fitr

July 1, 2018

Contents

Part I

Overview & Foundations

Part II

Tutorials

Part III

API

`fitr.data`

A module containing a generic class for behavioural data.

BehaviouralData

`fitr.data.BehaviouralData()`

A flexible and generic object to store and process behavioural data across tasks

Arguments:

- **ngroups**: Integer number of groups represented in the dataset. Only > 1 if data are merged
- **nsubjects**: Integer number of subjects in dataset
- **ntrials**: Integer number of trials done by each subject
- **dict**: Dictionary storage indexed by subject.
- **params**: `ndarray((nsubjects, nparams + 1))` parameters for each (simulated) subject
- **meta**: Array of covariates of type `ndarray((nsubjects, nmetadata_features+1))`
- **tensor**: Tensor representation of the behavioural data of type `ndarray((nsubjects, ntrials, nfeatures))`

BehaviouralData.add_subject

`fitr.data.add_subject(self, subject_index, parameters, subject_meta)`

Appends a new subject to the dataset

Arguments:

- **subject_index**: Integer identification for subject
- **parameters**: `list` of parameters for the subject
- **subject_meta**: Some covariates for the subject (`list`)

BehaviouralData.initialize_data_dictionary

```
fitr.data.initialize_data_dictionary(self)
```

BehaviouralData.make_behavioural_ngrams

```
fitr.data.make_behavioural_ngrams(self, n)
```

Creates N-grams of behavioural data

BehaviouralData.make_cooccurrence_matrix

```
fitr.data.make_cooccurrence_matrix(self, k, dtype=<class 'numpy.float32'>)
```

BehaviouralData.make_tensor_representations

```
fitr.data.make_tensor_representations(self)
```

Creates a tensor with all subjects' data

Notes

Assumes that all subjects did same number of trials.

BehaviouralData.numpy_tensor_to_bdf

```
fitr.data.numpy_tensor_to_bdf(self, X)
```

Creates `BehaviouralData` formatted set from a dataset stored in a numpy `ndarray`.

Arguments:

- **X**: `ndarray((nsubjects, ntrials, m))` with `m` being the size of flattened single-trial data

---

BehaviouralData.unpack_tensor

```
fitr.data.unpack_tensor(self, x_dim, u_dim, r_dim=1, terminal_dim=1, get='sarsat')
```

Unpacks data stored in tensor format into separate arrays for states, actions, rewards, next states, and next actions.

Arguments:

x_dim : Task state space dimensionality (`int`) u_dim : Task action space dimensionality (`int`) r_dim : Reward dimensionality (`int`, default=1) terminal_dim : Dimensionality of the terminal state indicator (`int` , default=1) get : String indicating the order that data are stored in the array. Can also be shortened such that fewer elements are returned. For example, the default is `sarsat`.

Returns:

List with data, where each element is in the order of the argument `get`

BehaviouralData.update

```
fitr.data.update(self, subject_index, behav_data)
```

Adds behavioural data to the dataset

Arguments:

- **subject_index**: Integer index for the subject
- **behav_data**: 1-dimensional `ndarray` of flattened data

---

merge_behavioural_data

```
fitr.data.merge_behavioural_data(datalist)
```

Combines BehaviouralData objects.

Arguments:

- **datalist**: List of BehaviouralData objects

Returns:

`BehaviouralData` with data from multiple groups merged.

---

fitr.environments

Functions to synthesize data from behavioural tasks.

Graph

`fitr.environments.Graph()`

Base object that defines a reinforcement learning task.

Definitions

- $\mathbf{x} \in \mathcal{X}$ be a one-hot state vector, where $|\mathcal{X}| = n_x$
- $\mathbf{u} \in \mathcal{U}$ be a one-hot action vector, where $|\mathcal{U}| = n_u$
- $\mathsf{T} = p(\mathbf{x}_{t+1}|\mathbf{x}_t, \mathbf{u}_t)$ be a transition tensor
- $p(\mathbf{x})$ be a distribution over starting states
- $\mathcal{J} : \mathcal{X} \to \mathcal{R}$, where $\mathcal{R} \subseteq \mathbb{R}$ be a reward function

Arguments:

- **T**: Transition tensor
- **R**: Vector of rewards for each state such that scalar reward $r_t = \mathbf{r}^o p\mathbf{x}$
- **end_states**: A vector $\{0, 1\}^{n_x}$ identifying which states terminate a trial (aka episode)
- **p_start**: Initial state distribution
- **label**: A string identifying a name for the task
- **state_labels**: A list or array of strings labeling the different states (for plotting purposes)
- **action_labels**: A list or array of strings labeling the different actions (for plotting purposes)
- **rng**: `np.random.RandomState` object
- **f_reward**: A function whose first argument is a vector of rewards for each state, and whose second argument is a state vector, and whose output is a scalar reward
- **cmap**: Matplotlib colormap for plotting.

Notes

There are two critical methods for the `Graph` class: `observation()` and `step`. All instances of a `Graph` must be able to call these functions. Let's say you have some bandit task `MyBanditTask` that inherits from `Graph`. To run such a task would look something like this:

```
env = MyBanditTask()             # Instantiate your environment object
agent = MyAgent()                # Some agent object (arbitrary, really)
for t in range(ntrials):
    x = env.observation()        # Samples initial state
    u = agent.action(x)          # Choose some action
    x_, r, done = agent.step(u) # Transition based on action
```

What differentiates tasks are the transition tensor `T`, starting state distribution $p(\mathbf{x})$ and reward function $\mathcal{J}$ (which here would include the reward vector $\mathbf{r}$).

Graph.adjacency_matrix_decomposition

```
fitr.environments.adjacency_matrix_decomposition(self)
```

Singular value decomposition of the graph adjacency matrix

Graph.get_graph_depth

```
fitr.environments.get_graph_depth(self)
```

Returns the depth of the task graph.

Calculated as the depth from `START` (pre-initial state) to `END` (which absorbs trial from all terminal states), minus 2 to account for the `START->node` & `node->END` transitions.

Returns:

An `int` identifying the depth of the current graph for a single trial of the task

---

Graph.laplacian_matrix_decomposition

`fitr.environments.laplacian_matrix_decomposition(self)`

Singular value decomposition of the graph Laplacian

Graph.make_action_labels

`fitr.environments.make_action_labels(self)`

Creates labels for the actions (for plotting) if none provided

---

Graph.make_digraph

`fitr.environments.make_digraph(self)`

Creates a `networkx` `DiGraph` object from the transition tensor for the purpose of plotting and some other analyses.

