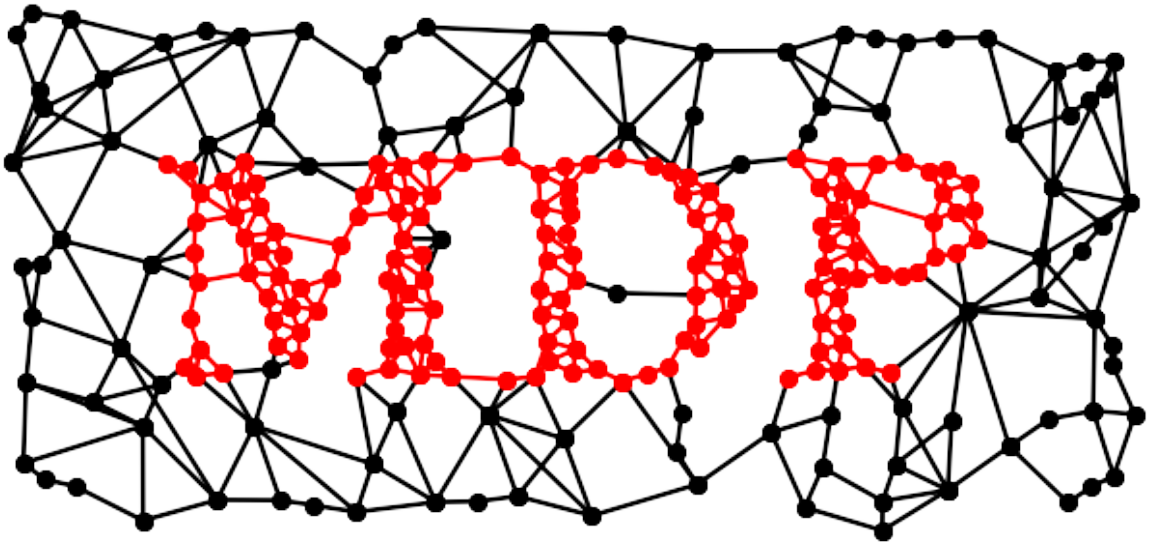


Modular toolkit for Data Processing

Tutorial

Release 3.3

Authors: MDP Developers

October 06, 2012

CONTENTS

Authors MDP Developers

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

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

Version 3.3

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
0.8
```

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)

[==================================100%==================================>]

Training finished
Training node #1 (BogusNode)
[==================================100%==================================>]

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%==================================>]

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%==================================>]

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
...         return ...
...     def _join(self):
...         # implement the joining for SFANode
...         return ...
```

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):
...     return ...
```

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)
```

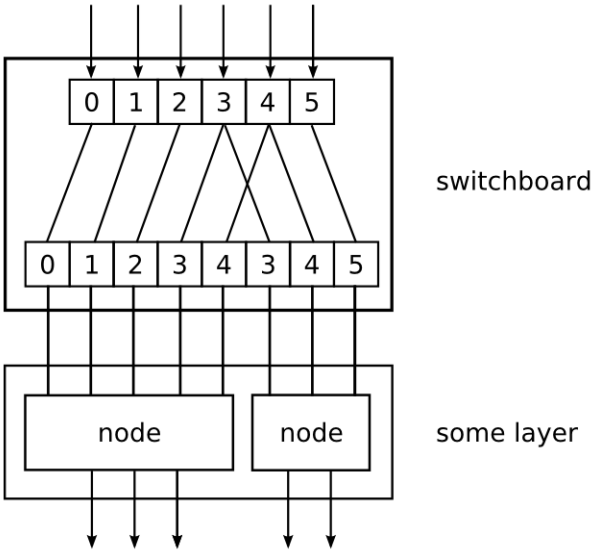
In this example `node1` has two training phases (one for each internal node). Therefore `layer` now has two training phases as well and behaves like any other node with two training phases. By combining and nesting `FlowNode` and `Layer`, it is thus possible to build modular node structures. Note that 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).

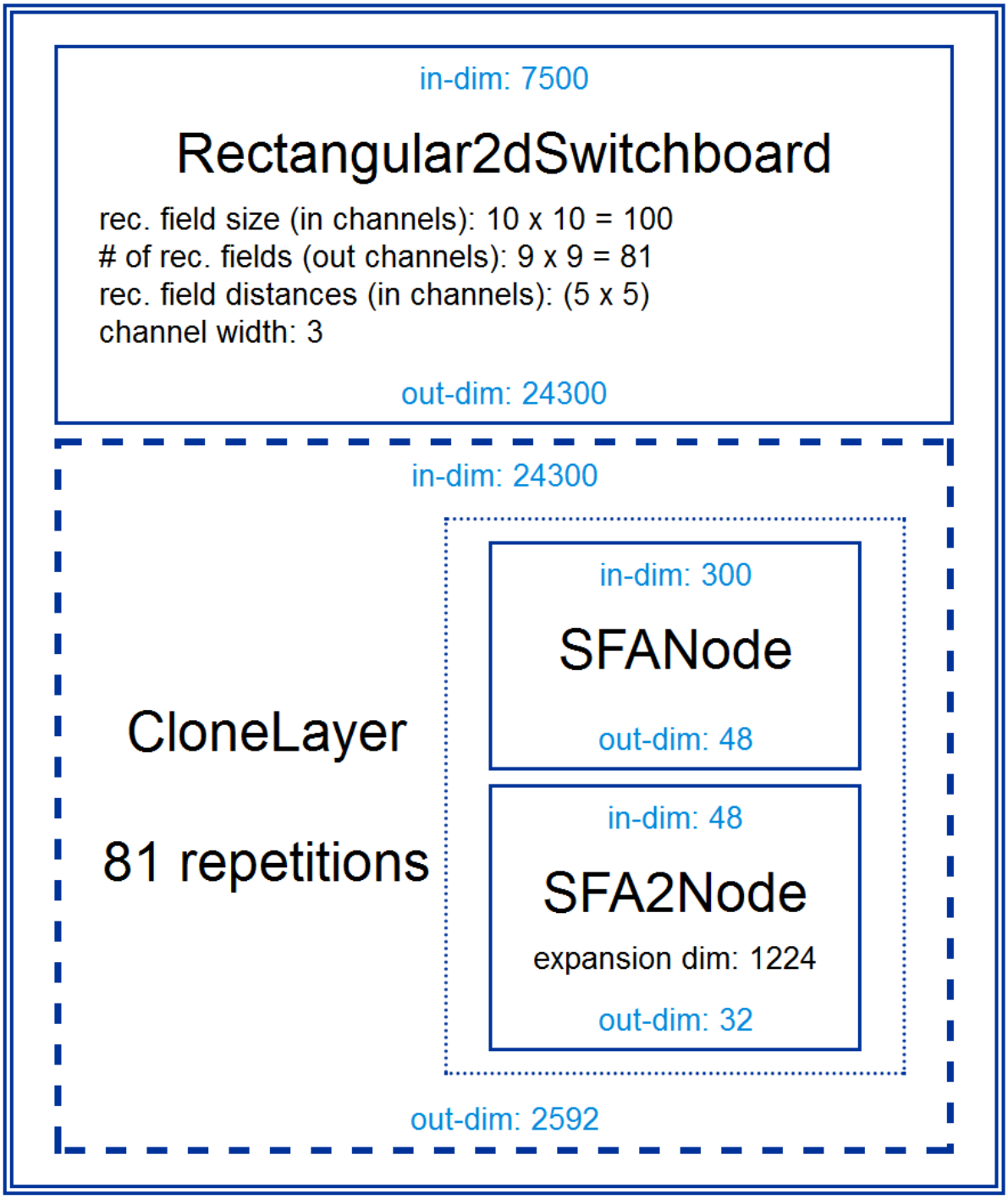
8.3 Example application (2-D image data)

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).

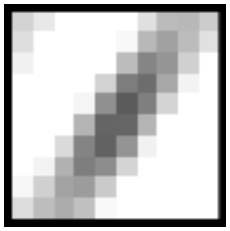
The processing layer consists of many parallel units, which only see a small image region with a size of 10 by 10 pixels. These so called receptive fields cover the whole image and have an overlap of five pixels. Note that the image data is represented as an 1-D array. Therefore we need the `Rectangular2dSwitchboard` 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

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.

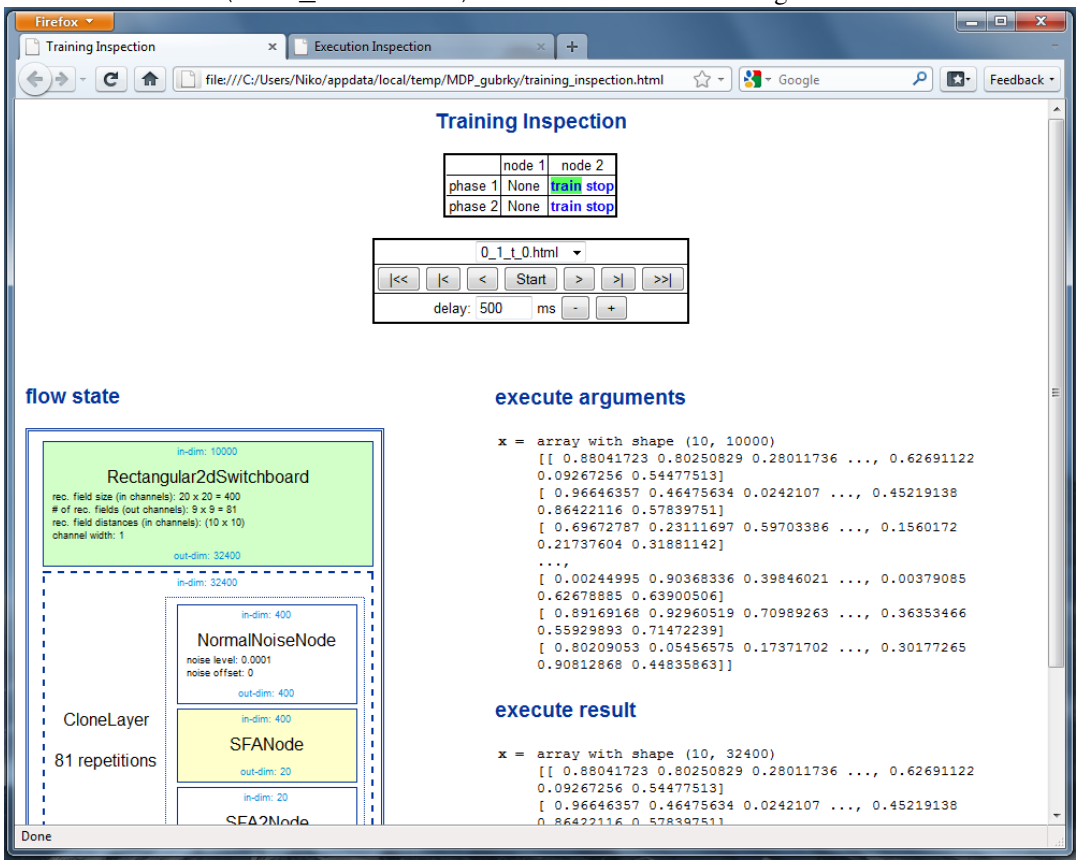
Some of these new options might be confusing at first. However, you can simply ignore those that you don't need and concentrate on the features that are useful for your current project. For example you could use messages without ever worrying about targets.

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

Full API documentation: `nodes`

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.

The algorithm expands on the original Neural Gas algorithm (see mdp.nodes NeuralGasNode) in that the algorithm adds new nodes are added to the graph as more data becomes available. Im this way, if the growth rate is appropriate, one can avoid overfitting or underfitting the data.

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.

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.`**`NeuralGasNode`**
Learn the topological structure of the input data by building a corresponding graph approximation (original Neural Gas algorithm).

The Neural Gas algorithm was originally published in Martinetz, T. and Schulten, K.: A "Neural-Gas" Network Learns Topologies. In Kohonen, T., Maekisara, K., Simula, O., and Kangas, J. (eds.), Artificial Neural Networks. Elsevier, North-Holland., 1991.

Attributes and methods of interest

- graph – The corresponding *mdp.graph.Graph* object
- max_epochs - maximum number of epochs until which to train.

Full API documentation: `NeuralGasNode`

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. 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. 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.**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**
Full API documentation: SGDRegressorScikitsLearnNode

class mdp.nodes.**PatchExtractorScikitsLearnNode**
Extracts patches from a collection of images

This node has been automatically generated by wrapping the sklearn.feature_extraction.image.PatchExtrac class from the sklearn library. The wrapped instance can be accessed through the scikits_alg attribute.

Parameters

patch_size: tuple of ints (patch_height, patch_width) the dimensions of one patch

max_patches: integer or float, optional default is None The maximum number of patches per image to extract. If max_patches is a float in (0, 1), it is taken to mean a proportion of the total number of patches.

random_state: int or RandomState Pseudo number generator state used for random sampling.

Full API documentation: PatchExtractorScikitsLearnNode

class mdp.nodes.**LinearModelCVScikitsLearnNode**
This node has been automatically generated by wrapping the sklearn.linear_model.coordinate_descent.Line class from the sklearn library. The wrapped instance can be accessed through the scikits_alg attribute.

Full API documentation: LinearModelCVScikitsLearnNode

class mdp.nodes.**DictionaryLearningScikitsLearnNode**
Dictionary learning

This node has been automatically generated by wrapping the sklearn.decomposition.dict_learning.Dictiona class from the sklearn library. The wrapped instance can be accessed through the scikits_alg attribute.

Finds a dictionary (a set of atoms) that can best be used to represent data using a sparse code.

Solves the optimization problem:

```
(U^*,V^*) = argmin 0.5 || Y - U V ||_2^2 + alpha * || U ||_1
            (U,V)
            with || V_k ||_2 = 1 for all  0 <= k < n_atoms
```

Parameters

n_atoms [int,] number of dictionary elements to extract

alpha [int,] sparsity controlling parameter

max_iter [int,] maximum number of iterations to perform

tol [float,] tolerance for numerical error

fit_algorithm [{'lars', 'cd'}] lars: uses the least angle regression method to solve the lasso problem (linear_model.lars_path) cd: uses the coordinate descent method to compute the Lasso solution (linear_model.Lasso). Lars will be faster if the estimated components are sparse.

transform_algorithm [{'lasso_lars', 'lasso_cd', 'lars', 'omp', 'threshold'}] Algorithm used to transform the data lars: uses the least angle regression method (linear_model.lars_path) lasso_lars: uses Lars to compute the Lasso solution lasso_cd: uses the coordinate descent method to compute the Lasso solution (linear_model.Lasso). lasso_lars will be faster if the estimated components are sparse. omp: uses orthogonal matching pursuit to estimate the sparse solution threshold: squashes to zero all coefficients less than alpha from the projection `dictionary * X'`

transform_n_nonzero_coefs [int, `0.1 * n_features` by default] Number of nonzero coefficients to target in each column of the solution. This is only used by *algorithm='lars'* and *algorithm='omp'* and is overridden by *alpha* in the *omp* case.

transform_alpha [float, 1. by default] If *algorithm='lasso_lars'* or *algorithm='lasso_cd'*, *alpha* is the penalty applied to the L1 norm. If *algorithm='threshold'*, *alpha* is the absolute value of the threshold below which coefficients will be squashed to zero. If *algorithm='omp'*, *alpha* is the tolerance parameter: the value of the reconstruction error targeted. In this case, it overrides *n_nonzero_coefs*.

split_sign [bool, False by default] Whether to split the sparse feature vector into the concatenation of its negative part and its positive part. This can improve the performance of downstream classifiers.

n_jobs [int,] number of parallel jobs to run

code_init [array of shape (n_samples, n_atoms),] initial value for the code, for warm restart

dict_init [array of shape (n_atoms, n_features),] initial values for the dictionary, for warm restart

verbose :

- degree of verbosity of the printed output

random_state [int or RandomState] Pseudo number generator state used for random sampling.

Attributes

components_ [array, [n_atoms, n_features]] dictionary atoms extracted from the data

error_ [array] vector of errors at each iteration

Notes

References:

J. Mairal, F. Bach, J. Ponce, G. Sapiro, 2009: Online dictionary learning for sparse coding (http://www.di.ens.fr/sierra/pdfs/icml09.pdf)

See also

SparseCoder MiniBatchDictionaryLearning SparsePCA MiniBatchSparsePCA

Full API documentation: `DictionaryLearningScikitsLearnNode`

class mdp.nodes.**PerceptronScikitsLearnNode**
Perceptron

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

Parameters

penalty [None, 'l2' or 'l1' or 'elasticnet'] The penalty (aka regularization term) to be used. Defaults to None.

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

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, optional The number of passes over the training data (aka epochs). Defaults to 5.

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

seed: int, optional The seed of the pseudo random number generator to use when shuffling the data.

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.

eta0 [double] Constant by which the updates are multiplied. Defaults to 1.

class_weight [dict, {class_label][weight} or "auto" or None, optional] Preset for the class_weight fit parameter.

Weights associated with classes. If not given, all classes are supposed to have weight one.

The "auto" mode uses the values of y to automatically adjust weights inversely proportional to class frequencies.

warm_start [bool, optional] When set to True, reuse the solution of the previous call to fit as initialization, otherwise, just erase the previous solution.

Attributes

*coef_* : array, shape = [1, n_features] if n_classes == 2 else [n_classes, n_features]

Weights assigned to the features.

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

Notes

Perceptron and *SGDClassifier* share the same underlying implementation. In fact, *Perceptron()* is equivalent to *SGDClassifier(loss="perceptron", eta0=1, learning_rate="constant", penalty=None)*.

See also

SGDClassifier

References

http://en.wikipedia.org/wiki/Perceptron and references therein.

Full API documentation: `PerceptronScikitsLearnNode`

class mdp.nodes.**RidgeClassifierScikitsLearnNode**
Classifier using Ridge regression.

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

Parameters

alpha [float] Small positive values of alpha improve the conditioning of the problem and reduce the variance of the estimates. Alpha corresponds to `(2*C)^-1` in other linear models such as LogisticRegression or LinearSVC.

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).

normalize [boolean, optional] If True, the regressors X are normalized

copy_X [boolean, optional, default True] If True, X will be copied; else, it may be overwritten.

tol [float] Precision of the solution.

class_weight [dict, optional] Weights associated with classes in the form {class_label : weight}. If not given, all classes are supposed to have weight one.

Attributes

coef_ [array, shape = [n_features] or [n_classes, n_features]] Weight vector(s).

See also

Ridge, RidgeClassifierCV

Notes

For multi-class classification, n_class classifiers are trained in a one-versus-all approach. Concretely, this is implemented by taking advantage of the multi-variate response support in Ridge.

Full API documentation: `RidgeClassifierScikitsLearnNode`

class `mdp.nodes.`**`WardAgglomerationScikitsLearnNode`**

Feature agglomeration based on Ward hierarchical clustering

This node has been automatically generated by wrapping the `sklearn.cluster.hierarchical.WardAgglomerati` class from the `sklearn` library. The wrapped instance can be accessed through the `scikits_alg` attribute.

Parameters

n_clusters [int or ndarray] The number of clusters.

connectivity [sparse matrix] connectivity matrix. Defines for each feature the neigbhoring features following a given structure of the data. Default is None, i.e, the hiearchical agglomeration algorithm is unstructured.

memory [Instance of joblib.Memory or string] Used to cache the output of the computation of the tree. By default, no caching is done. If a string is given, it is the path to the caching directory.

copy [bool] Copy the connectivity matrix or work inplace.

n_components [int (optional)] The number of connected components in the graph defined by the connectivity matrix. If not set, it is estimated.

compute_full_tree: bool or 'auto' (optional) Stop early the construction of the tree at n_clusters. This is useful to decrease computation time if the number of clusters is not small compared to the number of samples. This option is useful only when specifying a connectivity matrix. Note also that when varying the number of cluster and using caching, it may be advantageous to compute the full tree.

Attributes

children_ [array-like, shape = [n_nodes, 2]] List of the children of each nodes. Leaves of the tree do not appear.

labels_ [array [n_samples]] cluster labels for each point

n_leaves_ [int] Number of leaves in the hiearchical tree.

Full API documentation: `WardAgglomerationScikitsLearnNode`

class mdp.nodes.**KNeighborsClassifierScikitsLearnNode**

Classifier implementing the k-nearest neighbors vote.

This node has been automatically generated by wrapping the sklearn.neighbors.classification.KNeighborsC class from the sklearn library. The wrapped instance can be accessed through the scikits_alg attribute.

Parameters

n_neighbors [int, optional (default = 5)] Number of neighbors to use by default for k_neighbors() queries.

weights [str or callable] weight function used in prediction. Possible values:

- 'uniform' : uniform weights. All points in each neighborhood are weighted equally.
- 'distance' : weight points by the inverse of their distance. in this case, closer neighbors of a query point will have a greater influence than neighbors which are further away.
- [callable] : a user-defined function which accepts an array of distances, and returns an array of the same shape containing the weights.

Uniform weights are used by default.

algorithm [{'auto', 'ball_tree', 'kd_tree', 'brute'}, optional] Algorithm used to compute the nearest neighbors:

- 'ball_tree' will use BallTree
- 'kd_tree' will use scipy.spatial.cKDtree
- 'brute' will use a brute-force search.
- 'auto' will attempt to decide the most appropriate algorithm based on the values passed to fit() method.

Note: fitting on sparse input will override the setting of this parameter, using brute force.

leaf_size [int, optional (default = 30)] Leaf size passed to BallTree or cKDTree. This can affect the speed of the construction and query, as well as the memory required to store the tree. The optimal value depends on the nature of the problem.

warn_on_equidistant [boolean, optional. Defaults to True.] Generate a warning if equidistant neighbors are discarded. For classification or regression based on k-neighbors, if neighbor k and neighbor k+1 have identical distances but different labels, then the result will be dependent on the ordering of the training data. If the fit method is 'kd_tree', no warnings will be generated.

p: integer, optional (default = 2) Parameter for the Minkowski metric from sklearn.metrics.pairwise.pairwise_distances. When p = 1, this is equivalent to using manhattan_distance (l1), and euclidean_distance (l2) for p = 2. For arbitrary p, minkowski_distance (l_p) is used.

Examples

```
>>> X = [[0], [1], [2], [3]]
>>> y = [0, 0, 1, 1]
>>> from sklearn.neighbors import KNeighborsClassifier
>>> neigh = KNeighborsClassifier(n_neighbors=3)
>>> neigh.fit(X, y)
KNeighborsClassifier(...)
>>> print(neigh.predict([[1.1]]))
[0]
>>> print(neigh.predict_proba([[0.9]]))
[[ 0.66666667  0.33333333]]
```

See also

RadiusNeighborsClassifier KNeighborsRegressor RadiusNeighborsRegressor NearestNeighbors

Notes

See *Nearest Neighbors* in the online documentation for a discussion of the choice of `algorithm` and `leaf_size`.

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

Full API documentation: `KNeighborsClassifierScikitsLearnNode`

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

NuSVR for sparse matrices (csr)

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

See `sklearn.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).

Examples

```
>>> from sklearn.svm.sparse import NuSVR
>>> 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 = NuSVR(nu=0.1, C=1.0)
>>> clf.fit(X, y)
NuSVR(C=1.0, cache_size=200, coef0=0.0, degree=3, epsilon=0.1, gamma=0.0,
   kernel='rbf', nu=0.1, probability=False, shrinking=True, tol=0.001,
   verbose=False)
```

Full API documentation: `NuSVRScikitsLearnNode`

class `mdp.nodes.`**`NearestCentroidScikitsLearnNode`**

Nearest centroid classifier.

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

Each class is represented by its centroid, with test samples classified to the class with the nearest centroid.

Parameters

metric: string, or callable The metric to use when calculating distance between instances in a feature array. If metric is a string or callable, it must be one of the options allowed by metrics.pairwise.pairwise_distances for its metric parameter.

shrink_threshold [float, optional (default = None)] Threshold for shrinking centroids to remove features.

Attributes

centroids_ [array-like, shape = [n_classes, n_features]] Centroid of each class

Examples

```
>>> from sklearn.neighbors.nearest_centroid import NearestCentroid
>>> 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 = NearestCentroid()
>>> clf.fit(X, y)
NearestCentroid(metric='euclidean', shrink_threshold=None)
```

```
>>> print clf.predict([[-0.8, -1]])
[1]
```

See also

sklearn.neighbors.KNeighborsClassifier: nearest neighbors classifier

Notes

When used for text classification with tf–idf vectors, this classifier is also known as the Rocchio classifier.

