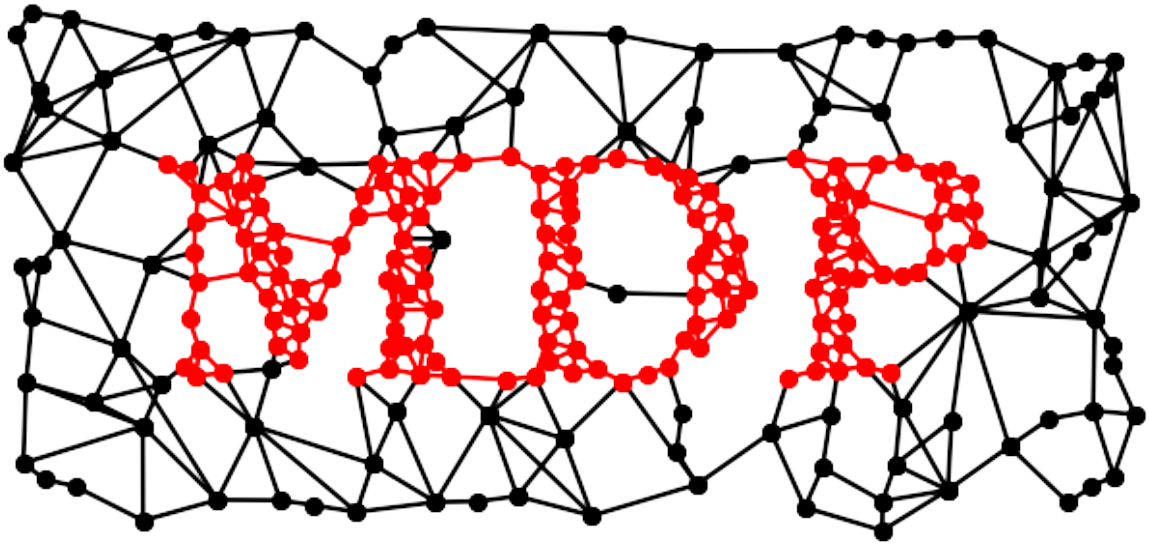


Modular toolkit for Data Processing

Tutorial

Release 3.0

Authors: MDP Developers

January 17, 2011

CONTENTS

Authors MDP Developers

Homepage http://mdp-toolkit.sourceforge.net

Contact mdp-toolkit-users@lists.sourceforge.net

Version 3.0

This document is also available online.

This is a guide to basic and some more advanced features of the MDP library. Besides the present tutorial, you can learn more about MDP by using the standard Python tools. All MDP nodes have doc-strings, the public attributes and methods have telling names: All information about a node can be obtained using the `help` and `dir` functions within the Python interpreter. In addition to that, an automatically generated API documentation is available.

Note: Code snippets throughout the script will be denoted by:

```
>>> print "Hello world!"
Hello world!
```

To run the following code examples don't forget to import `mdp` and `numpy` in your Python session with:

```
>>> import mdp
>>> import numpy as np
```

You'll find all the code of this tutorial online.

CHAPTER

ONE

QUICK START

Using MDP is as easy as:

```
>>> import mdp
>>> # perform pca on some data x
...
>>> y = mdp.pca(x)
>>> # perform ica on some data x using single precision
...
>>> y = mdp.fastica(x, dtype='float32')
```

MDP requires the numerical Python extensions NumPy or SciPy. At import time MDP will select `scipy` if available, otherwise `numpy` will be loaded. You can force the use of a numerical extension by setting the environment variable `MDPNUMX=numpy` or `MDPNUMX=scipy`.

An important remark

Input array data is typically assumed to be two-dimensional and ordered such that observations of the same variable are stored on rows and different variables are stored on columns.

CHAPTER

TWO

INTRODUCTION

The use of the Python programming language in computational neuroscience has been growing steadily over the past few years. The maturation of two important open source projects, the scientific libraries NumPy and SciPy, gives access to a large collection of scientific functions which rival in size and speed those from well known commercial alternatives such as *Matlab*® from The MathWorks™.

Furthermore, the flexible and dynamic nature of Python offers scientific programmers the opportunity to quickly develop efficient and structured software while maximizing prototyping and reusability capabilities.

The Modular toolkit for Data Processing (MDP) package is a library of widely used data processing algorithms, and the possibility to combine them together to form pipelines for building more complex data processing software.

MDP has been designed to be used as-is and as a framework for scientific data processing development.

From the user's perspective, MDP consists of a collection of *units*, which process data. For example, these include algorithms for supervised and unsupervised learning, principal and independent components analysis and classification.

These units can be chained into data processing flows, to create pipelines as well as more complex feed-forward network architectures. Given a set of input data, MDP takes care of training and executing all nodes in the network in the correct order and passing intermediate data between the nodes. This allows the user to specify complex algorithms as a series of simpler data processing steps.

The number of available algorithms is steadily increasing and includes signal processing methods (Principal Component Analysis, Independent Component Analysis, Slow Feature Analysis), manifold learning methods ([Hessian] Locally Linear Embedding), several classifiers, probabilistic methods (Factor Analysis, RBM), data preprocessing methods, and many others.

Particular care has been taken to make computations efficient in terms of speed and memory. To reduce the memory footprint, it is possible to perform learning using batches of data. For large data-sets, it is also possible to specify that MDP should use single precision floating point numbers rather than double precision ones. Finally, calculations can be parallelised using the `parallel` subpackage, which offers a parallel implementation of the basic nodes and flows.

From the developer's perspective, MDP is a framework that makes the implementation of new supervised and unsupervised learning algorithms easy and straightforward. The basic class, `Node`, takes care of tedious tasks like numerical type and dimensionality checking, leaving the developer free to concentrate on the implementation of the learning and execution phases. Because of the common interface, the node then automatically integrates with the rest of the library and can be used in a network together with other nodes.

A node can have multiple training phases and even an undetermined number of phases. Multiple training phases mean that the training data is presented multiple times to the same node. This allows the implementation of algorithms that need to collect some statistics on the whole input before proceeding with the actual training, and others that need to iterate over a training phase until a convergence criterion is satisfied. It is possible to train each phase using chunks of input data if the chunks are given as an iterable. Moreover, crash recovery can be optionally enabled, which will save the state of the flow in case of a failure for later inspection.

MDP is distributed under the open source BSD license. It has been written in the context of theoretical research in neuroscience, but it has been designed to be helpful in any context where trainable data processing algorithms

are used. Its simplicity on the user's side, the variety of readily available algorithms, and the reusability of the implemented nodes also make it a useful educational tool.

http://mdp-toolkit.sourceforge.net

With over 20,000 downloads since its first public release in 2004, MDP has become a widely used Python scientific software. It has minimal dependencies, requiring only the NumPy numerical extension, is completely platform-independent, and is available in several Linux distribution, and the Python(x,y) scientific Python distribution.

As the number of users and contributors is increasing, MDP appears to be a good candidate for becoming a community-driven common repository of user-supplied, freely available, Python implemented data processing algorithms.

CHAPTER

THREE

NODES

CodeSnippet

You can download all the code on this page from the code snippets directory

A *node* is the basic building block of an MDP application. It represents a data processing element, for example a learning algorithm, a data filter, or a visualization step (see the *Node List* section for an exhaustive list and references).

Each node can have one or more *training phases*, during which the internal structures are learned from training data (e.g. the weights of a neural network are adapted or the covariance matrix is estimated) and an *execution phase*, where new data can be processed forwards (by processing the data through the node) or backwards (by applying the inverse of the transformation computed by the node if defined).

Nodes have been designed to be applied to arbitrarily long sets of data; provided the underlying algorithms support it, the internal structures can be updated incrementally by sending multiple batches of data (this is equivalent to online learning if the chunks consists of single observations, or to batch learning if the whole data is sent in a single chunk). This makes it possible to perform computations on large amounts of data that would not fit into memory and to generate data on-the-fly.

A Node also defines some utility methods, for example copy, which returns an exact copy of a node, and save, which writes to in a file. Additional methods may also be present, depending on the algorithm.

3.1 Node Instantiation

A node can be obtained by creating an instance of the `Node` class.

Each node is characterized by an input dimension (i.e., the dimensionality of the input vectors), an output dimension, and a `dtype`, which determines the numerical type of the internal structures and of the output signal. By default, these attributes are inherited from the input data if left unspecified. The constructor of each node class can require other task-specific arguments. The full documentation is always available in the doc-string of the node's class.

3.1.1 Some examples of node instantiation

Create a node that performs Principal Component Analysis (PCA) whose input dimension and `dtype` are inherited from the input data during training. Output dimensions default to input dimensions.

```
>>> pcanode1 = mdp.nodes.PCANode()
>>> pcanode1
PCANode(input_dim=None, output_dim=None, dtype=None)
```

Setting `output_dim = 10` means that the node will keep only the first 10 principal components of the input.

```
>>> pcanode2 = mdp.nodes.PCANode(output_dim=10)
>>> pcanode2
PCANode(input_dim=None, output_dim=10, dtype=None)
```

The output dimensionality can also be specified in terms of the explained variance. If we want to keep the number of principal components which can account for 80% of the input variance, we set

```
>>> pcanode3 = mdp.nodes.PCANode(output_dim=0.8)
>>> pcanode3.desired_variance
0.80000000000000004
```

If `dtype` is set to `float32` (32-bit float), the input data is cast to single precision when received and the internal structures are also stored as `float32`. `dtype` influences the memory space necessary for a node and the precision with which the computations are performed.

```
>>> pcanode4 = mdp.nodes.PCANode(dtype='float32')
>>> pcanode4
PCANode(input_dim=None, output_dim=None, dtype='float32')
```

You can obtain a list of the numerical types supported by a node looking at its `supported_dtypes` property

```
>>> pcanode4.supported_dtypes
[dtype('float32'), dtype('float64')...]
```

This attribute is a list of `numpy.dtype` objects.

A `PolynomialExpansionNode` expands its input in the space of polynomials of a given degree by computing all monomials up to the specified degree. Its constructor needs as first argument the degree of the polynomials space (3 in this case):

```
>>> expnode = mdp.nodes.PolynomialExpansionNode(3)
```

3.2 Node Training

Some nodes need to be trained to perform their task. For example, the Principal Component Analysis (PCA) algorithm requires the computation of the mean and covariance matrix of a set of training data from which the principal eigenvectors of the data distribution are estimated.

This can be done during a training phases by calling the `train` method. MDP supports both supervised and unsupervised training, and algorithms with multiple training phases.

Some examples of node training:

Create some random data to train the node

```
>>> x = np.random.random((100, 25))  # 25 variables, 100 observations
```

Analyzes the batch of data `x` and update the estimation of mean and covariance matrix

```
>>> pcanode1.train(x)
```

At this point the input dimension and the `dtype` have been inherited from `x`

```
>>> pcanode1
PCANode(input_dim=25, output_dim=None, dtype='float64')
```

We can train our node with more than one chunk of data. This is especially useful when the input data is too long to be stored in memory or when it has to be created on-the-fly. (See also the *Iterables* section)

```
>>> for i in range(100):
...     x = np.random.random((100, 25))
...     pcanode1.train(x)
```

Some nodes don't need to or cannot be trained

```
>>> expnode.is_trainable()
False
```

Trying to train them anyway would raise an `IsNotTrainableException`.

The training phase ends when the `stop_training`, `execute`, `inverse`, and possibly some other node-specific methods are called. For example we can finalize the PCA algorithm by computing and selecting the principal eigenvectors

```
>>> pcanode1.stop_training()
```

If the `PCANode` was declared to have a number of output components dependent on the input variance to be explained, we can check after training the number of output components and the actually explained variance

```
>>> pcanode3.train(x)
>>> pcanode3.stop_training()
>>> pcanode3.output_dim
16
>>> pcanode3.explained_variance
0.85261144755506446
```

It is now possible to access the trained internal data. In general, a list of the interesting internal attributes can be found in the class documentation.

```
>>> avg = pcanode1.avg             # mean of the input data
>>> v = pcanode1.get_projmatrix() # projection matrix
```

Some nodes, namely the one corresponding to supervised algorithms, e.g. Fisher Discriminant Analysis (FDA), may need some labels or other supervised signals to be passed during training. Detailed information about the signature of the `train` method can be read in its doc-string.

```
>>> fdanode = mdp.nodes.FDANode()
>>> for label in ['a', 'b', 'c']:
...     x = np.random.random((100, 25))
...     fdanode.train(x, label)
```

A node could also require multiple training phases. For example, the training of `fdanode` is not complete yet, since it has two training phases: The first one computing the mean of the data conditioned on the labels, and the second one computing the overall and within-class covariance matrices and solving the FDA problem. The first phase must be stopped and the second one trained

```
>>> fdanode.stop_training()
>>> for label in ['a', 'b', 'c']:
...     x = np.random.random((100, 25))
...     fdanode.train(x, label)
```

The easiest way to train multiple phase nodes is using flows, which automatically handle multiple phases (see the *Flows* section).

3.3 Node Execution

Once the training is finished, it is possible to execute the node:

The input data is projected on the principal components learned in the training phase

```
>>> x = np.random.random((100, 25))
>>> y_pca = pcanode1.execute(x)
```

Calling a node instance is equivalent to executing it

```
>>> y_pca = pcanode1(x)
```

The input data is expanded in the space of polynomials of degree 3

```
>>> x = np.random.random((100, 5))
>>> y_exp = expnode(x)
```

The input data is projected to the directions learned by FDA

```
>>> x = np.random.random((100, 25))
>>> y_fda = fdanode(x)
```

Some nodes may allow for optional arguments in the `execute` method. As always the complete information can be found in the doc-string.

3.4 Node Inversion

If the operation computed by the node is invertible, the node can also be executed *backwards*, thus computing the inverse transformation:

In the case of PCA, for example, this corresponds to projecting a vector in the principal components space back to the original data space

```
>>> pcanode1.is_invertible()
True
>>> x = pcanode1.inverse(y_pca)
```

The expansion node in not invertible

```
>>> expnode.is_invertible()
False
```

Trying to compute the inverse would raise an `IsNotInvertibleException`.

3.5 Writing your own nodes: subclassing `Node`

MDP tries to make it easy to write new nodes that interface with the existing data processing elements.

The `Node` class is designed to make the implementation of new algorithms easy and intuitive. This base class takes care of setting input and output dimension and casting the data to match the numerical type (e.g. `float` or `double`) of the internal variables, and offers utility methods that can be used by the developer.

To expand the MDP library of implemented nodes with user-made nodes, it is sufficient to subclass `Node`, overriding some of the methods according to the algorithm one wants to implement, typically the `_train`, `_stop_training`, and `_execute` methods.

In its namespace MDP offers references to the main modules `numpy` or `scipy`, and the subpackages `linalg`, `random`, and `fft` as `mdp.numx`, `mdp.numx_linalg`, `mdp.numx_rand`, and `mdp.numx_fft`. This is done to possibly support additional numerical extensions in the future. For this reason it is recommended to refer to the `numpy` or `scipy` numerical extensions through the MDP aliases `mdp.numx`, `mdp.numx_linalg`, `mdp.numx_fft`, and `mdp.numx_rand` when writing `Node` subclasses. This shall ensure that your nodes can be used without modifications should MDP support alternative numerical extensions in the future.

We'll illustrate all this with some toy examples.

We start by defining a node that multiplies its input by 2.

Define the class as a subclass of `Node`:

```
>>> class TimesTwoNode(mdp.Node):
```

This node cannot be trained. To specify this, one has to overwrite the `is_trainable` method to return False:

```
...     def is_trainable(self):
...         return False
```

Execute only needs to multiply `x` by 2:

```
...     def _execute(self, x):
...         return 2*x
```

Note that the `execute` method, which should never be overwritten and which is inherited from the `Node` parent class, will perform some tests, for example to make sure that `x` has the right rank, dimensionality and casts it to have the right `dtype`. After that the user-supplied `_execute` method is called. Each subclass has to handle the `dtype` defined by the user or inherited by the input data, and make sure that internal structures are stored consistently. To help with this the `Node` base class has a method called `_refcast(array)` that casts the input `array` only when its `dtype` is different from the `Node` instance's `dtype`.

The inverse of the multiplication by 2 is of course the division by 2

```
...     def _inverse(self, y):
...         return y/2
```

Test the new node

```
>>> class TimesTwoNode(mdp.Node):
...      def is_trainable(self):
...          return False
...      def _execute(self, x):
...          return 2*x
...      def _inverse(self, y):
...          return y/2
>>> node = TimesTwoNode(dtype = 'float32')
>>> x = mdp.numx.array([[1.0, 2.0, 3.0]])
>>> y = node(x)
>>> print x, '* 2 =  ', y
[[ 1.  2.  3.]] * 2 =   [[ 2.  4.  6.]]
>>> print y, '/ 2 =', node.inverse(y)
[[ 2.  4.  6.]] / 2 = [[ 1.  2.  3.]]
```

We then define a node that raises the input to the power specified in the initialiser:

```
>>> class PowerNode(mdp.Node):
```

We redefine the init method to take the power as first argument. In general one should always give the possibility to set the `dtype` and the input dimensions. The default value is `None`, which means that the exact value is going to be inherited from the input data:

```
...     def __init__(self, power, input_dim=None, dtype=None):
```

Initialize the parent class:

```
...         super(PowerNode, self).__init__(input_dim=input_dim, dtype=dtype)
```

Store the power:

```
...         self.power = power
```

`PowerNode` is not trainable:

```
...     def is_trainable(self):
...         return False
```

nor invertible:

```
...     def is_invertible(self):
...         return False
```

It is possible to overwrite the function `_get_supported_dtypes` to return a list of `dtype` supported by the node:

```
...     def _get_supported_dtypes(self):
...         return ['float32', 'float64']
```

The supported types can be specified in any format allowed by the `numpy.dtype` constructor. The interface method `get_supported_dtypes` converts them and sets the property `supported_dtypes`, which is a list of `numpy.dtype` objects.

The `_execute` method:

```
...     def _execute(self, x):
...         return self._refcast(x**self.power)
```

Test the new node

```
>>> class PowerNode(mdp.Node):
...     def __init__(self, power, input_dim=None, dtype=None):
...         super(PowerNode, self).__init__(input_dim=input_dim, dtype=dtype)
...         self.power = power
...     def is_trainable(self):
...         return False
...     def is_invertible(self):
...         return False
...     def _get_supported_dtypes(self):
...         return ['float32', 'float64']
...     def _execute(self, x):
...         return self._refcast(x**self.power)
>>> node = PowerNode(3)
>>> x = mdp.numx.array([[1.0, 2.0, 3.0]])
>>> y = node(x)
>>> print x, '**', node.power, '=', node(x)
[[ 1.  2.  3.]] ** 3 = [[  1.   8.  27.]]
```

We now define a node that needs to be trained. The `MeanFreeNode` computes the mean of its training data and subtracts it from the input during execution:

```
>>> class MeanFreeNode(mdp.Node):
...     def __init__(self, input_dim=None, dtype=None):
...         super(MeanFreeNode, self).__init__(input_dim=input_dim,
...                                            dtype=dtype)
```

We store the mean of the input data in an attribute. We initialize it to `None` since we still don't know how large is an input vector:

```
...         self.avg = None
```

Same for the number of training points:

```
...         self.tlen = 0
```

The subclass only needs to overwrite the `_train` method, which will be called by the parent `train` after some testing and casting has been done:

```
...     def _train(self, x):
...         # Initialize the mean vector with the right
...         # size and dtype if necessary:
...         if self.avg is None:
...             self.avg = mdp.numx.zeros(self.input_dim,
...                                       dtype=self.dtype)
```

Update the mean with the sum of the new data:

```
...         self.avg += mdp.numx.sum(x, axis=0)
```

Count the number of points processed:

```
...         self.tlen += x.shape [0]
```

Note that the `train` method can have further arguments, which might be useful to implement algorithms that require supervised learning. For example, if you want to define a node that performs some form of classification

you can define a `_train(self, data, labels)` method. The parent `train` checks `data` and takes care to pass the `labels` on (cf. for example `mdp.nodes.FDANode`).

The `_stop_training` function is called by the parent `stop_training` method when the training phase is over. We divide the sum of the training data by the number of training vectors to obtain the mean:

```
...     def _stop_training(self):
...         self.avg /= self.tlen
...         if self.output_dim is None:
...             self.output_dim = self.input_dim
```

Note that we `input_dim` are set automatically by the `train` method, and we want to ensure that the node has `output_dim` set after training. For nodes that do not need training, the setting is performed automatically upon execution. The `_execute` and `_inverse` methods:

```
...     def _execute(self, x):
...         return x - self.avg
...     def _inverse(self, y):
...         return y + self.avg
```

Test the new node

```
>>> class MeanFreeNode(mdp.Node):
...     def __init__(self, input_dim=None, dtype=None):
...         super(MeanFreeNode, self).__init__(input_dim=input_dim,
...                                            dtype=dtype)
...         self.avg = None
...         self.tlen = 0
...     def _train(self, x):
...         # Initialize the mean vector with the right
...         # size and dtype if necessary:
...         if self.avg is None:
...             self.avg = mdp.numx.zeros(self.input_dim,
...                                       dtype=self.dtype)
...         self.avg += mdp.numx.sum(x, axis=0)
...         self.tlen += x.shape[0]
...     def _stop_training(self):
...         self.avg /= self.tlen
...         if self.output_dim is None:
...             self.output_dim = self.input_dim
...     def _execute(self, x):
...         return x - self.avg
...     def _inverse(self, y):
...         return y + self.avg
>>> node = MeanFreeNode()
>>> x = np.random.random((10,4))
>>> node.train(x)
>>> y = node(x)
>>> print 'Mean of y (should be zero):\n', np.abs(np.around(np.mean(y, 0), 15))
Mean of y (should be zero):
[ 0.  0.  0.  0.]
```

It is also possible to define nodes with multiple training phases. In such a case, calling the `train` and `stop_training` functions multiple times is going to execute successive training phases (this kind of node is much easier to train using *Flows*). Here we'll define a node that returns a meanfree, unit variance signal. We define two training phases: first we compute the mean of the signal and next we sum the squared, meanfree input to compute the standard deviation (of course it is possible to solve this problem in one single step - remember this is just a toy example).

```
>>> class UnitVarianceNode(mdp.Node):
...     def __init__(self, input_dim=None, dtype=None):
...         super(UnitVarianceNode, self).__init__(input_dim=input_dim,
...                                                dtype=dtype)
...         self.avg = None # average
```

```
...         self.std = None # standard deviation
...         self.tlen = 0
```

The training sequence is defined by the user-supplied method `_get_train_seq`, that returns a list of tuples, one for each training phase. The tuples contain references to the training and stop-training methods of each of them. The default output of this method is `[(_train, _stop_training)]`, which explains the standard behavior illustrated above. We overwrite the method to return the list of our training/stop_training methods:

```
...     def _get_train_seq(self):
...         return [(self._train_mean, self._stop_mean),
...                 (self._train_std, self._stop_std)]
```

Next we define the training methods. The first phase is identical to the one in the previous example:

```
...     def _train_mean(self, x):
...         if self.avg is None:
...             self.avg = mdp.numx.zeros(self.input_dim,
...                                       dtype=self.dtype)
...         self.avg += mdp.numx.sum(x, 0)
...         self.tlen += x.shape[0]
...     def _stop_mean(self):
...         self.avg /= self.tlen
```

The second one is only marginally different and does not require many explanations:

```
...     def _train_std(self, x):
...         if self.std is None:
...             self.tlen = 0
...             self.std = mdp.numx.zeros(self.input_dim,
...                                       dtype=self.dtype)
...         self.std += mdp.numx.sum((x - self.avg)**2., 0)
...         self.tlen += x.shape[0]
...     def _stop_std(self):
...         # compute the standard deviation
...         self.std = mdp.numx.sqrt(self.std/(self.tlen-1))
```

The `_execute` and `_inverse` methods are not surprising, either:

```
...     def _execute(self, x):
...         return (x - self.avg)/self.std
...     def _inverse(self, y):
...         return y*self.std + self.avg
```

Test the new node

```
>>> class UnitVarianceNode(mdp.Node):
...     def __init__(self, input_dim=None, dtype=None):
...         super(UnitVarianceNode, self).__init__(input_dim=input_dim,
...                                                dtype=dtype)
...         self.avg = None # average
...         self.std = None # standard deviation
...         self.tlen = 0
...     def _get_train_seq(self):
...         return [(self._train_mean, self._stop_mean),
...                 (self._train_std, self._stop_std)]
...     def _train_mean(self, x):
...         if self.avg is None:
...             self.avg = mdp.numx.zeros(self.input_dim,
...                                       dtype=self.dtype)
...         self.avg += mdp.numx.sum(x, 0)
...         self.tlen += x.shape[0]
...     def _stop_mean(self):
...         self.avg /= self.tlen
...     def _train_std(self, x):
```

```
...         if self.std is None:
...             self.tlen = 0
...             self.std = mdp.numx.zeros(self.input_dim,
...                                       dtype=self.dtype)
...         self.std += mdp.numx.sum((x - self.avg)**2., 0)
...         self.tlen += x.shape[0]
...     def _stop_std(self):
...         # compute the standard deviation
...         self.std = mdp.numx.sqrt(self.std/(self.tlen-1))
...     def _execute(self, x):
...         return (x - self.avg)/self.std
...     def _inverse(self, y):
...         return y*self.std + self.avg
>>> node = UnitVarianceNode()
>>> x = np.random.random((10,4))
>>> # loop over phases
... for phase in range(2):
...     node.train(x)
...     node.stop_training()
...
...
>>> # execute
... y = node(x)
>>> print 'Standard deviation of y (should be one): ', mdp.numx.std(y, axis=0, ddof=1)
Standard deviation of y (should be one):  [ 1.  1.  1.  1.]
```

In our last example we'll define a node that returns two copies of its input. The output is going to have twice as many dimensions.

```
>>> class TwiceNode(mdp.Node):
...     def is_trainable(self): return False
...     def is_invertible(self): return False
```

When `Node` inherits the input dimension, output dimension, and `dtype` from the input data, it calls the methods `set_input_dim`, `set_output_dim`, and `set_dtype`. Those are the setters for `input_dim`, `output_dim` and `dtype`, which are Python properties. If a subclass needs to change the default behavior, the internal methods `_set_input_dim`, `_set_output_dim` and `_set_dtype` can be overwritten. The property setter will call the internal method after some basic testing and internal settings. The private methods `_set_input_dim`, `_set_output_dim` and `_set_dtype` are responsible for setting the private attributes `_input_dim`, `_output_dim`, and `_dtype` that contain the actual value.

