## CONTENTS

1 Introduction
   1.1 What this Manual is NOT .......................................................... 3
   1.2 A bit of History . . ................................................................. 3
   1.3 Authors & Contributors ......................................................... 4
   1.4 How to cite PyMVPA ............................................................... 5
   1.5 Acknowledgements ............................................................... 5

2 Installation
   2.1 Dependencies . . ................................................................. 7
   2.2 Installing Binary Packages .................................................. 9
   2.3 Building from Source .......................................................... 12

3 Getting Started
   3.1 For the Impatient . . .......................................................... 17
   3.2 Module Overview ............................................................... 18

4 Datasets
   4.1 The Basic Concepts ........................................................... 19
   4.2 Data Mapping ................................................................. 20
   4.3 Data Access Sugaring ......................................................... 21
   4.4 Data Formats ................................................................. 21
   4.5 Data Splitting ................................................................. 22

5 Classifiers
   5.1 Stateful objects ............................................................... 26
   5.2 Error Calculation ............................................................ 27
   5.3 Error Reporting ............................................................. 28
   5.4 Basic Supervised Learning Methods ..................................... 30
   5.5 Meta-Classifiers ............................................................. 31
   5.6 Retrainable Classifiers .................................................... 33
   5.7 Classifiers “Warehouse” ................................................... 33

6 Measures
   6.1 Sensitivity Measures ........................................................ 35

7 Feature Selection
   7.1 Recursive Feature Elimination ............................................ 40
   7.2 Incremental Feature Search ................................................ 41

8 Miscellaneous
   8.1 Managing (Custom) Configurations ....................................... 43
   8.2 Progress Tracking ............................................................ 45
   8.3 Additional Little Helpers ................................................... 48
   8.4 FSL Bindings ................................................................. 49
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. Two very good examples are: Pattern Recognition and Machine Learning by Christopher M. Bishop, and The Elements of Statistical Learning: Data Mining, Inference, and Prediction by Trevor Hastie, Robert Tibshirani, and Jerome Friedman (PDF was generously made available online free of charge).

There is a growing number of introductory papers about the application of machine learning algorithms to (f)MRI data. A very high-level overview about the basic principles is available in Mur et al. (2009). A more detailed tutorial covering a wide variety of aspects is provided in Pereira et al. (in press). Two reviews by Norman et al. (2006) and Haynes and Rees (2006) give a broad overview about the literature.

This manual also does not describe every technical bit and piece of the PyMVPA package, but is instead focused on the user perspective. Developers should 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-Neighbor 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, Dartmouth College, USA
- Yaroslav O. Halchenko, Dartmouth College, USA
- Per B. Sederberg, Princeton University, USA
- Emanuele Olivetti, Fondazione Bruno Kessler, Italy

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

- Greg Detre, Princeton University, USA
- Ingo Fründ, TU Berlin, Germany
- Scott Gorlin, MIT, USA
- Valentin Haenel, TU Berlin, Germany
- James M. Hughes, Dartmouth College, USA
- James Kyle, UCLA, USA
- Tiziano Zito, BCCN, 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. In addition there is also a list of studies done by other groups employing PyMVPA somewhere in the analysis.

1.4.1 Peer-reviewed publications


1.4.2 Posters


1.4.3 Studies employing PyMVPA

- Manelis et al. (2010): Implicit memory for object locations depends on reactivation of encoding-related brain regions

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.
CHAPTER TWO

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 wrappers around LARS, ElasticNet, and GLMNet R libraries available from CRAN. On Debian-based machines you might like to install r-cran-* packages from cran2deb repository.

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.

**Note:** If you have difficulties deploying PyMVPA itself or third-party modules, such as Shogun, on non Debian-based systems, we would advise you to give a try to NeuroDebian virtual machine which would allow you to benefit from Debian packaging of PyMVPA and its dependencies by running Debian in a virtualized environment.

### 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 unofficial Ubuntu packages

Backports for the current Debian stable release and binary packages for recent Ubuntu releases are available from a Debian Neuroscience Repository. 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. You 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 prompt 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.

```python
>>> 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` (or `system` on 64-bit Windows 7) directory. Verify that it works by importing the `nifti` module in Python.

```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

The easiest installation method for OSX is via MacPorts. MacPorts is a package management system for MacOS, which is in some respects very similar to RPM or APT which are used in most GNU/Linux distributions. However, rather than installing binary packages, it compiles software from source on the target machine.

The MacPort of PyMVPA is kindly maintained by James Kyle <jameskyle@ucla.edu>.

**Note:** MacPorts needs XCode developer tools to be installed first, as the operating system does not come with a compiler by default.

In the context of PyMVPA MacPorts is much easier to handle than the previously available PyMVPA installer for Macs (which was discontinued with PyMVPA 0.4.1). Although the initial overhead to setup MacPorts on a machine is higher than simply installing PyMVPA using the former installer, MacPorts saves the user a significant amount of time (in the long run). This is due to the fact that this framework will not only take care of updating a PyMVPA installation automatically whenever a new release is available. It will also provide many of the optional dependencies of PyMVPA (e.g. NumPy, SciPy, matplotlib, IPython, Shogun, and pywt) in the same environment and therefore abolishes the need to manually check dozens of websites for updates and deal with an unbelievable number of different installation methods.

MacPorts provides a universal binary package installer that is downloadable at http://www.macports.org/install.php
After downloading, simply mount the dmg image and double click MacPorts.pkg.

By default, MacPorts installs to /opt/local. After the installation is completed, you must ensure that your paths are set up correctly in order to access the programs and utilities installed by MacPorts. For exhaustive details on editing shell paths please see:


A typical .bash_profile set up for MacPorts might look like:

```bash
> export PATH=/opt/local/bin:/opt/local/sbin:$PATH
```

Be sure to source your .bash_profile or close Terminal.app and reopen it for these changes to take effect.

Once MacPorts is installed and your environment is properly configured, PyMVPA is installed using a single command:

```bash
> $ sudo port install py25-pymvpa +scipy +pynifti +hcluster +libsvm +matplotlib +pywavelet
```

The +foo arguments add support within PyMVPA for these packages. For a full list of available 3rd party packages please see:

```bash
> $ port variants py25-pymvpa
```

If this is your first time using MacPorts Python 2.5 will be automatically installed for you. However, an additional step is needed:

```bash
$ sudo port install python_select
$ sudo python_select python25
```

MacPorts has the ability of installing several Python versions at a time, the python_select utility ensures that the default Python (located at /opt/local/bin/python) points to your preferred version.

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

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

If no error messages appear, you have successfully 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 9-12, RedHat Enterprise Linux 5, OpenSUSE 11.0 up to 11.2.

Once downloaded, open a console and invoke (the example command refers to PyMVPA 0.4.4):

```bash
rpm -i python-mvpa-0.4.4-1.1.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:/suse/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. There are two different repository groups, one for SUSE-related packages and another one for Fedora, Redhat and CentOS-related packages.

Please note that on Redhat and CentOS systems you will also have to enable the Extra Packages for Enterprise Linux (EPEL) repository.

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 GNU/Linux distribution should be capable of doing it (e.g. RedHat). Additionally, building PyMVPA also works on Mac OS X 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://github.com/PyMVPA/PyMVPA

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://github.com/PyMVPA/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 Microsoft 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:

```bash
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:

```bash
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

Since the MacPorts system basically compiles from source there should be no need to perform this step manually. However, if one intends to compile without MacPorts the XCode developer tools, have to be installed first, 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:

```python
>>> 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.

```python
>>> attr = SampleAttributes(os.path.join(pymvpa_dataroot, 'attributes.txt'))
>>> dataset = NiftiDataset(...
...     samples=os.path.join(pymvpa_dataroot, 'bold.nii.gz'),
...     labels=attr.labels,
...     chunks=attr.chunks,
...     mask=os.path.join(pymvpa_dataroot, '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).

```python
>>> 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.

```python
>>> dataset = dataset['labels', [1,2]]
```

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

```python
>>> 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 toplevel directory of the extracted source tarball, e.g.:
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.
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 samples, 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 chunks, 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.

