mlpy Documentation

Release 0.1.0

Astrid Jackson

Apr 24, 2017

Contents

CHAPTER 1

MLPy

A Machine Learning library for Python

- Free software: MIT license
- Documentation: https://mlpy.readthedocs.org.

Features

- TODO

CHAPTER 2

Installation

At the command line:

```
$ easy_install mlpy
```

Or, if you have virtualenvwrapper installed:

```
$ mkvirtualenv mlpy
$ pip install mlpy
```

CHAPTER 3

Usage

To use MLPy in a project:

```
import mlpy
```

CHAPTER 4

Contributing

Contributions are welcome, and they are greatly appreciated! Every little bit helps, and credit will always be given.

You can contribute in many ways:

Types of Contributions

Report Bugs

Report bugs at https://github.com/evenmarbles/mlpy/issues.

If you are reporting a bug, please include:

- Your operating system name and version.
- Any details about your local setup that might be helpful in troubleshooting.
- Detailed steps to reproduce the bug.

Fix Bugs

Look through the GitHub issues for bugs. Anything tagged with "bug" is open to whoever wants to implement it.

Implement Features

Look through the GitHub issues for features. Anything tagged with "feature" is open to whoever wants to implement it.

Write Documentation

MLPy could always use more documentation, whether as part of the official MLPy docs, in docstrings, or even on the web in blog posts, articles, and such.

Submit Feedback

The best way to send feedback is to file an issue at https://github.com/evenmarbles/mlpy/issues.

If you are proposing a feature:

- Explain in detail how it would work.
- Keep the scope as narrow as possible, to make it easier to implement.
- Remember that this is a volunteer-driven project, and that contributions are welcome :)

Get Started!

Ready to contribute? Here's how to set up *mlpy* for local development.

1. Fork the *mlpy* repo on GitHub.
2. Clone your fork locally:

```
$ git clone git@github.com:your_name_here/mlpy.git
```

3. Install your local copy into a virtualenv. Assuming you have virtualenvwrapper installed, this is how you set up your fork for local development:

```
$ mkvirtualenv mlpy
$ cd mlpy/
$ python setup.py develop
```

4. Create a branch for local development:

```
$ git checkout -b name-of-your-bugfix-or-feature
```

   Now you can make your changes locally.

5. When you're done making changes, check that your changes pass flake8 and the tests, including testing other Python versions with tox:

```
$ flake8 mlpy tests
$ python setup.py test
$ tox
```

   To get flake8 and tox, just pip install them into your virtualenv.

6. Commit your changes and push your branch to GitHub:

```
$ git add .
$ git commit -m "Your detailed description of your changes."
$ git push origin name-of-your-bugfix-or-feature
```

7. Submit a pull request through the GitHub website.

Pull Request Guidelines

Before you submit a pull request, check that it meets these guidelines:

1. The pull request should include tests.
2. If the pull request adds functionality, the docs should be updated. Put your new functionality into a function with a docstring, and add the feature to the list in README.rst.
3. The pull request should work for Python 2.6, 2.7, 3.3, and 3.4, and for PyPy. Check https://travis-ci.org/evenmarbles/mlpy/pull_requests and make sure that the tests pass for all supported Python versions.

Tips

To run a subset of tests:

```
$ python -m unittest tests.test_mlpy
```

CHAPTER 5

Credits

Development Lead

- Astrid Jackson <ajackson@eecs.ucf.edu>

Contributors

None yet. Why not be the first?

CHAPTER 6

History

CHAPTER 7

0.1.0 (2015-08-11)

- First release on PyPI.

CHAPTER 8

Agent design (`mlpy.agents`)

This module contains functionality for designing agents navigating inside an *`Environment`*.

Control of the agents is specified by an agent module which is handled by the *`Agent`* base class.

An agent class deriving from *`Agent`* can also make use of a finite state machine (FSM) to control the agent's behavior and a world model to maintain a notion of the current state of the world.

Agents

`Agent`	The agent base class.
`AgentModuleFactory`	The agent module factory.
`IAgentModule`	Agent module base interface class.
`LearningModule`	Learning agent module.
`FollowPolicyModule`	The follow policy agent module.
`UserModule`	The user agent module.

mlpy.agents.Agent

class `mlpy.agents.`**`Agent`** (*mid=None*, *module_type=None*, *task=None*, **args*, ***kwargs*)

Bases: *`mlpy.modules.Module`*

The agent base class.

Agents act in an environment (*`Environment`*) usually performing a task (*`Task`*). Depending on the agent module they either follow a policy (*`FollowPolicyModule`*), are user controlled via the keyboard or a PS2 controller (*`UserModule`*), or learn according to a learner (*`LearningModule`*). New agent modules can be created by inheriting from the *`IAgentModule`* class.

Parameters **mid** : str, optional

The agent's unique identifier.

module_type : str

The agent module type, which is the name of the class.

task: Task

The task the agent must complete.

args: tuple

Positional parameters passed to the agent module.

kwargs: dict

Non-positional parameters passed to the agent module.

Examples

```
>>> from mlpy.agents import Agent
>>> Agent(module_type='learningmodule', learner_type='qlearner', max_steps=10)
>>> Agent(None, 'learningmodule', None, 'qlearner', max_steps=10)
```

This creates an agent with a *LearningModule* agent module that performs qlearning. The parameters are given in the order in which the objects are created. Internally the agent creates the learning agent module and the learning agent module creates the qlearner.

Alternatively, non-positional arguments can be used:

```
>>> Agent(module_type='learningmodule', learner_type='qlearner', max_steps=10)
```

```
>>> from mlpy.experiments.task import Task
>>> task = Task()
>>> Agent(None, 'learningmodule', task, 'qlearner', max_steps=10)
```

This creates an agent that performs the given task. If a task is given, the method *Task.get_reward* is passed as the reward callback to the learning module to retrieve the reward. By default *Task.get_reward* returns *None*.

Attributes

mid	The module's unique identifier.
module	The agent module controlling the actions of the agent.
task	The task the agent is to perform.

mlpy.agents.Agent.mid

Agent.**mid**

The module's unique identifier.

Returns str :

The module's unique identifier

mlpy.agents.Agent.module

`Agent.module`

The agent module controlling the actions of the agent.

Returns IAgentModule :

The agent module instance.

mlpy.agents.Agent.task

`Agent.task`

The task the agent is to perform.

Returns Task :

The task to perform.

Methods

`enter`(t)	Enter the agent and the agent module.
`exit`()	Exit the agent and the agent module.
`is_task_complete`()	Check if the agent's task is completed.
`load`(filename)	Load the state of the module from file.
`reset`(t, **kwargs)	Reset the agent's state.
`save`(filename)	Save the current state of the module to file.
`update`(dt)	Update the agent and the agent module.

mlpy.agents.Agent.enter

`Agent.enter(t)`

Enter the agent and the agent module.

Perform initialization tasks here.

Parameters **t** : float

The current time (sec).

mlpy.agents.Agent.exit

`Agent.exit()`

Exit the agent and the agent module.

Perform cleanup tasks here.

mlpy.agents.Agent.is_task_complete

`Agent.is_task_complete()`

Check if the agent's task is completed.

This could be because a terminal state was reached.

Returns bool :

The value of the termination flag.

mlpy.agents.Agent.load

`Agent.load(filename)`

Load the state of the module from file.

Parameters **filename** : str

The name of the file to load from.

Notes

This is a class method, it can be accessed without instantiation.

mlpy.agents.Agent.reset

`Agent.reset(t, **kwargs)`

Reset the agent's state.

Parameters **t** : float

The current time (sec).

kwargs : dict, optional

Non-positional parameters.

mlpy.agents.Agent.save

`Agent.save(filename)`

Save the current state of the module to file.

Parameters **filename** : str

The name of the file to save to.

mlpy.agents.Agent.update

`Agent.update(dt)`

Update the agent and the agent module.

The agent and the agent module are updated at every time step in the program loop.

Parameters **dt** : float

The elapsed time (sec)

mlpy.agents.modules.AgentModuleFactory

class `mlpy.agents.modules.AgentModuleFactory`

Bases: `object`

The agent module factory.

An instance of an agent module can be created by passing the agent module type. The module type is the name of the module. The set of agent modules can be extended by inheriting from `IAgentModule`. However, for the agent module to be registered, the custom module must be imported by the time the agent module factory is called.

Notes

The agent module factory is being called by the `Agent` during initialization to create the agents controller.

Examples

```
>>> from mlpy.agents.modules import AgentModuleFactory
>>> AgentModuleFactory.create('learningmodule', 'qlearner', max_steps=10)
```

This creates a `LearningModule` instance performing q-learning with max_steps set to 10.

```
>>> def get_reward(state, action):
...     return 1.0
...
>>> AgentModuleFactory().create('learningmodule', 'qlearner', get_reward,
...                                 max_steps=10)
```

This creates a q-learning learning module instance passing a reward callback function and sets max_steps to 10.

```
>>> from mlpy.mdp.discrete import DiscreteModel
>>> from mlpy.planners.discrete import ValueIteration
>>>
>>> planner = ValueIteration(DiscreteModel(['out', 'in', 'kick']))
>>>
>>> AgentModuleFactory().create('learningmodule', 'rldtlearner', None, planner,
...                                 max_steps=10)
```

This creates a learning module using the `RLDTLearner`. The parameters for the learner are appended to the end of the argument list. Notice that since positional arguments are used to pass the planner, the reward callback must be accounted for by setting it to *None*.

Alternatively non-positional arguments can be used:

```
>>> AgentModuleFactory().create('learningmodule', 'rldtlearner', planner=planner,
...                                 max_steps=10)
```

Methods

`create`(_type, *args, **kwargs)	Create an agent module of the given type.

mlpy.agents.modules.AgentModuleFactory.create

static `AgentModuleFactory.`**`create`**(*_type*, **args*, ***kwargs*)

Create an agent module of the given type.

Parameters **_type** : str

The agent module type. Valid agent module types:

followpolicymodule The agent follows a given policy: a *`FollowPolicyModule`* is created.

learningmodule The agent learns according to a specified learner: a *`LearningModule`* is created.

usermodule The agent is user controlled via keyboard or PS2 controller: a *`UserModule`* is created.

args : tuple, optional

Positional arguments passed to the class of the given type for initialization.

kwargs : dict, optional

Non-positional arguments passed to the class of the given type for initialization.

Returns IAgentModule :

Agent model instance of given type.

mlpy.agents.modules.IAgentModule

class `mlpy.agents.modules.`**`IAgentModule`**

Bases: *`mlpy.modules.Module`*

Agent module base interface class.

The agent (*`Agent`*) uses an agent module, which specifies how the agent is controlled. Valid agent module types are:

followpolicymodule The agent follows a given policy (*`FollowPolicyModule`*)

learningmodule The agent learns according to a specified learner (*`LearningModule`*).

usermodule The agent is user controlled via keyboard or PS2 controller (*`UserModule`*).

Notes

Every class inheriting from IAgentModule must implement *`get_next_action`*.

Attributes

`mid`	The module's unique identifier.

mlpy.agents.modules.IAgentModule.mid

`IAgentModule.`**`mid`**

The module's unique identifier.

Returns str :

The module's unique identifier

Methods

`enter`(t)	Enter the module and perform initialization tasks.
`execute`(state)	Execute the agent module.
`exit`()	Exit the module and perform cleanup tasks.
`get_next_action`()	Return the next action the agent will execute.
`is_complete`()	Check if the agent module has completed.
`load`(filename)	Load the state of the module from file.
`reset`(t, **kwargs)	Reset the agent module.
`save`(filename)	Save the current state of the module to file.
`terminate`(value)	Set the termination flag.
`update`(dt)	Update the module at every delta time step dt.

mlpy.agents.modules.IAgentModule.enter

`IAgentModule.`**`enter`**`(`*t*`)`

Enter the module and perform initialization tasks.

Parameters **t** : float

The current time (sec)

mlpy.agents.modules.IAgentModule.execute

`IAgentModule.`**`execute`**`(`*state*`)`

Execute the agent module. This method can optionally be overwritten.

Parameters **state** : State

The current state

mlpy.agents.modules.IAgentModule.exit

`IAgentModule.`**`exit`**`()`

Exit the module and perform cleanup tasks.

mlpy.agents.modules.IAgentModule.get_next_action

`IAgentModule.`**`get_next_action`**`()`

Return the next action the agent will execute.

Returns Action :

The next action

Raises **NotImplementedError**

If the child class does not implement this function.

Notes

This is an abstract method and *must* be implemented by its deriving class.

mlpy.agents.modules.IAgentModule.is_complete

`IAgentModule.`**`is_complete`**`()`

Check if the agent module has completed.

Returns bool :

Whether the agent module has completed or not.

mlpy.agents.modules.IAgentModule.load

`IAgentModule.`**`load`**`(`*filename*`)`

Load the state of the module from file.

Parameters **filename** : str

The name of the file to load from.

Notes

This is a class method, it can be accessed without instantiation.

mlpy.agents.modules.IAgentModule.reset

`IAgentModule.`**`reset`**`(`*t*, ***kwargs*`)`

Reset the agent module.

Parameters **t** : float

The current time (sec)

kwargs : dict, optional

Non-positional parameters, optional.

mlpy.agents.modules.IAgentModule.save

`IAgentModule.`**`save`**`(`*filename*`)`

Save the current state of the module to file.

Parameters **filename** : str

The name of the file to save to.

mlpy.agents.modules.IAgentModule.terminate

`IAgentModule.`**`terminate`**`(`*value*`)`

Set the termination flag.

Parameters **value** : bool

The value of the termination flag.

mlpy.agents.modules.IAgentModule.update

`IAgentModule.`**`update`**(*dt*)

Update the module at every delta time step dt.

Parameters dt : float

The elapsed time (sec)

mlpy.agents.modules.LearningModule

class `mlpy.agents.modules.`**`LearningModule`**(*learner_type*, *cb_get_reward=None*, **args*, ***kwargs*)

Bases: *`mlpy.agents.modules.IAgentModule`*

Learning agent module.

The learning agent module allows the agent to learn from passed experiences.

Parameters learner_type : str

The learning type. Based on the type the appropriate learner module is created. Valid learning types are:

qlearner The learner performs q-learning, a reinforcement learning variant (*`QLearner`*).

rldtlearner The learner performs reinforcement learning with decision trees (RLDT), a method introduced by Hester, Quinlan, and Stone which builds a generalized model for the transitions and rewards of the environment (*`RLDTLearner`*).

apprenticeshiplearner The learner performs apprenticeship learning via inverse reinforcement learning, a method introduced by Abbeel and Ng which strives to imitate the demonstrations given by an expert (*`ApprenticeshipLearner`*).

incrapprenticeshiplearner The learner incrementally performs apprenticeship learning via inverse reinforcement learning. Inverse reinforcement learning assumes knowledge of the underlying model. However, this is not always feasible. The incremental apprenticeship learner updates its model after every iteration by executing the current policy (*`IncrApprenticeshipLearner`*).

cb_get_reward : callable, optional

A callback function to retrieve the reward based on the current state and action. Default is *None*.

The function must be of the following format:

```
>>> def callback(state, action):
>>>     pass
```

learner_params : dict, optional

Parameters passed to the learner for initialization. See the appropriate learner type for more information. Default is None.

Attributes

`mid`	The module's unique identifier.

mlpy.agents.modules.LearningModule.mid

`LearningModule.`**`mid`**

The module's unique identifier.

Returns str :

The module's unique identifier

Methods

`enter`(t)	Enter the module and perform initialization tasks.
`execute`(state)	Execute the learner.
`exit`()	Exit the module and perform cleanup tasks.
`get_next_action`()	Return the next action.
`is_complete`()	Check if the agent module has completed.
`load`(filename)	Load the state of the module from file.
`reset`(t, **kwargs)	Reset the module for the next iteration.
`save`(filename)	Save the current state of the module to file.
`terminate`(value)	Set the termination flag.
`update`(dt)	Update the module at every delta time step dt.

mlpy.agents.modules.LearningModule.enter

`LearningModule.`**`enter`**`(`*t*`)`

Enter the module and perform initialization tasks.

Parameters **t** : float

The current time (sec)

mlpy.agents.modules.LearningModule.execute

`LearningModule.`**`execute`**`(`*state*`)`

Execute the learner.

Update models with the current experience (`Experience`). Additionally, online learning is performed at this point.

Parameters **state** : State

The current state.

mlpy.agents.modules.LearningModule.exit

`LearningModule.`**`exit`**`()`

Exit the module and perform cleanup tasks.

mlpy.agents.modules.LearningModule.get_next_action

`LearningModule.`**`get_next_action`**`()`

Return the next action.

The next action the agent will execute is selected based on the current state and the policy that has been derived by the learner so far.

Returns Action :

The next action

mlpy.agents.modules.LearningModule.is_complete

`LearningModule.`**`is_complete`**`()`

Check if the agent module has completed.

Returns bool :

Whether the agent module has completed or not.

mlpy.agents.modules.LearningModule.load

`LearningModule.`**`load`**`(`*filename*`)`

Load the state of the module from file.

Parameters **filename** : str

The name of the file to load from.

Notes

This is a class method, it can be accessed without instantiation.

mlpy.agents.modules.LearningModule.reset

`LearningModule.`**`reset`**`(`*t*, ***kwargs*`)`

Reset the module for the next iteration.

Offline learning is performed at the end of the iteration.

Parameters **t** : float

The current time (sec).

kwargs : dict, optional

Non-positional parameters.

mlpy.agents.modules.LearningModule.save

`LearningModule.`**`save`**`(`*filename*`)`

Save the current state of the module to file.

Parameters **filename** : str

The name of the file to save to.

mlpy.agents.modules.LearningModule.terminate

`LearningModule.`**`terminate`**`(`*value*`)`

Set the termination flag.

Parameters **value** : bool

The value of the termination flag.

mlpy.agents.modules.LearningModule.update

`LearningModule.`**`update`**`(`*dt*`)`

Update the module at every delta time step dt.

Parameters **dt** : float

The elapsed time (sec)

mlpy.agents.modules.FollowPolicyModule

class `mlpy.agents.modules.`**`FollowPolicyModule`**`(`*policies*, *niter=None*, *start=None*`)`

Bases: *`mlpy.agents.modules.IAgentModule`*

The follow policy agent module.

The follow policy agent module follows a given policy choosing the next action based on that policy.

Parameters **policies** : array_like, shape (*n*, *nfeatures*, *ni*)

A list of policies (i.e., action sequences), where *n* is the number of policies, *nfeatures* is the number of action features, and *ni* is the sequence length.

niter : int, optional

The number of times each policy is repeated. Default is 1.

start : int, optional

The first policy to execute. Default is 0.

Attributes

`mid`	The module’s unique identifier.

mlpy.agents.modules.FollowPolicyModule.mid

`FollowPolicyModule.`**`mid`**

The module’s unique identifier.

Returns str :

The module’s unique identifier

Methods

change_policies(policies)	Exchange the list of policies.
enter(t)	Enter the module and perform initialization tasks.
execute(state)	Execute the agent module.
exit()	Exit the module and perform cleanup tasks.
get_next_action()	Return the next action.
is_complete()	Check if the agent module has completed.
load(filename)	Load the state of the module from file.
reset(t, **kwargs)	Reset the module for the next iteration.
save(filename)	Save the current state of the module to file.
terminate(value)	Set the termination flag.
update(dt)	Update the module at every delta time step dt.

mlpy.agents.modules.FollowPolicyModule.change_policies

`FollowPolicyModule.`**`change_policies`**(*policies*)

Exchange the list of policies.

Parameters **policies** : array_like, shape (*n*, *nfeatures*, *ni*)

A list of policies (i.e., action sequences), where *n* is the number of policies, *nfeatures* is the number of action features, and *ni* is the sequence length.

Raises **IndexError**

If the list is empty.

mlpy.agents.modules.FollowPolicyModule.enter

`FollowPolicyModule.`**`enter`**(*t*)

Enter the module and perform initialization tasks.

Parameters **t** : float

The current time (sec)

mlpy.agents.modules.FollowPolicyModule.execute

`FollowPolicyModule.`**`execute`**(*state*)

Execute the agent module. This method can optionally be overwritten.

Parameters **state** : State

The current state

mlpy.agents.modules.FollowPolicyModule.exit

`FollowPolicyModule.`**`exit`**()

Exit the module and perform cleanup tasks.

mlpy.agents.modules.FollowPolicyModule.get_next_action

`FollowPolicyModule.get_next_action()`

Return the next action.

The next action the agent will execute is selected based on the current state and the policy that has been derived by the learner so far.

Returns Action :

The next action

mlpy.agents.modules.FollowPolicyModule.is_complete

`FollowPolicyModule.is_complete()`

Check if the agent module has completed.

Returns bool :

Whether the agent module has completed or not.

mlpy.agents.modules.FollowPolicyModule.load

`FollowPolicyModule.load(filename)`

Load the state of the module from file.

Parameters **filename** : str

The name of the file to load from.

Notes

This is a class method, it can be accessed without instantiation.

mlpy.agents.modules.FollowPolicyModule.reset

`FollowPolicyModule.reset(t, **kwargs)`

Reset the module for the next iteration.

Offline learning is performed at the end of the iteration.

Parameters **t** : float

The current time (sec).

kwargs : dict, optional

Non-positional parameters.

mlpy.agents.modules.FollowPolicyModule.save

`FollowPolicyModule.save(filename)`

Save the current state of the module to file.

Parameters **filename** : str

The name of the file to save to.

mlpy.agents.modules.FollowPolicyModule.terminate

`FollowPolicyModule.`**`terminate`**`(`*value*`)`

Set the termination flag.

Parameters **value** : bool

The value of the termination flag.

mlpy.agents.modules.FollowPolicyModule.update

`FollowPolicyModule.`**`update`**`(`*dt*`)`

Update the module at every delta time step dt.

Parameters **dt** : float

The elapsed time (sec)

mlpy.agents.modules.UserModule

class `mlpy.agents.modules.`**`UserModule`**`(`*events_map*, *niter=None*`)`

Bases: *`mlpy.agents.modules.IAgentModule`*

The user agent module.

With the user agent module the agent is controlled by the user via the keyboard or a PS2 controller. The mapping of keyboard/joystick keys to events is given through a configuration file.

Parameters **events_map** : ConfigMgr

The configuration mapping keyboard/joystick keys to events that are translated into actions.

Example

```
{
    "keyboard": {
        "down": {
            "pygame.K_ESCAPE": "QUIT",
            "pygame.K_SPACE": [-1.0],
            "pygame.K_LEFT" : [-0.004],
            "pygame.K_RIGHT":  [0.004]
        }
    }
}
```

niter : int

The number of episodes to capture..

Notes

This agent module is requires the PyGame library.

Attributes

mid	The module's unique identifier.

mlpy.agents.modules.UserModule.mid

`UserModule.`**`mid`**

The module's unique identifier.

Returns str :

The module's unique identifier

Methods

enter(t)	Enter the module and perform initialization tasks.
execute(state)	Execute the agent module.
exit()	Exit the agent module.
get_next_action()	Return the next action.
is_complete()	Check if the agent module has completed.
load(filename)	Load the state of the module from file.
reset(t, **kwargs)	Reset the module for the next iteration.
save(filename)	Save the current state of the module to file.
terminate(value)	Set the termination flag.
update(dt)	Update the module at every delta time step dt.

mlpy.agents.modules.UserModule.enter

`UserModule.`**`enter`**`(`*t*`)`

Enter the module and perform initialization tasks.

Parameters **t** : float

The current time (sec)

mlpy.agents.modules.UserModule.execute

`UserModule.`**`execute`**`(`*state*`)`

Execute the agent module. This method can optionally be overwritten.

Parameters **state** : State

The current state

mlpy.agents.modules.UserModule.exit

`UserModule.`**`exit`**`()`

Exit the agent module.

mlpy.agents.modules.UserModule.get_next_action

`UserModule.`**`get_next_action`**`()`

Return the next action.

Return the next action the agent will execute depending on the key/button pressed.

Returns Action :

The next action

mlpy.agents.modules.UserModule.is_complete

`UserModule.is_complete()`

Check if the agent module has completed.

Returns bool :

Whether the agent module has completed or not.

mlpy.agents.modules.UserModule.load

`UserModule.load(`*filename*`)`

Load the state of the module from file.

Parameters **filename** : str

The name of the file to load from.

Notes

This is a class method, it can be accessed without instantiation.

mlpy.agents.modules.UserModule.reset

`UserModule.reset(`*t*, ***kwargs*`)`

Reset the module for the next iteration.

Offline learning is performed at the end of the iteration.

Parameters **t** : float

The current time (sec).

kwargs : dict, optional

Non-positional parameters.

mlpy.agents.modules.UserModule.save

`UserModule.save(`*filename*`)`

Save the current state of the module to file.

Parameters **filename** : str

The name of the file to save to.

mlpy.agents.modules.UserModule.terminate

`UserModule.terminate(value)`

Set the termination flag.

Parameters **value** : bool

The value of the termination flag.

mlpy.agents.modules.UserModule.update

`UserModule.update(dt)`

Update the module at every delta time step dt.

Parameters **dt** : float

The elapsed time (sec)

World Model

`WorldObject`	The world object base class.
`WorldModel`	The world model.

mlpy.agents.world.WorldObject

class `mlpy.agents.world.WorldObject`

Bases: `mlpy.modules.Module`

The world object base class.

The world object base class keeps track of the location of the object and its level of confidence for the information based on when the object was last seen.

Notes

All world objects should derive from this class.

Todo

Update the location based on localization.

Attributes

`confidence`	The level of confidence of the object's information based on when the object was last seen.

mlpy.agents.world.WorldObject.confidence

`WorldObject.`**`confidence`**

The level of confidence of the object's information based on when the object was last seen.

Returns float :

The level of confidence.

location	(Points3D) The objects current location.
timestamp	(float) The timestamp the object was last seen (the image was captured).

Methods

enter(t)	Enter the world object.
exit()	Exit the module and perform cleanup tasks.
load(filename)	Load the state of the module from file.
reset(t, **kwargs)	Reset the module.
save(filename)	Save the current state of the module to file.
update(dt)	Update the world object based on the elapsed time.
update_confidence()	Update the level of confidence.

mlpy.agents.world.WorldObject.enter

`WorldObject.`**`enter`**(*t*)

Enter the world object.

Parameters **t** : float

The current time (sec)

mlpy.agents.world.WorldObject.exit

`WorldObject.`**`exit`**()

Exit the module and perform cleanup tasks.

mlpy.agents.world.WorldObject.load

`WorldObject.`**`load`**(*filename*)

Load the state of the module from file.

Parameters **filename** : str

The name of the file to load from.

Notes

This is a class method, it can be accessed without instantiation.

mlpy.agents.world.WorldObject.reset

`WorldObject.`**`reset`**`(`*t*, ***kwargs*`)`

Reset the module.

Parameters **t** : float

The current time (sec)

kwargs : dict

Additional non-positional parameters.

mlpy.agents.world.WorldObject.save

`WorldObject.`**`save`**`(`*filename*`)`

Save the current state of the module to file.

Parameters **filename** : str

The name of the file to save to.

mlpy.agents.world.WorldObject.update

`WorldObject.`**`update`**`(`*dt*`)`

Update the world object based on the elapsed time.

Parameters **dt** : float

The elapsed time (sec)

mlpy.agents.world.WorldObject.update_confidence

`WorldObject.`**`update_confidence`**`()`

Update the level of confidence.

Based on when the object was last seen, the level of confidence for the information available on the object is update.

mlpy.agents.world.WorldModel

class `mlpy.agents.world.`**`WorldModel`**

Bases: *`mlpy.modules.Module`*

The world model.

The world model manages the world objects of type *`WorldObject`* by ensuring that the objects are updated with the latest information at every time step of the program loop. Furthermore, information of the world objects can be accessed from the world model.

Notes

The world module follows the singleton design pattern (*`Singleton`*), ensuring only one instances of the world module exist. This allows for accessing the information of the world model from anywhere in the program.

Examples

```
>>> from mlpy.agents.world import WorldModel, WorldObject
>>> WorldModel().add_object("ball", WorldObject)
```

```
>>> from mlpy.agents.world import WorldModel
>>> ball = WorldModel().get_object("ball")
```

Attributes

mid	The module's unique identifier.

mlpy.agents.world.WorldModel.mid

`WorldModel.`**`mid`**

The module's unique identifier.

Returns str :

The module's unique identifier

Methods

add_object(name, obj)	Add a world object.
enter(t)	Enter the world model.
exit()	Exit the module and perform cleanup tasks.
get_object(name)	Returns the object with the given name.
load(filename)	Load the state of the module from file.
reset(t, **kwargs)	Reset the module.
save(filename)	Save the current state of the module to file.
update(dt)	Update all world objects.

mlpy.agents.world.WorldModel.add_object

`WorldModel.`**`add_object`**(*name*, *obj*)

Add a world object.

Parameters **name** : str

The identifier of the world object.

obj : WorldObject

The world object instance.

Raises **AttributeError**

If an world object with the given name has already been registered.

mlpy.agents.world.WorldModel.enter

`WorldModel.enter(t)`
Enter the world model.

Parameters **t** : float

The current time (sec)

mlpy.agents.world.WorldModel.exit

`WorldModel.exit()`
Exit the module and perform cleanup tasks.

mlpy.agents.world.WorldModel.get_object

`WorldModel.get_object(name)`
Returns the object with the given name.

Parameters **name** : str

The identifier of the world object.

Returns WorldObject :

The world object

mlpy.agents.world.WorldModel.load

`WorldModel.load(filename)`
Load the state of the module from file.

Parameters **filename** : str

The name of the file to load from.

Notes

This is a class method, it can be accessed without instantiation.

mlpy.agents.world.WorldModel.reset

`WorldModel.reset(t, **kwargs)`
Reset the module.

Parameters **t** : float

The current time (sec)

kwargs : dict

Additional non-positional parameters.

mlpy.agents.world.WorldModel.save

`WorldModel.`**`save`**`(`*filename*`)`

Save the current state of the module to file.

Parameters **filename** : str

The name of the file to save to.

mlpy.agents.world.WorldModel.update

`WorldModel.`**`update`**`(`*dt*`)`

Update all world objects.

The world objects are updated at each time step of the program loop.

Parameters **dt** : float

The elapsed time (sec)

Finite State Machine

`Event`	Transition event definition.
`EmptyEvent`	A no-op transition event.
`FSMState`	State base class.
`Transition`	Transition class.
`OnUpdate`	OnUpdate class.
`StateMachine`	The finite state machine.

mlpy.agents.fsm.Event

class `mlpy.agents.fsm.`**`Event`**`(`*name*, *state=None*, *machine=None*, *delay=None*, **args*, ***kwargs*`)`

Bases: `object`

Transition event definition.

When transitioning from a source state to a destination state a transition event is fired.

Parameters **name** : str

Name of the event.

state : FSMState, optional

The current state.

machine : StateMachine, optional

Reference to the state machine.

delay : int, optional

The amount of time (milliseconds) by which checking this event is delayed. Default is 0.

args : tuple, optional

Positional parameters passed to the next state.

kwargs : dict, optional

Non-positional parameters passed to the next state.

Methods

ready()	Check if the event is ready.

mlpy.agents.fsm.Event.ready

`Event.ready()`

Check if the event is ready.

Check if the event has waited the requested amount of time. If so, the event fires.

mlpy.agents.fsm.EmptyEvent

class `mlpy.agents.fsm.`**`EmptyEvent`**(*state=None*, *machine=None*, *delay=None*, **args*, ***kwargs*)

Bases: *mlpy.agents.fsm.Event*

A no-op transition event.

A no-op transition event does nothing when it is fired; i.e. it stays in the same state without transitioning.

Parameters **state** : FSMState, optional

The current state.

machine : StateMachine, optional

Reference to the state machine.

delay : int, optional

The amount of time (milliseconds) by which checking this event is delayed. Default is 0.

args : tuple, optional

Positional parameters passed to the next state.

kwargs : dict, optional

Non-positional parameters passed to the next state.

Methods

ready()	Check if the event is ready.

mlpy.agents.fsm.EmptyEvent.ready

`EmptyEvent.ready()`

Check if the event is ready.

Check if the event has waited the requested amount of time. If so, the event fires.

mlpy.agents.fsm.FSMState

class `mlpy.agents.fsm.`**`FSMState`**

Bases: *`mlpy.modules.Module`*

State base class.

A state of the finite state machine.

Attributes

`mid`	The module's unique identifier.
`name`	Name of the state.

mlpy.agents.fsm.FSMState.mid

`FSMState.`**`mid`**

The module's unique identifier.

Returns str :

The module's unique identifier

mlpy.agents.fsm.FSMState.name

`FSMState.`**`name`**

Name of the state.

Returns str :

The state's name.

Methods

`enter`(t, *args, **kwargs)	State initialization.
`exit`()	Perform cleanup tasks.
`load`(filename)	Load the state of the module from file.
`reset`(t, **kwargs)	Reset the module.
`save`(filename)	Save the current state of the module to file.
`update`(dt)	Update the state.

mlpy.agents.fsm.FSMState.enter

`FSMState.`**`enter`**(*t*, **args*, ***kwargs*)

State initialization.

Parameters **t** : float

The current time (sec)

mlpy.agents.fsm.FSMState.exit

`FSMState.`**`exit`**`()`

Perform cleanup tasks.

mlpy.agents.fsm.FSMState.load

`FSMState.`**`load`**`(`*filename*`)`

Load the state of the module from file.

Parameters **filename** : str

The name of the file to load from.

Notes

This is a class method, it can be accessed without instantiation.

mlpy.agents.fsm.FSMState.reset

`FSMState.`**`reset`**`(`*t*, ***kwargs*`)`

Reset the module.

Parameters **t** : float

The current time (sec)

kwargs : dict

Additional non-positional parameters.

mlpy.agents.fsm.FSMState.save

`FSMState.`**`save`**`(`*filename*`)`

Save the current state of the module to file.

Parameters **filename** : str

The name of the file to save to.

mlpy.agents.fsm.FSMState.update

`FSMState.`**`update`**`(`*dt*`)`

Update the state.

Update the state and handle state transitions based on events.

Parameters **dt** : float

The elapsed time (sec)

Returns Event :

The transition event.

mlpy.agents.fsm.Transition

class `mlpy.agents.fsm.`**`Transition`**(*source*, *dest*, *conditions=None*, *before=None*, *after=None*)

Bases: `object`

Transition class.

Each transition contains a source and a destination state. Furthermore, conditions for transitioning and callbacks before and after transitioning can be specified.

Parameters **source** : str

The source state.

dest : str

The destination state.

conditions : list[callable]

The transition is only executed once the condition(s) have been met.

before : callable

Callback function to be called before exiting the source state.

after : callable

Callback function to be called after entering the destination state.

Methods

`execute`(event)	Execute the transition.

mlpy.agents.fsm.Transition.execute

`Transition.`**`execute`**(*event*)

Execute the transition.

The transition is only executed, if all conditions are met.

Parameters **event** : Event

The transition event.

Returns bool :

Whether the transition was executed or not.

mlpy.agents.fsm.OnUpdate

class `mlpy.agents.fsm.`**`OnUpdate`**(*source*, *onupdate=None*, *conditions=None*)

Bases: `object`

OnUpdate class.

On update of the current state, a callback can be specified which will be called if the conditions have been met.

Parameters **source** : str

The source state.

onupdate: callable

The callback function to be called

conditions : list[callable]

The condition(s) which have to be met in order for the callback to be called.

Methods

execute(machine)	Execute the callback.

mlpy.agents.fsm.OnUpdate.execute

OnUpdate.**execute**(*machine*)

Execute the callback.

The callbacks are only called if all conditions are met.

Parameters machine : StateMachine

Reference to the state machine.

mlpy.agents.fsm.StateMachine

class mlpy.agents.fsm.**StateMachine**(*states=None*, *initial=None*, *transitions=None*, *onupdate=None*)

Bases: *mlpy.modules.Module*

The finite state machine.

The finite state machine handles state transitions, by triggering events. Events can also be fired from outside the state machine to force a transition.

Parameters states : FSMState | list[FSMState], optional

A list of states.

initial : str, optional

The initial state.

transitions : list[dict] | list[list], optional

Transition information.

onupdate : list[dict] | list[list], optional

Callback information to be executed on update.

Attributes

current_state	The current event.
mid	The module's unique identifier.

mlpy.agents.fsm.StateMachine.current_state

StateMachine.**current_state**

The current event.

Returns FSMState :

the current state.

mlpy.agents.fsm.StateMachine.mid

StateMachine.**mid**

The module's unique identifier.

Returns str :

The module's unique identifier

Methods

add_onupdate(source[, onupdate, conditions])	Add a callback to be called on update.
add_states(states)	Add new state(s) to the managed states.
add_transition(event, source, dest[, ...])	Add a transition from a source state to a destination state.
clear_events([state_name])	Clear all events.
enter(t)	Enter the current state.
exit()	Exit the finite state machine.
get_state(state)	Return the FSMState instance with the given name.
load(filename)	Load the state of the module from file.
load_from_file(owner, filename, **kwargs)	Load the FSM from file.
post_event(e, *args, **kwargs)	An event is added to the events list.
reset(t, **kwargs)	Reset the finite state machine and all registered states.
save(filename)	Save the current state of the module to file.
set_state(state)	Set the current state.
update(dt)	Update state and handle event transitions.

mlpy.agents.fsm.StateMachine.add_onupdate

StateMachine.**add_onupdate**(*source*, *onupdate=None*, *conditions=None*)

Add a callback to be called on update.

Parameters **source** : str

The state for which the callback will be triggered.

onupdate : callable

The callback function.

conditions : list[callable]

The conditions that must be met in order to execute the callback.

Raises **ValueError**

If the source is not a registered state.

Notes

By setting source to *, the callbacks will be called for all states.

mlpy.agents.fsm.StateMachine.add_states

`StateMachine.`**`add_states`**`(`*states*`)`

Add new state(s) to the managed states.

Parameters **states** : FSMState | list[FSMState]

The state(s) to be added.

mlpy.agents.fsm.StateMachine.add_transition

`StateMachine.`**`add_transition`**`(`*event*, *source*, *dest*, *conditions=None*, *before=None*, *after=None*`)`

Add a transition from a source state to a destination state.

Parameters **event** : str

The event driving the transition.

source : str

The source state.

dest : str

The destination state.

conditions : list[callable]

The conditions that must be met for transition to execute.

before : callable

Callback function to be called before exiting the source state.

after : callable

Callback function to be called after entering the source state.

Raises **ValueError**

If the source or the destination is not a registered state.

Notes

By setting source to *, the callbacks will be called for all states.

mlpy.agents.fsm.StateMachine.clear_events

`StateMachine.`**`clear_events`**`(`*state_name=None*`)`

Clear all events.

If state_name is given, the events are only cleared for the given state.

Parameters **state_name** : str

The name of the state for which to clear all events. If *None* all events are removed. Default is None.

mlpy.agents.fsm.StateMachine.enter

`StateMachine.`**`enter`**`(`*t*`)`

Enter the current state.

Parameters **t** : float

The current time (sec)

mlpy.agents.fsm.StateMachine.exit

`StateMachine.`**`exit`**`()`

Exit the finite state machine.

mlpy.agents.fsm.StateMachine.get_state

`StateMachine.`**`get_state`**`(`*state*`)`

Return the FSMState instance with the given name.

Parameters **state** : str

The name of the state to retrieve.

Returns FSMState :

The state.

Raises **ValueError**

If the state is not a registered state.

mlpy.agents.fsm.StateMachine.load

`StateMachine.`**`load`**`(`*filename*`)`

Load the state of the module from file.

Parameters **filename** : str

The name of the file to load from.

Notes

This is a class method, it can be accessed without instantiation.

mlpy.agents.fsm.StateMachine.load_from_file

`StateMachine.`**`load_from_file`**`(`*owner*, *filename*, ***kwargs*`)`

Load the FSM from file.

Read the information of the state machine from file. The file contains information of the states, transitions and callback function.

Parameters owner : object

Reference to the object owning the FSM.

filename : str

The name of the file containing the FSM configuration information.

kwargs : dict

Non-positional arguments match with configuration parameters during state creation.

Notes

The FSM setup can be specified via a configuration file in `.json` format. The configuration file must follow a specific format.

The configuration file must contain the absolute path to the module containing the implementation of each state. Additionally, the configuration file must contain the name of the initial state, a list of states, their transitions, and information of the *onupdate* callback functions.

Example A skeleton configuration with two states named "<initial>" and "<next>" and one simple transition between them named "<event>". The implementation of the states are defined in the file specified in "Module".

```
{
    "Module": "absolute/path/to/the/fsm/states.py",
    "States": [
        "<initial>",
        "<next>"
    ],
    "Initial": "<initial>",
    "Transitions": [
        {"source": "<initial>", "event": "<event>", "dest": "<next>
↪"},
        {"source": "<next>", "event": "<event2>", "dest": "<initial>
↪"}
    ],
    "OnUpdate": [
    ]
}
```

If the states have initialization parameters these can be specified as follows:

```
{
    "States": [
        {"<initial>": {
            "args": "motion",
            "kwargs": {"support_leg": "right"}
        }}
    ]
}
```

This lets the FSM know that the state "<initial>" has two parameters. The positional arguments (specified in *args*) are compared to non-positional arguments in kwargs passed to *load_from_file*. If a match exists, the value of the match is passed as argument. If no match exist the value in "args" is send directly to the state. Multiple positional arguments can be specified by adding them in a list: ["arg1", "arg2", ...]. The non-positional arguments are send as is to the state.

To specify callback functions before and after transitioning from a source to the destination the following formats are available:

```
{
    "Transitions": [
        {"source": "<initial>", "event": "<event>", "dest": "<next>
↪",
            "before": {"model": "<ClassName>", "func": "<FuncName>"}
↪,
            "after": {"model": "<ClassName>", "func": "<FuncName>"}}
↪,
        {"source": "<next>", "event": "<event2>", "dest": "<initial>
↪",
            "before": "<FuncName>",
            "after": "<FuncName>"}
    ]
}
```

It is also possible to define conditions on the transitions, such that a transition between states is only performed when the condition(s) are met:

```
{
    "Transitions": [
        {"source": "<initial>", "event": "<event>", "dest": "<next>
↪",
            "conditions": ["lambda x: not x._motion.is_running",
                           "FuncName"]
            "before": {"model": "<ClassName>", "func": "<FuncName>"}
↪,
            "after": {"model": "<ClassName>", "func": "<FuncName>"}}
↪,
        {"source": "<next>", "event": "<event2>", "dest": "<initial>
↪",
            "conditions": "FuncName"
            "before": "<FuncName>",
            "after": "<FuncName>"}
    ]
}
```

It is also possible to add a transition to every state by using `*`:

```
{
    "Transitions": [
        {"source": "*", "event": "<event>", "dest": "*"},
    ]
}
```

This statement means that "<event>" is a valid event from every state to every other state.

To identify the *onupdate* callback functions use the following format:

```
{
    "OnUpdate": [
        {"source": "<initial>", "conditions": ["lambda x: not x._
↪motion.is_running",
                                               "FuncName"]
            "onupdate": {"model": "<ClassName>", "func": "<FuncName>
↪"}},
    ]
}
```

This lets the FSM know to call the function specified in "onupdate" when in state "<initial>" when the conditions are met. The conditions are optional. Also, instead of calling a class function a lambda or other function can be called here.

mlpy.agents.fsm.StateMachine.post_event

StateMachine.**post_event**(*e*, **args*, ***kwargs*)

An event is added to the events list.

The first event that meets all the conditions will be executed.

Parameters **e** : str | Event

The event

args : tuple, optional

Positional parameters send to the next state.

kwargs : dict, optional

Non-positional parameters send to the next state.

Raises **ValueError**

If the event *e* is not a string or an instance of Event.

ValueError

If event e is not a registered transition event or the event is not registered for the current state.

mlpy.agents.fsm.StateMachine.reset

StateMachine.**reset**(*t*, ***kwargs*)

Reset the finite state machine and all registered states.

Parameters **t** : float

The current time (sec).

kwargs : dict, optional

Non-positional parameters.

mlpy.agents.fsm.StateMachine.save

StateMachine.**save**(*filename*)

Save the current state of the module to file.

Parameters **filename** : str

The name of the file to save to.

mlpy.agents.fsm.StateMachine.set_state

`StateMachine.`**`set_state`**`(`*state*`)`
Set the current state.

Parameters **state** : str or FSMState

The (name of the) state.

Raises **ValueError**

If the state is not a string or an instance of FSMState.

mlpy.agents.fsm.StateMachine.update

`StateMachine.`**`update`**`(`*dt*`)`
Update state and handle event transitions.

Parameters **dt** : float

The elapsed time (sec)

CHAPTER 9

Auxiliary functions (`mlpy.auxiliary`)

This modules.

Array

`accum`	An accumulation function similar to Matlab's *accumarray* function.
`normalize`	Normalize the input array to sum to *1*.
`nunique`	Efficiently count the unique elements of *x* along the given axis.

mlpy.auxiliary.array.accum

`mlpy.auxiliary.array.`**`accum`**(*accmap*, *a*, *func=None*, *size=None*, *fill_value=0*, *dtype=None*)

An accumulation function similar to Matlab's *accumarray* function.

Parameters accmap : array_like

This is the "accumulation map". It maps input (i.e. indices into *a*) to their destination in the output array. The first *a.ndim* dimensions of *accmap* must be the same as *a.shape*. That is, *accmap.shape[:a.ndim]* must equal *a.shape*. For example, if *a* has shape (15,4), then *accmap.shape[:2]* must equal (15,4). In this case *accmap[i,j]* gives the index into the output array where element (i,j) of *a* is to be accumulated. If the output is, say, a 2D, then *accmap* must have shape (15,4,2). The value in the last dimension give indices into the output array. If the output is 1D, then the shape of *accmap* can be either (15,4) or (15,4,1)

a : array_like or float or int

The input data to be accumulated.

func : callable or None

The accumulation function. The function will be passed a list of values from *a* to be accumulated. If None, numpy.sum is assumed.

size : array_like or tuple

The size of the output array. If None, the size will be determined from *accmap*.

fill_value : scalar

The default value for elements of the output array.

dtype : dtype

The data type of the output array. If None, the data type of *a* is used.

Returns array_like :

The accumulated results.

The shape of *out* is *size* if *size* is given. Otherwise the shape is determined by the (lexicographically) largest indices of the output found in *accmap*.

Examples

```
>>> from numpy import array, prod, float64
>>> a = array([[1,2,3],[4,-1,6],[-1,8,9]])
>>> a
array([[ 1,  2,  3],
       [ 4, -1,  6],
       [-1,  8,  9]])
```

Sum the diagonals:

```
>>> accmap = array([[0,1,2],[2,0,1],[1,2,0]])
>>> s = accum(accmap, a)
array([9, 7, 15])
```

A 2D output, from sub-arrays with shapes and positions like this:

[(2,2) (2,1)]
[(1,2) (1,1)]

```
>>> accmap = array([
...     [[0,0],[0,0],[0,1]],
...     [[0,0],[0,0],[0,1]],
...     [[1,0],[1,0],[1,1]],
... ])
```

Accumulate using a product:

```
>>> accum(accmap, a, func=prod, dtype=float64)
array([[ -8.,  18.],
       [ -8.,   9.]])
```

Same accmap, but create an array of lists of values:

```
>>> accum(accmap, a, func=lambda x: x, dtype='O')
array([[[1, 2, 4, -1], [3, 6]],
       [[-1, 8], [9]]], dtype=object)
```

Note: Adapted from

Project: Code from SciPy Cookbook.
Code author: Warren Weckesser
License: CC-Wiki

mlpy.auxiliary.array.normalize

`mlpy.auxiliary.array.normalize(a, axis=None, return_scale=False)`

Normalize the input array to sum to *1*.

Parameters **a** : array_like, shape (*nsamples*, *nfeatures*)

Non-normalized input data array.

axis : int

Dimension along which normalization is performed.

Returns array_like, shape (*nsamples*, *nfeatures*) :

An array with values normalized (summing to 1) along the prescribed axis.

Examples

```
>>>
```

Attention: The input array *a* is modified inplace.

mlpy.auxiliary.array.nunique

`mlpy.auxiliary.array.nunique(x, axis=None)`

Efficiently count the unique elements of *x* along the given axis.

Parameters **x** : array_like

The array for which to count the unique elements.

axis : int

Dimension along which to count the unique elements.

Returns int or array_like :

The number of unique elements along the given axis.