Here we overwrite `_set_input_dim` to automatically set the output dimension to be twice the input one, and `_set_output_dim` to raise an exception, since the output dimension should not be set explicitly.

```
...     def _set_input_dim(self, n):
...         self._input_dim = n
...         self._output_dim = 2*n
...     def _set_output_dim(self, n):
...         raise mdp.NodeException, "Output dim can not be set explicitly!"
```

The `_execute` method:

```
...     def _execute(self, x):
...         return mdp.numx.concatenate((x, x), 1)
```

Test the new node

```
>>> class TwiceNode(mdp.Node):
...     def is_trainable(self): return False
...     def is_invertible(self): return False
...     def _set_input_dim(self, n):
...         self._input_dim = n
...         self._output_dim = 2*n
...     def _set_output_dim(self, n):
```

```
...             raise mdp.NodeException, "Output dim can not be set explicitly!"
...         def _execute(self, x):
...             return mdp.numx.concatenate((x, x), 1)
>>> node = TwiceNode()
>>> x = mdp.numx.zeros((5,2))
>>> x
array([[ 0.,  0.],
       [ 0.,  0.],
       [ 0.,  0.],
       [ 0.,  0.],
       [ 0.,  0.]])
>>> node.execute(x)
array([[ 0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.]])
```

CHAPTER

FOUR

FLOWS

CodeSnippet

You can download all the code on this page from the code snippets directory

A *flow* is a sequence of nodes that are trained and executed together to form a more complex algorithm. Input data is sent to the first node and is successively processed by the subsequent nodes along the sequence.

Using a flow as opposed to handling manually a set of nodes has a clear advantage: The general flow implementation automatizes the training (including supervised training and multiple training phases), execution, and inverse execution (if defined) of the whole sequence.

Crash recovery is optionally available: in case of failure the current state of the flow is saved for later inspection. A subclass of the basic flow class (`CheckpointFlow`) allows user-supplied checkpoint functions to be executed at the end of each phase, for example to save the internal structures of a node for later analysis. Flow objects are Python containers. Most of the builtin `list` methods are available. A `Flow` can be saved or copied using the corresponding `save` and `copy` methods.

4.1 Flow instantiation, training and execution

For example, suppose we need to analyze a very high-dimensional input signal using Independent Component Analysis (ICA). To reduce the computational load, we would like to reduce the input dimensionality of the data using PCA. Moreover, we would like to find the data that produces local maxima in the output of the ICA components on a new test set (this information could be used for instance to characterize the ICA filters).

We start by generating some input signal at random (which makes the example useless, but it's just for illustration...). Generate 1000 observations of 20 independent source signals

```
>>> inp = np.random.random((1000, 20))
```

Rescale x to have zero mean and unit variance

```
>>> inp = (inp - np.mean(inp, 0))/np.std(inp, axis=0, ddof=0)
```

We reduce the variance of the last 15 components, so that they are going to be eliminated by PCA

```
>>> inp[:,5:] /= 10.0
```

Mix the input signals linearly

```
>>> x = mdp.utils.mult(inp,np.random.random((20, 20)))
```

`x` is now the training data for our simulation. In the same way we also create a test set `x_test`.

```
>>> inp_test = np.random.random((1000, 20))
>>> inp_test = (inp_test - np.mean(inp_test, 0))/np.std(inp_test, 0)
>>> inp_test[:,5:] /= 10.0
>>> x_test = mdp.utils.mult(inp_test, np.random.random((20, 20)))
```

We could now perform our analysis using only nodes, that's the lengthy way...

1. Perform PCA

```
>>> pca = mdp.nodes.PCANode(output_dim=5)
>>> pca.train(x)
>>> out1 = pca(x)
```

2. Perform ICA using CuBICA algorithm

```
>>> ica = mdp.nodes.CuBICANode()
>>> ica.train(out1)
>>> out2 = ica(out1)
```

3. Find the three largest local maxima in the output of the ICA node when applied to the test data, using a `HitParadeNode`

```
>>> out1_test = pca(x_test)
>>> out2_test = ica(out1_test)
>>> hitnode = mdp.nodes.HitParadeNode(3)
>>> hitnode.train(out2_test)
>>> maxima, indices = hitnode.get_maxima()
```

or we could use flows, which is the best way

```
>>> flow = mdp.Flow([mdp.nodes.PCANode(output_dim=5), mdp.nodes.CuBICANode()])
```

Note that flows can be built simply by concatenating nodes

```
>>> flow = mdp.nodes.PCANode(output_dim=5) + mdp.nodes.CuBICANode()
```

Train the resulting flow

```
>>> flow.train(x)
```

Now the training phase of PCA and ICA are completed. Next we append a `HitParadeNode` which we want to train on the test data

```
>>> flow.append(mdp.nodes.HitParadeNode(3))
```

As before, new nodes can be appended to an existing flow by adding them ot it

```
>>> flow += mdp.nodes.HitParadeNode(3)
```

Train the `HitParadeNode` on the test data

```
>>> flow.train(x_test)
>>> maxima, indices = flow[2].get_maxima()
```

A single call to the `flow`'s `train` method will automatically take care of training nodes with multiple training phases, if such nodes are present.

Just to check that everything works properly, we can calculate covariance between the generated sources and the output (should be approximately 1)

```
>>> out = flow.execute(x)
>>> cov = np.amax(abs(mdp.utils.cov2(inp[:,:5], out)), axis=1)
>>> print cov
[ 0.9957042   0.98482351  0.99557617  0.99680391  0.99232424]
```

The `HitParadeNode` is an analysis node and as such does not interfere with the data flow.

Note that flows can be executed by calling the `Flow` instance directly

```
>>> out = flow(x)
```

4.2 Flow inversion

Flows can be inverted by calling their `inverse` method. In the case where the flow contains non-invertible nodes, trying to invert it would raise an exception. In this case, however, all nodes are invertible. We can reconstruct the mix by inverting the flow

```
>>> rec = flow.inverse(out)
```

Calculate covariance between input mix and reconstructed mix: (should be approximately 1)

```
>>> cov = np.amax(abs(mdp.utils.cov2(x/np.std(x,axis=0),
...                                  rec/np.std(rec,axis=0))))
>>> print cov
0.999622205447
```

4.3 Flows are container type objects

`Flow` objects are defined as Python containers, and thus are endowed with most of the methods of Python lists.

You can loop through a `Flow`

```
>>> for node in flow:
...     print repr(node)
PCANode(input_dim=20, output_dim=5, dtype='float64')
CuBICANode(input_dim=5, output_dim=5, dtype='float64')
HitParadeNode(input_dim=5, output_dim=5, dtype='float64')
HitParadeNode(input_dim=5, output_dim=5, dtype='float64')
```

You can get slices, `pop`, `insert`, and `append` nodes

```
>>> len(flow)
4
>>> print flow[::2]
[PCANode, HitParadeNode]
>>> nodetoberemoved = flow.pop(-1)
>>> nodetoberemoved
HitParadeNode(input_dim=5, output_dim=5, dtype='float64')
>>> len(flow)
3
```

Finally, you can concatenate flows

```
>>> dummyflow = flow[1:].copy()
>>> longflow = flow + dummyflow
>>> len(longflow)
5
```

The returned flow must always be consistent, i.e. input and output dimensions of successive nodes always have to match. If you try to create an inconsistent flow you'll get an exception.

4.4 Crash recovery

If a node in a flow fails, you'll get a traceback that tells you which node has failed. You can also switch the crash recovery capability on. If something goes wrong you'll end up with a pickle dump of the flow, that can be later inspected.

To see how it works let's define a bogus node that always throws an `Exception` and put it into a flow

```
>>> class BogusExceptNode(mdp.Node):
...     def train(self,x):
...         self.bogus_attr = 1
...         raise Exception, "Bogus Exception"
...     def execute(self,x):
...         raise Exception, "Bogus Exception"
...
>>> flow = mdp.Flow([BogusExceptNode()])
```

Switch on crash recovery

```
>>> flow.set_crash_recovery(1)
```

Attempt to train the flow

```
>>> flow.train(x)
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
  [...]
mdp.linear_flows.FlowExceptionCR:
----------------------------------------
! Exception in node #0 (BogusExceptNode):
Node Traceback:
Traceback (most recent call last):
  [...]
Exception: Bogus Exception
----------------------------------------
A crash dump is available on: "/tmp/MDPcrash_LmISO_.pic"
```

You can give a file name to tell the flow where to save the dump:

```
>>> flow.set_crash_recovery('/home/myself/mydumps/MDPdump.pic')
```

CHAPTER

FIVE

ITERABLES

CodeSnippet

You can download all the code on this page from the code snippets directory

Python allows user-defined classes to support iteration, as described in the Python docs. A class is a so called iterable if it defines a method `__iter__` that returns an iterator instance. An iterable is typically some kind of container or collection (e.g. `list` and `tuple` are iterables).

The iterator instance must have a `next` method that returns the next element in the iteration. In Python an iterable also has to have an `__iter__` method itself that returns `self` instead of a new iterator. It is important to understand that an iterator only manages a single iteration. After this iteration it is spend and cannot be used for a second iteration (it cannot be restarted). An iterable on the other hand can create as many iterators as needed and therefore supports multiple iterations. Even though both iterables and iterators have an `__iter__` method they are semantically very different (duck-typing can be misleading in this case).

In the context of MDP this means that an iterator can only be used for a single training phase, while iterables also support multiple training phases. So if you use a node with multiple training phases and train it in a flow make sure that you provide an iterable for this node (otherwise an exception will be raised). For nodes with a single training phase you can use either an iterable or an iterator.

A convenient implementation of the iterator protocol is provided by generators: see this article for an introduction, and the official **PEP 255** for a complete description.

Let us define two bogus node classes to be used as examples of nodes

```
>>> class BogusNode(mdp.Node):
...     """This node does nothing."""
...     def _train(self, x):
...         pass
>>> class BogusNode2(mdp.Node):
...     """This node does nothing. But it's neither trainable nor invertible.
...     """
...     def is_trainable(self): return False
...     def is_invertible(self): return False
```

This generator generates `blocks` input blocks to be used as training set. In this example one block is a 2-dimensional time series. The first variable is [2,4,6,....,1000] and the second one [0,1,3,5,...,999]. All blocks are equal, this of course would not be the case in a real-life example.

In this example we use a progress bar to get progress information.

```
>>> def gen_data(blocks):
...     for i in mdp.utils.progressinfo(xrange(blocks)):
...         block_x = np.atleast_2d(np.arange(2.,1001,2))
...         block_y = np.atleast_2d(np.arange(1.,1001,2))
...         # put variables on columns and observations on rows
...         block = np.transpose(np.concatenate([block_x,block_y]))
...         yield block
```

The `progressinfo` function is a fully configurable text-mode progress info box tailored to the command-line die-hards. Have a look at its doc-string and prepare to be amazed!

Let's define a bogus flow consisting of 2 `BogusNodes`

```
>>> flow = mdp.Flow([BogusNode(),BogusNode()], verbose=1)
```

Train the first node with 5000 blocks and the second node with 3000 blocks. Note that the only allowed argument to `train` is a sequence (list or tuple) of iterables or iterators. In case you don't want or need to use incremental learning and want to do a one-shot training, you can use as argument to `train` a single array of data.

5.1 Block-mode training

```
>>> flow.train([gen_data(5000),gen_data(3000)])
Training node #0 (BogusNode)
<BLANKLINE>
[==================================100%==================================>]
<BLANKLINE>
Training finished
Training node #1 (BogusNode)
[==================================100%==================================>]
<BLANKLINE>
Training finished
Close the training phase of the last node
```

5.2 One-shot training using one single set of data for both nodes

```
>>> flow = BogusNode() + BogusNode()
>>> block_x = np.atleast_2d(np.arange(2.,1001,2))
>>> block_y = np.atleast_2d(np.arange(1.,1001,2))
>>> single_block = np.transpose(np.concatenate([block_x,block_y]))
>>> flow.train(single_block)
```

If your flow contains non-trainable nodes, you must specify a `None` for the non-trainable nodes

```
>>> flow = mdp.Flow([BogusNode2(),BogusNode()], verbose=1)
>>> flow.train([None, gen_data(5000)])
Training node #0 (BogusNode2)
Training finished
Training node #1 (BogusNode)
[==================================100%==================================>]
<BLANKLINE>
Training finished
Close the training phase of the last node
```

You can use the one-shot training

```
>>> flow = mdp.Flow([BogusNode2(),BogusNode()], verbose=1)
>>> flow.train(single_block)
Training node #0 (BogusNode2)
Training finished
Training node #1 (BogusNode)
Training finished
Close the training phase of the last node
```

Iterators can always be safely used for execution and inversion, since only a single iteration is needed

```
>>> flow = mdp.Flow([BogusNode(),BogusNode()], verbose=1)
>>> flow.train([gen_data(1), gen_data(1)])
Training node #0 (BogusNode)
```

```
Training finished
Training node #1 (BosgusNode)
[==================================100%==================================>]
<BLANKLINE>
Training finished
Close the training phase of the last node
>>> output = flow.execute(gen_data(1000))
[==================================100%==================================>]
>>> output = flow.inverse(gen_data(1000))
[==================================100%==================================>]
```

Execution and inversion can be done in one-shot mode also. Note that since training is finished you are not going to get a warning

```
>>> output = flow(single_block)
>>> output = flow.inverse(single_block)
```

If a node requires multiple training phases (e.g., `GaussianClassifierNode`), `Flow` automatically takes care of using the iterable multiple times. In this case generators (and iterators) are not allowed, since they are spend after yielding the last data block.

However, it is fairly easy to wrap a generator in a simple iterable if you need to

```
>>> class SimpleIterable(object):
...     def __init__(self, blocks):
...         self.blocks = blocks
...     def __iter__(self):
...         # this is a generator
...         for i in range(self.blocks):
...             yield generate_some_data()
```

Note that if you use random numbers within the generator, you usually would like to reset the random number generator to produce the same sequence every time

```
>>> class RandomIterable(object):
...     def __init__(self):
...         self.state = None
...     def __iter__(self):
...         if self.state is None:
...             self.state = np.random.get_state()
...         else:
...             np.random.set_state(self.state)
...         for i in range(2):
...             yield np.random.random((1,4))
>>> iterable = RandomIterable()
>>> for x in iterable:
...     print x
[[ 0.5488135   0.71518937  0.60276338  0.54488318]]
[[ 0.4236548   0.64589411  0.43758721  0.891773  ]]
>>> for x in iterable:
...     print x
[[ 0.5488135   0.71518937  0.60276338  0.54488318]]
[[ 0.4236548   0.64589411  0.43758721  0.891773  ]]
```

CHAPTER

SIX

CHECKPOINTS

CodeSnippet

You can download all the code on this page from the code snippets directory

It can sometimes be useful to execute arbitrary functions at the end of the training or execution phase, for example to save the internal structures of a node for later analysis. This can easily be done by defining a `CheckpointFlow`. As an example imagine the following situation: you want to perform Principal Component Analysis (PCA) on your data to reduce the dimensionality. After this you want to expand the signals into a nonlinear space and then perform Slow Feature Analysis to extract slowly varying signals. As the expansion will increase the number of components, you don't want to run out of memory, but at the same time you want to keep as much information as possible after the dimensionality reduction. You could do that by specifying the percentage of the total input variance that has to be conserved in the dimensionality reduction. As the number of output components of the PCA node now can become as large as the that of the input components, you want to check, after training the PCA node, that this number is below a certain threshold. If this is not the case you want to abort the execution and maybe start again requesting less variance to be kept.

Let start defining a generator to be used through the whole example

```
>>> def gen_data(blocks,dims):
...     mat = np.random.random((dims,dims))-0.5
...     for i in xrange(blocks):
...         # put variables on columns and observations on rows
...         block = mdp.utils.mult(np.random.random((1000,dims)), mat)
...         yield block
```

Define a `PCANode` which reduces dimensionality of the input, a `PolynomialExpansionNode` to expand the signals in the space of polynomials of degree 2 and a `SFANode` to perform SFA

```
>>> pca = mdp.nodes.PCANode(output_dim=0.9)
>>> exp = mdp.nodes.PolynomialExpansionNode(2)
>>> sfa = mdp.nodes.SFANode()
```

As you see we have set the output dimension of the `PCANode` to be `0.9`. This means that we want to keep at least 90% of the variance of the original signal. We define a `PCADimensionExceededException` that has to be thrown when the number of output components exceeds a certain threshold

```
>>> class PCADimensionExceededException(Exception):
...     """Exception base class for PCA exceeded dimensions case."""
...     pass
```

Then, write a `CheckpointFunction` that checks the number of output dimensions of the `PCANode` and aborts if this number is larger than `max_dim`

```
>>> class CheckPCA(mdp.CheckpointFunction):
...     def __init__(self,max_dim):
...         self.max_dim = max_dim
...     def __call__(self,node):
...         node.stop_training()
...         act_dim = node.get_output_dim()
```

```
...         if act_dim > self.max_dim:
...             errstr = 'PCA output dimensions exceeded maximum '+\
...                      '(%d > %d)'%(act_dim,self.max_dim)
...             raise PCADimensionExceededException, errstr
...         else:
...             print 'PCA output dimensions = %d'%(act_dim)
```

Define the CheckpointFlow

```
>>> flow = mdp.CheckpointFlow([pca, exp, sfa])
```

To train it we have to supply 3 generators and 3 checkpoint functions

```
>>> flow.train([gen_data(10, 50), None, gen_data(10, 50)],
...            [CheckPCA(10), None, None])
Traceback (most recent call last):
  File "<stdin>", line 2, in ?
  [...]
__main__.PCADimensionExceededException: PCA output dimensions exceeded maximum (25 > 10)
```

The training fails with a `PCADimensionExceededException`. If we only had 12 input dimensions instead of 50 we would have passed the checkpoint

```
>>> flow[0] = mdp.nodes.PCANode(output_dim=0.9)
>>> flow.train([gen_data(10, 12), None, gen_data(10, 12)],
...            [CheckPCA(10), None, None])
PCA output dimensions = 7
```

We could use the built-in `CheckpoinSaveFunction` to save the `SFANode` and analyze the results later

```
>>> pca = mdp.nodes.PCANode(output_dim=0.9)
>>> exp = mdp.nodes.PolynomialExpansionNode(2)
>>> sfa = mdp.nodes.SFANode()
>>> flow = mdp.CheckpointFlow([pca, exp, sfa])
>>> flow.train([gen_data(10, 12), None, gen_data(10, 12)],
...            [CheckPCA(10),
...             None,
...             mdp.CheckpointSaveFunction('dummy.pic',
...                                        stop_training = 1,
...                                        protocol = 0)])
PCA output dimensions = 6
```

We can now reload and analyze the `SFANode`

```
>>> fl = file('dummy.pic')
>>> import cPickle
>>> sfa_reloaded = cPickle.load(fl)
>>> sfa_reloaded
SFANode(input_dim=27, output_dim=27, dtype='float64')
```

Don't forget to clean the rubbish

```
>>> fl.close()
>>> import os
>>> os.remove('dummy.pic')
```

CHAPTER

SEVEN

NODE EXTENSIONS

CodeSnippet

You can download all the code on this page from the code snippets directory

The node extension mechanism is an advanced topic, so you might want to skip this section at first. The examples here partly use the `parallel` and `hinet` packages, which are explained later in the tutorial.

The node extension mechanism makes it possible to dynamically add methods or class attributes for specific features to node classes (e.g. for parallelization the nodes need a `_fork` and `_join` method). Note that methods are just a special case of class attributes, the extension mechanism treats them like any other class attributes. It is also possible for users to define custom extensions to introduce new functionality for MDP nodes without having to directly modify any MDP code. The node extension mechanism basically enables some form of *Aspect-oriented programming* (AOP) to deal with *cross-cutting concerns* (i.e., you want to add a new aspect to node classes which are spread all over MDP and possibly your own code). In the AOP terminology any new methods you introduce contain *advice* and the *pointcut* is effectively defined by the calling of these methods.

Without the extension mechanism the adding of new aspects to nodes would be done through inheritance, deriving new node classes that implement the aspect for the parent node class. This is fine unless one wants to use multiple aspects, requiring multiple inheritance for every combination of aspects one wants to use. Therefore this approach does not scale well with the number of aspects.

The node extension mechanism does not directly depend on inheritance, instead it adds the methods or class attributes to the node classes dynamically at runtime (like *method injection*). This makes it possible to activate extensions just when they are needed, reducing the risk of interference between different extensions. One can also use multiple extensions at the same time, as long as there is no interference, i.e., both extensions do not use any attributes with the same name.

The node extension mechanism uses a special Metaclass, which allows it to define the node extensions as classes derived from nodes (bascially just what one would do without the extension mechanism). This keeps the code readable and avoids some problems when using automatic code checkers (like the background pylint checks in the Eclipse IDE with PyDev).

In MDP the node extension mechanism is currently used by the `parallel` package and for the the HTML representation in the `hinet` package, so the best way to learn more is to look there. We also use these packages in the following examples.

7.1 Using Extensions

First of all you can get all the available node extensions by calling the `get_extensions` function, or to get just a list of their names use `get_extensions().keys()`. Be careful not to modify the dict returned by `get_extensions`, since this will actually modify the registered extensions. The currently activated extensions are returned by `get_active_extensions`. To activate an extension use `activate_extension`, e.g. to activate the parallel extension write:

```
>>> mdp.activate_extension("parallel")
>>> # now you can use the added attributes / methods
```

```
>>> mdp.deactivate_extension("parallel")
>>> # the additional attributes are no longer available
```

Note: As a user you will never have to activate the parallel extension yourself, this is done automatically by the `ParallelFlow` class. The parallel package will be explained later, it is used here only as an example.

Activating an extension adds the available extensions attributes to the supported nodes. MDP also provides a context manager for the `with` statement:

```
>>> with mdp.extension("parallel"):
...     pass
```

The `with` statement ensures that the activated extension is deactivated after the code block, even if there is an exception. But the deactivation at the end happens only for the extensions that were activated by this context manager (not for those that were already active when the context was entered). This prevents unintended side effects.

Finally there is also a function decorator:

```
>>> @mdp.with_extension("parallel")
... def f():
...     pass
```

Again this ensures that the extension is deactivated after the function call, even in the case of an exception. The deactivation happens only if the extension was activated by the decorator (not if it was already active before).

7.2 Writing Extension Nodes

Suppose you have written your own nodes and would like to make them compatible with a particular extension (e.g. add the required methods). The first way to do this is by using multiple inheritance to derive from the base class of this extension and your custom node class. For example the parallel extension of the SFA node is defined in a class

```
>>> class ParallelSFANode(mdp.parallel.ParallelExtensionNode,
...                       mdp.nodes.SFANode):
...     def _fork(self):
...         # implement the forking for SFANode
...         pass
...     def _join(self):
...         # implement the joining for SFANode
...         pass
```

Here `ParallelExtensionNode` is the base class of the extension. Then you define the required methods or attributes just like in a normal class. If you want you could even use the new `ParallelSFANode` class like a normal class, ignoring the extension mechanism. Note that your extension node is automatically registered in the extension mechanism (through a little metaclass magic).

