fitr

July 1, 2018

Contents

1 Overview & Foundations **3**

2 Tutorials **4**

Getting Started 4

Installation 4

Simulating and Fitting a Two-Armed Bandit 4

I API **6**

3 Environments **7**

`fitr.environments` 7

Graph 7

TwoArmedBandit 11

OrthogonalGoNoGo 14

TwoStep 17

ReverseTwoStep 21

RandomContextualBandit 24

4 Agents **28**

`fitr.agents` 28

SoftmaxPolicy 28

StickySoftmaxPolicy 29

EpsilonGreedyPolicy 31

ValueFunction 32

DummyLearner 34

InstrumentalRescorlaWagnerLearner 36

QLearner 39

SARSALearner 42

Agent 44

BanditAgent 45

MDPAgent 47

RandomBanditAgent 48

RandomMDPAgent 50

Notes 50

SARSASoftmaxAgent 51

QLearningSoftmaxAgent 53

RWSoftmaxAgent 55

RWSoftmaxAgentRewardSensitivity . . . 57

5 Data **60**
`fitr.data` . . . 60
BehaviouralData . . . 60
merge_behavioural_data . . . 62

6 Inference **63**
`fitr.inference` . . . 63
OptimizationResult . . . 63
mlepar . . . 64
l_bfgs_b . . . 64

7 Criticism **66**
`fitr.criticism` . . . 66
actual_estimate . . . 66

8 Metrics **67**
`fitr.metrics` . . . 67
bic . . . 67
linear_correlation . . . 67
lme . . . 68
log_loss . . . 68

9 Utilities **69**
`fitr.utils` . . . 69
logsumexp . . . 69
relu . . . 69
scale_data . . . 70
sigmoid . . . 70
softmax . . . 71
stable_exp . . . 71

Chapter 1

Overview & Foundations

Chapter 2

Tutorials

Getting Started

Installation

```
pip install git+https://github.com/abrahamnunes/fitr.git
```

Simulating and Fitting a Two-Armed Bandit

```python
import numpy as np
import matplotlib.pyplot as plt
from fitr import generate_behavioural_data
from fitr.environments import TwoArmedBandit
from fitr.agents import RWSoftmaxAgent
from fitr.inference import mlepar
from fitr.utils import sigmoid
from fitr.utils import relu
from fitr.criticism.plotting import actual_estimate

N = 50   # number of subjects
T = 200 # number of trials

# Generate synthetic data
data = generate_behavioural_data(TwoArmedBandit(), RWSoftmaxAgent, N, T)

# Create log-likelihood function
def log_prob(w, D):
    lr  = sigmoid(w[0], a_min=-6, a_max=6)
    ist = relu(w[1], a_max=10)
    agent = RWSoftmaxAgent(TwoArmedBandit(), lr, ist)
    L = 0
    for t in range(D.shape[0]):
        x  = D[t, :3]
```

```
        u  = D[t, 3:5]
        r  = D[t, 5]
        x_ = D[t, 6:]
        L += u@agent.log_prob(x)
        agent.learning(x, u, r, x_, None)
    return L

# Fit model
res = mlepar(log_prob, data.tensor, nparams=2, maxstarts=5)
X = res.transform_xmin([sigmoid, relu])

# Criticism
lr_fig  = actual_estimate(data.params[:,1], X[:,0]); plt.show()
ist_fig = actual_estimate(data.params[:,2], X[:,1]); plt.show()
```

Part I

API

Chapter 3

Environments

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.`

Plots the directed graph of the task

Graph.plot_spectral_properties

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

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.
```

Plots the directed graph of the task

TwoArmedBandit.plot_spectral_properties

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

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.
```

Plots the directed graph of the task

OrthogonalGoNoGo.plot_spectral_properties

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

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.