```python
>>> 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:

```python
>>> 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.

```python
>>> 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 neighborhood 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.

```python
>>> 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.

```python
>>> 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 overlaid over some anatomical image. This is trivial with PyMVPA, because the mapping is automatically updated upon feature selection.

```python
>>> mdata.mapReverse(N.arange(1,mdata.nfeatures+1))
array([[ 1,  2,  3,  4],
       [ 5,  6,  7,  8],
       [ 9, 10, 11, 12]])
```

```python
>>> sdata = mdata.selectFeatures([2,7,9,10])
>>> sdata
<Dataset / float64 5 x 4 uniq: 5 chunks 5 labels>
```

```python
>>> sdata.mapReverse(N.arange(1,sdata.nfeatures+1))
array([[0, 0, 1, 0],
       [0, 0, 0, 2],
       [1, 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. Resultant dataset contains a copy of the corresponding features of the original dataset. 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

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Abbreviation</th>
<th>Definition class</th>
</tr>
</thead>
<tbody>
<tr>
<td>samples</td>
<td>S</td>
<td>Dataset</td>
</tr>
<tr>
<td>labels</td>
<td>L</td>
<td>Dataset</td>
</tr>
<tr>
<td>uniquelabels</td>
<td>UL</td>
<td>Dataset</td>
</tr>
<tr>
<td>chunks</td>
<td>C</td>
<td>Dataset</td>
</tr>
<tr>
<td>uniquechunks</td>
<td>UC</td>
<td>Dataset</td>
</tr>
<tr>
<td>origids</td>
<td>I</td>
<td>Dataset</td>
</tr>
<tr>
<td>samples_original</td>
<td>O</td>
<td>MappedDataset</td>
</tr>
</tbody>
</table>

### 4.4 Data Formats

The concept of mappers in conjunction 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`).

---

## 4.3 Data Access Sugaring

By passing their ids to the `selectFeatures()` method. The method returns a new dataset only containing the four selected features. Resultant dataset contains a copy of the corresponding features of the original dataset. 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.

```python
>>> mdata.mapReverse(N.arange(1,mdata.nfeatures+1))
array([[ 1,  2,  3,  4],
       [ 5,  6,  7,  8],
       [ 9, 10, 11, 12]])
```

```python
>>> sdata = mdata.selectFeatures([2,7,9,10])
>>> sdata
<Dataset / float64 5 x 4 uniq: 5 chunks 5 labels>
```

```python
>>> sdata.mapReverse(N.arange(1,sdata.nfeatures+1))
array([[0, 0, 1, 0],
       [0, 0, 0, 2],
       [1, 3, 4, 0]])
```

---

## 4.4 Data Formats

The concept of mappers in conjunction 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:

```python
>>> from mvpa.datasets.splitters 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:

```python
>>> 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.

Behavior of the splitters can be heavily customized by additional arguments to the constructor (see `Splitter` for extended help on the arguments). For instance, in the analysis in fMRI data it might be important to assure that samples in the training and testing parts of the split are not neighboring samples (unless it is otherwise assured by the presence of baseline condition on the boundaries between chunks, samples of which are discarded
prior the statistical learning analysis). Providing argument `discard_boundary=1` to the splitter, would remove from both training and testing parts a single sample, which lie on the boundary between chunks. Providing `discard_boundary=(2,0)` would remove 2 samples only from training part of the split (which is desired strategy for `NFoldSplitter` where training part contains majority of the data).

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.
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.

```python
>>> 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 figures 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,
```python
>>> 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:

```python
>>> 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

```python
>>> 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.

```python
>>> 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:

```python
>>> 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. Optionnally 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 re-
turned. `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.

#### 5.2. Error Calculation
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.

```python
>>> 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.

```python
>>> # 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.splitters 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.aasrt(description=True) \...
-------- 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 NPV TPR SPC FDR MCC
3kHz / 38 84 42 27 4 4 2 1 0 15 19 198 1351 114 90 0.42 0.93 0.48 0.92 0.58 0.36
7kHz / 39 43 94 16 0 1 1 2 1 24 183 1331 89 80 0.51 0.94 0.54 0.93 0.49 0.43
12kHz / 40 21 16 103 5 2 2 0 0 6 13 168 1312 65 70 0.61 0.95 0.6 0.95 0.39 0.51
20kHz / 41 1 2 13 158 1 0 0 1 3 180 1202 22 15 0.88 0.99 0.91 0.98 0.12 0.77
30kHz / 42 3 0 2 162 0 0 0 0 0 170 1194 8 11 0.95 0.99 0.94 0.99 0.05 0.82
song1 / 43 3 1 1 0 1 160 0 0 2 5 173 1199 13 14 0.92 ...
song2 / 44 1 1 0 0 0 171 0 0 0 0 173 1176 2 2 0.99 ...
song3 / 45 1 1 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 ...
```
N 1560 1560 1561 1561 1561 1560 1561 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 0.97 0.78 0.97 0.22 0.66
ACC 0.78
ACC% 77.57
# of sets 8

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)
  \[ TPR = \frac{TP}{TP + FN} \]
- **FPR**: false positive rate (AKA false alarm rate, fall-out)
  \[ FPR = \frac{FP}{FP + TN} \]
- **ACC**: accuracy
  \[ ACC = \frac{(TP + TN)}{(P + N)} \]
- **SPC**: specificity
  \[ SPC = \frac{TN}{(FP + TN)} = 1 - FPR \]
- **PPV**: positive predictive value (AKA precision)
  \[ PPV = \frac{TP}{TP + FP} \]
- **NPV**: negative predictive value
  \[ NPV = \frac{TN}{(TN + FN)} \]
- **FDR**: false discovery rate
  \[ FDR = \frac{FP}{(FP + TP)} \]
- **MCC**: Matthews Correlation Coefficient
  \[ MCC = \frac{(TP*TN - FP*FN)}{\sqrt{P N P' N'}} \]

# 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`:

```python
>>> import pylab as P
>>> cvterr.confusion.plot() \...    \...
>>> P.show() \...    \...
```
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 Euclidean 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:

```python
>>> from mvpa.clfs.meta 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(),
    feature_selection = RFE(
        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. \texttt{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 \texttt{retrainable} in their \_clf\_internals are capable of efficient retraining. To enable retraining, just provide \texttt{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 \texttt{retrain()} and \texttt{repredict()} methods could be called. \texttt{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 \texttt{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 \texttt{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 \texttt{!} being excluded:

```python
>>> from mvpa.clfs.warehouse import clfswh
>>> tryme = clfswh[‘multiclass’, ‘!svm’]
```

to simply sweep through classifiers which are capable of multiclass classification and are not SVM based.
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 objects. 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 behavior 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 `mvpa.clfs.libsvm.sens.LinearSVMWeights` and `mvpa.clfs.sg.sens.LinearSVMWeights` classes 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` argument in the constructor. 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).

