Value Awareness Experiment

Nieves Montes

Oct 17, 2023

CONTENTS

CHAPTER

ONE

INTRODUCTION

This library analyzes the data of the *value awareness experiment*, part of the VALAWAI project. In fact, the experiment conducted is a variation of the *minimal group paradigm study*. In that experiment, a set of participants are recruited. They are told that they have all been assigned to one of two groups, which are mutually exclusive but nonetheless arbitrary (e.g., Klee-lovers vs Kandinsky lovers).

During the experiment, participants are presented with a series of games. Every game consists of a sequence of "domino chips", displaying a number at the top and at the bottom of the chip. These quantities indicate an allocation of tokens or utility that is allocated to the participant's in-group or out-group. The chip's top number may correspond to the in-group and the bottom number to the out-group, or vice-versa. For every game, the participant must select one chip of score to assign to someone in his in-group and someone in his out-group.

1.1 Formal Modeling of Games

Formally, we model a game in the minimal group paradigm study as:

$$g = \left[\langle u^i_{in}, u^i_{out}\rangle\right]_{i=1,\dots,n_g} \tag{1.1}$$

−19	−16	−13	−10	−7	−4	−1	0	1	2	3	4	5	6
6	5	4	3	2	1	0	−1	−4	−7	−10	−13	−16	−19

12	10	8	6	4	2	0	−1	−5	−9	−13	−17	−21	−25
−25	−21	−17	−13	−9	−5	−1	0	2	4	6	8	10	12

Fig. 1.1: Two example games in the minimal group paradigm study.

that is, a game g is an ordered sequence of n_g tuples $\langle u^i_{in}, u^i_{out}\rangle$, which allocates u_{in} tokens to the in-group and u_{out} tokens to the out-group. Additionally, the following two constraints hold:

- either $u^j_{in} > u^i_{in}$ or $u^j_{in} < u^i_{in}$ where $i > j$, and
- either $u^j_{out} > u^i_{out}$ or $u^j_{out} < u^i_{out}$ where $i > j$.

In other words, the sequences of in-group $\left[u^1_{in}, u^2_{in}, \ldots\right]$ and out-group tokens $\left[u^1_{out}, u^2_{out}, \ldots\right]$ are *monotonic*, either increasing or decreasing. They need not be monotonic in the same direction (e.g. the sequence of in-group tokens might ascend while the sequence of out-group tokens might descend).

1.2 Ideal Profiles

When a participant is faced with a set of games, they decide which chip to chose based on their *profile*. We define a *profile* p as a function $f_p : \mathbb{Z}^2 \to \mathbb{R}$. In essence, $f_p(\langle u_{in}, u_{out}\rangle)$ returns the valuation that a participant makes of a choice in the game. The higher $f_p(u_{in}, u_{out})$, the more the participant likes choice $\langle u_{in}, u_{out}\rangle$.

We have eight *ideal profiles* corresponding to stereotypical behaviors. These functions come in pairs f_1, f_2 where $f_2 = f_1^{-1}$:

- **Egalitarian** *vs* **Differentarian**
  - Egalitarians minimize the difference between in- and out-group.

$$f_{eq}(u_{in}, u_{out}) = \frac{1}{|u_{in} - u_{out}|} \tag{1.2}$$

  - Differentarians maximize the difference between in- and out-group.

$$f_{diff}(u_{in}, u_{out}) = |u_{in} - u_{out}| \tag{1.3}$$

- **Sectarian** *vs* **Self-hater**
  - Sectarians maximize the in-group.

$$f_{sec}(u_{in}, u_{out}) = u_{in} \tag{1.4}$$

  - Self-haters minimize in-group.

$$f_{sh}(u_{in}, u_{out}) = \frac{1}{u_{in}} \tag{1.5}$$

- **Bad person** *vs* **Altruist**
  - Bad people minimize the out-group.

$$f_{bp}(u_{in}, u_{out}) = \frac{1}{u_{out}} \tag{1.6}$$

  - Altruists maximize the out-group.

$$f_{alt}(u_{in}, u_{out}) = u_{out} \tag{1.7}$$

- **Utilitarian** *vs* **Sociopath**

– Utilitarians maximize the total utility.

$$f_{ut}(u_{in}, u_{out}) = u_{in} + u_{out} \tag{1.8}$$

– Sociopaths minimize the total utility.

$$f_{soc}(u_{in}, u_{out}) = \frac{1}{u_{in} + u_{out}} \tag{1.9}$$

We will these eight ideal profiles to analyze the empirical behavior of participants in our study.

