smallerize Documentation

Release 0.5.0

Marius Mather

Jan 27, 2020

CONTENTS:

A Python implementation of minimisation for clinical trials

- Open source: Mozilla Public License 2.0
- Documentation: https://smallerize.readthedocs.io.
- Source: https://gitlab.com/warsquid/smallerize.

CHAPTER

ONE

FEATURES

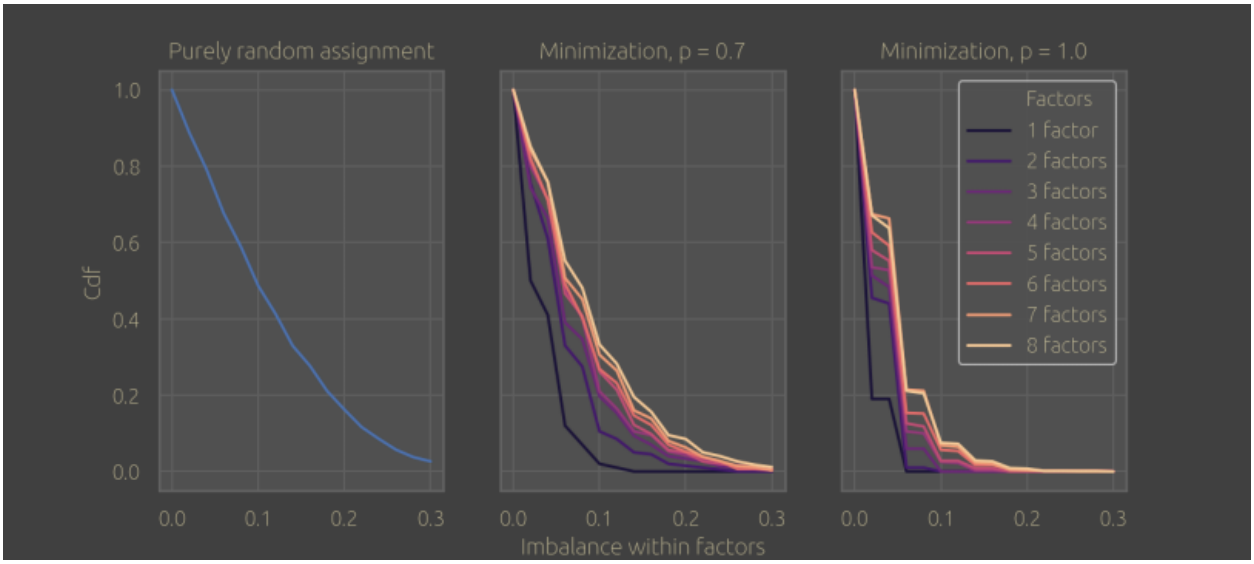
- Implements minimization as described in Pocock + Simon (1975): *Sequential treatment assignment with balancing for prognostic factors in the controlled clinical trial*
- Tested using `pytest` to ensure the results match the original implementation.
- Pure Python module with no dependencies (`pandas` is useful when conducting simulations but is optional)
- Includes all functions described in the article: range, standard deviation, variance, etc.
- Also implements the biased-coin minimization method described in Han et al. (2009): *Randomization by minimization for unbalanced treatment allocation*, to allow for unequal allocation ratios.
- Allows pure random assignment for comparison
- Simulation module to allow simulating the effects of different assignment schemes.

CHAPTER

TWO

EXAMPLE

Comparing minimization to purely random assignment by simulation:



See the example notebook for details of the simulation.

CHAPTER

THREE

CREDITS

This package was created with Cookiecutter and the audreyr/cookiecutter-pypackage project template.

CHAPTER

FOUR

DETAILED DOCUMENTATION

4.1 Installation

4.1.1 Stable release

To install smallerize, run this command in your terminal:

```
$ pip install smallerize
```

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

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

4.2 Usage

To use smallerize in a project:

```
import smallerize
```

Or:

```
from smallerize import Arm, Factor, Minimizer
```

A simple example of setting up a trial and assigning a participant is:

```
from smallerize import Arm, Factor, Minimizer

sex = Factor('Sex', levels=['Male', 'Female'])
site = Factor('Site', levels=['A', 'B', 'C'])

minimizer = Minimizer(
    factors=[sex, site],
    arms=[Arm('Treat'), Arm('Control')],
    d_imbalance_method='standard_deviation',
    probability_method='best_only',
    preferred_p=0.75
)

new_participant = {'Sex': 'Female', 'Site': 'B'}
assigned_arm = minimizer.assign_participant(new_participant)
```

4.2.1 Simulation

You can also use the `simulate` module to compare the performance of different allocation methods and settings:

```
from smallerize import simulate
```

See *the API docs* for further details..