```python
>>> 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 sensitivity 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 sensitivity 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 original 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.
CHAPTER
SEVEN

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):

```python
>>> 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:

```python
>>> 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)
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

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

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)
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:

```python
>>> 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:

```ini
[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:

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

`MVPA_VERBOSE_OUTPUT=stdout` becomes:

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

Any length of variable name is allowed, e.g. `MVPA_SEC1_LONG_VARIABLE_NAME=1` becomes:

```ini
[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:

```python
>>> 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 no 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:

```plaintext
# 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 =
# either to use custom (improved) exception handler to report
# information about pymvpa useful during bug reporting
#wtf = no

[examples]
interactive = yes

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

[matplotlib]
# override the default matplotlib's backend
# backend = pdf

[rpy]
# to prevent stalled execution of PyMVPA upon problems in R
# session of R is always responding '1' whenever R asks for input.
# 1 corresponds to "abort (with core dump, if enabled)".
# Unfortunately such callback does not work reliably, thus disabled
# by default
interactive = yes

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

# whether to issue warning when an externals test fails _and_
# issuing a warning was requested
issue warning = True

# whether to retest the availability of an external dependency, despite 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

# verbosity level of the unittest runner
verbosity = 1

# scale SNR of simulated data more than 1 to reduce failures of labile tests
snr scale = 1.0

[doc]
# whether to enhance the docstrings with base class and state information
pimp docstrings = yes

8.2 Progress Tracking

There are 3 types of messages PyMVPA can produce:

verbatim
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_X_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/2
```

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
```

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
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` - controls 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` - 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 backtraces 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 base 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

8.2. Progress Tracking
Traceback (module1:line_number1[,line_number2...]>module2:line_number..)
where this debug statement was requested

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.

**8.2.5 PyMVPA Status Summary**

While reporting found bugs, it is advised to provide information about the operating system/environment and availability of PyMVPA externals. Please use wtf() to collect such useful information to be included with the bug reports.

Alternatively, same printout can be obtained upon not handled exception automagically, if environment variable MVPA_DEBUG_WTF is set.

**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 unitests 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 text files, 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('mvpa', 'data', 'bold_mc.par'))

# simple plot using pylab (use pylab.show() or pylab.savefig())
# afterwards)
>>> mc.plot()

# detrend some dataset with mc params as additional regressors
>>> from mvpa.datasets.miscfx import detrend
>>> res = detrend(ds, model='regress', opt_reg=mc.toarray())

All FSL bindings are located in the mvpa.misc.fsl module.
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) < OJA+05).

Note: The original 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 voxel size 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 text file with the stimulation condition and the corresponding experimental run for each volume in the timeseries image. The labels are given in literal form (e.g. ‘face’).

attributes.txt
Similar to 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:

```python
>>> from mvpa.misc.io.base import SampleAttributes
>>> from mvpa.datasets.nifti import NiftiDataset

>>> 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, that instead of specific import statements, it is usually more convinient, but slower, to import all functionality from PyMVPA at once with `from mvpa.suite import *` statement.

---

**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 Visualization of Data Projection Methods

```python
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)
mappers = 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).

9.2.2 Simple Data-Exploration

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

```python
import numpy as N
import pylab as P
import os

from mvpa import pymvpa_dataroot
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(os.path.join(pymvpa_dataroot, 'attributes.txt'))
```
ds = NiftiDataset(samples=os.path.join(pymvpa_dataroot, 'bold.nii.gz'),
                  labels=attr.labels,
                  chunks=attr.chunks,
                  mask=os.path.join(pymvpa_dataroot, '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 similarity
# 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
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.

```python
# get PyMVPA running
from mvpa.suite import *

# load PyMVPA example dataset
attr = SampleAttributes(os.path.join(pymvpa_dataroot, 'attributes.txt'))
dataset = NiftiDataset(samples=os.path.join(pymvpa_dataroot, 'bold.nii.gz'),
                        labels=attr.labels,
                        chunks=attr.chunks,
                        mask=os.path.join(pymvpa_dataroot, 'mask.nii.gz'))

# do chunkwise 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'))
```

See Also:
The full source code of this example is included in the PyMVPA source distribution (`doc/examples/smellit.py`).
# 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 classifiers can be combined.

```python
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 patterns 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: %g%% (Retained %d/%d features)" % 
    (smlr_confusion.percentCorrect,
    (smlr.weights!=0).sum(), N.prod(smlr.weights.shape))
print "linear-SVM Percent Correct: %g%%" % 
    (lsvm_confusion.percentCorrect)
print "linear-SVM Percent Correct (with %d features from SMLR): %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.

```python
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
    attrs = SampleAttributes(os.path.join(pymvpa_dataroot,
        'attributes_literal.txt'))
    haxby8 = NiftiDataset(samples=os.path.join(pymvpa_dataroot,
        'bold.nii.gz'),
        labels=attrs.labels,
        labels_map=True,
        chunks=attrs.chunks,
        mask=os.path.join(pymvpa_dataroot, 'mask.nii.gz'),
        dtype=N.float32)

    # preprocess slightly
    rest_label = haxby8.labels_map['rest']
detrend(haxby8, perchunk=True, model='linear')
zscore(haxby8, perchunk=True, baseline_labels=[rest_label],
    target_dtype='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 in [(dummy2, "Dummy 2-class univariate with 2 useful features out of 100"),
    clfswh[:]),
    (pureMultivariateSignal(8, 3),
    "Dummy XOR-pattern"),
    clfswh['non-linear']),
    (haxby8_no0,
    "Haxby 8-cat subject 1"),
    clfswh['multiclass']),
    ]:
    print "%s
%s" % (datasetdescr, dataset.summary(idhash=False))
    print " Classifier "
        "%corr %features\t train predict full"
for clf in clfs_in clfs_in:
    print "%-40s: " % clf.descr,
    # Lets do splits/train/predict explicitely 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 "$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 The effect of different hyperparameters in GPR

The following example runs Gaussian Process Regression (GPR) on a simple 1D dataset using squared exponential (i.e., Gaussian or RBF) kernel and different hyperparameters. The resulting classifier solutions are finally visualized in a single figure.

As usual we start by importing all of PyMVPA:

```
# Lets use LaTex for proper rendering of greek
from matplotlib import rc
rc('text', usetex=True)

from mvpa.suite import *
```

The next lines build two datasets using one of PyMVPA's data generators.

```
# Generate dataset for training:
train_size = 40
F = 1
dataset = data_generators.sinModulated(train_size, F)

# Generate dataset for testing:
test_size = 100
dataset_test = data_generators.sinModulated(test_size, F, flat=True)
```

The last configuration step is the definition of four sets of hyperparameters to be used for GPR.

```
# Hyperparameters. Each row is [sigma_f, length_scale, sigma_noise]
hyperparameters = N.array([[1.0, 0.2, 0.4], [1.0, 0.1, 0.1], [1.0, 1.0, 0.1], [1.0, 0.1, 1.0]])
```

The plotting of the final figure and the actually GPR runs are performed in a single loop.
rows = 2
columns = 2
P.figure(figsize=(12, 12))
for i in range(rows*columns):
    P.subplot(rows, columns, i+1)
    regression = True
    logml = True
    data_train = dataset.samples
    label_train = dataset.labels
    data_test = dataset_test.samples
    label_test = dataset_test.labels

    The next lines configure a squared exponential kernel with the set of hyperparameters for the current subplot and
    assign the kernel to the GPR instance.
    sigma_f, length_scale, sigma_noise = hyperparameters[i, :]
kse = KernelSquaredExponential(length_scale=length_scale, sigma_f=sigma_f)
g = GPR(kse, sigma_noise=sigma_noise, regression=regression)
print g
if regression:
g.states.enable("predicted_variances")
if logml:
g.states.enable("log_marginal_likelihood")

After training GPR the predictions are queried by passing the test dataset samples and accuracy measures are
computed.
g.train(dataset)
prediction = g.predict(data_test)

# print label_test
# print prediction
accuracy = None
if regression:
    accuracy = N.sqrt(((prediction-label_test)**2).sum()/prediction.size)
    print "RMSE: ", accuracy
else:
    accuracy = (prediction.astype('l')==label_test.astype('l')).sum() \
                / float(prediction.size)
    print "accuracy: ", accuracy

The remaining code simply plots both training and test datasets, as well as the GPR solutions.
if F == 1:
P.title(r"\$\sigma_f=%0.2f$, \$\sigma_n=%0.2f$" \
% (sigma_f,length_scale,sigma_noise))
P.plot(data_train, label_train, "ro", label="train")
P.plot(data_test, prediction, "b-", label="prediction")
P.plot(data_test, label_test, "g+", label="test")
if regression:
P.plot(data_test, prediction-N.sqrt(g.predicted_variances), "b--", label=None)
P.plot(data_test, prediction+N.sqrt(g.predicted_variances), "b--", label=None)
P.text(0.5, -0.8, "$\text{RMSE}=%.3f$" % (accuracy))
P.text(0.5, -0.95, "$\text{LML}=%.3f$" % (g.log_marginal_likelihood))
else:
P.text(0.5, -0.8, "$\text{accuracy}=\%s\ accuracy")
P.legend(loc='lower right')
print "LML:", g.log_marginal_likelihood

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/gpr.py).

9.3.5 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 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.splitters 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.dataGenerators 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:', min(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.

.. seealso::

    The full source code of this example is included in the PyMVPA source distribution (doc/examples/searchlight_minimal.py).
9.3.6 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 resources, 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(os.path.join(pymvpa_dataroot, 'attributes.txt'))
dataset = NiftiDataset(samples=os.path.join(pymvpa_dataroot, 'bold.nii.gz'),
    labels=attr.labels,
    chunks=attr.chunks,
    mask=os.path.join(pymvpa_dataroot, '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 N-fold 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.