CHAPTER

TWO

ANALYZING EMPIRICAL BEHAVIOR

From the experiment, we gather data on choices that players make (i.e. the chip they select from each game matrix, like the ones in Fig. 1.1). Hence, for every participant we gather the set of games $G = \{g_i\}$ they are presented with and the choices $C = \{\langle u^*_{in}, u^*_{out}\rangle_i\}_{g_i \in G}$ they have made in each of those games.

Given a set of games G and a set of choices C, we compute the *affinity* of the corresponding participant for each of the eight ideal profiles as:

$$\mathsf{Affty}(p) = \frac{1}{\|G\|} \sum_{g_i \in G} \mathsf{eval}_p(\langle u^*_{in}, u^*_{out}\rangle_i \mid g_i) \tag{2.1}$$

where $\mathsf{eval}_p(\langle u^*_{in}, u^*_{out}\rangle_i \mid g_i)$ is the *evaluation* that profile p makes of the choice $\langle u^*_{in}, u^*_{out}\rangle_i$ in game g_i. The conditioning on the game g_i is important, since we must take into account that participants are limited to the chips they are presented with for every game. For example, a perfect sectarian participant would, ideally, select an infinite amount of tokens for their in-group. However, they can only select the chip with the largest in-group allocation (e.g. +6 or +12 in the top row of the games in Fig. 1.1).

2.1 Evaluation Methods

The $\mathsf{eval}_p(\langle u^*_{in}, u^*_{out}\rangle_i \mid g_i)$ in Eq. (2.1) has not been yet specified. We propose *three different evaluation methods*. They differ on how strict they, and some attempt to counterbalance the fact that participants are constraint in the choices they can make.

1. **Ideal choice sets**

   When using ideal choice sets, we consider that a profile function f_p select the subset of choices in a game g that maximize f_p:

   $$\mathbb{S}_p(g) = \{\langle u_{in}, u_{out}\rangle \in g \mid \langle u_{in}, u_{out}\rangle = \mathsf{argmax}_{\langle u_{in}, u_{out}\rangle \in g} f(u_{in}, u_{out})\} \quad (2.2)$$

   where $\mathbb{S}_p(g)$ denotes the subset set of choices prescribed by profile p in game g. Then, the ideal choice set is used to evaluate an empirical choice in a game as:

   $$\mathsf{eval}_p(\langle u^*_{in}, u^*_{out}\rangle \mid g) = \begin{cases} 1 & \text{if } \langle u^*_{in}, u^*_{out}\rangle \in \mathbb{S}_p(g) \\ 0 & \text{otherwise} \end{cases} \quad (2.3)$$

   Evaluation using ideal choice sets is a very *maximalist* approach. In other words, a choice is only "good" (according to profile p) if it is the very best choice. The other evaluation methods offer a more nuance and less strict approach.

2. **Continuous evaluation**

   In continuous evaluation, a choice in a game is evaluated according to its relative position with respect to all other choices in the game. Mathemtically:

   $$\mathsf{eval}_p(\langle u^*_{in}, u^*_{out}\rangle \mid g) = \frac{f_p(u^*_{in}, u^*_{out}) - \mathsf{Min}}{\mathsf{Max} - \mathsf{Min}} \quad (2.4)$$

where:

$$\begin{aligned} \text{Min} &= \min_{\langle u_{in}, u_{out} \rangle \in g} f_p(u_{in}, u_{out}) \\ \text{Max} &= \max_{\langle u_{in}, u_{out} \rangle \in g} f_p(u_{in}, u_{out}) \end{aligned} \tag{2.5}$$

Continuous evaluation is more balanced in the sense that choices are assigned a score ranging from 0 to 1, depending on whether they agree the least or the most with profile p. Nonetheless, continuous evaluation may somewhat not fully reflect the attitude of a participant. Since participants are constraint to the set of available choices, their actual choice $\langle u^*_{in}, u^*_{out} \rangle$ may not have corresponded to the one they would select if they had complete freedom to propose an allocation of token to their in- and out-group. This issue is mitigated in the following evaluation method.