Graph.make_state_labels

`fitr.environments.make_state_labels(self)`

Creates labels for the states (for plotting) if none provided

---

Graph.make_undirected_graph

`fitr.environments.make_undirected_graph(self)`

Converts the DiGraph to undirected and computes some stats

Graph.observation

```
fitr.environments.observation(self)
```

Samples an initial state from the start-state distribution $p(\mathbf{x})$

$$\mathbf{x}_0 \sim p(\mathbf{x})$$

Returns:

A one-hot vector `ndarray((nstates,))` indicating the starting state.

Examples:

```
x = env.observation()
```

Graph.plot_action_outcome_probabilities

```
fitr.environments.plot_action_outcome_probabilities(self, figsize=None, outfile=None,
```

Plots the probabilities of different outcomes given actions.

Each plot is a heatmap for a starting state showing the transition probabilities for each action-outcome pair within that state.

Graph.plot_graph

```
fitr.environments.plot_graph(self, figsize=None, node_size=2000, arrowsize=20, lw=1.5
```

Plots the directed graph of the task

Graph.plot_spectral_properties

```
fitr.environments.plot_spectral_properties(self, figsize=None, outfile=None, outfilet
```

Creates a set of subplots depicting the graph Laplacian and its spectral decomposition.

Graph.random_action

`fitr.environments.random_action(self)`

Samples a random one-hot action vector uniformly over the action space.

Useful for testing that your environment works, without having to create an agent.

$$\mathbf{u} \sim \mathrm{Multinomial}\Big(1, \mathbf{p} = \{p_i = \frac{1}{|\mathcal{U}|}\}_{i=1}^{|\mathcal{U}|}\Big)$$

Returns:

A one-hot action vector of type `ndarray((nactions,))`

Examples:

```
u = env.random_action()
```

Graph.step

`fitr.environments.step(self, action)`

Executes a state transition in the environment.

Arguments:

action : A one-hot vector of type `ndarray((naction,))` indicating the action selected at the current state.

Returns:

A 3-tuple representing the next state (`ndarray((noutcomes,))`), scalar reward, and whether the current step terminates a trial (`bool`).

Raises:

`RuntimeError` if `env.observation()` not called after a previous `env.step(...)` call yielded a terminal state.

TwoArmedBandit

`fitr.environments.TwoArmedBandit()`

Two armed bandit just as a tester

TwoArmedBandit.adjacency_matrix_decomposition

`fitr.environments.adjacency_matrix_decomposition(self)`

Singular value decomposition of the graph adjacency matrix

TwoArmedBandit.get_graph_depth

`fitr.environments.get_graph_depth(self)`

Returns the depth of the task graph.

Calculated as the depth from `START` (pre-initial state) to `END` (which absorbs trial from all terminal states), minus 2 to account for the `START->node` & `node->END` transitions.

Returns:

An `int` identifying the depth of the current graph for a single trial of the task

---

TwoArmedBandit.laplacian_matrix_decomposition

`fitr.environments.laplacian_matrix_decomposition(self)`

Singular value decomposition of the graph Laplacian

TwoArmedBandit.make_action_labels

`fitr.environments.make_action_labels(self)`

Creates labels for the actions (for plotting) if none provided

---

TwoArmedBandit.make_digraph

`fitr.environments.make_digraph(self)`

Creates a `networkx` `DiGraph` object from the transition tensor for the purpose of plotting and some other analyses.

TwoArmedBandit.make_state_labels

`fitr.environments.make_state_labels(self)`

Creates labels for the states (for plotting) if none provided

---

TwoArmedBandit.make_undirected_graph

`fitr.environments.make_undirected_graph(self)`

Converts the DiGraph to undirected and computes some stats

TwoArmedBandit.observation

`fitr.environments.observation(self)`

Samples an initial state from the start-state distribution $p(\mathbf{x})$

$$\mathbf{x}_0 \sim p(\mathbf{x})$$

Returns:

A one-hot vector `ndarray((nstates,))` indicating the starting state.

Examples:

```
x = env.observation()
```

TwoArmedBandit.plot_action_outcome_probabilities

```
fitr.environments.plot_action_outcome_probabilities(self, figsize=None, outfile=None,
```

Plots the probabilities of different outcomes given actions.

Each plot is a heatmap for a starting state showing the transition probabilities for each action-outcome pair within that state.

TwoArmedBandit.plot_graph

```
fitr.environments.plot_graph(self, figsize=None, node_size=2000, arrowsize=20, lw=1.5
```

Plots the directed graph of the task

TwoArmedBandit.plot_spectral_properties

```
fitr.environments.plot_spectral_properties(self, figsize=None, outfile=None, outfilet
```

Creates a set of subplots depicting the graph Laplacian and its spectral decomposition.

TwoArmedBandit.random_action

```
fitr.environments.random_action(self)
```

Samples a random one-hot action vector uniformly over the action space.

Useful for testing that your environment works, without having to create an agent.

$$\mathbf{u} \sim \mathrm{Multinomial}\Big(1, \mathbf{p} = \{p_i = \frac{1}{|\mathcal{U}|}\}_{i=1}^{|\mathcal{U}|}\Big)$$

Returns:

A one-hot action vector of type `ndarray((nactions,))`

Examples:

```
u = env.random_action()
```

TwoArmedBandit.step

```
fitr.environments.step(self, action)
```

Executes a state transition in the environment.

Arguments:

action : A one-hot vector of type `ndarray((naction,))` indicating the action selected at the current state.

Returns:

A 3-tuple representing the next state (`ndarray((noutcomes,))`), scalar reward, and whether the current step terminates a trial (`bool`).

Raises:

`RuntimeError` if `env.observation()` not called after a previous `env.step(...)` call yielded a terminal state.

OrthogonalGoNoGo

```
fitr.environments.OrthogonalGoNoGo()
```

The orthogonal GoNogo task from Guitart-Masip et al. (2012)

OrthogonalGoNoGo.adjacency_matrix_decomposition

`fitr.environments.adjacency_matrix_decomposition(self)`

Singular value decomposition of the graph adjacency matrix

OrthogonalGoNoGo.get_graph_depth

`fitr.environments.get_graph_depth(self)`

Returns the depth of the task graph.

Calculated as the depth from `START` (pre-initial state) to `END` (which absorbs trial from all terminal states), minus 2 to account for the `START->node` & `node->END` transitions.