References

Tibshirani, R., Hastie, T., Narasimhan, B., & Chu, G. (2002). Diagnosis of multiple cancer types by shrunken centroids of gene expression. Proceedings of the National Academy of Sciences of the United States of America, 99(10), 6567-6572. The National Academy of Sciences.

Full API documentation: `NearestCentroidScikitsLearnNode`

class `mdp.nodes.`**`ExtraTreeRegressorScikitsLearnNode`**

An extremely randomized tree regressor.

This node has been automatically generated by wrapping the `sklearn.tree.tree.ExtraTreeRegressor` class from the `sklearn` library. The wrapped instance can be accessed through the `scikits_alg` attribute.

Extra-trees differ from classic decision trees in the way they are built. When looking for the best split to separate the samples of a node into two groups, random splits are drawn for each of the *max_features* randomly selected features and the best split among those is chosen. When *max_features* is set 1, this amounts to building a totally random decision tree.

Warning: Extra-trees should only be used within ensemble methods.

See also

ExtraTreeClassifier : A classifier base on extremely randomized trees sklearn.ensemble.ExtraTreesClassifier : An ensemble of extra-trees for

classification

sklearn.ensemble.ExtraTreesRegressor [An ensemble of extra-trees for] regression

References

Full API documentation: `ExtraTreeRegressorScikitsLearnNode`

class `mdp.nodes.`**`ExtraTreesClassifierScikitsLearnNode`**

An extra-trees classifier.

This node has been automatically generated by wrapping the `sklearn.ensemble.forest.ExtraTreesClassifier` class from the `sklearn` library. The wrapped instance can be accessed through the `scikits_alg` attribute.

This class implements a meta estimator that fits a number of randomized decision trees (a.k.a. extra-trees) on various sub-samples of the dataset and use averaging to improve the predictive accuracy and control over-fitting.

Parameters

n_estimators [integer, optional (default=10)] The number of trees in the forest.

criterion [string, optional (default="gini")] The function to measure the quality of a split. Supported criteria are "gini" for the Gini impurity and "entropy" for the information gain. Note: this parameter is tree-specific.

max_depth [integer or None, optional (default=None)] The maximum depth of the tree. If None, then nodes are expanded until all leaves are pure or until all leaves contain less than min_samples_split samples. Note: this parameter is tree-specific.

min_samples_split [integer, optional (default=1)] The minimum number of samples required to split an internal node. Note: this parameter is tree-specific.

min_samples_leaf [integer, optional (default=1)] The minimum number of samples in newly created leaves. A split is discarded if after the split, one of the leaves would contain less then `min_samples_leaf` samples. Note: this parameter is tree-specific.

min_density [float, optional (default=0.1)] This parameter controls a trade-off in an optimization heuristic. It controls the minimum density of the *sample_mask* (i.e. the fraction of samples in the mask). If the density falls below this threshold the mask is recomputed and the input data is packed which results in data copying. If *min_density* equals to one, the partitions are always represented as copies of the original data. Otherwise, partitions are represented as bit masks (aka sample masks). Note: this parameter is tree-specific.

max_features [int, string or None, optional (default="auto")]

The number of features to consider when looking for the best split.

- If "auto", then *max_features=sqrt(n_features)* on classification tasks and *max_features=n_features* on regression problems.
- If "sqrt", then *max_features=sqrt(n_features)*.
- If "log2", then *max_features=log2(n_features)*.
- If None, then *max_features=n_features*.

Note: this parameter is tree-specific.

bootstrap [boolean, optional (default=False)] Whether bootstrap samples are used when building trees.

compute_importances [boolean, optional (default=True)] Whether feature importances are computed and stored into the `feature_importances_` attribute when calling fit.

oob_score [bool] Whether to use out-of-bag samples to estimate the generalization error.

n_jobs [integer, optional (default=1)] The number of jobs to run in parallel. If -1, then the number of jobs is set to the number of cores.

random_state [int, RandomState instance or None, optional (default=None)] If int, random_state is the seed used by the random number generator; If RandomState instance, random_state is the random number generator; If None, the random number generator is the RandomState instance used by *np.random*.

verbose [int, optional (default=0)] Controlls the verbosity of the tree building process.

Attributes

***estimators_*: list of DecisionTreeClassifier** The collection of fitted sub-estimators.

feature_importances_ [array of shape = [n_features]] The feature mportances (the higher, the more important the feature).

oob_score_ [float] Score of the training dataset obtained using an out-of-bag estimate.

oob_decision_function_ [array, shape = [n_samples, n_classes]] Decision function computed with out-of-bag estimate on the training set.

References

See also

sklearn.tree.ExtraTreeClassifier : Base classifier for this ensemble. RandomForestClassifier : Ensemble Classifier based on trees with optimal

splits.

Full API documentation: `ExtraTreesClassifierScikitsLearnNode`

class mdp.nodes.**LassoCVScikitsLearnNode**

Lasso linear model with iterative fitting along a regularization path

This node has been automatically generated by wrapping the sklearn.linear_model.coordinate_descent.Lass class from the sklearn library. The wrapped instance can be accessed through the scikits_alg attribute.

The best model is selected by cross-validation.

The optimization objective for Lasso is:

```
(1 / (2 * n_samples)) * ||y - Xw||^2_2 + alpha * ||w||_1
```

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

precompute [True | False | 'auto' | array-like] Whether to use a precomputed Gram matrix to speed up calculations. If set to 'auto' let us decide. The Gram matrix can also be passed as argument.

max_iter: int, optional The maximum number of iterations

tol: float, optional The tolerance for the optimization: if the updates are smaller than 'tol', the optimization code checks the dual gap for optimality and continues until it is smaller than tol.

cv [integer or crossvalidation generator, optional] If an integer is passed, it is the number of fold (default 3). Specific crossvalidation objects can be passed, see sklearn.cross_validation module for the list of possible objects

verbose [bool or integer] amount of verbosity

Attributes

***alpha_*: float** The amount of penalization choosen by cross validation

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

intercept_ [float] independent term in decision function.

***mse_path_*: array, shape = (n_alphas, n_folds)** mean square error for the test set on each fold, varying alpha

***alphas_*: numpy array** The grid of alphas used for fitting

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.

See also

lars_path lasso_path LassoLars Lasso LassoLarsCV

Full API documentation: LassoCVScikitsLearnNode

class mdp.nodes.**OneClassSVMScikitsLearnNode**

Unsupervised Outliers Detection.

This node has been automatically generated by wrapping the sklearn.svm.classes.OneClassSVM class from the sklearn library. The wrapped instance can be accessed through the scikits_alg attribute.

Estimate the support of a high-dimensional distribution.

The implementation is based on libsvm.

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 (default=0.0)] kernel coefficient for rbf and poly, if gamma is 0.0 then 1/n_features will be taken.

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

tol: float, optional Tolerance for stopping criterion.

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

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

verbose [bool, default: False] Enable verbose output. Note that this setting takes advantage of a per-process runtime setting in libsvm that, if enabled, may not work properly in a multithreaded context.

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.

*coef_* is readonly property derived from *dual_coef_* and *support_vectors_*

intercept_ [array, shape = [n_classes-1]] Constants in decision function.

Full API documentation: `OneClassSVMScikitsLearnNode`

class `mdp.nodes.`**`RidgeCVScikitsLearnNode`**

Ridge regression with built-in cross-validation.

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

By default, it performs Generalized Cross-Validation, which is a form of efficient Leave-One-Out cross-validation.

Parameters

alphas: numpy array of shape [n_alphas] Array of alpha values to try. Small positive values of alpha improve the conditioning of the problem and reduce the variance of the estimates. Alpha corresponds to `(2*C)^-1` in other linear models such as LogisticRegression or LinearSVC.

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).

normalize [boolean, optional] If True, the regressors X are normalized

score_func: callable, optional function that takes 2 arguments and compares them in order to evaluate the performance of prediction (big is good) if None is passed, the score of the estimator is maximized

loss_func: callable, optional function that takes 2 arguments and compares them in order to evaluate the performance of prediction (small is good) if None is passed, the score of the estimator is maximized

cv [cross-validation generator, optional] If None, Generalized Cross-Validation (efficient Leave-One-Out) will be used.

gcv_mode [{None, 'auto', 'svd', eigen'}, optional] Flag indicating which strategy to use when performing Generalized Cross-Validation. Options are:

```
'auto' : use svd if n_samples > n_features, otherwise use eigen
'svd' : force computation via singular value decomposition of X
'eigen' : force computation via eigendecomposition of X^T X
```

The 'auto' mode is the default and is intended to pick the cheaper option of the two depending upon the shape of the training data.

store_cv_values [boolean, default=False] Flag indicating if the cross-validation values corresponding to each alpha should be stored in the *cv_values_* attribute (see below). This flag is only compatible with *cv=None* (i.e. using Generalized Cross-Validation).

Attributes

cv_values_ [array, shape = [n_samples, n_alphas] or shape = [n_samples, n_responses, n_alphas], optional] Cross-validation values for each alpha (if *store_cv_values=True* and *cv=None*). After *fit()* has been called, this attribute will contain the mean squared errors (by default) or the values of the *{loss,score}_func* function (if provided in the constructor).

coef_ [array, shape = [n_features] or [n_responses, n_features]] Weight vector(s).

alpha_ [float] Estimated regularization parameter.

See also

Ridge: Ridge regression RidgeClassifier: Ridge classifier RidgeClassifierCV: Ridge classifier with built-in cross validation

Full API documentation: `RidgeCVScikitsLearnNode`

class `mdp.nodes.`**`PriorProbabilityEstimatorScikitsLearnNode`**
An estimator predicting the probability of each

This node has been automatically generated by wrapping the `sklearn.ensemble.gradient_boosting.PriorProb` class from the `sklearn` library. The wrapped instance can be accessed through the `scikits_alg` attribute.

Full API documentation: `PriorProbabilityEstimatorScikitsLearnNode`

class `mdp.nodes.`**`ARDRegressionScikitsLearnNode`**
Bayesian ARD regression.

This node has been automatically generated by wrapping the `sklearn.linear_model.bayes.ARDRegression` class from the `sklearn` 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 iterations. Default is 300

tol [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.

normalize [boolean, optional] If True, the regressors X are normalized

copy_X [boolean, optional, default True.] If True, X will be copied; else, it may be overwritten.

verbose [boolean, optional, default False] Verbose mode when fitting the model.

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)

Examples

```
>>> from sklearn import linear_model
>>> clf = linear_model.ARDRegression()
>>> clf.fit([[0,0], [1, 1], [2, 2]], [0, 1, 2])
...
ARDRegression(alpha_1=1e-06, alpha_2=1e-06, compute_score=False,
        copy_X=True, fit_intercept=True, lambda_1=1e-06, lambda_2=1e-06,
        n_iter=300, normalize=False, threshold_lambda=10000.0, tol=0.001,
        verbose=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.`**`GradientBoostingRegressorScikitsLearnNode`**

Gradient Boosting for regression.

This node has been automatically generated by wrapping the `sklearn.ensemble.gradient_boosting.GradientB` class from the `sklearn` library. The wrapped instance can be accessed through the `scikits_alg` attribute.

GB builds an additive model in a forward stage-wise fashion; it allows for the optimization of arbitrary differentiable loss functions. In each stage a regression tree is fit on the negative gradient of the given loss function.

Parameters

loss [{'ls', 'lad', 'huber', 'quantile'}, optional (default='ls')] loss function to be optimized. 'ls' refers to least squares regression. 'lad' (least absolute deviation) is a highly robust loss function soley based on order information of the input variables. 'huber' is a combination of the two. 'quantile' allows quantile regression (use *alpha* to specify the quantile).

learn_rate [float, optional (default=0.1)] learning rate shrinks the contribution of each tree by *learn_rate*. There is a trade-off between learn_rate and n_estimators.

n_estimators [int (default=100)] The number of boosting stages to perform. Gradient boosting is fairly robust to over-fitting so a large number usually results in better performance.

max_depth [integer, optional (default=3)] maximum depth of the individual regression estimators. The maximum depth limits the number of nodes in the tree. Tune this parameter for best performance; the best value depends on the interaction of the input variables.

min_samples_split [integer, optional (default=1)] The minimum number of samples required to split an internal node.

min_samples_leaf [integer, optional (default=1)] The minimum number of samples required to be at a leaf node.

subsample [float, optional (default=1.0)] The fraction of samples to be used for fitting the individual base learners. If smaller than 1.0 this results in Stochastic Gradient Boosting. *subsample* interacts with the parameter *n_estimators*. Choosing *subsample* < *1.0* leads to a reduction of variance and an increase in bias.

max_features [int, None, optional (default=None)] The number of features to consider when looking for the best split. Features are choosen randomly at each split point. If None, then *max_features=n_features*. Choosing *max_features* < *n_features* leads to a reduction of variance and an increase in bias.

alpha [float (default=0.9)] The alpha-quantile of the huber loss function and the quantile loss function. Only if `loss='huber'` or `loss='quantile'`.

Attributes

feature_importances_ [array, shape = [n_features]] The feature importances (the higher, the more important the feature).

oob_score_ [array, shape = [n_estimators]] Score of the training dataset obtained using an out-of-bag estimate. The i-th score `oob_score_[i]` is the deviance (= loss) of the model at iteration `i` on the out-of-bag sample.

train_score_ [array, shape = [n_estimators]] The i-th score `train_score_[i]` is the deviance (= loss) of the model at iteration `i` on the in-bag sample. If `subsample == 1` this is the deviance on the training data.

Examples

```
>>> samples = [[0, 0, 2], [1, 0, 0]]
>>> labels = [0, 1]
>>> from sklearn.ensemble import GradientBoostingRegressor
>>> gb = GradientBoostingRegressor().fit(samples, labels)
>>> print gb.predict([[0, 0, 0]])
[  1.32806...
```

See also

DecisionTreeRegressor, RandomForestRegressor

References

J. Friedman, Greedy Function Approximation: A Gradient Boosting Machine, The Annals of Statistics, Vol. 29, No. 5, 2001.

10.Friedman, Stochastic Gradient Boosting, 1999

T. Hastie, R. Tibshirani and J. Friedman. Elements of Statistical Learning Ed. 2, Springer, 2009.

Full API documentation: `GradientBoostingRegressorScikitsLearnNode`

class `mdp.nodes.`**`PLSCanonicalScikitsLearnNode`**

PLSCanonical implements the 2 blocks canonical PLS of the original Wold algorithm [Tenenhaus 1998] p.204, refered as PLS-C2A in [Wegelin 2000].

This node has been automatically generated by wrapping the `sklearn.pls.PLSCanonical` class from the `sklearn` library. The wrapped instance can be accessed through the `scikits_alg` attribute.

This class inherits from PLS with mode="A" and deflation_mode="canonical", norm_y_weights=True and algorithm="nipals", but svd should provide similar results up to numerical errors.

Parameters

X [array-like of predictors, shape = [n_samples, p]] Training vectors, where n_samples in the number of samples and p is the number of predictors.

Y [array-like of response, shape = [n_samples, q]] Training vectors, where n_samples in the number of samples and q is the number of response variables.

n_components : int, number of components to keep. (default 2).

scale : boolean, scale data? (default True)

algorithm [string, "nipals" or "svd"] The algorithm used to estimate the weights. It will be called n_components times, i.e. once for each iteration of the outer loop.

max_iter [an integer, (default 500)] the maximum number of iterations of the NIPALS inner loop (used only if algorithm="nipals")

tol [non-negative real, default 1e-06] the tolerance used in the iterative algorithm

copy [boolean, default True] Whether the deflation should be done on a copy. Let the default value to True unless you don't care about side effect

Attributes

x_weights_ [array, shape = [p, n_components]] X block weights vectors.

y_weights_ [array, shape = [q, n_components]] Y block weights vectors.

x_loadings_ [array, shape = [p, n_components]] X block loadings vectors.

y_loadings_ [array, shape = [q, n_components]] Y block loadings vectors.

x_scores_ [array, shape = [n_samples, n_components]] X scores.

y_scores_ [array, shape = [n_samples, n_components]] Y scores.

x_rotations_ [array, shape = [p, n_components]] X block to latents rotations.

y_rotations_ [array, shape = [q, n_components]] Y block to latents rotations.

Notes

For each component k, find weights u, v that optimize:

max corr(Xk u, Yk v) * var(Xk u) var(Yk u), such that `|u| = |v| = 1`

Note that it maximizes both the correlations between the scores and the intra-block variances.

The residual matrix of X (Xk+1) block is obtained by the deflation on the current X score: x_score.

The residual matrix of Y (Yk+1) block is obtained by deflation on the current Y score. This performs a canonical symetric version of the PLS regression. But slightly different than the CCA. This is mode mostly used for modeling.

This implementation provides the same results that the "plspm" package provided in the R language (R-project), using the function plsca(X, Y). Results are equal or colinear with the function `pls(..., mode = "canonical")` of the "mixOmics" package. The difference relies in the fact that mixOmics implmentation does not exactly implement the Wold algorithm since it does not normalize y_weights to one.

Examples

```
>>> from sklearn.pls import PLSCanonical, PLSRegression, CCA
>>> X = [[0., 0., 1.], [1.,0.,0.], [2.,2.,2.], [2.,5.,4.]]
>>> Y = [[0.1, -0.2], [0.9, 1.1], [6.2, 5.9], [11.9, 12.3]]
>>> plsca = PLSCanonical(n_components=2)
```

```
>>> plsca.fit(X, Y)
...
PLSCanonical(algorithm='nipals', copy=True, max_iter=500, n_components=2,
             scale=True, tol=1e-06)
>>> X_c, Y_c = plsca.transform(X, Y)
```

References

Jacob A. Wegelin. A survey of Partial Least Squares (PLS) methods, with emphasis on the two-block case. Technical Report 371, Department of Statistics, University of Washington, Seattle, 2000.

Tenenhaus, M. (1998). La regression PLS: theorie et pratique. Paris:

Editions Technic.

See also

CCA PLSSVD

Full API documentation: `PLSCanonicalScikitsLearnNode`

class `mdp.nodes.`**`SelectPercentileScikitsLearnNode`**

Filter: Select the best percentile of the p_values

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

Parameters

score_func: callable Function taking two arrays X and y, and returning 2 arrays:

- both scores and pvalues

percentile: int, optional Percent of features to keep

Full API documentation: `SelectPercentileScikitsLearnNode`

class `mdp.nodes.`**`RandomForestRegressorScikitsLearnNode`**

A random forest regressor.

This node has been automatically generated by wrapping the `sklearn.ensemble.forest.RandomForestRegresso` class from the `sklearn` library. The wrapped instance can be accessed through the `scikits_alg` attribute.

A random forest is a meta estimator that fits a number of classifical decision trees on various sub-samples of the dataset and use averaging to improve the predictive accuracy and control over-fitting.

Parameters

n_estimators [integer, optional (default=10)] The number of trees in the forest.

criterion [string, optional (default="mse")] The function to measure the quality of a split. The only supported criterion is "mse" for the mean squared error. Note: this parameter is tree-specific.

max_depth [integer or None, optional (default=None)] The maximum depth of the tree. If None, then nodes are expanded until all leaves are pure or until all leaves contain less than min_samples_split samples. Note: this parameter is tree-specific.

min_samples_split [integer, optional (default=1)] The minimum number of samples required to split an internal node. Note: this parameter is tree-specific.

min_samples_leaf [integer, optional (default=1)] The minimum number of samples in newly created leaves. A split is discarded if after the split, one of the leaves would contain less then `min_samples_leaf` samples. Note: this parameter is tree-specific.

min_density [float, optional (default=0.1)] This parameter controls a trade-off in an optimization heuristic. It controls the minimum density of the *sample_mask* (i.e. the fraction of samples in the mask). If the density falls below this threshold the mask is recomputed and the input data is packed which results in data copying. If *min_density* equals to one, the partitions are always represented as copies of the

original data. Otherwise, partitions are represented as bit masks (aka sample masks). Note: this parameter is tree-specific.

max_features [int, string or None, optional (default="auto")] The number of features to consider when looking for the best split:

- – If "auto", then *max_features=sqrt(n_features)* on
- classification tasks and *max_features=n_features*
- on regression problems.
- – If "sqrt", then *max_features=sqrt(n_features)*.
- – If "log2", then *max_features=log2(n_features)*.
- – If None, then *max_features=n_features*.

Note: this parameter is tree-specific.

bootstrap [boolean, optional (default=True)] Whether bootstrap samples are used when building trees.

compute_importances [boolean, optional (default=True)] Whether feature importances are computed and stored into the `feature_importances_` attribute when calling fit.

oob_score [bool] whether to use out-of-bag samples to estimate the generalization error.

n_jobs [integer, optional (default=1)] The number of jobs to run in parallel. If -1, then the number of jobs is set to the number of cores.

random_state [int, RandomState instance or None, optional (default=None)] If int, random_state is the seed used by the random number generator; If RandomState instance, random_state is the random number generator; If None, the random number generator is the RandomState instance used by *np.random*.

verbose [int, optional (default=0)] Controlls the verbosity of the tree building process.

Attributes

***estimators_*: list of DecisionTreeRegressor** The collection of fitted sub-estimators.

feature_importances_ [array of shape = [n_features]] The feature mportances (the higher, the more important the feature).

oob_score_ [float] Score of the training dataset obtained using an out-of-bag estimate.

oob_prediction_ [array, shape = [n_samples]] Prediction computed with out-of-bag estimate on the training set.

References

See also

DecisionTreeRegressor, ExtraTreesRegressor

Full API documentation: `RandomForestRegressorScikitsLearnNode`

class `mdp.nodes.`**`GaussianNBScikitsLearnNode`**

Gaussian Naive Bayes (GaussianNB)

This node has been automatically generated by wrapping the `sklearn.naive_bayes.GaussianNB` class from the `sklearn` 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

class_prior_ [array, shape = [n_classes]] probability of each class.

theta_ [array, shape = [n_classes, n_features]] mean of each feature per class

sigma_ [array, shape = [n_classes, n_features]] variance of each feature per class

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 sklearn.naive_bayes import GaussianNB
>>> clf = GaussianNB()
>>> clf.fit(X, Y)
GaussianNB()
>>> print(clf.predict([[-0.8, -1]]))
[1]
```

Full API documentation: `GaussianNBScikitsLearnNode`

class `mdp.nodes.`**`GaussianHMMScikitsLearnNode`**

Hidden Markov Model with Gaussian emissions

This node has been automatically generated by wrapping the `sklearn.hmm.GaussianHMM` class from the `sklearn` 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.

Parameters

n_components [int] Number of states.

`_covariance_type` [string] String describing the type of covariance parameters to use. Must be one of 'spherical', 'tied', 'diag', 'full'. Defaults to 'diag'.

Attributes

`_covariance_type` [string] String describing the type of covariance parameters used by the model. Must be one of 'spherical', 'tied', 'diag', 'full'.

n_features [int] Dimensionality of the Gaussian emissions.

n_components [int] Number of states in the model.

transmat [array, shape (*n_components*, *n_components*)] Matrix of transition probabilities between states.

startprob [array, shape ('n_components',)] Initial state occupation distribution.

means [array, shape (*n_components*, *n_features*)] Mean parameters for each state.

covars [array] Covariance parameters for each state. The shape depends on `_covariance_type`:

```
('n_components',)                   if 'spherical',
('n_features', 'n_features')          if 'tied',
('n_components', 'n_features')      if 'diag',
('n_components', 'n_features', 'n_features')  if 'full'
```

random_state: RandomState or an int seed (0 by default) A random number generator instance

n_iter [int, optional] Number of iterations to perform.

thresh [float, optional] Convergence threshold.

params [string, optional] Controls which parameters are updated in the training process. Can contain any combination of 's' for startprob, 't' for transmat, 'm' for means, and 'c' for covars, etc. Defaults to all parameters.

init_params [string, optional] Controls which parameters are initialized prior to training. Can contain any combination of 's' for startprob, 't' for transmat, 'm' for means, and 'c' for covars, etc. Defaults to all parameters.

Examples

```
>>> from sklearn.hmm import GaussianHMM
>>> GaussianHMM(n_components=2)
...
GaussianHMM(algorithm='viterbi',...
```

See Also

GMM : Gaussian mixture model

Full API documentation: `GaussianHMMScikitsLearnNode`

class `mdp.nodes.`**`LabelSpreadingScikitsLearnNode`**

LabelSpreading model for semi-supervised learning

This node has been automatically generated by wrapping the `sklearn.semi_supervised.label_propagation.La` class from the `sklearn` library. The wrapped instance can be accessed through the `scikits_alg` attribute.