3. **Equidistant ranks**

   In evaluation by equidistant ranks, the set of choices in a game are ordered according to their value $f_p(u_{in}, u_{out})$. They are then assigned a score between 0 and 1 that is proportional to their position in their ranking.

   Equidistant ranks is, like continuous evaluation and unlike ideal choice sets, a non-maximalist evaluation metric. Additionally, it recognizes the fact that participants can only select available choices, and compromises on this point by equally spacing out all the available choices.

2.2 Visualizing Empirical Profiles

Given a set of games g presented to a participant and the set of choices C they have selected, we compute the affinity of the participant with respect to the 8 ideal profiles. We present the results as lollipop plots along 4 axes, where ideal behavior with respect to opposing profiles corresponds to opposing ends of one axis.

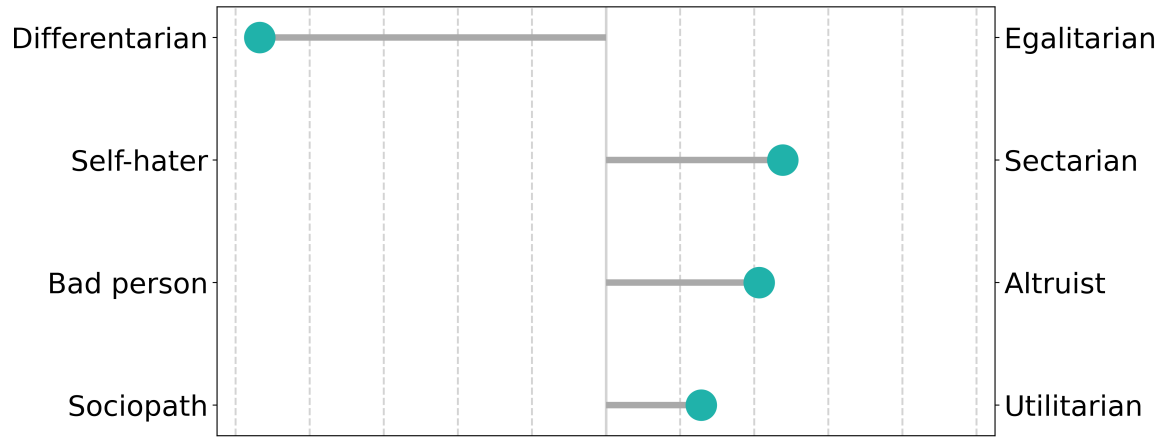

Fig. 2.1: Visualization of a profile computed with continuous evaluation (data from pilot study).

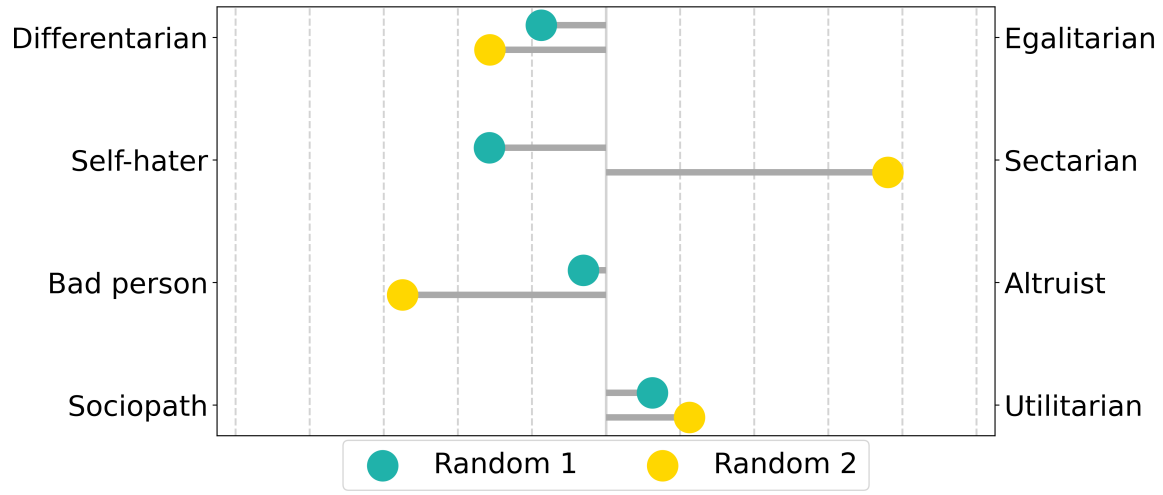

Fig. 2.2: Visualization of two (random) profiles being compared.