```python
# setup plotting
fig = 0
plt.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
    plt.subplot(1,3,fig)
    plt.title('Radius %i' % radius)
    plt.imshow(masked_orig_sl_map[0], interpolation='nearest', aspect=1.25, cmap=plt.cm.autumn)
    plt.clim(0.5, 0.65)
    plt.colorbar(shrink=0.6)

if cfg.getboolean('examples', 'interactive', True):
    # show all the cool figures
    plt.show()
```

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

### 9.3.7 A searchlight computing a dissimilarity matrix measure

This example extends the minimal Searchlight example to use a dissimilarity matrix-based DatasetMetric to compute Searchlight-center significance. This is based on representational similarity analysis (RSA) as presented in Kriegeskorte et al. (2008).

First import all necessary parts of PyMVPA.

```python
from mvpa.suite import *
```

Create a small artificial dataset.
Create a dissimilarity matrix based on the labels of the data points in our test dataset. This will allow us to see if there is a correlation between any given searchlight sphere and the experimental conditions.

```
# create dissimilarity matrix using the 'confusion' distance
# metric
dsm = DSMatrix(dataset.labels, 'confusion')
```

Now it only takes three lines for a searchlight analysis.

```
# setup measure to be computed in each sphere (correlation
# distance between dissimilarity matrix and the dissimilarities
# of a particular searchlight sphere across experimental
# conditions), N.B. in this example between-condition
# dissimilarity is also pearson’s r (i.e., correlation distance)
dsmetric = DSMDatasetMeasure(dsm, 'pearson', 'pearson')
```

```
# setup searchlight with 5 mm radius and measure configured above
sl = Searchlight(dsmetric, 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_dsm.py).

### 9.3.8 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(os.path.join(pymvpa_dataroot, 'attributes.txt'))
dataset = NiftiDataset(samples=os.path.join(pymvpa_dataroot, 'bold.nii.gz'),
labels=attr.labels,
chunks=attr.chunks,
mask=os.path.join(pymvpa_dataroot, '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 = {
```
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.

```python
# do chunkwise 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.

```python
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 sensitivities
    # 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, 30)
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:

![Output of the example analysis](image)

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

### 9.3.9 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 classifier.

```python
from mvpa.suite import *
if __debug__:
```
debug.active += ["CROSSC"]

# load PyMVPA example dataset
attr = SampleAttributes(os.path.join(pymvpa_dataroot, 'attributes.txt'))
dataset = NiftiDataset(samples=os.path.join(pymvpa_dataroot, 'bold.nii.gz'),
                      labels=attr.labels,
                      chunks=attr.chunks,
                      mask=os.path.join(pymvpa_dataroot, '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

clf = 
    [('All orig.\n features (%)' % dataset.nfeatures, Clf()),
     ('All Comps\n(%i)\n' % (dataset.nsamples - (dataset.nsamples / len(dataset.uniquechunks))),
      MappedClassifier(Clf(), SVDMapper()))

# 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:

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

9.3.10 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 empirical (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 MCNullDist, 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.

```python
# lazy import
from mvpa.suite import *

# enable progress output for MC estimation
if __debug__:
    debug.active += ['STATMC']

# some example data with signal
train = normalFeatureDataset(perlabel=50, nlabels=2, nfeatures=3,
                              nonbogus_features=[0,1], snr=0.3, 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 larger
# to get stable estimates
terr = TransferError(clf=SMLR(),
                      null_dist=MCNullDist(permutations=100,
                                           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).

9.3.11 Determine the Distribution of some Variable

This is an example demonstrating discovery of the distribution facility.

```python
from mvpa.suite import *
```
verbose.level = 2
if __debug__:
    # report useful debug information for the example
    debug.active += ['STAT', 'STAT_']

report = Report(name='match_distribution_report',
                 title='PyMVPA Example: match_distribution.py')
verbose.handlers += [report]  # Lets add verbose output to the report.
    # Similar action could be done to 'debug'

# 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')

# we are done with the figure -- add it to report
report.figure()

# Figure for fMRI data sample we have
#
verbose(1, "Load sample fMRI dataset")
attr = SampleAttributes(os.path.join(pymvpa_dataroot, 'attributes.txt'))
dataset = NiftiDataset(samples=os.path.join(pymvpa_dataroot, 'bold.nii.gz'),
                       labels=attr.labels,
                       chunks=attr.chunks,
                       mask=os.path.join(pymvpa_dataroot, 'mask.nii.gz'))

# select random voxel
dataset = dataset.selectFeatures(
    [int(N.random.uniform(0)*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)
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).

9.3.12 Spatio-temporal Analysis of event-related fMRI data

The purpose of this example is to show how to use spatio-temporal samples in an event-related fMRI data analysis. We start as usual by loading the PyMVPA suite. The tiny fMRI dataset, included in the sources will server as an example dataset. Although the original paradigm of this dataset is a block-design experiment, we’ll analyze it in an event-related fashion, where each block will be considered as an individual event.

from mvpa.suite import *

# filename of the source fMRI timeseries image
fmri_src = os.path.join(pymvpa_dataroot, 'bold.nii.gz')
mask = NiftiImage(os.path.join(pymvpa_dataroot, 'mask.nii.gz'))

# load the samples attributes as usual and preserve the literal labels
attr = SampleAttributes(
    os.path.join(pymvpa_dataroot, 'attributes_literal.txt'),
    literallabels=True)

For an event-related analysis most of the processing is done on data samples that are somehow derived from a set of events. The rest of the data could be considered irrelevant. However, some e.g. preprocessing is only meaningful when performed on the full timeseries and not the segmented event samples. An example is the detrending that typically needs to be done on the original, continuous timeseries. Therefore we are going to load the data twice: first as a simple volume-based dataset for an initial preprocessing, and second to extract the events of interest.

verbose(1, "Load data for preprocessing")
pre_ds = NiftiImage(fmri_src)

# actual labels are not important here, could be 'labels=1'
pre_ds = NiftiDataset(samples=fmri_src, labels=attr.labels,
    chunks=attr.chunks, mask=mask)

# convert to floats
pre_ds.setSamplesDType('float')

# detrend on full timeseries
detrend(pre_ds, perchunk=True, model='linear')

After the detrending, we can now segment the timeseries into a set of events. To achieve this we have to compile a list of event definitions first. In this example we will simply convert the block-design setup defined by the samples attributes into events, so that each block become an event with an associated onset and duration. The necessary chunk settings are taken from these attributes as well. Finally, we are only interested in face or house blocks.

evs = [ev for ev in attr.toEvents()
    if ev['label'] in ['face', 'house']]

Since we might want to take a look at the sensitivity profile ranging from just before until a little after each block, we are slightly moving the event onsets (one volume prior the actual event) and request to extract a set of twelve consecutive volume a as sample for each event.

    for ev in evs:
        ev['onset'] -= 1
        ev['duration'] = 12

A ERNiftiDataset can now be used to segment the timeseries and automatically extract boxcar-shaped multi-volume samples. It is also capable of applying a volume mask.

    # could use evconv...
    verbose(1, "Segmenting timeseries into events")
    ds = ERNiftiDataset(samples=pre_ds.map2Nifti(),
        events=evs,
        mask=mask,
        labels_map={'face': 1,
            'house': 2})

For demonstration purposes we copy the pristine dataset before any further processing is done.