Examples

```
>>>
```

Note: Ported from Matlab:

Project: Probabilistic Modeling Toolkit for Matlab/Octave.
Copyright (2010) Kevin Murphy and Matt Dunham
License: MIT

I/O

import_module_from_path	Import a module from a file path and return the module object.
load_from_file	Load data from file.
save_to_file	Saves data to file.
is_pickle	Check if the file with the given name is `pickle` encoded.
txt2pickle	Converts a text file into a `pickle` encoded file.

mlpy.auxiliary.io.import_module_from_path

`mlpy.auxiliary.io.`**`import_module_from_path`**(*full_path*, *global_name*)

Import a module from a file path and return the module object.

Allows one to import from anywhere, something `__import__` does not do. The module is added to `sys.modules` as *global_name*.

Parameters **full_path** : str

The absolute path to the module .py file

global_name : str

The name assigned to the module in `sys.modules`. To avoid confusion, the global_name should be the same as the variable to which you're assigning the returned module.

Examples

```
>>> from mlpy.auxiliary.io import import_module_from_path
```

Note:

Project: Code from Trigger.
Copyright (c) 2006-2012, AOL Inc.
License: BSD

mlpy.auxiliary.io.load_from_file

`mlpy.auxiliary.io.`**`load_from_file`**(*filename*, *import_modules=None*)

Load data from file.

Different formats are supported.

Parameters **filename** : str

Name of the file to load data from.

import_modules : str or list

List of modules that may be required by the data that need to be imported.

Returns dict or list :

The loaded data. If any errors occur, `None` is returned.

mlpy.auxiliary.io.save_to_file

`mlpy.auxiliary.io.`**`save_to_file`**(*filename*, *data*)

Saves data to file.

The data can be a dictionary or an object's state and is saved in `pickle` format.

Parameters **filename** : str

Name of the file to which to save the data to.

data : dict or object

The data to be saved.

mlpy.auxiliary.io.is_pickle

`mlpy.auxiliary.io.`**`is_pickle`**(*filename*)

Check if the file with the given name is `pickle` encoded.

Parameters **filename** : str

The name of the file to check.

Returns bool :

Whether the file is `pickle` encoded or not.

mlpy.auxiliary.io.txt2pickle

`mlpy.auxiliary.io.`**`txt2pickle`**(*filename*, *new_filename=None*, *func=None*)

Converts a text file into a `pickle` encoded file.

Parameters **filename** : str

The name of the file to encode.

new_filename : str

New file name to which the encoded data is saved to.

func : callable

A data encoding helper function.

Returns str :

The name of the file to which the encoded data was saved to.

Data structures

Array	The managed array class.
Point2D	The 2d-point class.
Point3D	The 3d-point class.
Vector3D	The 3d-vector class.
Queue	The abstract queue base class.
FIFOQueue	The first-in-first-out (FIFO) queue.
PriorityQueue	The priority queue.

mlpy.auxiliary.datastructs.Array

class mlpy.auxiliary.datastructs.**Array**(*size*)

Bases: object

The managed array class.

The managed array class pre-allocates memory to the given size automatically resizing as needed.

Parameters **size** : int

The size of the array.

Examples

```
>>> a = Array(5)
>>> a[0] = 3
>>> a[1] = 6
```

Retrieving an elements:

```
>>> a[0]
3
>>> a[2]
0
```

Finding the length of the array:

```
>>> len(a)
2
```

mlpy.auxiliary.datastructs.Point2D

class `mlpy.auxiliary.datastructs.`**`Point2D`**(*x=0.0*, *y=0.0*)
Bases: `object`

The 2d-point class.

The 2d-point class is a container for positions in a 2d-coordinate system.

Parameters **x** : float, optional

The x-position in a 2d-coordinate system. Default is 0.0.

y : float, optional

The y-position in a 2d-coordinate system. Default is 0.0.

Attributes

x	(float) The x-position in a 2d-coordinate system.
y	(float) The y-position in a 2d-coordinate system.

mlpy.auxiliary.datastructs.Point3D

class `mlpy.auxiliary.datastructs.`**`Point3D`**(*x=0.0*, *y=0.0*, *z=0.0*)
Bases: `object`

The 3d-point class.

The 3d-point class is a container for positions in a 3d-coordinate system.

Parameters **x** : float, optional

The x-position in a 2d-coordinate system. Default is 0.0.

y : float, optional

The y-position in a 2d-coordinate system. Default is 0.0.

z : float, optional

The z-position in a 3d-coordinate system. Default is 0.0.

Attributes

x	(float) The x-position in a 2d-coordinate system.
y	(float) The y-position in a 2d-coordinate system.
z	(float) The z-position in a 3d-coordinate system.

mlpy.auxiliary.datastructs.Vector3D

class `mlpy.auxiliary.datastructs.`**`Vector3D`**(*x=0.0*, *y=0.0*, *z=0.0*)
Bases: *`mlpy.auxiliary.datastructs.Point3D`*

The 3d-vector class.

Todo

Implement vector functionality.

Parameters **x** : float, optional

The x-position in a 2d-coordinate system. Default is 0.0.

y : float, optional

The y-position in a 2d-coordinate system. Default is 0.0.

z : float, optional

The z-position in a 3d-coordinate system. Default is 0.0.

Attributes

x	(float) The x-position in a 2d-coordinate system.
y	(float) The y-position in a 2d-coordinate system.
z	(float) The z-position in a 3d-coordinate system.

mlpy.auxiliary.datastructs.Queue

class `mlpy.auxiliary.datastructs.`**`Queue`**

Bases: `object`

The abstract queue base class.

The queue class handles core functionality common for any type of queue. All queues inherit from the queue base class.

See also:

`FIFOQueue`, *`PriorityQueue`*

Methods

`empty`()	Check if the queue is empty.
`extend`(items)	Extend the queue by a number of elements.
`get`(item)	Return the element in the queue identical to *item*.
`pop`()	Pop an element from the queue.
`push`(item)	Push a new element on the queue
`remove`(item)	Remove an element from the queue.

mlpy.auxiliary.datastructs.Queue.empty

`Queue.`**`empty`**`()`

Check if the queue is empty.

Returns bool :

Whether the queue is empty.

mlpy.auxiliary.datastructs.Queue.extend

`Queue.extend(items)`
Extend the queue by a number of elements.

Parameters **items** : list

A list of items.

mlpy.auxiliary.datastructs.Queue.get

`Queue.get(item)`
Return the element in the queue identical to *item*.

Parameters **item :**

The element to search for.

Returns The element in the queue identical to *item*. If the element

was not found, None is returned.

mlpy.auxiliary.datastructs.Queue.pop

`Queue.pop()`
Pop an element from the queue.

mlpy.auxiliary.datastructs.Queue.push

`Queue.push(item)`
Push a new element on the queue

Parameters **item :**

The element to push on the queue

mlpy.auxiliary.datastructs.Queue.remove

`Queue.remove(item)`
Remove an element from the queue.

Parameters **item :**

The element to remove.

mlpy.auxiliary.datastructs.FIFOQueue

class `mlpy.auxiliary.datastructs.FIFOQueue`
Bases: *`mlpy.auxiliary.datastructs.Queue`*

The first-in-first-out (FIFO) queue.

In a FIFO queue the first element added to the queue is the first element to be removed.

See also:

`PriorityQueue`

Examples

```
>>> q = FIFOQueue()
>>> q.push(5)
>>> q.extend([1, 3, 7])
>>> print q
[5, 1, 3, 7]
```

Retrieving an element:

```
>>> q.pop()
5
```

Removing an element:

```
>>> q.remove(3)
>>> print q
[1, 7]
```

Get the element in the queue identical to the given item:

```
>>> q.get(7)
7
```

Check if the queue is empty:

```
>>> q.empty()
False
```

Loop over the elements in the queue:

```
>>> for x in q:
>>>     print x
1
7
```

Check if an element is in the queue:

```
>>> if 7 in q:
>>>     print "yes"
yes
```

Methods

`empty`()	Check if the queue is empty.
`extend`(items)	Append a list of elements at the end of the queue.
`get`(item)	Return the element in the queue identical to *item*.
`pop`()	Return the element at the front of the queue.
`push`(item)	Push an element to the end of the queue.
`remove`(item)	Remove an element from the queue.

mlpy.auxiliary.datastructs.FIFOQueue.empty

`FIFOQueue.`**`empty`**`()`

Check if the queue is empty.

Returns bool :

Whether the queue is empty.

mlpy.auxiliary.datastructs.FIFOQueue.extend

`FIFOQueue.`**`extend`**`(`*items*`)`

Append a list of elements at the end of the queue.

Parameters **items** : list

List of elements.

mlpy.auxiliary.datastructs.FIFOQueue.get

`FIFOQueue.`**`get`**`(`*item*`)`

Return the element in the queue identical to *item*.

Parameters **item :**

The element to search for.

Returns The element in the queue identical to *item*. If the element

was not found, None is returned.

mlpy.auxiliary.datastructs.FIFOQueue.pop

`FIFOQueue.`**`pop`**`()`

Return the element at the front of the queue.

Returns The first element in the queue.

mlpy.auxiliary.datastructs.FIFOQueue.push

`FIFOQueue.`**`push`**`(`*item*`)`

Push an element to the end of the queue.

Parameters **item :**

The element to append.

mlpy.auxiliary.datastructs.FIFOQueue.remove

`FIFOQueue.`**`remove`**`(`*item*`)`

Remove an element from the queue.

Parameters **item :**

The element to remove.

mlpy.auxiliary.datastructs.PriorityQueue

class `mlpy.auxiliary.datastructs.`**`PriorityQueue`**(*func=<function <lambda>>*)

Bases: *`mlpy.auxiliary.datastructs.Queue`*

The priority queue.

In a priority queue each element has a priority associated with it. An element with high priority (i.e., smallest value) is served before an element with low priority (i.e., largest value). The priority queue is implemented with a heap.

Parameters **func** : callable

A callback function handling the priority. By default the priority is the value of the element.

See also:

`FIFOQueue`

Examples

```
>>> q = PriorityQueue()
>>> q.push(5)
>>> q.extend([1, 3, 7])
>>> print q
[(1,1), (5,5), (3,3), (7,7)]
```

Retrieving the element with highest priority:

```
>>> q.pop()
1
```

Removing an element:

```
>>> q.remove((3, 3))
>>> print q
[(5,5), (7,7)]
```

Get the element in the queue identical to the given item:

```
>>> q.get(7)
7
```

Check if the queue is empty:

```
>>> q.empty()
False
```

Loop over the elements in the queue:

```
>>> for x in q:
>>>     print x
(5, 5)
(7, 7)
```

Check if an element is in the queue:

```
>>> if 7 in q:
>>>     print "yes"
yes
```

Methods

`empty`()	Check if the queue is empty.
`extend`(items)	Extend the queue by a number of elements.
`get`(item)	Return the element in the queue identical to *item*.
`pop`()	Get the element with the highest priority.
`push`(item)	Push an element on the priority queue.
`remove`(item)	Remove an element from the queue.

mlpy.auxiliary.datastructs.PriorityQueue.empty

`PriorityQueue.`**`empty`**`()`

Check if the queue is empty.

Returns bool :

Whether the queue is empty.

mlpy.auxiliary.datastructs.PriorityQueue.extend

`PriorityQueue.`**`extend`**`(`*items*`)`

Extend the queue by a number of elements.

Parameters **items** : list

A list of items.

mlpy.auxiliary.datastructs.PriorityQueue.get

`PriorityQueue.`**`get`**`(`*item*`)`

Return the element in the queue identical to *item*.

Parameters **item :**

The element to search for.

Returns The element in the queue identical to *item*. If the element

was not found, None is returned.

mlpy.auxiliary.datastructs.PriorityQueue.pop

`PriorityQueue.`**`pop`**`()`

Get the element with the highest priority.

Get the element with the highest priority (i.e., smallest value).

Returns The element with the highest priority.

mlpy.auxiliary.datastructs.PriorityQueue.push

`PriorityQueue.push(item)`

Push an element on the priority queue.

The element is pushed on the priority queue according to its priority.

Parameters item :

The element to push on the queue.

mlpy.auxiliary.datastructs.PriorityQueue.remove

`PriorityQueue.remove(item)`

Remove an element from the queue.

Parameters item :

The element to remove.

Data sets

`DataSet`	The data set.

mlpy.auxiliary.datasets.DataSet

class `mlpy.auxiliary.datasets.DataSet(`*capacity=None, filename=None, append=None*`)`

Bases: `object`

The data set.

The data set class a container for tracked data. Data can be tracked by adding a `field` for the data of interest. A `numpy.ndarray` is created for every field that is added for recording. Optionally a *description* and a `numpy.dtype` can be associated with the field.

Parameters capacity : int

The initial capacity of the record. Defaults to 10.

filename : str

The name of the file to load from/save to the record.

append : bool

Whether to append to the existing records loaded from file or to overwrite data. Defaults to `False`.

Examples

Creating a new dataset that stores its records in `my_history.pkl`:

```
>>> history = DataSet(capacity=2, filename="my_history.pkl")
```

Adding a new field:

```
>>> history.add_field("state", 3, dtype=DataSet.DTYPE_FLOAT)
>>> print history
state: dim(2,)
[]
```

Adding a new data record:

```
>>> import numpy as np
>>> history.append("state", np.ones(3))
```

Add a new sequence:

```
>>> history.new_sequence()
```

Save the dataset to file:

```
>>> history.save()
```

Methods

DTYPE_FLOAT	
DTYPE_INT	
DTYPE_OBJECT	
add_field(name, dim[, dtype, description])	Add a field with given the specifications.
append(name, data)	Append a new data record.
get_field(name)	Returns the field with the given name.
get_field_names()	Returns all field names.
has_field(name)	Checks if a field with that name exists.
load([filename])	Load the records from file.
new_sequence()	Adds a new sequence.
save([filename])	Save the record to file.

mlpy.auxiliary.datasets.DataSet.add_field

DataSet.**add_field**(*name*, *dim*, *dtype=None*, *description=None*)

Add a field with given the specifications.

Parameters **name** : str

The name of the field.

dim : int

The dimensions of the field

dtype : dtype

The `numpy.dtype` for the underlying `numpy.ndarray`.

description : str

An optional description of the field.

mlpy.auxiliary.datasets.DataSet.append

`DataSet.`**`append`**`(`*name*, *data*`)`

Append a new data record.

Append a new data record to the current sequence of samples of the field with the given *name*.

Parameters **name** : str

The name of the field.

data : str or int or float or ndarray

The data record.

mlpy.auxiliary.datasets.DataSet.get_field

`DataSet.`**`get_field`**`(`*name*`)`

Returns the field with the given name.

Parameters **name** : str

The name of the field.

Returns ndarray :

If a field with that name exists, the field data is returned.

mlpy.auxiliary.datasets.DataSet.get_field_names

`DataSet.`**`get_field_names`**`()`

Returns all field names.

Returns tuple[str] :

A list of field names.

mlpy.auxiliary.datasets.DataSet.has_field

`DataSet.`**`has_field`**`(`*name*`)`

Checks if a field with that name exists.

Parameters **name** : str

The name of the field.

Returns bool :

Whether a field with that name exists.

mlpy.auxiliary.datasets.DataSet.load

`DataSet.`**`load`**`(`*filename=None*`)`

Load the records from file.

If filename is `None`, the record is loaded from the class variable filename.

Parameters **filename** : str

The name of the file.

Raises ValueError

If no filename is passed to the function and the member variable filename is *None*

IOError

If a file with the name does not exist.

mlpy.auxiliary.datasets.DataSet.new_sequence

`DataSet.`**`new_sequence`**`()`

Adds a new sequence.

Adds a new sequence of samples for all fields and increments the sequence counter.

mlpy.auxiliary.datasets.DataSet.save

`DataSet.`**`save`**`(`*filename=None*`)`

Save the record to file.

If filename is *None*, the record is saved to the class variable filename.

Parameters filename : str

The name of the file

Raises ValueError

If no filename is passed to the function and the member variable filename is *None*.

Notes

If an error occurred during saving, the function fails silently.

Miscellaneous

`remove_key`	Safely remove the *key* from the dictionary.
`listify`	Ensure that the object *obj* is of type list.
`stdout_redirected`	Preventing a C shared library to print on stdout.

mlpy.auxiliary.misc.remove_key

`mlpy.auxiliary.misc.`**`remove_key`**`(`*d*, *key*`)`

Safely remove the *key* from the dictionary.

Safely remove the *key* from the dictionary *d* by first making a copy of dictionary. Return the new dictionary together with the value stored for the *key*.

Parameters d : dict

The dictionary from which to remove the *key*.

key :

The key to remove

Returns v :

The value for the key

r : dict

The dictionary with the key removed.

mlpy.auxiliary.misc.listify

`mlpy.auxiliary.misc.`**`listify`**`(`*obj*`)`

Ensure that the object *obj* is of type list.

If the object is not of type *list*, the object is converted into a list.

Parameters obj :

The object.

Returns list :

The object inside a list.

mlpy.auxiliary.misc.stdout_redirected

`mlpy.auxiliary.misc.`**`stdout_redirected`**`(`**args*, ***kwds*`)`

Preventing a C shared library to print on stdout.

Examples

```
>>> import os
>>>
>>> with stdout_redirected(to="filename"):
>>>     print("from Python")
>>>     os.system("echo non-Python applications are also supported")
```

Note:

Project: Code from StackOverflow.
Code author: J.F. Sebastian
License: CC-Wiki

Plotting

Arrow3D	Create a new *Mock* object.

mlpy.auxiliary.plotting.Arrow3D

`mlpy.auxiliary.plotting.`**`Arrow3D`** **= <Mock spec='str' id='140284520602512'>**

Create a new *Mock* object. *Mock* takes several optional arguments that specify the behaviour of the Mock object:

- *spec*: This can be either a list of strings or an existing object (a class or instance) that acts as the specification for the mock object. If you pass in an object then a list of strings is formed by calling dir on the object (excluding unsupported magic attributes and methods). Accessing any attribute not in this list will raise an *AttributeError*.

  If *spec* is an object (rather than a list of strings) then *mock.__class__* returns the class of the spec object. This allows mocks to pass *isinstance* tests.

- *spec_set*: A stricter variant of *spec*. If used, attempting to *set* or get an attribute on the mock that isn't on the object passed as *spec_set* will raise an *AttributeError*.

- *side_effect*: A function to be called whenever the Mock is called. See the *side_effect* attribute. Useful for raising exceptions or dynamically changing return values. The function is called with the same arguments as the mock, and unless it returns *DEFAULT*, the return value of this function is used as the return value.

  Alternatively *side_effect* can be an exception class or instance. In this case the exception will be raised when the mock is called.

  If *side_effect* is an iterable then each call to the mock will return the next value from the iterable. If any of the members of the iterable are exceptions they will be raised instead of returned.

- *return_value*: The value returned when the mock is called. By default this is a new Mock (created on first access). See the *return_value* attribute.

- *wraps*: Item for the mock object to wrap. If *wraps* is not None then calling the Mock will pass the call through to the wrapped object (returning the real result). Attribute access on the mock will return a Mock object that wraps the corresponding attribute of the wrapped object (so attempting to access an attribute that doesn't exist will raise an *AttributeError*).

  If the mock has an explicit *return_value* set then calls are not passed to the wrapped object and the *return_value* is returned instead.

- *name*: If the mock has a name then it will be used in the repr of the mock. This can be useful for debugging. The name is propagated to child mocks.

Mocks can also be called with arbitrary keyword arguments. These will be used to set attributes on the mock after it is created.

CHAPTER 10

Clustering package (`mlpy.cluster`)

K-means clustering

kmeans	Hard cluster data using kmeans.

mlpy.cluster.vq.kmeans

`mlpy.cluster.vq.`**`kmeans`**(*x*, *k*, *n_iter=None*, *thresh=None*, *mean=None*, *fn_plot=None*, *return_assignment=False*, *return_err_hist=False*, *verbose=False*)

Hard cluster data using kmeans.

Parameters **x** : array_like, shape (*n*, *dim*)

List of dim-dimensional data points. Each row corresponds to a single data point.

k : int

The number of clusters to fit.

n_iter : int, optional

Number of iterations to perform. Default is 100.

thresh : float, optional

Convergence threshold. Default is 1e-3.

mean : array_like, shape (ncomponents,), optional

Initial guess for the cluster centers.

fn_plot : callable, optional

A plotting callback function.

return_assignment : bool, optional

Whether to return the assignments or not. Default is False.

return_err_hist : bool, optional

Whether to return the error history. Default is False.

verbose : bool, optional

Controls if debug information is printed to the console. Default is False.

Returns ndarray or tuple :

The cluster centers and optionally the assignments and error history.

Examples

```
>>> from mlpy.cluster.vq import kmeans
```

Note: Ported from Matlab:

Project: Probabilistic Modeling Toolkit for Matlab/Octave.
Copyright (2010) Kevin Murphy and Matt Dunham
License: MIT

CHAPTER 11

Constants (`mlpy.constants`)

Mathematical constants and units used in artificial intelligence.

Mathematical constants

epsilon	10^{-5}
infty	10^{308}

Units

SI prefixes

`micro`	10^{-6}

CHAPTER 12

Environments (`mlpy.environments`)

`Environment`	The environment base class.

mlpy.environments.Environment

class `mlpy.environments.`**`Environment`**`(`*agents=None*`)`

Bases: *`mlpy.modules.Module`*

The environment base class.

The environment specifies the setting in which the agent(s) act. The class is responsible to update the agent(s) at each time step of the program loop and keeps track if the agents' task is complete.

Parameters **agents** : Agent or list[Agent], optional

A list of agents that act in the environment.

Attributes

`mid`	The module's unique identifier.

mlpy.environments.Environment.mid

`Environment.`**`mid`**

The module's unique identifier.

Returns str :

The module's unique identifier

Methods

add_agents(agent)	Add an agent to the environment.
enter(t)	Enter the environment and all agents.
exit()	Exit the environment and all agents.
get_agent(mid)	Return the agent specified by the id.
is_complete()	Checks if the environment has completed.
load(filename)	Load the state of the module from file.
reset(t, **kwargs)	Resets the environment.
save(filename)	Save the current state of the module to file.
update(dt)	Update the environment and all agents.

mlpy.environments.Environment.add_agents

`Environment.`**`add_agents`**`(`*agent*`)`

Add an agent to the environment.

Parameters **agents** : Agent or list[Agent], optional

A list of agents added to the environment.

mlpy.environments.Environment.enter

`Environment.`**`enter`**`(`*t*`)`

Enter the environment and all agents.

Parameters **t** : float

The current time (sec).

mlpy.environments.Environment.exit

`Environment.`**`exit`**`()`

Exit the environment and all agents.

Perform cleanup tasks here.

mlpy.environments.Environment.get_agent

`Environment.`**`get_agent`**`(`*mid*`)`

Return the agent specified by the id.

Returns Agent :

The agent identified by the id.

mlpy.environments.Environment.is_complete

`Environment.`**`is_complete`**`()`

Checks if the environment has completed.

This is dependent on whether the agent(s) have completed their task.

Returns bool :

Whether the environment has reached some end goal.

mlpy.environments.Environment.load

`Environment.load(filename)`

Load the state of the module from file.

Parameters **filename** : str

The name of the file to load from.

Notes

This is a class method, it can be accessed without instantiation.

mlpy.environments.Environment.reset

`Environment.reset(t, **kwargs)`

Resets the environment.

The environment and all agents are reset.

Parameters **t** : float

The current time (sec)

kwargs : dict, optional

Non-positional parameters, optional.

mlpy.environments.Environment.save

`Environment.save(filename)`

Save the current state of the module to file.

Parameters **filename** : str

The name of the file to save to.

mlpy.environments.Environment.update

`Environment.update(dt)`

Update the environment and all agents.

Parameters **dt** : float

The elapsed time (sec)

Gridworld

Cell	The abstract cell module.
GridWorld	A gridworld consisting of a 2d-grid.

mlpy.environments.gridworld.Cell

class `mlpy.environments.gridworld.`**`Cell`**(*x*, *y*, *func*)

Bases: `object`

The abstract cell module.

A cell is a base unit in a 2d-grid. The *GridWorld* is composed of cells.

Parameters **x** : int

The x-position of the cell.

y : int

The y-position of the cell.

func : callable

A callback function to find the neighboring cells.

Notes

Every class inheriting from Cell must implement *is_occupied*.

Attributes

neighbors	The cell's neighbors.
x	The x-position of the cell.
y	The y-position of the cell.

mlpy.environments.gridworld.Cell.neighbors

`Cell.`**`neighbors`**

The cell's neighbors.

Returns list[Point2D] :

A list of neighbors.

mlpy.environments.gridworld.Cell.x

`Cell.`**`x`**

The x-position of the cell.

Returns int :

The x-position.

mlpy.environments.gridworld.Cell.y

`Cell.y`

The y-position of the cell.

Returns int :

The y-position.

Methods

is_occupied()	Determines if the cell is occupied.

mlpy.environments.gridworld.Cell.is_occupied

`Cell.is_occupied()`

Determines if the cell is occupied.

Returns bool :

Whether the cell is occupied.

Raises NotImplementedError

If the child class does not implement this function.

mlpy.environments.gridworld.GridWorld

class `mlpy.environments.gridworld.GridWorld`(*width=20*, *height=20*, *agents=None*, *filename=None*)

Bases: *mlpy.environments.Environment*

A gridworld consisting of a 2d-grid.

A gridworld's basic unit is a cell. Each cell has four neighbors corresponding to the actions the agent can take in the four compass directions (N, S, E, W).

Parameters **width** : int

The number of cells in the x-direction.

height : int

The number of cells in the y-direction.

agents : Agent or list[Agent]

A list of agents that act in the gridworld.

filename : str

The name of the file containing the configuration of the gridworld.

Notes

Within the gridworld, the agent's location is denoted by *o*.

Attributes

height	The number of cells in the y-direction.
mid	The module's unique identifier.
width	The number of cells in the x-direction.

mlpy.environments.gridworld.GridWorld.height

`GridWorld.`**`height`**

The number of cells in the y-direction.

Returns int :

The height.

mlpy.environments.gridworld.GridWorld.mid

`GridWorld.`**`mid`**

The module's unique identifier.

Returns str :

The module's unique identifier

mlpy.environments.gridworld.GridWorld.width

`GridWorld.`**`width`**

The number of cells in the x-direction.

Returns int :

The width.

Methods

add_agents(agent)	Add an agent to the environment.
enter(t)	Enter the environment and all agents.
exit()	Exit the environment and all agents.
find_cells(data)	Find the cells containing given data.
get_agent(mid)	Return the agent specified by the id.
get_cell(loc)	Return the cell based on its x/y-coordinates.
is_complete()	Checks if the environment has completed.
load(f)	Loads the world from file.
make_cell(x, y)	Create the new cell.
move_coords(cell, move)	Returns the coordinates of the neighboring cell the agent transitions to following the given move.
random_location()	Find a random unoccupied location within the grid.
reset(t, **kwargs)	Reset the agent's state.
save([f])	Save the world to file.
set_start_loc(loc)	Set the agent's starting location.
update(dt)	Update the agents.

mlpy.environments.gridworld.GridWorld.add_agents

`GridWorld.add_agents(agent)`

Add an agent to the environment.

Parameters **agents** : Agent or list[Agent], optional

A list of agents added to the environment.

mlpy.environments.gridworld.GridWorld.enter

`GridWorld.enter(t)`

Enter the environment and all agents.

Parameters **t** : float

The current time (sec).

mlpy.environments.gridworld.GridWorld.exit

`GridWorld.exit()`

Exit the environment and all agents.

Perform cleanup tasks here.

mlpy.environments.gridworld.GridWorld.find_cells

`GridWorld.find_cells(data)`

Find the cells containing given data.

Parameters **data** : str

The data to match the cell to.

Returns list[Cell] :

All cells that contain the specified data.

mlpy.environments.gridworld.GridWorld.get_agent

`GridWorld.get_agent(mid)`

Return the agent specified by the id.

Returns Agent :

The agent identified by the id.

mlpy.environments.gridworld.GridWorld.get_cell

`GridWorld.get_cell(loc)`

Return the cell based on its x/y-coordinates.

Parameters **loc** : Point2D

The x-/y-coordinates of the cell.

Returns Cell :

The cell at the specified location.

mlpy.environments.gridworld.GridWorld.is_complete

`GridWorld.`**`is_complete`**`()`

Checks if the environment has completed.

This is dependent on whether the agent(s) have completed their task.

Returns bool :

Whether the environment has reached some end goal.

mlpy.environments.gridworld.GridWorld.load

`GridWorld.`**`load`**`(`*f*`)`

Loads the world from file.

If a **.cfg* with the same name exists, the configurations are being loaded as well.

Parameters **f** : str or file

The file instance or the filename.

mlpy.environments.gridworld.GridWorld.make_cell

`GridWorld.`**`make_cell`**`(`*x*, *y*`)`

Create the new cell.

x [int] The x-coordinate within the gridworld.

y [int] The y-coordinate within the gridworld.

Returns Cell :

The created cell.

mlpy.environments.gridworld.GridWorld.move_coords

`GridWorld.`**`move_coords`**`(`*cell*, *move*`)`

Returns the coordinates of the neighboring cell the agent transitions to following the given move.

Parameters **cell** : Cell

The current cell.

move : int

The action performed.

Returns Point2D :

The x-/y-coordinates of the resulting cell.

mlpy.environments.gridworld.GridWorld.random_location

`GridWorld.`**`random_location`**`()`

Find a random unoccupied location within the grid.

Returns Point2D :

The random x-/y-coordinates.

mlpy.environments.gridworld.GridWorld.reset

`GridWorld.`**`reset`**`(`*t*, ***kwargs*`)`

Reset the agent's state.

Parameters **t** : float

The current time (sec).

kwargs : dict, optional

Non-positional parameters.

mlpy.environments.gridworld.GridWorld.save

`GridWorld.`**`save`**`(`*f=None*`)`

Save the world to file.

Parameters **f** : str or file

The file instance or the filename.

mlpy.environments.gridworld.GridWorld.set_start_loc

`GridWorld.`**`set_start_loc`**`(`*loc*`)`

Set the agent's starting location.

Parameters **loc** : Point2D

The x-/y-coordinates the agent starts out in.

mlpy.environments.gridworld.GridWorld.update

`GridWorld.`**`update`**`(`*dt*`)`

Update the agents.

Parameters **dt** : float

The elapsed time (sec)

Nao

NaoEnvFactory	The NAO environment factory.
PhysicalWorld	The physical (real) environment.
Webots	Simulated environment using the Webots simulator.

mlpy.environments.nao.NaoEnvFactory

class `mlpy.environments.nao.`**`NaoEnvFactory`**

Bases: `object`

The NAO environment factory.

An instance of a NAO environment can be created by passing the environment type.

Notes

Currently only soccer related events are handled by the Nao worlds. To be more specific the environment attempts to detect whether a goal was scored. The physical world will prompt the user to reset the experiment while the supervisor in the Webots world is responsible for resetting the environment for the next experiment.

Examples

```
>>> from mlpy.environments.nao import NaoEnvFactory
>>> NaoEnvFactory.create('nao.physicalworld')
```

This creates a *`PhysicalWorld`* instance controlling agents in the real world.

```
>>> NaoEnvFactory.create('nao.webots', 12345)
```

This creates a *`Webots`* instance controlling simulated agents in Webots. The port '12345' is the port the controller of the supervisor in the Webots world listens to.

Methods

`create`(_type, *args, **kwargs)	Create a Nao environment of the given type.

mlpy.environments.nao.NaoEnvFactory.create

static `NaoEnvFactory.`**`create`**(*_type*, **args*, ***kwargs*)

Create a Nao environment of the given type.

Parameters **_type** : str

The Nao environment type. Valid environment types:

nao.physicalworld This controls the robots in the real world. The environment interacts with the user to inquire about events happening in the real world. A *`PhysicalWorld`* instance is created.

nao.webots This controls the simulated robots in the Webots simulator. The world in the simulator should include a supervisor using a controller to handle the required events. A sample controller can be found in *environments/webots/controllers/serverc*. A *`Webots`* instance is created.

args : tuple, optional

Positional arguments to pass to the class of the given type for initialization.

kwargs : dict, optional

Non-positional arguments to pass to the class of the given type for initialization.

Returns Environment

A Nao environment instance of the given type.

mlpy.environments.nao.PhysicalWorld

class `mlpy.environments.nao.`**`PhysicalWorld`**(*agents=None*)

Bases: *`mlpy.environments.Environment`*

The physical (real) environment.

Parameters **agents** : Agent or list[Agent], optional

A list of agents that act in the environment.

Notes

The agents are acting in the real world. To capture events happening in the real world the user is prompted to provide the information.

Attributes

`mid`	The module's unique identifier.

mlpy.environments.nao.PhysicalWorld.mid

`PhysicalWorld.`**`mid`**

The module's unique identifier.

Returns str :

The module's unique identifier

Methods

`add_agents`(agent)	Add an agent to the environment.
`check_data`(value)	Request to check for data.
`enter`(t)	Enter the environment and all agents.
`exit`()	Exit the environment and all agents.
`get_agent`(mid)	Return the agent specified by the id.
`is_complete`()	Checks if the environment has completed.
`load`(filename)	Load the state of the module from file.
`reset`(t, **kwargs)	Reset the environment and all agents.
`save`(filename)	Save the current state of the module to file.
`update`(dt)	Update the environment and all agents.

mlpy.environments.nao.PhysicalWorld.add_agents

`PhysicalWorld.`**`add_agents`**`(`*agent*`)`

Add an agent to the environment.

Parameters **agents** : Agent or list[Agent], optional

A list of agents added to the environment.

mlpy.environments.nao.PhysicalWorld.check_data

`PhysicalWorld.`**`check_data`**`(`*value*`)`

Request to check for data.

Parameters **value** : str

The request identifier.

Returns The result returned by the user.

Notes

When checking for data, the user is prompted to provide the information via the console.

mlpy.environments.nao.PhysicalWorld.enter

`PhysicalWorld.`**`enter`**`(`*t*`)`

Enter the environment and all agents.

Parameters **t** : float

The current time (sec).

mlpy.environments.nao.PhysicalWorld.exit

`PhysicalWorld.`**`exit`**`()`

Exit the environment and all agents.

Perform cleanup tasks here.

mlpy.environments.nao.PhysicalWorld.get_agent

`PhysicalWorld.`**`get_agent`**`(`*mid*`)`

Return the agent specified by the id.

Returns Agent :

The agent identified by the id.

mlpy.environments.nao.PhysicalWorld.is_complete

`PhysicalWorld.`**`is_complete`**`()`

Checks if the environment has completed.

This is dependent on whether the agent(s) have completed their task.

Returns bool :

Whether the environment has reached some end goal.

mlpy.environments.nao.PhysicalWorld.load

`PhysicalWorld.`**`load`**`(`*filename*`)`

Load the state of the module from file.

Parameters **filename** : str

The name of the file to load from.

Notes

This is a class method, it can be accessed without instantiation.

mlpy.environments.nao.PhysicalWorld.reset

`PhysicalWorld.`**`reset`**`(`*t*, ***kwargs*`)`

Reset the environment and all agents.

The user is prompted to reset the environment (i.e., experiment). Ones the user has reset the environment all agents are reset.

Parameters **t** : float

The current time (sec)

kwargs : dict, optional

Non-positional parameters, optional.

mlpy.environments.nao.PhysicalWorld.save

`PhysicalWorld.`**`save`**`(`*filename*`)`

Save the current state of the module to file.

Parameters **filename** : str

The name of the file to save to.

mlpy.environments.nao.PhysicalWorld.update

`PhysicalWorld.`**`update`**`(`*dt*`)`

Update the environment and all agents.

Parameters **dt** : float

The elapsed time (sec)

mlpy.environments.nao.Webots

class `mlpy.environments.nao.`**`Webots`**(*port=12345*, *agents=None*)

Bases: *`mlpy.environments.Environment`*

Simulated environment using the Webots simulator.

The Webots environment works in conjunction with a controller specified for a supervisor. A sample controller can be found in *webots/controllers/serverc*. This controller listens on port *12345* for the following events:

request reset Requests an environment reset from the controller.

check goal Requests from the controller a check whether a goal was scored or not. The result of that check is send back to the client.

Parameters **port** : int, optional

The port the controller listens to. If using the environment in conjunction with controller *serverc* the port number is *12345*. Default is 12345.

agents : Agent or list[Agent], optional

A list of agents that act in the environment.

Notes

When requested to reset, the environment will request to reset the simulated environment in Webots from the controller. It is also possible to check if a goal was scored by calling the function *`check_data`* with the argument 'check goal'. This sends a request to the controller to check if a goal was scored.

Attention: The Webots environment class requires the NAOqi API from Aldebaran be installed on your machine. A separate license is required for the API.

Attributes

`mid`	The module's unique identifier.

mlpy.environments.nao.Webots.mid

`Webots.`**`mid`**

The module's unique identifier.

Returns str :

The module's unique identifier

Methods

`add_agents`(agent)	Add an agent to the environment.
`check_data`(value)	Request to check for data.
`enter`(t)	Enter the environment and all agents.

Continued on next page

Table 12.14 – continued from previous page

`exit`()	Exit the environment and all agents.
`get_agent`(mid)	Return the agent specified by the id.
`is_complete`()	Checks if the environment has completed.
`load`(filename)	Load the state of the module from file.
`reset`(t, **kwargs)	Reset the environment and all agents.
`save`(filename)	Save the current state of the module to file.
`update`(dt)	Update the environment and all agents.

mlpy.environments.nao.Webots.add_agents

`Webots.`**`add_agents`**`(`*agent*`)`

Add an agent to the environment.

Parameters agents : Agent or list[Agent], optional

A list of agents added to the environment.

mlpy.environments.nao.Webots.check_data

`Webots.`**`check_data`**`(`*value*`)`

Request to check for data.

Parameters value : str

The request identifier.

Returns The result returned by the controller.

Notes

The Webots environment works in conjunction with a controller specified for a supervisor. When checking for data, a request with the *value* is send to the controller which extracts the information and returns the results.

mlpy.environments.nao.Webots.enter

`Webots.`**`enter`**`(`*t*`)`

Enter the environment and all agents.

Parameters t : float

The current time (sec).

mlpy.environments.nao.Webots.exit

`Webots.`**`exit`**`()`

Exit the environment and all agents.

Perform cleanup tasks here.

mlpy.environments.nao.Webots.get_agent

`Webots.`**`get_agent`**`(`*mid*`)`

Return the agent specified by the id.

Returns Agent :

The agent identified by the id.

mlpy.environments.nao.Webots.is_complete

`Webots.`**`is_complete`**`()`

Checks if the environment has completed.

This is dependent on whether the agent(s) have completed their task.

Returns bool

Whether the environment has reached some end goal.

mlpy.environments.nao.Webots.load

`Webots.`**`load`**`(`*filename*`)`

Load the state of the module from file.

Parameters **filename** : str

The name of the file to load from.

Notes

This is a class method, it can be accessed without instantiation.

mlpy.environments.nao.Webots.reset

`Webots.`**`reset`**`(`*t*, ***kwargs*`)`

Reset the environment and all agents.

A request is send to the controller to reset the environment. Once the environment is reset all agents acting in the environment are reset.

Parameters **t** : float

The current time (sec)

kwargs : dict, optional

Non-positional parameters, optional.

mlpy.environments.nao.Webots.save

`Webots.`**`save`**`(`*filename*`)`

Save the current state of the module to file.

Parameters **filename** : str

The name of the file to save to.

mlpy.environments.nao.Webots.update

`Webots.`**`update`**`(`*dt*`)`

Update the environment and all agents.

Parameters **dt** : float

The elapsed time (sec)

CHAPTER 13

Experiment Infrastructure (`mlpy.experiments`)

Experiment	The experiment class.

mlpy.experiments.Experiment

class `mlpy.experiments.`**`Experiment`**(*env*)

Bases: `object`

The experiment class.

An experiment sets up an agent in an environment and runs until the environment is considered to have completed. This can be the case when all agents acting in the environment have reached their goal state.

An experiment can consist of multiple episodes and rests itself at the end of each episode.

Parameters **env** : Environment

The environment in which to run the agent(s).

Methods

enter()	Enter the experiment.
exit()	Exit the experiment.
reset()	Reset the experiment.
run()	Run the experiment.
update()	Update all modules during the program loop.

mlpy.experiments.Experiment.enter

`Experiment.`**`enter`**()

Enter the experiment.

mlpy.experiments.Experiment.exit

`Experiment.exit()`

Exit the experiment.

mlpy.experiments.Experiment.reset

`Experiment.reset()`

Reset the experiment.

mlpy.experiments.Experiment.run

`Experiment.run()`

Run the experiment.

The experiment finishes when the environment is considered to have completed. Possible causes for completing the environment is that all agents have reached a goal state.

mlpy.experiments.Experiment.update

`Experiment.update()`

Update all modules during the program loop.

Tasks

Task	The task description base class.
EpisodicTask	The episodic task description base class.
SearchTask	The abstract class for a search task definition.

mlpy.experiments.task.Task

class `mlpy.experiments.task.Task(`*env=None*`)`

Bases: `object`

The task description base class.

A task description describes the task the agent is to perform. The task description allows to configure *State* and *Action* by setting the number of features, the description and by overwriting the static functions *is_valid*, *encode*, and *decode* at runtime.

Parameters **env** : Environment, optional

The environment in which the agent performs the task.

See also:

EpisodicTask, *SearchTask*

Notes

Any task should inherit from this base class or any class deriving from this class. Every deriving class must overwrite the methods `_configure_state` and `_configure_action` to configure the classes *State* and *Action*, respectively.

For both *State* and *Action* the appropriate class variables can be set by calling the following functions:

- *set_nfeatures*
- *set_dtype*
- *set_description*
- *set_discretized*
- *set_minmax_features*
- *set_states_per_dim*

Overwrite the following *State* and *Action* methods to allow for more readable descriptions:

- *encode*
- *decode*

Additionally, the *State* class provides a method to check a state's validity:

- *is_valid*

Attributes

event_delay	Event delay.
is_episodic	Identifies if the task is episodic or not.

mlpy.experiments.task.Task.event_delay

`Task.`**`event_delay`**

Event delay.

The time in milliseconds (ms) by which the fsm event is delayed once termination is requested.

Returns float :

The time in milliseconds.

mlpy.experiments.task.Task.is_episodic

`Task.`**`is_episodic`**

Identifies if the task is episodic or not.

Returns bool :

Whether this task is episodic or not.

Methods

`get_reward`(state, action)	Retrieve the reward.
`is_complete`()	Check if the task has completed.
`request_termination`(value)	Request termination of the task.
`reset`(t, **kwargs)	Reset the task.
`sensation`(**kwargs)	Gather the state feature information.
`terminate`(value)	Set the termination flag.
`termination_requested`()	Check if termination was requested.

mlpy.experiments.task.Task.get_reward

`Task.`**`get_reward`**(*state*, *action*)

Retrieve the reward.

Retrieve the reward for the given state and action from the environment.

Parameters **state** : State

The current state.

action : Action

The current action.

Returns float :

The reward.

mlpy.experiments.task.Task.is_complete

`Task.`**`is_complete`**()

Check if the task has completed.

Returns bool :

Whether the task has completed or not.

mlpy.experiments.task.Task.request_termination

`Task.`**`request_termination`**(*value*)

Request termination of the task.

Parameters **value** : bool

The value to set the termination requested flag to.

mlpy.experiments.task.Task.reset

`Task.`**`reset`**(*t*, ***kwargs*)

Reset the task.

Parameters **t** : float

The current time (sec).

kwargs : dict, optional

Non-positional parameters.

mlpy.experiments.task.Task.sensation

`Task.`**`sensation`**`(`*`**kwargs`*`)`

Gather the state feature information.

Gather the state information (i.e. features) according to the task from the agent's senses.

Parameters **kwargs: dict**

Non-positional arguments needed for gathering the information.

Returns **features** : array, shape (*nfeatures*,)

The sensed features

Raises **NotImplementedError**

If the child class does not implement this function.

mlpy.experiments.task.Task.terminate

`Task.`**`terminate`**`(`*`value`*`)`

Set the termination flag.

Parameters **value** : bool

The value to set the termination flag to.

mlpy.experiments.task.Task.termination_requested

`Task.`**`termination_requested`**`()`

Check if termination was requested.

Returns bool :

Whether termination was requested or not.

mlpy.experiments.task.EpisodicTask

class `mlpy.experiments.task.`**`EpisodicTask`**`(`*`initial_states`*, *`terminal_states`*, *`env=None`*`)`

Bases: *`mlpy.experiments.task.Task`*

The episodic task description base class.

This class automatically identifies the task as an episodic task. An episodic task has a set of actions that transitions the agent into a terminal state. Once a terminal state is reached the task is complete.

Parameters **initial_states** : str or State or list[str or State]

List of possible initial states.

terminal_states : str or State or list[str or State]

List of terminal states.

env : Environment, optional

The environment in which the agent performs the task.

Notes

Every deriving class must overwrite the methods `_configure_state` and `_configure_action` to configure the classes *`State`* and *`Action`*, respectively.

For both *`State`* and *`Action`* the appropriate class variables can be set by calling the following functions:

- *`set_nfeatures`*
- *`set_dtype`*
- *`set_description`*
- *`set_discretized`*
- *`set_minmax_features`*
- *`set_states_per_dim`*

Overwrite the following *`State`* and *`Action`* methods to allow for more readable descriptions:

- *`encode`*
- *`decode`*

Additionally, the *`State`* class provides a method to check a state's validity. Overwrite this function to specify valid states:

- *`is_valid`*

Attributes

`event_delay`	Event delay.
`is_episodic`	Identifies if the task is episodic or not.

mlpy.experiments.task.EpisodicTask.event_delay

`EpisodicTask.`**`event_delay`**

Event delay.

The time in milliseconds (ms) by which the fsm event is delayed once termination is requested.

Returns float :

The time in milliseconds.

mlpy.experiments.task.EpisodicTask.is_episodic

`EpisodicTask.`**`is_episodic`**

Identifies if the task is episodic or not.

Returns bool :

Whether this task is episodic or not.

Methods

get_reward(state, action)	Retrieve the reward.
is_complete()	Check if the task has completed.
random_initial_state()	Return a random initial state.
request_termination(value)	Request termination of the task.
reset(t, **kwargs)	Reset the task.
sensation(**kwargs)	Gather the state feature information.
terminate(value)	Set the termination flag.
termination_requested()	Check if termination was requested.

mlpy.experiments.task.EpisodicTask.get_reward

EpisodicTask.**get_reward**(*state*, *action*)

Retrieve the reward.

Retrieve the reward for the given state and action from the environment.

Parameters **state** : State

The current state.

action : Action

The current action.

Returns float :

The reward.

mlpy.experiments.task.EpisodicTask.is_complete

EpisodicTask.**is_complete**()

Check if the task has completed.

Returns bool :

Whether the task has completed or not.

mlpy.experiments.task.EpisodicTask.random_initial_state

static EpisodicTask.**random_initial_state**()

Return a random initial state.

Returns str or State :

A random initial state.

mlpy.experiments.task.EpisodicTask.request_termination

EpisodicTask.**request_termination**(*value*)

Request termination of the task.

Parameters **value** : bool

The value to set the termination requested flag to.

mlpy.experiments.task.EpisodicTask.reset

`EpisodicTask.`**`reset`**`(`*t*, ***kwargs*`)`

Reset the task.

Parameters **t** : float

The current time (sec).

kwargs : dict, optional

Non-positional parameters.

mlpy.experiments.task.EpisodicTask.sensation

`EpisodicTask.`**`sensation`**`(`***kwargs*`)`

Gather the state feature information.

Gather the state information (i.e. features) according to the task from the agent's senses.

Parameters **kwargs: dict**

Non-positional arguments needed for gathering the information.

Returns **features** : array, shape (*nfeatures*,)

The sensed features

Raises **NotImplementedError**

If the child class does not implement this function.

mlpy.experiments.task.EpisodicTask.terminate

`EpisodicTask.`**`terminate`**`(`*value*`)`

Set the termination flag.

Parameters **value** : bool

The value to set the termination flag to.

mlpy.experiments.task.EpisodicTask.termination_requested

`EpisodicTask.`**`termination_requested`**`()`

Check if termination was requested.

Returns bool :

Whether termination was requested or not.

mlpy.experiments.task.SearchTask

class `mlpy.experiments.task.`**`SearchTask`**`(`*initial_states*, *terminal_states=None*, *env=None*`)`

Bases: *`mlpy.experiments.task.EpisodicTask`*

The abstract class for a search task definition.