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

Returns:

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

Examples:

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

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.
```

Plots the directed graph of the task

TwoStep.plot_reward_paths

```
fitr.environments.plot_reward_paths(self, outfile=None, outfiletype='pdf', figsize=N
```

TwoStep.plot_spectral_properties

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

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.
```

Plots the directed graph of the task

ReverseTwoStep.plot_spectral_properties

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

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.
```

Plots the directed graph of the task

RandomContextualBandit.plot_spectral_properties

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

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.

Chapter 4

Agents

`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 $f : \mathcal{X} \rightarrow \mathcal{U}$
- `fitr.agents.value_functions`: which describe a class of functions essentially representing $\mathcal{V} : \mathcal{X} \rightarrow \mathbb{R}$ and/or $\mathcal{Q} : \mathcal{Q} \times \mathcal{U} \rightarrow \mathbb{R}$
- `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:

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

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 $\mathbf{u}$

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

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()`

A policy that takes the maximally valued action with probability $1 - \epsilon$, otherwise chooses randomlyself.

Arguments:

- **epsilon**: Probability of not taking the action with highest value
- **rng**: `numpy.random.RandomState` object

EpsilonGreedyPolicy.action_prob

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

Creates vector of action probabilities for e-greedy policy

Arguments:

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

Returns:

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

EpsilonGreedyPolicy.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

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)`

Compute value of state $\mathbf{x}$

$$\mathcal{V}(\mathbf{x}) = \mathbf{v}^\top \mathbf{x}$$

Arguments:

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

Returns:

Scalar value of state $\mathbf{x}$

ValueFunction.uQx

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

Compute value of taking action $\mathbf{u}$ in state $\mathbf{x}$

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

Arguments:

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

Returns:

Scalar value of action $\mathbf{u}$ in state $\mathbf{x}$

ValueFunction.update

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

Updates the value function

In the context of the base `ValueFunction` class, this is merely a placeholder. The specific update rule will depend on the specific value function desired.

Arguments:

- **x**: `ndarray((nstates,))` one-hot state vector
- **u**: `ndarray((nactions,))` one-hot action vector
- **r**: Scalar reward
- **x_**: `ndarray((nstates,))` one-hot next-state vector
- **u_**: `ndarray((nactions,))` one-hot next-action vector

DummyLearner

```
fitr.agents.value_functions.DummyLearner()
```

A critic/value function for the random learner

This class actually contributes nothing except identifying that a value function has been chosen for an `Agent` object

Arguments:

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

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)
```

Compute value of state $\mathbf{x}$

$$\mathcal{V}(\mathbf{x}) = \mathbf{v}^\top \mathbf{x}$$

Arguments:

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

Returns:

Scalar value of state $\mathbf{x}$

DummyLearner.uQx

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

Compute value of taking action $\mathbf{u}$ in state $\mathbf{x}$

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

Arguments:

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

Returns:

Scalar value of action $\mathbf{u}$ in state $\mathbf{x}$

DummyLearner.update

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

Updates the value function

In the context of the base `ValueFunction` class, this is merely a placeholder. The specific update rule will depend on the specific value function desired.

Arguments:

- **x**: `ndarray((nstates,))` one-hot state vector
- **u**: `ndarray((nactions,))` one-hot action vector
- **r**: Scalar reward
- **x_**: `ndarray((nstates,))` one-hot next-state vector
- **u_**: `ndarray((nactions,))` one-hot next-action vector

InstrumentalRescorlaWagnerLearner

`fitr.agents.value_functions.InstrumentalRescorlaWagnerLearner()`

Learns an instrumental control policy through one-step error-driven updates of the state-action value function

The instrumental Rescorla-Wagner rule is as follows:

$$\mathbf{Q} \leftarrow \mathbf{Q} + \alpha(r - \mathbf{u}^\top \mathbf{Q}\mathbf{x})\mathbf{u}\mathbf{x}^\top,$$

where $0 < \alpha < 1$ is the learning rate, and where the reward prediction error (RPE) is $\delta = (r - \mathbf{u}^\top \mathbf{Q}\mathbf{x})$.

$$

Arguments:

- **env**: A `fitr.environments.Graph`
- **learning_rate**: Learning rate α

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)`

Compute value of state $\mathbf{x}$

$$\mathcal{V}(\mathbf{x}) = \mathbf{v}^\top \mathbf{x}$$

Arguments:

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

Returns:

Scalar value of state $\mathbf{x}$

InstrumentalRescorlaWagnerLearner.uQx

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

Compute value of taking action $\mathbf{u}$ in state $\mathbf{x}$

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

Arguments:

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

Returns:

Scalar value of action $\mathbf{u}$ in state $\mathbf{x}$

InstrumentalRescorlaWagnerLearner.update

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

Updates the value function

In the context of the base `ValueFunction` class, this is merely a placeholder. The specific update rule will depend on the specific value function desired.