4.3 Setting up a trial

Setting up a trial to use the minimization method requires you to choose a few settings, based on the design of the trial and what you think will be most important to balance. These are:

- The *arms* that participants will be assigned to.
- The prognostic *factors* that need to be balanced.
- The function that calculates the amount of imbalance *within each factor*
- The function that combines the factor imbalance scores to find the *total imbalance*
- How *probabilities* will be assigned to each arm, based on the imbalance scores.

Some of those pieces might require additional settings, e.g. providing the probability that the most favoured arm will be assigned.

Putting these all together, a full specification for a trial might look like:

```
from smallerize import Arm, Factor, Minimizer

arms = [Arm('Control', allocation_ratio=1),
        Arm('Active', allocation_ratio=2)]
factors = [Factor('Sex', levels=['Female', 'Male']),
           Factor('Site', levels=['Site A', 'Site B', 'Site C'])]

minimizer = Minimizer(
    factors=factors,
    arms=arms,
    d_imbalance_method='marginal_balance',
    total_imbalance_method='sum',
    probability_method='biased_coin',
    preferred_p=0.8
)
```

4.3.1 Trial arms

First we outline all the arms that participants will be assigned to.

- Each arm requires a unique name
- Each arm might have a different allocation ratio, if the trial design requires different proportions in each group. This is optional, if you don't provide them all arms will be assigned in equal proportions.

Example:

```
# Equal allocation ratios
equal_arms = [Arm('Placebo'), Arm('Surgery')]
# Different allocation ratios
unbalanced_arms = [
    Arm('Control', allocation_ratio=1),
    Arm('Active', allocation_ratio=2),
    Arm('Waitlist', allocation_ratio=1)
]
```

Note: When using unequal allocation ratios, you might want to use the 'biased coin' method to assign probabilities

4.3.2 Factors

Factors are participant characteristics that we want to balance because they might affect the trial outcomes. The minimization method allows you to balance participants across multiple factors simultaneously - unlike in stratification methods, this doesn't split participants up into increasingly small strata.

Factors must be categorical - they must have a finite set of levels that each participant will match up to.

- Each factor requires a unique name.
- Each factor will have its own factor levels.
- *(Optional):* factors can be given different weights, so that they contribute different amounts to the total imbalance score. When using different weights, you must use the "weighted sum" method to calculate the total imbalance score.

Examples:

```
sex = Factor('Sex', levels=['Female', 'Male'])
# Factors must be categorical - bin numeric variables
age = Factor('Age', levels=['20-29', '30-39', '40+'])
# Optional weights
weighted = [
    Factor('Severity', levels=['Low', 'High'], weight=2.0),
    Factor('Sex', levels=['Female', 'Male'], weight=1.0)
]
```

4.3.3 Imbalance within each factor

When assigning a new participant, we could consider what would happen if the new participant was assigned to each arm.

Within each factor, we look at the current participants who match the new participant's factor level, e.g. if for sex we currently have

Arm	Male	Female
Placebo	9	11
Active	12	8

and we are assigning a female participant, then assigning the participant to Placebo would result in `Placebo: 12, Active: 8` and assigning them to Active would result in `Placebo: 11, Active: 9`. While assigning a

female participant, the number of male participants assigned to each arm doesn't contribute to any imbalance calculations.

We take these potential counts, and use a function that scores the amount of imbalance between arms. E.g. if we use the range, then assigning to Placebo would give range$(12, 8) = 4$ and assigning to Active would give range$(11, 9) = 2$.