This model is similar to the basic Label Propgation algorithm, but uses affinity matrix based on the normalized graph Laplacian and soft clamping across the labels.

Parameters

kernel [{'knn', 'rbf'}] String identifier for kernel function to use. Only 'rbf' and 'knn' kernels are currently supported.

gamma [float] parameter for rbf kernel

n_neighbors [integer > 0] parameter for knn kernel

alpha [float] clamping factor

max_iters [float] maximum number of iterations allowed

tol [float] Convergence tolerance: threshold to consider the system at steady state

Examples

```
>>> from sklearn import datasets
>>> from sklearn.semi_supervised import LabelSpreading
>>> label_prop_model = LabelSpreading()
>>> iris = datasets.load_iris()
>>> random_unlabeled_points = np.where(np.random.random_integers(0, 1,
...    size=len(iris.target)))
>>> labels = np.copy(iris.target)
>>> labels[random_unlabeled_points] = -1
>>> label_prop_model.fit(iris.data, labels)
...
LabelSpreading(...)
```

References

Dengyong Zhou, Olivier Bousquet, Thomas Navin Lal, Jason Weston, Bernhard Schölkopf. Learning with local and global consistency (2004) http://citeseer.ist.psu.edu/viewdoc/summary?doi=10.1.1.115.3219

See Also

LabelPropagation : Unregularized graph based semi-supervised learning

Full API documentation: `LabelSpreadingScikitsLearnNode`

class `mdp.nodes.`**`NMFScikitsLearnNode`**

Non-Negative matrix factorization by Projected Gradient (NMF)

This node has been automatically generated by wrapping the `sklearn.decomposition.nmf.NMF` class from the `sklearn` library. The wrapped instance can be accessed through the `scikits_alg` attribute.

Parameters

X: {array-like, sparse matrix}, shape = [n_samples, n_features] Data the model will be fit to.

n_components: int or None Number of components, if n_components is not set all components are kept

init: 'nndsvd' | 'nndsvda' | 'nndsvdar' | int | RandomState Method used to initialize the procedure. Default: 'nndsvdar' Valid options:

```
'nndsvd': Nonnegative Double Singular Value Decomposition (NNDSVD)
    initialization (better for sparseness)
'nndsvda': NNDSVD with zeros filled with the average of X
    (better when sparsity is not desired)
'nndsvdar': NNDSVD with zeros filled with small random values
    (generally faster, less accurate alternative to NNDSVDa
    for when sparsity is not desired)
int seed or RandomState: non-negative random matrices
```

sparseness: 'data' | 'components' | None, default: None Where to enforce sparsity in the model.

beta: double, default: 1 Degree of sparseness, if sparseness is not None. Larger values mean more sparseness.

eta: double, default: 0.1 Degree of correctness to mantain, if sparsity is not None. Smaller values mean larger error.

tol: double, default: 1e-4 Tolerance value used in stopping conditions.

max_iter: int, default: 200 Number of iterations to compute.

nls_max_iter: int, default: 2000 Number of iterations in NLS subproblem.

Attributes

components_ [array, [n_components, n_features]] Non-negative components of the data

reconstruction_err_ [number] Frobenius norm of the matrix difference between the training data and the reconstructed data from the fit produced by the model. `|| X - WH ||_2` Not computed for sparse input matrices because it is too expensive in terms of memory.

Examples

```
>>> import numpy as np
>>> X = np.array([[1,1], [2, 1], [3, 1.2], [4, 1], [5, 0.8], [6, 1]])
>>> from sklearn.decomposition import ProjectedGradientNMF
>>> model = ProjectedGradientNMF(n_components=2, init=0)
>>> model.fit(X)
ProjectedGradientNMF(beta=1, eta=0.1, init=0, max_iter=200, n_components=2,
                     nls_max_iter=2000, sparseness=None, tol=0.0001)
>>> model.components_
array([[ 0.77032744,  0.11118662],
       [ 0.38526873,  0.38228063]])
>>> model.reconstruction_err_
0.00746...
>>> model = ProjectedGradientNMF(n_components=2, init=0,
...                              sparseness='components')
>>> model.fit(X)
ProjectedGradientNMF(beta=1, eta=0.1, init=0, max_iter=200, n_components=2,
           nls_max_iter=2000, sparseness='components', tol=0.0001)
>>> model.components_
array([[ 1.67481991,  0.29614922],
       [-0.        ,  0.4681982 ]])
>>> model.reconstruction_err_
0.513...
```

Notes

This implements

C.-J. Lin. Projected gradient methods for non-negative matrix factorization. Neural Computation, 19(2007), 2756-2779. http://www.csie.ntu.edu.tw/~cjlin/nmf/

P. Hoyer. Non-negative Matrix Factorization with Sparseness Constraints. Journal of Machine Learning Research 2004.

NNDSVD is introduced in

C. Boutsidis, E. Gallopoulos: SVD based initialization: A head start for nonnegative matrix factorization - Pattern Recognition, 2008 http://www.cs.rpi.edu/~boutsc/files/nndsvd.pdf

Full API documentation: `NMFScikitsLearnNode`

class `mdp.nodes.`**`SparseBaseLibSVMScikitsLearnNode`**
Full API documentation: `SparseBaseLibSVMScikitsLearnNode`

class `mdp.nodes.`**`DPGMMScikitsLearnNode`**
Variational Inference for the Infinite Gaussian Mixture Model.

This node has been automatically generated by wrapping the `sklearn.mixture.dpgmm.DPGMM` class from the `sklearn` library. The wrapped instance can be accessed through the `scikits_alg` attribute.

DPGMM stands for Dirichlet Process Gaussian Mixture Model, and it is an infinite mixture model with the Dirichlet Process as a prior distribution on the number of clusters. In practice the approximate inference algorithm uses a truncated distribution with a fixed maximum number of components, but almost always the number of components actually used depends on the data.

Stick-breaking Representation of a Gaussian mixture model probability distribution. This class allows for easy and efficient inference of an approximate posterior distribution over the parameters of a Gaussian mixture model with a variable number of components (smaller than the truncation parameter n_components).

Initialization is with normally-distributed means and identity covariance, for proper convergence.

Parameters

n_components: int, optional Number of mixture components. Defaults to 1.

covariance_type: string, optional String describing the type of covariance parameters to use. Must be one of 'spherical', 'tied', 'diag', 'full'. Defaults to 'diag'.

alpha: float, optional Real number representing the concentration parameter of the dirichlet process. Intuitively, the Dirichlet Process is as likely to start a new cluster for a point as it is to add that point to a cluster with alpha elements. A higher alpha means more clusters, as the expected number of clusters is `alpha*log(N)`. Defaults to 1.

thresh [float, optional] Convergence threshold.

n_iter [int, optional] Maximum number of iterations to perform before convergence.

params [string, optional] Controls which parameters are updated in the training process. Can contain any combination of 'w' for weights, 'm' for means, and 'c' for covars. Defaults to 'wmc'.

init_params [string, optional] Controls which parameters are updated in the initialization process. Can contain any combination of 'w' for weights, 'm' for means, and 'c' for covars. Defaults to 'wmc'.

Attributes

covariance_type [string] String describing the type of covariance parameters used by the DP-GMM. Must be one of 'spherical', 'tied', 'diag', 'full'.

n_components [int] Number of mixture components.

weights_ [array, shape (*n_components*,)] Mixing weights for each mixture component.

means_ [array, shape (*n_components*, *n_features*)] Mean parameters for each mixture component.

precisions_ [array] Precision (inverse covariance) parameters for each mixture component. The shape depends on *covariance_type*:

```
('n_components', 'n_features')                if 'spherical',
('n_features', 'n_features')                  if 'tied',
('n_components', 'n_features')                if 'diag',
('n_components', 'n_features', 'n_features')  if 'full'
```

converged_ [bool] True when convergence was reached in fit(), False otherwise.

See Also

GMM : Finite Gaussian mixture model fit with EM

VBGMM [Finite Gaussian mixture model fit with a variational] algorithm, better for situations where there might be too little data to get a good estimate of the covariance matrix.

Full API documentation: `DPGMMScikitsLearnNode`

class `mdp.nodes.`**`SVCScikitsLearnNode`**

C-Support Vector Classification.

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

The implementations is a based on libsvm. The fit time complexity is more than quadratic with the number of samples which makes it hard to scale to dataset with more than a couple of 10000 samples.

The multiclass support is handled according to a one-vs-one scheme.

For details on the precise mathematical formulation of the provided kernel functions and how *gamma*, *coef0* and *degree* affect each, see the corresponding section in the narrative documentation:

svm_kernels.

Parameters

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

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

degree [int, optional (default=3)] Degree of kernel function. It is significant only in 'poly' and 'sigmoid'.

gamma [float, optional (default=0.0)] Kernel coefficient for 'rbf' and 'poly'. If gamma is 0.0 then 1/n_features will be used instead.

coef0 [float, optional (default=0.0)] Independent term in kernel function. It is only significant in 'poly' and 'sigmoid'.

probability: boolean, optional (default=False) Whether to enable probability estimates. This must be enabled prior to calling predict_proba.

shrinking: boolean, optional (default=True) Whether to use the shrinking heuristic.

tol: float, optional (default=1e-3) Tolerance for stopping criterion.

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

class_weight [{dict, 'auto'}, optional] Set the parameter C of class i to class_weight[i]*C for SVC. If not given, all classes are supposed to have weight one. The 'auto' mode uses the values of y to automatically adjust weights inversely proportional to class frequencies.

verbose [bool, default: False] Enable verbose output. Note that this setting takes advantage of a per-process runtime setting in libsvm that, if enabled, may not work properly in a multithreaded context.

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. For multiclass, coefficient for all 1-vs-1 classifiers. The layout of the coefficients in the multiclass case is somewhat non-trivial. See the section about multi-class classification in the SVM section of the User Guide for details.

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.

*coef_* is readonly property derived from *dual_coef_* and *support_vectors_*

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 sklearn.svm import SVC
>>> clf = SVC()
>>> clf.fit(X, y)
SVC(C=1.0, cache_size=200, class_weight=None, coef0=0.0, degree=3,
        gamma=0.0, kernel='rbf', probability=False, shrinking=True,
        tol=0.001, verbose=False)
>>> print(clf.predict([[-0.8, -1]]))
[ 1.]
```

See also

SVR Support Vector Machine for Regression implemented using libsvm.

LinearSVC Scalable Linear Support Vector Machine for classififcation implemented using liblinear. Check the See also section of LinearSVC for more comparison element.

Full API documentation: `SVCScikitsLearnNode`

class `mdp.nodes.`**`VBGMMScikitsLearnNode`**

Variational Inference for the Gaussian Mixture Model

This node has been automatically generated by wrapping the `sklearn.mixture.dpgmm.VBGMM` class from the `sklearn` library. The wrapped instance can be accessed through the `scikits_alg` attribute.

Variational inference for a Gaussian mixture model probability distribution. This class allows for easy and efficient inference of an approximate posterior distribution over the parameters of a Gaussian mixture model with a fixed number of components.

Initialization is with normally-distributed means and identity covariance, for proper convergence.

Parameters

n_components: int, optional Number of mixture components. Defaults to 1.

covariance_type: string, optional String describing the type of covariance parameters to use. Must be one of 'spherical', 'tied', 'diag', 'full'. Defaults to 'diag'.

alpha: float, optional Real number representing the concentration parameter of the dirichlet distribution. Intuitively, the higher the value of alpha the more likely the variational mixture of Gaussians model will use all components it can. Defaults to 1.

Attributes

covariance_type [string] String describing the type of covariance parameters used by the DP-GMM. Must be one of 'spherical', 'tied', 'diag', 'full'.

n_features [int] Dimensionality of the Gaussians.

n_components [int (read-only)] Number of mixture components.

weights_ [array, shape (*n_components*,)] Mixing weights for each mixture component.

means_ [array, shape (*n_components*, *n_features*)] Mean parameters for each mixture component.

precisions_ [array] Precision (inverse covariance) parameters for each mixture component. The shape depends on *covariance_type*:

```
('n_components', 'n_features')                if 'spherical',
('n_features', 'n_features')                  if 'tied',
('n_components', 'n_features')                if 'diag',
('n_components', 'n_features', 'n_features')  if 'full'
```

converged_ [bool] True when convergence was reached in fit(), False otherwise.

See Also

GMM : Finite Gaussian mixture model fit with EM DPGMM : Ininite Gaussian mixture model, using the dirichlet

process, fit with a variational algorithm

Full API documentation: `VBGMMScikitsLearnNode`

class `mdp.nodes.`**`DictVectorizerScikitsLearnNode`**

Transforms lists of feature-value mappings to vectors.

This node has been automatically generated by wrapping the `sklearn.feature_extraction.dict_vectorizer.D` class from the `sklearn` library. The wrapped instance can be accessed through the `scikits_alg` attribute.

This transformer turns lists of mappings (dict-like objects) of feature names to feature values into Numpy arrays or scipy.sparse matrices for use with scikit-learn estimators.

When feature values are strings, this transformer will do a binary one-hot (aka one-of-K) coding: one boolean-valued feature is constructed for each of the possible string values that the feature can take on. For instance, a feature "f" that can take on the values "ham" and "spam" will become two features in the output, one signifying "f=ham", the other "f=spam".

Features that do not occur in a sample (mapping) will have a zero value in the resulting array/matrix.

Parameters

dtype [callable, optional] The type of feature values. Passed to Numpy array/scipy.sparse matrix constructors as the dtype argument.

separator: string, optional Separator string used when constructing new features for one-hot coding.

sparse: boolean, optional. Whether transform should produce scipy.sparse matrices. True by default.

Examples

```
>>> from sklearn.feature_extraction import DictVectorizer
>>> v = DictVectorizer(sparse=False)
>>> D = [{'foo': 1, 'bar': 2}, {'foo': 3, 'baz': 1}]
>>> X = v.fit_transform(D)
>>> X
array([[ 2.,  0.,  1.],
       [ 0.,  1.,  3.]])
>>> v.inverse_transform(X) ==         [{'bar': 2.0, 'foo': 1.0}, {'baz': 1.0, 'foo': 3.0}]
True
>>> v.transform({'foo': 4, 'unseen_feature': 3})
array([[ 0.,  0.,  4.]])
```

Full API documentation: `DictVectorizerScikitsLearnNode`

class `mdp.nodes.`**`LinearSVCScikitsLearnNode`**

Full API documentation: `LinearSVCScikitsLearnNode`

class `mdp.nodes.`**`RandomizedLassoScikitsLearnNode`**

Randomized Lasso

This node has been automatically generated by wrapping the `sklearn.linear_model.randomized_l1.Randomize` class from the `sklearn` library. The wrapped instance can be accessed through the `scikits_alg` attribute.

Randomized Lasso works by resampling the train data and computing a Lasso on each resampling. In short, the features selected more often are good features. It is also known as stability selection.

Parameters

alpha [float, 'aic', or 'bic'] The regularization parameter alpha parameter in the Lasso. Warning: this is not the alpha parameter in the stability selection article which is scaling.

scaling [float] The alpha parameter in the stability selection article used to randomly scale the features. Should be between 0 and 1.

sample_fraction [float] The fraction of samples to be used in each randomized design. Should be between 0 and 1. If 1, all samples are used.

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).

verbose [boolean or integer, optional] Sets the verbosity amount

normalize [boolean, optional] If True, the regressors X are normalized

precompute [True | False | 'auto'] Whether to use a precomputed Gram matrix to speed up calculations. If set to 'auto' let us decide. The Gram matrix can also be passed as argument.

max_iter [integer, optional] Maximum number of iterations to perform in the Lars algorithm.

eps [float, optional] The machine-precision regularization in the computation of the Cholesky diagonal factors. Increase this for very ill-conditioned systems. Unlike the 'tol' parameter in some iterative optimization-based algorithms, this parameter does not control the tolerance of the optimization.

n_jobs [integer, optional] Number of CPUs to use during the resampling. If '-1', use all the CPUs

random_state [int, RandomState instance or None, optional (default=None)] If int, random_state is the seed used by the random number generator; If RandomState instance, random_state is the random number generator; If None, the random number generator is the RandomState instance used by *np.random*.

pre_dispatch [int, or string, optional] Controls the number of jobs that get dispatched during parallel execution. Reducing this number can be useful to avoid an explosion of memory consumption when more jobs get dispatched than CPUs can process. This parameter can be:

- None, in which case all the jobs are immediatly created and spawned. Use this for lightweight and fast-running jobs, to avoid delays due to on-demand spawning of the jobs
- An int, giving the exact number of total jobs that are spawned
- A string, giving an expression as a function of n_jobs, as in '2*n_jobs'

memory [Instance of joblib.Memory or string] Used for internal caching. By default, no caching is done. If a string is given, it is thepath to the caching directory.

Attributes

scores_ [array, shape = [n_features]] Feature scores between 0 and 1.

all_scores_ [array, shape = [n_features, n_reg_parameter]] Feature scores between 0 and 1 for all values of the regularization parameter. The reference article suggests `scores_` is the max of `all_scores_`.

Examples

```
>>> from sklearn.linear_model import RandomizedLasso
>>> randomized_lasso = RandomizedLasso()
```

Notes

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

References

Stability selection Nicolai Meinshausen, Peter Buhlmann Journal of the Royal Statistical Society: Series B Volume 72, Issue 4, pages 417-473, September 2010 DOI: 10.1111/j.1467-9868.2010.00740.x

See also

RandomizedLogisticRegression, LogisticRegression

Full API documentation: `RandomizedLassoScikitsLearnNode`

class `mdp.nodes.`**`MultinomialNBScikitsLearnNode`**

Naive Bayes classifier for multinomial models

This node has been automatically generated by wrapping the `sklearn.naive_bayes.MultinomialNB` class from the `sklearn` library. The wrapped instance can be accessed through the `scikits_alg` attribute.

The multinomial Naive Bayes classifier is suitable for classification with discrete features (e.g., word counts for text classification). The multinomial distribution normally requires integer feature counts. However, in practice, fractional counts such as tf-idf may also work.

Parameters

alpha: float, optional (default=1.0) Additive (Laplace/Lidstone) smoothing parameter (0 for no smoothing).

fit_prior: boolean Whether to learn class prior probabilities or not. If false, a uniform prior will be used.

Attributes

intercept_, class_log_prior_ [array, shape = [n_classes]] Smoothed empirical log probability for each class.

feature_log_prob_, coef_ [array, shape = [n_classes, n_features]] Empirical log probability of features given a class, P(x_ily).

(*intercept_* and *coef_* are properties referring to *class_log_prior_* and *feature_log_prob_*, respectively.)

Examples

```
>>> import numpy as np
>>> X = np.random.randint(5, size=(6, 100))
>>> Y = np.array([1, 2, 3, 4, 5, 6])
>>> from sklearn.naive_bayes import MultinomialNB
>>> clf = MultinomialNB()
>>> clf.fit(X, Y)
MultinomialNB(alpha=1.0, fit_prior=True)
>>> print(clf.predict(X[2]))
[3]
```

Notes

For the rationale behind the names *coef_* and *intercept_*, i.e. naive Bayes as a linear classifier, see J. Rennie et al. (2003), Tackling the poor assumptions of naive Bayes text classifiers, ICML.

Full API documentation: `MultinomialNBScikitsLearnNode`

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

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

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

The optimization objective for Lasso is:

```
(1 / (2 * n_samples)) * ||y - Xw||^2_2 + alpha * ||w||_1
```

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).

normalize [boolean, optional] If True, the regressors X are normalized

copy_X [boolean, optional, default True] If True, X will be copied; else, it may be overwritten.

precompute [True | False | 'auto' | array-like] Whether to use a precomputed Gram matrix to speed up calculations. If set to 'auto' let us decide. The Gram matrix can also be passed as argument. For sparse input this option is always True to preserve sparsity.

max_iter: int, optional The maximum number of iterations

tol [float, optional] The tolerance for the optimization: if the updates are smaller than 'tol', the optimization code checks the dual gap for optimality and continues until it is smaller than tol.

warm_start [bool, optional] When set to True, reuse the solution of the previous call to fit as initialization, otherwise, just erase the previous solution.

positive [bool, optional] When set to True, forces the coefficients to be positive.

Attributes

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

sparse_coef_ [scipy.sparse matrix, shape = (n_features, 1)] *sparse_coef_* is a readonly property derived from *coef_*

intercept_ [float] independent term in decision function.

Examples

```
>>> from sklearn 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, copy_X=True, fit_intercept=True, max_iter=1000,
   normalize=False, positive=False, precompute='auto', tol=0.0001,
   warm_start=False)
>>> print(clf.coef_)
[ 0.85  0.  ]
>>> print(clf.intercept_)
0.15
```

See also

lars_path lasso_path LassoLars LassoCV LassoLarsCV sklearn.decomposition.sparse_encode

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.`**`LocallyLinearEmbeddingScikitsLearnNode`**

Locally Linear Embedding

This node has been automatically generated by wrapping the `sklearn.manifold.locally_linear.LocallyLinea` class from the `sklearn` library. The wrapped instance can be accessed through the `scikits_alg` attribute.

Parameters

n_neighbors [integer] number of neighbors to consider for each point.

n_components [integer] number of coordinates for the manifold

reg [float] regularization constant, multiplies the trace of the local covariance matrix of the distances.

eigen_solver [string, {'auto', 'arpack', 'dense'}] auto : algorithm will attempt to choose the best method for input data

arpack [use arnoldi iteration in shift-invert mode.] For this method, M may be a dense matrix, sparse matrix, or general linear operator. Warning: ARPACK can be unstable for some problems. It is best to try several random seeds in order to check results.

dense [use standard dense matrix operations for the eigenvalue] decomposition. For this method, M must be an array or matrix type. This method should be avoided for large problems.

tol [float, optional] Tolerance for 'arpack' method Not used if eigen_solver=='dense'.

max_iter [integer] maximum number of iterations for the arpack solver. Not used if eigen_solver=='dense'.

method [string ['standard' | 'hessian' | 'modified']]

standard [use the standard locally linear embedding algorithm.] see reference [1]

hessian [use the Hessian eigenmap method. This method requires] n_neighbors > n_components * (1 + (n_components + 1) / 2. see reference [2]

modified [use the modified locally linear embedding algorithm.] see reference [3]

ltsa [use local tangent space alignment algorithm] see reference [4]

hessian_tol [float, optional] Tolerance for Hessian eigenmapping method. Only used if method == 'hessian'

modified_tol [float, optional] Tolerance for modified LLE method. Only used if method == 'modified'

neighbors_algorithm [string ['auto'|'brute'|'kd_tree'|'ball_tree']] algorithm to use for nearest neighbors search, passed to neighbors.NearestNeighbors instance

random_state: numpy.RandomState or int, optional The generator or seed used to determine the starting vector for arpack iterations. Defaults to numpy.random.

Attributes

embedding_vectors_ [array-like, shape [n_components, n_samples]] Stores the embedding vectors

reconstruction_error_ [float] Reconstruction error associated with *embedding_vectors_*

nbrs_ [NearestNeighbors object] Stores nearest neighbors instance, including BallTree or KDtree if applicable.

References

Full API documentation: `LocallyLinearEmbeddingScikitsLearnNode`

class `mdp.nodes.`**`LarsCVScikitsLearnNode`**

Cross-validated Least Angle Regression model

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

Parameters

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).

verbose [boolean or integer, optional] Sets the verbosity amount

normalize [boolean, optional] If True, the regressors X are normalized

copy_X [boolean, optional, default True] If True, X will be copied; else, it may be overwritten.

precompute [True | False | 'auto' | array-like] Whether to use a precomputed Gram matrix to speed up calculations. If set to 'auto' let us decide. The Gram matrix can also be passed as argument.

max_iter: integer, optional Maximum number of iterations to perform.

cv [crossvalidation generator, optional] see sklearn.cross_validation module. If None is passed, default to a 5-fold strategy

max_n_alphas [integer, optional] The maximum number of points on the path used to compute the residuals in the cross-validation

n_jobs [integer, optional] Number of CPUs to use during the cross validation. If '-1', use all the CPUs

eps: float, optional The machine-precision regularization in the computation of the Cholesky diagonal factors. Increase this for very ill-conditioned systems.

Attributes

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

intercept_ [float] independent term in decision function

***coef_path_*: array, shape = [n_features, n_alpha]** the varying values of the coefficients along the path

***alpha_*: float** the estimated regularization parameter alpha

***alphas_*: array, shape = [n_alpha]** the different values of alpha along the path

***cv_alphas_*: array, shape = [n_cv_alphas]** all the values of alpha along the path for the different folds

***cv_mse_path_*: array, shape = [n_folds, n_cv_alphas]** the mean square error on left-out for each fold along the path (alpha values given by cv_alphas)

See also

lars_path, LassoLARS, LassoLarsCV

Full API documentation: `LarsCVScikitsLearnNode`

class `mdp.nodes.`**`LDAScikitsLearnNode`**

Linear Discriminant Analysis (LDA)

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

A classifier with a linear decision boundary, generated by fitting class conditional densities to the data and using Bayes' rule.

The model fits a Gaussian density to each class, assuming that all classes share the same covariance matrix.

The fitted model can also be used to reduce the dimensionality of the input, by projecting it to the most discriminative directions.

Parameters

n_components: int Number of components (< n_classes - 1) for dimensionality reduction

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 sklearn.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(n_components=None, priors=None)
>>> print(clf.predict([[-0.8, -1]]))
[1]
```

See also

sklearn.qda.QDA: Quadratic discriminant analysis

Full API documentation: `LDAScikitsLearnNode`

class `mdp.nodes.`**`QuantileEstimatorScikitsLearnNode`**

This node has been automatically generated by wrapping the `sklearn.ensemble.gradient_boosting.QuantileE` class from the `sklearn` library. The wrapped instance can be accessed through the `scikits_alg` attribute.

