], Any]`

Returns
A mapping from candidates to their aggregate scores.

`corrected_scores`(*votes*)

Correct score votes to an intermediate format.

This forms the first part of the conversion (the second is *`aggregate()`*) and is exposed independently to allow for repeated aggregation which is used by some score voting evaluators.

The correction adds values for unscored candidates, assigns a fixed value to candidates with too few votes, and potentially truncates the scores to enable truncated central values.

Parameters
`votes` (`Dict[FrozenSet[Tuple[Union[str, CandidateObject], Any]], int]`) – Uncorrected score votes.

Return type
`Dict[Union[str, CandidateObject], Dict[Any, int]]`

Returns
Corrected scores in a nested dictionary (candidates are mapped to dictionaries mapping score values to numbers of their occurrences).

class `votelib.convert.`**`RankedToPositionalVotes`**(*rank_scorer*, *unranked_scoring='zero'*)

Aggregate ranked votes to simple votes.

Useful for Borda count systems. Assigns a score to each rank and then sums the scores. The complete Borda count system is obtained by coupling this converter with the Plurality evaluator.

Parameters

- **`rank_scorer`** (*`RankScorer`*) – A rank scorer that determines which score to assign to which rank through its *scores()* method. You can use any object that honors the interface of `rankscore.RankScorer`, such as any of its subclasses.
- **`unranked_scoring`** (`str`) – What score to assign to candidates not ranked by the given voter. So far only the *'zero'* option, which assigns a score of zero, is supported.

`convert`(*votes*)

Convert ranked votes to simple votes by scoring their positions.

Return type

`Dict`[`Union`[`str`, `CandidateObject`], `Number`]

class `votelib.convert.`**`RankedToCondorcetVotes`**(*unranked_at_bottom=True*)

Aggregate ranked votes to counts of pairwise wins.

Basic component for Condorcet methods. For each ballot that ranks a pair of candidates in a given order, adds one to the count of the first candidate over the second.

Parameters

`unranked_at_bottom` (`bool`) – Whether to consider candidates not ranked on a ballot as being ranked last. If False, these candidates are not considered (the voter is assumed not to have any preferences there).

`convert`(*votes*)

Convert ranked votes to counts of pairwise wins.

Return type

`Dict`[`Tuple`[`Union`[`str`, `CandidateObject`], `Union`[`str`, `CandidateObject`]], `int`]

class `votelib.convert.`**`ScoreToRankedVotes`**(*unscored_value=None*)

Convert score votes to ranked votes.

This is useful to employ ranked voting methods for run-off in some score voting systems to reduce their susceptibility to tactical voting (e.g. STAR voting).

Parameters

`unscored_value` (`Union`[`str`, `Number`, `None`]) – Score to give to a candidate that was not assigned a score by the voter. None means such ballots will not be considered for the candidate. A callable is not accepted.

`convert`(*votes*)

Convert score votes to ranked votes.

Parameters

`votes` (`Dict`[`FrozenSet`[`Tuple`[`Union`[`str`, `CandidateObject`], `Any`]], `int`]) – Score votes.

Return type

`Dict`[`Tuple`[`Union`[`str`, `CandidateObject`, `FrozenSet`[`Union`[`str`, `CandidateObject`]]], `...`], `int`]

1.2.2 Vote inverters to negative votes

class `votelib.convert.`**`InvertedSimpleVotes`**

Vote inverter to represent negative simple votes.

In some voting systems, voters vote against rather than for candidates.

static convert(*votes*)

Invert the count signs of single votes.

Return type
`Dict`[`Union`[`str`, `CandidateObject`], `Number`]

1.2.3 Individual candidate/party conversion

class `votelib.convert.`**`IndividualToPartyVotes`**(*mapper=<votelib.candidate.IndividualToPartyMapper object>*)

Aggregate votes for individual candidates to votes for their parties.

Useful for cases where votes are received by candidates but also considered by party, e.g. in panachage systems.

Parameters
mapper (*IndividualToPartyMapper*) – A mapper object specifying the mapping from individuals to parties.

convert(*votes*)

Convert individual simple votes to party-based simple votes.

Return type
`Dict`[*ElectionParty*, `int`]

class `votelib.convert.`**`IndividualToPartyResult`**(*mapper=<votelib.candidate.IndividualToPartyMapper object>*)

Aggregate individual elected candidates to results for their parties.

Useful to determine party results in systems (or parts thereof) where party affiliation is not taken into account during evaluation, e.g. in STV systems (Ireland) or in mixed-member proportional systems (Germany) to determine the number of directly elected candidates before party-based seats are allocated.

Parameters
mapper (*IndividualToPartyMapper*) – A mapper object specifying the mapping from individuals to parties.

convert(*results*)

Convert individual selection results to party-based counts.

Return type
`Dict`[*ElectionParty*, `int`]

1.2.4 Result conversion and merging

class `votelib.convert.`**`SelectionToDistribution`**(*amount=1*)

Adapt selection results to distribution (seat count) format.

This can be used e.g. for majority bonus systems where the largest party gets a predetermined amount of additional reserved seats, or in mixed-member proportional systems to determine party-wise results of the constituency round.

Parameters

`amount` (`Number`) – How many votes to attribute to each winner of the selection.

`convert`(*elected*)

Convert selection results to distribution results.

Return type

`Dict[Union[str, CandidateObject], int]`

class `votelib.convert.`**`MergedSelections`**

Compile candidates elected in constituencies to a single result list.

The candidates are ordered by their positions in the district-wide result lists.

Useful e.g. in mixed-member proportional systems to list all candidates elected in constituencies.

`convert`(*elected*)

Compile constituency election results to a single result list.

Parameters

`elected` (`Union[Dict[`*`Constituency`*`, List[Union[str, CandidateObject]]], List[List[Union[str, CandidateObject]]]]`) – Partial selection results in a list or dictionary; if a dictionary, its keys are ignored and values treated as a list.

Return type

`List[Union[str, CandidateObject]]`

class `votelib.convert.`**`MergedDistributions`**

Merge many distribution election results into one.

Aggregates distribution election results from a list or dictionary of partial results (e.g. by constituency) into a single candidate-wise dictionary. Neither reconciles ties nor preserves result ordering.

Use *`VoteTotals`* to aggregate votes.

`convert`(*elected*)

Aggregate many distribution election results into one.

Parameters

`elected` (`Union[Dict[`*`Constituency`*`, Dict[Union[str, CandidateObject], int]], List[Dict[Union[str, CandidateObject], int]]]`) – Partial distribution election results in a list or dictionary; if a dictionary, its keys are ignored and values treated as a list.

Return type

`Dict[Union[str, CandidateObject], int]`

1.2.5 Constituency-based vote handling and aggregation

class `votelib.convert.`**`VoteTotals`**

Count total votes/results for each candidate in all constituencies.

If votes of other type than simple are given, counts the totals of votes. Use *`MergedDistributions`* to aggregate distribution election results.

`convert`(*votes*)

Count total votes for each candidate in all constituencies.

Parameters
votes (`Dict[`*`Constituency`*`, Dict[Any, int]]`) – Votes of any type.

Return type
`Dict[Any, int]`

class `votelib.convert.`**`ConstituencyTotals`**

Count total votes (or results) for all candidates in each constituency.

Accepts any type of votes or distribution election results.

Useful for apportionment of seats to districts.

`convert`(*votes*)

Return total votes for all candidates in each constituency.

Parameters
votes (`Dict[`*`Constituency`*`, Dict[Any, int]]`) – Votes of any type, or distribution election results.

Return type
`Dict[`*`Constituency`*`, int]`

class `votelib.convert.`**`PartyTotals`**

Count total votes for parties from grouped votes for its candidates.

Accepts any type of votes or distribution election results.

Useful for evaluating party-based results in systems where votes can be received by individual candidates (panachage) if the results are already grouped by party in a nested dictionary, e.g. after applying `GroupVotesByParty`.

`convert`(*votes*)

Return total votes for all candidates of each party.

Parameters
votes (`Dict[`*`ElectionParty`*`, Dict[Any, int]]`) – Votes of any type, or distribution election results.

Return type
`Dict[`*`ElectionParty`*`, int]`

class `votelib.convert.`**`ByConstituency`**(*converter*)

Perform conversion for each constituency votes/results separately.

Parameters
converter (`Converter`) – A converter to wrap. Will be called to convert votes (or results) for each constituency separately.

`convert`(*values*)

Convert the votes/results for each constituency.

Parameters

values (Dict[*Constituency*, Dict[Any, Any]]) – Mapping of constituencies to votes or results. The keys of this dictionary will be transferred unchanged to the result, with their values converted by the wrapped converter.

Return type

Dict[*Constituency*, Dict[Any, Any]]

1.2.6 Vote corrections and subsetting

class votelib.convert.**RoundedVotes**(*decimals*, *round_method='ROUND_HALF_UP'*)

Round vote counts to the given number of decimal digits.

In some systems, rounding of fractional values to a given number of digits is specified (e.g. Scottish STV). This rounds vote counts of any vote type to this number of decimals.

Parameters

- **decimals** (int) – Number of digits to round to. Negative values are not accepted.
- **round_method** – A rounding method accepted by Python's `decimal` module.

Raises

ValueError – If an invalid number of decimal digits is given.

`convert`(*votes*)

Round all vote counts to the specified number of votes.

The vote counts must be convertible to `Decimal` (`Fraction` and any types accepted by the decimal constructor are supported).

Parameters

votes (Dict[Any, Number]) – Votes whose counts should be rounded.

Return type

Dict[Any, Decimal]

class votelib.convert.**SubsettedVotes**(*vote_subsetter=<votelib.vote.SimpleSubsetter object>*, *depth=0*)

Subset the votes to only concern a subset of candidates.

A wrapper over `VoteSubsetter` that takes the entire vote count dictionary, not just a single vote object (key). Useful when some candidates should be excluded because they do not pass an electoral threshold, or when evaluating ties.

Parameters

vote_subsetter (VoteSubsetter) – A vote subsetter that turns a single vote object (key) into a vote object that only concerns the specified candidates, with other candidates removed.

`convert`(*votes*, *subset*)

Subset the votes to only concern a subset of candidates.

Parameters

- **votes** (Dict[Any, Number]) – Votes to be subsetted. Their type should be in accordance with the wrapped vote subsetter.
- **subset** (Collection[Union[str, CandidateObject]]) – The only candidates that should be contained in the output.

Return type
`Dict[Any, Number]`

class `votelib.convert.`**`InvalidVoteEliminator`**`(`*validator*`)`

Only allow through votes that are declared valid by a given validator.

Calls the *validate()* method of the validator for each of the keys of the input dictionary; if an `vote.VoteError` is raised, does not include the vote in the output.

Parameters
`validator` (*`VoteValidator`*) – The vote validator to use. Look for some in the `vote` module.

`convert``(`*votes*`)`

Copy the votes dictionary, removing invalid votes.

If no invalid votes are detected, does not make a copy.

Return type
`Dict[Any, int]`

1.3 Quality criteria API

1.3.1 Yee diagrams

Create and plot Yee diagrams outlining single-winner voting system quality.

Yee diagrams distribute some candidates in a 2D issue space and then evaluate a selected voting system on a grid over that space. In each cell of the grid, an election is simulated with voter preferences proportional to the distances of the grid cell to the candidate positions. Use `diagram()` to produce this.

A good voting system's Yee diagram should closely approximate a Voronoi diagram over the candidates. (A Voronoi diagram would assign each point of space to the closest candidate.)[1] This can be produced by calling *`voronoi()`*.

The Yee diagram here is represented by a 2D list as a rectangular lattice covering the 0-1 in both dimensions.

`votelib.crit.yee.`**`bounded_sampler`**`(`*samp*`)`

Bound a vote sampler to the Yee diagram bounding box.

Return type
`BoundedSampler`

`votelib.crit.yee.`**`circular_candidates`**`(`*n*, *r=0.25*, *center=(0.5, 0.5)*`)`

Generate Yee diagram candidate issue space positions in a circle.

Parameters

- **`n`** (`int`) – Number of candidates. The candidates will be evenly spaced along the circumference of the circle.
- **`r`** (`float`) – Radius of the circle.
- **`center`** (`Tuple[float, float]`) – Center of the circle coordinates.

Return type
`Dict[Union[str, CandidateObject], Tuple[float, float]]`

[1] Warren D. Smith. "Yee Pictures", Range Voting, 2007. https://rangevoting.org/IEVS/Pictures.html

`votelib.crit.yee.`**`diagram`**(*evaluator*, *candidates*, *shape=(200, 200)*, *n_votes=1000*, *random_state=None*, ***kwargs*)

Produce a Yee diagram for the evaluator.

Any superfluous keyword arguments are passed on to the vote generation sampler using *`sampler()`*.