Arguments:

- **x**: `ndarray((nstates,))` one-hot state vector
- **u**: `ndarray((nactions,))` one-hot action vector
- **r**: Scalar reward
- **x_**: `ndarray((nstates,))` one-hot next-state vector
- **u_**: `ndarray((nactions,))` one-hot next-action vector

QLearner

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

Learns an instrumental control policy through Q-learning

The Q-learning rule is as follows:

$$\mathbf{Q} \leftarrow \mathbf{Q} + \alpha(r + \gamma \max_{\mathbf{u}'} \mathbf{u}'^\top \mathbf{Q}\mathbf{x}' - \mathbf{u}^\top \mathbf{Q}\mathbf{x})\mathbf{z},$$

where $0 < \alpha < 1$ is the learning rate, $0 \leq \gamma \leq 1$ is a discount factor, and where the reward prediction error (RPE) is $\delta = (r + \gamma \max_{\mathbf{u}'} \mathbf{u}'^\top \mathbf{Q}\mathbf{x}' - \mathbf{u}^\top \mathbf{Q}\mathbf{x})$. We have also included an eligibility trace $\mathbf{z}$ defined as

$$\mathbf{z} = \mathbf{u}\mathbf{x}^\top + \gamma\lambda\mathbf{z}$$

Arguments:

- **env**: A `fitr.environments.Graph`
- **learning_rate**: Learning rate α
- **discount_factor**: Discount factor γ
- **trace_decay**: Eligibility trace decay λ