    # preserve
    orig_ds = deepcopy(ds)

The rest is pretty much standard. A dataset with spatio-temporal fMRI samples behaves just as any other dataset type. We perform normalization by Z-scoring the data and settle on a linear SVM classifier to perform a cross-validated sensitivity analysis.
# using rest as baseline
zscore(ds, perchunk=True)

clf = LinearCSVMC()
sclf = SplitClassifier(clf, NFoldSplitter(),
    enable_states=['confusion', 'training_confusion'])

# Compute sensitivity, which in turn trains the sclf
sensitivities = \
    sclf.getSensitivityAnalyzer(combiner=None,
    slave_combiner=None)(ds)

Before looking at the sensitivity profile we first have to inspect the classifier performance in the cross-validation,
since only for a model with reasonable generalization performance it would make sense to interpret the model
parameters, i.e. classifier weights. If this is done we could dump the spatio-temporal sensitivity profile, which
covers all voxels in the dataset for the full duration of the events, into a NIfTI file.

print sclf.confusion

#ds.map2Nifti(N.mean(sensitivities, axis=0)).save('fs_sens.nii.gz')

However, we are going to plot it for some target voxel right away, and compare it to the actual signal timecourse
prior and after normalization. We can use the dataset’s mapper to convert the sensitivity vector for each CV-fold
back into a 4D snippet.

# reverse map sensitivities -> fold x volumes x Z x Y x X
smaps = N.array([ds.mapReverse(s) for s in sensitivities])

# extract sensitivity profile for target voxel ijk(33,10,0)
v = (0, 3, 15)
smap = smaps[:,:,v[0],v[1],v[2]]

Now, we plot the orginal signal after initial detrending.

P.subplot(311)
P.title('Voxel zyx%s\nblock-onset@1, block-offset@8' % 'v')
for l in ds.uniquelabels:
P.plot(
    ds.mapReverse(
        orig_ds.samples[ds.labels==l].mean(axis=0)
    )[:,:,v[0],v[1],v[2]])
P.ylabel('Signal after detrending')
P.axhline(linestyle='--', color='0.6')

the peristimulus timecourse after Z-scoring.

P.subplot(312)
for l in ds.uniquelabels:
P.plot(
    ds.mapReverse(
        ds.samples[ds.labels==l].mean(axis=0)
    )[:,:,v[0],v[1],v[2]])
P.ylabel('Signal after normalization')
P.axhline(linestyle='--', color='0.6')

and finally the associated SVM weight profile for each peristimulus timepoint of the voxel.

P.subplot(313)
plotErrLine(smap)
P.ylabel('Sensitivity')
P.xlabel('Peristimulus volumes')
P.axhline(linestyle='--', color='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/eventrelated.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(pymvpa_dataroot, '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',
    'alilinewidth': 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)
```
The output 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.

```
    # Plot EEG sensor
    ax = fig.add_subplot(2, 1, 2, frame_on=False)
    plotERPs(plots[2:3], ax=ax, ymult=-1e6, **cargs)
```

```
# Plot EEG sensor
ax = fig.add_subplot(2, 1, 2, frame_on=False)
plotERPs(plots[2:3], ax=ax, ymult=-1e6, **cargs)
```

The output 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.

### 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
# Absolute max value allowed. Just to assure proper plots
xyamax = 10
feat_pos = N.random.randn(2, num_dat)
feat_pos[0, :] *= 2.
feat_pos[1, :] *= .5
feat_pos[0, :] += dist
feat_pos = feat_pos.clip(-xyamax, xyamax)
feat_neg = N.random.randn(2, num_dat)
feat_neg[0, :] *= .5
feat_neg[1, :] *= 2.
feat_neg[0, :] -= dist
feat_neg = feat_neg.clip(-xyamax, xyamax)

# set up the testing features
npoints = 101
x1 = N.linspace(-xyamax, xyamax, npoints)
x2 = N.linspace(-xyamax, xyamax, npoints)
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)
 ds_lin = patternsPos + patternsNeg

Let’s add another dataset: XOR. This problem is not linear separable and therefore need a non-linear classifier to be solved. The dataset is provided by the PyMVPA dataset warehouse.

# 30 samples per condition, SNR 3
ds_nl = pureMultivariateSignal(30, 3)
datasets = {'linear': ds_lin, 'non-linear': ds_nl}

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),
        'GNB': GNB(common_variance=True),
        'GNB(common_variance=False)': GNB(common_variance=False),
    }

Now we are ready to run the classifiers. The following loop trains and queries each classifier to finally generate a nice plot showing the decision surface of each individual classifier, both for the linear and the non-linear dataset.

for id, ds in datasets.iteritems():
    # loop over classifiers and show how they do
    fig = 0
    # make a new figure
    P.figure(figsize=(9, 9))
print "Processing %s problem..." % id

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, 3, fig)

    # plot the training points
    P.plot(ds.samples[ds.labels == 1, 0],
           ds.samples[ds.labels == 1, 1],
           "r."
    P.plot(ds.samples[ds.labels == 0, 0],
           ds.samples[ds.labels == 0, 1],
           "b."

    # select the classifier
    clf = clfs[c]

    # enable saving of the values used for the prediction
    clf.states.enable('values')

    # train with the known points
    clf.train(ds)

    # 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.startswith('k-Nearest'):
        # 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 in ['SMLR']:
        res = N.asarray(clf.values[:, 1])
    elif c.startswith('GNB'):
        # Since probabilities are raw: for visualization lets
        # operate on logprobs and in comparison one to another
        res = clf.values[:, 1] - clf.values[:, 0]
        # Scale and position around 0.5
        res = 0.5 + res/max(N.abs(res))
    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((npoints, npoints))

    # 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)
    P.axis('tight')

    # add the title
    P.title(c)

if cfg.getboolean('examples', 'interactive', True):
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.

```python
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(pymvpa_dataroot, 'xavr1010.dat'))

# make up some artificial 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(pymvpa_dataroot, '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 output of the provided example should look like
9.4.4 Self-organizing Maps

This is a demonstration of how a self-organizing map (SOM), also known as a Kohonen network, can be used to map high-dimensional data into a two-dimensional representation. For the sake of an easy visualization ‘high-dimensional’ in this case is 3D.

In general, SOMs might be useful for visualizing high-dimensional data in terms of its similarity structure. Especially large SOMs (i.e. with large number of Kohonen units) are known to perform mappings that preserve the topology of the original data, i.e. neighboring data points in input space will also be represented in adjacent locations on the SOM.

The following code shows the ‘classic’ color mapping example, i.e. the SOM will map a number of colors into a rectangular area.

```python
from mvpa.suite import *

colors = [[0., 0., 0.],
          [0., 0., 1.],
          [0., 0., 0.5],
          [0.125, 0.529, 1.0],
          [0.33, 0.4, 0.67],
          [0.6, 0.5, 1.0],
          [0., 1., 0.],
          [1., 0., 0.],
          [0., 1., 1.],
          [1., 0., 1.],
          [1., 1., 0.],
          [1., 1., 1.],
          [.33, .33, .33],
          [.5, .5, .5],
          [.66, .66, .66]]
```

See Also:
The full source code of this example is included in the PyMVPA source distribution (doc/examples/topo_plot.py).
# store the names of the colors for visualization later on

```python
color_names = \\
    ['black', 'blue', 'darkblue', 'skyblue',
     'greyblue', 'lilac', 'green', 'red',
     'cyan', 'violet', 'yellow', 'white',
     'darkgrey', 'mediumgrey', 'lightgrey']
```

Since we are going to use a mapper, we will put the color vectors into a dataset. To be able to do this, we will assign an arbitrary label, although it will not be used at all, since this SOM mapper uses an unsupervised training algorithm.

```python
ds = Dataset(samples=colors, labels=1)
```

Now we can instantiate the mapper. It will internally use a so-called Kohonen layer to map the data onto. We tell the mapper to use a rectangular layer with 20 x 30 units. This will be the output space of the mapper. Additionally, we tell it to train the network using 400 iterations and to use custom learning rate.

```python
som = SimpleSOMMapper((20, 30), 400, learning_rate=0.05)
```