For methods you can alternatively use the `extension_method` function decorator. You define the extension method like a normal function, but add the function decorator on top. For example to define the `_fork` method for the `SFANode` we could have also used

```
>>> @mdp.extension_method("parallel", mdp.nodes.SFANode)
... def _fork(self):
...     pass
```

The first decorator argument is the name of the extension, the second is the class you want to extend. You can also specify the method name as a third argument, then the name of the function is ignored (this allows you to get rid of warnings about multiple functions with the same name).

7.3 Creating Extensions

To create a new node extension you just have to create a new extension base class. For example the HTML representation extension in `mdp.hinet` is created with

```
>>> class  HTMLExtensionNode(mdp.ExtensionNode, mdp.Node):
...     """Extension node for HTML representations of individual nodes."""
...     extension_name = "html2"
...     def html_representation(self):
...         pass
...     def _html_representation(self):
...         pass
```

Note that you must derive from `ExtensionNode`. If you also derive from `mdp.Node` then the methods (and attributes) in this class are the default implementation for the `mdp.Node` class. So they will be used by all nodes without a more specific implementation. If you do not derive from `mdp.Node` then there is no such default implementation. You can also derive from a more specific node class if your extension only applies to these specific nodes.

When you define a new extension then you must define the `extension_name` attribute. This magic attribute is used to register the new extension and you can activate or deactivate the extension by using this name.

Note that extensions can override attributes and methods that are defined in a node class. The original attributes can still be accessed by prefixing the name with `_non_extension_` (the prefix string is also available as `mdp.ORIGINAL_ATTR_PREFIX`). On the other hand one extension is not allowed to override attributes that were defined by another currently active extension.

The extension mechanism uses some magic to make the behavior more intuitive with respect to inheritance. Basically methods or attributes defined by extensions shadow those which are not defined in the extension. Here is an example

```
>>> class TestExtensionNode(mdp.ExtensionNode):
...     extension_name = "test"
...     def _execute(self):
...         return 0
>>> class TestNode(mdp.Node):
...     def _execute(self):
...         return 1
>>> class ExtendedTestNode(TestExtensionNode, TestNode):
...     pass
```

After this extension is activated any calls of `_execute` in instances of `TestNode` will return 0 instead of 1. The `_execute` from the extension base-class shadows the method from `TestNode`. This makes it easier to share behavior for different classes. Without this magic one would have to explicitly override `_execute` in `ExtendedTestNode` (or derive the extension base-class from `Node`, but that would give this behavior to all node classes). Note that there is a `verbose` argument in `activate_extension` which can help with debugging.

CHAPTER

EIGHT

HIERARCHICAL NETWORKS

CodeSnippet

You can download all the code on this page from the code snippets directory

The `hinet` subpackage makes it possible to construct arbitrary feed-forward architectures, and in particular hierarchical networks (networks which are organized in layers).

8.1 Building blocks

The `hinet` package contains three basic building blocks, all of which are derived from the `Node` class: `Layer`, `FlowNode`, and `Switchboard`.

The first building block is the `Layer` node, which works like a horizontal version of flow. It acts as a wrapper for a set of nodes that are trained and executed in parallel. For example, we can combine two nodes with 100 dimensional input to construct a layer with a 200-dimensional input:

```
>>> node1 = mdp.nodes.PCANode(input_dim=100, output_dim=10)
>>> node2 = mdp.nodes.SFANode(input_dim=100, output_dim=20)
>>> layer = mdp.hinet.Layer([node1, node2])
>>> layer
Layer(input_dim=200, output_dim=30, dtype=None)
```

The first half of the 200 dimensional input data is then automatically assigned to `node1` and the second half to `node2`. A layer `Layer` node can be trained and executed just like any other node. Note that the dimensions of the nodes must be already set when the layer is constructed.

In order to be able to build arbitrary feed-forward node structures, `hinet` provides a wrapper class for flows (i.e., vertical stacks of nodes) called `FlowNode`. For example, we can replace `node1` in the above example with a `FlowNode`:

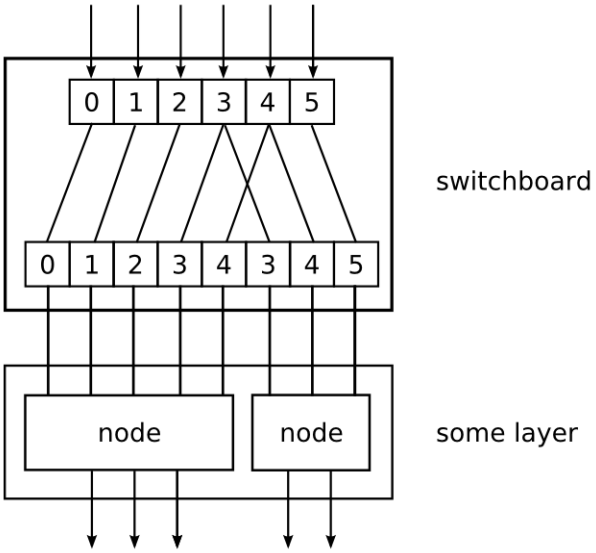
```
>>> node1_1 = mdp.nodes.PCANode(input_dim=100, output_dim=50)
>>> node1_2 = mdp.nodes.SFANode(input_dim=50, output_dim=10)
>>> node1_flow = mdp.Flow([node1_1, node1_2])
>>> node1 = mdp.hinet.FlowNode(node1_flow)
>>> layer = mdp.hinet.Layer([node1, node2])
>>> layer
Layer(input_dim=200, output_dim=30, dtype=None)
```

In this example `node1` has two training phases (one for each internal node). Therefore `layer` now has two training phases as well and behaves like any other node with two training phases. By combining and nesting `FlowNode` and `Layer`, it is thus possible to build modular node structures. Note that while the `Flow` interface looks pretty similar to that of `Node` it is not compatible and therefore we must use `FlowNode` as an adapter.

When implementing networks one might have to route different parts of the data to different nodes in a layer. This functionality is provided by the `Switchboard` node. A basic `Switchboard` is initialized with a 1-D Array with one entry for each output connection, containing the corresponding index of the input connection that it receives its input from, e.g.:

```
>>> switchboard = mdp.hinet.Switchboard(input_dim=6, connections=[0,1,2,3,4,3,4,5])
>>> switchboard
Switchboard(input_dim=6, output_dim=8, dtype=None)
>>> x = mdp.numx.array([[2,4,6,8,10,12]])
>>> switchboard.execute(x)
array([[ 2,  4,  6,  8, 10,  8, 10, 12]])
```

The switchboard can then be followed by a layer that splits the routed input to the appropriate nodes, as illustrated in following picture:



By combining layers with switchboards one can realize any feed-forward network topology. Defining the switchboard routing manually can be quite tedious. One way to automatize this is by defining switchboard subclasses for special routing situations. The `Rectangular2dSwitchboard` class is one such example and will be briefly described in a later example.

8.2 HTML representation

Since hierarchical networks can be quite complicated, `hinet` includes the class `HiNetHTMLTranslator` that translates an MDP flow into a graphical visualization in an HTML file. We also provide the helper function `show_flow` which creates a complete HTML file with the flow visualization in it and opens it in your standard browser.

```
>>> mdp.hinet.show_flow(flow)
```

To integrate the HTML representation into your own custom HTML file you can take a look at `show_flow` to learn the usage of `HiNetHTMLTranslator`. You can also specify custom translations for node types via the extension mechanism (e.g to define which parameters are displayed).

8.3 Example application (2-D image data)

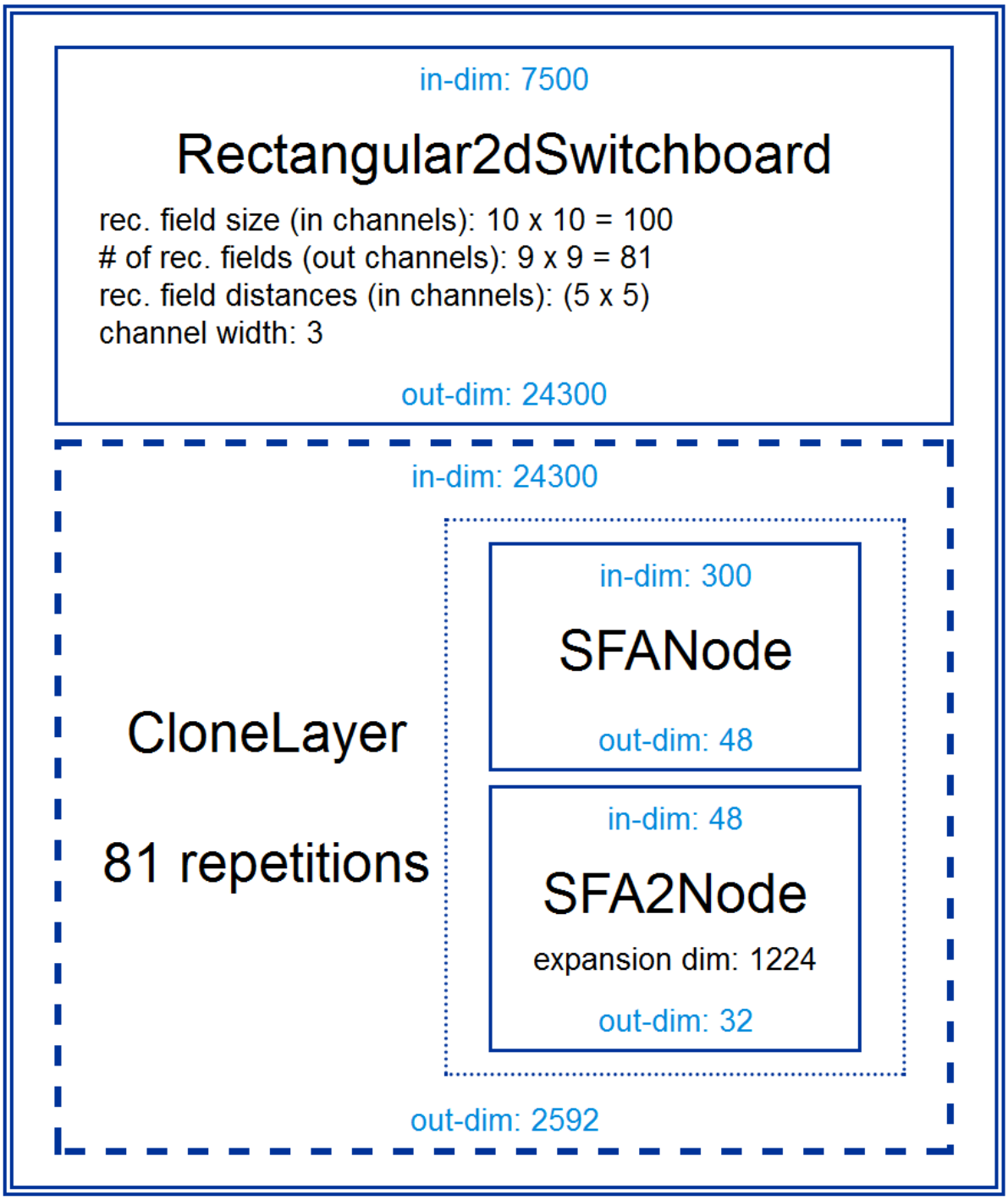
As promised we now present a more complicated example. We define the lowest layer for some kind of image processing system. The input data is assumed to consist of image sequences, with each image having a size of 50 by 50 pixels. We take color images, so after converting the images to one dimensional numpy arrays each pixel corresponds to three numeric values in the array, which the values just next to each other (one for each color channel).

The processing layer consists of many parallel units, which only see a small image region with a size of 10 by 10 pixels. These so called receptive fields cover the whole image and have an overlap of five pixels. Note

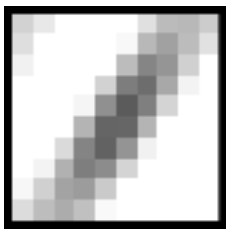
that the image data is represented as an 1-D array. Therefore we need the `Rectangular2dSwitchboard` class to correctly route the data for each receptive field to the corresponding unit in the following layer. We also call the switchboard output for a single receptive field an output channel and the three RGB values for a single pixel form an input channel. Each processing unit is a flow consisting of an `SFANode` (to somewhat reduce the dimensionality) that is followed by an `SFA2Node`. Since we assume that the statistics are similar in each receptive filed we actually use the same nodes for each receptive field. Therefore we use a `CloneLayer` instead of the standard `Layer`. Here is the actual code:

```
>>> switchboard = mdp.hinet.Rectangular2dSwitchboard(in_channels_xy=(50, 50),
...                                                  field_channels_xy=(10, 10),
...                                                  field_spacing_xy=(5, 5),
...                                                  in_channel_dim=3)
>>> sfa_dim = 48
>>> sfa_node = mdp.nodes.SFANode(input_dim=switchboard.out_channel_dim,
...                              output_dim=sfa_dim)
>>> sfa2_dim = 32
>>> sfa2_node = mdp.nodes.SFA2Node(input_dim=sfa_dim,
...                                output_dim=sfa2_dim)
>>> flownode = mdp.hinet.FlowNode(mdp.Flow([sfa_node, sfa2_node]))
>>> sfa_layer = mdp.hinet.CloneLayer(flownode,
...                                  n_nodes=switchboard.output_channels)
>>> flow = mdp.Flow([switchboard, sfa_layer])
```

The HTML representation of the the constructed flow looks like this in your browser:



Now one can train this flow for example with image sequences from a movie. After the training phase one can compute the image pattern that produces the highest response in a given output coordinate (use `mdp.utils.QuadraticForm`). One such optimal image pattern may look like this (only a grayscale version is shown):



So the network units have developed some kind of primitive line detector. More on this topic can be found in: Berkes, P. and Wiskott, L., *Slow feature analysis yields a rich repertoire of complex cell properties*. Journal of Vision, 5(6):579-602.

One could also add more layers on top of this first layer to do more complicated stuff. Note that the `in_channel_dim` in the next `Rectangular2dSwitchboard` would be 32, since this is the output dimension of one unit in the `CloneLayer` (instead of 3 in the first switchboard, corresponding to the three RGB colors).

CHAPTER

NINE

PARALLELIZATION

CodeSnippet

You can download all the code on this page from the code snippets directory

The `parallel` package adds the ability to parallelize the training and execution of MPD flows. This package is split into two decoupled parts.

The first part consists of a parallel extension for the familiar MDP structures of nodes and flows. In principle all MDP nodes aldready support parallel execution, since copies of a node can be made and used in parallel. Parallelization of the training on the other hand depends on the specific node or algorithm. For nodes which can be trained in a parallelized way there is the extension class `ParallelExtensionNode`. It adds the `fork` and `join` methods. When providing a parallel extension for custom node classes you should implement `_fork` and `_join`. Secondly there is the `ParallelFlow` class, which internally splits the training or execution into tasks which are then processed in parallel.

The second part consists of the schedulers. A scheduler takes tasks and processes them in a more or less parallel way (e.g. in multiple Python processes). A scheduler deals with the more technical aspects of the parallelization, but does not need to know anything about nodes and flows.

9.1 Basic Examples

In the following example we parallelize a simple `Flow` consisting of PCA and quadratic SFA, so that it makes use of multiple cores on a modern CPU:

```
>>> node1 = mdp.nodes.PCANode(input_dim=100, output_dim=10)
>>> node2 = mdp.nodes.SFA2Node(input_dim=10, output_dim=10)
>>> parallel_flow = mdp.parallel.ParallelFlow([node1, node2])
>>> parallel_flow2 = parallel_flow.copy()
>>> parallel_flow3 = parallel_flow.copy()
>>> n_data_chunks = 10
>>> data_iterables = [[np.random.random((50, 100))
...                    for _ in range(n_data_chunks)]] * 2
>>> scheduler = mdp.parallel.ProcessScheduler()
>>> parallel_flow.train(data_iterables, scheduler=scheduler)
>>> scheduler.shutdown()
```

Only two additional lines were needed to parallelize the training of the flow. All one has to do is use a `ParallelFlow` instead of the normal `Flow` and provide a scheduler. The `ProcessScheduler` will automatically create as many Python processes as there are CPU cores. The parallel flow gives the training task for each data chunk over to the scheduler, which in turn then distributes them across the available worker processes. The results are then returned to the flow, which puts them together in the right way. Note that the `shutdown` method should be always called at the end to make sure that the recources used by the scheduler are cleaned up properly. One should therefore put the `shutdown` call into a safe try/finally statement

```
>>> scheduler = mdp.parallel.ProcessScheduler()
>>> try:
```

```
...     parallel_flow2.train(data_iterables, scheduler=scheduler)
... finally:
...     scheduler.shutdown()
```

The `Scheduler` class also supports the context manager interface of Python. One can therefore use a `with` statement

```
>>> with mdp.parallel.ProcessScheduler() as scheduler:
...     parallel_flow3.train(data_iterables, scheduler=scheduler)
```

The `with` statement ensures that `scheduler.shutdown` is automatically called (even if there is an exception).

9.2 Scheduler

The scheduler classes in MDP are derived from the `Scheduler` base class (which itself does not implement any parallelization). The standard choice at the moment is the `ProcessScheduler`, which distributes the incoming tasks over multiple Python processes (circumventing the global interpreter lock or GIL). The performance gain is highly dependent on the specific situation, but can potentially scale well with the number of CPU cores (in one real world case we saw a speed-up factor of 4.2 on an Intel Core i7 processor with 4 physical / 8 logical cores).

MDP has experimental support for the Parallel Python library in the `mdp.parallel.pp_support` package. In principle this makes it possible to parallelize across multiple machines. Recently we also added the thread based scheduler `ThreadScheduler`. While it is limited by the GIL it can still achieve a real-world speedup (since NumPy releases the GIL when possible) and it causes less overhead compared to the `ProcessScheduler`.

(The following information is only releveant for people who want to implement custom scheduler classes.)

The first important method of the scheduler class is `add_task`. This method takes two arguments: `data` and `task_callable`, which can be a function or an object with a `__call__` method. The return value of the `task_callable` is the result of the task. If `task_callable` is `None` then the last provided `task_callable` will be used. This splitting into callable and data makes it possible to implement caching of the `task_callable` in the scheduler and its workers (caching is turned on by default in the `ProcessScheduler`). To further influence caching one can derive from the `TaskCallable` class, which has a `fork` method to generate new callables in order to preserve the original cached callable. For MDP training and execution there are corresponding classes derived from `TaskCallable` which are automatically used, so normally there is no need to worry about this.

After submitting all the tasks with `add_task` you can then call the `get_results` method. This method returns all the task results, normally in a list. If there are open tasks in the scheduler then `get_results` will wait until all the tasks are finished (it blocks). You can also check the status of the scheduler by looking at the `n_open_tasks` property, which gives you the number of open tasks. After using the scheduler you should always call the `shutdown` method, otherwise you might get error messages from not properly closed processes.

Internally an instance of the base class `mdp.parallel.ResultContainer` is used for the storage of the results in the scheduler. By providing your own result container to the scheduler you modify the storage. For example the default result container is an instance of `OrderedResultContainer`. The `ParallelFlow` class by default makes sure that the right container is used for the task (this can be overriden manually via the `overwrite_result_container` parameter of the `train` and `execute` methods).

9.3 Parallel Nodes

If you want to parallelize your own nodes you have to provide parallel extensions for them. The `ParallelExtensionNode` base class has the new template methods `fork` and `join`. `fork` should return a new node instance. This new instance can then be trained somewhere else (e.g. in a different process) with the usual `train` method. Afterwards `join` is called on the original node, with the forked node as the argument. This should be equivalent to calling `train` directly on the original node.

During Execution nodes are not forked by default, instead they are just copied (for example they are pickled and send to the Python worker processes). It is possible for nodes during execution to explicitly request that they are

forked and joined (like during training). This is done by overriding the `use_execute_fork` method, which by default returns `False`. For example nodes that record data during execution can use this feature to become compatible with parallelization.

When writing custom parallel node extension you should only overwrite the `_fork` and `_join` methods, which are automatically called by `fork` and `join`. The `fork` and `join` take care of the standard node attributes like the dimensions. You should also look at the source code of a parallel node like `ParallelPCANode` to get a better idea of how to parallelize nodes. By overwriting `use_execute_fork` to return `True` you can force forking and joining during execution. Note that the same `_fork` and `_join` implementation is called as during training, so if necessary one should add an `node.is_training()` check there to determine the correct action.

Currently we provide the following parallel nodes: `ParallelPCANode`, `ParallelWhiteningNode`, `ParallelSFANode`, `ParallelSFA2Node`, `ParallelFDANode`, `ParallelHistogramNode`, `ParallelAdaptiveCutoffNode`, `ParallelFlowNode`, `ParallelLayer`, `ParallelCloneLayer` (the last three are derived from the `hinet` package).

CHAPTER

TEN

CACHING EXECUTION RESULTS

CodeSnippet

You can download all the code on this page from the code snippets directory

10.1 Introduction

It is relatively common for nodes to process the same data several times. Usually this happens when training a long sequence of nodes using a fixed data set: to train the nodes at end of the sequence, the data has to be processed by all the preceding ones. This duplication of efforts may be costly, for example in image processing, when one needs to repeatedly filter the images (*as in this example*).

MDP offers a *node extension* that automatically caches the result of the `execute` method, which can boost the speed of an application considerably in such scenarios. The cache can be activated globally (i.e., for all node instances), for some node classes only, or for specific instances.

The caching mechanism is based on the library joblib, version 0.4.3 or higher.

10.2 Activating the caching extension

It is possible to activate the caching extension as for regular extension using the extension name `'cache_execute'`. By default, the cached results will be stored in a database created in a temporary directory for the duration of the Python session. To change the caching directory, which may be useful to create a permanent cache over multiple sessions, one can call the function `mdp.caching.set_cachedir`.

We will illustrate the caching extension using a simple but relatively large Principal Component Analysis problem:

```
>>> # set up a relatively large PCA run
>>> import mdp
>>> import numpy as np
>>> from timeit import Timer
>>> x = np.random.rand(3000,1000)
>>> # create a PCANode and train it using the random data in 'x'
>>> pca_node = mdp.nodes.PCANode()
>>> pca_node.train(x)
>>> pca_node.stop_training()
```

The time for projecting the data `x` on the principal components drops dramatically after the caching extension is activated:

```
>>> # we will use this timer to measure the speed of 'pca_node.execute'
>>> timer = Timer("pca_node.execute(x)", "from __main__ import pca_node, x")
>>> mdp.caching.set_cachedir("/tmp/my_cache")
>>> mdp.activate_extension("cache_execute")
>>> # all calls to the 'execute' method will now be cached in 'my_cache'
>>> # the first time execute is called, the method is run
```

```
>>> # and the result is cached
>>> print timer.repeat(1, 1)[0], 'sec'
1.188946008682251 sec
>>> # the second time, the result is retrieved from the cache
>>> print timer.repeat(1, 1)[0], 'sec'
0.112375974655 sec
>>> mdp.deactivate_extension("cache_execute")
>>> # when the cache extension is deactivated, the 'execute' method is
>>> # called as usual
>>> print timer.repeat(1, 1)[0], 'sec'
0.801102161407 sec
```

Alternative ways to activate the caching extension, which also expose more functionalities, can be found in the `mdp.caching` module. The functions `activate_caching` and `deactivate_caching` allow activating the cache only on certain Node classes, or specific instances. For example, the following line starts the cache extension, caching only instances of the classes `SFANode` and `FDANode`, and the instance `pca_node`.

```
>>> mdp.caching.activate_caching(cachedir='/tmp/my_cache',
...     cache_classes=[mdp.nodes.SFANode, mdp.nodes.FDANode],
...     cache_instances=[pca_node])
>>> # all calls to the 'execute' method of instances of 'SFANode' and
>>> # 'FDANode', and of 'pca_node' will now be cached in 'my_cache'
>>> mdp.caching.deactivate_caching()
```

Make sure to call the `deactivate_caching` method before the end of the session, or the cache directory may remain in a broken state.

Finally, the module `mdp.caching` also defines a context manager that closes the cache properly at the end of the block:

```
>>> with mdp.caching.cache(cachedir='/tmp/my_cache', cache_instances=[pca_node]):
...     # in the block, the cache is active
...     print timer.repeat(1, 1)[0], 'sec'
...
0.101263999939 sec
>>> # at the end of the block, the cache is deactivated
>>> print timer.repeat(1, 1)[0], 'sec'
0.801436901093 sec
```

CHAPTER

ELEVEN

CLASSIFIER NODES

CodeSnippet

You can download all the code on this page from the code snippets directory

New in MDP 2.6 is the `ClassifierNode` base class which offers a simple interface for creating classification tasks. Usually, one does not want to use the classification output in a flow but extract this information independently. By default classification nodes will therefore simply return the identity function on `execute`; all classification work is done with the new methods `label`, `prob` and `rank`. However, if a classification node is the last node in a flow then it is possible to perform the classification as part of the normal flow execution by setting the `execute_method` attribute (more on this later).