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)
```

Compute value of state $\mathbf{x}$

$$\mathcal{V}(\mathbf{x}) = \mathbf{v}^\top \mathbf{x}$$

Arguments:

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

Returns:

Scalar value of state $\mathbf{x}$

QLearner.uQx

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

Compute value of taking action $\mathbf{u}$ in state $\mathbf{x}$

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

Arguments:

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

Returns:

Scalar value of action $\mathbf{u}$ in state $\mathbf{x}$

QLearner.update

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

Updates the value function

In the context of the base `ValueFunction` class, this is merely a placeholder. The specific update rule will depend on the specific value function desired.

Arguments:

- **x**: `ndarray((nstates,))` one-hot state vector
- **u**: `ndarray((nactions,))` one-hot action vector
- **r**: Scalar reward
- **x_**: `ndarray((nstates,))` one-hot next-state vector
- **u_**: `ndarray((nactions,))` one-hot next-action vector

SARSALearner

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

Learns an instrumental control policy through the SARSA learning rule

The SARSA learning rule is as follows:

$$\mathbf{Q} \leftarrow \mathbf{Q} + \alpha(r + \gamma \mathbf{u'}^\top \mathbf{Q} \mathbf{x'} - \mathbf{u}^\top \mathbf{Q} \mathbf{x})\mathbf{z},$$

where $0 < \alpha < 1$ is the learning rate, $0 \leq \gamma \leq 1$ is a discount factor, and where the reward prediction error (RPE) is $\delta = (r + \gamma \mathbf{u'}^\top \mathbf{Q} \mathbf{x'} - \mathbf{u}^\top \mathbf{Q} \mathbf{x})$. We have also included an eligibility trace $\mathbf{z}$ defined as

$$\mathbf{z} = \mathbf{u} \mathbf{x}^\top + \gamma \lambda \mathbf{z}$$

Arguments:

- **env**: A `fitr.environments.Graph`
- **learning_rate**: Learning rate α
- **discount_factor**: Discount factor γ
- **trace_decay**: Eligibility trace decay λ

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)`

Compute value of state $\mathbf{x}$

$$\mathcal{V}(\mathbf{x}) = \mathbf{v}^\top \mathbf{x}$$

Arguments:

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

Returns:

Scalar value of state $\mathbf{x}$

SARSALearner.uQx

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

Compute value of taking action $\mathbf{u}$ in state $\mathbf{x}$

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

Arguments:

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

Returns:

Scalar value of action $\mathbf{u}$ in state $\mathbf{x}$

SARSALearner.update

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

Updates the value function

In the context of the base `ValueFunction` class, this is merely a placeholder. The specific update rule will depend on the specific value function desired.

Arguments:

- **x**: `ndarray((nstates,))` one-hot state vector
- **u**: `ndarray((nactions,))` one-hot action vector
- **r**: Scalar reward
- **x_**: `ndarray((nstates,))` one-hot next-state vector
- **u_**: `ndarray((nactions,))` one-hot next-action vector

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.action

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

Selects an action given the current state of environment.

The implementation will vary depending on the type of agent and environment.

Arguments:

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

Agent.learning

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

Updates the model's parameters.

The implementation will vary depending on the type of agent and environment.

Arguments:

- **state**: `ndarray((nstates,))` one-hot state vector
- **action**: `ndarray((nactions,))` one-hot action vector
- **reward**: scalar reward
- **next_state**: `ndarray((nstates,))` one-hot next-state vector
- **next_action**: `ndarray((nactions,))` one-hot action vector

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).

Arguments:

- **task**: `fitr.environments.Graph`

BanditAgent.action

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

Selects an action given the current state of environment.

The implementation will vary depending on the type of agent and environment.

Arguments:

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

BanditAgent.generate_data

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

For the parent agent, this function generates data from a bandit task

Arguments:

- **ntrials**: `int` number of trials

Returns:

`fitr.data.BehaviouralData`

BanditAgent.learning

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

Updates the model's parameters.

The implementation will vary depending on the type of agent and environment.

Arguments:

- **state**: `ndarray((nstates,))` one-hot state vector
- **action**: `ndarray((nactions,))` one-hot action vector
- **reward**: scalar reward
- **next_state**: `ndarray((nstates,))` one-hot next-state vector
- **next_action**: `ndarray((nactions,))` one-hot action vector