CHAPTER

THREE

INFERRING PROFILES FROM OBSERVATION

Despite analyzing behavior in terms of a set of predefined profiles, we also look to extract the profile function f_p that is actually guiding the choices of a participant. This can be framed as an *optimization problem* over the space of functions $\mathcal{F} : \mathbb{Z}^2 \to \mathbb{R}$ that map a tuple of two (u_{in}, u_{out}) to a real number.

The objective is to find, for each participant, the expression $f(u_{in}, u_{out})$ that best predicts the observed choices they have taken. To solve this problem, we implement an Evolutionary Strategy search algorithm. To generate potential candidates for $f(\cdot)$, we use the following grammar of arithmetic expressions:

$$\begin{aligned} \text{Arg} &::= \quad u_{in} \mid u_{out} \mid z, z \in \mathbb{Z} \\ \text{Op} &::= \quad + \mid - \mid \cdot \mid / \\ \text{Expr} &::= \quad \text{Arg} \ \ [\text{Op Expr}] \end{aligned} \tag{3.1}$$

To evaluate how well does a candidate function $f(\cdot)$ fit the experimental data, we weight consider two opposite factors.

1. What is the affinity of that function for the experimental data, as defined in Eq. (2.1).

2. How complex is that function. We favor simpler, less complex expressions. We define the *complexity* of an expression $\text{Comp}(f)$ as the number of operations Op required to generate it.

When evaluating the suitability of an expression, we consider its affinity for the experimental data and add a penalizing factor for the complexity of the expression. Hence, our optimization target is the following:

$$F(f) = \mathsf{Affty}(f) - \lambda \cdot \alpha(\mathsf{Comp}(f)) \tag{3.2}$$

where the function $\alpha : \mathbb{N} \to \mathbb{R}$ is an increasing function. Then, the best profile that describes a participant is:

$$f^* = \mathsf{argmax}_{f \in \mathcal{F}} F(f)$$

where $\mathcal{F}$ is the set of expressions that can be generated with the grammar.

3.1 Clustering

Given a group of participants, once we have found the expressions that optimally predict their behavior, we can try to partition them into undetected categories using *clustering*. We have done this for the pilot study data and obtained satisfactory results.

Clustering algorithms typically operate on either vector representation of the data points, or the matrix of distance between points. Since we are working with arithmetic expressions that cannot be represented in vector form, we need to define the *distance between two expressions* f_1 and f_2. We denote this distance as $\mathsf{dist}(f_1, f_2)$. Once this distance is defined, the matrix of distances can be defined and fed to an off-the-shelf clustering algorithms.

In order to compute the distance between two behavioral expressions f_1 and f_2, we start by defining the *most general game* MGG:

$$\mathrm{MGG} = \{\langle u_{in}, u_{out}\rangle \mid u_{in}, u_{out} \in [\min, \max]\} \tag{3.3}$$

In other words, the MGG is a game containing the set of all possible choices. That is, all tuples where the in-group and out-group tokens fall within some range. This range (limited by min and max) is selected in light of the choices that are presented to participants during the experiment.

We propose two different distance metrics. The first is based on the ideal choice sets. Here, the distance between two profiles f_1 and f_2 is computed from the ideal choice sets for MGG:

$$\mathsf{dist}(f_1, f_2) = \sqrt{1 - \frac{\|\mathbb{S}_1(\mathrm{MGG}) \cap \mathbb{S}_2(\mathrm{MGG})\|^2}{\|\mathbb{S}_1(\mathrm{MGG})\| \cdot \|\mathbb{S}_2(\mathrm{MGG})\|}} \tag{3.4}$$

The second distance metric is based on the *continuous evaluation* and *equidistant ranks* evaluation methods:

$$\mathsf{dist}(f_1, f_2) = 1 - \frac{\sum\limits_{u_{in}, u_{out}} \mathsf{eval}_1(u_{in}, u_{out}) \cdot \mathsf{eval}_2(u_{in}, u_{out})}{\left(\sum\limits_{u_{in}, u_{out}} \mathsf{eval}_1(u_{in}, u_{out})\right) \cdot \left(\sum\limits_{u_{in}, u_{out}} \mathsf{eval}_2(u_{in}, u_{out})\right)} \tag{3.5}$$

where the sums are taken over all the tuples $\langle u_{in}, u_{out} \rangle \in \mathrm{MGG}$.