As a first example, we will use the `GaussianClassifier`.

```
>>> gc = mdp.nodes.GaussianClassifier()
>>> gc.train(np.random.random((50, 3)), +1)
>>> gc.train(np.random.random((50, 3)) - 0.8, -1)
```

We have trained the node and assigned the labels +1 and -1 to the sample points. Note that in this simple case we do not need to give a label to each individual point, when only a single label is given, it is assigned to the whole batch of features. However, it is also possible to use the more explicit form:

```
>>> gc.train(np.random.random((50, 3)), [+1] * 50)
```

We can then retrieve the most probable labels for some testing data,

```
>>> test_data = np.array([[0.1, 0.2, 0.1], [-0.1, -0.2, -0.1]])
>>> gc.label(test_data)
[1, -1]
```

and also get the probability for each label.

```
>>> prob = gc.prob(test_data)
>>> print prob[0][-1], prob[0][+1]
0.188737388144 0.811262611856
>>> print prob[1][-1], prob[1][+1]
0.992454101588 0.00754589841187
```

Finally, it is possible to get the ranking of the labels, starting with the likeliest.

```
>>> gc.rank(test_data)
[[1, -1], [-1, 1]]
```

New nodes should inherit from `ClassifierNode` and implement the `_label` and `_prob` methods. The public `rank` method will be created automatically from `prob`.

As mentioned earlier it is possible to perform the classification in via the `execute` method of a classifier node. Every classifier node has an `execute_method` attribite which can be set to the string values `"label"`, `"rank"`, or `"prob"`. The `execute` method of the node will then automatically call the indicated classification method and return the result. This is especially useful when the classification node is the last node in a flow,

because then the normal flow execution can be used to get the classification results. An example application is given in the MNSIT handwritten digits classification example.

The `execute_method` attribute can be also set when the node is created via the `execute_method` argument of the `__init__` method.

CHAPTER

TWELVE

INTERFACING WITH OTHER LIBRARIES

CodeSnippet

You can download all the code on this page from the code snippets directory

MDP is, of course, not the only Python library to offer an implementation of signal processing and machine learning methods. Several other projects, often specialized in different algorithms, or based on different approaches, are being developed in parallel. In order to avoid an excessive duplication of efforts, the long-term philosophy of MDP is that of automatically wrapping the algorithms defined in external libraries, if these are installed. In this way, MDP users have access to a larger number of algorithms, and at the same time,we offer the MDP infrastructure (flows, caching, etc.) to users of the wrapped libraries.

At present, MDP automatically creates wrapper nodes for the following libraries if they are installed:

- **Shogun** (http://www.shogun-toolbox.org/): The Shogun machine learning toolbox provides a large set of different support vector machine implementations and classifiers. Each of them can be combined with another large set of kernels.

  The MDP wrapper simplifies setting the parameters for the kernels and classifiers, and provides reasonable default values. In order to avoid conflicts, users are encouraged to keep an eye on the original C++ API and provide as many parameters as specified.

- **libsvm** (http://www.csie.ntu.edu.tw/~cjlin/libsvm/): libsvm is a library for support vector machines. Even though there is also a libsvm wrapper in the Shogun toolbox, the direct libsvm interface is simpler to use and it provides estimates of the probability of different labels.

  Note that starting with MDP 3.0 we only support the Python API for the recent libsvm versions 2.91 and 3.0.

- **scikits.learn** (http://scikit-learn.sourceforge.net/index.html): scikits.learn is a collection of efficient machine learning algorithms. We offer automatic wrappers to all algorithms defined by in the library scikits.learn, and there are a lot of them! The wrapped algorithms can be recognised as their name end with `ScikitsLearnNode`.

  All `ScikitsLearnNode` contain an instance of the wrapped scikits.learn instance in the attribute `scikits_alg`, and allow setting all the parameters using the original keywords. You can see the scikits.learn wrapper in action in this *example application* that uses scikits.learn to perform handwritten digits recognition.

  As of MDP 3.0, the wrappers must be considered experimental, because there are still a few inconsistencies in the scikits.learn interface that we need to address.

CHAPTER

THIRTEEN

BIMDP

CodeSnippet

You can download all the code on this page from the code snippets directory

BiMDP defines a framework for more general flow sequences, involving top-down processes (e.g. for error back-propagation) and loops. So the *bi* in BiMDP primarily stands for *bidirectional*. It also adds a couple of other features, like a standardized way to transport additional data, and a HTML based flow inspection utility. Because BiMDP is a rather large addition and changes a few things compared to standard MDP it is not included in `mdp` but must be imported separately as `bimdp` (BiMDP is included in the standard MDP installation)

```
>>> import bimdp
```

Warning: BiMDP is a relatively new addition to MDP (it was added in MDP 2.6). Even though it already went through long testing and several refactoring rounds it is still not as mature and polished as the rest of MDP. The API of BiMDP should be stable now, we don't expect any significant breakages in the future.

Here is a brief summary of the most important new features in BiMDP:

- Nodes can specify other nodes as jump targets, where the execution or training will be continued. It is now possible to use loops or backpropagation, in contrast to the strictly linear execution of a normal MDP flow. This is enabled by the new `BiFlow` class. The new `BiNode` base class adds a `node_id` string attribute, which can be used to target a node.

  The complexities of arbitrary data flows are evenly split up between `BiNode` and `BiFlow`: Nodes specify their data and target using a standardized interface, which is then interpreted by the flow (somewhat like a very primitive domain specific language). The alternative approach would have been to use specialized flow classes or container nodes for each use case, which ultimately comes down to a design decision. Of course you can (and should) still take that route if for some reason BiMDP is not an adequate solution for your problem.

- In addition to the standard array data, nodes can transport more data in a message dictionary (these are really just standard Python dictionaries, so they are `dict` instances). The new `BiNode` base class provides functionality to make this as convenient as possible.

- An interactive HTML-based inspection for flow training and execution is available. This allows you to step through your flow for debugging or add custom visualizations to analyze what is going on.

- BiMDP supports and extends the `hinet` and the `parallel` packages from MDP. BiMDP in general is compatible with MDP, so you can use standard MDP nodes in a `BiFlow`. You can also use `BiNode` instances in a standard MDP flow, as long as you don't use certain BiMDP features.

The structure of BiMDP closely follows that of MDP, so there are submodules `bimdp.nodes`, `bimdp.parallel`, and `bimdp.hinet`. The module `bimdp.nodes` contains `BiNode` versions of nearly all MDP nodes. For example `bimdp.nodes.PCABiNode` is derived from both `BiNode` and `mdp.nodes.PCANode`.

There are several examples available in the `mdp-examples` repository, which demonstrate how BiMDP can be used. For example `backpropagation` demonstrates how to implement a simple multilayer perceptron, using

backpropagation for learning. The example `binetdbn` is a proof-of-concept implementation of a deep belief network. In addition there are a couple of smaller examples in `bimdp_examples`.

Finally note that this tutorial is intended to serve as an introduction, covering all the basic aspects of BiMDP. For more detailed specifications have a look at the docstrings.

13.1 Targets, id's and Messages

In a normal MDP node the return value of the `execute` method is restricted to a single 2d array. A BiMDP `BiNode` on the other hand can optionally return a tuple containing an additional message dictionary and a target value. So in general the return value is a tuple `(x, msg, target)`, where `x` is a the usual 2d array. Alternatively a `BiNode` is also allowed to return only the array `x` or a 2-tuple `(x, msg)` (specifying no target value). Unless stated otherwise the last entry in the tuple should not be `None`, but all the other values are allowed to be `None` (so if you specify a target then `msg` can be `None`, and even `x` can be `None`).

The `msg` message is a normal Python dictionary. You can use it to transport any data that does not fit into the `x` 2d data array. Nodes can take data from to the message and add data to it. The message is propagated along with the `x` data. If a normal MDP node is contained in a `BiFlow` then the message is simply passed around it. A `BiNode` can freely decide how to interact with the message (see the BiNode section for more information).

The target value is either a string or a number. The number is the relative position of the target node in the flow, so a target value of 1 corresponds to the following node, while -1 is the previous node. The `BiNode` base class also allows the specification of a `node_id` string in the `__init__` method. This string can then be used as a target value.

The `node_id` string is also useful to access nodes in a `BiFlow` instance. The standard MDP `Flow` class already implements standard Python container methods, so `flow[2]` will return the third node in the flow. `BiFlow` in addition enables you to use the `node_id` to index nodes in the flow, just like for a dictionary. Here is a simple example

```
>>> pca_node = bimdp.nodes.PCABiNode(node_id="pca")
>>> biflow = bimdp.BiFlow([pca_node])
>>> biflow["pca"]
PCABiNode(input_dim=None, output_dim=None, dtype=None, node_id="pca")
```

13.2 BiFlow

The `BiFlow` class mostly works in the same way as the normal `Flow` class. We already mentioned several of the new features, like support for targets, messages, and retrieving nodes based on their `node_id`. Apart from that the only major difference is the way in which you can provide additional arguments for nodes. For example the `FDANode` in MDP requires class labels in addition to the data array (telling the node to which class each data point belongs). In the `Flow` class the additional training data (the class labels) is provided by the same iterable as the data. In a `BiFlow` this is no longer allowed, since this functionality is provided by the more general message mechanism. In addition to the `data_iterables` keyword argument of `train` there is a new `msg_iterables` argument, to provide iterables for the message dictionary. The structure of the `msg_iterables` argument must be the same as that of `data_iterables`, but instead of yielding arrays it should yield dictionaries (containing the additional data values with the corresponding keys). Here is an example

```
>>> samples = np.random.random((100,10))
>>> labels = np.arange(100)
>>> biflow = bimdp.BiFlow([mdp.nodes.PCANode(), bimdp.nodes.FDABiNode()])
>>> biflow.train([[samples],[samples]], msg_iterables=[None,[{"labels": labels}]])
```

The `_train` method of `FDANode` requires the `labels` argument, so this is used as the key value. Note that we have to use the `BiNode` version of `FDANode`, called `FDABiNode` (alomost every MDP node has a `BiNode` version following this naming convention). The `BiNode` class provides the `cl` value from the message to the `_train` method.

In a normal `Flow` the additional arguments can only be given to the node which is currently in training. This limitation does not apply to a `BiFlow`, where the message can be accessed by all nodes (more on this later). Message iterators can also be used during execution, via the `msg_iterable` argument in `BiFlow.execute`. Of course messages can be also returned by `BiFlow.execute`, so the return value always has the form `(y, msg)` (where `msg` can be an empty dictionary). For example:

```
>>> biflow = bimdp.nodes.PCABiNode(output_dim=10) + bimdp.nodes.SFABiNode()
>>> x = np.random.random((100,20))
>>> biflow.train(x)
>>> y, msg = biflow.execute(x)
>>> msg
{}
>>> # include a message that is not used
>>> y, msg = biflow.execute(x, msg_iterable={"test": 1})
>>> msg
{'test': 1}
```

Note that `BiNode` overloads the plus operator to create a `BiFlow`. If iterables are used for execution then the `BiFlow` not only concatenates the `y` result arrays, but also tries to join the `msg` dictionaries into a single one. Arrays in the `msg` will be concatenated, for all other types the plus operator is used.

The `train` method of `BiFlow` also has an additional argument called `stop_messages`, which can be used to provide message iterables for `stop_training`. The `execute` method on the other hand has an argument `target_iterable`, which can be used to specify the initial target in the flow execution (if the `iterable` is just a single array then of course the `target_iterable` should be just a single `node_id`).

13.3 BiNode

We now want to give an overview of the `BiNode` API, which is mostly an extension of the `Node` API. First we take a look at the possible return values of a `BiNode` and briefly explain their meaning:

- **`execute`**
  - `x` or `(x, msg)` or `(x, msg, target)`. Normal execution continues, directly jumping to the target if one is specified.
- **`train`**
  - `None` terminates training.
  - `x` or `(x, msg)` or `(x, msg, target)`. Means that execution is continued and that this node will be reached again to terminate training. If `x` is `None` and no target is specified then the remaining `msg` is dropped (so it is not required to "clear" the message manually in `_train` for custom nodes to terminate training).
- **`stop_training`**
  - `None` doesn't do anything, like the normal MDP `stop_training`.
  - `x` or `(x, msg)` or `(x, msg, target)`. Causes an execute like phase, which terminates when the end of the flow is reached or when `EXIT_TARGET` is given as target value (just like during a normal execute phase, `EXIT_TARGET` is explained later).

Of course all these methods also accept messages. Compared to `Node` methods they have a new `msg` argument. The `target` part on the other hand is only used by the `BiFlow`.

As you can see from `train`, the training does not always stop when the training node is reached. Instead it is possible to continue with the execution to come back later. For example this is used in the backpropagation example (in the MDP examples repository). There are also the new `stop_training` result options that start an execute phase. This can be used to propagate results from the node training or to prepare nodes for their upcoming training.

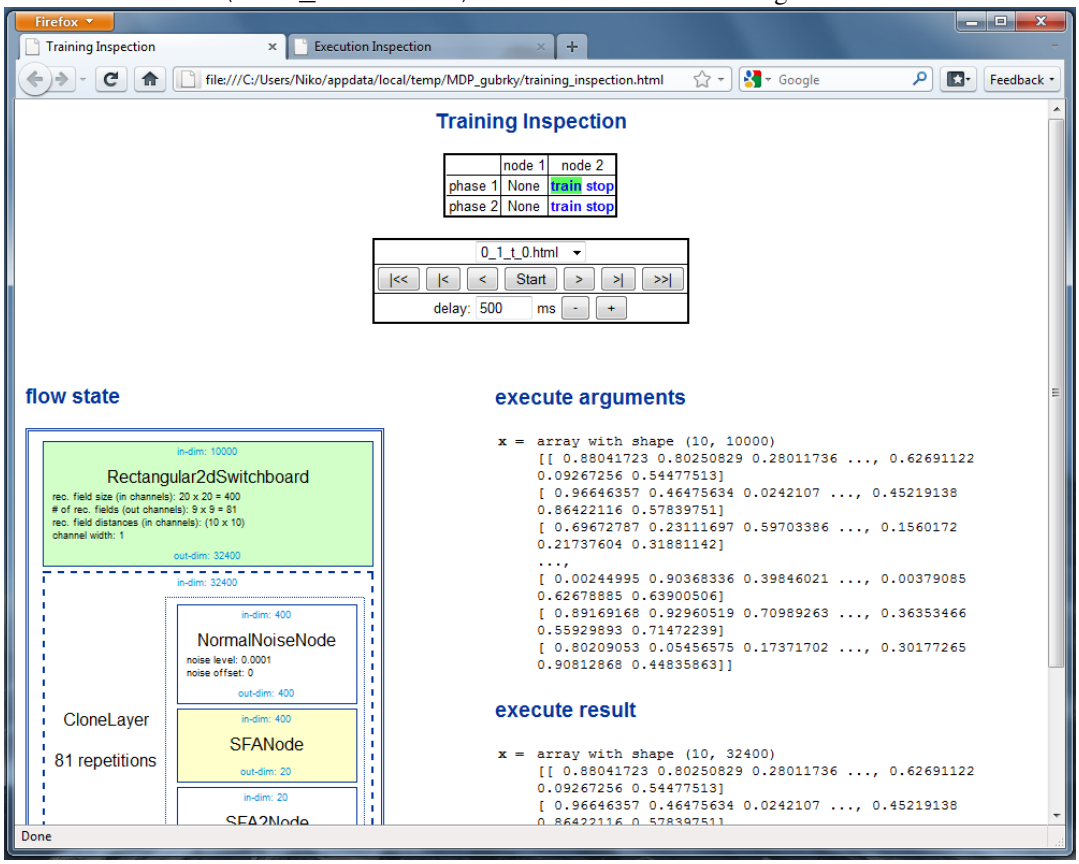
Some of these new options might be confusing at first. However, you can simply ignore those that you don't need and concentrate on the features that are useful for your current project. For example you could use messages without ever worrying about targets.

There are also two more additions to the `BiNode` API:

- **`node_id`** This is a read-only property, which returns the node id (which is `None` if it wasn't specified). The `__init__` method of a `BiNode` generally accepts a `node_id` keyword argument to set this value.
- **`bi_reset`** This method is called by the `BiFlow` before and after training and execution (and after the `stop_training` execution phase). You can be override the private `_bi_reset` method to reset internal state variables (`_bi_reset` is called by `bi_reset`).

13.4 Inspection

Using jumps and messages can result in complex data flows. Therefore BiMDP offers some convenient inspection capabilities to help with debugging and analyzing what is going on. This functionality is based on the static HTML view from the `mdp.hinet` module. Instead of a static view of the flow you get an animated slideshow of the flow training or execution. An example is provided in `bimdp/test/demo_hinet_inspection.py`. You can simply call `bimdp.show_execution(flow, data)` instead of the normal `flow.execute(data)`. This will automatically perform the inspection and open it in your webbrowser. Similar functionality is available for training. Just call `bimdp.show_execution(flow, data_iterables)`, which will perform training as in `flow.train(data_iterables)`. Have a look at the docstrings to learn about additional options.



The BiMDP inspection is also useful to visualize the data processing that is happening inside a flow. This is especially handy if you are trying to build or understand new algorithms and want to know what is going on. Therefore we made it very easy to customize the HTML views in the inspection. One simple example is provided in `bimdp/test/demo_custom_inspection.py`, where we use matplotlib to plot the data and present it inside the HTML view. Note that `bimdp.show_training` and `bimdp.show_execution` are just helper

functions. If you need more flexibility you can directly access the machinery below (but this is rather messy and hardly ever needed).

Browser Compatibility

The inspection works with all browser except Chrome. This is due to a controversial chromium issue. Until this is fixed by the Chrome developers the only workarounds are to either start Chrome with the `--allow-file-access-from-files` flag or to access the inspection via a webserver.

13.5 Extending BiNode and Message Handling

As in the `Node` class any derived `BiNode` classes should not directly overwrite the public `execute` or `train` methods but instead the private versions with an underscore in front (for training you can of course also overwrite `_get_train_seq`). In addition to the dimensionality checks performed on `x` by the `Node` class this enables a couple of message handling features.

The automatic message handling is a major feature in `BiNode` and relies on the dynamic nature of Python. In the `FDABiNode` and `BiFlow` example we have already seen how a value from the message is automatically passed to the `_train` method, because the key of the value is also the name of a keyword argument.

Public methods like `execute` in `BiNode` accept not only a data array `x`, but also a message dictionary `msg`. When given a message they perform introspection to determine the arguments for the corresponding private methods (like `_train`). If there is a matching key for an argument in the message then the value is provided as a keyword argument. It remains in the dictionary and can therefore be used by other nodes in the flow as well.

A private method like `_train` has the same return options as the public `train` method, so one can for example return a tuple `(x, msg)`. The `msg` in the return value from `_train` is then used by `train` to update the original `msg`. Thereby `_train` can overwrite or add new values to the message. There are also some special features ("magic") to make handling messages more convenient:

- You can use message keys of the form `node_id->argument_key` to address parts of the message to a specific node. When the node with the corresponding id is reached then the value is not only provided as an argument, but the key is also deleted from the message. If the `argument_key` is not an argument of the method then the whole key is simply erased.
- If a private method like `_train` has a keyword argument called `msg` then the complete message is provided. The message from the return value replaces the original message in this case. For example this makes it possible to delete parts of the message (instead of just updating them with new values).
- The key `"method"` is treated in a special way. Instead of calling the standard private method like `_train` (or `_execute`, depending on the called public method) the `"method"` value will be used as the method name, with an underscore in front. For example the message `{"method": "classify"}` has the effect that a method `_classify` will be called. Note that this feature can be combined with the extension mechanism, when methods are added at runtime.
- The key `"target"` is treated in a special way. If the called private method does not return a target value (e.g., if it just returned `x`) then the `"target"` value is used as target return value (e.g, instead of `x` the return value of `execute` would then have the form `x, None, target`).
- If the key `"method"` has the value `inverse` then, as expected, the `_inverse` method is called. However, additionally the checks from `inverse` are run on the data array. If `_inverse` does not return a target value then the target -1 is returned. So with the message `{"method": "inverse"}` one can execute a `BiFlow` in inverse node (note that one also has to provide the last node in the flow as the initial target to the flow).
- This is more of a `BiFlow` feature, but the target value specified in `bimdp.EXIT_TARGET` (currently set to `"exit"`) causes `BiFlow` to terminate the execution and to return the last return value.

Of course all these features can be combined, or can be ignored when they are not needed.

13.6 HiNet in BiMDP

BiMDP is mostly compatible with the hierarchical networks introduced in `mdp.hinet`. For the full BiMDP functionality it is of required to use the BiMDP versions of the the building blocks.

The `bimdp.hinet` module provides a `BiFlowNode` class, which is offers the same functionality as a `FlowNode` but with the added capability of handling messages, targets, and all other BiMDP concepts.

There is also a new `BiSwitchboard` base class, which is able to deal with messages. Arrays present in the message are mapped with the switchboard routing if the second axis matches the switchboard dimension (this works for both execute and inverse).

Finally there is a `CloneBiLayer` class, which is the BiMDP version of the `CloneLayer` class in `mdp.hinet`. To support all the features of BiMDP some significant functionality has been added to this class. The most important new aspect is the `use_copies` property. If it is set to `True` then multiple deep copies are used instead of just a reference to the same node. This makes it possible to use internal variables in a node that persist while the node is left and later reentered. You can set this property as often as you like (note that there is of course some overhead for the deep copying). You can also set the `use_copies` property via the message mechanism by simply adding a `"use_copies"` key with the required boolean value. The `CloneBiLayer` class also looks for this key in outgoing messages (so it can be send by nodes inside the layer). A `CloneBiLayer` can also split arrays in the message to feed them to the nodes (see the doctring for more details). `CloneBiLayer` is compatible with the target mechanism (e.g. if the `CloneBiLayer` contains a `BiFlowNode` you can target an internal node).

13.7 Parallel in BiMDP

The parallelisation capabilites introduced in `mdp.parallel` can be used for BiMDP. The `bimdp.parallel` module provides a `ParallelBiFlow` class which can be used like the normal `ParallelFlow`. No changes to schedulers are required.

Note that a `ParallelBiFlow` uses a special callable class to handle the message data. So if you want to use a custom callable you will have to make a few modifications (compared to the standard callable class used by `ParallFlow`).

13.8 Coroutine Decorator

For complex flow control (like in the DBN example) one might need a node that keeps track of the current status in the execution. The standard pattern for this is to implement a state machine, which would require some boilerplate code. Python on the other hand supports so called *continuations* via *coroutines*. A coroutine is very similar to a generator function, but the `yield` statement can also return a value (i.e., the coroutine is receiving a value). Coroutines might be difficult to grasp, but they are well documented on the web. Most importantly, coroutines can be a very elegant implementation of the state machine pattern.

Using a couroutine in a BiNode to maintain a state would still require some boilerplate code. Therefore BiMDP provides a special function decorator to minimize the effort, making it extremely convenient to use coroutines. This is demonstrated in the `gradnewton` and `binetdbn` examples. For example decorating the `_execute` method can be done like this:

```
>>> class SimpleCoroutineNode(bimdp.nodes.IdentityBiNode):
...     # the arg ["b"] means that that the signature will be (x, b)
...     @bimdp.binode_coroutine(["b"])
...     def _execute(self, x, n_iterations):
...         """Gather all the incomming b and return them finally."""
...         bs = []
...         for _ in range(n_iterations):
...             x, b = yield x
...             bs.append(b)
...         raise StopIteration(x, {"all the b": bs})
```

```
>>> n_iterations = 3
>>> x = np.random.random((1,1))
>>> node = SimpleCoroutineNode()
>>> # during the first call the decorator creates the actual coroutine
>>> x, msg = node.execute(x, {"n_iterations": n_iterations})
>>> # the following calls go to the yield statement,
>>> # finally the bs are returned
>>> for i in range(n_iterations-1):
...     x, msg = node.execute(x, {"b": i})
>>> x, msg = node.execute(x, {"b": n_iterations-1})
```

You can find the complete runable code in the `bimdp_simple_coroutine.py` example.

13.9 Classifiers in BiMDP

BiMDP introduces a special `BiClassifier` base class for the new `Classifier` nodes in MDP. This makes it possible to fully use classifiers in a normal `BiFlow`. Just like for normal nodes the BiMDP versions of the classifier are available in `bimdp.nodes` (the SVM classifiers are currently not available by default, but it is possible to manually derive a `BiClassifier` version of them).

The `BiClassifier` class makes it possible to provide the training labels via the message mechanism (simply store the labels with a `"labels"` key in the `msg` dict). It is also possible to transport the classification results in the outgoing message. The `_execute` method of a `BiClassifier` has three keyword arguments called `return_labels`, `return_ranks`, and `return_probs`. These can be set via the message mechanism. If for example `return_labels` is set to `True` then `execute` will call the `label` method from the classifier node and store the result in the outgoing message (under the key `"labels"`). The `return_labels` argument (and the other two) can also be set to a string value, which is then used as a prefix for the `"labels"` key in the outgoing message (e.g., to target this information at a specific node in the flow).

CHAPTER

FOURTEEN

NODE LIST

class mdp.nodes.**PCANode**
Filter the input data through the most significatives of its principal components.