BanditAgent.log_prob

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

Computes the log-likelihood over actions for a given state under the present agent parameters.

Presently this only works for the state-action value function. In all other cases, you should define your own log-likelihood function. However, this can be used as a template.

Arguments:

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

Returns:

`ndarray((nactions,))` log-likelihood vector

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.

Arguments:

- **task**: `fitr.environments.Graph`

MDPAgent.action

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

Selects an action given the current state of environment.

The implementation will vary depending on the type of agent and environment.

Arguments:

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

MDPAgent.generate_data

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

For the parent agent, this function generates data from a Markov Decision Process (MDP) task

Arguments:

- **ntrials**: `int` number of trials

Returns:

`fitr.data.BehaviouralData`

MDPAgent.learning

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

Updates the model's parameters.

The implementation will vary depending on the type of agent and environment.

Arguments:

- **state**: `ndarray((nstates,))` one-hot state vector
- **action**: `ndarray((nactions,))` one-hot action vector
- **reward**: scalar reward
- **next_state**: `ndarray((nstates,))` one-hot next-state vector
- **next_action**: `ndarray((nactions,))` one-hot action vector

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)`

Selects an action given the current state of environment.

The implementation will vary depending on the type of agent and environment.

Arguments:

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

RandomBanditAgent.generate_data

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

For the parent agent, this function generates data from a bandit task

Arguments:

- **ntrials**: `int` number of trials

Returns:

`fitr.data.BehaviouralData`

RandomBanditAgent.learning

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

Updates the model's parameters.

The implementation will vary depending on the type of agent and environment.

Arguments:

- **state**: `ndarray((nstates,))` one-hot state vector
- **action**: `ndarray((nactions,))` one-hot action vector
- **reward**: scalar reward
- **next_state**: `ndarray((nstates,))` one-hot next-state vector
- **next_action**: `ndarray((nactions,))` one-hot action vector

RandomBanditAgent.log_prob

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

Computes the log-likelihood over actions for a given state under the present agent parameters.

Presently this only works for the state-action value function. In all other cases, you should define your own log-likelihood function. However, this can be used as a template.

Arguments:

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

Returns:

`ndarray((nactions,))` log-likelihood vector

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)`

Selects an action given the current state of environment.

The implementation will vary depending on the type of agent and environment.

Arguments:

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

RandomMDPAgent.generate_data

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

For the parent agent, this function generates data from a Markov Decision Process (MDP) task

Arguments:

- **ntrials**: `int` number of trials

Returns:

`fitr.data.BehaviouralData`

RandomMDPAgent.learning

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

Updates the model's parameters.

The implementation will vary depending on the type of agent and environment.

Arguments:

- **state**: `ndarray((nstates,))` one-hot state vector
- **action**: `ndarray((nactions,))` one-hot action vector
- **reward**: scalar reward
- **next_state**: `ndarray((nstates,))` one-hot next-state vector
- **next_action**: `ndarray((nactions,))` one-hot action vector

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

The softmax policy selects actions from a multinomial

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

whose parameters are

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

The value function is SARSA:

$$\mathbf{Q} \leftarrow \mathbf{Q} + \alpha (r + \gamma \mathbf{u}'^\top \mathbf{Q} \mathbf{x}' - \mathbf{u}^\top \mathbf{Q} \mathbf{x}) \mathbf{z},$$

where $0 < \alpha < 1$ is the learning rate, $0 \leq \gamma \leq 1$ is a discount factor, and where the reward prediction error (RPE) is $\delta = (r + \gamma \mathbf{u}'^\top \mathbf{Q} \mathbf{x}' - \mathbf{u}^\top \mathbf{Q} \mathbf{x})$. We have also included an eligibility trace $\mathbf{z}$ defined as

$$\mathbf{z} = \mathbf{u}\mathbf{x}^\top + \gamma\lambda\mathbf{z}$$

Arguments:

- **task**: `fitr.environments.Graph`
- **learning_rate**: Learning rate α
- **discount_factor**: Discount factor γ
- **trace_decay**: Eligibility trace decay λ