Full API documentation: `QuantileEstimatorScikitsLearnNode`

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 `sklearn.feature_extraction.text.CountVectori` class from the `sklearn` library. The wrapped instance can be accessed through the `scikits_alg` attribute.

This implementation produces a sparse representation of the counts using scipy.sparse.coo_matrix.

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 will be equal to the vocabulary size found by analysing the data. The default analyzer does simple stop word filtering for English.

Parameters

input: string {'filename', 'file', 'content'} If filename, the sequence passed as an argument to fit is expected to be a list of filenames that need reading to fetch the raw content to analyze.

If 'file', the sequence items must have 'read' method (file-like object) it is called to fetch the bytes in memory.

Otherwise the input is expected to be the sequence strings or bytes items are expected to be analyzed directly.

charset: string, 'utf-8' by default. If bytes or files are given to analyze, this charset is used to decode.

charset_error: {'strict', 'ignore', 'replace'} Instruction on what to do if a byte sequence is given to analyze that contains characters not of the given *charset*. By default, it is 'strict', meaning that a UnicodeDecodeError will be raised. Other values are 'ignore' and 'replace'.

strip_accents: {'ascii', 'unicode', None} Remove accents during the preprocessing step. 'ascii' is a fast method that only works on characters that have an direct ASCII mapping. 'unicode' is a slightly slower method that works on any characters. None (default) does nothing.

analyzer: string, {'word', 'char', 'char_wb'} or callable Whether the feature should be made of word or character n-grams. Option 'char_wb' creates character n-grams only from text inside word boundaries.

If a callable is passed it is used to extract the sequence of features out of the raw, unprocessed input.

preprocessor: callable or None (default) Override the preprocessing (string transformation) stage while preserving the tokenizing and n-grams generation steps.

tokenizer: callable or None (default) Override the string tokenization step while preserving the preprocessing and n-grams generation steps.

ngram_range: tuple (min_n, max_n) The lower and upper boundary of the range of n-values for different n-grams to be extracted. All values of n such that min_n <= n <= max_n will be used.

stop_words: string {'english'}, list, or None (default) If a string, it is passed to _check_stop_list and the appropriate stop list is returned is currently the only supported string value.

If a list, that list is assumed to contain stop words, all of which will be removed from the resulting tokens.

If None, no stop words will be used. max_df can be set to a value in the range [0.7, 1.0) to automatically detect and filter stop words based on intra corpus document frequency of terms.

lowercase: boolean, default True Convert all characters to lowercase befor tokenizing.

token_pattern: string Regular expression denoting what constitutes a "token", only used if *tokenize == 'word'*. The default regexp select tokens of 2 or more letters characters (punctuation is completely ignored and always treated as a token separator).

max_df [float in range [0.0, 1.0] or int, optional, 1.0 by default] When building the vocabulary ignore terms that have a term frequency strictly higher than the given threshold (corpus specific stop words). If float, the parameter represents a proportion of documents, integer absolute counts. This parameter is ignored if vocabulary is not None.

min_df [float in range [0.0, 1.0] or int, optional, 2 by default] When building the vocabulary ignore terms that have a term frequency strictly lower than the given threshold. This value is also called cut-off in the literature. If float, the parameter represents a proportion of documents, integer absolute counts. This parameter is ignored if vocabulary is not None.

max_features [optional, None by default] If not None, build a vocabulary that only consider the top max_features ordered by term frequency across the corpus.

This parameter is ignored if vocabulary is not None.

vocabulary: Mapping or iterable, optional Either a Mapping (e.g., a dict) where keys are terms and values are indices in the feature matrix, or an iterable over terms. If not given, a vocabulary is determined from the input documents.

binary: boolean, False by default. If True, all non zero counts are set to 1. This is useful for discrete probabilistic models that model binary events rather than integer counts.

dtype: type, optional Type of the matrix returned by fit_transform() or transform().

Full API documentation: `CountVectorizerScikitsLearnNode`

class `mdp.nodes.`**`ExtraTreesRegressorScikitsLearnNode`**

An extra-trees regressor.

This node has been automatically generated by wrapping the `sklearn.ensemble.forest.ExtraTreesRegressor` class from the `sklearn` library. The wrapped instance can be accessed through the `scikits_alg` attribute.

This class implements a meta estimator that fits a number of randomized decision trees (a.k.a. extra-trees) on various sub-samples of the dataset and use averaging to improve the predictive accuracy and control over-fitting.

Parameters

n_estimators [integer, optional (default=10)] The number of trees in the forest.

criterion [string, optional (default="mse")] The function to measure the quality of a split. The only supported criterion is "mse" for the mean squared error. Note: this parameter is tree-specific.

max_depth [integer or None, optional (default=None)] The maximum depth of the tree. If None, then nodes are expanded until all leaves are pure or until all leaves contain less than min_samples_split samples. Note: this parameter is tree-specific.

min_samples_split [integer, optional (default=1)] The minimum number of samples required to split an internal node. Note: this parameter is tree-specific.

min_samples_leaf [integer, optional (default=1)] The minimum number of samples in newly created leaves. A split is discarded if after the split, one of the leaves would contain less then `min_samples_leaf` samples. Note: this parameter is tree-specific.

min_density [float, optional (default=0.1)] This parameter controls a trade-off in an optimization heuristic. It controls the minimum density of the *sample_mask* (i.e. the fraction of samples in the mask). If the density falls below this threshold the mask is recomputed and the input data is packed which results in data copying. If *min_density* equals to one, the partitions are always represented as copies of the original data. Otherwise, partitions are represented as bit masks (aka sample masks). Note: this parameter is tree-specific.

max_features [int, string or None, optional (default="auto")] The number of features to consider when looking for the best split:

- – If "auto", then *max_features=sqrt(n_features)* on
- classification tasks and *max_features=n_features*
- on regression problems.
- – If "sqrt", then *max_features=sqrt(n_features)*.
- – If "log2", then *max_features=log2(n_features)*.
- – If None, then *max_features=n_features*.

Note: this parameter is tree-specific.

bootstrap [boolean, optional (default=False)] Whether bootstrap samples are used when building trees. Note: this parameter is tree-specific.

compute_importances [boolean, optional (default=True)] Whether feature importances are computed and stored into the `feature_importances_` attribute when calling fit.

oob_score [bool] Whether to use out-of-bag samples to estimate the generalization error.

n_jobs [integer, optional (default=1)] The number of jobs to run in parallel. If -1, then the number of jobs is set to the number of cores.

random_state [int, RandomState instance or None, optional (default=None)] If int, random_state is the seed used by the random number generator; If RandomState instance, random_state is the random number generator; If None, the random number generator is the RandomState instance used by *np.random*.

verbose [int, optional (default=0)] Controlls the verbosity of the tree building process.

Attributes

***estimators_*: list of DecisionTreeRegressor** The collection of fitted sub-estimators.

feature_importances_ [array of shape = [n_features]] The feature mportances (the higher, the more important the feature).

oob_score_ [float] Score of the training dataset obtained using an out-of-bag estimate.

oob_prediction_ [array, shape = [n_samples]] Prediction computed with out-of-bag estimate on the training set.

References

See also

sklearn.tree.ExtraTreeRegressor: Base estimator for this ensemble. RandomForestRegressor: Ensemble regressor using trees with optimal splits.

Full API documentation: `ExtraTreesRegressorScikitsLearnNode`

class `mdp.nodes.`**`MultinomialHMMScikitsLearnNode`**

Hidden Markov Model with multinomial (discrete) emissions

This node has been automatically generated by wrapping the `sklearn.hmm.MultinomialHMM` class from the `sklearn` library. The wrapped instance can be accessed through the `scikits_alg` attribute.

Attributes

n_components [int] Number of states in the model.

n_symbols [int] Number of possible symbols emitted by the model (in the observations).

transmat [array, shape (*n_components*, *n_components*)] Matrix of transition probabilities between states.

startprob [array, shape ('n_components',)] Initial state occupation distribution.

emissionprob [array, shape ('n_components', 'n_symbols')] Probability of emitting a given symbol when in each state.

random_state: RandomState or an int seed (0 by default) A random number generator instance

n_iter [int, optional] Number of iterations to perform.

thresh [float, optional] Convergence threshold.

params [string, optional] Controls which parameters are updated in the training process. Can contain any combination of 's' for startprob, 't' for transmat, 'm' for means, and 'c' for covars, etc. Defaults to all parameters.

init_params [string, optional] Controls which parameters are initialized prior to training. Can contain any combination of 's' for startprob, 't' for transmat, 'm' for means, and 'c' for covars, etc. Defaults to all parameters.

Examples

```
>>> from sklearn.hmm import MultinomialHMM
>>> MultinomialHMM(n_components=2)
...
MultinomialHMM(algorithm='viterbi',...
```

See Also

GaussianHMM : HMM with Gaussian emissions

Full API documentation: `MultinomialHMMScikitsLearnNode`

class `mdp.nodes.`**`LabelPropagationScikitsLearnNode`**

Label Propagation classifier

This node has been automatically generated by wrapping the `sklearn.semi_supervised.label_propagation.La` class from the `sklearn` library. The wrapped instance can be accessed through the `scikits_alg` attribute.

Parameters

kernel [{'knn', 'rbf'}] String identifier for kernel function to use. Only 'rbf' and 'knn' kernels are currently supported..

gamma [float] parameter for rbf kernel

n_neighbors [integer > 0] parameter for knn kernel

alpha [float] clamping factor

max_iters [float] change maximum number of iterations allowed

tol [float] Convergence tolerance: threshold to consider the system at steady state

Examples

```
>>> from sklearn import datasets
>>> from sklearn.semi_supervised import LabelPropagation
>>> label_prop_model = LabelPropagation()
>>> iris = datasets.load_iris()
```

```
>>> random_unlabeled_points = np.where(np.random.random_integers(0, 1,
...    size=len(iris.target)))
>>> labels = np.copy(iris.target)
>>> labels[random_unlabeled_points] = -1
>>> label_prop_model.fit(iris.data, labels)
...
LabelPropagation(...)
```

References

Xiaojin Zhu and Zoubin Ghahramani. Learning from labeled and unlabeled data with label propagation. Technical Report CMU-CALD-02-107, Carnegie Mellon University, 2002 http://pages.cs.wisc.edu/~jerryzhu/pub/CMU-CALD-02-107.pdf

See Also

LabelSpreading : Alternate label proagation strategy more robust to noise

Full API documentation: `LabelPropagationScikitsLearnNode`

class `mdp.nodes.`**`GaussianProcessScikitsLearnNode`**

The Gaussian Process model class.

This node has been automatically generated by wrapping the `sklearn.gaussian_process.gaussian_process.Ga` class from the `sklearn` 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. Available built-in regression models are:

```
'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. Built-in correlation models are:

```
'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 or ndarray, optional] Introduce a nugget effect to allow smooth predictions from noisy data. If nugget is an ndarray, it must be the same length as the number of data points used for the fit. The nugget is added to the diagonal of the assumed training covariance; in this way it acts as a Tikhonov regularization in the problem. In the special case of the squared exponential correlation function, the nugget mathematically represents the variance of the input values. 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. Available optimizers are:

```
'fmin_cobyla', 'Welch'
```

'Welch' optimizer is dued to Welch et al., see reference [WBSWM1992]. 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).

random_state: integer or numpy.RandomState, optional The generator used to shuffle the sequence of coordinates of theta in the Welch optimizer. If an integer is given, it fixes the seed. Defaults to the global numpy random number generator.

Attributes

***theta_*: array** Specified theta OR the best set of autocorrelation parameters (the sought maximizer of the reduced likelihood function).

***reduced_likelihood_function_value_*: array** The optimal reduced likelihood function value.

Examples

```
>>> import numpy as np
>>> from sklearn.gaussian_process import GaussianProcess
>>> X = np.array([[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(beta0=None...
        ...
```

Notes

The presentation implementation is based on a translation of the DACE Matlab toolbox, see reference [NLNS2002].

References

Full API documentation: `GaussianProcessScikitsLearnNode`

class `mdp.nodes.`**`MeanEstimatorScikitsLearnNode`**

This node has been automatically generated by wrapping the `sklearn.ensemble.gradient_boosting.MeanEstim` class from the `sklearn` library. The wrapped instance can be accessed through the `scikits_alg` attribute.

Full API documentation: `MeanEstimatorScikitsLearnNode`

class `mdp.nodes.`**`RadiusNeighborsRegressorScikitsLearnNode`**

Regression based on neighbors within a fixed radius.

This node has been automatically generated by wrapping the `sklearn.neighbors.regression.RadiusNeighbors` class from the `sklearn` 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 nearest neighbors in the training set.

Parameters

radius [float, optional (default = 1.0)] Range of parameter space to use by default for :meth‘radius_neighbors‘ queries.

weights [str or callable] weight function used in prediction. Possible values:

- ‘uniform’ : uniform weights. All points in each neighborhood are weighted equally.
- ‘distance’ : weight points by the inverse of their distance. in this case, closer neighbors of a query point will have a greater influence than neighbors which are further away.
- [callable] : a user-defined function which accepts an array of distances, and returns an array of the same shape containing the weights.

Uniform weights are used by default.

algorithm [{‘auto’, ‘ball_tree’, ‘kd_tree’, ‘brute’}, optional] Algorithm used to compute the nearest neighbors:

- ‘ball_tree’ will use `BallTree`
- ‘kd_tree’ will use `scipy.spatial.cKDtree`
- ‘brute’ will use a brute-force search.
- ‘auto’ will attempt to decide the most appropriate algorithm based on the values passed to `fit()` method.

Note: fitting on sparse input will override the setting of this parameter, using brute force.

leaf_size [int, optional (default = 30)] Leaf size passed to BallTree or cKDTree. This can affect the speed of the construction and query, as well as the memory required to store the tree. The optimal value depends on the nature of the problem.

p: integer, optional (default = 2) Parameter for the Minkowski metric from sklearn.metrics.pairwise.pairwise_distances. When p = 1, this is equivalent to using manhattan_distance (l1), and euclidean_distance (l2) for p = 2. For arbitrary p, minkowski_distance (l_p) is used.

Examples

```
>>> X = [[0], [1], [2], [3]]
>>> y = [0, 0, 1, 1]
>>> from sklearn.neighbors import RadiusNeighborsRegressor
>>> neigh = RadiusNeighborsRegressor(radius=1.0)
>>> neigh.fit(X, y)
RadiusNeighborsRegressor(...)
>>> print(neigh.predict([[1.5]]))
[ 0.5]
```

See also

NearestNeighbors KNeighborsRegressor KNeighborsClassifier RadiusNeighborsClassifier

Notes

See *Nearest Neighbors* in the online documentation for a discussion of the choice of `algorithm` and `leaf_size`.

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

Full API documentation: `RadiusNeighborsRegressorScikitsLearnNode`

class `mdp.nodes.`**`PLSSVDScikitsLearnNode`**

Partial Least Square SVD

This node has been automatically generated by wrapping the `sklearn.pls.PLSSVD` class from the `sklearn` library. The wrapped instance can be accessed through the `scikits_alg` attribute.

Simply perform a svd on the crosscovariance matrix: X'Y The are no iterative deflation here.

Parameters

X [array-like of predictors, shape = [n_samples, p]] Training vector, where n_samples in the number of samples and p is the number of predictors. X will be centered before any analysis.

Y [array-like of response, shape = [n_samples, q]] Training vector, where n_samples in the number of samples and q is the number of response variables. X will be centered before any analysis.

n_components [int, (default 2).] number of components to keep.

scale [boolean, (default True)] scale X and Y

Attributes

x_weights_ [array, [p, n_components]] X block weights vectors.

y_weights_ [array, [q, n_components]] Y block weights vectors.

x_scores_ [array, [n_samples, n_components]] X scores.

y_scores_ [array, [n_samples, n_components]] Y scores.

See also

PLSCanonical CCA

Full API documentation: `PLSSVDScikitsLearnNode`

class `mdp.nodes.`**`LassoLarsCVScikitsLearnNode`**

Cross-validated Lasso, using the LARS algorithm

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

The optimization objective for Lasso is:

```
(1 / (2 * n_samples)) * ||y - Xw||^2_2 + alpha * ||w||_1
```

Parameters

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).

verbose [boolean or integer, optional] Sets the verbosity amount

normalize [boolean, optional] If True, the regressors X are normalized

precompute [True | False | 'auto' | array-like] Whether to use a precomputed Gram matrix to speed up calculations. If set to 'auto' let us decide. The Gram matrix can also be passed as argument.

max_iter: integer, optional Maximum number of iterations to perform.

cv [crossvalidation generator, optional] see sklearn.cross_validation module. If None is passed, default to a 5-fold strategy

max_n_alphas [integer, optional] The maximum number of points on the path used to compute the residuals in the cross-validation

n_jobs [integer, optional] Number of CPUs to use during the cross validation. If '-1', use all the CPUs

eps: float, optional The machine-precision regularization in the computation of the Cholesky diagonal factors. Increase this for very ill-conditioned systems.

copy_X [boolean, optional, default True] If True, X will be copied; else, it may be overwritten.

Attributes

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

intercept_ [float] independent term in decision function.

***coef_path_*: array, shape = [n_features, n_alpha]** the varying values of the coefficients along the path

***alpha_*: float** the estimated regularization parameter alpha

***alphas_*: array, shape = [n_alpha]** the different values of alpha along the path

***cv_alphas_*: array, shape = [n_cv_alphas]** all the values of alpha along the path for the different folds

***cv_mse_path_*: array, shape = [n_folds, n_cv_alphas]** the mean square error on left-out for each fold along the path (alpha values given by cv_alphas)

Notes

The object solves the same problem as the LassoCV object. However, unlike the LassoCV, it find the relevent alphas values by itself. In general, because of this property, it will be more stable. However, it is more fragile to heavily multicollinear datasets.

It is more efficient than the LassoCV if only a small number of features are selected compared to the total number, for instance if there are very few samples compared to the number of features.

See also

lars_path, LassoLars, LarsCV, LassoCV

Full API documentation: `LassoLarsCVScikitsLearnNode`

class `mdp.nodes.`**KNeighborsRegressorScikitsLearnNode**

Regression based on k-nearest neighbors.

This node has been automatically generated by wrapping the `sklearn.neighbors.regression.KNeighborsRegre` class from the `sklearn` 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 nearest neighbors in the training set.

Parameters

n_neighbors [int, optional (default = 5)] Number of neighbors to use by default for `k_neighbors()` queries.

weights [str or callable] weight function used in prediction. Possible values:

- 'uniform' : uniform weights. All points in each neighborhood are weighted equally.
- 'distance' : weight points by the inverse of their distance. in this case, closer neighbors of a query point will have a greater influence than neighbors which are further away.
- [callable] : a user-defined function which accepts an array of distances, and returns an array of the same shape containing the weights.

Uniform weights are used by default.

algorithm [{'auto', 'ball_tree', 'kd_tree', 'brute'}, optional] Algorithm used to compute the nearest neighbors:

- 'ball_tree' will use `BallTree`
- 'kd_tree' will use `scipy.spatial.cKDtree`
- 'brute' will use a brute-force search.
- 'auto' will attempt to decide the most appropriate algorithm based on the values passed to `fit()` method.

Note: fitting on sparse input will override the setting of this parameter, using brute force.

leaf_size [int, optional (default = 30)] Leaf size passed to BallTree or cKDTree. This can affect the speed of the construction and query, as well as the memory required to store the tree. The optimal value depends on the nature of the problem.

warn_on_equidistant [boolean, optional. Defaults to True.] Generate a warning if equidistant neighbors are discarded. For classification or regression based on k-neighbors, if neighbor k and neighbor k+1 have identical distances but different labels, then the result will be dependent on the ordering of the training data. If the fit method is `'kd_tree'`, no warnings will be generated.

p: integer, optional (default = 2) Parameter for the Minkowski metric from sklearn.metrics.pairwise.pairwise_distances. When p = 1, this is equivalent to using manhattan_distance (l1), and euclidean_distance (l2) for p = 2. For arbitrary p, minkowski_distance (l_p) is used.

Examples

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

See also

NearestNeighbors RadiusNeighborsRegressor KNeighborsClassifier RadiusNeighborsClassifier

Notes

See *Nearest Neighbors* in the online documentation for a discussion of the choice of `algorithm` and `leaf_size`.

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

Full API documentation: `KNeighborsRegressorScikitsLearnNode`

class `mdp.nodes.`**`RandomForestClassifierScikitsLearnNode`**

A random forest classifier.

This node has been automatically generated by wrapping the `sklearn.ensemble.forest.RandomForestClassifi` class from the `sklearn` library. The wrapped instance can be accessed through the `scikits_alg` attribute.

A random forest is a meta estimator that fits a number of classifical decision trees on various sub-samples of the dataset and use averaging to improve the predictive accuracy and control over-fitting.

Parameters

n_estimators [integer, optional (default=10)] The number of trees in the forest.

criterion [string, optional (default="gini")] The function to measure the quality of a split. Supported criteria are "gini" for the Gini impurity and "entropy" for the information gain. Note: this parameter is tree-specific.

max_depth [integer or None, optional (default=None)] The maximum depth of the tree. If None, then nodes are expanded until all leaves are pure or until all leaves contain less than min_samples_split samples. Note: this parameter is tree-specific.

min_samples_split [integer, optional (default=1)] The minimum number of samples required to split an internal node. Note: this parameter is tree-specific.

min_samples_leaf [integer, optional (default=1)] The minimum number of samples in newly created leaves. A split is discarded if after the split, one of the leaves would contain less then `min_samples_leaf` samples. Note: this parameter is tree-specific.

min_density [float, optional (default=0.1)] This parameter controls a trade-off in an optimization heuristic. It controls the minimum density of the *sample_mask* (i.e. the fraction of samples in the mask). If the density falls below this threshold the mask is recomputed and the input data is packed which results in data copying. If *min_density* equals to one, the partitions are always represented as copies of the original data. Otherwise, partitions are represented as bit masks (aka sample masks). Note: this parameter is tree-specific.

max_features [int, string or None, optional (default="auto")] The number of features to consider when looking for the best split:

- – If "auto", then *max_features=sqrt(n_features)* on
- classification tasks and *max_features=n_features* on regression
- problems.
- – If "sqrt", then *max_features=sqrt(n_features)*.
- – If "log2", then *max_features=log2(n_features)*.
- – If None, then *max_features=n_features*.

Note: this parameter is tree-specific.

bootstrap [boolean, optional (default=True)] Whether bootstrap samples are used when building trees.

compute_importances [boolean, optional (default=True)] Whether feature importances are computed and stored into the `feature_importances_` attribute when calling fit.

oob_score [bool] Whether to use out-of-bag samples to estimate the generalization error.

n_jobs [integer, optional (default=1)] The number of jobs to run in parallel. If -1, then the number of jobs is set to the number of cores.

random_state [int, RandomState instance or None, optional (default=None)] If int, random_state is the seed used by the random number generator; If RandomState instance, random_state is the random number generator; If None, the random number generator is the RandomState instance used by *np.random*.

verbose [int, optional (default=0)] Controlls the verbosity of the tree building process.

