`fitr`

Abraham Nunes

October 30, 2018

Contents

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

```
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: Actual vs. Estimate Plots
lr_fig  = actual_estimate(data.params[:,1], X[:,0]); plt.show()
ist_fig = actual_estimate(data.params[:,2], X[:,1]); plt.show()
```

Simulating and Fitting Data from a Random Contextual Bandit Task

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

class MyBanditTask(RandomContextualBandit):
    def __init__(self):
        super().__init__(nactions=4,
                         noutcomes=3,
                         nstates=4,
                         min_actions_per_context=None,
                         alpha=0.1,
                         alpha_start=1.,
                         shift_flip='shift',
                         reward_lb=-1,
                         reward_ub=1,
                         reward_drift='on',
                         drift_mu=np.zeros(3),
                         drift_sd=1.)

data = generate_behavioural_data(MyBanditTask, RWSoftmaxAgent, 20, 200)

def log_prob(w, D):
    agent = RWSoftmaxAgent(task=MyBanditTask(),
                           learning_rate=w[0],
                           inverse_softmax_temp=w[1])
    L=0
    for t in range(D.shape[0]):
        x=D[t,:7]; u=D[t,7:11]; r=D[t,11]; x_=D[t,12:]
```

```
        L += u@agent.log_prob(x)
        agent.learning(x, u, r, x_, None)
    return L

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

# Criticism: Actual vs. Estimate Plots
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.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
- $\mathcal{J} : \mathcal{X} \rightarrow \mathcal{R}$, 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.graph.adjacency_matrix_decomposition(self)
```

Singular value decomposition of the graph adjacency matrix

Graph.get_graph_depth

```
fitr.environments.graph.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.graph.laplacian_matrix_decomposition(self)
```

Singular value decomposition of the graph Laplacian

Graph.make_action_labels

```
fitr.environments.graph.make_action_labels(self)
```

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

---

Graph.make_digraph

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

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

---

Graph.make_undirected_graph

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

Converts the DiGraph to undirected and computes some stats

Graph.observation

```
fitr.environments.graph.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.graph.plot_action_outcome_probabilities(self, figsize=None, outfil
```

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.graph.plot_graph(self, figsize=None, node_size=2000, arrowsize=20,
```

Plots the directed graph of the task

Graph.plot_spectral_properties

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

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

Graph.random_action

```
fitr.environments.graph.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.set_seed

`fitr.environments.graph.set_seed(self, seed=None)`

Allows user to specify a seed for the pseudorandom number generator.

Arguments:

- **seed**: `int`. Seed value. Default is `None`, which results in a default random state object. If user enters a non-integer value, the default random state object will still be used and no error will be thrown!

Graph.step

`fitr.environments.graph.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.TwoArmedBandit()`

A simple 2-armed bandit task

TwoArmedBandit.adjacency_matrix_decomposition

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

Singular value decomposition of the graph adjacency matrix

TwoArmedBandit.get_graph_depth

`fitr.environments.graph.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.graph.laplacian_matrix_decomposition(self)`

Singular value decomposition of the graph Laplacian

TwoArmedBandit.make_action_labels

`fitr.environments.graph.make_action_labels(self)`

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

---

TwoArmedBandit.make_digraph

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

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

---

TwoArmedBandit.make_undirected_graph

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

Converts the DiGraph to undirected and computes some stats

TwoArmedBandit.observation

```
fitr.environments.graph.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.graph.plot_action_outcome_probabilities(self, figsize=None, outfil
```

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.graph.plot_graph(self, figsize=None, node_size=2000, arrowsize=20,
```

Plots the directed graph of the task

TwoArmedBandit.plot_spectral_properties

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

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

TwoArmedBandit.random_action

```
fitr.environments.graph.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.set_seed

```
fitr.environments.graph.set_seed(self, seed=None)
```

Allows user to specify a seed for the pseudorandom number generator.

Arguments:

- **seed**: `int`. Seed value. Default is `None`, which results in a default random state object. If user enters a non-integer value, the default random state object will still be used and no error will be thrown!

TwoArmedBandit.step

```
fitr.environments.graph.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.orthogonal_gonogo.OrthogonalGoNoGo()`

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

OrthogonalGoNoGo.adjacency_matrix_decomposition

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

Singular value decomposition of the graph adjacency matrix

OrthogonalGoNoGo.get_graph_depth

`fitr.environments.graph.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.graph.laplacian_matrix_decomposition(self)`

Singular value decomposition of the graph Laplacian

OrthogonalGoNoGo.make_action_labels

`fitr.environments.graph.make_action_labels(self)`

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

---

OrthogonalGoNoGo.make_digraph

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

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

---

OrthogonalGoNoGo.make_undirected_graph

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

Converts the DiGraph to undirected and computes some stats

OrthogonalGoNoGo.observation

```
fitr.environments.graph.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.graph.plot_action_outcome_probabilities(self, figsize=None, outfil
```

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.graph.plot_graph(self, figsize=None, node_size=2000, arrowsize=20,
```

Plots the directed graph of the task

OrthogonalGoNoGo.plot_spectral_properties

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

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

OrthogonalGoNoGo.random_action

```
fitr.environments.graph.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.set_seed

```
fitr.environments.graph.set_seed(self, seed=None)
```

Allows user to specify a seed for the pseudorandom number generator.

Arguments:

- **seed**: `int`. Seed value. Default is `None`, which results in a default random state object. If user enters a non-integer value, the default random state object will still be used and no error will be thrown!

OrthogonalGoNoGo.step

`fitr.environments.graph.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.

DawTwoStep

`fitr.environments.dawtwostep.DawTwoStep()`

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

---

DawTwoStep.adjacency_matrix_decomposition

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

Singular value decomposition of the graph adjacency matrix

DawTwoStep.f_reward

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

DawTwoStep.get_graph_depth

`fitr.environments.graph.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

---

DawTwoStep.laplacian_matrix_decomposition

`fitr.environments.graph.laplacian_matrix_decomposition(self)`

Singular value decomposition of the graph Laplacian

DawTwoStep.make_action_labels

`fitr.environments.graph.make_action_labels(self)`

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

---

DawTwoStep.make_digraph

`fitr.environments.graph.make_digraph(self)`

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

DawTwoStep.make_state_labels

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

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

---

DawTwoStep.make_undirected_graph

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

Converts the DiGraph to undirected and computes some stats

DawTwoStep.observation

```
fitr.environments.graph.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()
```

DawTwoStep.plot_action_outcome_probabilities

```
fitr.environments.graph.plot_action_outcome_probabilities(self, figsize=None, outfil
```

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.

DawTwoStep.plot_graph

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

Plots the directed graph of the task

DawTwoStep.plot_reward_paths

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

DawTwoStep.plot_spectral_properties

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

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

DawTwoStep.random_action

```
fitr.environments.graph.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()
```

DawTwoStep.set_seed

```
fitr.environments.graph.set_seed(self, seed=None)
```

Allows user to specify a seed for the pseudorandom number generator.

Arguments:

- **seed**: `int`. Seed value. Default is `None`, which results in a default random state object. If user enters a non-integer value, the default random state object will still be used and no error will be thrown!