Parameters **initial_states** : str or State or list[str or State]

List of possible initial states.

terminal_states : str or State or list[str or State]

List of terminal states.

env : Environment, optional

The environment in which the agent performs the task.

Attributes

event_delay	Event delay.
is_episodic	Identifies if the task is episodic or not.

mlpy.experiments.task.SearchTask.event_delay

SearchTask.**event_delay**

Event delay.

The time in milliseconds (ms) by which the fsm event is delayed once termination is requested.

Returns float :

The time in milliseconds.

mlpy.experiments.task.SearchTask.is_episodic

SearchTask.**is_episodic**

Identifies if the task is episodic or not.

Returns bool :

Whether this task is episodic or not.

Methods

get_path_cost(c, _)	Returns the cost for the current path.
get_reward(state, action)	Retrieve the reward.
get_successor(state)	Find valid successors.
is_complete()	Check if the task has completed.
random_initial_state()	Return a random initial state.
request_termination(value)	Request termination of the task.
reset(t, **kwargs)	Reset the task.
sensation(**kwargs)	Gather the state feature information.
terminate(value)	Set the termination flag.
termination_requested()	Check if termination was requested.

mlpy.experiments.task.SearchTask.get_path_cost

static SearchTask.**get_path_cost**(*c*, *_*)

Returns the cost for the current path.

Parameters **c** : float

The current cost for the path.

Returns float :

The updated cost.

mlpy.experiments.task.SearchTask.get_reward

`SearchTask.get_reward(state, action)`

Retrieve the reward.

Retrieve the reward for the given state and action from the environment.

Parameters **state** : State

The current state.

action : Action

The current action.

Returns float :

The reward.

mlpy.experiments.task.SearchTask.get_successor

`SearchTask.get_successor(state)`

Find valid successors.

Finds all valid successors (state-action pairs) for the given `state`.

Parameters **state** : int or tuple[int]

The state from which to find successors.

Returns list[tuple(str, str or tuple[str])] :

A list of all successor.

Raises **NotImplementedError**

If the child class does not implement this function.

mlpy.experiments.task.SearchTask.is_complete

`SearchTask.is_complete()`

Check if the task has completed.

Returns bool :

Whether the task has completed or not.

mlpy.experiments.task.SearchTask.random_initial_state

`SearchTask.random_initial_state()`

Return a random initial state.

Returns str or State :

A random initial state.

mlpy.experiments.task.SearchTask.request_termination

`SearchTask.`**`request_termination`**`(`*value*`)`

Request termination of the task.

Parameters **value** : bool

The value to set the termination requested flag to.

mlpy.experiments.task.SearchTask.reset

`SearchTask.`**`reset`**`(`*t*, ***kwargs*`)`

Reset the task.

Parameters **t** : float

The current time (sec).

kwargs : dict, optional

Non-positional parameters.

mlpy.experiments.task.SearchTask.sensation

`SearchTask.`**`sensation`**`(`***kwargs*`)`

Gather the state feature information.

Gather the state information (i.e. features) according to the task from the agent's senses.

Parameters **kwargs: dict**

Non-positional arguments needed for gathering the information.

Returns **features** : array, shape (*nfeatures*,)

The sensed features

Raises **NotImplementedError**

If the child class does not implement this function.

mlpy.experiments.task.SearchTask.terminate

`SearchTask.`**`terminate`**`(`*value*`)`

Set the termination flag.

Parameters **value** : bool

The value to set the termination flag to.

mlpy.experiments.task.SearchTask.termination_requested

`SearchTask.`**`termination_requested`**`()`

Check if termination was requested.

Returns bool :

Whether termination was requested or not.

CHAPTER 14

Knowledge representations (`mlpy.knowledgerep`)

Case base reasoning (`mlpy.knowledgerep.cbr`)

Engine

`CaseMatch`	Case match information.
`Case`	The representation of a case in the case base.
`CaseBaseEntry`	The case base entry class.
`CaseBase`	The case base engine.

mlpy.knowledgerep.cbr.engine.CaseMatch

class `mlpy.knowledgerep.cbr.engine.`**`CaseMatch`**(*case*, *key*, *similarity=None*)

Bases: `object`

Case match information.

Parameters **case** : Case

The matching case.

similarity :

A measure for the similarity to the query case.

Attributes

is_solution	(bool) Whether this case match is a solution to the query case or not.
error	(float) The error of the prediction.
predicted	(bool) Whether the query case could be correctly predicted using this case match.

Methods

get_similarity(key)	Retrieve the similarity measure for the feature identified by the key.
set_similarity(key, value)	Set the similarity measure for the feature identified by the key.

mlpy.knowledgerep.cbr.engine.CaseMatch.get_similarity

`CaseMatch.`**`get_similarity`**`(`*key*`)`

Retrieve the similarity measure for the feature identified by the key.

Returns float :

The similarity measure.

mlpy.knowledgerep.cbr.engine.CaseMatch.set_similarity

`CaseMatch.`**`set_similarity`**`(`*key*, *value*`)`

Set the similarity measure for the feature identified by the key.

mlpy.knowledgerep.cbr.engine.Case

class `mlpy.knowledgerep.cbr.engine.`**`Case`**`(`*cid*, *name=None*, *description=None*, *features=None*`)`

Bases: `object`

The representation of a case in the case base.

A case is composed of one or more `Feature`.

Parameters **cid** : int

The case's unique identifier.

name : str

The name for the case.

description : str

Text describing the case, optional.

features : dict

A list of features describing the case.

Attributes

id	The case's unique identifier.

mlpy.knowledgerep.cbr.engine.Case.id

`Case.`**`id`**

The case's unique identifier.

Return type int

Methods

`add_feature`(name, _type, value, **kwargs)	Add a new feature.
`compute_similarity`(other)	Computes how similar two cases are.
`get_features`(names)	Return sorted collection of features with the specified name.
`get_indexed`()	Return sorted collection of all indexed features.
`get_retrieval_method`(names)	Returns the retrieval method for the given features.
`get_retrieval_params`(names)	Return the retrieval parameters for the given features.
`next`()	

mlpy.knowledgerep.cbr.engine.Case.add_feature

`Case.`**`add_feature`**(*name*, *_type*, *value*, ***kwargs*)

Add a new feature.

Parameters **name** : str

The name of the feature (this also serves as the features identifying key).

_type : str

The type of the feature. Valid feature types are:

bool The feature values are boolean types (`BoolFeature`).

string The feature values are of types sting (`StringFeature`).

int The feature values are of type integer (`IntFeature`).

float The feature values are of type float (`FloatFeature`).

value : bool or string or int or float or list

The feature value.

Other Parameters **weight** : float or list[float]

The weights given to each feature value.

is_index : bool

Flag indicating whether this feature is an index.

retrieval_method : str

The similarity model used for retrieval. Refer to `Feature.retrieval_method` for valid methods.

retrieval_method_params : dict

Parameters relevant to the selected retrieval method.

retrieval_algorithm : str

The internal indexing structure of the training data. Refer to `Feature.retrieval_method` for valid algorithms.

retrieval_metric : str

The metric used to compute the distances between pairs of points. Refer to `sklearn.neighbors.DistanceMetric` for valid identifiers.

retrieval_metric_params : dict

Parameters relevant to the specified metric.

mlpy.knowledgerep.cbr.engine.Case.compute_similarity

`Case.`**`compute_similarity`**`(`*other*`)`

Computes how similar two cases are.

Parameters **other** : Case

The other case this case is compared to.

Returns float :

The similarity measure between the two cases.

mlpy.knowledgerep.cbr.engine.Case.get_features

`Case.`**`get_features`**`(`*names*`)`

Return sorted collection of features with the specified name.

Parameters **names** : str or list[str]

The name(s) of the features to retrieve.

Returns list or int or str or bool or float :

List of features with the specified names(s)

mlpy.knowledgerep.cbr.engine.Case.get_indexed

`Case.`**`get_indexed`**`()`

Return sorted collection of all indexed features.

Returns list :

The names of the indexed features in ascending order.

mlpy.knowledgerep.cbr.engine.Case.get_retrieval_method

`Case.`**`get_retrieval_method`**`(`*names*`)`

Returns the retrieval method for the given features.

Parameters **names** : str or list[str]

The name(s) of the feature for which to retrieve the retrieval method.

Returns str :

The retrieval method for all feature. Features grouped together for retrieval must use the same retrieval method.

Raises **UserWarning**

If not all features use the same retrieval method.

mlpy.knowledgerep.cbr.engine.Case.get_retrieval_params

`Case.`**`get_retrieval_params`**`(`*names*`)`

Return the retrieval parameters for the given features.

Parameters **names** : str or list[str]

The name(s) of the feature for which to retrieve the retrieval parameters.

Returns dict :

The retrieval parameters for the feature(s). Features grouped together for retrieval must use the same retrieval parameters.

Raises **UserWarning**

If not all features use the same retrieval parameters.

mlpy.knowledgerep.cbr.engine.Case.next

`Case.`**`next`**`()`

mlpy.knowledgerep.cbr.engine.CaseBaseEntry

class `mlpy.knowledgerep.cbr.engine.`**`CaseBaseEntry`**`(`*model*, *validity_check=True*`)`

Bases: `object`

The case base entry class.

The entry maintains a similarity model from which similar cases can be derived.

Internally the similarity model maintains an indexing structure dependent on the similarity model type for efficient computation of the similarity between cases. The case base is responsible for updating the indexing structure as cases are added and removed.

Parameters **model** : ISimilarity

The similarity model.

validity_check : bool

This flag controls whether the dirty flag is being checked before determining whether to rebuild the model or not.

Attributes

dirty	(bool) A flag which identifies whether the model needs to be rebuild. The indexing structure of the similarity model is always rebuild unless a validity check is required. If a validity check is required the indexing structure is only rebuild if the entry is considered dirty.

Methods

`compute_similarity`(data_point, **kwargs)	Computes the similarity.

mlpy.knowledgerep.cbr.engine.CaseBaseEntry.compute_similarity

`CaseBaseEntry.`**`compute_similarity`**`(`*data_point*, ***kwargs*`)`

Computes the similarity.

Computes the similarity between the data point and each entry in the similarity model's indexing structure.

Parameters **data_point** : list[float]

The data point to each entry in the similarity model is compared to.

Returns list[Stat] :

The similarity statistics of all entries in the model's indexing structure.

Other Parameters **cases** : dict[int, Case]

The complete case base from which to build the indexing structure used by the similarity model.

data : ndarray[ndarray[float]]

If this keyword is set, the cases in the case base are ignored and the data entries specified in this variable are used to build the indexing structure.

names : str or list[str]

The feature name(s) relevant for the similarity computation. This field is only required if the cases for building the indexing structure comes from the *cases* field.

id_map : dict[int, int]

The mapping from the data stored in the *data* field to their case ids. This field is only required if the data for building the indexing structure comes from the *data* field.

mlpy.knowledgerep.cbr.engine.CaseBase

class `mlpy.knowledgerep.cbr.engine.`**`CaseBase`**`(`*case_template*, *reuse_method=None*, *reuse_method_params=None*, *revision_method=None*, *revision_method_params=None*, *retention_method=None*, *retention_method_params=None*, *plot_retrieval=None*, *plot_retrieval_names=None*`)`

Bases: `object`

The case base engine.

The case base engine maintains the a database of all cases entered into the case base. It manages retrieval, revision, reuse, and retention of cases.

Parameters **case_template: dict**

The template from which to create a new case.

Example An example template for a feature named `state` with the specified feature parameters. `data` is the data from which to extract the case from. In this example it is expected that `data` has a member variable `state`.

```
{
    "state": {
        "type": "float",
        "value": "data.state",
        "is_index": True,
        "retrieval_method": "radius-n",
        "retrieval_method_params": 0.01
    },
    "delta_state": {
        "type": "float",
        "value": "data.next_state - data.state",
        "is_index": False,
    }
}
```

reuse_method : str

The reuse method name to be used during the reuse step. Default is *defaultreusemethod*.

reuse_method_params : dict

Non-positional initialization parameters for the reuse method instantiation.

revision_method : str

The revision method name to be used during the revision step. Default is *defaultrevisionmethod*.

revision_method_params : dict

Non-positional initialization parameters for the revision method instantiation.

retention_method : str

The retention method name to be used during the retention step. Default is *defaultretentionmethod*.

retention_method_params : dict

Non-positional initialization parameters for the retention method instantiation.

plot_retrieval : bool

Whether to plot the result or not. Default is False.

plot_retrieval_names : str or list[str]

The names of the feature which to plot.

Examples

Create a case base:

```
>>> from mlpy.auxiliary.io import load_from_file
>>>
>>> template = {}
>>> cb = CaseBase(template)
```

Fill case base with data read from file:

```
>>> from mlpy.mdp.stateaction import Experience, State, Action
>>>
>>> data = load_from_file("data/jointsAndActionsData.pkl")
>>> for i in xrange(len(data.itervalues().next())):
...     for j in xrange(len(data.itervalues().next()[0][i]) - 1):
...         if not j == 10:  # exclude one experience as test case
...             experience = Experience(State(data["states"][i][:, j]),
...                                     Action(data["actions"][i][:, j]),
...                                     State(data["states"][i][:, j + 1]))
...             cb.run(cb.case_from_data(experience))
```

Loop over all cases in the case base:

```
>>> for i in len(cb):
...     pass
```

Retrieve case with `id=0`:

```
>>> case = cb[0]
```

Attributes

`counter`	The case counter.

mlpy.knowledgerep.cbr.engine.CaseBase.counter

`CaseBase.`**`counter`**

The case counter.

The counter is increased with every case added to the case base.

Returns int :

The current count.

Methods

`add`(case)	Add a new case without any checks.
`case_from_data`(data)	Convert data into a case using the case template.
`get_new_id`()	Return an unused case id.
`load`(filename)	
`next`()	
`plot_retention`(case, case_matches)	Plot the retention result.
`plot_retrieval`(case, case_id_list[, names])	Plot the retrieved data.
`plot_reuse`(case, case_matches, revised_matches)	Plot the reuse result.
`plot_revision`(case, case_matches)	Plot revision results.
`retain`(case, case_matches)	Retain new case.
`retrieve`(case[, names, validity_check])	Retrieve cases similar to the query case.
`reuse`(case, case_matches)	Performs the reuse step

Continued on next page

Table 14.7 – continued from previous page

revision(case, case_matches)	Evaluate solution provided by problem-solving experience.
run(case)	Run the case base.
save(filename)	

mlpy.knowledgerep.cbr.engine.CaseBase.add

`CaseBase.`**`add`**`(`*case*`)`

Add a new case without any checks.

Parameters **case** : Case

The case to add to the case base.

mlpy.knowledgerep.cbr.engine.CaseBase.case_from_data

`CaseBase.`**`case_from_data`**`(`*data*`)`

Convert data into a case using the case template.

Parameters **data :**

The data from which to extract the case.

Returns Case :

The case extracted from the data.

mlpy.knowledgerep.cbr.engine.CaseBase.get_new_id

`CaseBase.`**`get_new_id`**`()`

Return an unused case id.

Returns int :

Unused case ID.

mlpy.knowledgerep.cbr.engine.CaseBase.load

`CaseBase.`**`load`**`(`*filename*`)`

mlpy.knowledgerep.cbr.engine.CaseBase.next

`CaseBase.`**`next`**`()`

mlpy.knowledgerep.cbr.engine.CaseBase.plot_retention

`CaseBase.`**`plot_retention`**`(`*case*, *case_matches*`)`

Plot the retention result.

Parameters **case** : Case

The query case.

case_matches : dict[int, CaseMatch]

The corrected solution

mlpy.knowledgerep.cbr.engine.CaseBase.plot_retrieval

`CaseBase.`**`plot_retrieval`**(*case*, *case_id_list*, *names=None*)

Plot the retrieved data.

Parameters **case** : Case

The query case.

case_id_list : list[int]

The ids of the cases identified to be similar.

names : str or list[str]

The name(s) of the features for which similar cases were retrieve.

mlpy.knowledgerep.cbr.engine.CaseBase.plot_reuse

`CaseBase.`**`plot_reuse`**(*case*, *case_matches*, *revised_matches*)

Plot the reuse result.

Parameters **case** : Case

The query case.

case_matches : dict[int, CaseMatch]

The solution to the problem-solving experience.

revised_matches : dict[int, CaseMatch]

The revised solution to the problem-solving experience.

mlpy.knowledgerep.cbr.engine.CaseBase.plot_revision

`CaseBase.`**`plot_revision`**(*case*, *case_matches*)

Plot revision results.

Parameters **case** : Case

The query case.

case_matches : dict[int, CaseMatch]

The revised solution to the problem-solving experience.

mlpy.knowledgerep.cbr.engine.CaseBase.retain

`CaseBase.`**`retain`**(*case*, *case_matches*)

Retain new case.

Retain new case depending on the revise outcomes and the CBR policy regarding case retention.

Parameters **case** : Case

The query case.

case_matches : dict[int, CaseMatch]

The corrected solution

mlpy.knowledgerep.cbr.engine.CaseBase.retrieve

`CaseBase.`**`retrieve`**(*case*, *names=None*, *validity_check=True*, ***kwargs*)

Retrieve cases similar to the query case.

Parameters **case** : Case

The query case.

names : str or list[str]

The name(s) of the features for which to retrieve similar cases.

validity_check : bool

This flag controls whether the dirty flag is being checked before determining whether to rebuild the indexing structure or not.

Returns dict[int, CaseMatch] :

The solution to the problem-solving experience.

Other Parameters **cases** : dict[int, Case]

The complete case base from which to build the indexing structure used by the similarity model.

data : ndarray[ndarray[float]]

If this keyword is set, the cases in the case base are ignored and the data entries specified in this variable are used to build the indexing structure.

names : str or list[str]

The feature name(s) relevant for the similarity computation. This field is only required if the cases for building the indexing structure comes from the *cases* field.

id_map : dict[int, int]

The mapping from the data stored in the *data* field to their case ids. This field is only required if the data for building the indexing structure comes from the *data* field.

mlpy.knowledgerep.cbr.engine.CaseBase.reuse

`CaseBase.`**`reuse`**(*case*, *case_matches*)

Performs the reuse step

Performs new generalizations and specializations as a consequence of the solution transformation.

Parameters **case** : Case

The query case.

case_matches : dict[int, CaseMatch]

The solution to the problem-solving experience.

Returns dict[int, CaseMatch] :

The revised solution to the problem-solving experience.

mlpy.knowledgerep.cbr.engine.CaseBase.revision

`CaseBase.`**`revision`**`(`*case*, *case_matches*`)`

Evaluate solution provided by problem-solving experience.

Parameters **case** : Case

The query case.

case_matches : dict[int, CaseMatch]

The revised solution to the problem-solving experience.

Returns dict[int, CaseMatch] :

The corrected solution.

mlpy.knowledgerep.cbr.engine.CaseBase.run

`CaseBase.`**`run`**`(`*case*`)`

Run the case base.

Run the case base using the CBR methods retrieve, reuse, revision and retention.

Parameters **case** : Case

The query case

Returns dict[int, CaseMatch] :

The solution to the problem-solving experience

mlpy.knowledgerep.cbr.engine.CaseBase.save

`CaseBase.`**`save`**`(`*filename*`)`

Features

FeatureFactory	The feature factory.
Feature	The abstract feature class.
BoolFeature	The boolean feature.
StringFeature	The string feature.
IntFeature	The integer feature.
FloatFeature	The float feature.

mlpy.knowledgerep.cbr.features.FeatureFactory

class `mlpy.knowledgerep.cbr.features.`**`FeatureFactory`**

Bases: `object`

The feature factory.

An instance of a feature can be created by passing the feature type.

Examples

```
>>> from mlpy.knowledgerep.cbr.features import FeatureFactory
>>> FeatureFactory.create('float', **{})
```

Methods

create(_type, name, value, **kwargs)	Create a feature of the given type.

mlpy.knowledgerep.cbr.features.FeatureFactory.create

static `FeatureFactory.`**`create`**(*_type*, *name*, *value*, ***kwargs*)

Create a feature of the given type.

Parameters **_type: str**

The feature type. Valid feature types are:

bool The feature values are boolean types (*BoolFeature*).

string The feature values are of types sting (*StringFeature*).

int The feature values are of type integer (*IntFeature*).

float The feature values are of type float (*FloatFeature*).

kwargs : dict, optional

Non-positional arguments to pass to the class of the given type for initialization.

Returns Feature :

A feature instance of the given type.

mlpy.knowledgerep.cbr.features.Feature

class `mlpy.knowledgerep.cbr.features.`**`Feature`**(*name*, *value*, ***kwargs*)

Bases: `object`

The abstract feature class.

A feature consists of one or more feature values.

Parameters **name** : str

The name of the feature (this also serves as the features identifying key).

value : bool or string or int or float or list

The feature value.

Other Parameters **weight** : float or list[float]

The weights given to each feature value.

is_index : bool

Flag indicating whether this feature is an index.

retrieval_method : str

The similarity model used for retrieval. Refer to *Feature.retrieval_method* for valid methods.

retrieval_method_params : dict

Parameters relevant to the selected retrieval method.

retrieval_algorithm : str

The internal indexing structure of the training data. Refer to *Feature.retrieval_method* for valid algorithms.

retrieval_metric : str

The metric used to compute the distances between pairs of points. Refer to `sklearn.neighbors.DistanceMetric` for valid identifiers.

retrieval_metric_params : dict

Parameters relevant to the specified metric.

Attributes

is_index	Flag indicating whether this feature is an index.
name	The name of the feature (this also serves as the features identifying key).
retrieval_algorithm	The internal indexing structure of the training data.
retrieval_method	The similarity model used during retrieval.
retrieval_method_params	Parameters relevant to the specified retrieval method.
retrieval_metric	The metric used to compute the distances between pairs of points.
retrieval_metric_params	Parameters relevant to the specified metric.
value	The feature value.
weight	The weights given to each feature value

mlpy.knowledgerep.cbr.features.Feature.is_index

`Feature.`**`is_index`**

Flag indicating whether this feature is an index.

Returns bool :

Whether the feature is an index.

mlpy.knowledgerep.cbr.features.Feature.name

`Feature.`**`name`**

The name of the feature (this also serves as the features identifying key).

Returns str :

The name of the feature.

mlpy.knowledgerep.cbr.features.Feature.retrieval_algorithm

`Feature.`**`retrieval_algorithm`**

The internal indexing structure of the training data.

The retrieval algorithm is only relevant for `NeighborSimilarity`. Valid algorithms are:

ball_tree A ball tree data structure is used for computational efficiency of the calculation of the distances between pairs of points.

kd_tree A K-D Tree data structure is used for computational efficiency of the calculation of the distances between pairs of points.

brute The nearest neighbors are determined by brute-force computation of distances between all pairs of points in the dataset.

auto When `auto` is passed, the algorithm attempts to determine the best approach from the training data.

Returns str :

The retrieval algorithm.

mlpy.knowledgerep.cbr.features.Feature.retrieval_method

`Feature.`**`retrieval_method`**

The similarity model used during retrieval.

Valid models are:

knn A k-nearest-neighbor algorithm is used to determine similarity between cases (`NeighborSimilarity`). The value *k* must be specified.

radius-n Similarity between cases is determined by the nearest neighbors within a radius (`NeighborSimilarity`). The radius *n* must be specified.

kmeans Similarity is determined by a kmeans clustering algorithm (`KMeansSimilarity`).

exact-match Only exact matches are considered similar (`ExactMatchSimilarity`).

cosine A cosine similarity measure is used to determine similarity between cases (`CosineSimilarity`).

Returns str :

The retrieval method.

mlpy.knowledgerep.cbr.features.Feature.retrieval_method_params

`Feature.`**`retrieval_method_params`**

Parameters relevant to the specified retrieval method.

Returns dict :

Retrieval parameters.

mlpy.knowledgerep.cbr.features.Feature.retrieval_metric

`Feature.`**`retrieval_metric`**

The metric used to compute the distances between pairs of points.

The retrieval metric is only relevant for `NeighborSimilarity`. Refer to `sklearn.neighbors.DistanceMetric` for valid metric identifiers.

Returns str :

The retrieval metric.

mlpy.knowledgerep.cbr.features.Feature.retrieval_metric_params

`Feature.`**`retrieval_metric_params`**

Parameters relevant to the specified metric.

Returns dict :

The retrieval metric parameters.

mlpy.knowledgerep.cbr.features.Feature.value

`Feature.`**`value`**

The feature value.

Returns bool or string or int or float :

The feature value.

mlpy.knowledgerep.cbr.features.Feature.weight

`Feature.`**`weight`**

The weights given to each feature value

Returns float or list[float] :

The feature weights.

Methods

compare(other)	Compare this feature to another feature.

mlpy.knowledgerep.cbr.features.Feature.compare

`Feature.`**`compare`**(*other*)

Compare this feature to another feature.

Parameters **other** : Feature

The other feature to compare this feature to.

Returns float :

The similarity metric.

Raises NotImplementedError:

If the child class does not implement this function.

mlpy.knowledgerep.cbr.features.BoolFeature

class `mlpy.knowledgerep.cbr.features.`**`BoolFeature`**(*name*, *value*, ***kwargs*)

Bases: *`mlpy.knowledgerep.cbr.features.Feature`*

The boolean feature.

The boolean feature is represented by a scalar.

Parameters name : str

The name of the feature (this also serves as the features identifying key).

value : bool or string or int or float or list

The feature value.

Other Parameters weight : float or list[float]

The weights given to each feature value.

is_index : bool

Flag indicating whether this feature is an index.

retrieval_method : str

The similarity model used for retrieval. Refer to *`Feature.retrieval_method`* for valid methods.

retrieval_method_params : dict

Parameters relevant to the selected retrieval method.

retrieval_algorithm : str

The internal indexing structure of the training data. Refer to *`Feature.retrieval_method`* for valid algorithms.

retrieval_metric : str

The metric used to compute the distances between pairs of points. Refer to `sklearn.neighbors.DistanceMetric` for valid identifiers.

retrieval_metric_params : dict

Parameters relevant to the specified metric.

Raises ValueError :

If the feature values is not of type *boolean*.

Attributes

`is_index`	Flag indicating whether this feature is an index.

Continued on next page

Table 14.12 – continued from previous page

name	The name of the feature (this also serves as the features identifying key).
retrieval_algorithm	The internal indexing structure of the training data.
retrieval_method	The similarity model used during retrieval.
retrieval_method_params	Parameters relevant to the specified retrieval method.
retrieval_metric	The metric used to compute the distances between pairs of points.
retrieval_metric_params	Parameters relevant to the specified metric.
value	The feature value.
weight	The weights given to each feature value

mlpy.knowledgerep.cbr.features.BoolFeature.is_index

`BoolFeature.`**`is_index`**

Flag indicating whether this feature is an index.

Returns bool :

Whether the feature is an index.

mlpy.knowledgerep.cbr.features.BoolFeature.name

`BoolFeature.`**`name`**

The name of the feature (this also serves as the features identifying key).

Returns str :

The name of the feature.

mlpy.knowledgerep.cbr.features.BoolFeature.retrieval_algorithm

`BoolFeature.`**`retrieval_algorithm`**

The internal indexing structure of the training data.

The retrieval algorithm is only relevant for `NeighborSimilarity`. Valid algorithms are:

ball_tree A ball tree data structure is used for computational efficiency of the calculation of the distances between pairs of points.

kd_tree A K-D Tree data structure is used for computational efficiency of the calculation of the distances between pairs of points.

brute The nearest neighbors are determined by brute-force computation of distances between all pairs of points in the dataset.

auto When `auto` is passed, the algorithm attempts to determine the best approach from the training data.

Returns str :

The retrieval algorithm.

mlpy.knowledgerep.cbr.features.BoolFeature.retrieval_method

`BoolFeature.`**`retrieval_method`**

The similarity model used during retrieval.

Valid models are:

knn A k-nearest-neighbor algorithm is used to determine similarity between cases (`NeighborSimilarity`). The value *k* must be specified.

radius-n Similarity between cases is determined by the nearest neighbors within a radius (`NeighborSimilarity`). The radius *n* must be specified.

kmeans Similarity is determined by a kmeans clustering algorithm (`KMeansSimilarity`).

exact-match Only exact matches are considered similar (`ExactMatchSimilarity`).

cosine A cosine similarity measure is used to determine similarity between cases (`CosineSimilarity`).

Returns str :

The retrieval method.

mlpy.knowledgerep.cbr.features.BoolFeature.retrieval_method_params

`BoolFeature.`**`retrieval_method_params`**

Parameters relevant to the specified retrieval method.

Returns dict :

Retrieval parameters.

mlpy.knowledgerep.cbr.features.BoolFeature.retrieval_metric

`BoolFeature.`**`retrieval_metric`**

The metric used to compute the distances between pairs of points.

The retrieval metric is only relevant for `NeighborSimilarity`. Refer to `sklearn.neighbors.DistanceMetric` for valid metric identifiers.

Returns str :

The retrieval metric.

mlpy.knowledgerep.cbr.features.BoolFeature.retrieval_metric_params

`BoolFeature.`**`retrieval_metric_params`**

Parameters relevant to the specified metric.

Returns dict :

The retrieval metric parameters.

mlpy.knowledgerep.cbr.features.BoolFeature.value

`BoolFeature.value`
The feature value.

Returns bool or string or int or float :

The feature value.

mlpy.knowledgerep.cbr.features.BoolFeature.weight

`BoolFeature.weight`
The weights given to each feature value

Returns float or list[float] :

The feature weights.

Methods

compare(other)	Compare this feature to another feature.

mlpy.knowledgerep.cbr.features.BoolFeature.compare

`BoolFeature.compare(other)`
Compare this feature to another feature.

The strings are compared directly and receive a similarity measure of *1* if they are the same, *0* otherwise.

Parameters **other** : Feature

The other feature to compare this feature to.

Returns float :

The similarity metric.

mlpy.knowledgerep.cbr.features.StringFeature

class `mlpy.knowledgerep.cbr.features.StringFeature(name, value, **kwargs)`
Bases: *mlpy.knowledgerep.cbr.features.Feature*

The string feature.

The string feature is represented by a scalar.

Parameters **name** : str

The name of the feature (this also serves as the features identifying key).

value : bool or string or int or float or list

The feature value.

Other Parameters **weight** : float or list[float]

The weights given to each feature value.

is_index : bool

Flag indicating whether this feature is an index.

retrieval_method : str

The similarity model used for retrieval. Refer to `Feature.retrieval_method` for valid methods.

retrieval_method_params : dict

Parameters relevant to the selected retrieval method.

retrieval_algorithm : str

The internal indexing structure of the training data. Refer to `Feature.retrieval_method` for valid algorithms.

retrieval_metric : str

The metric used to compute the distances between pairs of points. Refer to `sklearn.neighbors.DistanceMetric` for valid identifiers.

retrieval_metric_params : dict

Parameters relevant to the specified metric.

Raises ValueError

If the feature values is not of type *string*.

Attributes

`is_index`	Flag indicating whether this feature is an index.
`name`	The name of the feature (this also serves as the features identifying key).
`retrieval_algorithm`	The internal indexing structure of the training data.
`retrieval_method`	The similarity model used during retrieval.
`retrieval_method_params`	Parameters relevant to the specified retrieval method.
`retrieval_metric`	The metric used to compute the distances between pairs of points.
`retrieval_metric_params`	Parameters relevant to the specified metric.
`value`	The feature value.
`weight`	The weights given to each feature value

mlpy.knowledgerep.cbr.features.StringFeature.is_index

StringFeature.**is_index**

Flag indicating whether this feature is an index.

Returns bool :

Whether the feature is an index.

mlpy.knowledgerep.cbr.features.StringFeature.name

StringFeature.**name**

The name of the feature (this also serves as the features identifying key).

Returns str :

The name of the feature.

mlpy.knowledgerep.cbr.features.StringFeature.retrieval_algorithm

`StringFeature.`**`retrieval_algorithm`**

The internal indexing structure of the training data.

The retrieval algorithm is only relevant for `NeighborSimilarity`. Valid algorithms are:

ball_tree A ball tree data structure is used for computational efficiency of the calculation of the distances between pairs of points.

kd_tree A K-D Tree data structure is used for computational efficiency of the calculation of the distances between pairs of points.

brute The nearest neighbors are determined by brute-force computation of distances between all pairs of points in the dataset.

auto When `auto` is passed, the algorithm attempts to determine the best approach from the training data.

Returns str :

The retrieval algorithm.

mlpy.knowledgerep.cbr.features.StringFeature.retrieval_method

`StringFeature.`**`retrieval_method`**

The similarity model used during retrieval.

Valid models are:

knn A k-nearest-neighbor algorithm is used to determine similarity between cases (`NeighborSimilarity`). The value *k* must be specified.

radius-n Similarity between cases is determined by the nearest neighbors within a radius (`NeighborSimilarity`). The radius *n* must be specified.

kmeans Similarity is determined by a kmeans clustering algorithm (`KMeansSimilarity`).

exact-match Only exact matches are considered similar (`ExactMatchSimilarity`).

cosine A cosine similarity measure is used to determine similarity between cases (`CosineSimilarity`).

Returns str :

The retrieval method.

mlpy.knowledgerep.cbr.features.StringFeature.retrieval_method_params

`StringFeature.`**`retrieval_method_params`**

Parameters relevant to the specified retrieval method.

Returns dict :

Retrieval parameters.

mlpy.knowledgerep.cbr.features.StringFeature.retrieval_metric

`StringFeature.`**`retrieval_metric`**

The metric used to compute the distances between pairs of points.

The retrieval metric is only relevant for `NeighborSimilarity`. Refer to `sklearn.neighbors.DistanceMetric` for valid metric identifiers.

Returns str :

The retrieval metric.

mlpy.knowledgerep.cbr.features.StringFeature.retrieval_metric_params

`StringFeature.`**`retrieval_metric_params`**

Parameters relevant to the specified metric.

Returns dict :

The retrieval metric parameters.

mlpy.knowledgerep.cbr.features.StringFeature.value

`StringFeature.`**`value`**

The feature value.

Returns bool or string or int or float :

The feature value.

mlpy.knowledgerep.cbr.features.StringFeature.weight

`StringFeature.`**`weight`**

The weights given to each feature value

Returns float or list[float] :

The feature weights.

Methods

`compare`(other)	Compare this feature to another feature.

mlpy.knowledgerep.cbr.features.StringFeature.compare

`StringFeature.`**`compare`**(*other*)

Compare this feature to another feature.

The strings are compared directly and receive a similarity measure of *1* if they are the same, *0* otherwise.

Parameters **other** : Feature

The other feature to compare this feature to.

Returns float :

The similarity metric.

mlpy.knowledgerep.cbr.features.IntFeature

class `mlpy.knowledgerep.cbr.features.`**`IntFeature`**(*name*, *value*, ***kwargs*)

Bases: *`mlpy.knowledgerep.cbr.features.Feature`*

The integer feature.

The integer feature is either represented by a scalar or by a list or values.

Parameters **name** : str

The name of the feature (this also serves as the features identifying key).

value : bool or string or int or float or list

The feature value.

Other Parameters **weight** : float or list[float]

The weights given to each feature value.

is_index : bool

Flag indicating whether this feature is an index.

retrieval_method : str

The similarity model used for retrieval. Refer to *`Feature.retrieval_method`* for valid methods.

retrieval_method_params : dict

Parameters relevant to the selected retrieval method.

retrieval_algorithm : str

The internal indexing structure of the training data. Refer to *`Feature.retrieval_method`* for valid algorithms.

retrieval_metric : str

The metric used to compute the distances between pairs of points. Refer to `sklearn.neighbors.DistanceMetric` for valid identifiers.

retrieval_metric_params : dict

Parameters relevant to the specified metric.

Raises **ValueError**

If not all feature values are of type *integer*.

Attributes

`is_index`	Flag indicating whether this feature is an index.
`name`	The name of the feature (this also serves as the features identifying key).
`retrieval_algorithm`	The internal indexing structure of the training data.
`retrieval_method`	The similarity model used during retrieval.

Continued on next page

Table 14.16 – continued from previous page

retrieval_method_params	Parameters relevant to the specified retrieval method.
retrieval_metric	The metric used to compute the distances between pairs of points.
retrieval_metric_params	Parameters relevant to the specified metric.
value	The feature value.
weight	The weights given to each feature value

mlpy.knowledgerep.cbr.features.IntFeature.is_index

IntFeature.**is_index**

Flag indicating whether this feature is an index.

Returns bool :

Whether the feature is an index.

mlpy.knowledgerep.cbr.features.IntFeature.name

IntFeature.**name**

The name of the feature (this also serves as the features identifying key).

Returns str :

The name of the feature.

mlpy.knowledgerep.cbr.features.IntFeature.retrieval_algorithm

IntFeature.**retrieval_algorithm**

The internal indexing structure of the training data.

The retrieval algorithm is only relevant for `NeighborSimilarity`. Valid algorithms are:

ball_tree A ball tree data structure is used for computational efficiency of the calculation of the distances between pairs of points.

kd_tree A K-D Tree data structure is used for computational efficiency of the calculation of the distances between pairs of points.

brute The nearest neighbors are determined by brute-force computation of distances between all pairs of points in the dataset.

auto When `auto` is passed, the algorithm attempts to determine the best approach from the training data.

Returns str :

The retrieval algorithm.

mlpy.knowledgerep.cbr.features.IntFeature.retrieval_method

IntFeature.**retrieval_method**

The similarity model used during retrieval.

Valid models are:

knn A k-nearest-neighbor algorithm is used to determine similarity between cases (`NeighborSimilarity`). The value *k* must be specified.

radius-n Similarity between cases is determined by the nearest neighbors within a radius (`NeighborSimilarity`). The radius *n* must be specified.

kmeans Similarity is determined by a kmeans clustering algorithm (`KMeansSimilarity`).

exact-match Only exact matches are considered similar (`ExactMatchSimilarity`).

cosine A cosine similarity measure is used to determine similarity between cases (`CosineSimilarity`).

Returns str :

The retrieval method.

mlpy.knowledgerep.cbr.features.IntFeature.retrieval_method_params

`IntFeature.`**`retrieval_method_params`**

Parameters relevant to the specified retrieval method.

Returns dict :

Retrieval parameters.

mlpy.knowledgerep.cbr.features.IntFeature.retrieval_metric

`IntFeature.`**`retrieval_metric`**

The metric used to compute the distances between pairs of points.

The retrieval metric is only relevant for `NeighborSimilarity`. Refer to `sklearn.neighbors.DistanceMetric` for valid metric identifiers.

Returns str :

The retrieval metric.

mlpy.knowledgerep.cbr.features.IntFeature.retrieval_metric_params

`IntFeature.`**`retrieval_metric_params`**

Parameters relevant to the specified metric.

Returns dict :

The retrieval metric parameters.

mlpy.knowledgerep.cbr.features.IntFeature.value

`IntFeature.`**`value`**

The feature value.

Returns bool or string or int or float :

The feature value.

mlpy.knowledgerep.cbr.features.IntFeature.weight

`IntFeature.`**`weight`**

The weights given to each feature value

Returns float or list[float] :

The feature weights.

Methods

compare(other)	Compare this feature to another feature.

mlpy.knowledgerep.cbr.features.IntFeature.compare

`IntFeature.`**`compare`**(*other*)

Compare this feature to another feature.

If the feature is represented by a list the similarity between the two features is determined by the Euclidean distance of the feature values.

Parameters **other** : Feature

The other feature to compare this feature to.

Returns float :

The similarity metric.

mlpy.knowledgerep.cbr.features.FloatFeature

class `mlpy.knowledgerep.cbr.features.`**`FloatFeature`**(*name*, *value*, ***kwargs*)

Bases: *mlpy.knowledgerep.cbr.features.Feature*

The float feature.

The float feature is either represented by a scalar or by a list or values.

Parameters **name** : str

The name of the feature (this also serves as the features identifying key).

value : bool or string or int or float or list

The feature value.

Other Parameters **weight** : float or list[float]

The weights given to each feature value.

is_index : bool

Flag indicating whether this feature is an index.

retrieval_method : str

The similarity model used for retrieval. Refer to *Feature.retrieval_method* for valid methods.

retrieval_method_params : dict

Parameters relevant to the selected retrieval method.

retrieval_algorithm : str

The internal indexing structure of the training data. Refer to `Feature.retrieval_method` for valid algorithms.

retrieval_metric : str

The metric used to compute the distances between pairs of points. Refer to `sklearn.neighbors.DistanceMetric` for valid identifiers.

retrieval_metric_params : dict

Parameters relevant to the specified metric.

Raises ValueError

If not all feature values are of type *float*.

Attributes

`is_index`	Flag indicating whether this feature is an index.
`name`	The name of the feature (this also serves as the features identifying key).
`retrieval_algorithm`	The internal indexing structure of the training data.
`retrieval_method`	The similarity model used during retrieval.
`retrieval_method_params`	Parameters relevant to the specified retrieval method.
`retrieval_metric`	The metric used to compute the distances between pairs of points.
`retrieval_metric_params`	Parameters relevant to the specified metric.
`value`	The feature value.
`weight`	The weights given to each feature value

mlpy.knowledgerep.cbr.features.FloatFeature.is_index

`FloatFeature.`**`is_index`**

Flag indicating whether this feature is an index.

Returns bool :

Whether the feature is an index.

mlpy.knowledgerep.cbr.features.FloatFeature.name

`FloatFeature.`**`name`**

The name of the feature (this also serves as the features identifying key).

Returns str :

The name of the feature.

mlpy.knowledgerep.cbr.features.FloatFeature.retrieval_algorithm

`FloatFeature.`**`retrieval_algorithm`**

The internal indexing structure of the training data.

The retrieval algorithm is only relevant for `NeighborSimilarity`. Valid algorithms are:

ball_tree A ball tree data structure is used for computational efficiency of the calculation of the distances between pairs of points.

kd_tree A K-D Tree data structure is used for computational efficiency of the calculation of the distances between pairs of points.

brute The nearest neighbors are determined by brute-force computation of distances between all pairs of points in the dataset.

auto When `auto` is passed, the algorithm attempts to determine the best approach from the training data.

Returns str :

The retrieval algorithm.

mlpy.knowledgerep.cbr.features.FloatFeature.retrieval_method

`FloatFeature.`**`retrieval_method`**

The similarity model used during retrieval.

Valid models are:

knn A k-nearest-neighbor algorithm is used to determine similarity between cases (`NeighborSimilarity`). The value *k* must be specified.

radius-n Similarity between cases is determined by the nearest neighbors within a radius (`NeighborSimilarity`). The radius *n* must be specified.

kmeans Similarity is determined by a kmeans clustering algorithm (`KMeansSimilarity`).

exact-match Only exact matches are considered similar (`ExactMatchSimilarity`).

cosine A cosine similarity measure is used to determine similarity between cases (`CosineSimilarity`).

Returns str :

The retrieval method.

mlpy.knowledgerep.cbr.features.FloatFeature.retrieval_method_params

`FloatFeature.`**`retrieval_method_params`**

Parameters relevant to the specified retrieval method.

Returns dict :

Retrieval parameters.

mlpy.knowledgerep.cbr.features.FloatFeature.retrieval_metric

`FloatFeature.`**`retrieval_metric`**

The metric used to compute the distances between pairs of points.

The retrieval metric is only relevant for `NeighborSimilarity`. Refer to `sklearn.neighbors.DistanceMetric` for valid metric identifiers.

Returns str :

The retrieval metric.

mlpy.knowledgerep.cbr.features.FloatFeature.retrieval_metric_params

`FloatFeature.`**`retrieval_metric_params`**

Parameters relevant to the specified metric.

Returns dict :

The retrieval metric parameters.

mlpy.knowledgerep.cbr.features.FloatFeature.value

`FloatFeature.`**`value`**

The feature value.

Returns bool or string or int or float :

The feature value.

mlpy.knowledgerep.cbr.features.FloatFeature.weight

`FloatFeature.`**`weight`**

The weights given to each feature value

Returns float or list[float] :

The feature weights.

Methods

compare(other)	Compare this feature to another feature.

mlpy.knowledgerep.cbr.features.FloatFeature.compare

`FloatFeature.`**`compare`**`(`*other*`)`

Compare this feature to another feature.

If the feature is represented by a list the similarity between the two features is determined by the Euclidean distance of the feature values.

Parameters **other** : Feature

The other feature to compare this feature to.

Returns float :

The similarity metric.

Similarity measures

Stat	The similarity statistics container.
SimilarityFactory	The similarity factory.
ISimilarity	The similarity model interface.
NeighborSimilarity	The neighborhood similarity model.
KMeansSimilarity	The KMeans similarity model.
ExactMatchSimilarity	The exact match similarity model.
CosineSimilarity	The cosine similarity model.

mlpy.knowledgerep.cbr.similarity.Stat

class `mlpy.knowledgerep.cbr.similarity.`**`Stat`**(*case_id*, *similarity=None*)

Bases: `object`

The similarity statistics container.

The similarity statistics is a container to pass the calculated measure of similarity between the case identified by the case id and the query case between functions.

Parameters **case_id** : int

The case's id.

similarity : float

The similarity measure.

Attributes

case_id	The case's id.
similarity	The similarity measure.

mlpy.knowledgerep.cbr.similarity.Stat.case_id

`Stat.`**`case_id`**

The case's id.

Returns int :

The case's id

mlpy.knowledgerep.cbr.similarity.Stat.similarity

`Stat.`**`similarity`**

The similarity measure.

Returns float :

The similarity measure.

mlpy.knowledgerep.cbr.similarity.SimilarityFactory

class `mlpy.knowledgerep.cbr.similarity.`**`SimilarityFactory`**

Bases: `object`

The similarity factory.

An instance of a similarity model can be created by passing the similarity model type.

Examples

```
>>> from mlpy.knowledgerep.cbr.similarity import SimilarityFactory
>>> SimilarityFactory.create('float', **{})
```

Methods

`create`(_type, **kwargs)	Create a feature of the given type.

mlpy.knowledgerep.cbr.similarity.SimilarityFactory.create

static `SimilarityFactory.`**`create`**(*_type*, ***kwargs*)

Create a feature of the given type.

Parameters **_type** : str

The feature type. Valid feature types are:

knn A k-nearest-neighbor algorithm is used to determine similarity between cases (*`NeighborSimilarity`*). The value `n_neighbors` must be specified.

radius-n Similarity between cases is determined by the nearest neighbors within a radius (*`NeighborSimilarity`*). The value `radius` must be specified.

kmeans Similarity is determined by a KMeans clustering algorithm (*`KMeansSimilarity`*). The value `n_clusters` must be specified.

exact-match Only exact matches are considered similar (*`ExactMatchSimilarity`*).

cosine A cosine similarity measure is used to determine similarity between cases (*`CosineSimilarity`*).

kwargs : dict, optional

Non-positional arguments to pass to the class of the given type for initialization.

Returns ISimilarity :

A similarity instance of the given type.

mlpy.knowledgerep.cbr.similarity.ISimilarity

class `mlpy.knowledgerep.cbr.similarity.`**`ISimilarity`**

Bases: `object`

The similarity model interface.

The similarity model keeps an internal indexing structure of the relevant case data to efficiently computing the similarity measure between data points.

Notes

All similarity models must inherit from this class.

Methods

`build_indexing_structure`(data, id_map)	Build the indexing structure.
`compute_similarity`(data_point)	Computes the similarity.

mlpy.knowledgerep.cbr.similarity.ISimilarity.build_indexing_structure

`ISimilarity.`**`build_indexing_structure`**(*data*, *id_map*)

Build the indexing structure.

Parameters **data** : ndarray[ndarray[float]]

The raw data points to be indexed.

id_map : dict[int, int]

The mapping from the data points to their case ids.

Raises **NotImplementedError**

If the child class does not implement this function.

mlpy.knowledgerep.cbr.similarity.ISimilarity.compute_similarity

`ISimilarity.`**`compute_similarity`**(*data_point*)

Computes the similarity.

Computes the similarity between the data point and the data in the indexing structure returning the results in a collection of similarity statistics (*`Stat`*).

Parameters **data_point** : list[float]

The raw data point to compare against the data points stored in the indexing structure.

Returns list[Stat] :

A collection of similarity statistics.

Raises **NotImplementedError**

If the child class does not implement this function.

mlpy.knowledgerep.cbr.similarity.NeighborSimilarity

class `mlpy.knowledgerep.cbr.similarity.`**`NeighborSimilarity`**(*n_neighbors=None*, *radius=None*, *algorithm=None*, *metric=None*, *metric_params=None*)

Bases: *`mlpy.knowledgerep.cbr.similarity.ISimilarity`*

The neighborhood similarity model.