Attributes

***estimators_*: list of DecisionTreeClassifier** The collection of fitted sub-estimators.

feature_importances_ [array, shape = [n_features]] The feature importances (the higher, the more important the feature).

oob_score_ [float] Score of the training dataset obtained using an out-of-bag estimate.

oob_decision_function_ [array, shape = [n_samples, n_classes]] Decision function computed with out-of-bag estimate on the training set.

References

See also

DecisionTreeClassifier, ExtraTreesClassifier

Full API documentation: `RandomForestClassifierScikitsLearnNode`

class `mdp.nodes.`**`ForestRegressorScikitsLearnNode`**

Base class for forest of trees-based regressors.

This node has been automatically generated by wrapping the `sklearn.ensemble.forest.ForestRegressor` class from the `sklearn` library. The wrapped instance can be accessed through the `scikits_alg` attribute.

Warning: This class should not be used directly. Use derived classes instead.

Full API documentation: `ForestRegressorScikitsLearnNode`

class mdp.nodes.**LarsScikitsLearnNode**
Least Angle Regression model a.k.a. LAR

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

Parameters

n_nonzero_coefs [int, optional] Target number of non-zero coefficients. Use np.inf for no limit.

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).

verbose [boolean or integer, optional] Sets the verbosity amount

normalize [boolean, optional] If True, the regressors X are normalized

precompute [True | False | 'auto' | array-like] Whether to use a precomputed Gram matrix to speed up calculations. If set to 'auto' let us decide. The Gram matrix can also be passed as argument.

copy_X [boolean, optional, default True] If True, X will be copied; else, it may be overwritten.

eps: float, optional The machine-precision regularization in the computation of the Cholesky diagonal factors. Increase this for very ill-conditioned systems. Unlike the 'tol' parameter in some iterative optimization-based algorithms, this parameter does not control the tolerance of the optimization.

fit_path [boolean] If True the full path is stored in the *coef_path_* attribute. If you compute the solution for a large problem or many targets, setting fit_path to False will lead to a speedup, especially with a small alpha.

Attributes

coef_path_ [array, shape = [n_features, n_alpha]] The varying values of the coefficients along the path. It is not present if the fit_path parameter is False.

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

intercept_ [float] Independent term in decision function.

Examples

```
>>> from sklearn import linear_model
>>> clf = linear_model.Lars(n_nonzero_coefs=1)
>>> clf.fit([[-1, 1], [0, 0], [1, 1]], [-1.1111, 0, -1.1111])
...
Lars(copy_X=True, eps=..., fit_intercept=True, fit_path=True,
   n_nonzero_coefs=1, normalize=True, precompute='auto', verbose=False)
>>> print(clf.coef_)
[ 0. -1.11...]
```

See also

lars_path, LarsCV sklearn.decomposition.sparse_encode

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

Full API documentation: `LarsScikitsLearnNode`

class mdp.nodes.**ElasticNetScikitsLearnNode**
Linear Model trained with L1 and L2 prior as regularizer

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

Minimizes the objective function:

```
1 / (2 * n_samples) * ||y - Xw||^2_2 +
+ alpha * rho * ||w||_1 + 0.5 * alpha * (1 - rho) * ||w||^2_2
```

If you are interested in controlling the L1 and L2 penalty separately, keep in mind that this is equivalent to:

```
a * L1 + b * L2
```

where:

```
alpha = a + b and rho = a / (a + b)
```

The parameter rho corresponds to alpha in the glmnet R package while alpha corresponds to the lambda parameter in glmnet. Specifically, 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 penalty terms. Defaults to 1.0 See the notes for the exact mathematical meaning of this parameter

rho [float] The ElasticNet mixing parameter, with 0 < rho <= 1. For rho = 0 the penalty is an L1 penalty. For rho = 1 it is an L2 penalty. For 0 < rho < 1, the penalty is a combination of L1 and L2

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

normalize [boolean, optional] If True, the regressors X are normalized

precompute [True | False | 'auto' | array-like] Whether to use a precomputed Gram matrix to speed up calculations. If set to 'auto' let us decide. The Gram matrix can also be passed as argument. For sparse input this option is always True to preserve sparsity.

max_iter: int, optional The maximum number of iterations

copy_X [boolean, optional, default False] If True, X will be copied; else, it may be overwritten.

tol: float, optional The tolerance for the optimization: if the updates are smaller than 'tol', the optimization code checks the dual gap for optimality and continues until it is smaller than tol.

warm_start [bool, optional] When set to True, reuse the solution of the previous call to fit as initialization, otherwise, just erase the previous solution.

positive: bool, optional When set to True, forces the coefficients to be positive.

Attributes

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

sparse_coef_ [scipy.sparse matrix, shape = (n_features, 1)] *sparse_coef_* is a readonly property derived from *coef_*

intercept_ [float | array, shape = (n_targets,)] independent term in decision function.

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.`**`IsomapScikitsLearnNode`**

Isomap Embedding

This node has been automatically generated by wrapping the `sklearn.manifold.isomap.Isomap` class from the `sklearn` library. The wrapped instance can be accessed through the `scikits_alg` attribute.

Non-linear dimensionality reduction through Isometric Mapping

Parameters

n_neighbors [integer] number of neighbors to consider for each point.

n_components [integer] number of coordinates for the manifold

eigen_solver [['auto'|'arpack'|'dense']]

'auto' [attempt to choose the most efficient solver] for the given problem.

'arpack' [use Arnoldi decomposition to find the eigenvalues] and eigenvectors. Note that arpack can handle both dense and sparse data efficiently

'dense' [use a direct solver (i.e. LAPACK)] for the eigenvalue decomposition.

tol [float] convergence tolerance passed to arpack or lobpcg. not used if eigen_solver == 'dense'

max_iter [integer] maximum number of iterations for the arpack solver. not used if eigen_solver == 'dense'

path_method [string ['auto'|'FW'|'D']] method to use in finding shortest path. 'auto' : attempt to choose the best algorithm automatically 'FW' : Floyd-Warshall algorithm 'D' : Dijkstra algorithm with Fibonacci Heaps

neighbors_algorithm [string ['auto'|'brute'|'kd_tree'|'ball_tree']] algorithm to use for nearest neighbors search, passed to neighbors.NearestNeighbors instance

Attributes

embedding_ [array-like, shape (n_samples, n_components)] Stores the embedding vectors

*kernel_pca_* : *KernelPCA* object used to implement the embedding

training_data_ [array-like, shape (n_samples, n_features)] Stores the training data

nbrs_ [sklearn.neighbors.NearestNeighbors instance] Stores nearest neighbors instance, including BallTree or KDtree if applicable.

dist_matrix_ [array-like, shape (n_samples, n_samples)] Stores the geodesic distance matrix of training data

References

[1] Tenenbaum, J.B.; De Silva, V.; & Langford, J.C. A global geometric framework for nonlinear dimensionality reduction. Science 290 (5500)

Full API documentation: `IsomapScikitsLearnNode`

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

Binarize data (set feature values to 0 or 1) according to a threshold

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

The default threshold is 0.0 so that any non-zero values are set to 1.0 and zeros are left untouched.

Binarization is a common operation on text count data where the analyst can decide to only consider the presence or absence of a feature rather than a quantified number of occurences for instance.

It can also be used as a pre-processing step for estimators that consider boolean random variables (e.g. modeled using the Bernoulli distribution in a Bayesian setting).

Parameters

threshold [float, optional (0.0 by default)] The lower bound that triggers feature values to be replaced by 1.0.

copy [boolean, optional, default is True] set to False to perform inplace binarization and avoid a copy (if the input is already a numpy array or a scipy.sparse CSR matrix).

Notes

If the input is a sparse matrix, only the non-zero values are subject to update by the Binarizer class.

This estimator is stateless (besides constructor parameters), the fit method does nothing but is useful when used in a pipeline.

Full API documentation: `BinarizerScikitsLearnNode`

class mdp.nodes.**MiniBatchDictionaryLearningScikitsLearnNode**

Mini-batch dictionary learning

This node has been automatically generated by wrapping the `sklearn.decomposition.dict_learning.MiniBatc` class from the `sklearn` library. The wrapped instance can be accessed through the `scikits_alg` attribute.

Finds a dictionary (a set of atoms) that can best be used to represent data using a sparse code.

Solves the optimization problem:

```
(U^*,V^*) = argmin 0.5 || Y - U V ||_2^2 + alpha * || U ||_1
             (U,V)
             with || V_k ||_2 = 1 for all  0 <= k < n_atoms
```

Parameters

n_atoms [int,] number of dictionary elements to extract

alpha [int,] sparsity controlling parameter

n_iter [int,] total number of iterations to perform

fit_algorithm [{'lars', 'cd'}] lars: uses the least angle regression method to solve the lasso problem (linear_model.lars_path) cd: uses the coordinate descent method to compute the Lasso solution (linear_model.Lasso). Lars will be faster if the estimated components are sparse.

transform_algorithm [{'lasso_lars', 'lasso_cd', 'lars', 'omp', 'threshold'}] Algorithm used to transform the data. lars: uses the least angle regression method (linear_model.lars_path) lasso_lars: uses Lars to compute the Lasso solution lasso_cd: uses the coordinate descent method to compute the Lasso solution (linear_model.Lasso). lasso_lars will be faster if the estimated components are sparse. omp: uses orthogonal matching pursuit to estimate the sparse solution threshold: squashes to zero all coefficients less than alpha from the projection dictionary * X'

transform_n_nonzero_coefs [int, `0.1 * n_features` by default] Number of nonzero coefficients to target in each column of the solution. This is only used by *algorithm='lars'* and *algorithm='omp'* and is overridden by *alpha* in the *omp* case.

transform_alpha [float, 1. by default] If *algorithm='lasso_lars'* or *algorithm='lasso_cd'*, *alpha* is the penalty applied to the L1 norm. If *algorithm='threshold'*, *alpha* is the absolute value of the threshold below which coefficients will be squashed to zero. If *algorithm='omp'*, *alpha* is the tolerance parameter: the value of the reconstruction error targeted. In this case, it overrides *n_nonzero_coefs*.

split_sign [bool, False by default] Whether to split the sparse feature vector into the concatenation of its negative part and its positive part. This can improve the performance of downstream classifiers.

n_jobs [int,] number of parallel jobs to run

dict_init [array of shape (n_atoms, n_features),] initial value of the dictionary for warm restart scenarios

verbose :

- degree of verbosity of the printed output

chunk_size [int,] number of samples in each mini-batch

shuffle [bool,] whether to shuffle the samples before forming batches

random_state [int or RandomState] Pseudo number generator state used for random sampling.

Attributes

components_ [array, [n_atoms, n_features]] components extracted from the data

Notes

References:

J. Mairal, F. Bach, J. Ponce, G. Sapiro, 2009: Online dictionary learning for sparse coding (http://www.di.ens.fr/sierra/pdfs/icml09.pdf)

See also

SparseCoder DictionaryLearning SparsePCA MiniBatchSparsePCA

Full API documentation: `MiniBatchDictionaryLearningScikitsLearnNode`

class `mdp.nodes.`**`TfidfVectorizerScikitsLearnNode`**

Convert a collection of raw documents to a matrix of TF-IDF features.

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

Equivalent to CountVectorizer followed by TfidfTransformer.

Parameters

input: string {'filename', 'file', 'content'} If filename, the sequence passed as an argument to fit is expected to be a list of filenames that need reading to fetch the raw content to analyze.

If 'file', the sequence items must have 'read' method (file-like object) it is called to fetch the bytes in memory.

Otherwise the input is expected to be the sequence strings or bytes items are expected to be analyzed directly.

charset: string, 'utf-8' by default. If bytes or files are given to analyze, this charset is used to decode.

charset_error: {'strict', 'ignore', 'replace'} Instruction on what to do if a byte sequence is given to analyze that contains characters not of the given *charset*. By default, it is 'strict', meaning that a UnicodeDecodeError will be raised. Other values are 'ignore' and 'replace'.

strip_accents: {'ascii', 'unicode', None} Remove accents during the preprocessing step. 'ascii' is a fast method that only works on characters that have an direct ASCII mapping. 'unicode' is a slightly slower method that works on any characters. None (default) does nothing.

analyzer: string, {'word', 'char'} or callable Whether the feature should be made of word or character n-grams.

If a callable is passed it is used to extract the sequence of features out of the raw, unprocessed input.

preprocessor: callable or None (default) Override the preprocessing (string transformation) stage while preserving the tokenizing and n-grams generation steps.

tokenizer: callable or None (default) Override the string tokenization step while preserving the preprocessing and n-grams generation steps.

ngram_range: tuple (min_n, max_n) The lower and upper boundary of the range of n-values for different n-grams to be extracted. All values of n such that min_n <= n <= max_n will be used.

stop_words: string {'english'}, list, or None (default) If a string, it is passed to _check_stop_list and the appropriate stop list is returned is currently the only supported string value.

If a list, that list is assumed to contain stop words, all of which will be removed from the resulting tokens.

If None, no stop words will be used. max_df can be set to a value in the range [0.7, 1.0) to automatically detect and filter stop words based on intra corpus document frequency of terms.

lowercase: boolean, default True Convert all characters to lowercase befor tokenizing.

token_pattern: string Regular expression denoting what constitutes a "token", only used if *tokenize == 'word'*. The default regexp select tokens of 2 or more letters characters (punctuation is completely ignored and always treated as a token separator).

max_df [float in range [0.0, 1.0] or int, optional, 1.0 by default] When building the vocabulary ignore terms that have a term frequency strictly higher than the given threshold (corpus specific stop words). If float, the parameter represents a proportion of documents, integer absolute counts. This parameter is ignored if vocabulary is not None.

min_df [float in range [0.0, 1.0] or int, optional, 2 by default] When building the vocabulary ignore terms that have a term frequency strictly lower than the given threshold. This value is also called cut-off in the literature. If float, the parameter represents a proportion of documents, integer absolute counts. This parameter is ignored if vocabulary is not None.

max_features [optional, None by default] If not None, build a vocabulary that only consider the top max_features ordered by term frequency across the corpus.

This parameter is ignored if vocabulary is not None.

vocabulary: Mapping or iterable, optional Either a Mapping (e.g., a dict) where keys are terms and values are indices in the feature matrix, or an iterable over terms. If not given, a vocabulary is determined from the input documents.

binary: boolean, False by default. If True, all non zero counts are set to 1. This is useful for discrete probabilistic models that model binary events rather than integer counts.

dtype: type, optional Type of the matrix returned by fit_transform() or transform().

norm ['l1', 'l2' or None, optional] Norm used to normalize term vectors. None for no normalization.

use_idf [boolean, optional] Enable inverse-document-frequency reweighting.

smooth_idf [boolean, optional] Smooth idf weights by adding one to document frequencies, as if an extra document was seen containing every term in the collection exactly once. Prevents zero divisions.

sublinear_tf [boolean, optional] Apply sublinear tf scaling, i.e. replace tf with 1 + log(tf).

See also

CountVectorizer Tokenize the documents and count the occurrences of token and return them as a sparse matrix

TfidfTransformer Apply Term Frequency Inverse Document Frequency normalization to a sparse matrix of occurrence counts.

Full API documentation: `TfidfVectorizerScikitsLearnNode`

class `mdp.nodes.`**`RandomizedPCAScikitsLearnNode`**

Principal component analysis (PCA) using randomized SVD

This node has been automatically generated by wrapping the `sklearn.decomposition.pca.RandomizedPCA` class from the `sklearn` 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

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 their data respect some hard-wired assumptions.

random_state [int or RandomState instance or None (default)] Pseudo Random Number generator seed control. If None, use the numpy.random singleton.

Attributes

components_ [array, [n_components, n_features]] 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

Examples

```
>>> import numpy as np
>>> from sklearn.decomposition 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, iterated_power=3, n_components=2,
       random_state=<mtrand.RandomState object at 0x...>, whiten=False)
>>> print(pca.explained_variance_ratio_)
[ 0.99244...  0.00755...]
```

See also

PCA ProbabilisticPCA

References

Full API documentation: `RandomizedPCAScikitsLearnNode`

class `mdp.nodes.`**`MiniBatchSparsePCAScikitsLearnNode`**

Mini-batch Sparse Principal Components Analysis

This node has been automatically generated by wrapping the `sklearn.decomposition.sparse_pca.MiniBatchSp` class from the `sklearn` library. The wrapped instance can be accessed through the `scikits_alg` attribute.

Finds the set of sparse components that can optimally reconstruct the data. The amount of sparseness is controllable by the coefficient of the L1 penalty, given by the parameter alpha.

Parameters

n_components [int,] number of sparse atoms to extract

alpha [int,] Sparsity controlling parameter. Higher values lead to sparser components.

ridge_alpha [float,] Amount of ridge shrinkage to apply in order to improve conditioning when calling the transform method.

n_iter [int,] number of iterations to perform for each mini batch

callback [callable,] callable that gets invoked every five iterations

chunk_size [int,] the number of features to take in each mini batch

verbose :

- degree of output the procedure will print

shuffle [boolean,] whether to shuffle the data before splitting it in batches

n_jobs [int,] number of parallel jobs to run, or -1 to autodetect.

method [{'lars', 'cd'}] lars: uses the least angle regression method to solve the lasso problem (linear_model.lars_path) cd: uses the coordinate descent method to compute the Lasso solution (linear_model.Lasso). Lars will be faster if the estimated components are sparse.

random_state [int or RandomState] Pseudo number generator state used for random sampling.

Attributes

components_ [array, [n_components, n_features]] Sparse components extracted from the data.

error_ [array] Vector of errors at each iteration.

See also

PCA SparsePCA DictionaryLearning

Full API documentation: `MiniBatchSparsePCAScikitsLearnNode`

class `mdp.nodes.`**`ProjectedGradientNMFScikitsLearnNode`**

Non-Negative matrix factorization by Projected Gradient (NMF)

This node has been automatically generated by wrapping the `sklearn.decomposition.nmf.ProjectedGradientN` class from the `sklearn` library. The wrapped instance can be accessed through the `scikits_alg` attribute.

Parameters

X: {array-like, sparse matrix}, shape = [n_samples, n_features] Data the model will be fit to.

n_components: int or None Number of components, if n_components is not set all components are kept

init: 'nndsvd' | 'nndsvda' | 'nndsvdar' | int | RandomState Method used to initialize the procedure. Default: 'nndsvdar' Valid options:

```
'nndsvd': Nonnegative Double Singular Value Decomposition (NNDSVD)
    initialization (better for sparseness)
'nndsvda': NNDSVD with zeros filled with the average of X
    (better when sparsity is not desired)
'nndsvdar': NNDSVD with zeros filled with small random values
    (generally faster, less accurate alternative to NNDSVDa
    for when sparsity is not desired)
int seed or RandomState: non-negative random matrices
```

sparseness: 'data' | 'components' | None, default: None Where to enforce sparsity in the model.

beta: double, default: 1 Degree of sparseness, if sparseness is not None. Larger values mean more sparseness.

eta: double, default: 0.1 Degree of correctness to mantain, if sparsity is not None. Smaller values mean larger error.

tol: double, default: 1e-4 Tolerance value used in stopping conditions.

max_iter: int, default: 200 Number of iterations to compute.

nls_max_iter: int, default: 2000 Number of iterations in NLS subproblem.

Attributes

components_ [array, [n_components, n_features]] Non-negative components of the data

reconstruction_err_ [number] Frobenius norm of the matrix difference between the training data and the reconstructed data from the fit produced by the model. `|| X - WH ||_2` Not computed for sparse input matrices because it is too expensive in terms of memory.

Examples

```
>>> import numpy as np
>>> X = np.array([[1,1], [2, 1], [3, 1.2], [4, 1], [5, 0.8], [6, 1]])
>>> from sklearn.decomposition import ProjectedGradientNMF
>>> model = ProjectedGradientNMF(n_components=2, init=0)
>>> model.fit(X)
ProjectedGradientNMF(beta=1, eta=0.1, init=0, max_iter=200, n_components=2,
                     nls_max_iter=2000, sparseness=None, tol=0.0001)
>>> model.components_
array([[ 0.77032744,  0.11118662],
```

```
       [ 0.38526873,  0.38228063]])
>>> model.reconstruction_err_
0.00746...
>>> model = ProjectedGradientNMF(n_components=2, init=0,
...                              sparseness='components')
>>> model.fit(X)
ProjectedGradientNMF(beta=1, eta=0.1, init=0, max_iter=200, n_components=2,
           nls_max_iter=2000, sparseness='components', tol=0.0001)
>>> model.components_
array([[ 1.67481991,  0.29614922],
       [-0.        ,  0.4681982 ]])
>>> model.reconstruction_err_
0.513...
```

Notes

This implements

C.-J. Lin. Projected gradient methods for non-negative matrix factorization. Neural Computation, 19(2007), 2756-2779. http://www.csie.ntu.edu.tw/~cjlin/nmf/

P. Hoyer. Non-negative Matrix Factorization with Sparseness Constraints. Journal of Machine Learning Research 2004.

NNDSVD is introduced in

C. Boutsidis, E. Gallopoulos: SVD based initialization: A head start for nonnegative matrix factorization - Pattern Recognition, 2008 http://www.cs.rpi.edu/~boutsc/files/nndsvd.pdf

Full API documentation: `ProjectedGradientNMFScikitsLearnNode`

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

Elastic Net model with iterative fitting along a regularization path

This node has been automatically generated by wrapping the `sklearn.linear_model.coordinate_descent.Elas` class from the `sklearn` 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). For rho = 0 the penalty is an L1 penalty. For rho = 1 it is an L2 penalty. For 0 < rho < 1, the penalty is a combination of L1 and L2 This parameter can be a list, in which case the different values are tested by cross-validation and the one giving the best prediction score is used. Note that a good choice of list of values for rho is often to put more values close to 1 (i.e. Lasso) and less close to 0 (i.e. Ridge), as in [.1, .5, .7, .9, .95, .99, 1]

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

precompute [True | False | 'auto' | array-like] Whether to use a precomputed Gram matrix to speed up calculations. If set to 'auto' let us decide. The Gram matrix can also be passed as argument.

max_iter [int, optional] The maximum number of iterations

tol [float, optional] The tolerance for the optimization: if the updates are smaller than 'tol', the optimization code checks the dual gap for optimality and continues until it is smaller than tol.

cv [integer or crossvalidation generator, optional] If an integer is passed, it is the number of fold (default 3). Specific crossvalidation objects can be passed, see sklearn.cross_validation module for the list of possible objects

verbose [bool or integer] amount of verbosity

n_jobs [integer, optional] Number of CPUs to use during the cross validation. If '-1', use all the CPUs. Note that this is used only if multiple values for rho are given.

Attributes

alpha_ [float] The amount of penalization choosen by cross validation

rho_ [float] The compromise between l1 and l2 penalization choosen by cross validation

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

intercept_ [float] independent term in decision function.

mse_path_ [array, shape = (n_rho, n_alpha, n_folds)] mean square error for the test set on each fold, varying rho and alpha

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.

The parameter rho corresponds to alpha in the glmnet R package while alpha corresponds to the lambda parameter in glmnet. More specifically, the optimization objective is:

```
1 / (2 * n_samples) * ||y - Xw||^2_2 +
+ alpha * rho * ||w||_1 + 0.5 * alpha * (1 - rho) * ||w||^2_2
```

If you are interested in controlling the L1 and L2 penalty separately, keep in mind that this is equivalent to:

```
a * L1 + b * L2
```

for:

```
alpha = a + b and rho = a / (a + b)
```

See also

enet_path ElasticNet

Full API documentation: `ElasticNetCVScikitsLearnNode`

class `mdp.nodes.`**`LassoLarsICScikitsLearnNode`**

Lasso model fit with Lars using BIC or AIC for model selection

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

The optimization objective for Lasso is:

```
(1 / (2 * n_samples)) * ||y - Xw||^2_2 + alpha * ||w||_1
```

AIC is the Akaike information criterion and BIC is the Bayes Information criterion. Such criteria are useful to select the value of the regularization parameter by making a trade-off between the goodness of fit and the complexity of the model. A good model should explain well the data while being simple.

Parameters

criterion: 'bic' | 'aic' The type of criterion to use.