DawTwoStep.step

```
fitr.environments.graph.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.

KoolTwoStep

`fitr.environments.kooltwostep.KoolTwoStep()`

From Kool & Gershman 2016.

KoolTwoStep.adjacency_matrix_decomposition

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

Singular value decomposition of the graph adjacency matrix

KoolTwoStep.f_reward

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

KoolTwoStep.get_graph_depth

`fitr.environments.graph.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

---

KoolTwoStep.laplacian_matrix_decomposition

`fitr.environments.graph.laplacian_matrix_decomposition(self)`

Singular value decomposition of the graph Laplacian

KoolTwoStep.make_action_labels

`fitr.environments.graph.make_action_labels(self)`

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

---

KoolTwoStep.make_digraph

```
fitr.environments.graph.make_digraph(self)
```

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

KoolTwoStep.make_state_labels

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

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

---

KoolTwoStep.make_undirected_graph

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

Converts the DiGraph to undirected and computes some stats

KoolTwoStep.observation

```
fitr.environments.graph.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()
```

KoolTwoStep.plot_action_outcome_probabilities

```
fitr.environments.graph.plot_action_outcome_probabilities(self, figsize=None, outfil
```

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.

KoolTwoStep.plot_graph

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

Plots the directed graph of the task

KoolTwoStep.plot_spectral_properties

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

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

KoolTwoStep.random_action

```
fitr.environments.graph.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()
```

KoolTwoStep.set_seed

```
fitr.environments.graph.set_seed(self, seed=None)
```

Allows user to specify a seed for the pseudorandom number generator.

Arguments:

- **seed**: `int`. Seed value. Default is `None`, which results in a default random state object. If user enters a non-integer value, the default random state object will still be used and no error will be thrown!

KoolTwoStep.step

`fitr.environments.graph.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.

MouthTask

`fitr.environments.mouthtask.MouthTask()`

The Pizzagalli reward sensitivity signal-detection task

MouthTask.adjacency_matrix_decomposition

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

Singular value decomposition of the graph adjacency matrix

MouthTask.get_graph_depth

`fitr.environments.graph.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

---

MouthTask.laplacian_matrix_decomposition

`fitr.environments.graph.laplacian_matrix_decomposition(self)`

Singular value decomposition of the graph Laplacian

MouthTask.make_action_labels

`fitr.environments.graph.make_action_labels(self)`

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

---

MouthTask.make_digraph

`fitr.environments.graph.make_digraph(self)`

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

MouthTask.make_state_labels

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

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

---

MouthTask.make_undirected_graph

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

Converts the DiGraph to undirected and computes some stats

MouthTask.observation

`fitr.environments.graph.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()
```

MouthTask.plot_action_outcome_probabilities

```
fitr.environments.graph.plot_action_outcome_probabilities(self, figsize=None, outfil
```

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.

MouthTask.plot_graph

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

Plots the directed graph of the task

MouthTask.plot_spectral_properties

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

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

MouthTask.random_action

```
fitr.environments.graph.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()
```

MouthTask.set_seed

`fitr.environments.graph.set_seed(self, seed=None)`

Allows user to specify a seed for the pseudorandom number generator.

Arguments:

- **seed**: `int`. Seed value. Default is `None`, which results in a default random state object. If user enters a non-integer value, the default random state object will still be used and no error will be thrown!

MouthTask.step

`fitr.environments.graph.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.

IGT

`fitr.environments.igt.IGT()`

Iowa Gambling Task

IGT.adjacency_matrix_decomposition

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

Singular value decomposition of the graph adjacency matrix

IGT.get_graph_depth

`fitr.environments.graph.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

---

IGT.laplacian_matrix_decomposition

`fitr.environments.graph.laplacian_matrix_decomposition(self)`

Singular value decomposition of the graph Laplacian

IGT.make_action_labels

`fitr.environments.graph.make_action_labels(self)`

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

---

IGT.make_digraph

`fitr.environments.graph.make_digraph(self)`

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

IGT.make_state_labels

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

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

---

IGT.make_undirected_graph

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

Converts the DiGraph to undirected and computes some stats

IGT.observation

```
fitr.environments.graph.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()
```

IGT.plot_action_outcome_probabilities

```
fitr.environments.graph.plot_action_outcome_probabilities(self, figsize=None, outfil
```

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.

IGT.plot_graph

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

Plots the directed graph of the task

IGT.plot_spectral_properties

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

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

IGT.random_action

```
fitr.environments.graph.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()
```

IGT.set_seed

```
fitr.environments.graph.set_seed(self, seed=None)
```

Allows user to specify a seed for the pseudorandom number generator.

Arguments:

- **seed**: `int`. Seed value. Default is `None`, which results in a default random state object. If user enters a non-integer value, the default random state object will still be used and no error will be thrown!

IGT.step

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

Singular value decomposition of the graph adjacency matrix

RandomContextualBandit.f_reward

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

RandomContextualBandit.get_graph_depth

```
fitr.environments.graph.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.graph.laplacian_matrix_decomposition(self)`

Singular value decomposition of the graph Laplacian

RandomContextualBandit.make_action_labels

`fitr.environments.graph.make_action_labels(self)`

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

---

RandomContextualBandit.make_digraph

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

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

---

RandomContextualBandit.make_undirected_graph

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

Converts the DiGraph to undirected and computes some stats

RandomContextualBandit.observation

`fitr.environments.graph.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.graph.plot_action_outcome_probabilities(self, figsize=None, outfil
```

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.graph.plot_graph(self, figsize=None, node_size=2000, arrowsize=20,
```

Plots the directed graph of the task

RandomContextualBandit.plot_spectral_properties

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

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

RandomContextualBandit.random_action

```
fitr.environments.graph.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.set_seed

```
fitr.environments.graph.set_seed(self, seed=None)
```

Allows user to specify a seed for the pseudorandom number generator.

Arguments:

- **seed**: `int`. Seed value. Default is `None`, which results in a default random state object. If user enters a non-integer value, the default random state object will still be used and no error will be thrown!

RandomContextualBandit.step

```
fitr.environments.graph.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 $\mathbf{u}$, in addition to computing derivatives up to second order

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

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((nactions,))` action value vector

Returns:

`ndarray((nactions,))` 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((nactions,))`

Returns:

Scalar log-probability

StickySoftmaxPolicy.sample

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

Samples from the action distribution

Arguments:

- **x**: `ndarray((nactions,))` action value vector

Returns:

`ndarray((nactions,))` 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 $\mathbf{Q} = \mathcal{Q}(\mathbf{x}, \mathbf{u})$
- `rpe`: Reward prediction error history
- `etrace`: An eligibility trace (optional)
- `dV`: A dictionary storing gradients with respect to parameters (named keys)
- `dQ`: A dictionary storing gradients with respect to parameters (named keys)

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.grad_Qx

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

Compute gradient of action values for a given state

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

where the gradient is defined as

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

Arguments:

- $\mathbf{x}$: `ndarray((nstates,))` one-hot state vector

Returns:

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

ValueFunction.grad_Vx

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

Compute the gradient of state value function with respect to parameters $\mathbf{v}$

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

where the gradient is defined as

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

Arguments:

- $\mathbf{x}$: `ndarray((nstates,))` one-hot state vector

Returns:

Scalar value of state $\mathbf{x}$

ValueFunction.grad_uQx

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

Compute derivative of value of taking action $\mathbf{u}$ in state $\mathbf{x}$ with respect to value function parameters $\mathbf{Q}$

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

where the derivative is defined as

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

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.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.grad_Qx

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

Compute gradient of action values for a given state

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

where the gradient is defined as

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

Arguments:

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

Returns:

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

DummyLearner.grad_Vx

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

Compute the gradient of state value function with respect to parameters $\mathbf{v}$

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

where the gradient is defined as

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

Arguments:

- $\mathbf{x}$: `ndarray((nstates,))` one-hot state vector

Returns:

Scalar value of state $\mathbf{x}$

DummyLearner.grad_uQx

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

Compute derivative of value of taking action $\mathbf{u}$ in state $\mathbf{x}$ with respect to value function parameters $\mathbf{Q}$

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

where the derivative is defined as

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

Arguments:

- $\mathbf{u}$: `ndarray((nactions,))` one-hot action vector
- $\mathbf{x}$: `ndarray((nstates,))` one-hot state vector

Returns:

Scalar value of action $\mathbf{u}$ in 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.grad_Qx

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

Compute gradient of action values for a given state

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

where the gradient is defined as

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

Arguments:

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

Returns:

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

InstrumentalRescorlaWagnerLearner.grad_Vx

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

Compute the gradient of state value function with respect to parameters $\mathbf{v}$

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

where the gradient is defined as

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

Arguments:

- $\mathbf{x}$: `ndarray((nstates,))` one-hot state vector

Returns:

Scalar value of state $\mathbf{x}$

InstrumentalRescorlaWagnerLearner.grad_uQx

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

Compute derivative of value of taking action $\mathbf{u}$ in state $\mathbf{x}$ with respect to value function parameters $\mathbf{Q}$

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

where the derivative is defined as

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

Arguments:

- $\mathbf{u}$: `ndarray((nactions,))` one-hot action vector
- $\mathbf{x}$: `ndarray((nstates,))` one-hot state vector

Returns:

Scalar value of action $\mathbf{u}$ in 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_)
```

Computes the value function update of the instrumental Rescorla-Wagner learning rule and computes derivative with respect to the learning rate.