From Pocock + Simon (1975) and Han (2009), the different functions we can use for `d_imbalance_method` are:

- `'range'`: The highest count in an arm minus the smallest.
- `'standard_deviation'`: The standard deviation of counts.
- `'variance'`: Variance. Assigns an increasingly high score to large amounts of imbalance, so may be better at preventing extreme imbalance than the range or standard deviation.
- `'over_max_range'`: A binary indicator that is 1 if the range exceeds a specified threshold, and 0 otherwise.
  - If using this function, you must also supply the threshold as an additional argument `d_max_range`, e.g. `d_max_range=2`
- `'is_largest'`: A binary indicator that is 1 if the potential arm will have the highest count, and 0 otherwise (only works when the trial has exactly 2 arms).
- `'marginal_balance'`: A relative measure between 0.0 (all treatments equal) and 1.0 (all participants in 1 treatment), as outlined in Han (2009). Weights factor levels with many participants lower, so that rare factor levels also contribute to the total imbalance.

We carry out this calculation for each factor, generating a set of scores like:

Potential arm	Resulting imbalance within Sex	Resulting imbalance within Severity
Placebo	2	1
Active	1	3

Note: When arms have different allocation ratios, we divide the count in each arm by its allocation ratio first, meaning the degree of imbalance is calculated relative to the desired counts. This doesn't entirely account for the different allocation ratios though, so using the `'biased_coin'` method to assign probabilities is still recommended.

Examples:

```
minimizer = Minimizer(
    factors=[sex, age],
    arms=[Arm('Placebo'), Arm('Active')],
    d_imbalance_method='over_max_range',
    # over_max_range requires an additional argument
    d_max_range=3
)
```

```
preferred_p argument was not provided. Using default value of 0.75
```

4.3.4 Total imbalance

Once we have the degree of imbalance within each factor, we combine them into a total score, the overall imbalance that would result from assigning each potential arm. We can just use the sum here, although we can also apply different weights to each factor and use a weighted sum. Valid values for `total_imbalance_method` are:

- `'sum'`

- `'weighted_sum'`: Weight the sum by the `weight` attribute of each factor.

E.g. using the table above with Sex and Severity, and using the sum to combine scores, assigning to Placebo would result in a score of 3, and assigning to Active would result in a score of 4.

4.3.5 Assigning probabilities

After we have the total imbalance that would result from assigning the new participant to each arm, we rank the arms in increasing order of imbalance, and apply different probabilities. The different options for `probability_method` are:

- `'best_only'`: The arm that would result in the least imbalance is assigned with a high probability, and the remaining probability is divided among the remaining arms.
  - An additional argument `preferred_p` sets the probability for the arm with lowest imbalance, e.g. `preferred_p=0.7`. Setting this to 1.0 means the arm that produces the lowest imbalance is always assigned, but may make the allocation sequence too predictable.
- `'rank_all'`: The full ranking of arms is taken into account, with each successive arm being assigned a lower probability than the last.
  - An additional argument `q` sets the degree to which the arm with lowest imbalance is favoured. `q` must be between $1/N$ and $2/(N-1)$, where N is the number of arms in the trial, and higher values mean the arm with lower imbalance is favoured more. E.g. with $N = 4, q = 1/2$, the probabilities assigned are 0.4, 0.3, 0.2, 0.1.
- `'biased_coin'`: The biased-coin minimization method outlined in Han (2009), which correctly accounts for trials where arms have different allocation ratios.
  - An additional argument `preferred_p` sets the probability that the arm with the lowest allocation ratio will be assigned, when it produces the lowest imbalance. Arms with other allocation ratios have their probabilities adjusted accordingly.

4.4 Minimization

Note: This page tries to summarize the details of how minimization works as a method of treatment allocation. It mostly just summarises the original Pocock and Simon article found here

---

Minimization balances treatment assignments across multiple prognostic factors simultaneously. Let's start with some notation - we're running a trial and we have:

- N different treatments to assign participants to.
- M prognostic factors that we want to balance (all categorical[1]).
  - Each factor has n_i levels.

At any point during the trial, x_{ijk} is the number of participants assigned to treatment k, who have level j of factor i.

A new patient coming into the trial has a factor level for each of the M factors $r_1, \ldots, r_M$. If that patient is assigned to treatment k, there will be new values x^k_{ijl}:

$$x^k_{ir_ik} = x_{ijk} + 1$$

$$x^k_{ijl} = x_{ijl} \text{ for } l \neq k, j \neq r_i$$

[1] Continuous factors can be accounted for in some extensions of the minimization method, but they are not implemented in `smallerize`

i.e. we add 1 to all the counts x_{ijk} that match the treatment k and the factor levels the participant has, and all the other counts are unchanged.