Parameters

- **`evaluator`** (*`Selector`*) – The voting system evaluator for which to construct a Yee diagram.
- **`candidates`** (`Dict[Union[str, CandidateObject], Tuple[float, float]]`) – Candidate positions in the 2D issue space. The coordinates should both be between 0 and 1.
- **`shape`** (`Tuple[int, int]`) – Number of cells along each dimension of the issue space. Enlarging this gives more detail but increases computing time.
- **`n_votes`** (`int`) – Number of voters to consider in each simulated election. Lowering this will speed up the computation but may give unstable results.
- **`random_state`** (`Optional[int]`) – Seed for the vote generator.

Return type

`List[List[Union[str, CandidateObject]]]`

`votelib.crit.yee.`**`plot`**(*results*, *candidates*, *cmap='tab10'*, *ax=None*)

Plot the Yee diagram.

Requires Matplotlib. Plots the Yee diagram cells and candidate positions on the current/chosen axes and adds a legend. You normally need to follow up on this with the `show()` call to show the figure.

Parameters

- **`results`** (`List[List[Union[str, CandidateObject]]]`) – The Yee diagram to plot.
- **`candidates`** (`Dict[Union[str, CandidateObject], Tuple[float, float]]`) – Candidate positions in the 2D issue space. The coordinates should both be between 0 and 1.
- **`cmap`** (`str`) – Colormap to use for plotting. The default is sufficient for up to 10 candidates unambiguously. References the Matplotlib colormap register.
- **`ax`** – Axes to plot on.

Return type

`None`

`votelib.crit.yee.`**`random_candidates`**(*n*, *cand_sampler=None*)

Generate Yee diagram candidate issue space positions randomly.

Parameters

- **`n`** (`int`) – Number of candidates.
- **`cand_sampler`** (`Optional[Sampler]`) – A sampler to generate the candidate positions. The default (None) is to sample from a uniform 2D distribution, i.e. all points of the diagram are equally probable.

Return type

`Dict[Union[str, CandidateObject], Tuple[float, ...]]`

`votelib.crit.yee.`**`sampler`**(*x*, *y*, *sigma=(0.25, 0.25)*)

Create a vote generation sampler for a given Yee issue space point.

Gives a Gaussian sampler centered on the x-y point with a standard deviation given by sigma.

Return type
BoundedSampler

`votelib.crit.yee.`**`voronoi`**`(`*candidates*, *shape=(200, 200)*`)`

Produce a Voronoi Yee diagram for the given candidates.

This gives the ideal Yee diagram that good voting systems should be close to.

Parameters

- **`candidates`** (`Dict[Union[str, CandidateObject], Tuple[float, float]]`) – Candidate positions in the 2D issue space. The coordinates should both be between 0 and 1.
- **`shape`** (`Tuple[int, int]`) – Number of cells along each dimension of the issue space. Enlarging this gives more detail but increases computing time.

Return type
`List[List[Union[str, CandidateObject]]]`

`votelib.crit.yee.`**`voronoi_conformity`**`(`*results*, *candidates*`)`

Calculate the percentage of how a Yee diagram matches a Voronoi diagram.

Parameters

- **`results`** (`List[List[Union[str, CandidateObject]]]`) – A Yee diagram of winning candidates.
- **`candidates`** (`Dict[Union[str, CandidateObject], Tuple[float, float]]`) – Candidate positions in the 2D issue space. The coordinates should both be between 0 and 1.

Return type
`float`

`votelib.crit.yee.`**`voronoi_matches`**`(`*results*, *candidates*`)`

Give a 2D boolean mask how a Yee diagram matches a Voronoi diagram.

Parameters

- **`results`** (`List[List[Union[str, CandidateObject]]]`) – A Yee diagram of winning candidates.
- **`candidates`** (`Dict[Union[str, CandidateObject], Tuple[float, float]]`) – Candidate positions in the 2D issue space. The coordinates should both be between 0 and 1.

Return type
`List[List[bool]]`

1.3.2 Election result proportionality measurements

Measure proportionality of election results.

These functions evaluate how proportionally the seats are allocated to parties according to their votes received. For a review of such indicators, see[2] or[3].

The functions only accept votes in simple format. To evaluate disproportionality for elections using other vote types, you must use an appropriate converter first; however, using the index for those election systems might not be meaningful.

The seat counts must be in a dictionary format as returned by votelib's distributors.

[2] "polrep: Calculate Political Representation Scores", Didier Ruedin. https://rdrr.io/rforge/polrep/
[3] "Measures of disproportionality", Kalogirou. http://www2.stat-athens.aueb.gr/~jpan/diatrives/Kalogirou/chapter5.pdf

`votelib.crit.proportionality.d_hondt(votes, results)`

Compute the D'Hondt index of disproportionality.[Page 45, 3]

This is the index of disproportionality minimized by the D'Hondt highest averages evaluator. It uses the maximum ratio of seats to votes across parties.

Parameters

- **votes** (`Dict[Union[str, CandidateObject], Number]`) – Numbers of votes for each candidate.
- **results** (`Dict[Union[str, CandidateObject], Number]`) – Seat counts awarded to each candidate.

Return type

`float`

`votelib.crit.proportionality.gallagher(votes, results)`

Compute the Gallagher index of election result disproportionality.

The Gallagher (LSq) index[4] expresses the mismatch between the fraction of votes received and seats allocated for each candidate or party. The index ranges from zero (no disproportionality) to 1 (total disproportionality). Compared to the Loosemore–Hanby index, it highlights large deviations rather than small ones.

Parameters

- **votes** (`Dict[Union[str, CandidateObject], Number]`) – Numbers of votes for each candidate.
- **results** (`Dict[Union[str, CandidateObject], Number]`) – Seat counts awarded to each candidate.

Return type

`float`

`votelib.crit.proportionality.lijphart(votes, results)`

Compute the Lijphart's index of disproportionality.[Page 45, 3]

Lijphart's index takes the single largest difference between vote and seat fractions.

Parameters

- **votes** (`Dict[Union[str, CandidateObject], Number]`) – Numbers of votes for each candidate.
- **results** (`Dict[Union[str, CandidateObject], Number]`) – Seat counts awarded to each candidate.

Return type

`float`

`votelib.crit.proportionality.loosemore_hanby(votes, results)`

Compute the Loosemore–Hanby index of election result disproportionality.

The Loosemore–Hanby (LH) index[5] expresses the mismatch between the fraction of votes received and seats allocated for each candidate or party. The index ranges from zero (no disproportionality) to 1 (total disproportionality). Compared to the Gallagher index, it does not diminish the effect of smaller deviations.

Parameters

[4] "Gallagher index", Wikipedia. https://en.wikipedia.org/wiki/Gallagher_index

[5] "Loosemore–Hanby index", Wikipedia. https://en.wikipedia.org/wiki/Loosemore%E2%80%93Hanby_index

- **votes** (Dict[Union[str, CandidateObject], Number]) – Numbers of votes for each candidate.
- **results** (Dict[Union[str, CandidateObject], Number]) – Seat counts awarded to each candidate.

Return type
`float`

`votelib.crit.proportionality.`**`rae`**`(`*votes*, *results*`)`

Compute Rae's index of disproportionality.[Page 45, 3]

Rae's index is the earliest known disproportionality measure. It is known to underestimate disproportionality in the presence of small parties.

Parameters

- **votes** (Dict[Union[str, CandidateObject], Number]) – Numbers of votes for each candidate.
- **results** (Dict[Union[str, CandidateObject], Number]) – Seat counts awarded to each candidate.

Return type
`float`

`votelib.crit.proportionality.`**`regression`**`(`*votes*, *results*`)`

Compute the regression index of disproportionality.[Page 45, 3]

This is obtained by performing linear regression to predict seat fractions from vote fractions. If the index is one, the allocation is perfectly proportional; values below one signal preference of the system for smaller parties, while values above one signal preference for larger parties.

Parameters

- **votes** (Dict[Union[str, CandidateObject], Number]) – Numbers of votes for each candidate.
- **results** (Dict[Union[str, CandidateObject], Number]) – Seat counts awarded to each candidate.

Return type
`float`

`votelib.crit.proportionality.`**`rose`**`(`*votes*, *results*`)`

Compute the Rose disproportionality index (inverse LH).[Page 45, 3]

This is an inverted version of the Loosemore-Hanby index which ranges from 1 (no disproportionality) to 0 (total disproportionality).

Parameters

- **votes** (Dict[Union[str, CandidateObject], Number]) – Numbers of votes for each candidate.
- **results** (Dict[Union[str, CandidateObject], Number]) – Seat counts awarded to each candidate.

Return type
`float`

`votelib.crit.proportionality.`**`sainte_lague`**(*votes*, *results*)

Compute the Sainte-Laguë index of disproportionality.[Page 45, 3]

Sainte-Laguë index takes fractional differences.

Parameters

- **`votes`** (`Dict`[`Union`[`str`, `CandidateObject`], `Number`]) – Numbers of votes for each candidate.
- **`results`** (`Dict`[`Union`[`str`, `CandidateObject`], `Number`]) – Seat counts awarded to each candidate.

Return type

`float`

1.4 Vote generation API

Generate votes for voting system simulations.

The generators implemented here mostly produce score votes since those are the most general type. For other types, convert the result using converters:

- To get ranked votes from score votes, use *`votelib.convert.ScoreToRankedVotes`*.
- To get simple votes from score votes, use *`votelib.convert.ScoreToRankedVotes`* and `votelib.convert.RankedToFirstPreference` (wrapped in `votelib.convert.Chain`).
- To get approval votes over a threshold, use `votelib.convert.ScoreToApprovalVotesThreshold`.

1.4.1 Vote generators

`class votelib.generate.`**`IssueSpaceGenerator`**(*candidates*, *sampler='gauss'*, *vote_creation='minmax'*, *random_state=None*)

Generate random votes by sampling from a multidimensional issue space.

The most common paradigm for random vote generation is to spread some candidates as points into a multidimensional space (representing their opinions or characteristics), then sample voter points from a statistical distribution on that space and determine their votes based on their proximity to candidates. This class implements this paradigm, delegating the sampling to a contained instance of `Sampler`.

Parameters

- **`candidates`** (`Union`[`int`, `Dict`[`Union`[`str`, `CandidateObject`], `Tuple`[`float`, `...`]]]) – Positions of candidates in the issue space. It is also possible to specify just an integer; in that case, the candidate positions will be sampled in the same way the voters are.
- **`sampler`** (`Union`[`str`, `Sampler`]) – How to sample the voter points. Either an object with a `sample()` method that yields numerical tuples with the correct number of dimensions (such as instances of `Sampler` subclasses), or a string referencing a name of a statistical distribution; in that case, *`DistributionSampler`* will be invoked in two dimensions with default settings on the specified distribution. See the class documentation for more on supported distributions.
- **`vote_creation`** (`str`) – How to transform voter-to-candidate proximities in the issue space to score votes:

  - `minmax`: The closest candidate is assigned a score of 1, the furthest is assigned a score of 0. Score votes are produced.
  - `closest`: The closest candidate is voted for. Simple votes are produced.
  - Other transformations are not implemented.
- **random_state** (`Optional[int]`) – Seed for the sampler.

generate(*n*)

Generate n votes for the candidate setup.

Return type
`Dict[Any, int]`

samples_to_votes(*sample*, *candidates*)

Convert issue space samples to votes.

Return type
`Dict[Any, int]`

class votelib.generate.**ScoreSpaceGenerator**(*candidates*, *sampler='uniform'*, *round_scores=False*, *random_state=None*)

Randomly sample independent candidate scorings.

This generator is simpler than *IssueSpaceGenerator* since it uses the underlying sampler to produce candidate scorings directly. The candidate scores are independent of each other.

Under default settings, this generator produces the *Impartial Culture* (IC). By providing different settings for different dimensions of the passed sampler, score probabilities for individual candidates can be set. An example for two candidates, in which A will, on average, get higher scores (mean 0.7) than B (mean 0.3):

```
ScoreSpaceGenerator(
    candidates=['A', 'B'],
    sampler=BoundedSampler(
        DistributionSampler('gauss', mu=(0.7, 0.3), sigma=(1, 1)),
        bbox=(0, 1)
    )
)
```

Parameters

- **candidates** (`Union[int, List[Union[str, CandidateObject]]]`) – Candidate objects (most straightforwardly, a list of candidate names as strings). It is also possible to specify just an integer; in that case, the candidate names are autogenerated as uppercase ASCII letters (A, B, C…).
- **sampler** (`Union[str, Sampler]`) – How to sample the scores. Either an object with a `sample()` method that yields numerical tuples with the correct number of dimensions (such as instances of `Sampler` subclasses), or a string referencing a name of a statistical distribution; in that case, *DistributionSampler* will be invoked with default settings on the specified distribution. See the class documentation for more on supported distributions.
- **round_scores** (`bool`) – Whether to round the scores (using round half to even) to integers.
- **random_state** (`Optional[int]`) – Seed for the sampler.

generate(*n*)

Generate n votes.

Return type
`Dict[Any, int]`

1.4.2 Random samplers

class votelib.generate.**DistributionSampler**(*distribution='gauss'*, *n_dims=None*, ***kwargs*)

Sample points from the issue or score space by specifying a distribution.