This derivative is defined as

$$\frac{\partial}{\partial \alpha} \mathcal{Q}(\mathbf{x}, \mathbf{u}; \alpha) = \delta \mathbf{u}\mathbf{x}^\top + \frac{\partial}{\partial \alpha} \mathcal{Q}(\mathbf{x}, \mathbf{u}; \alpha)(1 - \alpha \mathbf{u}\mathbf{x}^\top)$$

and the second order derivative with respect to learning rate is

$$\frac{\partial}{\partial \alpha} \mathcal{Q}(\mathbf{x}, \mathbf{u}; \alpha) = -2\mathbf{u}\mathbf{x}^\top \partial_\alpha \mathcal{Q}(\mathbf{x}, \mathbf{u}; \alpha) + \partial^2_\alpha \mathcal{Q}(\mathbf{x}, \mathbf{u}; \alpha)(1 - \alpha \mathbf{u}\mathbf{x}^\top)$$

Arguments:

- **x**: `ndarray((nstates, ))`. State vector
- **u**: `ndarray((nactions, ))`. Action vector
- **r**: `float`. Reward received
- **x_**: `ndarray((nstates, ))`. For compatibility
- **u_**: `ndarray((nactions, ))`. For compatibility

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.grad_Qx

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

Compute gradient of action values for a given state

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

where the gradient is defined as

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

Arguments:

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

Returns:

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

QLearner.grad_Vx

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

Compute the gradient of state value function with respect to parameters $\mathbf{v}$

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

where the gradient is defined as

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

Arguments:

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

Returns:

Scalar value of state $\mathbf{x}$

QLearner.grad_uQx

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

Compute derivative of value of taking action $\mathbf{u}$ in state $\mathbf{x}$ with respect to value function parameters $\mathbf{Q}$

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

where the derivative is defined as

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

Arguments:

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

- $\mathbf{x}$: `ndarray((nstates,))` one-hot state vector

Returns:

Scalar value of action $\mathbf{u}$ in 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:

- $\mathbf{u}$: `ndarray((nactions,))` one-hot action vector
- $\mathbf{x}$: `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_)`

Computes value function updates and their derivatives for the Q-learning model

---

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.grad_Qx

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

Compute gradient of action values for a given state

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

where the gradient is defined as

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

Arguments:

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

Returns:

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

SARSALearner.grad_Vx

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

Compute the gradient of state value function with respect to parameters $\mathbf{v}$

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

where the gradient is defined as

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

Arguments:

- $\mathbf{x}$: `ndarray((nstates,))` one-hot state vector

Returns:

Scalar value of state $\mathbf{x}$

SARSALearner.grad_uQx

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

Compute derivative of value of taking action $\mathbf{u}$ in state $\mathbf{x}$ with respect to value function parameters $\mathbf{Q}$

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

where the derivative is defined as

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

Arguments:

- $\mathbf{u}$: `ndarray((nactions,))` one-hot action vector
- $\mathbf{x}$: `ndarray((nstates,))` one-hot state vector

Returns:

Scalar value of action $\mathbf{u}$ in 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_)`

Computes value function updates and their derivatives for the SARSA model

---

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 and computes gradients

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, state_only=False)
```

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

Arguments:

- **state_only**: `bool`. If the eligibility trace is only an `nstate` dimensional vector (i.e. for a Pavlovian conditioning model) then set to `True`. For instumental models, the eligibility trace should be an `nactions` by `nstates` matrix, so keep this to `False` in that case.

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 and computes gradients

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, state_only=False)`

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

Arguments:

- **state_only**: `bool`. If the eligibility trace is only an `nstate` dimensional vector (i.e. for a Pavlovian conditioning model) then set to `True`. For instumental models, the eligibility trace should be an `nactions` by `nstates` matrix, so keep this to `False` in that case.

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, state_only=False)`

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

Arguments:

- **ntrials**: `int` number of trials
- **state_only**: `bool`. If the eligibility trace is only an `nstate` dimensional vector (i.e. for a Pavlovian conditioning model) then set to `True`. For instumental models, the eligibility trace should be an `nactions` by `nstates` matrix, so keep this to `False` in that case.

Returns:

`fitr.data.BehaviouralData`

MDPAgent.learning

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

Updates the model's parameters and computes gradients

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, state_only=False)
```

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

Arguments:

- **state_only**: `bool`. If the eligibility trace is only an `nstate` dimensional vector (i.e. for a Pavlovian conditioning model) then set to `True`. For instumental models, the eligibility trace should be an `nactions` by `nstates` matrix, so keep this to `False` in that case.

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 and computes gradients

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, state_only=False)`

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

Arguments:

- **state_only**: `bool`. If the eligibility trace is only an `nstate` dimensional vector (i.e. for a Pavlovian conditioning model) then set to `True`. For instumental models, the eligibility trace should be an `nactions` by `nstates` matrix, so keep this to `False` in that case.

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, state_only=False)`

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

Arguments:

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

- **state_only**: `bool`. If the eligibility trace is only an `nstate` dimensional vector (i.e. for a Pavlovian conditioning model) then set to `True`. For instumental models, the eligibility trace should be an `nactions` by `nstates` matrix, so keep this to `False` in that case.

Returns:

`fitr.data.BehaviouralData`

RandomMDPAgent.learning

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

Updates the model's parameters and computes gradients

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, state_only=False)
```

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

Arguments:

- **state_only**: `bool`. If the eligibility trace is only an `nstate` dimensional vector (i.e. for a Pavlovian conditioning model) then set to `True`. For instumental models, the eligibility trace should be an `nactions` by `nstates` matrix, so keep this to `False` in that case.

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, state_only=False)
```

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

Arguments:

- **ntrials**: `int` number of trials
- **state_only**: `bool`. If the eligibility trace is only an `nstate` dimensional vector (i.e. for a Pavlovian conditioning model) then set to `True`. For instumental models, the eligibility trace should be an `nactions` by `nstates` matrix, so keep this to `False` in that case.

Returns:

`fitr.data.BehaviouralData`

SARSASoftmaxAgent.learning

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

Updates the model's parameters and computes gradients

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.log_prob

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

Computes the log-probability of the given action and state under the model, while also computing first and second order derivatives.