CHAPTER

FOUR

MODULES

4.1 valawex.analysis

class valawex.analysis.**Choice**(*top: int*, *bottom: int*)

Bases: object

Class for a choice in a game.

Examples

```
>>> c = Choice(2, 5)
>>> c['top']
2
>>> c['bottom']
5
>>> print(c)
(2, 5)
```

class valawex.analysis.**Game**(**choices: Choice*, *game_id: Any | None = None*)

Bases: object

Class for a game as a set of possible choices.

Examples

```
>>> g = Game(Choice(2, 3), Choice(7, 1), game_id='my_game')
>>> print(g)
my_game:
(2, 3)
(7, 1)
```

```
>>> g[0]
(2, 3)
>>> g[1]
(7, 1)
>>> g.get_choices()
[(2, 3), (7, 1)]
>>> g.num_choices()
2
```

assign_groups(*top_group: str*, *bottom_group: str*) → None

Assign the tokens in the top or bottom to the in- and/or out-group.

Parameters

- **top_group** (*str*) –
- **bottom_group** (*str*) –

Raises

ValueError – If `top_group` or `bottom_group` is not `in` or `out`.

get_groups() → Dict[str, str]

classmethod from_csv(*filename: str*, *top_col: int = 0*, *bottom_col: int = 1*, *game_id: Any | None = None*, ***kwargs*) → G

Parse a game from a csv file.

The csv file must contain one row per choice if the game. Which column represents the points at the top or bottom of the chip is customizable.

Parameters

- **filename** (*str*) – The csv file.
- **top_col** (*int, optional*) – The column index containing the points at the top of the chip, by default 0.
- **bottom_col** (*int, optional*) – The column index containing the points at the bottom of the chip, by default 1.
- **game_id** (*Any, optional*) –
- ****kwargs** – Additional arguments for csv parsing.

Returns

G – A game with the set of choices parsed form the csv file.

Return type

Game

Raises

ValueError – If top_col and bottom_col are equal.

See also:

`csv.reader`

get_choices() → List[*Choice*]

Get the choices in the game as a list.

Return type

List[*Choice*]

num_choices() → int

Get the number of choices in the game.

Return type

int

get_choice_index(*top: int*, *bottom: int*) → int

Get the index of a choice in the choice list of the game.

The choice is not passed as a Choice object, but as the tuple of in- and out-group tokens.

Parameters

- **top** (*int*) –
- **bottom** (*int*) –

Return type

int

Raises

LookupError – If the choice is not found.

ideal_choice_set(*function: Expr*, *tol: float = 1e-05*) → List[int]

Compute the ideal choice set for some pre-defined profile.

Parameters

- **function** (*Expr*) – Function that the profile seeks to maximize
- **tol** (*float, optional*) – When to consider an evaluation is a new maximum, by default 1.E-5

Return type

List[*Choice*]

Raises

ValueError – if *tol* is negative

continuous_evaluation(*function: Expr*) → List[float]

Evaluate the choices continously according to some profile.

The choices are assigned a score ranging from 0 (for the choice that is the least aligned with a profile) to 1 (for the choice that is the most aligned with a profile). Intermediate choices are assigned a score that is normalized by the range of the evaluations.

Parameters

function (*Expr*) –

Returns

- *List[float]* – The order in the evaluations returned respects the order of the choices in the game.
- *None* – If all the choices in the game are equally evaluated.

equidistant_ranks(*function: Expr*) → List[float]

Rank the choices in the game according to some profile.

The choices are assigned a rank from 0 (for the choice that is the least aligned with a profile) to 1 (for the choice that is the most aligned with a profile). Intermediate choices are assigned a score with *equal distancing* between pairs of consecutive choices.