The neighbor similarity model determines similarity between the data in the indexing structure and the query data by using the nearest neighbor algorithm `sklearn.neighbors.NearestNeighbors`.

Both a k-neighbors classifier and a radius-neighbor-classifier are implemented. To choose between the classifiers either *n_neighbors* or *radius* must be specified.

Parameters **n_neighbors** : int

The number of data points considered to be closest neighbors.

radius : int

The radius around the query data point, within which the data points are considered closest neighbors.

algorithm : str

The internal indexing structure of the training data. Defaults to *kd-tree*.

metric : str

The metric used to compute the distances between pairs of points. Refer to `sklearn.neighbors.DistanceMetric` for valid identifiers. Default is *euclidean*.

metric_params : dict

Parameters relevant to the specified metric.

Raises **UserWarning :**

If the either both or none of *n_neighbors* and *radius* are given.

See also:

`sklearn.neighbors.KNeighborsClassifier`, `sklearn.neighbors.RadiusNeighborsClassifier`

Methods

`build_indexing_structure`(data, id_map)	Build the indexing structure.
`compute_similarity`(data_point)	Computes the similarity.

mlpy.knowledgerep.cbr.similarity.NeighborSimilarity.build_indexing_structure

`NeighborSimilarity.`**`build_indexing_structure`**(*data*, *id_map*)

Build the indexing structure.

Build the indexing structure by fitting the data according to the specified algorithm.

Parameters **data** : ndarray[ndarray[float]]

The raw data points to be indexed.

id_map : dict[int, int]

The mapping from the data points to their case ids.

mlpy.knowledgerep.cbr.similarity.NeighborSimilarity.compute_similarity

NeighborSimilarity.**compute_similarity**(*data_point*)

Computes the similarity.

Computes the similarity between the data point and the data in the indexing structure using the `sklearn.neighbors.NearestNeighbors` algorithm. The results are returned in a collection of similarity statistics (*Stat*).

Parameters **data_point** : list[float]

The raw data point to compare against the data points stored in the indexing structure.

Returns list[Stat] :

A collection of similarity statistics.

mlpy.knowledgerep.cbr.similarity.KMeansSimilarity

class mlpy.knowledgerep.cbr.similarity.**KMeansSimilarity**(*n_cluster=None*)

Bases: *mlpy.knowledgerep.cbr.similarity.ISimilarity*

The KMeans similarity model.

The KMeans similarity model determines similarity between the data in the indexing structure and the query data by using the `sklearn.cluster.KMeans` algorithm.

Parameters **n_cluster** : int

The number of clusters to fit the raw data in.

Methods

build_indexing_structure(data, id_map)	Build the indexing structure.
compute_similarity(data_point)	Computes the similarity.

mlpy.knowledgerep.cbr.similarity.KMeansSimilarity.build_indexing_structure

KMeansSimilarity.**build_indexing_structure**(*data*, *id_map*)

Build the indexing structure.

Build the indexing structure by fitting the data into *n_cluster* clusters.

Parameters **data** : ndarray[ndarray[float]]

The raw data points to be indexed.

id_map : dict[int, int]

The mapping from the data points to their case ids.

mlpy.knowledgerep.cbr.similarity.KMeansSimilarity.compute_similarity

KMeansSimilarity.**compute_similarity**(*data_point*)

Computes the similarity.

Computes the similarity between the data point and the data in the indexing structure using the `sklearn.cluster.KMeans` clustering algorithm. The results are returned in a collection of similarity statistics (*Stat*).

Parameters **data_point** : list[float]

The raw data point to compare against the data points stored in the indexing structure.

Returns list[Stat] :

A collection of similarity statistics.

mlpy.knowledgerep.cbr.similarity.ExactMatchSimilarity

class mlpy.knowledgerep.cbr.similarity.**ExactMatchSimilarity**(***kwargs*)

Bases: *mlpy.knowledgerep.cbr.similarity.ISimilarity*

The exact match similarity model.

The exact match similarity model considered only exact matches between the data in the indexing structure and the query data as similar.

Methods

build_indexing_structure(data, id_map)	Build the indexing structure.
compute_similarity(data_point)	Computes the similarity.

mlpy.knowledgerep.cbr.similarity.ExactMatchSimilarity.build_indexing_structure

ExactMatchSimilarity.**build_indexing_structure**(*data*, *id_map*)

Build the indexing structure.

To determine exact matches a brute-force algorithm is used thus the data remains as is and no special indexing structure is implemented.

Parameters **data** : ndarray[ndarray[float]]

The raw data points to be indexed.

id_map : dict[int, int]

The mapping from the data points to their case ids.

.. todo::

It might be worth looking into a more efficient way of determining exact matches.

mlpy.knowledgerep.cbr.similarity.ExactMatchSimilarity.compute_similarity

`ExactMatchSimilarity.`**`compute_similarity`**`(`*data_point*`)`

Computes the similarity.

Computes the similarity between the data point and the data in the indexing structure identifying exact matches. The results are returned in a collection of similarity statistics (*`Stat`*).

Parameters **data_point** : list[float]

The raw data point to compare against the data points stored in the indexing structure.

Returns list[Stat] :

A collection of similarity statistics.

mlpy.knowledgerep.cbr.similarity.CosineSimilarity

class `mlpy.knowledgerep.cbr.similarity.`**`CosineSimilarity`**`(`***kwargs*`)`

Bases: *`mlpy.knowledgerep.cbr.similarity.ISimilarity`*

The cosine similarity model.

Cosine similarity is a measure of similarity between two vectors of an inner product space that measures the cosine of the angle between them. The cosine of 0 degree is 1, and it is less than 1 for any other angle. It is thus a judgement of orientation and not magnitude: tow vectors with the same orientation have a cosine similarity of 1, two vectors at 90 degrees have a similarity of 0, and two vectors diametrically opposed have a similarity of -1, independent of their magnitude *[R1]*.

The cosine model employs the cosine_similarity function from the `sklearn.metrics.pairwise` module to determine similarity.

See also:

Machine Learning::Cosine Similarity for Vector Space Models (Part III)

References

[R1]

Methods

`build_indexing_structure`(data, id_map)	Build the indexing structure.
`compute_similarity`(data_point)	Computes the similarity.

mlpy.knowledgerep.cbr.similarity.CosineSimilarity.build_indexing_structure

`CosineSimilarity.`**`build_indexing_structure`**`(`*data*, *id_map*`)`

Build the indexing structure.

The cosine_similarity function from `sklearn.metrics.pairwise` takes the raw data as input. Thus the data remains as is and no special indexing structure is implemented.

Parameters **data** : ndarray[ndarray[float]]

The raw data points to be indexed.

id_map : dict[int, int]

The mapping from the data points to their case ids.

mlpy.knowledgerep.cbr.similarity.CosineSimilarity.compute_similarity

`CosineSimilarity.`**`compute_similarity`**(*data_point*)

Computes the similarity.

Computes the similarity between the data point and the data in the indexing structure using the function `cosine_similarity` from `sklearn.metrics.pairwise`.

The resulting similarity ranges from -1 meaning exactly opposite, to 1 meaning exactly the same, with 0 indicating orthogonality (decorrelation), and in-between values indicating intermediate similarity or dissimilarity. The results are returned in a collection of similarity statistics (*`Stat`*).

Parameters **data_point** : list[float]

The raw data point to compare against the data points stored in the indexing structure.

Returns list[Stat] :

A collection of similarity statistics.

Problem solving methods

`CBRMethodFactory`	The case base reasoning factory.
`ICBRMethod`	The method interface.
`IReuseMethod`	The reuse method interface.
`IRevisionMethod`	The revision method interface.
`IRetentionMethod`	The retention method interface.
`DefaultReuseMethod`	The default reuse method implementation.
`DefaultRevisionMethod`	The default revision method implementation called from the case base.
`DefaultRetentionMethod`	The default retention method implementation called from the case base.

mlpy.knowledgerep.cbr.methods.CBRMethodFactory

class `mlpy.knowledgerep.cbr.methods.`**`CBRMethodFactory`**

Bases: `object`

The case base reasoning factory.

An instance of a case base reasoning method can be created by passing the case base reasoning method type. Case base reasoning methods are used to implement task specific reuse, revision, and retention.

Examples

```
>>> from mlpy.knowledgerep.cbr.methods import CBRMethodFactory
>>> CBRMethodFactory.create('defaultreusemethod', **{})
```

Methods

create(_type, *args, **kwargs)	Create an case base reasoning method of the given type.

mlpy.knowledgerep.cbr.methods.CBRMethodFactory.create

static `CBRMethodFactory.`**`create`**(*_type*, **args*, ***kwargs*)
Create an case base reasoning method of the given type.

_type [str] The case base reasoning method type. The method type should be equal to the class name of the method.

args [tuple, optional] Positional arguments passed to the class of the given type for initialization.

kwargs [dict, optional] Non-positional arguments passed to the class of the given type for initialization.

Returns ICBRMethod :

A case base method instance of the given type.

mlpy.knowledgerep.cbr.methods.ICBRMethod

class `mlpy.knowledgerep.cbr.methods.`**`ICBRMethod`**
Bases: `object`

The method interface.

This is the interface for reuse, revision, and retention methods and handles registration of all subclasses.

It uses the *RegistryInterface* ensuring that all subclasses are registered. Therefore, new specialty case base reasoning method classes can be defined and are recognized by the case base reasoning engine.

Notes

All case base reasoning method must inherit from this class.

Methods

execute(case, case_matches, **kwargs)	Execute reuse step.
plot_data(case, case_matches, **kwargs)	Plot the data.

mlpy.knowledgerep.cbr.methods.ICBRMethod.execute

`ICBRMethod.`**`execute`**(*case*, *case_matches*, ***kwargs*)
Execute reuse step.

Parameters **case** : Case

The query case.

case_matches : dict[int, CaseMatch]

The solution identified through the similarity measure.

kwargs : dict, optional

Non-positional arguments passed to the class of the given type for initialization.

Raises NotImplementedError

If the child class does not implement this function.

mlpy.knowledgerep.cbr.methods.ICBRMethod.plot_data

`ICBRMethod.`**`plot_data`**`(`*case*, *case_matches*, ***kwargs*`)`

Plot the data.

Parameters case : Case

The query case.

case_matches : dict[int, CaseMatch]

The solution identified through the similarity measure.

kwargs : dict, optional

Non-positional arguments passed to the class of the given type for initialization.

mlpy.knowledgerep.cbr.methods.IReuseMethod

class `mlpy.knowledgerep.cbr.methods.`**`IReuseMethod`**

Bases: *mlpy.knowledgerep.cbr.methods.ICBRMethod*

The reuse method interface.

The solutions of the best (or set of best) retrieved cases are used to construct the solution for the query case; new generalizations and specializations may occur as a consequence of the solution transformation.

Notes

All reuse method implementations must inherit from this class.

Methods

execute(case, case_matches[, fn_retrieve])	Execute reuse step.
plot_data(case, case_matches[, revised_matches])	Plot the data.

mlpy.knowledgerep.cbr.methods.IReuseMethod.execute

`IReuseMethod.`**`execute`**`(`*case*, *case_matches*, *fn_retrieve=None*`)`

Execute reuse step.

Parameters case : Case

The query case.

case_matches : dict[int, CaseMatch]

The solution identified through the similarity measure.

fn_retrieve : callable

Callback for accessing the case base's 'retrieval' method.

Returns dict[int, CaseMatch] :

The revised solution.

Raises NotImplementedError

If the child class does not implement this function.

mlpy.knowledgerep.cbr.methods.IReuseMethod.plot_data

`IReuseMethod.`**`plot_data`**`(`*case*, *case_matches*, *revised_matches=None*`)`

Plot the data.

Parameters case : Case

The query case.

case_matches : dict[int, CaseMatch]

The solution identified through the similarity measure.

revised_matches : dict[int, CaseMatch]

The revised solution.

mlpy.knowledgerep.cbr.methods.IRevisionMethod

class `mlpy.knowledgerep.cbr.methods.`**`IRevisionMethod`**

Bases: *`mlpy.knowledgerep.cbr.methods.ICBRMethod`*

The revision method interface.

The solutions provided by the query case is evaluated and information about whether the solution has or has not provided a desired outcome is gathered.

Notes

All revision method implementations must inherit from this class.

Methods

`execute`(case, case_matches, **kwargs)	Execute the revision step.
`plot_data`(case, case_matches, **kwargs)	Plot the data.

mlpy.knowledgerep.cbr.methods.IRevisionMethod.execute

`IRevisionMethod.`**`execute`**`(`*case*, *case_matches*, ***kwargs*`)`

Execute the revision step.

Parameters case : Case

The query case.

case_matches : dict[int, CaseMatch]

The solution identified through the similarity measure.

Returns dict[int, CaseMatch] :

The corrected solution.

Raises NotImplementedError

If the child class does not implement this function.

mlpy.knowledgerep.cbr.methods.IRevisionMethod.plot_data

`IRevisionMethod.`**`plot_data`**`(`*case*, *case_matches*, ***kwargs*`)`

Plot the data.

Parameters case : Case

The query case.

case_matches : dict[int, CaseMatch]

The solution identified through the similarity measure.

mlpy.knowledgerep.cbr.methods.IRetentionMethod

class `mlpy.knowledgerep.cbr.methods.`**`IRetentionMethod`**

Bases: *mlpy.knowledgerep.cbr.methods.ICBRMethod*

The retention method interface.

When the new problem-solving experience can be stored or not stored in memory, depending on the revision outcomes and the case base reasoning policy regarding case retention.

Notes

All retention method implementations must inherit from this class.

Methods

execute(case, case_matches[, fn_add])	Execute retention step.
plot_data(case, case_matches, **kwargs)	Plot the data.

mlpy.knowledgerep.cbr.methods.IRetentionMethod.execute

`IRetentionMethod.`**`execute`**`(`*case*, *case_matches*, *fn_add=None*`)`

Execute retention step.

Parameters case : Case

The query case.

case_matches : dict[int, CaseMatch]

The solution identified through the similarity measure.

fn_add : callable

Callback for accessing the case base's 'add' method.

Raises NotImplementedError

If the child class does not implement this function.

mlpy.knowledgerep.cbr.methods.IRetentionMethod.plot_data

`IRetentionMethod.`**`plot_data`**`(`*case*, *case_matches*, ***kwargs*`)`

Plot the data.

Parameters case : Case

The query case.

case_matches : dict[int, CaseMatch]

The solution identified through the similarity measure.

mlpy.knowledgerep.cbr.methods.DefaultReuseMethod

class `mlpy.knowledgerep.cbr.methods.`**`DefaultReuseMethod`**

Bases: *`mlpy.knowledgerep.cbr.methods.IReuseMethod`*

The default reuse method implementation.

The solutions of the best (or set of best) retrieved cases are used to construct the solution for the query case; new generalizations and specializations may occur as a consequence of the solution transformation.

Notes

The default reuse method does not perform any solution transformations.

Methods

`execute`(case, case_matches[, fn_retrieve])	Execute reuse step.
`plot_data`(case, case_matches[, revised_matches])	Plot the data.

mlpy.knowledgerep.cbr.methods.DefaultReuseMethod.execute

`DefaultReuseMethod.`**`execute`**`(`*case*, *case_matches*, *fn_retrieve=None*`)`

Execute reuse step.

Parameters case : Case

The query case.

case_matches : dict[int, CaseMatch]

The solution identified through the similarity measure.

fn_retrieve : callable

Callback for accessing the case base's 'retrieval' method.

Returns dict[int, CaseMatch] :

The revised solution.

mlpy.knowledgerep.cbr.methods.DefaultReuseMethod.plot_data

`DefaultReuseMethod.`**`plot_data`**(*case*, *case_matches*, *revised_matches=None*)

Plot the data.

Parameters **case** : Case

The query case.

case_matches : dict[int, CaseMatch]

The solution identified through the similarity measure.

revised_matches : dict[int, CaseMatch]

The revised solution.

mlpy.knowledgerep.cbr.methods.DefaultRevisionMethod

class `mlpy.knowledgerep.cbr.methods.`**`DefaultRevisionMethod`**

Bases: *`mlpy.knowledgerep.cbr.methods.IRevisionMethod`*

The default revision method implementation called from the case base.

The solutions provided by the query case is evaluated and information about whether the solution has or has not provided a desired outcome is gathered.

Notes

The default revision method returns the original solution without making any modifications.

Methods

`execute`(case, case_matches, **kwargs)	Execute the revision step.
`plot_data`(case, case_matches, **kwargs)	Plot the data.

mlpy.knowledgerep.cbr.methods.DefaultRevisionMethod.execute

`DefaultRevisionMethod.`**`execute`**(*case*, *case_matches*, ***kwargs*)

Execute the revision step.

Parameters **case** : Case

The query case.

case_matches : dict[int, CaseMatch]

The solution identified through the similarity measure.

Returns dict[int, CaseMatch] :

The corrected solution.

mlpy.knowledgerep.cbr.methods.DefaultRevisionMethod.plot_data

`DefaultRevisionMethod.`**`plot_data`**(*case*, *case_matches*, ***kwargs*)

Plot the data.

Parameters **case** : Case

The query case.

case_matches : dict[int, CaseMatch]

The solution identified through the similarity measure.

mlpy.knowledgerep.cbr.methods.DefaultRetentionMethod

class `mlpy.knowledgerep.cbr.methods.`**`DefaultRetentionMethod`**(*max_error=None*)

Bases: *`mlpy.knowledgerep.cbr.methods.IRetentionMethod`*

The default retention method implementation called from the case base.

When the new problem-solving experience can be stored or not stored in memory, depending on the revision outcomes and the case base reasoning policy regarding case retention.

Parameters **max_error** : float

The maximum permitted error.

Notes

The default retention method adds the new experience only if the query case is within the maximum permitted error of the most similar solution case:

$$d(\text{case}, \text{solution}[0]) < \text{max\_error}.$$

Methods

`execute`(case, case_matches[, fn_add])	Execute retention step.
`plot_data`(case, case_matches, **kwargs)	Plot the data.

mlpy.knowledgerep.cbr.methods.DefaultRetentionMethod.execute

`DefaultRetentionMethod.`**`execute`**(*case*, *case_matches*, *fn_add=None*)

Execute retention step.

Parameters **case** : Case

The query case.

case_matches : dict[int, CaseMatch]

The solution identified through the similarity measure.

fn_add : callable

Callback for accessing the case base's 'add' method.

mlpy.knowledgerep.cbr.methods.DefaultRetentionMethod.plot_data

`DefaultRetentionMethod.`**`plot_data`**(*case*, *case_matches*, ***kwargs*)

Plot the data.

Parameters **case** : Case

The query case.

case_matches : dict[int, CaseMatch]

The solution identified through the similarity measure.

CHAPTER 15

Learning algorithms (`mlpy.learners`)

LearnerFactory	The learner factory.
ILearner	The learner interface.

mlpy.learners.LearnerFactory

class `mlpy.learners.`**`LearnerFactory`**

Bases: `object`

The learner factory.

An instance of a learner can be created by passing the learner type.

Examples

```
>>> from mlpy.learners import LearnerFactory
>>> q0 = LearnerFactory.create('qlearner')
```

This creates a *QLearner* instance with default parameters.

```
>>> q1 = LearnerFactory.create('qlearner', max_steps=10)
```

This creates a *QLearner* instance with max_steps set to 10.

Methods

create(_type, *args, **kwargs)	Create an learner of the given type.

mlpy.learners.LearnerFactory.create

static `LearnerFactory.`**`create`**`(`*_type*, **args*, ***kwargs*`)`
Create an learner of the given type.

A new learner of the given type is created. If *progress* is among the keywords in *kwargs*, the factory attempts to recover the learner from the learner state saved to file *filename*. If the factory fails to load the learners state from file, a new learner is created.

Parameters **_type** : str

The learner type. Valid learner types:

qlearner Performs q-learning, a reinforcement learning variant. A *`QLearner`* module is created.

rldtlearner The learner performs reinforcement learning with decision trees (RLDT), a method introduced by Hester, Quinlan, and Stone which builds a generalized model for the transitions and rewards of the environment. A *`RLDTLearner`* module is created.

apprenticeshiplearner The learner performs apprenticeship learning via inverse reinforcement learning, a method introduced by Abbeel and Ng which strives to imitate the demonstrations given by an expert. A *`ApprenticeshipLearner`* module is create.

incrapprenticeshiplearner The learner incrementally performs apprenticeship learning via inverse reinforcement learning. Inverse reinforcement learning assumes knowledge of the underlying model. However, this is not always feasible. The incremental apprenticeship learner updates its model after every iteration by executing the current policy. A *`IncrApprenticeshipLearner`* module is create.

args : tuple, optional

Positional arguments passed to the class of the given type for initialization.

kwargs : dict, optional

Non-positional arguments passed to the class of the given type for initialization.

Returns ILearner :

A learner instance of the given type.

mlpy.learners.ILearner

class `mlpy.learners.`**`ILearner`**`(`*filename=None*`)`
Bases: *`mlpy.modules.UniqueModule`*

The learner interface.

Both online and offline learner inherit from this interface.

Parameters **filename** : str, optional

The name of the file to save the learner state to after each iteration. If None is given, the learner state is not saved. Default is None.

Attributes

mid	The module's unique identifier.
type	The type of the learner (i.e., *online* and *offline*).

mlpy.learners.ILearner.mid

`ILearner.`**`mid`**

The module's unique identifier.

Returns str :

The module's unique identifier

mlpy.learners.ILearner.type

`ILearner.`**`type`**

The type of the learner (i.e., *online* and *offline*).

During online learning the learning is performed during the episode or iteration, while offline learner do not perform the learning step until the end of the episode or iteration.

This property must be overwritten by its deriving class.

Returns str :

The type. Values can be either *online* or *offline*.

Raises NotImplementedError

If the child class does not implement this function.

Methods

choose_action(state)	Choose the next action
execute(experience)	Execute learning specific updates.
learn()	Learn a policy from the experience.
load(filename)	Load the state of the module from file.
reset(t, **kwargs)	Reset reinforcement learner.
save(filename)	Save the current state of the module to file.

mlpy.learners.ILearner.choose_action

`ILearner.`**`choose_action`**`(`*state*`)`

Choose the next action

The next action is chosen according to the current policy and the selected exploration strategy.

Parameters state : State

The current state.

Returns Action :

The chosen action.

Raises NotImplementedError

If the child class does not implement this function.

mlpy.learners.ILearner.execute

`ILearner.`**`execute`**`(`*experience*`)`

Execute learning specific updates.

Learning specific updates are performed, e.g. model updates.

Parameters experience : Experience

The actor's current experience consisting of previous state, the action performed in that state, the current state, and the reward awarded.

Raises NotImplementedError

If the child class does not implement this function.

mlpy.learners.ILearner.learn

`ILearner.`**`learn`**`()`

Learn a policy from the experience.

Perform the learning step to derive a new policy taking the latest experience into account.

Parameters experience : Experience

The agent's experience consisting of the previous state, the action performed in that state, the current state and the reward awarded.

Raises NotImplementedError

If the child class does not implement this function.

mlpy.learners.ILearner.load

`ILearner.`**`load`**`(`*filename*`)`

Load the state of the module from file.

Parameters filename : str

The name of the file to load from.

Notes

This is a class method, it can be accessed without instantiation.

mlpy.learners.ILearner.reset

`ILearner.`**`reset`**`(`*t*, ***kwargs*`)`

Reset reinforcement learner.

Reset the learner before start of a new episode or iteration and save the state of the learner to file.

Parameters t : float

The current time (sec)

kwargs : dict, optional

Non-positional parameters, optional.

mlpy.learners.ILearner.save

`ILearner.`**`save`**(*filename*)

Save the current state of the module to file.

Parameters **filename** : str

The name of the file to save to.

Online learners (`mlpy.learners.online`)

`IOnlineLearner`	The online learner base class.

mlpy.learners.online.IOnlineLearner

class `mlpy.learners.online.`**`IOnlineLearner`**(*filename=None*)

Bases: *`mlpy.learners.ILearner`*

The online learner base class.

The learning step is performed during the episode or iteration after each step.

Parameters **filename** : str, optional

The name of the file to save the learner state to after each iteration. If None is given, the learner state is not saved. Default is None.

Attributes

`mid`	The module's unique identifier.
`type`	This learner is of type *online*.

mlpy.learners.online.IOnlineLearner.mid

`IOnlineLearner.`**`mid`**

The module's unique identifier.

Returns str :

The module's unique identifier

mlpy.learners.online.IOnlineLearner.type

`IOnlineLearner.`**`type`**

This learner is of type *online*.

Returns str :

The learner type

Methods

`choose_action`(state)	Choose the next action
`execute`(experience)	Execute learning specific updates.
`learn`()	Learn a policy from the experience.
`load`(filename)	Load the state of the module from file.
`reset`(t, **kwargs)	Reset reinforcement learner.
`save`(filename)	Save the current state of the module to file.

mlpy.learners.online.IOnlineLearner.choose_action

`IOnlineLearner.`**`choose_action`**`(`*state*`)`

Choose the next action

The next action is chosen according to the current policy and the selected exploration strategy.

Parameters **state** : State

The current state.

Returns Action :

The chosen action.

Raises **NotImplementedError**

If the child class does not implement this function.

mlpy.learners.online.IOnlineLearner.execute

`IOnlineLearner.`**`execute`**`(`*experience*`)`

Execute learning specific updates.

Learning specific updates are performed, e.g. model updates.

Parameters **experience** : Experience

The actor's current experience consisting of previous state, the action performed in that state, the current state, and the reward awarded.

Raises **NotImplementedError**

If the child class does not implement this function.

mlpy.learners.online.IOnlineLearner.learn

`IOnlineLearner.`**`learn`**`()`

Learn a policy from the experience.

Perform the learning step to derive a new policy taking the latest experience into account.

Parameters **experience** : Experience

The agent's experience consisting of the previous state, the action performed in that state, the current state and the reward awarded.

Raises NotImplementedError

If the child class does not implement this function.

mlpy.learners.online.IOnlineLearner.load

`IOnlineLearner.`**`load`**`(`*filename*`)`

Load the state of the module from file.

Parameters filename : str

The name of the file to load from.

Notes

This is a class method, it can be accessed without instantiation.

mlpy.learners.online.IOnlineLearner.reset

`IOnlineLearner.`**`reset`**`(`*t*, ***kwargs*`)`

Reset reinforcement learner.

Reset the learner before start of a new episode or iteration and save the state of the learner to file.

Parameters t : float

The current time (sec)

kwargs : dict, optional

Non-positional parameters, optional.

mlpy.learners.online.IOnlineLearner.save

`IOnlineLearner.`**`save`**`(`*filename*`)`

Save the current state of the module to file.

Parameters filename : str

The name of the file to save to.

Reinforcement learning

RLLearner	The reinforcement learning learner interface.
QLearner	Performs q-learning.
RLDTLearner	Performs reinforcement learning using decision trees.

mlpy.learners.online.rl.RLLearner

class `mlpy.learners.online.rl.`**`RLLearner`**`(`*max_steps=None*, *filename=None*, *profile=False*`)`

Bases: *mlpy.learners.online.IOnlineLearner*

The reinforcement learning learner interface.

Parameters **max_steps** : int, optional

The maximum number of steps in an iteration. Default is 100.

filename : str, optional

The name of the file to save the learner state to after each iteration. If None is given, the learner state is not saved. Default is None.

profile : bool, optional

Turn on profiling at which point profiling data is collected and saved to a text file. Default is False.

Attributes

`mid`	The module's unique identifier.
`type`	This learner is of type *online*.

mlpy.learners.online.rl.RLLearner.mid

`RLLearner.`**`mid`**

The module's unique identifier.

Returns str :

The module's unique identifier

mlpy.learners.online.rl.RLLearner.type

`RLLearner.`**`type`**

This learner is of type *online*.

Returns str :

The learner type

Methods

`choose_action`(state)	Choose the next action
`execute`(experience)	Execute learning specific updates.
`learn`([experience])	Learn a policy from the experience.
`load`(filename)	Load the state of the module from file.
`reset`(t, **kwargs)	Reset reinforcement learner.
`save`(filename)	Save the learners state.

mlpy.learners.online.rl.RLLearner.choose_action

`RLLearner.`**`choose_action`**(*state*)

Choose the next action

The next action is chosen according to the current policy and the selected exploration strategy.

Parameters **state** : State

The current state.

Returns Action :

The chosen action.

Raises **NotImplementedError**

If the child class does not implement this function.

mlpy.learners.online.rl.RLLearner.execute

`RLLearner.`**`execute`**`(`*experience*`)`

Execute learning specific updates.

Learning specific updates are performed, e.g. model updates.

Parameters **experience** : Experience

The actor's current experience consisting of previous state, the action performed in that state, the current state, and the reward awarded.

Raises **NotImplementedError**

If the child class does not implement this function.

mlpy.learners.online.rl.RLLearner.learn

`RLLearner.`**`learn`**`(`*experience=None*`)`

Learn a policy from the experience.

Parameters **experience** : Experience

The agent's experience consisting of the previous state, the action performed in that state, the current state and the reward awarded.

mlpy.learners.online.rl.RLLearner.load

`RLLearner.`**`load`**`(`*filename*`)`

Load the state of the module from file.

Parameters **filename** : str

The name of the file to load from.

Notes

This is a class method, it can be accessed without instantiation.

mlpy.learners.online.rl.RLLearner.reset

`RLLearner.`**`reset`**(*t*, ***kwargs*)

Reset reinforcement learner.

Parameters **t** : float

The current time (sec)

kwargs : dict, optional

Non-positional parameters, optional.

mlpy.learners.online.rl.RLLearner.save

`RLLearner.`**`save`**(*filename*)

Save the learners state.

If profiling is turned on, profile information is saved to a *txt* file with the same name.

Parameters **filename** : str

The filename to save the information to.

mlpy.learners.online.rl.QLearner

class `mlpy.learners.online.rl.`**`QLearner`**(*explorer=None*, *max_steps=None*, *alpha=None*, *gamma=None*, *filename=None*, *profile=False*)

Bases: *`mlpy.learners.online.rl.RLLearner`*

Performs q-learning.

Q-learning is a reinforcement learning variant.

Parameters **explorer** : Explorer, optional

The exploration strategy used. Default is no exploration.

max_steps : int, optional

The maximum number of steps in an iteration. Default is 100

alpha : float, optional

The learning rate. Default is 0.5.

gamma : float, optional

The discounting factor. Default is 0.9.

filename : str, optional

The name of the file to save the learner state to after each iteration. If None is given, the learner state is not saved. Default is None.

profile : bool, optional

Turn on profiling at which point profiling data is collected and saved to a text file. Default is False.

Attributes

mid	The module's unique identifier.
type	This learner is of type *online*.

mlpy.learners.online.rl.QLearner.mid

`QLearner.`**`mid`**

The module's unique identifier.

Returns str :

The module's unique identifier

mlpy.learners.online.rl.QLearner.type

`QLearner.`**`type`**

This learner is of type *online*.

Returns str :

The learner type

Methods

choose_action(state)	Choose the next action
execute(experience)	Execute learning specific updates.
learn([experience])	Learn a policy from the experience.
load(filename)	Load the state of the module from file.
reset(t, **kwargs)	Reset reinforcement learner.
save(filename)	Save the learners state.

mlpy.learners.online.rl.QLearner.choose_action

`QLearner.`**`choose_action`**(*state*)

Choose the next action

The next action is chosen according to the current policy and the selected exploration strategy.

Parameters **state** : State

The current state.

Returns Action :

The chosen action.

mlpy.learners.online.rl.QLearner.execute

`QLearner.`**`execute`**(*experience*)

Execute learning specific updates.

Learning specific updates are performed, e.g. model updates.

Parameters **experience** : Experience

The actor's current experience consisting of previous state, the action performed in that state, the current state, and the reward awarded.

mlpy.learners.online.rl.QLearner.learn

`QLearner.`**`learn`**`(`*experience=None*`)`

Learn a policy from the experience.

By updating the Q table according to the experience a policy is learned.

Parameters **experience** : Experience

The actor's current experience consisting of previous state, the action performed in that state, the current state, and the reward awarded.

mlpy.learners.online.rl.QLearner.load

`QLearner.`**`load`**`(`*filename*`)`

Load the state of the module from file.

Parameters **filename** : str

The name of the file to load from.

Notes

This is a class method, it can be accessed without instantiation.

mlpy.learners.online.rl.QLearner.reset

`QLearner.`**`reset`**`(`*t*, ***kwargs*`)`

Reset reinforcement learner.

Parameters **t** : float

The current time (sec)

kwargs : dict, optional

Non-positional parameters, optional.

mlpy.learners.online.rl.QLearner.save

`QLearner.`**`save`**`(`*filename*`)`

Save the learners state.

If profiling is turned on, profile information is saved to a *txt* file with the same name.

Parameters **filename** : str

The filename to save the information to.

mlpy.learners.online.rl.RLDTLearner

class `mlpy.learners.online.rl.`**`RLDTLearner`**(*planner*, *max_steps=None*, *filename=None*, *profile=False*)

Bases: *`mlpy.learners.online.rl.RLLearner`*

Performs reinforcement learning using decision trees.

Reinforcement learning using decision trees (RL-DT) use decision trees to build the transition and reward models as described by Todd Hester and Peter Stone *[R3]*.

Parameters **planner** : IPlanner

The planner to use to determine the best action.

max_steps : int, optional

The maximum number of steps in an iteration. Default is 100.

filename : str, optional

The name of the file to save the learner state to after each iteration. If None is given, the learner state is not saved. Default is None.

profile : bool, optional

Turn on profiling at which point profiling data is collected and saved to a text file. Default is False.

References

[R3]

Attributes

`mid`	The module's unique identifier.
`type`	This learner is of type *online*.

mlpy.learners.online.rl.RLDTLearner.mid

`RLDTLearner.`**`mid`**

The module's unique identifier.

Returns str :

The module's unique identifier

mlpy.learners.online.rl.RLDTLearner.type

`RLDTLearner.`**`type`**

This learner is of type *online*.

Returns str :

The learner type

Methods

choose_action(state)	Choose the next action
execute(experience)	Execute learning specific updates.
learn([experience])	Learn a policy from the experience.
load(filename)	Load the state of the module from file.
reset(t, **kwargs)	Reset reinforcement learner.
save(filename)	Save the learners state.

mlpy.learners.online.rl.RLDTLearner.choose_action

RLDTLearner.**choose_action**(*state*)

Choose the next action

The next action is chosen according to the current policy and the selected exploration strategy.

Parameters **state** : State

The current state.

Returns Action :

The chosen action.

mlpy.learners.online.rl.RLDTLearner.execute

RLDTLearner.**execute**(*experience*)

Execute learning specific updates.

Learning specific updates are performed, e.g. model updates.

Parameters **experience** : Experience

The actor's current experience consisting of previous state, the action performed in that state, the current state, and the reward awarded.

mlpy.learners.online.rl.RLDTLearner.learn

RLDTLearner.**learn**(*experience=None*)

Learn a policy from the experience.

A policy is learned from the experience by building the MDP model.

Parameters **experience** : Experience

The actor's current experience consisting of previous state, the action performed in that state, the current state, and the reward awarded.

mlpy.learners.online.rl.RLDTLearner.load

RLDTLearner.**load**(*filename*)

Load the state of the module from file.

Parameters **filename** : str

The name of the file to load from.

Notes

This is a class method, it can be accessed without instantiation.

mlpy.learners.online.rl.RLDTLearner.reset

`RLDTLearner.`**`reset`**`(`*t*, ***kwargs*`)`

Reset reinforcement learner.

Parameters **t** : float

The current time (sec)

kwargs : dict, optional

Non-positional parameters, optional.

mlpy.learners.online.rl.RLDTLearner.save

`RLDTLearner.`**`save`**`(`*filename*`)`

Save the learners state.

If profiling is turned on, profile information is saved to a *txt* file with the same name.

Parameters **filename** : str

The filename to save the information to.

Offline learners (`mlpy.learners.offline`)

IOfflineLearner	The offline learner base class.

mlpy.learners.offline.IOfflineLearner

class `mlpy.learners.offline.`**`IOfflineLearner`**`(`*filename=None*`)`

Bases: *mlpy.learners.ILearner*

The offline learner base class.

In offline learning the learning step is performed at the end of the episode or iteration.

Parameters **filename** : str, optional

The name of the file to save the learner state to after each iteration. If None is given, the learner state is not saved. Default is None.

Attributes

mid	The module's unique identifier.
type	This learner is of type *offline*.

mlpy.learners.offline.IOfflineLearner.mid

`IOfflineLearner.`**`mid`**

The module's unique identifier.

Returns str :

The module's unique identifier

mlpy.learners.offline.IOfflineLearner.type

`IOfflineLearner.`**`type`**

This learner is of type *offline*.

Returns str :

The learner type

Methods

`choose_action`(state)	Choose the next action
`execute`(experience)	Execute learning specific updates.
`learn`()	Learn a policy from the experience.
`load`(filename)	Load the state of the module from file.
`reset`(t, **kwargs)	Reset reinforcement learner.
`save`(filename)	Save the current state of the module to file.

mlpy.learners.offline.IOfflineLearner.choose_action

`IOfflineLearner.`**`choose_action`**(*state*)

Choose the next action

The next action is chosen according to the current policy and the selected exploration strategy.

Parameters **state** : State

The current state.

Returns Action :

The chosen action.

Raises NotImplementedError

If the child class does not implement this function.

mlpy.learners.offline.IOfflineLearner.execute

`IOfflineLearner.`**`execute`**(*experience*)

Execute learning specific updates.

Learning specific updates are performed, e.g. model updates.

Parameters **experience** : Experience

The actor's current experience consisting of previous state, the action performed in that state, the current state, and the reward awarded.

Raises **NotImplementedError**

If the child class does not implement this function.

mlpy.learners.offline.IOfflineLearner.learn

`IOfflineLearner.`**`learn`**`()`

Learn a policy from the experience.

Perform the learning step to derive a new policy taking the latest experience into account.

Parameters **experience** : Experience

The agent's experience consisting of the previous state, the action performed in that state, the current state and the reward awarded.

Raises **NotImplementedError**

If the child class does not implement this function.

mlpy.learners.offline.IOfflineLearner.load

`IOfflineLearner.`**`load`**`(`*filename*`)`

Load the state of the module from file.

Parameters **filename** : str

The name of the file to load from.

Notes

This is a class method, it can be accessed without instantiation.

mlpy.learners.offline.IOfflineLearner.reset

`IOfflineLearner.`**`reset`**`(`*t*, ***kwargs*`)`

Reset reinforcement learner.

Reset the learner before start of a new episode or iteration and save the state of the learner to file.

Parameters **t** : float

The current time (sec)

kwargs : dict, optional

Non-positional parameters, optional.

mlpy.learners.offline.IOfflineLearner.save

`IOfflineLearner.`**`save`**`(`*filename*`)`

Save the current state of the module to file.

Parameters **filename** : str

The name of the file to save to.

Inverse reinforcement learning

`ApprenticeshipLearner`	The apprenticeship learner.
`IncrApprenticeshipLearner`	Incremental apprenticeship learner.

mlpy.learners.offline.irl.ApprenticeshipLearner

class `mlpy.learners.offline.irl.`**`ApprenticeshipLearner`**(*obs*, *planner*, *method=None*, *max_iter=None*, *thresh=None*, *gamma=None*, *nsamples=None*, *max_steps=None*, *filename=None*, ***kwargs*)

Bases: `mlpy.learners.offline.IOfflineLearner`

The apprenticeship learner.

The apprenticeship learner is an inverse reinforcement learner, a method introduced by Abbeel and Ng [R2] which strives to imitate the demonstrations given by an expert.

Parameters **obs** : array_like, shape (*n*, *nfeatures*, *ni*)

List of trajectories provided by demonstrator, which the learner is trying to emulate, where *n* is the number of sequences, *ni* is the length of the i_th demonstration, and each demonstration has *nfeatures* features.

planner : IPlanner

The planner to use to determine the best action.

method : {'projection', 'maxmargin'}, optional

The IRL method to employ. Default is *projection*.

max_iter : int, optional

The maximum number of iteration after which learning will be terminated. It is assumed that a policy close enough to the experts demonstrations was found. Default is *inf*.

thresh : float, optional

The learning is considered having converged to the demonstrations once the threshold has been reach. Default is *eps*.

gamma : float, optional

The discount factor. Default is 0.9.

nsamples : int, optional

The number of samples taken during Monte Carlo sampling. Default is 100.

max_steps : int, optional

The maximum number of steps in an iteration (during MonteCarlo sampling). Default is 100.

filename : str, optional

The name of the file to save the learner state to after each iteration. If None is given, the learner state is not saved. Default is None.

Other Parameters **mix_policies** : bool

Whether to create a new policy by mixing from policies seen so far or by considering the best valued action. Default is False.

rescale : bool

If set to True, the feature expectations are rescaled to be between 0 and 1. Default is False.

visualize : bool

Visualize each iteration of the IRL step if set to True. Default is False.

See also:

IncrApprenticeshipLearner

Notes

Method **maxmargin** using a QP solver to solve the following equation:

$$\begin{aligned} \underset{t,w}{\text{maximize}} \quad & t \\ \text{subject to} \quad & w^T \mu_E > w^T \mu^{(j)} + t, j = 0, \ldots, i-1 \\ & ||w||_2 \leq 1. \end{aligned}$$

and mixing policies is realized by solving the quadratic problem:

$$\begin{aligned} \text{minimize} \quad & ||\mu_E - \mu||_2 \\ \text{subject to} \quad & \mu = \sum_i (\lambda_i \mu^{(i)}) \\ & \lambda_i \geq 0 \\ & \sum_i \lambda_i = 1 \end{aligned}$$

The QP solver used for the implementation is the IBM ILOG CPLEX Optimizer which requires a separate license. If you are unable to obtain a license, the 'projection' method can be used instead.

References

[R2]

Attributes

`mid`	The module's unique identifier.
`type`	This learner is of type *offline*.

mlpy.learners.offline.irl.ApprenticeshipLearner.mid

`ApprenticeshipLearner.`**`mid`**

The module's unique identifier.

Returns str :

The module's unique identifier

mlpy.learners.offline.irl.ApprenticeshipLearner.type

`ApprenticeshipLearner.`**`type`**

This learner is of type *offline*.

Returns str :

The learner type

Methods

choose_action(state)	Choose the next action
execute(experience)	Execute learning specific updates.
learn()	Learn the optimal policy via apprenticeship learning.
load(filename)	Load the state of the module from file.
reset(t, **kwargs)	Reset reinforcement learner.
save(filename)	Save the current state of the module to file.

mlpy.learners.offline.irl.ApprenticeshipLearner.choose_action

`ApprenticeshipLearner.`**`choose_action`**(*state*)

Choose the next action

The next action is chosen according to the current policy and the selected exploration strategy.

Parameters **state** : State

The current state.

Returns Action :

The chosen action.

Raises **NotImplementedError**

If the child class does not implement this function.

mlpy.learners.offline.irl.ApprenticeshipLearner.execute

`ApprenticeshipLearner.`**`execute`**(*experience*)

Execute learning specific updates.

Learning specific updates are performed, e.g. model updates.

Parameters **experience** : Experience

The actor's current experience consisting of previous state, the action performed in that state, the current state, and the reward awarded.

Raises NotImplementedError

If the child class does not implement this function.

mlpy.learners.offline.irl.ApprenticeshipLearner.learn

`ApprenticeshipLearner.`**`learn`**`()`

Learn the optimal policy via apprenticeship learning.

The apprenticeship learning algorithm for finding a policy $\tilde{\pi}$, that induces feature expectations $\mu(\tilde{\pi})$ close to μ_E is as follows:

1. Randomly pick some policy $\pi^{(0)}$, compute (or approximate via Monte Carlo) $\mu^{(0)} = \mu(\pi^{(0)})$, and set $i = 1$.
2. Compute $t^{(i)} = \max_{w:||w||_2 \leq 1} \min_{j \in 0 \ldots (i-1)} w^T(\mu_E = \mu^{(j)})$, and let $w^{(i)}$ be the value of w that attains this maximum. This can be achieved by either the **max-margin** method or by the **projection** method.
3. If $t^{(i)} \leq \epsilon$, then terminate.
4. Using the RL algorithm, compute the optimal policy $\pi^{(i)}$ for the MDP using rewards $R = (w^{(i)})^T \phi$.
5. Compute (or estimate) $\mu^{(i)} = \mu(\pi^{(i)})$.
6. Set $i = i + 1$, and go back to step 2.

mlpy.learners.offline.irl.ApprenticeshipLearner.load

`ApprenticeshipLearner.`**`load`**`(`*filename*`)`

Load the state of the module from file.

Parameters filename : str

The name of the file to load from.

Notes

This is a class method, it can be accessed without instantiation.

mlpy.learners.offline.irl.ApprenticeshipLearner.reset

`ApprenticeshipLearner.`**`reset`**`(`*t*, ***kwargs*`)`

Reset reinforcement learner.

Reset the learner before start of a new episode or iteration and save the state of the learner to file.

Parameters t : float

The current time (sec)

kwargs : dict, optional

Non-positional parameters, optional.

mlpy.learners.offline.irl.ApprenticeshipLearner.save

`ApprenticeshipLearner.save`(*filename*)

Save the current state of the module to file.

Parameters **filename** : str

The name of the file to save to.

mlpy.learners.offline.irl.IncrApprenticeshipLearner

class `mlpy.learners.offline.irl.IncrApprenticeshipLearner`(*obs*, *planner*, *method=None*, *max_iter=None*, *thresh=None*, *gamma=None*, *nsamples=None*, *max_steps=None*, *filename=None*, ***kwargs*)

Bases: *`mlpy.learners.offline.irl.ApprenticeshipLearner`*

Incremental apprenticeship learner.

The model under which the apprenticeship is operating is updated incrementally while learning a policy that emulates the expert's demonstrations.

Parameters **obs** : array_like, shape (*n*, *nfeatures*, *ni*)

List of trajectories provided by demonstrator, which the learner is trying to emulate, where *n* is the number of sequences, *ni* is the length of the i_th demonstration, and each demonstration has *nfeatures* features.

planner : IPlanner

The planner to use to determine the best action.

method : {'projection', 'maxmargin'}, optional

The IRL method to employ. Default is *projection*.

max_iter : int, optional

The maximum number of iteration after which learning will be terminated. It is assumed that a policy close enough to the experts demonstrations was found. Default is *inf*.

thresh : float, optional

The learning is considered having converged to the demonstrations once the threshold has been reach. Default is *eps*.

gamma : float, optional

The discount factor. Default is 0.9.

nsamples : int, optional

The number of samples taken during Monte Carlo sampling. Default is 100.

max_steps : int, optional

The maximum number of steps in an iteration (during MonteCarlo sampling). Default is 100.

filename : str, optional

The name of the file to save the learner state to after each iteration. If None is given, the learner state is not saved. Default is None.

Other Parameters **mix_policies** : bool

Whether to create a new policy by mixing from policies seen so far or by considering the best valued action. Default is False.

rescale : bool

If set to True, the feature expectations are rescaled to be between 0 and 1. Default is False.

visualize : bool

Visualize each iteration of the IRL step if set to True. Default is False.

See also:

`ApprenticeshipLearner`

Notes

Inverse reinforcement learning assumes knowledge of the underlying model. However, this is not always feasible. The incremental apprenticeship learner updates its model after every iteration by executing the current policy. Thus, it provides an extension to the original apprenticeship learner.

Attributes

`mid`	The module's unique identifier.
`type`	This learner is of type *offline*.

mlpy.learners.offline.irl.IncrApprenticeshipLearner.mid

`IncrApprenticeshipLearner.`**`mid`**

The module's unique identifier.

Returns str :

The module's unique identifier

mlpy.learners.offline.irl.IncrApprenticeshipLearner.type

`IncrApprenticeshipLearner.`**`type`**

This learner is of type *offline*.

Returns str :

The learner type

Methods

`choose_action`(state)	Choose the next action
`execute`(experience)	Execute learning specific updates.
`learn`()	Learn a policy from the experience.
`load`(filename)	Load the state of the module from file.
`reset`(t, **kwargs)	Reset the apprenticeship learner.
`save`(filename)	Save the current state of the module to file.

mlpy.learners.offline.irl.IncrApprenticeshipLearner.choose_action

`IncrApprenticeshipLearner.`**`choose_action`**`(`*state*`)`

Choose the next action

The next action is chosen according to the current policy and the selected exploration strategy.

Parameters **state** : State

The current state.