This model has four free parameters:

- Learning rate α
- Inverse softmax temperature β
- Discount factor γ
- Trace decay λ

First-order partial derivatives

We can break down the computation using the chain rule to reuse previously computed derivatives:

$$\frac{\partial \mathcal{L}}{\partial \alpha} = \frac{\partial \mathcal{L}}{\partial \boldsymbol{\pi}} \frac{\partial \boldsymbol{\pi}}{\partial \mathbf{q}} \frac{\partial \mathbf{q}}{\partial \mathbf{Q}} \frac{\partial \mathbf{Q}}{\partial \alpha}$$

$$\frac{\partial \mathcal{L}}{\partial \beta} = \frac{\partial \mathcal{L}}{\partial \boldsymbol{\pi}} \frac{\partial \boldsymbol{\pi}}{\partial \beta}$$

$$\frac{\partial \mathcal{L}}{\partial \gamma} = \frac{\partial \mathcal{L}}{\partial \boldsymbol{\pi}} \frac{\partial \boldsymbol{\pi}}{\partial \mathbf{q}} \frac{\partial \mathbf{q}}{\partial \mathbf{Q}} \frac{\partial \mathbf{Q}}{\partial \gamma}$$

$$\frac{\partial \mathcal{L}}{\partial \lambda} = \frac{\partial \mathcal{L}}{\partial \boldsymbol{\pi}} \frac{\partial \boldsymbol{\pi}}{\partial \mathbf{q}} \frac{\partial \mathbf{q}}{\partial \mathbf{Q}} \frac{\partial \mathbf{Q}}{\partial \lambda}$$

Action Probabilities

$$\partial_\alpha \varsigma = \frac{\partial \varsigma}{\partial \boldsymbol{\pi}} \frac{\partial \boldsymbol{\pi}}{\partial \mathbf{q}} \frac{\partial \mathbf{q}}{\partial \mathbf{Q}} (\partial_\alpha \mathbf{Q}) = \beta \big(\partial_{\boldsymbol{\pi}} \varsigma\big)_i \big(\partial_\alpha Q\big)^i_j x^j$$

Value Function

$$\partial_\alpha Q_{ij} = \partial_\alpha Q_{ij} + (\delta + \alpha \partial_\alpha \delta) z_{ij}$$

$$\partial_\gamma Q_{ij} = \partial_\gamma Q_{ij} + \alpha\big((\partial_\gamma \delta) z_{ij} + \delta(\partial_\gamma z_{ij})\big)$$

$$\partial_\lambda Q_{ij} = \partial_\lambda Q_{ij} + \alpha\big((\partial_\lambda \delta) z_{ij} + \delta(\partial_\lambda z_{ij})\big)$$

Reward Prediction Error

$$\partial_\alpha \delta = (\partial_{\mathbf{Q}} \delta)_{ij} (\partial_\alpha Q)^{ij}$$

$$\partial_\gamma \delta = (\partial_{\mathbf{Q}} \delta)_{ij} (\partial_\gamma Q)^{ij} + \tilde{u}_i Q^i_j \tilde{x}^j$$

$$\partial_\lambda \delta = (\partial_{\mathbf{Q}} \delta)_{ij} (\partial_\lambda Q)^{ij}$$

Trace Decay

$$\partial_\gamma z_{ij} = \lambda\big(z_{ij} + \gamma(\partial_\gamma z_{ij})\big)$$

$$\partial_\lambda z_{ij} = \gamma\big(z_{ij} + \lambda(\partial_\lambda z_{ij})\big)$$

Simplified Components of the Gradient Vector

$$\frac{\partial \mathcal{L}}{\partial \alpha} = \beta\big[\mathbf{u} - \varsigma(\boldsymbol{\pi})\big]_i \big(\partial_\alpha Q\big)^i_j x^j = \beta\big[u_i (\partial_\alpha Q)^i_j x^j - p(u_i)(\partial_\alpha Q)^i_j x^j\big]$$

$$\frac{\partial \mathcal{L}}{\partial \beta} = \big[\mathbf{u} - \varsigma(\boldsymbol{\pi})\big]_i Q^i_j x^j = u_i Q^i_j x^j - p(u_i) Q^i_j x^j$$

$$\frac{\partial \mathcal{L}}{\partial \gamma} = \beta\big[\mathbf{u} - \varsigma(\boldsymbol{\pi})\big]_i \big(\partial_\gamma Q\big)^i_j x^j$$

$$\frac{\partial \mathcal{L}}{\partial \lambda} = \beta\big[\mathbf{u} - \varsigma(\boldsymbol{\pi})\big]_i \big(\partial_\lambda Q\big)^i_j x^j$$

Second-Order Partial Derivatives

The Hessian matrix for this model is

$$\mathbf{H} = \begin{bmatrix} \frac{\partial^2 \mathcal{L}}{\partial \alpha^2} & \frac{\partial^2 \mathcal{L}}{\partial \alpha \partial \beta} & \frac{\partial^2 \mathcal{L}}{\partial \alpha \partial \gamma} & \frac{\partial^2 \mathcal{L}}{\partial \alpha \partial \lambda} \\ \frac{\partial^2 \mathcal{L}}{\partial \beta \partial \alpha} & \frac{\partial^2 \mathcal{L}}{\partial \beta^2} & \frac{\partial^2 \mathcal{L}}{\partial \beta \partial \gamma} & \frac{\partial^2 \mathcal{L}}{\partial \beta \partial \lambda} \\ \frac{\partial^2 \mathcal{L}}{\partial \gamma \partial \alpha} & \frac{\partial^2 \mathcal{L}}{\partial \gamma \partial \beta} & \frac{\partial^2 \mathcal{L}}{\partial \gamma^2} & \frac{\partial^2 \mathcal{L}}{\partial \gamma \partial \lambda} \\ \frac{\partial^2 \mathcal{L}}{\partial \lambda \partial \alpha} & \frac{\partial^2 \mathcal{L}}{\partial \lambda \partial \beta} & \frac{\partial^2 \mathcal{L}}{\partial \lambda \partial \gamma} & \frac{\partial^2 \mathcal{L}}{\partial \lambda^2} \end{bmatrix},$$

where the second-order partial derivatives are such that $\mathbf{H}$ is symmetrical. We must therefore compute 10 second order partial derivatives, shown below:

$$\frac{\partial^2 \mathcal{L}}{\partial \alpha^2} = \beta \Big[(\mathbf{u} - \varsigma(\boldsymbol{\pi}))_i (\partial_\alpha^2 Q)^i - (\partial_\alpha \varsigma)_j (\partial_\alpha Q)^j_k x^k \Big]_l x^l$$

$$\frac{\partial^2 \mathcal{L}}{\partial \beta^2} = \Big(q_i \varsigma(\boldsymbol{\pi})^i \Big)^2 - \mathbf{q} \odot \mathbf{q} \odot \varsigma(\boldsymbol{\pi})$$

$$\frac{\partial^2 \mathcal{L}}{\partial \gamma^2} = \beta \Big[(\mathbf{u} - \varsigma(\boldsymbol{\pi}))_i (\partial_\gamma^2 Q)^i - (\partial_\gamma \varsigma)_j (\partial_\gamma Q)^j_k x^k \Big]_l x^l$$

$$\frac{\partial^2 \mathcal{L}}{\partial \lambda^2} = \beta \Big[(\mathbf{u} - \varsigma(\boldsymbol{\pi}))_i (\partial_\lambda^2 Q)^i - (\partial_\lambda \varsigma)_j (\partial_\lambda Q)^j_k x^k \Big]_l x^l$$

The off diagonal elements of the Hessian are as follows:

$$\frac{\partial^2 \mathcal{L}}{\partial \alpha \partial \beta} = \Big(\mathbf{u} - \varsigma(\boldsymbol{\pi}) - \beta(\partial_\beta \varsigma) \Big)_i (\partial_\alpha Q)^i_j x^j$$

$$\frac{\partial^2 \mathcal{L}}{\partial \beta \partial \gamma} = \Big(\mathbf{u} - \varsigma(\boldsymbol{\pi}) - \beta(\partial_\beta \varsigma) \Big)_i (\partial_\gamma Q)^i_j x^j$$

$$\frac{\partial^2 \mathcal{L}}{\partial \beta \partial \lambda} = \Big(\mathbf{u} - \varsigma(\boldsymbol{\pi}) - \beta(\partial_\beta \varsigma) \Big)_i (\partial_\lambda Q)^i_j x^j$$

$$\frac{\partial^2 \mathcal{L}}{\partial \alpha \partial \gamma} = \beta \Big((\mathbf{u} - \varsigma(\boldsymbol{\pi}))_i (\partial_\alpha \partial_\gamma Q)^i - (\partial_\gamma \varsigma)_j (\partial_\alpha Q)^j \Big)_k x^k$$

$$\frac{\partial^2 \mathcal{L}}{\partial \alpha \partial \lambda} = \beta \Big((\mathbf{u} - \varsigma(\boldsymbol{\pi}))_i (\partial_\alpha \partial_\lambda Q)^i - (\partial_\lambda \varsigma)_j (\partial_\alpha Q)^j \Big)_k x^k$$

$$\frac{\partial^2 \mathcal{L}}{\partial \gamma \partial \lambda} = \beta \Big((\mathbf{u} - \varsigma(\boldsymbol{\pi}))_i (\partial_\lambda \partial_\gamma Q)^i - (\partial_\lambda \varsigma)_j (\partial_\gamma Q)^j \Big)_k x^k$$