Internal variables of interest

self.avg Mean of the input data (available after training).

self.v Transposed of the projection matrix (available after training).

self.d Variance corresponding to the PCA components (eigenvalues of the covariance matrix).

self.explained_variance When output_dim has been specified as a fraction of the total variance, this is the fraction of the total variance that is actually explained.

More information about Principal Component Analysis, a.k.a. discrete Karhunen-Loeve transform can be found among others in I.T. Jolliffe, Principal Component Analysis, Springer-Verlag (1986).

Full API documentation: PCANode

class mdp.nodes.**WhiteningNode**
Whiten the input data by filtering it through the most significatives of its principal components. All output signals have zero mean, unit variance and are decorrelated.

Internal variables of interest

self.avg Mean of the input data (available after training).

self.v Transpose of the projection matrix (available after training).

self.d Variance corresponding to the PCA components (eigenvalues of the covariance matrix).

self.explained_variance When output_dim has been specified as a fraction of the total variance, this is the fraction of the total variance that is actually explained.

Full API documentation: WhiteningNode

class mdp.nodes.**NIPALSNode**
Perform Principal Component Analysis using the NIPALS algorithm. This algorithm is particularyl useful if you have more variable than observations, or in general when the number of variables is huge and calculating a full covariance matrix may be unfeasable. It's also more efficient of the standard PCANode if you expect the number of significant principal components to be a small. In this case setting output_dim to be a certain fraction of the total variance, say 90%, may be of some help.

Internal variables of interest

self.avg Mean of the input data (available after training).

self.d Variance corresponding to the PCA components.

self.v Transposed of the projection matrix (available after training).

self.explained_variance When output_dim has been specified as a fraction of the total variance, this is the fraction of the total variance that is actually explained.

Reference for NIPALS (Nonlinear Iterative Partial Least Squares): Wold, H. Nonlinear estimation by iterative least squares procedures. in David, F. (Editor), Research Papers in Statistics, Wiley, New York, pp 411-444 (1966).

More information about Principal Component Analysis, a.k.a. discrete Karhunen-Loeve transform can be found among others in I.T. Jolliffe, Principal Component Analysis, Springer-Verlag (1986).

Original code contributed by: Michael Schmuker, Susanne Lezius, and Farzad Farkhooi (2008).

Full API documentation: NIPALSNode

class mdp.nodes.**FastICANode**

Perform Independent Component Analysis using the FastICA algorithm. Note that FastICA is a batch-algorithm. This means that it needs all input data before it can start and compute the ICs. The algorithm is here given as a Node for convenience, but it actually accumulates all inputs it receives. Remember that to avoid running out of memory when you have many components and many time samples.

FastICA does not support the telescope mode (the convergence criterium is not robust in telescope mode).

Reference: Aapo Hyvarinen (1999). Fast and Robust Fixed-Point Algorithms for Independent Component Analysis IEEE Transactions on Neural Networks, 10(3):626-634.

Internal variables of interest

self.white The whitening node used for preprocessing.

self.filters The ICA filters matrix (this is the transposed of the projection matrix after whitening).

self.convergence The value of the convergence threshold.

History:

- 1.4.1998 created for Matlab by Jarmo Hurri, Hugo Gavert, Jaakko Sarela, and Aapo Hyvarinen
- 7.3.2003 modified for Python by Thomas Wendler
- 3.6.2004 rewritten and adapted for scipy and MDP by MDP's authors
- 25.5.2005 now independent from scipy. Requires Numeric or numarray
- 26.6.2006 converted to numpy
- 14.9.2007 updated to Matlab version 2.5

Full API documentation: FastICANode

class mdp.nodes.**CuBICANode**

Perform Independent Component Analysis using the CuBICA algorithm. Note that CuBICA is a batch-algorithm, which means that it needs all input data before it can start and compute the ICs. The algorithm is here given as a Node for convenience, but it actually accumulates all inputs it receives. Remember that to avoid running out of memory when you have many components and many time samples.

As an alternative to this batch mode you might consider the telescope mode (see the docs of the `__init__` method).

Reference: Blaschke, T. and Wiskott, L. (2003). CuBICA: Independent Component Analysis by Simultaneous Third- and Fourth-Order Cumulant Diagonalization. IEEE Transactions on Signal Processing, 52(5), pp. 1250-1256.

Internal variables of interest

self.white The whitening node used for preprocessing.

self.filters The ICA filters matrix (this is the transposed of the projection matrix after whitening).

self.convergence The value of the convergence threshold.

Full API documentation: CuBICANode

class mdp.nodes.**TDSEPNode**

Perform Independent Component Analysis using the TDSEP algorithm. Note that TDSEP, as implemented in this Node, is an online algorithm, i.e. it is suited to be trained on huge data sets, provided that the training is done sending small chunks of data for each time.

Reference: Ziehe, Andreas and Muller, Klaus-Robert (1998). TDSEP an efficient algorithm for blind separation using time structure. in Niklasson, L, Boden, M, and Ziemke, T (Editors), Proc. 8th Int. Conf. Artificial Neural Networks (ICANN 1998).

Internal variables of interest

self.white The whitening node used for preprocessing.

self.filters The ICA filters matrix (this is the transposed of the projection matrix after whitening).

self.convergence The value of the convergence threshold.

Full API documentation: TDSEPNode

class mdp.nodes.**JADENode**

Perform Independent Component Analysis using the JADE algorithm. Note that JADE is a batch-algorithm. This means that it needs all input data before it can start and compute the ICs. The algorithm is here given as a Node for convenience, but it actually accumulates all inputs it receives. Remember that to avoid running out of memory when you have many components and many time samples.

JADE does not support the telescope mode.

Main references:

•Cardoso, Jean-Francois and Souloumiac, Antoine (1993). Blind beamforming for non Gaussian signals. Radar and Signal Processing, IEE Proceedings F, 140(6): 362-370.

•Cardoso, Jean-Francois (1999). High-order contrasts for independent component analysis. Neural Computation, 11(1): 157-192.

Original code contributed by: Gabriel Beckers (2008).

History:

•May 2005 version 1.8 for MATLAB released by Jean-Francois Cardoso

•Dec 2007 MATLAB version 1.8 ported to Python/NumPy by Gabriel Beckers

•Feb 15 2008 Python/NumPy version adapted for MDP by Gabriel Beckers

Full API documentation: JADENode

class mdp.nodes.**SFANode**

Extract the slowly varying components from the input data. More information about Slow Feature Analysis can be found in Wiskott, L. and Sejnowski, T.J., Slow Feature Analysis: Unsupervised Learning of Invariances, Neural Computation, 14(4):715-770 (2002).

Instance variables of interest

self.avg Mean of the input data (available after training)

self.sf Matrix of the SFA filters (available after training)

self.d Delta values corresponding to the SFA components (generalized eigenvalues). [See the docs of the `get_eta_values` method for more information]

Special arguments for constructor

include_last_sample If `False` the *train* method discards the last sample in every chunk during training when calculating the covariance matrix. The last sample is in this case only used for calculating the covariance matrix of the derivatives. The switch should be set to `False` if you plan to train with several small chunks. For example we can split a sequence (index is time):

```
x_1 x_2 x_3 x_4
```

in smaller parts like this:

```
x_1 x_2
x_2 x_3
x_3 x_4
```

The SFANode will see 3 derivatives for the temporal covariance matrix, and the first 3 points for the spatial covariance matrix. Of course you will need to use a generator that *connects* the small chunks (the last sample needs to be sent again in the next chunk). If `include_last_sample` was True, depending on the generator you use, you would either get:

```
x_1 x_2
x_2 x_3
x_3 x_4
```

in which case the last sample of every chunk would be used twice when calculating the covariance matrix, or:

```
x_1 x_2
x_3 x_4
```

in which case you loose the derivative between `x_3` and `x_2`.

If you plan to train with a single big chunk leave `include_last_sample` to the default value, i.e. `True`.

You can even change this behaviour during training. Just set the corresponding switch in the *train* method.

Full API documentation: SFANode

class `mdp.nodes.`**`SFA2Node`**

Get an input signal, expand it in the space of inhomogeneous polynomials of degree 2 and extract its slowly varying components. The `get_quadratic_form` method returns the input-output function of one of the learned unit as a `QuadraticForm` object. See the documentation of `mdp.utils.QuadraticForm` for additional information.

More information about Slow Feature Analysis can be found in Wiskott, L. and Sejnowski, T.J., Slow Feature Analysis: Unsupervised Learning of Invariances, Neural Computation, 14(4):715-770 (2002).

Full API documentation: SFA2Node

class `mdp.nodes.`**`ISFANode`**

Perform Independent Slow Feature Analysis on the input data.

Internal variables of interest

`self.RP` The global rotation-permutation matrix. This is the filter applied on input_data to get output_data

`self.RPC` The *complete* global rotation-permutation matrix. This is a matrix of dimension input_dim x input_dim (the 'outer space' is retained)

`self.covs` A *mdp.utils.MultipleCovarianceMatrices* instance containing the current time-delayed covariance matrices of the input_data. After convergence the uppermost `output_dim` x `output_dim` submatrices should be almost diagonal.

`self.covs[n-1]` is the covariance matrix relative to the `n`-th time-lag

Note: they are not cleared after convergence. If you need to free some memory, you can safely delete them with:

```
>>> del self.covs
```

self.initial_contrast A dictionary with the starting contrast and the SFA and ICA parts of it.

self.final_contrast Like the above but after convergence.

Note: If you intend to use this node for large datasets please have a look at the `stop_training` method documentation for speeding things up.

References: Blaschke, T. , Zito, T., and Wiskott, L. (2007). Independent Slow Feature Analysis and Nonlinear Blind Source Separation. Neural Computation 19(4):994-1021 (2007) http://itb.biologie.hu-berlin.de/~wiskott/Publications/BlasZitoWisk2007-ISFA-NeurComp.pdf

Full API documentation: ISFANode

class `mdp.nodes.`**XSFANode**

Perform Non-linear Blind Source Separation using Slow Feature Analysis.

This node is designed to iteratively extract statistically independent sources from (in principle) arbitrary invertible nonlinear mixtures. The method relies on temporal correlations in the sources and consists of a combination of nonlinear SFA and a projection algorithm. More details can be found in the reference given below (once it's published).

The node has multiple training phases. The number of training phases depends on the number of sources that must be extracted. The recommended way of training this node is through a container flow:

```
>>> flow = mdp.Flow([XSFANode()])
>>> flow.train(x)
```

doing so will automatically train all training phases. The argument `x` to the `Flow.train` method can be an array or a list of iterables (see the section about Iterators in the MDP tutorial for more info).

If the number of training samples is large, you may run into memory problems: use data iterators and chunk training to reduce memory usage.

If you need to debug training and/or execution of this node, the suggested approach is to use the capabilities of BiMDP. For example:

```
>>> flow = mdp.Flow([XSFANode()])
>>> tr_filename = bimdp.show_training(flow=flow, data_iterators=x)
>>> ex_filename, out = bimdp.show_execution(flow, x=x)
```

this will run training and execution with bimdp inspection. Snapshots of the internal flow state for each training phase and execution step will be opened in a web brower and presented as a slideshow.

References: Sprekeler, H., Zito, T., and Wiskott, L. (2009). An Extension of Slow Feature Analysis for Nonlinear Blind Source Separation. Journal of Machine Learning Research. http://cogprints.org/7056/1/SprekelerZitoWiskott-Cogprints-2010.pdf

Full API documentation: XSFANode

class `mdp.nodes.`**FDANode**

Perform a (generalized) Fisher Discriminant Analysis of its input. It is a supervised node that implements FDA using a generalized eigenvalue approach.

FDANode has two training phases and is supervised so make sure to pay attention to the following points when you train it:

- call the `train` method with *two* arguments: the input data and the labels (see the doc string of the `train` method for details).
- if you are training the node by hand, call the `train` method twice.
- if you are training the node using a flow (recommended), the only argument to `Flow.train` must be a list of `(data_point, label)` tuples or an iterator returning lists of such tuples, *not* a generator.

The `Flow.train` function can be called just once as usual, since it takes care of *rewinding* the iterator to perform the second training step.

More information on Fisher Discriminant Analysis can be found for example in C. Bishop, Neural Networks for Pattern Recognition, Oxford Press, pp. 105-112.

Internal variables of interest

`self.avg` Mean of the input data (available after training)

`self.v` Transposed of the projection matrix, so that `output = dot(input-self.avg, self.v)` (available after training).

Full API documentation: FDANode

class `mdp.nodes.`**`FANode`**

Perform Factor Analysis.

The current implementation should be most efficient for long data sets: the sufficient statistics are collected in the training phase, and all EM-cycles are performed at its end.

The `execute` method returns the Maximum A Posteriori estimate of the latent variables. The `generate_input` method generates observations from the prior distribution.

Internal variables of interest

`self.mu` Mean of the input data (available after training)

`self.A` Generating weights (available after training)

`self.E_y_mtx` Weights for Maximum A Posteriori inference

`self.sigma` Vector of estimated variance of the noise for all input components

More information about Factor Analysis can be found in Max Welling's classnotes: http://www.ics.uci.edu/~welling/classnotes/classnotes.html , in the chapter 'Linear Models'.

Full API documentation: FANode

class `mdp.nodes.`**`RBMNode`**

Restricted Boltzmann Machine node. An RBM is an undirected probabilistic network with binary variables. The graph is bipartite into observed (*visible*) and hidden (*latent*) variables.

By default, the `execute` method returns the *probability* of one of the hiden variables being equal to 1 given the input.

Use the `sample_v` method to sample from the observed variables given a setting of the hidden variables, and `sample_h` to do the opposite. The `energy` method can be used to compute the energy of a given setting of all variables.

The network is trained by Contrastive Divergence, as described in Hinton, G. E. (2002). Training products of experts by minimizing contrastive divergence. Neural Computation, 14(8):1711-1800

Internal variables of interest

`self.w` Generative weights between hidden and observed variables

`self.bv` bias vector of the observed variables

`self.bh` bias vector of the hidden variables

For more information on RBMs, see Geoffrey E. Hinton (2007) Boltzmann machine. Scholarpedia, 2(5):1668

Full API documentation: RBMNode

class `mdp.nodes.`**`RBMWithLabelsNode`**

Restricted Boltzmann Machine with softmax labels. An RBM is an undirected probabilistic network with binary variables. In this case, the node is partitioned into a set of observed (*visible*) variables, a set of hidden (*latent*) variables, and a set of label variables (also observed), only one of which is active at any time. The node is able to learn associations between the visible variables and the labels.

By default, the `execute` method returns the *probability* of one of the hiden variables being equal to 1 given the input.

Use the `sample_v` method to sample from the observed variables (visible and labels) given a setting of the hidden variables, and `sample_h` to do the opposite. The `energy` method can be used to compute the energy of a given setting of all variables.

The network is trained by Contrastive Divergence, as described in Hinton, G. E. (2002). Training products of experts by minimizing contrastive divergence. Neural Computation, 14(8):1711-1800

Internal variables of interest:

`self.w` Generative weights between hidden and observed variables

`self.bv` bias vector of the observed variables

`self.bh` bias vector of the hidden variables

For more information on RBMs with labels, see

- Geoffrey E. Hinton (2007) Boltzmann machine. Scholarpedia, 2(5):1668.
- Hinton, G. E, Osindero, S., and Teh, Y. W. (2006). A fast learning algorithm for deep belief nets. Neural Computation, 18:1527-1554.

Full API documentation: RBMWithLabelsNode

class `mdp.nodes.`**`GrowingNeuralGasNode`**

Learn the topological structure of the input data by building a corresponding graph approximation.

More information about the Growing Neural Gas algorithm can be found in B. Fritzke, A Growing Neural Gas Network Learns Topologies, in G. Tesauro, D. S. Touretzky, and T. K. Leen (editors), Advances in Neural Information Processing Systems 7, pages 625-632. MIT Press, Cambridge MA, 1995.

A java implementation is available at: http://www.neuroinformatik.ruhr-uni-bochum.de/ini/VDM/research/gsn/DemoGNG/GNG.html

Attributes and methods of interest

- graph – The corresponding *mdp.graph.Graph* object

Full API documentation: GrowingNeuralGasNode

class `mdp.nodes.`**`LLENode`**

Perform a Locally Linear Embedding analysis on the data.

Internal variables of interest

`self.training_projection` The LLE projection of the training data (defined when training finishes).

`self.desired_variance` variance limit used to compute intrinsic dimensionality.

Based on the algorithm outlined in *An Introduction to Locally Linear Embedding* by L. Saul and S. Roweis, using improvements suggested in *Locally Linear Embedding for Classification* by D. deRidder and R.P.W. Duin.

References: Roweis, S. and Saul, L., Nonlinear dimensionality reduction by locally linear embedding, Science 290 (5500), pp. 2323-2326, 2000.

Original code contributed by: Jake VanderPlas, University of Washington,

Full API documentation: LLENode

class `mdp.nodes.`**`HLLENode`**

Perform a Hessian Locally Linear Embedding analysis on the data.

Internal variables of interest

`self.training_projection` the HLLE projection of the training data (defined when training finishes)

self.desired_variance variance limit used to compute intrinsic dimensionality.

Implementation based on algorithm outlined in Donoho, D. L., and Grimes, C., Hessian Eigenmaps: new locally linear embedding techniques for high-dimensional data, Proceedings of the National Academy of Sciences 100(10): 5591-5596, 2003.

Original code contributed by: Jake Vanderplas, University of Washington

Full API documentation: HLLENode

class mdp.nodes.**LinearRegressionNode**
Compute least-square, multivariate linear regression on the input data, i.e., learn coefficients b_j so that:

```
y_i = b_0 + b_1 x_1 + ... b_N x_N ,
```

for i = 1 ... M, minimizes the square error given the training x's and y's.

This is a supervised learning node, and requires input data x and target data y to be supplied during training (see train docstring).

Internal variables of interest

self.beta The coefficients of the linear regression

Full API documentation: LinearRegressionNode

class mdp.nodes.**QuadraticExpansionNode**
Perform expansion in the space formed by all linear and quadratic monomials. QuadraticExpansionNode() is equivalent to a PolynomialExpansionNode(2)

Full API documentation: QuadraticExpansionNode

class mdp.nodes.**PolynomialExpansionNode**
Perform expansion in a polynomial space.

Full API documentation: PolynomialExpansionNode

class mdp.nodes.**RBFExpansionNode**
Expand input space with Gaussian Radial Basis Functions (RBFs).

The input data is filtered through a set of unnormalized Gaussian filters, i.e.:

```
y_j = exp(-0.5/s_j * ||x - c_j||^2)
```

for isotropic RBFs, or more in general:

```
y_j = exp(-0.5 * (x-c_j)^T S^-1 (x-c_j))
```

for anisotropic RBFs.

Full API documentation: RBFExpansionNode

class mdp.nodes.**GeneralExpansionNode**
Expands the input signal x according to a list [f_0, ... f_k] of functions.

Each function f_i should take the whole two-dimensional array x as input and output another two-dimensional array. Moreover the output dimension should depend only on the input dimension. The output of the node is [f_0[x], ... f_k[x]], that is, the concatenation of each one of the outputs f_i[x].

Original code contributed by Alberto Escalante.

Full API documentation: GeneralExpansionNode

class mdp.nodes.**GrowingNeuralGasExpansionNode**
Perform a trainable radial basis expansion, where the centers and sizes of the basis functions are learned through a growing neural gas.

positions of RBFs position of the nodes of the neural gas

sizes of the RBFs mean distance to the neighbouring nodes.

Important: Adjust the maximum number of nodes to control the dimension of the expansion.

More information on this expansion type can be found in: B. Fritzke. Growing cell structures-a self-organizing network for unsupervised and supervised learning. Neural Networks 7, p. 1441–1460 (1994).

Full API documentation: GrowingNeuralGasExpansionNode

class mdp.nodes.**SignumClassifier**
This classifier node classifies as `1` if the sum of the data points is positive and as `-1` if the data point is negative

Full API documentation: SignumClassifier

class mdp.nodes.**PerceptronClassifier**
A simple perceptron with input_dim input nodes.

Full API documentation: PerceptronClassifier

class mdp.nodes.**SimpleMarkovClassifier**
A simple version of a Markov classifier. It can be trained on a vector of tuples the label being the next element in the testing data.

Full API documentation: SimpleMarkovClassifier

class mdp.nodes.**DiscreteHopfieldClassifier**
Node for simulating a simple discrete Hopfield model

Full API documentation: DiscreteHopfieldClassifier

class mdp.nodes.**KMeansClassifier**
Employs K-Means Clustering for a given number of centroids.

Full API documentation: KMeansClassifier

class mdp.nodes.**NormalizeNode**
Make input signal meanfree and unit variance

Full API documentation: NormalizeNode

class mdp.nodes.**GaussianClassifier**
Perform a supervised Gaussian classification.

Given a set of labelled data, the node fits a gaussian distribution to each class.

Full API documentation: GaussianClassifier

class mdp.nodes.**NearestMeanClassifier**
Nearest-Mean classifier.

Full API documentation: NearestMeanClassifier

class mdp.nodes.**KNNClassifier**
K-Nearest-Neighbour Classifier.

Full API documentation: KNNClassifier

class mdp.nodes.**EtaComputerNode**
Compute the eta values of the normalized training data.

The delta value of a signal is a measure of its temporal variation, and is defined as the mean of the derivative squared, i.e. `delta(x) = mean(dx/dt(t)^2)`. `delta(x)` is zero if `x` is a constant signal, and increases if the temporal variation of the signal is bigger.

The eta value is a more intuitive measure of temporal variation, defined as:

```
eta(x) = T/(2*pi) * sqrt(delta(x))
```

If `x` is a signal of length `T` which consists of a sine function that accomplishes exactly `N` oscillations, then `eta(x)=N`.

`EtaComputerNode` normalizes the training data to have unit variance, such that it is possible to compare the temporal variation of two signals independently from their scaling.

Reference: Wiskott, L. and Sejnowski, T.J. (2002). Slow Feature Analysis: Unsupervised Learning of Invariances, Neural Computation, 14(4):715-770.

Important: if a data chunk is tlen data points long, this node is going to consider only the first tlen-1 points together with their derivatives. This means in particular that the variance of the signal is not computed on all data points. This behavior is compatible with that of `SFANode`.

This is an analysis node, i.e. the data is analyzed during training and the results are stored internally. Use the method `get_eta` to access them.

Full API documentation: EtaComputerNode

class `mdp.nodes.`**`HitParadeNode`**

Collect the first `n` local maxima and minima of the training signal which are separated by a minimum gap `d`.

This is an analysis node, i.e. the data is analyzed during training and the results are stored internally. Use the `get_maxima` and `get_minima` methods to access them.

Full API documentation: HitParadeNode

class `mdp.nodes.`**`NoiseNode`**

Inject multiplicative or additive noise into the input data.

Original code contributed by Mathias Franzius.

Full API documentation: NoiseNode

class `mdp.nodes.`**`NormalNoiseNode`**

Special version of `NoiseNode` for Gaussian additive noise.

Unlike `NoiseNode` it does not store a noise function reference but simply uses `numx_rand.normal`.

Full API documentation: NormalNoiseNode

class `mdp.nodes.`**`TimeFramesNode`**

Copy delayed version of the input signal on the space dimensions.

For example, for `time_frames=3` and `gap=2`:

```
[ X(1) Y(1)         [ X(1) Y(1) X(3) Y(3) X(5) Y(5)
  X(2) Y(2)           X(2) Y(2) X(4) Y(4) X(6) Y(6)
  X(3) Y(3)   -->     X(3) Y(3) X(5) Y(5) X(7) Y(7)
  X(4) Y(4)           X(4) Y(4) X(6) Y(6) X(8) Y(8)
  X(5) Y(5)           ...  ...  ...  ...  ...  ... ]
  X(6) Y(6)
  X(7) Y(7)
  X(8) Y(8)
  ...  ...  ]
```

It is not always possible to invert this transformation (the transformation is not surjective. However, the `pseudo_inverse` method does the correct thing when it is indeed possible.

Full API documentation: TimeFramesNode

class `mdp.nodes.`**`TimeDelayNode`**

Copy delayed version of the input signal on the space dimensions.

For example, for `time_frames=3` and `gap=2`:

```
[ X(1) Y(1)         [ X(1) Y(1)   0    0    0    0
  X(2) Y(2)           X(2) Y(2)   0    0    0    0
  X(3) Y(3)   -->     X(3) Y(3) X(1) Y(1)   0    0
  X(4) Y(4)           X(4) Y(4) X(2) Y(2)   0    0
  X(5) Y(5)           X(5) Y(5) X(3) Y(3) X(1) Y(1)
  X(6) Y(6)           ...  ...  ...  ...  ...  ... ]
```

```
X(7)  Y(7)
X(8)  Y(8)
...   ...   ]
```

This node provides similar functionality as the `TimeFramesNode`, only that it performs a time embedding into the past rather than into the future.