Parameters

function (*Expr*) –

Returns

- *List[float]* – The order in the ranks returned respects the order of the choices in the game.
- *None* – If all the choices in the game are equally evaluated.

valawex.analysis.**U_IN = u_{in}**

Symbols for in- and out-group tokens arguments.

valawex.analysis.**U_OUT = u_{out}**

Symbols for in- and out-group tokens arguments.

valawex.analysis.**EGALITARIAN = Piecewise((1/(u_{in} - u_{out}), u_{in} > u_{out}), (1/(-u_{in} + u_{out}), u_{in} < u_{out}), (100, Eq(u_{in}, u_{out})))**

Egalitarian profile minimizes difference between in- and out-group.

valawex.analysis.**DIFFERENTARIAN = Piecewise((u_{in} - u_{out}, u_{in} > u_{out}), (-u_{in} + u_{out}, u_{in} < u_{out}), (1/100, Eq(u_{in}, u_{out})))**

Differentarian profile maximizes difference between in- and out-group.

valawex.analysis.**SECTARIAN = u_{in}**

Sectarian profile maximizes in-group.

valawex.analysis.**SELFHATER = 1/u_{in}**

Self-hater profile minimizes in-group.

valawex.analysis.**BADPERSON = 1/u_{out}**

Bad person profile minimizes out-group.

valawex.analysis.**ALTRUIST = u_{out}**

Altruist profile maximizes out-group.

valawex.analysis.**UTILITARIAN = u_{in} + u_{out}**

Utilitarian profile maximizes total payoff.

valawex.analysis.**SOCIOPATH = 1/(u_{in} + u_{out})**

Sociopath profile minimizes total payoff.

valawex.analysis.**get_opposite_profile**(*prof: str*) → str

Given an ideal profile, find its opposite.

Parameters

prof (*str*) –

Return type

str

valawex.analysis.**compute_participant_profile**(*games: List[Game]*, *choices: List[int]*, *method: str*, ***kwargs*) → Dict[str, float]

Compute the profile of a participant against all the ideal profiles.

If a game is not informative with respect to a particular profile, it is note considered when taking the average over the evaluation of all games.

Parameters

- **games** (*List[Game]*) –
- **choices** (*List[int]*) – Index of the choices selected at each game.
- **method** (*str*) – The method used to evaluate a choice by a profile in a given game. Options are: `ideal_choice_set`, `continuous_evaluation` or `equidistant_ranks`.

Returns

A map from each ideal profile to the affinity shown by the set of choices.

Return type

Dict[str, float]

Raises

ValueError – If the number of games and choices do not match.

valawex.analysis.**compute_plot_participant_profile**(*games: List[Game]*, *choices: List[int]*, *method: str*, *filename: str*, ***kwargs*) → Dict[str, float]

Compute and plot a profile for a participant.

Parameters

- **games** (*List[Game]*) –
- **choices** (*List[int]*) –
- **method** (*str*) –
- **filename** (*str*) – Where to save the generated lollipop plot.

Returns

The profile for the participant.

Return type

Dict[str, float]

See also:

compute_participant_profile

valawex.analysis.**compute_profile_4d**(*games: List[Game]*, *choices: List[int]*, *method: str*, *pid: Any*, ***kwargs*) → Dict[str, int]

Compute and transform the profile along four dimensions.

Parameters

- **games** (*List[Game]*) –
- **choices** (*List[int]*) –
- **method** (*str*) –

- **pid** (*Any*) – An identifier for the participant.

Returns

The adherence to each profile is quantified from 0 to 100.

Return type

Dict[str, int]

valawex.analysis.**analyze_participant**(*participant_data: Dict[str, str]*, *method: str*, ***kwargs*) → Dict[str, Any]

Analyze raw data in the value awareness experiment.

Parameters

- **participant_data** (*Dict[str, str]*) – A dictionary describing the games the participant is presented with and the decision they make.
- **method** (*str*) – Use 'equidistant_ranks'.

Returns

A dictionary describing the value profile of the participant.

Return type

Dict[str, Any]

valawex.analysis.**lollipop_plot**(*profile: Dict[str, float]*, *profile_compare: Dict[str, float] | None = None*, *profile_name: str | None = None*, *profile_compare_name: str | None = None*, *colour: str = 'lightseagreen'*, *colour_compare: str = 'gold'*)

Plot a profile as a lollipop plot.

If two profiles are provided, the plot compares the two and adds a legend at the bottom of the plot.

Parameters

- **profile** (*Dict[str, float]*) – Primary profile to display.
- **profile_compare** (*Dict[str, float], optional*) – Secondary profile. If it is not empty, the primary profile is compared to the secondary profile, by default None.
- **profile_name** (*str, optional*) – Name of the primary profile, by default None.
- **profile_compare_name** (*str, optional*) – Name of the secondary profile, by default None.
- **colour** (*str, optional*) – Colour of the lollipop for the primary profile, by default 'lightseagreen'.
- **colour_compare** (*str, optional*) – Colour of the lollipop for the secondary profile, by default 'gold'.