Reward Prediction Error

$$\partial_\alpha^2 \delta = (\partial_{\mathbf{Q}} \delta)_{ij} (\partial_\alpha^2 Q)^{ij}$$

$$\partial_\gamma^2 \delta = (\partial_{\mathbf{Q}} \delta)_{ij} (\partial_\gamma^2 Q)^{ij} + 2\tilde{u}_i (\partial_\gamma Q)^i_j \tilde{x}^j$$

$$\partial_\lambda^2 \delta = (\partial_{\mathbf{Q}} \delta)_{ij} (\partial_\lambda^2 Q)^{ij}$$

$$\partial_\alpha \partial_\gamma \delta = (\partial_{\mathbf{Q}} \delta)_{ij} (\partial_\gamma \partial_\alpha Q)^{ij} + \tilde{u}_i (\partial_\alpha Q)^i_j \tilde{x}^j$$

$$\partial_\alpha \partial_\lambda \delta = (\partial_{\mathbf{Q}} \delta)_{ij} (\partial_\alpha \partial_\lambda Q)^{ij}$$

$$\partial_\gamma \partial_\lambda \delta = (\partial_{\mathbf{Q}} \delta)_{ij} (\partial_\gamma \partial_\lambda Q)^{ij} + \tilde{u}_i (\partial_\lambda Q)^i_j \tilde{x}^j$$

Value Function

$$\partial_\alpha^2 Q_{ij} = \partial_\alpha^2 Q_{ij} + 2(\partial_\alpha \delta) z_{ij} + \alpha(\partial_\alpha^2 \delta) z_{ij}$$

$$\partial_\gamma^2 Q_{ij} = \partial_\gamma^2 Q_{ij} + \alpha\Big((\partial_\gamma^2 \delta) z_{ij} + (\partial_\gamma \delta)(\partial_\gamma z_{ij}) + (\partial_\gamma \delta)(\partial_\gamma^2 z_{ij})\Big)$$

$$\partial_\lambda^2 Q_{ij} = \partial_\lambda^2 Q_{ij} + \alpha\Big((\partial_\lambda^2 \delta) z_{ij} + (\partial_\lambda \delta)(\partial_\lambda z_{ij}) + (\partial_\lambda \delta)(\partial_\lambda^2 z_{ij})\Big)$$

$$\partial_\alpha \partial_\gamma Q_{ij} = \partial_\alpha \partial_\gamma Q_{ij} + (\partial_\gamma \delta) z_{ij} + \delta(\partial_\gamma z_{ij}) + \alpha(\partial_\alpha \delta)(\partial_\gamma z_{ij}) + \alpha(\partial_\alpha \partial_\gamma \delta) z_{ij}$$

$$\partial_\alpha \partial_\lambda Q_{ij} = \partial_\alpha \partial_\lambda Q_{ij} + (\partial_\lambda \delta) z_{ij} + \delta(\partial_\lambda z_{ij}) + \alpha(\partial_\alpha \delta)(\partial_\lambda z_{ij}) + \alpha(\partial_\alpha \partial_\lambda \delta) z_{ij}$$

$$\partial_\gamma \partial_\lambda Q_{ij} = \partial_\gamma \partial_\lambda Q_{ij} + \alpha\Big[(\partial_\lambda \partial_\gamma \delta) z_{ij} + (\partial_\gamma \delta)(\partial_\lambda z_{ij}) + (\partial_\lambda \delta)(\partial_\gamma z_{ij}) + \delta(\partial_\lambda \partial_\gamma z_{ij})\Big]$$

Trace Decay

$$\partial_\gamma^2 z = \lambda\Big(2(\partial_\gamma z) + \gamma(\partial_\gamma^2 z)\Big)$$

$$\partial_\lambda^2 z = \gamma\Big(2(\partial_\lambda z) + \lambda(\partial_\lambda^2 z)\Big)$$

$$\partial_\gamma \partial_\lambda z = z + \gamma(\partial_\gamma z) + \lambda(\partial_\lambda z) + \lambda\gamma(\partial_\gamma \partial_\lambda z)$$

Arguments:

- **action**: `ndarray(nactions)`. One-hot action vector
- **state**: `ndarray(nstates)`. One-hot state vector

SARSASoftmaxAgent.reset_trace

```
fitr.agents.agents.reset_trace(self, state_only=False)
```

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

Arguments:

- **state_only**: `bool`. If the eligibility trace is only an `nstate` dimensional vector (i.e. for a Pavlovian conditioning model) then set to `True`. For instumental models, the eligibility trace should be an `nactions` by `nstates` matrix, so keep this to `False` in that case.

SARSAStickySoftmaxAgent

```
fitr.agents.agents.SARSAStickySoftmaxAgent()
```

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

The sticky softmax policy selects actions from a multinomial

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

whose parameters are

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

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
- **perseveration**: Perseveration parameter β^ρ
- **rng**: `np.random.RandomState`

SARSAStickySoftmaxAgent.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

SARSAStickySoftmaxAgent.generate_data

```
fitr.agents.agents.generate_data(self, ntrials, state_only=False)
```

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

Arguments:

- **ntrials**: `int` number of trials
- **state_only**: `bool`. If the eligibility trace is only an `nstate` dimensional vector (i.e. for a Pavlovian conditioning model) then set to `True`. For instumental models, the eligibility trace should be an `nactions` by `nstates` matrix, so keep this to `False` in that case.

Returns:

`fitr.data.BehaviouralData`

SARSAStickySoftmaxAgent.learning

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

Updates the model's parameters and computes gradients

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

SARSAStickySoftmaxAgent.reset_trace

```
fitr.agents.agents.reset_trace(self, state_only=False)
```

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

Arguments:

- **state_only**: `bool`. If the eligibility trace is only an `nstate` dimensional vector (i.e. for a Pavlovian conditioning model) then set to `True`. For instumental models, the eligibility trace should be an `nactions` by `nstates` matrix, so keep this to `False` in that case.

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, state_only=False)
```

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

Arguments:

- **ntrials**: `int` number of trials
- **state_only**: `bool`. If the eligibility trace is only an `nstate` dimensional vector (i.e. for a Pavlovian conditioning model) then set to `True`. For instumental models, the eligibility trace should be an `nactions` by `nstates` matrix, so keep this to `False` in that case.

Returns:

`fitr.data.BehaviouralData`

QLearningSoftmaxAgent.learning

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

Updates the model's parameters and computes gradients

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, state_only=False)
```

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

Arguments:

- **state_only**: `bool`. If the eligibility trace is only an `nstate` dimensional vector (i.e. for a Pavlovian conditioning model) then set to `True`. For instumental models, the eligibility trace should be an `nactions` by `nstates` matrix, so keep this to `False` in that case.