Finally, we train the mapper with the previously defined ‘color’ dataset.

```python
som.train(ds)
```

Each unit in the Kohonen layer can be treated as a pointer into the high-dimensional input space, that can be queried to inspect which input subspaces the SOM maps onto certain sections of its 2D output space. The color-mapping generated by this example’s SOM can be shown with a single matplotlib call:

```python
P.imshow(som.K, origin='lower')
```

And now, let’s take a look onto which coordinates the initial training prototypes were mapped to. The get those coordinates we can simply feed the training data to the mapper and plot the output.

```python
mapped = som(colors)
```

```python
P.title('Color SOM')
```

```python
# SOM's kshape is (rows x columns), while matplotlib wants (X x Y)
for i, m in enumerate(mapped):
    P.text(m[1], m[0], color_names[i], ha='center', va='center',
           bbox=dict(facecolor='white', alpha=0.5, lw=0))
```

The text labels of the original training colors will appear at the ‘mapped’ locations in the SOM – and should match with the underlying color.

```python
# show the figure
if cfg.getboolean('examples', 'interactive', True):
    P.show()
```

The following figure shows an exemplary solution of the SOM mapping of the 3D color-space onto the 2D SOM node layer:
9.5 Miscellaneous

9.5.1 Kernel-Demo

This is an example demonstrating various kernel implementation in PyMVPA.

```python
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)
```

See Also:

The full source code of this example is included in the PyMVPA source distribution (`doc/examples/som.py`).
**print** kernel
    result = kernel.compute(data)

# In the following we draw some 2D functions at random from the
# distribution N(0,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/classification 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.

First, importing the necessary pieces:

```python
import numpy as N
import pylab as P
from mvpa.misc.plot import plotErrLine
from mvpa.misc.fx import singleGammaHRF, leastSqFit
from mvpa import cfg
```

Now, we generate some noisy “trial time courses” from a simple gamma function (40 samples, 6s time-to-peak, 7s FWHM and no additional scaling:

```python
a = N.asarray([singleGammaHRF(N.arange(20), A=6, W=7, K=1) * 40])
# get closer to reality with noise
a += N.random.normal(size=a.shape)
```

Fitting a gamma function to this data is easy (using resonable seeds for the parameter search (5s time-to-peak, 5s FWHM, and no scaling):

```python
fpar, succ = leastSqFit(singleGammaHRF, [5,5,1], a)
```

Generate high-resolution curves for the ’true’ time course and the fitted one for visualization and plot them together with the data:
x = N.linspace(0, 20)
curves = [(x, singleGammaHRF(x, 6, 7, 1)),
          (x, singleGammaHRF(x, *fpar))]

# plot data (with error bars) and both curves
plotErrLine(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 output 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.

A recent paper by Jurica and van Leeuwen (2009) describes an open-source MATLAB®-to-Python compiler which might be a very useful tool to migrate a substantial amount of Matlab-based source code to Python and therefore also aids the migration of developers from Matlab to the new “general open-source lingua franca for scientific computation”.
FREQUENTLY ASKED QUESTIONS

11.1 General

11.1.1 It is sloooooow. 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:

```python
>>> 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:

```python
>>> 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 initial 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:
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](http://git-scm.com/docs) or a more comprehensive [Git tutorial](http://git-scm.com/book).

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 your 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 behavior (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:

```python
>>> 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.

```python
>>> 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.

```python
>>> 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:

```python
>>> 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:

```python
>>> # 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:

```python
>>> 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:

```python
>>> 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:
```python
>>> 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 explicitly:

```python
>>> 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.
```
The literature concerning the application of multivariate pattern analysis procedures to neuro-scientific datasets contains a lot of specific terms to refer to procedures or types of data, that are of particular importance. Unfortunately, sometimes various terms refer to the same construct and even worse these terms do not necessarily match the terminology used in the machine learning literature. The following 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 continuous 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 comparison 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-quality 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 measure 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 whose correlate is measured 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 acquired samples, and, thus, related to a chunk.

**Example**
Another term for sample.

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

**fMRI**
This abbreviation stands for functional magnetic resonance imaging.

**Label**
A label associates each 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 Discrimination 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), where instead of raw sensitivity significance testing result is assigned.

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

Weight Vector
See sensitivity.
This list aims to be a collection of literature, that is of particular interest in the context of multivariate 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.


*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


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

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


**Keywords:** least angle regression, LARS

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


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


*Focused review article emphasizing the role of transparency to facilitate adoption and evaluation of statistical learning techniques in neuroimaging research.*


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

**Keywords:** PyMVPA, fMRI

**DOI:** http://dx.doi.org/10.1007/s12021-008-9041-y


*Demonstration of PyMVPA capabilities concerning multi-modal or modality-agnostic data analysis.*

**Keywords:** PyMVPA, fMRI, EEG, MEG, extracellular recordings
DOI: http://dx.doi.org/10.3389/neuro.11.003.2009


**Keywords:** support vector machine, SVM, recursive feature elimination, RFE

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


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


**Keywords:** split-correlation classifier

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


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

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


DOI: http://dx.doi.org/10.3389/neuro.11.005.2009


A summary of the relationship of machine learning and cognitive science. Moreover it also points out the role of kernel-based methods in this context.

**Keywords:** kernel, similarity

DOI: http://dx.doi.org/10.1016/j.tics.2009.06.002


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

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


This paper offers an approach to make sense out of feature sensitivities of non-linear classifiers.


*Paper introducing the searchlight algorithm.*

**Keywords:** searchlight

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


DOI: http://dx.doi.org/10.3389/neuro.06.004.2008


**Keywords:** sparse multinomial logistic regression, SMLR

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

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

Manelis, A., Hanson, C. & Hanson, S. J. (2010). Implicit memory for object locations depends on reactivation of encoding-related brain regions. Human Brain Mapping.

Keywords: PyMVPA, implicit memory, MRI


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


DOI: http://dx.doi.org/10.1093/scan/nsn044


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


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


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


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


DOI: http://dx.doi.org/10.1016/j.neuroimage.2008.11.007


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


Keywords: support vector machine, SVM, sensitivity
DOI: http://dx.doi.org/10.1016/j.jneumeth.2008.04.008


First published study employing PyMVPA for MRI-based analysis of Psychosis.
Keywords: PyMVPA, psychosis, MRI
DOI: http://dx.doi.org/10.1016/j.biopsych.2009.07.019

Keywords: support vector machine, SVM

Keywords: support vector machine, SVM, group analysis
DOI: http://dx.doi.org/10.1016/j.neuroimage.2007.03.072

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

The MIT License

Copyright (c) 2006-2009 Michael Hanke
2007-2009 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.
This changelog only lists rather macroscopic changes to PyMVPA. The full VCS changelog for 0.4.x series of PyMVPA is available here:

https://github.com/PyMVPA/PyMVPA/commits/maint%2F0.4

In addition there is also a somewhat unconventional visual changelog:

http://www.pymvpa.org/history.html

‘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.

• Many, many, many

15.1 Releases

• 0.4.8 (Tue, Apr 23 2012) (Total: 14 commits)

A bugfix release

  – Fixed

    * Compatibility with libsvm 3.10, shogun >= 1.0 (Closes: #655643)
    * import ma directly from numpy
    * GPRLinearWeights – correct access to weights
    * FslEV3 – gzip import and getNEVs
    * read_fsl_design() – correct splitting of the fields
    * score() – assure std to be an array during application

  – RF

    * extensions are built inplace

• 0.4.7 (Tue, Mar 07 2011) (Total: 12 commits)

A bugfix release

  – Fixed
* Addressed the issue with input NIfTI files having scl_* fields set: it could result in incorrect analyses and map2nifti-produced NIfTI files. Now input files account for scaling/offset if scl_ fields direct to do so. Moreover upon map2nifti, those fields get reset.