See `TimeDelaySlidingWindowNode` for a sliding window delay node for application in a non-batch manner.

Original code contributed by Sebastian Hoefer <mail@sebastianhoefer.de> Dec 31, 2010

Full API documentation: TimeDelayNode

class `mdp.nodes.`**`TimeDelaySlidingWindowNode`**

`TimeDelaySlidingWindowNode` is an alternative to `TimeDelayNode` which should be used for online learning/execution. Whereas the `TimeDelayNode` works in a batch manner, for online application a sliding window is necessary which yields only one row per call.

Applied to the same data the collection of all returned rows of the `TimeDelaySlidingWindowNode` is equivalent to the result of the `TimeDelayNode`.

Original code contributed by Sebastian Hoefer <mail@sebastianhoefer.de> Dec 31, 2010

Full API documentation: TimeDelaySlidingWindowNode

class `mdp.nodes.`**`CutoffNode`**

Node to cut off values at specified bounds.

Works similar to `numpy.clip`, but also works when only a lower or upper bound is specified.

Full API documentation: CutoffNode

class `mdp.nodes.`**`AdaptiveCutoffNode`**

Node which uses the data history during training to learn cutoff values.

As opposed to the simple `CutoffNode`, a different cutoff value is learned for each data coordinate. For example if an upper cutoff fraction of 0.05 is specified, then the upper cutoff bound is set so that the upper 5% of the training data would have been clipped (in each dimension). The cutoff bounds are then applied during execution. This node also works as a `HistogramNode`, so the histogram data is stored.

When `stop_training` is called the cutoff values for each coordinate are calculated based on the collected histogram data.

Full API documentation: AdaptiveCutoffNode

class `mdp.nodes.`**`HistogramNode`**

Node which stores a history of the data during its training phase.

The data history is stored in `self.data_hist` and can also be deleted to free memory. Alternatively it can be automatically pickled to disk.

Note that data is only stored during training.

Full API documentation: HistogramNode

class `mdp.nodes.`**`IdentityNode`**

Execute returns the input data and the node is not trainable.

This node can be instantiated and is for example useful in complex network layouts.

Full API documentation: IdentityNode

class `mdp.nodes.`**`Convolution2DNode`**

Convolve input data with filter banks.

The `filters` argument specifies a set of 2D filters that are convolved with the input data during execution. Convolution can be selected to be executed by linear filtering of the data, or in the frequency domain using a Discrete Fourier Transform.

Input data can be given as 3D data, each row being a 2D array to be convolved with the filters, or as 2D data, in which case the `input_shape` argument must be specified.

This node depends on `scipy`.

Full API documentation: Convolution2DNode

class `mdp.nodes.`**`ShogunSVMClassifier`**

The `ShogunSVMClassifier` works as a wrapper class for accessing the SHOGUN machine learning toolbox for support vector machines.

Most kernel machines and linear classifier should work with this class.

Currently, distance machines such as the K-means classifier are not supported yet.

Information to paramters and additional options can be found on http://www.shogun-toolbox.org/

Note that some parts in this classifier might receive some refinement in the future.

This node depends on `shogun`.

Full API documentation: ShogunSVMClassifier

class `mdp.nodes.`**`LibSVMClassifier`**

The `LibSVMClassifier` class acts as a wrapper around the LibSVM library for support vector machines.

Information to the parameters can be found on http://www.csie.ntu.edu.tw/~cjlin/libsvm/

The class provides access to change kernel and svm type with a text string.

Additionally `self.parameter` is exposed which allows to change all other svm parameters directly.

This node depends on `libsvm`.

Full API documentation: LibSVMClassifier

class `mdp.nodes.`**`SGDRegressorScikitsLearnNode`**

Linear model fitted by minimizing a regularized empirical loss with SGD

This node has been automatically generated by wrapping the `scikits.learn.linear_model.sparse.stochastic` class from the `scikits.learn` library. The wrapped instance can be accessed through the `scikits_alg` attribute.

SGD stands for Stochastic Gradient Descent: the gradient of the loss is estimated each sample at a time and the model is updated along the way with a decreasing strength schedule (aka learning rate).

The regularizer is a penalty added to the loss function that shrinks model parameters towards the zero vector using either the squared euclidean norm L2 or the absolute norm L1 or a combination of both (Elastic Net). If the parameter update crosses the 0.0 value because of the regularizer, the update is truncated to 0.0 to allow for learning sparse models and achieve online feature selection.

This implementation works with data represented as dense numpy arrays of floating point values for the features.

Parameters

loss [str, 'squared_loss' or 'huber'] The loss function to be used. Defaults to 'squared_loss' which refers to the ordinary least squares fit. 'huber' is an epsilon insensitive loss function for robust regression.

penalty [str, 'l2' or 'l1' or 'elasticnet'] The penalty (aka regularization term) to be used. Defaults to 'l2' which is the standard regularizer for linear SVM models. 'l1' and 'elasticnet' migh bring sparsity to the model (feature selection) not achievable with 'l2'.

alpha [float] Constant that multiplies the regularization term. Defaults to 0.0001

rho [float] The Elastic Net mixing parameter, with 0 < rho <= 1. Defaults to 0.85.

fit_intercept: bool Whether the intercept should be estimated or not. If False, the data is assumed to be already centered. Defaults to True.

n_iter: int The number of passes over the training data (aka epochs). Defaults to 5.

shuffle: bool Whether or not the training data should be shuffled after each epoch. Defaults to False.

verbose: integer, optional The verbosity level

p [float] Epsilon in the epsilon insensitive huber loss function; only if *loss=='huber'*.

Attributes

coef_ [array, shape = [n_features]] Weights asigned to the features.

intercept_ [array, shape = [1]] The intercept term.

Examples

```
>>> import numpy as np
>>> from scikits.learn import linear_model
>>> n_samples, n_features = 10, 5
>>> np.random.seed(0)
>>> y = np.random.randn(n_samples)
>>> X = np.random.randn(n_samples, n_features)
>>> clf = linear_model.sparse.SGDRegressor()
>>> clf.fit(X, y)
SGDRegressor(loss='squared_loss', shuffle=False, verbose=0, n_iter=5,
       fit_intercept=True, penalty='l2', p=0.1, rho=1.0, alpha=0.0001)
```

See also

RidgeRegression, ElasticNet, Lasso, SVR

Full API documentation: SGDRegressorScikitsLearnNode

class mdp.nodes.**RFEScikitsLearnNode**

Feature ranking with Recursive feature elimination

This node has been automatically generated by wrapping the `scikits.learn.feature_selection.rfe.RFE` class from the `scikits.learn` library. The wrapped instance can be accessed through the `scikits_alg` attribute.

Parameters

estimator [object] A supervised learning estimator with a fit method that updates a `coef_` attributes that holds the fitted parameters. The first dimension of the `coef_` array must be equal n_features an important features must yield high absolute values in the `coef_` array.

For instance this is the case for most supervised learning algorithms such as Support Vector Classifiers and Generalized Linear Models from the svm and linear_model package.

n_features [int] Number of features to select

percentage [float] The percentage of features to remove at each iteration Should be between (0, 1]. By default 0.1 will be taken.

Attributes

support_ [array-like, shape = [n_features]] Mask of estimated support

ranking_ [array-like, shape = [n_features]] Mask of the ranking of features

Methods

fit(X, y) [self] Fit the model

transform(X) [array] Reduce X to support

Examples

```
>>> # TODO!
```

References

Guyon, I., Weston, J., Barnhill, S., & Vapnik, V. (2002). Gene selection for cancer classification using support vector machines. Mach. Learn., 46(1-3), 389–422.

Full API documentation: RFEScikitsLearnNode

class mdp.nodes.**SparseBaseLibSVMScikitsLearnNode**

class mdp.nodes.**VectorizerScikitsLearnNode**
Convert a collection of raw documents to a matrix

This node has been automatically generated by wrapping the scikits.learn.feature_extraction.text.dense class from the scikits.learn library. The wrapped instance can be accessed through the scikits_alg attribute.

Equivalent to CountVectorizer followed by TfidfTransformer.

Full API documentation: VectorizerScikitsLearnNode

class mdp.nodes.**SGDClassifierScikitsLearnNode**
Linear model fitted by minimizing a regularized empirical loss with SGD.

This node has been automatically generated by wrapping the scikits.learn.linear_model.stochastic_gradie class from the scikits.learn library. The wrapped instance can be accessed through the scikits_alg attribute.

SGD stands for Stochastic Gradient Descent: the gradient of the loss is estimated each sample at a time and the model is updated along the way with a decreasing strength schedule (aka learning rate).

The regularizer is a penalty added to the loss function that shrinks model parameters towards the zero vector using either the squared euclidean norm L2 or the absolute norm L1 or a combination of both (Elastic Net). If the parameter update crosses the 0.0 value because of the regularizer, the update is truncated to 0.0 to allow for learning sparse models and achieve online feature selection.

This implementation works with data represented as dense numpy arrays of floating point values for the features.

Parameters

loss [str, 'hinge' or 'log' or 'modified_huber'] The loss function to be used. Defaults to 'hinge'. The hinge loss is a margin loss used by standard linear SVM models. The 'log' loss is the loss of logistic regression models and can be used for probability estimation in binary classifiers. 'modified_huber' is another smooth loss that brings tolerance to outliers.

penalty [str, 'l2' or 'l1' or 'elasticnet'] The penalty (aka regularization term) to be used. Defaults to 'l2' which is the standard regularizer for linear SVM models. 'l1' and 'elasticnet' migh bring sparsity to the model (feature selection) not achievable with 'l2'.

alpha [float] Constant that multiplies the regularization term. Defaults to 0.0001

rho [float] The Elastic Net mixing parameter, with 0 < rho <= 1. Defaults to 0.85.

fit_intercept: bool Whether the intercept should be estimated or not. If False, the data is assumed to be already centered. Defaults to True.

n_iter: int The number of passes over the training data (aka epochs). Defaults to 5.

shuffle: bool Whether or not the training data should be shuffled after each epoch. Defaults to False.

verbose: integer, optional The verbosity level

n_jobs: integer, optional The number of CPUs to use to do the OVA (One Versus All, for multi-class problems) computation. -1 means 'all CPUs'. Defaults to 1.

Attributes

*coef_* : array, shape = [n_features] if n_classes == 2 else [n_classes, n_features]

Weights asigned to the features.

intercept_ [array, shape = [1] if n_classes == 2 else [n_classes]] Constants in decision function.

Examples

```
>>> import numpy as np
>>> from scikits.learn import linear_model
>>> X = np.array([[-1, -1], [-2, -1], [1, 1], [2, 1]])
>>> Y = np.array([1, 1, 2, 2])
>>> clf = linear_model.SGDClassifier()
>>> clf.fit(X, Y)
SGDClassifier(loss='hinge', n_jobs=1, shuffle=False, verbose=0, n_iter=5,
       fit_intercept=True, penalty='l2', rho=1.0, alpha=0.0001)
>>> print clf.predict([[-0.8, -1]])
[ 1.]
```

See also

LinearSVC, LogisticRegression

Full API documentation: SGDClassifierScikitsLearnNode

class mdp.nodes.**LinearModelCVScikitsLearnNode**

This node has been automatically generated by wrapping the scikits.learn.linear_model.coordinate_descen class from the scikits.learn library. The wrapped instance can be accessed through the scikits_alg attribute.

Full API documentation: LinearModelCVScikitsLearnNode

class mdp.nodes.**SVCScikitsLearnNode**

C-Support Vector Classification.

This node has been automatically generated by wrapping the scikits.learn.svm.libsvm.SVC class from the scikits.learn library. The wrapped instance can be accessed through the scikits_alg attribute.

Parameters

C [float, optional (default=1.0)] penalty parameter C of the error term.

kernel [string, optional] Specifies the kernel type to be used in the algorithm. one of 'linear', 'poly', 'rbf', 'sigmoid', 'precomputed'. If none is given 'rbf' will be used.

degree [int, optional] degree of kernel function is significant only in poly, rbf, sigmoid

gamma [float, optional] kernel coefficient for rbf and poly, by default 1/n_features will be taken.

coef0 [float, optional] independent term in kernel function. It is only significant in poly/sigmoid.

probability: boolean, optional (False by default) enable probability estimates. This must be enabled prior to calling prob_predict.

shrinking: boolean, optional wether to use the shrinking heuristic.

eps: float, optional precision for stopping criteria

cache_size: float, optional specify the size of the cache (in MB)

Attributes

support_ [array-like, shape = [n_SV]] Index of support vectors.

support_vectors_ [array-like, shape = [n_SV, n_features]] Support vectors.

n_support_ [array-like, dtype=int32, shape = [n_class]] number of support vector for each class.

dual_coef_ [array, shape = [n_class-1, n_SV]] Coefficients of the support vector in the decision function.

coef_ [array, shape = [n_class-1, n_features]] Weights asigned to the features (coefficients in the primal problem). This is only available in the case of linear kernel.

intercept_ [array, shape = [n_class * (n_class-1) / 2]] Constants in decision function.

Examples

```
>>> import numpy as np
>>> X = np.array([[-1, -1], [-2, -1], [1, 1], [2, 1]])
>>> y = np.array([1, 1, 2, 2])
>>> from scikits.learn.svm import SVC
>>> clf = SVC()
>>> clf.fit(X, y)
SVC(kernel='rbf', C=1.0, probability=False, degree=3, coef0=0.0, eps=0.001,
  cache_size=100.0, shrinking=True, gamma=0.25)
>>> print clf.predict([[-0.8, -1]])
[ 1.]
```

See also

SVR, LinearSVC

Full API documentation: SVCScikitsLearnNode

class mdp.nodes.**SelectFprScikitsLearnNode**

This node has been automatically generated by wrapping the scikits.learn.feature_selection.univariate_s class from the scikits.learn library. The wrapped instance can be accessed through the scikits_alg attribute.

Full API documentation: SelectFprScikitsLearnNode

class mdp.nodes.**ScalerScikitsLearnNode**

Object to standardize a dataset

This node has been automatically generated by wrapping the scikits.learn.preprocessing.Scaler class from the scikits.learn library. The wrapped instance can be accessed through the scikits_alg attribute.

It centers the dataset and optionaly scales to fix the variance to 1 for each feature

Full API documentation: ScalerScikitsLearnNode

class mdp.nodes.**ElasticNetScikitsLearnNode**

Linear Model trained with L1 and L2 prior as regularizer

This node has been automatically generated by wrapping the scikits.learn.linear_model.coordinate_descen class from the scikits.learn library. The wrapped instance can be accessed through the scikits_alg attribute.

rho=1 is the lasso penalty. Currently, rho <= 0.01 is not reliable, unless you supply your own sequence of alpha.

Parameters

alpha [float] Constant that multiplies the L1 term. Defaults to 1.0

rho [float] The ElasticNet mixing parameter, with 0 < rho <= 1.

coef_: ndarray of shape n_features The initial coeffients to warm-start the optimization

fit_intercept: bool Whether the intercept should be estimated or not. If False, the data is assumed to be already centered.

Notes

To avoid unnecessary memory duplication the X argument of the fit method should be directly passed as a fortran contiguous numpy array.

Full API documentation: ElasticNetScikitsLearnNode

class mdp.nodes.**LinearSVCScikitsLearnNode**

Linear Support Vector Classification, Sparse Version

This node has been automatically generated by wrapping the scikits.learn.svm.sparse.liblinear.LinearSVC class from the scikits.learn library. The wrapped instance can be accessed through the scikits_alg attribute.

Similar to SVC with parameter kernel='linear', but uses internally liblinear rather than libsvm, so it has more flexibility in the choice of penalties and loss functions and should be faster for huge datasets.

Parameters

loss [string, 'l1' or 'l2' (default 'l2')] Specifies the loss function. With 'l1' it is the standard SVM loss (a.k.a. hinge Loss) while with 'l2' it is the squared loss. (a.k.a. squared hinge Loss)

penalty [string, 'l1' or 'l2' (default 'l2')] Specifies the norm used in the penalization. The 'l2' penalty is the standard used in SVC. The 'l1' leads to `coef_` vectors that are sparse.

dual [bool, (default True)] Select the algorithm to either solve the dual or primal optimization problem.

Attributes

coef_ [array, shape = [n_features] if n_classes == 2 else [n_classes, n_features]] Wiehgiths asigned to the features (coefficients in the primal problem). This is only available in the case of linear kernel.

intercept_ [array, shape = [1] if n_classes == 2 else [n_classes]] constants in decision function

Notes

The underlying C implementation uses a random number generator to select features when fitting the model. It is thus not uncommon, to have slightly different results for the same input data. If that happens, try with a smaller eps parameter.

See also

SVC

References

LIBLINEAR – A Library for Large Linear Classification http://www.csie.ntu.edu.tw/~cjlin/liblinear/

Full API documentation: LinearSVCScikitsLearnNode

class `mdp.nodes.`**`BinarizerScikitsLearnNode`**

This node has been automatically generated by wrapping the `scikits.learn.preprocessing.Binarizer` class from the `scikits.learn` library. The wrapped instance can be accessed through the `scikits_alg` attribute.

Full API documentation: BinarizerScikitsLearnNode

class `mdp.nodes.`**`FastICAScikitsLearnNode`**

FastICA; a fast algorithm for Independent Component Analysis

This node has been automatically generated by wrapping the `scikits.learn.fastica.FastICA` class from the `scikits.learn` library. The wrapped instance can be accessed through the `scikits_alg` attribute.

Parameters

n_components [int, optional] Number of components to use. If none is passed, all are used.

algorithm: {'parallel', 'deflation'} Apply parallel or deflational algorithm for FastICA

whiten: boolean, optional If whiten is false, the data is already considered to be whitened, and no whitening is performed.

fun: {'logcosh', 'exp', or 'cube'}, or a callable The non-linear function used in the FastICA loop to approximate negentropy. If a function is passed, it derivative should be passed as the 'fun_prime' argument.

fun_prime: None or a callable The derivative of the non-linearity used.

max_iter [int, optional] Maximum number of iterations during fit

tol [float, optional] Tolerance on update at each iteration

w_init: None of an (n_components, n_components) ndarray The mixing matrix to be used to initialize the algorithm.

Attributes

`unmixing_matrix_` : 2D array, [n_components, n_samples]

Methods

get_mixing_matrix() :

- Returns an estimate of the mixing matrix

Notes

Implementation based on :

1.Hyvarinen and E. Oja, Independent Component Analysis:

Algorithms and Applications, Neural Networks, 13(4-5), 2000, pp. 411-430

Full API documentation: FastICAScikitsLearnNode

class `mdp.nodes.`**`NormalizerScikitsLearnNode`**

This node has been automatically generated by wrapping the `scikits.learn.preprocessing.Normalizer` class from the `scikits.learn` library. The wrapped instance can be accessed through the `scikits_alg` attribute.

Full API documentation: NormalizerScikitsLearnNode

class `mdp.nodes.`**`LassoScikitsLearnNode`**

Linear Model trained with L1 prior as regularizer (aka the Lasso)

This node has been automatically generated by wrapping the `scikits.learn.linear_model.coordinate_descer` class from the `scikits.learn` library. The wrapped instance can be accessed through the `scikits_alg` attribute.

Technically the Lasso model is optimizing the same objective function as the Elastic Net with rho=1.0 (no L2 penalty).

Parameters

alpha [float, optional] Constant that multiplies the L1 term. Defaults to 1.0

fit_intercept [boolean] whether to calculate the intercept for this model. If set to false, no intercept will be used in calculations (e.g. data is expected to be already centered).

Attributes

coef_ [array, shape = [n_features]] parameter vector (w in the fomulation formula)

intercept_ [float] independent term in decision function.

Examples

```
>>> from scikits.learn import linear_model
>>> clf = linear_model.Lasso(alpha=0.1)
>>> clf.fit([[0,0], [1, 1], [2, 2]], [0, 1, 2])
Lasso(alpha=0.1, fit_intercept=True)
>>> print clf.coef_
[ 0.85  0.  ]
>>> print clf.intercept_
0.15
```

See also

LassoLARS

Notes

The algorithm used to fit the model is coordinate descent.

To avoid unnecessary memory duplication the X argument of the fit method should be directly passed as a fortran contiguous numpy array.

Full API documentation: LassoScikitsLearnNode

class mdp.nodes.**SelectFdrScikitsLearnNode**
Filter : Select the p-values corresponding to an estimated false

This node has been automatically generated by wrapping the scikits.learn.feature_selection.univariate_s class from the scikits.learn library. The wrapped instance can be accessed through the scikits_alg attribute.

Full API documentation: SelectFdrScikitsLearnNode

class mdp.nodes.**SparseBaseLibLinearScikitsLearnNode**

class mdp.nodes.**LassoLARSScikitsLearnNode**
Lasso model fit with Least Angle Regression a.k.a. LARS

This node has been automatically generated by wrapping the scikits.learn.linear_model.least_angle.Lasso class from the scikits.learn library. The wrapped instance can be accessed through the scikits_alg attribute.

It is a Linear Model trained with an L1 prior as regularizer. lasso).

Parameters

alpha [float, optional] Constant that multiplies the L1 term. Defaults to 1.0

fit_intercept [boolean] whether to calculate the intercept for this model. If set to false, no intercept will be used in calculations (e.g. data is expected to be already centered).

Attributes

coef_ [array, shape = [n_features]] parameter vector (w in the fomulation formula)

intercept_ [float] independent term in decision function.

Examples

```
>>> from scikits.learn import linear_model
>>> clf = linear_model.LassoLARS(alpha=0.1)
>>> clf.fit([[-1,1], [0, 0], [1, 1]], [-1, 0, -1])
LassoLARS(alpha=0.1, verbose=False, fit_intercept=True)
>>> print clf.coef_
[ 0.          0.08350342]
```

References

http://en.wikipedia.org/wiki/Least_angle_regression

See also

lars_path, Lasso

Full API documentation: LassoLARSScikitsLearnNode

class mdp.nodes.**RandomizedPCAScikitsLearnNode**
Principal component analysis (PCA) using randomized SVD

This node has been automatically generated by wrapping the scikits.learn.pca.RandomizedPCA class from the scikits.learn library. The wrapped instance can be accessed through the scikits_alg attribute.

Linear dimensionality reduction using approximated Singular Value Decomposition of the data and keeping only the most significant singular vectors to project the data to a lower dimensional space.

This implementation uses a randomized SVD implementation and can handle both scipy.sparse and numpy dense arrays as input.

Parameters

X: array-like or scipy.sparse matrix, shape (n_samples, n_features) Training vector, where n_samples in the number of samples and n_features is the number of features.

n_components: int Maximum number of components to keep: default is 50.

copy: bool If False, data passed to fit are overwritten

iterated_power: int, optional Number of iteration for the power method. 3 by default.

whiten: bool, optional When True (False by default) the `components_` vectors are divided by the singular values to ensure uncorrelated outputs with unit component-wise variances.

Whitening will remove some information from the transformed signal (the relative variance scales of the components) but can sometime improve the predictive accuracy of the downstream estimators by making there data respect some hard-wired assumptions.

Attributes

components_: array, [n_features, n_components] Components with maximum variance.

explained_variance_ratio_: array, [n_components] Percentage of variance explained by each of the selected components. k is not set then all components are stored and the sum of explained variances is equal to 1.0

References

Finding structure with randomness: Stochastic algorithms for constructing approximate matrix decompositions Halko, et al., 2009 (arXiv:909)

A randomized algorithm for the decomposition of matrices Per-Gunnar Martinsson, Vladimir Rokhlin and Mark Tygert

Examples

```
>>> import numpy as np
>>> from scikits.learn.pca import RandomizedPCA
>>> X = np.array([[-1, -1], [-2, -1], [-3, -2], [1, 1], [2, 1], [3, 2]])
>>> pca = RandomizedPCA(n_components=2)
>>> pca.fit(X)
RandomizedPCA(copy=True, n_components=2, iterated_power=3, whiten=False)
>>> print pca.explained_variance_ratio_
[ 0.99244289  0.00755711]
```

See also

PCA ProbabilisticPCA

Full API documentation: RandomizedPCAScikitsLearnNode

class `mdp.nodes.`**`NuSVRScikitsLearnNode`**

NuSVR for sparse matrices (csr)

This node has been automatically generated by wrapping the `scikits.learn.svm.sparse.libsvm.NuSVR` class from the `scikits.learn` library. The wrapped instance can be accessed through the `scikits_alg` attribute.

See `scikits.learn.svm.NuSVC` for a complete list of parameters

Notes

For best results, this accepts a matrix in csr format (scipy.sparse.csr), but should be able to convert from any array-like object (including other sparse representations).

Full API documentation: NuSVRScikitsLearnNode

class `mdp.nodes.`**`ElasticNetCVScikitsLearnNode`**

Elastic Net model with iterative fitting along a regularization path

This node has been automatically generated by wrapping the `scikits.learn.linear_model.coordinate_descer` class from the `scikits.learn` library. The wrapped instance can be accessed through the `scikits_alg` attribute.

The best model is selected by cross-validation.