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).

verbose [boolean or integer, optional] Sets the verbosity amount

normalize [boolean, optional] If True, the regressors X are normalized

copy_X [boolean, optional, default True] If True, X will be copied; else, it may be overwritten.

precompute [True | False | 'auto' | array-like] Whether to use a precomputed Gram matrix to speed up calculations. If set to 'auto' let us decide. The Gram matrix can also be passed as argument.

max_iter: integer, optional Maximum number of iterations to perform. Can be used for early stopping.

eps: float, optional The machine-precision regularization in the computation of the Cholesky diagonal factors. Increase this for very ill-conditioned systems. Unlike the 'tol' parameter in some iterative optimization-based algorithms, this parameter does not control the tolerance of the optimization.

Attributes

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

intercept_ [float] independent term in decision function.

alpha_ [float] the alpha parameter chosen by the information criterion

Examples

```
>>> from sklearn import linear_model
>>> clf = linear_model.LassoLarsIC(criterion='bic')
>>> clf.fit([[-1, 1], [0, 0], [1, 1]], [-1.1111, 0, -1.1111])
...
LassoLarsIC(copy_X=True, criterion='bic', eps=..., fit_intercept=True,
      max_iter=500, normalize=True, precompute='auto',
      verbose=False)
>>> print(clf.coef_)
[ 0.  -1.11...]
```

Notes

The estimation of the number of degrees of freedom is given by:

"On the degrees of freedom of the lasso" Hui Zou, Trevor Hastie, and Robert Tibshirani Ann. Statist. Volume 35, Number 5 (2007), 2173-2192.

http://en.wikipedia.org/wiki/Akaike_information_criterion http://en.wikipedia.org/wiki/Bayesian_information_criterion

See also

lars_path, LassoLars, LassoLarsCV

Full API documentation: `LassoLarsICScikitsLearnNode`

class `mdp.nodes.`**`RFEScikitsLearnNode`**

Feature ranking with recursive feature elimination.

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

Given an external estimator that assigns weights to features (e.g., the coefficients of a linear model), the goal of recursive feature elimination (RFE) is to select features by recursively considering smaller and smaller sets of features. First, the estimator is trained on the initial set of features and weights are assigned to each one of them. Then, features whose absolute weights are the smallest are pruned from the current set features. That procedure is recursively repeated on the pruned set until the desired number of features to select is eventually reached.

Parameters

estimator [object] A supervised learning estimator with a *fit* method that updates a *coef_* attribute that holds the fitted parameters. Important features must correspond to 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* modules.

n_features_to_select [int or None (default=None)] The number of features to select. If *None*, half of the features are selected.

step [int or float, optional (default=1)] If greater than or equal to 1, then *step* corresponds to the (integer) number of features to remove at each iteration. If within (0.0, 1.0), then *step* corresponds to the percentage (rounded down) of features to remove at each iteration.

Attributes

n_features_ [int] The number of selected features.

support_ [array of shape [n_features]] The mask of selected features.

ranking_ [array of shape [n_features]] The feature ranking, such that *ranking_[i]* corresponds to the ranking position of the i-th feature. Selected (i.e., estimated best) features are assigned rank 1.

Examples

The following example shows how to retrieve the 5 right informative features in the Friedman #1 dataset.

```
>>> from sklearn.datasets import make_friedman1
>>> from sklearn.feature_selection import RFE
>>> from sklearn.svm import SVR
>>> X, y = make_friedman1(n_samples=50, n_features=10, random_state=0)
>>> estimator = SVR(kernel="linear")
>>> selector = RFE(estimator, 5, step=1)
>>> selector = selector.fit(X, y)
>>> selector.support_
array([ True,  True,  True,  True,  True,
        False, False, False, False, False], dtype=bool)
>>> selector.ranking_
array([1, 1, 1, 1, 1, 6, 4, 3, 2, 5])
```

References

Full API documentation: `RFEScikitsLearnNode`

class `mdp.nodes.`**`PCAScikitsLearnNode`**

Principal component analysis (PCA)

This node has been automatically generated by wrapping the `sklearn.decomposition.pca.PCA` class from the `sklearn` 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

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 if `0 < n_components < 1`, select the number of components such that the amount of variance that needs to be explained is greater than the percentage specified by n_components

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_components, n_features]] 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'

Due to implementation subtleties of the Singular Value Decomposition (SVD), which is used in this implementation, running fit twice on the same matrix can lead to principal components with signs flipped (change in direction). For this reason, it is important to always use the same estimator object to transform data in a consistent fashion.

Examples

```
>>> import numpy as np
>>> from sklearn.decomposition 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_)
[ 0.99244...  0.00755...]
```

See also

ProbabilisticPCA RandomizedPCA KernelPCA SparsePCA

Full API documentation: `PCAScikitsLearnNode`

class `mdp.nodes.`**`MultiTaskLassoScikitsLearnNode`**

Multi-task Lasso model trained with L1/L2 mixed-norm as regularizer

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

The optimization objective for Lasso is:

```
(1 / (2 * n_samples)) * ||Y - XW||^2_Fro + alpha * ||W||_21
```

Where:

```
||W||_21 = \sum_i \sqrt{\sum_j w_{ij}^2}
```

i.e. the sum of norm of earch row.

Parameters

alpha [float, optional] Constant that multiplies the L1/L2 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).

normalize [boolean, optional] If True, the regressors X are normalized

copy_X [boolean, optional, default True] If True, X will be copied; else, it may be overwritten.

max_iter [int, optional] The maximum number of iterations

tol [float, optional] The tolerance for the optimization: if the updates are smaller than 'tol', the optimization code checks the dual gap for optimality and continues until it is smaller than tol.

warm_start [bool, optional] When set to True, reuse the solution of the previous call to fit as initialization, otherwise, just erase the previous solution.

Attributes

coef_ [array, shape = (n_tasks, n_features)] parameter vector (W in the cost function formula)

intercept_ [array, shape = (n_tasks,)] independent term in decision function.

Examples

```
>>> from sklearn import linear_model
>>> clf = linear_model.MultiTaskLasso(alpha=0.1)
>>> clf.fit([[0,0], [1, 1], [2, 2]], [[0, 0], [1, 1], [2, 2]])
MultiTaskLasso(alpha=0.1, copy_X=True, fit_intercept=True, max_iter=1000,
        normalize=False, tol=0.0001, warm_start=False)
>>> print clf.coef_
[[ 0.89393398  0.        ]
 [ 0.89393398  0.        ]]
>>> print clf.intercept_
[ 0.10606602  0.10606602]
```

See also

Lasso, MultiTaskElasticNet

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: `MultiTaskLassoScikitsLearnNode`

class `mdp.nodes.`**`RandomizedLogisticRegressionScikitsLearnNode`**

Randomized Logistic Regression

This node has been automatically generated by wrapping the `sklearn.linear_model.randomized_l1.Randomize` class from the `sklearn` library. The wrapped instance can be accessed through the `scikits_alg` attribute.

Randomized Regression works by resampling the train data and computing a LogisticRegression on each resampling. In short, the features selected more often are good features. It is also known as stability selection.

Parameters

C [float] The regularization parameter C in the LogisticRegression.

scaling [float] The alpha parameter in the stability selection article used to randomly scale the features. Should be between 0 and 1.

sample_fraction [float] The fraction of samples to be used in each randomized design. Should be between 0 and 1. If 1, all samples are used.

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).

verbose [boolean or integer, optional] Sets the verbosity amount

normalize [boolean, optional] If True, the regressors X are normalized

tol [float, optional] tolerance for stopping criteria of LogisticRegression

n_jobs [integer, optional] Number of CPUs to use during the resampling. If '-1', use all the CPUs

random_state [int, RandomState instance or None, optional (default=None)] If int, random_state is the seed used by the random number generator; If RandomState instance, random_state is the random number generator; If None, the random number generator is the RandomState instance used by *np.random*.

pre_dispatch [int, or string, optional] Controls the number of jobs that get dispatched during parallel execution. Reducing this number can be useful to avoid an explosion of memory consumption when more jobs get dispatched than CPUs can process. This parameter can be:

- None, in which case all the jobs are immediatly created and spawned. Use this for lightweight and fast-running jobs, to avoid delays due to on-demand spawning of the jobs
- An int, giving the exact number of total jobs that are spawned
- A string, giving an expression as a function of n_jobs, as in '2*n_jobs'

memory [Instance of joblib.Memory or string] Used for internal caching. By default, no caching is done. If a string is given, it is thepath to the caching directory.

Attributes

scores_ [array, shape = [n_features]] Feature scores between 0 and 1.

all_scores_ [array, shape = [n_features, n_reg_parameter]] Feature scores between 0 and 1 for all values of the regularization parameter. The reference article suggests `scores_` is the max of `all_scores_`.

Examples

```
>>> from sklearn.linear_model import RandomizedLogisticRegression
>>> randomized_logistic = RandomizedLogisticRegression()
```

Notes

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

References

Stability selection Nicolai Meinshausen, Peter Buhlmann Journal of the Royal Statistical Society: Series B Volume 72, Issue 4, pages 417-473, September 2010 DOI: 10.1111/j.1467-9868.2010.00740.x

See also

RandomizedLasso, Lasso, ElasticNet

Full API documentation: `RandomizedLogisticRegressionScikitsLearnNode`

class `mdp.nodes.`**`SelectFweScikitsLearnNode`**
Filter: Select the p-values corresponding to Family-wise error rate

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

Parameters

score_func: callable Function taking two arrays X and y, and returning 2 arrays:

- both scores and pvalues

alpha: float, optional The highest uncorrected p-value for features to keep

Full API documentation: `SelectFweScikitsLearnNode`

class `mdp.nodes.`**`MultiTaskElasticNetScikitsLearnNode`**
Multi-task ElasticNet model trained with L1/L2 mixed-norm as regularizer

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

The optimization objective for Lasso is:

```
(1 / (2 * n_samples)) * ||Y - XW||^Fro_2
+ alpha * rho * ||W||_21 + 0.5 * alpha * (1 - rho) * ||W||_Fro^2
```

Where:

```
||W||_21 = \sum_i \sqrt{\sum_j w_{ij}^2}
```

i.e. the sum of norm of earch row.

Parameters

alpha [float, optional] Constant that multiplies the L1/L2 term. Defaults to 1.0

rho [float] The ElasticNet mixing parameter, with 0 < rho <= 1. For rho = 0 the penalty is an L1/L2 penalty. For rho = 1 it is an L2 penalty. For 0 < rho < 1, the penalty is a combination of L1/L2 and L2

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).

normalize [boolean, optional] If True, the regressors X are normalized

copy_X [boolean, optional, default True] If True, X will be copied; else, it may be overwritten.

max_iter [int, optional] The maximum number of iterations

tol [float, optional] The tolerance for the optimization: if the updates are smaller than 'tol', the optimization code checks the dual gap for optimality and continues until it is smaller than tol.

warm_start [bool, optional] When set to True, reuse the solution of the previous call to fit as initialization, otherwise, just erase the previous solution.

Attributes

intercept_ [array, shape = (n_tasks,)] Independent term in decision function.

coef_ [array, shape = (n_tasks, n_features)] Parameter vector (W in the cost function formula). If a 1D y is passed in at fit (non multi-task usage), *coef_* is then a 1D array

Examples

```
>>> from sklearn import linear_model
>>> clf = linear_model.MultiTaskElasticNet(alpha=0.1)
>>> clf.fit([[0,0], [1, 1], [2, 2]], [[0, 0], [1, 1], [2, 2]])
...
MultiTaskElasticNet(alpha=0.1, copy_X=True, fit_intercept=True,
        max_iter=1000, normalize=False, rho=0.5, tol=0.0001,
        warm_start=False)
>>> print clf.coef_
[[ 0.45663524  0.45612256]
 [ 0.45663524  0.45612256]]
>>> print clf.intercept_
[ 0.0872422  0.0872422]
```

See also

ElasticNet, MultiTaskLasso

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: `MultiTaskElasticNetScikitsLearnNode`

class `mdp.nodes.`**`SparseCoderScikitsLearnNode`**

Sparse coding

This node has been automatically generated by wrapping the `sklearn.decomposition.dict_learning.SparseC` class from the `sklearn` library. The wrapped instance can be accessed through the `scikits_alg` attribute.

Finds a sparse representation of data against a fixed, precomputed dictionary.

Each row of the result is the solution to a sparse coding problem. The goal is to find a sparse array *code* such that:

```
X ~= code * dictionary
```

Parameters

dictionary [array, [n_atoms, n_features]] The dictionary atoms used for sparse coding. Lines are assumed to be normalized to unit norm.

transform_algorithm [{'lasso_lars', 'lasso_cd', 'lars', 'omp', 'threshold'}] Algorithm used to transform the data:

- lars: uses the least angle regression method (linear_model.lars_path)
- lasso_lars: uses Lars to compute the Lasso solution
- lasso_cd: uses the coordinate descent method to compute the
- Lasso solution (linear_model.Lasso). lasso_lars will be faster if
- the estimated components are sparse.
- omp: uses orthogonal matching pursuit to estimate the sparse solution
- threshold: squashes to zero all coefficients less than alpha from
- the projection `dictionary * X'`

transform_n_nonzero_coefs [int, `0.1 * n_features` by default] Number of nonzero coefficients to target in each column of the solution. This is only used by *algorithm='lars'* and *algorithm='omp'* and is overridden by *alpha* in the *omp* case.

transform_alpha [float, 1. by default] If *algorithm='lasso_lars'* or *algorithm='lasso_cd'*, *alpha* is the penalty applied to the L1 norm. If *algorithm='threshold'*, *alpha* is the absolute value of the threshold below which coefficients will be squashed to zero. If *algorithm='omp'*, *alpha* is the tolerance parameter: the value of the reconstruction error targeted. In this case, it overrides *n_nonzero_coefs*.

split_sign [bool, False by default] Whether to split the sparse feature vector into the concatenation of its negative part and its positive part. This can improve the performance of downstream classifiers.

n_jobs [int,] number of parallel jobs to run

Attributes

components_ [array, [n_atoms, n_features]] The unchanged dictionary atoms

See also

DictionaryLearning MiniBatchDictionaryLearning SparsePCA MiniBatchSparsePCA sparse_encode

Full API documentation: `SparseCoderScikitsLearnNode`

class `mdp.nodes.`**`GMMScikitsLearnNode`**

Gaussian Mixture Model

This node has been automatically generated by wrapping the `sklearn.mixture.gmm.GMM` class from the `sklearn` 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_components [int, optional] Number of mixture components. Defaults to 1.

covariance_type [string, optional] String describing the type of covariance parameters to use. Must be one of 'spherical', 'tied', 'diag', 'full'. Defaults to 'diag'.

random_state: RandomState or an int seed (0 by default) A random number generator instance

min_covar [float, optional] Floor on the diagonal of the covariance matrix to prevent overfitting. Defaults to 1e-3.

thresh [float, optional] Convergence threshold.

n_iter [int, optional] Number of EM iterations to perform.

n_init [int, optional] Number of initializations to perform. the best results is kept

params [string, optional] Controls which parameters are updated in the training process. Can contain any combination of 'w' for weights, 'm' for means, and 'c' for covars. Defaults to 'wmc'.

init_params [string, optional] Controls which parameters are updated in the initialization process. Can contain any combination of 'w' for weights, 'm' for means, and 'c' for covars. Defaults to 'wmc'.

Attributes

weights_ [array, shape (*n_components*,)] This attribute stores the mixing weights for each mixture component.

means_ [array, shape (*n_components*, *n_features*)] Mean parameters for each mixture component.

covars_ [array] Covariance parameters for each mixture component. The shape depends on *covariance_type*:

```
(n_components,)                         if 'spherical',
(n_features, n_features)                if 'tied',
(n_components, n_features)              if 'diag',
(n_components, n_features, n_features) if 'full'
```

converged_ [bool] True when convergence was reached in fit(), False otherwise.

See Also

DPGMM [Ininite gaussian mixture model, using the dirichlet] process, fit with a variational algorithm

VBGMM [Finite gaussian mixture model fit with a variational] algorithm, better for situations where there might be too little data to get a good estimate of the covariance matrix.

Examples

```
>>> import numpy as np
>>> from sklearn import mixture
>>> np.random.seed(1)
>>> g = mixture.GMM(n_components=2)
>>> # Generate random observations with two modes centered on 0
>>> # and 10 to use for training.
>>> obs = np.concatenate((np.random.randn(100, 1),
...                       10 + np.random.randn(300, 1)))
>>> g.fit(obs)
GMM(covariance_type='diag', init_params='wmc', min_covar=0.001,
        n_components=2, n_init=1, n_iter=100, params='wmc',
        random_state=None, thresh=0.01)
>>> np.round(g.weights_, 2)
array([ 0.75,  0.25])
>>> np.round(g.means_, 2)
array([[ 10.05],
       [  0.06]])
>>> np.round(g.covars_, 2)
array([[[ 1.02]],
       [[ 0.96]]])
>>> g.predict([[0], [2], [9], [10]])
```

```
array([1, 1, 0, 0]...)
>>> np.round(g.score([[0], [2], [9], [10]]), 2)
array([-2.19, -4.58, -1.75, -1.21])
>>> # 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(covariance_type='diag', init_params='wmc', min_covar=0.001,
        n_components=2, n_init=1, n_iter=100, params='wmc',
        random_state=None, thresh=0.01)
>>> np.round(g.weights_, 2)
array([ 0.5,  0.5])
```

Full API documentation: `GMMScikitsLearnNode`

class `mdp.nodes.`**`DecisionTreeClassifierScikitsLearnNode`**

A decision tree classifier.

This node has been automatically generated by wrapping the `sklearn.tree.tree.DecisionTreeClassifier` class from the `sklearn` library. The wrapped instance can be accessed through the `scikits_alg` attribute.

Parameters

criterion [string, optional (default="gini")] The function to measure the quality of a split. Supported criteria are "gini" for the Gini impurity and "entropy" for the information gain.

max_depth [integer or None, optional (default=None)] The maximum depth of the tree. If None, then nodes are expanded until all leaves are pure or until all leaves contain less than min_samples_split samples.

min_samples_split [integer, optional (default=1)] The minimum number of samples required to split an internal node.

min_samples_leaf [integer, optional (default=1)] The minimum number of samples required to be at a leaf node.

min_density [float, optional (default=0.1)] This parameter controls a trade-off in an optimization heuristic. It controls the minimum density of the *sample_mask* (i.e. the fraction of samples in the mask). If the density falls below this threshold the mask is recomputed and the input data is packed which results in data copying. If *min_density* equals to one, the partitions are always represented as copies of the original data. Otherwise, partitions are represented as bit masks (aka sample masks).

max_features [int, string or None, optional (default=None)] The number of features to consider when looking for the best split. If "auto", then *max_features=sqrt(n_features)* on classification tasks and *max_features=n_features* on regression problems. If "sqrt", then *max_features=sqrt(n_features)*. If "log2", then *max_features=log2(n_features)*. If None, then *max_features=n_features*.

compute_importances [boolean, optional (default=True)] Whether feature importances are computed and stored into the `feature_importances_` attribute when calling fit.

random_state [int, RandomState instance or None, optional (default=None)] If int, random_state is the seed used by the random number generator; If RandomState instance, random_state is the random number generator; If None, the random number generator is the RandomState instance used by *np.random*.

Attributes

tree_ [Tree object] The underlying Tree object.

feature_importances_ [array of shape = [n_features]] The feature importances (the higher, the more important the feature). The importance I(f) of a feature f is computed as the (normalized) total reduction of error brought by that feature. It is also known as the Gini importance [4]_.

See also

DecisionTreeRegressor

References

Examples

```
>>> from sklearn.datasets import load_iris
>>> from sklearn.cross_validation import cross_val_score
>>> from sklearn.tree import DecisionTreeClassifier

>>> clf = DecisionTreeClassifier(random_state=0)
>>> iris = load_iris()

>>> cross_val_score(clf, iris.data, iris.target, cv=10)
...
...
array([ 1.     ,  0.93...,  0.86...,  0.93...,  0.93...,
        0.93...,  0.93...,  1.     ,  0.93...,  1.      ])
```

Full API documentation: `DecisionTreeClassifierScikitsLearnNode`

class `mdp.nodes.`**`PipelineScikitsLearnNode`**

Pipeline of transforms with a final estimator.

This node has been automatically generated by wrapping the `sklearn.pipeline.Pipeline` class from the `sklearn` library. The wrapped instance can be accessed through the `scikits_alg` attribute.

Sequentially apply a list of transforms and a final estimator. Intermediate steps of the pipeline must be 'transforms', that is, they must implements fit and transform methods. The final estimator needs only implements fit.

The purpose of the pipeline is to assemble several steps that can be cross-validated together while setting different parameters. For this, it enables setting parameters of the various steps using their names and the parameter name separated by a '__', as in the example below.

Parameters

steps: list List of (name, transform) tuples (implementing fit/transform) that are chained, in the order in which they are chained, with the last object an estimator.

Attributes

steps [list of (name, object)] List of the named object that compose the pipeline, in the order that they are applied on the data.

Examples

```
>>> from sklearn import svm
>>> from sklearn.datasets import samples_generator
>>> from sklearn.feature_selection import SelectKBest
>>> from sklearn.feature_selection import f_regression
>>> from sklearn.pipeline import Pipeline

>>> # generate some data to play with
>>> X, y = samples_generator.make_classification(
...     n_informative=5, n_redundant=0, random_state=42)

>>> # ANOVA SVM-C
>>> anova_filter = SelectKBest(f_regression, k=5)
>>> clf = svm.SVC(kernel='linear')
>>> anova_svm = Pipeline([('anova', anova_filter), ('svc', clf)])

>>> # You can set the parameters using the names issued
>>> # For instance, fit using a k of 10 in the SelectKBest
>>> # and a parameter 'C' of the svn
>>> anova_svm.set_params(anova__k=10, svc__C=.1).fit(X, y)
...
Pipeline(steps=[...])
```

```
>>> prediction = anova_svm.predict(X)
>>> anova_svm.score(X, y)
0.75
```

Full API documentation: `PipelineScikitsLearnNode`

class `mdp.nodes.`**`GenericUnivariateSelectScikitsLearnNode`**

Univariate feature selector with configurable strategy

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

Parameters

score_func: callable Function taking two arrays X and y, and returning 2 arrays:

- both scores and pvalues

mode: {'percentile', 'k_best', 'fpr', 'fdr', 'fwe'} Feature selection mode

param: float or int depending on the feature selection mode Parameter of the corresponding mode

Full API documentation: `GenericUnivariateSelectScikitsLearnNode`

class `mdp.nodes.`**`BernoulliNBScikitsLearnNode`**

Naive Bayes classifier for multivariate Bernoulli models.

This node has been automatically generated by wrapping the `sklearn.naive_bayes.BernoulliNB` class from the `sklearn` library. The wrapped instance can be accessed through the `scikits_alg` attribute.

Like MultinomialNB, this classifier is suitable for discrete data. The difference is that while MultinomialNB works with occurrence counts, BernoulliNB is designed for binary/boolean features.

Parameters

alpha: float, optional (default=1.0) Additive (Laplace/Lidstone) smoothing parameter (0 for no smoothing).

binarize: float or None, optional Threshold for binarizing (mapping to booleans) of sample features. If None, input is presumed to already consist of binary vectors.

fit_prior: boolean Whether to learn class prior probabilities or not. If false, a uniform prior will be used.

Attributes

class_log_prior_ [array, shape = [n_classes]] Log probability of each class (smoothed).

feature_log_prob_ [array, shape = [n_classes, n_features]] Empirical log probability of features given a class, P(x_ily).

Examples

```
>>> import numpy as np
>>> X = np.random.randint(2, size=(6, 100))
>>> Y = np.array([1, 2, 3, 4, 4, 5])
>>> from sklearn.naive_bayes import BernoulliNB
>>> clf = BernoulliNB()
>>> clf.fit(X, Y)
BernoulliNB(alpha=1.0, binarize=0.0, fit_prior=True)
>>> print(clf.predict(X[2]))
[3]
```

References

C.D. Manning, P. Raghavan and H. Schütze (2008). Introduction to Information Retrieval. Cambridge University Press, pp. 234–265.

A. McCallum and K. Nigam (1998). A comparison of event models for naive Bayes text classification. Proc. AAAI/ICML-98 Workshop on Learning for Text Categorization, pp. 41–48.