* doc/examples/searchlight_minimal.py - best error is the minimal one

  - Enhancements
    * GNB can now tolerate training datasets with a single label
    * TreeClassifier can have trailing nodes with no classifier assigned

• 0.4.6 (Tue, Feb 01 2011) (Total: 20 commits)

  A bugfix release

  - Fixed (few BF commits):
    * Compatibility with numpy 1.5.1 (histogram) and scipy 0.8.0 (workaround for a regression in legende)
    * Compatibility with libsvm 3.0
    * PLR robustification

  - Enhancements
    * Enforce suppression of numpy warnings while running unitests. Also setting verbosity >= 3 enables all warnings (Python, NumPy, and PyMVPA)
    * doc/examples/nested_cv.py example (adopted from 0.5)
    * Introduced base class LearnerError for classifiers' exceptions (adopted from 0.5)
    * Adjusted example data to live up to nibabel's warranty of NIfTI standard-compliance
    * More robust operation of MC iterations – skip iterations where classifier experienced difficulties and raise an exception (e.g. due to degenerate data)

• 0.4.5 (Fri, Oct 01 2010) (Total: 27 commits)

  A bugfix release

  - Fixed (13 BF commits):
    * Compatible with LIBSVM >= 2.91 (Closes: #583018)
    * No string exceptions raised (Python 2.6 compatibility)
    * Setting of shrinking parameter in sg interface
    * Deducing number of SVs for SVR (LIBSVM)
    * Correction of significance in the tails of non-parametric tests

  - Miscellaneous:
    * Development repository moved to http://github.com/PyMVPA/PyMVPA

• 0.4.4 (Mon, Feb 2 2010) (Total: 144 commits)

  Primarily a bugfix release, probably the last in 0.4 series since development for 0.5 release is leaping forward.

  - New functionality (19 NF commits):
    * GNB implements Gaussian Naïve Bayes Classifier.
    * read_fsl_design() to read FSL FEAT design.fsf files (Contributed by Russell A. Poldrack).
    * SequenceStats to provide basic statistics on labels sequence (counter-balancing, autocorrelation).
    * New exceptions DegenerateInputError and FailedToTrainError to be thrown by classifiers primarily during training/testing.
* Debug target `STATMC` to report on progress of Monte-Carlo sampling (during permutation testing).

-- Refactored (15 RF commits):
* To get users prepared to 0.5 release, internally and in some examples/documentation, access to states and parameters is done via corresponding collections, not from the top level object (e.g. `clf.states.predictions` instead of soon-to-be-deprecated `clf.predictions`). That should lead also to improved performance.
* Adopted copy.py from python2.6 (support Ellipsis as well).

-- Fixed (38 BF commits):
* GLM output does not depend on the enabled states any more.
* Variety of docstrings fixed and/or improved.
* Do not derive NaN scaling for SVM’s C whenever data is degenerate (lead to never finishing SVM training).
* `sg`:
  - KRR is optional now – avoids crashing if KRR is not available.
  - tolerance to absent `set_precompute_matrix` in svmlight in recent shogun versions.
  - support for recent (present in 0.9.1) API change in exposing debug levels.
* Python 2.4 compatibility issues: `kNN` and `IFS`

* 0.4.3 (Sat, 5 Sep 2009) (Total: 165 commits)
  -- Online documentation editor is no longer available due to low demand – please submit changes via email.
  -- Performance (Contributed by Valentin Haenel) (3 OPT commits):
    * Further optimized LIBSVM bindings.
    * Copy-if-sorted in `selectFeatures`.
  -- New functionality (25 NF commits):
    * `ProcrusteanMapper` with orthogonal and oblique transformations.
    * Ability to generate simple reports using `reportlab`. See/run examples/match_distribution.py for example.
    * `TreeClassifier` – construct simple hierarchies of classifiers.
    * `wtf()` to report information about the system/PyMVPA to be included in the bug reports.
    * Parameter ‘reverse’ to swap training/testing splits in `Splitter`.
    * Example code for the analysis of event-related dataset using `ERNiftiDataset`.
    * `toEvents()` to create lists of `Event`.
    * `mvpa-prep-fmri` was extended with plotting of motion correction parameters.
    * `ColumnData` can be explicitly told either file contains a header.
    * In `XMLBasedAtlas` (e.g. fsl atlases) it is now possible to provide custom ‘image_file’ to get maps or indexes for the areas given an atlas’s volume registered into subject space.
    * Updated included LIBSVM version to 2.89 and provided support for its “silencing”.
  -- Refactored (27 RF commits):
    * Dataset’s `copy()` with deep=False allows for shallow copying the dataset.
    * `FeatureSelectionClassifier` in `warehouse` not to reuse the same classifiers, but to use clones.
– Fixed (70 BF commits):
  * OneWayAnova: previously degrees of freedom were not considered while computing F-scores.
  * Majority voting strategy in kNN: it was not working.
  * Various fixes to ensure cross-platform building (numpy header locations, etc).
  * Stability fixes in ConfusionMatrix.
  * idsonboundaries(): samples at the end of the sequence were not handled properly.
  * Proper “untraining” of FeatureSelectionClassifier’s classifiers which use sensitivities: it could lead to various unpleasant side-effects if the same slave classifier was used simultaneously by multiple MetaClassifiers (like TreeClassifier).

– Documentation (25 DOC commits): citations, spelling corrections, etc.

* 0.4.2 (Mon, 25 May 2009)
  – New correlation stability measure (CorrStability).
  – New elastic net classifier (ENET).
  – New GLM-Net regression/classifier (GLMNET).
  – New measure CompoundOneWayAnova.
  – New measure DSMDatasetMeasure.
  – New meta-measure TScoredFeaturewiseMeasure.
  – New basic GLM implementation.
  – New examples for Gaussian process regression.
  – New example showing a searchlight analysis employing a dissimilarity matrix based measure.
  – New ZScoreMapper.
  – New import helper for FSL design matrices (FslGLMDesign).
  – New implementation of a mapper using a self-organizing map (SimpleSOMMapper) and a corresponding example.
  – Matplotlib backend is now configurable via MVPA_MATPLOTLIB_BACKEND.
  – PyMVPA version is now available from mvpa.__version__.
  – Renamed mvpa.misc.plot.errLinePlot to plotErrLine() for consistency.
  – Fixed NFoldSplitter to support N-3 and larger splits.
  – Improved speed of LIBSVM backend. Thanks to Valentin Haenel and Tiziano Zito.
  – Updated included LIBSVM version to 2.89.
  – Adjust LIBSVM Python interface for recent NumPy API and latest LIBSVM release 2.89.
  – Refactored examples parser into a standalone tool to turn PyMVPA examples into restructured text sources.

* 0.4.1 (Sat, 24 Jan 2009)
  – Unit tests and example data are now also installed. In conjunction with mvpa.test(), this allow to easily run unittests from within Python.
  – NiftiDataset capable to handle files with less than 4 dimensions, which can, optionally, be provided as a list of filenames or NiftiImage objects. That makes it easy to load data from a sequence of files.
  – Changes (code refactorings) which might impact any user who imports from suite:
    * Pre-populated warehouses of classifiers and regressions are renamed from clfs and regrs into clfswh and regrswh respectively.
**Hamster** is not derived from `dict` any longer – just from a basic `object` class. API includes methods ‘dump’, ‘adict’ and a property ‘registered’.

- Changes (code refactorings) which *should not impact* any user who imports from `suite`:
  - * Meta classifiers definitions moved from `base` into `meta`.
  - * Splitters definitions moved from `splitter` into `splitters`.

### 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 arbitrary length (e.g. to build preprocessing pipelines).
- New **EventData** 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 as DOI or similar URL reference to the original document.
- Added initial glossary to the manual.
- 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.
  – Improved and more flexible `plotERPs()`.
  – New `IterativeRelief` sensitivity analyzer.
  – 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 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 grouped 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 compatibility with MacOS.
- Extended FAQ section of the manual.
- Bugfix: doubleGammaHRF() ignoring K2 argument.

• 0.2.2 (Tue, 17 Jun 2008)
  - 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.


