PyMVPA Manual

Release 0.4.0

**Michael Hanke, Yaroslav O. Halchenko,
Per B. Sederberg, James M. Hughes**

CONTENTS

The PDF version of the manual is available for download.

INTRODUCTION

PyMVPA is a Python module intended to ease pattern classification analysis of large datasets. It provides high-level abstraction of typical processing steps and a number of implementations of some popular algorithms. While it is not limited to neuroimaging data it is eminently suited for such datasets. PyMVPA is truly free software (in every respect) and additionally requires nothing but free software to run. Theoretically PyMVPA should run on anything that can run a Python interpreter, although the proof is yet to come.

PyMVPA stands for *Multivariate Pattern Analysis* in Python.

1.1 What this Manual is NOT

This manual does not make an attempt to be a comprehensive introduction into machine learning *theory*. There is a wealth of high-quality text books about this field available. A very good example is: Pattern Recognition and Machine Learning by Christopher M. Bishop.

A good starting point to learn about the application of machine learning algorithms to (f)MRI data are two recent reviews by *Norman et al. (2006)* and *Haynes and Rees (2006)*.

This manual also does not describe every technical bit and piece of the PyMVPA package, but is instead focused on the user perspective. Developers shoud have a look at the API documentation, which is a detailed, comprehensive and up-to-date description of the whole package. Users looking for an overview of the public programming interface of the framework are referred to the *Module Reference*. The *Module Reference* is similar to the API reference, but hides overly technical information, which are only relevant for people intending to extend the framework by adding more functionality.

More examples and usage patterns extending the ones described here can be taken from the examples shipped with the PyMVPA source distribution (*doc/examples/*; some of them are also available in the *Full Examples* chapter of this manual) or even the unit test battery, also part of the source distribution (in the *tests/* directory).

1.2 A bit of History

The roots of PyMVPA date back to early 2005. At that time it was a C++ library (no Python yet) developed by Michael Hanke and Sebastian Krüger, intended to make it easy to apply artificial neural networks to pattern recognition problems.

During a visit to Princeton University in spring 2005, Michael Hanke was introduced to the MVPA toolbox for Matlab, which had several advantages over a C++ library. Most importantly it was easier to use. While a user of a C++ library is forced to write a significant amount of front-end code, users of the MVPA toolbox could simply load their data and start analyzing it, providing a common interface to functions drawn from a variety of libraries.

However, there are some disadvantages when writing a toolbox in Matlab. While users in general benefit from the powers of Matlab, they are at the same time bound to the goodwill of a commercial company. That this is indeed a problem becomes obvious when one considers the time when the vendor of Matlab was not willing to support the Mac platform. Therefore even if the MVPA toolbox is GPL-licensed it cannot fully benefit from the enormous advantages of the free software development model environment (free as in free speech, not only free beer).

For these reasons, Michael thought that a successor to the C++ library should remain truly free software, remain fully object-oriented (in contrast to the MVPA toolbox), but should be at least as easy to use and extensible as the MVPA toolbox.

After evaluating some possibilities Michael decided that Python is the most promising candidate that was fully capable of fulfilling the intended development goal. Python is a very powerful language that magically combines the possibility to write really fast code and a simplicity that allows one to learn the basic concepts within a few days. One of the major advantages of Python is the availability of a huge amount of so called *modules*. Modules can include extensions written in a hardcore language like C (or even FORTRAN) and therefore allow one to incorporate high-performance code without having to leave the Python environment. Additionally some Python modules even provide links to other toolkits. For example RPy allows to use the full functionality of R from inside Python. Even Matlab can be used via some Python modules (see PyMatlab for an example).

After the decision for Python was made, Michael started development with a simple k-Nearest-Neighbour classifier and a cross-validation class. Using the mighty NumPy package made it easy to support data of any dimensionality. Therefore PyMVPA can easily be used with 4d fMRI dataset, but equally well with EEG/MEG data (3d) or even non-neuroimaging datasets. By September 2007 PyMVPA included support for reading and writing datasets from and to the NIfTI format, kNN and Support Vector Machine classifiers, as well as several analysis algorithms (e.g. searchlight and incremental feature search).

During another visit in Princeton in October 2007 Michael met with Yaroslav Halchenko and Per B. Sederberg. That incident and the following discussions and hacking sessions of Michael and Yaroslav lead to a major refactoring of the PyMVPA codebase, making it much more flexible/extensible, faster and easier than it has ever been before.

1.3 Authors & Contributors

The PyMVPA developers team currently consists of:

- Michael Hanke, University of Magdeburg, Germany

- Yaroslav O. Halchenko, Rutgers University Newark, USA

- Per B. Sederberg, Princeton University, USA

- Emanuele Olivetti, University of Trento, Italy

We are very grateful to the following people, who have contributed valueable advice, code or documentation to PyMVPA:

- Greg Detre, Princeton University, USA

- James M. Hughes, Dartmouth College, USA

- Ingo Fründ, University of Magdeburg, Germany

1.4 How to cite PyMVPA

Below is a list of all publications about PyMVPA that have been published so far (in chronological order). If you use PyMVPA in your research please cite the one that matches best.

Hanke, M., Halchenko, Y. O., Sederberg, P. B., Hanson, S. J., Haxby, J. V. & Pollmann, S. (2008). PyMVPA: A Python toolbox for classifier-based data analysis.
> First presentation of PyMVPA at the conference *Psychologie und Gehirn* [Psychology and Brain], Magdeburg, 2008. This poster received the poster prize of the *German Society for Psychophysiology and its Application*.

Hanke, M., Halchenko, Y. O., Sederberg, P. B., Hanson, S. J., Haxby, J. V. & Pollmann, S. (in press). PyMVPA: A Python toolbox for multivariate pattern analysis of fMRI data. Neuroinformatics.

First paper introducing fMRI data analysis with PyMVPA.

Hanke, M., Halchenko, Y. O., Sederberg, P. B., Hanson, S. J., Haxby, J. V. & Pollmann, S. (2008). PyMVPA: A Python toolbox for machine-learning based data analysis.

Poster emphasizing PyMVPA's capabilities concerning multi-modal data analysis at the annual meeting of the Society for Neuroscience, Washington, 2008.

1.5 Acknowledgements

We are greatful to the developers and contributers of NumPy, SciPy and IPython for providing an excellent Python-based computing environment.

Additionally, as PyMVPA makes use of a lot of external software packages (e.g. classifier implementations), we want to acknowledge the authors of the respective tools and libraries (e.g. LIBSVM or Shogun) and thank them for developing their packages as free and open source software.

Finally, we would like to express our acknowledgements to the Debian project for providing us with hosting facilities for mailing lists and source code repositories. But most of all for developing the *universal operating system*.

INSTALLATION

This section covers the necessary steps to install and run PyMVPA. It contains a comprehensive list of software dependencies, as well as recommendation for additional software packages that further enhance the functionality provided by PyMVPA.

2.1 Dependencies

PyMVPA is designed to be able to easily interface with various libraries and computing environments. However, most of these external software packages only enhance functionality built into PyMVPA or add a different flavor of some algorithm (e.g. yet another classifier). In fact, the framework itself has only two mandatory dependencies (see below), which are known to be very portable. It is therefore possible to run PyMVPA on a wide variety of platforms and operating systems, ranging from computing mainframes, to regular desktop machines. It even runs on a cell phone.

This picture shows PyMVPA on an OpenMoko cell phone — running the *pylab_2d.py* example in an IPython session.

Note: In general a phone might not be the optimal environment for data analysis with PyMVPA, but PyMVPA itself does not restrict the user's choice of the platform to the usual suspects. (A highres image is available, if you want to double check. ;-)

2.1.1 Must Have

The following software packages are required or PyMVPA will not work at all.

Python 2.4 with ctypes 1.0.1 or a later Python 2.X release
> With some modifications PyMVPA could probably work with Python 2.3, but as it is quite old already and Python 2.4 is widely available there should be no need to do this.

NumPy
> PyMVPA makes extensive use of NumPy to store and handle data. There is no way around it.

2.1.2 Strong Recommendations

While most parts of PyMVPA will work without any additional software, some functionality makes use (or can optionally make use) of external software packages. It is strongly recommended to install these packages as well, if they are available on a particular target platform.

SciPy: linear algebra, standard distributions, signal processing, data IO
> SciPy is mainly used by the statistical testing and the logistic regression classifier code. However, the SciPy package provides a lot of functionality that might be relevant in the context of PyMVPA, e.g. IO support for Matlab .mat files.

PyNIfTI (>= 0.20081017.1): access to NIfTI files
> PyMVPA provides a convenient wrapper for datasets stored in the NIfTI format, that internally uses PyNIfTI. If you don't need that, PyNIfTI is not necessary, but otherwise it makes it really easy to read from and write to NIfTI images. All dataset types dealing with NIfTI data will not be available without a functional PyNIfTI installation. Since PyMVPA 0.4.0 at least PyNIfTI version 0.20081017.1 (or later) is required.

2.1.3 Suggestions

The following list of software is again not required by PyMVPA, but these packages add additional functionality (e.g. classifiers implemented in external libraries) and might make life a lot easier by leading to more efficiency when using PyMVPA.

IPython: frontend
> If you want to use PyMVPA interactively it is strongly recommend to use IPython. If you think: *"Oh no, not another one, I already have to learn about PyMVPA."* please invest a tiny bit of time to watch the Five Minutes with IPython screencasts at showmedo.com, so at least you know what you are missing. In the context of cluster computing IPython is also the way to go.

FSL: preprocessing and analysis of (f)MRI data
> PyMVPA provides some simple bindings to FSL output and filetypes (e.g. EV files, estimated motion correct parameters and MELODIC output directories). This makes it fairly easy to e.g. use FSL's implementation of ICA for data reduction and proceed with analyzing the estimated ICs in PyMVPA.

AFNI: preprocessing and analysis of (f)MRI data
> Similar to FSL, AFNI is a free package for processing (f)MRI data. Though its primary data file format is BRIK files, it has the ability to read and write NIFTI files, which easily integrate with PyMVPA.

Shogun: various classifiers
> PyMVPA currently can make use of several SVM implementations of the Shogun toolbox. It requires the modular python interface of Shogun to be installed. Any version from 0.6 on should work.

LIBSVM: fast SVM classifier
> Only the C library is required and none of the Python bindings that are available on the upstream website. PyMVPA provides its own Python wrapper for LIBSVM which is a fork based on the one included in the LIBSVM package. Additionally the upstream LIBSVM distribution causes flooding of the console with a huge amount of debugging messages. Please see the Building from Source section for information on how to build an alternative version that does not have this problem. Since version 0.2.2, PyMVPA contains a minimal copy of LIBSVM in its source distribution.

R and RPy: more classifiers
> Currently PyMVPA provides a wrapper around the LARS library.

matplotlib: Matlab-style plotting library for Python
> This is a very powerful plotting library that allows you to export into a large variety of raster and vector formats (e.g. SVG), and thus, is ideal to produce publication quality figures. The examples shipped with PyMVPA show a number of possibilities how to use matplotlib for data visualization.

hcluster: generating, visualizing, and analyzing hierarchical clusters
> This module is a nice addition to SciPy and can be used to perform cluster analyses and plot dendrograms of their results.

2.2 Installing Binary Packages

The easiest way to obtain PyMVPA is to use pre-built binary packages. Currently we provide such packages or installers for the Debian/Ubuntu family, several RPM-based GNU/Linux distributions, MacOS X and 32-bit Windows (see below). If there are no binary packages for your operating system or platform yet, you can build PyMVPA from source. Please refer to Building from Source for more information.

2.2.1 Debian

PyMVPA is available as an official Debian package (*python-mvpa*; since *lenny*). The documentation is provided by the optional *python-mvpa-doc* package. To install PyMVPA simply do:

```
sudo aptitude install python-mvpa
```

2.2.2 Debian backports and inofficial Ubuntu packages

Backports for the current Debian stable release and binary packages for recent Ubuntu releases are available from a repository at the University of Magdeburg. Please read the package repository instructions to learn about how to obtain them. Otherwise install as you would do with any other Debian package.

2.2.3 Windows

There are a few Python distributions for Windows. In theory all of them should work equally well. However, we only tested the standard Python distribution from www.python.org (with version 2.5.2).

First you need to download and install Python. Use the Python installer for this job. Yo do not need to install the Python test suite and utility scripts. From now on we will assume that Python was installed in *C:\Python25* and that this directory has been added to the *PATH* environment variable.

For a minimal installation of PyMVPA the only thing you need in addition is NumPy. Download a matching NumPy windows installer for your Python version (in this case 2.5) from the SciPy download page and install it.

Now, you can use the PyMVPA windows installer to install PyMVPA on your system. If done, verify that everything went fine by opening a command promt and start Python by typing *python* and hit enter. Now you should see the Python prompt. Import the mvpa module, which should cause no error messages.

```
>>> import mvpa
>>>
```

Although you have a working installation already, most likely you want to install some additional software. First and foremost install SciPy – download from the same page where you also got the NumPy installer.

If you want to use PyMVPA to analyze fMRI datasets, you probably also want to install PyNIfTI. Download the corresponding installer from the website of the NIfTI libraries and install it. PyNIfTI does not come with the required *zlib* library, so you also need to download and install it. A binary installer is available from the GnuWin32 project. Install it in some arbitrary folder (just the binaries nothing else), find the *zlib1.dll* file in the *bin* subdirectory and move it in the Windows *system32* directory. Verify that it works by importing the *nifti* module in Python.

```
>>> import nifti
>>>
```

Another piece of software you might want to install is matplotlib. The project website offers a binary installer for Windows. If you are using the standard Python distribution and matplotlib complains about a missing *msvcp71.dll*, be sure to obey the installation instructions for Windows on the matplotlib website.

With this set of packages you should be able to run most of the PyMVPA examples which are shipped with the source code in the *doc/examples* directory.

2.2.4 MacOS X

Similar to the situation on Windows, there are multiple Python distributions available for the Mac. A convenient way to get PyMVPA running is to use the Enthought Python distribution, which is available free of charge for academic use. The main advange of this distribution is, that it already includes NumPy, SciPy, matplotlib, and IPython, hence only few additional software packages have to be installed manually. The procedure outlined here has been tested with MacOS X 10.5 (Leopard). It may, however, also work on 10.4 (Tiger).

First download Enthought Python and install it as usual. Now you can download the PyMVPA installer from the project's download area. It should be automatically decompressed after it has been downloaded (gzipped tar file). Simply click the dowloaded file to start the installer and follow the instructions.

Upon success, open a terminal window and start Python by typing *python* and hit return. Now try to import the PyMVPA module by doing:

```
>>> import mvpa
>>>
```

If no error messages appear, you have succesfully installed PyMVPA.

2.2.5 RPM-based GNU/Linux Distributions

To install one of the RPM packages provided through the OpenSUSE Build Service, first download it from the OpenSUSE software website.

Note: This site does not only offer OpenSUSE packages, but also binaries for other distributions, including: CentOS 5, Fedora 9, Mandriva 2007-2008, RedHat Enterprise Linux 5, SUSE Linux Enterprise 10, OpenSUSE 10.2 up to 11.0.

Once downloaded, open a console and invoke (the example command refers to PyMVPA 0.3.1):

```
rpm -i python-mvpa-0.3.1-19.1.i386.rpm
```

The OpenSUSE website also offers 1-click-installations for distributions supporting it.

A more convenient way to install PyMVPA and automatically receive software updates is to included one of the RPM-package repositories in the system's package management configuration. For e.g. OpenSUSE 11.0, simply use Yast to add another repository, using the following URL:

> http://download.opensuse.org/repositories/home:/hankem/openSUSE_11.0/

For other distributions use the respective package managers (e.g. Yum) to setup the repository URL. The repositories include all core dependencies of PyMVPA (usually Numpy and PyNIfTI), if they are not available from other repositories of the respective distribution.

2.3 Building from Source

If a binary package for your platform and operating system is provided, you do not have to build the packages on your own – use the corresponding pre-build packages instead. However, if there are no binary packages for your system, or you want to try a new (unreleased) version of PyMVPA, you can easily build PyMVPA on your own. Any recent linux distribution should be capable of doing it (e.g. RedHat). Additionally, building PyMVPA also works on Mac OSX and Windows systems.

2.3.1 Three Ways to Obtain the Sources

The first step is obtaining the sources. The source code tarballs of all PyMVPA releases are available from the PyMVPA project website. Alternatively, one can also download a tarball of the latest development snapshot (i.e. the current state of the *master* branch of the PyMVPA source code repository). If you want to have access to both, the full PyMVPA history and the latest development code, you can use the PyMVPA Git repository, which is publicly available. To view the repository, please point your web browser to gitweb:

> http://git.debian.org/?p=pkg-exppsy/pymvpa.git

The gitweb browser also allows to download arbitrary development snapshots of PyMVPA. For a full clone (aka checkout) of the PyMVPA repository simply do:

> **git clone git://git.debian.org/git/pkg-exppsy/pymvpa.git**

After a short while you will have a *pymvpa* directory below your current working directory, that contains the PyMVPA repository.

2.3.2 Build it (General instructions)

In general you can build PyMVPA like any other Python module (using the Python *distutils*). This general method will be outline first. However, in some situations or on some platforms alternative ways of building PyMVPA might be more convenient – alternative approaches are listed at the end of this section.

To build PyMVPA from source simply enter the root of the source tree (obtained by either extracting the source package or cloning the repository) and run:

> **python setup.py build_ext**

If you are using a Python version older than 2.5, you need to have python-ctypes (>= 1.0.1) installed to be able to do this.

Now, you are ready to install the package. Do this by invoking:

python setup.py install

Most likely you need superuser privileges for this step. If you want to install in a non-standard location, please take a look at the **–prefix** option. You also might want to consider **–optimize**.

Now you should be ready to use PyMVPA on your system.

2.3.3 Build with enabled LIBSVM bindings

From the 0.2 release of PyMVPA on, the LIBSVM classifier extension is not build by default anymore. However, it is still shipped with PyMVPA and can be enabled at build time. To be able to do this you need to have SWIG installed on your system.

PyMVPA needs a patched LIBSVM version, as the original distribution generates a huge amount of debugging messages and therefore makes the console and PyMVPA output almost unusable. Debian (since lenny: 2.84.0-1) and Ubuntu (since gutsy) already include the patched version. For all other systems a minimal copy of the patched sources is included in the PyMVPA source package (*3rd/libsvm*).

If you do not have a proper LIBSVM package, you can build the library from the copy of the code that is shipped with PyMVPA. To do this, simply invoke:

```
make 3rd
```

Now build PyMVPA as described above. The build script will automatically detect that LIBSVM is available and builds the LIBSVM wrapper module for you.

If your system provides an appropriate LIBSVM version, you need to have the development files (headers and library) installed. Depending on where you installed them, it might be necessary to specify the full path to that location with the *–include-dirs*, *–library-dirs* and *–swig* options. Now add the '–with-libsvm' flag when building PyMVPA:

```
python setup.py build_ext --with-libsvm \
    [ -I<LIBSVM_INCLUDEDIR> -L<LIBSVM_LIBDIR> ]
```

The installation procedure is equivalent to the build setup without LIBSVM, except that the '–with–libsvm' flag also has to be set when installing:

```
python setup.py install --with-libsvm
```

2.3.4 Alternative build procedure

Alternatively, if you are doing development in PyMVPA or if you simply do not want (or do not have sufficient permissions to do so) to install PyMVPA system wide, you can simply call *make* (same *make build*) in the top-level directory of the source tree to build PyMVPA. Then extend or define your environment variable *PYTHONPATH* to point to the root of PyMVPA sources (i.e. where you invoked all previous commands from):

export PYTHONPATH=$PWD

Note: This procedure also always builds the LIBSVM extension and therefore also requires the patched LIBSVM version and SWIG to be available.

2.3.5 Windows

On Windows the whole situation is a little more tricky, as the system doesn't come with a compiler by default. Nevertheless, it is easily possible to build PyMVPA from source. One could use the Microsoft compiler that comes with Visual Studio to do it, but as this is commercial software and not everybody has access to it, we will outline a way that exclusively involves free and open source software.

First one needs to install the packages required to run PyMVPA as explained *above*.

Next we need to obtain and install the MinGW compiler collection. Download the *Automated MinGW Installer* from the MinGW project website. Now, run it and choose to install the *current* package. You will need the *MinGW base tools*, *g++* compiler and *MinGW Make*. For the remaining parts of the section, we will assume that MinGW got installed in *C:\MinGW* and the directory *C:\MinGW\bin* has been added to the *PATH* environment variable, to be able to easily access all MinGW tools.

Note: It is not necessary to install MSYS to build PyMVPA, but it might handy to have it.

If you want to build the LIBSVM wrapper for PyMVPA, you also need to download SWIG (actually *swigwin*, the distribution for Windows). SWIG does not have to be installed, just unzip the file you downloaded and add the root directory of the extracted sources to the *PATH* environment variable (make sure that this directory contains *swig.exe*, if not, you haven't downloaded *swigwin*).

PyMVPA comes with a specific build setup configuration for Windows – *setup.cfg.win* in the root of the source tarball. Please rename this file to *setup.cfg*. This is only necessary, if you have *not* configured your Python distutils installation to always use MinGW instead of the Mircrosoft compilers.

Now, we are ready to build PyMVPA. The easiest way to do this, is to make use of the *Makefile.win* that is shipped with PyMVPA to build a binary installer package (*.exe*). Make sure, that the settings at the top of *Makefile.win* (the file is located in the root directory of the source distribution) correspond to your Python installation – if not, first adjust them accordingly before your proceed. When everything is set, do:

```
mingw32-make -f Makefile.win installer
```

Upon success you can find the installer in the *dist* subdirectory. Install it as described *above*.

2.3.6 OpenSUSE

Building PyMVPA on OpenSUSE involves the following steps (tested with 10.3): First add the OpenSUSE science repository, that contains most of the required packages (e.g. NumPy, SciPy, matplotlib), to the Yast configuration. The URL for OpenSUSE 10.3 is:

```
http://download.opensuse.org/repositories/science/openSUSE_10.3/
```

Now, install the following required packages:

- a recent C and C++ compiler (e.g. GCC 4.1)
- *python-devel* (Python development package)
- *python-numpy* (NumPy)
- *swig* (SWIG is only necessary, if you want to make use of LIBSVM)

Now you can simply compile and install PyMVPA, as outlined above, in the general build instructions (or alternatively using the method with LIBSVM).

If you have problems compiling the NIfTI libraries and PyNIfTI on OpenSUSE, try the following: Download the *nifticlib* source tarball, extract it and run *make* in the top-level source directory. Be sure to install the *zlib-devel* package before. Now, download the *pynifti* source tarball extract it, and edit *setup.py*. Change the line:

```
libraries = [ 'niftiio' ],
```

to:

```
libraries = [ 'niftiio', 'znz', 'z' ],
```

as mentioned in the PyNIfTI installation instructions. This is necessary, as the above approach does only generate static NIfTI libraries which are not properly linked with all dependencies. Now, compile PyNIfTI with:

```
python setup.py build_ext -I <path_to_nifti>/include \
    -L <path_to_nifti>/lib --swig-opts="-I<path_to_nifti>/include"
```

where *<path_to_nifti>* is the directory that contains the extracted *nifticlibs* sources. Finally, install PyNIfTI with:

```
sudo python setup.py install
```

If you want to run the PyMVPA examples including the ones that make use of the plotting capabilities of *matplotlib* you need to install of few more packages (mostly due to broken dependencies in the corresponding OpenSUSE packages):

- *python-scipy*

- *python-gobject2*

- *python-gtk*

2.3.7 Fedora

On Fedora (tested with Fedora 9) you first have to install a few required packages, that are not installed by default. Simply do:

```
yum install numpy gcc gcc-c++ python-devel swig
```

You might also want to consider installing some more packages, that will make your life significantly easier:

```
yum install scipy ipython python-matplotlib
```

Now, you are ready to compile and install PyMVPA as describe in the *general build instructions*.

2.3.8 MacOS X

To be able to compile PyMVPA on a Mac you first have to install the XCode developer tools, as the operating system does not come with a compiler by default. If you want to use or even work on the latest development code, you should also install Git. There is a MacOS installer for Git, that make this step very easy.

Otherwise follow the *general build instructions*.

GETTING STARTED

3.1 For the Impatient

If you only have five minutes to decide whether you want to use PyMVPA, take the first minute to look at the following example of a cross-validation procedure on an fMRI dataset (the full source code!). It is not heavily commented, but should simply give you an idea how PyMVPA feels like.

First import the whole PyMVPA module:

```
>>> from mvpa.suite import *
```

Now, load the dataset from a NIfTI file. An additional 2-column textfile has the label and associated experimental run of each volume in the dataset (one volume per line). Finally, a mask is loaded to exclude non-brain voxels.

```
>>> attr = SampleAttributes('data/attributes.txt')
>>> dataset = NiftiDataset(samples='data/bold.nii.gz',
...                        labels=attr.labels,
...                        chunks=attr.chunks,
...                        mask='data/mask.nii.gz')
```

Perform linear detrending and afterwards zscore the timeseries of each voxel using the mean and standard deviation determined from *rest* volumes (all done for each experimental run individually).

```
>>> detrend(dataset, perchunk=True, model='linear')
>>> zscore(dataset, perchunk=True, baselinelabels=[0],
...        targetdtype='float32')
```

Select a subset of two stimulation conditions from the whole dataset.

```
>>> dataset = dataset.selectSamples(
...              N.array([l in [1, 2] for l in dataset.labels],
...                      dtype='bool'))
```

Finally, setup the cross-validation procedure using an odd-even split of the dataset and a *SMLR* classifier – and run it.

```
>>> cv = CrossValidatedTransferError(
...          TransferError(SMLR()),
...          OddEvenSplitter())
>>> error = cv(dataset)
```

Done. The mean error of classifier predictions on the test dataset across dataset splits is stored in *error*.

If you think that is a good start, take the remaining four minutes to take a look at the examples shipped in the source distribution of PyMVPA (*doc/examples/*; some of them are also listed in *Full Examples* section of this manual).

The examples provide a coarse overview of a substantial portion of the functionality provided by PyMVPA, ranging from basic classifier usage, over more sophisticated analysis strategies to simple visualization demos.

All examples are executable scripts that are meant to be run from to toplevel directory of the extracted source tarball, e.g.:

```
$ doc/examples/start_easy.py
```

which would run the example shown in the first part of this section.

However, once you found something interesting in the examples you should consider skipping through this manual, as it contains a lot of information that is complementary to the API reference and the examples.

And now for the details ...

3.2 Module Overview

The PyMVPA package consists of three major parts: *Data handling*, *Classifiers* and various algorithms and measures that operate on datasets and classifiers. In the following sections the basic concept of all three parts will be described and examples using certain parts of the PyMVPA package will be given.
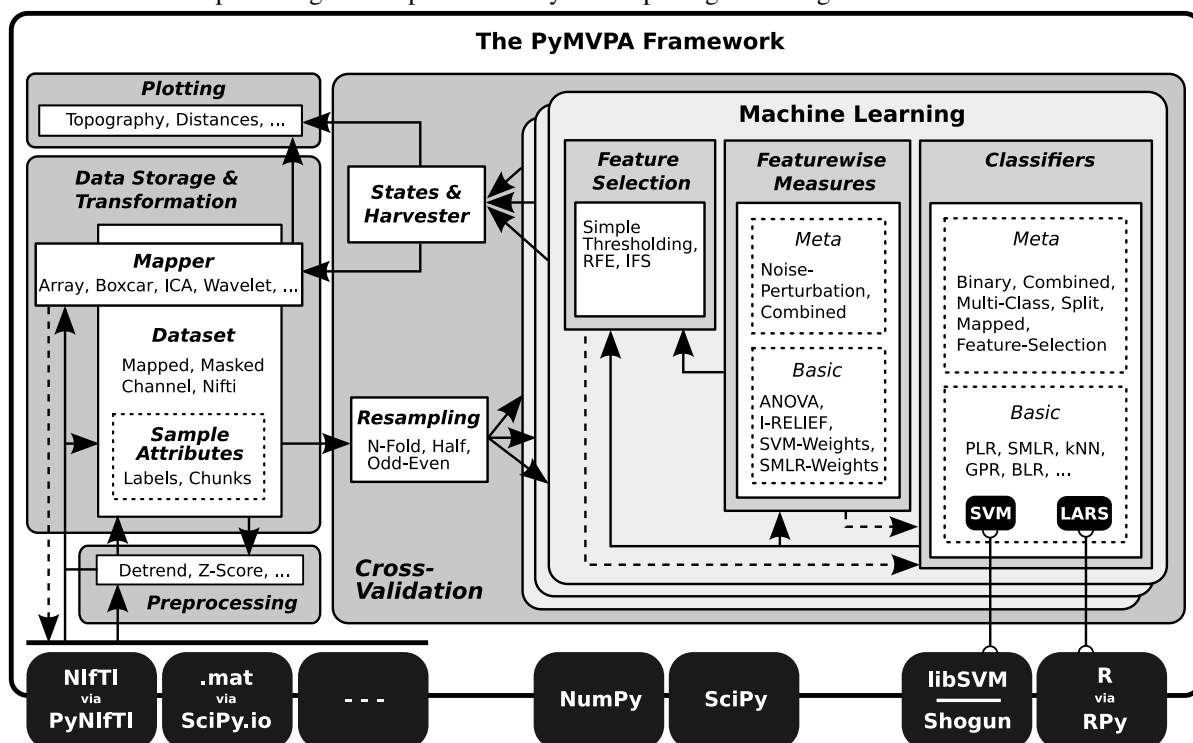
The manual does not cover all bits and pieces of PyMVPA. Detailed information about the module layout and additional documentation about all included functionality is available from the *Module Reference* – or the API Reference if you are interested in a more technical document. The main purpose of the manual is to give an idea how the individual parts of PyMVPA can be combined to perform complex analyses – easily.

DATASETS

The first step of any analysis in PyMVPA involves reading the data and putting it into the necessary shape for the intended analysis. But even after the initial setup, many algorithms have to modify datasets, e.g. by selecting a subset of it, or simple transformations of the data (e.g. z-scoring), or more complex things like projections into alternative representations/spaces.

This section introduces the basic concepts of a dataset in PyMVPA and shows useful operations typically performed on datasets.

4.1 The Basic Concepts

A minimal dataset in PyMVPA consists of a number of *sample*s, where each individual sample is nothing more than a vector of values. Each sample is associated with a *label*, which defines the category the respective sample belongs to, or in more general terms, defines the model that should be learned by a classifier. Moreover, samples can be grouped into so-called *chunk*s, where each chunk is assumed to be statistically independent from all other data chunks.

The foundation of PyMVPA's data handling is the `Dataset` class. Basically, this class stores data samples, sample attributes and dataset attributes. By definition, sample attributes assign a value to each data sample (e.g. labels, or chunks) and dataset attributes are additional information or functionality that apply to the whole dataset.

Most likely the `Dataset` class will not be used directly, but through one of the derived classes. However, it is perfectly possible to use it directly. In the simplest case a dataset can be constructed by specifying some data samples and the corresponding class labels.

```
>>> import numpy as N
>>> from mvpa.datasets import Dataset
>>> data = Dataset(samples=N.random.normal(size=(10,5)), labels=1)
>>> data
<Dataset / float64 10 x 5 uniq: 1 labels 10 chunks>
```

The above example creates a dataset with 10 samples and 5 features each. The values of all features stem from normally distributed random noise. The class label '1' is assigned to all samples. Instead of a single scalar value *labels* can also be a sequence with individual labels for each data sample. In this case the length of this sequence has to match the number of samples.

Interestingly, the dataset object tells us about 10 *chunks*. In PyMVPA chunks are used to group subsets of data samples. However, if no grouping information is provided all data samples are assumed to be in their own group, hence no sample grouping is performed.

Both *labels* and *chunks* are so called *sample attributes*. All sample attributes are stored in sequence-type containers consisting of one value per sample. These containers can be accessed by properties with the same as the attribute:

```
>>> data.labels
array([1, 1, 1, 1, 1, 1, 1, 1, 1, 1])
>>> data.chunks
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

The *data samples* themselves are stored as a two-dimensional matrix where each row vector is a *sample* and each column vector contains the values of a *feature* across all *samples*. The `Dataset` class provides access to the samples matrix via the *samples* property.

```
>>> data.samples.shape
(10, 5)
```

The `Dataset` class itself can only deal with 2d sample matrices. However, PyMVPA provides a very easy way to deal with data where each data sample is more than a 1d vector: Data Mapping

4.2 Data Mapping

It was already mentioned that the `Dataset` class cannot deal with data samples that are more than simple vectors. This could be a problem in cases where the data has a higher dimensionality, e.g. functional brain-imaging data where each data sample is typically a three-dimensional volume.

One approach to deal with this situation would be to concatenate the whole volume into a 1d vector. While this would work in certain cases there is definitely information lost. Especially for brain-imaging data one would most likely want keep information about neighbourhood and distances between data sample elements.

In PyMVPA this is done by mappers that transform data samples from their original *dataspace* into the so-called *features space*. In the above neuro-imaging example the *dataspace* is three-dimensional and the *feature space* always refers to the 2d *samples x features* representation that is required by the `Dataset` class. In the context of mappers the dataspace is sometimes also referred to as *in-space* (i.e. the initial data that goes into the mapper) while the feature space is labeled as *out-space* (i.e. the mapper output when doing forward mapping).

The task of a mapper, besides transforming samples into 1d vectors, is to retain as much information of the dataspace as possible. Some mappers provide information about dataspace metrics and feature neighbourhood, but all mappers are able to do reverse mapping from feature space into the original dataspace.

Usually one does not have to deal with mappers directly. PyMVPA provides some convenience subclasses of `Dataset` that automatically perform the necessary mapping operations internally. For an introduction into to concept of a dataset with mapping capabilities we can take a look at the `MaskedDataset` class. This dataset class works almost exactly like the basic `Dataset` class, except that it provides some additional methods and is more flexible with respect to the format of the sample data. A masked dataset can be created just like a normal dataset.

```
>>> from mvpa.datasets.masked import MaskedDataset
>>> mdata = MaskedDataset(samples=N.random.normal(size=(5,3,4)),
...                       labels=[1,2,3,4,5])
>>> mdata
<Dataset / float64 5 x 12 uniq: 5 chunks 5 labels>
```

However, unlike `Dataset` the `MaskedDataset` class can deal with sample data arrays with more than two dimensions. More precisely it handles arrays of any dimensionality. The only assumption that is made is that the first axis of a sample array separates the sample data points. In the above example we therefore have 5 samples, where each sample is a 3x4 plane. If we look at the self-description of the created dataset we can see that it doesn't tell us about 3x4 plane, but simply 12 features. That is because internally the sample array is automatically reshaped into the aforementioned 2d matrix representation of the `Dataset` class. However, the information about the original dataspace is not lost, but kept inside the mapper used by `MaskedDataset`. Two useful methods of `MaskedDataset` make use of the mapper: *mapForward()* and *mapReverse()*. The former can be used to transform additional data from dataspace into the feature space and the latter performs the same in the opposite direction.

```
>>> mdata.mapForward(N.arange(12).reshape(3,4)).shape
(12,)
>>> mdata.mapReverse(N.array([1]*mdata.nfeatures)).shape
(3, 4)
```

Especially reverse mapping can be very useful when visualizing classification results and information maps on the original dataspace.

Another feature of mapped datasets is that valid mapping information is maintained even when the feature space changes. When running some feature selection algorithm (see *Feature Selection*) some features of the original features set will be removed, but after feature selection one will most likely want to know where the selected (or removed) features are in the original dataspace. To make use of the neuro-imaging example again: The most convenient way to access this kind of information would be a map of the selected features that can be overlayed over some anatomical image. This is trivial with PyMVPA, because the mapping is automatically updated upon feature selection.

```
>>> mdata.mapReverse(N.arange(1,mdata.nfeatures+1))
array([[ 1,  2,  3,  4],
       [ 5,  6,  7,  8],
       [ 9, 10, 11, 12]])
>>> sdata = mdata.selectFeatures([2,7,9,10])
>>> sdata
<Dataset / float64 5 x 4 uniq: 5 chunks 5 labels>
>>> sdata.mapReverse(N.arange(1,sdata.nfeatures+1))
array([[0, 0, 1, 0],
       [0, 0, 0, 2],
       [0, 3, 4, 0]])
```

The above example selects four features from the set of the 12 original ones, by passing their ids to the *selectFeatures()* method. The method returns a new dataset only containing the four selected features. Both datasets share the sample data (using a NumPy array view). Using *selectFeatures()* is therefore both memory efficient and relatively fast. All other information like class labels and chunks are maintained. By calling *mapReverse()* on the new dataset one can see that the remaining four features are precisely mapped back onto their original locations in the data space.

4.3 Data Access Sugaring

Complementary to self-descriptive attribute names (e.g. *labels*, *samples*) datasets have a few concise shortcuts to get quick access to some attributes or perform some common action

Attribute	Abbreviation	Definition class
samples	S	`Dataset`
labels	L	`Dataset`
uniquelabels	UL	`Dataset`
chunks	O	`Dataset`
uniquechunks	UC	`Dataset`
origids	I	`Dataset`
samples_original	O	`MappedDataset`

4.4 Data Formats

The concept of mappers in conjuction with the functionality provided by the `Dataset` class, makes it very easy to create new dataset types with support for specialized data types and formats. The following is a non-exhaustive list of data formats currently supported by PyMVPA (for additional formats take a look at the subclasses of `Dataset`):

- NumPy arrays

  PyMVPA builds its dataset facilities on NumPy arrays. Basically, anything that can be converted into a NumPy array can also be converted into a dataset. Together with the corresponding labels, NumPy arrays can simply be passed to the `Dataset` constructor to create a dataset. With arrays it is possible to use the classes `Dataset`, `MappedDataset` (to combine the samples with any custom mapping algorithm) or `MaskedDataset` (readily provides a `DenseArrayMapper`).

- Plain text

  Using the NumPy function *fromfile()* a variety of text file formats (e.g. CSV) can be read and converted into NumPy arrays.

- NIfTI/Analyze images

  PyMVPA provides a specialized dataset for MRI data in the NIfTI format. `NiftiDataset` uses PyNIfTI to read the data and automatically configures an appropriate `DenseArrayMapper` with metric information read from the NIfTI file header.

- EEP binary files

  Another special dataset type is `EEPDataset`. It reads data from binary EEP file (written by eeprobe)

4.5 Data Splitting

In many cases some algorithm should not run on a complete dataset, but just some parts of it. One well-known example is leave-one-out cross-validation, where a dataset is typically split into a number of training and validation datasets. A classifier is trained on the training set and its generalization performance is tested using the validation set.

It is important to strictly separate training and validation datasets as otherwise no valid statement can be made whether a classifier really generated an appropriate model of the training data. Violating this requirement spuriously elevates the classification performance, often termed 'peeking' in the literature. However, they provide no relevant information because they are based on cheating or peeking and do not describe signal similarities between training and validation datasets.

With the splitter classes derived from the base `Splitter`, PyMVPA makes dataset splitting easy. All dataset splitters in PyMVPA are implemented as Python generators, meaning that when called with a dataset once, they return one dataset split per iteration and an appropriate Exception when they are done. This is exactly the same behavior as of e.g. the Python *xrange()* function. To perform data splitting for the already mentioned cross-validation, PyMVPA provides the `NFoldSplitter` class. It implements a method to generate arbitrary N-M splits, where N is the number of different chunks in a dataset and M is any non-negative integer smaller than N. Doing a leave-one-out split of our example dataset looks like this:

```
>>> from mvpa.datasets.splitter import NFoldSplitter
>>> splitter = NFoldSplitter(cvtype=1)   # Do N-1
>>> for wdata, vdata in splitter(data):
...     pass
```

where *wdata* is the *working dataset* and *vdata* is the *validation dataset*. If we have a look a those datasets we can see that the splitter did what we intended:

```
>>> split = [ i for i in splitter(data)][0]
>>> for s in split:
...     print s
Dataset / float64 9 x 5 uniq: 1 labels 9 chunks
Dataset / float64 1 x 5 uniq: 1 labels 1 chunks
>>> split[0].uniquechunks
array([1, 2, 3, 4, 5, 6, 7, 8, 9])
>>> split[1].uniquechunks
array([0])
```

In the first split, the working dataset contains nine chunks of the original dataset and the validation set contains the remaining chunk. The usage of the splitter, creating a splitter object and calling it with a dataset, is a very common design pattern in the PyMVPA package. Like splitters, there are many more so called *processing objects*. These classes or objects are instantiated by passing all relevant parameters to the constructor. Processing objects can then be called multiple times with different datasets to perform their algorithm on the respective dataset. This design applies to the majority of the algorithms implemented in PyMVPA.

CLASSIFIERS

PyMVPA includes a number of ready-to-use classifiers, which are described in the following sections. All classifiers implement the same, very simple interface. Each classifier object takes all relevant parameters as arguments to its constructor. Once instantiated, the classifier object's `train()` method can be called with some dataset. This trains the classifier using *all* samples in the respective dataset.

The major task for a classifier is to make predictions. Predictions are made by calling the classifier's `predict()` method with one or multiple data samples. `predict()` operates on pure sample data and not datasets, as in some cases the true label for a sample might be totally unknown.

This examples demonstrates the typical daily life of a classifier.

```
>>> import numpy as N
>>> from mvpa.clfs.knn import kNN
>>> from mvpa.datasets import Dataset
>>> training = Dataset(samples=N.array(
...                             N.arange(100),ndmin=2, dtype='float').T,
...                 labels=[0] * 50 + [1] * 50)
>>> rand100 = N.random.rand(10)*100
>>> validation = Dataset(samples=N.array(rand100, ndmin=2, dtype='float').T,
...                 labels=[ int(i>50) for i in rand100 ])
>>> clf = kNN(k=10)
>>> clf.train(training)
>>> N.mean(clf.predict(training.samples) == training.labels)
1.0
>>> N.mean(clf.predict(validation.samples) == validation.labels)
1.0
```

Two datasets with 100 and 10 samples each are generated. Both datasets only have one feature and the associated label is 0 if the feature value is below 50 or 1 otherwise. The larger dataset contains all integers in the interval (0,100) and is used to train the classifier. The smaller is used as a validation dataset, to check whether the classifier learned something that generalizes well across samples not included in the training dataset. In this case the validation dataset consists of 10 random floating point values in the interval (0,100).

The classifier in this example is a kNN (k-Nearest-Neighbour) classifier that makes use of the 10 nearest neighbours of a data sample to make its predictions (k=10). One can see that after the training the classifier performs optimally on the training dataset as well as on the validation data samples.

The choice of the classifier in the above example is more or less arbitrary. Any classifier in PyMVPA could be used in place of kNN. This demonstrates another useful feature of PyMVPA's classifiers. Due to the high-level abstraction and the simple interface, almost all classifiers can be combined with most algorithms in PyMVPA. This makes it very easy to test different classifiers on some dataset (see Fig. 1).
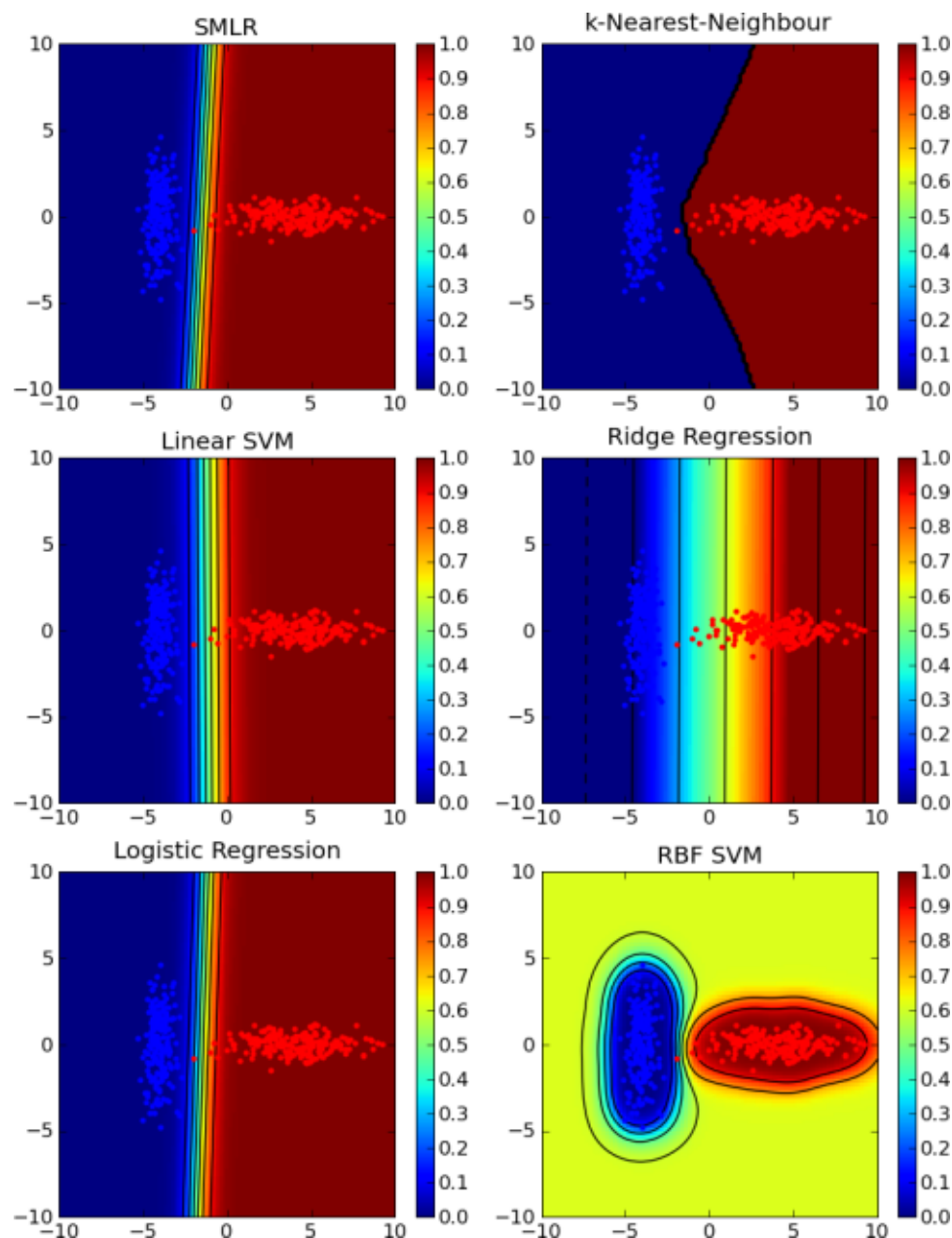
A comparison of the behavior of different classifiers (k-Nearest-Neighbour, linear SVM, logistic regression, ridge regression and SVM with radial basis function kernel) on a simple classification problem. The code to generate these figure can be found in the *pylab_2d.py* example in the *Simple Plotting of Classifier Behavior* section.

5.1 Stateful objects

Before looking at the different classifiers in more detail, it is important to mention another feature common to all of them. While their interface is simple, classifiers are in no way limited to report only predictions. All classifiers implement an additional interface: Objects of any class that are derived from `ClassWithCollections` have attributes (we refer to such attributes as state variables), which are conditionally computed and stored by PyMVPA. Such conditional storage and access is handy if a variable of interest might consume a lot of memory or needs intensive computation, and not needed in most (or in some) of the use cases.

For instance, the `Classifier` class defines the *trained_labels* state variable, which just stores the unique labels for which the classifier was trained. Since *trained_labels* stores meaningful information only for a trained classifier, attempt to access 'clf.trained_labels' before training would result in an error,

```
>>> from mvpa.misc.exceptions import UnknownStateError
>>> try:
...     untrained_clf = kNN()
...     labels = untrained_clf.trained_labels
... except UnknownStateError:
...     "Does not work"
'Does not work'
```

since the classifier has not seen the data yet and, thus, does not know the labels. In other words, it is not yet in the state to know anything about the labels. Any state variable can be enabled or disabled on per instance basis at any time of the execution (see `ClassWithCollections`).

To continue the last example, each classifier, or more precisely every stateful object, can be asked to report existing state-related attributes:

```
>>> list_with_verbose_explanations = clf.states.listing
```

'clf.states' is an instance of `StateCollection` class which is a container for all state variables of the given class. Although values can be queried or set (if state is enabled) operating directly on the stateful object

```
>>> clf.trained_labels
array([0, 1])
```

any other operation on the state (e.g. enabling, disabling) has to be carried out through the *states* attribute.

```
>>> print clf.states
states{trained_dataset predicting_time*+ training_confusion predictions*+...}
>>> clf.states.enable('values')
>>> print clf.states
states{trained_dataset predicting_time*+ training_confusion predictions*+...}
>>> clf.states.disable('values')
```

A string representation of the state collection mentioned above lists all state variables present accompanied with 2 markers: '+' for an enabled state variable, and '*' for a variable that stores some value (but might have been disabled already and, therefore, would have no '+' and attempts to reassign it would result in no action).

By default all classifiers provide state variables *values*, *predictions*. The latter is simply the set of predictions that was returned by the last call to the objects `predict()` method. The former is heavily classifier-specific. By convention the *values* key provides access to the raw values that a classifier prediction is based on. Depending on the classifier, this information might required significant resources when stored. Therefore all states can be disabled or enabled (*states.disable()*, *states.enable()*) and their current status can be queried like this:

```
>>> clf.states.isActive('predictions')
True
>>> clf.states.isActive('values')
False
```

States can be enabled or disabled during stateful object construction, if *enable_states* or *disable_states* (or both) arguments, which store the list of desired state variables names, passed to the object constructor. Keyword 'all' can be used to select all known states for that stateful object.

5.2 Error Calculation

The `TransferError` class provides a convenient way to determine the transfer error of a trained classifier on some validation dataset, i.e. the accuracy of the classifier's predictions on a novel, independent dataset. A `TransferError` object is instanciated by passing a classifier object to the constructor. Optionally a custom error function can be specified (see *errorfx* argument).

To compute the transfer error simply call the object with a validation dataset. The computed error value is returned. `TransferError` also supports a state variable *confusion* that contains the full confusion matrix of the predictions made on the validation dataset. The confusion matrix is disabled by default.

If the `TransferError` object is called with an optional training dataset, the contained classifier is first training using this dataset before predictions on the validation dataset are made.

```
>>> from mvpa.clfs.transerror import TransferError
>>> clf = kNN(k=10)
>>> terr = TransferError(clf)
>>> terr(validation, training )
0.0
```

5.2.1 Cross-validated Transfer Error

Often one is not only interested in a single transfer error on one validation or test dataset, but on a cross-validated estimate of the transfer error. A popular method is the so-called leave-one-out cross-validation.

The `CrossValidatedTransferError` class provides a simple way to compute such measure. It utilizes a `TransferError` object and a `Splitter`. When called with a `Dataset` the splitter generates splits of the Dataset and the transfer error for all splits is computed by training on one of the splitted datasets and making predictions on the other. By default the mean of transfer errors is returned (but the actual *combiner* function is customizable).

The following example shows the minimal code for a leave-one-out cross-validation reusing the transfer error object from the previous example and some `Dataset` *data*.