V. Metsis, I. Androutsopoulos and G. Paliouras (2006). Spam filtering with naive Bayes – Which naive Bayes? 3rd Conf. on Email and Anti-Spam (CEAS).

Full API documentation: `BernoulliNBScikitsLearnNode`

class `mdp.nodes.`**`LogisticRegressionScikitsLearnNode`**

Logistic Regression (aka logit, MaxEnt) classifier.

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

In the multiclass case, the training algorithm uses a one-vs.-all (OvA) scheme, rather than the "true" multinomial LR.

This class implements L1 and L2 regularized logistic regression using the *liblinear* library. It can handle both dense and sparse input. Use C-ordered arrays or CSR matrices containing 64-bit floats for optimal performance; any other input format will be converted (and copied).

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. Prefer dual=False when n_samples > n_features.

C [float, optional (default=1.0)] Specifies the strength of the regularization. The smaller it is the bigger is the regularization.

fit_intercept [bool, default: True] Specifies if a constant (a.k.a. bias or intercept) should be added the decision function

intercept_scaling [float, default: 1] when self.fit_intercept is True, instance vector x becomes [x, self.intercept_scaling], i.e. a "synthetic" feature with constant value equals to intercept_scaling is appended to the instance vector. The intercept becomes intercept_scaling * synthetic feature weight Note! the synthetic feature weight is subject to l1/l2 regularization as all other features. To lessen the effect of regularization on synthetic feature weight (and therefore on the intercept) intercept_scaling has to be increased

class_weight [{dict, 'auto'}, optional] Set the parameter C of class i to class_weight[i]*C for SVC. If not given, all classes are supposed to have weight one. The 'auto' mode uses the values of y to automatically adjust weights inversely proportional to class frequencies.

tol: float, optional tolerance for stopping criteria

Attributes

coef_ [array, shape = [n_classes-1, n_features]] Coefficient of the features in the decision function.

*coef_* is readonly property derived from *raw_coef_* that follows the internal memory layout of liblinear.

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 tol parameter.

References:

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

Hsiang-Fu Yu, Fang-Lan Huang, Chih-Jen Lin (2011). Dual coordinate descent methods for logistic regression and maximum entropy models. Machine Learning 85(1-2):41-75. http://www.csie.ntu.edu.tw/~cjlin/papers/maxent_dual.pdf

Full API documentation: `LogisticRegressionScikitsLearnNode`

class `mdp.nodes.`**`NuSVCScikitsLearnNode`**

NuSVC for sparse matrices (csr).

This node has been automatically generated by wrapping the `sklearn.svm.sparse.classes.NuSVC` class from the `sklearn` library. The wrapped instance can be accessed through the `scikits_alg` attribute.

See `sklearn.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).

Examples

```
>>> import numpy as np
>>> X = np.array([[-1, -1], [-2, -1], [1, 1], [2, 1]])
>>> y = np.array([1, 1, 2, 2])
>>> from sklearn.svm.sparse import NuSVC
>>> clf = NuSVC()
>>> clf.fit(X, y)
NuSVC(cache_size=200, class_weight=None, coef0=0.0, degree=3, gamma=0.0,
        kernel='rbf', nu=0.5, probability=False, shrinking=True, tol=0.001,
        verbose=False)
>>> print(clf.predict([[-0.8, -1]]))
[ 1.]
```

Full API documentation: `NuSVCScikitsLearnNode`

class `mdp.nodes.`**`SparsePCAScikitsLearnNode`**

Sparse Principal Components Analysis (SparsePCA)

This node has been automatically generated by wrapping the `sklearn.decomposition.sparse_pca.SparsePCA` class from the `sklearn` library. The wrapped instance can be accessed through the `scikits_alg` attribute.

Finds the set of sparse components that can optimally reconstruct the data. The amount of sparseness is controllable by the coefficient of the L1 penalty, given by the parameter alpha.

Parameters

n_components [int,] Number of sparse atoms to extract.

alpha [float,] Sparsity controlling parameter. Higher values lead to sparser components.

ridge_alpha [float,] Amount of ridge shrinkage to apply in order to improve conditioning when calling the transform method.

max_iter [int,] Maximum number of iterations to perform.

tol [float,] Tolerance for the stopping condition.

method [{'lars', 'cd'}] lars: uses the least angle regression method to solve the lasso problem (linear_model.lars_path) cd: uses the coordinate descent method to compute the Lasso solution (linear_model.Lasso). Lars will be faster if the estimated components are sparse.

n_jobs [int,] Number of parallel jobs to run.

U_init [array of shape (n_samples, n_atoms),] Initial values for the loadings for warm restart scenarios.

V_init [array of shape (n_atoms, n_features),] Initial values for the components for warm restart scenarios.

verbose :

•Degree of verbosity of the printed output.

random_state [int or RandomState] Pseudo number generator state used for random sampling.

Attributes

components_ [array, [n_components, n_features]] Sparse components extracted from the data.

error_ [array] Vector of errors at each iteration.

See also

PCA MiniBatchSparsePCA DictionaryLearning

Full API documentation: `SparsePCAScikitsLearnNode`

class `mdp.nodes.`**`OrthogonalMatchingPursuitScikitsLearnNode`**
Orthogonal Mathching Pursuit model (OMP)

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

Parameters

n_nonzero_coefs [int, optional] Desired number of non-zero entries in the solution. If None (by default) this value is set to 10% of n_features.

tol [float, optional] Maximum norm of the residual. If not None, overrides n_nonzero_coefs.

fit_intercept [boolean, optional] 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).

normalize [boolean, optional] If False, the regressors X are assumed to be already normalized.

precompute_gram [{True, False, 'auto'},] Whether to use a precomputed Gram and Xy matrix to speed up calculations. Improves performance when *n_targets* or *n_samples* is very large. Note that if you already have such matrices, you can pass them directly to the fit method.

copy_X [bool, optional] Whether the design matrix X must be copied by the algorithm. A false value is only helpful if X is already Fortran-ordered, otherwise a copy is made anyway.

copy_Gram [bool, optional] Whether the gram matrix must be copied by the algorithm. A false value is only helpful if X is already Fortran-ordered, otherwise a copy is made anyway.

copy_Xy [bool, optional] Whether the covariance vector Xy must be copied by the algorithm. If False, it may be overwritten.

Attributes

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

intercept_ [float or array, shape =(n_targets,)] independent term in decision function.

Notes

Orthogonal matching pursuit was introduced in G. Mallat, Z. Zhang, Matching pursuits with time-frequency dictionaries, IEEE Transactions on Signal Processing, Vol. 41, No. 12. (December 1993), pp. 3397-3415. (http://blanche.polytechnique.fr/~mallat/papiers/MallatPursuit93.pdf)

This implementation is based on Rubinstein, R., Zibulevsky, M. and Elad, M., Efficient Implementation of the K-SVD Algorithm using Batch Orthogonal Matching Pursuit Technical Report - CS Technion, April 2008. http://www.cs.technion.ac.il/~ronrubin/Publications/KSVD-OMP-v2.pdf

See also

orthogonal_mp orthogonal_mp_gram lars_path Lars LassoLars decomposition.sparse_encode

Full API documentation: `OrthogonalMatchingPursuitScikitsLearnNode`

class mdp.nodes.**SelectFprScikitsLearnNode**
Filter: Select the pvalues below alpha based on a FPR test.

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

FPR test stands for False Positive Rate test. It controls the total amount of false detections.

Parameters

score_func: callable Function taking two arrays X and y, and returning 2 arrays:

- both scores and pvalues

alpha: float, optional The highest p-value for features to be kept

Full API documentation: SelectFprScikitsLearnNode

class mdp.nodes.**LabelEncoderScikitsLearnNode**
Encode labels with value between 0 and n_classes-1.

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

Attributes

***classes_*: array of shape [n_class]** Holds the label for each class.

Examples

LabelEncoder can be used to normalize labels.

```
>>> from sklearn import preprocessing
>>> le = preprocessing.LabelEncoder()
>>> le.fit([1, 2, 2, 6])
LabelEncoder()
>>> le.classes_
array([1, 2, 6])
>>> le.transform([1, 1, 2, 6])
array([0, 0, 1, 2])
>>> le.inverse_transform([0, 0, 1, 2])
array([1, 1, 2, 6])
```

It can also be used to transform non-numerical labels (as long as they are hashable and comparable) to numerical labels.

```
>>> le = preprocessing.LabelEncoder()
>>> le.fit(["paris", "paris", "tokyo", "amsterdam"])
LabelEncoder()
>>> list(le.classes_)
['amsterdam', 'paris', 'tokyo']
>>> le.transform(["tokyo", "tokyo", "paris"])
array([2, 2, 1])
>>> list(le.inverse_transform([2, 2, 1]))
['tokyo', 'tokyo', 'paris']
```

Full API documentation: LabelEncoderScikitsLearnNode

class mdp.nodes.**QDAScikitsLearnNode**
Quadratic Discriminant Analysis (QDA)

This node has been automatically generated by wrapping the sklearn.qda.QDA class from the sklearn library. The wrapped instance can be accessed through the scikits_alg attribute.

A classifier with a quadratic decision boundary, generated by fitting class conditional densities to the data and using Bayes' rule.

The model fits a Gaussian density to each class.

Parameters

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 sklearn.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

sklearn.lda.LDA: Linear discriminant analysis

Full API documentation: `QDAScikitsLearnNode`

class `mdp.nodes.`**`LogOddsEstimatorScikitsLearnNode`**

This node has been automatically generated by wrapping the `sklearn.ensemble.gradient_boosting.LogOddsEs` class from the `sklearn` library. The wrapped instance can be accessed through the `scikits_alg` attribute.

Full API documentation: `LogOddsEstimatorScikitsLearnNode`

class `mdp.nodes.`**`VectorizerScikitsLearnNode`**

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

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 `sklearn.linear_model.stochastic_gradient.SGD` class from the `sklearn` 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 or sparse 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, optional The number of passes over the training data (aka epochs). Defaults to 5.

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

seed: int, optional The seed of the pseudo random number generator to use when shuffling the data.

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.

learning_rate [string, optional] The learning rate:

- constant: eta = eta0
- optimal: eta = 1.0/(t+t0) [default]
- invscaling: eta = eta0 / pow(t, power_t)

eta0 [double] The initial learning rate [default 0.01].

power_t [double] The exponent for inverse scaling learning rate [default 0.25].

class_weight [dict, {class_label][weight} or "auto" or None, optional] Preset for the class_weight fit parameter.

Weights associated with classes. If not given, all classes are supposed to have weight one.

The "auto" mode uses the values of y to automatically adjust weights inversely proportional to class frequencies.

warm_start [bool, optional] When set to True, reuse the solution of the previous call to fit as initialization, otherwise, just erase the previous solution.

Attributes

*coef_* : array, shape = [1, n_features] if n_classes == 2 else [n_classes, n_features]

Weights assigned to the features.

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

Examples

```
>>> import numpy as np
>>> from sklearn 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(alpha=0.0001, class_weight=None, epsilon=0.1, eta0=0.0,
        fit_intercept=True, learning_rate='optimal', loss='hinge',
        n_iter=5, n_jobs=1, penalty='l2', power_t=0.5, rho=0.85, seed=0,
        shuffle=False, verbose=0, warm_start=False)
```

```
>>> print(clf.predict([[-0.8, -1]]))
[1]
```

See also

LinearSVC, LogisticRegression, Perceptron

Full API documentation: `SGDClassifierScikitsLearnNode`

class `mdp.nodes.`**`LassoLarsScikitsLearnNode`**

Lasso model fit with Least Angle Regression a.k.a. Lars

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

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

The optimization objective for Lasso is:

```
(1 / (2 * n_samples)) * ||y - Xw||^2_2 + alpha * ||w||_1
```

Parameters

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).

verbose [boolean or integer, optional] Sets the verbosity amount

normalize [boolean, optional] If True, the regressors X are normalized

copy_X [boolean, optional, default True] If True, X will be copied; else, it may be overwritten.

precompute [True | False | 'auto' | array-like] Whether to use a precomputed Gram matrix to speed up calculations. If set to 'auto' let us decide. The Gram matrix can also be passed as argument.

max_iter: integer, optional Maximum number of iterations to perform.

eps: float, optional The machine-precision regularization in the computation of the Cholesky diagonal factors. Increase this for very ill-conditioned systems. Unlike the 'tol' parameter in some iterative optimization-based algorithms, this parameter does not control the tolerance of the optimization.

fit_path [boolean] If True the full path is stored in the *coef_path_* attribute. If you compute the solution for a large problem or many targets, setting fit_path to False will lead to a speedup, especially with a small alpha.

Attributes

coef_path_ [array, shape = [n_features, n_alpha]] The varying values of the coefficients along the path. It is not present if fit_path parameter is False.

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

intercept_ [float] Independent term in decision function.

Examples

```
>>> from sklearn import linear_model
>>> clf = linear_model.LassoLars(alpha=0.01)
>>> clf.fit([[-1, 1], [0, 0], [1, 1]], [-1, 0, -1])
...
LassoLars(alpha=0.01, copy_X=True, eps=..., fit_intercept=True,
     fit_path=True, max_iter=500, normalize=True, precompute='auto',
     verbose=False)
>>> print(clf.coef_)
[ 0.         -0.963257...]
```

See also

lars_path lasso_path Lasso LassoCV LassoLarsCV sklearn.decomposition.sparse_encode

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

Full API documentation: `LassoLarsScikitsLearnNode`

class `mdp.nodes.`**`KernelPCAScikitsLearnNode`**

Kernel Principal component analysis (KPCA)

This node has been automatically generated by wrapping the `sklearn.decomposition.kernel_pca.KernelPCA` class from the `sklearn` library. The wrapped instance can be accessed through the `scikits_alg` attribute.

Non-linear dimensionality reduction through the use of kernels.

Parameters

n_components: int or None Number of components. If None, all non-zero components are kept.

kernel: "linear" | "poly" | "rbf" | "sigmoid" | "precomputed" Kernel. Default: "linear"

degree [int, optional] Degree for poly, rbf and sigmoid kernels. Default: 3.

gamma [float, optional] Kernel coefficient for rbf and poly kernels. Default: 1/n_features.

coef0 [float, optional] Independent term in poly and sigmoid kernels.

alpha: int Hyperparameter of the ridge regression that learns the inverse transform (when fit_inverse_transform=True). Default: 1.0

fit_inverse_transform: bool Learn the inverse transform for non-precomputed kernels. (i.e. learn to find the pre-image of a point) Default: False

eigen_solver: string ['auto'|'dense'|'arpack'] Select eigensolver to use. If n_components is much less than the number of training samples, arpack may be more efficient than the dense eigensolver.

tol: float convergence tolerance for arpack. Default: 0 (optimal value will be chosen by arpack)

max_iter [int] maximum number of iterations for arpack Default: None (optimal value will be chosen by arpack)

Attributes

*lambdas_*, *alphas_*:

- Eigenvalues and eigenvectors of the centered kernel matrix

*dual_coef_*:

- Inverse transform matrix

*X_transformed_fit_*:

- Projection of the fitted data on the kernel principal components

References

Kernel PCA was intoduced in:

- Bernhard Schoelkopf, Alexander J. Smola,
- and Klaus-Robert Mueller. 1999. Kernel principal
- component analysis. In Advances in kernel methods,
- MIT Press, Cambridge, MA, USA 327-352.

Full API documentation: `KernelPCAScikitsLearnNode`

class `mdp.nodes.`**`ScalerScikitsLearnNode`**

Standardize features by removing the mean and scaling to unit variance

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

Centering and scaling happen indepently on each feature by computing the relevant statistics on the samples in the training set. Mean and standard deviation are then stored to be used on later data using the *transform* method.

Standardization of a dataset is a common requirement for many machine learning estimators: they might behave badly if the individual feature do not more or less look like standard normally distributed data (e.g. Gaussian with 0 mean and unit variance).

For instance many elements used in the objective function of a learning algorithm (such as the RBF kernel of Support Vector Machines or the L1 and L2 regularizers of linear models) assume that all features are centered around 0 and have variance in the same order. If a feature has a variance that is orders of magnitude larger that others, it might dominate the objective function and make the estimator unable to learn from other features correctly as expected.

Parameters

with_mean [boolean, True by default] If True, center the data before scaling.

with_std [boolean, True by default] If True, scale the data to unit variance (or equivalently, unit standard deviation).

copy [boolean, optional, default is True] set to False to perform inplace row normalization and avoid a copy (if the input is already a numpy array or a scipy.sparse CSR matrix and if axis is 1).

Attributes

mean_ [array of floats with shape [n_features]] The mean value for each feature in the training set.

std_ [array of floats with shape [n_features]] The standard deviation for each feature in the training set.

See also

`sklearn.preprocessing.scale()` to perform centering and scaling without using the `Transformer` object oriented API

`sklearn.decomposition.RandomizedPCA` with *whiten=True* to further remove the linear correlation across features.

Full API documentation: `ScalerScikitsLearnNode`

class `mdp.nodes.`**`CCAScikitsLearnNode`**

CCA Canonical Correlation Analysis. CCA inherits from PLS with mode="B" and deflation_mode="canonical".

This node has been automatically generated by wrapping the `sklearn.pls.CCA` class from the `sklearn` library. The wrapped instance can be accessed through the `scikits_alg` attribute.

Parameters

X [array-like of predictors, shape = [n_samples, p]] Training vectors, where n_samples in the number of samples and p is the number of predictors.

Y [array-like of response, shape = [n_samples, q]] Training vectors, where n_samples in the number of samples and q is the number of response variables.

n_components [int, (default 2).] number of components to keep.

scale [boolean, (default True)] whether to scale the data?

max_iter [an integer, (default 500)] the maximum number of iterations of the NIPALS inner loop (used only if algorithm="nipals")

tol [non-negative real, default 1e-06.] the tolerance used in the iterative algorithm

copy [boolean] Whether the deflation be done on a copy. Let the default value to True unless you don't care about side effects

Attributes

x_weights_ [array, [p, n_components]] X block weights vectors.

y_weights_ [array, [q, n_components]] Y block weights vectors.

x_loadings_ [array, [p, n_components]] X block loadings vectors.

y_loadings_ [array, [q, n_components]] Y block loadings vectors.

x_scores_ [array, [n_samples, n_components]] X scores.

y_scores_ [array, [n_samples, n_components]] Y scores.

x_rotations_ [array, [p, n_components]] X block to latents rotations.

y_rotations_ [array, [q, n_components]] Y block to latents rotations.

Notes

For each component k, find the weights u, v that maximizes max corr(Xk u, Yk v), such that `|u| = |v| = 1`

Note that it maximizes only the correlations between the scores.

The residual matrix of X (Xk+1) block is obtained by the deflation on the current X score: x_score.

The residual matrix of Y (Yk+1) block is obtained by deflation on the current Y score.

Examples

```
>>> from sklearn.pls import PLSCanonical, PLSRegression, CCA
>>> X = [[0., 0., 1.], [1.,0.,0.], [2.,2.,2.], [3.,5.,4.]]
>>> Y = [[0.1, -0.2], [0.9, 1.1], [6.2, 5.9], [11.9, 12.3]]
>>> cca = CCA(n_components=1)
>>> cca.fit(X, Y)
...
CCA(copy=True, max_iter=500, n_components=1, scale=True, tol=1e-06)
>>> X_c, Y_c = cca.transform(X, Y)
```

References

Jacob A. Wegelin. A survey of Partial Least Squares (PLS) methods, with emphasis on the two-block case. Technical Report 371, Department of Statistics, University of Washington, Seattle, 2000.

In french but still a reference:

Tenenhaus, M. (1998). La regression PLS: theorie et pratique. Paris:

Editions Technic.

See also

PLSCanonical PLSSVD

Full API documentation: `CCAScikitsLearnNode`

class `mdp.nodes.`**`KernelCentererScikitsLearnNode`**

Center a kernel matrix

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

This is equivalent to centering phi(X) with sklearn.preprocessing.Scaler(with_std=False).

Full API documentation: `KernelCentererScikitsLearnNode`

class `mdp.nodes.`**`SelectFdrScikitsLearnNode`**

Filter: Select the p-values for an estimated false discovery rate

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

This uses the Benjamini-Hochberg procedure. `alpha` is the target false discovery rate.

Parameters

score_func: callable Function taking two arrays X and y, and returning 2 arrays:

- both scores and pvalues

alpha: float, optional The highest uncorrected p-value for features to keep

Full API documentation: `SelectFdrScikitsLearnNode`

class `mdp.nodes.`**`ExtraTreeClassifierScikitsLearnNode`**
An extremely randomized tree classifier.

This node has been automatically generated by wrapping the `sklearn.tree.tree.ExtraTreeClassifier` class from the `sklearn` library. The wrapped instance can be accessed through the `scikits_alg` attribute.

Extra-trees differ from classic decision trees in the way they are built. When looking for the best split to separate the samples of a node into two groups, random splits are drawn for each of the *max_features* randomly selected features and the best split among those is chosen. When *max_features* is set 1, this amounts to building a totally random decision tree.

Warning: Extra-trees should only be used within ensemble methods.

See also

ExtraTreeRegressor, ExtraTreesClassifier, ExtraTreesRegressor

References

Full API documentation: `ExtraTreeClassifierScikitsLearnNode`

class `mdp.nodes.`**`SelectKBestScikitsLearnNode`**
Filter: Select the k lowest p-values.

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

Parameters

score_func: callable Function taking two arrays X and y, and returning 2 arrays:

- both scores and pvalues

k: int, optional Number of top features to select.

Notes

Ties between features with equal p-values will be broken in an unspecified way.

Full API documentation: `SelectKBestScikitsLearnNode`

class `mdp.nodes.`**`NormalizerScikitsLearnNode`**
Normalize samples individually to unit norm

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

Each sample (i.e. each row of the data matrix) with at least one non zero component is rescaled independently of other samples so that its norm (l1 or l2) equals one.

This transformer is able to work both with dense numpy arrays and scipy.sparse matrix (use CSR format if you want to avoid the burden of a copy / conversion).

Scaling inputs to unit norms is a common operation for text classification or clustering for instance. For instance the dot product of two l2-normalized TF-IDF vectors is the cosine similarity of the vectors and is the base similarity metric for the Vector Space Model commonly used by the Information Retrieval community.

Parameters

norm ['l1' or 'l2', optional ('l2' by default)] The norm to use to normalize each non zero sample.

copy [boolean, optional, default is True] set to False to perform inplace row normalization and avoid a copy (if the input is already a numpy array or a scipy.sparse CSR matrix).

Notes

This estimator is stateless (besides constructor parameters), the fit method does nothing but is useful when used in a pipeline.

See also

`sklearn.preprocessing.normalize()` equivalent function without the object oriented API

Full API documentation: `NormalizerScikitsLearnNode`

class `mdp.nodes.`**`TfidfTransformerScikitsLearnNode`**

Transform a count matrix to a normalized tf or tf–idf representation

This node has been automatically generated by wrapping the `sklearn.feature_extraction.text.TfidfTransfo` class from the `sklearn` 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. This is a common term weighting scheme in information retrieval, that has also found good use in document classification.

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 features that occur in a small fraction of the training corpus.

In the SMART notation used in IR, this class implements several tf–idf variants. Tf is always "n" (natural), idf is "t" iff use_idf is given, "n" otherwise, and normalization is "c" iff norm='l2', "n" iff norm=None.

Parameters

norm ['l1', 'l2' or None, optional] Norm used to normalize term vectors. None for no normalization.

use_idf [boolean, optional] Enable inverse-document-frequency reweighting.

smooth_idf [boolean, optional] Smooth idf weights by adding one to document frequencies, as if an extra document was seen containing every term in the collection exactly once. Prevents zero divisions.

sublinear_tf [boolean, optional] Apply sublinear tf scaling, i.e. replace tf with 1 + log(tf).

References

Full API documentation: `TfidfTransformerScikitsLearnNode`

class `mdp.nodes.`**`GradientBoostingClassifierScikitsLearnNode`**

Gradient Boosting for classification.

This node has been automatically generated by wrapping the `sklearn.ensemble.gradient_boosting.GradientB` class from the `sklearn` library. The wrapped instance can be accessed through the `scikits_alg` attribute.

GB builds an additive model in a forward stage-wise fashion; it allows for the optimization of arbitrary differentiable loss functions. In each stage `n_classes_` regression trees are fit on the negative gradient of the binomial or multinomial deviance loss function. Binary classification is a special case where only a single regression tree is induced.