Returns Action :

The chosen action.

mlpy.learners.offline.irl.IncrApprenticeshipLearner.execute

`IncrApprenticeshipLearner.`**`execute`**`(`*experience*`)`

Execute learning specific updates.

Learning specific updates are performed, e.g. model updates.

Parameters **experience** : Experience

The actor's current experience consisting of previous state, the action performed in that state, the current state, and the reward awarded.

mlpy.learners.offline.irl.IncrApprenticeshipLearner.learn

`IncrApprenticeshipLearner.`**`learn`**`()`

Learn a policy from the experience.

Learn the optimal policy using an apprenticeship learning algorithm incrementally.

Returns bool :

Whether the found policy is considered to have converged. The algorithm is considered to have converged on the optimal policy if either the performance is within a certain threshold or if the maximum number of iterations has been reached.

mlpy.learners.offline.irl.IncrApprenticeshipLearner.load

`IncrApprenticeshipLearner.`**`load`**`(`*filename*`)`

Load the state of the module from file.

Parameters **filename** : str

The name of the file to load from.

Notes

This is a class method, it can be accessed without instantiation.

mlpy.learners.offline.irl.IncrApprenticeshipLearner.reset

`IncrApprenticeshipLearner.`**`reset`**(*t*, ***kwargs*)

Reset the apprenticeship learner.

Parameters **t** : float

The current time (sec)

kwargs : dict, optional

Non-positional parameters, optional.

mlpy.learners.offline.irl.IncrApprenticeshipLearner.save

`IncrApprenticeshipLearner.`**`save`**(*filename*)

Save the current state of the module to file.

Parameters **filename** : str

The name of the file to save to.

CHAPTER 16

Markov decision process (MDP) (`mlpy.mdp`)

Transition and reward models

MDPModelFactory	The Markov decision process (MDP) model factory.
IMDPModel	The Markov decision process interface.

mlpy.mdp.MDPModelFactory

class `mlpy.mdp.`**`MDPModelFactory`**

Bases: `object`

The Markov decision process (MDP) model factory.

An instance of an MDP model can be created by passing the MDP model type.

Examples

```
>>> from mlpy.mdp import MDPModelFactory
>>> MDPModelFactory.create('discretemodel')
```

This creates a *DiscreteModel* instance with default settings.

```
>>> MDPModelFactory.create('decisiontreemodel', explorer_type=
↪'leastvisitedbonusexplorer',
...                          explorer_params={'rmax': 1.0})
```

This creates a *DecisionTreeModel* instance using *LeastVisitedBonusExplorer* with *rmax* set to 1.0.

Methods

create(_type, *args, **kwargs)	Create an MDP model of the given type.

mlpy.mdp.MDPModelFactory.create

static `MDPModelFactory.`**`create`**(*_type*, **args*, ***kwargs*)
Create an MDP model of the given type.

Parameters **_type** : str

The MDP model type. Valid model types:

discretemodel A model for discrete state and actions deriving transition and reward information from empirical data. A *DiscreteModel* instance is created.

decisiontreemodel A model for discrete state and actions deriving transition and reward information from empirical data generalized using decision trees. A *DecisionTreeModel* instance model is created.

casml A model for continuous state and actions deriving transition information from empirical data fit to a case base and a Hidden Markov Model (*HMM*). Rewards are derived from empirical data. A *CASML* instance is created.

args : tuple, optional

Positional arguments to pass to the class of the given type for initialization.

kwargs : dict, optional

Non-positional arguments to pass to the class of the given type for initialization.

Returns IMDPModel :

A MDP model instance of the given type.

mlpy.mdp.IMDPModel

class `mlpy.mdp.`**`IMDPModel`**(*proba_calc_method=None*)
Bases: *mlpy.modules.UniqueModule*

The Markov decision process interface.

All Markov decision process (MDP) models are derived from the base class. The base class maintains an initial probability distribution from which the initial state can be sampled.

Parameters **proba_calc_method** : str

The method used to calculate the probability distribution for the initial state. Defaults to DefaultProbaCalcMethod.

Attributes

mid	The module's unique identifier.

mlpy.mdp.IMDPModel.mid

`IMDPModel.`**`mid`**

The module's unique identifier.

Returns str :

The module's unique identifier

Methods

`fit`(obs, actions, **kwargs)	Fit the model to the observations and actions of the trajectory.
`load`(filename)	Load the state of the module from file.
`predict_proba`(state, action)	Predict the probability distribution.
`sample`([state, action])	Sample from the probability distribution.
`save`(filename)	Save the current state of the module to file.
`update`(experience)	Update the model with the agent's experience.

mlpy.mdp.IMDPModel.fit

`IMDPModel.`**`fit`**(*obs*, *actions*, ***kwargs*)

Fit the model to the observations and actions of the trajectory.

Parameters **obs** : array_like, shape (*nfeatures*, *n*)

Trajectory of observations, where each observation has *nfeatures* features and *n* is the length of the trajectory.

actions : array_like, shape (*nfeatures*, *n*)

Trajectory of actions, where each action has *nfeatures* features and *n* is the length of the trajectory.

kwargs: dict, optional

Non-positional parameters, optional

Raises **NotImplementedError**

If the child class does not implement this function.

Notes

This is an abstract method and *must* be implemented by its deriving class.

mlpy.mdp.IMDPModel.load

`IMDPModel.`**`load`**(*filename*)

Load the state of the module from file.

Parameters **filename** : str

The name of the file to load from.

Notes

This is a class method, it can be accessed without instantiation.

mlpy.mdp.IMDPModel.predict_proba

`IMDPModel.`**`predict_proba`**`(`*state*, *action*`)`

Predict the probability distribution.

The probability distribution for state transitions is predicted for the given state and an action.

Parameters **state** : State

The current state the robot is in.

action : Action

The action perform in state *state*.

Returns dict[tuple[float], float] :

The probability distribution for the state-action pair.

Raises **NotImplementedError**

If the child class does not implement this function.

Notes

This is an abstract method and *must* be implemented by its deriving class.

mlpy.mdp.IMDPModel.sample

`IMDPModel.`**`sample`**`(`*state=None*, *action=None*`)`

Sample from the probability distribution.

The next state is sampled for the given state and action from the probability distribution. If either state or action is `None` the next state is sampled from the initial distribution.

Parameters **state** : State, optional

The current state the robot is in.

action : Action, optional

The action perform in state *state*.

Returns State :

The sampled next state.

mlpy.mdp.IMDPModel.save

`IMDPModel.`**`save`**`(`*filename*`)`

Save the current state of the module to file.

Parameters **filename** : str

The name of the file to save to.

mlpy.mdp.IMDPModel.update

IMDPModel.**update**(*experience*)

Update the model with the agent's experience.

Parameters experience : Experience

The agent's experience, consisting of state, action, next state(, and reward).

Returns bool :

Return True if the model has changed, False otherwise.

Raises NotImplementedError

If the child class does not implement this function.

Notes

Optionally this method can be overwritten if the model supports incrementally updating the model.

Discrete models

DiscreteModel	The MDP model for discrete states and actions.
DecisionTreeModel	The MDP model for discrete states and actions realized with decision trees.

mlpy.mdp.discrete.DiscreteModel

class mlpy.mdp.discrete.**DiscreteModel**(*actions=None*, ***kwargs*)

Bases: *mlpy.mdp.IMDPModel*

The MDP model for discrete states and actions.

Parameters actions : list[Action] or dict[State, list[Action]

The available actions. If not given, the actions are read from the Action description.

Attributes

mid	The module's unique identifier.
statespace	Collection of states and their state-action information.

mlpy.mdp.discrete.DiscreteModel.mid

DiscreteModel.**mid**

The module's unique identifier.

Returns str :

The module's unique identifier

mlpy.mdp.discrete.DiscreteModel.statespace

`DiscreteModel.`**`statespace`**

Collection of states and their state-action information.

Returns dict[State, StateData] :

The state space.

Methods

add_state(state)	Add a new state to the statespace.
fit(obs, actions[, labels])	Fit the model to the observations and actions of the trajectory.
get_actions([state])	Retrieve the available actions for the given state.
load(filename)	Load the state of the module from file.
predict_proba(state, action)	Predict the probability distribution.
print_rewards()	Print the state rewards for debugging purposes.
print_transitions()	Print the state transitions for debugging purposes.
sample([state, action])	Sample from the probability distribution.
save(filename)	Save the current state of the module to file.
update([experience])	Update the model with the agent's experience.

mlpy.mdp.discrete.DiscreteModel.add_state

`DiscreteModel.`**`add_state`**(*state*)

Add a new state to the statespace.

Add a new state to the statespace (a collection of states that have already been seen).

Parameters **state** : State

The state to add to the state space.

Returns bool :

Whether the state was a new state or not.

mlpy.mdp.discrete.DiscreteModel.fit

`DiscreteModel.`**`fit`**(*obs*, *actions*, *labels=None*)

Fit the model to the observations and actions of the trajectory.

Parameters **obs** : array_like, shape (*nfeatures*, *n*)

Trajectory of observations, where each observation has *nfeatures* features and *n* is the length of the trajectory.

actions : array_like, shape (*nfeatures*, *n*)

Trajectory of actions, where each action has *nfeatures* features and *n* is the length of the trajectory.

labels : array_like, shape (*n*,)

Label identifying each step in the trajectory, where *n* is the length of the trajectory.

mlpy.mdp.discrete.DiscreteModel.get_actions

`DiscreteModel.`**`get_actions`**`(`*state=None*`)`

Retrieve the available actions for the given state.

Parameters **state** : State

The state for which to get the actions.

Returns list :

The actions that can be taken in this state.

mlpy.mdp.discrete.DiscreteModel.load

`DiscreteModel.`**`load`**`(`*filename*`)`

Load the state of the module from file.

Parameters **filename** : str

The name of the file to load from.

Notes

This is a class method, it can be accessed without instantiation.

mlpy.mdp.discrete.DiscreteModel.predict_proba

`DiscreteModel.`**`predict_proba`**`(`*state*, *action*`)`

Predict the probability distribution.

Predict the probability distribution for state transitions given a state and an action.

Parameters **state** : State

The current state the robot is in.

action : Action

The action perform in state *state*.

Returns dict[tuple[float]], float] :

The probability distribution for the state-action pair.

mlpy.mdp.discrete.DiscreteModel.print_rewards

`DiscreteModel.`**`print_rewards`**`()`

Print the state rewards for debugging purposes.

mlpy.mdp.discrete.DiscreteModel.print_transitions

`DiscreteModel.`**`print_transitions`**`()`

Print the state transitions for debugging purposes.

mlpy.mdp.discrete.DiscreteModel.sample

`DiscreteModel.`**`sample`**`(`*state=None*, *action=None*`)`

Sample from the probability distribution.

The next state is sampled for the given state and action from the probability distribution. If either state or action is `None` the next state is sampled from the initial distribution.

Parameters **state** : State, optional

The current state the robot is in.

action : Action, optional

The action perform in state *state*.

Returns State :

The sampled next state.

mlpy.mdp.discrete.DiscreteModel.save

`DiscreteModel.`**`save`**`(`*filename*`)`

Save the current state of the module to file.

Parameters **filename** : str

The name of the file to save to.

mlpy.mdp.discrete.DiscreteModel.update

`DiscreteModel.`**`update`**`(`*experience=None*`)`

Update the model with the agent's experience.

Parameters **experience** : Experience

The agent's experience, consisting of state, action, next state(, and reward).

Returns bool :

Return True if the model has changed, False otherwise.

mlpy.mdp.discrete.DecisionTreeModel

class `mlpy.mdp.discrete.`**`DecisionTreeModel`**`(`*actions=None*, *explorer_type=None*, *use_reward_trees=None*, **args*, ***kwargs*`)`

Bases: *`mlpy.mdp.discrete.DiscreteModel`*

The MDP model for discrete states and actions realized with decision trees.

The MDP model with decision trees is implemented as described by Todd Hester and Peter Stone *[R4]*. Transitions are learned for each feature; i.e. there is a decision tree for each state feature, and the predictions $P(x_i^r|s,a)$ for the `n` state features are combined to create a prediction of probabilities of the relative change of the state $s^r = \langle x_1^r, x_2^r, \ldots, x_n^r \rangle$ by calculating:

$$P(s^r|s,a) = \Pi_{i=0}^n P(x_i^r|s,a)$$

Optionally, the reward can also be learned by generating a decision tree for it.

The MDP model with decision trees can optionally specify an RMax based exploration model to drive exploration of unseen states.

Parameters **actions** : list[Action] | dict[State, list[Action]

The available actions. If not given, the actions are read from the Action description.

explorer_type : str

The type of exploration policy to perform. Valid explorer types:

unvisitedbonusexplorer: In unvisited-bonus exploration mode, if a state is experienced that has not been seen before the decision trees are considered to have changed and thus are being updated, otherwise, the decision trees are only considered to have changed based on the C45Tree algorithm.

leastvisitedbonusexplorer: In least-visited-bonus exploration mode, the states that have been visited the least are given a bonus of RMax. A `LeastVisitedBonusExplorer` instance is create.

unknownbonusexplorer: In unknown-bonus exploration mode states for which the decision tree was unable to predict a reward are considered unknown and are given a bonus of RMax. A `UnknownBonusExplorer` instance is create.

use_reward_trees : bool

If True, decision trees are used for the rewards model, otherwise a standard reward function is used.

args: tuple

Positional parameters passed to the model explorer.

kwargs: dict

Non-positional parameters passed to the model explorer.

Other Parameters **explorer_params** : dict

Parameters specific to the given exploration type.

Raises **ValueError**

If explorer type is not valid.

Notes

A C4.5 algorithm is used to generate the decision trees. The implementation of the algorithm that was improved to make the algorithm incremental. This is realized by checking at each node whether the new experience changes the optimal split and only rebuilds the the tree from that node if it does.

References

[R4]

Attributes

`mid`	The module's unique identifier.

Continued on next page

Table 16.8 – continued from previous page

statespace	Collection of states and their state-action information.

mlpy.mdp.discrete.DecisionTreeModel.mid

`DecisionTreeModel.`**`mid`**

The module's unique identifier.

Returns str :

The module's unique identifier

mlpy.mdp.discrete.DecisionTreeModel.statespace

`DecisionTreeModel.`**`statespace`**

Collection of states and their state-action information.

Returns dict[State, StateData] :

The state space.

Methods

activate_exploration()	Turn the explorer on.
add_state(state)	Add a new state to the statespace.
deactivate_exploration()	Turn the explorer off.
fit(obs, actions[, rewards])	Fit the model to the trajectory data.
get_actions([state])	Retrieve the available actions for the given state.
load(filename)	Load the state of the module from file.
predict_proba(state, action)	Predict the probability distribution.
print_rewards()	Print the state rewards for debugging purposes.
print_transitions()	Print the state transitions for debugging purposes.
sample([state, action])	Sample from the probability distribution.
save(filename)	Save the current state of the module to file.
update([experience])	Update the model with the agent's experience.

mlpy.mdp.discrete.DecisionTreeModel.activate_exploration

`DecisionTreeModel.`**`activate_exploration`**`()`

Turn the explorer on.

mlpy.mdp.discrete.DecisionTreeModel.add_state

`DecisionTreeModel.`**`add_state`**`(`*state*`)`

Add a new state to the statespace.

Add a new state to the statespace (a collection of states that have already been seen).

Parameters state : State

The state to add to the state space.

Returns bool :

Whether the state was a new state or not.

mlpy.mdp.discrete.DecisionTreeModel.deactivate_exploration

`DecisionTreeModel.`**`deactivate_exploration`**`()`

Turn the explorer off.

mlpy.mdp.discrete.DecisionTreeModel.fit

`DecisionTreeModel.`**`fit`**`(`*obs*, *actions*, *rewards=None*`)`

Fit the model to the trajectory data.

Parameters **obs** : array_like, shape (*nfeatures*, *n*)

Trajectory of observations, where each observation has *nfeatures* features and *n* is the length of the trajectory.

actions : array_like, shape (*nfeatures*, *n*)

Trajectory of actions, where each action has *nfeatures* features and *n* is the length of the trajectory.

rewards : array_like, shape (*n*,)

List of rewards, a reward is awarded for each observation.

mlpy.mdp.discrete.DecisionTreeModel.get_actions

`DecisionTreeModel.`**`get_actions`**`(`*state=None*`)`

Retrieve the available actions for the given state.

Parameters **state** : State

The state for which to get the actions.

Returns list :

The actions that can be taken in this state.

mlpy.mdp.discrete.DecisionTreeModel.load

`DecisionTreeModel.`**`load`**`(`*filename*`)`

Load the state of the module from file.

Parameters **filename** : str

The name of the file to load from.

Notes

This is a class method, it can be accessed without instantiation.

mlpy.mdp.discrete.DecisionTreeModel.predict_proba

`DecisionTreeModel.`**`predict_proba`**`(`*state*, *action*`)`

Predict the probability distribution.

Predict the probability distribution for state transitions given a state and an action.

Parameters **state** : State

The current state the robot is in.

action : Action

The action perform in state *state*.

Returns dict[tuple[float]], float] :

The probability distribution for the state-action pair.

mlpy.mdp.discrete.DecisionTreeModel.print_rewards

`DecisionTreeModel.`**`print_rewards`**`()`

Print the state rewards for debugging purposes.

mlpy.mdp.discrete.DecisionTreeModel.print_transitions

`DecisionTreeModel.`**`print_transitions`**`()`

Print the state transitions for debugging purposes.

mlpy.mdp.discrete.DecisionTreeModel.sample

`DecisionTreeModel.`**`sample`**`(`*state=None*, *action=None*`)`

Sample from the probability distribution.

The next state is sampled for the given state and action from the probability distribution. If either state or action is `None` the next state is sampled from the initial distribution.

Parameters **state** : State, optional

The current state the robot is in.

action : Action, optional

The action perform in state *state*.

Returns State :

The sampled next state.

mlpy.mdp.discrete.DecisionTreeModel.save

`DecisionTreeModel.`**`save`**`(`*filename*`)`

Save the current state of the module to file.

Parameters **filename** : str

The name of the file to save to.

mlpy.mdp.discrete.DecisionTreeModel.update

`DecisionTreeModel.`**`update`**`(`*experience=None*`)`

Update the model with the agent's experience.

The decision trees for transition and reward functions are being updated.

Parameters **experience** : Experience

The agent's experience, consisting of state, action, next state(, and reward).

Returns bool :

Return True if the model has changed, False otherwise.

Model explorer

ExplorerFactory	The model explorer factory.
RMaxExplorer	RMax based exploration base class.
LeastVisitedBonusExplorer	Least visited bonus explorer, a RMax based exploration model.
UnknownBonusExplorer	Unknown bonus explorer, a RMax based exploration model.

mlpy.mdp.discrete.ExplorerFactory

class `mlpy.mdp.discrete.`**`ExplorerFactory`**

Bases: `object`

The model explorer factory.

An instance of an explorer can be created by passing the explorer type.

Examples

```
>>> from mlpy.mdp.discrete import ExplorerFactory
>>> ExplorerFactory.create('unknownbonusexplorer', 1.0)
```

This creates a *UnknownBonusExplorer* with *rmax* set to 1.0.

Methods

create(_type, *args, **kwargs)	Create an MDP model of the given type.

mlpy.mdp.discrete.ExplorerFactory.create

static `ExplorerFactory.`**`create`**`(`*_type*, **args*, ***kwargs*`)`

Create an MDP model of the given type.

Parameters **_type** : str

The model explorer type. Valid model types:

leastvisitedbonusexplorer: In least-visited-bonus exploration mode, the states that have been visited the least are given a bonus of RMax. A `LeastVisitedBonusExplorer` instance is create.

unknownbonusexplorer: In unknown-bonus exploration mode states for which the decision tree was unable to predict a reward are considered unknown and are given a bonus of RMax. A `UnknownBonusExplorer` instance is create.

args : tuple, optional

Positional arguments to pass to the class of the given type for initialization.

kwargs : dict, optional

Non-positional arguments to pass to the class of the given type for initialization.

Returns RMaxExplorer :

An explorer instance of the given type.

mlpy.mdp.discrete.RMaxExplorer

class `mlpy.mdp.discrete.`**`RMaxExplorer`**(*rmax*)

Bases: `mlpy.modules.UniqueModule`

RMax based exploration base class.

Parameters rmax : float

The maximum achievable reward.

Attributes

`mid`	The module's unique identifier.

mlpy.mdp.discrete.RMaxExplorer.mid

`RMaxExplorer.`**`mid`**

The module's unique identifier.

Returns str :

The module's unique identifier

Methods

`activate`(*args, **kwargs)	Turn on exploration mode.
`deactivate`()	Turn off exploration mode.
`load`(filename)	Load the state of the module from file.
`save`(filename)	Save the current state of the module to file.
`update`(model)	Update the reward model according to a RMax based exploration policy.

mlpy.mdp.discrete.RMaxExplorer.activate

`RMaxExplorer.`**`activate`**`(`**args*, ***kwargs*`)`
Turn on exploration mode.

mlpy.mdp.discrete.RMaxExplorer.deactivate

`RMaxExplorer.`**`deactivate`**`()`
Turn off exploration mode.

mlpy.mdp.discrete.RMaxExplorer.load

`RMaxExplorer.`**`load`**`(`*filename*`)`
Load the state of the module from file.

Parameters **filename** : str

The name of the file to load from.

Notes

This is a class method, it can be accessed without instantiation.

mlpy.mdp.discrete.RMaxExplorer.save

`RMaxExplorer.`**`save`**`(`*filename*`)`
Save the current state of the module to file.

Parameters **filename** : str

The name of the file to save to.

mlpy.mdp.discrete.RMaxExplorer.update

`RMaxExplorer.`**`update`**`(`*model*`)`
Update the reward model according to a RMax based exploration policy.

Parameters **model** : StateActionInfo

The state-action information.

mlpy.mdp.discrete.LeastVisitedBonusExplorer

class `mlpy.mdp.discrete.`**`LeastVisitedBonusExplorer`**`(`*rmax*, *func*, *thresh=None*`)`
Bases: *`mlpy.mdp.discrete.RMaxExplorer`*

Least visited bonus explorer, a RMax based exploration model.

Least visited bonus exploration only goes into exploration mode whether it is predicted that only states with rewards less than a given threshold can be reached. Once in exploration mode, states that have been visited least are given a bonus of RMax to drive exploration.

Parameters **rmax** : float

The maximum achievable reward.

func : callable

Callback function to retrieve the minimum number of times a state has been visited.

thresh : float

If all states that can be reached from the current state have a value less than the threshold, exploration mode is turned on.

Attributes

mid	The module's unique identifier.

mlpy.mdp.discrete.LeastVisitedBonusExplorer.mid

`LeastVisitedBonusExplorer.`**`mid`**

The module's unique identifier.

Returns str :

The module's unique identifier

Methods

activate([qvalues])	Turn on exploration mode.
deactivate()	Turn off exploration mode.
load(filename)	Load the state of the module from file.
save(filename)	Save the current state of the module to file.
update(model)	Update the reward model.

mlpy.mdp.discrete.LeastVisitedBonusExplorer.activate

`LeastVisitedBonusExplorer.`**`activate`**(*qvalues=None*, **args*, ***kwargs*)

Turn on exploration mode.

If it is predicted that only states with rewards less than the threshold can be reached then the agent goes into exploration mode.

Parameters **qvalues** : dict

The qvalues for all actions from the current state

mlpy.mdp.discrete.LeastVisitedBonusExplorer.deactivate

`LeastVisitedBonusExplorer.`**`deactivate`**()

Turn off exploration mode.

mlpy.mdp.discrete.LeastVisitedBonusExplorer.load

`LeastVisitedBonusExplorer.`**`load`**`(`*filename*`)`

Load the state of the module from file.

Parameters **filename** : str

The name of the file to load from.

Notes

This is a class method, it can be accessed without instantiation.

mlpy.mdp.discrete.LeastVisitedBonusExplorer.save

`LeastVisitedBonusExplorer.`**`save`**`(`*filename*`)`

Save the current state of the module to file.

Parameters **filename** : str

The name of the file to save to.

mlpy.mdp.discrete.LeastVisitedBonusExplorer.update

`LeastVisitedBonusExplorer.`**`update`**`(`*model*`)`

Update the reward model.

Update the reward model according to a RMax based exploration policy. To drive exploration a bonus of RMax is given to the least visited states.

Parameters **model** : StateActionInfo

The states-action information.

mlpy.mdp.discrete.UnknownBonusExplorer

class `mlpy.mdp.discrete.`**`UnknownBonusExplorer`**`(`*rmax*`)`

Bases: *`mlpy.mdp.discrete.RMaxExplorer`*

Unknown bonus explorer, a RMax based exploration model.

States for which the decision tree was unable to predict a reward are given a bonus of RMax to drive exploration, since these states are considered to be unknown under the model.

Parameters **rmax** : float

The maximum achievable reward.

Attributes

mid	The module's unique identifier.

mlpy.mdp.discrete.UnknownBonusExplorer.mid

`UnknownBonusExplorer.mid`
The module's unique identifier.

Returns str :

The module's unique identifier

Methods

`activate`(*args, **kwargs)	Turn on exploration mode.
`deactivate`()	Turn off exploration mode.
`load`(filename)	Load the state of the module from file.
`save`(filename)	Save the current state of the module to file.
`update`(model)	Update the reward model.

mlpy.mdp.discrete.UnknownBonusExplorer.activate

`UnknownBonusExplorer.activate(*args, **kwargs)`
Turn on exploration mode.

mlpy.mdp.discrete.UnknownBonusExplorer.deactivate

`UnknownBonusExplorer.deactivate()`
Turn off exploration mode.

mlpy.mdp.discrete.UnknownBonusExplorer.load

`UnknownBonusExplorer.load(filename)`
Load the state of the module from file.

Parameters **filename** : str

The name of the file to load from.

Notes

This is a class method, it can be accessed without instantiation.

mlpy.mdp.discrete.UnknownBonusExplorer.save

`UnknownBonusExplorer.save(filename)`
Save the current state of the module to file.

Parameters **filename** : str

The name of the file to save to.

mlpy.mdp.discrete.UnknownBonusExplorer.update

UnknownBonusExplorer.**update**(*model*)

Update the reward model.

Update the reward model according to a RMax based exploration policy. States for which the decision tree was unable to predict a reward are considered unknown. These states are given a bonus of RMax to drive exploration.

Parameters model : StateActionInfo

The states-action information.

Contiguous models

casml	Continuous Action and State Model Learner (CASML)

mlpy.mdp.continuous.casml

Continuous Action and State Model Learner (CASML)

CASMLReuseMethod	The reuse method implementation for *CASML*.
CASMLRevisionMethod	The revision method implementation for *CASML*.
CASMLRetentionMethod	The retention method implementation for *CASML*.
CASML	Continuous Action and State Model Learner (CASML).

mlpy.mdp.continuous.casml.CASMLReuseMethod

class mlpy.mdp.continuous.casml.**CASMLReuseMethod**

Bases: *mlpy.knowledgerep.cbr.methods.IReuseMethod*

The reuse method implementation for *CASML*.

The solutions of the best (or set of best) retrieved cases are used to construct the solution for the query case; new generalizations and specializations may occur as a consequence of the solution transformation.

The CASML reuse method further specializes the solution by identifying cases similar in both state and action.

Methods

execute(case, case_matches[, fn_retrieve])	Execute reuse step.
plot_data(case, case_matches[, revised_matches])	Plot the data.

mlpy.mdp.continuous.casml.CASMLReuseMethod.execute

CASMLReuseMethod.**execute**(*case*, *case_matches*, *fn_retrieve=None*)

Execute reuse step.

Take both similarity in state and in actions into account.

Parameters case : Case

The query case.

case_matches : dict[int, CaseMatch]

The solution identified through the similarity measure.

fn_retrieve : callable

Callback for accessing the case base's 'retrieval' method.

Returns **revised_matches** : dict[int, CaseMatch]

The revised solution.

mlpy.mdp.continuous.casml.CASMLReuseMethod.plot_data

`CASMLReuseMethod.`**`plot_data`**(*case*, *case_matches*, *revised_matches=None*)

Plot the data.

Parameters **case** : Case

The query case.

case_matches : dict[int, CaseMatch]

The solution identified through the similarity measure.

revised_matches : dict[int, CaseMatch]

The revised solution.

mlpy.mdp.continuous.casml.CASMLRevisionMethod

class `mlpy.mdp.continuous.casml.`**`CASMLRevisionMethod`**(*rho=None*, *plot_revision=None*, *plot_revision_params=None*)

Bases: *`mlpy.knowledgerep.cbr.methods.IRevisionMethod`*

The revision method implementation for *`CASML`*.

The solutions provided by the query case is evaluated and information about whether the solution has or has not provided a desired outcome is gathered.

Parameters **rho** : float, optional

The permitted error of the similarity measure. Default is 0.99.

plot_revision : bool, optional

Whether to plot the vision step or not. Default is False.

plot_revision_params : {'origin_to_query', 'original_origin'}

Parameters used for plotting. Valid parameters are:

origin_to_query Which moves the origins of all actions to the query case's origin.

original_origin The origins remain unchanged and the states are plotted at their original origins.

Notes

The CASML revision method further narrows down the solutions to the query case by identifying whether the actions are similar by ensuring that the actions are cosine similar within the permitted error ρ:

$$d(c_{q.\text{action}}, c.\text{action}) >= \rho$$

Methods

execute(case, case_matches, **kwargs)	Execute the revision step.
plot_data(case, case_matches, **kwargs)	Plot the data during the revision step.

mlpy.mdp.continuous.casml.CASMLRevisionMethod.execute

`CASMLRevisionMethod.`**`execute`**(*case*, *case_matches*, ***kwargs*)

Execute the revision step.

Parameters **case** : Case

The query case.

case_matches : dict[int, CaseMatch]

The solution identified through the similarity measure.

Returns **case_matches** : dict[int, CaseMatch]

the corrected solution.

mlpy.mdp.continuous.casml.CASMLRevisionMethod.plot_data

`CASMLRevisionMethod.`**`plot_data`**(*case*, *case_matches*, ***kwargs*)

Plot the data during the revision step.

Parameters **case** : Case

The query case.

case_matches : dict[int, CaseMatch]

The solution identified through the similarity measure.

mlpy.mdp.continuous.casml.CASMLRetentionMethod

class `mlpy.mdp.continuous.casml.`**`CASMLRetentionMethod`**(*tau=None*, *sigma=None*, *plot_retention=None*)

Bases: *mlpy.knowledgerep.cbr.methods.IRetentionMethod*

The retention method implementation for *CASML*.

When the new problem-solving experience can be stored or not stored in memory, depending on the revision outcomes and the CBR policy regarding case retention.

Parameters **tau** : float, optional

The maximum permitted error when comparing most similar solution. Default is 0.8.

sigma : float, optional

The maximum permitted error when comparing actual with estimated transitions. Default is 0.2

plot_retention : bool, optional

Whether to plot the data during the retention step or not. Default is False.

Notes

The CASML retention method considers query cases as predicted correctly if

1.the query case is within the maximum permitted error τ of the most similar solution case:

$$d(\text{case}, 1\text{NN}(C_T, \text{case})) < \tau$$

2.the difference between the actual and the estimated transitions are less than the permitted error σ:

$$d(\text{case}.\Delta_\text{state}, T(s_{i-1}, a_{i-1}) < \sigma$$

Methods

`execute`(case, case_matches[, fn_add])	Execute the retention step.
`plot_data`(case, case_matches, **kwargs)	Plot the data during the retention step.

mlpy.mdp.continuous.casml.CASMLRetentionMethod.execute

`CASMLRetentionMethod.`**`execute`**(*case*, *case_matches*, *fn_add=None*)
Execute the retention step.

Parameters **case** : Case

The query case

case_matches : dict[int, CaseMatch]

The solution identified through the similarity measure.

fn_add : callable

Callback for accessing the case base's 'add' method.

mlpy.mdp.continuous.casml.CASMLRetentionMethod.plot_data

`CASMLRetentionMethod.`**`plot_data`**(*case*, *case_matches*, ***kwargs*)
Plot the data during the retention step.

Parameters **case** : Case

The query case.

case_matches : dict[int, CaseMatch]

The solution identified through the similarity measure.

mlpy.mdp.continuous.casml.CASML

class `mlpy.mdp.continuous.casml.`**`CASML`**(*case_template*, *rho=None*, *tau=None*, *sigma=None*, *ncomponents=1*, *revision_method_params=None*, *retention_method_params=None*, *case_base_params=None*, *hmm_params=None*, *proba_calc_method=None*)

Bases: *`mlpy.mdp.IMDPModel`*

Continuous Action and State Model Learner (CASML).

Parameters **case_template** : dict

The template from which to create a new case.

Example An example template for a feature named *state* with the specified feature parameters. *data* is the data from which to extract the case from. In this example it is expected that *data* has a member variable *state*.

```
{
    "state": {
        "type": "float",
        "value": "data.state",
        "is_index": True,
        "retrieval_method": "radius-n",
        "retrieval_method_params": 0.01
    },
    "delta_state": {
        "type": "float",
        "value": "data.next_state - data.state",
        "is_index": False,
    }
}
```

rho : float, optional

The maximum permitted error when comparing cosine similarity of actions. Default is 0.99.

tau : float, optional

The maximum permitted error when comparing most similar solution. Default is 0.8.

sigma : float, optional

The maximum permitted error when comparing actual with estimated transitions. Default is 0.2.

ncomponents : int, optional

Number of states of the hidden Markov model. Default is 1.

revision_method_params : dict, optional

Additional initialization parameters for *`CASMLRevisionMethod`*.

retention_method_params : dict, optional

Additional initialization parameters for *`CASMLRetentionMethod`*.

case_base_params : dict, optional

Initialization parameters for `CaseBase`.

hmm_params : dict, optional

Additional initialization parameters for `GaussianHMM`.

proba_calc_method : str, optional

The method used to calculate the probability distribution for the initial states. Default is DefaultProbaCalcMethod.

Attributes

`mid`	The module's unique identifier.

mlpy.mdp.continuous.casml.CASML.mid

CASML.**mid**

The module's unique identifier.

Returns str :

The module's unique identifier

Methods

`fit`(obs, actions[, n_init])	Fit the `CaseBase` and the `HMM`.
`load`(filename)	Load the state of the module from file.
`predict_proba`(state, action)	Predict the probability distribution.
`sample`([state, action])	Sample from the probability distribution.
`save`(filename)	Save the current state of the module to file.
`update`(experience)	Update the model with the agent's experience.

mlpy.mdp.continuous.casml.CASML.fit

CASML.**fit** (*obs*, *actions*, *n_init=1*, ***kwargs*)

Fit the `CaseBase` and the `HMM`.

The model is fit to the observations and actions of the trajectory by updating the case base and the HMM.

Parameters obs : array_like, shape (*nfeatures*, *n*)

Trajectory of observations, where each observation has *nfeatures* features and *n* is the length of the trajectory.

actions : array_like, shape (*nfeatures*, *n*)

Trajectory of actions, where each action has *nfeatures* features and *n* is the length of the trajectory.

n_init : int, optional

Number of restarts to prevent the HMM from getting stuck in a local minimum. Default is 1.

mlpy.mdp.continuous.casml.CASML.load

`CASML.load`(*filename*)

Load the state of the module from file.

Parameters **filename** : str

The name of the file to load from.

Notes

This is a class method, it can be accessed without instantiation.

mlpy.mdp.continuous.casml.CASML.predict_proba

`CASML.predict_proba`(*state*, *action*)

Predict the probability distribution.

Predict the probability distribution for state transitions given a state and an action.

Parameters **state** : State

The current state the robot is in.

action : Action

The action perform in state *state*.

Returns dict[tuple[float]], float] :

The probability distribution for the state-action pair.

mlpy.mdp.continuous.casml.CASML.sample

`CASML.sample`(*state=None*, *action=None*)

Sample from the probability distribution.

The next state is sampled for the given state and action from the probability distribution. If either state or action is `None` the next state is sampled from the initial distribution.

Parameters **state** : State, optional

The current state the robot is in.

action : Action, optional

The action perform in state *state*.

Returns State :

The sampled next state.

mlpy.mdp.continuous.casml.CASML.save

`CASML.save`(*filename*)

Save the current state of the module to file.

Parameters **filename** : str

The name of the file to save to.

mlpy.mdp.continuous.casml.CASML.update

CASML.**update**(*experience*)

Update the model with the agent's experience.

Parameters **experience** : Experience

The agent's experience, consisting of state, action, next state(, and reward).

Returns bool :

Return True if the model has changed, False otherwise.

Raises **NotImplementedError**

If the child class does not implement this function.

Notes

Optionally this method can be overwritten if the model supports incrementally updating the model.

Probability distributions

ProbaCalcMethodFactory	The probability calculation method factory.
IProbaCalcMethod	The Probability calculation method interface.
DefaultProbaCalcMethod	The default probability calculation method.
ProbabilityDistribution	Probability Distribution.

mlpy.mdp.distrib.ProbaCalcMethodFactory

class mlpy.mdp.distrib.**ProbaCalcMethodFactory**

Bases: object

The probability calculation method factory.

An instance of a probability calculation method can be created by passing the probability calculation method type.

Examples

```
>>> from mlpy.mdp.distrib import ProbaCalcMethodFactory
>>> ProbaCalcMethodFactory.create('defaultprobacalcmethod')
```

This creates a *DefaultProbaCalcMethod* instance.

Methods

create(_type, *args, **kwargs)	Create a probability calculation method of the given type.

mlpy.mdp.distrib.ProbaCalcMethodFactory.create

static `ProbaCalcMethodFactory.`**`create`**`(`*_type*, **args*, ***kwargs*`)`

Create a probability calculation method of the given type.

Parameters **_type** : str

The probability calculation method type. The method type should be equal to the class name of the method.

args : tuple

Positional arguments passed to the class of the given type for initialization.

kwargs : dict

Non-positional arguments passed to the class of the given type for initialization.

Returns IProbaCalcMethod :

A probability calculation method instance of the given type.

mlpy.mdp.distrib.IProbaCalcMethod

class `mlpy.mdp.distrib.`**`IProbaCalcMethod`**

Bases: `object`

The Probability calculation method interface.

The probability calculation method is responsible for calculating the probability distribution based on the state transitions seen so far.

Notes

To create custom probability calculation methods, derive from this class.

Methods

execute(states)	Execute the calculation.

mlpy.mdp.distrib.IProbaCalcMethod.execute

`IProbaCalcMethod.`**`execute`**`(`*states*`)`

Execute the calculation.

Parameters **states** : dict[State, dict[str, int | float]]

The list of next states to consider.

Returns dict[State, dict[str, int | float]] :

The updated states information including the probabilities.

Raises **NotImplementedError :**

If the child class does not implement this function.

mlpy.mdp.distrib.DefaultProbaCalcMethod

class `mlpy.mdp.distrib.`**`DefaultProbaCalcMethod`**

Bases: *`mlpy.mdp.distrib.IProbaCalcMethod`*

The default probability calculation method.

The default probability calculation method determines the probability distribution by normalizing the state count over all state.

Methods

`execute`(states)	Execute the calculation.

mlpy.mdp.distrib.DefaultProbaCalcMethod.execute

`DefaultProbaCalcMethod.`**`execute`**(*states*)

Execute the calculation.

Calculate the probability distribution based on the number of times the states have been seen so far.

Parameters **states** : dict[State, dict[str, int | float]]

The list of next states to consider.

Returns dict[State, dict[str, int | float]] :

The updated states information including the probabilities.

mlpy.mdp.distrib.ProbabilityDistribution

class `mlpy.mdp.distrib.`**`ProbabilityDistribution`**(*proba_calc_method=None*)

Bases: `object`

Probability Distribution.

This class handles evaluation of empirically derived states and calculates the probability distribution from them.

Parameters **proba_calc_method** : str

The method used to calculate the probability distribution for the initial state. Defaults to 'defaultprobacalcmethod'.

Methods

`add_state`(state)	Adds a state to the states list.
`clear`()	Clear the probability distribution.
`get`()	Retrieve the probability distribution.

Continued on next page

Table 16.29 – continued from previous page

iadd(state, proba)	In-place addition of the probability to the states probability.
sample()	Returns a next state according to the probability distribution.

mlpy.mdp.distrib.ProbabilityDistribution.add_state

`ProbabilityDistribution.`**`add_state`**`(`*state*`)`

Adds a state to the states list.

Adds a state to the states list in order to build the probability distribution.

Parameters **state** : State

An initial state.

mlpy.mdp.distrib.ProbabilityDistribution.clear

`ProbabilityDistribution.`**`clear`**`()`

Clear the probability distribution.

mlpy.mdp.distrib.ProbabilityDistribution.get

`ProbabilityDistribution.`**`get`**`()`

Retrieve the probability distribution.

Returns dict[State, float] :

A list of probabilities for all possible transitions.

mlpy.mdp.distrib.ProbabilityDistribution.iadd

`ProbabilityDistribution.`**`iadd`**`(`*state*, *proba*`)`

In-place addition of the probability to the states probability.

If the state does not exist in the list of states, it will be added.

Parameters **state** : State

The state for which the probability is updated.

proba : float

The probability value to add to the state's probability.

mlpy.mdp.distrib.ProbabilityDistribution.sample

`ProbabilityDistribution.`**`sample`**`()`

Returns a next state according to the probability distribution.

Returns State :

The next state sampled from the probability distribution.

State and action information

Experience	Experience base class.
RewardFunction	The reward function.
StateActionInfo	The models interface.
StateData	State information interface.
MDPPrimitive	A Markov decision process primitive.
State	Representation of the state.
Action	Representation of an action.

mlpy.mdp.stateaction.Experience

class `mlpy.mdp.stateaction.`**`Experience`**(*state*, *action*, *next_state*, *reward=None*)

Bases: `object`

Experience base class.

Representation of an experience occurring from acting in the environment.

Parameters **state** : State

The representation of the current state.

action : Action

The executed action.

next_state : State

The representation of the state following from acting with *action* in state *state*.

reward : int or float

The reward awarded by the environment for the state-action pair.

Attributes

state	(State) The experienced state
action	(Action) The experienced action.
next_state	(State) The experienced next state.
reward	(float) The experienced reward.

mlpy.mdp.stateaction.RewardFunction

class `mlpy.mdp.stateaction.`**`RewardFunction`**

Bases: `object`

The reward function.

The reward function is responsible for calculating the proper value of the reward. Callback functions can be specified for custom calculation of the reward value.

Notes

To ensure that the correct value of the reward is being accessed, the user should not access the class variables directly but instead use the methods *set* and *get* to set and get the reward respectively.

Examples

```
>>> RewardFunction.cb_get = staticmethod(lambda r, s: np.dot(s, RewardFunction.
↪reward))
```

In this cas the reward function is calculated by taking the dot product of the stored reward and a passed in value.

```
>>> RewardFunction.reward = [0.1, 0.9. 1.0, 0.0]
```

This sets the reward for all instances of the reward function.

```
>>> reward_func = RewardFunction()
>>> print reward_func.get([0.9, 0.5, 0.0, 1.0])
0.54
```

This calculates the reward *r* according to previously defined the callback function.

Attributes

bonus	The bonus added to the reward to encourage exploration.

mlpy.mdp.stateaction.RewardFunction.bonus

`RewardFunction.`**`bonus`**

The bonus added to the reward to encourage exploration.

Returns float :

The bonus added to the reward.

cb_get	(callable) Callback function to retrieve the reward value.
cb_set	(callable) Callback function to set the reward value.
reward	(float) The reward value.
rmax	(float) The maximum possible reward.
activate_bonus	(bool) Flag activating/deactivating the bonus.

Methods

get(*args, **kwargs)	Retrieve the reward value.
set(value, *args, **kwargs)	Set the reward value.

mlpy.mdp.stateaction.RewardFunction.get

`RewardFunction.`**`get`**`(`**args*, ***kwargs*`)`

Retrieve the reward value.

If `cb_get` is set, the callback will be called to retrieve the value.

Parameters **args** : tuple

Positional arguments passed to the callback.

kwargs : dict

Non-positional arguments passed to the callback.

Returns float :

The (calculated) reward value.

mlpy.mdp.stateaction.RewardFunction.set

`RewardFunction.`**`set`**`(`*value*, **args*, ***kwargs*`)`

Set the reward value.

If `cb_set` is set, the callback will be called to set the value.

Parameters **args** : tuple

Positional arguments passed to the callback.

kwargs : dict

Non-positional arguments passed to the callback.

mlpy.mdp.stateaction.StateActionInfo

class `mlpy.mdp.stateaction.`**`StateActionInfo`**

Bases: `object`

The models interface.

Contains all relevant information predicted by a model for a given state-action pair. This includes the (predicted) reward and transition probabilities to possible next states.

Attributes

transition_proba	(ProbabilityDistribution) The transition probability distribution.
reward_func	(RewardFunction) The reward function.
visits	(int) The number of times the state-action pair has been visited.
known	(bool) Flag indicating whether a reward value is known or not.

mlpy.mdp.stateaction.StateData

class `mlpy.mdp.stateaction.`**`StateData`**`(`*state_id*, *actions*`)`

Bases: `object`

State information interface.

Information about the state can be accessed here.

Parameters **state_id** : int

The unique id of the state

actions : list[Action]

List of actions that can be taken in this state.

Attributes

id	(int) The unique id of the state.
models	(dict) The reward and transition models for each action.
q	(dict) The q-table, containing a q-value for each action.
steps_away	(int) The number of steps the state is away from its closest neighbor.

mlpy.mdp.stateaction.MDPPrimitive

class `mlpy.mdp.stateaction.`**`MDPPrimitive`**(*features*, *name=None*)

Bases: `object`

A Markov decision process primitive.

The base class for *`State`* and *`Action`*. Primitives are represented by a list of features. They optionally can have a *name*.

Parameters **features** : array_like, shape (*nfeatures*,)

List of features, where *nfeatures* is the number of features identifying the primitive.

name : str, optional

The name of the primitive. Default is "".

Raises **ValueError**

If the feature array is not one-dimensional.

Notes

Use the *description* to encode action information. The information should contain the list of all available feature combinations, the name of each feature.

Examples A description of an action with three possible discrete actions:

```
{
    "out": {"value": [-0.004]},
    "in": {"value": [0.004]},
    "kick": {"value": [-1.0]}
}
```

A description of an action with one possible continuous action with name *move*, a value of * allows to find the action for every feature array. Additional information encodes the feature name together with its index into the feature array are given for each higher level element of feature array:

```
{
    "move": {
        "value": "*",
        "descr": {
            "LArm": {"dx": 0, "dy": 1, "dz": 2},
            "RArm": {"dx": 3, "dy": 4, "dz": 5},
            "LLeg": {"dx": 6, "dy": 7, "dz": 8},
            "RLeg": {"dx": 9, "dy": 10, "dz": 11},
            "Torso": {"dx": 12, "dy": 13, "dz": 14}
        }
    }
}
```

Similarly, a continuous state can be encoded as follows, which identifies the name of each feature together with its index into the feature array:

```
{
    "LArm": {"x": 0, "y": 1, "z": 2},
    "RArm": {"x": 3, "y": 4, "z": 5},
    "LLeg": {"x": 6, "y": 7, "z": 8},
    "RLeg": {"x": 9, "y": 10, "z": 11},
    "Torso": {"x": 12, "y": 13, "z": 14}
}
```

A discrete state can be encoded by identifying the position of each feature:

```
{
    "image x-position": 0,
    "displacement (mm)": 1
}
```

Alternatively, the feature can be identified by a list of features, giving he positional description:

```
["image x-position", "displacement (mm)"]
```

Rather then setting the attributes directly, use the methods *set_nfeatures*, *set_dtype*, *set_description*, *set_discretized*, *set_minmax_features*, and *set_states_per_dim* in order to enforce type checking.

Attributes

name	The name of the MDP primitive.
dtype	

mlpy.mdp.stateaction.MDPPrimitive.name

MDPPrimitive.**name**

The name of the MDP primitive.

Returns str :

The name of the primitive.

mlpy.mdp.stateaction.MDPPrimitive.dtype

`MDPPrimitive.`**`dtype`** **= <Mock name='mock.float64' id='140284520557840'>**

nfeatures	(int) The number of features.
discretized	(bool) Flag indicating whether the features are discretized or not.
min_features	(list) The minimum value for each feature.
max_features	(list) The minimum value for each feature.
states_per_dim	(list) The number of states per dimension.
description	(dict) A description of the features.