```
>>> # create some dataset
>>> from mvpa.misc.data_generators import normalFeatureDataset
>>> data = normalFeatureDataset(perlabel=50, nlabels=2,
...                             nfeatures=20, nonbogus_features=[3, 7],
...                             snr=3.0)
>>> # now cross-validation
>>> from mvpa.algorithms.cvtranserror import CrossValidatedTransferError
>>> from mvpa.datasets.splitter import NFoldSplitter
>>> cvterr = CrossValidatedTransferError(terr,
...                                       NFoldSplitter(cvtype=1),
...                                       enable_states=['confusion'])
>>> error = cvterr(data)
```

5.3 Error Reporting

PyMVPA is equipped with easy ways to have a glance overview over the generalization performance of a cross-validated classifier. Such summary is provided by instances of a `ConfusionMatrix` class, and is accompanied by various performance metrics. For example, the 8-fold cross-validation of the dataset with 8 labels with the SMLR classifier produced the following confusion matrix:

```
>>> # Simple 'print cvterr.confusion' provides the same output
>>> # without the description of abbreviations
>>> print cvterr.confusion.asstring(description=True) \
... # doctest: +SKIP
--------.       3kHz  7kHz  12kHz 20kHz 30kHz song1 song2 song3 song4 song5
predict.\targets 38   39    40    41    42    43    44    45    46    47
        `------ ---- ----- ----- ----- ----- ----- ----- ----- ----- -----  P'   N'   FP  FN   PPV
    3kHz / 38 84     42    27    4     4     2     1     0     15    19   198 1351 114 90   0.42
    7kHz / 39 43     94    16    0     1     1     1     2     1     24   183 1331 89  80   0.51
    12kHz / 40 21    16    103   5     2     2     0     0     6     13   168 1312 65  70   0.61
    20kHz / 41 1     2     13    158   1     0     0     1     3     1    180 1202 22  15   0.88
```

```
        30kHz / 42  3     0     2     3    162    0     0     0     0     170 1194  8  11   0.95
        song1 / 43  3     1     1     0     1    160    0     0     2     5     173 1199 13  14   0.92
        song2 / 44  1     1     0     0     0     0    171    0     0     0     173 1176  2   2   0.99
        song3 / 45  1     1     1     0     0     0     0    170    2     0     175 1179  5   4   0.97
        song4 / 46  7     3     3     2     2     2     0     0    139    7     165 1240 26  34   0.84
        song5 / 47 10    14     7     1     0     7     0     1     5    104    149 1310 45  69   0.7
Per target:       ---- ----- ----- ----- ----- ----- ----- ----- ----- -----
        P          174   174   173   173   173   174   173   174   173   173
        N          1560  1560  1561  1561  1561  1560  1561  1560  1561  1561
        TP          84    94   103   158   162   160   171   170   139   104
        TN         1261  1251  1242  1187  1183  1185  1174  1175  1206  1241
Summary\Means:    ---- ----- ----- ----- ----- ----- ----- ----- ----- ----- 173 1249 38  39   0.78
      ACC          0.78
      ACC%        77.57
   # of sets         8
<BLANKLINE>
Statistics computed in 1-vs-rest fashion per each target.
Abbreviations (for details see http://en.wikipedia.org/wiki/ROC_curve):
 TP : true positive (AKA hit)
 TN : true negative (AKA correct rejection)
 FP : false positive (AKA false alarm, Type I error)
 FN : false negative (AKA miss, Type II error)
 TPR: true positive rate (AKA hit rate, recall, sensitivity)
     TPR = TP / P = TP / (TP + FN)
 FPR: false positive rate (AKA false alarm rate, fall-out)
     FPR = FP / N = FP / (FP + TN)
 ACC: accuracy
     ACC = (TP + TN) / (P + N)
 SPC: specificity
     SPC = TN / (FP + TN) = 1 - FPR
 PPV: positive predictive value (AKA precision)
     PPV = TP / (TP + FP)
 NPV: negative predictive value
     NPV = TN / (TN + FN)
 FDR: false discovery rate
     FDR = FP / (FP + TP)
 MCC: Matthews Correlation Coefficient
     MCC = (TP*TN - FP*FN)/sqrt(P N P' N')
 # of sets: number of target/prediction sets which were provided
```

In addition to the abusively informative textual representation, there is an alternative graphical representation of the confusion matrix via the `plot()` method of a `ConfusionMatrix`:

```
>>> import pylab as P
>>> cvterr.confusion.plot() \
... # doctest: +SKIP
>>> P.show() \
... # doctest: +SKIP
```
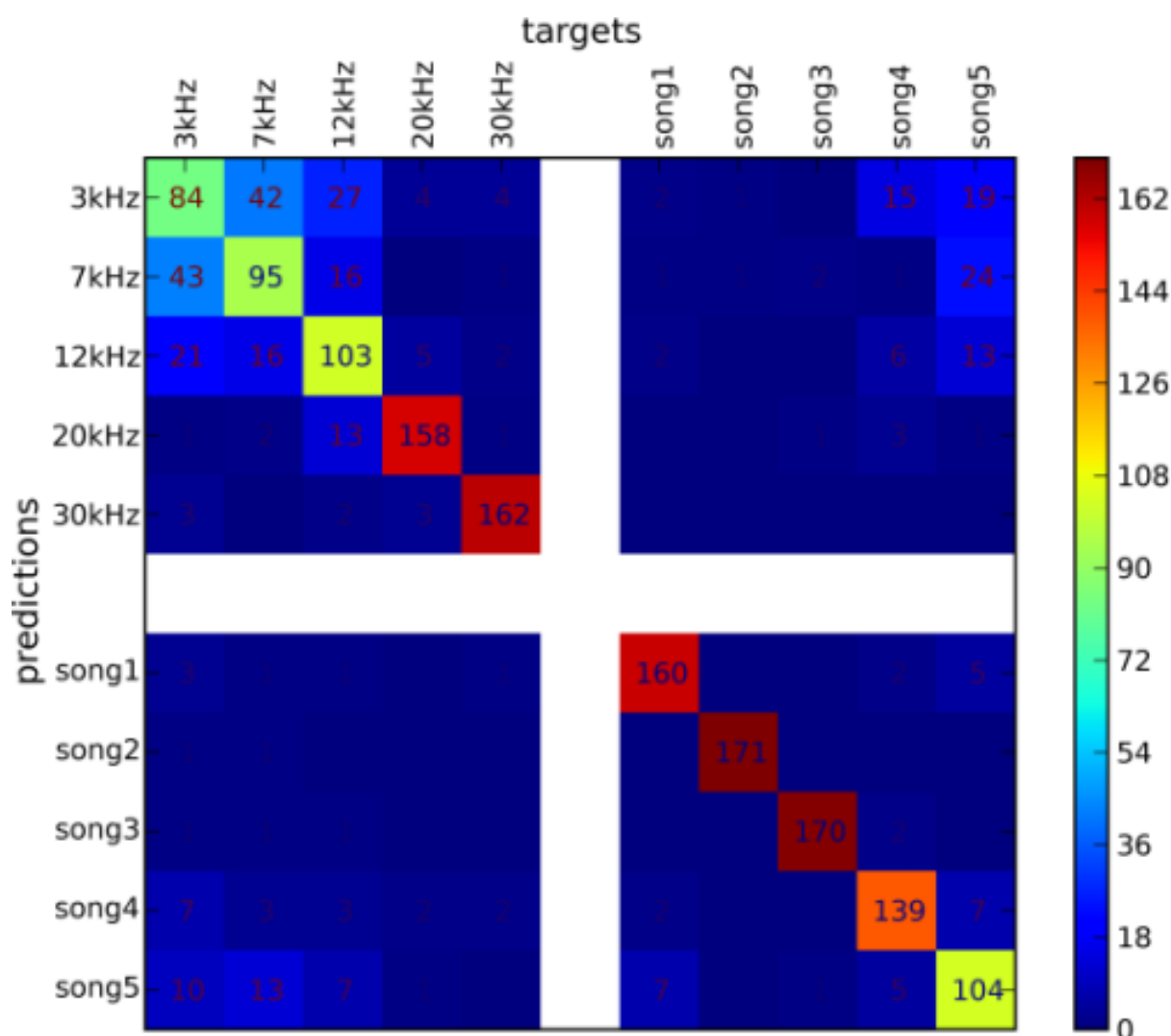
5.4 Basic Supervised Learning Methods

PyMVPA provides a number of learning methods (i.e. classifiers or regression algorithms) that can be plug into the various algorithms that are also part of the framework. Most importantly they all can be combined or enhanced with *Meta-Classifiers*.

5.4.1 Gaussian Process Regression

`GPR` (Wikipedia entry about gaussian process regression).

5.4.2 k-Nearest-Neighbour

The `kNN` classifier makes predictions based on the labels of nearby samples. It currently uses Euclidian distance to determine the nearest neighbours, but future enhancements may include support for other kernels.

5.4.3 Least Angle Regression

`LARS` *Efron et al. (2004)*

5.4.4 Penalized Logistic Regression

The penalized logistic regression (`PLR`) is similar to the ridge in that it has a penalty term, however, it is trained to predict a binary outcome by means of the logistic function (Wikipedia entry about logistic regression).

5.4.5 Ridge Regression

Ridge regression (aka Tikhonov regularization) is a variant of a linear regression (Wikipedia entry about ridge regression).

The ridge regression classifier (`RidgeReg`) performs a simple linear regression with a penalty parameter to help avoid over-fitting. The regression inserts an intercept term so that you do not have to center your data.

5.4.6 Sparse Multinomial Logistic Regression

Sparse Multinomial Logistic Regression (`SMLR`; *Krishnapuram et al., 2005*) is a fast multi-class classifier that can easily deal with high-dimensional problems (research paper about SMLR). PyMVPA includes two implementations: one in pure Python and a faster one that makes use of a C extension for the performance critical pieces of the code.

5.4.7 Support Vector Machines

Support vector machine (*Vapnik, 1995*) classifiers (and regressions) are popular since they can deal with very high dimensional problems (Wikipedia entry about SVM), while maintaining reasonable generalization performance.

The support vector machine classes provide a family of classifiers by wrapping libsvm and Shogun libraries, with corresponding base classes `SVM` and `SVM` accordingly. By default SVM class is bound to libsvm's implementation if such is available (shogun otherwise).

While any SVM class provides a complete interface, the others child classes make it easy to run some subset of standard classifiers, such as linear SVM, with a default set of parameters (see `LinearCSVMC`, `LinearNuSVMC`, `RbfNuSVMC` and `RbfCSVMC`).

5.5 Meta-Classifiers

This section has been contributed by James M. Hughes.

A meta-classifier is essentially a blanket term used to describe all classes that appear functionally equivalent to a regular `Classifier`, but which in reality provide some extra amount of functionality on top of a normal classifier. Furthermore, they generally do not implement a `Classifier` *per se*, but rather take a `Classifier` as input. The methods then typically called on a classifier (e.g., *train* or *predict*) can be called on the meta-classifier, but will call the input classifier's routines, before or after some other function that the meta-classifier provides.

5.5.1 Examples of Meta-Classifiers

At present, there are two primary meta-classifiers implemented in the PyMVPA package, beneath which there are several specific options:

`BoostedClassifier`
    typically uses multiple classifiers internally

`ProxyClassifier`
    typically performs some action on the data/labels before classification is performed

Within these more general categories, specific classifiers are implemented. For example, there are several `BoostedClassifier` subclasses:

`CombinedClassifier`
    combines predictions using a `PredictionsCombiner` functor

`MulticlassClassifier`
    performs multi-class classification by means of a list of `BinaryClassifier` instances. Typical use-case is to wrap a binary classifier to give it ability to operate on multiple classes via voting over classifiers for all possible pairs of the categories

`SplitClassifier`
    combines a `Classifier` and an arbitrary `Splitter`

Furthermore, there are also several `ProxyClassifier` subclasses:

`BinaryClassifier`
    maps a set of labels into two categories (+1 and -1)

`MappedClassifier`
    uses a mapper on input data prior to training/testing

`FeatureSelectionClassifier`
    performs some kind of `FeatureSelection` prior to training/testing

5.5.2 Implementation Examples

Classifiers such as the `FeatureSelectionClassifier` are particularly useful because they simplify the process of selecting features and then using only that subset of features to classify novel exemplars (the *predict* stage). They become even more powerful when combined with `SplitClassifier`, so that even the task of withholding certain data points to create statistically valid training and testing datasets is abstracted and wrapped up within a single object (and, ultimately, very few method calls). Consider the following code, which can be found in mvpa/clfs/warehouse.py:

```
>>> from mvpa.clfs.base import SplitClassifier, FeatureSelectionClassifier
>>> from mvpa.clfs.svm import LinearCSVMC
>>> from mvpa.clfs.transerror import ConfusionBasedError
>>> from mvpa.featsel.rfe import RFE
>>> from mvpa.featsel.helpers import FractionTailSelector
>>>
>>> rfesvm_split = SplitClassifier(LinearCSVMC())
>>> clf = \
...   FeatureSelectionClassifier(
...     clf = LinearCSVMC(),
...     # on features selected via RFE
...     feature_selection = RFE(
...         # based on sensitivity of a clf which does
...         # splitting internally
...         sensitivity_analyzer=rfesvm_split.getSensitivityAnalyzer(),
...         transfer_error=ConfusionBasedError(
...             rfesvm_split,
...             confusion_state="confusion"),
...             # and whose internal error we use
...         feature_selector=FractionTailSelector(
...                         0.2, mode='discard', tail='lower'),
...                         # remove 20% of features at each step
...         update_sensitivity=True),
...         # update sensitivity at each step
...     descr='LinSVM+RFE(splits_avg)' )
```

This analysis combines the `FeatureSelectionClassifier` and the `SplitClassifier` to perform internal splitting of the data and then perform FeatureSelection based on those splits. Such analyses can be easily created due to the straightforward way that classifier and meta-classifiers can be combined. Please refer to the relevant documentation sections for more information about the specifics of each meta-classifier.

5.6 Retrainable Classifiers

Some classifiers have ability to provide quick (i.e in terms of performance) re-training if they were previously trained, and only part of their specification got changed. For instance, for kernel-based classifier (e.g. GPR) it makes no sense to recompute kernel matrix, if only a classifier (not kernel) parameter (e.g. `sigma_noise`) was changed. Another similar usecase: for *null-hypothesis statistical testing* it might be needed to train classifier multiple times on a randomized set of labels.

Only classifiers which have `retrainable` in their `_clf_internals` are capable of efficient retraining. To enable retraining, just provide `retrainable=True` to the constructor of the classifier. Internally retrainable classifiers will try to deduce what was changed in the specification of the classifier (e.g. training/testing datasets, parameters) and act accordingly. To reduce training/prediction time even more, classifier might directly be instructed with what aspects were changed. It must be previously trained / predicted, so later on `retrain()` and `repredict()` methods could be called. `repredict()` can be called only with the same data, for which it was earlier predicted. See API doc for more information.

Implementation of efficient retraining is not straightforward, thus it is strongly advised to

- enable `CHECK_RETRAIN` debug target while developing the code for analysis. That might guard you against obvious misuses of retraining feature, as well as to spot bugs in the code

- validate on a simple dataset that analysis code provides the same results if classifier was created retrainable or not

5.7 Classifiers "Warehouse"

To facilitate easy trial of different classifiers for any specific task, `Warehouse` of classifiers clfs.warehouse.clfs was defined to create a sample collection of some commonly used parameterizations of the classifiers present in PyMVPA. Such collection can be queried by any set of known keywords/tags with tags prefixed with `!` being excluded:

```
>>> from mvpa.clfs.warehouse import clfs
>>> tryme = clfs['multiclass', '!svm']
```

to simply sweep through classifiers which are capable of multiclass classification and are not SVM based.

MEASURES

PyMVPA provides a number of useful measures. The vast majority of them are dedicated to feature selection. To increase analysis flexibility, PyMVPA distinguishes two parts of a feature selection procedure.

First, the impact of each individual feature on a classification has to be determined. The resulting map reflects the sensitivities of all features with respect to a certain decision and, therefore, algorithms generating these maps are summarized as `Sensitivity` in PyMVPA. Second, once the feature sensitivities are known, they can be used as criteria for feature selection. However, possible selection strategies range from very simple *Go with the 10% best features* to more complicated algorithms like *Recursive Feature Elimination*. Because *Sensitivity Measures* and selections strategies can be arbitrarily combined, PyMVPA offers a quite flexible framework for feature selection. Similar to dataset splitters, all PyMVPA algorithms are implemented and behave like *processing object*s. To recap, this means that they are instantiated by passing all relevant arguments to the constructor. Once created, they can be used multiple times by calling them with different datasets.

6.1 Sensitivity Measures

It was already mentioned that a `Sensitivity` computes a featurewise score that indicates how much interesting signal each feature contains – hoping that this score somehow correlates with the impact of the features on a classifier's decision for a certain problem.

Every sensitivity analyzer object computes a one-dimensional array with the respective score for every feature, when called with a `Dataset`. Due to this common behaviour all `Sensitivity` types are interchangeable and can be combined with any other algorithm requiring a sensitivity analyzer.

By convention higher sensitivity values indicate more interesting features.

There are two types of sensitivity analyzers in PyMVPA. Basic sensitivity analyzers directly compute a score from a Dataset. Meta sensitivity analyzers on the other hand utilize another sensitivity analyzer to compute their sensitivity maps.

6.1.1 Basic Sensitivity (and related Measures)

ANOVA

The `OneWayAnova` class provides a simple (and fast) univariate measure, that can be used for feature selection, although it is not a proper sensitivity measure. For each feature an individual F-score is computed as the fraction of between and within group variances. Groups are defined by samples with unique labels.

Higher F-scores indicate higher sensitivities, as with all other sensitivity analyzers.

Linear SVM Weights

The featurewise weights of a trained support vector machine are another possible sensitivity measure. The libsvm.LinearSVMWeights and sg.LinearSVMWeights class can internally train all types of *linear* support vector machines and report those weights.

In contrast to the F-scores computed by an ANOVA, the weights can be positive or negative, with both extremes indicating higher sensitivities. To deal with this property all subclasses of `DatasetMeasure` support a *transformer* arguments in the contructor. A transformer is a functor that is finally called with the computed sensitivity map. PyMVPA already comes with some convenience functors which can be used for this purpose (see `transformers`).

```
>>> from mvpa.misc.data_generators import normalFeatureDataset
>>> from mvpa.clfs.svm import LinearCSVMC
>>> from mvpa.misc.transformers import Absolute
>>>
>>> ds = normalFeatureDataset()
>>> ds
<Dataset / float64 100 x 4 uniq: 2 labels 5 chunks labels_mapped>
>>>
>>> clf = LinearCSVMC()
>>> sensana = clf.getSensitivityAnalyzer()
>>> sens = sensana(ds)
>>> sens.shape
(4,)
>>> (sens < 0).any()
True
>>> sensana_abs = clf.getSensitivityAnalyzer(transformer=Absolute)
>>> (sensana_abs(ds) < 0).any()
False
```

Above example shows how to use an existing classifier instance to report sensity values (a linear SVM in this case). The computed sensitivity vector contains one element for each feature in the dataset. `transformers` can be used to post-process the sensitvity scores, e.g. reporting absolute values for feature selection purposes, instead of raw sensitivities.

Note: The *SVMWeights* classes *cannot* extract reasonable weights from non-linear SVMs (e.g. with RBF kernels).

Other linear Classifier Weights

Any linear classifier in PyMVPA can report its weights. The procedure is identical for all of them. As outlined in the example using linear SVM weights, simply call `getSensitivityAnalyzer()` on a classifier instance and you'll get an appropriate `Sensitivity` object. Additionally, it is possible to force (re)training of the underlying classifier or simply report the weights computed during a previous training run.

Examples of other classifier-based linear sensitivity analyzers are: `SMLRWeights` and `GPRLinearWeights`.

Noise Perturbation

Noise perturbation is a generic approach to determine feature sensitivity. The sensitivity analyzer `NoisePerturbationSensitivity`) computes a scalar `DatasetMeasure` using the original dataset. Afterwards, for each single feature a noise pattern is added to the respective feature and the dataset measure is recomputed. The sensitivity of each feature is the difference between the dataset measure of the orginal dataset and the one with added noise. The reasoning behind this algorithm is that adding noise to *important* features will impair a dataset measure like cross-validated classifier transfer error. However, adding noise to a feature that already only contains noise, will not change such a measure.

Depending on the used scalar `DatasetMeasure` using the sensitivity analyzer might be really CPU-intensive! Also depending on the measure, it might be necessary to use appropriate `transformers` (see `transformers` constructor arguments) to ensure that higher values represent higher sensitivities.

6.1.2 Meta Sensitivity Measures

Meta Sensitivity Measures are FeaturewiseDatasetMeasures that internally use one of the Basic Sensitivity (and related Measures) to compute their sensitivity scores.

Splitting Measures

The SplittingFeaturewiseMeasure uses a `Splitter` to generate dataset splits. A FeaturewiseDatasetMeasure is then used to compute sensitivity maps for all these dataset splits. At the end a *combiner* function is called with all sensitivity maps to produce the final sensitivity map. By default the mean sensitivity maps across all splits is computed.

FEATURE SELECTION

This section has been contributed by James M. Hughes.

It is often the case in machine learning problems that we wish to reduce a feature space of high dimensionality into something more manageable by selecting only those features that contribute most to classification performance. Feature selection methods attempt to achieve this goal in an algorithmic fashion. PyMVPA's flexible framework allows various feature selection methods to take place within a small block of code. `FeatureSelectionClassifier` extends the basic classifier framework to allow for the use of arbitrary methods of feature selection according to whatever ranking metric, feature selection criteria, and stopping criterion the user chooses for a given application. Examples of the code/classification algorithms presented here can be found in mvpa/clfs/warehouse.py.

More formally, a `FeatureSelectionClassifier` is a meta-classifier. That is, it is not a classifier itself – it can take any *slave* `Classifier`, perform some feature selection in advance, select those features, and then train the provided *slave* `Classifier` on those features. Externally, however, it looks like a `Classifier`, in that it fulfills the specialization of the Classifier base class. The following are the relevant arguments to the constructor of such a `Classifier`:

clf: `Classifier`
> classifier based on which mask classifiers is created

feature_selection: FeatureSelection
> whatever feature selection is considered best

testdataset: `Dataset` (optional)
> dataset which would be given on call to feature_selection

Let us turn out attention to the second argument, FeatureSelection. As noted above, this feature selection can be arbitrary and should be chosen appropriately for the task at hand. For example, we could perform a one-way ANOVA statistic to select features, then keep only the most important 5% of them. It is crucial to note that, in PyMVPA, the way in which features are selected (in this example by keeping only 5% of them) is wholly independent of the way features are ranked (in this example, by using a one-way ANOVA). Feature selection using this method could be accomplished using the following code (from mvpa/clfs/warehouse.py):

```
>>> from mvpa.suite import *
>>> FeatureSelection = SensitivityBasedFeatureSelection(
...     OneWayAnova(),
...     FractionTailSelector(0.05, mode='select', tail='upper'))
```

A more interesting analysis is one in which we use the weights (hyperplane coefficients) to rank features. This allows us to use the same classifier to train the selected features as we used to select them:

It bears mentioning at this point that caution must be exercised when selecting features. The process of feature selection must be performed on an independent training dataset: it is not possible to select features using the entire dataset, re-train a classifier on a subset of the original data (but using only the selected features) and then test on a held-out testing dataset. This results in an obvious positive bias in classification performance. PyMVPA allows for easy dataset splitting, however, so creating independent training and testing datasets is easily accomplished, for instance using an `NFoldSplitter`, `OddEvenSplitter`, etc.

7.1 Recursive Feature Elimination

Recursive feature elimination (RFE, applied to fMRI data in (*Hanson et al., 2008*)) is a technique that falls under the larger umbrella of feature selection. Recursive feature elimination specifically attempts to reduce the number of selected features used for classification in the following way:

- A classifier is trained on a subset of the data and features are ranked according to an arbitrary metric.

- Some amount of those features is either selected or discarded according to a pre-selected rule.

- The classifier is retrained and features are once again ranked; this process continues until some criterion determined textit{a priori} (such as classification error) is reached.

- One or more classifiers trained only on the final set of selected features are used on a generalization dataset and performance is calculated.

PyMVPA's flexible framework allows each of these steps to take place within a small block of code. To actually perform recursive feature elimination, we consider two separate analysis scenarios that deal with a pre-selected training dataset:

- We split the training dataset into an arbitrary number of independent datasets and perform RFE on each of these; the sensitivity analysis of features is performed independently for each split and features are selected based on those independent measures.

- We split the training dataset into an arbitrary number of independent datasets (as before), but we average the feature sensitivities and select which features to prune/select based on that one average measure.

We will concentrate on the second approach. The following code can be used to perform such an analysis:

```
>>> rfesvm_split = SplitClassifier(LinearCSVMC())
>>> clf = \
...   FeatureSelectionClassifier(
...     clf = LinearCSVMC(),
...     # on features selected via RFE
...     feature_selection = RFE(
...         # based on sensitivity of a clf which does splitting internally
...         sensitivity_analyzer=rfesvm_split.getSensitivityAnalyzer(),
...         transfer_error=ConfusionBasedError(
...           rfesvm_split,
...           confusion_state="confusion"),
...           # and whose internal error we use
...         feature_selector=FractionTailSelector(
...                         0.2, mode='discard', tail='lower'),
...                         # remove 20% of features at each step
...         update_sensitivity=True),
...         # update sensitivity at each step
...     descr='LinSVM+RFE(splits_avg)' )
```

The code above introduces the `SplitClassifier`, which in this case is yet another *meta-classifier* that takes in a `Classifier` (in this case a LinearCSVMC) and an arbitrary `Splitter` object, so that the dataset can be split in whatever way the user desires. Prior to training, the `SplitClassifier` splits the training dataset, dedicates a separate classifier to each split, trains each on the training part of the split, and then computes transfer error on the testing part of the split. If a `SplitClassifier` instance is later on asked to *predict* some new data, it uses (by default) the MaximalVote strategy to derive an answer. A summary about the performance of a `SplitClassifier` internally on each split of the training dataset is available by accessing the *confusion* state variable.

To summarize somewhat, RFE is just one method of feature selection, so we use a `FeatureSelectionClassifier` to facilitate this. To parameterize the RFE process, we refer above to the following:

sensitivity_analyzer
> in this case just the default from a linear C-SVM (the SVM weights), taken as an average over all splits (in accordance with scenario 2 as above)

transfer_error
> confusion-based error that relies on the confusion matrices computed during splitting of the dataset by the `SplitClassifier`; this is used to provide a value that can be compared against a stopping criterion to stop eliminating features

feature_selector
> in this example we simply discard the 20% of features deemed least important

update_sensitivity
> true to retrain the classifiers each time we eliminate features; should be false if a non-classifier-based sensitivity measure (such as one-way ANOVA) is used

As has been shown, recursive feature elimination is an easy-to-implement, flexible, and powerful tool within the PyMVPA framework. Various ranking methods for selecting features have been discussed. Additionally, several analysis scenarios have been presented, along with enough requisite knowledge that the user can plug in whatever classifiers, error metrics, or sensitivity measures are most appropriate for the task at hand.

7.2 Incremental Feature Search

IFS

(to be written)

MISCELLANEOUS

8.1 Managing (Custom) Configurations

PyMVPA provides a facility to handle arbitrary configuration settings. This facility can be used to control some aspects of the behavior of PyMVPA itself, as well as to store and query custom configuration items, e.g. to control one's own analysis scripts.

An instance of this configuration manager is loaded whenever the *mvpa* module is imported. It can be used from any script like this:

```
>>> from mvpa import cfg
```

By default the config manager reads settings from two config files (if any of them exists). The first is a file named *.pymvpa.cfg* and located in the user's home directory. The second is *pymvpa.cfg* in the current directory. Please note, that settings found in the second file override the ones in the first.

The syntax of both files is the one also known from the Windows INI files. Basically, Python's ConfigParser is used to read those file and the config supports whatever this parser can read. A minimal example config file might look like this:

```
[general]
verbose = 1
```

It consists of a section *general* containing a single setting *verbose*, which is set to *1*. PyMVPA recognizes a number of such sections and configuration variables. A full list is shown at the end of this section and is also available in the source package (*doc/examples/pymvpa.cfg*).

In addition to configuration files, the config manager also looks for special environment variables to read settings from. Names of such variables have to start with *MVPA_* following by the an optional section name and the variable name itself (with _ as delimiter). If no section name is provided, the variables will be associated with section *general*. Some examples:

```
MVPA_VERBOSE=1
```

will become:

```
[general]
verbose = 1
```

However, **MVPA_VERBOSE_OUTPUT** = *stdout* becomes:

```
[verbose]
output = stdout
```

Any lenght of variable name is allowed, e.g. *MVPA_SEC1_LONG_VARIABLE_NAME=1* becomes:

```
[sec1]
long variable name = 1
```

Settings read from environment variables have the highest priority and override settings found in the config files. Therefore environment variables can be used to quickly adjust some setting without having to edit the config files.

The config manager can easily be queried from inside scripts. In addition to the interface of Python's ConfigParser it has a few convenience functions mostly to allow for a default value in case no setting was found. For example:

```
>>> cfg.getboolean('warnings', 'suppress', default=False)
False
```

queries the config manager whether warnings should be suppressed (i.e. if there is a variable *suppress* in section *warnings*). In case, there is now such setting, i.e. neither config files nor environment variables defined it, the *default* values is returned. Please see the documentation of ConfigManager for its full functionality. The source tarballs includes an example configuration file (*doc/examples/pymvpa.cfg*) with the comprehensive list of settings recognized by PyMVPA itself:

```
### ### ### ### ### ### ### ### ### ### ### ### ### ### ### ### ### ### ### ##
#
#   Example configuration file to be used with PyMVPA
#
#
#   See COPYING file distributed along with the PyMVPA package for the
#   copyright and license terms.
#
### ### ### ### ### ### ### ### ### ### ### ### ### ### ### ### ### ### ### ##

# This is a comprehensive list of all settings currently recognized by PyMVPA.
# Users can add arbitrary additional settings, both in new and already existing
# sections.
[general]
#debug =
#verbose =
#seed = 12345

[verbose]
# comma-separated list of handlers, e.g. stdout
#output =

[error]
#output =

[warnings]
# integer
#bt =
# integer
#count =
# comma-separated list of handlers, e.g. stdout
#output =
# Boolean (former: MVPA_NO_WARNINGS)
suppress = no

[debug]
# comma-separated list of handlers, e.g. stdout
#output =
#metrics =

[examples]
interactive = yes
```

```
[svm]
# which SVM implementation to use by default: libsvm or shogun
backend = libsvm

[externals]
# whether to really raise an exception when an externals test fails _and_
# raising an exception was requested
raise exception = True

# whether to retest the availability of an external dependency, depite an
# already present (but possibly outdated) test result
retest = no

# options starting with 'have ' indicate the presence or absence of external
# dependencies
#have scipy = no

[tests]
# whether to perform tests where the outcome is not deterministic
labile = yes

# if enabled, the unit tests will not run multiple classifiers on the same
# test, which reduces the time to run a full test significantly.
quick = no

# if enabled, unit tests consuming lots of memory will not automatically run
# as part of the main unittest battery
lowmem = no
```

8.2 Progress Tracking

There are 3 types of messages PyMVPA can produce:

> verbose
> > regular informative messages about generic actions being performed
>
> debug
> > messages about the progress of computation, manipulation on data structures
>
> warning
> > messages which are reported by mvpa if something goes a little unexpected but not critical

8.2.1 Redirecting Output

By default, all types of messages are printed by PyMVPA to the standard output. It is possible to redirect them to standard error, or a file, or a list of multiple such targets, by using environment variable MVPA_?_OUTPUT, where X is either VERBOSE, DEBUG, or WARNING correspondingly. E.g.:

```
export MVPA_VERBOSE_OUTPUT=stdout,/tmp/1 MVPA_WARNING_OUTPUT=/tmp/3 MVPA_DEBUG_OUTPUT=stderr,/tmp,
```

would direct verbose messages to standard output as well as to /tmp/1 file, warnings will be stored only in /tmp/3, and debug output would appear on standard error output, as well as in the file /tmp/2.

PyMVPA output redirection though has no effect on external libraries debug output if corresponding debug target is enabled

> shogun
> > debug output (if any of internal SG_ debug targets is enabled) appears on standard output

SMLR
> debug output (if `SMLR_` debug target is enabled) appears on standard output

libsvm
> debug output (if `LIBSVM` debug target is enabled) appears on standard error

One of the possible redirections is Python's `StringIO` class. Instance of such class can be added to the `handlers` and queried later on for the information to be dumped to a file later on. It is useful if output path is specified at run time, thus it is impossible to redirect verbose or debug from the start of the program:

```
>>> import sys
>>> from mvpa.base import verbose
>>> from StringIO import StringIO
>>> stringout = StringIO()
>>> verbose.handlers = [sys.stdout, stringout]
>>> verbose.level = 3
>>>
>>> verbose(1, 'msg1')
 msg1
>>> out_prefix='/tmp/'
>>>
>>> verbose(2, 'msg2')
  msg2
>>> # open('%sverbose.log' % out_prefix, 'w').write(stringout.getvalue())
>>> print stringout.getvalue(),
 msg1
  msg2
>>>
```

8.2.2 Verbose Messages

Primarily for a user of PyMVPA to provide information about the progress of their scripts. Such messages are printed out if their level specified as the first parameter to verbose function call is less than specified. There are two easy ways to specify verbosity level:

- command line: you can use opt.verbose for precrafted command line option for to give facility to change it from your script (see examples)

- environment variable **MVPA_VERBOSE**

- code: verbose.level property

The following verbosity levels are supported:

0	nothing besides errors
1	high level stuff – top level operation or file operations
2	cmdline handling
3	n.a.
4	computation/algorithm relevant thing

8.2.3 Warning Messages

Reported by PyMVPA if something goes a little unexpected but not critical. By default they are printed just once per occasion, i.e. once per piece of code where it is called. Following environment variables control the behavior of warnings:

- **MVPA_WARNINGS_COUNT** =*<int>* controls for how many invocations of specific warning it gets printed (default behavior is 1 for once). Specification of negative count results in all invocations being printed, and value of 0 obviously suppresses the warnings

- **MVPA_WARNINGS_SUPPRESS** analogous to **MVPA_WARNINGS_COUNT** =*0* it resultant behavior

- **MVPA_WARNINGS_BT** =*<int>* controls up to how many lines of traceback is printed for the warnings

In python code, invocation of warning with argument `bt = True` enforces printout of traceback whenever warning tracebacks are disabled by default.

8.2.4 Debug Messages

Debug messages are used to track progress of any computation inside PyMVPA while the code run by python without optimization (i.e. without `-O` switch to python). They are specified not by the level but by some id usually specific for a particular PyMVPA routine. For example `RFEC` id causes debugging information about Recursive Feature Elimination call to be printed (See misc module sources for the list of all ids, or print `debug.registered` property).

Analogous to verbosity level there are two easy ways to specify set of ids to be enabled (reported):

- command line: you can use optDebug for precrafted command line option to provide it from your script (see examples). If in command line if optDebug is used, `-d list` is given, PyMVPA will print out list of known ids.

- environment: variable **MVPA_DEBUG** can contain comma-separated list of ids or python regular expressions to match multiple ids. Thus specifying **MVPA_DEBUG** =*CLF.** would enable all ids which start with `CLF`, and **MVPA_DEBUG** =*.** would enable all known ids.

- code: debug.active property (e.g. `debug.active = [ 'RFEC', 'CLF' ]`)

Besides printing debug messages, it is also possible to print some metric. You can define new metrics or select predefined ones:

vmem
     (Linux specific): amount of virtual memory consumed by the task
pid
     (Linux specific): PID of the process
reltime
     How many seconds passed since previous debug printout
asctime
     Time stamp
tb   Traceback (`module1:line_number1[,line_number2...]>module2:line_number..)`
     where this debug statement was requested
tbc
     Concise traceback printout – prefix common with the previous invocation is replaced with `...`

To enable list of metrics you can use **MVPA_DEBUG_METRICS** environment variable to list desired metric names comma-separated. If `ALL` is provided, it enables all the metrics.

As it was mentioned earlier, debug messages are printed only in non-optimized python invocation. That was done to eliminate any slowdown introduced by such 'debugging' output, which might appear at some computational bottleneck places in the code.

Some of the debug ids are defined to facilitate additional checking of the validity of the analysis. Their debug ids a prefixed by `CHECK_`. E.g. `CHECK_RETRAIN` id would cause additional checking of the data in retraining phase. Such additional testing might spot out some bugs in the internal logic, thus enabled when full test suite is ran.

8.3 Additional Little Helpers

8.3.1 Random Number Generation

To facilitate reproducible troubleshooting, a seed value of random generator of NumPy can be provided in debug mode (python is called without `-O`) via environment variable **MVPA_SEED =<*int*>**. Otherwise it gets seeded with random integer which can be displayed with debug id `RANDOM` e.g.:

```
> MVPA_SEED=123 MVPA_DEBUG=RANDOM python test_clf.py
[RANDOM] DBG: Seeding RNG with 123
...
> MVPA_DEBUG=RANDOM python test_clf.py
[RANDOM] DBG: Seeding RNG with 1447286079
...
```

8.3.2 Unittests at a Grasp

If it is needed to just quickly grasp through all unittests without making them to test multiple classifiers (implemented with sweeparg), define environmental variable **MVPA_TESTS_QUICK** e.g.:

```
> MVPA_WARNINGS_SUPPRESS=no MVPA_TESTS_QUICK=yes python test_clf.py
...............
----------------------------------------------------------------------
Ran 15 tests in 0.845s
```

Some tests are not 100% deterministic as they operate on random data (e.g. the performance of a randomly initialized classifier). Therefore, in some cases, specific unit tests might fail when running the full test battery. To exclude these test cases (and only those where non-deterministic behavior immanent) one can use the **MVPA_TESTS_LABILE** configuration and set it to 'off'.

8.3.3 Others

(to be written)

8.4 FSL Bindings

PyMVPA contains a few little helpers to make interfacing with FSL easier. The purpose of these helpers is to increase the efficiency when doing an analysis by (re)using useful information that is already available from some FSL output. FSL usually stores most interesting information in the NIfTI format. Therefore it can be easily imported into PyMVPA using PyNIfTI. However, some information is stored in textfiles, e.g. estimated motion correction parameters and *FEAT's three-column custom EV* files. PyMVPA provides import and export helpers for both of them (among other stuff like a *MELODIC* results import helper). Here is an example how the *McFlirt* parameter output can be used to perform motion-aware data detrending:

```python
>>> from os import path
>>> import numpy as N
>>>
>>> # some dummy dataset
>>> from mvpa.datasets import Dataset
>>> ds = Dataset(samples=N.random.normal(size=(19, 3)), labels=1)
>>>
>>> # load motion correction output
>>> from mvpa.misc.fsl.base import McFlirtParams
>>> mc = McFlirtParams(path.join('data', 'bold_mc.par'))
>>>
```

```
>>> # simple plot using pylab (use pylab.show() or pylab.savefig()
>>> # afterwards)
>>> mc.plot()
>>>
>>> # detrend some dataset with mc params as additonal regressors
>>> from mvpa.datasets.miscfx import detrend
>>> res = detrend(ds, model='regress', opt_reg=mc.toarray())
>>> # 'res' contains all regressors and their associated weights
```

All FSL bindings are located in the mvpa.misc.fsl module.

FULL EXAMPLES

Each of the examples in this section is a stand-alone script containing all necessary code to run some analysis. All examples are shipped with PyMVPA and can be found in the *doc/examples/* directory in the source package. This directory might include some more special-interest examples which are not listed here.

Some examples need to access a sample dataset available in the *data/* directory within the root of the PyMVPA hierarchy, and thus have to be invoked directly from PyMVPA root (e.g. *doc/examples/searchlight_2d.py*). Alternatively, one can download a full example dataset, which is explained in the next section.

9.1 Example fMRI Dataset

For an easy start with PyMVPA an example fMRI dataset is provided. This is a single subject from a study published by *Haxby et al. (2001)*. This dataset has already been repeatedly reanalyzed since its first publication (e.g. *Hanson et al (2004)* and *O'Toole et al. (2005)*).

Note: The orginal authors of *Haxby et al. (2001)* hold the copyright of this dataset and made it available under the terms of the Creative Commons Attribution-Share Alike 3.0 license.

The subset of the dataset that is available here has been converted into the NIfTI dataformat and is preprocessed to a degree that should allow people without prior fMRI experience to perform meaningful analyses. Moreover, it should not require further preprocessing with external tools.

All preprocessing has been performed using tools from FSL. Specifically, the 4D fMRI timeseries has been skull-stripped and thresholded to zero-out non-brain voxels (using a brain outline estimate significantly larger than the brain, to prevent removal of edge voxels actually covering brain tissue). The corresponding commandline call to BET was:

```
bet bold bold_brain -F -f 0.5 -g 0
```

Afterwards the timeseries has been motion-corrected using MCFLIRT:

```
mcflirt -in bold_brain -out bold_mc -plots
```

The following files are available in the example fMRI dataset download (approx. 100 MB):

bold.nii.gz

The motion-corrected and skull-stripped 4D timeseries (1452 volumes with 40 x 64 x 64 voxels, corresponding to a voxelsize of 3.5 x 3.75 x 3.75 mm and a volume repetition time of 2.5 seconds). The timeseries contains all 12 runs of the original experiment, concatenated in a single file. Please note, that the timeseries signal is *not* detrended.

bold_mc.par

The motion correction parameter output. This is a 6-column textfile with three rotation and three translation parameters respectively. This information can be used e.g. as additional regressors for *motion-aware timeseries detrending*.

mask.nii.gz
> A binary mask with a conservative brain outline estimate, i.e. including some non-brain voxels to prevent the exclusion of brain tissue.

attributes_literal.txt
> A two-column textfile with the stimulation condition and the corresponding experimental run for each volume in the timeseries image. The labels a given in literal form (e.g. 'face').

attributes.txt
> Similiar as *attributes_literal.txt*, but with the condition labels encoded as integers. This file is only provided for earlier PyMVPA version, that could not handle *literal labels*.

Once downloaded and extracted (e.g. into a folder *data/*), the dataset can be easily loaded like this:

```
>>> attrs = SampleAttributes('data/attributes_literal.txt',
...                          literallabels=True)
>>> ds = NiftiDataset(samples='data/bold.nii.gz',
...                   labels=attrs.labels,
...                   chunks=attrs.chunks,
...                   labels_map=True,
...                   mask='data/mask.nii.gz')
```

Note: The dataset used in the *examples* shipped with PyMVPA is actually a minimal version (posterior half of a single brain slice) of this full dataset. After appropriately adjusting the path, it is possible to run several of the examples on this full dataset.

9.2 Preprocessing

9.2.1 Simple preprocessing of event-related data using FSL EV3 design files

Events are defined using FSL's EV3 format (onset, duration, intensity). For each event a number of volumes is selected and the mean volume is computed. The selection method is a boxcar with defined length and offset from the event onset (–boxlength, –boxoffset). The computed event data samples are written back to a NIfTI file using the header information of the source timeseries.

Additonally a file listing the sample attributes (label, chunk) is created. All samples are assumed to be from the same chunk, where the chunk id can be set using the –chunk option. If the source NIfTI data file contains more than one chunk, one can be selected using the –chunklimits option. This will have the effect that linear detrending (–detrend) will perform a separate fit for this chunk and will not simply remove a global trend.

```
>>> from mvpa.misc.fsl.base import FslEV3
>>> from mvpa.suite import *
>>> from nifti.utils import time2vol
>>>
>>> def main():
>>>     """ Wrapped into a function call for easy profiling later on
>>>     """
>>>
>>>     parser.usage = """\
>>>     %s [options] <NIfTI data> <output prefix> <EV file 1> [ <EV file 2> ... ]
>>>     """ \
>>>     % sys.argv[0]
>>>
>>>     parser.option_groups += [ opts.common, opts.box, opts.chunk]
>>>     parser.option_list += [opt.tr, opt.detrend]
>>>
>>>     (options, args) = parser.parse_args()
>>>
>>>     if not len(args) >= 3:
```

```
>>>            parser.error("Insufficient arguments.")
>>>            sys.exit(1)
>>>
>>>      verbose(1, "Loading data")
>>>
>>>      # data filename
>>>      dfile = args[0]
>>>      # output prefix
>>>      oprefix = args[1]
>>>      # list of EV files
>>>      evfiles = args[2:]
>>>
>>>      verbose(2, "Reading conditions from files")
>>>      evs = [ FslEV3(evfile) for evfile in evfiles ]
>>>
>>>      verbose(2, "Loading volume file %s" % dfile)
>>>      nimg = NiftiImage(dfile)
>>>
>>>      verbose(1, "Preprocess data")
>>>      # force float32 to prevent unecessary upcasting of int to float64
>>>      data = nimg.data.astype('float32')
>>>
>>>      if options.detrend:
>>>          if options.chunklimits == None:
>>>              verbose(2, "Linear detrending (whole dataset)")
>>>              data = detrend(data, axis=0)
>>>          else:
>>>              verbose(2, "Linear detrending (data chunk only)")
>>>              limits = [int(i) for i in options.chunklimits.split(',')]
>>>
>>>              if data[limits[0]:limits[1],:].shape[0] == 0:
>>>                  raise ValueError, 'Invalid chunklimit value [%s].' \
>>>                                    % options.chunklimits
>>>              # use limits to do a piecewise linear detrending (separate
>>>              # linear fit of the interesting chunk
>>>              data = detrend(data, axis=0, bp=limits)
>>>
>>>
>>>      verbose(2, "Convert onsets into volume ids")
>>>      # transform onset time into a volume id
>>>      onset_vols = [ time2vol(ev.onsets, options.tr, 0.0,
>>>                  decimals = 0 ).astype('int') for ev in evs ]
>>>
>>>      verbose(2, "Compute EV samples")
>>>
>>>      labels = []
>>>      chunks = []
>>>      samples = []
>>>
>>>      print data.shape
>>>      # for each condition -> label
>>>      for label, onsets in enumerate(onset_vols):
>>>          # mean of all volumes in a window after each onset vol
>>>          samples.append(transformWithBoxcar(data,
>>>                                             onsets,
>>>                                             options.boxlength,
>>>                                             offset=options.boxoffset,
>>>                                             fx=N.mean))
>>>          labels += [label] * len(onsets)
>>>          chunks += [options.chunk] * len(onsets)
>>>
>>>      # concatenate into a single array (assumes that each entry in
```

```
>>>     # samples already is 4d which is true, because
>>>     # transformWithBoxcar does it like that
>>>     samples = N.concatenate(samples, axis=0)
>>>
>>>     attrs = SampleAttributes({'labels': labels, 'chunks': chunks})
>>>
>>>     verbose(1, "Store results")
>>>     NiftiImage(samples, nimg.header).save(oprefix + '.nii.gz')
>>>     attrs.tofile(oprefix + '.attrs.txt')
>>>
>>>
>>> # if ran stand-alone
>>> if __name__ == "__main__":
>>>     main()
```

See Also:

The full source code of this example is included in the PyMVPA source distribution (*doc/examples/preproc_fsl_ev.py*).

9.2.2 Visualization of Data Projection Methods

```
>>> from mvpa.misc.data_generators import noisy_2d_fx
>>> from mvpa.mappers.pca import PCAMapper
>>> from mvpa.mappers.svd import SVDMapper
>>> from mvpa.mappers.ica import ICAMapper
>>> from mvpa import cfg
>>>
>>> import pylab as P
>>> import numpy as N
>>> center = [10, 20]
>>> axis_range = 7
>>>
>>> def plotProjDir(mproj):
>>>     p = mproj + N.array(center).T
>>>
>>>     P.plot([center[0], p[0,0]], [center[1], p[0,1]], hold=True)
>>>     P.plot([center[0], p[1,0]], [center[1], p[1,1]], hold=True)
>>>
>>>
>>>
>>> mappers = {
>>>             'PCA': PCAMapper(),
>>>             'SVD': SVDMapper(),
>>>             'ICA': ICAMapper(),
>>>         }
>>> datasets = [
>>>     noisy_2d_fx(100, lambda x: x, [lambda x: x],
>>>                 center, noise_std=.5),
>>>     noisy_2d_fx(50, lambda x: x, [lambda x: x, lambda x: -x],
>>>                 center, noise_std=.5),
>>>     noisy_2d_fx(50, lambda x: x, [lambda x: x, lambda x: 0],
>>>                 center, noise_std=.5),
>>>     ]
>>>
>>> ndatasets = len(datasets)
>>> nmappers = len(mappers.keys())
>>>
>>> P.figure(figsize=(8,8))
>>> fig = 1
>>>
```

```
>>> for ds in datasets:
>>>     for mname, mapper in mappers.iteritems():
>>>         mapper.train(ds)
>>>
>>>         dproj = mapper.forward(ds.samples)
>>>         mproj = mapper.proj
>>>         print mproj
>>>
>>>         P.subplot(ndatasets, nmappers, fig)
>>>         if fig <= 3:
>>>             P.title(mname)
>>>         P.axis('equal')
>>>
>>>         P.scatter(ds.samples[:, 0],
>>>                   ds.samples[:, 1],
>>>                   s=30, c=(ds.labels) * 200)
>>>         plotProjDir(mproj)
>>>         fig += 1
>>>
>>>
>>> if cfg.getboolean('examples', 'interactive', True):
>>>     P.show()
>>>
```
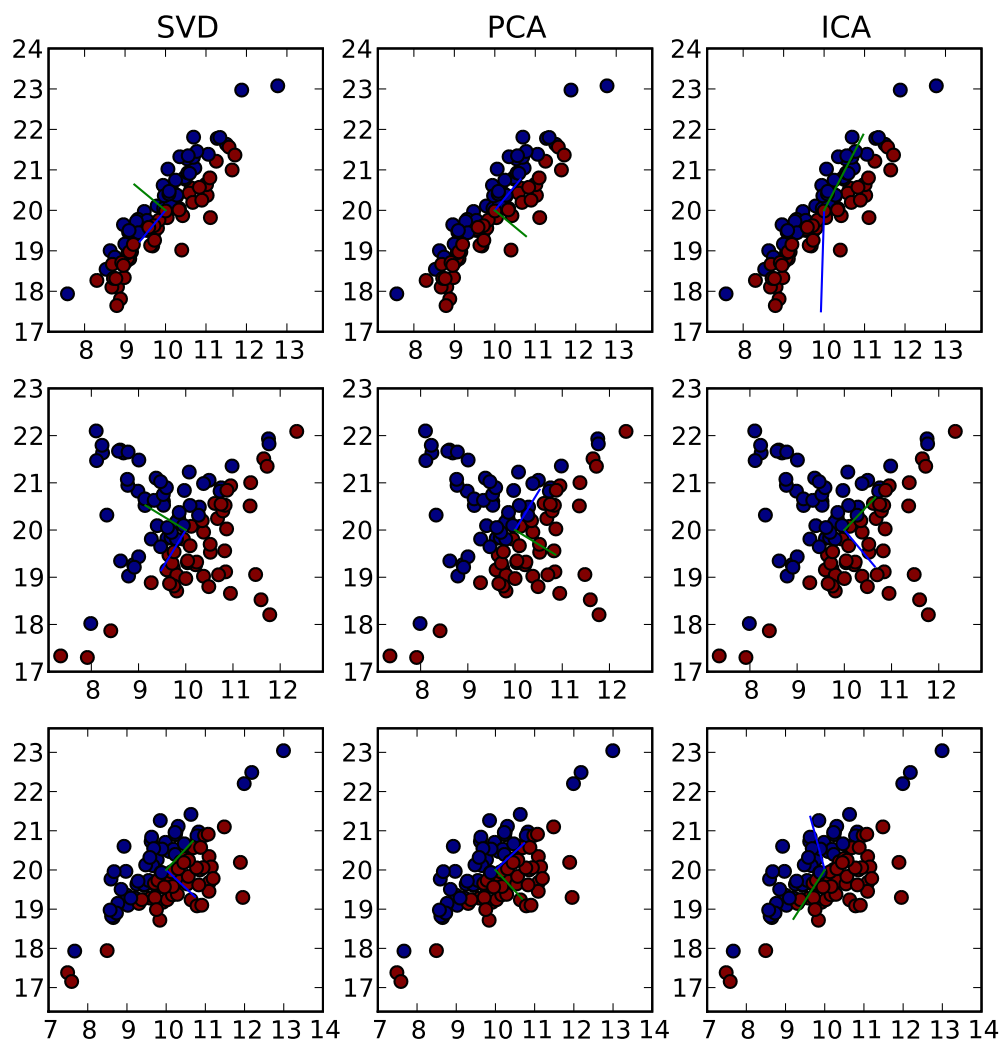
Output of the example:

See Also:

The full source code of this example is included in the PyMVPA source distribution (*doc/examples/projections.py*).

9.2.3 Simple Data-Exploration

Example showing some possibilities of data exploration (i.e. to 'smell' data).