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 and computes gradients

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, action)
```

Computes the log-probability of an action taken by the agent in a given state, as well as updates all partial derivatives with respect to the parameters.

This function overrides the `log_prob` method of the parent class.

Let

- $n_u \in \mathbb{N}_+$ be the dimensionality of the action space
- $n_x \in \mathbb{N}_+$ be the dimensionality of the state space
- $\mathbf{u} = (u_0, u_1, u_{n_u})^\top$ be a one-hot action vector
- $\mathbf{x} = (x_0, x_1, x_{n_x})^\top$ be a one-hot action vector
- $\mathbf{Q} \in \mathbb{R}^{n_u \times n_x}$ be the state-action value function parameters
- $\beta \in \mathbb{R}$ be the inverse softmax temperature
- $\alpha \in [0, 1]$ be the learning rate

- $\varsigma(\boldsymbol{\pi}) = p(\mathbf{u}|\mathbf{Q}, \beta)$ be a softmax function with logits $\pi_i = \beta Q_{ij} x^j$ (shown in Einstein summation convention).
- $\mathcal{L} = \log p(\mathbf{u}|\mathbf{Q}, \beta)$ be the log-likelihood function for trial t
- $q_i = Q_{ij} x^j$ be the value of the state x^j
- $v^i = e^{\beta q_i}$ be the softmax potential
- $\eta(\boldsymbol{\pi})$ be the softmax partition function.

Then we have the partial derivative of $\mathcal{L}$ at trial t with respect to α

$$\partial_\alpha \mathcal{L} = \beta \Big[\big(\mathbf{u} - \varsigma(\pi)\big)_i (\partial_\alpha Q)^i_j x^j \Big],$$

and with respect to β

$$\partial_\beta \mathcal{L} = u_i \Big(\mathbf{I}_{n_u \times n_u} - \varsigma(\boldsymbol{\pi}) \Big)^i_j Q_{jk} x^k.$$

We also compute the Hessian $\mathbf{H}$, defined as

$$\mathbf{H} = \left[\begin{array}{cc} \partial^2_\alpha \mathcal{L} & \partial_\alpha \partial_\beta \mathcal{L} \\ \partial_\beta \partial_\alpha \mathcal{L} & \partial^2_\beta \mathcal{L} \end{array} \right].$$

The components of $\mathbf{H}$ are

$$\partial^2_\alpha \mathcal{L} = \beta \Big((\mathbf{u} - \varsigma(\boldsymbol{\pi}))_i (\partial^2_\alpha \mathbf{Q})^i - \partial_\alpha \varsigma(\boldsymbol{\pi})_i (\partial_\alpha \mathbf{Q})^i \Big)_j x^j,$$

$$\partial^2_\beta \mathcal{L} = u_i \Big(\Big),$$

$$\partial_\alpha \partial_\beta \mathcal{L} = \Bigg[(u - \varsigma(\boldsymbol{\pi})) - \beta \partial_\beta \varsigma(\boldsymbol{\pi}) \Bigg]_i (\partial_\alpha Q)^i_k x^k.$$

and where $\partial_\beta \partial_\alpha \mathcal{L} = \partial_\alpha \partial_\beta \mathcal{L}$ since the second derivatives of $\mathcal{L}$ are continuous in the neighbourhood of the parameters.

Arguments:

- **action**: `ndarray(nactions)`. One-hot action vector
- **state**: `ndarray(nstates)`. One-hot state vector

RWSoftmaxAgent.reset_trace

```
fitr.agents.agents.reset_trace(self, state_only=False)
```

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

Arguments:

- **state_only**: `bool`. If the eligibility trace is only an `nstate` dimensional vector (i.e. for a Pavlovian conditioning model) then set to `True`. For instumental models, the eligibility trace should be an `nactions` by `nstates` matrix, so keep this to `False` in that case.

RWStickySoftmaxAgent

`fitr.agents.agents.RWStickySoftmaxAgent()`

An instrumental Rescorla-Wagner agent with a 'sticky' softmax policy

The softmax policy selects actions from a multinomial

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

whose parameters are

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

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
- **perseveration**: Perseveration parameter $\beta^h o$
- **rng**: `np.random.RandomState`

RWStickySoftmaxAgent.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

RWStickySoftmaxAgent.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`

RWStickySoftmaxAgent.learning

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

Updates the model's parameters and computes gradients

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

RWStickySoftmaxAgent.log_prob

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

Computes the log-probability of an action taken by the agent in a given state, as well as updates all partial derivatives with respect to the parameters.

This function overrides the `log_prob` method of the parent class.

Let

- $n_u \in \mathbb{N}_+$ be the dimensionality of the action space
- $n_x \in \mathbb{N}_+$ be the dimensionality of the state space
- $\mathbf{u} = (u_0, u_1, u_{n_u})^\top$ be a one-hot action vector
- $\tilde{\mathbf{u}}$ be a one-hot vector representing the last trial's action, where at trial 0, $\tilde{\mathbf{u}} = \mathbf{0}$.
- $\mathbf{x} = (x_0, x_1, x_{n_x})^\top$ be a one-hot action vector
- $\mathbf{Q} \in \mathbb{R}^{n_u \times n_x}$ be the state-action value function parameters
- $\beta \in \mathbb{R}$ be the inverse softmax temperature scaling the action values
- $\rho \in \mathbb{R}$ be the inverse softmax temperature scaling the influence of the past trial's action
- $\alpha \in [0, 1]$ be the learning rate

- $\varsigma(\boldsymbol{\pi}) = p(\mathbf{u}|\mathbf{Q}, \beta, \rho)$ be a softmax function with logits $\pi_i = \beta Q_{ij} x^j + \rho \tilde{u}_i$ (shown in Einstein summation convention).
- $\mathcal{L} = \log p(\mathbf{u}|\mathbf{Q}, \beta, \rho)$ be the log-likelihood function for trial t
- $q_i = Q_{ij} x^j$ be the value of the state x^j
- $v^i = e^{\beta q_i + \rho \tilde{u}_i}$ be the softmax potential
- $\eta(\boldsymbol{\pi})$ be the softmax partition function.

Then we have the partial derivative of $\mathcal{L}$ at trial t with respect to α

$$\partial_\alpha \mathcal{L} = \beta \Big[\big(\mathbf{u} - \varsigma(\pi)\big)_i (\partial_\alpha Q)^i_j x^j \Big],$$

and with respect to β

$$\partial_\beta \mathcal{L} = u_i \Big(\mathbf{I}_{n_u \times n_u} - \varsigma(\boldsymbol{\pi}) \Big)^i_j Q_{jk} x^k$$

and with respect to ρ

$$\partial_\rho \mathcal{L} = u_i \Big(\mathbf{I}_{n_u \times n_u} - \varsigma(\boldsymbol{\pi}) \Big)^i_j \tilde{u}^j.$$

We also compute the Hessian $\mathbf{H}$, defined as

$$\mathbf{H} = \begin{bmatrix} \partial^2_\alpha \mathcal{L} & \partial_\alpha \partial_\beta \mathcal{L} & \partial_\alpha \partial_\rho \mathcal{L} \\ \partial_\beta \partial_\alpha \mathcal{L} & \partial^2_\beta \mathcal{L} & \partial_\beta \partial_\rho \mathcal{L} \\ \partial_\rho \partial_\alpha \mathcal{L} & \partial_\rho \partial_\beta \mathcal{L} & \partial^2_\rho \mathcal{L} \end{bmatrix}.$$

The components of $\mathbf{H}$ are virtually identical to that of `RWSoftmaxAgent`, with the exception of the $\partial_\rho \partial_\alpha \mathcal{L}$ and $\partial_\beta \partial_\rho \mathcal{L}$

$$\partial^2_\alpha \mathcal{L} = \beta \Big((\mathbf{u} - \varsigma(\boldsymbol{\pi}))_i (\partial^2_\alpha \mathbf{Q})^i - \partial_\alpha \varsigma(\boldsymbol{\pi})_i (\partial_\alpha \mathbf{Q})^i \Big)_j x^j,$$

$$\partial^2_\beta \mathcal{L} = u_k \left(\frac{(q_i q_i v^i v^i}{z^2} - \frac{q_i q_i v^i}{z} \right)^k$$

$$\partial_\alpha \partial_\beta \mathcal{L} = \Bigg[(u - \varsigma(\boldsymbol{\pi})) - \beta \partial_\beta \varsigma(\boldsymbol{\pi}) \Bigg]_i (\partial_\alpha Q)^i_k x^k$$

$$\partial_\alpha \partial_\rho \mathcal{L} = -\beta \Big(\partial_{\boldsymbol{\pi}} \varsigma(\boldsymbol{\pi})_i \tilde{u}^i \Big)_j (\partial_\alpha Q)^j_k x^k$$

and where $\mathbf{H}$ is symmetric since the second derivatives of $\mathcal{L}$ are continuous in the neighbourhood of the parameters.

Arguments:

- **action**: `ndarray(nactions)`. One-hot action vector
- **state**: `ndarray(nstates)`. One-hot state vector

Returns:

`float`

RWStickySoftmaxAgent.reset_trace

```
fitr.agents.agents.reset_trace(self, state_only=False)
```

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

Arguments:

- **state_only**: `bool`. If the eligibility trace is only an `nstate` dimensional vector (i.e. for a Pavlovian conditioning model) then set to `True`. For instumental models, the eligibility trace should be an `nactions` by `nstates` matrix, so keep this to `False` in that case.

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 and computes gradients

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, state_only=False)
```

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

Arguments:

- **state_only**: `bool`. If the eligibility trace is only an `nstate` dimensional vector (i.e. for a Pavlovian conditioning model) then set to `True`. For instumental models, the eligibility trace should be an `nactions` by `nstates` matrix, so keep this to `False` in that case.