Methods

`DTYPE_FLOAT`	
`DTYPE_INT`	
`DTYPE_OBJECT`	
`decode`(_repr)	Decodes the state into its original representation.
`discretize`()	Discretizes the state.
`dtype`	
`encode`()	Encodes the state into a human readable representation.
`get`()	Return the feature array.
`key_to_index`(key)	Maps internal name to group index.
`next`()	
`set`(features)	Sets the feature array to the given array.
`set_description`(descr)	Set the feature description.
`set_discretized`([val])	Sets the *discretized* flag.
`set_dtype`([value])	Set the feature's data type.
`set_minmax_features`(_min, _max)	Sets the minimum and maximum value for each feature.
`set_nfeatures`(n)	Set the number of features.
`set_states_per_dim`(nstates)	Sets the number of states per feature.
`tolist`()	Returns the feature array as a list.

mlpy.mdp.stateaction.MDPPrimitive.decode

classmethod `MDPPrimitive.`**`decode`**`(`*_repr*`)`

Decodes the state into its original representation.

Parameters **_repr** : tuple

The readable representation of the primitive.

Returns State :

The decoded state.

Notes

Optionally this method can be overwritten at runtime.

Examples

```
>>> def my_decode(cls, _repr)
...     pass
...
>>> MDPPrimitive.decode = classmethod(my_decode)
```

mlpy.mdp.stateaction.MDPPrimitive.discretize

MDPPrimitive.**discretize**()

Discretizes the state.

Discretize the state using the information from the minimum and maximum values for each feature and the number of states attributed to each feature.

mlpy.mdp.stateaction.MDPPrimitive.encode

MDPPrimitive.**encode**()

Encodes the state into a human readable representation.

Returns ndarray :

The encoded state.

Notes

Optionally this method can be overwritten at runtime.

Examples

```
>>> def my_encode(self)
...     pass
...
>>> MDPPrimitive.encode = my_encode
```

mlpy.mdp.stateaction.MDPPrimitive.get

MDPPrimitive.**get**()

Return the feature array.

Returns ndarray :

The feature array.

mlpy.mdp.stateaction.MDPPrimitive.key_to_index

static MDPPrimitive.**key_to_index**(*key*)

Maps internal name to group index.

Maps the internal name of a feature to the index of the corresponding feature grouping. For example for a feature vector consisting of the x-y-z position of the left and the right arm, the features for the left and the right arm can be extracted separately as a group, effectively splitting the feature vector into two vectors with x, y, and z at the positions specified by the the mapping of this function.

Parameters **key** : str

The key into the mapping

Returns int :

The index in the feature array.

Raises **NotImplementedError**

If the child class does not implement this function.

Notes

Optionally this method can be overwritten at runtime.

Examples

```
>>> def my_key_to_index(key)
...     return {
...         "x": 0,
...         "y": 1,
...         "z": 2
...     }[key]
...
>>> State.description = {'LArm': {'x': 0, 'y': 1, 'z': 2}
...                      'RArm': {'x': 3, 'y': 4, 'z': 5}}
>>> State.key_to_index = staticmethod(my_key_to_index)
```

This specifies the mapping in both direction.

```
>>> state = [0.1, 0.4, 0.3. 4.6. 2.5. 0.9]
>>>
>>> mapping = State.description['LArm']
>>>
>>> larm = np.zeros[len(mapping.keys())]
>>> for key, axis in mapping.iteritems():
...     larm[State.key_to_index(key)] = state[axis]
...
>>> print larm
[0.1, 0.4, 0.3]
```

This extracts the features for the left arm from the *state* vector.

mlpy.mdp.stateaction.MDPPrimitive.next

`MDPPrimitive.`**`next`**`()`

mlpy.mdp.stateaction.MDPPrimitive.set

`MDPPrimitive.`**`set`**`(`*features*`)`

Sets the feature array to the given array.

Parameters **features** : array_like, shape (*nfeatures*,)

The new feature values.

mlpy.mdp.stateaction.MDPPrimitive.set_description

classmethod `MDPPrimitive.`**`set_description`**(*descr*)

Set the feature description.

This extracts the number of features from the description and checks that it matches with the *nfeatures*. If *nfeatures* is None, *nfeatures* is set to the extracted value.

Parameters **descr** : dict

The feature description.

Raises **ValueError**

If the number of features extracted from the description does not match *nfeatures* or if *name* isn't of type string.

Notes

Use the *description* to encode action information. The information should contain the list of all available feature combinations, the name of each feature.

Examples

A description of an action with three possible discrete actions:

```
{
    "out": {"value": [-0.004]},
    "in": {"value": [0.004]},
    "kick": {"value": [-1.0]}
}
```

A description of an action with one possible continuous action with name *move*, a value of * allows to find the action for every feature array. Additional information encodes the feature name together with its index into the feature array are given for each higher level element of feature array:

```
{
    "move": {
        "value": "*",
        "descr": {
            "LArm": {"dx": 0, "dy": 1, "dz": 2},
            "RArm": {"dx": 3, "dy": 4, "dz": 5},
            "LLeg": {"dx": 6, "dy": 7, "dz": 8},
            "RLeg": {"dx": 9, "dy": 10, "dz": 11},
            "Torso": {"dx": 12, "dy": 13, "dz": 14}
        }
    }
}
```

Similarly, a continuous state can be encoded as follows, which identifies the name of each feature together with its index into the feature array:

```
{
    "LArm": {"x": 0, "y": 1, "z": 2},
    "RArm": {"x": 3, "y": 4, "z": 5},
    "LLeg": {"x": 6, "y": 7, "z": 8},
    "RLeg": {"x": 9, "y": 10, "z": 11},
    "Torso": {"x": 12, "y": 13, "z": 14}
}
```

A discrete state can be encoded by identifying the position of each feature:

```
"descr": {
    "image x-position": 0,
    "displacement (mm)": 1
}
```

Alternatively, the feature can be identified by a list of features, giving he positional description:

```
["image x-position", "displacement (mm)"]
```

mlpy.mdp.stateaction.MDPPrimitive.set_discretized

classmethod `MDPPrimitive.`**`set_discretized`**(*val=False*)

Sets the *discretized* flag.

Parameters **val** : bool

Flag identifying whether the features are discretized or not. Default is False.

Raises **ValueError**

If *val* is not boolean type.

mlpy.mdp.stateaction.MDPPrimitive.set_dtype

classmethod `MDPPrimitive.`**`set_dtype`**(*value=<Mock name='mock.float64' id='140284520557840'>*)

Set the feature's data type.

Parameters **value** : {DTYPE_FLOAT, DTYPE_INT, DTYPE_OBJECT}

The data type.

Raises **ValueError**

If the data type is not one of the allowed types.

mlpy.mdp.stateaction.MDPPrimitive.set_minmax_features

classmethod `MDPPrimitive.`**`set_minmax_features`**(*_min*, *_max*)

Sets the minimum and maximum value for each feature.

This extracts the number of features from the *_min* and *_max* values and ensures that it matches with *nfeatures*. If *nfeatures* is None, the *nfeatures* attribute is set to the extracted value.

Parameters **_min** : array_like, shape(*nfeatures*,)

The minimum value for each feature

_max : array_like, shape(*nfeatures*,)

The maximum value for each feature

Raises ValueError

If the arrays are not one-dimensional vectors, the shapes of the arrays don't match, or the number of features does not agree with the attribute *nfeatures*.

mlpy.mdp.stateaction.MDPPrimitive.set_nfeatures

classmethod `MDPPrimitive.`**`set_nfeatures`**`(`*n*`)`

Set the number of features.

Parameters n : int

The number of features.

Raises ValueError

If *n* is not of type integer.

mlpy.mdp.stateaction.MDPPrimitive.set_states_per_dim

classmethod `MDPPrimitive.`**`set_states_per_dim`**`(`*nstates*`)`

Sets the number of states per feature.

This extracts the number of features from *nstates* and compares it to the attribute *nfeatures*. If it doesn't match, an exception is thrown. If the *nfeatures* attribute is None, *nfeatures* is set to the extracted value.

Parameters nstates : array_like, shape (*nfeatures*,)

The number of states per features

Raises ValueError

If the array is not a vector of length *nfeatures*.

mlpy.mdp.stateaction.MDPPrimitive.tolist

`MDPPrimitive.`**`tolist`**`()`

Returns the feature array as a list.

Returns list :

The features list.

mlpy.mdp.stateaction.State

class `mlpy.mdp.stateaction.`**`State`**`(`*features*, *name=None*`)`

Bases: *`mlpy.mdp.stateaction.MDPPrimitive`*

Representation of the state.

States are represented by an array of features.

Parameters features : array_like, shape (*nfeatures*,)

List of features, where *nfeatures* is the number of features identifying the primitive.

name : str, optional

The name of the primitive. Default is ‘’.

Notes

Use the *description* to encode action information. The information should contain the list of all available feature combinations, the name of each feature.

Examples A description of an action with three possible discrete actions:

```
{
    "out": {"value": [-0.004]},
    "in": {"value": [0.004]},
    "kick": {"value": [-1.0]}
}
```

A description of an action with one possible continuous action with name *move*, a value of * allows to find the action for every feature array. Additional information encodes the feature name together with its index into the feature array are given for each higher level element of feature array:

```
{
    "move": {
        "value": "*",
        "descr": {
            "LArm": {"dx": 0, "dy": 1, "dz": 2},
            "RArm": {"dx": 3, "dy": 4, "dz": 5},
            "LLeg": {"dx": 6, "dy": 7, "dz": 8},
            "RLeg": {"dx": 9, "dy": 10, "dz": 11},
            "Torso": {"dx": 12, "dy": 13, "dz": 14}
        }
    }
}
```

Similarly, a continuous state can be encoded as follows, which identifies the name of each feature together with its index into the feature array:

```
{
    "LArm": {"x": 0, "y": 1, "z": 2},
    "RArm": {"x": 3, "y": 4, "z": 5},
    "LLeg": {"x": 6, "y": 7, "z": 8},
    "RLeg": {"x": 9, "y": 10, "z": 11},
    "Torso": {"x": 12, "y": 13, "z": 14}
}
```

A discrete state can be encoded by identifying the position of each feature:

```
{
    "image x-position": 0,
    "displacement (mm)": 1
}
```

Alternatively, the feature can be identified by a list of features, giving he positional description:

```
["image x-position", "displacement (mm)"]
```

Rather then setting the attributes directly, use the methods *set_nfeatures*, *set_dtype*, *set_description*, *set_discretized*, *set_minmax_features*, and *set_states_per_dim* in order to enforce type checking.

Examples

```
>>> State.description = {'LArm': {'x': 0, 'y': 1, 'z': 2}
...                      'RArm': {'x': 3, 'y': 4, 'z': 5}}
```

This description identifies the features to be the x-y-z-position of the left and the right arm. The position into the feature array is given by the integer numbers.

```
>>> def my_key_to_index(key)
...     return {
...         "x": 0,
...         "y": 1,
...         "z": 2
...     }[key]
...
>>> State.key_to_index = staticmethod(my_key_to_index)
```

This defines a mapping for each key.

```
>>> state = [0.1, 0.4, 0.3. 4.6. 2.5. 0.9]
>>>
>>> mapping = State.description['LArm']
>>>
>>> larm = np.zeros[len(mapping.keys())]
>>> for key, axis in mapping.iteritems():
...     larm[State.key_to_index(key)] = state[axis]
...
>>> print larm
[0.1, 0.4, 0.3]
```

This extracts the features for the left arm from the *state* vector.

```
>>> s1 = State([0.1, 0.4, 0.2])
>>> s2 = State([0.5, 0.3, 0.5])
>>> print s1 - s2
[-0.4, 0.1, -0.3]
```

Subtract states from each other.

```
>>> print s1 * s2
[0.05, 0.12, 0.1]
```

Multiplies two states with each other.

```
>>> s1 *= s2
>>> print s1
[0.05, 0.12, 0.1]
```

Multiplies two states in place.

Attributes

name	The name of the MDP primitive.
dtype	

mlpy.mdp.stateaction.State.name

State.**name**
The name of the MDP primitive.

Returns str :

The name of the primitive.

mlpy.mdp.stateaction.State.dtype

State.**dtype = <Mock name='mock.float64' id='140284520557840'>**

nfeatures	(int) The number of features.
discretized	(bool) Flag indicating whether the features are discretized or not.
min_features	(list) The minimum value for each feature.
max_features	(list) The minimum value for each feature.
states_per_dim	(list) The number of states per dimension.
description	(dict) A description of the features.

Methods

DTYPE_FLOAT	
DTYPE_INT	
DTYPE_OBJECT	
decode(_repr)	Decodes the state into its original representation.
discretize()	Discretizes the state.
dtype	
encode()	Encodes the state into a human readable representation.
get()	Return the feature array.
is_initial()	Checks if the state is an initial state.
is_terminal()	Checks if the state is a terminal state.
is_valid()	Check if this state is a valid state.
key_to_index(key)	Maps internal name to group index.
next()	
set(features)	Sets the feature array to the given array.
set_description(descr)	Set the feature description.
set_discretized([val])	Sets the *discretized* flag.
set_dtype([value])	Set the feature's data type.
set_minmax_features(_min, _max)	Sets the minimum and maximum value for each feature.
set_nfeatures(n)	Set the number of features.
set_states_per_dim(nstates)	Sets the number of states per feature.
tolist()	Returns the feature array as a list.

mlpy.mdp.stateaction.State.decode

`State.decode(_repr)`

Decodes the state into its original representation.

Parameters **_repr** : tuple

The readable representation of the primitive.

Returns State :

The decoded state.

Notes

Optionally this method can be overwritten at runtime.

Examples

```
>>> def my_decode(cls, _repr)
...     pass
...
>>> MDPPrimitive.decode = classmethod(my_decode)
```

mlpy.mdp.stateaction.State.discretize

`State.discretize()`

Discretizes the state.

Discretize the state using the information from the minimum and maximum values for each feature and the number of states attributed to each feature.

mlpy.mdp.stateaction.State.encode

`State.encode()`

Encodes the state into a human readable representation.

Returns ndarray :

The encoded state.

Notes

Optionally this method can be overwritten at runtime.

Examples

```
>>> def my_encode(self)
...     pass
...
>>> MDPPrimitive.encode = my_encode
```

mlpy.mdp.stateaction.State.get

`State.get()`

Return the feature array.

Returns ndarray :

The feature array.

mlpy.mdp.stateaction.State.is_initial

`State.is_initial()`

Checks if the state is an initial state.

Returns bool :

Whether the state is an initial state or not.

mlpy.mdp.stateaction.State.is_terminal

`State.is_terminal()`

Checks if the state is a terminal state.

Returns bool :

Whether the state is a terminal state or not.

mlpy.mdp.stateaction.State.is_valid

`State.is_valid()`

Check if this state is a valid state.

Returns bool :

Whether the state is valid or not.

Notes

Optionally this method can be overwritten at runtime.

Examples

```
>>> def my_is_valid(self)
...     pass
...
>>> MDPPrimitive.is_valid = my_is_valid
```

mlpy.mdp.stateaction.State.key_to_index

`State.key_to_index(`*key*`)`

Maps internal name to group index.

Maps the internal name of a feature to the index of the corresponding feature grouping. For example for a feature vector consisting of the x-y-z position of the left and the right arm, the features for the left and the right arm can be extracted separately as a group, effectively splitting the feature vector into two vectors with x, y, and z at the positions specified by the the mapping of this function.

Parameters **key** : str

The key into the mapping

Returns int :

The index in the feature array.

Raises **NotImplementedError**

If the child class does not implement this function.

Notes

Optionally this method can be overwritten at runtime.

Examples

```
>>> def my_key_to_index(key)
...     return {
...         "x": 0,
...         "y": 1,
...         "z": 2
...     }[key]
...
>>> State.description = {'LArm': {'x': 0, 'y': 1, 'z': 2}
...                      'RArm': {'x': 3, 'y': 4, 'z': 5}}
>>> State.key_to_index = staticmethod(my_key_to_index)
```

This specifies the mapping in both direction.

```
>>> state = [0.1, 0.4, 0.3. 4.6. 2.5. 0.9]
>>>
>>> mapping = State.description['LArm']
>>>
>>> larm = np.zeros[len(mapping.keys())]
>>> for key, axis in mapping.iteritems():
...     larm[State.key_to_index(key)] = state[axis]
...
>>> print larm
[0.1, 0.4, 0.3]
```

This extracts the features for the left arm from the *state* vector.

mlpy.mdp.stateaction.State.next

`State.next()`

mlpy.mdp.stateaction.State.set

`State.`**`set`**(*features*)

Sets the feature array to the given array.

Parameters **features** : array_like, shape (*nfeatures*,)

The new feature values.

mlpy.mdp.stateaction.State.set_description

`State.`**`set_description`**(*descr*)

Set the feature description.

This extracts the number of features from the description and checks that it matches with the *nfeatures*. If *nfeatures* is None, *nfeatures* is set to the extracted value.

Parameters **descr** : dict

The feature description.

Raises **ValueError**

If the number of features extracted from the description does not match *nfeatures* or if *name* isn't of type string.

Notes

Use the *description* to encode action information. The information should contain the list of all available feature combinations, the name of each feature.

Examples

A description of an action with three possible discrete actions:

```
{
    "out": {"value": [-0.004]},
    "in": {"value": [0.004]},
    "kick": {"value": [-1.0]}
}
```

A description of an action with one possible continuous action with name *move*, a value of * allows to find the action for every feature array. Additional information encodes the feature name together with its index into the feature array are given for each higher level element of feature array:

```
{
    "move": {
        "value": "*",
        "descr": {
            "LArm": {"dx": 0, "dy": 1, "dz": 2},
            "RArm": {"dx": 3, "dy": 4, "dz": 5},
            "LLeg": {"dx": 6, "dy": 7, "dz": 8},
            "RLeg": {"dx": 9, "dy": 10, "dz": 11},
            "Torso": {"dx": 12, "dy": 13, "dz": 14}
        }
    }
}
```

Similarly, a continuous state can be encoded as follows, which identifies the name of each feature together with its index into the feature array:

```
{
    "LArm": {"x": 0, "y": 1, "z": 2},
    "RArm": {"x": 3, "y": 4, "z": 5},
    "LLeg": {"x": 6, "y": 7, "z": 8},
    "RLeg": {"x": 9, "y": 10, "z": 11},
    "Torso": {"x": 12, "y": 13, "z": 14}
}
```

A discrete state can be encoded by identifying the position of each feature:

```
"descr": {
    "image x-position": 0,
    "displacement (mm)": 1
}
```

Alternatively, the feature can be identified by a list of features, giving he positional description:

```
["image x-position", "displacement (mm)"]
```

mlpy.mdp.stateaction.State.set_discretized

`State.`**`set_discretized`**`(`*val=False*`)`

Sets the *discretized* flag.

Parameters **val** : bool

Flag identifying whether the features are discretized or not. Default is False.

Raises **ValueError**

If *val* is not boolean type.

mlpy.mdp.stateaction.State.set_dtype

`State.`**`set_dtype`**`(`*value=<Mock name='mock.float64' id='140284520557840'>*`)`

Set the feature's data type.

Parameters **value** : {DTYPE_FLOAT, DTYPE_INT, DTYPE_OBJECT}

The data type.

Raises **ValueError**

If the data type is not one of the allowed types.

mlpy.mdp.stateaction.State.set_minmax_features

`State.`**`set_minmax_features`**`(`*_min*, *_max*`)`

Sets the minimum and maximum value for each feature.

This extracts the number of features from the *_min* and *_max* values and ensures that it matches with *nfeatures*. If *nfeatures* is None, the *nfeatures* attribute is set to the extracted value.

Parameters **_min** : array_like, shape(*nfeatures*,)

The minimum value for each feature

_max : array_like, shape(*nfeatures*,)

The maximum value for each feature

Raises **ValueError**

If the arrays are not one-dimensional vectors, the shapes of the arrays don't match, or the number of features does not agree with the attribute *nfeatures*.

mlpy.mdp.stateaction.State.set_nfeatures

`State.`**`set_nfeatures`**`(`*n*`)`

Set the number of features.

Parameters **n** : int

The number of features.

Raises **ValueError**

If *n* is not of type integer.

mlpy.mdp.stateaction.State.set_states_per_dim

`State.`**`set_states_per_dim`**`(`*nstates*`)`

Sets the number of states per feature.

This extracts the number of features from *nstates* and compares it to the attribute *nfeatures*. If it doesn't match, an exception is thrown. If the *nfeatures* attribute is None, *nfeatures* is set to the extracted value.

Parameters **nstates** : array_like, shape (*nfeatures*,)

The number of states per features

Raises **ValueError**

If the array is not a vector of length *nfeatures*.

mlpy.mdp.stateaction.State.tolist

`State.`**`tolist`**`()`

Returns the feature array as a list.

Returns list :

The features list.

mlpy.mdp.stateaction.Action

class `mlpy.mdp.stateaction.`**`Action`**`(`*features*, *name=None*`)`

Bases: *`mlpy.mdp.stateaction.MDPPrimitive`*

Representation of an action.

Actions are represented by an array of features.

Parameters **features** : array_like, shape (*nfeatures*,)

List of features, where *nfeatures* is the number of features identifying the primitive.

name : str, optional

The name of the primitive. Default is ‘’.

Notes

Use the *description* to encode action information. The information should contain the list of all available feature combinations, the name of each feature.

Examples A description of an action with three possible discrete actions:

```
{
    "out": {"value": [-0.004]},
    "in": {"value": [0.004]},
    "kick": {"value": [-1.0]}
}
```

A description of an action with one possible continuous action with name *move*, a value of * allows to find the action for every feature array. Additional information encodes the feature name together with its index into the feature array are given for each higher level element of feature array:

```
{
    "move": {
        "value": "*",
        "descr": {
            "LArm": {"dx": 0, "dy": 1, "dz": 2},
            "RArm": {"dx": 3, "dy": 4, "dz": 5},
            "LLeg": {"dx": 6, "dy": 7, "dz": 8},
            "RLeg": {"dx": 9, "dy": 10, "dz": 11},
            "Torso": {"dx": 12, "dy": 13, "dz": 14}
        }
    }
}
```

Similarly, a continuous state can be encoded as follows, which identifies the name of each feature together with its index into the feature array:

```
{
    "LArm": {"x": 0, "y": 1, "z": 2},
    "RArm": {"x": 3, "y": 4, "z": 5},
    "LLeg": {"x": 6, "y": 7, "z": 8},
    "RLeg": {"x": 9, "y": 10, "z": 11},
    "Torso": {"x": 12, "y": 13, "z": 14}
}
```

A discrete state can be encoded by identifying the position of each feature:

```
{
    "image x-position": 0,
    "displacement (mm)": 1
}
```

Alternatively, the feature can be identified by a list of features, giving he positional description:

```
["image x-position", "displacement (mm)"]
```

Rather then setting the attributes directly, use the methods *set_nfeatures*, *set_dtype*, *set_description*, *set_discretized*, *set_minmax_features*, and *set_states_per_dim* in order to enforce type checking.

Examples

```
>>> Action.set_description({'LArm': {'dx': 0, 'dy': 1, 'dz': 2}
...                         'RArm': {'dx': 3, 'dy': 4, 'dz': 5}})
```

This description identifies the features to be the delta x-y-z-position of the left and the right arm. The position into the feature array is given by the integer numbers.

```
>>> def my_key_to_index(key)
...     return {
...         "dx": 0,
...         "dy": 1,
...         "dz": 2
...     }[key]
...
>>> Action.key_to_index = staticmethod(my_key_to_index)
```

This defines a mapping for each key.

```
>>> action = [0.1, 0.4, 0.3. 4.6. 2.5. 0.9]
>>>
>>> mapping = Action.description['LArm']
>>>
>>> larm = np.zeros[len(mapping.keys())]
>>> for key, axis in mapping.iteritems():
...     larm[Action.key_to_index(key)] = action[axis]
...
>>> print larm
[0.1, 0.4, 0.3]
```

This extracts the features for the left arm from the *action* vector.

```
>>> a1 = Action([0.1, 0.4, 0.2])
>>> a2 = Action([0.5, 0.3, 0.5])
>>> print a1 - a2
[-0.4, 0.1, -0.3]
```

Subtract actions from each other.

```
>>> print a1 * a2
[0.05, 0.12, 0.1]
```

Multiplies two actions with each other.

```
>>> a1 *= a2
>>> print a1
[0.05, 0.12, 0.1]
```

Multiplies two actions in place.

Attributes

`name`	The name of the MDP primitive.
`dtype`	

mlpy.mdp.stateaction.Action.name

`Action.name`
The name of the MDP primitive.

Returns str :

The name of the primitive.

mlpy.mdp.stateaction.Action.dtype

`Action.dtype` **= <Mock name='mock.float64' id='140284520557840'>**

nfeatures	(int) The number of features.
discretized	(bool) Flag indicating whether the features are discretized or not.
min_features	(list) The minimum value for each feature.
max_features	(list) The minimum value for each feature.
states_per_dim	(list) The number of states per dimension.
description	(dict) A description of the features.

Methods

`DTYPE_FLOAT`	
`DTYPE_INT`	
`DTYPE_OBJECT`	
`decode`(_repr)	Decodes the state into its original representation.
`discretize`()	Discretizes the state.
`dtype`	
`encode`()	Encodes the state into a human readable representation.
`get`()	Return the feature array.
`get_name`(features)	Retrieves the name of the action.
`get_noop_action`()	Creates a *no-op* action.
`key_to_index`(key)	Maps internal name to group index.
`next`()	
`set`(features)	Sets the feature array to the given array.
`set_description`(descr)	Set the feature description.
`set_discretized`([val])	Sets the *discretized* flag.
`set_dtype`([value])	Set the feature's data type.
`set_minmax_features`(_min, _max)	Sets the minimum and maximum value for each feature.
`set_nfeatures`(n)	Set the number of features.
`set_states_per_dim`(nstates)	Sets the number of states per feature.
`tolist`()	Returns the feature array as a list.

mlpy.mdp.stateaction.Action.decode

`Action.decode(_repr)`

Decodes the state into its original representation.

Parameters _repr : tuple

The readable representation of the primitive.

Returns State :

The decoded state.

Notes

Optionally this method can be overwritten at runtime.

Examples

```
>>> def my_decode(cls, _repr)
...     pass
...
>>> MDPPrimitive.decode = classmethod(my_decode)
```

mlpy.mdp.stateaction.Action.discretize

`Action.discretize()`

Discretizes the state.

Discretize the state using the information from the minimum and maximum values for each feature and the number of states attributed to each feature.

mlpy.mdp.stateaction.Action.encode

`Action.encode()`

Encodes the state into a human readable representation.

Returns ndarray :

The encoded state.

Notes

Optionally this method can be overwritten at runtime.

Examples

```
>>> def my_encode(self)
...     pass
...
>>> MDPPrimitive.encode = my_encode
```

mlpy.mdp.stateaction.Action.get

`Action.`**`get`**`()`

Return the feature array.

Returns ndarray :

The feature array.

mlpy.mdp.stateaction.Action.get_name

classmethod `Action.`**`get_name`**`(`*features*`)`

Retrieves the name of the action.

Retrieve the name of the action using the action's description. In the case that all features are zero the action is considered a *no-op* action.

Parameters **features** : ndarray

A feature array.

Returns str :

The name of the action.

mlpy.mdp.stateaction.Action.get_noop_action

classmethod `Action.`**`get_noop_action`**`()`

Creates a *no-op* action.

A *no-op* action does not have any effect.

Returns Action :

A *no-op* action.

mlpy.mdp.stateaction.Action.key_to_index

`Action.`**`key_to_index`**`(`*key*`)`

Maps internal name to group index.

Maps the internal name of a feature to the index of the corresponding feature grouping. For example for a feature vector consisting of the x-y-z position of the left and the right arm, the features for the left and the right arm can be extracted separately as a group, effectively splitting the feature vector into two vectors with x, y, and z at the positions specified by the the mapping of this function.

Parameters **key** : str

The key into the mapping

Returns int :

The index in the feature array.

Raises **NotImplementedError**

If the child class does not implement this function.

Notes

Optionally this method can be overwritten at runtime.

Examples

```
>>> def my_key_to_index(key)
...     return {
...         "x": 0,
...         "y": 1,
...         "z": 2
...     }[key]
...
>>> State.description = {'LArm': {'x': 0, 'y': 1, 'z': 2}
...                      'RArm': {'x': 3, 'y': 4, 'z': 5}}
>>> State.key_to_index = staticmethod(my_key_to_index)
```

This specifies the mapping in both direction.

```
>>> state = [0.1, 0.4, 0.3. 4.6. 2.5. 0.9]
>>>
>>> mapping = State.description['LArm']
>>>
>>> larm = np.zeros[len(mapping.keys())]
>>> for key, axis in mapping.iteritems():
...     larm[State.key_to_index(key)] = state[axis]
...
>>> print larm
[0.1, 0.4, 0.3]
```

This extracts the features for the left arm from the *state* vector.

mlpy.mdp.stateaction.Action.next

`Action.`**`next`**`()`

mlpy.mdp.stateaction.Action.set

`Action.`**`set`**`(`*features*`)`

Sets the feature array to the given array.

Parameters **features** : array_like, shape (*nfeatures*,)

The new feature values.

mlpy.mdp.stateaction.Action.set_description

`Action.`**`set_description`**`(`*descr*`)`

Set the feature description.

This extracts the number of features from the description and checks that it matches with the *nfeatures*. If *nfeatures* is None, *nfeatures* is set to the extracted value.

Parameters **descr** : dict

The feature description.

Raises ValueError

If the number of features extracted from the description does not match *nfeatures* or if *name* isn't of type string.

Notes

Use the *description* to encode action information. The information should contain the list of all available feature combinations, the name of each feature.

Examples

A description of an action with three possible discrete actions:

```
{
    "out": {"value": [-0.004]},
    "in": {"value": [0.004]},
    "kick": {"value": [-1.0]}
}
```

A description of an action with one possible continuous action with name *move*, a value of * allows to find the action for every feature array. Additional information encodes the feature name together with its index into the feature array are given for each higher level element of feature array:

```
{
    "move": {
        "value": "*",
        "descr": {
            "LArm": {"dx": 0, "dy": 1, "dz": 2},
            "RArm": {"dx": 3, "dy": 4, "dz": 5},
            "LLeg": {"dx": 6, "dy": 7, "dz": 8},
            "RLeg": {"dx": 9, "dy": 10, "dz": 11},
            "Torso": {"dx": 12, "dy": 13, "dz": 14}
        }
    }
}
```

Similarly, a continuous state can be encoded as follows, which identifies the name of each feature together with its index into the feature array:

```
{
    "LArm": {"x": 0, "y": 1, "z": 2},
    "RArm": {"x": 3, "y": 4, "z": 5},
    "LLeg": {"x": 6, "y": 7, "z": 8},
    "RLeg": {"x": 9, "y": 10, "z": 11},
    "Torso": {"x": 12, "y": 13, "z": 14}
}
```

A discrete state can be encoded by identifying the position of each feature:

```
"descr": {
    "image x-position": 0,
    "displacement (mm)": 1
}
```

Alternatively, the feature can be identified by a list of features, giving he positional description:

```
["image x-position", "displacement (mm)"]
```

mlpy.mdp.stateaction.Action.set_discretized

`Action.`**`set_discretized`**(*val=False*)

Sets the *discretized* flag.

Parameters val : bool

Flag identifying whether the features are discretized or not. Default is False.

Raises ValueError

If *val* is not boolean type.

mlpy.mdp.stateaction.Action.set_dtype

`Action.`**`set_dtype`**(*value=<Mock name='mock.float64' id='140284520557840'>*)

Set the feature's data type.

Parameters value : {DTYPE_FLOAT, DTYPE_INT, DTYPE_OBJECT}

The data type.

Raises ValueError

If the data type is not one of the allowed types.

mlpy.mdp.stateaction.Action.set_minmax_features

`Action.`**`set_minmax_features`**(*_min*, *_max*)

Sets the minimum and maximum value for each feature.

This extracts the number of features from the *_min* and *_max* values and ensures that it matches with *nfeatures*. If *nfeatures* is None, the *nfeatures* attribute is set to the extracted value.

Parameters _min : array_like, shape(*nfeatures*,)

The minimum value for each feature

_max : array_like, shape(*nfeatures*,)

The maximum value for each feature

Raises ValueError

If the arrays are not one-dimensional vectors, the shapes of the arrays don't match, or the number of features does not agree with the attribute *nfeatures*.

mlpy.mdp.stateaction.Action.set_nfeatures

`Action.`**`set_nfeatures`**(*n*)

Set the number of features.

Parameters n : int

The number of features.

Raises ValueError

If *n* is not of type integer.

mlpy.mdp.stateaction.Action.set_states_per_dim

`Action.`**`set_states_per_dim`**`(`*nstates*`)`

Sets the number of states per feature.

This extracts the number of features from *nstates* and compares it to the attribute *nfeatures*. If it doesn't match, an exception is thrown. If the *nfeatures* attribute is None, *nfeatures* is set to the extracted value.

Parameters nstates : array_like, shape (*nfeatures*,)

The number of states per features

Raises ValueError

If the array is not a vector of length *nfeatures*.

mlpy.mdp.stateaction.Action.tolist

`Action.`**`tolist`**`()`

Returns the feature array as a list.

Returns list :

The features list.

CHAPTER 17

Modules and design patterns (`mlpy.modules`)

This module contains various modules and design patterns.

Modules

`UniqueModule`	Class ensuring each instance has a unique name.
`Module`	Base module class from which most modules inherit from.

mlpy.modules.UniqueModule

class `mlpy.modules.`**`UniqueModule`**(*mid=None*)

Bases: `object`

Class ensuring each instance has a unique name.

The unique id can either be passed to the class or if none is passed, it will be generated using the module and class name.

Parameters **mid** : str

The module's unique identifier

Examples

```
>>> from mlpy.modules import UniqueModule
>>> class MyClass(UniqueModule):
>>>
>>>     def __init__(self, mid=None):
>>>         super(MyClass, self).__init__(mid)
```

This creates a unique model.

Attributes

mid	The module's unique identifier.

mlpy.modules.UniqueModule.mid

`UniqueModule.`**`mid`**

The module's unique identifier.

Returns str :

The module's unique identifier

Methods

load(filename)	Load the state of the module from file.
save(filename)	Save the current state of the module to file.

mlpy.modules.UniqueModule.load

classmethod `UniqueModule.`**`load`**(*filename*)

Load the state of the module from file.

Parameters **filename** : str

The name of the file to load from.

Notes

This is a class method, it can be accessed without instantiation.

mlpy.modules.UniqueModule.save

`UniqueModule.`**`save`**(*filename*)

Save the current state of the module to file.

Parameters **filename** : str

The name of the file to save to.

mlpy.modules.Module

class `mlpy.modules.`**`Module`**(*mid=None*)

Bases: *mlpy.modules.UniqueModule*

Base module class from which most modules inherit from.

The base module class handles processing of the program loop. A module inherits from the unique module class, thus every module has a unique name.

Parameters **mid** : str

The module's unique identifier

Examples

To create a module handling the program loop, write

```
>>> from mlpy.modules import Module
>>> class MyClass(Module):
>>>     pass
```

Attributes

mid	The module's unique identifier.

mlpy.modules.Module.mid

Module.**mid**

The module's unique identifier.

Returns str :

The module's unique identifier

Methods

enter(t)	Enter the module and perform initialization tasks.
exit()	Exit the module and perform cleanup tasks.
load(filename)	Load the state of the module from file.
reset(t, **kwargs)	Reset the module.
save(filename)	Save the current state of the module to file.
update(dt)	Update the module at every delta time step dt.

mlpy.modules.Module.enter

Module.**enter**(*t*)

Enter the module and perform initialization tasks.

Parameters **t** : float

The current time (sec)

mlpy.modules.Module.exit

Module.**exit**()

Exit the module and perform cleanup tasks.

mlpy.modules.Module.load

`Module.load(filename)`

Load the state of the module from file.

Parameters **filename** : str

The name of the file to load from.

Notes

This is a class method, it can be accessed without instantiation.

mlpy.modules.Module.reset

`Module.reset(t, **kwargs)`

Reset the module.

Parameters **t** : float

The current time (sec)

kwargs : dict

Additional non-positional parameters.

mlpy.modules.Module.save

`Module.save(filename)`

Save the current state of the module to file.

Parameters **filename** : str

The name of the file to save to.

mlpy.modules.Module.update

`Module.update(dt)`

Update the module at every delta time step dt.

Parameters **dt** : float

The elapsed time (sec)

Patterns

Borg	Class ensuring that all instances share the same state.
Observable	The observable base class.
Listener	The listener interface.

mlpy.modules.patterns.Borg

class `mlpy.modules.patterns.`**`Borg`**

Bases: `object`

Class ensuring that all instances share the same state.

The borg design pattern ensures that all instances of a class share the same state and provides a global point of access to the shared state.

Rather than enforcing that only ever one instance of a class exists, the borg design pattern ensures that all instances share the same state. That means every the values of the member variables are the same for every instance of the borg class.

The member variables which are to be shared among all instances must be declared as class variables.

See also:

`Singleton`

Notes

One side effect is that if you subclass a borg, the objects all have the same state, whereas subclass objects of a singleton have different states.

Examples

Create a borg class:

```
>>> from mlpy.modules.patterns import Borg
>>> class MyClass(Borg):
>>>     shared_variable = None
```

Note:

Project: Code from ActiveState.
Code author: Alex Naanou
License: CC-Wiki

mlpy.modules.patterns.Observable

class `mlpy.modules.patterns.`**`Observable`**`(`*mid=None*`)`

Bases: *`mlpy.modules.UniqueModule`*

The observable base class.

The observable keeps a record of all listeners and notifies them of the events they have subscribed to by calling *`Listener.notify`*.

The listeners are notified by calling *`dispatch`*. Listeners are notified if either the event that is being dispatched is `None` or the listener has subscribed to a `None` event, or the name of the event the listener has subscribed to is equal to the name of the dispatching event.

An event is an object consisting of the *source*; i.e. the observable, the event *name*, and the event *data* to be passed to the listener.

Parameters **mid** : str

The module's unique identifier

Examples

```
>>> from mlpy.modules.patterns import Observable
>>>
>>> class MyObservable(Observable):
>>>     pass
>>>
>>> o = MyObservable()
```

This defines the observable *MyObservable* and creates an instance of it.

```
>>> from mlpy.modules.patterns import Listener
>>>
>>> class MyListener(Listener):
>>>
>>>     def notify(self, event):
>>>         print "I have been notified!"
>>>
>>> l = MyListener(o, "test")
```

This defines the listener *MyListener* that when notified will print the same text to the console regardless of which event has been thrown (as long as the listener has subscribed to the event). Then an instance of MyListener is created that subscribes to the event *test* of *MyObservable*.

When the event *test* is dispatched by the observable, the listener is notified and the text is printed on the stdout:

```
>>> o.dispatch("test", **{})
I have been notified!
```

Attributes

`mid`	The module's unique identifier.

mlpy.modules.patterns.Observable.mid

`Observable.`**`mid`**

The module's unique identifier.

Returns str :

The module's unique identifier

Methods

`dispatch`(name, **attrs)	Dispatch the event to all listeners.
`load`(filename)	Load the state of the module from file.
`save`(filename)	Save the current state of the module to file.

Continued on next page

Table 17.8 – continued from previous page

subscribe(listener[, events])	Subscribe to the observable.
unsubscribe(listener)	Unsubscribe from the observable.

mlpy.modules.patterns.Observable.dispatch

`Observable.`**`dispatch`**(*name*, ***attrs*)

Dispatch the event to all listeners.

Parameters **name** : str

The name of the event to dispatch.

attrs : dict

The information send to the listeners.

mlpy.modules.patterns.Observable.load

`Observable.`**`load`**(*filename*)

Load the state of the module from file.

Parameters **filename** : str

The name of the file to load from.

Notes

This is a class method, it can be accessed without instantiation.

mlpy.modules.patterns.Observable.save

`Observable.`**`save`**(*filename*)

Save the current state of the module to file.

Parameters **filename** : str

The name of the file to save to.

mlpy.modules.patterns.Observable.subscribe

`Observable.`**`subscribe`**(*listener*, *events=None*)

Subscribe to the observable.

Parameters **listener** : Listener

The listener instance.

events : str or list[str] or tuple[str] or None

The event names the listener wants to be notified about.

mlpy.modules.patterns.Observable.unsubscribe

Observable.**unsubscribe**(*listener*)

Unsubscribe from the observable.

The listener is removed from the list of listeners.

Parameters **listener** : Listener

The listener instance.

mlpy.modules.patterns.Listener

class mlpy.modules.patterns.**Listener**(*o=None*, *events=None*)

Bases: object

The listener interface.

A listener subscribes to an observable identifying the events the listener is interested in. The observable calls *notify* to send relevant event information.

Parameters **o** : Observable, optional

The observable instance.

events : str or list[str], optional

The event names the listener wants to be notified about.

Notes

Every class inheriting from Listener must implement *notify*, which defines what to do with the information send by the observable.

Examples

```
>>> from mlpy.modules.patterns import Observable
>>>
>>> class MyObservable(Observable):
>>>     pass
>>>
>>> o = MyObservable()
```

This defines the observable *MyObservable* and creates an instance of it.

```
>>> from mlpy.modules.patterns import Listener
>>>
>>> class MyListener(Listener):
>>>
>>>     def notify(self, event):
>>>         print "I have been notified!"
>>>
>>> l = MyListener(o, "test")
```

This defines the listener *MyListener* that when notified will print the same text to the console regardless of which event has been thrown (as long as the listener has subscribed to the event). Then an instance of MyListener is created that subscribes to the event *test* of *MyObservable*.

When the event *test* is dispatched by the observable, the listener is notified and the text is printed on the stdout:

```
>>> o.dispatch("test", **{})
I have been notified!
```

Methods

`notify`(event)	Notification from the observable.

mlpy.modules.patterns.Listener.notify

`Listener.`**`notify`**`(`*event*`)`

Notification from the observable.

Parameters **event** : Observable.Event

The event object dispatched by the observable consisting of *source*; i.e. the observable, the event *name*, and the event *data*.

Raises **NotImplementedError**

If the child class does not implement this function.

Notes

This is an abstract method and *must* be implemented by its deriving class.

Meta classes

`Singleton`	Metaclass ensuring only one instance of the class exists.
`RegistryInterface`	Metaclass registering all subclasses derived from a given class.

mlpy.modules.patterns.Singleton

class `mlpy.modules.patterns.`**`Singleton`**

Bases: `type`

Metaclass ensuring only one instance of the class exists.

The singleton pattern ensures that a class has only one instance and provides a global point of access to that instance.

See also:

`Borg`

Notes

To define a class as a singleton include the `__metaclass__` directive.

Examples

Define a singleton class:

```
>>> from mlpy.modules.patterns import Singleton
>>> class MyClass(object):
>>>     __metaclass__ = Singleton
```

Note:

Project: Code from StackOverflow.
Code author: theheadofabroom
License: CC-Wiki

Methods

__call__(*args, **kwargs)	Returns instance to object.

mlpy.modules.patterns.Singleton.__call__

`Singleton.__call__`(**args*, ***kwargs*)
Returns instance to object.

mlpy.modules.patterns.RegistryInterface

class `mlpy.modules.patterns.`**`RegistryInterface`**(*name*, *bases*, *dct*)
Bases: `type`

Metaclass registering all subclasses derived from a given class.

The registry interface adds every class derived from a given class to its registry dictionary. The *registry* attribute is a class variable and can be accessed anywhere. Therefore, this interface can be used to find all subclass of a given class.

One use case are factory classes.

Examples

Create a registry class:

```
>>> from mlpy.modules.patterns import RegistryInterface
>>> class MyRegistryClass(object):
...     __metaclass__ = RegistryInterface
```

Note:

Project: Code from A Primer on Python Metaclasses.
Code author: Jake Vanderplas
License: CC-Wiki

Attributes

registry	(list) List of all classes deriving from a registry class.

Methods

`__init__`(name, bases, dct)	Register the deriving class on instantiation.

CHAPTER 18

Optimization tools (`mlpy.optimize`)

Algorithms

EM	Expectation-Maximization module base class.

mlpy.optimize.algorithms.EM

class `mlpy.optimize.algorithms.`**`EM`**(*n_iter=None*, *thresh=None*, *verbose=None*)

Bases: `object`

Expectation-Maximization module base class.

Representation of the expectation-maximization (EM) model. This class allows for the execution of the expectation- maximization algorithm by providing functionality for random restarts and convergence checking.

See the instance documentation for details specific to a particular implementation of the EM algorithm.

Parameters **n_iter** : int, optional

The number of iterations to perform. Default is 100.

thresh : float, optional

The convergence threshold. Default is 1e-4.

verbose : bool, optional

Controls if debug information is printed to the console. Default is False.

See also:

HMM, *GMM*

Notes

Classes that deriving from the EM base class must overwrite the following private functions:

_initialize(obs, init_count) Perform initialization before entering the EM algorithm. The expected parameters are:

obs [array_like, shape (*n*, *ni*, *nfeatures*)] List of observation sequences, where *n* is the number of sequences, *ni* is the length of the i_th observation, and each observation has *nfeatures* features.

init_count [int] Restart counter.

_estep(obs) Perform the expectation step of the EM algorithm and return the log likelihood of the observation *obs*. The expected parameters are:

obs [array_like, shape (*n*, *ni*, *nfeatures*)] List of observation sequences, where *n* is the number of sequences, *ni* is the length of the i_th observation, and each observation has *nfeatures* features.

_mstep() Perform maximization step of the EM algorithm.

Optionally, the private function `_plot` can be overwritten to visualize the results at each iteration. The `_plot` function is called by the EM algorithm before the maximization step is performed.

The deriving class must call the private method `_em(x, n_init=None)` to initiate the the EM algorithm. Pass the following parameters:

x [array_like, shape (*n*, *ni*, *ndim*)] List of data sequences, where *n* is the number of sequences, *ni* is the length of the i_th sequence, and each data point in the sequence has *ndim* dimensions.

n_init [int, optional] Number of restarts to prevent getting stuck in a local minimum. Default is 1.

The function returns the log likelihood of the data sequences *x*.

Examples

```
>>> from mlpy.optimize.algorithms import EM
>>>
>>> class MyEM(EM):
...     def _initialize(self, obs, init_count):
...         pass
...
...     def _estep(self, obs):
...         pass
...
...     def _mstep(self):
...         pass
...
...     def _plot(self):
...         pass
...
...     def fit(self, x):
...         return self._em(x, n_init=5)
...
```

This creates a new class capable of performing the expectation-maximization algorithm.

Note: Adapted from:

Utilities

is_converged	Check if an objective function has converged.

mlpy.optimize.utils.is_converged

`mlpy.optimize.utils.`**`is_converged`**(*fval*, *prev_fval*, *thresh=0.0001*, *warn=False*)

Check if an objective function has converged.

Parameters **fval** : float

The current value.

prev_fval : float

The previous value.

thresh : float

The convergence threshold.

warn : bool

Flag indicating whether to warn the user when the fval decreases.

Returns bool :

Flag indicating whether the objective function has converged or not.

Notes

The test returns true if the slope of the function falls below the threshold; i.e.

$$\frac{|f(t) - f(t-1)|}{\text{avg}} < \text{thresh},$$

where

$$\text{avg} = \frac{|f(t)| + |f(t+1)|}{2}$$

Note: Ported from Matlab:

CHAPTER 19

Planning tools (`mlpy.planners`)

Explorers

`ExplorerFactory`	The explorer factory.
`IExplorer`	The explorer interface class.

mlpy.planners.explorers.ExplorerFactory

class `mlpy.planners.explorers.`**`ExplorerFactory`**

Bases: `object`

The explorer factory.

An instance of an explorer can be created by passing the explorer type.

Examples

```
>>> from mlpy.planners.explorers import ExplorerFactory
>>> ExplorerFactory.create('egreedyexplorer', 0.8)
```

This creates a :class:.EGreedyExplorer‘ instance with epsilon set to 0.8.

```
>>> ExplorerFactory.create('softmaxexplorer', tau=3.0, decay=0.4)
```

This creates a *`SoftmaxExplorer`* instance with tau set to 3.0 and decay set to 0.4

Methods

create(_type, *args, **kwargs)	Create an explorer of the given type .

mlpy.planners.explorers.ExplorerFactory.create

static ExplorerFactory.**create**(*_type*, **args*, ***kwargs*)

Create an explorer of the given type .