```python
>>> import numpy as N
>>> import pylab as P
>>>
>>> from mvpa.misc.plot import plotFeatureHist, plotSamplesDistance
>>> from mvpa import cfg
>>> from mvpa.datasets.nifti import NiftiDataset
>>> from mvpa.misc.io import SampleAttributes
>>> from mvpa.datasets.miscfx import zscore, detrend
>>>
>>> # load example fmri dataset
>>> attr = SampleAttributes('data/attributes.txt')
>>> ds = NiftiDataset(samples='data/bold.nii.gz',
```

```
>>>                    labels=attr.labels,
>>>                    chunks=attr.chunks,
>>>                    mask='data/mask.nii.gz')
>>>
>>> # only use the first 5 chunks to save some cpu-cycles
>>> ds = ds.selectSamples(ds.chunks < 5)
>>>
>>> # take a look at the distribution of the feature values in all
>>> # sample categories and chunks
>>> plotFeatureHist(ds, perchunk=True, bins=20, normed=True,
>>>                  xlim=(0, ds.samples.max()))
>>> if cfg.getboolean('examples', 'interactive', True):
>>>     P.show()
>>>
>>> # next only works with floating point data
>>> ds.setSamplesDType('float')
>>>
>>> # look at sample similiarity
>>> # Note, the decreasing similarity with increasing temporal distance
>>> # of the samples
>>> P.subplot(121)
>>> plotSamplesDistance(ds, sortbyattr='chunks')
>>> P.title('Sample distances (sorted by chunks)')
>>>
>>> # similar distance plot, but now samples sorted by their
>>> # respective labels, i.e. samples with same labels are plotted
>>> # in adjacent columns/rows.
>>> # Note, that the first and largest group corresponds to the
>>> # 'rest' condition in the dataset
>>> P.subplot(122)
>>> plotSamplesDistance(ds, sortbyattr='labels')
>>> P.title('Sample distances (sorted by labels)')
>>> if cfg.getboolean('examples', 'interactive', True):
>>>     P.show()
>>>
>>>
>>> # z-score features individually per chunk
>>> print 'Detrending data'
>>> detrend(ds, perchunk=True, model='regress', polyord=2)
>>> print 'Z-Scoring data'
>>> zscore(ds)
>>>
>>> P.subplot(121)
>>> plotSamplesDistance(ds, sortbyattr='chunks')
>>> P.title('Distances: z-scored, detrended (sorted by chunks)')
>>> P.subplot(122)
>>> plotSamplesDistance(ds, sortbyattr='labels')
>>> P.title('Distances: z-scored, detrended (sorted by labels)')
>>> if cfg.getboolean('examples', 'interactive', True):
>>>     P.show()
>>>
>>> # XXX add some more, maybe show effect of preprocessing
>>>
```

Outputs of the example script. Data prior to preprocessing
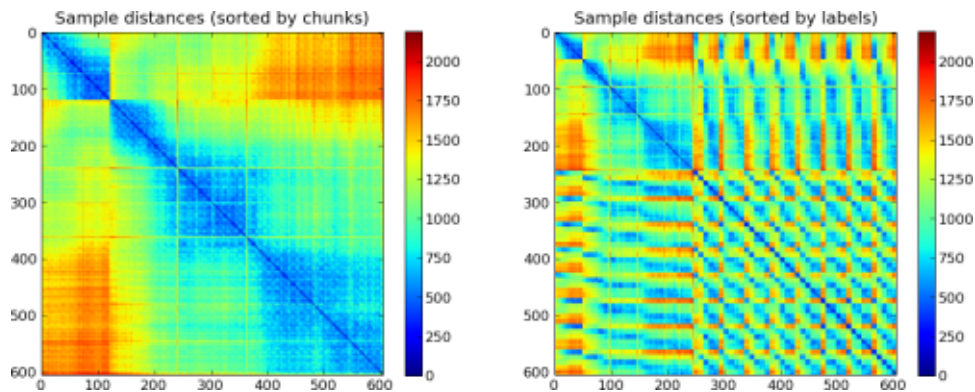
Data after minimal preprocessing
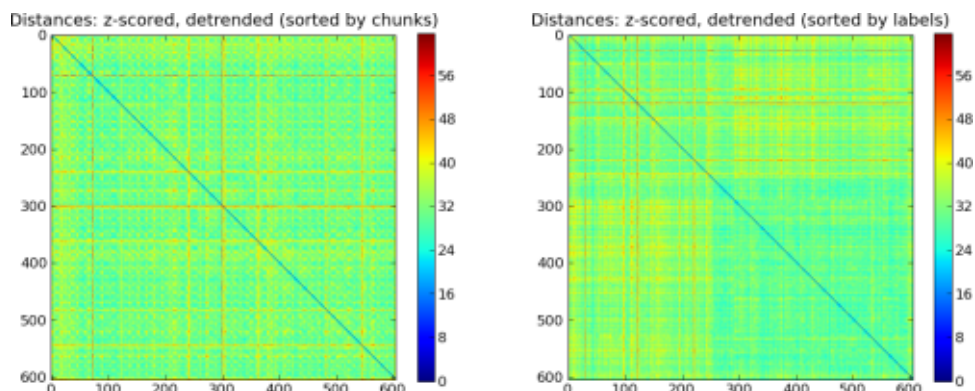
See Also:

The full source code of this example is included in the PyMVPA source distribution (*doc/examples/smellit.py*).

9.3 Analysis

9.3.1 Tiny Example of a Full Cross-Validation

Very, very simple example showing a complete cross-validation procedure with no fancy additions whatsoever.

```
>>> # get PyMVPA running
>>> from mvpa.suite import *
>>>
>>> # load PyMVPA example dataset
>>> attr = SampleAttributes('data/attributes.txt')
>>> dataset = NiftiDataset(samples='data/bold.nii.gz',
>>>                        labels=attr.labels,
>>>                        chunks=attr.chunks,
>>>                        mask='data/mask.nii.gz')
>>>
>>> # do chunkswise linear detrending on dataset
>>> detrend(dataset, perchunk=True, model='linear')
>>>
>>> # zscore dataset relative to baseline ('rest') mean
>>> zscore(dataset, perchunk=True, baselinelabels=[0],
>>>        targetdtype='float32')
>>>
>>> # select class 1 and 2 for this demo analysis
>>> # would work with full datasets (just a little slower)
>>> dataset = dataset.selectSamples(
>>>                 N.array([l in [1, 2] for l in dataset.labels],
>>>                         dtype='bool'))
```

```
>>>
>>> # setup cross validation procedure, using SMLR classifier
>>> cv = CrossValidatedTransferError(
>>>             TransferError(SMLR()),
>>>             OddEvenSplitter())
>>> # and run it
>>> error = cv(dataset)
>>>
>>> print "Error for %i-fold cross-validation on %i-class problem: %f" \
>>>        % (len(dataset.uniquechunks), len(dataset.uniquelabels), error)
```

See Also:

The full source code of this example is included in the PyMVPA source distribution (*doc/examples/start_easy.py*).

9.3.2 Compare SMLR to Linear SVM Classifier

Runs both classifiers on the the same dataset and compare their performance. This example also shows an example usage of confusion matrices and how two classifers can be combined.

```
>>> from mvpa.suite import *
>>>
>>> if __debug__:
>>>     debug.active.append('SMLR_')
>>>
>>> # features of sample data
>>> print "Generating samples..."
>>> nfeat = 10000
>>> nsamp = 100
>>> ntrain = 90
>>> goodfeat = 10
>>> offset = .5
>>>
>>> # create the sample datasets
>>> samp1 = N.random.randn(nsamp,nfeat)
>>> samp1[:,:goodfeat] += offset
>>>
>>> samp2 = N.random.randn(nsamp,nfeat)
>>> samp2[:,:goodfeat] -= offset
>>>
>>> # create the pymvpa training dataset from the labeled features
>>> patternsPos = Dataset(samples=samp1[:ntrain,:], labels=1)
>>> patternsNeg = Dataset(samples=samp2[:ntrain,:], labels=0)
>>> trainpat = patternsPos + patternsNeg
>>>
>>> # create patters for the testing dataset
>>> patternsPos = Dataset(samples=samp1[ntrain:,:], labels=1)
>>> patternsNeg = Dataset(samples=samp2[ntrain:,:], labels=0)
>>> testpat = patternsPos + patternsNeg
>>>
>>> # set up the SMLR classifier
>>> print "Evaluating SMLR classifier..."
>>> smlr = SMLR(fit_all_weights=True)
>>>
>>> # enable saving of the values used for the prediction
>>> smlr.states.enable('values')
>>>
>>> # train with the known points
>>> smlr.train(trainpat)
>>>
>>> # run the predictions on the test values
```

```
>>> pre = smlr.predict(testpat.samples)
>>>
>>> # calculate the confusion matrix
>>> smlr_confusion = ConfusionMatrix(
>>>     labels=trainpat.uniquelabels, targets=testpat.labels,
>>>     predictions=pre)
>>>
>>> # now do the same for a linear SVM
>>> print "Evaluating Linear SVM classifier..."
>>> lsvm = LinearNuSVMC(probability=1)
>>>
>>> # enable saving of the values used for the prediction
>>> lsvm.states.enable('values')
>>>
>>> # train with the known points
>>> lsvm.train(trainpat)
>>>
>>> # run the predictions on the test values
>>> pre = lsvm.predict(testpat.samples)
>>>
>>> # calculate the confusion matrix
>>> lsvm_confusion = ConfusionMatrix(
>>>     labels=trainpat.uniquelabels, targets=testpat.labels,
>>>     predictions=pre)
>>>
>>> # now train SVM with selected features
>>> print "Evaluating Linear SVM classifier with SMLR's features..."
>>>
>>> keepInd = (N.abs(smlr.weights).mean(axis=1)!=0)
>>> newtrainpat = trainpat.selectFeatures(keepInd, sort=False)
>>> newtestpat = testpat.selectFeatures(keepInd, sort=False)
>>>
>>> # train with the known points
>>> lsvm.train(newtrainpat)
>>>
>>> # run the predictions on the test values
>>> pre = lsvm.predict(newtestpat.samples)
>>>
>>> # calculate the confusion matrix
>>> lsvm_confusion_sparse = ConfusionMatrix(
>>>     labels=newtrainpat.uniquelabels, targets=newtestpat.labels,
>>>     predictions=pre)
>>>
>>>
>>> print "SMLR Percent Correct:\t%g%% (Retained %d/%d features)" % \
>>>     (smlr_confusion.percentCorrect,
>>>      (smlr.weights!=0).sum(), N.prod(smlr.weights.shape))
>>> print "linear-SVM Percent Correct:\t%g%%" % \
>>>     (lsvm_confusion.percentCorrect)
>>> print "linear-SVM Percent Correct (with %d features from SMLR):\t%g%%" % \
>>>     (keepInd.sum(), lsvm_confusion_sparse.percentCorrect)
```

See Also:

The full source code of this example is included in the PyMVPA source distribution (*doc/examples/smlr.py*).

9.3.3 Classifier Sweep

This examples shows a test of various classifiers on different datasets.

```
>>> from mvpa.suite import *
>>>
>>> # no MVPA warnings during whole testsuite
>>> warning.handlers = []
>>>
>>> def main():
>>>
>>>     # fix seed or set to None for new each time
>>>     N.random.seed(44)
>>>
>>>
>>>     # Load Haxby dataset example
>>>     haxby1path = 'data'
>>>     attrs = SampleAttributes(os.path.join(haxby1path,
>>>                                           'attributes_literal.txt'))
>>>     haxby8 = NiftiDataset(samples=os.path.join(haxby1path,
>>>                                                'bold.nii.gz'),
>>>                         labels=attrs.labels,
>>>                         labels_map=True,
>>>                         chunks=attrs.chunks,
>>>                         mask=os.path.join(haxby1path, 'mask.nii.gz'),
>>>                         dtype=N.float32)
>>>
>>>     # preprocess slightly
>>>     rest_label = haxby8.labels_map['rest']
>>>     detrend(haxby8, perchunk=True, model='linear')
>>>     zscore(haxby8, perchunk=True, baselinelabels=[rest_label],
>>>           targetdtype='float32')
>>>     haxby8_no0 = haxby8.selectSamples(haxby8.labels != rest_label)
>>>
>>>     dummy2 = normalFeatureDataset(perlabel=30, nlabels=2,
>>>                                   nfeatures=100,
>>>                                   nchunks=6, nonbogus_features=[11, 10],
>>>                                   snr=3.0)
>>>
>>>     for (dataset, datasetdescr), clfs_ in \
>>>         [
>>>         ((dummy2,
>>>           "Dummy 2-class univariate with 2 useful features out of 100"),
>>>           clfs[:]),
>>>         ((pureMultivariateSignal(8, 3),
>>>           "Dummy XOR-pattern"),
>>>           clfs['non-linear']),
>>>         ((haxby8_no0,
>>>           "Haxby 8-cat subject 1"),
>>>           clfs['multiclass']),
>>>         ]:
>>>         print "%s\n %s" % (datasetdescr, dataset.summary(idhash=False))
>>>         print " Classifier                            " \
>>>               "%corr  #features\t train predict  full"
>>>         for clf in clfs_:
>>>             print "  %-40s: " % clf.descr,
>>>             # Lets do splits/train/predict explicitly so we could track
>>>             # timing otherwise could be just
>>>             #cv = CrossValidatedTransferError(
>>>             #        TransferError(clf),
>>>             #        NFoldSplitter(),
>>>             #        enable_states=['confusion'])
>>>             #error = cv(dataset)
>>>             #print cv.confusion
>>>
>>>             # to report transfer error
```

```
>>>              confusion = ConfusionMatrix(labels_map=dataset.labels_map)
>>>              times = []
>>>              nf = []
>>>              t0 = time.time()
>>>              clf.states.enable('feature_ids')
>>>              for nfold, (training_ds, validation_ds) in \
>>>                      enumerate(NFoldSplitter()(dataset)):
>>>                  clf.train(training_ds)
>>>                  nf.append(len(clf.feature_ids))
>>>                  if nf[-1] == 0:
>>>                      break
>>>                  predictions = clf.predict(validation_ds.samples)
>>>                  confusion.add(validation_ds.labels, predictions)
>>>                  times.append([clf.training_time, clf.predicting_time])
>>>              if nf[-1] == 0:
>>>                  print "no features were selected. skipped"
>>>                  continue
>>>              tfull = time.time() - t0
>>>              times = N.mean(times, axis=0)
>>>              nf = N.mean(nf)
>>>              # print "\n", confusion
>>>              print "%5.1f%%   %-4d\t %.2fs  %.2fs   %.2fs" % \
>>>                      (confusion.percentCorrect, nf, times[0], times[1], tfull)
>>>
>>>
>>> if __name__ == "__main__":
>>>     main()
```

See Also:

The full source code of this example is included in the PyMVPA source distribution (*doc/examples/clfs_examples.py*).

9.3.4 Minimal Searchlight Example

The term Searchlight refers to an algorithm that runs a scalar DatasetMeasure on all possible spheres of a certain size within a dataset (that provides information about distances between feature locations). The measure typically computed is a cross-validated transfer error (see *CrossValidatedTransferError*). The idea to use a searchlight as a sensitivity analyzer on fMRI datasets stems from *Kriegeskorte et al. (2006)*.

A searchlight analysis is can be easily performed. This examples shows a minimal draft of a complete analysis.

First import a necessary pieces of PyMVPA – this time each bit individually.

```
>>> from mvpa.datasets.masked import MaskedDataset
>>> from mvpa.datasets.splitter import OddEvenSplitter
>>> from mvpa.clfs.svm import LinearCSVMC
>>> from mvpa.clfs.transerror import TransferError
>>> from mvpa.algorithms.cvtranserror import CrossValidatedTransferError
>>> from mvpa.measures.searchlight import Searchlight
>>> from mvpa.misc.data_generators import normalFeatureDataset
>>>
```

For the sake of simplicity, let's use a small artificial dataset.

```
>>> # overcomplicated way to generate an example dataset
>>> ds = normalFeatureDataset(perlabel=10, nlabels=2, nchunks=2,
>>>                           nfeatures=10, nonbogus_features=[3, 7],
>>>                           snr=5.0)
>>> dataset = MaskedDataset(samples=ds.samples, labels=ds.labels,
```

```
>>>                            chunks=ds.chunks)
>>>
```

Now it only takes three lines for a searchlight analysis.

```
>>> # setup measure to be computed in each sphere (cross-validated
>>> # generalization error on odd/even splits)
>>> cv = CrossValidatedTransferError(
>>>         TransferError(LinearCSVMC()),
>>>         OddEvenSplitter())
>>>
>>> # setup searchlight with 5 mm radius and measure configured above
>>> sl = Searchlight(cv, radius=5)
>>>
>>> # run searchlight on dataset
>>> sl_map = sl(dataset)
>>>
>>> print 'Best performing sphere error:', max(sl_map)
>>>
```

If this analysis is done on a fMRI dataset using *NiftiDataset* the resulting searchlight map (*sl_map*) can be mapped back into the original dataspace and viewed as a brain overlay. *Another example* shows a typical application of this algorithm.

See Also:

The full source code of this example is included in the PyMVPA source distribution (*doc/examples/searchlight_minimal.py*).

9.3.5 Searchlight on fMRI data

The example shows how to run a searchlight analysis on the example fMRI dataset that is shipped with PyMVPA.

As always, we first have to import PyMVPA.

```
>>> from mvpa.suite import *
>>>
```

As searchlight analyses are usually quite expensive in term of computational ressources, we are going to enable some progress output, to entertain us while we are waiting.

```
>>> # enable debug output for searchlight call
>>> if __debug__:
>>>     debug.active += ["SLC"]
>>>
```

The next section simply loads the example dataset and performs some standard preprocessing steps on it.

```
>>> #
>>> # load PyMVPA example dataset
>>> #
>>> attr = SampleAttributes('data/attributes.txt')
>>> dataset = NiftiDataset(samples='data/bold.nii.gz',
>>>                        labels=attr.labels,
>>>                        chunks=attr.chunks,
>>>                        mask='data/mask.nii.gz')
>>>
>>> #
>>> # preprocessing
>>> #
>>>
```

```
>>> # do chunkswise linear detrending on dataset
>>> detrend(dataset, perchunk=True, model='linear')
>>>
>>> # only use 'rest', 'house' and 'scrambled' samples from dataset
>>> dataset = dataset.selectSamples(
>>>                 N.array([ l in [0,2,6] for l in dataset.labels],
>>>                 dtype='bool'))
>>>
>>> # zscore dataset relative to baseline ('rest') mean
>>> zscore(dataset, perchunk=True, baselinelabels=[0], targetdtype='float32')
>>>
>>> # remove baseline samples from dataset for final analysis
>>> dataset = dataset.selectSamples(N.array([l != 0 for l in dataset.labels],
>>>                                     dtype='bool'))
>>>
```

But now for the interesting part: Next we define the measure that shall be computed for each sphere. Theoretically, this can be anything, but here we choose to compute a full leave-one-out cross-validation using a linear Nu-SVM classifier.

```
>>> #
>>> # Run Searchlight
>>> #
>>>
>>> # choose classifier
>>> clf = LinearNuSVMC()
>>>
>>> # setup measure to be computed by Searchlight
>>> # cross-validated mean transfer using an odd-even dataset splitter
>>> cv = CrossValidatedTransferError(TransferError(clf),
>>>                                 NFoldSplitter())
>>>
```

Finally, we run the searchlight analysis for three different radii, each time computing an error for each sphere. To achieve this, we simply use the Searchlight class, which takes any *processing object* and a radius as arguments. The *processing object* has to compute the intended measure, when called with a dataset. The Searchlight object will do nothing more than generating small datasets for each sphere, feeding it to the processing object and storing its result.

After the errors are computed for all spheres, the resulting vector is then mapped back into the original fMRI dataspace and plotted.

```
>>> # setup plotting
>>> fig = 0
>>> P.figure(figsize=(12,4))
>>>
>>>
>>> for radius in [1,5,10]:
>>>     # tell which one we are doing
>>>     print "Running searchlight with radius: %i ..." % (radius)
>>>
>>>     # setup Searchlight with a custom radius
>>>     # radius has to be in the same unit as the nifti file's pixdim
>>>     # property.
>>>     sl = Searchlight(cv, radius=radius)
>>>
>>>     # run searchlight on example dataset and retrieve error map
>>>     sl_map = sl(dataset)
>>>
>>>     # map sensitivity map into original dataspace
>>>     orig_sl_map = dataset.mapReverse(N.array(sl_map))
>>>     masked_orig_sl_map = N.ma.masked_array(orig_sl_map,
```

```
>>>                                                      mask=orig_sl_map == 0)
>>>
>>>     # make a new subplot for each classifier
>>>     fig += 1
>>>     P.subplot(1,3,fig)
>>>
>>>     P.title('Radius %i' % radius)
>>>
>>>     P.imshow(masked_orig_sl_map[0],
>>>             interpolation='nearest',
>>>             aspect=1.25,
>>>             cmap=P.cm.autumn)
>>>     P.clim(0.5, 0.65)
>>>     P.colorbar(shrink=0.6)
>>>
>>>
>>> if cfg.getboolean('examples', 'interactive', True):
>>>     # show all the cool figures
>>>     P.show()
```

See Also:

The full source code of this example is included in the PyMVPA source distribution
(*doc/examples/searchlight_2d.py*).

9.3.6 Sensitivity Measure

Run some basic and meta sensitivity measures on the example fMRI dataset that comes with PyMVPA and plot the
computed featurewise measures for each. The generated figure shows sensitivity maps computed by six sensitivity
analyzers.

We start by loading PyMVPA and the example fMRI dataset.

```
>>> from mvpa.suite import *
>>>
>>> # load PyMVPA example dataset
>>> attr = SampleAttributes('data/attributes.txt')
>>> dataset = NiftiDataset(samples='data/bold.nii.gz',
>>>                        labels=attr.labels,
>>>                        chunks=attr.chunks,
>>>                        mask='data/mask.nii.gz')
>>>
```

As with classifiers it is easy to define a bunch of sensitivity analyzers. It is usually possible to simply call *getSensitivityAnalyzer()* on any classifier to get an instance of an appropriate sensitivity analyzer that uses this particular
classifier to compute and extract sensitivity scores.

```
>>> # define sensitivity analyzer
>>> sensanas = {
>>>     'a) ANOVA': OneWayAnova(transformer=N.abs),
>>>     'b) Linear SVM weights': LinearNuSVMC().getSensitivityAnalyzer(
>>>                                         transformer=N.abs),
>>>     'c) I-RELIEF': IterativeRelief(transformer=N.abs),
>>>     'd) Splitting ANOVA (odd-even)':
>>>         SplitFeaturewiseMeasure(OneWayAnova(transformer=N.abs),
>>>                                 OddEvenSplitter()),
>>>     'e) Splitting SVM (odd-even)':
>>>         SplitFeaturewiseMeasure(
>>>             LinearNuSVMC().getSensitivityAnalyzer(transformer=N.abs),
>>>                         OddEvenSplitter()),
>>>     'f) I-RELIEF Online':
```

```
>>>           IterativeReliefOnline(transformer=N.abs),
>>>       'g) Splitting ANOVA (nfold)':
>>>           SplitFeaturewiseMeasure(OneWayAnova(transformer=N.abs),
>>>                                   NFoldSplitter()),
>>>       'h) Splitting SVM (nfold)':
>>>           SplitFeaturewiseMeasure(
>>>             LinearNuSVMC().getSensitivityAnalyzer(transformer=N.abs),
>>>                             NFoldSplitter()),
>>>           }
>>>
```

Now, we are performing some a more or less standard preprocessing steps: detrending, selecting a subset of the experimental conditions, normalization of each feature to a standard mean and variance.

```
>>> # do chunkswise linear detrending on dataset
>>> detrend(dataset, perchunk=True, model='linear')
>>>
>>> # only use 'rest', 'shoe' and 'bottle' samples from dataset
>>> dataset = dataset.selectSamples(
>>>                 N.array([ l in [0,3,7] for l in dataset.labels],
>>>                 dtype='bool'))
>>>
>>> # zscore dataset relative to baseline ('rest') mean
>>> zscore(dataset, perchunk=True, baselinelabels=[0], targetdtype='float32')
>>>
>>> # remove baseline samples from dataset for final analysis
>>> dataset = dataset.selectSamples(N.array([l != 0 for l in dataset.labels],
>>>                                 dtype='bool'))
>>>
```

Finally, we will loop over all defined analyzers and let them compute the sensitivity scores. The resulting vectors are then mapped back into the dataspace of the original fMRI samples, which are then plotted.

```
>>> fig = 0
>>> P.figure(figsize=(14,8))
>>>
>>> keys = sensanas.keys()
>>> keys.sort()
>>>
>>> for s in keys:
>>>     # tell which one we are doing
>>>     print "Running %s ..." % (s)
>>>
>>>     # compute sensitivies
>>>     # I-RELIEF assigns zeros, which corrupts voxel masking for pylab's
>>>     # imshow, so adding some epsilon :)
>>>     smap = sensanas[s](dataset)+0.00001
>>>
>>>     # map sensitivity map into original dataspace
>>>     orig_smap = dataset.mapReverse(smap)
>>>     masked_orig_smap = N.ma.masked_array(orig_smap, mask=orig_smap == 0)
>>>
>>>     # make a new subplot for each classifier
>>>     fig += 1
>>>     P.subplot(3,3,fig)
>>>
>>>     P.title(s)
>>>
>>>     P.imshow(masked_orig_smap[0],
>>>             interpolation='nearest',
>>>             aspect=1.25,
>>>             cmap=P.cm.autumn)
```

```
>>>
>>>        # uniform scaling per base sensitivity analyzer
>>>        if s.count('ANOVA'):
>>>            P.clim(0, 0.4)
>>>        elif s.count('SVM'):
>>>            P.clim(0, 0.055)
>>>        else:
>>>            pass
>>>
>>>        P.colorbar(shrink=0.6)
>>>
>>> if cfg.getboolean('examples', 'interactive', True):
>>>        # show all the cool figures
>>>        P.show()
>>>
```

Output of the example analysis:
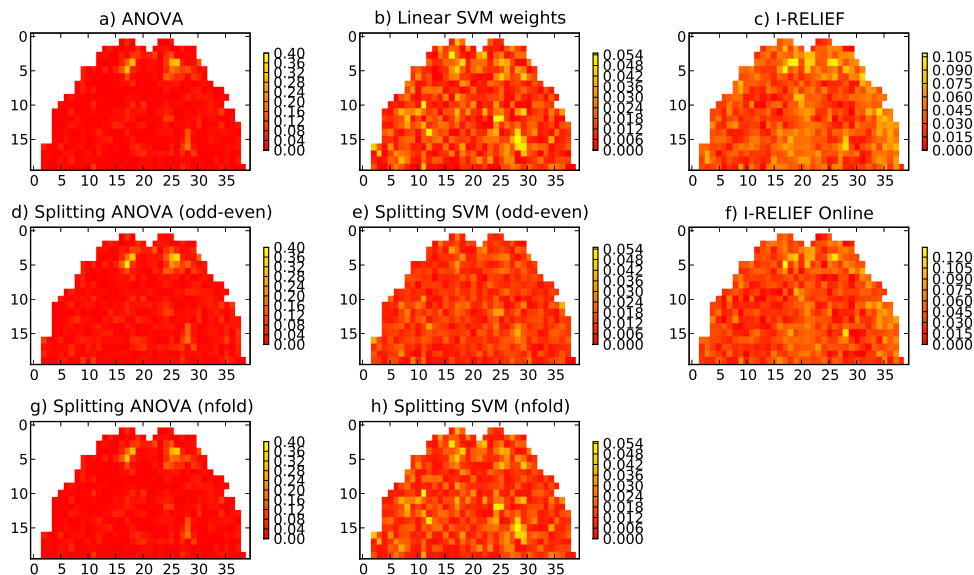
See Also:

The full source code of this example is included in the PyMVPA source distribution (*doc/examples/sensanas.py*).

9.3.7 Classification of SVD-mapped Datasets

Demonstrate the usage of a dataset mapper performing data projection onto singular value components within a cross-validation – for *any* clasifier.

```
>>> from mvpa.suite import *
>>>
>>> if __debug__:
>>>      debug.active += ["CROSSC"]
>>>
>>> #
>>> # load PyMVPA example dataset
>>> #
>>> attr = SampleAttributes('data/attributes.txt')
>>> dataset = NiftiDataset(samples='data/bold.nii.gz',
```

```
>>>                         labels=attr.labels,
>>>                         chunks=attr.chunks,
>>>                         mask='data/mask.nii.gz')
>>>
>>> #
>>> # preprocessing
>>> #
>>>
>>> # do chunkswise linear detrending on dataset
>>> detrend(dataset, perchunk=True, model='linear')
>>>
>>> # only use 'rest', 'cats' and 'scissors' samples from dataset
>>> dataset = dataset.selectSamples(
>>>                 N.array([ l in [0,4,5] for l in dataset.labels],
>>>                 dtype='bool'))
>>>
>>> # zscore dataset relative to baseline ('rest') mean
>>> zscore(dataset, perchunk=True, baselinelabels=[0], targetdtype='float32')
>>>
>>> # remove baseline samples from dataset for final analysis
>>> dataset = dataset.selectSamples(N.array([l != 0 for l in dataset.labels],
>>>                                   dtype='bool'))
>>> print dataset
>>>
>>> # Specify the base classifier to be used
>>> # To parametrize the classifier to be used
>>> #   Clf = lambda *args:LinearCSVMC(C=-10, *args)
>>> # Just to assign a particular classifier class
>>> Clf = LinearCSVMC
>>>
>>> # define some classifiers: a simple one and several classifiers with
>>> # built-in SVDs
>>> clfs = [('All orig.\nfeatures (%i)' % dataset.nfeatures, Clf()),
>>>         ('All Comps\n(%i)' % (dataset.nsamples \
>>>                 - (dataset.nsamples / len(dataset.uniquechunks)),),
>>>                     MappedClassifier(Clf(), SVDMapper())),
>>>         ('First 5\nComp.', MappedClassifier(Clf(),
>>>                     SVDMapper(selector=range(5)))),
>>>         ('First 30\nComp.', MappedClassifier(Clf(),
>>>                     SVDMapper(selector=range(30)))),
>>>         ('Comp.\n6-30', MappedClassifier(Clf(),
>>>                     SVDMapper(selector=range(5,30))))]
>>>
>>>
>>> # run and visualize in barplot
>>> results = []
>>> labels = []
>>>
>>> for desc, clf in clfs:
>>>     print desc
>>>     cv = CrossValidatedTransferError(
>>>             TransferError(clf),
>>>             NFoldSplitter(),
>>>             enable_states=['results'])
>>>     cv(dataset)
>>>
>>>     results.append(cv.results)
>>>     labels.append(desc)
>>>
>>> plotBars(results, labels=labels,
>>>         title='Linear C-SVM classification (cats vs. scissors)',
>>>         ylabel='Mean classification error (N-1 cross-validation, 12-fold)',
```

```
>>>              distance=0.5)
>>>
>>> if cfg.getboolean('examples', 'interactive', True):
>>>     P.show()
>>>
```
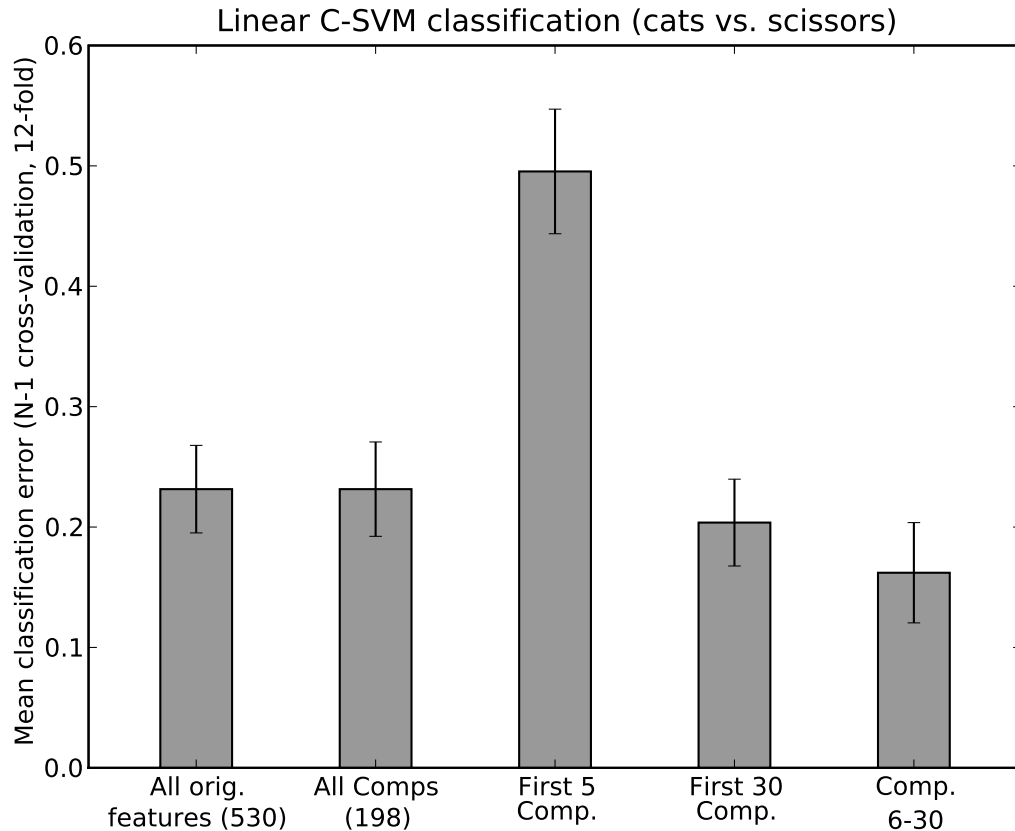
Output of the example analysis:

Linear C-SVM classification (cats vs. scissors)

See Also:

The full source code of this example is included in the PyMVPA source distribution (*doc/examples/svdclf.py*).

9.3.8 Monte-Carlo testing of Classifier-based Analyses

It is often desirable to be able to make statements like *"Performance is significantly above chance-level"*. PyMVPA supports *NULL* (aka *H0*) hypothesis testing for *transfer errors* and all *dataset measures*. In both cases the object computing the measure or transfer error takes an optional constructor argument *null_dist*. The value of this argument is an instance of some **NullDist_** estimator. If *NULL* distribution is luckily a-priori known, it is possible to reuse any distribution specified in *scipy.stats* module. If the parameters of the distribution are known, such distribution instance can be used to initialize FixedNullDist instance to be specified in *null_dist* parameter.

However, as with other applications of statistics in classifier-based analyses there is the problem that we do not know the distribution of a variable like error or performance under the *NULL* hypothesis to assign the adored p-values, i.e. the probability of a result given that there is no signal. Even worse, the chance-level or guess probability of a classifier depends on the content of a validation dataset, e.g. balanced or unbalanced number of samples per label and total number of labels).

One approach to deal with this situation is to estimate the *NULL* distribution. A generic way to do this are permutation tests (aka *Monte Carlo*, *Nichols et al. (2006)*). Then *NULL* distribution is estimated by computing some measure multiple times using datasets with no relevant signal in them. These datasets are generated by

permuting the labels of all samples in the training dataset each time the measure is computed, and therefore randomizing/removing any possible relevant information.

Given the measures computed using the permuted datasets one can now determine the probability of the empirical measure (i.e. the one computed from the original training dataset) under the *no signal* condition. This is simply the fraction of measures from the permutation runs that is larger or smaller than the emprical (depending on whether on is looking at performances or errors).

If the family of the distribution is known (e.g. Gaussian/Normal) and provided in *dist_class* parameter of MC-NullDist, then permutation tests done by MCNullDist allow to determine the distribution parameters. Under strong assumption of Gaussian distribution, 20-30 permutations should be sufficient to get sensible estimates of the distribution parameters. If no distribution family can be assumed, with a larger number of permutations, derivation of CDF out of population is possible with Nonparametric probability function (which is the default value of *dist_class* for MCNullDist). If *null_dist* is provided, the respective `TransferError` or `DatasetMeasure` instance will automatically use it to estimate the *NULL* distribution and store the associated *p*-values in a state variable named *null_prob*.

```
>>> # lazy import
>>> from mvpa.suite import *
>>>
>>> # some example data with signal
>>> train = normalFeatureDataset(perlabel=50, nlabels=2, nfeatures=3,
>>>                              nonbogus_features=[0,1], snr=3, nchunks=1)
>>>
>>> # define class to estimate NULL distribution of errors
>>> # use left tail of the distribution since we use MeanMatchFx as error
>>> # function and lower is better
>>> # in a real analysis the number of permutations should be MUCH larger
>>> terr = TransferError(clf=SMLR(),
>>>                     null_dist=MCNullDist(permutations=10,
>>>                                          tail='left'))
>>>
>>> # compute classifier error on training dataset (should be low :)
>>> err = terr(train, train)
>>> print 'Error on training set:', err
>>>
>>> # check that the result is highly significant since we know that the
>>> # data has signal
>>> print 'Corresponding p-value: 'terr.null_prob
```

See Also:

The full source code of this example is included in the PyMVPA source distribution (*doc/examples/permutation_test.py*).

9.3.9 Determine the Distribution of some Variable

This is an example demonstrating discovery of the distribution facility.

```
>>> from mvpa.suite import *
>>>
>>> verbose.level = 2
>>> if __debug__:
>>>     # report useful debug information for the example
>>>     debug.active += ['STAT', 'STAT_']
>>>
>>> #
>>> # Figure for just normal distribution
>>> #
>>>
>>> # generate random signal from normal distribution
```

```
>>> verbose(1, "Random signal with normal distribution")
>>> data = N.random.normal(size=(1000,1))
>>>
>>> # find matching distributions
>>> # NOTE: since kstest is broken in older versions of scipy
>>> #       p-roc testing is done here, which aims to minimize
>>> #       false positives/negatives while doing H0-testing
>>> test = 'p-roc'
>>> figsize = (15,10)
>>> verbose(1, "Find matching datasets")
>>> matches = matchDistribution(data, test=test, p=0.05)
>>>
>>> P.figure(figsize=figsize)
>>> P.subplot(2,1,1)
>>> plotDistributionMatches(data, matches, legend=1, nbest=5)
>>> P.title('Normal: 5 best distributions')
>>>
>>> P.subplot(2,1,2)
>>> plotDistributionMatches(data, matches, nbest=5, p=0.05,
>>>                         tail='any', legend=4)
>>> P.title('Accept regions for two-tailed test')
>>>
>>> #
>>> # Figure for fMRI data sample we have
>>> #
>>> verbose(1, "Load sample fMRI dataset")
>>> attr = SampleAttributes('data/attributes.txt')
>>> dataset = NiftiDataset(samples='data/bold.nii.gz',
>>>                        labels=attr.labels,
>>>                        chunks=attr.chunks,
>>>                        mask='data/mask.nii.gz')
>>> # select random voxel
>>> dataset = dataset.selectFeatures(
>>>              [int(N.random.uniform()*dataset.nfeatures)])
>>>
>>> verbose(2, "Minimal preprocessing to remove the bias per each voxel")
>>> detrend(dataset, perchunk=True, model='linear')
>>> zscore(dataset, perchunk=True, baselinelabels=[0],
>>>        targetdtype='float32')
>>>
>>> # on all voxels at once, just for the sake of visualization
>>> data = dataset.samples.ravel()
>>> verbose(2, "Find matching distribution")
>>> matches = matchDistribution(data, test=test, p=0.05)
>>>
>>> P.figure(figsize=figsize)
>>> P.subplot(2,1,1)
>>> plotDistributionMatches(data, matches, legend=1, nbest=5)
>>> P.title('Random voxel: 5 best distributions')
>>>
>>> P.subplot(2,1,2)
>>> plotDistributionMatches(data, matches, nbest=5, p=0.05,
>>>                         tail='any', legend=4)
>>> P.title('Accept regions for two-tailed test')
>>>
>>> if cfg.getboolean('examples', 'interactive', True):
>>>     # show the cool figure
>>>     P.show()
>>>
```

Example output for a random voxel is
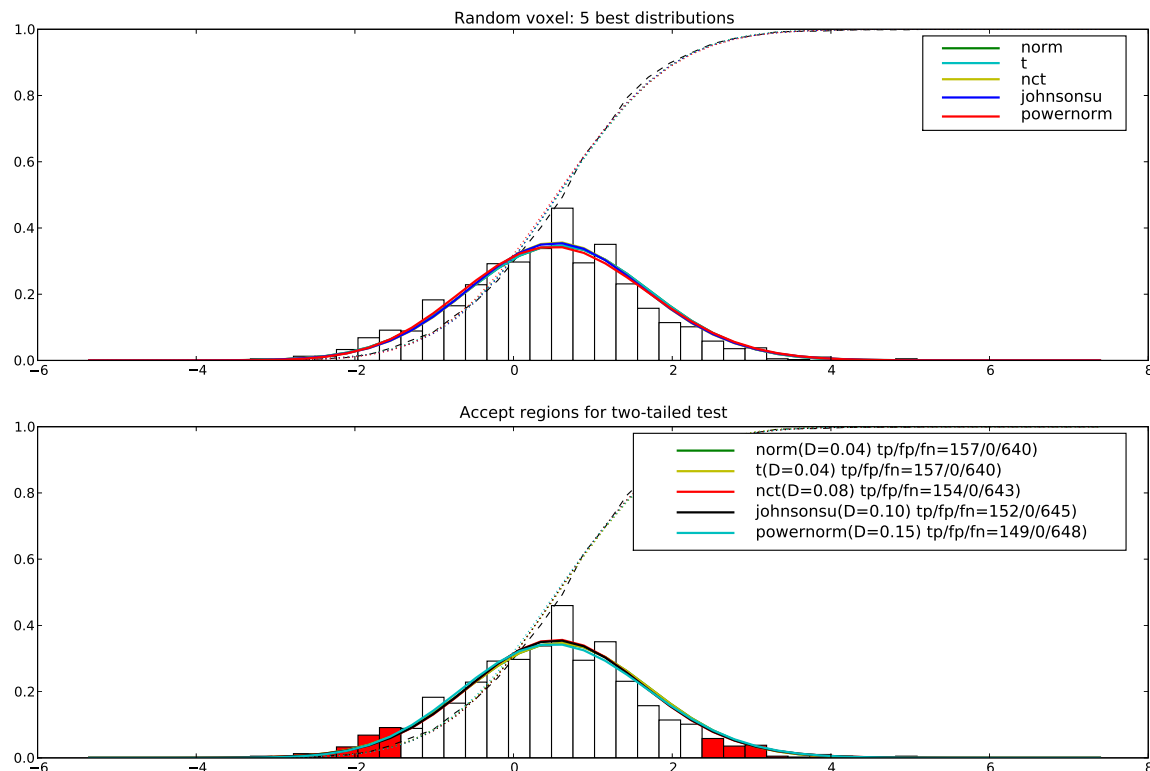
See Also:

The full source code of this example is included in the PyMVPA source distribution (*doc/examples/match_distribution.py*).

9.4 Visualization

9.4.1 ERP/ERF-Plots

Example demonstrating an ERP-style plots. Actually, this code can be used to plot various time-locked data types. This example uses MEG data and therefore generates an ERF-plot.

```
>>> from mvpa.suite import *
>>>
>>> # load data
>>> meg = TuebingenMEG(os.path.join('data', 'tueb_meg.dat.gz'))
>>>
>>>
>>> # Define plots for easy feeding into plotERP
>>> plots = []
>>> colors = ['r', 'b', 'g']
>>>
>>> # figure out pre-stimulus onset interval
>>> t0 = -meg.timepoints[0]
>>>
>>> plots = [ {'label' : meg.channelids[i],
>>>            'color' : colors[i],
>>>            'data' : meg.data[:, i, :]}
>>>           for i in xrange(len(meg.channelids)) ]
>>>
>>> # Common arguments for all plots
>>> cargs = {
```

```
>>>     'SR' : meg.samplingrate,
>>>     'pre_onset' : t0,
>>>     # Plot only 50ms before and 100ms after the onset since we have
>>>     # just few trials
>>>     'pre' : 0.05, 'post' : 0.1,
>>>     # Plot all 'errors' in different degrees of shadings
>>>     'errtype' : ['ste', 'ci95', 'std'],
>>>     # Set to None if legend manages to obscure the plot
>>>     'legend' : 'best',
>>>     'alinewidth' : 1  # assume that we like thin lines
>>>     }
>>>
>>> # Create a new figure
>>> fig = P.figure(figsize=(12, 8))
>>>
>>> # Following plots are plotted inverted (negative up) for the
>>> # demonstration of this capability and elderly convention for ERP
>>> # plots. That is controlled with ymult (negative gives negative up)
>>>
>>>
>>> # Plot MEG sensors
>>>
>>> # frame_on=False guarantees abent outside rectangular axis with
>>> # labels. plotERP recreates its own axes centered at (0,0)
>>> ax = fig.add_subplot(2, 1, 1, frame_on=False)
>>> plotERPs(plots[:2], ylabel='$pT$', ymult=-1e12, ax=ax, **cargs)
>>>
>>> # Plot EEG sensor
>>> ax = fig.add_subplot(2, 1, 2, frame_on=False)
>>> plotERPs(plots[2:3], ax=ax, ymult=-1e6, **cargs)
>>>
>>> # Additional example: plotting a single ERP on an existing plot
>>> # without drawing axis:
>>> #
>>> # plotERP(data=meg.data[:, 0, :], SR=meg.samplingrate, pre=pre,
>>> #         pre_mean=pre, errtype=errtype, ymult=-1.0)
>>>
>>> if cfg.getboolean('examples', 'interactive', True):
>>>     # show all the cool figures
>>>     P.show()
>>>
```

The ouput of the provided example is presented below. It is not a very fascinating one due to the limited number of samples provided in the dataset shipped within the toolbox.
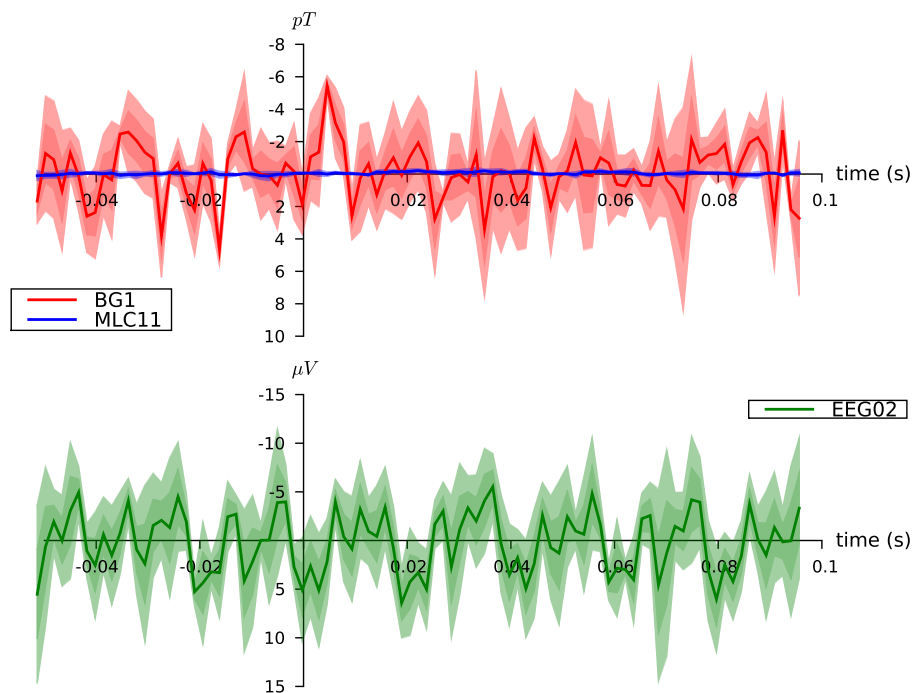
See Also:

The full source code of this example is included in the PyMVPA source distribution (*doc/examples/erp_plot.py*).

9.4.2 Simple Plotting of Classifier Behavior

This example runs a number of classifiers on a simple 2D dataset and plots the decision surface of each classifier.

First compose some sample data – no PyMVPA involved.

```
>>> import numpy as N
>>>
>>> # set up the labeled data
>>> # two skewed 2-D distributions
>>> num_dat = 200
>>> dist = 4
>>> feat_pos=N.random.randn(2, num_dat)
>>> feat_pos[0, :] *= 2.
>>> feat_pos[1, :] *= .5
>>> feat_pos[0, :] += dist
>>> feat_neg=N.random.randn(2, num_dat)
>>> feat_neg[0, :] *= .5
>>> feat_neg[1, :] *= 2.
>>> feat_neg[0, :] -= dist
>>>
>>> # set up the testing features
>>> x1 = N.linspace(-10, 10, 100)
>>> x2 = N.linspace(-10, 10, 100)
>>> x,y = N.meshgrid(x1, x2);
>>> feat_test = N.array((N.ravel(x), N.ravel(y)))
>>>
```

Now load PyMVPA and convert the data into a proper `Dataset`.

```
>>> from mvpa.suite import *
>>>
>>> # create the pymvpa dataset from the labeled features
>>> patternsPos = Dataset(samples=feat_pos.T, labels=1)
>>> patternsNeg = Dataset(samples=feat_neg.T, labels=0)
>>> patterns = patternsPos + patternsNeg
>>>
```

This demo utilizes a number of classifiers. The instantiation of a classifier involves almost no runtime costs, so it is easily possible compile a long list, if necessary.

```
>>> # set up classifiers to try out
>>> clfs = {'Ridge Regression': RidgeReg(),
>>>         'Linear SVM': LinearNuSVMC(probability=1,
>>>                     enable_states=['probabilities']),
>>>         'RBF SVM': RbfNuSVMC(probability=1,
>>>                     enable_states=['probabilities']),
>>>         'SMLR': SMLR(lm=0.01),
>>>         'Logistic Regression': PLR(criterion=0.00001),
>>>         'k-Nearest-Neighbour': kNN(k=10)}
>>>
```

Now we are ready to run the classifiers. The folowing loop trains and queries each classifier to finally generate a nice plot showing the decision surface of each individual classifier.