- **inverse_softmax_temp**: Inverse softmax temperature β
- **rng**: `np.random.RandomState`

SARSASoftmaxAgent.action

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

Selects an action given the current state of environment.

The implementation will vary depending on the type of agent and environment.

Arguments:

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

SARSASoftmaxAgent.generate_data

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

For the parent agent, this function generates data from a Markov Decision Process (MDP) task

Arguments:

- **ntrials**: `int` number of trials

Returns:

`fitr.data.BehaviouralData`

SARSASoftmaxAgent.learning

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

Updates the model's parameters.

The implementation will vary depending on the type of agent and environment.

Arguments:

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

- **action**: `ndarray((nactions,))` one-hot action vector
- **reward**: scalar reward
- **next_state**: `ndarray((nstates,))` one-hot next-state vector
- **next_action**: `ndarray((nactions,))` one-hot action vector

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

The softmax policy selects actions from a multinomial

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

whose parameters are

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

The value function is Q-learning:

$$\mathbf{Q} \leftarrow \mathbf{Q} + \alpha(r + \gamma \max_{\mathbf{u}'} \mathbf{u}'^\top \mathbf{Q}\mathbf{x}' - \mathbf{u}^\top \mathbf{Q}\mathbf{x})\mathbf{z},$$

where $0 < \alpha < 1$ is the learning rate, $0 \leq \gamma \leq 1$ is a discount factor, and where the reward prediction error (RPE) is $\delta = (r + \gamma \max_{\mathbf{u}'} \mathbf{u}'^\top \mathbf{Q}\mathbf{x}' - \mathbf{u}^\top \mathbf{Q}\mathbf{x})$. The eligibility trace $\mathbf{z}$ is defined as

$$\mathbf{z} = \mathbf{u}\mathbf{x}^\top + \gamma\lambda\mathbf{z}$$

Arguments:

- **task**: `fitr.environments.Graph`
- **learning_rate**: Learning rate α
- **discount_factor**: Discount factor γ

- **trace_decay**: Eligibility trace decay λ
- **inverse_softmax_temp**: Inverse softmax temperature β
- **rng**: `np.random.RandomState`

QLearningSoftmaxAgent.action

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

Selects an action given the current state of environment.

The implementation will vary depending on the type of agent and environment.

Arguments:

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

QLearningSoftmaxAgent.generate_data

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

For the parent agent, this function generates data from a Markov Decision Process (MDP) task

Arguments:

- **ntrials**: `int` number of trials

Returns:

`fitr.data.BehaviouralData`

QLearningSoftmaxAgent.learning

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

Updates the model's parameters.

The implementation will vary depending on the type of agent and environment.

Arguments:

- **state**: `ndarray((nstates,))` one-hot state vector
- **action**: `ndarray((nactions,))` one-hot action vector
- **reward**: scalar reward
- **next_state**: `ndarray((nstates,))` one-hot next-state vector
- **next_action**: `ndarray((nactions,))` one-hot action vector

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()`

An instrumental Rescorla-Wagner agent with a softmax policy

The softmax policy selects actions from a multinomial

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

whose parameters are

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

The value function is the Rescorla-Wagner learning rule:

$$\mathbf{Q} \leftarrow \mathbf{Q} + \alpha(r - \mathbf{u}^\top \mathbf{Q}\mathbf{x})\mathbf{u}\mathbf{x}^\top,$$

where $0 < \alpha < 1$ is the learning rate, $0 \leq \gamma \leq 1$ is a discount factor, and where the reward prediction error (RPE) is $\delta = (r - \mathbf{u}^\top \mathbf{Q}\mathbf{x})$.

Arguments:

- **task**: `fitr.environments.Graph`
- **learning_rate**: Learning rate α
- **inverse_softmax_temp**: Inverse softmax temperature β
- **rng**: `np.random.RandomState`

RWSoftmaxAgent.action

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

Selects an action given the current state of environment.

The implementation will vary depending on the type of agent and environment.

Arguments:

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

RWSoftmaxAgent.generate_data

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

For the parent agent, this function generates data from a bandit task

Arguments:

- **ntrials**: `int` number of trials

Returns:

`fitr.data.BehaviouralData`

RWSoftmaxAgent.learning

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

Updates the model's parameters.

The implementation will vary depending on the type of agent and environment.

Arguments:

- **state**: `ndarray((nstates,))` one-hot state vector
- **action**: `ndarray((nactions,))` one-hot action vector
- **reward**: scalar reward