Parameters **_type** : str

The explorer type. Valid explorer types:

egreedyexplorer With ϵ probability, a random action is chosen, otherwise the action resulting in the highest q-value is selected. An *EGreedyExplorer* is created.

softmaxexplorer The softmax explorer varies the action probability as a graded function of estimated value. The greedy action is still given the highest selection probability, but all the others are ranked and weighted according to their value estimates. A *SoftmaxExplorer* is created.

args : tuple

Positional arguments passed to the class of the given type for initialization.

kwargs : dict

Non-positional arguments passed to the class of the given type for initialization.

Returns IExplorer :

An explorer instance of the given type.

mlpy.planners.explorers.IExplorer

class mlpy.planners.explorers.**IExplorer**

Bases: object

The explorer interface class.

The explorer class executes the exploration policy.

Notes

All explorers should derive from this class.

Methods

activate()	Turn on exploration mode.
choose_action(*args, **kwargs)	Choose the next action according to the exploration strategy.
deactivate()	Turn off exploration mode.

mlpy.planners.explorers.IExplorer.activate

`IExplorer.activate()`
Turn on exploration mode.

mlpy.planners.explorers.IExplorer.choose_action

`IExplorer.choose_action(*args, **kwargs)`
Choose the next action according to the exploration strategy.

Parameters **args** : tuple

Positional arguments.

kwargs : dict

Non-positional arguments.

Returns Action :

The next action to taken.

Raises **NotImplementedError:**

If the child class does not implement this function.

mlpy.planners.explorers.IExplorer.deactivate

`IExplorer.deactivate()`
Turn off exploration mode.

Discrete explorers

DiscreteExplorer	The discrete explorer base class.
EGreedyExplorer	The ϵ-greedy explorer.
SoftmaxExplorer	The softmax explorer.

mlpy.planners.explorers.discrete.DiscreteExplorer

class `mlpy.planners.explorers.discrete.DiscreteExplorer`
Bases: *mlpy.planners.explorers.IExplorer*

The discrete explorer base class.

The explorer class executes the exploration policy by choosing a next action based on the current qvalues of the state-action pairs.

Notes

All discrete explorers should derive from this class.

Methods

`activate`()	Turn on exploration mode.
`choose_action`(actions, qvalues)	Choose the next action according to the exploration strategy.
`deactivate`()	Turn off exploration mode.

mlpy.planners.explorers.discrete.DiscreteExplorer.activate

`DiscreteExplorer.`**`activate`**`()`

Turn on exploration mode.

mlpy.planners.explorers.discrete.DiscreteExplorer.choose_action

`DiscreteExplorer.`**`choose_action`**`(`*actions*, *qvalues*`)`

Choose the next action according to the exploration strategy.

Parameters **actions** : list[Actions]

The available actions.

qvalues : list[float]

The q-value for each action.

Returns Action :

The action with maximum q-value that can be taken from the given state.

mlpy.planners.explorers.discrete.DiscreteExplorer.deactivate

`DiscreteExplorer.`**`deactivate`**`()`

Turn off exploration mode.

mlpy.planners.explorers.discrete.EGreedyExplorer

class `mlpy.planners.explorers.discrete.`**`EGreedyExplorer`**`(`*epsilon=None*, *decay=None*`)`

Bases: `mlpy.planners.explorers.discrete.DiscreteExplorer`

The ϵ-greedy explorer.

The ϵ-greedy explorer policy chooses as next action the action with the highest q-value, however with ϵ-probability a random action is chosen to drive exploration of unknown states.

Parameters **epsilon** : float, optional

The ϵ probability. Default is 0.5.

decay : float, optional

The value by which ϵ decays. This value should be between 0 and 1. The probability ϵ to decreases over time with a factor of *decay*. Set this value to 1 if ϵ should remain the same throughout the experiment. Default is 1.

Methods

`activate`()	Turn on exploration mode.
`choose_action`(actions, qvalues)	Choose the next action.
`deactivate`()	Turn off exploration mode.

mlpy.planners.explorers.discrete.EGreedyExplorer.activate

EGreedyExplorer.**activate**()

Turn on exploration mode.

mlpy.planners.explorers.discrete.EGreedyExplorer.choose_action

EGreedyExplorer.**choose_action**(*actions*, *qvalues*)

Choose the next action.

With ϵ probability, a random action is chosen, otherwise the action resulting in the highest q-value is selected.

Parameters **actions** : list[Actions]

The available actions.

qvalues : list[float]

The q-value for each action.

Returns Action :

The action with maximum qvalue that can be taken from the given state.

mlpy.planners.explorers.discrete.EGreedyExplorer.deactivate

EGreedyExplorer.**deactivate**()

Turn off exploration mode.

mlpy.planners.explorers.discrete.SoftmaxExplorer

class mlpy.planners.explorers.discrete.**SoftmaxExplorer**(*tau=None*, *decay=None*)

Bases: *mlpy.planners.explorers.discrete.DiscreteExplorer*

The softmax explorer.

The softmax explorer varies the action probability as a graded function of estimated value. The greedy action is still given the highest selection probability, but all the others are ranked and weighted according to their value estimates.

Parameters **tau** : float, optional

The temperature value. Default is 2.0.

decay : float, optional

The value by which τ decays. This value should be between 0 and 1. The temperature τ to decrease over time with a factor of *decay*. Set this value to 1 if τ should remain the same throughout the experiment. Default is 1.

Notes

The softmax function implemented uses the Gibbs distribution. It chooses action *a* on the *t*-th play with probability:

$$\frac{e^{Q_t(a)/\tau}}{\sum_{b=1}^{n} e^{Q_t(b)/\tau}}$$

where τ is a positive parameter called the *temperature*. High temperatures cause all actions to be equiprobable. Low temperatures cause a greater difference in the selection probability. For τ close to zero, the action selection because the same as greedy.

Methods

`activate`()	Turn on exploration mode.
`choose_action`(actions, qvalues)	Choose the next action.
`deactivate`()	Turn off exploration mode.

mlpy.planners.explorers.discrete.SoftmaxExplorer.activate

`SoftmaxExplorer.`**`activate`**`()`

Turn on exploration mode.

mlpy.planners.explorers.discrete.SoftmaxExplorer.choose_action

`SoftmaxExplorer.`**`choose_action`**`(`*actions*, *qvalues*`)`

Choose the next action.

Choose the next action according to the Gibbs distribution.

Parameters **actions** : list[Actions]

The available actions.

qvalues : list[float]

The q-value for each action.

Returns Action :

The action with maximum q-value that can be taken from the given state.

mlpy.planners.explorers.discrete.SoftmaxExplorer.deactivate

`SoftmaxExplorer.`**`deactivate`**`()`

Turn off exploration mode.

Planners

`IPlanner`	The planner interface class.

mlpy.planners.IPlanner

class `mlpy.planners.`**`IPlanner`**(*explorer=None*)

Bases: *`mlpy.modules.UniqueModule`*

The planner interface class.

Parameters **explorer** : Explorer

The exploration strategy to employ. Available explorers are:

`EGreedyExplorer` With ϵ probability, a random action is chosen, otherwise the action resulting in the highest q-value is selected.

`SoftmaxExplorer` The softmax explorer varies the action probability as a graded function of estimated value. The greedy action is still given the highest selection probability, but all the others are ranked and weighted according to their value estimates.

Attributes

`mid`	The module's unique identifier.

mlpy.planners.IPlanner.mid

`IPlanner.`**`mid`**

The module's unique identifier.

Returns str :

The module's unique identifier

Methods

`activate_exploration`()	Turn the explorer on.
`create_policy`([func])	Creates a policy (i.e., a state-action association).
`deactivate_exploration`()	Turn the explorer off.
`get_best_action`(state)	Choose the best next action for the agent to take.
`get_next_action`(state[, use_policy])	Returns the optimal action for a state according to the current policy.
`load`(filename)	Load the state of the module from file.
`plan`()	Plan for the optimal policy.
`save`(filename)	Save the current state of the module to file.
`visualize`()	Visualize of the planning data.

mlpy.planners.IPlanner.activate_exploration

`IPlanner.`**`activate_exploration`**()

Turn the explorer on.

mlpy.planners.IPlanner.create_policy

`IPlanner.`**`create_policy`**(*func=None*)

Creates a policy (i.e., a state-action association).

Parameters **func** : callable, optional

A callback function for mixing policies.

mlpy.planners.IPlanner.deactivate_exploration

`IPlanner.`**`deactivate_exploration`**()

Turn the explorer off.

mlpy.planners.IPlanner.get_best_action

`IPlanner.`**`get_best_action`**(*state*)

Choose the best next action for the agent to take.

Parameters **state** : State

The state for which to choose the action for.

Returns Action :

The best action.

Raises **NotImplementedError**

If the child class does not implement this function.

mlpy.planners.IPlanner.get_next_action

`IPlanner.`**`get_next_action`**(*state*, *use_policy=False*)

Returns the optimal action for a state according to the current policy.

Parameters **state** : State

The state for which to choose the next action for.

use_policy : bool, optional

When using a policy the next action is chosen according to the current policy, otherwise the best action is selected. Default is False.

Returns Action :

The next action.

mlpy.planners.IPlanner.load

`IPlanner.`**`load`**(*filename*)

Load the state of the module from file.

Parameters **filename** : str

The name of the file to load from.

Notes

This is a class method, it can be accessed without instantiation.

mlpy.planners.IPlanner.plan

`IPlanner.plan()`

Plan for the optimal policy.

Raises NotImplementedError

If the child class does not implement this function.

mlpy.planners.IPlanner.save

`IPlanner.save(`*filename*`)`

Save the current state of the module to file.

Parameters filename : str

The name of the file to save to.

mlpy.planners.IPlanner.visualize

`IPlanner.visualize()`

Visualize of the planning data.

Raises NotImplementedError

If the child class does not implement this function.

Discrete planners

`ValueIteration`	Planning through value Iteration.

mlpy.planners.discrete.ValueIteration

class `mlpy.planners.discrete.ValueIteration(`*model, explorer=None, gamma=None, ignore_unreachable=False*`)`

Bases: `mlpy.planners.IPlanner`

Planning through value Iteration.

Parameters model : DiscreteModel

The Markov decision model.

explorer : Explorer, optional

The exploration strategy to employ. Available explorers are:

`EGreedyExplorer` With ϵ probability, a random action is chosen, otherwise the action resulting in the highest q-value is selected.

`SoftmaxExplorer` The softmax explorer varies the action probability as a graded function of estimated value. The greedy action is still given the highest selection probability, but all the others are ranked and weighted according to their value estimates.

By default no explorer is used and the greedy action is chosen.

gamma : float, optional

The discount factor. Default is 0.9.

ignore_unreachable : bool, optional

Whether to ignore unreachable states or not. Unreachability is determined by how many steps a state is are away from the closest neighboring state. Default is False.

Raises AttributeError

If both the Markov model and the planner define an explorer. Only one explorer can be specified.

Attributes

mid	The module's unique identifier.
model	The Markov decision process model.

mlpy.planners.discrete.ValueIteration.mid

`ValueIteration.`**`mid`**

The module's unique identifier.

Returns str :

The module's unique identifier

mlpy.planners.discrete.ValueIteration.model

`ValueIteration.`**`model`**

The Markov decision process model.

The Markov decision process model contain information about the states, actions, and their transitions and the reward function.

Returns IMDPModel :

The model.

Methods

activate_exploration()	Turn the explorer on.
create_policy([func])	Creates a policy (i.e., a state-action association).
deactivate_exploration()	Turn the explorer off.
get_best_action(state)	Choose the best next action for the agent to take.
get_next_action(state[, use_policy])	Returns the optimal action for a state according to the current policy.

Continued on next page

Table 19.13 – continued from previous page

load(filename)	Load the state of the module from file.
plan()	Plan for the optimal policy.
save(filename)	Save the current state of the module to file.
visualize()	Visualize of the planning data.

mlpy.planners.discrete.ValueIteration.activate_exploration

ValueIteration.**activate_exploration**()

Turn the explorer on.

mlpy.planners.discrete.ValueIteration.create_policy

ValueIteration.**create_policy**(*func=None*)

Creates a policy (i.e., a state-action association).

Parameters **func** : callable, optional

A callback function for mixing policies.

mlpy.planners.discrete.ValueIteration.deactivate_exploration

ValueIteration.**deactivate_exploration**()

Turn the explorer off.

mlpy.planners.discrete.ValueIteration.get_best_action

ValueIteration.**get_best_action**(*state*)

Choose the best next action for the agent to take.

Parameters **state** : State

The state for which to choose the action for.

Returns Action :

The best action.

mlpy.planners.discrete.ValueIteration.get_next_action

ValueIteration.**get_next_action**(*state*, *use_policy=False*)

Returns the optimal action for a state according to the current policy.

Parameters **state** : State

The state for which to choose the next action for.

use_policy : bool, optional

When using a policy the next action is chosen according to the current policy, otherwise the best action is selected. Default is False.

Returns Action :

The next action.

mlpy.planners.discrete.ValueIteration.load

`ValueIteration.load(filename)`

Load the state of the module from file.

Parameters **filename** : str

The name of the file to load from.

Notes

This is a class method, it can be accessed without instantiation.

mlpy.planners.discrete.ValueIteration.plan

`ValueIteration.plan()`

Plan for the optimal policy.

Perform value iteration and build the Q-table.

mlpy.planners.discrete.ValueIteration.save

`ValueIteration.save(filename)`

Save the current state of the module to file.

Parameters **filename** : str

The name of the file to save to.

mlpy.planners.discrete.ValueIteration.visualize

`ValueIteration.visualize()`

Visualize of the planning data.

The results in the Q table are visualized via a heat map.

CHAPTER 20

Search tools (`mlpy.search`)

`Node`	The node class.
`ISearch`	The search class interface.

mlpy.search.Node

class `mlpy.search.`**`Node`** (*state*, *parent=None*, *action=None*, *cost=0*)

Bases: `object`

The node class.

The node within the graph that is being built while searching for the optimal path

Parameters **state** : int or tuple[int]

The state.

parent : Node

The parent node.

action : str

The action performed in the state.

cost : float

The path cost to the state.

Attributes

`depth`	The depth of the node.
`g`	The path cost.

Continued on next page

Table 20.2 – continued from previous page

parent	The node's parent.
state	The state associated the node.

mlpy.search.Node.depth

`Node.`**`depth`**

The depth of the node.

The number of steps between the start node and this node.

Returns int :

The depth.

mlpy.search.Node.g

`Node.`**`g`**

The path cost.

Returns float :

The cost.

mlpy.search.Node.parent

`Node.`**`parent`**

The node's parent.

Returns Node :

The parent node.

mlpy.search.Node.state

`Node.`**`state`**

The state associated the node.

Returns int or tuple[int] :

The state.

Methods

expand(task)	Expands a node's neighbors.

mlpy.search.Node.expand

`Node.`**`expand`**(*task*)

Expands a node's neighbors.

Parameters **task** : SearchTask

A search task instance.

mlpy.search.ISearch

class `mlpy.search.`**`ISearch`**(*task*)

Bases: *`mlpy.modules.UniqueModule`*

The search class interface.

Attributes

`mid`	The module's unique identifier.

mlpy.search.ISearch.mid

`ISearch.`**`mid`**

The module's unique identifier.

Returns str :

The module's unique identifier

Methods

`get_path`()	Return the optimal path from the start to the goal node.
`load`(filename)	Load the state of the module from file.
`save`(filename)	Save the current state of the module to file.
`save_path`(path, filename)	Save the path to file.
`search`()	Search for the optimal path.

mlpy.search.ISearch.get_path

`ISearch.`**`get_path`**()

Return the optimal path from the start to the goal node.

Returns list[int or tuple[int]] :

The optimal path.

mlpy.search.ISearch.load

`ISearch.`**`load`**(*filename*)

Load the state of the module from file.

Parameters **filename** : str

The name of the file to load from.

Notes

This is a class method, it can be accessed without instantiation.

mlpy.search.ISearch.save

`ISearch.`**`save`**`(`*filename*`)`

Save the current state of the module to file.

Parameters **filename** : str

The name of the file to save to.

mlpy.search.ISearch.save_path

`ISearch.`**`save_path`**`(`*path*, *filename*`)`

Save the path to file.

Parameters **path** : list[int or tuple[int]]

The found path.

filename : str

The filename to save the path to.

mlpy.search.ISearch.search

`ISearch.`**`search`**`()`

Search for the optimal path.

Informed Search

`AStar`	A* algorithm.

mlpy.search.informed.AStar

class `mlpy.search.informed.`**`AStar`**`(`*task*`)`

Bases: *`mlpy.search.ISearch`*

A* algorithm.

This class implements the A* algorithm

Parameters **task** : SearchTask

The search task to perform

Attributes

`mid`	The module's unique identifier.

mlpy.search.informed.AStar.mid

`AStar.`**`mid`**

The module's unique identifier.

Returns str :

The module's unique identifier

Methods

`get_path()`	Return the optimal path from the start to the goal node.
`get_results()`	Display the search results visually.
`load(filename)`	Load the state of the module from file.
`save(filename)`	Save the current state of the module to file.
`save_path(path, filename)`	Save the path to file.
`search()`	Perform the search.

mlpy.search.informed.AStar.get_path

`AStar.get_path()`

Return the optimal path from the start to the goal node.

Returns list[int or tuple[int]] :

The optimal path.

mlpy.search.informed.AStar.get_results

`AStar.get_results()`

Display the search results visually.

mlpy.search.informed.AStar.load

`AStar.load(filename)`

Load the state of the module from file.

Parameters **filename** : str

The name of the file to load from.

Notes

This is a class method, it can be accessed without instantiation.

mlpy.search.informed.AStar.save

`AStar.save(filename)`

Save the current state of the module to file.

Parameters **filename** : str

The name of the file to save to.

mlpy.search.informed.AStar.save_path

`AStar.save_path(path, filename)`

Save the path to file.

Parameters **path** : list[int or tuple[int]]

The found path.

filename : str

The filename to save the path to.

mlpy.search.informed.AStar.search

`AStar.search()`

Perform the search.

Performs the actual search for the optimal path using the A* algorithm

CHAPTER 21

Statistical functions (`mlpy.stats`)

Discrete distributions

mlpy.stats.nonuniform

`mlpy.stats.`**`nonuniform`** **= <Mock name='mock.rv_discrete' id='140284521135952'>**

Create a new *Mock* object. *Mock* takes several optional arguments that specify the behaviour of the Mock object:

- *spec*: This can be either a list of strings or an existing object (a class or instance) that acts as the specification for the mock object. If you pass in an object then a list of strings is formed by calling dir on the object (excluding unsupported magic attributes and methods). Accessing any attribute not in this list will raise an *AttributeError*.

  If *spec* is an object (rather than a list of strings) then *mock.__class__* returns the class of the spec object. This allows mocks to pass *isinstance* tests.

- *spec_set*: A stricter variant of *spec*. If used, attempting to *set* or get an attribute on the mock that isn't on the object passed as *spec_set* will raise an *AttributeError*.

- *side_effect*: A function to be called whenever the Mock is called. See the *side_effect* attribute. Useful for raising exceptions or dynamically changing return values. The function is called with the same arguments as the mock, and unless it returns *DEFAULT*, the return value of this function is used as the return value.

  Alternatively *side_effect* can be an exception class or instance. In this case the exception will be raised when the mock is called.

  If *side_effect* is an iterable then each call to the mock will return the next value from the iterable. If any of the members of the iterable are exceptions they will be raised instead of returned.

- *return_value*: The value returned when the mock is called. By default this is a new Mock (created on first access). See the *return_value* attribute.

- *wraps*: Item for the mock object to wrap. If *wraps* is not None then calling the Mock will pass the call through to the wrapped object (returning the real result). Attribute access on the mock will return a Mock object that wraps the corresponding attribute of the wrapped object (so attempting to access an attribute that doesn't exist will raise an *AttributeError*).

If the mock has an explicit *return_value* set then calls are not passed to the wrapped object and the *return_value* is returned instead.

- *name*: If the mock has a name then it will be used in the repr of the mock. This can be useful for debugging. The name is propagated to child mocks.

Mocks can also be called with arbitrary keyword arguments. These will be used to set attributes on the mock after it is created.

mlpy.stats.gibbs

`mlpy.stats.`**`gibbs`** **= <Mock name='mock.rv_discrete' id='140284521135952'>**

Create a new *Mock* object. *Mock* takes several optional arguments that specify the behaviour of the Mock object:

- *spec*: This can be either a list of strings or an existing object (a class or instance) that acts as the specification for the mock object. If you pass in an object then a list of strings is formed by calling dir on the object (excluding unsupported magic attributes and methods). Accessing any attribute not in this list will raise an *AttributeError*.

  If *spec* is an object (rather than a list of strings) then *mock.__class__* returns the class of the spec object. This allows mocks to pass *isinstance* tests.

- *spec_set*: A stricter variant of *spec*. If used, attempting to *set* or get an attribute on the mock that isn't on the object passed as *spec_set* will raise an *AttributeError*.

- *side_effect*: A function to be called whenever the Mock is called. See the *side_effect* attribute. Useful for raising exceptions or dynamically changing return values. The function is called with the same arguments as the mock, and unless it returns *DEFAULT*, the return value of this function is used as the return value.

  Alternatively *side_effect* can be an exception class or instance. In this case the exception will be raised when the mock is called.

  If *side_effect* is an iterable then each call to the mock will return the next value from the iterable. If any of the members of the iterable are exceptions they will be raised instead of returned.

- *return_value*: The value returned when the mock is called. By default this is a new Mock (created on first access). See the *return_value* attribute.

- *wraps*: Item for the mock object to wrap. If *wraps* is not None then calling the Mock will pass the call through to the wrapped object (returning the real result). Attribute access on the mock will return a Mock object that wraps the corresponding attribute of the wrapped object (so attempting to access an attribute that doesn't exist will raise an *AttributeError*).

  If the mock has an explicit *return_value* set then calls are not passed to the wrapped object and the *return_value* is returned instead.

- *name*: If the mock has a name then it will be used in the repr of the mock. This can be useful for debugging. The name is propagated to child mocks.

Mocks can also be called with arbitrary keyword arguments. These will be used to set attributes on the mock after it is created.

`nonuniform`	A non-uniform discrete random variable.
`gibbs`	A Gibbs distribution discrete random variable.

Conditional distributions

mlpy.stats.conditional_normal

`mlpy.stats.`**`conditional_normal`** **= <mlpy.stats._conditional.conditional_normal_gen object>**

mlpy.stats.conditional_student

`mlpy.stats.`**`conditional_student`** **= <mlpy.stats._conditional.conditional_student_gen object>**

mlpy.stats.conditional_mix_normal

`mlpy.stats.`**`conditional_mix_normal`** **= <mlpy.stats._conditional.conditional_mix_normal_gen object>**

conditional_normal	Conditional Normal random variable.
conditional_student	Conditional Student random variable.
conditional_mix_normal	Conditional Mix-Normal random variable.

Multivariate distributions

mlpy.stats.multivariate_normal

`mlpy.stats.`**`multivariate_normal`** **= <mlpy.stats._multivariate.multivariate_normal_gen object>**

mlpy.stats.multivariate_student

`mlpy.stats.`**`multivariate_student`** **= <mlpy.stats._multivariate.multivariate_student_gen object>**

mlpy.stats.invwishart

`mlpy.stats.`**`invwishart`** **= <mlpy.stats._multivariate.invwishart_gen object>**

mlpy.stats.normal_invwishart

`mlpy.stats.`**`normal_invwishart`** **= <mlpy.stats._multivariate.normal_invwishart_gen object>**

multivariate_normal	Multivariate Normal random variable.
multivariate_student	Multivariate Student random variable.
invwishart	Inverse Wishart random variable.
normal_invwishart	Normal-Inverse Wishart random variable.

Statistical Models

mlpy.stats.models.markov

`mlpy.stats.models.`**`markov`** **= <mlpy.stats.models._basic.markov_gen object>**

`markov`	Markov model.

Mixture Models

`MixtureModel`	Mixture model base class.
`DiscreteMM`	Discrete mixture model class.
`GMM`	Gaussian mixture model class.
`StudentMM`	Student mixture model class.

mlpy.stats.models.mixture.MixtureModel

class `mlpy.stats.models.mixture.`**`MixtureModel`**(*ncomponents=1*, *prior=None*, *mix_prior=None*, *mix_weight=None*, *n_iter=None*, *thresh=None*, *verbose=None*)

Bases: `mlpy.optimize.algorithms.EM`

Mixture model base class.

Representation of a mixture model probability distribution. This class allows for easy evaluation of, sampling from, and maximum-likelihood estimation of the parameters of a distribution.

Parameters **ncomponents** : int, optional

Number of mixture components. Default is 1.

prior : normal_invwishart, optional

A `normal_invwishart` distribution.

mix_prior : float or array_like, shape (*ncomponents*,), optional

Prior mixture probabilities.

mix_weight : array_like, shape (*ncomponents*,), optional

Mixture weights.

n_iter : int, optional

Number of EM iterations to perform. Default is 100.

thresh : float, optional

Convergence threshold. EM iterations will stop when average gain in log-likelihood is below this threshold. Default is 1e-4.

verbose : bool, optional

Controls if debug information is printed to the console. Default is False.

Examples

```
>>> from mlpy.stats.models.mixture import GMM
```

```
>>> m = GMM()
```

Note: Adapted from Matlab:

Project: Probabilistic Modeling Toolkit for Matlab/Octave.
Copyright (2010) Kevin Murphy and Matt Dunham
License: MIT

Attributes

ncomponents	(int) Number of mixture components.
dim	(int) Dimensionality of the each component.
prior	(normal_invwishart) A `normal_invwishart` distribution.
mix_prior	(array_like, shape (*ncomponents*,)) Prior mixture probabilities.
mix_weight	(array_like, shape (*ncomponents*,)) Mixture weights.
cond_proba	(cond_rv_frozen) Conditional probability distribution.
n_iter	(int) Number of EM iterations to perform.
thresh	(float) Convergence threshold.
verbose	(bool) Controls if debug information is printed to the console.

Methods

`fit`(x[, n_init])	Fit the mixture model from the data *x*.
`predict`(x)	Predict label for data.
`predict_proba`(x)	Predict posterior probability of data under the model.
`sample`([size])	Generate random samples from the model.
`score`(x[, y])	Compute the log probability under the model.
`score_samples`(x)	Return the per-sample likelihood of the data under the model.

mlpy.stats.models.mixture.MixtureModel.fit

`MixtureModel.`**`fit`**(*x*, *n_init=1*)

Fit the mixture model from the data *x*.

Estimate model parameters with the expectation-maximization algorithm.

Parameters **x** : array_like, shape (*n*, *dim*)

List of dim-dimensional data points. Each row corresponds to a single data point.

n_init : int, optional

Number of random restarts to avoid a local minimum. Default is 1.

mlpy.stats.models.mixture.MixtureModel.predict

`MixtureModel.`**`predict`**(*x*)

Predict label for data.

Parameters **x** : array_like, shape (*size*, *dim*)

Returns **C** : array, shape = (*size*,)

mlpy.stats.models.mixture.MixtureModel.predict_proba

`MixtureModel.`**`predict_proba`**(*x*)

Predict posterior probability of data under the model.

Parameters **x** : array_like, shape (*size*, *dim*)

Returns **responsibilities** : array_like, shape = (*nsamples*, *ncomponents*)

Returns the probability of the sample for each Gaussian (state) in the model.

mlpy.stats.models.mixture.MixtureModel.sample

`MixtureModel.`**`sample`**(*size=1*)

Generate random samples from the model.

Parameters **size** : int, optional

Number of samples to generate. Default is 1.

Returns **x** : array_like, shape (*size*, *dim*)

List of samples

Raises **NotImplementedError**

If the child class does not implement this function.

mlpy.stats.models.mixture.MixtureModel.score

`MixtureModel.`**`score`**(*x*, *y=None*)

Compute the log probability under the model.

Parameters **x** : array_like, shape (size, dim)

List of dim-dimensional data points. Each row corresponds to a single data point.

y : Not used.

Returns **logp** : array_like, shape (*size*,)

Log probabilities of each data point in *x*.

mlpy.stats.models.mixture.MixtureModel.score_samples

`MixtureModel.`**`score_samples`**(*x*)

Return the per-sample likelihood of the data under the model.

Compute the log probability of x under the model and return the posterior distribution (responsibilities) of each mixture component for each element of x.

Parameters **x** : array_like, shape (*size*, *dim*)

List of *dim*-dimensional data points. Each row corresponds to a single data point.

Returns **responsibilities** : array_like, shape (*size*, *ncomponents*)

Posterior probabilities of each mixture component for each observation.

loglik : array_like, shape (size,)

Log probabilities of each data point in *x*.

Raises NotImplementedError

If the child class does not implement this function.

mlpy.stats.models.mixture.DiscreteMM

class `mlpy.stats.models.mixture.DiscreteMM`(*ncomponents=1*, *prior=None*, *mix_prior=None*, *mix_weight=None*, *transmat=None*, *alpha=None*, *n_iter=None*, *thresh=None*, *verbose=None*)

Bases: `mlpy.stats.models.mixture.MixtureModel`

Discrete mixture model class.

Representation of a discrete mixture model probability distribution. This class allows for easy evaluation of, sampling from, and maximum-likelihood estimation of the parameters of a distribution.

Parameters ncomponents : int, optional

Number of mixture components. Default is 1.

prior : normal_invwishart, optional

A `normal_invwishart` distribution.

mix_prior : float or array_like, shape (*ncomponents*,), optional

Prior mixture probabilities.

mix_weight : array_like, shape (*ncomponents*,), optional

Mixture weights.

transmat : array_like, shape (*ncomponents*, *ncomponents*), optional

Matrix of transition probabilities between states.

alpha : float

Value of Dirichlet prior on observations. Default is 1.1 (1=MLE)

n_iter : int, optional

Number of EM iterations to perform. Default is 100.

thresh : float, optional

Convergence threshold. EM iterations will stop when average gain in log-likelihood is below this threshold. Default is 1e-4.

verbose : bool, optional

Controls if debug information is printed to the console. Default is False.

Examples

```
>>> from mlpy.stats.models.mixture import DiscreteMM
```

```
>>> m = DiscreteMM()
```

Note: Adapted from Matlab:

Project: Probabilistic Modeling Toolkit for Matlab/Octave.
Copyright (2010) Kevin Murphy and Matt Dunham
License: MIT

Attributes

ncomponents	(int) Number of mixture components.
dim	(int) Dimensionality of the each component.
prior	(normal_invwishart) A `normal_invwishart` distribution.
mix_prior	(array_like, shape (*ncomponents*,)) Prior mixture probabilities.
mix_weight	(array_like, shape (*ncomponents*,)) Mixture weights.
transmat	(array_like, shape (*ncomponents*, *ncomponents*)) Matrix of transition probabilities between states.
alpha	(float) Value of Dirichlet prior on observations.
cond_proba	(cond_rv_frozen) Conditional probability distribution.
n_iter	(int) Number of EM iterations to perform.
thresh	(float) Convergence threshold.
verbose	(bool) Controls if debug information is printed to the console.

Methods

`fit`(x[, n_init])	Fit the mixture model from the data *x*.
`predict`(x)	Predict label for data.
`predict_proba`(x)	Predict posterior probability of data under the model.
`sample`([size])	Generate random samples from the model.
`score`(x[, y])	Compute the log probability under the model.
`score_samples`(x)	Return the per-sample likelihood of the data under the model.

mlpy.stats.models.mixture.DiscreteMM.fit

`DiscreteMM.`**`fit`**(*x*, *n_init=1*)

Fit the mixture model from the data *x*.

Estimate model parameters with the expectation-maximization algorithm.

Parameters **x** : array_like, shape (*n*, *dim*)

List of dim-dimensional data points. Each row corresponds to a single data point.

n_init : int, optional

Number of random restarts to avoid a local minimum. Default is 1.

mlpy.stats.models.mixture.DiscreteMM.predict

`DiscreteMM.`**`predict`**(*x*)

Predict label for data.

Parameters **x** : array_like, shape (*size*, *dim*)

Returns **C** : array, shape = (*size*,)

mlpy.stats.models.mixture.DiscreteMM.predict_proba

`DiscreteMM.`**`predict_proba`**(*x*)

Predict posterior probability of data under the model.

Parameters **x** : array_like, shape (*size*, *dim*)

Returns **responsibilities** : array_like, shape = (*nsamples*, *ncomponents*)

Returns the probability of the sample for each Gaussian (state) in the model.

mlpy.stats.models.mixture.DiscreteMM.sample

`DiscreteMM.`**`sample`**(*size=1*)

Generate random samples from the model.

Parameters **size** : int, optional

Number of samples to generate. Default is 1.

Returns **x** : array_like, shape (*size*, *dim*)

List of samples

Raises **NotImplementedError**

If the child class does not implement this function.

mlpy.stats.models.mixture.DiscreteMM.score

`DiscreteMM.`**`score`**(*x*, *y=None*)

Compute the log probability under the model.

Parameters **x** : array_like, shape (size, dim)

List of dim-dimensional data points. Each row corresponds to a single data point.

y : Not used.

Returns **logp** : array_like, shape (*size*,)

Log probabilities of each data point in *x*.

mlpy.stats.models.mixture.DiscreteMM.score_samples

`DiscreteMM.`**`score_samples`**(*x*)

Return the per-sample likelihood of the data under the model.

Compute the log probability of x under the model and return the posterior distribution (responsibilities) of each mixture component for each element of x.

Parameters **x** : array_like, shape (*size*, *dim*)

List of *dim*-dimensional data points. Each row corresponds to a single data point.

Returns **responsibilities** : array_like, shape (*size*, *ncomponents*)

Posterior probabilities of each mixture component for each observation.

loglik : array_like, shape (size,)

Log probabilities of each data point in *x*.

mlpy.stats.models.mixture.GMM

class `mlpy.stats.models.mixture.`**`GMM`**(*ncomponents=1*, *prior=None*, *mix_prior=None*, *mix_weight=None*, *mean=None*, *cov=None*, *n_iter=None*, *thresh=None*, *verbose=None*)

Bases: `mlpy.stats.models.mixture.MixtureModel`

Gaussian mixture model class.

Representation of a gaussian mixture model probability distribution. This class allows for easy evaluation of, sampling from, and maximum-likelihood estimation of the parameters of a distribution.

Parameters **ncomponents** : int, optional

Number of mixture components. Default is 1.

prior : normal_invwishart, optional

A `normal_invwishart` distribution.

mix_prior : float or array_like, shape (*ncomponents*,), optional

Prior mixture probabilities.

mix_weight : array_like, shape (*ncomponents*,), optional

Mixture weights.

mean : array, shape (*ncomponents*, *nfeatures*)

Mean parameters for each state.

cov : array, shape (*ncomponents*, *nfeatures*, *nfeatures*)

Covariance parameters for each state.

n_iter : int, optional

Number of EM iterations to perform. Default is 100.

thresh : float, optional

Convergence threshold. EM iterations will stop when average gain in log-likelihood is below this threshold. Default is 1e-4.

verbose : bool, optional

Controls if debug information is printed to the console. Default is False.

Examples

```
>>> from mlpy.stats.models.mixture import GMM
```

```
>>> m = GMM()
```

Note: Adapted from Matlab:

Project: Probabilistic Modeling Toolkit for Matlab/Octave.
Copyright (2010) Kevin Murphy and Matt Dunham
License: MIT

Attributes

`mean`	Mean parameters of the emission.
`cov`	Covariance parameters of the emission.

mlpy.stats.models.mixture.GMM.mean

GMM.**mean**
Mean parameters of the emission.

Returns array, shape (*ncomponents*, *nfeatures*) :

The mean parameters.

mlpy.stats.models.mixture.GMM.cov

GMM.**cov**
Covariance parameters of the emission.

array, shape (*ncomponents*, *nfeatures*, *nfeatures*): Covariance parameters.

ncomponents	(int) Number of mixture components.
dim	(int) Dimensionality of the each component.
prior	(normal_invwishart) A `normal_invwishart` distribution.
mix_prior	(array_like, shape (*ncomponents*,)) Prior mixture probabilities.
mix_weight	(array_like, shape (*ncomponents*,)) Mixture weights.
cond_proba	(conditional_normal) A `conditional_normal` probability distribution.
n_iter	(int) Number of EM iterations to perform.
thresh	(float) Convergence threshold.
verbose	(bool) Controls if debug information is printed to the console.

Methods

`fit`(x[, n_init])	Fit the mixture model from the data *x*.
`predict`(x)	Predict label for data.
`predict_proba`(x)	Predict posterior probability of data under the model.
`sample`([size])	Generate random samples from the model.
`score`(x[, y])	Compute the log probability under the model.

Continued on next page

Table 21.5 – continued from previous page

score_samples(x)	Return the per-sample likelihood of the data under the model.

mlpy.stats.models.mixture.GMM.fit

GMM.**fit**(*x*, *n_init=1*)

Fit the mixture model from the data *x*.

Estimate model parameters with the expectation-maximization algorithm.

Parameters **x** : array_like, shape (*n*, *dim*)

List of dim-dimensional data points. Each row corresponds to a single data point.

n_init : int, optional

Number of random restarts to avoid a local minimum. Default is 1.

mlpy.stats.models.mixture.GMM.predict

GMM.**predict**(*x*)

Predict label for data.

Parameters **x** : array_like, shape (*size*, *dim*)

Returns **C** : array, shape = (*size*,)

mlpy.stats.models.mixture.GMM.predict_proba

GMM.**predict_proba**(*x*)

Predict posterior probability of data under the model.

Parameters **x** : array_like, shape (*size*, *dim*)

Returns **responsibilities** : array_like, shape = (*nsamples*, *ncomponents*)

Returns the probability of the sample for each Gaussian (state) in the model.

mlpy.stats.models.mixture.GMM.sample

GMM.**sample**(*size=1*)

Generate random samples from the model.

Parameters **size** : int, optional

Number of samples to generate. Default is 1.

Returns **x** : array_like, shape (*size*, *dim*)

List of samples

Raises **NotImplementedError**

If the child class does not implement this function.

mlpy.stats.models.mixture.GMM.score

`GMM.`**`score`**(*x*, *y=None*)

Compute the log probability under the model.

Parameters **x** : array_like, shape (size, dim)

List of dim-dimensional data points. Each row corresponds to a single data point.

y : Not used.

Returns **logp** : array_like, shape (*size*,)

Log probabilities of each data point in *x*.

mlpy.stats.models.mixture.GMM.score_samples

`GMM.`**`score_samples`**(*x*)

Return the per-sample likelihood of the data under the model.

Compute the log probability of x under the model and return the posterior distribution (responsibilities) of each mixture component for each element of x.

Parameters **x** : array_like, shape (*size*, *dim*)

List of *dim*-dimensional data points. Each row corresponds to a single data point.

Returns **responsibilities** : array_like, shape (*size*, *ncomponents*)

Posterior probabilities of each mixture component for each observation.

loglik : array_like, shape (size,)

Log probabilities of each data point in *x*.

mlpy.stats.models.mixture.StudentMM

class `mlpy.stats.models.mixture.`**`StudentMM`**(*ncomponents=1*, *prior=None*, *mix_prior=None*, *mix_weight=None*, *mean=None*, *cov=None*, *df=None*, *n_iter=None*, *thresh=None*, *verbose=None*)

Bases: `mlpy.stats.models.mixture.GMM`

Student mixture model class.

Representation of a student mixture model probability distribution. This class allows for easy evaluation of, sampling from, and maximum-likelihood estimation of the parameters of a distribution.

Parameters **ncomponents** : int, optional

Number of mixture components. Default is 1.

prior : normal_invwishart, optional

A `normal_invwishart` distribution.

mix_prior : float or array_like, shape (*ncomponents*,), optional

Prior mixture probabilities.

mix_weight : array_like, shape (*ncomponents*,), optional

Mixture weights.

mean : array, shape (*ncomponents*, *nfeatures*)

Mean parameters for each state.

cov : array, shape (*ncomponents*, *nfeatures*, *nfeatures*)

Covariance parameters for each state.

df : array, shape (*ncomponents*,)

Degrees of freedom.

n_iter : int, optional

Number of EM iterations to perform. Default is 100.

thresh : float, optional

Convergence threshold. EM iterations will stop when average gain in log-likelihood is below this threshold. Default is 1e-4.

verbose : bool, optional

Controls if debug information is printed to the console. Default is False.

Examples

```
>>> from mlpy.stats.models.mixture import StudentMM
```

```
>>> m = StudentMM()
```

Note: Adapted from Matlab:

Project: Probabilistic Modeling Toolkit for Matlab/Octave.
Copyright (2010) Kevin Murphy and Matt Dunham
License: MIT

Attributes

mean	Mean parameters of the emission.
cov	Covariance parameters of the emission.

mlpy.stats.models.mixture.StudentMM.mean

`StudentMM.`**`mean`**

Mean parameters of the emission.

Returns array, shape (*ncomponents*, *nfeatures*) :

The mean parameters.

mlpy.stats.models.mixture.StudentMM.cov

StudentMM.**cov**

Covariance parameters of the emission.

array, shape (*ncomponents*, *nfeatures*, *nfeatures*): Covariance parameters.

ncomponents	(int) Number of mixture components.
dim	(int) Dimensionality of the each component.
prior	(normal_invwishart) A `normal_invwishart` distribution.
mix_prior	(array_like, shape (*ncomponents*,)) Prior mixture probabilities.
mix_weight	(array_like, shape (*ncomponents*,)) Mixture weights.
df	(array, shape (*ncomponents*,)) Degrees of freedom.
cond_proba	(conditional_student) A `conditional_student` probability distribution.
n_iter	(int) Number of EM iterations to perform.
thresh	(float) Convergence threshold.
verbose	(bool) Controls if debug information is printed to the console.

Methods

`fit`(x[, n_init])	Fit the mixture model from the data *x*.
`predict`(x)	Predict label for data.
`predict_proba`(x)	Predict posterior probability of data under the model.
`sample`([size])	Generate random samples from the model.
`score`(x[, y])	Compute the log probability under the model.
`score_samples`(x)	Return the per-sample likelihood of the data under the model.

mlpy.stats.models.mixture.StudentMM.fit

StudentMM.**fit**(*x*, *n_init=1*)

Fit the mixture model from the data *x*.

Estimate model parameters with the expectation-maximization algorithm.

Parameters **x** : array_like, shape (*n*, *dim*)

List of dim-dimensional data points. Each row corresponds to a single data point.

n_init : int, optional

Number of random restarts to avoid a local minimum. Default is 1.

mlpy.stats.models.mixture.StudentMM.predict

StudentMM.**predict**(*x*)

Predict label for data.

Parameters **x** : array_like, shape (*size*, *dim*)

Returns **C** : array, shape = (*size*,)

mlpy.stats.models.mixture.StudentMM.predict_proba

`StudentMM.`**`predict_proba`**`(`*x*`)`

Predict posterior probability of data under the model.

Parameters **x** : array_like, shape (*size*, *dim*)

Returns **responsibilities** : array_like, shape = (*nsamples*, *ncomponents*)

Returns the probability of the sample for each Gaussian (state) in the model.

mlpy.stats.models.mixture.StudentMM.sample

`StudentMM.`**`sample`**`(`*size=1*`)`

Generate random samples from the model.

Parameters **size** : int, optional

Number of samples to generate. Default is 1.

Returns **x** : array_like, shape (*size*, *dim*)

List of samples

Raises **NotImplementedError**

If the child class does not implement this function.

mlpy.stats.models.mixture.StudentMM.score

`StudentMM.`**`score`**`(`*x*, *y=None*`)`

Compute the log probability under the model.

Parameters **x** : array_like, shape (size, dim)

List of dim-dimensional data points. Each row corresponds to a single data point.

y : Not used.

Returns **logp** : array_like, shape (*size*,)

Log probabilities of each data point in *x*.

mlpy.stats.models.mixture.StudentMM.score_samples

`StudentMM.`**`score_samples`**`(`*x*`)`

Return the per-sample likelihood of the data under the model.

Compute the log probability of x under the model and return the posterior distribution (responsibilities) of each mixture component for each element of x.

Parameters **x** : array_like, shape (*size*, *dim*)

List of *dim*-dimensional data points. Each row corresponds to a single data point.

Returns **responsibilities** : array_like, shape (*size*, *ncomponents*)

Posterior probabilities of each mixture component for each observation.

loglik : array_like, shape (size,)

Log probabilities of each data point in *x*.

Statistical functions

mlpy.stats.canonize_labels

`mlpy.stats.canonize_labels(labels, support=None)`

Transform labels to 1:k.

The size of canonized is the same as ladles but every label is transformed to its corresponding 1:k. If labels does not span the support, specify the support explicitly as the 2nd argument.

Parameters **labels** : array_like

support : optional

Returns Transformed labels.

Examples

```
>>> canonize_labels()
```

Note: Adapted from Matlab:

Project: Probabilistic Modeling Toolkit for Matlab/Octave.
Copyright (2010) Kevin Murphy and Matt Dunham
License: MIT

Warning: This is only a stub function. Implementation is still missing

mlpy.stats.is_posdef

`mlpy.stats.is_posdef(a)`

Test if matrix *a* is positive definite.

The method uses Cholesky decomposition to determine if the matrix is positive definite.

Parameters **a** : ndarray

A matrix.

Returns bool :

Whether the matrix is positive definite.

Examples

```
>>> is_posdef()
```

Note: Adapted from Matlab:

Project: Probabilistic Modeling Toolkit for Matlab/Octave.
Copyright (2010) Kevin Murphy and Matt Dunham
License: MIT

mlpy.stats.normalize_logspace

`mlpy.stats.`**`normalize_logspace`**`(`*a*`)`

Normalizes the array *a* in the log domain.

Each row of *a* is a log discrete distribution. Returns the array normalized in the log domain while minimizing the possibility of numerical underflow.

Parameters a : ndarray

The array to normalize in the log domain.

Returns a : ndarray

The array normalized in the log domain.

lnorm : float

log normalization constant.

Examples

```
>>> normalize_logspace()
```

Note: Adapted from Matlab:

Project: Probabilistic Modeling Toolkit for Matlab/Octave.
Copyright (2010) Kevin Murphy and Matt Dunham
License: MIT

mlpy.stats.partitioned_cov

`mlpy.stats.`**`partitioned_cov`**`(`*x*, *y*, *c=None*`)`

Covariance of groups.

Partition the rows of *x* according to class labels in *y* and take the covariance of each group.

Parameters **x** : array_like, shape (*n*, *dim*)

The data to group, where *n* is the number of data points and *dim* is the dimensionality of each data point.

y : array_like, shape (*n*,)

The class label for each data point.

c : int

The number of components in *y*.

Returns **cov** : array_like

The covariance of each group.

Examples

```
>>> partitioned_cov()
```

Note: Adapted from Matlab:

Project: Probabilistic Modeling Toolkit for Matlab/Octave.
Copyright (2010) Kevin Murphy and Matt Dunham
License: MIT

Warning: Implementation of this function is not finished yet.

mlpy.stats.partitioned_mean

`mlpy.stats.partitioned_mean`(*x*, *y*, *c=None*, *return_counts=False*)

Mean of groups.

Groups the rows of *x* according to the class labels in y and takes the mean of each group.

Parameters **x** : array_like, shape (*n*, *dim*)

The data to group, where *n* is the number of data points and *dim* is the dimensionality of each data point.

y : array_like, shape (*n*,)

The class label for each data point.

return_counts : bool

Whether to return the number of elements in each group or not.

Returns **mean** : array_like

The mean of each group.

counts : int

The number of elements in each group.

Examples

```
>>> partitioned_mean()
```

Note: Adapted from Matlab:

Project: Probabilistic Modeling Toolkit for Matlab/Octave.
Copyright (2010) Kevin Murphy and Matt Dunham
License: MIT

mlpy.stats.partitioned_sum

`mlpy.stats.`**`partitioned_sum`**(*x*, *y*, *c=None*)

Sums of groups.

Groups the rows of *x* according to the class labels in *y* and sums each group.

Parameters **x** : array_like, shape (*n*, *dim*)

The data to group, where *n* is the number of data points and *dim* is the dimensionality of each data point.

y : array_like, shape (*n*,)

The class label for each data point.

c : int

The number of components in *y*.

Returns **sums** : array_like

The sum of each group.

Examples

```
>>> partitioned_sum()
```

Note: Adapted from Matlab:

Project: Probabilistic Modeling Toolkit for Matlab/Octave.
Copyright (2010) Kevin Murphy and Matt Dunham
License: MIT

mlpy.stats.randpd

`mlpy.stats.randpd(dim)`

Create a random positive definite matrix of size *dim*-by-*dim*.