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, maxs`

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
- **maxstarts_without_improvement**: `int`. Maximum number of restarts without improvement in objective function value
- **init_sd**: Standard deviation for Gaussian initial values
- **jac**: `bool`. Set to `True` if `f` returns a Jacobian as the second element of the returned values
- **hess**: `bool`. Set to `True` if third output value of `f` is the Hessian matrix
- **method**: `str`. One of the `scipy.optimize` methods.

Returns:

`fitr.inference.OptimizationResult`

Todo:

- [ ] Raise errors when user selects inappropriate optimization function given values for `jac` and `hess`

---

l_bfgs_b

`fitr.inference.mle_parallel.l_bfgs_b(f, i, data, nparams, jac, minstarts=2, maxstart`

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**: (Negative!) 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
- **jac**: `bool`. Set to `True` if `f` returns a Jacobian as the second element of the returned values
- **minstarts**: `int`. Minimum number of restarts with new initial values
- **maxstarts**: `int`. Maximum number of restarts with new initial values
- **maxstarts_without_improvement**: `int`. Maximum number of restarts without improvement in objective function value
- **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

---

bms

```
fitr.inference.bms.bms(L, ftol=1e-12, nsamples=1000000, rng=<mtrand.RandomState obje
```

Implements variational Bayesian Model Selection as per Rigoux et al. (2014).

Arguments:

- **L**: `ndarray((nsubjects, nmodels))`. Log model evidence
- **ftol**: `float`. Threshold for convergence of prediction error
- **nsamples**: `int>0`. Number of samples to draw from Dirichlet distribution for computation of exceedence probabilities
- **rng**: `np.random.RandomState`
- **verbose**: `bool (default=True)`. If `False`, no output provided.

Returns:

- **pxp**: `ndarray(nmodels)`. Protected exceedance probabilities
- **xp**: `ndarray(nmodels)`. Exceedance probabilities
- **bor**: `ndarray(nmodels)`. Bayesian Omnibus Risk
- **q_m**: `ndarray((nsubjects, nmodels))`. Posterior distribution over models for each subject
- **alpha**: `ndarray(nmodels)`. Posterior estimates of Dirichlet parameters
- **f0**: `float`. Free energy of null model
- **f1**: `float`. Free energy of alternative model
- **niter**: `int`. Number of iterations of posterior optimization

Examples:

Assuming one is given a matrix of (log-) model evidence values `L` of type `ndarray((nsubjects, nmodels))`,

```
from fitr.inference import spm_bms

pxp, xp, bor, q_m, alpha, f0, f1, niter = bms(L)
```

Todos:

- [ ] Add notes on derivation

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

Statistics

`fitr.stats`

Functions for statistical analyses.

bic

`fitr.stats.model_evaluation.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.

---

lme

`fitr.stats.model_evaluation.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

---

pearson_rho

```
fitr.stats.correlations.pearson_rho(X, Y, comparison='diagonal')
```

Linear (Pearson) 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.

Also returns a two-tailed p-value where the hypotheses being tested are

$$H_o : \rho = 0$$

$$H_a : \rho \neq 0$$

and where the test statistic is

$$T = \frac{\rho \sqrt{n_s - 2}}{\sqrt{1 - \rho^2}}$$

and the p-value is thus

$$p = 2 * (1 - \mathcal{T}(T, n_s - 2))$$

given the CDF of the Student T-distribution with degrees of freedom $n_s - 2$.

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
- **comparison**: `str`. Here `'diagonal'` computes correlations individually, column-for-column between matrices. Otherwise `'pairwise'` computes pairwise correlations between columns in `X` and `Y`.

Returns:

- **rho**: `ndarray((nfeatures,))`. Correlation coefficient(s). Will be an `X.shape[1]` by `Y.shape[1]` matrix if `comparison='pairwise'`
- **p**: `ndarray((nfeatures,))`. P-values for correlation coefficient(s). Will be an `X.shape[1]` by `Y.shape[1]` matrix if `comparison='pairwise'`

TODO:

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

---

spearman_rho

`fitr.stats.correlations.spearman_rho(X, Y, comparison='diagonal')`

Spearman's rank correlation

Note this function takes correlations between the columns of `X` and `Y`.

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
- **comparison**: `str`. Here `'diagonal'` computes correlations individually, column-for-column between matrices. Otherwise `'pairwise'` computes pairwise correlations between columns in `X` and `Y`.

Returns:

- **rho**: `ndarray((nfeatures,))`. Correlation coefficient(s). Will be an `X.shape[1]` by `Y.shape[1]` matrix if `comparison='pairwise'`
- **p**: `ndarray((nfeatures,))`. P-values for correlation coefficient(s). Will be an `X.shape[1]` by `Y.shape[1]` matrix if `comparison='pairwise'`

---

linear_regression

`fitr.stats.linear_regression.linear_regression(X, y, add_intercept=True, scale_x=Fal`

Performs ordinary least squares linear regression, returning MLEs of the coefficients

Hypothesis testing on the model

Compute sum of squares:

$$SS_R = (\mathbf{y} - \bar{y})^o p(\mathbf{y} - \bar{y})$$

$$SS_{Res} = \mathbf{y}^\top \mathbf{y} - \mathbf{w}^\top \mathbf{X}^\top \mathbf{y}$$

$$SS_T = \mathbf{y}^\top \mathbf{y} - \frac{(\mathbf{1}^\top \mathbf{y})^\top}{n_s}$$

The test statistic is defined as follows:

$$F = \frac{SS_R(n-k-1)}{SS_{Res}k} \sim F(k, n-k-1)$$

The adjusted R^2 is

$$R^2_{Adj} = 1 - \frac{SS_R(n-1)}{SS_T(n-k-1)}$$

Hypothesis testing on the coefficients

The test statistic is

$$\frac{w_i}{SE(w_i)} \sim StudentT(n-k-1)$$

Arguments:

- **X**: `ndarray((nsamples, nfeatures))`. Predictors
- **y**: `ndarray(nsamples)`. Target
- **add_intercept**: `bool`. Whether to add an intercept term (pads on LHS of `X` with column of ones)
- **scale_x**: `bool`. Whether to scale the columns of `X`
- **scale_y**: `bool`. Whether to scale the columns of `y`

Returns:

`LinearRegressionResult`

---

kruskal_wallis

```
fitr.stats.nonparametric.kruskal_wallis(x, g, dist='beta')
```

Kruskal-Wallis one-way analysis of variance (one-way ANOVA on ranks)

Arguments:

- **x**: `ndarray(nsamples)`. Vector of data to be compared
- **g**: `ndarray(nsamples)`. Group ID's
- **dist**: `str {'chi2', 'beta'}`. Which distributional approximation to make

Returns:

- **T**: `float`. Test statistic
- **p**: `float`. P-value for the comparison

---

conover

```
fitr.stats.nonparametric.conover(x, g, alpha=0.05, adjust='bonferroni')
```

Conover's nonparametric test of homogeneity.