Parameters

rho [float, optional] float between 0 and 1 passed to ElasticNet (scaling between l1 and l2 penalties)

eps [float, optional] Length of the path. eps=1e-3 means that alpha_min / alpha_max = 1e-3.

n_alphas [int, optional] Number of alphas along the regularization path

alphas [numpy array, optional] List of alphas where to compute the models. If None alphas are set automatically

Notes

See examples/linear_model/lasso_path_with_crossvalidation.py for an example.

To avoid unnecessary memory duplication the X argument of the fit method should be directly passed as a fortran contiguous numpy array.

Full API documentation: ElasticNetCVScikitsLearnNode

class mdp.nodes.**LDAScikitsLearnNode**

Linear Discriminant Analysis (LDA)

This node has been automatically generated by wrapping the `scikits.learn.lda.LDA` class from the `scikits.learn` library. The wrapped instance can be accessed through the `scikits_alg` attribute.

Parameters

X [array-like, shape = [n_samples, n_features]] Training vector, where n_samples in the number of samples and n_features is the number of features.

y [array, shape = [n_samples]] Target vector relative to X

priors [array, optional, shape = [n_classes]] Priors on classes

Attributes

means_ [array-like, shape = [n_classes, n_features]] Class means

xbar_ [float, shape = [n_features]] Over all mean

priors_ [array-like, shape = [n_classes]] Class priors (sum to 1)

covariance_ [array-like, shape = [n_features, n_features]] Covariance matrix (shared by all classes)

Examples

```
>>> import numpy as np
>>> from scikits.learn.lda import LDA
>>> X = np.array([[-1, -1], [-2, -1], [-3, -2], [1, 1], [2, 1], [3, 2]])
>>> y = np.array([1, 1, 1, 2, 2, 2])
>>> clf = LDA()
>>> clf.fit(X, y)
LDA(priors=None)
>>> print clf.predict([[-0.8, -1]])
[1]
```

See also

QDA

Full API documentation: LDAScikitsLearnNode

class mdp.nodes.**SelectFweScikitsLearnNode**

This node has been automatically generated by wrapping the `scikits.learn.feature_selection.univariate_s` class from the `scikits.learn` library. The wrapped instance can be accessed through the `scikits_alg` attribute.

Full API documentation: SelectFweScikitsLearnNode

class mdp.nodes.**TfidfTransformerScikitsLearnNode**
Transform a count matrix to a TF or TF-IDF representation

This node has been automatically generated by wrapping the `scikits.learn.feature_extraction.text.dense.` class from the `scikits.learn` library. The wrapped instance can be accessed through the `scikits_alg` attribute.

TF means term-frequency while TF-IDF means term-frequency times inverse document-frequency:

http://en.wikipedia.org/wiki/TF-IDF

The goal of using TF-IDF instead of the raw frequencies of occurrence of a token in a given document is to scale down the impact of tokens that occur very frequently in a given corpus and that are hence empirically less informative than feature that occur in a small fraction of the training corpus.

TF-IDF can be seen as a smooth alternative to the stop words filtering.

Parameters

use_tf: boolean enable term-frequency normalization

use_idf: boolean enable inverse-document-frequency reweighting

Full API documentation: TfidfTransformerScikitsLearnNode

class mdp.nodes.**PCAScikitsLearnNode**
Principal component analysis (PCA)

This node has been automatically generated by wrapping the `scikits.learn.pca.PCA` class from the `scikits.learn` library. The wrapped instance can be accessed through the `scikits_alg` attribute.

Linear dimensionality reduction using Singular Value Decomposition of the data and keeping only the most significant singular vectors to project the data to a lower dimensional space.

This implementation uses the scipy.linalg implementation of the singular value decomposition. It only works for dense arrays and is not scalable to large dimensional data.

The time complexity of this implementation is O(n ** 3) assuming n ~ n_samples ~ n_features.

Parameters

X: array-like, shape (n_samples, n_features) Training vector, where n_samples in the number of samples and n_features is the number of features.

n_components: int, none or string Number of components to keep. if n_components is not set all components are kept:

- n_components == min(n_samples, n_features)

if n_components == 'mle', Minka's MLE is used to guess the dimension

copy: bool If False, data passed to fit are overwritten

whiten: bool, optional When True (False by default) the `components_` vectors are divided by n_samples times singular values to ensure uncorrelated outputs with unit component-wise variances.

Whitening will remove some information from the transformed signal (the relative variance scales of the components) but can sometime improve the predictive accuracy of the downstream estimators by making there data respect some hard-wired assumptions.

Attributes

components_: array, [n_features, n_components] Components with maximum variance.

explained_variance_ratio_: array, [n_components] Percentage of variance explained by each of the selected components. k is not set then all components are stored and the sum of explained variances is equal to 1.0

Notes

For n_components='mle', this class uses the method of Thomas P. Minka:

Automatic Choice of Dimensionality for PCA. NIPS 2000: 598-604

Examples

```
>>> import numpy as np
>>> from scikits.learn.pca import PCA
>>> X = np.array([[-1, -1], [-2, -1], [-3, -2], [1, 1], [2, 1], [3, 2]])
>>> pca = PCA(n_components=2)
>>> pca.fit(X)
PCA(copy=True, n_components=2, whiten=False)
>>> print pca.explained_variance_ratio_
[ 0.99244289  0.00755711]
```

See also

ProbabilisticPCA RandomizedPCA

Full API documentation: PCAScikitsLearnNode

class mdp.nodes.**NeighborsBarycenterScikitsLearnNode**

Regression based on k-Nearest Neighbor Algorithm.

This node has been automatically generated by wrapping the `scikits.learn.neighbors.NeighborsBarycenter` class from the `scikits.learn` library. The wrapped instance can be accessed through the `scikits_alg` attribute.

The target is predicted by local interpolation of the targets associated of the k-Nearest Neighbors in the training set. The interpolation weights correspond to barycenter weights.

Parameters

X [array-like, shape (n_samples, n_features)] The data points to be indexed. This array is not copied, and so modifying this data will result in bogus results.

y [array] An array representing labels for the data (only arrays of integers are supported).

n_neighbors [int] default number of neighbors.

window_size [int] Window size passed to BallTree

Examples

```
>>> X = [[0], [1], [2], [3]]
>>> y = [0, 0, 1, 1]
>>> from scikits.learn.neighbors import NeighborsBarycenter
>>> neigh = NeighborsBarycenter(n_neighbors=2)
>>> neigh.fit(X, y)
NeighborsBarycenter(n_neighbors=2, window_size=1)
>>> print neigh.predict([[1.5]])
[ 0.5]
```

Notes

http://en.wikipedia.org/wiki/K-nearest_neighbor_algorithm

Full API documentation: NeighborsBarycenterScikitsLearnNode

class mdp.nodes.**GaussianProcessScikitsLearnNode**

The Gaussian Process model class.

This node has been automatically generated by wrapping the `scikits.learn.gaussian_process.gaussian_proc` class from the `scikits.learn` library. The wrapped instance can be accessed through the `scikits_alg` attribute.

Parameters

regr [string or callable, optional] A regression function returning an array of outputs of the linear regression functional basis. The number of observations n_samples should be greater than the size p of this basis. Default assumes a simple constant regression trend. Here is the list of built-in regression models:

- 'constant', 'linear', 'quadratic'

corr [string or callable, optional] A stationary autocorrelation function returning the autocorrelation between two points x and x'. Default assumes a squared-exponential autocorrelation model. Here is the list of built-in correlation models:

- 'absolute_exponential', 'squared_exponential',
- 'generalized_exponential', 'cubic', 'linear'

beta0 [double array_like, optional] The regression weight vector to perform Ordinary Kriging (OK). Default assumes Universal Kriging (UK) so that the vector beta of regression weights is estimated using the maximum likelihood principle.

storage_mode [string, optional] A string specifying whether the Cholesky decomposition of the correlation matrix should be stored in the class (storage_mode = 'full') or not (storage_mode = 'light'). Default assumes storage_mode = 'full', so that the Cholesky decomposition of the correlation matrix is stored. This might be a useful parameter when one is not interested in the MSE and only plan to estimate the BLUP, for which the correlation matrix is not required.

verbose [boolean, optional] A boolean specifying the verbose level. Default is verbose = False.

theta0 [double array_like, optional] An array with shape (n_features,) or (1,). The parameters in the autocorrelation model. If thetaL and thetaU are also specified, theta0 is considered as the starting point for the maximum likelihood rstimation of the best set of parameters. Default assumes isotropic autocorrelation model with theta0 = 1e-1.

thetaL [double array_like, optional] An array with shape matching theta0's. Lower bound on the autocorrelation parameters for maximum likelihood estimation. Default is None, so that it skips maximum likelihood estimation and it uses theta0.

thetaU [double array_like, optional] An array with shape matching theta0's. Upper bound on the autocorrelation parameters for maximum likelihood estimation. Default is None, so that it skips maximum likelihood estimation and it uses theta0.

normalize [boolean, optional] Input X and observations y are centered and reduced wrt means and standard deviations estimated from the n_samples observations provided. Default is normalize = True so that data is normalized to ease maximum likelihood estimation.

nugget [double, optional] Introduce a nugget effect to allow smooth predictions from noisy data. Default assumes a nugget close to machine precision for the sake of robustness (nugget = 10. * MACHINE_EPSILON).

optimizer [string, optional] A string specifying the optimization algorithm to be used. Default uses 'fmin_cobyla' algorithm from scipy.optimize. Here is the list of available optimizers:

- 'fmin_cobyla', 'Welch'

'Welch' optimizer is dued to Welch et al., see reference [2]. It consists in iterating over several one-dimensional optimizations instead of running one single multi-dimensional optimization.

random_start [int, optional] The number of times the Maximum Likelihood Estimation should be performed from a random starting point. The first MLE always uses the specified starting point (theta0), the next starting points are picked at random according to an exponential distribution (log-uniform on [thetaL, thetaU]). Default does not use random starting point (random_start = 1).

Example

```
>>> import numpy as np
>>> from scikits.learn.gaussian_process import GaussianProcess
>>> X = np.atleast_2d([1., 3., 5., 6., 7., 8.]).T
>>> y = (X * np.sin(X)).ravel()
>>> gp = GaussianProcess(theta0=0.1, thetaL=.001, thetaU=1.)
>>> gp.fit(X, y)
GaussianProcess(normalize=True, ...)
```

Implementation details

The presentation implementation is based on a translation of the DACE Matlab toolbox, see reference [1].

References

[1] H.B. Nielsen, S.N. Lophaven, H. B. Nielsen and J. Sondergaard (2002). DACE - A MATLAB Kriging Toolbox. http://www2.imm.dtu.dk/~hbn/dace/dace.pdf

[2] W.J. Welch, R.J. Buck, J. Sacks, H.P. Wynn, T.J. Mitchell, and M.D. Morris (1992). Screening, predicting, and computer experiments. Technometrics, 34(1) 15–25. http://www.jstor.org/pss/1269548

Full API documentation: GaussianProcessScikitsLearnNode

class mdp.nodes.**LassoCVScikitsLearnNode**

Lasso linear model with iterative fitting along a regularization path

This node has been automatically generated by wrapping the `scikits.learn.linear_model.coordinate_descer` class from the `scikits.learn` library. The wrapped instance can be accessed through the `scikits_alg` attribute.

The best model is selected by cross-validation.

Parameters

eps [float, optional] Length of the path. eps=1e-3 means that alpha_min / alpha_max = 1e-3.

n_alphas [int, optional] Number of alphas along the regularization path

alphas [numpy array, optional] List of alphas where to compute the models. If None alphas are set automatically

Notes

See examples/linear_model/lasso_path_with_crossvalidation.py for an example.

To avoid unnecessary memory duplication the X argument of the fit method should be directly passed as a fortran contiguous numpy array.

Full API documentation: LassoCVScikitsLearnNode

class mdp.nodes.**BayesianRidgeScikitsLearnNode**

Bayesian ridge regression

This node has been automatically generated by wrapping the `scikits.learn.linear_model.bayes.BayesianRi` class from the `scikits.learn` library. The wrapped instance can be accessed through the `scikits_alg` attribute.

Fit a Bayesian ridge model and optimize the regularization parameters lambda (precision of the weights) and alpha (precision of the noise).

Parameters

X [array, shape = (n_samples, n_features)] Training vectors.

y [array, shape = (length)] Target values for training vectors

n_iter [int, optional] Maximum number of interations. Default is 300.

eps [float, optional] Stop the algorithm if w has converged. Default is 1.e-3.

alpha_1 [float, optional] Hyper-parameter : shape parameter for the Gamma distribution prior over the alpha parameter. Default is 1.e-6

alpha_2 [float, optional] Hyper-parameter : inverse scale parameter (rate parameter) for the Gamma distribution prior over the alpha parameter. Default is 1.e-6.

lambda_1 [float, optional] Hyper-parameter : shape parameter for the Gamma distribution prior over the lambda parameter. Default is 1.e-6.

lambda_2 [float, optional] Hyper-parameter : inverse scale parameter (rate parameter) for the Gamma distribution prior over the lambda parameter. Default is 1.e-6

compute_score [boolean, optional] If True, compute the objective function at each step of the model. Default is False

fit_intercept [boolean, optional] wether to calculate the intercept for this model. If set to false, no intercept will be used in calculations (e.g. data is expected to be already centered). Default is True.

Attributes

coef_ [array, shape = (n_features)] Coefficients of the regression model (mean of distribution)

alpha_ [float] estimated precision of the noise.

lambda_ [array, shape = (n_features)] estimated precisions of the weights.

scores_ [float] if computed, value of the objective function (to be maximized)

Methods

fit(X, y) [self] Fit the model

predict(X) [array] Predict using the model.

Examples

```
>>> from scikits.learn import linear_model
>>> clf = linear_model.BayesianRidge()
>>> clf.fit([[0,0], [1, 1], [2, 2]], [0, 1, 2])
BayesianRidge(n_iter=300, verbose=False, lambda_1=1e-06, lambda_2=1e-06,
       fit_intercept=True, eps=0.001, alpha_2=1e-06, alpha_1=1e-06,
       compute_score=False)
>>> clf.predict([[1, 1]])
array([ 1.])
```

Notes

See examples/linear_model/plot_bayesian_ridge.py for an example.

Full API documentation: BayesianRidgeScikitsLearnNode

class mdp.nodes.**OneClassSVMScikitsLearnNode**

Unsupervised Outliers Detection.

This node has been automatically generated by wrapping the `scikits.learn.svm.libsvm.OneClassSVM` class from the `scikits.learn` library. The wrapped instance can be accessed through the `scikits_alg` attribute.

Estimate the support of a high-dimensional distribution.

Parameters

kernel [string, optional] Specifies the kernel type to be used in the algorithm. Can be one of 'linear', 'poly', 'rbf', 'sigmoid', 'precomputed'. If none is given 'rbf' will be used.

nu [float, optional] An upper bound on the fraction of training errors and a lower bound of the fraction of support vectors. Should be in the interval (0, 1]. By default 0.5 will be taken.

degree [int, optional] Degree of kernel function. Significant only in poly, rbf, sigmoid.

gamma [float, optional] kernel coefficient for rbf and poly, by default 1/n_features will be taken.

coef0 [float, optional] Independent term in kernel function. It is only significant in poly/sigmoid.

eps: float, optional precision for stopping criteria

shrinking: boolean, optional wether to use the shrinking heuristic.

cache_size: float, optional specify the size of the cache (in MB)

Attributes

support_ [array-like, shape = [n_SV]] Index of support vectors.

support_vectors_ [array-like, shape = [nSV, n_features]] Support vectors.

dual_coef_ [array, shape = [n_classes-1, n_SV]] Coefficient of the support vector in the decision function.

coef_ [array, shape = [n_classes-1, n_features]] Weights asigned to the features (coefficients in the primal problem). This is only available in the case of linear kernel.

intercept_ [array, shape = [n_classes-1]] Constants in decision function.

Full API documentation: OneClassSVMScikitsLearnNode

class `mdp.nodes.`**`GMMScikitsLearnNode`**

Gaussian Mixture Model

This node has been automatically generated by wrapping the `scikits.learn.mixture.GMM` class from the `scikits.learn` library. The wrapped instance can be accessed through the `scikits_alg` attribute.

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 GMM distribution.

Initializes parameters such that every mixture component has zero mean and identity covariance.

Parameters

n_states [int] Number of mixture components.

cvtype [string (read-only)] String describing the type of covariance parameters to use. Must be one of 'spherical', 'tied', 'diag', 'full'. Defaults to 'diag'.

Attributes

cvtype [string (read-only)] String describing the type of covariance parameters used by the GMM. Must be one of 'spherical', 'tied', 'diag', 'full'.

n_features [int] Dimensionality of the Gaussians.

n_states [int (read-only)] Number of mixture components.

weights [array, shape (*n_states*,)] Mixing weights for each mixture component.

means [array, shape (*n_states*, *n_features*)] Mean parameters for each mixture component.

covars [array] Covariance parameters for each mixture component. The shape depends on *cvtype*:

- (*n_states*,) if 'spherical',
- (*n_features*, *n_features*) if 'tied',
- (*n_states*, *n_features*) if 'diag',
- (*n_states*, *n_features*, *n_features*) if 'full'

Methods

decode(X) Find most likely mixture components for each point in *X*.

eval(X) Compute the log likelihood of *X* under the model and the posterior distribution over mixture components.

fit(X) Estimate model parameters from *X* using the EM algorithm.

predict(X) Like decode, find most likely mixtures components for each observation in *X*.

rvs(n=1) Generate *n* samples from the model.

score(X) Compute the log likelihood of *X* under the model.

Examples

```
>>> import numpy as np
>>> from scikits.learn import mixture
>>> g = mixture.GMM(n_states=2)

>>> # Generate random observations with two modes centered on 0
>>> # and 10 to use for training.
>>> np.random.seed(0)
>>> obs = np.concatenate((np.random.randn(100, 1),
...                       10 + np.random.randn(300, 1)))
>>> g.fit(obs)
GMM(cvtype='diag', n_states=2)
>>> g.weights
array([ 0.25,  0.75])
>>> g.means
array([[ 0.05980802],
       [ 9.94199467]])
>>> g.covars
[array([[ 1.01682662]]), array([[ 0.96080513]])]
>>> np.round(g.weights, 2)
array([ 0.25,  0.75])
>>> np.round(g.means, 2)
array([[ 0.06],
       [ 9.94]])
>>> np.round(g.covars, 2)
...
array([[[ 1.02]],
       [[ 0.96]]])
>>> g.predict([[0], [2], [9], [10]])
array([0, 0, 1, 1])
>>> np.round(g.score([[0], [2], [9], [10]]), 2)
array([-2.32, -4.16, -1.65, -1.19])

>>> # Refit the model on new data (initial parameters remain the
>>> # same), this time with an even split between the two modes.
>>> g.fit(20 * [[0]] +  20 * [[10]])
GMM(cvtype='diag', n_states=2)
>>> np.round(g.weights, 2)
array([ 0.5,  0.5])
```

Full API documentation: GMMScikitsLearnNode

class mdp.nodes.**GMMHMMScikitsLearnNode**

Hidden Markov Model with Gaussin mixture emissions

This node has been automatically generated by wrapping the `scikits.learn.hmm.GMMHMM` class from the `scikits.learn` library. The wrapped instance can be accessed through the `scikits_alg` attribute.

Attributes

n_states [int (read-only)] Number of states in the model.

transmat [array, shape (*n_states*, *n_states*)] Matrix of transition probabilities between states.

startprob [array, shape ('n_states',)] Initial state occupation distribution.

gmms: array of GMM objects, length 'n_states' GMM emission distributions for each state

Methods

eval(X) Compute the log likelihood of *X* under the HMM.

decode(X) Find most likely state sequence for each point in *X* using the Viterbi algorithm.

rvs(n=1) Generate *n* samples from the HMM.

init(X) Initialize HMM parameters from *X*.

fit(X) Estimate HMM parameters from *X* using the Baum-Welch algorithm.

predict(X) Like decode, find most likely state sequence corresponding to *X*.

score(X) Compute the log likelihood of *X* under the model.

Examples

```
>>> from scikits.learn.hmm import GMMHMM
>>> GMMHMM(n_states=2, n_mix=10, cvtype='diag')
...
GMMHMM(n_mix=10, cvtype='diag', n_states=2, startprob_prior=1.0,
    startprob=array([ 0.5,  0.5]),
    transmat=array([[ 0.5,  0.5],
       [ 0.5,  0.5]]),
    transmat_prior=1.0,
    gmms=[GMM(cvtype='diag', n_states=10), GMM(cvtype='diag',
          n_states=10)])
```

See Also

GaussianHMM : HMM with Gaussian emissions

Full API documentation: GMMHMMScikitsLearnNode

class mdp.nodes.**ARDRegressionScikitsLearnNode**

Bayesian ARD regression.

This node has been automatically generated by wrapping the `scikits.learn.linear_model.bayes.ARDRegressi` class from the `scikits.learn` library. The wrapped instance can be accessed through the `scikits_alg` attribute.

Fit the weights of a regression model, using an ARD prior. The weights of the regression model are assumed to be in Gaussian distributions. Also estimate the parameters lambda (precisions of the distributions of the weights) and alpha (precision of the distribution of the noise). The estimation is done by an iterative procedures (Evidence Maximization)

Parameters

X [array, shape = (n_samples, n_features)] Training vectors.

y [array, shape = (n_samples)] Target values for training vectors

n_iter [int, optional] Maximum number of interations. Default is 300

eps [float, optional] Stop the algorithm if w has converged. Default is 1.e-3.

alpha_1 [float, optional] Hyper-parameter : shape parameter for the Gamma distribution prior over the alpha parameter. Default is 1.e-6.

alpha_2 [float, optional] Hyper-parameter : inverse scale parameter (rate parameter) for the Gamma distribution prior over the alpha parameter. Default is 1.e-6.

lambda_1 [float, optional] Hyper-parameter : shape parameter for the Gamma distribution prior over the lambda parameter. Default is 1.e-6.

lambda_2 [float, optional] Hyper-parameter : inverse scale parameter (rate parameter) for the Gamma distribution prior over the lambda parameter. Default is 1.e-6.

compute_score [boolean, optional] If True, compute the objective function at each step of the model. Default is False.

threshold_lambda [float, optional] threshold for removing (pruning) weights with high precision from the computation. Default is 1.e+4.

fit_intercept [boolean, optional] wether to calculate the intercept for this model. If set to false, no intercept will be used in calculations (e.g. data is expected to be already centered). Default is True.

verbose [boolean, optional] Verbose mode when fitting the model. Default is False.

Attributes

coef_ [array, shape = (n_features)] Coefficients of the regression model (mean of distribution)

alpha_ [float] estimated precision of the noise.

lambda_ [array, shape = (n_features)] estimated precisions of the weights.

sigma_ [array, shape = (n_features, n_features)] estimated variance-covariance matrix of the weights

scores_ [float] if computed, value of the objective function (to be maximized)

Methods

fit(X, y) [self] Fit the model

predict(X) [array] Predict using the model.

Examples

```
>>> from scikits.learn import linear_model
>>> clf = linear_model.ARDRegression()
>>> clf.fit([[0,0], [1, 1], [2, 2]], [0, 1, 2])
ARDRegression(n_iter=300, verbose=False, lambda_1=1e-06, lambda_2=1e-06,
       fit_intercept=True, eps=0.001, threshold_lambda=10000.0,
       alpha_2=1e-06, alpha_1=1e-06, compute_score=False)
>>> clf.predict([[1, 1]])
array([ 1.])
```

Notes

See examples/linear_model/plot_ard.py for an example.

Full API documentation: ARDRegressionScikitsLearnNode

class mdp.nodes.**GenericUnivariateSelectScikitsLearnNode**

class mdp.nodes.**CountVectorizerScikitsLearnNode**

Convert a collection of raw documents to a matrix of token counts

This node has been automatically generated by wrapping the `scikits.learn.feature_extraction.text.dense` class from the `scikits.learn` library. The wrapped instance can be accessed through the `scikits_alg` attribute.

This implementation produces a dense representation of the counts using a numpy array.

If you do not provide an a-priori dictionary and you do not use an analyzer that does some kind of feature selection then the number of features (the vocabulary size found by analysing the data) might be very large and the count vectors might not fit in memory.

For this case it is either recommended to use the sparse.CountVectorizer variant of this class or a HashingVectorizer that will reduce the dimensionality to an arbitrary number by using random projection.

Parameters

analyzer: WordNGramAnalyzer or CharNGramAnalyzer, optional

vocabulary: dict, optional A dictionary where keys are tokens and values are indices in the matrix. This is useful in order to fix the vocabulary in advance.

dtype: type, optional Type of the matrix returned by fit_transform() or transform().

Full API documentation: CountVectorizerScikitsLearnNode

class mdp.nodes.**RidgeScikitsLearnNode**

Ridge regression.

This node has been automatically generated by wrapping the `scikits.learn.linear_model.ridge.Ridge` class from the `scikits.learn` library. The wrapped instance can be accessed through the `scikits_alg` attribute.