Uses a statistical probability distribution to produce randomly located points within the issue space or score space of given dimensionality. The distributions are taken from Python's *random* module by referencing the names of the generating functions.

The outputs from this sampler need to be fed into *IssueSpaceGenerator* to produce votes or specify candidate positions or into *ScoreSpaceGenerator* to be converted to candidate scorings directly.

Any superfluous keyword arguments are passed to the generating function from the random module. If no keyword arguments are given but are required (i.e. the distribution parameters need to be specified), for some distributions (uniform, gauss, triangular, beta), defaults are specified in this class and automatically used if necessary.

Parameters

- **distribution** (`str`) – The name of the distribution to use. Must refer to a name of a function in Python stdlib random module that produces random floats.
- **n_dims** (`Optional[int]`) – Dimensionality of the issue or score space to sample from.

sample(*n*, *n_dims=None*)

Sample n issue space samples from the distribution.

Return type
`Iterable[Tuple[float, ...]]`

class votelib.generate.**BoundedSampler**(*inner*, *bbox*)

A sampler from a bounded issue space.

Wraps another sampler to only produce issue space samples that lie within a specified multidimensional bounding box. Useful e.g. for the generation of Yee diagrams.

The outputs from this sampler need to be fed into `SamplingGenerator` to produce votes or specify candidate positions.

Parameters

- **inner** (`Sampler`) – A sampler (e.g. *DistributionSampler*) to wrap. Its samples are filtered by the specified bounding box.
- **bbox** (`Tuple[Number, ...]`) – The bounding box to restrict the samples to. First, all minima per dimension are specified, then all maxima; for two dimensions, this would be `(minx, miny, maxx, maxy)`. If the inner sampler has a defined number of dimensions, it is also possible to specify only two numbers, which will be interpreted as the minimum and maximum in each dimension.

sample(*n*, **args*, ***kwargs*)

A generator to sample n bbox-restricted issue space samples.

Return type
`Iterable[Tuple[float, ...]]`

1.5 Candidate objects and candidacy (nomination) validation API

Candidate specifications and nomination validators.

Contains definitions of candidate types and interfaces (`Candidate`, `ElectionParty`, `Coalition`), constituencies (`Constituency`), special vote variants (`NoneOfTheAbove`, `ReopenNominations`) and candidate/nomination validators (`BasicNominator`, `PersonNominator`, `PartyNominator`).

Apart from the candidate objects specified here, Votelib also accepts strings with candidate names in all concerned places. The subclasses of this class are thus only needed for systems considering candidate properties such as coalition size-dependent electoral thresholds. For a particular example of this, the `votelib.evaluate.threshold.PropertyBracketer` chooses different rules based on a value of an arbitrary property of the candidate. You can use any object that has the specified property defined; the classes defined here can be assigned arbitrary additional properties, so they, too, can be used.

1.5.1 Candidate objects and interfaces

`votelib.candidate.`**`Candidate`**

alias of `Union`[`str`, `CandidateObject`]

`class votelib.candidate.`**`IndividualElectionOption`**

An abstract class for individuals standing for an election.

`candidacy_for: Optional[ElectionParty] = NotImplemented`

The party for which this choice is candidating for the election.

This is important for counting party-based election results where the votes are cast and counted for candidates.

If the candidate is endorsed by multiple parties, use a `Coalition` object.

`membership: Optional[PoliticalParty] = NotImplemented`

The party the candidate is member of.

This is important for counting party-based election results where the votes are cast and counted for candidates.

`class votelib.candidate.`**`Person`**(*name*, *number=None*, *membership=None*, *candidacy_for=None*, *properties=None*, *withdrawn=False*)

A physical person standing for the election.

Parameters

- **`name`** (`str`) – Name of the person, in any customary text format.
- **`number`** (`Optional[int]`) – Candidacy number assigned to the person for the purpose of the election; usually drawn by lot.
- **`membership`** (`Optional[PoliticalParty]`) – A political party the person is member of, if any.
- **`candidacy_for`** (`Optional[ElectionParty]`) – A political party for which the person is standing in the election. If there is no such party, the person is considered an independent candidate.
- **`withdrawn`** (`bool`) – Whether the candidate withdrew from the election (and is thus ineligible to get elected).

`class votelib.candidate.`**`ElectionParty`**

A subject that is regarded as a political party for the election.

`is_coalition = NotImplemented`

Whether the party is a coalition.

This makes a difference for some systems; the most common case is a heightened vote threshold in proportional elections.

`class votelib.candidate.`**`PoliticalParty`**`(`*name*, *number=None*, *affiliations=None*, *lead=None*, *properties=None*, *withdrawn=False*`)`

A political party or movement that is eligible to stand in elections.

Parameters

- **`name`** (`str`) – Name of the party, in any customary text format.
- **`number`** (`Optional[int]`) – Candidacy number assigned to the party for the purpose of the election; usually drawn by lot.
- **`affiliation`** – Other (e.g. national or supranational) parties this party is affiliated with, if any.
- **`lead`** (`Optional[`*`Person`*`]`) – A person that leads the party into the elections.

`is_coalition = False`

Whether the party is a coalition.

This makes a difference for some systems; the most common case is a heightened vote threshold in proportional elections.

`class votelib.candidate.`**`Coalition`**`(`*parties*, *name=None*, *number=None*, *affiliations=None*, *lead=None*, *withdrawn=False*`)`

A coalition of two or more election-eligible parties.

Parameters

- **`parties`** (`List[`*`PoliticalParty`*`]`) – Parties involved in the coalition.
- **`name`** (`Optional[str]`) – Name of the coalition, in any customary text format. If not given, it is derived from the names of the member parties.
- **`number`** (`Optional[int]`) – Candidacy number assigned to the coalition for the purpose of the election; usually drawn by lot.
- **`affiliations`** (`Optional[List[`*`PoliticalParty`*`]]`) – Other (e.g. national or supranational) parties this coalition is affiliated with, if any.
- **`lead`** (`Optional[`*`Person`*`]`) – A person that leads the coalition into the elections.

`get_n_coalition_members``()`

Return the number of member parties in the coalition.

This is important in some elections which specify different thresholds for coalitions with a given number of members.

Return type

`int`

`is_coalition = True`

Whether the party is a coalition.

This makes a difference for some systems; the most common case is a heightened vote threshold in proportional elections.

Blank votes and other special vote options

`class votelib.candidate.BlankVoteOption(name)`

A base class for blank (non-partisan) vote choices.

This includes votes that are not counted to any candidate. In some rare cases, these votes have a special effect (such as triggering a new election if there is a sufficient number of them), but most of the time they can be safely disregarded.

`candidacy_for: Optional[ElectionParty] = None`

The party for which this choice is candidating for the election.

This is important for counting party-based election results where the votes are cast and counted for candidates.

If the candidate is endorsed by multiple parties, use a `Coalition` object.

`is_coalition = False`

Whether the party is a coalition.

This makes a difference for some systems; the most common case is a heightened vote threshold in proportional elections.

`membership: Optional[PoliticalParty] = None`

The party the candidate is member of.

This is important for counting party-based election results where the votes are cast and counted for candidates.

`class votelib.candidate.NoneOfTheAbove(name)`

None of the above (NOTA) vote option (often called white ballots).

In some contexts, a sufficient number of these votes can trigger a special effect such as a new election.

Parameters

name (`str`) – Actual name of the NOTA option in the given context.

`class votelib.candidate.ReopenNominations(name)`

Reopen Nominations vote option.

In some elections, this option is present; if it prevails, it can trigger a new round of nominations and a new election.

Parameters

name (`str`) – Actual name of the Reopen Nominations option in the given context.

Mapping candidates to parties

`class votelib.candidate.IndividualToPartyMapper(affiliation='candidacy_for', independents='aggregate')`

Define mapping from individuals to parties.

A candidate object such as a `Person` might define multiple party relationships or define that the candidate is standing for the election as an independent. This object can be added as a component to some converters or evaluators to map individuals to parties correctly according to the rules of the particular election system.

Parameters

- **affiliation** (`str`) – How to allocate the candidate to the party:

  - *'candidacy_for'* (default) uses the party the candidate stood for (was endorsed by) in the election,
  - *'membership'* uses the party the candidate is a member of.
- **independents** (`str`) – How to handle individual candidates not candidating for a party:
  - *'aggregate'* (default) aggregates them under the None key to a total count,
  - *'keep'* keeps them separately,
  - *'ignore'* omits them from the result,
  - *'error'* raises an error.

1.5.2 Constituency objects

`class votelib.candidate.`**`Constituency`**

A constituency for elections with multiple sets of candidates.

Constituencies are used where the electorate is separated into more than one group. The most common variant is spatial - electoral or voting districts (also called wards or precincts). This object should be used only for cases where such spatial division is used to evaluate the result, not merely used to collect and count the ballots.

Non-spatial variants include curias (used in 19th century Austria-Hungary for different social classes), ethnical divisions (Maori electorates in New Zealand) or qualificational divisions (University constituencies in Ireland).

In the current version of Votelib, special constituency objects are not needed (any hashable objects such as strings can be used for all functionality implemented so far and will continue to be supported by the existing features in the future) so this is essentially just a type marker, but further development might necessitate creating some subclasses of this, and more specific interfaces.

1.5.3 Nomination (candidate) validators

`class votelib.candidate.`**`BasicNominator`**(*allow_blank=True*)

Validate that the election candidates are valid objects.

Does not do any logical checks; only validates that the candidate instance passes the criteria of the `Candidate` class (such as checking against most types of collections).

Parameters
allow_blank (`bool`) – Whether to allow blank votes (NOTA, ReopenNominations).

`validate`(*candidate*)

Check whether a candidate is valid.

Parameters
candidate (`Union[str, CandidateObject]`) – Candidate to be checked.

Raises
`CandidateError` – If a candidate is invalid.

Return type
`None`

`class votelib.candidate.`**`PersonNominator`**(*allow_independents=True*, *allow_blank=True*)

Validate that election candidates are physical persons and not parties.

Parameters

- `allow_independents` (`bool`) – Whether persons candidating without a support of a political party can stand in the election.
- `allow_blank` (`bool`) – Whether to allow blank votes (NOTA, ReopenNominations).

`validate`(*candidate*)

Check whether a candidate is valid.

Parameters
`candidate` (`Union[str, CandidateObject]`) – Candidate to be checked.

Raises
`CandidateError` – If a candidate is invalid.

Return type
`None`

`class votelib.candidate.PartyNominator`(*allow_coalitions=True*, *allow_blank=True*)

Check that election candidates are electoral parties, not persons.

This includes political parties in the broadest sense of the term, as well as their coalitions formed for the purpose of the election.

Only instances of *`ElectionParty`* and its subclasses are accepted, as are classes that pass the interface check of the class, if there is one. Blank votes are accepted by default and can be disallowed, like coalitions.

Parameters

- `allow_coalitions` (`bool`) – Whether to allow coalitions.
- `allow_blank` (`bool`) – Whether to allow blank votes (NOTA, ReopenNominations).

`validate`(*candidate*)

Check whether a candidate is a valid election party.

Parameters
`candidate` (`Union[str, CandidateObject]`) – Candidate to be checked.

Raises
`CandidateError` – If a candidate is not a valid election party candidate.

Return type
`None`

An abstract class defining the nominator interface is also present.

`class votelib.candidate.Nominator`

An abstract class for nominators (candidacy validators).

`abstract validate`(*candidate*)

Check if the candidate satisfies criteria given by the system.

Raises
`NotImplementedError` –

Return type
`None`

1.5.4 Nomination (candidate) validation errors

class `votelib.candidate.`**`CandidateError`**(*candidate*, *expected=None*)

A candidate is invalid in the given context.

E.g. parties in place of individual candidates, or candidates ineligible to stand for given seats.

Parameters

- **`candidate`** (`Any`) – Candidate that was found to be invalid.
- **`expected`** (`Optional[Any]`) – Definition of a candidate that was expected.

1.6 Vote objects and vote validation API

Vote type specifications and vote validators.

Vote types vary from system to system and are only loosely tied to the method of evaluation (usually, the richer vote types can be reduced to use simple evaluators but not vice versa). The following vote types are recognized by Votelib:

- **Simple** votes - a voter votes for a single candidate. Represented by the candidate object itself.
- **Approval** votes - a voter selects a number of candidates and votes for them equally. Represented by a frozen set of candidate objects.
- **Ranked** votes - a voter ranks a number of candidates. Represented by a tuple of candidate objects or frozen sets of them (to account for possible tied rankings).
- **Score** votes - a voter assigns a score (label) to a number of candidates. The scores might either be selected from a predefined set or arbitrary numbers from a range.

Voting systems generally require one of the above types and usually impose additional restrictions such as minimum and maximum number of candidates voted for or ranked. These restrictions are handled by parameters accepted by the vote validators.

Vote validators validate individual votes, i.e. the keys of the dictionaries that should be passed to the evaluators. If a vote is invalid, they raise a subclass of *`VoteError`* (or `CandidateError`, if a candidate contained in the vote is invalid). Some converter objects wrap these validators and catch these errors to remove invalid votes, for example.