```
>>> # loop over classifiers and show how they do
>>> fig = 0
>>>
>>> # make a new figure
>>> P.figure(figsize=(8,12))
>>> for c in clfs:
>>>     # tell which one we are doing
>>>     print "Running %s classifier..." % (c)
>>>
>>>     # make a new subplot for each classifier
>>>     fig += 1
>>>     P.subplot(3,2,fig)
>>>
>>>     # plot the training points
>>>     P.plot(feat_pos[0, :], feat_pos[1, :], "r.")
>>>     P.plot(feat_neg[0, :], feat_neg[1, :], "b.")
>>>
>>>     # select the clasifier
>>>     clf = clfs[c]
>>>
>>>     # enable saving of the values used for the prediction
>>>     clf.states.enable('values')
>>>
>>>     # train with the known points
>>>     clf.train(patterns)
>>>
>>>     # run the predictions on the test values
>>>     pre = clf.predict(feat_test.T)
>>>
>>>     # if ridge, use the prediction, otherwise use the values
>>>     if c == 'Ridge Regression' or c == 'k-Nearest-Neighbour':
>>>         # use the prediction
>>>         res = N.asarray(pre)
>>>     elif c == 'Logistic Regression':
>>>         # get out the values used for the prediction
>>>         res = N.asarray(clf.values)
```

```
>>>     elif c == 'SMLR':
>>>         res = N.asarray(clf.values[:, 1])
>>>     else:
>>>         # get the probabilities from the svm
>>>         res = N.asarray([(q[1][1] - q[1][0] + 1) / 2
>>>                 for q in clf.probabilities])
>>>
>>>     # reshape the results
>>>     z = N.asarray(res).reshape((100, 100))
>>>
>>>     # plot the predictions
>>>     P.pcolor(x, y, z, shading='interp')
>>>     P.clim(0, 1)
>>>     P.colorbar()
>>>     P.contour(x, y, z, linewidths=1, colors='black', hold=True)
>>>
>>>     # add the title
>>>     P.title(c)
>>>
>>> if cfg.getboolean('examples', 'interactive', True):
>>>     # show all the cool figures
>>>     P.show()
```

See Also:

The full source code of this example is included in the PyMVPA source distribution (*doc/examples/pylab_2d.py*).

9.4.3 Generating Topography plots

Example demonstrating a topography plot.

```
>>> from mvpa.suite import *
>>>
>>> # Sanity check if we have griddata available
>>> externals.exists("griddata", raiseException=True)
>>>
>>> # EEG example splot
>>> P.subplot(1, 2, 1)
>>>
>>> # load the sensor information from their definition file.
>>> # This file has sensor names, as well as their 3D coordinates
>>> sensors=XAVRSensorLocations(os.path.join('data', 'xavr1010.dat'))
>>>
>>> # make up some artifical topography
>>> # 'enable' to channels, all others set to off ;-)
>>> topo = N.zeros(len(sensors.names))
>>> topo[sensors.names.index('O1')] = 1
>>> topo[sensors.names.index('F4')] = 1
>>>
>>> # plot with sensor locations shown
>>> plotHeadTopography(topo, sensors.locations(), plotsensors=True)
>>>
>>>
>>> # MEG example plot
>>> P.subplot(1, 2, 2)
>>>
>>> # load MEG sensor locations
>>> sensors=TuebingenMEGSensorLocations(
>>>             os.path.join('data', 'tueb_meg_coord.xyz'))
>>>
>>> # random values this time
```

```
>>> topo = N.random.randn(len(sensors.names))
>>>
>>> # plot without additional interpolation
>>> plotHeadTopography(topo, sensors.locations(),
>>>                     interpolation='nearest')
>>>
>>>
>>> if cfg.getboolean('examples', 'interactive', True):
>>>     # show all the cool figures
>>>     P.show()
>>>
```

The ouput of the provided example should look like
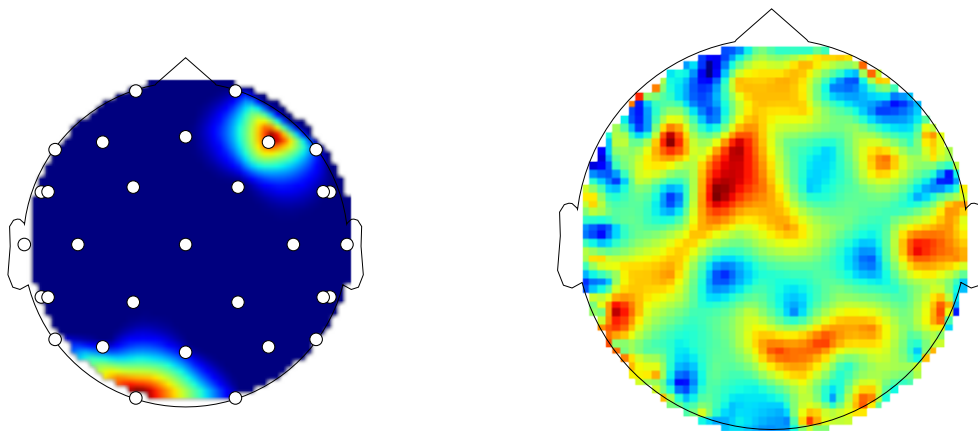
See Also:

The full source code of this example is included in the PyMVPA source distribution (*doc/examples/topo_plot.py*).

9.5 Miscellaneous

9.5.1 Kernel-Demo

This is an example demonstrating various kernel implementation in PyMVPA.

```
>>> from mvpa.suite import *
>>> from mvpa.clfs.kernel import *
>>> import pylab as P
>>>
>>>
>>> # N.random.seed(1)
>>> data = N.random.rand(4, 2)
>>>
>>> for kernel_class, kernel_args in (
>>>     (KernelConstant, {'sigma_0':1.0}),
>>>     (KernelConstant, {'sigma_0':1.0}),
>>>     (KernelLinear, {'Sigma_p':N.eye(data.shape[1])}),
>>>     (KernelLinear, {'Sigma_p':N.ones(data.shape[1])}),
```

```
>>>        (KernelLinear, {'Sigma_p':2.0}),
>>>        (KernelLinear, {}),
>>>        (KernelExponential, {}),
>>>        (KernelSquaredExponential, {}),
>>>        (KernelMatern_3_2, {}),
>>>        (KernelMatern_5_2, {}),
>>>        (KernelRationalQuadratic, {}),
>>>        ):
>>>        kernel = kernel_class(**kernel_args)
>>>        print kernel
>>>        result = kernel.compute(data)
>>>
>>> # In the following we draw some 2D functions at random from the
>>> # distribution N(O,kernel) defined by each available kernel and
>>> # plot them. These plots shows the flexibility of a given kernel
>>> # (with default parameters) when doing interpolation. The choice
>>> # of a kernel defines a prior probability over the function space
>>> # used for regression/classfication with GPR/GPC.
>>> count = 1
>>> for k in kernel_dictionary.keys():
>>>        P.subplot(3,4,count)
>>>        # X = N.random.rand(size)*12.0-6.0
>>>        # X.sort()
>>>        X = N.arange(-1,1,.02)
>>>        X = X[:,N.newaxis]
>>>        ker = kernel_dictionary[k]()
>>>        K = ker.compute(X,X)
>>>        for i in range(10):
>>>            f = N.random.multivariate_normal(N.zeros(X.shape[0]),K)
>>>            P.plot(X[:,0],f,"b-")
>>>
>>>        P.title(k)
>>>        P.axis('tight')
>>>        count += 1
>>>
>>> if cfg.getboolean('examples', 'interactive', True):
>>>     # show all the cool figures
>>>     P.show()
```

See Also:

The full source code of this example is included in the PyMVPA source distribution (*doc/examples/kerneldemo.py*).

9.5.2 Curve-Fitting

An example showing how to fit an HRF model to noisy peristimulus time-series data.

```
>>> import numpy as N
>>> import pylab as P
>>>
>>> from mvpa.misc.plot import errLinePlot
>>> from mvpa.misc.fx import singleGammaHRF, leastSqFit
>>> from mvpa import cfg
>>>
>>> # make dataset
>>> # 40 identical 'trial time courses' generated from a simple
>>> # gamma function
>>> #   time-to-peak: 6s
>>> #   FWHM: 7s
>>> #   Scaling: 1
```

```
>>> a = N.asarray([singleGammaHRF(N.arange(20), A=6, W=7, K=1)] * 40)
>>> # get closer to reality
>>> a += N.random.normal(size=a.shape)
>>>
>>>
>>> # now fit a gamma function, parameter start values:
>>> #    time-to-peak: 5s
>>> #    FWHM: 5s
>>> #    Scaling: 1
>>> fpar, succ = leastSqFit(singleGammaHRF, [5,5,1], a)
>>>
>>> # generate high-reslution curves for the 'true' time course
>>> # and the fitted one
>>> curves = [singleGammaHRF(N.linspace(0,20), 6, 7, 1),
>>>           singleGammaHRF(N.linspace(0,20), *fpar)]
>>>
>>> # plot data (with error bars) and both curves
>>> errLinePlot(a, curves=curves, linestyle='-')
>>>
>>> # add legend to plot
>>> P.legend(('original', 'fit'))
>>>
>>> if cfg.getboolean('examples', 'interactive', True):
>>>     # show the cool figure
>>>     P.show()
>>>
```

The ouput of the provided example should look like
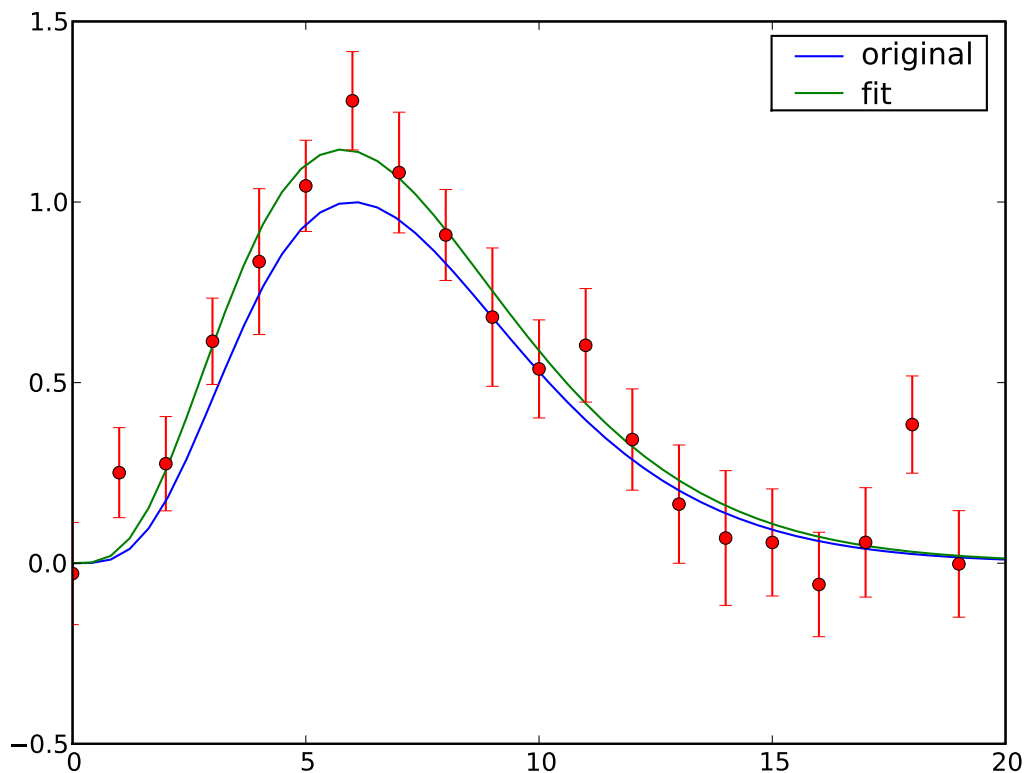
See Also:

The full source code of this example is included in the PyMVPA source distribution (*doc/examples/curvefitting.py*).

PYMVPA FOR MATLAB USERS

If you are coming from Matlab, you will soon notice a lot of similarities between Matlab and Python (besides the huge advantages of Python over Matlab). For an easy transition you might want to have a look at a basic comparison of Matlab and NumPy. It would be nice to have some guidelines on how to use PyMVPA for users who are already familiar with the Matlab MVPA toolbox. If you are using both packages and could compile a few tips, your contribution would be most welcome.

FREQUENTLY ASKED QUESTIONS

11.1 General

11.1.1 It is slooooow. What can I do?

Have you tried running the Python interpreter with *-O*? PyMVPA provides lots of debug messages with information that is computed in addition to the work that really has to be done. However, if Python is running in *optimized* mode, PyMVPA will not waste time on this and really tries to be fast.

If you are already running it optimized, then maybe you are doing something really demanding...

11.1.2 I am tired of writing these endless import blocks. Any alternative?

Sure. Instead of individually importing all pieces that are required by a script, you can import them all at once. A simple:

```
>>> import mvpa.suite as mvpa
```

makes everything directly accessible through the mvpa namespace, e.g. *mvpa.datasets.base.Dataset* becomes *mvpa.Dataset*. Really lazy people can even do:

```
>>> from mvpa.suite import *
```

However, as always there is a price to pay for this convenience. In contrast to the individual imports there is some intial performance and memory cost. In the worst case you'll get all external dependencies loaded (e.g. a full R session), just because you have them installed. Therefore, it might be better to limit this use to case where individual key presses matter and use individual imports for production scripts.

11.1.3 I feel like I want to contribute something, do you mind?

Not at all! If you think there is something that is not well explained in the documentation, send us an improvement. If you implemented a new algorithm using PyMVPA that you want to share, please share. If you have an idea for some other improvement (e.g. speed, functionality), but you have no time/cannot/do not want to implement it yourself, please post your idea to the PyMVPA mailing list.

11.1.4 I want to develop a new feature for PyMVPA. How can I do it efficiently?

The best way is to use Git for both, getting the latest code from the repository and preparing the patch. Here is a quick sketch of the workflow.

First get the latest code:

```
git clone git://git.debian.org/git/pkg-exppsy/pymvpa.git
```

This will create a new *pymvpa* subdirectory, that contains the complete repository. Enter this directory and run *gitk –all* to browse the full history and *all* branches that have ever been published.

You can run:

```
git fetch origin
```

in this directory at any time to get the latest changes from the main repository.

Next, you have to decide what you want to base your new feature on. In the simplest case this is the *master* branch (the one that contains the code that will become the next release). Creating a local branch based on the (remote) *master* branch is:

```
git checkout -b my_hack origin/master
```

Now you are ready to start hacking. You are free to use all powers of Git (and yours, of course). You can do multiple commits, fetch new stuff from the repository, and merge it into your local branch, ... To get a feeling what can be done, take a look very short description of Git or a more comprehensive Git tutorial.

When you are done with the new feature, you can prepare the patch for inclusion into PyMVPA. If you have done multiple commits you might want to squash them into a single patch containing the new feature. You can do this with *git-rebase*. In recent version *git-rebase* has an option *–interactive*, which allows you to easily pick, squash or even further edit any of the previous commits you have made. Rebase your local branch against the remote branch you started hacking on (*origin/master* in this example):

```
git rebase --interactive origin/master
```

When you are done, you can generate the final patch file:

```
git-format-patch origin/master
```

Above command will generate a file for each commit in you local branch that is not yet part of *origin/master*. The patch files can then be easily emailed.

11.1.5 The manual is quite insufficient. When will you improve it?

Writing a manual can be a tricky task if you already know the details and have to imagine what might be the most interesting information for someone who is just starting. If you feel that something is missing which has cost you some time to figure out, please drop us a note and we will add it as soon as possible. If you have developed some code snippets to demonstrate some feature or non-trivial behaviour (maybe even trivial ones, which are not as obvious as they should be), please consider sharing this snippet with us and we will put it into the example collection or the manual. Thanks!

11.2 Data import, export and storage

11.2.1 What file formats are understood by PyMVPA?

Please see the *Data Formats* section.

11.2.2 What if there is no special file format for some particular datatype?

With the `Hamster` class, PyMVPA supports storing *any* kind of serializable data into a (compressed) file (see the class documentation for a trivial usage example). The facility is particularly useful for storing any number of intermediate analysis results, e.g. for post-processing.

11.3 Data preprocessing

11.3.1 Is there an easy way to remove invariant features from a dataset?

You might have to deal with invariant features in case like an fMRI dataset, where the *brain mask* is slightly larger than the thresholded fMRI timeseries image. Such invariant features (i.e. features with zero variance) are sometime a problem, e.g. they will lead to numerical difficulties when z-scoring the features of a dataset (i.e. division by zero).

The *mvpa.datasets.miscfx* module provides a convenience function *removeInvariantFeatures()* that strips such features from a dataset.

11.3.2 How can I do *block-averaging* of my block-design fMRI dataset?

The easiest way is to use a mapper to transform/average the respective samples. Suppose you have a dataset:

```
>>> dataset = normalFeatureDataset()
>>> dataset
<Dataset / float64 100 x 4 uniq: 2 labels 5 chunks labels_mapped>
```

Averaging all samples with the same label in each chunk individually is done by applying a samples mapper to the dataset.

```
>>> from mvpa.mappers.samplegroup import SampleGroupMapper
>>> from mvpa.misc.transformers import FirstAxisMean
>>>
>>> m = SampleGroupMapper(fx=FirstAxisMean)
>>> mapped_dataset = dataset.applyMapper(samplesmapper=m)
>>> mapped_dataset
<Dataset / float64 10 x 4 uniq: 2 labels 5 chunks labels_mapped>
```

SampleGroupMapper applies a function to every group of samples in each chunk individually. Using *FirstAxisMean* as function, therefore yields one sample of each label per chunk.

11.4 Data analysis

11.4.1 How do I know which features were finally selected by a classifier doing feature selection?

All classifier possess a state variable *feature_ids*. When enable, the classifier stores the ids of all features that were finally used to train the classifier.

```
>>> clf = FeatureSelectionClassifier(
...          kNN(k=5),
...          SensitivityBasedFeatureSelection(
...              SMLRWeights(SMLR(lm=1.0), transformer=Absolute),
...              FixedNElementTailSelector(1, tail='upper', mode='select')),
...          enable_states = ['feature_ids'])
```

```
>>> clf.train(dataset)
>>> final_dataset = dataset.selectFeatures(clf.feature_ids)
>>> final_dataset
<Dataset / float64 100 x 1 uniq: 2 labels 5 chunks labels_mapped>
```

In the above code snippet a kNN classifier is defined, that performs a feature selection step prior training. Features are selected according to the absolute magnitude of the weights of a SMLR classifier trained on the data (same training data that will also go into kNN). Absolute SMLR weights are used for feature selection as large negative values also indicate important information. Finally, the classifier is configured to select the single most important feature (given the SMLR weights). After enabling the *feature_ids* state, the classifier provides the desired information, that can e.g. be applied to generate a stripped dataset for an analysis of the similarity structure.

11.4.2 How do I extract sensitivities from a classifier used within a cross-validation?

`CrossValidatedTransferError` provides an interface to access any classifier-related information: *harvest_attribs*. Harvesting the sensitivities computed by all classifiers (without recomputing them again) looks like this:

```
>>> cv = CrossValidatedTransferError(
...         TransferError(SMLR()),
...         OddEvenSplitter(),
...         harvest_attribs=\
...          ['transerror.clf.getSensitivityAnalyzer(force_training=False)()'])
>>> merror = cv(dataset)
>>> sensitivities = cv.harvested.values()[0]
>>> N.array(sensitivities).shape == (2, dataset.nfeatures)
True
```

First, we define an instance of `CrossValidatedTransferError` that uses an SMLR classifier to perform the cross-validation on odd-even splits of a dataset. The important piece is the definition of the *harvest_attribs*. It takes a list of code snippets that will be executed in the local context of the cross-validation function. The `TransferError` instance used to train and test the classifier on each split is available via *transerror*. The rest is easy: `TransferError` provides access to its classifier and any classifier can in turn generate an appropriate `Sensitivity` instance via *getSensitivityAnalyzer()*. This generator method takes additional arguments to the constructor of the `mvpa.measures.base.Sensitivity` class. In this case we want to prevent retraining the classifiers, as they will be trained anyway by the `TransferError` instance they belong to.

The return values of all code snippets defined in *harvest_attribs* are available in the *harvested* state variable. *harvested* is a dictionary where the keys are the code snippets used to compute the value. As the key in this case is pretty long, we simply take the first (and only) value from the dictionary. The value is actually a list of sensitivity vectors, one per split.

11.4.3 Can PyMVPA deal with literal class labels?

Yes and no. In general the classifiers wrapped or implemented in PyMVPA are not capable of handling literal labels, some even might require binary labels. However, PyMVPA datasets provide functionality to map any set of literal labels to a corresponding set of numerical labels. Let's take a look:

```
>>> # invent some samples (arbitrary in this example)
>>> samples = N.random.randn(3).reshape(3,1)
```

First we will construct a Dataset the usual way (3 samples with unique numerical labels, all in one chunk:

```
>>> Dataset(samples=samples, labels=range(3), chunks=1)
<Dataset / float64 3 x 1 uniq: 3 labels 1 chunks>
```

Now, we are trying to create the same dataset using literal labels:

```
>>> # now create the same dataset using literal labels
>>> ds = Dataset(samples=samples,
...              labels=['one', 'two', 'three'],
...              chunks=1)
>>> ds.labels[0]
'one'
```

This approach simply stored the literal labels in the dataset and will most likely lead to unpredictable behavior of classifiers that cannot handle them. A more flexible approach is to let the dataset map the literal labels to numerical ones:

```
>>> ds = Dataset(samples=samples,
...              labels=['one', 'two', 'three'],
...              chunks=1,
...              labels_map=True)
>>> ds
<Dataset / float64 3 x 1 uniq: 3 labels 1 chunks labels_mapped>
>>> ds.labels[0]
0
>>> for k in sorted(ds.labels_map.keys()):
...     print k, ds.labels_map[k]
one 0
three 1
two 2
```

With this approach the labels stored in the dataset are now numerical. However, the mapping between literal and numerical labels is somewhat arbitrary. If a fixed mapping is possible or intended (e.g. same mapping for multiple dataset), the mapping can be set explicitely:

```
>>> ds = Dataset(samples=samples,
...              labels=['one', 'two', 'three'],
...              chunks=1,
...              labels_map={'one': 1, 'two': 2, 'three': 3})
>>> for k in sorted(ds.labels_map.keys()):
...     print k, ds.labels_map[k]
one 1
three 3
two 2
```

PyMVPA will use the labels mapping to display literal instead of numerical labels e.g. in confusion matrices.

GLOSSARY

The literature concerning the application of multivariate pattern analysis procedures to neuroscientific datasets contains a lot of specific terms to refer to procedures or types of data, that are of particular importants. Unfortunately, sometimes various terms refer to the same contruct and even worse these terms do not necessarily match the terminology used in the machine learning literature. The foloowing glossary is an attempt to map the various terms found in the literature to the terminology used in this manual.

Block-averaging

Averaging all samples recorded during a block of continous stimulation in a block-design fMRI experiment. The rationale behind this technique is, that a averaging might lead to an improved signal-to-noise ratio. However, averaging further decreases the number of samples in a dataset, which is already very low in typical fMRI datasets, especially in comparsion to the number of features/voxels. Block-averaging might nevertheless improve the classifier performance, *if* it indeed improves signal-to-noise *and* the respective classifier benefits more from few high-quaility samples than from a larger set of lower-quality samples.

Chunk

A chunk is a group of samples. In PyMVPA chunks define *independent* groups of samples (note: the groups are independent from each other, not the samples in each particular group). This information is important in the context of a cross-validation procedure, as it is required to measured the classifier performance on independent test datasets to be able to compute unbiased generalization estimates. This is of particular importance in the case of fMRI data, where two successively recorded volumes cannot be considered as independent measurements. This is due to the significant temporal forward contamination of the hemodynamic response whos correlate is measures by the MR scanner.

Dataset

In PyMVPA a dataset is the combination of samples, their ...

Decoding

This term is usually used to refer to the application of machine learning or pattern recognition techniques to brainimaging datasets, and therefore is another term for *MVPA*. Sometimes also 'brain-reading' is used as another alternative.

Epoch

Sometimes used to refer to a group of successively aqcuired samples, and, thus, related to a *chunk*.

Example

Another term for *sample*.

Feature

This is a name for a variable in the *dataset*.

Label

A label associates each *sample* sample in the *dataset* with a certain category, experimental condition or, in case of a regression problem, with some metric variable. The label therefore defines the model that a classifier has to learn. The labels also provide the "true" model value when computing classifier errors.

MVPA

This term originally stems from the authors of the Matlab MVPA toolbox, and in that context stands for *multi-voxel pattern analysis* (see *Norman et al., 2006*). PyMVPA obviously adopted this acronym. However,

as PyMVPA is explicitly designed to operate on non-fMRI data as well, the 'voxel' term is not appropriate and therefore MVPA in this context stands for the more general term *multivariate pattern analysis*.

Processing object

Most objects dealing with data are implemented as processing objects. Such objects are instantiated *once*, with all appropriate parameters configured as desired. When created, they can be used multiple time by simply calling them with new data.

Sample

A sample a vector with observations for all *feature* variables.

Sensitivity

The sensitivity is a score assigned to a particular *feature* with respect to its impact on a classifier's decision. The sensitivity is often available from a classifier's *weight vector*. There are some more scores which are similar to a sensitivity in terms of indicating the "importance" of a particular feature – examples are a univariate *ANOVA* score or a *Noise Perturbation* measure.

Sensitivity Map

A vector of several sensitivity scores – one for each feature in a dataset.

Spatial Discriminanckonq e Map (SDM)

This is another term for a *sensitivity map*, used in e.g. *Wang et al. (2007)*.

Statistical Discrimination Map (SDM)

This is another term for a *sensitivity map*, used in e.g. *Sato et al. (2008)*.

Time-compression

This usually refers to the *block-averaging* of samples from a block-design fMRI dataset.

Weight Vector

See *sensitivity*.

REFERENCES

This list aims to be a collection of literature, that is of particular interest in the context of multivarite pattern analysis. It includes all references cited throughout this manual, but also a number of additional manuscripts containing descriptions of interesting analysis methods or fruitful experiments.

Chen, X., Pereira, F., Lee, W., Strother, S. & Mitchell, T. (2006). Exploring predictive and reproducible modeling with the single-subject FIAC dataset. *Human Brain Mapping*, *27*, 452–461.

> *This paper illustrates the necessity to consider the stability or reproducibility of a classifier's feature selection as at least equally important to it's generalization performance.*

> Keywords: `feature selection stability`

> DOI: http://dx.doi.org/10.1002/hbm.20243

Demšar, J. (2006). Statistical Comparisons of Classifiers over Multiple Data Sets. *Journal of Machine Learning Research*, *7*, 1–30.

> *This is a review of several classifier benchmark procedures.*

> URL: http://portal.acm.org/citation.cfm?id=1248548

Efron, B., Trevor, H., Johnstone, I. & Tibshirani, R. (2004). Least Angle Regression. *Annals of Statistics*, *32*, 407–499.

> Keywords: `least angle regression`, `LARS`

> DOI: http://dx.doi.org/10.1214/009053604000000067

Guyon, I. & Elisseeff, A. (2003). An Introduction to Variable and Feature Selection. *Journal of Machine Learning*, *3*, 1157–1182.

> URL: http://www.jmlr.org/papers/v3/guyon03a.html

Hanke, M., Halchenko, Y. O., Sederberg, P. B. & Hughes, J. M. The PyMVPA Manual. Available online at http://www.pymvpa.org/PyMVPA-Manual.pdf.

Hanke, M., Halchenko, Y. O., Sederberg, P. B., Hanson, S. J., Haxby, J. V. & Pollmann, S. (in press). PyMVPA: A Python toolbox for multivariate pattern analysis of fMRI data. *Neuroinformatics*.

> *Introduction into the analysis of fMRI data using PyMVPA.*

Hanson, S. J. & Halchenko, Y. O. (2008). Brain reading using full brain support vector machines for object recognition: there is no "face" identification area. *Neural Computation*, *20*, 486–503.

> Keywords: `support vector machine`, `SVM`, `recursive feature elimination`, `RFE`

> DOI: http://dx.doi.org/10.1162/neco.2007.09-06-340

Hanson, S., Matsuka, T. & Haxby, J. (2004). Combinatorial codes in ventral temporal lobe for object recognition: Haxby (2001) revisited: is there a "face" area?. *Neuroimage*, *23*, 156–166.

> DOI: http://dx.doi.org/10.1016/j.neuroimage.2004.05.020

Haxby, J., Gobbini, M., Furey, M., Ishai, A., Schouten, J. & Pietrini, P. (2001). Distributed and overlapping representations of faces and objects in ventral temporal cortex. *Science*, *293*, 2425–2430.

> Keywords: `split-correlation classifier`

> DOI: http://dx.doi.org/10.1126/science.1063736

Haynes, J. & Rees, G. (2006). Decoding mental states from brain activity in humans. *Nature Reviews Neuroscience*, *7*, 523–534.

> *Review of decoding studies, emphasizing the importance of ethical issues concerning the privacy of personal thought.*

> DOI: http://dx.doi.org/10.1038/nrn1931

Kamitani, Y. & Tong, F. (2005). Decoding the visual and subjective contents of the human brain. *Nature Neuroscience*, *8*, 679–685.

> *One of the two studies showing the possibility to read out orientation information from visual cortex.*

> DOI: http://dx.doi.org/10.1038/nn1444

Kriegeskorte, N., Goebel, R. & Bandettini, P. (2006). Information-based functional brain mapping. *Proceedings of the National Academy of Sciences of the USA*, *103*, 3863–3868.

> *Paper introducing the searchlight algorithm.*

> Keywords: `searchlight`

> DOI: http://dx.doi.org/10.1073/pnas.0600244103

Krishnapuram, B., Carin, L., Figueiredo, M. A. & Hartemink, A. J. (2005). Sparse multinomial logistic regression: fast algorithms and generalization bounds. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, *27*, 957–968.

> Keywords: `sparse multinomial logistic regression`, `SMLR`

> DOI: http://dx.doi.org/10.1109/TPAMI.2005.127

LaConte, S., Strother, S., Cherkassky, V., Anderson, J. & Hu, X. (2005). Support vector machines for temporal classification of block design fMRI data. *Neuroimage*, *26*, 317–329.

> *Comprehensive evaluation of preprocessing options with respect to SVM-classifier (and others) performance on block-design fMRI data.*

> Keywords: `SVM`

> DOI: http://dx.doi.org/10.1016/j.neuroimage.2005.01.048

Mitchell, T., Hutchinson, R., Niculescu, R. S., Pereira, F., Wang, X., Just, M. & Newman, S. (2004). Learning to Decode Cognitive States from Brain Images. *Machine Learning*, *57*, 145–175.

> DOI: http://dx.doi.org/10.1023/B:MACH.0000035475.85309.1b

Nichols, T. E. & Holmes, A. P. (2002). Nonparametric permutation tests for functional neuroimaging: a primer with examples. *Human Brain Mapping*, *15*, 1–25.

> *Overview of standard nonparametric randomization and permutation testing applied to neuroimaging data (e.g. fMRI)*

> DOI: http://dx.doi.org/10.1002/hbm.1058

Norman, K. A., Polyn, S. M., Detre, G. J. & Haxby, J. V. (2006). Beyond mind-reading: multi-voxel pattern analysis of fMRI data. *Trends in Cognitive Science*, *10*, 424–430.

> DOI: http://dx.doi.org/10.1016/j.tics.2006.07.005

O'Toole, A. J., Jiang, F., Abdi, H. & Haxby, J. V. (2005). Partially Distributed Representations of Objects and Faces in Ventral Temporal Cortex . *Journal of Cognitive Neuroscience*, *17*, 580–590.

> DOI: http://dx.doi.org/10.1162/0898929053467550

O'Toole, A. J., Jiang, F., Abdi, H., Penard, N., Dunlop, J. P. & Parent, M. A. (2007). Theoretical, statistical, and practical perspectives on pattern-based classification approaches to the analysis of functional neuroimaging data. *Journal of Cognitive Neuroscience*, *19*, 1735–1752.

> DOI: http://dx.doi.org/10.1162/jocn.2007.19.11.1735

Pessoa, L. & Padmala, S. (2007). Decoding near-threshold perception of fear from distributed single-trial brain activation. *Cerebral Cortex*, *17*, 691–701.

> *Analysis of slow event-related fMRI data using patter classification techniques.*

> DOI: http://dx.doi.org/10.1093/cercor/bhk020

Sato, J. R., Mourão-Miranda, J., Martin, M. d. G. M., Amaro, E., Morettin, P. A. & Brammer, M. J. (2008). The impact of functional connectivity changes on support vector machines mapping of fMRI data. *Journal of Neuroscience Methods*, *172*, 94–104.

> *Discussion of possible scenarios where univariate and multivariate (SVM) sensitivity maps derived from the same dataset could differ. Including the case were univariate methods would assign a substantially larger score to some features.*

> Keywords: `support vector machine`, `SVM`, `sensitivity`

> DOI: http://dx.doi.org/10.1016/j.jneumeth.2008.04.008

Vapnik, V. (1995). The Nature of Statistical Learning Theory. Springer: New York.

> Keywords: `support vector machine`, `SVM`

Wang, Z., Childress, A. R., Wang, J. & Detre, J. A. (2007). Support vector machine learning-based fMRI data group analysis. *Neuroimage*, *36*, 1139–51.

> Keywords: `support vector machine`, `SVM`, `group analysis`

> DOI: http://dx.doi.org/10.1016/j.neuroimage.2007.03.072

LICENSE

The PyMVPA package, including all examples, code snippets and attached documentation is covered by the MIT license.

```
The MIT License

Copyright (c) 2006-2008 Michael Hanke
              2007-2008 Yaroslav Halchenko

Permission is hereby granted, free of charge, to any person obtaining a copy
of this software and associated documentation files (the "Software"), to deal
in the Software without restriction, including without limitation the rights
to use, copy, modify, merge, publish, distribute, sublicense, and/or sell
copies of the Software, and to permit persons to whom the Software is
furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in
all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE
AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,
OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN
THE SOFTWARE.
```

PYMVPA DEVELOPMENT CHANGELOG

This changelog only lists rather macroscopic changes to PyMVPA. The full VCS changelog is available here:

> http://git.debian.org/?p=pkg-exppsy/pymvpa.git;a=summary

'Closes' statement IDs refer to the Debian bug tracking system and can be queried by visiting the URL:

```
http://bugs.debian.org/<bug id>
```

Unreleased changes
> Changes described here are not yet released, but available from VCS repository.
> None yet.

15.1 Releases

- 0.4.0 (Sat, 15 Nov 2008)
    - Add `Hamster`, as a simple facility to easily store any serializable objects in a compressed file and later on resurrect all of them with a single line of code.
    - SVM backend is now configurable via **MVPA_SVM_BACKEND** (libsvm or shogun).
    - Non-deterministic tests in the unittest battery are now configurable via **MVPA_TESTS_LABILE**.
    - New helper to determine and plot the best matching distribution(s) for the data (matchDistribution, plotDistributionMatches). It is WiP thus API can change in the upcoming release.
    - Simplifies API of mappers.
    - Splitters can now limit the number of splits automatically.
    - New `CombinedMapper` to map between multiple, independent dataspace and a common feature space.
    - New `ChainMapper` to create chains of mappers of abitrary lenght (e.g. to build preprocessing pipelines).
    - New `EventDataset` to rapidly extract boxcar-shaped samples from data array using a simple list of `Event` definitions.
    - Removed obsolete MetricMapper class. `Mapper` itself provides the facilities for dealing with metrics.
    - `BoxcarMapper` can now handle data with more than four dimensions/axis and also performs reverse mapping of single boxcar samples.
    - `FslEV3` can now convert EV3 files into a list of `Event` instances.
    - Results of tests for external dependencies are now stored in PyMVPA's config manager (*mvpa.cfg*) and can be stored to a file (not done automatically at the moment). This will significantly decrease the time needed to import the *mvpa* module, as it prevents the repeated and lengthy tests for working externals.

- Initial support for ROC computing and AUC as an accuracy measure.

- Weights of LARS are now available via `LARSWeights`.

- Added an initial list of MVPA-related references to the manual, tagged with keywords and comments as well is DOI or similar URL reference to the original document.

- Added initial glossary to the manual.

- New 'Module reference', as a middle-ground between manual and API reference.

- New manual section about meta-classifiers (contributed by James M. Hughes).

- New minimal example for a 'getting started' section in the manual.

- Former **MVPA_QUICKTEST** was renamed to **MVPA_TESTS_QUICK**.

- Update installation instructions for RPM-based distributions to make use of the OpenSUSE Build Service.

- Updated install instructions for several RPM-based GNU/Linux distributions.

- Switch from distutils to numpy.distutils (no change in dependencies).

- Depend on PyNIfTI >= 0.20081017.1 and gain a smaller memory footprint when accessing NIfTI files via all datasets with NIfTI support.

- Added workaround to make PyMVPA work with older Shogun releases and those from 0.6.4 on, which introduced backward-incompatible API changes.

- 0.3.1 (Sun, 14 Sep 2008)

  - New manual section about feature selection with a focus on RFE. Contributed by James M. Hughes.

  - New dataset type `ChannelDataset` for data structured in channels. Might be useful for data modalities like EEG and MEG. This dataset includes support for common preprocessing steps like resampling and baseline signal substraction.

  - Plotting of topographies on heads. Thanks to Ingo Fründ for contributing this code. Additionally, a new example shows how to do such plots.

  - New general purpose function for generating barplots and candlestick plots with error bars (`plotBars()`).

  - Dataset supports mapping of string labels onto numerical labels, removing the need to perform this mapping manually in user code. 'clfs_examples.py' is adjusted accordingly to demonstrate the new feature.

  - New `mvpa.clfs.base.Classifier.summary()` method to dump classifier settings.

  - Improved and more flexible `plotERPs()`.

  - New `IterativeRelief` sensitivity analyzer.

  - Added visualization of confusion matrices via `mvpa.clfs.transerror.ConfusionMatrix.plot()` inspired by Ingo Fründ.

  - The PyMVPA version is now globally available in `mvpa.pymvpa_version`.

  - BugFix: `TuebingenMEG` reader failed in some cases.

  - Several improvements (docs and implementation) for building PyMVPA on MacOS X.

  - New convenience accessor methods (`select()`, `where()` and `__getitem__()`) for :class'~mvpa.datasets.base.Dataset'.

  - New `mvpa.seed()` function to configure the random number generators from user code.

  - Added reader for a MEG sensor locations format (`TuebingenMEGSensorLocations`).

  - Initial model selection support for GRP (using openopt).

  - And tons of minor bugfixes, additional tests and improved documentation.

- 0.3.0 (Mon, 18 Aug 2008)

  - Import of binary EEP files (used by EEProbe) and EEPDataset class.

  - Initial version of a meta dataset class (MetaDataset). This is a container for multiple datasets, which behaves like a dataset itself.

- Regression performance is summarized now within RegressionStatistics.

- Error functions: CorrErrorPFx, RelativeRMSErrorFx.

- Measures: CorrCoef.

- Data generators: chirp, wr1996

- Few more examples: curvefitting, kerneldemo, smellit, projections

- Updated kNN classifier. kNN is now able to use custom distance function to determine that nearest neighbors. It also (re)gained the ability to do simple majority or weighted voting.

- Some initial convenience functions for plotting typical results and data exploration.

- Unified configuration handling with support for user-specific and analysis-specific config files, as well as the ability to override all config settings via environment variables. The configuration handling is used for PyMVPA internal settings, but can also be easily used for custom (user-)settings.

- Improved modularity, e.g. SciPy is not required anymore, but still very useful.

- Initial implementations of ICA and PCA mapper using functionality provided by MDP. These mappers are more or less untested and should be used with great care.

- Further improved docstrings of some classes, but still a long way to go.

- New 'boxcar' mapper, which is the similar to the already present transformWithBoxCar() function, but implemented as a mapper.

- New SampleGroupMapper that can be used for e.g. block averaging of samples. See new FAQ item.

- Stripped redundant suffixes from module names, e.g. mvpa.datasets.niftidataset -> mvpa.datasets.nifti

- mvpa.misc.cmdline variables opt* and opts* were groupped within opt and optss class instances. Also names of the options were changed to match 'dest' of the options. Use tools/refactor.py to quickly fix your custom code.

- Change all references to PyMVPA website to www.pymvpa.org.

- Make website stylesheet compatible with sphinx 0.4.

- Several minor improvements of the compatibilty with MacOS.

- Extended FAQ section of the manual.

- Bugfix: doubleGammaHRF() ignoring K2 argument.

• 0.2.2 (Tue, 17 Jun 2008)

- Extended build instructions: Added section on OpenSUSE.

- Replaced ugly PYMVPA_LIBSVM environment variable to trigger compiling the LIBSVM wrapper with a proper '–with-libsvm' switch in setup.py. Additionally, setup.py now detects if included LIBSVM has been built and enables LIBSVM wrapper automatically in this case.

- Added proper Makefiles for LIBSVM copy, with configurable compiler flags.

- Added 'setup.cfg' to remove the need to manually specify swig-opts (Windows specific configuration is in 'setup.cfg.win').

• 0.2.1 (Sun, 15 Jun 2008)

- Several improvements to make building PyMVPA on Windows systems easy (e.g. added dedicated Makefile.win to build a binary installer).

- Improved and extended documentation for building and installing PyMVPA.

- Include a minimal copy of the required (patched) LIBSVM library (currently version 2.85.0) for convenience. This copy is automatically compiled and used for the LIBSVM wrapper when PyMVPA built using the *Make* approach.

• 0.2.0 (Wed, 29 May 2008)

- New Splitter class (HalfSplitter) to split into first and second half.

- New Splitter class (CustomSplitter) to allow for splits with an arbitrary number of datasets per split and the ability to specify the association of samples with any of those datasets (not just the validation set).

- New sparse multinomial logistic regression (SMLR) classifier and associated sensitivity analyzer.
- New least angle regression classifier (LARS).
- New gaussian process regression classifier (GPR).
- Initial documentation on extending PyMVPA.
- Switch to Sphinx for documentation handling.
- New example comparing the performance of all classifiers on some artificial datasets.
- New data mapper performing singular value decomposition (SVDMapper) and an example showing its usage.
- More sophisticated data preprocessing: removal of non-linear trends and other arbitrary confounding regressors.
- New *Harvester* class to feed data from arbitrary generators into multiple objects and store results of returned values and arbitrary properties.
- Added documentation about how to build patched libsvm version with sane debug output.
- libsvm bindings are not build by default anymore. Instructions on how to reenable them are available in the manual.
- New wrapper from SVM implementation of the Shogun toolbox.
- Important bugfix in RFE, which reported incorrect feature ids in some cases.
- Added ability to compute stats/probabilities for all measures and transfer errors.

- 0.1.0 (Wed, 20 Feb 2008)

  - First public release.

MODULE REFERENCE

This module reference extends the manual with a comprehensive overview of the currently available functionality, that is built into PyMVPA. However, instead of a full list including every single line of the PyMVPA code base, this reference limits itself to the relevant pieces of the application programming interface (API) that are of particular interest to users of this framework.

Each module in the package is documented by a general summary of its purpose and the list of classes and functions it provides.

For developers, more detailed (technical) information is available in the API reference.

16.1 Global Facilities

16.1.1 `mvpa`

MultiVariate Pattern Analysis

Package Organization

The mvpa package contains the following subpackages and modules:

group Basic Data Structures
>   datasets

group Classifiers
>   clf

group Algorithms
>   algorithms

group Miscellaneous
>   misc

author
>   Michael Hanke, Yaroslav Halchenko, Per B. Sederberg

requires
>   Python 2.4+

version
>   0.4.0

see
>   The PyMVPA webpage

see
>   GIT Repository Browser

license
>   The MIT License

copyright
        © 2006-2008 Michael Hanke <michael.hanke@gmail.com>

newfield contributor
        Contributor, Contributors (Alphabetical Order)

contributor
        Per B. Sederberg

contributor
        Yaroslav O. Halchenko

The comprehensive API documentation for this module, including all technical details, is available in the Epydoc-generated API reference for mvpa (for developers).

seed(*random_seed*)
        Uniform and combined seeding of all relevant random number generators.

See Also:

Full API documentation of seed() in module mvpa.

16.2 Datasets: Input, Output, Storage and Preprocessing

16.2.1 `mvpa.datasets`

PyMVPA datasets and helper classes such as mappers, splitters

Module Description

Dataset and derived classes are dedicated to contain the data and associated information (such as labels, chunk(session) identifiers.

Module Organization

The mvpa.datasets module contains the following modules:

        group Generic Datasets
                base, mapped, masked, meta
        group Specialized Datasets
                nifti, eep
        group Mappers
                mapper, maskmapper
        group Metrics
                metric
        group Splitters
                splitter, nfoldsplitter
        group Miscellaneous
                miscfx

The mvpa.datasets module contains the following modules:

The comprehensive API documentation for this module, including all technical details, is available in the Epydoc-generated API reference for mvpa.datasets (for developers).

16.2.2 `mvpa.datasets.base`

Dataset container

The comprehensive API documentation for this module, including all technical details, is available in the Epydoc-generated API reference for mvpa.datasets.base (for developers).

Dataset

class Dataset (*data=None, dsattr=None, dtype=None, samples=None, labels=None, labels_map=None, chunks=None, origids=None, check_data=True, copy_samples=False, copy_data=True, copy_dsattr=True*)

Bases: `object`

The Dataset.

This class provides a container to store all necessary data to perform MVPA analyses. These are the data samples, as well as the labels associated with the samples. Additionally, samples can be grouped into chunks.

> •*Creators*: *__init__*, *selectFeatures*, *selectSamples*, *applyMapper*
> •*Mutators*: *permuteLabels*

Important: labels assumed to be immutable, i.e. noone should modify them externally by accessing indexed items, ie something like `dataset.labels[1] += "_bad"` should not be used. If a label has to be modified, full copy of labels should be obtained, operated on, and assigned back to the dataset, otherwise dataset.uniquelabels would not work. The same applies to any other attribute which has corresponding unique* access property.

Initialize dataset instance

There are basically two different way to create a dataset:

1. Create a new dataset from samples and sample attributes. In this mode a two-dimensional *ndarray* has to be passed to the *samples* keyword argument and the corresponding samples attributes are provided via the *labels* and *chunks* arguments.

   Copy contructor mode
   > The second way is used internally to perform quick coyping of datasets, e.g. when performing feature selection. In this mode and the two dictionaries (*data* and *dsattr*) are required. For performance reasons this mode bypasses most of the sanity check performed by the previous mode, as for internal operations data integrity is assumed.

2.
   > •*data* (dict) – Dictionary with an arbitrary number of entries. The value for each key in the dict has to be an ndarray with the same length as the number of rows in the samples array. A special entry in this dictionary is 'samples', a 2d array (samples x features). A shallow copy is stored in the object.
   > •*dsattr* (dict) – Dictionary of dataset attributes. An arbitrary number of arbitrarily named and typed objects can be stored here. A shallow copy of the dictionary is stored in the object.
   > •*dtype* (type | None) – If None – do not change data type if samples is an ndarray. Otherwise convert samples to dtype.

   samples
   > [ndarray] 2d array (samples x features)

   labels
   > An array or scalar value defining labels for each samples

   labels_map
   > [None or bool or dict] Map from labels into literal names. If is None or True, the mapping is computed, from labels which must be literal. If is False, no mapping is computed. If dict – mapping is verified and taken, labels get remapped. Dict must map literal -> number

   chunks
   > An array or scalar value defining chunks for each sample

Each of the Keywords arguments overwrites what is/might be already in the *data* container.

C
> chunks

I
origids

L
labels

S
samples

UC
attrib

UL
attrib

applyMapper (*featuresmapper=None, samplesmapper=None, train=True*)
Obtain new dataset by applying mappers over features and/or samples.

While featuresmappers leave the sample attributes information unchanged, as the number of samples in the dataset is invariant, samplesmappers are also applied to the samples attributes themselves!

Applying a featuresmapper will destroy any feature grouping information.

•*featuresmapper* (Mapper) – *Mapper* to somehow transform each sample's features

•*samplesmapper* (Mapper) – *Mapper* to transform each feature across samples

•*train* (bool) – Flag whether to train the mapper with this dataset before applying it.

TODO: selectFeatures is pretty much
applyMapper(featuresmapper=MaskMapper(...))

chunks
chunks

convertFeatureIds2FeatureMask (*ids*)
Returns a boolean mask with all features in *ids* selected.

•*ids* (list or 1d array) – To be selected features ids.

Return type
ndarray

Returns
All selected features are set to True; False otherwise.

convertFeatureMask2FeatureIds (*mask*)
Returns feature ids corresponding to non-zero elements in the mask.

•*mask* (1d ndarray) – Feature mask.

Return type
ndarray

Returns
Ids of non-zero (non-False) mask elements.

copy ()
Create a copy (clone) of the dataset, by fully copying current one

defineFeatureGroups (*definition*)

getLabelsMap ()
Stored labels map (if any)

getNFeatures ()
Number of features per pattern.

getNSamples ()
Currently available number of patterns.

getRandomSamples (*nperlabel*)
Select a random set of samples.

If 'nperlabel' is an integer value, the specified number of samples is randomly choosen from the group of samples sharing a unique label value (total number of selected samples: nperlabel x len(uniquelabels).

If 'nperlabel' is a list which's length has to match the number of unique label values. In this case 'nper-label' specifies the number of samples that shall be selected from the samples with the corresponding label.

The method returns a Dataset object containing the selected samples.

idhash

> To verify if dataset is in the same state as when smth else was done
>
> Like if classifier was trained on the same dataset as in question

idsbychunks (*x*)

> attrib

idsbylabels (*x*)

> attrib

idsonboundaries (*prior=0, post=0, attributes_to_track=, ['labels', 'chunks'], affected_labels=None, revert=False*)

> Find samples which are on the boundaries of the blocks
>
> Such samples might need to be removed. By default (with prior=0, post=0) ids of the first samples in a 'block' are reported
>
> > •*prior* (int) – how many samples prior to transition sample to include
> >
> > •*post* (int) – how many samples post the transition sample to include
> >
> > •*attributes_to_track* (list of basestring) – which attributes to track to decide on the boundary condition
> >
> > •*affected_labels* (list of basestring) – for which labels to perform selection. If None - for all
> >
> > •*revert* (bool) – either to revert the meaning and provide ids of samples which are found to not to be boundary samples

index (**args, **kwargs*)

> Universal indexer to obtain indexes of interesting samples/features. See .select() for more information
>
> > Return
> >
> > > tuple of (samples indexes, features indexes). Each item could be also None, if no selection on samples or features was requested (to discriminate between no selected items, and no selections)

labels

> labels

labels_map

> Stored labels map (if any)

nfeatures

> Number of features per pattern.

nsamples

> Currently available number of patterns.

origids

> origids

permuteLabels (*status, perchunk=True, assure_permute=False*)

> Permute the labels.
>
> TODO: rename status into something closer in semantics.
>
> > •*status* (bool) – Calling this method with set to True, the labels are permuted among all samples. If 'status' is False the original labels are restored.
> >
> > •*perchunk* (bool) – If True permutation is limited to samples sharing the same chunk value. Therefore only the association of a certain sample with a label is permuted while keeping the absolute number of occurences of each label value within a certain chunk constant.
> >
> > •*assure_permute* (bool) – If True, assures that labels are permutted, ie any one is different from the original one

samples

> samples

---

samplesperchunk
    attrib

samplesperlabel
    attrib

select(**args, **kwargs*)
    Universal selector

    WARNING: if you need to select duplicate samples (e.g. samples=[5,5]) or order of selected samples of features is important and has to be not ordered (e.g. samples=[3,2,1]), please use selectFeatures or selectSamples functions directly

    Examples:
        Mimique plain selectSamples:

```
dataset.select([1,2,3])
dataset[[1,2,3]]
```

        Mimique plain selectFeatures:

```
dataset.select(slice(None), [1,2,3])
dataset.select('all', [1,2,3])
dataset[:, [1,2,3]]
```

        Mixed (select features and samples):

```
dataset.select([1,2,3], [1, 2])
dataset[[1,2,3], [1, 2]]
```

        Select samples matching some attributes:

```
dataset.select(labels=[1,2], chunks=[2,4])
dataset.select('labels', [1,2], 'chunks', [2,4])
dataset['labels', [1,2], 'chunks', [2,4]]
```

        Mixed – out of first 100 samples, select only those with labels 1 or 2 and belonging to chunks 2 or 4, and select features 2 and 3:

```
dataset.select(slice(0,100), [2,3], labels=[1,2], chunks=[2,4])
dataset[:100, [2,3], 'labels', [1,2], 'chunks', [2,4]]
```

selectFeatures(*ids=None, sort=True, groups=None*)
    Select a number of features from the current set.

            •*ids* – iterable container to select ids

            •*sort* (bool) – if to sort Ids. Order matters and *selectFeatures* assumes incremental order. If not such, in non-optimized code selectFeatures would verify the order and sort

    Returns a new Dataset object with a view of the original samples array (no copying is performed).

    WARNING: The order of ids determines the order of features in the returned dataset. This might be useful sometimes, but can also cause major headaches! Order would is verified when running in non-optimized code (if __debug__)

selectSamples(*ids*)
    Choose a subset of samples defined by samples IDs.

    Returns a new dataset object containing the selected sample subset.

    TODO: yoh, we might need to sort the mask if the mask is a list of ids and is not ordered. Clarify with Michael what is our intent here!

setLabelsMap(*lm*)
    Set labels map.

    Checks for the validity of the mapping – values should cover all existing labels in the dataset

setSamplesDType(*dtype*)
    Set the data type of the samples array.

summary(*uniq=True, stats=True, idhash=False, lstats=True, maxc=30, maxl=20*)
    String summary over the object

•*uniq* (bool) – Include summary over data attributes which have unique

•*idhash* (bool) – Include idhash value for dataset and samples

•*stats* (bool) – Include some basic statistics (mean, std, var) over dataset samples

•*lstats* (bool) – Include statistics on chunks/labels

•*maxc* (int) – Maximal number of chunks when provide details on labels/chunks

•*maxl* (int) – Maximal number of labels when provide details on labels/chunks

summary_labels (*maxc=30, maxl=20*)

Provide summary statistics over the labels and chunks

•*maxc* (int) – Maximal number of chunks when provide details

•*maxl* (int) – Maximal number of labels when provide details

uniquechunks

attrib

uniquelabels

attrib

where (**args, **kwargs*)

Obtain indexes of interesting samples/features. See select() for more information

XXX somewhat obsoletes idsby...

See Also:

Derived classes might provide additional methods via their base classes. Please refer to the list of base classes (if it exists) at the begining of the `Dataset` documentation.

Full API documentation of Dataset in module mvpa.datasets.base.

datasetmethod (*func*)

Decorator to easily bind functions to a Dataset class

See Also:

Full API documentation of datasetmethod() in module mvpa.datasets.base.

16.2.3 `mvpa.datasets.channel`

Dataset handling data structured in channels.

The comprehensive API documentation for this module, including all technical details, is available in the Epydoc-generated API reference for mvpa.datasets.channel (for developers).

ChannelDataset

class ChannelDataset (*samples=None, dsattr=None, t0=None, dt=None, channelids=None, **kwargs*)

Bases: `mvpa.datasets.mapped.MappedDataset`

Dataset handling data structured into channels.

Channels are assumes to contain several timepoints, thus this Dataset stores some additional properties (reference time *t0*, temporal distance of two timepoints *dt* and *channelids* (names)).

Initialize ChannelDataset.

•*samples* (ndarray) – Three-dimensional array: (samples x channels x timepoints).

•*t0* (float) – Reference time of the first timepoint. Can be used to preserve information about the onset of some stimulation. Preferably in seconds.

•*dt* (float) – Temporal distance between two timepoints. Has to be given in seconds. Otherwise *samplingrate* property will not return *Hz*.

•*channelids* (list) – List of channel names.

channelids

List of channel IDs

dt
  Time difference between two samples (in seconds).

samplingrate
  Yeah, sampling rate.

substractBaseline(*t=None*)
  Substract mean baseline signal from the each timepoint.

  The baseline is determined by computing the mean over all timepoints specified by *t*.

  The samples of the dataset are modified in-place and nothing is returned.

    •*t* (int | float | None) – If an integer, *t* denotes the number of timepoints in the from the start of each sample to be used to compute the baseline signal. If a floating point value, *t* is the duration of the baseline window from the start of each sample in whatever unit corresponding to the datasets *samplingrate*. Finally, if *None* the *t0* property of the dataset is used to determine *t* as it would have been specified as duration.

t0
  Temporal position of first sample in the timeseries (in seconds) – possibly relative to stimulus onset.

See Also:

Derived classes might provide additional methods via their base classes. Please refer to the list of base classes (if it exists) at the begining of the ChannelDataset documentation.

Full API documentation of ChannelDataset in module mvpa.datasets.channel.

16.2.4 `mvpa.datasets.eep`

Dataset that gets its samples from an EEP binary file

The comprehensive API documentation for this module, including all technical details, is available in the Epydoc-generated API reference for mvpa.datasets.eep (for developers).

EEPDataset

class **EEPDataset**(*samples=None, **kwargs*)
  Bases: mvpa.datasets.channel.ChannelDataset

  Dataset using a EEP binary file as source.

  EEP files are used by *eeprobe* a software for analysing even-related potentials (ERP), which was developed at the Max-Planck Institute for Cognitive Neuroscience in Leipzig, Germany.

    http://www.ant-neuro.com/products/eeprobe

  Initialize EEPDataset.

    •*samples* (Filename (string) of a EEP binary file or an *EEPBin*) – object

See Also:

Derived classes might provide additional methods via their base classes. Please refer to the list of base classes (if it exists) at the begining of the EEPDataset documentation.

Full API documentation of EEPDataset in module mvpa.datasets.eep.

16.2.5 `mvpa.datasets.event`

Event-based dataset type

The comprehensive API documentation for this module, including all technical details, is available in the Epydoc-generated API reference for mvpa.datasets.event (for developers).

EventDataset

class EventDataset (*samples=None, events=None, mask=None, bcshape=None, dametric=None, **kwargs*)
Bases: `mvpa.datasets.mapped.MappedDataset`

Event-based dataset

This dataset type can be used to segment 'raw' data input into meaningful boxcar-shaped samples, by simply defining a list of events (see `Event`).

Additionally, it can be used to add arbitrary information (as features) to each event-sample (extracted from the event list itself). An appropriate mapper is automatically constructed, that merges original samples and additional features into a common feature space and also separates them again during reverse-mapping. Otherwise, this dataset type is a regular dataset (in contrast to *MetaDataset*).

The properties of an `Event` supported/required by this class are:

onset An integer indicating the startpoint of an event as the sample index in the input data.

duration
How many input data samples following the onset sample should be considered for an event. The embedded `BoxcarMapper` will use the maximum boxlength (i.e., *duration*) of all defined events to create a regular-shaped data array.

label The corresponding label of that event (numeric or literal).

chunk
An optional chunk id.

features
A list with an arbitrary number of features values (floats), that will be added to the feature vector of the corresponding sample.

- *samples* (ndarray) – 'Raw' input data from which boxcar-shaped samples will be extracted.
- *events* (sequence of *Event* instances) – Both an events *onset* and *duration* are assumed to be provided as #samples. The boxlength will be determined by the maximum duration of all events.
- *mask* (boolean array) – Only features corresponding to non-zero mask elements will be considered for the final dataset. The mask shape either has to match the shape of the generated boxcar-samples, or the shape of the 'raw' input samples. In the latter case, the mask is automatically expanded to cover the whole boxcar. If no mask is provided, a full mask will be constructed automatically.
- *bcshape* (tuple) – Shape of the boxcar samples generated by the embedded boxcar mapper. If not provided this is determined automatically. However, this required an extra mapping step.
- *dametric* (Metric) – Custom metric to be used by the embedded DenseArrayMapper.
- ***kwargs* – All additional arguments are passed to the base class.

See Also:

Derived classes might provide additional methods via their base classes. Please refer to the list of base classes (if it exists) at the begining of the `EventDataset` documentation.

Full API documentation of EventDataset in module mvpa.datasets.event.

16.2.6 `mvpa.datasets.mapped`

Mapped dataset

The comprehensive API documentation for this module, including all technical details, is available in the Epydoc-generated API reference for mvpa.datasets.mapped (for developers).

MappedDataset

class MappedDataset (*samples=None, mapper=None, dsattr=None, **kwargs*)
> Bases: `mvpa.datasets.base.Dataset`

> A *Dataset* which is created by applying a *Mapper* to the data.

> Upon contruction *MappedDataset* uses a *Mapper* to transform the samples from their original into the two-dimensional matrix representation that is required by the *Dataset* class.

> This class enhanced the *Dataset* interface with two additional methods: *mapForward()* and *mapReverse()*. Both take arbitrary data arrays (with matching shape) and transform them using the embedded mapper from the original dataspace into a one- or two-dimensional representation (for arrays corresponding to the shape of a single or multiple samples respectively) or vice versa.

> Most likely, this class will not be used directly, but rather indirectly through one of its subclasses (e.g. *MaskedDataset*).

> If *samples* and *mapper* arguments are not *None* the mapper is used to forward-map the samples array and the result is passed to the *Dataset* constructor.

> > •*mapper* (Instance of *Mapper*) – This mapper will be embedded in the dataset and is used and updated, by all subsequent mapping or feature selection procedures.
> > •***kwargs* – All other arguments are simply passed to and handled by the constructor of *Dataset*.

> **O**
> > Return samples in the original shape

> **mapForward** (*data*)
> > Map data from the original dataspace into featurespace.

> **mapReverse** (*data*)
> > Reverse map data from featurespace into the original dataspace.

> **mapSelfReverse** ()
> > Reverse samples from featurespace into the original dataspace.

> **mapper**
> > mapper

> **samples_original**
> > Return samples in the original shape

> **selectFeatures** (*ids, plain=False, sort=False*)
> > Select features given their ids.

> > The methods behaves similar to Dataset.selectFeatures(), but additionally takes care of adjusting the embedded mapper appropriately.

> > > •*ids* (sequence) – Iterable container to select ids
> > > •*plain* (boolean) – Flag whether to return MappedDataset (or just Dataset)
> > > •*sort* (boolean) – Flag whether to sort Ids. Order matters and selectFeatures assumes incremental order. If not such, in non-optimized code selectFeatures would verify the order and sort

See Also:

Derived classes might provide additional methods via their base classes. Please refer to the list of base classes (if it exists) at the begining of the `MappedDataset` documentation.

Full API documentation of MappedDataset in module mvpa.datasets.mapped.

16.2.7 `mvpa.datasets.masked`

Dataset with applied mask

The comprehensive API documentation for this module, including all technical details, is available in the Epydoc-generated API reference for mvpa.datasets.masked (for developers).

MaskedDataset

class MaskedDataset (*samples=None, mask=None, **kwargs*)
Bases: `mvpa.datasets.mapped.MappedDataset`

Helper class which is *MappedDataset* with using *MaskMapper*.

TODO: since what it does is simply some checkes/data_mangling in the constructor, it might be absorbed inside generic *MappedDataset*

•*mask* (ndarray) – the chosen features equal the non-zero mask elements.

selectFeaturesByMask (*mask, plain=False*)
Use a mask array to select features from the current set.

•*mask* (ndarray) – input mask
•*plain* (bool) – *True* directs to return a simple *Dataset*, *False* – a new *MaskedDataset* object

Returns a new MaskedDataset object with a view of the original pattern array (no copying is performed). The final selection mask only contains features that are present in the current feature mask AND the selection mask passed to this method.

See Also:

Derived classes might provide additional methods via their base classes. Please refer to the list of base classes (if it exists) at the begining of the `MaskedDataset` documentation.

Full API documentation of MaskedDataset in module mvpa.datasets.masked.

16.2.8 `mvpa.datasets.meta`

Dataset container

The comprehensive API documentation for this module, including all technical details, is available in the Epydoc-generated API reference for mvpa.datasets.meta (for developers).

MetaDataset

class MetaDataset (*datasets*)
Bases: `object`

Dataset container

The class is useful to combine several Datasets with different origin and type and bind them together. Such a combined dataset can then by used to e.g. pass it to a classifier.

MetaDataset does not permanently duplicate data stored in the dataset it contains. The combined samples matrix is build on demand and samples attribute access is redirected to the first dataset in the container.

Currently operations other than samples or feature selection are not fully supported, e.g. passing a Meta-Dataset to detrend() will initially result in a detrended MetaDataset, but the combined and detrended samples matrix will be lost after the next call to selectSamples() or selectFeatures(), which freshly pulls samples from all datasets in the container.

Initialize dataset instance

•*datasets* (list) –

applyMapper (*args, **kwargs*)
Apply a mapper on all underlying datasets.

datasets

getNFeatures ()
Number of features per sample.

getNSamples ()
Currently available number of samples.

getRandomSamples (*nperlabel*)
> Return a MetaDataset with a random subset of samples.

mapReverse (*val*)

nfeatures
> Number of features per sample.

nsamples
> Currently available number of samples.

permuteLabels (**args, **kwargs*)
> Toggle label permutation.

rebuildSamples ()
> Update the combined samples matrix from all underlying datasets.

selectFeatures (*ids, sort=True*)
> Do feature selection on all underlying datasets at once.

selectSamples (**args, **kwargs*)
> Select samples from all underlying datasets at once.

setSamplesDType (*dtype*)
> Set the data type of the samples array.

See Also:

Derived classes might provide additional methods via their base classes. Please refer to the list of base classes (if it exists) at the begining of the MetaDataset documentation.

Full API documentation of MetaDataset in module mvpa.datasets.meta.

16.2.9 `mvpa.datasets.miscfx`

Misc function performing operations on datasets.

All the functions defined in this module must accept dataset as the first argument since they are bound to Dataset class in the trailer.

The comprehensive API documentation for this module, including all technical details, is available in the Epydoc-generated API reference for mvpa.datasets.miscfx (for developers).

Functions

aggregateFeatures (*dataset, fx=<function mean at 0x879909c>*)
> Apply a function to each row of the samples matrix of a dataset.

> The functor given as *fx* has to honour an *axis* keyword argument in the way that NumPy used it (e.g. NumPy.mean, var).

>> Return type
>>> a new *Dataset* object with the aggregated feature(s).

See Also:

Full API documentation of aggregateFeatures() in module mvpa.datasets.miscfx.

coarsenChunks (*source, nchunks=4*)
> Change chunking of the dataset

> Group chunks into groups to match desired number of chunks. Makes sense if originally there were no strong groupping into chunks or each sample was independent, thus belonged to its own chunk

>> • *source* (Dataset or list of chunk ids) – dataset or list of chunk ids to operate on. If Dataset, then its chunks get modified
>> • *nchunks* (int) – desired number of chunks

See Also:

Full API documentation of coarsenChunks() in module mvpa.datasets.miscfx.

getSamplesPerChunkLabel(*dataset*)
    Returns an array with the number of samples per label in each chunk.

    Array shape is (chunks x labels).

> •*dataset* (Dataset) – Source dataset.

See Also:

Full API documentation of getSamplesPerChunkLabel() in module mvpa.datasets.miscfx.

removeInvariantFeatures(*dataset*)
    Returns a new dataset with all invariant features removed.

See Also:

Full API documentation of removeInvariantFeatures() in module mvpa.datasets.miscfx.

zscore(*dataset, mean=None, std=None, perchunk=True, baselinelabels=None, pervoxel=True, targetdtype='float64'*)
    Z-Score the samples of a *Dataset* (in-place).

    mean and *std* can be used to pass custom values to the z-scoring. Both may be scalars or arrays.

    All computations are done *in place*. Data upcasting is done automatically if necessary into *targetdtype*

    If *baselinelabels* provided, and *mean* or *std* aren't provided, it would compute the corresponding measure based only on labels in *baselinelabels*

    If *perchunk* is True samples within the same chunk are z-scored independent of samples from other chunks, e.i. mean and standard deviation are calculated individually.

See Also:

Full API documentation of zscore() in module mvpa.datasets.miscfx.

16.2.10 `mvpa.datasets.miscfx_sp`

The comprehensive API documentation for this module, including all technical details, is available in the Epydoc-generated API reference for mvpa.datasets.miscfx_sp (for developers).

See Also:

Full API documentation of detrend() in module mvpa.datasets.miscfx_sp.

16.2.11 `mvpa.datasets.nifti`

The comprehensive API documentation for this module, including all technical details, is available in the Epydoc-generated API reference for mvpa.datasets.nifti (for developers).

Classes

ERNiftiDataset

See Also:

Derived classes might provide additional methods via their base classes. Please refer to the list of base classes (if it exists) at the begining of the `ERNiftiDataset` documentation.

Full API documentation of ERNiftiDataset in module mvpa.datasets.nifti.

NiftiDataset

See Also:

Derived classes might provide additional methods via their base classes. Please refer to the list of base classes (if it exists) at the begining of the `NiftiDataset` documentation.

Full API documentation of NiftiDataset in module mvpa.datasets.nifti.

Functions

See Also:

Full API documentation of getNiftiData() in module mvpa.datasets.nifti.

See Also:

Full API documentation of getNiftiFromAnySource() in module mvpa.datasets.nifti.

16.2.12 `mvpa.datasets.splitter`

Collection of dataset splitters.

The comprehensive API documentation for this module, including all technical details, is available in the Epydoc-generated API reference for mvpa.datasets.splitter (for developers).

Classes

CustomSplitter

class CustomSplitter(*splitrule, **kwargs*)
   Bases: `mvpa.datasets.splitter.Splitter`

   Split a dataset using an arbitrary custom rule.

   The splitter is configured by passing a custom spitting rule (*splitrule*) to its constructor. Such a rule is basically a sequence of split definitions. Every single element in this sequence results in excatly one split generated by the Splitter. Each element is another sequence for sequences of sample ids for each dataset that shall be generated in the split.

   Example:

   • Generate two splits. In the first split the *second* dataset contains all samples with sample attributes corresponding to either 0, 1 or 2. The *first* dataset of the first split contains all samples which are not split into the second dataset.
     The second split yields three datasets. The first with all samples corresponding to sample attributes 1 and 2, the second dataset contains only samples with attrbiute 3 and the last dataset contains the samples with attribute 5 and 6.
     CustomSplitter([(None, [0, 1, 2]), ([1,2], [3], [5, 6])])

   Cheap init.

See Also:

Derived classes might provide additional methods via their base classes. Please refer to the list of base classes (if it exists) at the begining of the `CustomSplitter` documentation.

Full API documentation of CustomSplitter in module mvpa.datasets.splitter.

HalfSplitter

class HalfSplitter(***kwargs*)

Bases: `mvpa.datasets.splitter.Splitter`

Split a dataset into two halves of the sample attribute.

The splitter yields to splits: first (1st half, 2nd half) and second (2nd half, 1st half).

Cheap init.

See Also:

Derived classes might provide additional methods via their base classes. Please refer to the list of base classes (if it exists) at the begining of the `HalfSplitter` documentation.

Full API documentation of HalfSplitter in module mvpa.datasets.splitter.

NFoldSplitter

class NFoldSplitter(*cvtype=1, **kwargs*)

Bases: `mvpa.datasets.splitter.Splitter`

Generic N-fold data splitter.

XXX: This docstring is a shame for such an important class!

Initialize the N-fold splitter.

- *cvtype* (Int) – Type of cross-validation: N-(cvtype)
- *kwargs* – Additional parameters are passed to the *Splitter* base class.

See Also:

Derived classes might provide additional methods via their base classes. Please refer to the list of base classes (if it exists) at the begining of the `NFoldSplitter` documentation.

Full API documentation of NFoldSplitter in module mvpa.datasets.splitter.

NoneSplitter

class NoneSplitter(*mode='second', **kwargs*)

Bases: `mvpa.datasets.splitter.Splitter`

This is a dataset splitter that does **not** split. It simply returns the full dataset that it is called with.

The passed dataset is returned as the second element of the 2-tuple. The first element of that tuple will always be 'None'.

Cheap init – nothing special

- *mode* – Either 'first' or 'second' (default) – which output dataset would actually contain the samples

See Also:

Derived classes might provide additional methods via their base classes. Please refer to the list of base classes (if it exists) at the begining of the `NoneSplitter` documentation.

Full API documentation of NoneSplitter in module mvpa.datasets.splitter.

OddEvenSplitter

class OddEvenSplitter(*usevalues=False, **kwargs*)

Bases: `mvpa.datasets.splitter.Splitter`

Split a dataset into odd and even values of the sample attribute.

The splitter yields to splits: first (odd, even) and second (even, odd).

Cheap init.

> •*usevalues* (Boolean) – If True the values of the attribute used for splitting will be used to determine odd and even samples. If False odd and even chunks are defined by the order of attribute values, i.e. first unique attribute is odd, second is even, despite the corresponding values might indicate the opposite (e.g. in case of [2,3]).

See Also:

Derived classes might provide additional methods via their base classes. Please refer to the list of base classes (if it exists) at the begining of the `OddEvenSplitter` documentation.

Full API documentation of OddEvenSplitter in module mvpa.datasets.splitter.

Splitter

class Splitter(*nperlabel='all', nrunspersplit=1, permute=False, count=None, strategy='equidistant', attr='chunks'*)
Bases: `object`

Base class of dataset splitters.

Each splitter should be initialized with all its necessary parameters. The final splitting is done running the splitter object on a certain Dataset via __call__(). This method has to be implemented like a generator, i.e. it has to return every possible split with a yield() call.

Each split has to be returned as a sequence of Datasets. The properties of the splitted dataset may vary between implementations. It is possible to declare a sequence element as 'None'.

Please note, that even if there is only one Dataset returned it has to be an element in a sequence and not just the Dataset object!

Initialize splitter base.

> •*nperlabel* (int or str (or list of them)) – Number of dataset samples per label to be included in each split. Two special strings are recognized: 'all' uses all available samples (default) and 'equal' uses the maximum number of samples the can be provided by all of the classes. This value might be provided as a sequence whos length matches the number of datasets per split and indicates the configuration for the respective dataset in each split.
>
> •*nrunspersplit* (int) – Number of times samples for each split are chosen. This is mostly useful if a subset of the available samples is used in each split and the subset is randomly selected for each run (see the *nperlabel* argument).
>
> •*permute* (bool) – If set to *True*, the labels of each generated dataset will be permuted on a per-chunk basis.
>
> •*count* (None or int) – Desired number of splits to be output. It is limited by the number of splits possible for a given splitter (e.g. *OddEvenSplitter* can have only up to 2 splits). If None, all splits are output (default).
>
> •*strategy* (str) – If *count* is not None, possible strategies are possible: first First *count* splits are chosen random Random (without replacement) *count* splits are chosen equidistant Splits which are equidistant from each other
>
> •*attr* (str) – Sample attribute used to determine splits.

setNPerLabel(*value*)
Set the number of samples per label in the split datasets.

'equal' sets sample size to highest possible number of samples that can be provided by each class. 'all' uses all available samples (default).

splitDataset(*dataset, specs*)
Split a dataset by separating the samples where the configured sample attribute matches an element of *specs*.

> • *dataset* (Dataset) – This is this source dataset.
> • *specs* (sequence of sequences) – Contains ids of a sample attribute that shall be split into the another dataset.

Returns

Tuple of splitted datasets.

splitcfg(*dataset*)

Return splitcfg for a given dataset

strategy

See Also:

Derived classes might provide additional methods via their base classes. Please refer to the list of base classes (if it exists) at the begining of the `Splitter` documentation.

Full API documentation of Splitter in module mvpa.datasets.splitter.

16.3 Mappers: Data Transformations

16.3.1 `mvpa.mappers`

PyMVPA mappers.

The comprehensive API documentation for this module, including all technical details, is available in the Epydoc-generated API reference for mvpa.mappers (for developers).

16.3.2 `mvpa.mappers.array`

Data mapper

The comprehensive API documentation for this module, including all technical details, is available in the Epydoc-generated API reference for mvpa.mappers.array (for developers).

DenseArrayMapper

class **DenseArrayMapper**(*mask=None, metric=None, distance_function=<function cartesianDistance at 0x8fa7e9c>, elementsize=None, shape=None, **kwargs*)

Bases: `mvpa.mappers.mask.MaskMapper`

Mapper for equally spaced dense arrays.

Initialize DenseArrayMapper

> • *mask* (array) – an array in the original dataspace and its nonzero elements are used to define the features included in the dataset. alternatively, the *shape* argument can be used to define the array dimensions.
> • *metric* (Metric) – Corresponding metric for the space. No attempt is made to determine whether a certain metric is reasonable for this mapper. If *metric* is None – *DescreteMetric* is constructed that assumes an equal (1) spacing of all mask elements with a *distance_function* given as a parameter listed below.
> • *distance_function* (functor) – Distance function to use as the parameter to *DescreteMetric* if *metric* is not specified,
> • *elementsize* (list or scalar) – Determines spacing within *DescreteMetric*. If it is given as a scalar, corresponding value is assigned to all dimensions, which are found within *mask*
> • *shape* (tuple) – The shape of the array to be mapped. If *shape* is provided instead of *mask*, a full mask (all True) of the desired shape is constructed. If *shape* is specified in addition to *mask*, the provided mask is extended to have the same number of dimensions.

Note
> parameters *elementsize* and *distance_function* are relevant only if *metric* is None

See Also:

Derived classes might provide additional methods via their base classes. Please refer to the list of base classes (if it exists) at the begining of the `DenseArrayMapper` documentation.

Full API documentation of DenseArrayMapper in module mvpa.mappers.array.

16.3.3 `mvpa.mappers.base`

Data mapper

The comprehensive API documentation for this module, including all technical details, is available in the Epydoc-generated API reference for mvpa.mappers.base (for developers).

Classes

ChainMapper

class **ChainMapper** (*mappers, **kwargs*)
> Bases: `mvpa.mappers.base.Mapper`

> Meta mapper that embedded a chain of other mappers.

> Each mapper in the chain is called successively to perform forward or reverse mapping.

> **Note:** In its current implementation the *ChainMapper* treats all but the last mapper as simple pre-processing (in forward()) or post-processing (in reverse()) steps. All other capabilities, e.g. training and neighbor metrics are provided by or affect *only the last mapper in the chain*.

> With respect to neighbor metrics this means that they are determined based on the input space of the *last mapper* in the chain and *not* on the input dataspace of the *ChainMapper* as a whole

>> •*mappers* (list of Mapper instances) –
>> •***kwargs* – All additional arguments are passed to the base-class constructor.

> **forward** (*data*)
>> Calls all mappers in the chain successively.

>>> •*data* – data to be chain-mapped.

> **getInSize** ()
>> Returns the size of the entity in input space

> **getNeighbor** (*outId, *args, **kwargs*)
>> Get the ids of the neighbors of a single feature in output dataspace.

>> **Note:** The neighbors are determined based on the input space of the *last mapper* in the chain and *not* on the input dataspace of the *ChainMapper* as a whole!

>>> •*outId* (int) – Single id of a feature in output space, whos neighbors should be determined.
>>> •**args, **kwargs* – Additional arguments are passed to the metric of the embedded mapper, that is responsible for the corresponding feature.

>> Returns a list of outIds

> **getOutSize** ()
>> Returns the size of the entity in output space

> **reverse** (*data*)
>> Calls all mappers in the chain successively, in reversed order.

>>> •*data* (array) – data array to be reverse mapped into the orginal dataspace.

> **selectOut** (*outIds*)
>> Remove some elements from the *last* mapper in the chain.

•*outIds* (sequence) – All output feature ids to be selected/kept.

train(*dataset*)
Trains the *last* mapper in the chain.

•*dataset* (`Dataset` or subclass) – A dataset with the number of features matching the *outSize* of the last mapper in the chain (which is identical to the one of the *ChainMapper* itself).

See Also:

Derived classes might provide additional methods via their base classes. Please refer to the list of base classes (if it exists) at the begining of the `ChainMapper` documentation.

Full API documentation of ChainMapper in module mvpa.mappers.base.

CombinedMapper

class **CombinedMapper**(*mappers, **kwargs*)
Bases: `mvpa.mappers.base.Mapper`

Meta mapper that combines several embedded mappers.

This mapper can be used the map from several input dataspaces into a common output dataspace. When `forward()` is called with a sequence of data, each element in that sequence is passed to the corresponding mapper, which in turned forward-maps the data. The output of all mappers is finally stacked (horizontally or column or feature-wise) into a single large 2D matrix (nsamples x nfeatures).

Note: This mapper can only embded mappers that transform data into a 2D (nsamples x nfeatures) representation. For mappers not supporting this transformation, consider wrapping them in a `ChainMapper` with an appropriate post-processing mapper.

CombinedMapper fully supports forward and backward mapping, training, runtime selection of a feature subset (in output dataspace) and retrieval of neighborhood information.

•*mappers* (list of Mapper instances) – The order of the mappers in the list is important, as it will define the order in which data snippets have to be passed to `forward()`.

•***kwargs* – All additional arguments are passed to the base-class constructor.

forward(*data*)
Map data from the IN spaces into to common OUT space.

•*data* (sequence) – Each element in the *data* sequence is passed to the corresponding embedded mapper and is mapped individually by it. The number of elements in *data* has to match the number of embedded mappers. Each element is *data* has to provide the same number of samples (first dimension).

Return type
array

Returns
Horizontally stacked array of all embedded mapper outputs.

getInSize()
Returns the size of the entity in input space

getNeighbor(*outId, *args, **kwargs*)
Get the ids of the neighbors of a single feature in output dataspace.

•*outId* (int) – Single id of a feature in output space, whos neighbors should be determined.

•**args, **kwargs* – Additional arguments are passed to the metric of the embedded mapper, that is responsible for the corresponding feature.

Returns a list of outIds

getOutSize()
Returns the size of the entity in output space

---

reverse(*data*)
> Reverse map data from OUT space into the IN spaces.
>> •*data* (array) – Single data array to be reverse mapped into a sequence of data snippets in their individual IN spaces.
>
>> Return type
>>> list

selectOut(*outIds*)
> Remove some elements and leave only ids in 'out'/feature space.
> **Note:** The subset selection is done inplace
>> •*outIds* (sequence) – All output feature ids to be selected/kept.

train(*dataset*)
> Trains all embedded mappers.
> The provided training dataset is splitted appropriately and the corresponding pieces are passed to the train() method of each embedded mapper.
>> •*dataset* (Dataset or subclass) – A dataset with the number of features matching the *outSize* of the *CombinedMapper*.

See Also:

Derived classes might provide additional methods via their base classes. Please refer to the list of base classes (if it exists) at the begining of the CombinedMapper documentation.

Full API documentation of CombinedMapper in module mvpa.mappers.base.

Mapper

class **Mapper**(*metric=None*)
> Bases: object

> Interface to provide mapping between two spaces: IN and OUT. Methods are prefixed correspondingly. forward/reverse operate on the entire dataset. get(In|Out)Id[s] operate per element:

```
     forward
    --------->
IN              OUT
    <--------/
     reverse
```
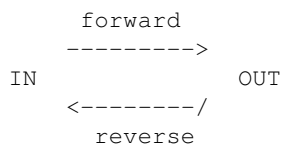
>> •*metric* (Metric) – Optional metric

forward(*data*)
> Map data from the IN dataspace into OUT space.

getInId(*outId*)
> Translate a feature id into a coordinate/index in input space.
> Such a translation might not be meaningful or even possible for a particular mapping algorithm and therefore cannot be relied upon.

getInSize()
> Returns the size of the entity in input space

getMetric()
> To make pylint happy

getNeighbor(*outId, *args, **kwargs*)
> Get feature neighbors in input space, given an id in output space.
> This method has to be reimplemented whenever a derived class does not provide an implementation for getInId().

getNeighborIn(*inId, *args, **kwargs*)
> Return the list of coordinates for the neighbors.

•*inId* – id (index) of an element in input dataspace.

•**args, **kwargs* – Any additional arguments are passed to the embedded metric of the mapper.

XXX See TODO below: what to return – list of arrays or list of tuples?

getNeighbors(*outId, *args, **kwargs*)
Return the list of coordinates for the neighbors.

By default it simply constructs the list based on the generator returned by getNeighbor()

getOutSize()
Returns the size of the entity in output space

isValidInId(*inId*)
Validate id in IN space.

Override if IN space is not simly a 1D vector

isValidOutId(*outId*)
Validate feature id in OUT space.

Override if OUT space is not simly a 1D vector

metric
To make pylint happy

nfeatures
str(object) -> string

Return a nice string representation of the object. If the argument is a string, the return value is the same object.

reverse(*data*)
Reverse map data from OUT space into the IN space.

selectOut(*outIds*)
Limit the OUT space to a certain set of features.

•*outIds* (sequence) – Subset of ids of the current feature in OUT space to keep.

setMetric(*metric*)
To make pylint happy

train(*dataset*)
Perform training of the mapper.

This method is called to put the mapper in a state that allows it to perform to intended mapping.

•*dataset* (Dataset or subclass) –

Note: The default behavior of this method is to do nothing.

See Also:

Derived classes might provide additional methods via their base classes. Please refer to the list of base classes (if it exists) at the begining of the Mapper documentation.

Full API documentation of Mapper in module mvpa.mappers.base.

ProjectionMapper

class **ProjectionMapper**(*selector=None, demean=True*)
Bases: mvpa.mappers.base.Mapper

Mapper using a projection matrix to transform the data.

This class cannot be used directly. Sub-classes have to implement the *_train()* method, which has to compute the projection matrix given a dataset (see *_train()* docstring for more information).

Once the projection matrix is available, this class provides functionality to perform forward and backwards mapping of data, the latter using the hermitian (conjugate) transpose of the projection matrix. Additionally, *ProjectionMapper* supports optional (but done by default) demeaning of the data and selection of arbitrary component (i.e. columns of the projection matrix) of the projection.

Forward and back-projection matrices (a.k.a. *projection* and *reconstruction*) are available via the *proj* and *recon* properties. the latter only after it has been computed (after first call to *reverse*).

Initialize the ProjectionMapper

- *selector* (None | list) – Which components (i.e. columns of the projection matrix) should be used for mapping. If *selector* is *None* all components are used. If a list is provided, all list elements are treated as component ids and the respective components are selected (all others are discarded).
- *demean* (bool) – Either data should be demeaned while computing projections and applied back while doing reverse()

forward(*data, demean=None*)
Perform forward projection.

- *data* (ndarray) – Data array to map
- *demean* (boolean | None) – Override demean setting for this method call.

Return type
NumPy array

getInSize()
Returns the number of original features.

getOutSize()
Returns the number of components to project on.

proj
Projection matrix

recon
Backprojection matrix

reverse(*data*)
Reproject (reconstruct) data into the original feature space.

Return type
NumPy array

selectOut(*outIds*)
Choose a subset of components (and remove all others).

train(*dataset*)
Determine the projection matrix.

See Also:

Derived classes might provide additional methods via their base classes. Please refer to the list of base classes (if it exists) at the begining of the `ProjectionMapper` documentation.

Full API documentation of ProjectionMapper in module mvpa.mappers.base.

16.3.4 `mvpa.mappers.boxcar`

Data mapper

The comprehensive API documentation for this module, including all technical details, is available in the Epydoc-generated API reference for mvpa.mappers.boxcar (for developers).

BoxcarMapper

class **BoxcarMapper**(*startpoints, boxlength, offset=0, collision_resolution='mean'*)
Bases: `mvpa.mappers.base.Mapper`

Mapper to combine multiple samples into a single sample.

Note: This mapper is somewhat unconventional since it doesn't preserve number of samples (ie the size of 0-th dimension).

> • *startpoints* (sequence) – Index values along the first axis of 'data'.
>
> • *boxlength* (int) – The number of elements after 'startpoint' along the first axis of 'data' to be considered for the boxcar.
>
> • *offset* (int) – The offset between the provided starting point and the actual start of the boxcar.
>
> • *collision_resolution* ('mean') – if a sample belonged to multiple output samples, then on reverse, how to resolve the value

> **forward**(*data*)
>
> > Project an ND matrix into N+1D matrix
> >
> > This method also handles the special of forward mapping a single 'raw' sample. Such a sample is extended (by concatenating clones of itself) to cover a full boxcar. This functionality is only availably after a full data array has been forward mapped once.
> >
> > > Return type
> > > > array

> **getInSize**()
>
> > Returns the number of original samples which were combined.

> **getOutSize**()
>
> > Returns the number of output samples.

> **isValidInId**(*inId*)
>
> > Validate if InId is valid

> **isValidOutId**(*outId*)
>
> > Validate if OutId is valid

> **reverse**(*data*)
>
> > Uncombine features back into original space.
> >
> > Samples which were not touched by forward will get value 0 assigned

> **selectOut**(*outIds*)
>
> > Just complain for now

See Also:

Derived classes might provide additional methods via their base classes. Please refer to the list of base classes (if it exists) at the begining of the `BoxcarMapper` documentation.

Full API documentation of BoxcarMapper in module mvpa.mappers.boxcar.

16.3.5 `mvpa.mappers.ica`

The comprehensive API documentation for this module, including all technical details, is available in the Epydoc-generated API reference for mvpa.mappers.ica (for developers).

`ICAMapper`

See Also:

Derived classes might provide additional methods via their base classes. Please refer to the list of base classes (if it exists) at the begining of the `ICAMapper` documentation.

Full API documentation of ICAMapper in module mvpa.mappers.ica.

16.3.6 `mvpa.mappers.mask`

Data mapper which applies mask to the data

The comprehensive API documentation for this module, including all technical details, is available in the Epydoc-generated API reference for mvpa.mappers.mask (for developers).

MaskMapper

class MaskMapper(*mask, **kwargs*)

      Bases: `mvpa.mappers.base.Mapper`

      Mapper which uses a binary mask to select "Features"

      Initialize MaskMapper

              •*mask* (array) – an array in the original dataspace and its nonzero elements are used to define the features included in the dataset

      convertOutIds2InMask(*outIds*)

            Returns a boolean mask with all features in *ouIds* selected.

            This method works exactly like Mapper.convertOutIds2OutMask(), but the feature mask is finally (reverse) mapped into in-space.

                •*outIds* (list or 1d array) – To be selected features ids in out-space.

            Return type

                ndarray

            Returns

                All selected features are set to True; False otherwise.

      convertOutIds2OutMask(*outIds*)

            Returns a boolean mask with all features in *outIds* selected.

                •*outIds* (list or 1d array) – To be selected features ids in out-space.

            Return type

                ndarray

            Returns

                All selected features are set to True; False otherwise.

      discardOut(*outIds*)

            Listed outIds would be discarded

      forward(*data*)

            Map data from the original dataspace into featurespace.

      getInId(*outId*)

            Returns a features coordinate in the original data space for a given feature id.

            If this method is called with a list of feature ids it returns a 2d-array where the first axis corresponds the dimensions in 'In' dataspace and along the second axis are the coordinates of the features on this dimension (like the output of NumPy.array.nonzero()).

            XXX it might become __get_item__ access method

      getInIds()

            Returns a 2d array where each row contains the coordinate of the feature with the corresponding id.

      getInSize()

            InShape is a shape of original mask

      getMask(*copy=True*)

            By default returns a copy of the current mask.

            If 'copy' is set to False a reference to the mask is returned instead. This shared mask must not be modified!

      getOutId(*coord*)

            Translate a feature mask coordinate into a feature ID.

      getOutSize()

            OutSize is a number of non-0 elements in the mask

      isValidInId(*inId*)

      mask

      reverse(*data*)

            Reverse map data from featurespace into the original dataspace.

selectOut (*outIds*)

    Only listed outIds would remain.

    Function assumes that outIds are sorted. In __debug__ mode selectOut would check if obtained IDs are sorted and would warn the user if they are not.

    Note: If you feel strongly that you need to remap features internally (ie to allow Ids with mixed order) please contact developers of mvpa to discuss your use case.

    The function used to accept a matrix-mask as the input but now it really has to be a list of IDs

        •Negative outIds would not raise exception - just would be treated 'from the tail'

See Also:

Derived classes might provide additional methods via their base classes. Please refer to the list of base classes (if it exists) at the begining of the `MaskMapper` documentation.

Full API documentation of MaskMapper in module mvpa.mappers.mask.

16.3.7 `mvpa.mappers.metric`

Classes and functions to provide sense of distances between sample points

The comprehensive API documentation for this module, including all technical details, is available in the Epydoc-generated API reference for mvpa.mappers.metric (for developers).

Classes

DescreteMetric

class DescreteMetric (*elementsize=1, distance_function=<function cartesianDistance at 0x8fa7e9c>*)

    Bases: `mvpa.mappers.metric.Metric`

    Find neighboring points in discretized space

    If input space is discretized and all points fill in N-dimensional cube, this finder returns list of neighboring points for a given distance.

    As input points it operates on discretized values, not absolute coordinates (which are e.g. in mm)

    Initialize the class provided @elementsize and @distance_function

    elementsize

    filter_coord

        Lets allow to specify some custom filter to use

    getNeighbors (*origin, radius=0*)

        Returns coordinates of the neighbors which are within distance from coord

        XXX radius might need to be not a scalar but a vector of scalars to specify search distance in different dimensions differently... but then may be it has to be a tensor to specify orientation etc? :-) so it might not be necessary for now

See Also:

Derived classes might provide additional methods via their base classes. Please refer to the list of base classes (if it exists) at the begining of the `DescreteMetric` documentation.

Full API documentation of DescreteMetric in module mvpa.mappers.metric.

Metric

class Metric ()

    Bases: `object`

    Abstract class for any finder.

Classes subclasses from this class show know about structure of the data and thus be able to provide information about the neighbors. At least one of the methods (getNeighbors, getNeighbor) has to be overriden in the derived class. NOTE: derived #2 from derived class #1 has to override all methods which were overrident in class #1

getNeighbor (**args, **kwargs*)
> Generator to return coordinate of the neighbor.

> Base class contains the simplest implementation, assuming that getNeighbors returns iterative structure to spit out neighbors 1-by-1

getNeighbors (**args, **kwargs*)
> Return the list of coordinates for the neighbors.

> By default it simply constracts the list based on the generator getNeighbor

See Also:

Derived classes might provide additional methods via their base classes. Please refer to the list of base classes (if it exists) at the begining of the `Metric` documentation.

Full API documentation of Metric in module mvpa.mappers.metric.

16.3.8 `mvpa.mappers.pca`

The comprehensive API documentation for this module, including all technical details, is available in the Epydoc-generated API reference for mvpa.mappers.pca (for developers).

`PCAMapper`

See Also:

Derived classes might provide additional methods via their base classes. Please refer to the list of base classes (if it exists) at the begining of the `PCAMapper` documentation.

Full API documentation of PCAMapper in module mvpa.mappers.pca.

16.3.9 `mvpa.mappers.samplegroup`

Data mapper

The comprehensive API documentation for this module, including all technical details, is available in the Epydoc-generated API reference for mvpa.mappers.samplegroup (for developers).

`SampleGroupMapper`

class SampleGroupMapper (*fx=<function FirstAxisMean at 0x8d1be2c>*)
> Bases: `mvpa.mappers.base.Mapper`

> Mapper to apply a mapping function to samples of the same type.

> A customimzable function is applied individually to all samples with the same unique label from the same chunk. This mapper is somewhat unconventional since it doesn't preserve number of samples (ie the size of 0-th dimension...)

> Initialize the PCAMapper

>> startpoints: A sequence of index value along the first axis of
>>> 'data'.

>> boxlength: The number of elements after 'startpoint' along the
>>> first axis of 'data' to be considered for averaging.

>> offset: The offset between the starting point and the
>>> averaging window (boxcar).

collision_resolution
[string] if a sample belonged to multiple output samples, then on reverse, how to resolve the value (choices: 'mean')

forward(*data*)

getInSize()
Returns the number of original samples which were combined.

getOutSize()
Returns the number of output samples.

reverse(*data*)
This is not implemented.

selectOut(*outIds*)
Just complain for now

train(*dataset*)

See Also:

Derived classes might provide additional methods via their base classes. Please refer to the list of base classes (if it exists) at the begining of the `SampleGroupMapper` documentation.

Full API documentation of SampleGroupMapper in module mvpa.mappers.samplegroup.

16.3.10 `mvpa.mappers.svd`

Data mapper

The comprehensive API documentation for this module, including all technical details, is available in the Epydoc-generated API reference for mvpa.mappers.svd (for developers).

SVDMapper

class **SVDMapper**(***kwargs*)
Bases: `mvpa.mappers.base.ProjectionMapper`

Mapper to project data onto SVD components estimated from some dataset.

Initialize the SVDMapper

•***kwargs* – All keyword arguments are passed to the ProjectionMapper constructor. Note, that for the 'selector' argument this class also supports passing a *ElementSelector* instance, which will be used to determine the to be selected features, based on the singular values of each component.

selectOut(*outIds*)
Choose a subset of SVD components (and remove all others).

sv
Singular values

See Also:

Derived classes might provide additional methods via their base classes. Please refer to the list of base classes (if it exists) at the begining of the `SVDMapper` documentation.

Full API documentation of SVDMapper in module mvpa.mappers.svd.

16.3.11 `mvpa.mappers.wavelet`

The comprehensive API documentation for this module, including all technical details, is available in the Epydoc-generated API reference for mvpa.mappers.wavelet (for developers).

Classes

`WaveletPacketMapper`

See Also:

Derived classes might provide additional methods via their base classes. Please refer to the list of base classes (if it exists) at the begining of the `WaveletPacketMapper` documentation.

Full API documentation of WaveletPacketMapper in module mvpa.mappers.wavelet.

`WaveletTransformationMapper`

See Also:

Derived classes might provide additional methods via their base classes. Please refer to the list of base classes (if it exists) at the begining of the `WaveletTransformationMapper` documentation.

Full API documentation of WaveletTransformationMapper in module mvpa.mappers.wavelet.

16.4 Classifiers and Errors

16.4.1 `mvpa.clfs`

Import helper for PyMVPA classifiers

Module Organization

mvpa.clfs module contains various classifiers

> group Basic
> > classifier
>
> group Specific Implementations
> > knn svm plr ridge smlr
>
> group Internal Implementations
> > libsvm
>
> group Utilities
> > transerror

The comprehensive API documentation for this module, including all technical details, is available in the Epydoc-generated API reference for mvpa.clfs (for developers).

16.4.2 `mvpa.clfs.base`

Base classes for all classifiers.

Base Classifiers can be grouped according to their function as

> group Basic Classifiers
> > Classifier BoostedClassifier ProxyClassifier
>
> group BoostedClassifiers
> > CombinedClassifier MulticlassClassifier SplitClassifier
>
> group ProxyClassifiers
> > BinaryClassifier MappedClassifier FeatureSelectionClassifier

group PredictionsCombiners for CombinedClassifier
    PredictionsCombiner MaximalVote MeanPrediction

The comprehensive API documentation for this module, including all technical details, is available in the Epydoc-generated API reference for mvpa.clfs.base (for developers).

Classes

BinaryClassifier

class **BinaryClassifier**(*clf, poslabels, neglabels, **kwargs*)
    Bases: `mvpa.clfs.base.ProxyClassifier`

    ProxyClassifier which maps set of two labels into +1 and -1

> •*clf* (Classifier) – classifier to use
> •*poslabels* (list) – list of labels which are treated as +1 category
> •*neglabels* (list) – list of labels which are treated as -1 category

See Also:

Derived classes might provide additional methods via their base classes. Please refer to the list of base classes (if it exists) at the begining of the `BinaryClassifier` documentation.

Full API documentation of BinaryClassifier in module mvpa.clfs.base.

BoostedClassifier

class **BoostedClassifier**(*clfs=None, propagate_states=True, harvest_attribs=None, copy_attribs='copy', **kwargs*)
    Bases: `mvpa.clfs.base.Classifier`, `mvpa.misc.state.Harvestable`

    Classifier containing the farm of other classifiers.

    Should rarely be used directly. Use one of its childs instead

    Initialize the instance.

> •*clfs* (list) – list of classifier instances to use (slave classifiers)
> •*propagate_states* (bool) – either to propagate enabled states into slave classifiers. It is in effect only when slaves get assigned - so if state is enabled not during construction, it would not necessarily propagate into slaves
> •*harvest_attribs* (list of basestr) – What attributes of call to store and return within harvested state variable
> •*copy_attribs* (None or basestr) – Force copying values of attributes on harvesting
> •*kwargs* (dict) – dict of keyworded arguments which might get used by State or Classifier

clfs
    Used classifiers

getSensitivityAnalyzer(*** kwargs*)
    Return an appropriate SensitivityAnalyzer

untrain()
    Untrain *BoostedClassifier*
    Has to untrain any known classifier

See Also:

Derived classes might provide additional methods via their base classes. Please refer to the list of base classes (if it exists) at the begining of the `BoostedClassifier` documentation.

Full API documentation of BoostedClassifier in module mvpa.clfs.base.

Classifier

class Classifier(***kwargs*)

> Bases: `mvpa.misc.state.ClassWithCollections`
>
> Abstract classifier class to be inherited by all classifiers
>
> Cheap initialization.
>
> **getSensitivityAnalyzer**(***kwargs*)
>
> > Factory method to return an appropriate sensitivity analyzer for the respective classifier.
>
> **isTrained**(*dataset=None*)
>
> > Either classifier was already trained.
> >
> > MUST BE USED WITH CARE IF EVER
>
> **predict**(*data*)
>
> > Predict classifier on data
> >
> > Shouldn't be overriden in subclasses unless explicitely needed to do so. Also subclasses trying to call super class's predict should call _predict if within _predict instead of predict() since otherwise it would loop
>
> **repredict**(*data, **kwargs*)
>
> > Helper to avoid check if data was changed actually changed
> >
> > Useful if classifier was (re)trained but with the same data (so just parameters were changed), so that it could be repredicted easily (on the same data as before) without recomputing for instance train/test kernel matrix. Should be used with caution and always compared to the results on not 'retrainable' classifier. Some additional checks are enabled if debug id 'CHECK_RETRAIN' is enabled, to guard against obvious mistakes.
> >
> > > •*data* – data which is conventionally given to predict
> > >
> > > •*kwargs* – that is what _changedData gets updated with. So, smth like (params=['C'], labels=True) if parameter C and labels got changed
>
> **retrain**(*dataset, **kwargs*)
>
> > Helper to avoid check if data was changed actually changed
> >
> > Useful if just some aspects of classifier were changed since its previous training. For instance if dataset wasn't changed but only classifier parameters, then kernel matrix does not have to be computed.
> >
> > Words of caution: classifier must be previously trained, results always should first be compared to the results on not 'retrainable' classifier (without calling retrain). Some additional checks are enabled if debug id 'CHECK_RETRAIN' is enabled, to guard against obvious mistakes.
> >
> > > •*kwargs* – that is what _changedData gets updated with. So, smth like (params=['C'], labels=True) if parameter C and labels got changed
>
> **summary**()
>
> > Providing summary over the classifier
>
> **train**(*dataset*)
>
> > Train classifier on a dataset
> >
> > Shouldn't be overriden in subclasses unless explicitely needed to do so
>
> **trained**
>
> > Either classifier was already trained
>
> **untrain**()
>
> > Reset trained state

See Also:

Derived classes might provide additional methods via their base classes. Please refer to the list of base classes (if it exists) at the begining of the `Classifier` documentation.

Full API documentation of Classifier in module mvpa.clfs.base.

ClassifierCombiner

class ClassifierCombiner(*clf, variables=None*)

    Bases: `mvpa.clfs.base.PredictionsCombiner`

    Provides a decision using training a classifier on predictions/values

    TODO

    Initialize *ClassifierCombiner*

> - *clf* (Classifier) – Classifier to train on the predictions
> - *variables* (list of basestring) – List of state variables stored in 'combined' classifiers, which to use as features for training this classifier

    untrain()

        It might be needed to untrain used classifier

See Also:

Derived classes might provide additional methods via their base classes. Please refer to the list of base classes (if it exists) at the begining of the `ClassifierCombiner` documentation.

Full API documentation of ClassifierCombiner in module mvpa.clfs.base.

CombinedClassifier

class CombinedClassifier(*clfs=None, combiner=None, **kwargs*)

    Bases: `mvpa.clfs.base.BoostedClassifier`

    BoostedClassifier which combines predictions using some *PredictionsCombiner* functor.

    Initialize the instance.

> - *clfs* (list of Classifier) – list of classifier instances to use
> - *combiner* (PredictionsCombiner) – callable which takes care about combining multiple results into a single one (e.g. maximal vote for classification, MeanPrediction for regression))
> - *kwargs* (dict) – dict of keyworded arguments which might get used by State or Classifier

    NB: *combiner* might need to operate not on 'predictions' descrete
        labels but rather on raw 'class' values classifiers estimate (which is pretty much what is stored under *values*

    combiner

        Used combiner to derive a single result

    summary()

    untrain()

        Untrain *CombinedClassifier*

See Also:

Derived classes might provide additional methods via their base classes. Please refer to the list of base classes (if it exists) at the begining of the `CombinedClassifier` documentation.

Full API documentation of CombinedClassifier in module mvpa.clfs.base.

FeatureSelectionClassifier

class FeatureSelectionClassifier(*clf, feature_selection, testdataset=None, **kwargs*)

    Bases: `mvpa.clfs.base.ProxyClassifier`

    ProxyClassifier which uses some *FeatureSelection* prior training.

FeatureSelection is used first to select features for the classifier to use for prediction. Internally it would rely on MappedClassifier which would use created MaskMapper.

TODO: think about removing overhead of retraining the same classifier if feature selection was carried out with the same classifier already. It has been addressed by adding .trained property to classifier, but now we should expclitely use isTrained here if we want... need to think more

Initialize the instance

> •*clf* (Classifier) – classifier based on which mask classifiers is created
>
> •*feature_selection* (FeatureSelection) – whatever *FeatureSelection* comes handy
>
> •*testdataset* (Dataset) – optional dataset which would be given on call to feature_selection

feature_selection
> Used *FeatureSelection*

getSensitivityAnalyzer(*args_, **kwargs_*)

maskclf
> Used *MappedClassifier*

setTestDataset(*testdataset*)
> Set testing dataset to be used for feature selection

testdataset

untrain()
> Untrain *FeatureSelectionClassifier*
>
> Has to untrain any known classifier

See Also:

Derived classes might provide additional methods via their base classes. Please refer to the list of base classes (if it exists) at the begining of the FeatureSelectionClassifier documentation.

Full API documentation of FeatureSelectionClassifier in module mvpa.clfs.base.

MappedClassifier

class MappedClassifier(*clf, mapper, **kwargs*)
> Bases: mvpa.clfs.base.ProxyClassifier
>
> *ProxyClassifier* which uses some mapper prior training/testing.
>
> *MaskMapper* can be used just a subset of features to train/classify. Having such classifier we can easily create a set of classifiers for BoostedClassifier, where each classifier operates on some set of features, e.g. set of best spheres from SearchLight, set of ROIs selected elsewhere. It would be different from simply applying whole mask over the dataset, since here initial decision is made by each classifier and then later on they vote for the final decision across the set of classifiers.
>
> Initialize the instance
>
> > •*clf* (Classifier) – classifier based on which mask classifiers is created
> >
> > •*mapper* – whatever *Mapper* comes handy
>
> **getSensitivityAnalyzer**(*args_, **kwargs_*)
>
> **mapper**
> > Used mapper

See Also:

Derived classes might provide additional methods via their base classes. Please refer to the list of base classes (if it exists) at the begining of the MappedClassifier documentation.

Full API documentation of MappedClassifier in module mvpa.clfs.base.

MaximalVote

class **MaximalVote**()
> Bases: `mvpa.clfs.base.PredictionsCombiner`

> Provides a decision using maximal vote rule

> XXX Might get a parameter to use raw decision values if voting is not unambigous (ie two classes have equal number of votes

See Also:

Derived classes might provide additional methods via their base classes. Please refer to the list of base classes (if it exists) at the begining of the `MaximalVote` documentation.

Full API documentation of MaximalVote in module mvpa.clfs.base.

MeanPrediction

class **MeanPrediction**(*descr=None, **kwargs*)
> Bases: `mvpa.clfs.base.PredictionsCombiner`

> Provides a decision by taking mean of the results

See Also:

Derived classes might provide additional methods via their base classes. Please refer to the list of base classes (if it exists) at the begining of the `MeanPrediction` documentation.

Full API documentation of MeanPrediction in module mvpa.clfs.base.

MulticlassClassifier

class **MulticlassClassifier**(*clf, bclf_type='1-vs-1', **kwargs*)
> Bases: `mvpa.clfs.base.CombinedClassifier`

> *CombinedClassifier* to perform multiclass using a list of *BinaryClassifier*.

> such as 1-vs-1 (ie in pairs like libsvm doesn) or 1-vs-all (which is yet to think about)

> Initialize the instance

>> • *clf* (Classifier) – classifier based on which multiple classifiers are created for multiclass
>> • *bclf_type* – "1-vs-1" or "1-vs-all", determines the way to generate binary classifiers

See Also:

Derived classes might provide additional methods via their base classes. Please refer to the list of base classes (if it exists) at the begining of the `MulticlassClassifier` documentation.

Full API documentation of MulticlassClassifier in module mvpa.clfs.base.

PredictionsCombiner

class **PredictionsCombiner**(*descr=None, **kwargs*)
> Bases: `mvpa.misc.state.ClassWithCollections`

> Base class for combining decisions of multiple classifiers

> **train**(*clfs, dataset*)
>> PredictionsCombiner might need to be trained

>>> • *clfs* (list of Classifier) – List of classifiers to combine. Has to be classifiers (not pure predictions), since combiner might use some other state variables (value's) instead of pure prediction's

•*dataset* (Dataset) – training data in this case

See Also:

Derived classes might provide additional methods via their base classes. Please refer to the list of base classes (if it exists) at the begining of the `PredictionsCombiner` documentation.

Full API documentation of PredictionsCombiner in module mvpa.clfs.base.

ProxyClassifier

class **ProxyClassifier**(*clf, **kwargs*)

Bases: `mvpa.clfs.base.Classifier`

Classifier which decorates another classifier

Possible uses:

•modify data somehow prior training/testing: * normalization * feature selection * modification

•optimized classifier?

Initialize the instance

•*clf* (Classifier) – classifier based on which mask classifiers is created

clf

Used *Classifier*

getSensitivityAnalyzer(**args_, **kwargs_*)

summary()

untrain()

Untrain ProxyClassifier

See Also:

Derived classes might provide additional methods via their base classes. Please refer to the list of base classes (if it exists) at the begining of the `ProxyClassifier` documentation.

Full API documentation of ProxyClassifier in module mvpa.clfs.base.

SplitClassifier

class **SplitClassifier**(*clf,   splitter=<mvpa.datasets.splitter.NFoldSplitter   object   at   0x8cc072c>,   **kwargs*)

Bases: `mvpa.clfs.base.CombinedClassifier`

BoostedClassifier to work on splits of the data

Initialize the instance

•*clf* (Classifier) – classifier based on which multiple classifiers are created for multiclass

•*splitter* (Splitter) – *Splitter* to use to split the dataset prior training

getSensitivityAnalyzer(**args_, **kwargs_*)

splitter

Splitter user by SplitClassifier

See Also:

Derived classes might provide additional methods via their base classes. Please refer to the list of base classes (if it exists) at the begining of the `SplitClassifier` documentation.

Full API documentation of SplitClassifier in module mvpa.clfs.base.

16.4.3 `mvpa.clfs.blr`

Bayesian Linear Regression (BLR).

The comprehensive API documentation for this module, including all technical details, is available in the Epydoc-generated API reference for mvpa.clfs.blr (for developers).

BLR

class **BLR** (*sigma_p=None, sigma_noise=1.0, **kwargs*)

Bases: `mvpa.clfs.base.Classifier`

Bayesian Linear Regression (BLR).

Initialize a BLR regression analysis.

• *sigma_noise* (float) – the standard deviation of the gaussian noise. (Defaults to 0.1)

compute_log_marginal_likelihood ()

Compute log marginal likelihood using self.train_fv and self.labels.

set_hyperparameters (**args*)

Set hyperparameters' values.

Note that this is a list so the order of the values is important.

See Also:

Derived classes might provide additional methods via their base classes. Please refer to the list of base classes (if it exists) at the begining of the BLR documentation.

Full API documentation of BLR in module mvpa.clfs.blr.

16.4.4 `mvpa.clfs.distance`

Distance functions to be used in kernels and elsewhere

The comprehensive API documentation for this module, including all technical details, is available in the Epydoc-generated API reference for mvpa.clfs.distance (for developers).

Functions

absminDistance (*a, b*)

Returns dinstance max(|a-b|) XXX There must be better name!

Useful to select a whole cube of a given "radius"

See Also:

Full API documentation of absminDistance() in module mvpa.clfs.distance.

cartesianDistance (*a, b*)

Return Cartesian distance between a and b

See Also:

Full API documentation of cartesianDistance() in module mvpa.clfs.distance.

mahalanobisDistance (*x, y=None, w=None*)

Caclulcate Mahalanobis distance of the pairs of points.

• *x* – first list of points. Rows are samples, columns are features.

• *y* – second list of points (optional)

• *w* (N.ndarray) – optional inverse covariance matrix between the points. It is computed if not given

Inverse covariance matrix can be calculated with the following

w = N.linalg.solve(N.cov(x.T), N.identity(x.shape[1]))

or

w = N.linalg.inv(N.cov(x.T))

See Also:

Full API documentation of mahalanobisDistance() in module mvpa.clfs.distance.

manhattenDistance(*a, b*)
    Return Manhatten distance between a and b

See Also:

Full API documentation of manhattenDistance() in module mvpa.clfs.distance.

pnorm_w_python(*data1, data2=None, weight=None, p=2, heuristic='auto', use_sq_euclidean=True*)
    Weighted p-norm between two datasets (pure Python implementation)

$\|x - x'\|_w = (sum_{i=1...N} (w_i*|x_i - x'_i|)**p)**(1/p)$

- *data1* (N.ndarray) – First dataset
- *data2* (N.ndarray or None) – Optional second dataset
- *weight* (N.ndarray or None) – Optional weights per 2nd dimension (features)
- *p* – Power
- *heuristic* (basestring) – Which heuristic to use: * 'samples' – python sweep over 0th dim * 'features' – python sweep over 1st dim * 'auto' decides automatically. If # of features (shape[1]) is much larger than # of samples (shape[0]) – use 'samples', and use 'features' otherwise
- *use_sq_euclidean* (bool) – Either to use squared_euclidean_distance_matrix for computation if p==2

See Also:

Full API documentation of pnorm_w_python() in module mvpa.clfs.distance.

squared_euclidean_distance(*data1, data2=None, weight=None*)
    Compute weighted euclidean distance matrix between two datasets.

- *data1* (N.ndarray) – first dataset
- *data2* (N.ndarray) – second dataset. If None, compute the euclidean distance between the first dataset versus itself. (Defaults to None)
- *weight* (N.ndarray) – vector of weights, each one associated to each dimension of the dataset (Defaults to None)

See Also:

Full API documentation of squared_euclidean_distance() in module mvpa.clfs.distance.

16.4.5 `mvpa.clfs.gpr`

The comprehensive API documentation for this module, including all technical details, is available in the Epydoc-generated API reference for mvpa.clfs.gpr (for developers).

Classes

GPR

See Also:

Derived classes might provide additional methods via their base classes. Please refer to the list of base classes (if it exists) at the begining of the `GPR` documentation.

Full API documentation of GPR in module mvpa.clfs.gpr.

GPRLinearWeights

See Also:

Derived classes might provide additional methods via their base classes. Please refer to the list of base classes (if it exists) at the begining of the GPRLinearWeights documentation.

Full API documentation of GPRLinearWeights in module mvpa.clfs.gpr.

16.4.6 `mvpa.clfs.kernel`

Kernels for Gaussian Process Regression and Classification.

The comprehensive API documentation for this module, including all technical details, is available in the Epydoc-generated API reference for mvpa.clfs.kernel (for developers).

Classes

Kernel

class Kernel()
    Bases: `object`

    Kernel function base class.

    compute(*data1, data2=None*)

    compute_gradient(*alphaalphaTK*)

    compute_lml_gradient(*alphaalphaT_Kinv, data*)

    compute_lml_gradient_logscale(*alphaalphaT_Kinv, data*)

    reset()
        Resets the kernel dropping internal variables to the original values

See Also:

Derived classes might provide additional methods via their base classes. Please refer to the list of base classes (if it exists) at the begining of the `Kernel` documentation.

Full API documentation of Kernel in module mvpa.clfs.kernel.

KernelConstant

class KernelConstant(*sigma_0=1.0, **kwargs*)
    Bases: `mvpa.clfs.kernel.Kernel`

    The constant kernel class.

    Initialize the constant kernel instance.

            •*sigma_0* (float) – standard deviation of the Gaussian prior probability N(0,sigma_0**2) of the intercept of the constant regression. (Defaults to 1.0)

    compute(*data1, data2=None*)
        Compute kernel matrix.

            •*data1* (numpy.ndarray) – data
            •*data2* (numpy.ndarray) – data (Defaults to None)

    compute_lml_gradient(*alphaalphaT_Kinv, data*)

    compute_lml_gradient_logscale(*alphaalphaT_Kinv, data*)

    set_hyperparameters(*hyperparameter*)

See Also:

Derived classes might provide additional methods via their base classes. Please refer to the list of base classes (if it exists) at the begining of the KernelConstant documentation.

Full API documentation of KernelConstant in module mvpa.clfs.kernel.

KernelExponential

class **KernelExponential**(*length_scale=1.0, sigma_f=1.0, **kwargs*)

Bases: mvpa.clfs.kernel.Kernel

The Exponential kernel class.

Note that it can handle a length scale for each dimension for Automtic Relevance Determination.

Initialize an Exponential kernel instance.

- *length_scale* (float OR numpy.ndarray) – the characteristic length-scale (or length-scales) of the phenomenon under investigation. (Defaults to 1.0)
- *sigma_f* (float) – Signal standard deviation. (Defaults to 1.0)

compute(*data1, data2=None*)
Compute kernel matrix.

- *data1* (numpy.ndarray) – data
- *data2* (numpy.ndarray) – data (Defaults to None)

compute_lml_gradient(*alphaalphaT_Kinv, data*)
Compute grandient of the kernel and return the portion of log marginal likelihood gradient due to the kernel. Shorter formula. Allows vector of lengthscales (ARD) BUT THIS LAST OPTION SEEMS NOT TO WORK FOR (CURRENTLY) UNKNOWN REASONS.

compute_lml_gradient_logscale(*alphaalphaT_Kinv, data*)
Compute grandient of the kernel and return the portion of log marginal likelihood gradient due to the kernel. Shorter formula. Allows vector of lengthscales (ARD). BUT THIS LAST OPTION SEEMS NOT TO WORK FOR (CURRENTLY) UNKNOWN REASONS.

gradient(*data1, data2*)
Compute gradient of the kernel matrix. A must for fast model selection with high-dimensional data.

set_hyperparameters(*hyperparameter*)
Set hyperaparmeters from a vector.

Used by model selection.

See Also:

Derived classes might provide additional methods via their base classes. Please refer to the list of base classes (if it exists) at the begining of the KernelExponential documentation.

Full API documentation of KernelExponential in module mvpa.clfs.kernel.

KernelLinear

class **KernelLinear**(*Sigma_p=None, sigma_0=1.0, **kwargs*)

Bases: mvpa.clfs.kernel.Kernel

The linear kernel class.

Initialize the linear kernel instance.

- *Sigma_p* (numpy.ndarray) – Covariance matrix of the Gaussian prior probability N(0,Sigma_p) on the weights of the linear regression. (Defaults to None)
- *sigma_0* (float) – the standard deviation of the Gaussian prior N(0,sigma_0**2) of the intercept of the linear regression. (Deafults to 1.0)

compute (*data1, data2=None*)
> Compute kernel matrix. Set Sigma_p to correct dimensions and default value if necessary.

> > •*data1* (numpy.ndarray) – data

> > •*data2* (numpy.ndarray) – data (Defaults to None)

compute_lml_gradient (*alphaalphaT_Kinv, data*)

compute_lml_gradient_logscale (*alphaalphaT_Kinv, data*)

reset ()

set_hyperparameters (*hyperparameter*)

See Also:

Derived classes might provide additional methods via their base classes. Please refer to the list of base classes (if it exists) at the begining of the KernelLinear documentation.

Full API documentation of KernelLinear in module mvpa.clfs.kernel.

KernelMatern_3_2

class **KernelMatern_3_2** (*length_scale=1.0, sigma_f=1.0, numerator=3.0, **kwargs*)
> Bases: mvpa.clfs.kernel.Kernel

> The Matern kernel class for the case ni=3/2 or ni=5/2.

> Note that it can handle a length scale for each dimension for Automtic Relevance Determination.

> Initialize a Squared Exponential kernel instance.

> > •*length_scale* (float OR numpy.ndarray) – the characteristic length-scale (or length-scales) of the phenomenon under investigation. (Defaults to 1.0)

> > •*sigma_f* (float) – Signal standard deviation. (Defaults to 1.0)

> > •*numerator* (float) – the numerator of parameter ni of Matern covariance functions. Currently only numerator=3.0 and numerator=5.0 are implemented. (Defaults to 3.0)

compute (*data1, data2=None*)
> Compute kernel matrix.

> > •*data1* (numpy.ndarray) – data

> > •*data2* (numpy.ndarray) – data (Defaults to None)

gradient (*data1, data2*)
> Compute gradient of the kernel matrix. A must for fast model selection with high-dimensional data.

set_hyperparameters (*hyperparameter*)
> Set hyperaparmeters from a vector.

> Used by model selection. Note: 'numerator' is not considered as an hyperparameter.

See Also:

Derived classes might provide additional methods via their base classes. Please refer to the list of base classes (if it exists) at the begining of the KernelMatern_3_2 documentation.

Full API documentation of KernelMatern_3_2 in module mvpa.clfs.kernel.

KernelMatern_5_2

class **KernelMatern_5_2** (***kwargs*)
> Bases: mvpa.clfs.kernel.KernelMatern_3_2

> The Matern kernel class for the case ni=5/2.

> This kernel is just KernelMatern_3_2(numerator=5.0).

> Initialize a Squared Exponential kernel instance.

•*length_scale* (float OR numpy.ndarray) – the characteristic length-scale (or length-scales) of the phenomenon under investigation. (Defaults to 1.0)

See Also:

Derived classes might provide additional methods via their base classes. Please refer to the list of base classes (if it exists) at the begining of the `KernelMatern_5_2` documentation.

Full API documentation of KernelMatern_5_2 in module mvpa.clfs.kernel.

KernelRationalQuadratic

class **KernelRationalQuadratic**(*length_scale=1.0, sigma_f=1.0, alpha=0.5, **kwargs*)

Bases: `mvpa.clfs.kernel.Kernel`

The Rational Quadratic (RQ) kernel class.

Note that it can handle a length scale for each dimension for Automtic Relevance Determination.

Initialize a Squared Exponential kernel instance.

•*length_scale* (float OR numpy.ndarray) – the characteristic length-scale (or length-scales) of the phenomenon under investigation. (Defaults to 1.0)

•*sigma_f* (float) – Signal standard deviation. (Defaults to 1.0)

•*alpha* (float) – The parameter of the RQ functions family. (Defaults to 2.0)

compute(*data1, data2=None*)

Compute kernel matrix.

•*data1* (numpy.ndarray) – data

•*data2* (numpy.ndarray) – data (Defaults to None)

gradient(*data1, data2*)

Compute gradient of the kernel matrix. A must for fast model selection with high-dimensional data.

set_hyperparameters(*hyperparameter*)

Set hyperaparmeters from a vector.

Used by model selection. Note: 'alpha' is not considered as an hyperparameter.

See Also:

Derived classes might provide additional methods via their base classes. Please refer to the list of base classes (if it exists) at the begining of the `KernelRationalQuadratic` documentation.

Full API documentation of KernelRationalQuadratic in module mvpa.clfs.kernel.

KernelSquaredExponential

class **KernelSquaredExponential**(*length_scale=1.0, sigma_f=1.0, **kwargs*)

Bases: `mvpa.clfs.kernel.Kernel`

The Squared Exponential kernel class.

Note that it can handle a length scale for each dimension for Automtic Relevance Determination.

Initialize a Squared Exponential kernel instance.

•*length_scale* (float OR numpy.ndarray) – the characteristic length-scale (or length-scales) of the phenomenon under investigation. (Defaults to 1.0)

•*sigma_f* (float) – Signal standard deviation. (Defaults to 1.0)

compute(*data1, data2=None*)

Compute kernel matrix.

•*data1* (numpy.ndarray) – data

•*data2* (numpy.ndarray) – data (Defaults to None)

compute_lml_gradient (*alphaalphaT_Kinv, data*)
> Compute grandient of the kernel and return the portion of log marginal likelihood gradient due to the kernel. Shorter formula. Allows vector of lengthscales (ARD).

compute_lml_gradient_logscale (*alphaalphaT_Kinv, data*)
> Compute grandient of the kernel and return the portion of log marginal likelihood gradient due to the kernel. Hyperparameters are in log scale which is sometimes more stable. Shorter formula. Allows vector of lengthscales (ARD).

reset ()

set_hyperparameters (*hyperparameter*)
> Set hyperaparmeters from a vector.
>
> Used by model selection.

See Also:

Derived classes might provide additional methods via their base classes. Please refer to the list of base classes (if it exists) at the begining of the KernelSquaredExponential documentation.

Full API documentation of KernelSquaredExponential in module mvpa.clfs.kernel.

16.4.7 `mvpa.clfs.knn`

k-Nearest-Neighbour classifier.

The comprehensive API documentation for this module, including all technical details, is available in the Epydoc-generated API reference for mvpa.clfs.knn (for developers).

kNN

class kNN (*k=2, dfx=<function squared_euclidean_distance at 0x8fa7f7c>, voting='weighted', **kwargs*)
> Bases: mvpa.clfs.base.Classifier
>
> k-nearest-neighbour classifier.
>
> If enabled, it stores the votes per class in the 'values' state after calling predict().
>
> > •*k* (unsigned integer) – Number of nearest neighbours to be used for voting.
> > •*dfx* (functor) – Function to compute the distances between training and test samples. Default: squared euclidean distance
> > •*voting* (str) – Voting method used to derive predictions from the nearest neighbors. Possible values are 'majority' (simple majority of classes determines vote) and 'weighted' (votes are weighted according to the relative frequencies of each class in the training data).
> > •***kwargs* – Additonal arguments are passed to the base class.

getMajorityVote (*knn_ids*)
> Simple voting by choosing the majority of class neighbours.

getWeightedVote (*knn_ids*)
> Vote with classes weighted by the number of samples per class.

untrain ()
> Reset trained state

See Also:

Derived classes might provide additional methods via their base classes. Please refer to the list of base classes (if it exists) at the begining of the kNN documentation.

Full API documentation of kNN in module mvpa.clfs.knn.

16.4.8 `mvpa.clfs.lars`

The comprehensive API documentation for this module, including all technical details, is available in the Epydoc-generated API reference for mvpa.clfs.lars (for developers).

Classes

LARS

See Also:

Derived classes might provide additional methods via their base classes. Please refer to the list of base classes (if it exists) at the begining of the `LARS` documentation.

Full API documentation of LARS in module mvpa.clfs.lars.

LARSWeights

See Also:

Derived classes might provide additional methods via their base classes. Please refer to the list of base classes (if it exists) at the begining of the `LARSWeights` documentation.

Full API documentation of LARSWeights in module mvpa.clfs.lars.

16.4.9 `mvpa.clfs.libsmlrc`

Wraper for the stepwise_regression function for SMLR.

The comprehensive API documentation for this module, including all technical details, is available in the Epydoc-generated API reference for mvpa.clfs.libsmlrc (for developers).

stepwise_regression(**args*)

See Also:

Full API documentation of stepwise_regression() in module mvpa.clfs.libsmlrc.

16.4.10 `mvpa.clfs.libsmlrc.ctypes_helper`

Helpers for wrapping C libraries with ctypes.

The comprehensive API documentation for this module, including all technical details, is available in the Epydoc-generated API reference for mvpa.clfs.libsmlrc.ctypes_helper (for developers).

Functions

extend_args(**args*)
    Turn ndarray arguments into dims and arrays.

See Also:

Full API documentation of extend_args() in module mvpa.clfs.libsmlrc.ctypes_helper.

get_argtypes(**args*)

See Also:

Full API documentation of get_argtypes() in module mvpa.clfs.libsmlrc.ctypes_helper.

process_args(**args*)
    Turn ndarray arguments into dims and array pointers for calling a ctypes-wrapped function.

See Also:

Full API documentation of process_args() in module mvpa.clfs.libsmlrc.ctypes_helper.

16.4.11 `mvpa.clfs.libsvmc`

Classifiers provied by LibSVM library

The comprehensive API documentation for this module, including all technical details, is available in the Epydoc-generated API reference for mvpa.clfs.libsvmc (for developers).

16.4.12 `mvpa.clfs.libsvmc.sens`

Provide sensitivity measures for libsvm's SVM.

The comprehensive API documentation for this module, including all technical details, is available in the Epydoc-generated API reference for mvpa.clfs.libsvmc.sens (for developers).

`LinearSVMWeights`

class **LinearSVMWeights**(*clf, **kwargs*)

> Bases: `mvpa.measures.base.Sensitivity`
>
> *SensitivityAnalyzer* for the LIBSVM implementation of a linear SVM.
>
> Initialize the analyzer with the classifier it shall use.
>
> > •*clf* (LinearSVM) – classifier to use. Only classifiers sub-classed from *LinearSVM* may be used.

See Also:

Derived classes might provide additional methods via their base classes. Please refer to the list of base classes (if it exists) at the begining of the `LinearSVMWeights` documentation.

Full API documentation of LinearSVMWeights in module mvpa.clfs.libsvmc.sens.

16.4.13 `mvpa.clfs.libsvmc.svm`

Wrap the libsvm package into a very simple class interface.

The comprehensive API documentation for this module, including all technical details, is available in the Epydoc-generated API reference for mvpa.clfs.libsvmc.svm (for developers).

Classes

`LinearSVMWeights`

class **LinearSVMWeights**(*clf, **kwargs*)

> Bases: `mvpa.measures.base.Sensitivity`
>
> *SensitivityAnalyzer* for the LIBSVM implementation of a linear SVM.
>
> Initialize the analyzer with the classifier it shall use.
>
> > •*clf* (LinearSVM) – classifier to use. Only classifiers sub-classed from *LinearSVM* may be used.

See Also:

Derived classes might provide additional methods via their base classes. Please refer to the list of base classes (if it exists) at the begining of the `LinearSVMWeights` documentation.

Full API documentation of LinearSVMWeights in module mvpa.clfs.libsvmc.svm.

SVM

class SVM (*kernel_type='linear', **kwargs*)

> Bases: `mvpa.clfs._svmbase._SVM`

> Support Vector Machine Classifier.

> This is a simple interface to the libSVM package.

> This is the base class of all classifier that utilize the libSVM package underneath. It is not really meant to be used directly. Unless you know what you are doing it is most likely better to use one of the subclasses.

> Here is the explaination for some of the parameters from the libSVM documentation:

> svm_type can be one of C_SVC, NU_SVC, ONE_CLASS, EPSILON_SVR, NU_SVR.

>> •*C_SVC*: C-SVM classification

>> •*NU_SVC*: nu-SVM classification

>> •*ONE_CLASS*: one-class-SVM

>> •*EPSILON_SVR*: epsilon-SVM regression

>> •*NU_SVR*: nu-SVM regression

> kernel_type can be one of LINEAR, POLY, RBF, SIGMOID.

>> •*LINEAR*: `u'*v`

>> •*POLY*: `(gamma*u'*v + coef0)^degree`

>> •*RBF*: `exp(-gamma*|u-v|^2)`

>> •*SIGMOID*: `tanh(gamma*u'*v + coef0)`

>> •*PRECOMPUTED*: kernel values in training_set_file

> cache_size is the size of the kernel cache, specified in megabytes. C is the cost of constraints violation. (we usually use 1 to 1000) eps is the stopping criterion. (we usually use 0.00001 in nu-SVC, 0.001 in others). nu is the parameter in nu-SVM, nu-SVR, and one-class-SVM. p is the epsilon in epsilon-insensitive loss function of epsilon-SVM regression. shrinking = 1 means shrinking is conducted; = 0 otherwise. probability = 1 means model with probability information is obtained; = 0 otherwise.

> nr_weight, weight_label, and weight are used to change the penalty for some classes (If the weight for a class is not changed, it is set to 1). This is useful for training classifier using unbalanced input data or with asymmetric misclassification cost.

> Each weight[i] corresponds to weight_label[i], meaning that the penalty of class weight_label[i] is scaled by a factor of weight[i].

> If you do not want to change penalty for any of the classes, just set nr_weight to 0.

> **model**

> **summary** ()

>> Provide quick summary over the SVM classifier

> **untrain** ()

See Also:

Derived classes might provide additional methods via their base classes. Please refer to the list of base classes (if it exists) at the begining of the SVM documentation.

Full API documentation of SVM in module mvpa.clfs.libsvmc.svm.

16.4.14 `mvpa.clfs.model_selector`

The comprehensive API documentation for this module, including all technical details, is available in the Epydoc-generated API reference for mvpa.clfs.model_selector (for developers).

ModelSelector

See Also:

Derived classes might provide additional methods via their base classes. Please refer to the list of base classes (if it exists) at the begining of the `ModelSelector` documentation.

Full API documentation of ModelSelector in module mvpa.clfs.model_selector.

16.4.15 `mvpa.clfs.plr`

Penalized logistic regression classifier.

The comprehensive API documentation for this module, including all technical details, is available in the Epydoc-generated API reference for mvpa.clfs.plr (for developers).

PLR

class **PLR** (*lm=1, criterion=1, reduced=False, maxiter=20, **kwargs*)
    Bases: `mvpa.clfs.base.Classifier`

    Penalized logistic regression *Classifier*.

    Initialize a penalized logistic regression analysis

- *lm* (int) – the penalty term lambda.
- *criterion* (int) – the criterion applied to judge convergence.
- *reduced* (Bool) – if not False, the rank of the data is reduced before performing the calculations. In that case, reduce is taken as the fraction of the first singular value, at which a dimension is not considered significant anymore. A reasonable criterion is reduced=0.01
- *maxiter* (int) – maximum number of iterations. If no convergence occurs after this number of iterations, an exception is raised.

See Also:

Derived classes might provide additional methods via their base classes. Please refer to the list of base classes (if it exists) at the begining of the `PLR` documentation.

Full API documentation of PLR in module mvpa.clfs.plr.

16.4.16 `mvpa.clfs.ridge`

The comprehensive API documentation for this module, including all technical details, is available in the Epydoc-generated API reference for mvpa.clfs.ridge (for developers).

RidgeReg

See Also:

Derived classes might provide additional methods via their base classes. Please refer to the list of base classes (if it exists) at the begining of the `RidgeReg` documentation.

Full API documentation of RidgeReg in module mvpa.clfs.ridge.

16.4.17 `mvpa.clfs.sg`

The comprehensive API documentation for this module, including all technical details, is available in the Epydoc-generated API reference for mvpa.clfs.sg (for developers).

16.4.18 `mvpa.clfs.sg.sens`

The comprehensive API documentation for this module, including all technical details, is available in the Epydoc-generated API reference for mvpa.clfs.sg.sens (for developers).

`LinearSVMWeights`

See Also:

Derived classes might provide additional methods via their base classes. Please refer to the list of base classes (if it exists) at the begining of the `LinearSVMWeights` documentation.

Full API documentation of LinearSVMWeights in module mvpa.clfs.sg.sens.

16.4.19 `mvpa.clfs.sg.svm`

The comprehensive API documentation for this module, including all technical details, is available in the Epydoc-generated API reference for mvpa.clfs.sg.svm (for developers).

`SVM`

See Also:

Derived classes might provide additional methods via their base classes. Please refer to the list of base classes (if it exists) at the begining of the `SVM` documentation.

Full API documentation of SVM in module mvpa.clfs.sg.svm.

16.4.20 `mvpa.clfs.smlr`

Sparse Multinomial Logistic Regression classifier.

The comprehensive API documentation for this module, including all technical details, is available in the Epydoc-generated API reference for mvpa.clfs.smlr (for developers).

Classes

SMLR

class **SMLR** (***kwargs*)
>   Bases: `mvpa.clfs.base.Classifier`

>   Sparse Multinomial Logistic Regression *Classifier*.

>   This is an implementation of the SMLR algorithm published in Krishnapuram et al. (2005, IEEE Transactions on Pattern Analysis and Machine Intelligence). Be sure to cite that article if you use this for your work.

>   Initialize an SMLR classifier.

>   **biases**

>   **getSensitivityAnalyzer** (***kwargs*)
>>   Returns a sensitivity analyzer for SMLR.

>   **weights**

See Also:

Derived classes might provide additional methods via their base classes. Please refer to the list of base classes (if it exists) at the begining of the `SMLR` documentation.

Full API documentation of SMLR in module mvpa.clfs.smlr.

SMLRWeights

class SMLRWeights (*clf, force_training=True, **kwargs*)
Bases: `mvpa.measures.base.Sensitivity`

SensitivityAnalyzer that reports the weights SMLR trained on a given *Dataset*.

By default SMLR provides multiple weights per feature (one per label in training dataset). By default, all weights are combined into a single sensitivity value. Please, see the *FeaturewiseDatasetMeasure* constructor arguments how to custmize this behavior.

Initialize the analyzer with the classifier it shall use.

> •*clf* (`Classifier`) – classifier to use.
> •*force_training* (Bool) – if classifier was already trained – do not retrain

See Also:

Derived classes might provide additional methods via their base classes. Please refer to the list of base classes (if it exists) at the begining of the `SMLRWeights` documentation.

Full API documentation of SMLRWeights in module mvpa.clfs.smlr.

16.4.21 `mvpa.clfs.stats`

Estimator for classifier error distributions.

The comprehensive API documentation for this module, including all technical details, is available in the Epydoc-generated API reference for mvpa.clfs.stats (for developers).

Classes

AdaptiveNormal

class AdaptiveNormal (*dist, **kwargs*)
Bases: `mvpa.clfs.stats.AdaptiveNullDist`

Adaptive rdist: params are (0, sqrt(1/nfeatures))

> •*dist* (distribution object) – This can be any object the has a *cdf()* method to report the cumulative distribition function values.

See Also:

Derived classes might provide additional methods via their base classes. Please refer to the list of base classes (if it exists) at the begining of the `AdaptiveNormal` documentation.

Full API documentation of AdaptiveNormal in module mvpa.clfs.stats.

AdaptiveNullDist

class AdaptiveNullDist (*dist, **kwargs*)
Bases: `mvpa.clfs.stats.FixedNullDist`

Adaptive distribution which adjusts parameters according to the data

WiP: internal implementation might change

> •*dist* (distribution object) – This can be any object the has a *cdf()* method to report the cumulative distribition function values.

fit (*measure, wdata, vdata=None*)
Cares about dimensionality of the feature space in measure

See Also:

Derived classes might provide additional methods via their base classes. Please refer to the list of base classes (if it exists) at the begining of the `AdaptiveNullDist` documentation.

Full API documentation of AdaptiveNullDist in module mvpa.clfs.stats.

AdaptiveRDist

class **AdaptiveRDist**(*dist, **kwargs*)

> Bases: `mvpa.clfs.stats.AdaptiveNullDist`
>
> Adaptive rdist: params are (nfeatures-1, 0, 1)
>
> > • *dist* (distribution object) – This can be any object the has a *cdf()* method to report the cumulative distribution function values.
>
> **cdf**(*x*)

See Also:

Derived classes might provide additional methods via their base classes. Please refer to the list of base classes (if it exists) at the begining of the `AdaptiveRDist` documentation.

Full API documentation of AdaptiveRDist in module mvpa.clfs.stats.

FixedNullDist

class **FixedNullDist**(*dist, **kwargs*)

> Bases: `mvpa.clfs.stats.NullDist`
>
> Proxy/Adaptor class for SciPy distributions.
>
> All distributions from SciPy's 'stats' module can be used with this class.