Parameters **dim** : int

The dimension of the matrix to create.

Returns ndarray :

A *dim*-by-*dim* positive definite matrix.

Examples

```
>>> randpd()
```

Note: Adapted from Matlab:

Project: Probabilistic Modeling Toolkit for Matlab/Octave.
Copyright (2010) Kevin Murphy and Matt Dunham
License: MIT

mlpy.stats.shrink_cov

`mlpy.stats.shrink_cov(x, return_lambda=False, return_estimate=False)`

Covariance shrinkage estimation.

Ledoit-Wolf optimal shrinkage estimator for cov(X) $C = \lambda * t + (1 - \lambda) * s$ using the diagonal variance 'target' t=np.diag(s) with the unbiased sample cov s as the unconstrained estimate.

Parameters **x** : array_like, shape (*n*, *dim*)

The data, where *n* is the number of data points and *dim* is the dimensionality of each data point.

return_lambda : bool

Whether to return lambda or not.

return_estimate : bool

Whether to return the unbiased estimate or not.

Returns **C** : array

The shrunk final estimate

lambda_ : float, optional

Lambda

estimate : array, optional

Unbiased estimate.

Examples

```
>>> shrink_cov()
```

Note: Adapted from Matlab:

Project: Probabilistic Modeling Toolkit for Matlab/Octave.
Copyright (2010) Kevin Murphy and Matt Dunham
License: MIT

mlpy.stats.sq_distance

`mlpy.stats.`**`sq_distance`**(*p*, *q*, *p_sos=None*, *q_sos=None*)

Efficiently compute squared Euclidean distances between stats of vectors.

Compute the squared Euclidean distances between every d-dimensional point in *p* to every *d*-dimensional point in q. Both *p* and *q* are n-point-by-n-dimensions.

Parameters **p** : array_like, shape (*n*, *dim*)

Array where *n* is the number of points and *dim* is the number of dimensions.

q : array_like, shape (*n*, *dim*)

Array where *n* is the number of points and *dim* is the number of dimensions.

p_sos : array_like, shape (*dim*,)

q_sos : array_like, shape (*dim*,)

Returns ndarray :

The squared Euclidean distance.

Examples

```
>>> sq_distance()
```

Note: Adapted from Matlab:

Project: Probabilistic Modeling Toolkit for Matlab/Octave.
Copyright (2010) Kevin Murphy and Matt Dunham
License: MIT

mlpy.stats.stacked_randpd

`mlpy.stats.stacked_randpd(dim, k, p=0)`

Create stacked positive definite matrices.

Create multiple random positive definite matrices of size dim-by-dim and stack them.

Parameters **dim** : int

The dimension of each matrix.

k : int

The number of matrices.

p : int

The diagonal value of each matrix.

Returns ndarray :

Multiple stacked random positive definite matrices.

Examples

```
>>> stacked_randpd()
```

Note: Adapted from Matlab:

Project: Probabilistic Modeling Toolkit for Matlab/Octave.
Copyright (2010) Kevin Murphy and Matt Dunham
License: MIT

is_posdef	Test if matrix *a* is positive definite.
randpd	Create a random positive definite matrix.
stacked_randpd	Create multiple random positive definite matrices.
normalize_logspace	Normalize in log space while avoiding numerical underflow.
sq_distance	Efficiently compute squared Euclidean distances between stats of vectors.
partitioned_cov	Partition the rows of *x* according to *y* and take the covariance of each group.
partitioned_mean	Groups the rows of x according to the class labels in y and takes the mean of each group.
partitioned_sum	Groups the rows of x according to the class labels in y and sums each group.
shrink_cov	Ledoit-Wolf optimal shrinkage estimator.
canonize_labels	Transform labels to 1:k.

CHAPTER 22

Dynamic Bayesian networks (`mlpy.stats.dbn`)

hmm	Hidden Markov Models

mlpy.stats.dbn.hmm

Hidden Markov Models

HMM	Hidden Markov Model base class.
DiscreteHMM	Hidden Markov Model with discrete(multinomial) emissions.
GaussianHMM	Hidden Markov Model with Gaussian emissions.
StudentHMM	Hidden Markov Model with Student emissions
GMMHMM	Hidden Markov Model with Gaussian mixture emissions.

mlpy.stats.dbn.hmm.HMM

class `mlpy.stats.dbn.hmm.`**`HMM`**(*ncomponents=1*, *startprob_prior=None*, *startprob=None*, *transmat_prior=None*, *transmat=None*, *emission_prior=None*, *emission=None*, *n_iter=None*, *thresh=None*, *verbose=None*)

Bases: *`mlpy.optimize.algorithms.EM`*

Hidden Markov Model base class.

Representation of a hidden Markov model probability distribution. This class allows for easy evaluation of, sampling from, and maximum-likelihood estimation of the parameters of a HMM.

See the instance documentation for details specific to a particular object.

Parameters **ncomponents** : int

Number of states in the model.

startprob_prior : array, shape (*ncomponents*,) Initial state occupation prior distribution.

startprob : array, shape (*ncomponents*,)

Initial state occupation distribution.

transmat_prior : array, shape (*ncomponents*, *ncomponents*)

Matrix of prior transition probabilities between states.

transmat : array, shape (*ncomponents*, *ncomponents*)

Matrix of transition probabilities between states.

emission : cond_rv_frozen

The conditional probability distribution used for the emission.

emission_prior : normal_invwishart

Initial emission parameters, a normal-inverse Wishart distribution.

n_iter : int

Number of iterations to perform during training, optional.

thresh : float

Convergence threshold, optional.

verbose : bool

Controls if debug information is printed to the console, optional.

Examples

```
>>> from mlpy.stats.dbn.hmm import GaussianHMM
```

```
>>> model = GaussianHMM(ncomponents=2, startprob_prior=[3, 2])
```

Create a gaussian hidden Markov model

```
>>> import scipy.io
>>> mat = scipy.io.loadmat('data/speechDataDigits4And5.mat'))
>>> x = np.hstack([mat['train4'][0], mat['train5'][0]])
```

Load data used for fitting the HMM and fit the HMM:

```
>>> model.fit(x, n_init=3)
```

Note: Adapted from Matlab:

Project: Probabilistic Modeling Toolkit for Matlab/Octave.
Copyright (2010) Kevin Murphy and Matt Dunham
License: MIT

Attributes

`startprob_prior`	Vector of initial probabilities for each state.
`transmat_prior`	Transition probability matrix.

mlpy.stats.dbn.hmm.HMM.startprob_prior

HMM.**startprob_prior**

Vector of initial probabilities for each state.

Returns startprob_prior : array, shape (*ncomponents*,)

The initial probabilities.

mlpy.stats.dbn.hmm.HMM.transmat_prior

HMM.**transmat_prior**

Transition probability matrix.

Returns transmat_prior : array, shape (*ncomponents*, *ncomponents*)

Matrix of transition probabilities from each state to every other state.

ncomponents	(int) The number of hidden states.
nfeatures	(int) Dimensionality of the Gaussian emission.
startprob	(array, shape (*ncomponents*,)) Initial state occupation distribution.
transmat	(array, shape (*ncomponents*, *ncomponents*)) Matrix of transition probabilities between states.
emission_prior	(normal_invwishart) Initial emission parameters, a normal-inverse Wishart distribution.
emission	(cond_rv_frozen) The conditional probability distribution used for the emission.

Methods

`decode`(obs[, algorithm])	Find the most likely state sequence.
`fit`(obs[, n_init])	Estimate model parameters.
`predict_proba`(obs)	Compute the posterior probability for each state in the model.
`sample`(length[, size])	Generates random samples from the model.
`score`(obs)	Compute log probability of the evidence (likelihood) under the model.
`score_samples`(obs)	Compute the log probability of the evidence.

mlpy.stats.dbn.hmm.HMM.decode

HMM.**decode**(*obs*, *algorithm='viterbi'*)

Find the most likely state sequence.

Find the most likely state sequence corresponding to the observation *obs*. Uses the given algorithm for decoding.

Parameters **obs** : array_like, shape (*nfeatures*, *T*)

The local evidence vector.

algorithm : {'viterbi', 'map'}

Decoder algorithm to be used.

Returns **best_path** : array_like, shape (*n*,)

The most likely states for each observation

loglik : float

Log probability of the maximum likelihood path through the HMM

mlpy.stats.dbn.hmm.HMM.fit

`HMM.`**`fit`**(*obs*, *n_init=1*)

Estimate model parameters.

Parameters **obs** : array_like, shape (*n*, *ni*, *nfeatures*)

List of observation sequences, where *n* is the number of sequences, *ni* is the length of the i_th observation, and each observation has *nfeatures* features.

Returns float :

log likelihood of the sequence *obs*

mlpy.stats.dbn.hmm.HMM.predict_proba

`HMM.`**`predict_proba`**(*obs*)

Compute the posterior probability for each state in the model.

Parameters **obs** : array_like, shape (*n*, *len*, *nfeatures*)

Sequence of *nfeatures*-dimensional data points. Each row corresponds to a single point in the sequence.

Returns **posteriors** : array_like, shape (*n*, *ncomponents*)

Posterior probabilities of each state for each observation

mlpy.stats.dbn.hmm.HMM.sample

`HMM.`**`sample`**(*length*, *size=1*)

Generates random samples from the model.

Parameters **length** : int or ndarray[int]

Length of a sample

size : int, optional

Number of samples to generate. Default is 1.

Returns **obs** : array_like, shape (*n*, *ni*, *nfeatures*)

List of samples, where *n* is the number of samples, *ni* is the length of the i-th sample, and each observation has *nfeatures*.

hidden_states : array_like, shape (*n*, *ni*)

List of hidden states, where *n* is the number of samples, *ni* is the i-th hidden state.

mlpy.stats.dbn.hmm.HMM.score

`HMM.`**`score`**(*obs*)

Compute log probability of the evidence (likelihood) under the model.

Parameters obs : array_like, shape (*n*, *len*, *nfeatures*)

Sequence of *nfeatures*-dimensional data points. Each row corresponds to a single point in the sequence.

Returns logp : float

Log likelihood of the sequence *obs*.

mlpy.stats.dbn.hmm.HMM.score_samples

`HMM.`**`score_samples`**(*obs*)

Compute the log probability of the evidence.

Compute the log probability of the evidence (likelihood) under the model and the posteriors.

Parameters obs : array_like, shape (*n*, *len*, *nfeatures*)

Sequence of *nfeatures*-dimensional data points. Each row corresponds to a single point in the sequence.

Returns logp : float

Log likelihood of the sequence *obs*.

posteriors : array_like, shape (*n*, *ncomponents*)

Posterior probabilities of each state for each observation

mlpy.stats.dbn.hmm.DiscreteHMM

class `mlpy.stats.dbn.hmm.`**`DiscreteHMM`**(*ncomponents=1*, *startprob_prior=None*, *startprob=None*, *transmat_prior=None*, *transmat=None*, *emission_prior=None*, *emission=None*, *n_iter=None*, *thresh=None*, *verbose=None*)

Bases: *`mlpy.stats.dbn.hmm.HMM`*

Hidden Markov Model with discrete(multinomial) emissions.

Representation of a hidden Markov model probability distribution. This class allows for easy evaluation of, sampling from, and maximum-likelihood estimation of the parameters of a HMM.

Parameters ncomponents : int

Number of states in the model.

startprob_prior : array, shape (*ncomponents*,)

Initial state occupation prior distribution.

startprob : array, shape (*ncomponents*,) Initial state occupation distribution.

transmat_prior : array, shape (*ncomponents*, *ncomponents*)

Matrix of prior transition probabilities between states.

transmat : array, shape (*ncomponents*, *ncomponents*)

Matrix of transition probabilities between states.

emission : cond_rv_frozen

The conditional probability distribution used for the emission.

emission_prior : normal_invwishart

Initial emission parameters, a normal-inverse Wishart distribution.

n_iter : int

Number of iterations to perform during training, optional.

thresh : float

Convergence threshold, optional.

verbose : bool

Controls if debug information is printed to the console, optional.

Examples

```
>>> from mlpy.stats.dbn.hmm import DiscreteHMM
>>> DiscreteHMM(ncomponents=2)
...
```

Note: Adapted from Matlab:

Project: Probabilistic Modeling Toolkit for Matlab/Octave.
Copyright (2010) Kevin Murphy and Matt Dunham
License: MIT

Attributes

startprob_prior	Vector of initial probabilities for each state.
transmat_prior	Transition probability matrix.

mlpy.stats.dbn.hmm.DiscreteHMM.startprob_prior

DiscreteHMM.**startprob_prior**
Vector of initial probabilities for each state.

Returns startprob_prior : array, shape (*ncomponents*,) The initial probabilities.

mlpy.stats.dbn.hmm.DiscreteHMM.transmat_prior

`DiscreteHMM.`**`transmat_prior`**

Transition probability matrix.

Returns **transmat_prior** : array, shape (*ncomponents*, *ncomponents*)

Matrix of transition probabilities from each state to every other state.

ncomponents	(int) The number of hidden states.
nfeatures	(int) Dimensionality of the Gaussian emission.
startprob	(array, shape (*ncomponents*,)) Initial state occupation distribution.
transmat	(array, shape (*ncomponents*, *ncomponents*)) Matrix of transition probabilities between states.
emis-sion_prior	(normal_invwishart) Initial emission parameters, a normal-inverse Wishart distribution.
emission	(cond_rv_frozen) The conditional probability distribution used for the emission.

Methods

`decode`(obs[, algorithm])	Find the most likely state sequence.
`fit`(obs[, n_init])	Estimate model parameters.
`predict_proba`(obs)	Compute the posterior probability for each state in the model.
`sample`(length[, size])	Generates random samples from the model.
`score`(obs)	Compute log probability of the evidence (likelihood) under the model.
`score_samples`(obs)	Compute the log probability of the evidence.

mlpy.stats.dbn.hmm.DiscreteHMM.decode

`DiscreteHMM.`**`decode`**(*obs*, *algorithm='viterbi'*)

Find the most likely state sequence.

Find the most likely state sequence corresponding to the observation *obs*. Uses the given algorithm for decoding.

Parameters **obs** : array_like, shape (*nfeatures*, *T*)

The local evidence vector.

algorithm : {'viterbi', 'map'}

Decoder algorithm to be used.

Returns **best_path** : array_like, shape (*n*,)

The most likely states for each observation

loglik : float

Log probability of the maximum likelihood path through the HMM

mlpy.stats.dbn.hmm.DiscreteHMM.fit

`DiscreteHMM.`**`fit`**(*obs*, *n_init=1*)

Estimate model parameters.

Parameters obs : array_like, shape (*n*, *ni*, *nfeatures*)

List of observation sequences, where *n* is the number of sequences, *ni* is the length of the i_th observation, and each observation has *nfeatures* features.

Returns float :

log likelihood of the sequence *obs*

mlpy.stats.dbn.hmm.DiscreteHMM.predict_proba

`DiscreteHMM.`**`predict_proba`**(*obs*)

Compute the posterior probability for each state in the model.

Parameters obs : array_like, shape (*n*, *len*, *nfeatures*)

Sequence of *nfeatures*-dimensional data points. Each row corresponds to a single point in the sequence.

Returns posteriors : array_like, shape (*n*, *ncomponents*)

Posterior probabilities of each state for each observation

mlpy.stats.dbn.hmm.DiscreteHMM.sample

`DiscreteHMM.`**`sample`**(*length*, *size=1*)

Generates random samples from the model.

Parameters length : int or ndarray[int]

Length of a sample

size : int, optional

Number of samples to generate. Default is 1.

Returns obs : array_like, shape (*n*, *ni*, *nfeatures*)

List of samples, where *n* is the number of samples, *ni* is the length of the i-th sample, and each observation has *nfeatures*.

hidden_states : array_like, shape (*n*, *ni*)

List of hidden states, where *n* is the number of samples, *ni* is the i-th hidden state.

mlpy.stats.dbn.hmm.DiscreteHMM.score

`DiscreteHMM.`**`score`**(*obs*)

Compute log probability of the evidence (likelihood) under the model.

Parameters obs : array_like, shape (*n*, *len*, *nfeatures*)

Sequence of *nfeatures*-dimensional data points. Each row corresponds to a single point in the sequence.

Returns **logp** : float

Log likelihood of the sequence *obs*.

mlpy.stats.dbn.hmm.DiscreteHMM.score_samples

`DiscreteHMM.`**`score_samples`**(*obs*)

Compute the log probability of the evidence.

Compute the log probability of the evidence (likelihood) under the model and the posteriors.

Parameters **obs** : array_like, shape (*n*, *len*, *nfeatures*)

Sequence of *nfeatures*-dimensional data points. Each row corresponds to a single point in the sequence.

Returns **logp** : float

Log likelihood of the sequence *obs*.

posteriors : array_like, shape (*n*, *ncomponents*)

Posterior probabilities of each state for each observation

mlpy.stats.dbn.hmm.GaussianHMM

class `mlpy.stats.dbn.hmm.`**`GaussianHMM`**(*ncomponents=1*, *startprob_prior=None*, *startprob=None*, *transmat_prior=None*, *transmat=None*, *emission_prior=None*, *emission=None*, *n_iter=None*, *thresh=None*, *verbose=None*)

Bases: *`mlpy.stats.dbn.hmm.HMM`*

Hidden Markov Model with Gaussian emissions.

Representation of a hidden Markov model probability distribution. This class allows for easy evaluation of, sampling from, and maximum-likelihood estimation of the parameters of a HMM.

Parameters **ncomponents** : int

Number of states in the model.

startprob_prior : array, shape (*ncomponents*,)

Initial state occupation prior distribution.

startprob : array, shape (*ncomponents*,)

Initial state occupation distribution.

transmat_prior : array, shape (*ncomponents*, *ncomponents*)

Matrix of prior transition probabilities between states.

transmat : array, shape (*ncomponents*, *ncomponents*)

Matrix of transition probabilities between states.

emission : conditional_normal_frozen

The conditional probability distribution used for the emission.

emission_prior : normal_invwishart

Initial emission parameters, a normal-inverse Wishart distribution.

n_iter : int

Number of iterations to perform during training, optional.

thresh : float

Convergence threshold, optional.

verbose : bool

Controls if debug information is printed to the console, optional.

Examples

```
>>> from mlpy.stats.dbn.hmm import GaussianHMM
>>> GaussianHMM(ncomponents=2)
...
```

Note: Adapted from Matlab:

Project: Probabilistic Modeling Toolkit for Matlab/Octave.
Copyright (2010) Kevin Murphy and Matt Dunham
License: MIT

Attributes

startprob_prior	Vector of initial probabilities for each state.
transmat_prior	Transition probability matrix.
mean	The mean parameters for each state
cov	Covariance parameters for each state.

mlpy.stats.dbn.hmm.GaussianHMM.startprob_prior

`GaussianHMM.`**`startprob_prior`**

Vector of initial probabilities for each state.

Returns **startprob_prior** : array, shape (*ncomponents*,)

The initial probabilities.

mlpy.stats.dbn.hmm.GaussianHMM.transmat_prior

`GaussianHMM.`**`transmat_prior`**

Transition probability matrix.

Returns **transmat_prior** : array, shape (*ncomponents*, *ncomponents*)

Matrix of transition probabilities from each state to every other state.

mlpy.stats.dbn.hmm.GaussianHMM.mean

`GaussianHMM.`**`mean`**

The mean parameters for each state

Returns array, shape (*ncomponents*, *nfeatures*) :

Mean parameters for each state.

mlpy.stats.dbn.hmm.GaussianHMM.cov

`GaussianHMM.`**`cov`**

Covariance parameters for each state.

Returns array, shape (*ncomponents*, *nfeatures*, *nfeatures*) :

Covariance parameters for each state as a full matrix

ncomponents	(int) The number of hidden states.
nfeatures	(int) Dimensionality of the Gaussian emission.
startprob	(array, shape (*ncomponents*,)) Initial state occupation distribution.
transmat	(array, shape (*ncomponents*, *ncomponents*)) Matrix of transition probabilities between states.
emission_prior	(normal_invwishart) Initial emission parameters, a normal-inverse Wishart distribution.
emission	(cond_rv_frozen) The conditional probability distribution used for the emission.

Methods

`decode`(obs[, algorithm])	Find the most likely state sequence.
`fit`(obs[, n_init])	Estimate model parameters.
`predict_proba`(obs)	Compute the posterior probability for each state in the model.
`sample`(length[, size])	Generates random samples from the model.
`score`(obs)	Compute log probability of the evidence (likelihood) under the model.
`score_samples`(obs)	Compute the log probability of the evidence.

mlpy.stats.dbn.hmm.GaussianHMM.decode

`GaussianHMM.`**`decode`**(*obs*, *algorithm='viterbi'*)

Find the most likely state sequence.

Find the most likely state sequence corresponding to the observation *obs*. Uses the given algorithm for decoding.

Parameters **obs** : array_like, shape (*nfeatures*, *T*)

The local evidence vector.

algorithm : {'viterbi', 'map'}

Decoder algorithm to be used.

Returns **best_path** : array_like, shape (*n*,) The most likely states for each observation

loglik : float

Log probability of the maximum likelihood path through the HMM

mlpy.stats.dbn.hmm.GaussianHMM.fit

`GaussianHMM.`**`fit`**`(`*obs*, *n_init=1*`)`

Estimate model parameters.

Parameters **obs** : array_like, shape (*n*, *ni*, *nfeatures*)

List of observation sequences, where *n* is the number of sequences, *ni* is the length of the i_th observation, and each observation has *nfeatures* features.

Returns float :

log likelihood of the sequence *obs*

mlpy.stats.dbn.hmm.GaussianHMM.predict_proba

`GaussianHMM.`**`predict_proba`**`(`*obs*`)`

Compute the posterior probability for each state in the model.

Parameters **obs** : array_like, shape (*n*, *len*, *nfeatures*)

Sequence of *nfeatures*-dimensional data points. Each row corresponds to a single point in the sequence.

Returns **posteriors** : array_like, shape (*n*, *ncomponents*)

Posterior probabilities of each state for each observation

mlpy.stats.dbn.hmm.GaussianHMM.sample

`GaussianHMM.`**`sample`**`(`*length*, *size=1*`)`

Generates random samples from the model.

Parameters **length** : int or ndarray[int]

Length of a sample

size : int, optional

Number of samples to generate. Default is 1.

Returns **obs** : array_like, shape (*n*, *ni*, *nfeatures*)

List of samples, where *n* is the number of samples, *ni* is the length of the i-th sample, and each observation has *nfeatures*.

hidden_states : array_like, shape (*n*, *ni*)

List of hidden states, where *n* is the number of samples, *ni* is the i-th hidden state.

mlpy.stats.dbn.hmm.GaussianHMM.score

`GaussianHMM.`**`score`**(*obs*)

Compute log probability of the evidence (likelihood) under the model.

Parameters **obs** : array_like, shape (*n*, *len*, *nfeatures*)

Sequence of *nfeatures*-dimensional data points. Each row corresponds to a single point in the sequence.

Returns **logp** : float

Log likelihood of the sequence *obs*.

mlpy.stats.dbn.hmm.GaussianHMM.score_samples

`GaussianHMM.`**`score_samples`**(*obs*)

Compute the log probability of the evidence.

Compute the log probability of the evidence (likelihood) under the model and the posteriors.

Parameters **obs** : array_like, shape (*n*, *len*, *nfeatures*)

Sequence of *nfeatures*-dimensional data points. Each row corresponds to a single point in the sequence.

Returns **logp** : float

Log likelihood of the sequence *obs*.

posteriors : array_like, shape (*n*, *ncomponents*)

Posterior probabilities of each state for each observation

mlpy.stats.dbn.hmm.StudentHMM

class `mlpy.stats.dbn.hmm.`**`StudentHMM`**(*ncomponents=1*, *startprob_prior=None*, *startprob=None*, *transmat_prior=None*, *transmat=None*, *emission_prior=None*, *emission=None*, *n_iter=None*, *thresh=None*, *verbose=None*)

Bases: *`mlpy.stats.dbn.hmm.HMM`*

Hidden Markov Model with Student emissions

Representation of a hidden Markov model probability distribution. This class allows for easy evaluation of, sampling from, and maximum-likelihood estimation of the parameters of a HMM.

Parameters **ncomponents** : int

Number of states in the model.

startprob_prior : array, shape (*ncomponents*,)

Initial state occupation prior distribution.

startprob : array, shape (*ncomponents*,)

Initial state occupation distribution.

transmat_prior : array, shape (*ncomponents*, *ncomponents*)

Matrix of prior transition probabilities between states.

transmat : array, shape (*ncomponents*, *ncomponents*)

Matrix of transition probabilities between states.

emission : conditional_student_frozen

The conditional probability distribution used for the emission.

emission_prior : normal_invwishart

Initial emission parameters, a normal-inverse Wishart distribution.

n_iter : int

Number of iterations to perform during training, optional.

thresh : float

Convergence threshold, optional.

verbose : bool

Controls if debug information is printed to the console, optional.

Examples

```
>>> from mlpy.stats.dbn.hmm import StudentHMM
>>> StudentHMM(ncomponents=2)
...
```

Note: Adapted from Matlab:

Project: Probabilistic Modeling Toolkit for Matlab/Octave.
Copyright (2010) Kevin Murphy and Matt Dunham
License: MIT

Attributes

startprob_prior	Vector of initial probabilities for each state.
transmat_prior	Transition probability matrix.

mlpy.stats.dbn.hmm.StudentHMM.startprob_prior

`StudentHMM.`**`startprob_prior`**

Vector of initial probabilities for each state.

Returns **startprob_prior** : array, shape (*ncomponents*,)

The initial probabilities.

mlpy.stats.dbn.hmm.StudentHMM.transmat_prior

`StudentHMM.`**`transmat_prior`**

Transition probability matrix.

Returns transmat_prior : array, shape (*ncomponents*, *ncomponents*)

Matrix of transition probabilities from each state to every other state.

ncomponents	(int) The number of hidden states.
nfeatures	(int) Dimensionality of the Gaussian emission.
startprob	(array, shape (*ncomponents*,)) Initial state occupation distribution.
transmat	(array, shape (*ncomponents*, *ncomponents*)) Matrix of transition probabilities between states.
emission_prior	(normal_invwishart) Initial emission parameters, a normal-inverse Wishart distribution.
emission	(cond_rv_frozen) The conditional probability distribution used for the emission.

Methods

decode(obs[, algorithm])	Find the most likely state sequence.
fit(obs[, n_init])	Estimate model parameters.
predict_proba(obs)	Compute the posterior probability for each state in the model.
sample(length[, size])	Generates random samples from the model.
score(obs)	Compute log probability of the evidence (likelihood) under the model.
score_samples(obs)	Compute the log probability of the evidence.

mlpy.stats.dbn.hmm.StudentHMM.decode

`StudentHMM.`**`decode`**(*obs*, *algorithm='viterbi'*)

Find the most likely state sequence.

Find the most likely state sequence corresponding to the observation *obs*. Uses the given algorithm for decoding.

Parameters obs : array_like, shape (*nfeatures*, *T*)

The local evidence vector.

algorithm : {'viterbi', 'map'}

Decoder algorithm to be used.

Returns best_path : array_like, shape (*n*,)

The most likely states for each observation

loglik : float

Log probability of the maximum likelihood path through the HMM

mlpy.stats.dbn.hmm.StudentHMM.fit

`StudentHMM.fit(obs, n_init=1)`

Estimate model parameters.

Parameters **obs** : array_like, shape (*n*, *ni*, *nfeatures*)

List of observation sequences, where *n* is the number of sequences, *ni* is the length of the i_th observation, and each observation has *nfeatures* features.

Returns float :

log likelihood of the sequence *obs*

mlpy.stats.dbn.hmm.StudentHMM.predict_proba

`StudentHMM.predict_proba(obs)`

Compute the posterior probability for each state in the model.

Parameters **obs** : array_like, shape (*n*, *len*, *nfeatures*)

Sequence of *nfeatures*-dimensional data points. Each row corresponds to a single point in the sequence.

Returns **posteriors** : array_like, shape (*n*, *ncomponents*)

Posterior probabilities of each state for each observation

mlpy.stats.dbn.hmm.StudentHMM.sample

`StudentHMM.sample(length, size=1)`

Generates random samples from the model.

Parameters **length** : int or ndarray[int]

Length of a sample

size : int, optional

Number of samples to generate. Default is 1.

Returns **obs** : array_like, shape (*n*, *ni*, *nfeatures*)

List of samples, where *n* is the number of samples, *ni* is the length of the i-th sample, and each observation has *nfeatures*.

hidden_states : array_like, shape (*n*, *ni*)

List of hidden states, where *n* is the number of samples, *ni* is the i-th hidden state.

mlpy.stats.dbn.hmm.StudentHMM.score

`StudentHMM.score(obs)`

Compute log probability of the evidence (likelihood) under the model.

Parameters **obs** : array_like, shape (*n*, *len*, *nfeatures*)

Sequence of *nfeatures*-dimensional data points. Each row corresponds to a single point in the sequence.

Returns **logp** : float

Log likelihood of the sequence *obs*.

mlpy.stats.dbn.hmm.StudentHMM.score_samples

`StudentHMM.`**`score_samples`**(*obs*)

Compute the log probability of the evidence.

Compute the log probability of the evidence (likelihood) under the model and the posteriors.

Parameters **obs** : array_like, shape (*n*, *len*, *nfeatures*)

Sequence of *nfeatures*-dimensional data points. Each row corresponds to a single point in the sequence.

Returns **logp** : float

Log likelihood of the sequence *obs*.

posteriors : array_like, shape (*n*, *ncomponents*)

Posterior probabilities of each state for each observation

mlpy.stats.dbn.hmm.GMMHMM

class `mlpy.stats.dbn.hmm.`**`GMMHMM`**(*ncomponents=1*, *nmix=1*, *startprob_prior=None*, *startprob=None*, *transmat_prior=None*, *transmat=None*, *emission_prior=None*, *emission=None*, *n_iter=None*, *thresh=None*, *verbose=None*)

Bases: *`mlpy.stats.dbn.hmm.HMM`*

Hidden Markov Model with Gaussian mixture emissions.

Representation of a hidden Markov model probability distribution. This class allows for easy evaluation of, sampling from, and maximum-likelihood estimation of the parameters of a HMM.

Parameters **ncomponents** : int

Number of states in the model.

nmix : int

Number of mixtures.

startprob_prior : array, shape (*ncomponents*,)

Initial state occupation prior distribution.

startprob : array, shape (*ncomponents*,)

Initial state occupation distribution.

transmat_prior : array, shape (*ncomponents*, *ncomponents*)

Matrix of prior transition probabilities between states.

transmat : array, shape (*ncomponents*, *ncomponents*)

Matrix of transition probabilities between states.

emission : conditional_mix_normal_frozen

The conditional probability distribution used for the emission.

emission_prior : normal_invwishart

Initial emission parameters, a normal-inverse Wishart distribution.

n_iter : int

Number of iterations to perform during training, optional.

thresh : float

Convergence threshold, optional.

verbose : bool

Controls if debug information is printed to the console, optional.

Examples

```
>>> from mlpy.stats.dbn.hmm import GMMHMM
>>> GMMHMM(ncomponents=2)
...
```

Note: Adapted from Matlab:

Project: Probabilistic Modeling Toolkit for Matlab/Octave.
Copyright (2010) Kevin Murphy and Matt Dunham
License: MIT

Attributes

startprob_prior	Vector of initial probabilities for each state.
transmat_prior	Transition probability matrix.

mlpy.stats.dbn.hmm.GMMHMM.startprob_prior

GMMHMM.**startprob_prior**

Vector of initial probabilities for each state.

Returns **startprob_prior** : array, shape (*ncomponents*,)

The initial probabilities.

mlpy.stats.dbn.hmm.GMMHMM.transmat_prior

GMMHMM.**transmat_prior**

Transition probability matrix.

Returns **transmat_prior** : array, shape (*ncomponents*, *ncomponents*)

Matrix of transition probabilities from each state to every other state.

ncomponents	(int) The number of hidden states.
nmix	(int) Number of mixtures.
nfeatures	(int) Dimensionality of the Gaussian emission.
startprob	(array, shape (*ncomponents*,)) Initial state occupation distribution.
transmat	(array, shape (*ncomponents*, *ncomponents*)) Matrix of transition probabilities between states.
emis-sion_prior	(normal_invwishart) Initial emission parameters, a normal-inverse Wishart distribution.
emission	(cond_rv_frozen) The conditional probability distribution used for the emission.

Methods

`decode`(obs[, algorithm])	Find the most likely state sequence.
`fit`(obs[, n_init])	Estimate model parameters.
`predict_proba`(obs)	Compute the posterior probability for each state in the model.
`sample`(length[, size])	Generates random samples from the model.
`score`(obs)	Compute log probability of the evidence (likelihood) under the model.
`score_samples`(obs)	Compute the log probability of the evidence.

mlpy.stats.dbn.hmm.GMMHMM.decode

GMMHMM.**decode**(*obs*, *algorithm='viterbi'*)

Find the most likely state sequence.

Find the most likely state sequence corresponding to the observation *obs*. Uses the given algorithm for decoding.

Parameters **obs** : array_like, shape (*nfeatures*, *T*)

The local evidence vector.

algorithm : {'viterbi', 'map'}

Decoder algorithm to be used.

Returns **best_path** : array_like, shape (*n*,)

The most likely states for each observation

loglik : float

Log probability of the maximum likelihood path through the HMM

mlpy.stats.dbn.hmm.GMMHMM.fit

GMMHMM.**fit**(*obs*, *n_init=1*)

Estimate model parameters.

Parameters **obs** : array_like, shape (*n*, *ni*, *nfeatures*)

List of observation sequences, where *n* is the number of sequences, *ni* is the length of the i_th observation, and each observation has *nfeatures* features.

Returns float :

log likelihood of the sequence *obs*

mlpy.stats.dbn.hmm.GMMHMM.predict_proba

GMMHMM.**predict_proba**(*obs*)

Compute the posterior probability for each state in the model.

Parameters **obs** : array_like, shape (*n*, *len*, *nfeatures*)

Sequence of *nfeatures*-dimensional data points. Each row corresponds to a single point in the sequence.

Returns **posteriors** : array_like, shape (*n*, *ncomponents*)

Posterior probabilities of each state for each observation

mlpy.stats.dbn.hmm.GMMHMM.sample

GMMHMM.**sample**(*length*, *size=1*)

Generates random samples from the model.

Parameters **length** : int or ndarray[int]

Length of a sample

size : int, optional

Number of samples to generate. Default is 1.

Returns **obs** : array_like, shape (*n*, *ni*, *nfeatures*)

List of samples, where *n* is the number of samples, *ni* is the length of the i-th sample, and each observation has *nfeatures*.

hidden_states : array_like, shape (*n*, *ni*)

List of hidden states, where *n* is the number of samples, *ni* is the i-th hidden state.

mlpy.stats.dbn.hmm.GMMHMM.score

GMMHMM.**score**(*obs*)

Compute log probability of the evidence (likelihood) under the model.

Parameters **obs** : array_like, shape (*n*, *len*, *nfeatures*)

Sequence of *nfeatures*-dimensional data points. Each row corresponds to a single point in the sequence.

Returns **logp** : float

Log likelihood of the sequence *obs*.

mlpy.stats.dbn.hmm.GMMHMM.score_samples

GMMHMM.**score_samples**(*obs*)

Compute the log probability of the evidence.

Compute the log probability of the evidence (likelihood) under the model and the posteriors.

Parameters **obs** : array_like, shape (*n*, *len*, *nfeatures*)

Sequence of *nfeatures*-dimensional data points. Each row corresponds to a single point in the sequence.

Returns **logp** : float

Log likelihood of the sequence *obs*.

posteriors : array_like, shape (*n*, *ncomponents*)

Posterior probabilities of each state for each observation

`is_posdef`	Test if matrix *a* is positive definite.
`randpd`	Create a random positive definite matrix of size *dim*-by-*dim*.
`stacked_randpd`	Create stacked positive definite matrices.
`normalize_logspace`	Normalizes the array *a* in the log domain.
`sq_distance`	Efficiently compute squared Euclidean distances between stats of vectors.
`partitioned_cov`	Covariance of groups.
`partitioned_mean`	Mean of groups.
`partitioned_sum`	Sums of groups.
`shrink_cov`	Covariance shrinkage estimation.
`canonize_labels`	Transform labels to 1:k.
`nonuniform`	Create a new *Mock* object.
`gibbs`	Create a new *Mock* object.
`conditional_normal`	
`conditional_student`	
`conditional_mix_normal`	
`multivariate_normal`	
`multivariate_student`	
`invwishart`	
`normal_invwishart`	
`models.markov`	

CHAPTER 23

Tools (mlpy.tools)

ConfigMgr	The configuration manager.
LoggingMgr	The logging manager *Singleton* class.

mlpy.tools.configuration.ConfigMgr

class mlpy.tools.configuration.**ConfigMgr**(*filename*, *import_modules=None*, *eval_key=None*)

Bases: object

The configuration manager.

The configuration manager provides access to configuration files (usually in JSON format) for client applications.

Parameters **filename** : str

The name of the configuration file.

import_modules : str or list[str]

Modules required by the configuration file that must be imported first.

eval_key : bool

Whether to evaluate the key. If this is *True*, the key will be evaluated as a statement by a call to eval.

Raises **TypeError**

If the configuration file is not read in a dictionary

Examples

Assuming there exists a file events_map.json containing the following configuration:

```
{
    "keyboard": {
        "down": {
            "pygame.K_ESCAPE": "QUIT",
            "pygame.K_SPACE": [-1.0],
            "pygame.K_LEFT" : [-0.004],
            "pygame.K_RIGHT":  [0.004]
        }
    }
}
```

The keys can be mapped to the PyGame keyboard constants, when the file is loaded by the configuration manager as follows:

```
>>> cfg = ConfigMgr("events_map.json", "pygame", eval_key=True)
```

This allows to retrieve the values for the keys in the configuration file by using the PyGame keyboard constants, which are returned in the *key* attribute of the PyGame event:

```
>>> import pygame
>>> for event in pygame.event.get():
>>>     print cfg.get("keyboard.down." + str(event.key))
```

Methods

get(key)	Return the value for the given key.
has_config(key)	Checks if the given key exists in the configuration.

mlpy.tools.configuration.ConfigMgr.get

ConfigMgr.**get**(*key*)

Return the value for the given key.

Parameters key : str

The key for the configuration. Concatenate keys by dots (.) to access keys at deeper levels in the configuration.

Raises KeyError

If the key does not exist in the configuration

mlpy.tools.configuration.ConfigMgr.has_config

ConfigMgr.**has_config**(*key*)

Checks if the given key exists in the configuration.

Parameters key : str

The key for the configuration. Concatenate keys by dots (`.`) to access keys at deeper levels in the configuration.

Returns bool :

Whether the key exists or not.

mlpy.tools.log.LoggingMgr

class `mlpy.tools.log.`**`LoggingMgr`**

Bases: `object`

The logging manager *`Singleton`* class.

The logger manager can be included as a member to any class to manager logging of information. Each logger is identified by the module id (*mid*), with which the logger settings can be changed.

By default a logger with log level LOG_INFO that is output to the stdout is created.

See also:

`logging`

Examples

```
>>> from mlpy.tools.log import LoggingMgr
>>> logger = LoggingMgr().get_logger('my_id')
>>> logger.info('This is a useful information.')
```

This gets a new logger. If a logger with the module id *my_id* already exists that logger will be returned, otherwise a logger with the default settings is created.

```
>>> LoggingMgr().add_handler('my_id', htype=LoggingMgr.LOG_TYPE_FILE)
```

This adds a new handler for the logger with module id *my_id* writing the logs to a file.

```
>>> LoggingMgr().remove_handler('my_id', htype=LoggingMgr.LOG_TYPE_STREAM)
```

This removes the stream handler from the logger with module id *my_id*.

```
>>> LoggingMgr().change_level('my_id', LoggingMgr.LOG_TYPE_ALL, LoggingMgr.LOG_
↪DEBUG)
```

This changes the log level for all attached handlers of the logger identified by *my_id* to LOG_DEBUG.

Attributes

LOG_TYPE_STREAM=0	Log only to output stream (stdout).
LOG_TYPE_FILE=1	Log only to an output file.
LOG_TYPE_ALL=2	Log to both output stream (stdout) and file.
LOG_DEBUG=10	Detailed information, typically of interest only when diagnosing problems.
LOG_INFO=20	Confirmation that things are working as expected.
LOG_WARNING=30	An indication that something unexpected happened, or indicative of some problem in the near future. The software is still working as expected.
LOG_ERROR=40	Due to a more serious problem, the software has not been able to perform some function.
LOG_CRITICAL=50	A serious error, indicating that the problem itself may be unable to continue running.

Methods

add_handler(mid[, htype, hlevel, fmt, filename])	Add a handler to the logger.
change_level(mid, hlevel[, htype])	Set the log level for a handler.
get_logger(mid[, level, htype, fmt, ...])	Get the logger instance with the identified *mid*.
get_verbosity(mid)	Gets the verbosity.
remove_handler(mid, htype)	Remove handlers.
set_verbosity(mid, value)	Sets the verbosity.

mlpy.tools.log.LoggingMgr.add_handler

`LoggingMgr.`**`add_handler`**(*mid*, *htype=0*, *hlevel=20*, *fmt=None*, *filename=None*)

Add a handler to the logger.

Parameters **mid** : str

The module id of the logger

htype : int, optional

The logging type to add to the handler. Default is LOG_TYPE_STREAM.

hlevel : int, optional

The logging level. Default is LOG_INFO.

fmt : str, optional

The format in which the information is presented. Default is "[%(levelname)-8s] %(name)s: %(funcName)s: %(message)s"

filename : str, optional

The name of the file the file handler writes the logs to. Default is a generated filename.

mlpy.tools.log.LoggingMgr.change_level

`LoggingMgr.`**`change_level`**(*mid*, *hlevel*, *htype=2*)

Set the log level for a handler.

Parameters **mid** : str

The module id of the logger

hlevel : int

The logging level.

htype : int, optional

The logging type of handler for which to change the log level. Default is LOG_TYPE_ALL.

mlpy.tools.log.LoggingMgr.get_logger

`LoggingMgr.`**`get_logger`**(*mid*, *level=20*, *htype=0*, *fmt=None*, *verbose=True*, *filename=None*)

Get the logger instance with the identified *mid*.

If a logger with the *mid* does not exist, a new logger will be created with the given settings. By default only a stream handler is attached to the logger.

Parameters **mid** : str

The module id of the logger.

level : int, optional

The top level logging level. Default is LOG_INFO.

htype : int, optional

The logging type of handler. Default is LOG_TYPE_STREAM.

fmt : str, optional

The format in which the information is presented. Default is "[%(levelname)-8s] %(name)s: %(funcName)s: %(message)s"

verbose : bool, optional

The verbosity setting of the logger. Default is True

filename : str, optional

The name of the file the file handler writes the logs to. Default is a generated filename.

Returns The logging instance.

mlpy.tools.log.LoggingMgr.get_verbosity

`LoggingMgr.`**`get_verbosity`**`(`*mid*`)`

Gets the verbosity.

The current setting of the verbosity of the logger identified by *mid* is returned.

Parameters **mid** : str

The module id of the logger to change the verbosity of.

Returns bool :

Whether to turn the verbosity on or off.

mlpy.tools.log.LoggingMgr.remove_handler

`LoggingMgr.`**`remove_handler`**`(`*mid*, *htype*`)`

Remove handlers.

Removes all handlers of the given handler type from the logger.

Parameters **mid** : str

The module id of the logger

htype : int

The logging type to remove from the handler.

mlpy.tools.log.LoggingMgr.set_verbosity

`LoggingMgr.set_verbosity(mid, value)`

Sets the verbosity.

Turn logging on/off for logger identified by *mid*.

Parameters **mid** : str

The module id of the logger to change the verbosity of.

value : bool

Whether to turn the verbosity on or off.

Waiting	The waiting class.

mlpy.tools.misc.Waiting

class `mlpy.tools.misc.Waiting(text=None)`

Bases: `threading.Thread`

The waiting class.

The waiting class prints dots (.) on stdout to indicate that a process is running. The waiting process runs on a different thread to not disturbed the running process.

Examples

```
>>> def long_process():
...     for i in xrange(20):
...         pass
...
>>> w = Waiting("processing")
>>>
>>> w.start()
>>> long_process()
>>> w.stop()
processing ......
```

Attributes

daemon	A boolean value indicating whether this thread is a daemon thread (True) or not (False).
ident	Thread identifier of this thread or None if it has not been started.
name	A string used for identification purposes only.

mlpy.tools.misc.Waiting.daemon

`Waiting.daemon`

A boolean value indicating whether this thread is a daemon thread (True) or not (False).

This must be set before start() is called, otherwise RuntimeError is raised. Its initial value is inherited from the creating thread; the main thread is not a daemon thread and therefore all threads created in the main thread default to daemon = False.

The entire Python program exits when no alive non-daemon threads are left.

mlpy.tools.misc.Waiting.ident

`Waiting.`**`ident`**

Thread identifier of this thread or None if it has not been started.

This is a nonzero integer. See the thread.get_ident() function. Thread identifiers may be recycled when a thread exits and another thread is created. The identifier is available even after the thread has exited.

mlpy.tools.misc.Waiting.name

`Waiting.`**`name`**

A string used for identification purposes only.

It has no semantics. Multiple threads may be given the same name. The initial name is set by the constructor.

Methods

getName()	
isAlive()	Return whether the thread is alive.
isDaemon()	
is_alive()	Return whether the thread is alive.
join([timeout])	Wait until the thread terminates.
run()	Run the process.
setDaemon(daemonic)	
setName(name)	
start()	Start the process.
stop()	End the process.

mlpy.tools.misc.Waiting.getName

`Waiting.`**`getName`**`()`

mlpy.tools.misc.Waiting.isAlive

`Waiting.`**`isAlive`**`()`

Return whether the thread is alive.

This method returns True just before the run() method starts until just after the run() method terminates. The module function enumerate() returns a list of all alive threads.

mlpy.tools.misc.Waiting.isDaemon

`Waiting.`**`isDaemon`**`()`

mlpy.tools.misc.Waiting.is_alive

`Waiting.`**`is_alive`**`()`

Return whether the thread is alive.

This method returns True just before the run() method starts until just after the run() method terminates. The module function enumerate() returns a list of all alive threads.

mlpy.tools.misc.Waiting.join

`Waiting.`**`join`**`(`*timeout=None*`)`

Wait until the thread terminates.

This blocks the calling thread until the thread whose join() method is called terminates – either normally or through an unhandled exception or until the optional timeout occurs.

When the timeout argument is present and not None, it should be a floating point number specifying a timeout for the operation in seconds (or fractions thereof). As join() always returns None, you must call isAlive() after join() to decide whether a timeout happened – if the thread is still alive, the join() call timed out.

When the timeout argument is not present or None, the operation will block until the thread terminates.

A thread can be join()ed many times.

join() raises a RuntimeError if an attempt is made to join the current thread as that would cause a deadlock. It is also an error to join() a thread before it has been started and attempts to do so raises the same exception.

mlpy.tools.misc.Waiting.run

`Waiting.`**`run`**`()`

Run the process.

This method is automatically called by the thead.

mlpy.tools.misc.Waiting.setDaemon

`Waiting.`**`setDaemon`**`(`*daemonic*`)`

mlpy.tools.misc.Waiting.setName

`Waiting.`**`setName`**`(`*name*`)`

mlpy.tools.misc.Waiting.start

`Waiting.`**`start`**`()`

Start the process.

mlpy.tools.misc.Waiting.stop

`Waiting.stop()`

End the process.

CHAPTER 24

Indices and tables

- genindex
- modindex
- search

Bibliography

[R1] Wikipidia::cosine_similarity

[R3] Hester, Todd, and Peter Stone. “Generalized model learning for reinforcement learning in factored domains.” Proceedings of The 8th International Conference on Autonomous Agents and Multiagent Systems-Volume 2. International Foundation for Autonomous Agents and Multiagent Systems, 2009.

[R2] Abbeel, Pieter, and Andrew Y. Ng. “Apprenticeship learning via inverse reinforcement learning.” Proceedings of the twenty-first international conference on Machine learning. ACM, 2004.

[R4] Hester, Todd, and Peter Stone. “Generalized model learning for reinforcement learning in factored domains.” Proceedings of The 8th International Conference on Autonomous Agents and Multiagent Systems-Volume 2. International Foundation for Autonomous Agents and Multiagent Systems, 2009.

Python Module Index

Index