1.6.1 Vote validators

class `votelib.vote.`**`SimpleVoteValidator`**(*nominator=<votelib.candidate.BasicNominator object>*)

Validate a simple vote (voting directly for a single candidate).

The candidate must be a valid atomic candidate object according to the nominator object specified.

Parameters

`nominator` (*`Nominator`*) – Nominator used to check candidates. The default uses only technical criteria specified by the `Candidate` class.

`validate`(*vote*)

Check if the candidate is valid.

Parameters

`vote` (`Union[str, CandidateObject]`) – Simple vote to be checked.

Raises

`CandidateError` – If the candidate is invalid.

Return type
`None`

class `votelib.vote.`**`ApprovalVoteValidator`**(*vote_count_bounds=(None, None)*, *count_checker=None*, *nominator=<votelib.candidate.BasicNominator object>*)

Validate an approval vote (voting for a number of candidates equally).

The vote must be a frozen set containing valid atomic candidates. The atomic candidates must not be tuples.

Parameters

- **`vote_count_bounds`** (`Tuple[Optional[int], Optional[int]]`) – A tuple with lower and upper bounds (inclusive) for the number of candidates any vote can contain. None means the respective bound is not checked. Ignored if count_checker is given.
- **`count_checker`** (`Optional[VoteMagnitudeChecker]`) – A `VoteMagnitudeChecker` that checks the number of candidates any vote can contain.
- **`nominator`** (*`Nominator`*) – Nominator used to check candidates. The default uses the technical criteria specified by the `Candidate` class.

`validate`(*vote*)

Check if the approval vote is valid.

Parameters
`vote` (`FrozenSet[Union[str, CandidateObject]]`) – Approval vote to be checked.

Raises

- *`VoteTypeError`* – If the vote is not a frozen set.
- *`CandidateError`* – If any of the contained candidates is invalid.
- *`VoteMagnitudeError`* – If the number of candidates voted for is out of allowed bounds.

Return type
`None`

class `votelib.vote.`**`RankedVoteValidator`**(*total_vote_count_bounds=(None, None)*, *rank_vote_count_bounds=(1, 1)*, *total_count_checker=None*, *rank_vote_count_checkers=None*, *nominator=<votelib.candidate.BasicNominator object>*)

Validate a ranked vote (ranking of a number of candidates).

The vote must be a tuple of candidates or frozen sets thereof. Usage of sets indicates tied rankings, which are not common but allowed in some systems.

Parameters

- **`total_vote_count_bounds`** (`Tuple[Optional[int], Optional[int]]`) – A tuple with lower and upper bounds (inclusive) for the number of candidates any vote can rank. None means the respective bound is not checked. Ignored if total_count_checker is given.
- **`rank_vote_count_bounds`** (`Union[Tuple[Optional[int], Optional[int]], Dict[int, Tuple[Optional[int], Optional[int]]]]`) – A tuple with lower and upper bounds (inclusive) for the number of candidates allowed to share any rank. The default settings disallow tied rankings. None means the respective bound is not checked. Alternatively, a dictionary can be specified that maps ranks (1-indexed) to bound tuples. Ignored if rank_vote_count_checkers is given.
- **`total_count_checker`** (`Optional[VoteMagnitudeChecker]`) – A `VoteMagnitudeChecker` that checks the total number of candidates any vote can rank.

- **rank_vote_count_checkers** (`Optional[Dict[int, VoteMagnitudeChecker]]`) – A mapping of integer ranks to instances of `VoteMagnitudeChecker` to check numbers of candidates allowed to share any rank.
- **nominator** (*`Nominator`*) – Nominator used to check candidates. The default uses only technical criteria specified by the `Candidate` class.

validate(*vote*)

Check if the ranked vote is valid.

Parameters

vote (`Tuple[Union[str, CandidateObject, FrozenSet[Union[str, CandidateObject]]], ...]`) – Ranked vote to be checked.

Raises

- *`VoteTypeError`* – If the vote is not a tuple.
- *`VoteError`* – If any candidate is specified more than once in the ranking.
- *`CandidateError`* – If any of the contained candidates is invalid.
- *`VoteMagnitudeError`* – If the number of candidates (total or at particular ranking tier) is out of the specified bounds.

Return type

`None`

class votelib.vote.**EnumScoreVoteValidator**(*score_levels*, *allowed_scorings=(None, None)*, *sum_bounds=(None, None)*, *n_scorings_checker=None*, *sum_checkers=None*, *nominator=<votelib.candidate.BasicNominator object>*)

Validate an enumeration-based score vote.

An enumeration-based score vote assigns scores from a predefined finite set to candidates. It should be represented as a frozen set of doubles containing a candidate and the associated score. This variant is the most common for score-based voting systems since the voters are usually given a predefined finite set of (possibly non-numeric) scores; for true range voting where the voters might specify arbitrary score values, use *`RangeVoteValidator`*.

If numeric scores are used, the usage of exact numeric types (integers, fractions, decimals) is encouraged.

Parameters

- **score_levels** (`Collection[Any]`) – A collection of allowed scores.
- **allowed_scorings** (`Tuple[Optional[int], Optional[int]]`) – A tuple with lower and upper bounds (inclusive) for the number of candidates allowed to appear in any single vote. None means the respective bound is not checked. Ignored if n_scorings_checker is given.
- **sum_bounds** (`Union[Tuple[Optional[Number], Optional[Number]], Dict[int, Tuple[Optional[Number], Optional[Number]]]]`) – A tuple with lower and upper bounds (inclusive) for the total sum of numeric scores allowed for any single vote. None means the respective bound is not checked. You can also provide a dictionary that gives different sum bounds for different numbers of candidates scored (numbers of candidates scored that do not have a corresponding key will not have their sums checked then). Ignored if sum_checkers are given.
- **n_scorings_checker** (`Optional[VoteMagnitudeChecker]`) – A `VoteMagnitudeChecker` that checks the total number of candidates any vote can score.

- **sum_checkers** (Optional[Dict[int, VoteMagnitudeChecker]]) – A mapping of integer numbers of scored candidates to instances of VoteMagnitudeChecker checking the allowed total score allocated to all candidates.
- **nominator** (*Nominator*) – Nominator used to check candidates. The default uses only technical criteria specified by the Candidate class.

validate(*vote*)

Check if the enumeration-based score vote is valid.

Parameters
vote (FrozenSet[Tuple[Union[str, CandidateObject], Any]]) – Score vote to be checked.

Raises

- *VoteTypeError* – If the vote is not a frozen set.
- *CandidateError* – If any of the contained candidates is invalid.
- *VoteMagnitudeError* – If the number of candidates scored or the total sum of scores is out of the specified bounds, or any of the items in the vote is not a tuple pair.
- *VoteValueError* – If any of the scores is not in the predefined set of allowed scores.

Return type
bool

class votelib.vote.**RangeVoteValidator**(*range=(None, None)*, *allowed_scorings=(None, None)*, *sum_bounds=(None, None)*, *range_checker=None*, *n_scorings_checker=None*, *sum_checkers=None*, *nominator=<votelib.candidate.BasicNominator object>*)

Validate a range (non-enumerative score) vote.

An range vote assigns scores from a predefined interval to candidates. It should be represented as a frozen set of doubles containing a candidate and the associated score. This variant is mostly theoretical since most systems only allow choice from a finite set of scores; for that variant, use *EnumScoreVoteValidator*.

The usage of exact numeric types (integers, fractions, decimals) for scores is encouraged.

Parameters

- **range** (Tuple[Optional[Number], Optional[Number]]) – A tuple with lower and upper bounds (inclusive) for any single score value. None means the respective bound is not checked.
- **allowed_scorings** (Tuple[Optional[int], Optional[int]]) – A tuple with lower and upper bounds (inclusive) for the number of candidates allowed to appear in any single vote. None means the respective bound is not checked. Ignored if n_scorings_checker is given.
- **sum_bounds** (Union[Tuple[Optional[Number], Optional[Number]], Dict[int, Tuple[Optional[Number], Optional[Number]]]]) – A tuple with lower and upper bounds (inclusive) for the total sum of numeric scores allowed for any single vote. None means the respective bound is not checked. Ignored if sum_checkers are given.
- **n_scorings_checker** (Optional[VoteMagnitudeChecker]) – A VoteMagnitudeChecker that checks the total number of candidates any vote can score.
- **sum_checkers** (Optional[Dict[int, VoteMagnitudeChecker]]) – A mapping of integer numbers of scored candidates to instances of VoteMagnitudeChecker checking the allowed total score allocated to all candidates.

- **nominator** (*Nominator*) – Nominator used to check candidates. The default uses only technical criteria specified by the `Candidate` class.

validate(*vote*)

Check if the range vote is valid.

Parameters

vote (`FrozenSet[Tuple[Union[str, CandidateObject], Any]]`) – Range (score) vote to be checked.

Raises

- *VoteTypeError* – If the vote is not a frozen set.
- *CandidateError* – If any of the contained candidates is invalid.
- *VoteMagnitudeError* – If the number of candidates scored, the total sum of scores or any single score is out of the specified bounds, or any of the items in the vote is not a tuple pair.

Return type

`bool`

An abstract class defining the vote validator interface is also present.

class votelib.vote.**VoteValidator**

Validate that a single vote is valid under the election rules.

Base class, not intended for direct use.

abstract validate(*vote*)

Check if the vote satisfies criteria given by the voting system.

Raises

NotImplementedError –

Return type

`None`

1.6.2 Vote validation errors

class votelib.vote.**VoteError**

A vote is invalid given the election rules.

class votelib.vote.**VoteTypeError**(*vtype*, *expected=None*)

A vote is of an invalid type.

E.g. ranked votes in place of simple votes.

Parameters

- **vtype** (`type`) – Vote type detected as invalid.
- **expected** (`Optional[type]`) – Vote type that was expected.

class votelib.vote.**VoteMagnitudeError**(*value*, *min_value=None*, *max_value=None*, *value_name='count'*)

A vote is too small or too large.

Parameters

- **value** (`Number`) – Size of the vote that was found to be invalid.
- **min_value** (`Optional[Number]`) – Minimum value permissible in the context.

- **max_value** (Optional[Number]) – Maximum value permissible in the context.
- **value_name** (str) – Role of the vote size (e.g. number of approval votes, number of ranked candidates. . .)

class votelib.vote.**VoteValueError**(*value*, *candidate=None*, *allowed=None*)

An explicitly given vote value is invalid.

Parameters

- **value** (Any) – Value of the vote that is invalid.
- **candidate** (Union[str, CandidateObject, None]) – A candidate that the vote was given for. If None, a specific candidate could not be pinpointed.
- **allowed** (Optional[Any]) – A spectrum of values that is allowed at the given point.

1.7 Components API

1.7.1 Quotas

Quota functions used in largest-remainder proportional voting systems.

Quotas can however be used as a component in many other voting systems, such as transferable vote or open list evaluators.

A quota function takes the total number of votes and the number of seats to allocate and returns the number of votes required to reach a seat. The unrounded quota functions return fractions to retain exact values.

All supported quota functions are assembled in the *QUOTAS* dictionary keyed by their name. *get()* retrieves from this dictionary by string key; *construct()* also accepts callables and passes them through.

votelib.component.quota.**droop**(*votes*, *seats*)

Droop quota, the most widely used one.

This it is the smallest integer quota guaranteeing the number of passing candidates will not be higher than the number of seats.

Return type
int

votelib.component.quota.**hagenbach_bischoff**(*votes*, *seats*)

Hagenbach-Bischoff quota.

This is the unrounded variant, giving the exact fraction.

Return type
Fraction

votelib.component.quota.**hagenbach_bischoff_ceil**(*votes*, *seats*)

Hagenbach-Bischoff quota, rounding up.

This is the rounded variant that is identical to the Droop quota in most cases.

Return type
int

votelib.component.quota.**hagenbach_bischoff_rounded**(*votes*, *seats*)

Hagenbach-Bischoff quota, rounding mathematically (round-to-even).

This is the rounded variant that is used in some rare cases, e.g. old Slovak regional apportionment of parliamentary seats. Half is rounded up.

Return type
`int`

votelib.component.quota.**hare**(*votes*, *seats*)

Hare quota, the most basic one.

This is the unrounded variant, giving the exact fraction.

Return type
`Fraction`

votelib.component.quota.**hare_rounded**(*votes*, *seats*)

Hare quota, the most basic one.

This is the rounded variant, which is used more often. Half is rounded up.

Return type
`int`

votelib.component.quota.**imperiali**(*votes*, *seats*)

Imperiali quota.

Imperiali quota can produce more candidates than seats to be filled in some cases; the results then usually need to be recalculated using a different quota.

Return type
`Fraction`

1.7.2 Divisors

Divisor functions used in highest-averages proportional voting systems.

This provides arguments for the *votelib.evaluate.proportional.HighestAverages* evaluator. However, divisors can be used as a component in many other voting systems, such as biproportional allocation.