4.4.1 Calculating imbalance

Imbalance within each factor

We start by calculating the imbalance within each factor. We choose a function $D(\{z\})$ that measures the variation in the participant counts $\{z\}$ (you can choose different functions like D = range). For each treatment k that a new participant could be assigned to, we can get the resulting counts that would be produced by assigning to k, and calculate:

$$d_{ik} = D(\{x^k_{ir_il}\}^N_{l=1})$$

i.e. the imbalance in treatment numbers across all treatments, for all participants with level r_i on factor i.

Note: Only participants who match the new participant's factor level r_i matter for the calculation of imbalance for factor i, participants with other factor levels don't affect the imbalance calculation

---

Total imbalance

Once we have the imbalance within each factor, we combine them to produce a total imbalance score. We choose a function G and calculate

$$G_k = G(d_{1k}, \cdots, d_{Mk})$$

that combines the d_{ik} for all the M factors (an obvious choice for G is just the sum). Then G_k is the total imbalance you would have, if the new patient is assigned to treatment k. We calculate G_k for all the N treatments.

Assigning probabilities

You can then rank the treatments in order of increasing G_k values (increasing = larger amount of imbalance), assigning ascending ranks $(1), (2), \ldots, (N)$.

Then pick some way of assigning decreasing probabilities of treatment assignment p_k to these, i.e.

$$p_1 \geq p_2 \geq \cdots \geq p_N$$

(p_k could be a function of G_k, the amount of imbalance, but this may be unnecessarily complex, and in practice just using the ranks should produce good balance).

4.4.2 Additional details

Unequal treatment allocation

Han et al. (2009) discussed how minimization could be adapted to trials with unequal treatment allocation ratios.

The original minimization method can be adapted by simply dividing the count of participants in each arm (within each factor) by that arm's allocation ratio - e.g. if the allocation ratio is 1:2 and the current counts for female participants are 11 and 20, then we would divide and get 11 and 10 - and then calculate the factor imbalance score from these values.

However this method is not perfect, as it results in a bias: slightly more participants are allocated to arms with low allocation ratios than desired, and slightly less to arms with high allocation ratios. Han et al. (2009) also propose a "biased coin minimization" method that solves this issue.

Biased coin minimization

In the biased coin minimization method, the treatment that would produce least imbalance is assigned with a probability given by:

$$p^H_{(i)} = 1 - \frac{\sum_{k \neq i}^{N} r_{(k)}}{\sum_{k \neq 1}^{N} r_{(k)}} \left(1 - p^H_{(1)}\right)$$

where $p^H_{(1)}$ is the probability that the treatment with the lowest allocation ratio will be assigned, when it produces the lowest imbalance.

The remaining probability is divided between treatments according to the formula:

$$p^L_{(i),H=(j)} = \frac{r_{(i)}}{\sum_{k \neq j}^{N} r_{(k)}} \left(1 - p^H_{(j)}\right)$$

4.5 API reference

4.5.1 Main module: smallerize

class smallerize.smallerize.**Factor**(*name: str, levels: Iterable[str], weight: float = 1.0*)
A prognostic factor with 2 or more levels, that needs to be balanced between treatment arms.

get_random_level() → str
Return one of the factor's levels at random (all levels have equal probability).

Returns Name of the chosen level

Return type str

get_random_level_multiple(*n*) → List[str]
Return n random levels, sampled with replacement. Faster than calling get_random_level multiple times.

Parameters **n** – Number of random levels to generate.

Returns List of level names.

class smallerize.smallerize.**Arm**(*name: str*, *allocation_ratio: int = 1*)
A treatment arm that participants will be assigned to. Different arms in the trial may have different allocation ratios, e.g. 1:2:1 for arms A, B and C.

class smallerize.smallerize.**Minimizer**(*factors: Iterable[smallerize.smallerize.Factor], arms: Iterable[smallerize.smallerize.Arm], d_imbalance_method: str = 'standard_deviation', total_imbalance_method: str = 'sum', probability_method: str = 'best_only', **method_args*)
Given a set of prognostic factors and treatment arms, assigns participants to arms based on the minimization algorithm (or purely random assignment for comparison).