```
>>> import numpy as N
>>> from scipy import stats
>>> from mvpa.clfs.stats import FixedNullDist
>>>
>>> dist = FixedNullDist(stats.norm(loc=2, scale=4))
>>> dist.p(2)
0.5
>>>
>>> dist.cdf(N.arange(5))
array([ 0.30853754,  0.40129367,  0.5       ,  0.59870633,  0.69146246])
>>>
>>> dist = FixedNullDist(stats.norm(loc=2, scale=4), tail='right')
>>> dist.p(N.arange(5))
array([ 0.69146246,  0.59870633,  0.5       ,  0.40129367,  0.30853754])
```

> > • *dist* (distribution object) – This can be any object the has a *cdf()* method to report the cumulative distribition function values.
>
> **cdf**(*x*)
> > Return value of the cumulative distribution function at *x*.
>
> **fit**(*measure, wdata, vdata=None*)
> > Does nothing since the distribution is already fixed.

See Also:

Derived classes might provide additional methods via their base classes. Please refer to the list of base classes (if it exists) at the begining of the `FixedNullDist` documentation.

Full API documentation of FixedNullDist in module mvpa.clfs.stats.

MCNullDist

class MCNullDist(*dist_class=<class 'mvpa.clfs.stats.Nonparametric'>, permutations=100, **kwargs*)

Bases: `mvpa.clfs.stats.NullDist`

Null-hypothesis distribution is estimated from randomly permuted data labels.

The distribution is estimated by calling fit() with an appropriate *DatasetMeasure* or *TransferError* instance and a training and a validation dataset (in case of a *TransferError*). For a customizable amount of cycles the training data labels are permuted and the corresponding measure computed. In case of a *TransferError* this is the error when predicting the *correct* labels of the validation dataset.

The distribution can be queried using the *cdf()* method, which can be configured to report probabilities/frequencies from *left* or *right* tail, i.e. fraction of the distribution that is lower or larger than some critical value.

This class also supports *FeaturewiseDatasetMeasure*. In that case *cdf()* returns an array of featurewise probabilities/frequencies.

Initialize Monte-Carlo Permutation Null-hypothesis testing

- *dist_class* (class) – This can be any class which provides parameters estimate using *fit()* method to initialize the instance, and provides *cdf(x)* method for estimating value of x in CDF. All distributions from SciPy's 'stats' module can be used.
- *permutations* (int) – This many permutations of label will be performed to determine the distribution under the null hypothesis.

cdf(*x*)

Return value of the cumulative distribution function at *x*.

clean()

Clean stored distributions

Storing all of the distributions might be too expensive (e.g. in case of Nonparametric), and the scope of the object might be too broad to wait for it to be destroyed. Clean would bind dist_samples to empty list to let gc revoke the memory.

fit(*measure, wdata, vdata=None*)

Fit the distribution by performing multiple cycles which repeatedly permuted labels in the training dataset.

- *measure* ((*Featurewise*)'DatasetMeasure' | *TransferError*) – TransferError instance used to compute all errors.
- *wdata* (*Dataset* which gets permuted and used to compute the) – measure/transfer error multiple times.
- *vdata* (*Dataset* used for validation.) – If provided measure is assumed to be a *TransferError* and working and validation dataset are passed onto it.

See Also:

Derived classes might provide additional methods via their base classes. Please refer to the list of base classes (if it exists) at the begining of the `MCNullDist` documentation.

Full API documentation of MCNullDist in module mvpa.clfs.stats.

Nonparametric

class Nonparametric(*dist_samples*)

Bases: `object`

Non-parametric 1d distribution – derives cdf based on stored values.

Introduced to complement parametric distributions present in scipy.stats.

cdf(*x*)

Returns the cdf value at *x*.

fit

See Also:

Derived classes might provide additional methods via their base classes. Please refer to the list of base classes (if it exists) at the begining of the `Nonparametric` documentation.

Full API documentation of Nonparametric in module mvpa.clfs.stats.

NullDist

class **NullDist** (*tail='both', **kwargs*)

    Bases: `mvpa.misc.state.ClassWithCollections`

    Base class for null-hypothesis testing.

    Cheap initialization.

> •*tail* (str ('left', 'right', 'any', 'both')) – Which tail of the distribution to report. For 'any' and 'both' it chooses the tail it belongs to based on the comparison to p=0.5. In the case of 'any' significance is taken like in a one-tailed test.

    cdf (*x*)

        Implementations return the value of the cumulative distribution function (left or right tail dpending on the setting).

    fit (*measure, wdata, vdata=None*)

        Implement to fit the distribution to the data.

    p (*x, **kwargs*)

        Returns the p-value for values of *x*. Returned values are determined left, right, or from any tail depending on the constructor setting.

        In case a *FeaturewiseDatasetMeasure* was used to estimate the distribution the method returns an array. In that case *x* can be a scalar value or an array of a matching shape.

    tail

See Also:

Derived classes might provide additional methods via their base classes. Please refer to the list of base classes (if it exists) at the begining of the `NullDist` documentation.

Full API documentation of NullDist in module mvpa.clfs.stats.

Functions

autoNullDist (*dist*)

    Cheater for human beings – wraps *dist* if needed with some NullDist

    tail and other arguments are assumed to be default as in NullDist/MCNullDist

See Also:

Full API documentation of autoNullDist() in module mvpa.clfs.stats.

nanmean (*x, axis=0*)

    Compute the mean over the given axis ignoring nans.

> •*x* (ndarray) – input array
> •*axis* (int) – axis along which the mean is computed.

    m   [float] the mean.

See Also:

Full API documentation of nanmean() in module mvpa.clfs.stats.

16.4.22 `mvpa.clfs.svm`

Wrap the libsvm package into a very simple class interface.

The comprehensive API documentation for this module, including all technical details, is available in the Epydoc-generated API reference for mvpa.clfs.svm (for developers).

16.4.23 `mvpa.clfs.transerror`

Utility class to compute the transfer error of classifiers.

The comprehensive API documentation for this module, including all technical details, is available in the Epydoc-generated API reference for mvpa.clfs.transerror (for developers).

Classes

ClassifierError

class **ClassifierError** (*clf, labels=None, train=True, **kwargs*)
 Bases: `mvpa.misc.state.ClassWithCollections`

 Compute (or return) some error of a (trained) classifier on a dataset.

 Initialization.

> •*clf* (Classifier) – Either trained or untrained classifier
>
> •*labels* (list) – if provided, should be a set of labels to add on top of the ones present in testdata
>
> •*train* (bool) – unless train=False, classifier gets trained if trainingdata provided to __call__

 clf

 labels

See Also:

Derived classes might provide additional methods via their base classes. Please refer to the list of base classes (if it exists) at the begining of the `ClassifierError` documentation.

Full API documentation of ClassifierError in module mvpa.clfs.transerror.

ConfusionBasedError

class **ConfusionBasedError** (*clf, labels=None, confusion_state='training_confusion', **kwargs*)
 Bases: `mvpa.clfs.transerror.ClassifierError`

 For a given classifier report an error based on internally computed error measure (given by some *Confusion-Matrix* stored in some state variable of *Classifier*).

 This way we can perform feature selection taking as the error criterion either learning error, or transfer to splits error in the case of SplitClassifier

 Initialization.

> •*clf* (Classifier) – Either trained or untrained classifier
>
> •*confusion_state* – Id of the state variable which stores *ConfusionMatrix*
>
> •*labels* (list) – if provided, should be a set of labels to add on top of the ones present in testdata

See Also:

Derived classes might provide additional methods via their base classes. Please refer to the list of base classes (if it exists) at the begining of the `ConfusionBasedError` documentation.

Full API documentation of ConfusionBasedError in module mvpa.clfs.transerror.

ConfusionMatrix

class ConfusionMatrix (*labels=None, labels_map=None, **kwargs*)

> Bases: `mvpa.clfs.transerror.SummaryStatistics`

> Class to contain information and display confusion matrix.

> Implementation of the *SummaryStatistics* in the case of classification problem. Actual computation of confusion matrix is delayed until all data is acquired (to figure out complete set of labels). If testing data doesn't have a complete set of labels, but you like to include all labels, provide them as a parameter to the constructor.

> Confusion matrix provides a set of performance statistics (use asstring(description=True) for the description of abbreviations), as well ROC curve (http://en.wikipedia.org/wiki/ROC_curve) plotting and analysis (AUC) in the limited set of problems: binary, multiclass 1-vs-all.

> Initialize ConfusionMatrix with optional list of *labels*

>> •*labels* (list) – Optional set of labels to include in the matrix
>> •*labels_map* (None or dict) – Dictionary from original dataset to show mapping into numerical labels
>> •*targets* – Optional set of targets
>> •*predictions* – Optional set of predictions

> **asstring** (*short=False, header=True, summary=True, description=False*)

>> 'Pretty print' the matrix

>>> •*short* (bool) – if True, ignores the rest of the parameters and provides consise 1 line summary
>>> •*header* (bool) – print header of the table
>>> •*summary* (bool) – print summary (accuracy)
>>> •*description* (bool) – print verbose description of presented statistics

> **error**

> **getLabels_map** ()

> **labels**

> **labels_map**

> **matrices**

>> Return a list of separate confusion matrix per each stored set

> **matrix**

> **percentCorrect**

> **plot** (*labels=None, numbers=False, origin='upper', numbers_alpha=None, xlabels_vertical=True, numbers_kwargs={}, **kwargs*)

>> Provide presentation of confusion matrix in image

>>> •*labels* (list of int or basestring) – Optionally provided labels guarantee the order of presentation. Also value of None places empty column/row, thus provides visual grouping of labels (Thanks Ingo)
>>> •*numbers* (bool) – Place values inside of confusion matrix elements
>>> •*numbers_alpha* (None or float) – Controls textual output of numbers. If None – all numbers are plotted in the same intensity. If some float – it controls alpha level – higher value would give higher contrast. (good value is 2)
>>> •*origin* (basestring) – Which left corner diagonal should start
>>> •*xlabels_vertical* (bool) – Either to plot xlabels vertical (benefitial if number of labels is large)
>>> •*numbers_kwargs* (dict) – Additional keyword parameters to be added to numbers (if numbers is True)

> •***kwargs** – Additional arguments given to imshow (eg me cmap)
>> Return type
>>> (fig, im, cb) – figure, imshow, colorbar

> **setLabels_map**(*val*)

See Also:

Derived classes might provide additional methods via their base classes. Please refer to the list of base classes (if it exists) at the begining of the `ConfusionMatrix` documentation.

Full API documentation of ConfusionMatrix in module mvpa.clfs.transerror.

ROCCurve

class ROCCurve(*labels, sets=None*)

> Bases: `object`

> Generic class for ROC curve computation and plotting

>> •*labels* (list) – labels which were used (in order of values if multiclass, or 1 per class for binary problems (e.g. in SMLR))
>> •*sets* (list of tuples) – list of sets for the analysis

> **ROCs**

> **aucs**
>> Compute and return set of AUC values 1 per label

> **plot**(*label_index=0*)

>> TODO: make it friendly to labels given by values?
>>> should we also treat labels_map?

See Also:

Derived classes might provide additional methods via their base classes. Please refer to the list of base classes (if it exists) at the begining of the `ROCCurve` documentation.

Full API documentation of ROCCurve in module mvpa.clfs.transerror.

RegressionStatistics

class RegressionStatistics(***kwargs*)

> Bases: `mvpa.clfs.transerror.SummaryStatistics`

> Class to contain information and display on regression results.

> Initialize RegressionStatistics

>> •*targets* – Optional set of targets
>> •*predictions* – Optional set of predictions

> **asstring**(*short=False, header=True, summary=True, description=False*)
>> 'Pretty print' the statistics

> **error**

See Also:

Derived classes might provide additional methods via their base classes. Please refer to the list of base classes (if it exists) at the begining of the `RegressionStatistics` documentation.

Full API documentation of RegressionStatistics in module mvpa.clfs.transerror.

SummaryStatistics

class **SummaryStatistics**(*targets=None, predictions=None, values=None, sets=None*)

Bases: `object`

Basic class to collect targets/predictions and report summary statistics

It takes care about collecting the sets, which are just tuples (targets, predictions, values). While 'computing' the matrix, all sets are considered together. Children of the class are responsible for computation and display.

Initialize SummaryStatistics

targets or predictions cannot be provided alone (ie targets without predictions)

> •*targets* – Optional set of targets
> •*predictions* – Optional set of predictions
> •*values* – Optional set of values (which served for prediction)
> •*sets* – Optional list of sets

add(*targets, predictions, values=None*)
Add new results to the set of known results

asstring(*short=False, header=True, summary=True, description=False*)
'Pretty print' the matrix

> •*short* (bool) – if True, ignores the rest of the parameters and provides consise 1 line summary
> •*header* (bool) – print header of the table
> •*summary* (bool) – print summary (accuracy)
> •*description* (bool) – print verbose description of presented statistics

compute()
Actually compute the confusion matrix based on all the sets

error

reset()
Cleans summary – all data/sets are wiped out

sets

stats

summaries
Return a list of separate summaries per each stored set

See Also:

Derived classes might provide additional methods via their base classes. Please refer to the list of base classes (if it exists) at the begining of the SummaryStatistics documentation.

Full API documentation of SummaryStatistics in module mvpa.clfs.transerror.

TransferError

class **TransferError**(*clf, errorfx=MeanMismatchErrorFx(), labels=None, null_dist=None, **kwargs*)
Bases: `mvpa.clfs.transerror.ClassifierError`

Compute the transfer error of a (trained) classifier on a dataset.

The actual error value is computed using a customizable error function. Optionally the classifier can be trained by passing an additional training dataset to the __call__() method.

Initialization.

> •*clf* (Classifier) – Either trained or untrained classifier
> •*errorfx* – Functor that computes a scalar error value from the vectors of desired and predicted values (e.g. subclass of *ErrorFunction*)

•*labels* (list) – if provided, should be a set of labels to add on top of the ones present in testdata

•*null_dist* (instance of distribution estimator) –

errorfx

null_dist

See Also:

Derived classes might provide additional methods via their base classes. Please refer to the list of base classes (if it exists) at the begining of the `TransferError` documentation.

Full API documentation of TransferError in module mvpa.clfs.transerror.

16.4.24 `mvpa.clfs.warehouse`

Collection of classifiers to ease the exploration.

The comprehensive API documentation for this module, including all technical details, is available in the Epydoc-generated API reference for mvpa.clfs.warehouse (for developers).

Warehouse

class **Warehouse** (*known_tags=None, matches={}*)

    Bases: `object`

    Class to keep known instantiated classifiers

    Should provide easy ways to select classifiers of needed kind: clfs['linear', 'svm'] should return all linear SVMs clfs['linear', 'multiclass'] should return all linear classifiers capable of doing multiclass classification

    Initialize warehouse

        •*known_tags* (list of basestring) – List of known tags

        •*matches* (dict) – Optional dictionary of additional matches. E.g. since any regression can be used as a binary classifier, matches={'binary':['regression']}, would allow to provide regressions also if 'binary' was requested

    internals

    items

    listing()

See Also:

Derived classes might provide additional methods via their base classes. Please refer to the list of base classes (if it exists) at the begining of the `Warehouse` documentation.

Full API documentation of Warehouse in module mvpa.clfs.warehouse.

16.5 Measures: Searchlights and Sensitivties

16.5.1 `mvpa.measures`

The comprehensive API documentation for this module, including all technical details, is available in the Epydoc-generated API reference for mvpa.measures (for developers).

16.5.2 `mvpa.measures.anova`

FeaturewiseDatasetMeasure performing a univariate ANOVA.

The comprehensive API documentation for this module, including all technical details, is available in the Epydoc-generated API reference for mvpa.measures.anova (for developers).

OneWayAnova

class OneWayAnova (***kwargs*)

> Bases: mvpa.measures.base.FeaturewiseDatasetMeasure
>
> *FeaturewiseDatasetMeasure* that performs a univariate ANOVA.
>
> F-scores are computed for each feature as the standard fraction of between and within group variances. Groups are defined by samples with unique labels.
>
> No statistical testing is performed, but raw F-scores are returned as a sensitivity map. As usual F-scores have a range of [0,inf] with greater values indicating higher sensitivity.
>
> Nothing special to do here.

See Also:

Derived classes might provide additional methods via their base classes. Please refer to the list of base classes (if it exists) at the begining of the OneWayAnova documentation.

Full API documentation of OneWayAnova in module mvpa.measures.anova.

16.5.3 `mvpa.measures.base`

Base class for data measures: algorithms that quantify properties of datasets.

Besides the *DatasetMeasure* base class this module also provides the (abstract) *FeaturewiseDatasetMeasure* class. The difference between a general measure and the output of the *FeaturewiseDatasetMeasure* is that the latter returns a 1d map (one value per feature in the dataset). In contrast there are no restrictions on the returned value of *DatasetMeasure* except for that it has to be in some iterable container.

The comprehensive API documentation for this module, including all technical details, is available in the Epydoc-generated API reference for mvpa.measures.base (for developers).

Classes

BoostedClassifierSensitivityAnalyzer

class BoostedClassifierSensitivityAnalyzer (**args_, **kwargs_*)

> Bases: mvpa.measures.base.Sensitivity
>
> Set sensitivity analyzers to be merged into a single output
>
> **combined_analyzer**

See Also:

Derived classes might provide additional methods via their base classes. Please refer to the list of base classes (if it exists) at the begining of the BoostedClassifierSensitivityAnalyzer documentation.

Full API documentation of BoostedClassifierSensitivityAnalyzer in module mvpa.measures.base.

CombinedFeaturewiseDatasetMeasure

class CombinedFeaturewiseDatasetMeasure (*analyzers=None, combiner=None, **kwargs*)

> Bases: mvpa.measures.base.FeaturewiseDatasetMeasure
>
> Set sensitivity analyzers to be merged into a single output
>
> Initialize CombinedFeaturewiseDatasetMeasure

> • *analyzers* (list or None) – List of analyzers to be used. There is no logic to populate such a list in __call__, so it must be either provided to the constructor or assigned to .analyzers prior calling

> **analyzers**
>> Used analyzers

See Also:

Derived classes might provide additional methods via their base classes. Please refer to the list of base classes (if it exists) at the begining of the `CombinedFeaturewiseDatasetMeasure` documentation.

Full API documentation of CombinedFeaturewiseDatasetMeasure in module mvpa.measures.base.

DatasetMeasure

class **DatasetMeasure** (*transformer=None, null_dist=None, **kwargs*)
> Bases: `mvpa.misc.state.ClassWithCollections`

> A measure computed from a *Dataset*

> All dataset measures support arbitrary transformation of the measure after it has been computed. Transformation are done by processing the measure with a functor that is specified via the *transformer* keyword argument of the constructor. Upon request, the raw measure (before transformations are applied) is stored in the *raw_result* state variable.

> Additionally all dataset measures support the estimation of the probabilit(y,ies) of a measure under some distribution. Typically this will be the NULL distribution (no signal), that can be estimated with permutation tests. If a distribution estimator instance is passed to the *null_dist* keyword argument of the constructor the respective probabilities are automatically computed and stored in the *null_prob* state variable.

>> Developer note
>>> All subclasses shall get all necessary parameters via their constructor, so it is possible to get the same type of measure for multiple datasets by passing them to the __call__() method successively.

> Does nothing special.

>> • *transformer* (Functor) – This functor is called in *__call__()* to perform a final processing step on the to be returned dataset measure. If None, nothing is called
>> • *null_dist* (instance of distribution estimator) –

> **null_dist**

See Also:

Derived classes might provide additional methods via their base classes. Please refer to the list of base classes (if it exists) at the begining of the `DatasetMeasure` documentation.

Full API documentation of DatasetMeasure in module mvpa.measures.base.

FeaturewiseDatasetMeasure

class **FeaturewiseDatasetMeasure** (*combiner=<function   SecondAxisSumOfAbs   at   0x8d1be9c>, **kwargs*)
> Bases: `mvpa.measures.base.DatasetMeasure`

> A per-feature-measure computed from a *Dataset* (base class).

> Should behave like a DatasetMeasure.

> Initialize

>> • *combiner* (Functor) – The combiner is only applied if the computed featurewise dataset measure is more than one-dimensional. This is different from a *transformer*, which

is always applied. By default, the sum of absolute values along the second axis is computed.

See Also:

Derived classes might provide additional methods via their base classes. Please refer to the list of base classes (if it exists) at the begining of the FeaturewiseDatasetMeasure documentation.

Full API documentation of FeaturewiseDatasetMeasure in module mvpa.measures.base.

MappedClassifierSensitivityAnalyzer

class **MappedClassifierSensitivityAnalyzer**(*args_, **kwargs_*)
   Bases: mvpa.measures.base.ProxyClassifierSensitivityAnalyzer

   Set sensitivity analyzer output be reverse mapped using mapper of the slave classifier

See Also:

Derived classes might provide additional methods via their base classes. Please refer to the list of base classes (if it exists) at the begining of the MappedClassifierSensitivityAnalyzer documentation.

Full API documentation of MappedClassifierSensitivityAnalyzer in module mvpa.measures.base.

ProxyClassifierSensitivityAnalyzer

class **ProxyClassifierSensitivityAnalyzer**(*args_, **kwargs_*)
   Bases: mvpa.measures.base.Sensitivity

   Set sensitivity analyzer output just to pass through

   analyzer

See Also:

Derived classes might provide additional methods via their base classes. Please refer to the list of base classes (if it exists) at the begining of the ProxyClassifierSensitivityAnalyzer documentation.

Full API documentation of ProxyClassifierSensitivityAnalyzer in module mvpa.measures.base.

Sensitivity

class **Sensitivity**(*clf, force_training=True, **kwargs*)
   Bases: mvpa.measures.base.FeaturewiseDatasetMeasure

   Initialize the analyzer with the classifier it shall use.

      •*clf* (Classifier) – classifier to use.
      •*force_training* (Bool) – if classifier was already trained – do not retrain

   clf

   feature_ids
      Return feature_ids used by the underlying classifier

See Also:

Derived classes might provide additional methods via their base classes. Please refer to the list of base classes (if it exists) at the begining of the Sensitivity documentation.

Full API documentation of Sensitivity in module mvpa.measures.base.

StaticDatasetMeasure

class **StaticDatasetMeasure** (*measure=None, bias=None, *args, **kwargs*)

> Bases: `mvpa.measures.base.DatasetMeasure`
>
> A static (assigned) sensitivity measure.
>
> Since implementation is generic it might be per feature or per whole dataset
>
> Initialize.
>
> > •*measure* – actual sensitivity to be returned
> > •*bias* – optionally available bias
>
> **bias**

See Also:

Derived classes might provide additional methods via their base classes. Please refer to the list of base classes (if it exists) at the begining of the `StaticDatasetMeasure` documentation.

Full API documentation of StaticDatasetMeasure in module mvpa.measures.base.

16.5.4 `mvpa.measures.corrcoef`

The comprehensive API documentation for this module, including all technical details, is available in the Epydoc-generated API reference for mvpa.measures.corrcoef (for developers).

CorrCoef

See Also:

Derived classes might provide additional methods via their base classes. Please refer to the list of base classes (if it exists) at the begining of the `CorrCoef` documentation.

Full API documentation of CorrCoef in module mvpa.measures.corrcoef.

16.5.5 `mvpa.measures.irelief`

FeaturewiseDatasetMeasure performing multivariate Iterative RELIEF (I-RELIEF) algorithm. See : Y. Sun, Iterative RELIEF for Feature Weighting: Algorithms, Theories, and Applications, IEEE Trans. on Pattern Analysis and Machine Intelligence (TPAMI), vol. 29, no. 6, pp. 1035-1051, June 2007.

The comprehensive API documentation for this module, including all technical details, is available in the Epydoc-generated API reference for mvpa.measures.irelief (for developers).

Classes

IterativeRelief

class **IterativeRelief** (*threshold=0.01, kernel_width=1.0, w_guess=None, **kwargs*)

> Bases: `mvpa.measures.base.FeaturewiseDatasetMeasure`
>
> *FeaturewiseDatasetMeasure* that performs multivariate I-RELIEF algorithm. Batch version.
>
> Batch I-RELIEF-2 feature weighting algorithm. Works for binary or multiclass class-labels. Batch version with complexity $O(T*N^2*I)$, where T is the number of iterations, N the number of instances, I the number of features.
>
> See: Y. Sun, Iterative RELIEF for Feature Weighting: Algorithms, Theories, and Applications, IEEE Trans. on Pattern Analysis and Machine Intelligence (TPAMI), vol. 29, no. 6, pp. 1035-1051, June 2007. http://plaza.ufl.edu/sunyijun/Paper/PAMI_1.pdf

Note that current implementation allows to use only exponential-like kernels. Support for linear kernel will be added later.

Constructor of the IRELIEF class.

compute_M_H (*label*)

Compute hit/miss dictionaries.

For each instance compute the set of indices having the same class label and different class label.

Note that this computation is independent of the number of features.

XXX should it be some generic function since it doesn't use self

k (*distances*)

Exponential kernel.

See Also:

Derived classes might provide additional methods via their base classes. Please refer to the list of base classes (if it exists) at the begining of the `IterativeRelief` documentation.

Full API documentation of IterativeRelief in module mvpa.measures.irelief.

IterativeReliefOnline

class **IterativeReliefOnline** (*a=10.0, permute=True, max_iter=3, **kwargs*)

Bases: `mvpa.measures.irelief.IterativeRelief`

FeaturewiseDatasetMeasure that performs multivariate I-RELIEF algorithm. Online version.

This algorithm is exactly the one in the referenced paper (I-RELIEF-2 online), using weighted 1-norm and Exponential Kernel.

Constructor of the IRELIEF class.

See Also:

Derived classes might provide additional methods via their base classes. Please refer to the list of base classes (if it exists) at the begining of the `IterativeReliefOnline` documentation.

Full API documentation of IterativeReliefOnline in module mvpa.measures.irelief.

IterativeReliefOnline_Devel

class **IterativeReliefOnline_Devel** (*a=5.0, permute=True, max_iter=3, **kwargs*)

Bases: `mvpa.measures.irelief.IterativeRelief_Devel`

FeaturewiseDatasetMeasure that performs multivariate I-RELIEF algorithm. Online version.

UNDER DEVELOPMENT

Online version with complexity O(T*N*I), where N is the number of instances and I the number of features.

See: Y. Sun, Iterative RELIEF for Feature Weighting: Algorithms, Theories, and Applications, IEEE Trans. on Pattern Analysis and Machine Intelligence (TPAMI), vol. 29, no. 6, pp. 1035-1051, June 2007. http://plaza.ufl.edu/sunyijun/Paper/PAMI_1.pdf

Note that this implementation is not fully online, since hit and miss dictionaries (H,M) are computed once at the beginning using full access to all labels. This can be easily corrected to a full online implementation. But this is not mandatory now since the major goal of this current online implementation is reduction of computational complexity.

Constructor of the IRELIEF class.

See Also:

Derived classes might provide additional methods via their base classes. Please refer to the list of base classes (if it exists) at the begining of the `IterativeReliefOnline_Devel` documentation.

Full API documentation of IterativeReliefOnline_Devel in module mvpa.measures.irelief.

IterativeRelief_Devel

class **IterativeRelief_Devel**(*threshold=0.01,   kernel=None,   kernel_width=1.0,   w_guess=None,* ***kwargs*)

Bases: `mvpa.measures.base.FeaturewiseDatasetMeasure`

FeaturewiseDatasetMeasure that performs multivariate I-RELIEF algorithm. Batch version allowing various kernels.

UNDER DEVELOPEMNT.

Batch I-RELIEF-2 feature weighting algorithm. Works for binary or multiclass class-labels. Batch version with complexity $O(T*N^2*I)$, where T is the number of iterations, N the number of instances, I the number of features.

See: Y. Sun, Iterative RELIEF for Feature Weighting: Algorithms, Theories, and Applications, IEEE Trans. on Pattern Analysis and Machine Intelligence (TPAMI), vol. 29, no. 6, pp. 1035-1051, June 2007. http://plaza.ufl.edu/sunyijun/Paper/PAMI_1.pdf

Note that current implementation allows to use only exponential-like kernels. Support for linear kernel will be added later.

Constructor of the IRELIEF class.

compute_M_H(*label*)

Compute hit/miss dictionaries.

For each instance compute the set of indices having the same class label and different class label.

Note that this computation is independent of the number of features.

See Also:

Derived classes might provide additional methods via their base classes. Please refer to the list of base classes (if it exists) at the begining of the `IterativeRelief_Devel` documentation.

Full API documentation of IterativeRelief_Devel in module mvpa.measures.irelief.

16.5.6 `mvpa.measures.noiseperturbation`

This is a *FeaturewiseDatasetMeasure* that uses a scalar *DatasetMeasure* and selective noise perturbation to compute a sensitivity map.

The comprehensive API documentation for this module, including all technical details, is available in the Epydoc-generated API reference for mvpa.measures.noiseperturbation (for developers).

NoisePerturbationSensitivity

class **NoisePerturbationSensitivity**(*datameasure,   noise=<built-in   method   normal   of* *mtrand.RandomState object at 0x401ec350>*)

Bases: `mvpa.measures.base.FeaturewiseDatasetMeasure`

This is a *FeaturewiseDatasetMeasure* that uses a scalar *DatasetMeasure* and selective noise perturbation to compute a sensitivity map.

First the scalar *DatasetMeasure* computed using the original dataset. Next the data measure is computed multiple times each with a single feature in the dataset perturbed by noise. The resulting difference in the scalar *DatasetMeasure* is used as the sensitivity for the respective perturbed feature. Large differences are treated as an indicator of a feature having great impact on the scalar *DatasetMeasure*.

The computed sensitivity map might have positive and negative values!

Cheap initialization.

> datameasure: *Datameasure* that is used to quantify the effect of
> > noise perturbation.
>
> noise: Functor to generate noise. The noise generator has to return
> > an 1d array of n values when called the *size=n* keyword argument. This is the default interface of
> > the random number generators in NumPy's *random* module.

See Also:

Derived classes might provide additional methods via their base classes. Please refer to the list of base classes (if it exists) at the begining of the `NoisePerturbationSensitivity` documentation.

Full API documentation of NoisePerturbationSensitivity in module mvpa.measures.noiseperturbation.

16.5.7 `mvpa.measures.pls`

The comprehensive API documentation for this module, including all technical details, is available in the Epydoc-generated API reference for mvpa.measures.pls (for developers).

Classes

PLS

class PLS (*num_permutations=200, num_bootstraps=100, **kwargs*)
    Bases: `mvpa.measures.base.FeaturewiseDatasetMeasure`

See Also:

Derived classes might provide additional methods via their base classes. Please refer to the list of base classes (if it exists) at the begining of the `PLS` documentation.

Full API documentation of PLS in module mvpa.measures.pls.

TaskPLS

class TaskPLS (*num_permutations=200, num_bootstraps=100, **kwargs*)
    Bases: `mvpa.measures.pls.PLS`

See Also:

Derived classes might provide additional methods via their base classes. Please refer to the list of base classes (if it exists) at the begining of the `TaskPLS` documentation.

Full API documentation of TaskPLS in module mvpa.measures.pls.

16.5.8 `mvpa.measures.searchlight`

Implementation of the Searchlight algorithm

The comprehensive API documentation for this module, including all technical details, is available in the Epydoc-generated API reference for mvpa.measures.searchlight (for developers).

Searchlight

class Searchlight (*datameasure, radius=1.0, center_ids=None, **kwargs*)
    Bases: `mvpa.measures.base.DatasetMeasure`

    Runs a scalar *DatasetMeasure* on all possible spheres of a certain size within a dataset.

    The idea for a searchlight algorithm stems from a paper by *Kriegeskorte et al. (2006)*.

    - *datameasure* (callable) – Any object that takes a `Dataset` and returns some measure when called.
    - *radius* (float) – All features within the radius around the center will be part of a sphere.
    - *center_ids* (list(int)) – List of feature ids (not coordinates) the shall serve as sphere centers. By default all features will be used.

> •**kwargs* – In additions this class supports all keyword arguments of its base-class `DatasetMeasure`.

Note: If *Searchlight* is used as *SensitivityAnalyzer* one has to make sure that the specified scalar *Dataset-Measure* returns large (absolute) values for high sensitivities and small (absolute) values for low sensitivities. Especially when using error functions usually low values imply high performance and therefore high sensitivity. This would in turn result in sensitivity maps that have low (absolute) values indicating high sensitives and this conflicts with the intended behavior of a *SensitivityAnalyzer*.

See Also:

Derived classes might provide additional methods via their base classes. Please refer to the list of base classes (if it exists) at the begining of the `Searchlight` documentation.

Full API documentation of Searchlight in module mvpa.measures.searchlight.

16.5.9 `mvpa.measures.splitmeasure`

This is a *FeaturewiseDatasetMeasure* that uses another *FeaturewiseDatasetMeasure* and runs it multiple times on differents splits of a *Dataset*.

The comprehensive API documentation for this module, including all technical details, is available in the Epydoc-generated API reference for mvpa.measures.splitmeasure (for developers).

`SplitFeaturewiseMeasure`

class **SplitFeaturewiseMeasure**(*sensana, splitter=<class 'mvpa.datasets.splitter.NoneSplitter'>, com-biner=<function FirstAxisMean at 0x8d1be2c>, **kwargs*)
  Bases: `mvpa.measures.base.FeaturewiseDatasetMeasure`

This is a *FeaturewiseDatasetMeasure* that uses another *FeaturewiseDatasetMeasure* and runs it multiple times on differents splits of a *Dataset*.

When called with a *Dataset* it returns the mean sensitivity maps of all data splits.

Additonally this class supports the *State* interface. Several postprocessing functions can be specifed to the constructor. The results of the functions specified in the *postproc* dictionary will be available via their respective keywords.

Cheap initialization.

> •*sensana* (FeaturewiseDatasetMeasure) – that shall be run on the *Dataset* splits.
> •*splitter* (Splitter) – used to split the *Dataset*. By convention the first dataset in the tuple returned by the splitter on each iteration is used to compute the sensitivity map.
> •*combiner* – This functor will be called on an array of sensitivity maps and the result will be returned by __call__(). The result of a combiner must be an 1d ndarray.

See Also:

Derived classes might provide additional methods via their base classes. Please refer to the list of base classes (if it exists) at the begining of the `SplitFeaturewiseMeasure` documentation.

Full API documentation of SplitFeaturewiseMeasure in module mvpa.measures.splitmeasure.

16.6 Feature Selection

16.6.1 `mvpa.featsel`

The comprehensive API documentation for this module, including all technical details, is available in the Epydoc-generated API reference for mvpa.featsel (for developers).

16.6.2 `mvpa.featsel.base`

Feature selection base class and related stuff base classes and helpers.

The comprehensive API documentation for this module, including all technical details, is available in the Epydoc-generated API reference for mvpa.featsel.base (for developers).

Classes

FeatureSelection

class **FeatureSelection** (***kwargs*)

> Bases: `mvpa.misc.state.ClassWithCollections`
>
> Base class for any feature selection
>
> Base class for Functors which implement feature selection on the datasets.

See Also:

Derived classes might provide additional methods via their base classes. Please refer to the list of base classes (if it exists) at the begining of the `FeatureSelection` documentation.

Full API documentation of FeatureSelection in module mvpa.featsel.base.

FeatureSelectionPipeline

class **FeatureSelectionPipeline** (*feature_selections, **kwargs*)

> Bases: `mvpa.featsel.base.FeatureSelection`
>
> Feature elimination through the list of FeatureSelection's.
>
> Given as list of FeatureSelections it applies them in turn.
>
> Initialize feature selection pipeline
>
> > •*feature_selections* (lisf of FeatureSelection) – selections which to use. Order matters

> **feature_selections**
> > List of *FeatureSelections*

See Also:

Derived classes might provide additional methods via their base classes. Please refer to the list of base classes (if it exists) at the begining of the `FeatureSelectionPipeline` documentation.

Full API documentation of FeatureSelectionPipeline in module mvpa.featsel.base.

SensitivityBasedFeatureSelection

class **SensitivityBasedFeatureSelection** (*sensitivity_analyzer,* *feature_selector=FractionTailSelector() fraction=0.050000,* ***kwargs*)

> Bases: `mvpa.featsel.base.FeatureSelection`
>
> Feature elimination.
>
> A *FeaturewiseDatasetMeasure* is used to compute sensitivity maps given a certain dataset. These sensitivity maps are in turn used to discard unimportant features.
>
> Initialize feature selection
>
> > •*sensitivity_analyzer* (FeaturewiseDatasetMeasure) – sensitivity analyzer to come up with sensitivity

•*feature_selector* (Functor) – Given a sensitivity map it has to return the ids of those features that should be kept.

sensitivity_analyzer
Measure which was used to do selection

See Also:

Derived classes might provide additional methods via their base classes. Please refer to the list of base classes (if it exists) at the begining of the `SensitivityBasedFeatureSelection` documentation.

Full API documentation of SensitivityBasedFeatureSelection in module mvpa.featsel.base.

16.6.3 `mvpa.featsel.helpers`

The comprehensive API documentation for this module, including all technical details, is available in the Epydoc-generated API reference for mvpa.featsel.helpers (for developers).

Classes

BestDetector

class **BestDetector** (*func=<built-in function min>, lastminimum=False*)
Bases: `object`

Determine whether the last value in a sequence is the best one given some criterion.

Initialize with number of steps

•*fun* (functor) – Functor to select the best results. Defaults to min
•*lastminimum* (bool) – Toggle whether the latest or the earliest minimum is used as optimal value to determine the stopping criterion.

bestindex

See Also:

Derived classes might provide additional methods via their base classes. Please refer to the list of base classes (if it exists) at the begining of the `BestDetector` documentation.

Full API documentation of BestDetector in module mvpa.featsel.helpers.

ElementSelector

class **ElementSelector** (*mode='discard', **kwargs*)
Bases: `mvpa.misc.state.ClassWithCollections`

Base class to implement functors to select some elements based on a sequence of values.

Cheap initialization.

•*mode* (['discard', 'select']) – Decides whether to *select* or to *discard* features.

mode

See Also:

Derived classes might provide additional methods via their base classes. Please refer to the list of base classes (if it exists) at the begining of the `ElementSelector` documentation.

Full API documentation of ElementSelector in module mvpa.featsel.helpers.

FixedErrorThresholdStopCrit

class **FixedErrorThresholdStopCrit** (*threshold*)

    Bases: `mvpa.featsel.helpers.StoppingCriterion`

    Stop computation if the latest error drops below a certain threshold.

    Initialize with threshold.

            •*threshold* (float [0,1]) – Error threshold.

    threshold

See Also:

Derived classes might provide additional methods via their base classes. Please refer to the list of base classes (if it exists) at the begining of the `FixedErrorThresholdStopCrit` documentation.

Full API documentation of FixedErrorThresholdStopCrit in module mvpa.featsel.helpers.

FixedNElementTailSelector

class **FixedNElementTailSelector** (*nelements, **kwargs*)

    Bases: `mvpa.featsel.helpers.TailSelector`

    Given a sequence, provide set of IDs for a fixed number of to be selected elements.

    Cheap initialization.

            •*nselect* (int) – Number of elements to select/discard.

    nelements

See Also:

Derived classes might provide additional methods via their base classes. Please refer to the list of base classes (if it exists) at the begining of the `FixedNElementTailSelector` documentation.

Full API documentation of FixedNElementTailSelector in module mvpa.featsel.helpers.

FractionTailSelector

class **FractionTailSelector** (*felements, **kwargs*)

    Bases: `mvpa.featsel.helpers.TailSelector`

    Given a sequence, provide Ids for a fraction of elements

    Cheap initialization.

            •*felements* (float (0,1.0]) – Fraction of elements to select/discard. Note: Even when 0.0 is specified at least one element will be selected.

    felements

See Also:

Derived classes might provide additional methods via their base classes. Please refer to the list of base classes (if it exists) at the begining of the `FractionTailSelector` documentation.

Full API documentation of FractionTailSelector in module mvpa.featsel.helpers.

MultiStopCrit

class **MultiStopCrit** (*crits, mode='or'*)

    Bases: `mvpa.featsel.helpers.StoppingCriterion`

    Stop computation if the latest error drops below a certain threshold.

•*crits* (list of StoppingCriterion instances) – For each call to MultiStopCrit all of these criterions will be evaluated.

•*mode* (any of ('and', 'or')) – Logical function to determine the multi criterion from the set of base criteria.

See Also:

Derived classes might provide additional methods via their base classes. Please refer to the list of base classes (if it exists) at the begining of the `MultiStopCrit` documentation.

Full API documentation of MultiStopCrit in module mvpa.featsel.helpers.

NBackHistoryStopCrit

class **NBackHistoryStopCrit** (*bestdetector=<mvpa.featsel.helpers.BestDetector object at 0x8fad50c>, steps=10*)
    Bases: `mvpa.featsel.helpers.StoppingCriterion`

Stop computation if for a number of steps error was increasing

Initialize with number of steps

•*bestdetector* (BestDetector instance) – used to determine where the best error is located.

•*steps* (int) – How many steps to check after optimal value.

steps

See Also:

Derived classes might provide additional methods via their base classes. Please refer to the list of base classes (if it exists) at the begining of the `NBackHistoryStopCrit` documentation.

Full API documentation of NBackHistoryStopCrit in module mvpa.featsel.helpers.

NStepsStopCrit

class **NStepsStopCrit** (*steps*)
    Bases: `mvpa.featsel.helpers.StoppingCriterion`

Stop computation after a certain number of steps.

Initialize with number of steps.

•*steps* (int) – Number of steps after which to stop.

steps

See Also:

Derived classes might provide additional methods via their base classes. Please refer to the list of base classes (if it exists) at the begining of the `NStepsStopCrit` documentation.

Full API documentation of NStepsStopCrit in module mvpa.featsel.helpers.

RangeElementSelector

class **RangeElementSelector** (*lower=None, upper=None, inclusive=False, mode='select', **kwargs*)
    Bases: `mvpa.featsel.helpers.ElementSelector`

Select elements based on specified range of values

Initialization *RangeElementSelector*

•*lower* – If not None – select elements which are above of specified value

•*upper* – If not None – select elements which are lower of specified value

•*inclusive* – Either to include end points

•*mode* – overrides parent's default to be 'select' since it is more native for RangeElementSelector XXX TODO – unify??

upper could be lower than *lower* – then selection is done on values <= lower or >=upper (ie tails). This would produce the same result if called with flipped values for mode and inclusive.

If no upper no lower is set, assuming upper,lower=0, thus outputing non-0 elements

See Also:

Derived classes might provide additional methods via their base classes. Please refer to the list of base classes (if it exists) at the begining of the `RangeElementSelector` documentation.

Full API documentation of RangeElementSelector in module mvpa.featsel.helpers.

StoppingCriterion

class StoppingCriterion()

Bases: `object`

Base class for all functors to decide when to stop RFE (or may be general optimization... so it probably will be moved out into some other module

See Also:

Derived classes might provide additional methods via their base classes. Please refer to the list of base classes (if it exists) at the begining of the `StoppingCriterion` documentation.

Full API documentation of StoppingCriterion in module mvpa.featsel.helpers.

TailSelector

class TailSelector(*tail='lower', sort=True, **kwargs*)

Bases: `mvpa.featsel.helpers.ElementSelector`

Select elements from a tail of a distribution.

The default behaviour is to discard the lower tail of a given distribution.

Initialize TailSelector

•*tail* (['lower', 'upper']) – Choose the tail to be processed.

•*sort* (bool) – Flag whether selected IDs will be sorted. Disable if not necessary to save some CPU cycles.

See Also:

Derived classes might provide additional methods via their base classes. Please refer to the list of base classes (if it exists) at the begining of the `TailSelector` documentation.

Full API documentation of TailSelector in module mvpa.featsel.helpers.

16.6.4 `mvpa.featsel.ifs`

Incremental feature search (IFS).

Very similar to Recursive feature elimination (RFE), but instead of begining with all features and stripping some sequentially, start with an empty feature set and include important features successively.

The comprehensive API documentation for this module, including all technical details, is available in the Epydoc-generated API reference for mvpa.featsel.ifs (for developers).

class IFS()
>   Bases: `mvpa.featsel.base.FeatureSelection`
>
>   Incremental feature search.
>
>   A scalar *DatasetMeasure* is computed multiple times on variations of a certain dataset. These measures are in turn used to incrementally select important features. Starting with an empty feature set the dataset measure is first computed for each single feature. A number of features is selected based on the resulting data measure map (using an *ElementSelector*).
>
>   Next the dataset measure is computed again using each feature in addition to the already selected feature set. Again the *ElementSelector* is used to select more features.
>
>   For each feature selection the transfer error on some testdatset is computed. This procedure is repeated until a given *StoppingCriterion* is reached.
>
>   Initialize incremental feature search
>
>> • *data_measure* (DatasetMeasure) – Computed for each candidate feature selection.
>> • *transfer_error* (TransferError) – Compute against a test dataset for each incremental feature set.
>> • *bestdetector* (Functor) – Given a list of error values it has to return a boolean that signals whether the latest error value is the total minimum.
>> • *stopping_criterion* (Functor) – Given a list of error values it has to return whether the criterion is fulfilled.

See Also:

Derived classes might provide additional methods via their base classes. Please refer to the list of base classes (if it exists) at the begining of the `IFS` documentation.

Full API documentation of IFS in module mvpa.featsel.ifs.

16.6.5 `mvpa.featsel.rfe`

Recursive feature elimination.

The comprehensive API documentation for this module, including all technical details, is available in the Epydoc-generated API reference for mvpa.featsel.rfe (for developers).

class RFE (*sensitivity_analyzer, transfer_error, feature_selector=FractionTailSelector() fraction=0.050000, bestdetector=<mvpa.featsel.helpers.BestDetector object at 0x8e8fa0c>, stopping_criterion=<mvpa.featsel.helpers.NBackHistoryStopCrit object at 0x8e8f9ec>, train_clf=None, update_sensitivity=True, **kargs*)
>   Bases: `mvpa.featsel.base.FeatureSelection`
>
>   Recursive feature elimination.
>
>   A *FeaturewiseDatasetMeasure* is used to compute sensitivity maps given a certain dataset. These sensitivity maps are in turn used to discard unimportant features. For each feature selection the transfer error on some testdatset is computed. This procedure is repeated until a given *StoppingCriterion* is reached.
>
>   Such strategy after
>>   Guyon, I., Weston, J., Barnhill, S., & Vapnik, V. (2002). Gene selection for cancer classification using support vector machines. Mach. Learn., 46(1-3), 389–422.
>
>   was applied to SVM-based analysis of fMRI data in
>>   Hanson, S. J. & Halchenko, Y. O. (2008). Brain reading using full brain support vector machines for object recognition: there is no "face identification area". Neural Computation, 20, 486–503.
>
>   Initialize recursive feature elimination

> •*sensitivity_analyzer* (FeaturewiseDatasetMeasure object) –
>
> •*transfer_error* (TransferError object) – used to compute the transfer error of a classifier based on a certain feature set on the test dataset. NOTE: If sensitivity analyzer is based on the same classifier as transfer_error is using, make sure you initialize transfer_error with train=False, otherwise it would train classifier twice without any necessity.
>
> •*feature_selector* (Functor) – Given a sensitivity map it has to return the ids of those features that should be kept.
>
> •*bestdetector* (Functor) – Given a list of error values it has to return a boolean that signals whether the latest error value is the total minimum.
>
> •*stopping_criterion* (Functor) – Given a list of error values it has to return whether the criterion is fulfilled.
>
> •*train_clf* (bool) – Flag whether the classifier in *transfer_error* should be trained before computing the error. In general this is required, but if the *sensitivity_analyzer* and *transfer_error* share and make use of the same classifier it can be switched off to save CPU cycles. Default *None* checks if sensitivity_analyzer is based on a classifier and doesn't train if so.
>
> •*update_sensitivity* (bool) – If False the sensitivity map is only computed once and reused for each iteration. Otherwise the senstitivities are recomputed at each selection step.

See Also:

Derived classes might provide additional methods via their base classes. Please refer to the list of base classes (if it exists) at the begining of the RFE documentation.

Full API documentation of RFE in module mvpa.featsel.rfe.

16.7 Additional Algorithms

16.7.1 `mvpa.algorithms`

Import helper for PyMVPA algorithms.

The comprehensive API documentation for this module, including all technical details, is available in the Epydoc-generated API reference for mvpa.algorithms (for developers).

16.7.2 `mvpa.algorithms.cvtranserror`

Cross-validate a classifier on a dataset

The comprehensive API documentation for this module, including all technical details, is available in the Epydoc-generated API reference for mvpa.algorithms.cvtranserror (for developers).

CrossValidatedTransferError

class CrossValidatedTransferError(*transerror, splitter=<mvpa.datasets.splitter.NoneSplitter object at 0x8b564cc>, combiner=<function GrandMean at 0x8d1bf0c>, expose_testdataset=False, harvest_attribs=None, copy_attribs='copy', **kwargs*)

   Bases: mvpa.measures.base.DatasetMeasure, mvpa.misc.state.Harvestable

   Cross validate a classifier on datasets generated by a splitter from a source dataset.

   Arbitrary performance/error values can be computed by specifying an error function (used to compute an error value for each cross-validation fold) and a combiner function that aggregates all computed error values across cross-validation folds.

   Cheap initialization.

   > •*transerror* (TransferError instance) – Provides the classifier used for cross-validation.

> •*splitter* (Splitter instance) – Used to split the dataset for cross-validation folds. By convention the first dataset in the tuple returned by the splitter is used to train the provided classifier. If the first element is 'None' no training is performed. The second dataset is used to generate predictions with the (trained) classifier.
>
> •*combiner* (Functor) – Used to aggregate the error values of all cross-validation folds.
>
> •*expose_testdataset* (bool) – In the proper pipeline, classifier must not know anything about testing data, but in some cases it might lead only to marginal harm, thus migth wanted to be enabled (provide testdataset for RFE to determine stopping point).
>
> •*harvest_attribs* (list of basestr) – What attributes of call to store and return within harvested state variable
>
> •*copy_attribs* (None or basestr) – Force copying values of attributes on harvesting

> **combiner**
>
> **splitter**
>
> **transerror**

See Also:

Derived classes might provide additional methods via their base classes. Please refer to the list of base classes (if it exists) at the begining of the `CrossValidatedTransferError` documentation.

Full API documentation of CrossValidatedTransferError in module mvpa.algorithms.cvtranserror.

16.8 Common Facilities

16.8.1 `mvpa.base`

Base functionality of PyMVPA

Module Organization

mvpa.base module contains various modules which are used through out PyMVPA code, and are generic building blocks

> group Basic
>> externals, config, verbosity, dochelpers

The comprehensive API documentation for this module, including all technical details, is available in the Epydoc-generated API reference for mvpa.base (for developers).

WarningLog

class **WarningLog** (*btlevels=10, btdefault=False, maxcount=1, *args, **kwargs*)
> Bases: `mvpa.base.verbosity.OnceLogger`
>
> Logging class of messsages to be printed just once per each messsage
>
> Define Warning logger.
>
> > btlevels
> >> [int] how many levels of backtrack to print to give a hint on WTF
> > btdefault
> >> [bool] if to print backtrace for all warnings at all
> > maxcount
> >> [int] how many times to print each warning
>
> **maxcount**

See Also:

Derived classes might provide additional methods via their base classes. Please refer to the list of base classes (if it exists) at the begining of the `WarningLog` documentation.

Full API documentation of WarningLog in module mvpa.base.

16.8.2 `mvpa.base.config`

Registry-like monster

The comprehensive API documentation for this module, including all technical details, is available in the Epydoc-generated API reference for mvpa.base.config (for developers).

`ConfigManager`

class `ConfigManager` (*filenames=None*)
　　Bases: `ConfigParser.SafeConfigParser`

　　Central configuration registry for PyMVPA.

　　The purpose of this class is to collect all configurable settings used by various parts of PyMVPA. It is fairly simple and does only little more than the standard Python ConfigParser. Like ConfigParser it is blind to the data that it stores, i.e. not type checking is performed.

　　Configuration files (INI syntax) in multiple location are passed when the class is instanciated or whenever *Config.reload()* is called later on. By default it looks for a config file named *pymvpa.cfg* in the current directory and *.pymvpa.cfg* in the user's home directory. Morever, the constructor takes an optional argument with a list of additional file names to parse.

　　In addition to configuration files, this class also looks for special environment variables to read settings from. Names of such variables have to start with *MVPA_* following by the an optional section name and the variable name itself ('_' as delimiter). If no section name is provided, the variables will be associated with section *general*. Some examples:

```
MVPA_VERBOSE=1
```

　　will become:

```
[general]
verbose = 1
```

　　However, *MVPA_VERBOSE_OUTPUT=stdout* becomes:

```
[verbose]
output = stdout
```

　　Any lenght of variable name as allowed, e.g. MVPA_SEC1_LONG_VARIABLE_NAME=1 becomes:

```
[sec1]
long variable name = 1
```

　　Settings from custom configuration files (specified by the constructor argument) have the highest priority and override settings found in the current directory. They in turn override user-specific settings and finally the content of any *MVPA_** environment variables overrides all settings read from any file.

　　Initialization reads settings from config files and env. variables.

　　　　　•*filenames* (list of filenames) –

get (*section, option, default=None, **kwargs*)
　　Wrapper around SafeConfigParser.get() with a custom default value.

　　This method simply wraps the base class method, but adds a *default* keyword argument. The value of *default* is returned whenever the config parser does not have the requested option and/or section.

> **getAsDType** (*section, option, dtype, default=None*)
>> Convenience method to query options with a custom default and type
>>
>> This method simply wraps the base class method, but adds a *default* keyword argument. The value of *default* is returned whenever the config parser does not have the requested option and/or section.
>>
>> In addition, the returned value is converted into the specified *dtype*.
>
> **getboolean** (*section, option, default=None*)
>> Wrapper around SafeConfigParser.getboolean() with a custom default.
>>
>> This method simply wraps the base class method, but adds a *default* keyword argument. The value of *default* is returned whenever the config parser does not have the requested option and/or section.
>
> **reload** ()
>> Re-read settings from all configured locations.
>
> **save** (*filename*)
>> Write current configuration to a file.

See Also:

Derived classes might provide additional methods via their base classes. Please refer to the list of base classes (if it exists) at the begining of the ConfigManager documentation.

Full API documentation of ConfigManager in module mvpa.base.config.

16.8.3 `mvpa.base.dochelpers`

The comprehensive API documentation for this module, including all technical details, is available in the Epydoc-generated API reference for mvpa.base.dochelpers (for developers).

Functions

enhancedClassDocString (*cls, *args*)
> Generate enhanced doc strings but given a class, not just a name.
>
> It is to be used from a collector, ie whenever class is already created

See Also:

Full API documentation of enhancedClassDocString() in module mvpa.base.dochelpers.

enhancedDocString (*name, lcl, *args*)
> Generate enhanced doc strings.

See Also:

Full API documentation of enhancedDocString() in module mvpa.base.dochelpers.

handleDocString (*text*)
> Take care of empty and non existing doc strings.

See Also:

Full API documentation of handleDocString() in module mvpa.base.dochelpers.

rstUnderline (*text, markup*)
> Add and underline RsT string matching the length of the given string.

See Also:

Full API documentation of rstUnderline() in module mvpa.base.dochelpers.

table2string (*table, out=None*)
> Given list of lists figure out their common widths and print to out
>
>> •*table* (list of lists of strings) – What is aimed to be printed
>>
>> •*out* (None or stream) – Where to print. If None – will print and return string

> Return type
>> string if out was None

See Also:

Full API documentation of table2string() in module mvpa.base.dochelpers.

16.8.4 `mvpa.base.externals`

Helper to verify presence of external libraries and modules

The comprehensive API documentation for this module, including all technical details, is available in the Epydoc-generated API reference for mvpa.base.externals (for developers).

Functions

exists (*dep, force=False, raiseException=False*)
> Test whether a known dependency is installed on the system.

> This method allows us to test for individual dependencies without testing all known dependencies. It also ensures that we only test for a dependency once.

>> •*dep* (string or list of string) – The dependency key(s) to test.
>> •*force* (boolean) – Whether to force the test even if it has already been performed.
>> •*raiseException* (boolean) – Whether to raise RuntimeError if dependency is missing.

See Also:

Full API documentation of exists() in module mvpa.base.externals.

testAllDependencies (*force=False*)
> Test for all known dependencies.

>> •*force* (boolean) – Whether to force the test even if it has already been performed.

See Also:

Full API documentation of testAllDependencies() in module mvpa.base.externals.

16.8.5 `mvpa.base.verbosity`

Verbose output and debugging facility

Examples: from verbosity import verbose, debug; debug.active = [1,2,3]; debug(1, "blah")

The comprehensive API documentation for this module, including all technical details, is available in the Epydoc-generated API reference for mvpa.base.verbosity (for developers).

Classes

`LevelLogger`

class **LevelLogger** (*level=0, indent=' ', *args, **kwargs*)
> Bases: `mvpa.base.verbosity.Logger`

> Logger which prints based on level – ie everything which is smaller than specified level

> Define level logger.

> It is defined by
>> *level*, int: to which messages are reported *indent*, string: symbol used to indent

> **indent**

level

See Also:

Derived classes might provide additional methods via their base classes. Please refer to the list of base classes (if it exists) at the begining of the `LevelLogger` documentation.

Full API documentation of LevelLogger in module mvpa.base.verbosity.

Logger

class Logger(*handlers=None*)

Bases: `object`

Base class to provide logging

Initialize the logger with a set of handlers to use for output

Each hanlder must have write() method implemented

handlers

Return active handlers

lfprev

See Also:

Derived classes might provide additional methods via their base classes. Please refer to the list of base classes (if it exists) at the begining of the `Logger` documentation.

Full API documentation of Logger in module mvpa.base.verbosity.

OnceLogger

class OnceLogger(**args, **kwargs*)

Bases: `mvpa.base.verbosity.Logger`

Logger which prints a message for a given ID just once.

It could be used for one-time warning to don't overfill the output with useless repeatative messages

Define once logger.

See Also:

Derived classes might provide additional methods via their base classes. Please refer to the list of base classes (if it exists) at the begining of the `OnceLogger` documentation.

Full API documentation of OnceLogger in module mvpa.base.verbosity.

SetLogger

class SetLogger(*register=None, active=None, printsetid=True, *args, **kwargs*)

Bases: `mvpa.base.verbosity.Logger`

Logger which prints based on defined sets identified by Id.

active

printsetid

register(*setid, description*)

"Register" a new setid with a given description for easy finding

registered

setActiveFromString(*value*)

Given a string listing registered(?) setids, make then active

---

See Also:

Derived classes might provide additional methods via their base classes. Please refer to the list of base classes (if it exists) at the begining of the `SetLogger` documentation.

Full API documentation of SetLogger in module mvpa.base.verbosity.

16.9 Miscellaneous

16.9.1 `mvpa.misc`

Import helper for PyMVPA misc modules

The comprehensive API documentation for this module, including all technical details, is available in the Epydoc-generated API reference for mvpa.misc (for developers).

16.9.2 `mvpa.misc.args`

Helpers for arguments handling.

The comprehensive API documentation for this module, including all technical details, is available in the Epydoc-generated API reference for mvpa.misc.args (for developers).

Functions

group_kwargs (*prefixes, assign=False, passthrough=False*)
    Decorator function to join parts of kwargs together

> •*prefixes* (list of basestrings) – Prefixes to split based on. See *split_kwargs*
> •*assign* (bool) – Flag to assign the obtained arguments to self._<prefix>_kwargs
> •*passthrough* (bool) – Flag to pass joined arguments as <prefix>_kwargs argument. Usually it is sufficient to have either assign or passthrough. If none of those is True, decorator simply filters out mentioned groups from being passed to the method

    Example: if needed to join all args which start with '**slave_**' together under slave_kwargs parameter

See Also:

Full API documentation of group_kwargs() in module mvpa.misc.args.

split_kwargs (*kwargs, prefixes=, []*)
    Helper to separate kwargs into multiple groups

> •*prefixes* (list of basestrings) – Each entry sets a prefix which puts entry with key starting with it into a separate group. Group '' corresponds to 'leftovers'

    Output
        dictionary with keys == *prefixes*

See Also:

Full API documentation of split_kwargs() in module mvpa.misc.args.

16.9.3 `mvpa.misc.bv`

Import helper for Brain Voyager

The comprehensive API documentation for this module, including all technical details, is available in the Epydoc-generated API reference for mvpa.misc.bv (for developers).

16.9.4 `mvpa.misc.bv.base`

Tiny snippets to interface with FSL easily.

The comprehensive API documentation for this module, including all technical details, is available in the Epydoc-generated API reference for mvpa.misc.bv.base (for developers).

BrainVoyagerRTC

class **BrainVoyagerRTC**(*source*)

> Bases: `mvpa.misc.io.base.ColumnData`
>
> IO helper to read BrainVoyager RTC files.
>
> This is a textfile format that is used to specify stimulation protocols for data analysis in BrainVoyager. It looks like
>
> FileVersion: 2 Type: DesignMatrix NrOfPredictors: 4 NrOfDataPoints: 147
>
> "fm_l_60dB" "fm_r_60dB" "fm_l_80dB" "fm_r_80dB" 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000
>
> Data is always read as *float* and header is actually ignored
>
> Read and write BrainVoyager RTC files.
>
> > •*source* (filename of an RTC file) –
>
> **toarray**()
> > Returns the data as an array

See Also:

Derived classes might provide additional methods via their base classes. Please refer to the list of base classes (if it exists) at the begining of the `BrainVoyagerRTC` documentation.

Full API documentation of BrainVoyagerRTC in module mvpa.misc.bv.base.

16.9.5 `mvpa.misc.cmdline`

Common functions and options definitions for command line

__docformat__ = 'restructuredtext'

Conventions: Every option (instance of optparse.Option) has prefix "opt". Lists of options has prefix opts (e.g. *opts.common*).

Option name should be camelbacked version of .dest for the option.

The comprehensive API documentation for this module, including all technical details, is available in the Epydoc-generated API reference for mvpa.misc.cmdline (for developers).

Classes

OptionGroups

class **OptionGroups**(*parser*)

> Bases: `object`
>
> Group creation is delayed until instance is requested.
>
> This allows to overcome the problem of poluting handled cmdline options
>
> **add**(*name, l, doc*)

See Also:

Derived classes might provide additional methods via their base classes. Please refer to the list of base classes (if it exists) at the begining of the `OptionGroups` documentation.

Full API documentation of OptionGroups in module mvpa.misc.cmdline.

Options

class Options()
    Bases: `object`

    Just a convinience placeholder for all available options

See Also:

Derived classes might provide additional methods via their base classes. Please refer to the list of base classes (if it exists) at the begining of the `Options` documentation.

Full API documentation of Options in module mvpa.misc.cmdline.

16.9.6 `mvpa.misc.data_generators`

Miscelaneous data generators for unittests and demos

The comprehensive API documentation for this module, including all technical details, is available in the Epydoc-generated API reference for mvpa.misc.data_generators (for developers).

Functions

chirpLinear(*n_instances, n_features=4, n_nonbogus_features=2, data_noise=0.40000000000000002, noise=0.10000000000000001*)
    Generates simple dataset for linear regressions

    Generates chirp signal, populates n_nonbogus_features out of n_features with it with different noise level and then provides signal itself with additional noise as labels

See Also:

Full API documentation of chirpLinear() in module mvpa.misc.data_generators.

dumbFeatureBinaryDataset()
    Very simple binary (2 labels) dataset

See Also:

Full API documentation of dumbFeatureBinaryDataset() in module mvpa.misc.data_generators.

dumbFeatureDataset()
    Create a very simple dataset with 2 features and 3 labels

See Also:

Full API documentation of dumbFeatureDataset() in module mvpa.misc.data_generators.

getMVPattern(*s2n*)
    Simple multivariate dataset

See Also:

Full API documentation of getMVPattern() in module mvpa.misc.data_generators.

linear_awgn(*size=10, intercept=0.0, slope=0.40000000000000002, noise_std=0.01, flat=False*)
    Generate a dataset from a linear function with Added White Gaussian Noise (AWGN). It can be multidimensional if 'slope' is a vector. If flat is True (in 1 dimesion) generate equally spaces samples instead of random ones. This is useful for the test phase.

See Also:

Full API documentation of linear_awgn() in module mvpa.misc.data_generators.

multipleChunks (*func, n_chunks, *args, **kwargs*)
> Replicate datasets multiple times raising different chunks
>
> Given some randomized (noisy) generator of a dataset with a single chunk call generator multiple times and place results into a distinct chunks

See Also:

Full API documentation of multipleChunks() in module mvpa.misc.data_generators.

noisy_2d_fx (*size_per_fx, dfx, sfx, center, noise_std=1*)

See Also:

Full API documentation of noisy_2d_fx() in module mvpa.misc.data_generators.

normalFeatureDataset (*perlabel=50, nlabels=2, nfeatures=4, nchunks=5, means=None, nonbogus_features=None, snr=1.0*)
> Generate a dataset where each label is some normally distributed beastie around specified mean (0 if None).
>
> snr is assuming that signal has std 1.0 so we just divide noise by snr
>
> Probably it is a generalization of pureMultivariateSignal where means=[[0,1], [1,0]]
>
> Specify either means or nonbogus_features so means get assigned accordingly

See Also:

Full API documentation of normalFeatureDataset() in module mvpa.misc.data_generators.

normalFeatureDataset__ (*dataset=None, labels=None, nchunks=None, perlabel=50, activation_probability_steps=1, randomseed=None, randomvoxels=False*)
> NOT FINISHED

See Also:

Full API documentation of normalFeatureDataset__() in module mvpa.misc.data_generators.

pureMultivariateSignal (*patterns, signal2noise=1.5, chunks=None*)
> Create a 2d dataset with a clear multivariate signal, but no univariate information.