- **next_state**: `ndarray((nstates,))` one-hot next-state vector
- **next_action**: `ndarray((nactions,))` one-hot action vector

RWSoftmaxAgent.log_prob

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

Computes the log-likelihood over actions for a given state under the present agent parameters.

Presently this only works for the state-action value function. In all other cases, you should define your own log-likelihood function. However, this can be used as a template.

Arguments:

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

Returns:

`ndarray((nactions,))` log-likelihood vector

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()`

An instrumental Rescorla-Wagner agent with a softmax policy, whose experienced reward is scaled by a factor ρ.

The softmax policy selects actions from a multinomial

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

whose parameters are

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

The value function is the Rescorla-Wagner learning rule with scaled reward ρr:

$$\mathbf{Q} \leftarrow \mathbf{Q} + \alpha(\rho r - \mathbf{u}^\top \mathbf{Q}\mathbf{x})\mathbf{u}\mathbf{x}^\top,$$

where $0 < \alpha < 1$ is the learning rate, $0 \leq \gamma \leq 1$ is a discount factor, and where the reward prediction error (RPE) is $\delta = (\rho r - \mathbf{u}^\top \mathbf{Q}\mathbf{x})$.

Arguments:

- **task**: `fitr.environments.Graph`
- **learning_rate**: Learning rate α
- **inverse_softmax_temp**: Inverse softmax temperature β
- **reward_sensitivity**: Reward sensitivity parameter ρ
- **rng**: `np.random.RandomState`

RWSoftmaxAgentRewardSensitivity.action

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

Selects an action given the current state of environment.

The implementation will vary depending on the type of agent and environment.

Arguments:

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

RWSoftmaxAgentRewardSensitivity.generate_data

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

For the parent agent, this function generates data from a bandit task

Arguments:

- **ntrials**: `int` number of trials

Returns:

`fitr.data.BehaviouralData`

RWSoftmaxAgentRewardSensitivity.learning

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

Updates the model's parameters.

The implementation will vary depending on the type of agent and environment.

Arguments:

- **state**: `ndarray((nstates,))` one-hot state vector
- **action**: `ndarray((nactions,))` one-hot action vector
- **reward**: scalar reward
- **next_state**: `ndarray((nstates,))` one-hot next-state vector
- **next_action**: `ndarray((nactions,))` one-hot action vector

RWSoftmaxAgentRewardSensitivity.log_prob

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

Computes the log-likelihood over actions for a given state under the present agent parameters.

Presently this only works for the state-action value function. In all other cases, you should define your own log-likelihood function. However, this can be used as a template.

Arguments:

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

Returns:

`ndarray((nactions,))` log-likelihood vector

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)

Chapter 5

Data

`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.

Chapter 6

Inference

`fitr.inference`

Methods for inferring the parameters of generative models for reinforcement learning data.

OptimizationResult

`fitr.inference.optimization_result.OptimizationResult()`

Container for the results of an optimization run on a generative model of behavioural data

Arguments:

- **subject_id**: `ndarray((nsubjects,))` or `None` (default). Integer ids for subjects
- **xmin**: `ndarray((nsubjects,nparams))` or `None` (default). Parameters that minimize objective function
- **fmin**: `ndarray((nsubjects,))` or `None` (default). Value of objective function at minimum
- **fevals**: `ndarray((nsubjects,))` or `None` (default). Number of function evaluations required to minimize objective function
- **niters**: `ndarray((nsubjects,))` or `None` (default). Number of iterations required to minimize objective function
- **lme**: `ndarray((nsubjects,))` or `None` (default). Log model evidence
- **bic**: `ndarray((nsubjects,))` or `None` (default). Bayesian Information Criterion
- **hess_inv**: `ndarray((nsubjects,nparams,nparams))` or `None` (default). Inverse Hessian at the optimum.
- **err**: `ndarray((nsubjects,nparams))` or `None` (default). Error of estimates at optimum.

OptimizationResult.transform_xmin

`fitr.inference.optimization_result.transform_xmin(self, transforms, inplace=False)`

Rescales the parameter estimates.

Arguments:

- **transforms**: `list`. Transformation functions where `len(transforms) == self.xmin.shape[1]`
- **inplace**: `bool`. Whether to change the values in `self.xmin`. Default is `False`, which returns an `ndarray((nsubjects, nparams))` of the transformed parameters.

Returns:

`ndarray((nsubjects, nparams))` of the transformed parameters if `inplace=False`

---

mlepar