A divisor function takes the order number (usually equal to the number of seats allocated so far) and returns the divisor by which to divide the number of votes for the given party or candidate. The party or candidate with the largest result then gets the next seat.

Some systems use a mathematically defined divisor but artificially change the result for parties with no seats so far (*order == 0*) to make it harder for parties to get their seats. Use *modified_first_coef()* for that.

All supported divisor functions are assembled in the *DIVISORS* dictionary keyed by their name. *get()* retrieves from this dictionary by string key; *construct()* also accepts callables and passes them through.

votelib.component.divisor.**d_hondt**(*order*)

D'Hondt divisor, the most commonly used divisor.

Forms a simple sequence 1, 2, 3… In the United States, this is known as the Jefferson divisor that was used for congressional apportionment 1792-1842.

Known to slightly favor larger parties.

Return type
`int`

`votelib.component.divisor.`**`danish`**`(`*order*`)`

Danish divisor.

Forms a sequence 1, 4, 7…

Extremely favors smaller parties.

Return type
`int`

`votelib.component.divisor.`**`huntington_hill`**`(`*order*`)`

Huntington-Hill divisor.

The divisors are defined by *sqrt(n*(n+1))*. This means the divisor is invalid for the zeroth order (passing a zero gives a divisor of zero, which raises an error in the subsequent division) and can thus only be used in cases where the first seat is already guaranteed (use the *prev_gains* argument for that).

Used for United States congressional apportionment as of 2020.

Return type
`Decimal`

`votelib.component.divisor.`**`imperiali`**`(`*order*`)`

Imperiali divisor. Not to be confused with the Imperiali quota.

Forms a sequence 1, 1.5, 2…

Known to favor large parties greatly.

Return type
`Fraction`

`votelib.component.divisor.`**`macau`**`(`*order*`)`

Macau modified D'Hondt divisor.

This uses order counts as exponents (1, 2, 4, 8…), and therefore favors smaller parties.

Return type
`int`

`votelib.component.divisor.`**`modified_first_coef`**`(`*divisor_fx*, *first_coef=Decimal('1.4')*`)`

Modify the divisor for the zeroth order to an apriori coefficient.

This can be used to raise the threshold for parties that have not yet obtained a seat. This is the case in the Czech regional election (Koudelka coefficient 1.42), Nepal, Norway, and Sweden.

Parameters

- **`divisor_fx`** (`Callable`[[`int`], `Number`]) – The ordinary divisor function to be wrapped and used for the first order and subsequent ones.
- **`first_coef`** (`Decimal`) – The coefficient to be used when order == 0.

Return type
`Callable`[[`int`], `Number`]

`votelib.component.divisor.`**`sainte_lague`**`(`*order*`)`

Sainte-Laguë (Webster, Schepers) divisor, a commonly used divisor.

Forms a sequence 1, 3, 5…

Known to favor mid-sized parties.

Return type
`int`

1.7.3 Pairwise win scorers

Functions to score magnitudes of wins between pairs of candidates.

These are used in some Condorcet methods to determine ranking priority.

`votelib.component.pairwin_scorer.`**`margins`**`(`*counts*`)`

Margins pairwise win scorer. Takes the difference from reverse option.

Also called margin of victory or defeat strength. Assigns the number of votes ranking the pair in the given order minus the number of votes doing the reverse as the win strength (which is thus negative for pairwise losses).

Parameters

`counts` (`Dict[Tuple[Union[str, CandidateObject], Union[str, CandidateObject]], Number]`) – Condorcet votes (counts of pairwise preferences).

Return type

`Dict[Tuple[Union[str, CandidateObject], Union[str, CandidateObject]], Number]`

`votelib.component.pairwin_scorer.`**`pairwise_opposition`**`(`*counts*`)`

Pairwise opposition win scorer. Returns the win counts unchanged.

This gives the number of votes ranking the pair in the given order directly as the measure of pairwise win, regardless of the number of votes preferring the opposite pairwise ranking.

Parameters

`counts` (`Dict[Tuple[Union[str, CandidateObject], Union[str, CandidateObject]], Number]`) – Condorcet votes (counts of pairwise preferences).

Return type

`Dict[Tuple[Union[str, CandidateObject], Union[str, CandidateObject]], Number]`

`votelib.component.pairwin_scorer.`**`winning_votes`**`(`*counts*`)`

Winning votes pairwise win scorer. Counts wins fully, zero otherwise.

This is the most common pairwise win scorer. When the number of votes for the pair ranked in one direction is larger than the other direction, assigns all those votes as the pairwise win strength.

Parameters

`counts` (`Dict[Tuple[Union[str, CandidateObject], Union[str, CandidateObject]], Number]`) – Condorcet votes (counts of pairwise preferences).

Return type

`Dict[Tuple[Union[str, CandidateObject], Union[str, CandidateObject]], Number]`

1.7.4 Rank scorers

Objects to assign scores to ranks in ranked voting systems such as Borda.

A rank scorer returns a list of numerical scores to be assigned to ranks given by voters. This is the essence of Borda count system, and rank scorers capture most of the variations there are in that system.

`class` `votelib.component.rankscore.`**`Borda`**`(`*base=1*`)`

Borda rank scorer, corresponding to the original Borda count variant.

Assigns the *base* score to the candidate ranked last, and one point more for each higher rank.

This rank scorer needs to be initialized by the `set_n_candidates()` before calling `scores()`.

Parameters
`base` (`int`) – The score to assign to the candidate ranked last. For the truly original Borda, this equals to 1; some variants set it to zero, and thus set the score for the first rank to the number of candidates minus one.

`scores`(*n_ranked*)

Return the scores for the first n_ranked ranks.

This gives (number of candidates + base - 1 - rank) for ranks running from 0 (best rank) to n_ranked.

Parameters
`n_ranked` (`int`) – Number of ranks to be returned. Equal to the length of the output list.

Raises
`RuntimeError` – If the scorer has not been initialized first by calling `set_n_candidates()`.

Return type
`List[int]`

`set_n_candidates`(*n_candidates*)

Set the total number of candidates that could be ranked.

This helps to account for rankings that do not rank all candidates.

Parameters
`n_candidates` (`int`) – The total number of candidates that could be ranked on any ballot (i.e. the number of candidates participating in the election in the particular constituency).

Return type
`None`

`class votelib.component.rankscore.Dowdall`

Dowdall (Nauru) rank scorer.

Assigns the numbers of the harmonic series (1, 1/2, 1/3…) to progressively lower ranks.

`scores`(*n_ranked*)

Return the scores for the first n_ranked ranks.

This gives *1 / (rank + 1)* for ranks running from 0 (best rank) to n_ranked.

Parameters
`n_ranked` (`int`) – Number of ranks to be returned. Equal to the length of the output list.

Return type
`List[Fraction]`

`class votelib.component.rankscore.FixedTop`(*top*)

A rank scorer with fixed score for the top rank.

Assigns scores progressively decreased by one until hitting zero.

Parameters
`top` (`int`) – The score for the top (best) ranked candidate on the ballot.

`scores`(*n_ranked*)

Return the scores for the first n_ranked ranks.

This gives *max(top - rank, 0)* for ranks running from 0 (best rank) to n_ranked.

Parameters
`n_ranked` (`int`) – Number of ranks to be returned. Equal to the length of the output list.

Return type
`List[int]`

class `votelib.component.rankscore.`**`Geometric`**(*base=2*)

A geometric progression rank scorer.

Assigns the numbers of a chosen inverse geometric progression (e.g. 1, 1/2, 1/4… for 2) to progressively lower ranks.

Parameters
`base` (`int`) – Base of the geometric progression.

`scores`(*n_ranked*)

Return the scores for the first n_ranked ranks.

This gives *1 / (2 ** rank)* for ranks running from 0 (best rank) to n_ranked.

Parameters
`n_ranked` (`int`) – Number of ranks to be returned. Equal to the length of the output list.

Return type
`List[Fraction]`

class `votelib.component.rankscore.`**`ModifiedBorda`**

Modified Borda count rank scorer.

In this system, the score for the highest rank is not constant (as is the case for the vanilla Borda count), but is equal to the number of ranked candidates; therefore, it encourages voters to rank many candidates.

`scores`(*n_ranked*)

Return the scores for the first n_ranked ranks.

This gives *n_ranked - rank* for ranks running from 0 (best rank) to n_ranked.

Parameters
`n_ranked` (`int`) – Number of ranks to be returned. Equal to the length of the output list.

Return type
`List[int]`

class `votelib.component.rankscore.`**`RankScorer`**

An abstract base class for rank scorers.

Rank scorers must provide a *scores()* method that returns a list of scores based on the number of ranks given. They may also provide a *set_n_candidates()* method that sets the total number of candidates participating in the election, which might be relevant for computing the rank scores. If this method is defined, it must be called before the *scores()* method is called first.

class `votelib.component.rankscore.`**`SequenceBased`**(*sequence*)

A rank scorer with a predetermined sequence of scores.

This is used in many competitions (Eurovision, Formula One, etc.)

Assigns scores according to the given sequence until hitting zero, and zero scores afterwards.

Parameters
`sequence` (`List[Number]`) – The scores for the top candidates on the ballot.

`scores`(*n_ranked*)

Return the scores for the first n_ranked ranks.

This gives values from the initial sequence, then zeros.

Parameters
`n_ranked` (`int`) – Number of ranks to be returned. Equal to the length of the output list.

Return type
`List[Number]`

`votelib.component.rankscore.select_padded`(*sequence*, *n*, *pad_with=0*)

Select n leading elements from sequence, padding with pad_with.

Padding with pad_with is used when sequence is not long enough to select n elements.

Return type
`List[Any]`

1.7.5 Vote transferers for STV

Objects to transfer votes between candidates for transferable vote systems.

Transfers of votes from eliminated and elected candidates to candidates staying in the contest are an essential part of any transferable vote system, such as the one of `votelib.evaluate.sequential.TransferableVoteSelector`. There are many variants how to achieve this, some are implemented here.

NOTE: The `VoteTransferer` interface is provisional and may be changed in the future, chiefly to accommodate a wider range of transfer methods (Meek, Wright…)

`class votelib.component.transfer.Gregory`

Gregory (fractional) vote transferer.

The variant used is the Weighted Inclusive Gregory Method (WIGM) used e.g. in Scottish local government elections.

When a candidate is elected by quota, the fraction corresponding to the quota divided by total votes for the candidate is used to multiply (lower) each vote allocated to that candidate; the votes are then transferred to next candidates on the ballots.

In case of shared ranks, the votes are evenly distributed between the candidates sharing the rank.

The implementation produces exact fractional votes. Rounding rules are not implemented yet.

`subtract`(*allocation*, *elected*)

Remove votes from elected candidates according to the quota.

Parameters

- **`allocation`** (`Dict[Union[str, CandidateObject, None], Dict[Tuple[Union[str, CandidateObject, FrozenSet[Union[str, CandidateObject]]], ...], Number]]`) – Current allocation of ranked votes to candidates. The votes allocated to elected candidates will be lowered by the amount corresponding to the quota for their election.
- **`elected`** (`Dict[Union[str, CandidateObject], int]`) – Elected candidates, mapped to the quota with which they were elected (or multiples thereof, if they were awarded multiple seats). These quotas should be removed from the candidates' votes (because so many votes were used).

Return type
`Dict[Union[str, CandidateObject, None], Dict[Tuple[Union[str, CandidateObject, FrozenSet[Union[str, CandidateObject]]], ...], Number]]`

`transfer`(*allocation*, *candidates*)

Transfer votes from eliminated or fully elected candidates.

Parameters

- `allocation` (`Dict[Union[str, CandidateObject, None], Dict[Tuple[Union[str, CandidateObject, FrozenSet[Union[str, CandidateObject]]], ...], Number]]`) – Current allocation of ranked votes to candidates. The votes allocated to specified candidates will be reallocated to other candidates that are present in the allocation, or reassigned to the None key as exhausted ballots.
- `candidates` (`List[Union[str, CandidateObject]]`) – Candidates to be removed from the allocation.

Return type

`Dict[Union[str, CandidateObject, None], Dict[Tuple[Union[str, CandidateObject, FrozenSet[Union[str, CandidateObject]]], ...], Number]]`

`class votelib.component.transfer.Hare(`*seed=None*`)`

Hare (random ballot selection) vote transferer.

This is the variant used (AFAIK) in Irish lower house legislative elections (Dáil Éireann).

When a candidate is elected by quota, the number of ballots corresponding to the quota is randomly selected and discarded, the rest is fully transferred to their next preferences.

In case of shared ranks, the votes are randomly distributed between the candidates sharing the rank.

Parameters

`seed` (`Optional[int]`) – Seed for the random generator.

`subtract(`*allocation*, *elected*`)`

Remove votes from elected candidates according to the quota.