Returns:

An `int` identifying the depth of the current graph for a single trial of the task

---

OrthogonalGoNoGo.laplacian_matrix_decomposition

`fitr.environments.laplacian_matrix_decomposition(self)`

Singular value decomposition of the graph Laplacian

OrthogonalGoNoGo.make_action_labels

`fitr.environments.make_action_labels(self)`

Creates labels for the actions (for plotting) if none provided

---

OrthogonalGoNoGo.make_digraph

`fitr.environments.make_digraph(self)`

Creates a `networkx` `DiGraph` object from the transition tensor for the purpose of plotting and some other analyses.

OrthogonalGoNoGo.make_state_labels

`fitr.environments.make_state_labels(self)`

Creates labels for the states (for plotting) if none provided

---

OrthogonalGoNoGo.make_undirected_graph

`fitr.environments.make_undirected_graph(self)`

Converts the DiGraph to undirected and computes some stats

OrthogonalGoNoGo.observation

`fitr.environments.observation(self)`

Samples an initial state from the start-state distribution $p(\mathbf{x})$

$$\mathbf{x}_0 \sim p(\mathbf{x})$$

Returns:

A one-hot vector `ndarray((nstates,))` indicating the starting state.

Examples:

```
x = env.observation()
```

OrthogonalGoNoGo.plot_action_outcome_probabilities

`fitr.environments.plot_action_outcome_probabilities(self, figsize=None, outfile=None,`

Plots the probabilities of different outcomes given actions.

Each plot is a heatmap for a starting state showing the transition probabilities for each action-outcome pair within that state.

OrthogonalGoNoGo.plot_graph

`fitr.environments.plot_graph(self, figsize=None, node_size=2000, arrowsize=20, lw=1.5`

Plots the directed graph of the task

OrthogonalGoNoGo.plot_spectral_properties

`fitr.environments.plot_spectral_properties(self, figsize=None, outfile=None, outfilet`

Creates a set of subplots depicting the graph Laplacian and its spectral decomposition.

OrthogonalGoNoGo.random_action

`fitr.environments.random_action(self)`

Samples a random one-hot action vector uniformly over the action space.

Useful for testing that your environment works, without having to create an agent.

$$\mathbf{u} \sim \text{Multinomial}\Big(1, \mathbf{p} = \{p_i = \frac{1}{|\mathcal{U}|}\}_{i=1}^{|\mathcal{U}|}\Big)$$

Returns:

A one-hot action vector of type `ndarray((nactions,))`

Examples:

```
u = env.random_action()
```

OrthogonalGoNoGo.step

```
fitr.environments.step(self, action)
```

Executes a state transition in the environment.

Arguments:

action : A one-hot vector of type `ndarray((naction,))` indicating the action selected at the current state.

Returns:

A 3-tuple representing the next state (`ndarray((noutcomes,))`), scalar reward, and whether the current step terminates a trial (`bool`).

Raises:

`RuntimeError` if `env.observation()` not called after a previous `env.step(...)` call yielded a terminal state.

TwoStep

```
fitr.environments.TwoStep()
```

An implementation of the Two-Step Task from Daw et al. (2011).

Arguments:

- **mu**: `float` identifying the drift of the reward-determining Gaussian random walks
- **sd**: `float` identifying the standard deviation of the reward-determining Gaussian random walks

---

TwoStep.adjacency_matrix_decomposition

`fitr.environments.adjacency_matrix_decomposition(self)`

Singular value decomposition of the graph adjacency matrix

TwoStep.f_reward

`fitr.environments.f_reward(self, R, x)`

TwoStep.get_graph_depth

`fitr.environments.get_graph_depth(self)`

Returns the depth of the task graph.

Calculated as the depth from `START` (pre-initial state) to `END` (which absorbs trial from all terminal states), minus 2 to account for the `START->node` & `node->END` transitions.

Returns:

An `int` identifying the depth of the current graph for a single trial of the task

---

TwoStep.laplacian_matrix_decomposition

`fitr.environments.laplacian_matrix_decomposition(self)`

Singular value decomposition of the graph Laplacian

TwoStep.make_action_labels

`fitr.environments.make_action_labels(self)`

Creates labels for the actions (for plotting) if none provided

---

TwoStep.make_digraph

`fitr.environments.make_digraph(self)`

Creates a `networkx` `DiGraph` object from the transition tensor for the purpose of plotting and some other analyses.

TwoStep.make_state_labels

`fitr.environments.make_state_labels(self)`

Creates labels for the states (for plotting) if none provided

---

TwoStep.make_undirected_graph

`fitr.environments.make_undirected_graph(self)`

Converts the DiGraph to undirected and computes some stats

TwoStep.observation

`fitr.environments.observation(self)`

Samples an initial state from the start-state distribution $p(\mathbf{x})$

$$\mathbf{x}_0 \sim p(\mathbf{x})$$

Returns:

A one-hot vector `ndarray((nstates,))` indicating the starting state.

Examples:

```
x = env.observation()
```

TwoStep.plot_action_outcome_probabilities

```
fitr.environments.plot_action_outcome_probabilities(self, figsize=None, outfile=None,
```

Plots the probabilities of different outcomes given actions.

Each plot is a heatmap for a starting state showing the transition probabilities for each action-outcome pair within that state.

TwoStep.plot_graph

```
fitr.environments.plot_graph(self, figsize=None, node_size=2000, arrowsize=20, lw=1.5
```

Plots the directed graph of the task

TwoStep.plot_reward_paths

```
fitr.environments.plot_reward_paths(self, outfile=None, outfiletype='pdf', figsize=No
```

TwoStep.plot_spectral_properties

```
fitr.environments.plot_spectral_properties(self, figsize=None, outfile=None, outfilet
```

Creates a set of subplots depicting the graph Laplacian and its spectral decomposition.

TwoStep.random_action

`fitr.environments.random_action(self)`

Samples a random one-hot action vector uniformly over the action space.

Useful for testing that your environment works, without having to create an agent.

$$\mathbf{u} \sim \mathrm{Multinomial}\Big(1, \mathbf{p} = \{p_i = \frac{1}{|\mathcal{U}|}\}_{i=1}^{|\mathcal{U}|}\Big)$$

Returns:

A one-hot action vector of type `ndarray((nactions,))`

Examples:

```
u = env.random_action()
```

TwoStep.step

`fitr.environments.step(self, action)`

Executes a state transition in the environment.

Arguments:

action : A one-hot vector of type `ndarray((naction,))` indicating the action selected at the current state.

Returns:

A 3-tuple representing the next state (`ndarray((noutcomes,))`), scalar reward, and whether the current step terminates a trial (`bool`).