```
%%%%%%%%%
% O % X %
%%%%%%%%%
% X % O %
%%%%%%%%%
```

See Also:

Full API documentation of pureMultivariateSignal() in module mvpa.misc.data_generators.

sinModulated (*n_instances, n_features, flat=False, noise=0.4000000000000002*)
> Generate a (quite) complex multidimensional non-linear dataset
>
> Used for regression testing. In the data label is a sin of a x^2 + uniform noise

See Also:

Full API documentation of sinModulated() in module mvpa.misc.data_generators.

wr1996 (*size=200*)
> Generate '6d robot arm' dataset (Williams and Rasmussen 1996)
>
> Was originally created in order to test the correctness of the implementation of kernel ARD. For full details see: http://www.gaussianprocess.org/gpml/code/matlab/doc/regression.html#ard
>
> x_1 picked randomly in [-1.932, -0.453] x_2 picked randomly in [0.534, 3.142] r_1 = 2.0 r_2 = 1.3 $f(x_1, x_2)$ = $r_1 \cos(x_1)$ + $r_2 \cos(x_1 + x_2)$ + $N(0, 0.0025)$ etc.
>
> Expected relevances: ell_1 1.804377 ell_2 1.963956 ell_3 8.884361 ell_4 34.417657 ell_5 1081.610451 ell_6 375.445823 sigma_f 2.379139 sigma_n 0.050835

16.9. Miscellaneous

See Also:

Full API documentation of wr1996() in module mvpa.misc.data_generators.

16.9.7 `mvpa.misc.errorfx`

Error functions helpers.

PyMVPA can use arbitrary function which takes 2 arguments: predictions and targets and spits out a scalar value. Functions below are for the convinience, and they confirm the agreement that 'smaller' is 'better'

The comprehensive API documentation for this module, including all technical details, is available in the Epydoc-generated API reference for mvpa.misc.errorfx (for developers).

Classes

AUCErrorFx

class **AUCErrorFx**()
    Bases: `mvpa.misc.errorfx._ErrorFx`
    Computes the area under the ROC for the given the target and predicted to make the prediction.

See Also:

Derived classes might provide additional methods via their base classes. Please refer to the list of base classes (if it exists) at the begining of the `AUCErrorFx` documentation.

Full API documentation of AUCErrorFx in module mvpa.misc.errorfx.

MeanMismatchErrorFx

class **MeanMismatchErrorFx**()
    Bases: `mvpa.misc.errorfx._ErrorFx`
    Computes the percentage of mismatches between some target and some predicted values.

See Also:

Derived classes might provide additional methods via their base classes. Please refer to the list of base classes (if it exists) at the begining of the `MeanMismatchErrorFx` documentation.

Full API documentation of MeanMismatchErrorFx in module mvpa.misc.errorfx.

RMSErrorFx

class **RMSErrorFx**()
    Bases: `mvpa.misc.errorfx._ErrorFx`
    Computes the root mean squared error of some target and some predicted values.

See Also:

Derived classes might provide additional methods via their base classes. Please refer to the list of base classes (if it exists) at the begining of the `RMSErrorFx` documentation.

Full API documentation of RMSErrorFx in module mvpa.misc.errorfx.

RelativeRMSErrorFx

class **RelativeRMSErrorFx**()
    Bases: `mvpa.misc.errorfx._ErrorFx`

Ratio between RMSE and root mean power of target output.

So it can be considered as a scaled RMSE – perfect reconstruction has values near 0, while no reconstruction has values around 1.0. Word of caution – it is not commutative, ie exchange of predicted and target might lead to completely different answers

See Also:

Derived classes might provide additional methods via their base classes. Please refer to the list of base classes (if it exists) at the begining of the `RelativeRMSErrorFx` documentation.

Full API documentation of RelativeRMSErrorFx in module mvpa.misc.errorfx.

`Variance1SVFx`

class **`Variance1SVFx`**`()`
> Bases: `mvpa.misc.errorfx._ErrorFx`
>
> Ratio of variance described by the first singular value component.
>
> Of limited use – left for the sake of not wasting it

See Also:

Derived classes might provide additional methods via their base classes. Please refer to the list of base classes (if it exists) at the begining of the `Variance1SVFx` documentation.

Full API documentation of Variance1SVFx in module mvpa.misc.errorfx.

Functions

`meanPowerFx``(data)`
> Returns mean power
>
> Similar to var but without demeaning

See Also:

Full API documentation of meanPowerFx() in module mvpa.misc.errorfx.

`rootMeanPowerFx``(data)`
> Returns root mean power
>
> to be comparable against RMSE

See Also:

Full API documentation of rootMeanPowerFx() in module mvpa.misc.errorfx.

16.9.8 `mvpa.misc.exceptions`

Exception classes which might get thrown

The comprehensive API documentation for this module, including all technical details, is available in the Epydoc-generated API reference for mvpa.misc.exceptions (for developers).

Classes

`ConvergenceError`

class **`ConvergenceError`**`()`
> Bases: `exceptions.Exception`
>
> Thrown if some algorithm does not converge to a solution.

See Also:

Derived classes might provide additional methods via their base classes. Please refer to the list of base classes (if it exists) at the begining of the ConvergenceError documentation.

Full API documentation of ConvergenceError in module mvpa.misc.exceptions.

DatasetError

class **DatasetError**(*msg=''*)
    Bases: exceptions.Exception

    Thrown if there is an internal problem with a Dataset.

    ValueError exception is too generic to be used for any needed case, thus this one is created

See Also:

Derived classes might provide additional methods via their base classes. Please refer to the list of base classes (if it exists) at the begining of the DatasetError documentation.

Full API documentation of DatasetError in module mvpa.misc.exceptions.

InvalidHyperparameterError

class **InvalidHyperparameterError**()
    Bases: exceptions.Exception

    Generic exception to be raised when setting improper values as hyperparameters.

See Also:

Derived classes might provide additional methods via their base classes. Please refer to the list of base classes (if it exists) at the begining of the InvalidHyperparameterError documentation.

Full API documentation of InvalidHyperparameterError in module mvpa.misc.exceptions.

UnknownStateError

class **UnknownStateError**(*msg=''*)
    Bases: exceptions.Exception

    Thrown if the internal state of the class is not yet defined.

    Classifiers and Algorithms classes might have properties, which are not defined prior to training or invocation has happened.

See Also:

Derived classes might provide additional methods via their base classes. Please refer to the list of base classes (if it exists) at the begining of the UnknownStateError documentation.

Full API documentation of UnknownStateError in module mvpa.misc.exceptions.

16.9.9 `mvpa.misc.fsl`

Import helper for FSL

The comprehensive API documentation for this module, including all technical details, is available in the Epydoc-generated API reference for mvpa.misc.fsl (for developers).

16.9.10 `mvpa.misc.fsl.base`

Tiny snippets to interface with FSL easily.

The comprehensive API documentation for this module, including all technical details, is available in the Epydoc-generated API reference for mvpa.misc.fsl.base (for developers).

Classes

FslEV3

class **FslEV3**(*source*)
> Bases: `mvpa.misc.io.base.ColumnData`

> IO helper to read FSL's EV3 files.

> This is a three-column textfile format that is used to specify stimulation protocols for fMRI data analysis in FSL's FEAT module.

> Data is always read as *float*.

> Read and write FSL EV3 files.

>> •*source* (filename of an EV3 file) –

> **durations**
>> durations

> **getEV**(*evid*)
>> Returns a tuple of (onset time, simulus duration, intensity) for a certain EV.

> **getNEVs**()
>> Returns the number of EVs in the file.

> **intensities**
>> intensities

> **nevs**
>> Returns the number of EVs in the file.

> **onsets**
>> onsets

> **toEvents**()
>> Convert into a list of *Event* instances.

> **tofile**(*filename*)
>> Write data to a FSL EV3 file.

See Also:

Derived classes might provide additional methods via their base classes. Please refer to the list of base classes (if it exists) at the begining of the `FslEV3` documentation.

Full API documentation of FslEV3 in module mvpa.misc.fsl.base.

McFlirtParams

class **McFlirtParams**(*source*)
> Bases: `mvpa.misc.io.base.ColumnData`

> Read and write McFlirt's motion estimation parameters from and to text files.

>> •*source* (str) – Filename of a parameter file.

> **plot**()
>> Produce a simple plot of the estimated translation and rotation parameters using.

>> You still need to can pylab.show() or pylab.savefig() if you want to see/get anything.

toarray()
   Returns the data as an array with six columns (same order as in file).

tofile(*filename*)
   Write motion parameters to file.

See Also:

Derived classes might provide additional methods via their base classes. Please refer to the list of base classes (if it exists) at the begining of the `McFlirtParams` documentation.

Full API documentation of McFlirtParams in module mvpa.misc.fsl.base.

16.9.11 `mvpa.misc.fsl.flobs`

Wrapper around FSLs halfcosbasis to generate HRF kernels

The comprehensive API documentation for this module, including all technical details, is available in the Epydoc-generated API reference for mvpa.misc.fsl.flobs (for developers).

makeFlobs(*pre=0, rise=5, fall=5, undershoot=5, undershootamp=0.29999999999999999, nsamples=1, resolution=0.05000000000000003, nsecs=-1, nbasisfns=2*)
   Wrapper around the FSL tool halfcosbasis.

   This function uses halfcosbasis to generate samples of HRF kernels. Kernel parameters can be modified analogous to the Make_flobs GUI which is part of FSL.

```
  ^         /-\
  |        /   \
  1       /     \
  |      /       \
  |     /         \
  |    /           \
-----/              \    /-----   |
                     \--/         |  undershootamp
  |    |      |      |            |
  |    |      |      |            |

   pre   rise  fall  undershoot
```
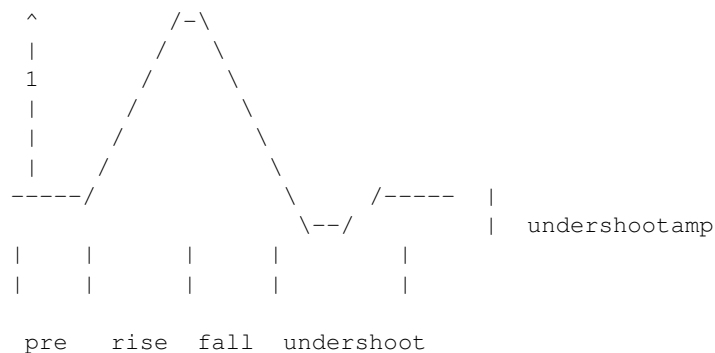
   Parameters 'pre', 'rise', 'fall', 'undershoot' and 'undershootamp' can be specified as 2-tuples (min-max range for sampling) and single value (setting exact values – no sampling).

   If 'nsec' is negative, the length of the samples is determined automatically to include the whole kernel function (until it returns to baseline). 'nsec' has to be an integer value and is set to the next greater integer value if it is not.

   All parameters except for 'nsamples' and 'nbasisfns' are in seconds.

See Also:

Full API documentation of makeFlobs() in module mvpa.misc.fsl.flobs.

16.9.12 `mvpa.misc.fsl.melodic`

The comprehensive API documentation for this module, including all technical details, is available in the Epydoc-generated API reference for mvpa.misc.fsl.melodic (for developers).

MelodicResults

See Also:

Derived classes might provide additional methods via their base classes. Please refer to the list of base classes (if it exists) at the begining of the `MelodicResults` documentation.

Full API documentation of MelodicResults in module mvpa.misc.fsl.melodic.

16.9.13 `mvpa.misc.fx`

Misc. functions (in the mathematical sense)

The comprehensive API documentation for this module, including all technical details, is available in the Epydoc-generated API reference for mvpa.misc.fx (for developers).

Functions

doubleGammaHRF (*t, A1=5.4000000000000004, W1=5.2000000000000002, K1=1.0, A2=10.800000000000001, W2=7.3499999999999996, K2=0.3499999999999998*)
Hemodynamic response function model.

The version is using two gamma functions (also see singleGammaHRF()).

- *t* (float) – Time.
- *A* (float) – Time to peak.
- *W* (float) – Full-width at half-maximum.
- *K* (float) – Scaling factor.

Parameters A, W and K exists individually for each of both gamma functions.

See Also:

Full API documentation of doubleGammaHRF() in module mvpa.misc.fx.

leastSqFit (*fx, params, y, x=None, **kwargs*)
Simple convenience wrapper around SciPy's optimize.leastsq.

The advantage of using this wrapper instead of optimize.leastsq directly is, that it automatically constructs an appropriate error function and easily deals with 2d data arrays, i.e. each column with multiple values for the same function argument (*x*-value).

- *fx* (functor) – Function to be fitted to the data. It has to take a vector with function arguments (*x*-values) as the first argument, followed by an arbitrary number of (to be fitted) parameters.
- *params* (sequence) – Sequence of start values for all to be fitted parameters. During fitting all parameters in this sequences are passed to the function in the order in which they appear in this sequence.
- *y* (1d or 2d array) – The data the function is fitted to. In the case of a 2d array, each column in the array is considered to be multiple observations or measurements of function values for the same *x*-value.
- *x* (Corresponding function arguments (*x*-values) for each datapoint, i.e.) – element in *y* or columns in *y', in the case of 'y* being a 2d array. If *x* is not provided it will be generated by *N.arange(m)*, where *m* is either the length of *y* or the number of columns in *y*, if *y* is a 2d array.
- ***kwargs* – All additonal keyword arguments are passed to *fx*.

Return type
    tuple

Returns
    i.e. 2-tuple with list of final (fitted) parameters of *fx* and an integer value indicating success or failure of the fitting procedure (see leastsq docs for more information).

See Also:

Full API documentation of leastSqFit() in module mvpa.misc.fx.

singleGammaHRF (*t, A=5.4000000000000004, W=5.2000000000000002, K=1.0*)
Hemodynamic response function model.

The version consists of a single gamma function (also see doubleGammaHRF()).

•*t* (float) – Time.

•*A* (float) – Time to peak.

•*W* (float) – Full-width at half-maximum.

•*K* (float) – Scaling factor.

See Also:

Full API documentation of singleGammaHRF() in module mvpa.misc.fx.

16.9.14 `mvpa.misc.io`

Import helper for IO helpers

The comprehensive API documentation for this module, including all technical details, is available in the Epydoc-generated API reference for mvpa.misc.io (for developers).

16.9.15 `mvpa.misc.io.base`

Some little helper for reading (and writing) common formats from and to disk.

The comprehensive API documentation for this module, including all technical details, is available in the Epydoc-generated API reference for mvpa.misc.io.base (for developers).

Classes

ColumnData

class **ColumnData**(*source, header=True, sep=None, headersep=None, dtype=<type 'float'>, skiplines=0*)

Bases: `dict`

Read data that is stored in columns of text files.

All read data is available via a dictionary-like interface. If column headers are available, the column names serve as dictionary keys. If no header exists an articfical key is generated: str(number_of_column).

Splitting of text file lines is performed by the standard split() function (which gets passed the *sep* argument as separator string) and each element is converted into the desired datatype.

Because data is read into a dictionary no two columns can have the same name in the header! Each column is stored as a list in the dictionary.

Read data from file into a dictionary.

•*source* (basestring or dict) – If values is given as a string all data is read from the file and additonal keyword arguments can be sued to customize the read procedure. If a dictionary is passed a deepcopy is performed.

•*header* (bool or list of basestring) – Indicates whether the column names should be read from the first line (*header=True*). If *header=False* unique column names will be generated (see class docs). If *header* is a python list, it's content is used as column header names and its length has to match the number of columns in the file.

•*sep* (basestring or None) – Separator string. The actual meaning depends on the output format (see class docs).

•*headersep* (basestring or None) – Separator string used in the header. The actual meaning depends on the output format (see class docs).

•*dtype* (type or list(types)) – Desired datatype(s). Datatype per column get be specified by passing a list of types.

•*skiplines* (int) – Number of lines to skip at the beginning of the file.

getNColumns()

Returns the number of columns.

getNRows()
> Returns the number of rows.

ncolumns
> Returns the number of columns.

nrows
> Returns the number of rows.

selectSamples(*selection*)
> Return new ColumnData with selected samples

tofile(*filename, header=True, header_order=None, sep=' '*)
> Write column data to a text file.

>> •*filename* (Think about it!) –
>> •*header* (If *True* a column header is written, using the column) – keys. If *False* no header is written.
>> •*header_order* (If it is a list of strings, they will be used instead) – of simply asking for the dictionary keys. However these strings must match the dictionary keys in number and identity. This argument type can be used to determine the order of the columns in the output file. The default value is *None*. In this case the columns will be in an arbitrary order.
>> •*sep* (String that is written as a separator between to data columns.) –

See Also:

Derived classes might provide additional methods via their base classes. Please refer to the list of base classes (if it exists) at the begining of the `ColumnData` documentation.

Full API documentation of ColumnData in module mvpa.misc.io.base.

DataReader

class DataReader()
> Bases: `object`

> Base class for data readers.

> Every subclass has to put all information into to variable:

> *self._data*: ndarray
>> The data array has to have the samples separating dimension along the first axis.

> *self._props*: dict
>> All other meaningful information has to be stored in a dictionary.

> This class provides two methods (and associated properties) to retrieve this information.

> Cheap init.

> **data**
>> Data array

> **getData**()
>> Return the data array.

> **getPropsAsDict**()
>> Return the dictionary with the data properties.

> **props**
>> Property dict

See Also:

Derived classes might provide additional methods via their base classes. Please refer to the list of base classes (if it exists) at the begining of the `DataReader` documentation.

Full API documentation of DataReader in module mvpa.misc.io.base.

SampleAttributes

class SampleAttributes (*source, literallabels=False*)

Bases: `mvpa.misc.io.base.ColumnData`

Read and write PyMVPA sample attribute definitions from and to text files.

Read PyMVPA sample attributes from disk.

> • *source* (filename of an atrribute file) –

getNSamples()

Returns the number of samples in the file.

nsamples

Returns the number of samples in the file.

tofile (*filename*)

Write sample attributes to a text file.

See Also:

Derived classes might provide additional methods via their base classes. Please refer to the list of base classes (if it exists) at the begining of the `SampleAttributes` documentation.

Full API documentation of SampleAttributes in module mvpa.misc.io.base.

SensorLocations

class SensorLocations (**args, **kwargs*)

Bases: `mvpa.misc.io.base.ColumnData`

Base class for sensor location readers.

Each subclass should provide x, y, z coordinates via the *pos_x*, *pos_y*, and *pos_z* attrbibutes.

Axes should follow the following convention:

> x-axis: left -> right y-axis: anterior -> posterior z-axis: superior -> inferior

Pass arguments to ColumnData.

locations()

Get the sensor locations as an array.

> Return type
>> (nchannels x 3) array with coordinates in (x, y, z)

See Also:

Derived classes might provide additional methods via their base classes. Please refer to the list of base classes (if it exists) at the begining of the `SensorLocations` documentation.

Full API documentation of SensorLocations in module mvpa.misc.io.base.

TuebingenMEGSensorLocations

class TuebingenMEGSensorLocations (*source*)

Bases: `mvpa.misc.io.base.SensorLocations`

Read sensor location definitions from a specific text file format.

File layout is assumed to be 7 columns:

> 1: sensor name 2: position on y-axis 3: position on x-axis 4: position on z-axis 5-7: same as 2-4, but for some outer surface thingie.

Note that x and y seem to be swapped, ie. y as defined by SensorLocations conventions seems to be first axis and followed by x.

Only inner surface coordinates are reported by *locations()*.

Read sensor locations from file.

> •*source* (filename of an attribute file) –

See Also:

Derived classes might provide additional methods via their base classes. Please refer to the list of base classes (if it exists) at the begining of the `TuebingenMEGSensorLocations` documentation.

Full API documentation of TuebingenMEGSensorLocations in module mvpa.misc.io.base.

XAVRSensorLocations

class XAVRSensorLocations(*source*)

> Bases: `mvpa.misc.io.base.SensorLocations`
>
> Read sensor location definitions from a specific text file format.
>
> File layout is assumed to be 5 columns:
>
> > 1.sensor name
> >
> > 2.some useless integer
> >
> > 3.position on x-axis
> >
> > 4.position on y-axis
> >
> > 5.position on z-axis
>
> Read sensor locations from file.
>
> > •*source* (filename of an attribute file) –

See Also:

Derived classes might provide additional methods via their base classes. Please refer to the list of base classes (if it exists) at the begining of the `XAVRSensorLocations` documentation.

Full API documentation of XAVRSensorLocations in module mvpa.misc.io.base.

Functions

design2labels(*columndata, baseline_label=0, func=<function <lambda> at 0x904f8b4>*)

> Helper to convert design matrix into a list of labels
>
> Given a design, assign a single label to any given sample
>
> TODO: fix description/naming
>
> > •*columndata* (ColumnData) – Attributes where each known will be considered as a separate explanatory variable (EV) in the design.
> >
> > •*baseline_label* – What label to assign for samples where none of EVs was given a value
> >
> > •*func* (functor) – Function which decides either a value should be considered
>
> > Output
> >
> > > list of labels which are taken from column names in ColumnData and baseline_label

See Also:

Full API documentation of design2labels() in module mvpa.misc.io.base.

labels2chunks(*labels, method='alllabels', ignore_labels=None*)

See Also:

Full API documentation of labels2chunks() in module mvpa.misc.io.base.

16.9.16 `mvpa.misc.io.eepbin`

Reader for binary EEP files.

The comprehensive API documentation for this module, including all technical details, is available in the Epydoc-generated API reference for mvpa.misc.io.eepbin (for developers).

EEPBin

class **EEPBin** (*source*)

>Bases: `mvpa.misc.io.base.DataReader`

>Read-access to binary EEP files.

>EEP files are used by *eeprobe* a software for analysing even-related potentials (ERP), which was developed at the Max-Planck Institute for Cognitive Neuroscience in Leipzig, Germany.

>>http://www.ant-neuro.com/products/eeprobe

>EEP files consist of a plain text header and a binary data block in a single file. The header starts with a line of the form

>';%d %d %d %g %g' % (Nchannels, Nsamples, Ntrials, t0, dt)

>where Nchannels, Nsamples, Ntrials are the numbers of channels, samples per trial and trials respectively. t0 is the time of the first sample of a trial relative to the stimulus onset and dt is the sampling interval.

>The binary data block consists of single precision floats arranged in the following way:

```
<trial1,channel1,sample1>,<trial1,channel1,sample2>,...
<trial1,channel2,sample1>,<trial1,channel2,sample2>,...
.
<trial2,channel1,sample1>,<trial2,channel1,sample2>,...
<trial2,channel2,sample1>,<trial2,channel2,sample2>,...
```

>Read EEP file and store header and data.

>>•*source* (str) – Filename.

>**channels**
>>List of channel names

>**dt**
>>Time difference between two adjacent samples

>**nchannels**
>>Number of channels

>**nsamples**
>>Number of trials/samples

>**ntimepoints**
>>Number of data timepoints

>**t0**
>>Relative start time of sampling interval

See Also:

Derived classes might provide additional methods via their base classes. Please refer to the list of base classes (if it exists) at the begining of the `EEPBin` documentation.

Full API documentation of EEPBin in module mvpa.misc.io.eepbin.

16.9.17 `mvpa.misc.io.hamster`

Helper for simple storage facility via cPickle and zlib

The comprehensive API documentation for this module, including all technical details, is available in the Epydoc-generated API reference for mvpa.misc.io.hamster (for developers).

Hamster

class **Hamster**(*args, **kwargs*)
> Bases: `dict`

> Simple container class which is derived from the dictionary

> It is capable of storing itself in a file, or loading from a file (using cPickle + zlib tandem). Any serializable object can be bound to a hamster to be stored.

> To undig burried hamster use Hamster(filename). Here is an example:

```
>>> h = Hamster(bla='blai')
>>> h.boo = N.arange(5)
>>> h.dump(filename)
...
>>> h = Hamster(filename)
```

> Initialize Hamster.

> Providing a single parameter string would treat it as a filename from which to undig the data. Otherwise all the parameters are equivalent to the ones of dict

> **dump** (*filename*)
> > Bury the hamster into the file

See Also:

Derived classes might provide additional methods via their base classes. Please refer to the list of base classes (if it exists) at the begining of the `Hamster` documentation.

Full API documentation of Hamster in module mvpa.misc.io.hamster.

16.9.18 `mvpa.misc.io.meg`

IO helper for MEG datasets.

The comprehensive API documentation for this module, including all technical details, is available in the Epydoc-generated API reference for mvpa.misc.io.meg (for developers).

TuebingenMEG

class **TuebingenMEG**(*source*)
> Bases: `object`

> Reader for MEG data from line-based textfile format.

> This class reads segmented MEG data from a textfile, which is created by converting the proprietary binary output files of a MEG device in Tuebingen (Germany) with an unkown tool.

> The file format is line-based, i.e. all timepoints for all samples/trials are written in a single line. Each line is prefixed with an identifier (using a colon as the delimiter between identifier and data). Two lines have a special purpose. The first 'Sample Number' is a list of timepoint ids, similar to *range(ntimepoints)* for each sample/trial (all concatenated into one line. The second 'Time' contains the timing information for each timepoint (relative to stimulus onset), again for all trials concatenated into a single line.

> All other lines contain various information (channels) recorded during the experiment. The meaning of some channels is unknown. Known ones are:

> > M*: MEG channels EEG*: EEG channels ADC*: Analog to digital converter output

> Dataset properties are available from various class attributes. The *data* member provides all data from all channels (except for 'Sample Number' and 'Time') in a NumPy array (nsamples x nchannels x ntimepoints).

> The reader supports uncompressed as well as gzipped input files (or other file-like objects).

> Reader MEG data from texfiles or file-like objects.

•*source* (str | file-like) – Strings are assumed to be filenames (with .*gz* suffix compressed), while all other object types are treated as file-like objects.

See Also:

Derived classes might provide additional methods via their base classes. Please refer to the list of base classes (if it exists) at the begining of the `TuebingenMEG` documentation.

Full API documentation of TuebingenMEG in module mvpa.misc.io.meg.

16.9.19 `mvpa.misc.param`

Parameter representation

The comprehensive API documentation for this module, including all technical details, is available in the Epydoc-generated API reference for mvpa.misc.param (for developers).

Classes

`KernelParameter`

class **KernelParameter** (*default, name=None, doc=None, **kwargs*)
    Bases: `mvpa.misc.param.Parameter`

    Just that it is different beast

    Specify a parameter by its default value and optionally an arbitrary number of additional parameters.

    TODO: :Parameters: for Parameter

See Also:

Derived classes might provide additional methods via their base classes. Please refer to the list of base classes (if it exists) at the begining of the `KernelParameter` documentation.

Full API documentation of KernelParameter in module mvpa.misc.param.

`Parameter`

class **Parameter** (*default, name=None, doc=None, **kwargs*)
    Bases: `mvpa.misc.state.CollectableAttribute`

    This class shall serve as a representation of a parameter.

    It might be useful if a little more information than the pure parameter value is required (or even only useful).

    Each parameter must have a value. However additional property can be passed to the constructor and will be stored in the object.

    BIG ASSUMPTION: stored values are not mutable, ie nobody should do

    cls.parameter1[:] = ...

    or we wouldn't know that it was changed

    Here is a list of possible property names:

        min - minimum value max - maximum value step - increment/decrement stepsize

    Specify a parameter by its default value and optionally an arbitrary number of additional parameters.

    TODO: :Parameters: for Parameter

    default

    equalDefault
        Returns True if current value is equal to default one

isDefault
    Returns True if current value is bound to default one

resetvalue()
    Reset value to the default

setDefault(*value*)

value

See Also:

Derived classes might provide additional methods via their base classes. Please refer to the list of base classes (if it exists) at the begining of the `Parameter` documentation.

Full API documentation of Parameter in module mvpa.misc.param.

16.9.20 `mvpa.misc.plot`

The comprehensive API documentation for this module, including all technical details, is available in the Epydoc-generated API reference for mvpa.misc.plot (for developers).

16.9.21 `mvpa.misc.plot.base`

The comprehensive API documentation for this module, including all technical details, is available in the Epydoc-generated API reference for mvpa.misc.plot.base (for developers).

Functions

See Also:

Full API documentation of errLinePlot() in module mvpa.misc.plot.base.

See Also:

Full API documentation of inverseCmap() in module mvpa.misc.plot.base.

See Also:

Full API documentation of plotBars() in module mvpa.misc.plot.base.

See Also:

Full API documentation of plotDatasetChunks() in module mvpa.misc.plot.base.

See Also:

Full API documentation of plotFeatureHist() in module mvpa.misc.plot.base.

See Also:

Full API documentation of plotSamplesDistance() in module mvpa.misc.plot.base.

16.9.22 `mvpa.misc.plot.erp`

The comprehensive API documentation for this module, including all technical details, is available in the Epydoc-generated API reference for mvpa.misc.plot.erp (for developers).

Functions

See Also:

Full API documentation of plotERP() in module mvpa.misc.plot.erp.

See Also:

Full API documentation of plotERPs() in module mvpa.misc.plot.erp.

16.9.23 `mvpa.misc.plot.topo`

The comprehensive API documentation for this module, including all technical details, is available in the Epydoc-generated API reference for mvpa.misc.plot.topo (for developers).

Functions

See Also:

Full API documentation of plotHeadOutline() in module mvpa.misc.plot.topo.

See Also:

Full API documentation of plotHeadTopography() in module mvpa.misc.plot.topo.

16.9.24 `mvpa.misc.state`

Classes to control and store state information.

It was devised to provide conditional storage

The comprehensive API documentation for this module, including all technical details, is available in the Epydoc-generated API reference for mvpa.misc.state (for developers).

Classes

`AttributeWithUnique`

class **AttributeWithUnique**(*name=None, hasunique=True, doc='Attribute with unique'*)

    Bases: `mvpa.misc.state.CollectableAttribute`

    Container which also takes care about recomputing unique values

    XXX may be we could better link original attribute to additional attribute which actually stores the values (and do reverse there as well).

> •don't need to mess with getattr since it would become just another attribute
> •might be worse design in terms of comprehension
> •take care about _set, since we shouldn't allow change it externally

    For now lets do it within a single class and tune up getattr

    hasunique

    reset()

    uniqueValues

See Also:

Derived classes might provide additional methods via their base classes. Please refer to the list of base classes (if it exists) at the begining of the `AttributeWithUnique` documentation.

Full API documentation of AttributeWithUnique in module mvpa.misc.state.

`AttributesCollector`

class **AttributesCollector**(*name, bases, dict*)

    Bases: `type`

    Intended to collect and compose StateCollection for any child class of this metaclass

See Also:

Derived classes might provide additional methods via their base classes. Please refer to the list of base classes (if it exists) at the begining of the `AttributesCollector` documentation.

Full API documentation of AttributesCollector in module mvpa.misc.state.

ClassWithCollections

class **ClassWithCollections** (*descr=None, **kwargs*)
>   Bases: `object`

>   Base class for objects which contain any known collection

>   Classes inherited from this class gain ability to access collections and their items as simple attributes. Access to collection items "internals" is done via <collection_name> attribute and interface of a corresponding *Collection*.

>   **descr**
>>   Description of the object if any

>   **reset** ()

See Also:

Derived classes might provide additional methods via their base classes. Please refer to the list of base classes (if it exists) at the begining of the `ClassWithCollections` documentation.

Full API documentation of ClassWithCollections in module mvpa.misc.state.

CollectableAttribute

class **CollectableAttribute** (*name=None, doc=None*)
>   Bases: `object`

>   Base class for any custom behaving attribute intended to become part of a collection.

>   Derived classes will have specific semantics:

>>   •StateVariable: conditional storage

>>   •AttributeWithUnique: easy access to a set of unique values within a container

>>   •Parameter: attribute with validity ranges.

>>>   –ClassifierParameter: specialization to become a part of Classifier's params collection
>>>   –KernelParameter: –//– to become a part of Kernel Classifier's kernel_params collection

>   Those CollectableAttributes are to be groupped into corresponding collections for each class by statecollector metaclass, ie it would be done on a class creation (ie not per each object)

>   **isSet**

>   **name**

>   **reset** ()
>>   Simply reset the flag

>   **value**

See Also:

Derived classes might provide additional methods via their base classes. Please refer to the list of base classes (if it exists) at the begining of the `CollectableAttribute` documentation.

Full API documentation of CollectableAttribute in module mvpa.misc.state.

Collection

class Collection(*items=None, owner=None, name=None*)

> Bases: `object`

> Container of some CollectableAttributes.

>> •*Public Access Functions*: *isKnown*
>> •*Access Implementors*: *_getListing*, *_getNames*
>> •*Mutators*: *__init__*
>> •*R/O Properties*: *listing*, *names*, *items*
>> XXX Seems to be not used and duplicating functionality: *_getListing* (thus *listing* property)

> Initialize the Collection

>> •*items* (dict of CollectableAttribute's) – items to initialize with
>> •*owner* (object) – an object to which collection belongs
>> •*name* (basestring) – name of the collection (as seen in the owner, e.g. 'states')

> **add**(*item*)
>> Add a new CollectableAttribute to the collection

>>> •*item* (CollectableAttribute) – or of derived class. Must have 'name' assigned

>> TODO: we should make it stricter to don't add smth of
>>> wrong type into Collection since it might lead to problems
>>> Also we might convert to __setitem__

> **get**(*index, default*)
>> Access the value by a given index.

>> Mimiquing regular dictionary behavior, if value cannot be obtained (i.e. if any exception is caught) return default value.

> **getvalue**(*index*)
>> Returns the value by index

> **isKnown**(*index*)
>> Returns *True* if state *index* is known at all

> **isSet**(*index=None*)
>> If item (or any in the present or listed) was set

>>> •*index* (None or basestring or list of basestring) – What items to check if they were set in the collection

> **items**

> **listing**
>> str(object) -> string

>> Return a nice string representation of the object. If the argument is a string, the return value is the same object.

> **name**

> **names**
>> Return ids for all registered state variables

> **owner**

> **remove**(*index*)
>> Remove item from the collection

> **reset**(*index=None*)
>> Reset the state variable defined by *index*

> **setvalue**(*index, value*)
>> Sets the value by index

> **whichSet**()
>> Return list of indexes which were set

See Also:

Derived classes might provide additional methods via their base classes. Please refer to the list of base classes (if it exists) at the begining of the `Collection` documentation.

Full API documentation of Collection in module mvpa.misc.state.

DatasetAttribute

class **DatasetAttribute**(*name=None, hasunique=True, doc='Attribute with unique'*)
> Bases: `mvpa.misc.state.AttributeWithUnique`

See Also:

Derived classes might provide additional methods via their base classes. Please refer to the list of base classes (if it exists) at the begining of the `DatasetAttribute` documentation.

Full API documentation of DatasetAttribute in module mvpa.misc.state.

FeatureAttribute

class **FeatureAttribute**(*name=None, hasunique=True, doc='Attribute with unique'*)
> Bases: `mvpa.misc.state.AttributeWithUnique`

See Also:

Derived classes might provide additional methods via their base classes. Please refer to the list of base classes (if it exists) at the begining of the `FeatureAttribute` documentation.

Full API documentation of FeatureAttribute in module mvpa.misc.state.

Harvestable

class **Harvestable**(*attribs=None, copy_attribs='copy', **kwargs*)
> Bases: `mvpa.misc.state.ClassWithCollections`

> Classes inherited from this class intend to collect attributes within internal processing.

> Subclassing Harvestable we gain ability to collect any internal data from the processing which is especially important if an object performs something in loop and discards some intermidiate possibly interesting results (like in case of CrossValidatedTransferError and states of the trained classifier or TransferError).

> Initialize state of harvestable

> > •*attribs* (list of basestr or dicts) – What attributes of call to store and return within harvested state variable. If an item is a dictionary, following keys are used ['name', 'copy']
> > •*copy_attribs* (None or basestr) – Default copying. If None – no copying, 'copy' - shallow copying, 'deepcopy' – deepcopying

> **harvest_attribs**

See Also:

Derived classes might provide additional methods via their base classes. Please refer to the list of base classes (if it exists) at the begining of the `Harvestable` documentation.

Full API documentation of Harvestable in module mvpa.misc.state.

ParameterCollection

class **ParameterCollection**(*items=None, owner=None, name=None*)
> Bases: `mvpa.misc.state.Collection`

Container of Parameters for a stateful object.

Initialize the Collection

> •*items* (dict of CollectableAttribute's) – items to initialize with
>
> •*owner* (object) – an object to which collection belongs
>
> •*name* (basestring) – name of the collection (as seen in the owner, e.g. 'states')

resetvalue(*index, missingok=False*)
> Reset all parameters to default values

See Also:

Derived classes might provide additional methods via their base classes. Please refer to the list of base classes (if it exists) at the begining of the `ParameterCollection` documentation.

Full API documentation of ParameterCollection in module mvpa.misc.state.

SampleAttribute

class **SampleAttribute**(*name=None, hasunique=True, doc='Attribute with unique'*)
> Bases: `mvpa.misc.state.AttributeWithUnique`

See Also:

Derived classes might provide additional methods via their base classes. Please refer to the list of base classes (if it exists) at the begining of the `SampleAttribute` documentation.

Full API documentation of SampleAttribute in module mvpa.misc.state.

SampleAttributesCollection

class **SampleAttributesCollection**(*items=None, owner=None, name=None*)
> Bases: `mvpa.misc.state.Collection`

> Container for data and attributes of samples (ie data/labels/chunks/...)

> Initialize the Collection

> > •*items* (dict of CollectableAttribute's) – items to initialize with
> >
> > •*owner* (object) – an object to which collection belongs
> >
> > •*name* (basestring) – name of the collection (as seen in the owner, e.g. 'states')

See Also:

Derived classes might provide additional methods via their base classes. Please refer to the list of base classes (if it exists) at the begining of the `SampleAttributesCollection` documentation.

Full API documentation of SampleAttributesCollection in module mvpa.misc.state.

StateCollection

class **StateCollection**(*items=None, owner=None*)
> Bases: `mvpa.misc.state.Collection`

> Container of StateVariables for a stateful object.

> > •*Public Access Functions*: *isKnown, isEnabled, isActive*
> >
> > •*Access Implementors*: *_getListing, _getNames, _getEnabled*
> >
> > •*Mutators*: *__init__, enable, disable, _setEnabled*
> >
> > •*R/O Properties*: *listing, names, items*
> >
> > •*R/W Properties*: *enabled*

Initialize the state variables of a derived class

> - *items* (dict) – dictionary of states
> - *owner* (Stateful) – object which owns the collection
> - *name* (basestring) – literal description. Usually just attribute name for the collection,
>   e.g. 'states'

disable(*index*)
> Disable state variable defined by *index* id

enable(*index, value=True, missingok=False*)
> Enable state variable given in *index*

enabled
> Return list of enabled states

> > - *nondefault* (bool) – Either to return also states which are enabled simply by default
> > - *invert* (bool) – Would invert the meaning, ie would return disabled states

isActive(*index*)
> Returns *True* if state *index* is known and is enabled

isEnabled(*index*)
> Returns *True* if state *index* is enabled

See Also:

Derived classes might provide additional methods via their base classes. Please refer to the list of base classes (if it exists) at the begining of the `StateCollection` documentation.

Full API documentation of StateCollection in module mvpa.misc.state.

StateVariable

class **StateVariable**(*name=None, enabled=True, doc='State variable'*)
> Bases: `mvpa.misc.state.CollectableAttribute`

> Simple container intended to conditionally store the value

> **enable**(*value=False*)

> **isEnabled**

> **reset**()
> > Simply detach the value, and reset the flag

See Also:

Derived classes might provide additional methods via their base classes. Please refer to the list of base classes (if it exists) at the begining of the `StateVariable` documentation.

Full API documentation of StateVariable in module mvpa.misc.state.

16.9.25 `mvpa.misc.stats`

The comprehensive API documentation for this module, including all technical details, is available in the Epydoc-generated API reference for mvpa.misc.stats (for developers).

See Also:

Full API documentation of chisquare() in module mvpa.misc.stats.

16.9.26 `mvpa.misc.support`

Support function – little helpers in everyday life

The comprehensive API documentation for this module, including all technical details, is available in the Epydoc-generated API reference for mvpa.misc.support (for developers).

Classes

Event

class **Event** (***kwargs*)

> Bases: `dict`
>
> Simple class to define properties of an event.
>
> The class is basically a dictionary. Any properties can be pass as keyword arguments to the constructor, e.g.:
>
> ```
> >>> ev = Event(onset=12, duration=2.45)
> ```
>
> Conventions for keys:
>
> *onset* The onset of the event in some unit.
>
> *duration*
> > The duration of the event in the same unit as *onset*.
>
> *label* E.g. the condition this event is part of.
>
> *chunk*
> > Group this event is part of (if any), e.g. experimental run.
>
> *features*
> > Any amount of additional features of the event. This might include things like physiological measures, stimulus intensity. Must be a mutable sequence (e.g. list), if present.
>
> **asDescreteTime** (*dt, storeoffset=False*)
> > Convert *onset* and *duration* information into descrete timepoints.
> >
> > > •*dt* (float) – Temporal distance between two timepoints in the same unit as *onset* and *duration*.
> > > •*storeoffset* (bool) – If True, the temporal offset between original *onset* and descretized *onset* is stored as an additional item in *features*.
> >
> > > Return
> > >
> > > A copy of the original *Event* with *onset* and optionally *duration* replaced by their corresponding descrete timepoint. The new onset will correspond to the timepoint just before or exactly at the original onset. The new duration will be the number of timepoints covering the event from the computed onset timepoint till the timepoint exactly at the end, or just after the event.
> > > Note again, that the new values are expressed as #timepoint and not in their original unit!

See Also:

Derived classes might provide additional methods via their base classes. Please refer to the list of base classes (if it exists) at the begining of the `Event` documentation.

Full API documentation of Event in module mvpa.misc.support.

Harvester

class **Harvester** (*source, calls, simplify_results=True*)

> Bases: `object`
>
> World domination helper: do whatever it is asked and accumulate results
>
> > •Might we need to deepcopy attributes values?
> > •Might we need to specify what attribs to copy and which just to bind?
>
> Initialize
>
> > •*source* – Generator which produce food for the calls.

•*calls* (sequence of HarvesterCall instances) – Calls which are processed in the loop. All calls are processed in order of apperance in the sequence.

•*simplify_results* (bool) – Remove unecessary overhead in results if possible (nested lists and dictionaries).

See Also:

Derived classes might provide additional methods via their base classes. Please refer to the list of base classes (if it exists) at the begining of the `Harvester` documentation.

Full API documentation of Harvester in module mvpa.misc.support.

HarvesterCall

class **HarvesterCall**(*call, attribs=None, argfilter=None, expand_args=True, copy_attribs=True*)

Bases: `object`

Initialize

•*expand_args* (bool) – Either to expand the output of looper into a list of arguments for call

•*attribs* (list of basestr) – What attributes of call to store and return later on?

•*copy_attribs* (bool) – Force copying values of attributes

See Also:

Derived classes might provide additional methods via their base classes. Please refer to the list of base classes (if it exists) at the begining of the `HarvesterCall` documentation.

Full API documentation of HarvesterCall in module mvpa.misc.support.

MapOverlap

class **MapOverlap**(*overlap_threshold=1.0*)

Bases: `object`

Compute some overlap stats from a sequence of binary maps.

When called with a sequence of binary maps (e.g. lists or arrays) the fraction of mask elements that are non-zero in a customizable proportion of the maps is returned. By default this threshold is set to 1.0, i.e. such an element has to be non-zero in *all* maps.

Three additional maps (same size as original) are computed:

•overlap_map: binary map which is non-zero for each overlapping element.

spread_map: binary map which is non-zero for each element that is non-zero in any map, but does not exceed the overlap threshold.

•   •ovstats_map: map of float with the raw elementwise fraction of overlap.

All maps are available via class members.

Nothing to be seen here.

See Also:

Derived classes might provide additional methods via their base classes. Please refer to the list of base classes (if it exists) at the begining of the `MapOverlap` documentation.

Full API documentation of MapOverlap in module mvpa.misc.support.

Functions

RFEHistory2maps (*history*)
> Convert history generated by RFE into the array of binary maps

> Example:
> > history2maps(N.array([3,2,1,0]))

> > array([[1., 1., 1., 1.],
> > > [1., 1., 1., 0.], [1., 1., 0., 0.], [1., 0., 0., 0.]])

See Also:

Full API documentation of RFEHistory2maps() in module mvpa.misc.support.

getBreakPoints (*items, contiguous=True*)
> Return a list of break points.

> > •*items* (iterable) – list of items, such as chunks
> > •*contiguous* (bool) – if *True* (default) then raise Value Error if items are not contiguous,
> > i.e. a label occur in multiple contiguous sets

> Raises
> > ValueError
> Returns
> > list of indexes for every new set of items

See Also:

Full API documentation of getBreakPoints() in module mvpa.misc.support.

getUniqueLengthNCombinations (*data, n*)
> Generates a list of lists containing all combinations of elements of data of length 'n' without repetitions.

> > data: list n: integer

> This function is adapted from a Java version posted in some forum on the web as an answer to the question 'How can I generate all possible combinations of length n?'. Unfortunately I cannot remember which forum it was.

See Also:

Full API documentation of getUniqueLengthNCombinations() in module mvpa.misc.support.

idhash (*val*)
> Craft unique id+hash for an object

See Also:

Full API documentation of idhash() in module mvpa.misc.support.

indentDoc (*v*)
> Given a *value* returns a string where each line is indented

> Needed for a cleaner __repr__ output *v* - arbitrary

See Also:

Full API documentation of indentDoc() in module mvpa.misc.support.

isInVolume (*coord, shape*)
> For given coord check if it is within a specified volume size.

> Returns True/False. Assumes that volume coordinates start at 0. No more generalization (arbitrary minimal coord) is done to save on performance

See Also:

Full API documentation of isInVolume() in module mvpa.misc.support.

isSorted (*items*)
> Check if listed items are in sorted order.

•*items* (iterable container) –

Returns
  True if were sorted. Otherwise *False* + Warning

See Also:

Full API documentation of isSorted() in module mvpa.misc.support.

reuseAbsolutePath (*file1, file2, force=False*)
 Use path to file1 as the path to file2 is no absolute path is given for file2

•*force* (bool) – if True, force it even if the file2 starts with /

See Also:

Full API documentation of reuseAbsolutePath() in module mvpa.misc.support.

transformWithBoxcar (*data, startpoints, boxlength, offset=0, fx=<function mean at 0x879909c>*)
 This function extracts boxcar windows from an array. Such a boxcar is defined by a starting point and the size of the window along the first axis of the array (*boxlength*). Afterwards a customizable function is applied to each boxcar individually (Default: averaging).

•*data* (array) – An array with an arbitrary number of dimensions.

•*startpoints* (sequence) – Boxcar startpoints as index along the first array axis

•*boxlength* (int) – Length of the boxcar window in #array elements

•*offset* (int) – Optional offset between the configured starting point and the actual beginning of the boxcar window.

Return type
  array (len(startpoints) x data.shape[1:])

See Also:

Full API documentation of transformWithBoxcar() in module mvpa.misc.support.

16.9.27 `mvpa.misc.transformers`

Simply functors that transform something.

The comprehensive API documentation for this module, including all technical details, is available in the Epydoc-generated API reference for mvpa.misc.transformers (for developers).

Class

OverAxis

class OverAxis (*transformer, axis=None*)
 Bases: `object`

 Helper to apply transformer over specific axis

 Initialize transformer wrapper with an axis.

•*transformer* – A callable to be used

•*axis* (None or int) – If None – apply transformer across all the data. If some int – over that axis

See Also:

Derived classes might provide additional methods via their base classes. Please refer to the list of base classes (if it exists) at the begining of the OverAxis documentation.

Full API documentation of OverAxis in module mvpa.misc.transformers.

Functions

Absolute(*x*)

>   Returns the elementwise absolute of any argument.

See Also:

Full API documentation of Absolute() in module mvpa.misc.transformers.

FirstAxisMean(*x*)

>   Mean computed along the first axis.

See Also:

Full API documentation of FirstAxisMean() in module mvpa.misc.transformers.

GrandMean(*x*)

>   Just what the name suggests.

See Also:

Full API documentation of GrandMean() in module mvpa.misc.transformers.

Identity(*x*)

>   Return whatever it was called with.

See Also:

Full API documentation of Identity() in module mvpa.misc.transformers.

L1Normed(*x, norm=1.0, reverse=False*)

>   Norm the values so that L_1 norm (sum|x|) becomes *norm*

See Also:

Full API documentation of L1Normed() in module mvpa.misc.transformers.

L2Normed(*x, norm=1.0, reverse=False*)

>   Norm the values so that regular vector norm becomes *norm*
>
>   More verbose: Norm that the sum of the squared elements of the returned vector becomes *norm*.

See Also:

Full API documentation of L2Normed() in module mvpa.misc.transformers.

OneMinus(*x*)

>   Returns elementwise '1 - x' of any argument.

See Also:

Full API documentation of OneMinus() in module mvpa.misc.transformers.

RankOrder(*x, reverse=False*)

>   Rank-order by value. Highest gets 0

See Also:

Full API documentation of RankOrder() in module mvpa.misc.transformers.

ReverseRankOrder(*x*)

>   Convinience functor

See Also:

Full API documentation of ReverseRankOrder() in module mvpa.misc.transformers.

SecondAxisMaxOfAbs(*x*)

>   Max of absolute values along the 2nd axis

See Also:

Full API documentation of SecondAxisMaxOfAbs() in module mvpa.misc.transformers.

SecondAxisMean(*x*)

Mean across 2nd axis

- to combine multiple sensitivities to get sense about mean sensitivity across splits

See Also:

Full API documentation of SecondAxisMean() in module mvpa.misc.transformers.

SecondAxisSumOfAbs(*x*)
    Sum of absolute values along the 2nd axis

- to combine multiple sensitivities to get sense about what features are most influential

See Also:

Full API documentation of SecondAxisSumOfAbs() in module mvpa.misc.transformers.

16.9.28 `mvpa.misc.vproperty`

C++-like virtual properties

The comprehensive API documentation for this module, including all technical details, is available in the Epydoc-generated API reference for mvpa.misc.vproperty (for developers).

VProperty

class **VProperty**(*fget=None, fset=None, fdel=None, doc=''*)
    Bases: `object`

    Provides "virtual" property: uses derived class's method

See Also:

Derived classes might provide additional methods via their base classes. Please refer to the list of base classes (if it exists) at the begining of the `VProperty` documentation.

Full API documentation of VProperty in module mvpa.misc.vproperty.

MODULE INDEX

INDEX

N