Parameters

- `allocation` (`Dict[Union[str, CandidateObject, None], Dict[Tuple[Union[str, CandidateObject, FrozenSet[Union[str, CandidateObject]]], ...], Number]]`) – Current allocation of ranked votes to candidates. The votes allocated to elected candidates will be lowered by the amount corresponding to the quota for their election.
- `elected` (`Dict[Union[str, CandidateObject], int]`) – Elected candidates, mapped to the quota with which they were elected (or multiples thereof, if they were awarded multiple seats). These quotas should be removed from the candidates' votes (because so many votes were used).

Return type

`Dict[Union[str, CandidateObject, None], Dict[Tuple[Union[str, CandidateObject, FrozenSet[Union[str, CandidateObject]]], ...], Number]]`

`transfer(`*allocation*, *candidates*`)`

Transfer votes from eliminated or fully elected candidates.

Parameters

- `allocation` (`Dict[Union[str, CandidateObject, None], Dict[Tuple[Union[str, CandidateObject, FrozenSet[Union[str, CandidateObject]]], ...], Number]]`) – Current allocation of ranked votes to candidates. The votes allocated to specified candidates will be reallocated to other candidates that are present in the allocation, or reassigned to the None key as exhausted ballots.
- `candidates` (`List[Union[str, CandidateObject]]`) – Candidates to be removed from the allocation.

Return type
`Dict[Union[str, CandidateObject, None], Dict[Tuple[Union[str, CandidateObject, FrozenSet[Union[str, CandidateObject]]], ...], Number]]`

1.8 Evaluator persistence API

`votelib.persist.`**`from_dict`**(*value*)

Parse an election evaluator object from a JSON-like dictionary.

Parameters
`value` (`Dict[str, Any]`) – A dictionary created by *`to_dict()`*.

Return type
`Any`

`votelib.persist.`**`to_dict`**(*obj*)

Serialize an election evaluator object to a JSON-ready dictionary.

Parameters
`obj` (`Any`) – An election evaluator object or similar. It should provide a *to_dict()* method (all the standard evaluators, converters and the like from Votelib should have it, courtesy of the simple_serialization decorator).

Return type
`Dict[str, Any]`

CHAPTER

TWO

EXAMPLES OF USING VOTELIB

This page groups links to example use cases of votelib.

2.1 Slovak 2020 parliamentary elections

A simple example: evaluating the result of the 2020 parliamentary elections in the Central European country of Slovakia. We choose it because its electoral system is quite simple, as is described below, yet it is not trivial to get the result.

[1]:
```
import sys
import os
import csv
import decimal

sys.path.append(os.path.join('..', '..'))
import votelib.candidate
import votelib.evaluate.core
import votelib.evaluate.threshold
import votelib.evaluate.proportional
```

2.1.1 Evaluator construction

First, we construct the evaluator.

All of Slovakia forms a single constituency so the votes may be summed over the whole country and there is a single evaluation taking place.

Slovakia uses the Hagenbach-Bischoff largest remainder system (rounded mathematically) to allocate the 150 seats in its National Council to parties. The candidates elected are determined through open lists, but we will not go into that detail here. The evaluator is simple to construct:

[2]:
```
core_evaluator = votelib.evaluate.proportional.LargestRemainder(
    'hagenbach_bischoff_rounded'
)
```

Slovakia also uses a minimum vote threshold (as a fraction of national level votes) to exclude small parties. The threshold is 5% for single parties, 7% for two- and three-member coalitions and 10% for four- and more member coalitions. We thus need to construct a pre-selection evaluator that selects only parties over the threshold:

[3]:
```
# using decimals to be precise
standard_elim = votelib.evaluate.threshold.RelativeThreshold(
    decimal.Decimal('.05'), accept_equal=True # reaching the exact count is sufficient
)
mem_2_3_elim = votelib.evaluate.threshold.RelativeThreshold(
    decimal.Decimal('.07'), accept_equal=True
)
mem_4plus_elim = votelib.evaluate.threshold.RelativeThreshold(
    decimal.Decimal('.1'), accept_equal=True
)
# combine the individual evaluators into a dispatcher based on coalition member count
preselector = votelib.evaluate.threshold.CoalitionMemberBracketer(
    {1: standard_elim, 2: mem_2_3_elim, 3: mem_2_3_elim},
    default=mem_4plus_elim
)
```

Now, we set up the pre-selector to be a condition for entering the evaluation (all votes for other parties will be simply discarded) and fix the number of seats (the Slovak National Council has 150 seats):

[4]:
```
evaluator = votelib.evaluate.core.FixedSeatCount(
    votelib.evaluate.core.Conditioned(preselector, core_evaluator), 150
)
```

2.1.2 Vote loading

Now we load the vote counts. Since all of Slovakia forms a single constituency the votes are already summed over the whole country.

[5]:
```
fpath = os.path.join('..', '..', 'tests', 'real', 'data', 'sk_nr_2020.csv')
with open(fpath, encoding='utf8') as infile:
    rows = list(csv.reader(infile, delimiter=';'))
    party_names, coalition_flags, votes, seats = [list(x) for x in zip(*rows)]
print(votes)
```

```
['721166', '527172', '237531', '229660', '200780', '179246', '166325', '134099', '112662
↪', '91171', '88220', '84507', '59174', '15925', '9260', '8191', '4194', '3296', '2018',
↪ '1966', '1887', '1261', '991', '809']
```

Since the `CoalitionMemberBracketer` needs `ElectionParty` objects as candidates to determine coalition sizes, we will need to construct those from the party names and coalition flags (1 if the election party is a coalition).

[6]:
```
parties = [
    votelib.candidate.Coalition(name=name, parties=[
        votelib.candidate.PoliticalParty(pname)
        for pname in name.split('-')
    ])
    if int(coalflag) else votelib.candidate.PoliticalParty(name)
    for name, coalflag in zip(party_names, coalition_flags)
]
```

Now, we assemble the votes into a dictionary to be accepted by the evaluator.

[7]:
```
votes = dict(zip(parties, [int(v) for v in votes]))
votes
```

[7]:
```
{<PoliticalParty(OĽANO)>: 721166,
 <PoliticalParty(Smer)>: 527172,
 <PoliticalParty(Sme rodina)>: 237531,
 <PoliticalParty(ĽSNS)>: 229660,
 Coalition([<PoliticalParty(Progresívne Slovensko)>, <PoliticalParty(SPOLU)>]): 200780,
 <PoliticalParty(Sloboda a Solidarita)>: 179246,
 <PoliticalParty(Za ľudí)>: 166325,
 <PoliticalParty(Kresťanskodemokratické hnutie)>: 134099,
 <PoliticalParty(MKÖ-MKS)>: 112662,
 <PoliticalParty(SNS)>: 91171,
 <PoliticalParty(Dobrá voľba)>: 88220,
 <PoliticalParty(Vlasť)>: 84507,
 <PoliticalParty(Most-Híd)>: 59174,
 <PoliticalParty(Socialisti.sk)>: 15925,
 <PoliticalParty(Máme toho dosť!)>: 9260,
 <PoliticalParty(Slovenská ľudová strana Andreja Hlinku)>: 8191,
 <PoliticalParty(Demokratická strana)>: 4194,
 <PoliticalParty(Solidarita-HPCh)>: 3296,
 <PoliticalParty(STAROSTOVIA A NEZÁVISLÍ KANDIDÁTI)>: 2018,
 <PoliticalParty(Slovenské Hnutie Obrody)>: 1966,
 <PoliticalParty(Hlas ľudu)>: 1887,
 <PoliticalParty(Práca slovenského národa)>: 1261,
 <PoliticalParty(99 % - občiansky hlas)>: 991,
 <PoliticalParty(Slovenská liga)>: 809}
```

2.1.3 Performing the evaluation

When the evaluator is set up correctly, obtaining the result is simple.

[8]:
```
evaluated = evaluator.evaluate(votes)
for party, mandates in evaluated.items():
    print(party.name.ljust(30), mandates)
```

```
OĽANO                          53
Smer                           38
Sme rodina                     17
ĽSNS                           17
Sloboda a Solidarita           13
Za ľudí                        12
```

We see that six parties were elected; although *PS-Spolu* had more votes than either *SaS* or *Za ľudí*, it was excluded by the higher threshold for coalitions.

2.2 Czech 2017 parliamentary elections

This example shows proportional elections in the presence of multiple constituencies, where seats are allocated to constituencies on-line by the number of votes cast in them.

[1]:
```
import sys
import os
import csv
import decimal

sys.path.append(os.path.join('..', '..'))
import votelib.candidate
import votelib.evaluate.core
import votelib.evaluate.threshold
import votelib.evaluate.proportional
```

2.2.1 Evaluator construction

First, we construct the evaluator.

Czechia uses the d'Hondt highest averages proportional system in fourteen regions that form separate constituencies with separate regional lists. There is a national 5% electoral threshold.

[2]:
```
eliminator = votelib.evaluate.threshold.RelativeThreshold(
    decimal.Decimal('.05'), accept_equal=True
)
regional_evaluator = votelib.evaluate.proportional.HighestAverages('d_hondt')
```

The number of seats for each region is determined by the total number of votes cast in the region by the Hare quota from a total of 200 seats. The candidates elected are determined through open lists, but we will not go into that detail here.

[3]:
```
apportioner = votelib.evaluate.proportional.LargestRemainder('hare')
combined_evaluator = votelib.evaluate.core.ByConstituency(
    regional_evaluator, apportioner, preselector=eliminator
)
evaluator = votelib.evaluate.core.FixedSeatCount(
    votelib.evaluate.core.PostConverted(
        combined_evaluator,
        votelib.convert.VoteTotals(),
    ),
    200
)
```

2.2.2 Vote loading

Now we load the vote counts, per region.

[4]:
```
fpath = os.path.join('..', '..', 'tests', 'real', 'data', 'cz_psp_2017.csv')
with open(fpath, encoding='utf8') as infile:
    rows = list(csv.reader(infile, delimiter=';'))
region_names = rows[0][1:]
votes = {region: {} for region in region_names}
for row in rows[1:]:
    party = row[0]
    for regname, n_votes in zip(region_names, row[1:]):
        votes[regname][party] = int(n_votes)
print(dict(sorted(votes['Královéhradecký kraj'].items(), key=lambda x: x[1],␣
↪reverse=True)))
```

```
{'ANO': 88551, 'ODS': 32242, 'Piráti': 29932, 'SPD': 28038, 'KSČM': 19792, 'ČSSD': 18128,
↪ 'KDU-ČSL': 16294, 'TOP 09': 14308, 'STAN': 14184, 'Svobodní': 4310, 'Zelení': 3650,
↪'Rozumní': 2388, 'REAL': 2037, 'SPO': 851, 'SPRRSČ M.Sládka': 619, 'DSSS': 600, 'ŘN -␣
↪VU': 531, 'SPORTOVCI': 431, 'DV 2016': 380, 'ODA': 322, 'CESTA': 297, 'BPI': 284,
↪'Referendum o EU': 276, 'RČ': 275, 'PB': 0, 'SPDV': 0, 'OBČANÉ 2011': 0, 'Unie H.A.V.E.
↪L.': 0, 'ČNF': 0, 'ČSNS': 0, 'NáS': 0}
```

2.2.3 Performing the evaluation

When the evaluator is set up correctly, obtaining the result is simple.

[5]:
```
print(evaluator.evaluate(votes))
```

```
{'ANO': 78, 'Piráti': 22, 'ODS': 25, 'TOP 09': 7, 'SPD': 22, 'ČSSD': 15, 'STAN': 6, 'KDU-
↪ČSL': 10, 'KSČM': 15}
```

We can also check the results for individual regions by taking the outputs of `combined_evaluator`.

[6]:
```
print(combined_evaluator.evaluate(votes, 200)['Královéhradecký kraj'])
```

```
{'ANO': 5, 'ODS': 1, 'Piráti': 1, 'SPD': 1, 'KSČM': 1, 'ČSSD': 1, 'KDU-ČSL': 1}
```

2.2.4 Alternative systems

In this section, we examine the results that would be achieved under different alternative voting systems.

Sainte-Laguë divisor

Using the Sainte-Laguë divisor helps smaller and mid-sized parties that d'Hondt tends to disfavor; however, the small size of some regions will still impact them. We construct the evaluator analogously, just using the different divisor.

[7]:
```
sainte_lague_evaluator = votelib.evaluate.core.PostConverted(
    votelib.evaluate.core.ByConstituency(
        votelib.evaluate.proportional.HighestAverages('sainte_lague'),
        apportioner,
        preselector=eliminator
```

(continues on next page)

(continued from previous page)

```
    ),
    votelib.convert.VoteTotals(),
)
print(sainte_lague_evaluator.evaluate(votes, 200))
```

```
{'ANO': 63, 'Piráti': 22, 'ODS': 24, 'TOP 09': 11, 'SPD': 23, 'ČSSD': 16, 'STAN': 14,
↪'KDU-ČSL': 11, 'KSČM': 16}
```

Abolishing the electoral threshold

If we abolish the electoral threshold and keep the d'Hondt divisor, nothing happens in this particular case - no party apart from those over 5% is strong enough to get hold of a single seat. (Under Sainte-Laguë, Zelení and Svobodní would each get a single seat.)

```
[8]: no_threshold_evaluator = votelib.evaluate.core.PostConverted(
    votelib.evaluate.core.ByConstituency(
        votelib.evaluate.proportional.HighestAverages('d_hondt'),
        apportioner,
    ),
    votelib.convert.VoteTotals(),
)
print(no_threshold_evaluator.evaluate(votes, 200))
```

```
{'ANO': 78, 'Piráti': 22, 'ODS': 25, 'TOP 09': 7, 'SPD': 22, 'ČSSD': 15, 'STAN': 6, 'KDU-
↪ČSL': 10, 'KSČM': 15}
```

Biproportional apportionment

Applying biproportional apportionment results in a similar result to applying the Sainte-Laguë divisor in this case even with the d'Hondt divisor, since we honor nationwide party vote fractions.