Raises:

`RuntimeError` if `env.observation()` not called after a previous `env.step(...)` call yielded a terminal state.

ReverseTwoStep

`fitr.environments.ReverseTwoStep()`

From Kool & Gershman 2016.

ReverseTwoStep.adjacency_matrix_decomposition

`fitr.environments.adjacency_matrix_decomposition(self)`

Singular value decomposition of the graph adjacency matrix

ReverseTwoStep.f_reward

`fitr.environments.f_reward(self, R, x)`

ReverseTwoStep.get_graph_depth

`fitr.environments.get_graph_depth(self)`

Returns the depth of the task graph.

Calculated as the depth from `START` (pre-initial state) to `END` (which absorbs trial from all terminal states), minus 2 to account for the `START->node` & `node->END` transitions.

Returns:

An `int` identifying the depth of the current graph for a single trial of the task

---

ReverseTwoStep.laplacian_matrix_decomposition

`fitr.environments.laplacian_matrix_decomposition(self)`

Singular value decomposition of the graph Laplacian

ReverseTwoStep.make_action_labels

`fitr.environments.make_action_labels(self)`

Creates labels for the actions (for plotting) if none provided

---

ReverseTwoStep.make_digraph

`fitr.environments.make_digraph(self)`

Creates a `networkx` `DiGraph` object from the transition tensor for the purpose of plotting and some other analyses.

ReverseTwoStep.make_state_labels

`fitr.environments.make_state_labels(self)`

Creates labels for the states (for plotting) if none provided

---

ReverseTwoStep.make_undirected_graph

`fitr.environments.make_undirected_graph(self)`

Converts the DiGraph to undirected and computes some stats

ReverseTwoStep.observation

```
fitr.environments.observation(self)
```

Samples an initial state from the start-state distribution $p(\mathbf{x})$

$$\mathbf{x}_0 \sim p(\mathbf{x})$$

Returns:

A one-hot vector `ndarray((nstates,))` indicating the starting state.

Examples:

```
x = env.observation()
```

ReverseTwoStep.plot_action_outcome_probabilities

```
fitr.environments.plot_action_outcome_probabilities(self, figsize=None, outfile=None,
```

Plots the probabilities of different outcomes given actions.

Each plot is a heatmap for a starting state showing the transition probabilities for each action-outcome pair within that state.

ReverseTwoStep.plot_graph

```
fitr.environments.plot_graph(self, figsize=None, node_size=2000, arrowsize=20, lw=1.5
```

Plots the directed graph of the task

ReverseTwoStep.plot_spectral_properties

```
fitr.environments.plot_spectral_properties(self, figsize=None, outfile=None, outfilet
```

Creates a set of subplots depicting the graph Laplacian and its spectral decomposition.

ReverseTwoStep.random_action

`fitr.environments.random_action(self)`

Samples a random one-hot action vector uniformly over the action space.

Useful for testing that your environment works, without having to create an agent.

$$\mathbf{u} \sim \mathrm{Multinomial}\Big(1, \mathbf{p} = \{p_i = \frac{1}{|\mathcal{U}|}\}_{i=1}^{|\mathcal{U}|}\Big)$$

Returns:

A one-hot action vector of type `ndarray((nactions,))`

Examples:

```
u = env.random_action()
```

ReverseTwoStep.step

`fitr.environments.step(self, action)`

Executes a state transition in the environment.

Arguments:

action : A one-hot vector of type `ndarray((naction,))` indicating the action selected at the current state.

Returns:

A 3-tuple representing the next state (`ndarray((noutcomes,))`), scalar reward, and whether the current step terminates a trial (`bool`).

Raises:

`RuntimeError` if `env.observation()` not called after a previous `env.step(...)` call yielded a terminal state.

RandomContextualBandit

`fitr.environments.RandomContextualBandit()`

Generates a random bandit task

Arguments:

- **nactions**: Number of actions
- **noutcomes**: Number of outcomes
- **nstates**: Number of contexts
- **min_actions_per_context**: Different contexts may have more or fewer actions than others (never more than `nactions`). This variable describes the minimum number of actions allowed in a context.
- **alpha**:
- **alpha_start**:
- **shift_flip**:
- **reward_lb**: Lower bound for drifting rewards
- **reward_ub**: Upper bound for drifting rewards
- **reward_drift**: Values (`on` or `off`) determining whether rewards are allowed to drift
- **drift_mu**: Mean of the Gaussian random walk determining reward
- **drift_sd**: Standard deviation of Gaussian random walk determining reward

---

RandomContextualBandit.adjacency_matrix_decomposition

`fitr.environments.adjacency_matrix_decomposition(self)`

Singular value decomposition of the graph adjacency matrix

RandomContextualBandit.f_reward

`fitr.environments.f_reward(self, R, x)`

RandomContextualBandit.get_graph_depth

`fitr.environments.get_graph_depth(self)`

Returns the depth of the task graph.

Calculated as the depth from `START` (pre-initial state) to `END` (which absorbs trial from all terminal states), minus 2 to account for the `START->node` & `node->END` transitions.

Returns:

An `int` identifying the depth of the current graph for a single trial of the task

---

RandomContextualBandit.laplacian_matrix_decomposition

`fitr.environments.laplacian_matrix_decomposition(self)`

Singular value decomposition of the graph Laplacian

RandomContextualBandit.make_action_labels

`fitr.environments.make_action_labels(self)`

Creates labels for the actions (for plotting) if none provided

---

RandomContextualBandit.make_digraph

`fitr.environments.make_digraph(self)`

Creates a `networkx` `DiGraph` object from the transition tensor for the purpose of plotting and some other analyses.

RandomContextualBandit.make_state_labels

```
fitr.environments.make_state_labels(self)
```

Creates labels for the states (for plotting) if none provided

---

RandomContextualBandit.make_undirected_graph

```
fitr.environments.make_undirected_graph(self)
```

Converts the DiGraph to undirected and computes some stats

RandomContextualBandit.observation

```
fitr.environments.observation(self)
```

Samples an initial state from the start-state distribution $p(\mathbf{x})$

$$\mathbf{x}_0 \sim p(\mathbf{x})$$

Returns:

A one-hot vector `ndarray((nstates,))` indicating the starting state.

Examples:

```
x = env.observation()
```

RandomContextualBandit.plot_action_outcome_probabilities

```
fitr.environments.plot_action_outcome_probabilities(self, figsize=None, outfile=None,
```

Plots the probabilities of different outcomes given actions.