Arguments:

- **x**: `ndarray(nsamples)`. Vector of data to be compared
- **g**: `ndarray(nsamples)`. Group ID's
- **alpha**: `0 < float < 1`. Significance threshold
- **adjust**: `str`. Method to adjust p-values (see below)

Returns:

- **T**: `float`. Test statistic
- **p**: `float`. P-value for the comparison

Notes:

Adjustment methods include the following:

- `bonferroni` : one-step correction
- `sidak` : one-step correction
- `holm-sidak` : step down method using Sidak adjustments
- `holm` : step-down method using Bonferroni adjustments
- `simes-hochberg` : step-up method (independent)
- `hommel` : closed method based on Simes tests (non-negative)
- `fdr_bh` : Benjamini/Hochberg (non-negative)
- `fdr_by` : Benjamini/Yekutieli (negative)
- `fdr_tsbh` : two stage fdr correction (non-negative)
- `fdr_tsbky` : two stage fdr correction (non-negative)

References:

W. J. Conover and R. L. Iman (1979), On multiple-comparisons procedures, Tech. Rep. LA-7677-MS, Los Alamos Scientific Laboratory.

Chapter 9

Hierarchical Convolutional Logistic Regression

fitr.hclr

Hierarchical convolutional logistic regression (HCLR): A general analysis method for trial-by-trial behavioural data with covariates.

HCLR

`fitr.hclr.HCLR()`

Hierarchical Convolutional Logistic Regression (HCLR) for general behavioural data.

Attributes:

- **X**: `ndarray((nsubjects,ntrials,nfeatures))`. The "experience" tensor.
- **y**: `ndarray((nsubjects,ntrials,ntargets))`. Tensor of "choices" we are trying to predict.
- **Z**: `ndarray((nsubjects,ncovariates))`. Covariates of interest
- **V**: `ndarray((naxes,nfeatures))`. Vectors identifying features of interest (i.e. to compute indices). If `add_intercept=True`, then the dimensionality of `V` should be `ndarray((naxes, nfeatures+1))`, where the first column represents the basis coordinate for the bias.
- **filter_size**: `int`. Number of steps prior to target included as features.
- **loading_matrix_scale**: `float > 0`. Scale of the loading matrix $\mathbf{\Phi}$, which is assumed that $\phi_{ij} \sim \mathcal{N}(0, 1)$, with the default scale being 1.
- **add_intercept**: 'bool'. Whether to add intercept
- **group_mean**: `ndarray`. Samples of the posterior group-level mean. `None` until model is fit
- **group_scale**: `ndarray`. Samples of the posterior group-level scale. `None` until model is fit
- **loading_matrix**: `ndarray`. Samples of the posterior loading matrix. `None` until model is fit
- **subject_parameters**: `ndarray`. Samples of the posterior subject-level parameters. `None` until model is fit
- **group_indices**: `ndarray`. Samples of the posterior group-level projections on to the basis. `None` until model is fit
- **covariate_effects**: `ndarray`. Samples of the posterior projection of the loading matrix onto the basis. `None` until model is fit

Notes

- When presenting `X` and `y`, note that the indices of `y` should correspond exactly to the trial indices in `X`, even though the HCLR analysis is predicting a trial ahead. In other words, there should be no lag in the `X`, `y` inputs. The HCLR setup will automatically set up the lag depending on how you set the `filter_size`.

HCLR.fit

```
fitr.hclr.fit(self, nchains=4, niter=1000, warmup=None, thin=1, seed=None, verbose=F
```

Fits the HCLR model

Arguments:

- **nchains**: `int`. Number of chains for the MCMC run.
- **niter**: `int`. Number of iterations over which to run MCMC.
- **warmup**: `int`. Number of warmup iterations
- **thin**: `int`. Periodicity of sample recording
- **seed**: `int`. Seed for pseudorandom number generator
- **algorithm**: `{'NUTS','HMC'}`
- **n_jobs**: `int`. Number of cores to use (default=-1, as many as possible and required)

Chapter 10

Utilities

`fitr.utils`

Functions used across `fitr`.

batch_softmax

`fitr.utils.batch_softmax(X, axis=1)`

Computes the softmax function for a batch of samples

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

Arguments:

- **x**: Softmax logits (`ndarray((nsamples,nfeatures))`)

Returns:

Matrix of probabilities of size `ndarray((nsamples,nfeatures))` such that sum over `nfeatures` is 1.

---

batch_transform

`fitr.utils.batch_transform(X, f_list)`

Applies the `fitr.utils.transform` function over a batch of parameters

Arguments:

- **X**: `ndarray((nsamples, nparams))`. Raw parameters
- **f_list**: `list` where `len(list) == nparams`. Functions defining coordinate transformations on each element of `x`.

Returns:

`ndarray((nsamples, nparams))`. Transformed parameters

I

`fitr.utils.I(x)`

Identity transformation.

Mainly for convenience when using `fitr.utils.transform` with some vector element that should not be transformed, despite changing the coordinates of other variables.

Arguments:

- **x**: `ndarray`

Returns:

`ndarray(shape=x.shape)`

---

log_loss

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

Computes log loss.

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

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

---

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`

---

rank_data

`fitr.utils.rank_data(x)`

Ranks a set of observations, assigning the average of ranks to ties.

Arguments:

- **x**: `ndarray(nsamples)`. Vector of data to be compared

Returns:

- **ranks**: `ndarray(nsamples)`. Ranks for each observation

---

rank_grouped_data

`fitr.utils.rank_grouped_data(x, g)`

Ranks observations taken across several groups

Arguments:

- **x**: `ndarray(nsamples)`. Vector of data to be compared
- **g**: `ndarray(nsamples)`. Group ID’s

Returns:

- **ranks**: `ndarray(nsamples)`. Ranks for each observation
- **G**: `ndarray(nsamples, ngroups)`. Matrix indicating whether sample i is in group j
- **R**: `ndarray((nsamples, ngroups))`. Matrix indicating the rank for sample i in group j
- **lab**: `ndarray(ngroups)`. Group labels

---

reduce_then_tile

`fitr.utils.reduce_then_tile(X, f, axis=1)`

Computes some reduction function over an axis, then tiles that vector to create matrix of original size

Arguments:

- **X**: `ndarray((n, m))`. Matrix.
- **f**: `function` that reduces data across some axis (e.g. `np.sum()`, `np.max()`)

- **axis**: `int` which axis the data should be reduced over (only goes over 2 axes for now)

Returns:res

`ndarray((n, m))`

Examples:

Here is one way to compute a softmax function over the columns of `X`, for each row.

```
import numpy as np
X = np.random.normal(0, 1, size=(10, 3))**2
max_x = reduce_then_tile(X, np.max, axis=1)
exp_x = np.exp(X - max_x)
sum_exp_x = reduce_then_tile(exp_x, np.sum, axis=1)
y = exp_x/sum_exp_x
```

---

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 standard deviation.

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

Arguments:

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

---

transform

`fitr.utils.transform(x, f_list)`

Transforms parameters from domain in `x` into some new domain defined by `f_list`

Arguments:

- **x**: `ndarray((nparams,))`. Parameter vector in some domain.
- **f_list**: `list` where `len(list) == nparams`. Functions defining coordinate transformations on each element of `x`.

Returns:

- **x_**: `ndarray((nparams,))`. Parameter vector in new coordinates.

Examples:

Applying `fitr` transforms can be done as follows.

```
import numpy as np
from fitr.utils import transform, sigmoid, relu

x = np.random.normal(0, 5, size=3)
x_= transform(x, [sigmoid, relu, relu])
```

You can also apply other functions, so long as dimensions are equal for input and output.

```
import numpy as np
from fitr.utils import transform

x  = np.random.normal(0, 10, size=3)
x_ = transform(x, [np.square, np.sqrt, np.exp])
```