Since `BiproportionalEvaluator` removes the need for `ByConstituency`, if we want to keep the electoral threshold, we need to introduce it using a separate `Conditioned` composite .

```
[9]: biprop_evaluator = votelib.evaluate.core.PostConverted(
    votelib.evaluate.Conditioned(
        votelib.evaluate.PreConverted(votelib.convert.VoteTotals(), eliminator),
        votelib.evaluate.proportional.BiproportionalEvaluator('d_hondt'),
        # depth=1 because the votes are nested by region and we want to subset parties
        subsetter=votelib.convert.SubsettedVotes(depth=1),
    ),
    votelib.convert.VoteTotals(),
)
print(biprop_evaluator.evaluate(votes, 200))
```

```
{'ANO': 64, 'ODS': 24, 'Piráti': 23, 'SPD': 23, 'KSČM': 17, 'ČSSD': 15, 'KDU-ČSL': 12,
↪'TOP 09': 11, 'STAN': 11}
```

2.3 Czech 2021 parliamentary elections

This example shows a complicated multi-constituency proportional system that was newly introduced the same year.

[1]:
```
%load_ext autoreload
%autoreload 2
```

[2]:
```
import sys
import os
import csv
import decimal

sys.path.append(os.path.join('..', '..'))
import votelib.candidate
import votelib.evaluate.core
import votelib.evaluate.threshold
import votelib.evaluate.proportional
```

2.3.1 Evaluator construction

First, we construct the evaluator.

The country is divided into fourteen regions that form separate constituencies with separate regional lists. The number of seats distributed within each region is determined from the national total of 200 seats by the Hare quota computed on the national vote count total. There is also a national 5% electoral threshold.

[3]:
```
eliminator = votelib.evaluate.threshold.RelativeThreshold(
    decimal.Decimal('.05'), accept_equal=True
)
apportioner = votelib.evaluate.proportional.LargestRemainder('hare')
```

The evaluator has two stages. In the first stage, seats are allocated within regions using the Imperiali quota. The awarded seat counts are rounded down - there is no largest remainder closeup, and so not all seats are awarded; these are transferred to the second stage.

[64]:
```
regional_evaluator = votelib.evaluate.proportional.QuotaDistributor('imperiali', on_
↪overaward='subtract')

stage1_evaluator = votelib.evaluate.core.ByConstituency(regional_evaluator)
```

In the second stage, the votes "unused" in the regional stage are summed across regions and the remaining seats allocated according to them using the Droop quota, now using the largest remainder rule to distribute all seats. The second stage seats are redistributed back to the regions according to the regional remainders and the candidates elected are determined through open lists, but we will not go into that detail here.

[52]:
```
stage2_evaluator = votelib.evaluate.core.RemovedApportionment(
    votelib.evaluate.core.ByParty(
        votelib.evaluate.proportional.LargestRemainder('droop')
    )
)

evaluator = votelib.evaluate.core.FixedSeatCount(
```

(continues on next page)

(continued from previous page)

```
    votelib.evaluate.core.PreApportioned(
        evaluator=votelib.evaluate.core.Conditioned(
            evaluator=votelib.evaluate.core.UnusedVotesDistributor(
                [stage1_evaluator, stage2_evaluator],
                quota_functions=[regional_evaluator.quota_function],
                depth=2
            ),
            eliminator=eliminator,
            depth=2,
        ),
        apportioner=apportioner,
    ),
    200
)

country_evaluator = votelib.evaluate.core.PostConverted(evaluator, votelib.convert.
↪MergedDistributions())
```

2.3.2 Vote loading

Now we load the vote counts, per region.

```
[6]: fpath = os.path.join('..', '..', 'tests', 'real', 'data', 'cz_psp_2021.csv')
     with open(fpath, encoding='utf8') as infile:
         rows = list(csv.reader(infile, delimiter=';'))
     region_names = rows[0][1:]
     votes = {region: {} for region in region_names}
     for row in rows[1:]:
         party = row[0]
         for regname, n_votes in zip(region_names, row[1:]):
             votes[regname][party] = int(n_votes)
     print(dict(sorted(votes['Královéhradecký kraj'].items(), key=lambda x: x[1],␣
     ↪reverse=True)))
```

```
{'SPOLU': 84166, 'ANO': 79463, 'Piráti+STAN': 44551, 'SPD': 26661, 'ČSSD': 14534,
↪'PŘÍSAHA': 13930, 'KSČM': 10150, 'TSS': 8663, 'Volný blok': 3941, 'Zelení': 3185, 'OtČe
↪': 1736, 'Švýcar. demokr.': 981, 'APB': 692, 'PRAMENY': 621, 'Monarchiste.cz': 438,
↪'Nevolte Urza.cz': 430, 'ANS': 273, 'PB': 0, 'Levice': 0, 'SENIOŘI': 0, 'MZH': 0,
↪'Moravané': 0}
```

2.3.3 Performing the evaluation

When the evaluator is set up correctly, obtaining the result is simple.

```
[65]: evaluator.evaluate(votes)
```

```
[65]: {'Středočeský kraj': {'SPD': 2, 'SPOLU': 10, 'Piráti+STAN': 6, 'ANO': 8},
       'Jihomoravský kraj': {'SPD': 2, 'SPOLU': 9, 'Piráti+STAN': 4, 'ANO': 8},
       'Olomoucký kraj': {'SPD': 1, 'SPOLU': 4, 'Piráti+STAN': 2, 'ANO': 5},
       'Karlovarský kraj': {'SPD': 1, 'SPOLU': 1, 'Piráti+STAN': 1, 'ANO': 2},
       'Moravskoslezský kraj': {'SPD': 3, 'SPOLU': 6, 'Piráti+STAN': 3, 'ANO': 10},
```

(continues on next page)

(continued from previous page)

```
 'Kraj Vysočina': {'SPD': 1, 'SPOLU': 4, 'Piráti+STAN': 1, 'ANO': 4},
 'Plzeňský kraj': {'SPD': 1, 'SPOLU': 4, 'Piráti+STAN': 2, 'ANO': 4},
 'Pardubický kraj': {'SPD': 1, 'SPOLU': 4, 'Piráti+STAN': 2, 'ANO': 3},
 'Liberecký kraj': {'SPD': 1, 'SPOLU': 2, 'Piráti+STAN': 2, 'ANO': 3},
 'Jihočeský kraj': {'SPD': 1, 'SPOLU': 5, 'Piráti+STAN': 2, 'ANO': 5},
 'Ústecký kraj': {'SPD': 2, 'SPOLU': 3, 'Piráti+STAN': 2, 'ANO': 7},
 'Zlínský kraj': {'SPD': 2, 'SPOLU': 4, 'Piráti+STAN': 2, 'ANO': 4},
 'Královéhradecký kraj': {'SPD': 1, 'SPOLU': 4, 'Piráti+STAN': 2, 'ANO': 4},
 'Hlavní město Praha': {'SPD': 1, 'SPOLU': 11, 'Piráti+STAN': 6, 'ANO': 5}}
```

[66]:
```
print(country_evaluator.evaluate(votes))
```

```
{'SPD': 20, 'SPOLU': 71, 'Piráti+STAN': 37, 'ANO': 72}
```

2.4 German 2017 federal legislative (Bundestag) elections

A hard-core example. Germany's federal legislative elections feature a mixed-member proportional (MMP) system where some candidates are elected in single-member constituencies while more are added through proportional allocation of party votes; additional seats are awarded to maintain the proportionality of the result.

This example was constructed according to the procedure and results laid out in the official document by the German Federal Returning Officer.

[1]:
```
import sys
import os
import csv
from decimal import Decimal

sys.path.append(os.path.join('..', '..'))
import votelib.convert
import votelib.candidate
import votelib.evaluate.core
import votelib.evaluate.threshold
import votelib.evaluate.proportional
```

2.4.1 Vote loading

Each voter casts two votes in German legislative election; the first one goes to a specific person standing in the local constituency, and the second one goes to a party. Independent constituency candidates are a minority and none has won a seat in the past fifty years; therefore, we omit this step and only consider votes for parties.

Since the voting system has two nesting levels - federal states *(Land)* and local constituency *(Wahlkreis)* -, we construct two doubly-nested dictionaries - one for the first votes *(Erststimme)*, and one for the second votes *(Zweitstimme)*.

[2]:
```
fpath = os.path.join('..', '..', 'tests', 'real', 'data', 'de_bdt_2017.csv')
with open(fpath, encoding='utf8') as infile:
    rows = list(csv.reader(infile, delimiter=';'))
party_names = [item for item in rows[0][2:] if item]
erst_stimmen, zweit_stimmen = {}, {}
for row in rows[2:]:
```

(continues on next page)

(continued from previous page)

```
    wahlkreis, land = row[:2]
    # Next cells in rows are first and second votes alternating, grouped by party. Empty␣
↪cells are zeroes.
    row[2:] = [int(x) if x else 0 for x in row[2:]]
    # Add the first and second votes to the register.
    erst_stimmen.setdefault(land, {})[wahlkreis] = dict(zip(party_names, row[2::2]))
    zweit_stimmen.setdefault(land, {})[wahlkreis] = dict(zip(party_names, row[3::2]))
print(erst_stimmen['Berlin']['Berlin-Mitte'])
```

```
{'CDU': 27654, 'SPD': 35036, 'DIE LINKE': 30492, 'GRÜNE': 26781, 'CSU': 0, 'FDP': 9017,
↪'AfD': 11782, 'PIRATEN': 0, 'NPD': 0, 'FREIE WÄHLER': 648}
```

2.4.2 Evaluator construction

We will need a complicated evaluator for the election.

The constituency winners are determined by simple plurality, each constituency giving a single winner. We will wrap this into a `ByConstituency` object to evaluate each constituency, and a combination of `SelectionToDistribution` and `MergedDistributions` to obtain party results at federal state level for each federal state:

[3]:
```
round1_eval = votelib.evaluate.core.ByConstituency(          # for each federal state
    votelib.evaluate.core.PostConverted(                     # sum all constituency␣
↪results
        votelib.evaluate.core.ByConstituency(                # evaluated over each␣
↪constituency
            votelib.evaluate.core.PostConverted(
                votelib.evaluate.core.Plurality(),           # by plurality selection
                votelib.convert.SelectionToDistribution()    # converted to a␣
↪distribution format {winner: 1}
            ),
            apportioner=1,                                   # each constituency has a␣
↪single winner
        ),
        votelib.convert.MergedDistributions()
    )
)
```

Before the elections, the shares of the nationwide total of 598 seats allocated to each federal state are determined according to their census population by the Sainte-Laguë (Schepers) method:

[4]:
```
land_inhab = {
    'Schleswig-Holstein': 2673803,
    'Hamburg': 1525090,
    'Niedersachsen': 7278789,
    'Bremen': 568510,
    'Nordrhein-Westfalen': 15707569,
    'Hessen': 5281198,
    'Rheinland-Pfalz': 3661245,
    'Baden-Württemberg': 9365001,
    'Bayern': 11362245,
    'Saarland': 899748,
    'Berlin': 2975745,
```

(continues on next page)

(continued from previous page)

```
    'Brandenburg': 2391746,
    'Mecklenburg-Vorpommern': 1548400,
    'Sachsen': 3914671,
    'Sachsen-Anhalt': 2145671,
    'Thüringen': 2077901,
}
prop_eval = votelib.evaluate.proportional.HighestAverages('sainte_lague')
land_seats = prop_eval.evaluate(land_inhab, 598)
print(land_seats)
```

```
{'Nordrhein-Westfalen': 128, 'Bayern': 93, 'Baden-Württemberg': 76, 'Niedersachsen': 59,
↪'Hessen': 43, 'Sachsen': 32, 'Rheinland-Pfalz': 30, 'Berlin': 24, 'Schleswig-Holstein':
↪ 22, 'Brandenburg': 20, 'Sachsen-Anhalt': 17, 'Thüringen': 17, 'Mecklenburg-Vorpommern
↪': 13, 'Hamburg': 12, 'Saarland': 7, 'Bremen': 5}
```

The seats to parties are distributed proportionally in each federal state according to their shares of second votes, again by the Sainte-Laguë method, eliminating parties that did not reach a 5 % statewide vote threshold:

[5]:
```
land_prop_eval = votelib.evaluate.core.ByConstituency(    # for each state
    prop_eval,
    apportioner=land_seats,                               # distribute a fixed given␣
↪amount of seats
    preselector=votelib.evaluate.threshold.RelativeThreshold(Decimal('.05'))
)
```

In addition to this, the system ensures that the number of seats awarded to each party is proportional to nationwide vote counts. These seat counts are computed by the Sainte-Laguë method. Only parties that gained at least 5 % of the nationwide vote or at least three constituency seats are entitled to national votes:

[6]:
```
nat_eval = votelib.evaluate.core.Conditioned(
    votelib.evaluate.threshold.AlternativeThresholds([
        votelib.evaluate.threshold.RelativeThreshold(Decimal('.05')),
        votelib.evaluate.threshold.PreviousGainThreshold(
            votelib.evaluate.threshold.AbsoluteThreshold(3)
        )
    ]),
    prop_eval
)
```

In mixed-member proportional system, *overhang seats* often arise when a party wins more constituency seats in the first round than it is entitled to by the result of the second round. There are multiple ways to deal with overhang; in Germany, the overhang seats are retained and some more seats *(leveling seats)* are added to the size of the parliament (`AdjustedSeatCount`) so that the proportionality with respect to the second round is satisfied *in each federal state* (`LevelOverhangByConstituency`).