Each plot is a heatmap for a starting state showing the transition probabilities for each action-outcome pair within that state.

RandomContextualBandit.plot_graph

```
fitr.environments.plot_graph(self, figsize=None, node_size=2000, arrowsize=20, lw=1.5
```

Plots the directed graph of the task

RandomContextualBandit.plot_spectral_properties

```
fitr.environments.plot_spectral_properties(self, figsize=None, outfile=None, outfilet
```

Creates a set of subplots depicting the graph Laplacian and its spectral decomposition.

RandomContextualBandit.random_action

```
fitr.environments.random_action(self)
```

Samples a random one-hot action vector uniformly over the action space.

Useful for testing that your environment works, without having to create an agent.

$$\mathbf{u} \sim \mathrm{Multinomial}\Big(1, \mathbf{p} = \{p_i = \frac{1}{|\mathcal{U}|}\}_{i=1}^{|\mathcal{U}|}\Big)$$

Returns:

A one-hot action vector of type `ndarray((nactions,))`

Examples:

```
u = env.random_action()
```

RandomContextualBandit.step

`fitr.environments.step(self, action)`

Executes a state transition in the environment.

Arguments:

action : A one-hot vector of type `ndarray((naction,))` indicating the action selected at the current state.

Returns:

A 3-tuple representing the next state (`ndarray((noutcomes,))`), scalar reward, and whether the current step terminates a trial (`bool`).

Raises:

`RuntimeError` if `env.observation()` not called after a previous `env.step(...)` call yielded a terminal state.

`fitr.agents`

A modular way to build and test reinforcement learning agents.

There are three main submodules:

- `fitr.agents.policies`: which describe a class of functions essentially representing $f : \mathcal{X} \to \mathcal{U}$
- `fitr.agents.value_functions`: which describe a class of functions essentially representing $\mathcal{V} : \mathcal{X} \to \mathbb{R}$ and/or $\mathcal{Q} : \mathcal{Q} \times \mathcal{U} \to \mathbb{R}$
- `fitr.agents.agents`: classes of agents that are combinations of policies and value functions, along with some convenience functions for generating data from `fitr.environments.Graph` environments.

SoftmaxPolicy

`fitr.agents.policies.SoftmaxPolicy()`

Action selection by sampling from a multinomial whose parameters are given by a softmax.

Action sampling is

$$\mathbf{u} \sim \mathrm{Multinomial}(1, \mathbf{p} = \varsigma(\mathbf{v})).$$

Parameters of that distribution are

$$p(\mathbf{u}|\mathbf{v}) = \varsigma(\mathbf{v}) = \frac{e^{\beta \mathbf{v}}}{\sum_i^{|\mathbf{v}|} e^{\beta v_i}}.$$

Arguments:

- **inverse_softmax_temp**: Inverse softmax temperature β
- **rng**: `np.random.RandomState` object

SoftmaxPolicy.action_prob

`fitr.agents.policies.action_prob(self, x)`

Computes the softmax

SoftmaxPolicy.log_prob

`fitr.agents.policies.log_prob(self, x)`

Computes the log-probability of an action $\mathbf{u}$

$$\log p(\mathbf{u}|\mathbf{v}) = \beta \mathbf{v} - \log \sum_{v_i} e^{\beta \mathbf{v}_i}$$

Arguments:

- **x**: State vector of type `ndarray((nstates,))`

Returns:

Scalar log-probability

SoftmaxPolicy.sample

`fitr.agents.policies.sample(self, x)`

Samples from the action distribution

StickySoftmaxPolicy

`fitr.agents.policies.StickySoftmaxPolicy()`

Action selection by sampling from a multinomial whose parameters are given by a softmax, but with accounting for the tendency to perseverate (i.e. choosing the previously used action without considering its value).

Let $\mathbf{u}_{t-1} = (u_{t-1}^{(i)})_{i=1}^{|\mathcal{U}|}$ be a one hot vector representing the action taken at the last step, and β^ρ be an inverse softmax temperature for the influence of this last action.

Action sampling is thus:

$$\mathbf{u} \sim \text{Multinomial}(1, \mathbf{p} = \varsigma(\mathbf{v}, \mathbf{u}_{t-1})).$$

Parameters of that distribution are

$$p(\mathbf{u}|\mathbf{v}, \mathbf{u}_{t-1}) = \varsigma(\mathbf{v}, \mathbf{u}_{t-1}) = \frac{e^{\beta \mathbf{v} + \beta^\rho \mathbf{u}_{t-1}}}{\sum_i^{|\mathbf{v}|} e^{\beta v_i + \beta^\rho u_{t-1}^{(i)}}}.$$

Arguments:

- **inverse_softmax_temp**: Inverse softmax temperature β
- **perseveration**: Inverse softmax temperature β^ρ capturing the tendency to repeat the last action taken.
- **rng**: `np.random.RandomState` object

StickySoftmaxPolicy.action_prob

`fitr.agents.policies.action_prob(self, x)`

Computes the softmax

Arguments:

- **x**: `ndarray((nstates,))` one-hot state vector

Returns:

`ndarray((nstates,))` vector of action probabilities

StickySoftmaxPolicy.log_prob

`fitr.agents.policies.log_prob(self, x)`

Computes the log-probability of an action **u**

$$\log p(\mathbf{u}|\mathbf{v}, \mathbf{u}_{t-1}) = \left(\beta \mathbf{v} + \beta^\rho \mathbf{u}_{t-1}\right) - \log \sum_{v_i} e^{\beta \mathbf{v}_i + \beta^\rho u_{t-1}^{(i)}}$$

Arguments:

- `x`: State vector of type `ndarray((nstates,))`

Returns:

Scalar log-probability

StickySoftmaxPolicy.sample

```
fitr.agents.policies.sample(self, x)
```

Samples from the action distribution

Arguments:

- `x`: `ndarray((nstates,))` one-hot state vector

Returns:

`ndarray((nstates,))` one-hot action vector

EpsilonGreedyPolicy

```
fitr.agents.policies.EpsilonGreedyPolicy()
```

EpsilonGreedyPolicy.action_prob

```
fitr.agents.policies.action_prob(self, x)
```

Creates vector of action probabilities for e-greedy policy

EpsilonGreedyPolicy.sample

```
fitr.agents.policies.sample(self, x)
```

ValueFunction

`fitr.agents.value_functions.ValueFunction()`

A general value function object.