4.2 valawex.extraction

valawex.extraction.**Subs**(*a*, *b*)

valawex.extraction.**Div**(*a*, *b*)

valawex.extraction.**generate_child_expression**(*expr: Expr*) → Expr

Expand an expression by adding an operation.

Parameters

expr (*Expr*) –

Return type

Expr

valawex.extraction.**generate_valid_child**(*expr: Expr*) → Expr

Generate a valid child.

A valid child expression is one which is not constant, i.e. not all variables have been removed and does not have imaginary constants.

Parameters

expr (*Expr*) –

Return type

Expr

valawex.extraction.**compute_affinity**(*games: List[Game]*, *choices: List[int]*, *expr: Expr*, *method: str*, ***kwargs*) → float

Affinity of a set of empirical choices for a behaviour expression.

Parameters

- **games** (*List[Game]*) –
- **choices** (*List[int]*) –
- **expr** (*Expr*) –
- **method** (*str*) – The evaluation method. Must be one of `ideal_choice_set`, `continuous_evaluation` or `equidistant_ranks`.

Return type

float

Raises

ValueError – If `games` and `choices` have different lengths.

valawex.extraction.**compute_fitness**(*games: ~typing.List[~valawex.analysis.Game], choices: ~typing.List[int], expr: ~sympy.core.expr.Expr, method: str, penal_function: ~typing.Callable[[~sympy.core.expr.Expr], float] = <function <lambda>>, lamb: float = 0.01, **kwargs*) → float

Compute the fitness of an expression given a set of choices in games.

Parameters

- **games** (*List[Game]*) –
- **choices** (*List[int]*) –
- **expr** (*Expr*) –
- **method** (*str*) – The evaluation method. Must be one of `ideal_choice_set`, `continuous_evaluation` or `equidistant_ranks`.
- **penal_function** (*Callable[[Expr], float], optional*) – Complexity penalization, by default the number of operations in the expression.
- **lamb** (*float, optional*) – Weight of the complexity penalization, by default 0.01.
- ****kwargs** – Keyword arguments to compute the affinity.

Return type

float

class valawex.extraction.**Candidate**(*expr: Expr*, *fitness: float | None = None*)

Bases: `object`

A candidate in the affinity-maximizing search.

expr: Expr

fitness: float = None

valawex.extraction.**evolution_strategy**(*rid: ~typing.Any, games: ~typing.List[~valawex.analysis.Game], choices: ~typing.List[int], method: str, mu: int, lam: int, penal_function: ~typing.Callable[[~sympy.core.expr.Expr], float] = <function <lambda>>, lamb: float = 0.01, max_partial_iters: int = 20, max_total_iters: int = 100, threshold: float = 0.95, **kwargs*) → Tuple[Expr, float]

Simple evolutionary search strategy.

Parameters

- **rid** (*Any*) – An identifier for the optimization run.
- **games** (*List[Game]*) –
- **choices** (*List[int]*) –
- **method** (*str*) – The evaluation method. Must be one of `ideal_choice_set`, `continuous_evaluation` or `equidistant_ranks`.
- **mu** (*int*) – Number of the best candidates that survive from one generation to the next.
- **lam** (*int*) – Number of total children generated at every iteration.
- **penal_function** (*Callable[[Expr], float], optional*) – Complexity penalization, by default the number of operations

in the expression.

- **lamb** (*float, optional*) – Weight of the complexity penalization, by default 0.01.
- **max_partial_iters** (*int, optional*) – Maximum number of interations without improvement before halting, by default 20.
- **max_total_iters** (*int, optional*) – Maximum number of iterations before halting regardless of improvement, by default 100.
- **threshold** (*float, optional*) – If a candidate surpassing this fitness in encountered, the search is automatically halted, by default 0.95.

Returns

The optimal expression that fits the set of choices in the set of games, and its affinity.

Return type

Tuple[Expr, float]

Raises

ValueError – If `mu` or `lam` are negative.

PYTHON MODULE INDEX

V

INDEX

G

I

L

M

N

S

U

V