Parameters

alpha [float] Small positive values of alpha improve the coditioning of the problem and reduce the variance of the estimates.

fit_intercept [boolean] wether to calculate the intercept for this model. If set to false, no intercept will be used in calculations (e.g. data is expected to be already centered).

Examples

```
>>> from scikits.learn.linear_model import Ridge
>>> import numpy as np
>>> n_samples, n_features = 10, 5
>>> np.random.seed(0)
>>> y = np.random.randn(n_samples)
>>> X = np.random.randn(n_samples, n_features)
>>> clf = Ridge(alpha=1.0)
>>> clf.fit(X, y)
Ridge(alpha=1.0, fit_intercept=True)
```

Full API documentation: RidgeScikitsLearnNode

class mdp.nodes.**MultinomialHMMScikitsLearnNode**

Hidden Markov Model with multinomial (discrete) emissions

This node has been automatically generated by wrapping the `scikits.learn.hmm.MultinomialHMM` class from the `scikits.learn` library. The wrapped instance can be accessed through the `scikits_alg` attribute.

Attributes

n_states [int (read-only)] Number of states in the model.

n_symbols [int] Number of possible symbols emitted by the model (in the observations).

transmat [array, shape (*n_states*, *n_states*)] Matrix of transition probabilities between states.

startprob [array, shape (‘n_states‘,)] Initial state occupation distribution.

emissionprob: array, shape (‘n_states‘, ‘n_symbols‘) Probability of emitting a given symbol when in each state.

Methods

eval(X) Compute the log likelihood of *X* under the HMM.

decode(X) Find most likely state sequence for each point in *X* using the Viterbi algorithm.

rvs(n=1) Generate *n* samples from the HMM.

init(X) Initialize HMM parameters from *X*.

fit(X) Estimate HMM parameters from *X* using the Baum-Welch algorithm.

predict(X) Like decode, find most likely state sequence corresponding to *X*.

score(X) Compute the log likelihood of *X* under the model.

Examples

```
>>> from scikits.learn.hmm import MultinomialHMM
>>> MultinomialHMM(n_states=2)
...
MultinomialHMM(transmat=array([[ 0.5,  0.5],
       [ 0.5,  0.5]]),
        startprob_prior=1.0, n_states=2, startprob=array([ 0.5,  0.5]),
       transmat_prior=1.0)
```

See Also

GaussianHMM : HMM with Gaussian emissions

Full API documentation: MultinomialHMMScikitsLearnNode

class mdp.nodes.**LogisticRegressionScikitsLearnNode**
Logistic Regression.

This node has been automatically generated by wrapping the scikits.learn.linear_model.logistic.Logistic class from the scikits.learn library. The wrapped instance can be accessed through the scikits_alg attribute.

Implements L1 and L2 regularized logistic regression.

Parameters

penalty [string, 'l1' or 'l2'] Used to specify the norm used in the penalization

dual [boolean] Dual or primal formulation. Dual formulation is only implemented for l2 penalty.

C [float] Specifies the strength of the regularization. The smaller it is the bigger in the regularization.

fit_intercept [bool, default: True] Specifies if a constant (a.k.a. bias or intercept) should be added the decision function

Attributes

coef_ [array, shape = [n_classes-1, n_features]] Coefficient of the features in the decision function.

intercept_ [array, shape = [n_classes-1]] intercept (a.k.a. bias) added to the decision function. It is available only when parameter intercept is set to True

See also

LinearSVC

Notes

The underlying C implementation uses a random number generator to select features when fitting the model. It is thus not uncommon, to have slightly different results for the same input data. If that happens, try with a smaller eps parameter.

References

LIBLINEAR – A Library for Large Linear Classification http://www.csie.ntu.edu.tw/~cjlin/liblinear/

Full API documentation: LogisticRegressionScikitsLearnNode

class mdp.nodes.**SVRScikitsLearnNode**
Support Vector Regression.

This node has been automatically generated by wrapping the scikits.learn.svm.libsvm.SVR class from the scikits.learn library. The wrapped instance can be accessed through the scikits_alg attribute.

Parameters

nu [float, optional] An upper bound on the fraction of training errors and a lower bound of the fraction of support vectors. Should be in the interval (0, 1]. By default 0.5 will be taken. Only available if impl='nu_svc'

kernel [string, optional] Specifies the kernel type to be used in the algorithm. one of 'linear', 'poly', 'rbf', 'sigmoid', 'precomputed'. If none is given 'rbf' will be used.

p [float] epsilon in the epsilon-SVR model.

degree [int, optional] degree of kernel function is significant only in poly, rbf, sigmoid

gamma [float, optional] kernel coefficient for rbf and poly, by default 1/n_features will be taken.

C [float, optional (default=1.0)] penalty parameter C of the error term.

probability: boolean, optional (False by default) enable probability estimates. This must be enabled prior to calling prob_predict.

eps: float, optional precision for stopping criteria

coef0 [float, optional] independent term in kernel function. It is only significant in poly/sigmoid.

cache_size: float, optional specify the size of the cache (in MB)

shrinking: boolean, optional wether to use the shrinking heuristic.

Attributes

support_ [array-like, shape = [n_SV]] Index of support vectors.

support_vectors_ [array-like, shape = [nSV, n_features]] Support vectors.

dual_coef_ [array, shape = [n_classes-1, n_SV]] Coefficients of the support vector in the decision function.

coef_ [array, shape = [n_classes-1, n_features]] Weights asigned to the features (coefficients in the primal problem). This is only available in the case of linear kernel.

intercept_ [array, shape = [n_class * (n_class-1) / 2]] Constants in decision function.

See also

NuSVR

Full API documentation: SVRScikitsLearnNode

class `mdp.nodes.`**`NuSVCScikitsLearnNode`**

Nu-Support Vector Classification.

This node has been automatically generated by wrapping the `scikits.learn.svm.libsvm.NuSVC` class from the `scikits.learn` library. The wrapped instance can be accessed through the `scikits_alg` attribute.

Parameters

nu [float, optional] An upper bound on the fraction of training errors and a lower bound of the fraction of support vectors. Should be in the interval (0, 1]. By default 0.5 will be taken.

kernel [string, optional] Specifies the kernel type to be used in the algorithm. one of 'linear', 'poly', 'rbf', 'sigmoid', 'precomputed'. If none is given 'rbf' will be used.

degree [int, optional] degree of kernel function is significant only in poly, rbf, sigmoid

gamma [float, optional] kernel coefficient for rbf and poly, by default 1/n_features will be taken.

probability: boolean, optional (False by default) enable probability estimates. This must be enabled prior to calling prob_predict.

coef0 [float, optional] independent term in kernel function. It is only significant in poly/sigmoid.

shrinking: boolean, optional wether to use the shrinking heuristic.

eps: float, optional precision for stopping criteria

cache_size: float, optional specify the size of the cache (in MB)

Attributes

support_ [array-like, shape = [n_SV]] Index of support vectors.

support_vectors_ [array-like, shape = [n_SV, n_features]] Support vectors.

n_support_ [array-like, dtype=int32, shape = [n_class]] number of support vector for each class.

dual_coef_ [array, shape = [n_classes-1, n_SV]] Coefficients of the support vector in the decision function.

coef_ [array, shape = [n_classes-1, n_features]] Weights asigned to the features (coefficients in the primal problem). This is only available in the case of linear kernel.

intercept_ [array, shape = [n_class * (n_class-1) / 2]] Constants in decision function.

Methods

fit(X, y) [self] Fit the model

predict(X) [array] Predict using the model.

predict_proba(X) [array] Return probability estimates.

predict_log_proba(X) [array] Return log-probability estimates.

decision_function(X) [array] Return distance to predicted margin.

Examples

```
>>> import numpy as np
>>> X = np.array([[-1, -1], [-2, -1], [1, 1], [2, 1]])
>>> y = np.array([1, 1, 2, 2])
>>> from scikits.learn.svm import NuSVC
>>> clf = NuSVC()
>>> clf.fit(X, y)
NuSVC(kernel='rbf', probability=False, degree=3, coef0=0.0, eps=0.001,
   cache_size=100.0, shrinking=True, nu=0.5, gamma=0.25)
>>> print clf.predict([[-0.8, -1]])
[ 1.]
```

See also

SVC, LinearSVC, SVR

Full API documentation: NuSVCScikitsLearnNode

class `mdp.nodes.`**`RFECVScikitsLearnNode`**

Feature ranking with Recursive feature elimination and cross validation

This node has been automatically generated by wrapping the `scikits.learn.feature_selection.rfe.RFECV` class from the `scikits.learn` library. The wrapped instance can be accessed through the `scikits_alg` attribute.

Parameters

estimator [object] A supervised learning estimator with a fit method that updates a `coef_` attributes that holds the fitted parameters. The first dimension of the `coef_` array must be equal n_features an important features must yield high absolute values in the `coef_` array.

For instance this is the case for most supervised learning algorithms such as Support Vector Classifiers and Generalized Linear Models from the svm and linear_model package.

n_features [int] Number of features to select

percentage [float] The percentage of features to remove at each iteration Should be between (0, 1]. By default 0.1 will be taken.

Attributes

support_ [array-like, shape = [n_features]] Mask of estimated support

ranking_ [array-like, shape = [n_features]] Mask of the ranking of features

Methods

fit(X, y) [self] Fit the model

transform(X) [array] Reduce X to support

Examples

```
>>> # TODO!
```

References

Guyon, I., Weston, J., Barnhill, S., & Vapnik, V. (2002). Gene selection for cancer classification using support vector machines. Mach. Learn., 46(1-3), 389–422.

Full API documentation: RFECVScikitsLearnNode

class `mdp.nodes.`**`GNBScikitsLearnNode`**

Gaussian Naive Bayes (GNB)

This node has been automatically generated by wrapping the `scikits.learn.naive_bayes.GNB` class from the `scikits.learn` library. The wrapped instance can be accessed through the `scikits_alg` attribute.

Parameters

X [array-like, shape = [n_samples, n_features]] Training vector, where n_samples in the number of samples and n_features is the number of features.

y [array, shape = [n_samples]] Target vector relative to X

Attributes

proba_y [array, shape = nb of classes] probability of each class.

theta [array of shape nb_class*nb_features] mean of each feature for the different class

sigma [array of shape nb_class*nb_features] variance of each feature for the different class

Methods

fit(X, y) [self] Fit the model

predict(X) [array] Predict using the model.

predict_proba(X) [array] Predict the probability of each class using the model.

Examples

```
>>> import numpy as np
>>> X = np.array([[-1, -1], [-2, -1], [-3, -2], [1, 1], [2, 1], [3, 2]])
>>> Y = np.array([1, 1, 1, 2, 2, 2])
>>> from scikits.learn.naive_bayes import GNB
>>> clf = GNB()
>>> clf.fit(X, Y)
GNB()
>>> print clf.predict([[-0.8, -1]])
[1]
```

See also

Full API documentation: GNBScikitsLearnNode

class `mdp.nodes.`**`SelectKBestScikitsLearnNode`**

This node has been automatically generated by wrapping the `scikits.learn.feature_selection.univariate_s` class from the `scikits.learn` library. The wrapped instance can be accessed through the `scikits_alg` attribute.

Full API documentation: SelectKBestScikitsLearnNode

class `mdp.nodes.`**`ProbabilisticPCAScikitsLearnNode`**

class `mdp.nodes.`**`LinearRegressionScikitsLearnNode`**

Ordinary least squares Linear Regression.

This node has been automatically generated by wrapping the `scikits.learn.linear_model.base.LinearRegres` class from the `scikits.learn` library. The wrapped instance can be accessed through the `scikits_alg` attribute.

Attributes

coef_ [array] Estimated coefficients for the linear regression problem.

intercept_ [array] Independent term in the linear model.

Notes

From the implementation point of view, this is just plain Ordinary Least Squares (numpy.linalg.lstsq) wrapped as a predictor object.

Full API documentation: LinearRegressionScikitsLearnNode

class mdp.nodes.**SelectPercentileScikitsLearnNode**
This node has been automatically generated by wrapping the scikits.learn.feature_selection.univariate_s class from the scikits.learn library. The wrapped instance can be accessed through the scikits_alg attribute.

Full API documentation: SelectPercentileScikitsLearnNode

class mdp.nodes.**LengthNormalizerScikitsLearnNode**
This node has been automatically generated by wrapping the scikits.learn.preprocessing.LengthNormalizer class from the scikits.learn library. The wrapped instance can be accessed through the scikits_alg attribute.

Full API documentation: LengthNormalizerScikitsLearnNode

class mdp.nodes.**GaussianHMMScikitsLearnNode**
Hidden Markov Model with Gaussian emissions

This node has been automatically generated by wrapping the scikits.learn.hmm.GaussianHMM class from the scikits.learn library. The wrapped instance can be accessed through the scikits_alg attribute.

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.

Attributes

cvtype [string (read-only)] String describing the type of covariance parameters used by the model. Must be one of 'spherical', 'tied', 'diag', 'full'.

n_features [int (read-only)] Dimensionality of the Gaussian emissions.

n_states [int (read-only)] Number of states in the model.

transmat [array, shape (*n_states*, *n_states*)] Matrix of transition probabilities between states.

startprob [array, shape ('n_states',)] Initial state occupation distribution.

means [array, shape (*n_states*, *n_features*)] Mean parameters for each state.

covars [array] Covariance parameters for each state. The shape depends on *cvtype*:

- (*n_states*,) if 'spherical',
- (*n_features*, *n_features*) if 'tied',
- (*n_states*, *n_features*) if 'diag',
- (*n_states*, *n_features*, *n_features*) if 'full'

Methods

eval(X) Compute the log likelihood of *X* under the HMM.

decode(X) Find most likely state sequence for each point in *X* using the Viterbi algorithm.

rvs(n=1) Generate *n* samples from the HMM.

init(X) Initialize HMM parameters from *X*.

fit(X) Estimate HMM parameters from *X* using the Baum-Welch algorithm.

predict(X) Like decode, find most likely state sequence corresponding to *X*.

score(X) Compute the log likelihood of *X* under the model.

Examples

```
>>> from scikits.learn.hmm import GaussianHMM
>>> GaussianHMM(n_states=2)
GaussianHMM(cvtype='diag', n_states=2, means_weight=0, startprob_prior=1.0,
      startprob=array([ 0.5,  0.5]),
      transmat=array([[ 0.5,  0.5],
```

```
 [ 0.5,  0.5]]),
transmat_prior=1.0, means_prior=None, covars_weight=1,
covars_prior=0.01)
```

See Also

GMM : Gaussian mixture model

Full API documentation: GaussianHMMScikitsLearnNode

class mdp.nodes.**LARSScikitsLearnNode**

Least Angle Regression model a.k.a. LAR

This node has been automatically generated by wrapping the `scikits.learn.linear_model.least_angle.LARS` class from the `scikits.learn` library. The wrapped instance can be accessed through the `scikits_alg` attribute.

Parameters

n_features [int, optional] Number of selected active features

fit_intercept [boolean] whether to calculate the intercept for this model. If set to false, no intercept will be used in calculations (e.g. data is expected to be already centered).

Attributes

coef_ [array, shape = [n_features]] parameter vector (w in the fomulation formula)

intercept_ [float] independent term in decision function.

Examples

```
>>> from scikits.learn import linear_model
>>> clf = linear_model.LARS()
>>> clf.fit([[-1,1], [0, 0], [1, 1]], [-1, 0, -1], max_features=1)
LARS(verbose=False, fit_intercept=True)
>>> print clf.coef_
[ 0.         -0.81649658]
```

References

http://en.wikipedia.org/wiki/Least_angle_regression

See also

lars_path, LassoLARS

Full API documentation: LARSScikitsLearnNode

class mdp.nodes.**QDAScikitsLearnNode**

Quadratic Discriminant Analysis (QDA)

This node has been automatically generated by wrapping the `scikits.learn.qda.QDA` class from the `scikits.learn` library. The wrapped instance can be accessed through the `scikits_alg` attribute.

Parameters

X [array-like, shape = [n_samples, n_features]] Training vector, where n_samples in the number of samples and n_features is the number of features.

y [array, shape = [n_samples]] Target vector relative to X

priors [array, optional, shape = [n_classes]] Priors on classes

Attributes

means_ [array-like, shape = [n_classes, n_features]] Class means

priors_ [array-like, shape = [n_classes]] Class priors (sum to 1)

covariances_ [list of array-like, shape = [n_features, n_features]] Covariance matrices of each class

Examples

```
>>> from scikits.learn.qda import QDA
>>> import numpy as np
>>> X = np.array([[-1, -1], [-2, -1], [-3, -2], [1, 1], [2, 1], [3, 2]])
>>> y = np.array([1, 1, 1, 2, 2, 2])
>>> clf = QDA()
>>> clf.fit(X, y)
QDA(priors=None)
>>> print clf.predict([[-0.8, -1]])
[1]
```

See also

LDA

Full API documentation: QDAScikitsLearnNode

class mdp.nodes.**NeighborsScikitsLearnNode**

Classifier implementing k-Nearest Neighbor Algorithm.

This node has been automatically generated by wrapping the `scikits.learn.neighbors.Neighbors` class from the `scikits.learn` library. The wrapped instance can be accessed through the `scikits_alg` attribute.

Parameters

data [array-like, shape (n, k)] The data points to be indexed. This array is not copied, and so modifying this data will result in bogus results.

labels [array] An array representing labels for the data (only arrays of integers are supported).

n_neighbors [int] default number of neighbors.

window_size [int] Window size passed to BallTree

Examples

```
>>> samples = [[0.,0.,1.], [1.,0.,0.], [2.,2.,2.], [2.,5.,4.]]
>>> labels = [0,0,1,1]
>>> from scikits.learn.neighbors import Neighbors
>>> neigh = Neighbors(n_neighbors=3)
>>> neigh.fit(samples, labels)
Neighbors(n_neighbors=3, window_size=1)
>>> print neigh.predict([[0,0,0]])
[ 0.]
```

Notes

http://en.wikipedia.org/wiki/K-nearest_neighbor_algorithm

Full API documentation: NeighborsScikitsLearnNode

CHAPTER

FIFTEEN

ADDITIONAL UTILITIES

MDP offers some additional utilities of general interest in the `mdp.utils` module. Refer to the API for the full documentation and interface description.

mdp.utils.CovarianceMatrix This class stores an empirical covariance matrix that can be updated incrementally. A call to the `fix` method returns the current state of the covariance matrix, the average and the number of observations, and resets the internal data.

Note that the internal sum is a standard `__add__` operation. We are not using any of the fancy sum algorithms to avoid round off errors when adding many numbers. If you want to contribute a `CovarianceMatrix` class that uses such algorithms we would be happy to include it in MDP. For a start see the Python recipe by Raymond Hettinger. For a review about floating point arithmetic and its pitfalls see *What every computer scientist should know about floating-point arithmetic* by David Goldberg, ACM Computing Surveys, Vol 23, No 1, March 1991.

mdp.utils.DelayCovarianceMatrix This class stores an empirical covariance matrix between the signal and time delayed signal that can be updated incrementally.

mdp.utils.MultipleCovarianceMatrices Container class for multiple covariance matrices to easily execute operations on all matrices at the same time.

mdp.utils.dig_node (node) Crawl recursively an MDP `Node` looking for arrays. Return (dictionary, string), where the dictionary is: { attribute_name: (size_in_bytes, array_reference)} and string is a nice string representation of it.

mdp.utils.get_node_size (node) Get `node` total byte-size using `cPickle` with protocol=2. (The byte-size is related the memory needed by the node).

mdp.utils.progressinfo (sequence, length, style, custom) A fully configurable text-mode progress info box tailored to the command-line die-hards. To get a progress info box for your loops use it like this:

```
>>> for i in progressinfo(sequence):
...     do_something(i)
```

You can also use it with generators, files or any other iterable object, but in this case you have to specify the total length of the sequence:

```
>>> for line in progressinfo(open_file, nlines):
...     do_something(line)
```

A few examples of the available layouts:

```
[==================================73%=============>..................]

Progress:  67%[====================================>                   ]

23% [02:01:28] - [00:12:37]
```

mdp.utils.QuadraticForm Define an inhomogeneous quadratic form as `1/2 x'Hx + f'x + c`. This class implements the quadratic form analysis methods presented in: Berkes, P. and Wiskott, L. On the analysis

and interpretation of inhomogeneous quadratic forms as receptive fields. *Neural Computation*, 18(8): 1868-1895. (2006).

mdp.utils.refcast (array, dtype) Cast the array to `dtype` only if necessary, otherwise return a reference.

mdp.utils.rotate (mat, angle, columns, units) Rotate in-place a NxM data matrix in the plane defined by the `columns` when observation are stored on rows. Observations are rotated counterclockwise. This corresponds to the following matrix-multiplication for each data-point (unchanged elements omitted):

```
[  cos(angle) -sin(angle)      [ x_i ]
   sin(angle)  cos(angle) ] * [ x_j ]
```

mdp.utils.random_rot (dim, dtype) Return a random rotation matrix, drawn from the Haar distribution (the only uniform distribution on SO(n)). The algorithm is described in the paper Stewart, G.W., *The efficient generation of random orthogonal matrices with an application to condition estimators*, SIAM Journal on Numerical Analysis, 17(3), pp. 403-409, 1980. For more information see this Wikipedia entry.

mdp.utils.symrand (dim_or_eigv, dtype) Return a random symmetric (Hermitian) matrix with eigenvalues uniformly distributed on (0,1].

15.1 HTML Slideshows

The `mdp.utils` module contains some classes and helper function to display animated results in a Webbrowser. This works by creating an HTML file with embedded JavaScript code, which dynamically loads image files (the images contain the content that you want to animate and can for example be created with matplotlib). MDP internally uses the open source Templete templating libray, written by David Bau.

The easiest way to create a slideshow it to use one of these two helper function:

mdp.utils.show_image_slideshow (filenames, image_size, filename=None, title=None, **kwargs) Write the slideshow into a HTML file, open it in the browser and return the file name. `filenames` is a list of the images files that you want to display in the slideshow. `image_size` is a 2-tuple containing the width and height at which the images should be displayed. There are also a couple of additional arguments, which are documented in the docstring.

mdp.utils.image_slideshow (filenames, image_size, title=None, **kwargs) This function is similar to `show_image_slideshow`, but it simply returns the slideshow HTML code (including the JavaScript code) which you can then embed into your own HTML file. Note that the default slideshow CSS code is not included, but it can be accessed in `mdp.utils.IMAGE_SLIDESHOW_STYLE`.

Note that there are also two demos for slideshows in the `mdp\demo` folder.

15.2 Graph module

MDP contains `mdp.graph`, a lightweight package to handle directed graphs.

mdp.graph.Graph Represent a directed graph. This class contains several methods to create graph structures and manipulate them, among which

- **`add_tree`: Add a tree to the graph.** The tree is specified with a nested list of tuple, in a LISP-like notation. The values specified in the list become the values of the single nodes. Return an equivalent nested list with the nodes instead of the values.

  Example::

  ```
  >>> g = mdp.graph.Graph()
  >>> a = b = c = d = e = None
  >>> nodes = g.add_tree( (a, b, (c, d ,e)) )
  ```

  Graph `g` corresponds to this tree, with all node values being `None`:

```
  a
 / \
b   c
   / \
  d   e
```

- `topological_sort`: Perform a topological sort of the nodes.
- `dfs`, `undirected_dfs`: Perform Depth First sort.
- `bfs`, `undirected_bfs`: Perform Breadth First sort.
- `connected_components`: Return a list of lists containing the nodes of all connected components of the graph.
- `is_weakly_connected`: Return True if the graph is weakly connected.

mdp.graph.GraphEdge Represent a graph edge and all information attached to it.

mdp.graph.GraphNode Represent a graph node and all information attached to it.

mdp.graph.recursive_map (fun, seq) Apply a function recursively on a sequence and all subsequences.

mdp.graph.recursive_reduce (func, seq, *argv) Apply `reduce(func, seq)` recursively to a sequence and all its subsequences.

CHAPTER

SIXTEEN

LICENSE

MDP is distributed under the open source BSD license.

```
This file is part of Modular toolkit for Data Processing (MDP).
All the code in this package is distributed under the following conditions:

Copyright (c) 2003-2011, MDP Developers <mdp-toolkit-devel@lists.sourceforge.net>

All rights reserved.

Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions are met:

    * Redistributions of source code must retain the above copyright
      notice, this list of conditions and the following disclaimer.
    * Redistributions in binary form must reproduce the above copyright
      notice, this list of conditions and the following disclaimer in the
      documentation and/or other materials provided with the distribution.
    * Neither the name of the Modular toolkit for Data Processing (MDP)
      nor the names of its contributors may be used to endorse or promote
      products derived from this software without specific prior written
      permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND
ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED
WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE
DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT HOLDER OR CONTRIBUTORS BE LIABLE
FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL
DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR
SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER
CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY,
OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE
OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
```

INDEX