A value function here is task specific and consists of several attributes:

- `nstates`: The number of states in the task, $|\mathcal{X}|$
- `nactions`: Number of actions in the task, $|\mathcal{U}|$
- `V`: State value function $\mathbf{v} = \mathcal{V}(\mathbf{x})$
- `Q`: State-action value function $\mathbf{Q} = \mathcal{Q}(\mathbf{x}, \mathbf{u})$
- `etrace`: An eligibility trace (optional)

Note that in general we rely on matrix-vector notation for value functions, rather than function notation. Vectors in the mathematical typesetting are by default column vectors.

Arguments:

- **env**: A `fitr.environments.Graph`

ValueFunction.Qmax

`fitr.agents.value_functions.Qmax(self, x)`

Return maximal action value for given state

$$\max_{u_i} \mathcal{Q}(\mathbf{x}, u_i) = \max_{\mathbf{u}'} \mathbf{u}'^\top \mathbf{Q}\mathbf{x}$$

Arguments:

- **x**: `ndarray((nstates,))` one-hot state vector

Returns:

Scalar value of the maximal action value at the given state

ValueFunction.Qmean

`fitr.agents.value_functions.Qmean(self, x)`

Return mean action value for given state

$$Mean(\mathcal{Q}(\mathbf{x}, :)) = \frac{1}{|\mathcal{U}|}\mathbf{1}^\top \mathbf{Q}\mathbf{x}$$

Arguments:

- `x`: `ndarray((nstates,))` one-hot state vector

Returns:

Scalar value of the maximal action value at the given state

ValueFunction.Qx

`fitr.agents.value_functions.Qx(self, x)`

Compute action values for a given state

$$\mathcal{Q}(\mathbf{x}, :) = \mathbf{Q}\mathbf{x}$$

Arguments:

- `x`: `ndarray((nstates,))` one-hot state vector

Returns:

`ndarray((nactions,))` vector of values for actions in the given state

ValueFunction.Vx

`fitr.agents.value_functions.Vx(self, x)`

ValueFunction.uQx

`fitr.agents.value_functions.uQx(self, u, x)`

DummyLearner

`fitr.agents.value_functions.DummyLearner()`

A critic for the random learner

DummyLearner.Qmax

`fitr.agents.value_functions.Qmax(self, x)`

Return maximal action value for given state

$$\max_{u_i} \mathcal{Q}(\mathbf{x}, u_i) = \max_{\mathbf{u}'} \mathbf{u}'^\top \mathbf{Q}\mathbf{x}$$

Arguments:

- `x`: `ndarray((nstates,))` one-hot state vector

Returns:

Scalar value of the maximal action value at the given state

DummyLearner.Qmean

`fitr.agents.value_functions.Qmean(self, x)`

Return mean action value for given state

$$Mean(\mathcal{Q}(\mathbf{x}, :)) = \frac{1}{|\mathcal{U}|} \mathbf{1}^\top \mathbf{Q}\mathbf{x}$$

Arguments:

- `x`: `ndarray((nstates,))` one-hot state vector

Returns:

Scalar value of the maximal action value at the given state

DummyLearner.Qx

`fitr.agents.value_functions.Qx(self, x)`

Compute action values for a given state

$$\mathcal{Q}(\mathbf{x}, :) = \mathbf{Q}\mathbf{x}$$

Arguments:

- `x`: `ndarray((nstates,))` one-hot state vector

Returns:

`ndarray((nactions,))` vector of values for actions in the given state

DummyLearner.Vx

`fitr.agents.value_functions.Vx(self, x)`

DummyLearner.uQx

`fitr.agents.value_functions.uQx(self, u, x)`

DummyLearner.update

`fitr.agents.value_functions.update(self, x, u, r, x_, u_)`

InstrumentalRescorlaWagnerLearner

`fitr.agents.value_functions.InstrumentalRescorlaWagnerLearner()`

A general value function object.

A value function here is task specific and consists of several attributes:

- `nstates`: The number of states in the task, $|\mathcal{X}|$
- `nactions`: Number of actions in the task, $|\mathcal{U}|$
- `V`: State value function $\mathbf{v} = \mathcal{V}(\mathbf{x})$
- `Q`: State-action value function $\mathbf{Q} = \mathcal{Q}(\mathbf{x}, \mathbf{u})$
- `etrace`: An eligibility trace (optional)

Note that in general we rely on matrix-vector notation for value functions, rather than function notation. Vectors in the mathematical typesetting are by default column vectors.

Arguments:

- **env**: A `fitr.environments.Graph`

InstrumentalRescorlaWagnerLearner.Qmax

`fitr.agents.value_functions.Qmax(self, x)`

Return maximal action value for given state

$$\max_{u_i} \mathcal{Q}(\mathbf{x}, u_i) = \max_{\mathbf{u}'} \mathbf{u}'^\top \mathbf{Q}\mathbf{x}$$

Arguments:

- **x**: `ndarray((nstates,))` one-hot state vector

Returns:

Scalar value of the maximal action value at the given state

InstrumentalRescorlaWagnerLearner.Qmean

```
fitr.agents.value_functions.Qmean(self, x)
```

Return mean action value for given state

$$Mean(\mathcal{Q}(\mathbf{x}, :)) = \frac{1}{|\mathcal{U}|}\mathbf{1}^\top \mathbf{Q}\mathbf{x}$$

Arguments:

- `x`: `ndarray((nstates,))` one-hot state vector

Returns:

Scalar value of the maximal action value at the given state

InstrumentalRescorlaWagnerLearner.Qx

```
fitr.agents.value_functions.Qx(self, x)
```

Compute action values for a given state

$$\mathcal{Q}(\mathbf{x}, :) = \mathbf{Q}\mathbf{x}$$

Arguments:

- `x`: `ndarray((nstates,))` one-hot state vector

Returns:

`ndarray((nactions,))` vector of values for actions in the given state

InstrumentalRescorlaWagnerLearner.Vx

```
fitr.agents.value_functions.Vx(self, x)
```

InstrumentalRescorlaWagnerLearner.uQx

```
fitr.agents.value_functions.uQx(self, u, x)
```

InstrumentalRescorlaWagnerLearner.update

```
fitr.agents.value_functions.update(self, x, u, r, x_, u_)
```

QLearner

```
fitr.agents.value_functions.QLearner()
```

A general value function object.