D_IMBALANCE_METHODS = {'is_largest': <function _get_imbalance_is_largest>, 'marginal_

PROBABILITY_METHODS = ['best_only', 'rank_all', 'pure_random', 'biased_coin']

TOTAL_IMBALANCE_METHODS = ['sum', 'weighted_sum']

add_existing_participant(*factor_levels: dict*, *arm: str*) → None
Add a participant who has already been assigned to the trial to the count table. Modifies the count table in place.

Parameters

- **factor_levels** – A dictionary where the keys are the names of the factors in the trial, and the values are the factor levels for the participant.
- **arm** – Name of the arm they are assigned to.

arm_names

assign_participant(*factor_levels: dict*) → str
Assign a new participant to the trial, using the minimization algorithm. Modifies the count table in place, and returns the chosen arm.

Parameters **factor_levels** (*dict*) – A dictionary that maps from factor names to participants.

Returns Name of chosen arm

Return type str

factor_names

factor_weights

get_all_new_counts(*factor_levels: dict*) → Dict[str, dict]
For each potential arm that a new participant could be assigned to, calculate the number of participants that would be in each arm if the potential arm was chosen, within each factor level that the new participant belongs to.

Parameters **factor_levels** (*dict*) – A dictionary that maps from factor names to factor levels, giving the participant's factor levels for each factor used in the trial.

Returns A nested dictionary, that maps from (potential arm) -> (factor name) -> dict of arm counts within that factor

get_arm_probability(*imbalances: dict*) → dict
Calculate the probability assigned to each arm, based on the current imbalances and the chosen probability method.

get_assignment_info(*factor_levels: dict*, *do_assignment: bool = False*) → dict
Alternative to assign_participant(), takes factor levels for a new participant, chooses an arm, and returns the arm along with extra details about whether the most-favoured arm was chosen. By default, the participant is not actually assigned. To assign the participant at the same time, set `do_assignment=True`.

Returns dict with keys: `'arm'`: chosen arm, `'prob'`: probability that was assigned to that arm before the selection, `'most_favoured'`: Whether the chosen arm was the arm with the highest probability (including if it was tied for highest probability).

get_current_x_counts(*factor_levels: dict*) → Dict[str, dict]
Return the current x_ijk (arm counts within each factor), when considering a participant with the given `factor_levels`.

Parameters **factor_levels** (*dict*) – A dictionary that maps from factor names to factor levels, giving the participant's factor levels for each factor used in the trial.

Returns

get_n() → int
Get the number of arms for the trial.

get_new_ds(*factor_levels: dict*) → Dict[str, dict]
Calculate the d_ik scores, the imbalance score within each factor, for each potential arm that a new participant could be assigned to.

Parameters **factor_levels** – A dict that maps from (factor name) -> (new participant's factor level)

Returns A dict that maps from (potential arm) -> (dict of scores for each factor)

get_new_total_imbalances(*factor_levels: dict*) → Dict[str, Union[int, float]]
Get the total imbalance scores that would result from assigning a participant with the given factor levels to each arm.

Parameters **factor_levels** (*dict*) – Maps from factor names to the participant's level for each factor.

Returns Total imbalance score for each potential arm.

Return type dict

reset_counts_to_zero()
Reset the counts of assigned participants to zero, i.e. starting over as if no participants had been assigned.

4.5.2 simulate

class smallerize.simulate.**SimulatedTrial**(*minimizers: Dict[str, smallerize.smallerize.Minimizer], factors: List[smallerize.smallerize.Factor]*)

assign_one(*factor_levels: dict*) → dict

create_random_participants(*n: int*)

simulate(*n_trials: int*)

4.6 Credits

4.6.1 Development Lead

- Marius Mather <marius.mather@gmail.com>

4.6.2 Contributors

None yet. Why not be the first?

4.7 History

4.7.1 0.5.0 (2019-05-08)

- Improved performance with large numbers of factors/factor levels by making the count table sparse.

4.7.2 0.4.0 (2019-02-02)

- Switched to Mozilla Public License.

4.7.3 0.3.0 (2019-01-02)

- Make it easier to check the valid total imbalance and probability methods.

4.7.4 0.2.0 (2018-10-22)

- Add the biased coin minimization method to allow for unequal treatment allocations.

4.7.5 0.1.0 (2018-10-07)

- First release on PyPI.

CHAPTER

FIVE

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

S

INDEX