After the total chamber seat count is determined, the German system specifies that the seats awarded on the national level to each party are to be distributed among that party's federal state lists in proportion of their votes (`ByParty`), again by the Sainte-Laguë evaluator (`prop_eval`).

[7]:
```
round2_eval = votelib.evaluate.core.PreConverted(
    votelib.convert.ByConstituency(votelib.convert.VoteTotals()),        # sum votes␣
↪from constituencies to fed states
    votelib.evaluate.core.AdjustedSeatCount(
        calculator=votelib.evaluate.core.LevelOverhangByConstituency(    # computes the␣
↪actual number of seats
```

(continues on next page)

(continued from previous page)

```
            constituency_evaluator=land_prop_eval,                          # by␣
↪distributing seats in each fed state
            overall_evaluator=nat_eval,                                     # and comparing␣
↪to nationwide distribution
        ),
        evaluator=votelib.evaluate.core.ByParty(                            # with the␣
↪total number of seats
            overall_evaluator=nat_eval,                                     # distribute␣
↪them among the parties nationally
            allocator=prop_eval,                                            # and␣
↪distribute those to its fed state lists.
        )
    )
)
```

Phew. Now we combine the two rounds into a single evaluator using `MultistageDistributor` (specifying that the votes are doubly nested by `depth=2`) and specify the normal count of seats (which will then be adjusted by the second round evaluator) to be 598.

[8]:
```
total_eval = votelib.evaluate.FixedSeatCount(
    votelib.evaluate.core.MultistageDistributor([round1_eval, round2_eval], depth=2),
    598
)
```

2.4.3 Performing the evaluation

The evaluator produces the seat counts for parties by federal states. We need to provide votes for both stages.

[9]:
```
land_result = total_eval.evaluate([erst_stimmen, zweit_stimmen])
for party, n_land_seats in land_result['Berlin'].items():
    print(party.ljust(10), n_land_seats)
```

```
SPD        5
DIE LINKE  6
CDU        6
GRÜNE      4
AfD        4
FDP        3
```

To determine the nationwide seat counts, we can use `MergedDistributions`.

[10]:
```
bund_result = votelib.convert.MergedDistributions().convert(land_result)
for party, n_land_seats in bund_result.items():
    print(party.ljust(10), n_land_seats)
print()
print('Total     ', sum(bund_result.values()))
```

```
SPD        153
CDU        200
AfD        94
FDP        80
DIE LINKE  69
```

(continues on next page)

(continued from previous page)

```
GRÜNE      67
CSU        46

Total      709
```

We see that seven parties are represented in the Bundestag after 2017, with the number of seats increased to 709 due to a high amount of overhang seats (for CDU and CSU, who won an overwhelming majority of the constituency seats) and leveling seats for other parties to offset that.

2.5 Irish 1990 presidential elections

A simple example of a transferable vote system: evaluating the result of the 1990 presidential elections in Ireland. We choose it because the transferable vote system variant is quite simple, as is described below, and the result differs from the one obtainable by simple plurality evaluation. (We will not go into what-if scenarios here; the change of a voting system usually means a change in voter behavior due to strategic voting.)

[1]:
```
import sys
import os

sys.path.append(os.path.join('..', '..'))
import votelib.candidate
import votelib.evaluate.sequential
```

2.5.1 Evaluator construction

First, we construct the evaluator.

All of Ireland forms a single constituency for presidential elections so the votes may be summed over the whole country and there is a single evaluation taking place.

Ireland uses the Single Transferable Vote system with the Hare (stochastic) variant of vote transfer and the Droop quota for election. There are no further thresholds or conditions in the evaluation.

[2]:
```
evaluator = votelib.evaluate.sequential.TransferableVoteSelector(
    quota_function='droop',
    transferer='Hare'
)
```

2.5.2 Vote construction

We use the vote counts from the official result. Full preference lists and their counts are not published; however, the vote transfer records allow us to infer enough to reconstruct the result.

Transferable voting uses ranked votes, so we use tuples of candidate names to encode voter preferences.

[3]:
```
votes = {
    ('Mary Robinson',): 612265,
    ('Brian Lenihan',): 694484,
    ('Austin Currie', 'Brian Lenihan'): 36789,
    ('Austin Currie', 'Mary Robinson'): 205565,
```

(continues on next page)

(continued from previous page)

```
    ('Austin Currie',): 25548,
}
```

2.5.3 Performing the evaluation

When the evaluator is set up correctly, obtaining the result is simple. If we do not specify the number of seats explicitly, most evaluators will provide 1 as the default value.

[4]:
```
print(evaluator.evaluate(votes))
```

```
['Mary Robinson']
```

2.6 Czech municipal elections 2018: Nové Město nad Metují

Czech municipal elections use an open-list proportional system that allows panachage - marking candidates across parties. Here, we give an example of how to evaluate such a composite election system.

[1]:
```
import sys
import os
import csv
import decimal

sys.path.append(os.path.join('..', '..'))
import votelib.candidate
import votelib.convert
import votelib.evaluate.core
import votelib.evaluate.threshold
import votelib.evaluate.proportional
import votelib.evaluate.openlist
```

2.6.1 Vote loading

The voters are given a huge ballot where they can vote for arbitrary candidates across parties, with the number of votes equal to the number of seats in the municipal council. In addition to this, they can also vote for a party, whose candidates obtain the votes not assigned to candidates elsewhere, counting from the top. The votes are thus counted for the candidates and can be aggregated to the parties.

We use the `Person` and `PoliticalParty` object to determine the relationships of candidates to their parties and aggregate their votes accordingly later.

[2]:
```
fpath = os.path.join('..', '..', 'tests', 'real', 'data', 'nmnmet_cc_2018.csv')
votes = {}
party_objs = {}
party_lists = {}
with open(fpath, encoding='utf8') as infile:
    for party, name, n_pers_votes in csv.reader(infile, delimiter=';'):
        # For each candidate: Get the according party object;
        party_obj = party_objs.setdefault(party, votelib.candidate.PoliticalParty(party))
        # Construct the person object with a reference to the party;
```

(continues on next page)

(continued from previous page)

```
        person = votelib.candidate.Person(name, candidacy_for=party_obj)
        # Record the candidate's votes;
        votes[person] = int(n_pers_votes)
        # Append the candidate to the party list of his or her party.
        party_lists.setdefault(party_obj, []).append(person)
```

An example of the votes and party list for the incumbent ruling party:

```
[3]: vpm_object = party_objs['VPM']
     print([cand.name for cand in party_lists[vpm_object]])
     {cand.name: n_votes for cand, n_votes in votes.items() if cand.candidacy_for == vpm_
     ↪object}
```

```
['Hable Petr', 'Beseda Michal Ing. MBA', 'Maur Vilém Ing. MBA', 'Němeček Jan Ing.',
↪'Petruželková Marie', 'Hladík Jiří', 'Neumann Jan Ing.', 'Novotný Jiří', 'Prouza Radek
↪', 'Hrnčíř Pavel', 'Bureš Michal Mgr.', 'Vlček Petr Mgr. A.', 'Rydlová Hana Mgr.',
↪'Minařík Jan Mgr.', 'Zimlová Věra', 'Roštlapil Tomáš Ing.', 'Krákorová Andrea Mgr.',
↪'Vintera Miroslav Ing.', 'Mach Martin Ing.', 'Reichmann David', 'Volf Martin Mgr.']
```

```
[3]: {'Hable Petr': 1479,
      'Beseda Michal Ing. MBA': 962,
      'Maur Vilém Ing. MBA': 1235,
      'Němeček Jan Ing.': 1013,
      'Petruželková Marie': 1053,
      'Hladík Jiří': 1200,
      'Neumann Jan Ing.': 1005,
      'Novotný Jiří': 808,
      'Prouza Radek': 1084,
      'Hrnčíř Pavel': 722,
      'Bureš Michal Mgr.': 793,
      'Vlček Petr Mgr. A.': 770,
      'Rydlová Hana Mgr.': 815,
      'Minařík Jan Mgr.': 770,
      'Zimlová Věra': 777,
      'Roštlapil Tomáš Ing.': 1006,
      'Krákorová Andrea Mgr.': 828,
      'Vintera Miroslav Ing.': 682,
      'Mach Martin Ing.': 664,
      'Reichmann David': 651,
      'Volf Martin Mgr.': 835}
```

2.6.2 Evaluator construction

Each Czech municipality forms a single constituency for the election.

The evaluation proceeds by first evaluating party results, so the results for the individual candidates must be grouped by their party. This mapping is defined by the candidates' `candidacy_for` attribute, which is recognized by the `IndividualToPartyMapper` object by default. Because independent candidates are not allowed to stand in the election, we add the behavior to recognize them as errors:

```
[4]: vote_grouper = votelib.convert.GroupVotesByParty(
         votelib.candidate.IndividualToPartyMapper(independents='error')
     )
```

The seats are allocated to the parties by the proportional D'Hondt system with a 5 % municipal vote threshold. We thus construct the proportional evaluator conditioned by the vote threshold and pre-aggregated by summing the grouped votes for parties:

[5]:
```
party_evaluator = votelib.evaluate.core.PreConverted(
    votelib.convert.PartyTotals(),
    votelib.evaluate.core.Conditioned(
        votelib.evaluate.threshold.RelativeThreshold(
            decimal.Decimal('.05'), accept_equal=True
        ),
        votelib.evaluate.proportional.HighestAverages('d_hondt'),
    )
)
```

Next, the party open lists are evaluated by the votes for their individual candidates. The candidate can advance forward in the list ranking if he or she has more than 5 % of the votes for the list; all such candidates are ranked first by the number of votes in descending order, and the rest goes after them in list order. We can use `PartyListEvaluator` to manage the list election and have `ThresholdOpenList` determine the elected candidates for each party. We use the vote grouper in two places to group both the party votes and list votes, which are passed separately:

[6]:
```
list_evaluator = votelib.evaluate.core.PreConverted(
    vote_grouper,
    votelib.evaluate.core.PartyListEvaluator(
        party_evaluator,
        votelib.evaluate.openlist.ThresholdOpenList(
            jump_fraction=decimal.Decimal('.05')
        ),
        list_votes_converter=vote_grouper,
    )
)
```

Finally, we fix the number of seats - the municipal council of Nové Město nad Metují has 21 seats:

[7]:
```
evaluator = votelib.evaluate.core.FixedSeatCount(
    list_evaluator, 21
)
```

2.6.3 Performing the evaluation

With the evaluator set up, we obtain the evaluation as lists of candidates per party.

[8]:
```
list_results = evaluator.evaluate(
    votes,
    list_votes=votes,
    party_lists=party_lists
)
for party, mandates in list_results.items():
    print(party.name.ljust(15), ', '.join([cand.name for cand in mandates]))
```

```
VPM             Hable Petr, Maur Vilém Ing. MBA, Hladík Jiří, Prouza Radek, Petruželková␣
↪Marie, Němeček Jan Ing.
ODS             Hovorka Libor Ing., Kupková Irena Mgr., Slavík Milan Ing., Jarolímek␣
↪Miroslav
```

(continues on next page)

(continued from previous page)

```
NM              Sláma Jiří Bc., Žahourková Markéta, Paarová Soňa Mgr.
VČ              Tymel Jiří Ing., Prouza Martin Ing., Balcarová Jana Mgr.
KDU-ČSL         Hylský Josef Mgr., Neumann Petr Ing., Dostál Pavel Ing. et Ing.
ČSSD            Čopík Jan Ing. Ph.D.
KSČM            Kulhavá Zdeňka PhDr.
```

We can see that VPM, the incumbent ruling party, has defended its first place with six seats, but its second place candidate from the original list was not elected because other candidates from the party with more votes jumped over him during open list evaluation.

CHAPTER

THREE

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

INDEX