A value function here is task specific and consists of several attributes:

- `nstates`: The number of states in the task, $|\mathcal{X}|$
- `nactions`: Number of actions in the task, $|\mathcal{U}|$
- `V`: State value function $\mathbf{v} = \mathcal{V}(\mathbf{x})$
- `Q`: State-action value function $\mathbf{Q} = \mathcal{Q}(\mathbf{x}, \mathbf{u})$
- `etrace`: An eligibility trace (optional)

Note that in general we rely on matrix-vector notation for value functions, rather than function notation. Vectors in the mathematical typesetting are by default column vectors.

Arguments:

- **env**: A `fitr.environments.Graph`

QLearner.Qmax

`fitr.agents.value_functions.Qmax(self, x)`

Return maximal action value for given state

$$\max_{u_i} \mathcal{Q}(\mathbf{x}, u_i) = \max_{\mathbf{u}'} \mathbf{u}'^{\top} \mathbf{Q}\mathbf{x}$$

Arguments:

- `x`: `ndarray((nstates,))` one-hot state vector

Returns:

Scalar value of the maximal action value at the given state

QLearner.Qmean

`fitr.agents.value_functions.Qmean(self, x)`

Return mean action value for given state

$$Mean(\mathcal{Q}(\mathbf{x}, :)) = \frac{1}{|\mathcal{U}|} \mathbf{1}^{\top} \mathbf{Q}\mathbf{x}$$

Arguments:

- `x`: `ndarray((nstates,))` one-hot state vector

Returns:

Scalar value of the maximal action value at the given state

QLearner.Qx

`fitr.agents.value_functions.Qx(self, x)`

Compute action values for a given state

$$\mathcal{Q}(\mathbf{x}, :) = \mathbf{Q}\mathbf{x}$$

Arguments:

- `x`: `ndarray((nstates,))` one-hot state vector

Returns:

`ndarray((nactions,))` vector of values for actions in the given state

QLearner.Vx

```
fitr.agents.value_functions.Vx(self, x)
```

QLearner.uQx

```
fitr.agents.value_functions.uQx(self, u, x)
```

QLearner.update

```
fitr.agents.value_functions.update(self, x, u, r, x_, u_)
```

SARSALearner

```
fitr.agents.value_functions.SARSALearner()
```

A general value function object.

A value function here is task specific and consists of several attributes:

- `nstates`: The number of states in the task, $|\mathcal{X}|$
- `nactions`: Number of actions in the task, $|\mathcal{U}|$

- `V`: State value function $\mathbf{v} = \mathcal{V}(\mathbf{x})$
- `Q`: State-action value function $\mathbf{Q} = \mathcal{Q}(\mathbf{x}, \mathbf{u})$
- `etrace`: An eligibility trace (optional)

Note that in general we rely on matrix-vector notation for value functions, rather than function notation. Vectors in the mathematical typesetting are by default column vectors.

Arguments:

- **env**: A `fitr.environments.Graph`

SARSALearner.Qmax

```
fitr.agents.value_functions.Qmax(self, x)
```

Return maximal action value for given state

$$\max_{u_i} \mathcal{Q}(\mathbf{x}, u_i) = \max_{\mathbf{u}'} \mathbf{u'}^\top \mathbf{Q}\mathbf{x}$$

Arguments:

- **x**: `ndarray((nstates,))` one-hot state vector

Returns:

Scalar value of the maximal action value at the given state

SARSALearner.Qmean

```
fitr.agents.value_functions.Qmean(self, x)
```

Return mean action value for given state

$$Mean(\mathcal{Q}(\mathbf{x}, :)) = \frac{1}{|\mathcal{U}|}\mathbf{1}^\top \mathbf{Q}\mathbf{x}$$

Arguments:

- `x`: `ndarray((nstates,))` one-hot state vector

Returns:

Scalar value of the maximal action value at the given state

SARSALearner.Qx

```
fitr.agents.value_functions.Qx(self, x)
```

Compute action values for a given state

$$\mathcal{Q}(\mathbf{x}, :) = \mathbf{Q}\mathbf{x}$$

Arguments:

- `x`: `ndarray((nstates,))` one-hot state vector

Returns:

`ndarray((nactions,))` vector of values for actions in the given state

SARSALearner.Vx

```
fitr.agents.value_functions.Vx(self, x)
```

SARSALearner.uQx

```
fitr.agents.value_functions.uQx(self, u, x)
```

SARSALearner.update

`fitr.agents.value_functions.update(self, x, u, r, x_, u_)`

Agent

`fitr.agents.agents.Agent()`

Base class for synthetic RL agents

Arguments:

meta : List of metadata of arbitrary type. e.g. labels, covariates, etc. params : List of parameters for the agent. Should be filled for specific agent.

Agent.reset_trace

`fitr.agents.agents.reset_trace(self, x, u=None)`

For agents with eligibility traces, this resets the eligibility trace (for episodic tasks)

Arguments:

- **x**: `ndarray((nstates,))` one-hot state vector
- **u**: `ndarray((nactions,))` one-hot action vector (optional)

BanditAgent

`fitr.agents.agents.BanditAgent()`

A base class for agents in bandit tasks (i.e. with one step).

This mainly has implications for generating data

BanditAgent.generate_data

```
fitr.agents.agents.generate_data(self, ntrials)
```

BanditAgent.reset_trace

```
fitr.agents.agents.reset_trace(self, x, u=None)
```

For agents with eligibility traces, this resets the eligibility trace (for episodic tasks)

Arguments:

- **x**: `ndarray((nstates,))` one-hot state vector
- **u**: `ndarray((nactions,))` one-hot action vector (optional)

MDPAgent

```
fitr.agents.agents.MDPAgent()
```

A base class for agents that operate on MDPs.

This mainly has implications for generating data

MDPAgent.generate_data

```
fitr.agents.agents.generate_data(self, ntrials)
```

MDPAgent.reset_trace

```
fitr.agents.agents.reset_trace(self, x, u=None)
```

For agents with eligibility traces, this resets the eligibility trace (for episodic tasks)

Arguments:

- **x**: `ndarray((nstates,))` one-hot state vector
- **u**: `ndarray((nactions,))` one-hot action vector (optional)

RandomBanditAgent

```
fitr.agents.agents.RandomBanditAgent()
```

An agent that simply selects random actions at each trial

RandomBanditAgent.action

```
fitr.agents.agents.action(self, state)
```

RandomBanditAgent.generate_data

```
fitr.agents.agents.generate_data(self, ntrials)
```

RandomBanditAgent.learning

```
fitr.agents.agents.learning(self, state, action, reward, next_state, next_action)
```

RandomBanditAgent.reset_trace

```
fitr.agents.agents.reset_trace(self, x, u=None)
```

For agents with eligibility traces, this resets the eligibility trace (for episodic tasks)

Arguments:

- **x**: `ndarray((nstates,))` one-hot state vector
- **u**: `ndarray((nactions,))` one-hot action vector (optional)

RandomMDPAgent

`fitr.agents.agents.RandomMDPAgent()`

An agent that simply selects random actions at each trial

Notes

This has been specified as an `OnPolicyAgent` arbitrarily.