```
fitr.inference.mle_parallel.mlepar(f, data, nparams, minstarts=2, maxstarts=10, init
```

Computes maximum likelihood estimates using parallel CPU resources.

Wraps over the `fitr.optimization.mle_parallel.mle` function.

Arguments:

- **f**: Likelihood function
- **data**: A subscriptable object whose first dimension indexes subjects
- **optimizer**: Optimization function (currently only `l_bfgs_b` supported)
- **nparams**: `int` number of parameters to be estimated
- **minstarts**: `int`. Minimum number of restarts with new initial values
- **maxstarts**: `int`. Maximum number of restarts with new initial values
- **init_sd**: Standard deviation for Gaussian initial values

Returns:

`fitr.inference.OptimizationResult`

---

l_bfgs_b

```
fitr.inference.mle_parallel.l_bfgs_b(f, i, data, nparams, minstarts=2, maxstarts=10,
```

Minimizes the negative log-probability of data with respect to some parameters under function `f` using the L-BFGS-B algorithm.

This function is specified for use with parallel CPU resources.

Arguments:

- **f**: Log likelihood function
- **i**: `int`. Subject being optimized (slices first dimension of `data`)
- **data**: Object subscriptable along first dimension to indicate subject being optimized
- **nparams**: `int`. Number of parameters in the model
- **minstarts**: `int`. Minimum number of restarts with new initial values
- **maxstarts**: `int`. Maximum number of restarts with new initial values
- **init_sd**: Standard deviation for Gaussian initial values

Returns:

- **i**: `int`. Subject being optimized (slices first dimension of `data`)
- **xmin**: `ndarray((nparams,))`. Parameter values at optimum
- **fmin**: Scalar objective function value at optimum
- **fevals**: `int`. Number of function evaluations
- **niters**: `int`. Number of iterations
- **lme_**: Scalar log-model evidence at optimum
- **bic_**: Scalar Bayesian Information Criterion at optimum
- **hess_inv**: `ndarray((nparams, nparams))`. Inv at optimum

Chapter 7

Criticism

fitr.criticism

Methods for criticism of model fits.

actual_estimate

`fitr.criticism.plotting.actual_estimate(y_true, y_pred, xlabel='Actual', ylabel='Est`

Plots parameter estimates against the ground truth values.

Arguments:

- **y_true**: `ndarray(nsamples)`. Vector of ground truth parameters
- **y_pred**: `ndarray(nsamples)`. Vector of parameter estimates
- **xlabel**: `str`. Label for x-axis
- **ylabel**: `str`. Label for y-axis
- **corr**: `bool`. Whether to plot correlation coefficient.
- **figsize**: `tuple`. Figure size (inches).

Returns:

`matplotlib.pyplot.Figure`

Chapter 8

Metrics

fitr.metrics

Metrics and performance statistics.

bic

`fitr.metrics.bic(log_prob, nparams, ntrials)`

Bayesian Information Criterion (BIC)

Arguments:

- **log_prob**: Log probability
- **nparams**: Number of parameters in the model
- **ntrials**: Number of trials in the time series

Returns:

Scalar estimate of BIC.

---

linear_correlation

`fitr.metrics.linear_correlation(X, Y)`

Linear correlation coefficient.

Will compute the following formula

$$\rho = \frac{\mathbf{x}^\top \mathbf{y}}{\|\mathbf{x}Vert \cdot \|\mathbf{y}Vert}$$

where each vector $\mathbf{x}$ and $\mathbf{y}$ are rows of the matrices $\mathbf{X}$ and $\mathbf{Y}$, respectively.

Arguments:

- **X**: `ndarray((nsamples, nfeatures))` of dimension 1 or 2. If `X` is a 1D array, it will be converted to 2D prior to computation
- **Y**: `ndarray((nsamples, nfeatures))` of dimension 1 or 2. If `Y` is a 1D array, it will be converted to 2D prior to computation

Returns:

- **rho**: `ndarray((nfeatures,))`. Correlation coefficient(s)

TODO:

- [ ] Create error raised when X and Y are not same dimension

---

lme

`fitr.metrics.lme(log_prob, nparams, hess_inv)`

Laplace approximation to the log model evidence

Arguments:

- **log_prob**: Log probability
- **nparams**: Number of parameters in the model
- **hess_inv**: Hessian at the optimum (shape is $K \times K$)

Returns:

Scalar approximation of the log model evidence

---

log_loss

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

Computes log loss.

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

Arguments:

- **p**: Binary vector of true labels `ndarray((nsamples,))`
- **q**: Vector of estimates (between 0 and 1) of type `ndarray((nsamples,))`

Returns:

Scalar log loss

Chapter 9

Utilities

`fitr.utils`

Functions used across `fitr`.

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`

---

relu

`fitr.utils.relu(x, a_max=None)`

Rectified linearity

$$\mathbf{x}' = \max(x_i, 0)_{i=1}^{|\mathbf{x}|}$$

Arguments:

- **x**: Vector of inputs

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

Returns:

Exponentiated values of `x`.

---

scale_data

`fitr.utils.scale_data(X, axis=0, with_mean=True, with_var=True)`

Rescales data by subtracting mean and dividing by variance

$$\mathbf{x}' = \frac{\mathbf{x} - \frac{1}{n}\mathbf{1}^\top\mathbf{x}}{Var(\mathbf{x})}$$

Arguments:

- **X**: `ndarray((nsamples, [nfeatures]))`. Data. May be 1D or 2D.
- **with_mean**: `bool`. Whether to subtract the mean
- **with_var**: `bool`. Whether to divide by variance

Returns:

`ndarray(X.shape)`. Rescaled data.

---

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`

---

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,))`

---

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`.