Parameters

loss [{'deviance'}, optional (default='deviance')] loss function to be optimized. 'deviance' refers to deviance (= logistic regression) for classification with probabilistic outputs.

learn_rate [float, optional (default=0.1)] learning rate shrinks the contribution of each tree by *learn_rate*. There is a trade-off between learn_rate and n_estimators.

n_estimators [int (default=100)] The number of boosting stages to perform. Gradient boosting is fairly robust to over-fitting so a large number usually results in better performance.

max_depth [integer, optional (default=3)] maximum depth of the individual regression estimators. The maximum depth limits the number of nodes in the tree. Tune this parameter for best performance; the best value depends on the interaction of the input variables.

min_samples_split [integer, optional (default=1)] The minimum number of samples required to split an internal node.

min_samples_leaf [integer, optional (default=1)] The minimum number of samples required to be at a leaf node.

subsample [float, optional (default=1.0)] The fraction of samples to be used for fitting the individual base learners. If smaller than 1.0 this results in Stochastic Gradient Boosting. *subsample* interacts with the parameter *n_estimators*. Choosing *subsample < 1.0* leads to a reduction of variance and an increase in bias.

max_features [int, None, optional (default=None)] The number of features to consider when looking for the best split. Features are choosen randomly at each split point. If None, then *max_features=n_features*. Choosing *max_features < n_features* leads to a reduction of variance and an increase in bias.

Attributes

feature_importances_ [array, shape = [n_features]] The feature importances (the higher, the more important the feature).

oob_score_ [array, shape = [n_estimators]] Score of the training dataset obtained using an out-of-bag estimate. The i-th score `oob_score_[i]` is the deviance (= loss) of the model at iteration `i` on the out-of-bag sample.

train_score_ [array, shape = [n_estimators]] The i-th score `train_score_[i]` is the deviance (= loss) of the model at iteration `i` on the in-bag sample. If `subsample == 1` this is the deviance on the training data.

Examples

```
>>> samples = [[0, 0, 2], [1, 0, 0]]
>>> labels = [0, 1]
>>> from sklearn.ensemble import GradientBoostingClassifier
>>> gb = GradientBoostingClassifier().fit(samples, labels)
>>> print gb.predict([[0.5, 0, 0]])
[0]
```

See also

sklearn.tree.DecisionTreeClassifier, RandomForestClassifier

References

J. Friedman, Greedy Function Approximation: A Gradient Boosting Machine, The Annals of Statistics, Vol. 29, No. 5, 2001.

10.Friedman, Stochastic Gradient Boosting, 1999

T. Hastie, R. Tibshirani and J. Friedman. Elements of Statistical Learning Ed. 2, Springer, 2009.

Full API documentation: `GradientBoostingClassifierScikitsLearnNode`

class `mdp.nodes.`**`GMMHMMScikitsLearnNode`**

Hidden Markov Model with Gaussin mixture emissions

This node has been automatically generated by wrapping the `sklearn.hmm.GMMHMM` class from the `sklearn` library. The wrapped instance can be accessed through the `scikits_alg` attribute.

Attributes

init_params [string, optional] Controls which parameters are initialized prior to training. Can contain any combination of 's' for startprob, 't' for transmat, 'm' for means, and 'c' for covars, etc. Defaults to all parameters.

params [string, optional] Controls which parameters are updated in the training process. Can contain any combination of 's' for startprob, 't' for transmat,'m' for means, and 'c' for covars, etc. Defaults to all parameters.

n_components [int] Number of states in the model.

transmat [array, shape (*n_components*, *n_components*)] Matrix of transition probabilities between states.

startprob [array, shape ('n_components',)] Initial state occupation distribution.

gmms [array of GMM objects, length *n_components*] GMM emission distributions for each state.

random_state [RandomState or an int seed (0 by default)] A random number generator instance

n_iter [int, optional] Number of iterations to perform.

thresh [float, optional] Convergence threshold.

Examples

```
>>> from sklearn.hmm import GMMHMM
>>> GMMHMM(n_components=2, n_mix=10, covariance_type='diag')
...
GMMHMM(algorithm='viterbi', covariance_type='diag',...
```

See Also

GaussianHMM : HMM with Gaussian emissions

Full API documentation: `GMMHMMScikitsLearnNode`

class `mdp.nodes.`**`DecisionTreeRegressorScikitsLearnNode`**

A tree regressor.

This node has been automatically generated by wrapping the `sklearn.tree.tree.DecisionTreeRegressor` class from the `sklearn` library. The wrapped instance can be accessed through the `scikits_alg` attribute.

Parameters

criterion [string, optional (default="mse")] The function to measure the quality of a split. The only supported criterion is "mse" for the mean squared error.

max_depth [integer or None, optional (default=None)] The maximum depth of the tree. If None, then nodes are expanded until all leaves are pure or until all leaves contain less than min_samples_split samples.

min_samples_split [integer, optional (default=1)] The minimum number of samples required to split an internal node.

min_samples_leaf [integer, optional (default=1)] The minimum number of samples required to be at a leaf node.

min_density [float, optional (default=0.1)] This parameter controls a trade-off in an optimization heuristic. It controls the minimum density of the *sample_mask* (i.e. the fraction of samples in the mask). If the density falls below this threshold the mask is recomputed and the input data is packed which results in data copying. If *min_density* equals to one, the partitions are always represented as copies of the original data. Otherwise, partitions are represented as bit masks (aka sample masks).

max_features [int, string or None, optional (default=None)] The number of features to consider when looking for the best split. If "auto", then *max_features=sqrt(n_features)* on classification tasks and *max_features=n_features* on regression problems. If "sqrt", then *max_features=sqrt(n_features)*. If "log2", then *max_features=log2(n_features)*. If None, then *max_features=n_features*.

compute_importances [boolean, optional (default=True)] Whether feature importances are computed and stored into the `feature_importances_` attribute when calling fit.

random_state [int, RandomState instance or None, optional (default=None)] If int, random_state is the seed used by the random number generator; If RandomState instance, random_state is the random

number generator; If None, the random number generator is the RandomState instance used by *np.random*.

Attributes

tree_ [Tree object] The underlying Tree object.

feature_importances_ [array of shape = [n_features]] The feature importances (the higher, the more important the feature). The importance I(f) of a feature f is computed as the (normalized) total reduction of error brought by that feature. It is also known as the Gini importance [4]_.

See also

DecisionTreeClassifier

References

Examples

```
>>> from sklearn.datasets import load_boston
>>> from sklearn.cross_validation import cross_val_score
>>> from sklearn.tree import DecisionTreeRegressor

>>> boston = load_boston()
>>> regressor = DecisionTreeRegressor(random_state=0)
```

R2 scores (a.k.a. coefficient of determination) over 10-folds CV:

```
>>> cross_val_score(regressor, boston.data, boston.target, cv=10)
...
...
array([ 0.61..., 0.57..., -0.34..., 0.41..., 0.75...,
        0.07..., 0.29..., 0.33..., -1.42..., -1.77...])
```

Full API documentation: `DecisionTreeRegressorScikitsLearnNode`

class `mdp.nodes.`**`RidgeScikitsLearnNode`**

Linear least squares with l2 regularization.

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

This model solves a regression model where the loss function is the linear least squares function and regularization is given by the l2-norm. Also known as Ridge Regression or Tikhonov regularization. This estimator has built-in support for multi-variate regression (i.e., when y is a 2d-array of shape [n_samples, n_responses]).

Parameters

alpha [float] Small positive values of alpha improve the conditioning of the problem and reduce the variance of the estimates. Alpha corresponds to `(2*C)^-1` in other linear models such as LogisticRegression or LinearSVC.

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).

normalize [boolean, optional] If True, the regressors X are normalized

copy_X [boolean, optional, default True] If True, X will be copied; else, it may be overwritten.

tol [float] Precision of the solution.

Attributes

coef_ [array, shape = [n_features] or [n_responses, n_features]] Weight vector(s).

See also

RidgeClassifier, RidgeCV

Examples

```
>>> from sklearn.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, copy_X=True, fit_intercept=True, normalize=False,
   tol=0.001)
```

Full API documentation: `RidgeScikitsLearnNode`

class `mdp.nodes.`**`SVRScikitsLearnNode`**

epsilon-Support Vector Regression.

This node has been automatically generated by wrapping the `sklearn.svm.classes.SVR` class from the `sklearn` library. The wrapped instance can be accessed through the `scikits_alg` attribute.

The free parameters in the model are C and epsilon.

The implementations is a based on libsvm.

Parameters

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

epsilon [float, optional (default=0.1)] epsilon in the epsilon-SVR model. It specifies the epsilon-tube within which no penalty is associated in the training loss function with points predicted within a distance epsilon from the actual value.

kernel [string, optional (default='rbf')] 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 (default=3)] degree of kernel function is significant only in poly, rbf, sigmoid

gamma [float, optional (default=0.0)] kernel coefficient for rbf and poly, if gamma is 0.0 then 1/n_features will be taken.

coef0 [float, optional (default=0.0)] independent term in kernel function. It is only significant in poly/sigmoid.

probability: boolean, optional (default=False) Whether to enable probability estimates. This must be enabled prior to calling predict_proba.

shrinking: boolean, optional (default=True) Whether to use the shrinking heuristic.

tol: float, optional (default=1e-3) Tolerance for stopping criterion.

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

verbose [bool, default: False] Enable verbose output. Note that this setting takes advantage of a per-process runtime setting in libsvm that, if enabled, may not work properly in a multithreaded context.

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.

*coef_* is readonly property derived from *dual_coef_* and *support_vectors_*

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

Examples

```
>>> from sklearn.svm import SVR
>>> 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 = SVR(C=1.0, epsilon=0.2)
>>> clf.fit(X, y)
SVR(C=1.0, cache_size=200, coef0=0.0, degree=3, epsilon=0.2, gamma=0.0,
  kernel='rbf', probability=False, shrinking=True, tol=0.001,
  verbose=False)
```

See also

NuSVR Support Vector Machine for regression implemented using libsvm using a parameter to control the number of support vectors.

Full API documentation: `SVRScikitsLearnNode`

class `mdp.nodes.`**`RFECVScikitsLearnNode`**

Feature ranking with recursive feature elimination and cross-validated selection of the best number of features.

This node has been automatically generated by wrapping the `sklearn.feature_selection.rfe.RFECV` class from the `sklearn` 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_* attribute that holds the fitted parameters. Important features must correspond to 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* modules.

step [int or float, optional (default=1)] If greater than or equal to 1, then *step* corresponds to the (integer) number of features to remove at each iteration. If within (0.0, 1.0), then *step* corresponds to the percentage (rounded down) of features to remove at each iteration.

cv [int or cross-validation generator, optional (default=None)] If int, it is the number of folds. If None, 3-fold cross-validation is performed by default. Specific cross-validation objects can also be passed, see *sklearn.cross_validation module* for details.

loss_function [function, optional (default=None)] The loss function to minimize by cross-validation. If None, then the score function of the estimator is maximized.

Attributes

n_features_ [int] The number of selected features with cross-validation.

support_ [array of shape [n_features]] The mask of selected features.

ranking_ [array of shape [n_features]] The feature ranking, such that *ranking_[i]* corresponds to the ranking position of the i-th feature. Selected (i.e., estimated best) features are assigned rank 1.

cv_scores_ [array of shape [n_subsets_of_features]] The cross-validation scores such that *cv_scores_[i]* corresponds to the CV score of the i-th subset of features.

Examples

The following example shows how to retrieve the a-priori not known 5 informative features in the Friedman #1 dataset.

```
>>> from sklearn.datasets import make_friedman1
>>> from sklearn.feature_selection import RFECV
```

```
>>> from sklearn.svm import SVR
>>> X, y = make_friedman1(n_samples=50, n_features=10, random_state=0)
>>> estimator = SVR(kernel="linear")
>>> selector = RFECV(estimator, step=1, cv=5)
>>> selector = selector.fit(X, y)
>>> selector.support_
array([ True,  True,  True,  True,  True,
        False, False, False, False, False], dtype=bool)
>>> selector.ranking_
array([1, 1, 1, 1, 1, 6, 4, 3, 2, 5])
```

References

Full API documentation: `RFECVScikitsLearnNode`

class `mdp.nodes.`**BayesianRidgeScikitsLearnNode**

Bayesian ridge regression

This node has been automatically generated by wrapping the `sklearn.linear_model.bayes.BayesianRidge` class from the `sklearn` 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 iterations. Default is 300.

tol [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.

normalize [boolean, optional, default False] If True, the regressors X are normalized

copy_X [boolean, optional, default True] If True, X will be copied; else, it may be overwritten.

verbose [boolean, optional, default False] Verbose mode when fitting the model.

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)

Examples

```
>>> from sklearn import linear_model
>>> clf = linear_model.BayesianRidge()
>>> clf.fit([[0,0], [1, 1], [2, 2]], [0, 1, 2])
...
BayesianRidge(alpha_1=1e-06, alpha_2=1e-06, compute_score=False,
        copy_X=True, fit_intercept=True, lambda_1=1e-06, lambda_2=1e-06,
        n_iter=300, normalize=False, tol=0.001, verbose=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.`**`PLSRegressionScikitsLearnNode`**

PLS regression

This node has been automatically generated by wrapping the `sklearn.pls.PLSRegression` class from the `sklearn` library. The wrapped instance can be accessed through the `scikits_alg` attribute.

PLSRegression implements the PLS 2 blocks regression known as PLS2 or PLS1 in case of one dimensional response. This class inherits from _PLS with mode="A", deflation_mode="regression", norm_y_weights=False and algorithm="nipals".

Parameters

X [array-like of predictors, shape = [n_samples, p]] Training vectors, where n_samples in the number of samples and p is the number of predictors.

Y [array-like of response, shape = [n_samples, q]] Training vectors, where n_samples in the number of samples and q is the number of response variables.

n_components [int, (default 2)] Number of components to keep.

scale [boolean, (default True)] whether to scale the data

max_iter [an integer, (default 500)] the maximum number of iterations of the NIPALS inner loop (used only if algorithm="nipals")

tol [non-negative real] Tolerance used in the iterative algorithm default 1e-06.

copy [boolean, default True] Whether the deflation should be done on a copy. Let the default value to True unless you don't care about side effect

Attributes

x_weights_ [array, [p, n_components]] X block weights vectors.

y_weights_ [array, [q, n_components]] Y block weights vectors.

x_loadings_ [array, [p, n_components]] X block loadings vectors.

y_loadings_ [array, [q, n_components]] Y block loadings vectors.

x_scores_ [array, [n_samples, n_components]] X scores.

y_scores_ [array, [n_samples, n_components]] Y scores.

x_rotations_ [array, [p, n_components]] X block to latents rotations.

y_rotations_ [array, [q, n_components]] Y block to latents rotations.

coefs: array, [p, q] The coeficients of the linear model: Y = X coefs + Err

Notes

For each component k, find weights u, v that optimizes:

`max corr(Xk u, Yk v) * var(Xk u) var(Yk u)`, such that `|u| = 1`

Note that it maximizes both the correlations between the scores and the intra-block variances.

The residual matrix of X (Xk+1) block is obtained by the deflation on the current X score: x_score.

The residual matrix of Y (Yk+1) block is obtained by deflation on the current X score. This performs the PLS regression known as PLS2. This mode is prediction oriented.

This implementation provides the same results that 3 PLS packages provided in the R language (R-project):

- "mixOmics" with function pls(X, Y, mode = "regression")
- "plspm " with function plsreg2(X, Y)
- "pls" with function oscorespls.fit(X, Y)

Examples

```
>>> from sklearn.pls import PLSCanonical, PLSRegression, CCA
>>> X = [[0., 0., 1.], [1.,0.,0.], [2.,2.,2.], [2.,5.,4.]]
>>> Y = [[0.1, -0.2], [0.9, 1.1], [6.2, 5.9], [11.9, 12.3]]
>>> pls2 = PLSRegression(n_components=2)
>>> pls2.fit(X, Y)
...
PLSRegression(copy=True, max_iter=500, n_components=2, scale=True,
        tol=1e-06)
>>> Y_pred = pls2.predict(X)
```

References

Jacob A. Wegelin. A survey of Partial Least Squares (PLS) methods, with emphasis on the two-block case. Technical Report 371, Department of Statistics, University of Washington, Seattle, 2000.

In french but still a reference:

Tenenhaus, M. (1998). La regression PLS: theorie et pratique. Paris:

Editions Technic.

Full API documentation: `PLSRegressionScikitsLearnNode`

class `mdp.nodes.`**`ProbabilisticPCAScikitsLearnNode`**

Additional layer on top of PCA that adds a probabilistic evaluationPrincipal component analysis (PCA)

This node has been automatically generated by wrapping the `sklearn.decomposition.pca.ProbabilisticPCA` class from the `sklearn` 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

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 if `0 < n_components < 1`, select the number of components such that the amount of variance that needs to be explained is greater than the percentage specified by n_components

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_components, n_features]] 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'

Due to implementation subtleties of the Singular Value Decomposition (SVD), which is used in this implementation, running fit twice on the same matrix can lead to principal components with signs flipped (change in direction). For this reason, it is important to always use the same estimator object to transform data in a consistent fashion.

Examples

```
>>> import numpy as np
>>> from sklearn.decomposition 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_)
[ 0.99244...  0.00755...]
```

See also

ProbabilisticPCA RandomizedPCA KernelPCA SparsePCA

Full API documentation: `ProbabilisticPCAScikitsLearnNode`

class `mdp.nodes.`**`LinearRegressionScikitsLearnNode`**

Ordinary least squares Linear Regression.

This node has been automatically generated by wrapping the `sklearn.linear_model.base.LinearRegression` class from the `sklearn` 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.

Parameters

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).

normalize [boolean, optional] If True, the regressors X are normalized

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.`**`LabelBinarizerScikitsLearnNode`**

Binarize labels in a one-vs-all fashion

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

Several regression and binary classification algorithms are available in the scikit. A simple way to extend these algorithms to the multi-class classification case is to use the so-called one-vs-all scheme.

At learning time, this simply consists in learning one regressor or binary classifier per class. In doing so, one needs to convert multi-class labels to binary labels (belong or does not belong to the class). LabelBinarizer makes this process easy with the transform method.

At prediction time, one assigns the class for which the corresponding model gave the greatest confidence. LabelBinarizer makes this easy with the inverse_transform method.

Parameters

neg_label: int (default: 0) Value with which negative labels must be encoded.

pos_label: int (default: 1) Value with which positive labels must be encoded.

Attributes

***classes_*: array of shape [n_class]** Holds the label for each class.

Examples

```
>>> from sklearn import preprocessing
>>> lb = preprocessing.LabelBinarizer()
>>> lb.fit([1, 2, 6, 4, 2])
LabelBinarizer(neg_label=0, pos_label=1)
>>> lb.classes_
array([1, 2, 4, 6])
>>> lb.transform([1, 6])
array([[1, 0, 0, 0],
       [0, 0, 0, 1]])

>>> lb.fit_transform([(1, 2), (3,)])
array([[1, 1, 0],
       [0, 0, 1]])
>>> lb.classes_
array([1, 2, 3])
```

Full API documentation: `LabelBinarizerScikitsLearnNode`

class `mdp.nodes.`**`RadiusNeighborsClassifierScikitsLearnNode`**

Classifier implementing a vote among neighbors within a given radius

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

Parameters

radius [float, optional (default = 1.0)] Range of parameter space to use by default for :meth‘radius_neighbors‘ queries.

weights [str or callable] weight function used in prediction. Possible values:

- ‘uniform’ : uniform weights. All points in each neighborhood are weighted equally.
- ‘distance’ : weight points by the inverse of their distance. in this case, closer neighbors of a query point will have a greater influence than neighbors which are further away.
- [callable] : a user-defined function which accepts an array of distances, and returns an array of the same shape containing the weights.

Uniform weights are used by default.

algorithm [{'auto', 'ball_tree', 'kd_tree', 'brute'}, optional] Algorithm used to compute the nearest neighbors:

- 'ball_tree' will use `BallTree`
- 'kd_tree' will use `scipy.spatial.cKDtree`
- 'brute' will use a brute-force search.
- 'auto' will attempt to decide the most appropriate algorithm based on the values passed to `fit()` method.

Note: fitting on sparse input will override the setting of this parameter, using brute force.

leaf_size [int, optional (default = 30)] Leaf size passed to BallTree or cKDTree. This can affect the speed of the construction and query, as well as the memory required to store the tree. The optimal value depends on the nature of the problem.

p: integer, optional (default = 2) Parameter for the Minkowski metric from sklearn.metrics.pairwise.pairwise_distances. When p = 1, this is equivalent to using manhattan_distance (l1), and euclidean_distance (l2) for p = 2. For arbitrary p, minkowski_distance (l_p) is used.

outlier_label: int, optional (default = None) Label, which is given for outlier samples (samples with no neighbors on given radius). If set to None, ValueError is raised, when outlier is detected.

Examples

```
>>> X = [[0], [1], [2], [3]]
>>> y = [0, 0, 1, 1]
>>> from sklearn.neighbors import RadiusNeighborsClassifier
>>> neigh = RadiusNeighborsClassifier(radius=1.0)
>>> neigh.fit(X, y)
RadiusNeighborsClassifier(...)
>>> print(neigh.predict([[1.5]]))
[0]
```

See also

KNeighborsClassifier RadiusNeighborsRegressor KNeighborsRegressor NearestNeighbors

Notes

See *Nearest Neighbors* in the online documentation for a discussion of the choice of `algorithm` and `leaf_size`.

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

Full API documentation: `RadiusNeighborsClassifierScikitsLearnNode`

class `mdp.nodes.`**`RidgeClassifierCVScikitsLearnNode`**

Ridge classifier with built-in cross-validation.

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

By default, it performs Generalized Cross-Validation, which is a form of efficient Leave-One-Out cross-validation. Currently, only the n_features > n_samples case is handled efficiently.

Parameters

alphas: numpy array of shape [n_alphas] Array of alpha values to try. Small positive values of alpha improve the conditioning of the problem and reduce the variance of the estimates. Alpha corresponds to (2*C)^-1 in other linear models such as LogisticRegression or LinearSVC.

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).

normalize [boolean, optional] If True, the regressors X are normalized

score_func: callable, optional function that takes 2 arguments and compares them in order to evaluate the performance of prediction (big is good) if None is passed, the score of the estimator is maximized

loss_func: callable, optional function that takes 2 arguments and compares them in order to evaluate the performance of prediction (small is good) if None is passed, the score of the estimator is maximized

cv [cross-validation generator, optional] If None, Generalized Cross-Validation (efficient Leave-One-Out) will be used.

class_weight [dict, optional] Weights associated with classes in the form {class_label : weight}. If not given, all classes are supposed to have weight one.

Attributes

cv_values_ [array, shape = [n_samples, n_alphas] or shape = [n_samples, n_responses, n_alphas], optional] Cross-validation values for each alpha (if *store_cv_values*=*True* and

cv=*None*). After *fit()* has been called, this attribute will contain the mean squared errors (by default) or the values of the *{loss,score}_func* function (if provided in the constructor).

coef_ [array, shape = [n_features] or [n_responses, n_features]] Weight vector(s).

alpha_ [float] Estimated regularization parameter

See also

Ridge: Ridge regression RidgeClassifier: Ridge classifier RidgeCV: Ridge regression with built-in cross validation

Notes

For multi-class classification, n_class classifiers are trained in a one-versus-all approach. Concretely, this is implemented by taking advantage of the multi-variate response support in Ridge.

Full API documentation: `RidgeClassifierCVScikitsLearnNode`

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 in the Examples section *slideshow*.

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) 2003-2012, 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.
```

BIBLIOGRAPHY

[NLNS2002] *H.B. Nielsen, S.N. Lophaven, H. B. Nielsen and J. Sondergaard. DACE - A MATLAB Kriging Toolbox.* (2002) http://www2.imm.dtu.dk/~hbn/dace/dace.pdf

[WBSWM1992] *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

[Halko2009] *Finding structure with randomness: Stochastic algorithms for constructing approximate matrix decompositions Halko, et al., 2009 (arXiv:909)*

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

[Yates2011] *R. Baeza-Yates and B. Ribeiro-Neto (2011). Modern Information Retrieval. Addison Wesley, pp. 68–74.*

[MSR2008] *C.D. Manning, H. Schütze and P. Raghavan (2008). Introduction to Information Retrieval. Cambridge University Press, pp. 121–125.*

INDEX

S

T

V

W

X