RandomMDPAgent.action

`fitr.agents.agents.action(self, state)`

RandomMDPAgent.generate_data

`fitr.agents.agents.generate_data(self, ntrials)`

RandomMDPAgent.learning

`fitr.agents.agents.learning(self, state, action, reward, next_state, next_action)`

RandomMDPAgent.reset_trace

`fitr.agents.agents.reset_trace(self, x, u=None)`

For agents with eligibility traces, this resets the eligibility trace (for episodic tasks)

Arguments:

- **x**: `ndarray((nstates,))` one-hot state vector
- **u**: `ndarray((nactions,))` one-hot action vector (optional)

SARSASoftmaxAgent

`fitr.agents.agents.SARSASoftmaxAgent()`

An agent that uses the SARSA learning rule and a softmax policy

---

SARSASoftmaxAgent.action

`fitr.agents.agents.action(self, state)`

SARSASoftmaxAgent.generate_data

`fitr.agents.agents.generate_data(self, ntrials)`

SARSASoftmaxAgent.learning

`fitr.agents.agents.learning(self, state, action, reward, next_state, next_action)`

SARSASoftmaxAgent.reset__trace

`fitr.agents.agents.reset_trace(self, x, u=None)`

For agents with eligibility traces, this resets the eligibility trace (for episodic tasks)

Arguments:

- **x**: `ndarray((nstates,))` one-hot state vector
- **u**: `ndarray((nactions,))` one-hot action vector (optional)

QLearningSoftmaxAgent

`fitr.agents.agents.QLearningSoftmaxAgent()`

An agent that uses the Q-learning rule and a softmax policy

QLearningSoftmaxAgent.action

`fitr.agents.agents.action(self, state)`

QLearningSoftmaxAgent.generate__data

`fitr.agents.agents.generate_data(self, ntrials)`

QLearningSoftmaxAgent.learning

`fitr.agents.agents.learning(self, state, action, reward, next_state, next_action)`

QLearningSoftmaxAgent.reset__trace

```
fitr.agents.agents.reset_trace(self, x, u=None)
```

For agents with eligibility traces, this resets the eligibility trace (for episodic tasks)

Arguments:

- **x**: `ndarray((nstates,))` one-hot state vector
- **u**: `ndarray((nactions,))` one-hot action vector (optional)

RWSoftmaxAgent

```
fitr.agents.agents.RWSoftmaxAgent()
```

A base class for agents in bandit tasks (i.e. with one step).

This mainly has implications for generating data

RWSoftmaxAgent.action

```
fitr.agents.agents.action(self, state)
```

RWSoftmaxAgent.generate__data

```
fitr.agents.agents.generate_data(self, ntrials)
```

RWSoftmaxAgent.learning

```
fitr.agents.agents.learning(self, state, action, reward, next_state, next_action)
```

RWSoftmaxAgent.reset_trace

```
fitr.agents.agents.reset_trace(self, x, u=None)
```

For agents with eligibility traces, this resets the eligibility trace (for episodic tasks)

Arguments:

- **x**: `ndarray((nstates,))` one-hot state vector
- **u**: `ndarray((nactions,))` one-hot action vector (optional)

RWSoftmaxAgentRewardSensitivity

```
fitr.agents.agents.RWSoftmaxAgentRewardSensitivity()
```

A base class for agents in bandit tasks (i.e. with one step).

This mainly has implications for generating data

RWSoftmaxAgentRewardSensitivity.action

```
fitr.agents.agents.action(self, state)
```

RWSoftmaxAgentRewardSensitivity.generate_data

```
fitr.agents.agents.generate_data(self, ntrials)
```

RWSoftmaxAgentRewardSensitivity.learning

```
fitr.agents.agents.learning(self, state, action, reward, next_state, next_action)
```

RWSoftmaxAgentRewardSensitivity.reset_trace

`fitr.agents.agents.reset_trace(self, x, u=None)`

For agents with eligibility traces, this resets the eligibility trace (for episodic tasks)

Arguments:

- **x**: `ndarray((nstates,))` one-hot state vector
- **u**: `ndarray((nactions,))` one-hot action vector (optional)

`fitr.utils`

Functions used across `fitr`.

softmax

`fitr.utils.softmax(x)`

Computes the softmax function

$$p(\mathbf{x}) = \frac{e^{\mathbf{x} - \max_i x_i}}{\mathbf{1}^\top e^{\mathbf{x} - \max_i x_i}}$$

Arguments:

- **x**: Softmax logits (`ndarray((N,))`)

Returns:

Vector of probabilities of size `ndarray((N,))`

---

sigmoid

`fitr.utils.sigmoid(x, a_min=-10, a_max=10)`

Sigmoid function

$$\sigma(x) = \frac{1}{1 + e^{-x}}$$

Arguments:

- **x**: Vector
- **a_min**: Lower bound at which to clip values of `x`
- **a_max**: Upper bound at which to clip values of `x`

Returns:

Vector between 0 and 1 of size `x.shape`

---

stable_exp

```
fitr.utils.stable_exp(x, a_min=-10, a_max=10)
```

Clipped exponential function

Avoids overflow by clipping input values.

Arguments:

- **x**: Vector of inputs
- **a_min**: Lower bound at which to clip values of `x`
- **a_max**: Upper bound at which to clip values of `x`

Returns:

Exponentiated values of `x`.

---

logsumexp

```
fitr.utils.logsumexp(x)
```

Numerically stable logsumexp.

Computed as follows:

$$\max x + \log \sum_x e^{x - \max x}$$

Arguments:

- **x**: ‘ndarray(shape=(nactions,))“

Returns:

`float`

---

log_loss

```
fitr.utils.log_loss(p, q)
```

Log-loss function.

$$\mathcal{L} = \mathbf{p}^\top \log \mathbf{q} + (1 - \mathbf{p})^\top \log(1 - \mathbf{q})$$

Arguments:

- $\mathbf{p}$: Binary vector of true labels `ndarray((nsamples,))`
- $\mathbf{q}$: Vector of estimates (between 0 and 1) of type `ndarray((nsamples,))`

Returns:

Scalar log loss