– 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.
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

**mvpa**

MultiVariate Pattern Analysis

**Package Organization**

The mvpa package contains the following subpackages and modules:

- **group Algorithms**
  - algorithms
- **group Anatomical Atlases**
  - atlases
- **group Basic Data Structures**
  - datasets
- **group Classifiers (supervised learners)**
  - clfs
- **group Feature Selections**
  - featsel
- **group Mappers (usually unsupervised learners)**
  - mappers
- **group Measures**
  - measures
- **group Miscellaneous**
  - base misc support
16.2 Datasets: Input, Output, Storage and Preprocessing

16.2.1 datasets.base

Module: datasets.base

Inheritance diagram for `mvpa.datasets.base`:

```
  datasets.base.Dataset | misc.state.ClassWthCollections
    |                   |
    |                   |
 datasets.base._Dataset
```

Dataset container
**Dataset**

```python
class mvpa.datasets.base.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.

Groups

- **Creators:** __init__, selectFeatures, selectSamples, applyMapper
- **Mutators:** permuteLabels

Important: labels assumed to be immutable, i.e. no one should modify them externally by accessing indexed items, i.e. something like dataset.labels[1] += 100 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 ways 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.

2. Copy constructor mode

   The second way is used internally to perform quick copying 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.

Parameters

- **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.

Keywords

samples

[ndarray] 2d array (samples x features)

labels

An array or scalar value defining labels for each samples. Generally labels should be numeric, unless labels_map is used
labels_map

[None or bool or dict] Map original labels into numeric labels. If True, the mapping is computed if labels are literal. If False, no mapping is computed. If dict instance – provided mapping is verified and applied. If you want to have labels_map just be present given already numeric labels, just assign labels_map dictionary to existing dataset instance.

cpyunks

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.

aggregateFeatures

(dataset, fx=<function mean at 0x2982c80>)  
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).

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.

Parameters

- **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.

NOTE: selectFeatures is pretty much

applyMapper(featuresmapper=MaskMapper(...))

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 grouping into chunks or each sample was independent, thus belonged to its own chunk.

Parameters

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

Parameters

\(ids\) (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.

Parameters

\(mask\) (1d ndarray) – Feature mask.

Return type
ndarray

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

copy \((deep=True)\)
Create a copy (clone) of the dataset, by fully copying current one

Keywords

deep
[bool] deep flag is provided to \_init\_ for copy_{samples,data,dsattr}. By default full copy is done.

defineFeatureGroups \((definition)\)
Assign \(definition\) to featuregroups

XXX Feature-groups was not finished to be useful

detrend \((dataset, perchunk=False, model='linear', polyord=None, opt_reg=None)\)
Given a dataset, detrend the data inplace either entirely or per each chunk

Parameters

\*dataset (Dataset) – dataset to operate on

\*perchunk (bool) – either to operate on whole dataset at once or on each chunk sepa-

\*model – Type of detrending model to run. If ‘linear’ or ‘constant’, scipy.signal.detrend
is used to perform a linear or demeaning detrend. Polynomial detrending is activated
when ‘regress’ is used or when polyord or opt_reg are specified.

\*polyord (int or list) – Order of the Legendre polynomial to remove from the data. This
will remove every polynomial up to and including the provided value. For example,
3 will remove 0th, 1st, 2nd, and 3rd order polynomials from the data. N.B.: The 0th
polynomial is the baseline shift, the 1st is the linear trend. If you specify a single int
and perchunk is True, then this value is used for each chunk. You can also specify a
different polyord value for each chunk by providing a list or ndarray of polyord values
the length of the number of chunks.

\*opt_reg (ndarray) – Optional ndarray of additional information to regress out from
the dataset. One example would be to regress out motion parameters. As with the
data, time is on the first axis.

getLabelsMap ()
Stored labels map (if any)
getNFeatures()
Number of features per pattern.

getNumSamples()
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 chosen 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.

generateSamplesPerChunkLabel(dataset)
Returns an array with the number of samples per label in each chunk.

Array shape is (chunks x labels).

Parameters

- dataset (Dataset) – Source dataset.

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)

idsbylabels(x)

idsonboundaries(prior=0, post=0, attributes_to_track=[‘labels’, ‘chunks’], af-
fected_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

Parameters

- 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 selec-
tion on samples or features was requested (to discriminate between no selected items,
and no selections)
labels_map
    Stored labels map (if any)

nfeatures
    Number of features per pattern.

nsamples
    Currently available number of patterns.

origids

permuteLabels (status, perchunk=True, assure_permute=False)
    Permute the labels.
    TODO: rename status into something closer in semantics.

    Parameters

    *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 occurrences of each label value within a certain
        chunk constant.

    *assure_permute (bool) – If True, assures that labels are permuted, ie any one is
        different from the original one

removeInvariantFeatures (dataset)
    Returns a new dataset with all invariant features removed.

samples

samplesperchunk

samplesperlabel

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:
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:

```python
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.

**Parameters**

*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 copy of corresponding features from the original samples array.

**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

**Parameters**

*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

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

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

`uniquechunks`

`uniquelabels`

`where (*args, **kwargs)`

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

XXX somewhat obsoletes idi by...

`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, i.e. mean and standard deviation are calculated individually.

`mvpa.datasets.base.datasetmethod(func)`

Decorator to easily bind functions to a Dataset class

### 16.2.2 datasets.channel

**Module:** `datasets.channel`

Inheritance diagram for `mvpa.datasets.channel`:

```
datasets.base.Dataset
    ↓
datasets.mapped.MappedDataset
    ↓
datasets.channel.ChannelDataset
```

Dataset handling data structured in channels.

**ChannelDataset**

```python
class mvpa.datasets.channel.ChannelDataset (samples=None, dsattr=None, t0=None, dt=None, channelids=None, **kwargs)
```

Bases: `mvpa.datasets.mapped.MappedDataset`

Dataset handling data structured into channels.

16.2. Datasets: Input, Output, Storage and Preprocessing
Channels are assumed to contain several timepoints, thus this Dataset stores some additional properties (reference time \( t_0 \), temporal distance of two timepoints \( dt \) and \( \text{channelids} \) (names)).

See Also:
Please refer to the documentation of the base class for more information:

\texttt{MappedDataset}

Initialize \texttt{ChannelDataset}.

Parameters

- **\texttt{samples} (ndarray)** – Three-dimensional array: (samples x channels x timepoints).
- **\texttt{t0} (float)** – Reference time of the first timepoint. Can be used to preserve information about the onset of some stimulation. Preferably in seconds.
- **\texttt{dt} (float)** – Temporal distance between two timepoints. Has to be given in seconds. Otherwise \texttt{samplingrate} property will not return Hz.
- **\texttt{channelids} (list)** – List of channel names.
- **\texttt{mapper} (Instance of \texttt{Mapper})** – This mapper will be embedded in the dataset and is used and updated, by all subsequent mapping or feature selection procedures.
- **\texttt{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.
- **\texttt{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.
- **\texttt{dtype} (type | None)** – If None – do not change data type if \texttt{samples} is an ndarray. Otherwise convert samples to dtype.

\texttt{channelids}
List of channel IDs

\texttt{dt}
Time difference between two samples (in seconds).

\texttt{resample (nt=None, sr=None, dt=None, window='ham', inplace=True, **kwargs)}
Convenience method to resample data sample channel-wise.

Resampling target can be specified by number of timepoint or temporal distance or sampling rate.

Please note that this method only operates on \texttt{ChannelDataset} and always returns such.

Parameters

- **\texttt{nt} (int)** – Number of timepoints to resample to.
- **\texttt{dt} (float)** – Temporal distance of samples after resampling.
- **\texttt{sr} (float)** – Target sampling rate.
- **\texttt{inplace} (bool)** – If inplace=False, it would create and return a new dataset with new samples
- **\texttt{**kwargs}** – All additional arguments are passed to resample() from scipy.signal

Return
\texttt{ChannelDataset}

\texttt{samplingrate}
Yeah, sampling rate.
substractBaseline \((t=None)\)

Subtract 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.

Parameters

- \(t\) \((\text{int} | \text{float} | \text{None})\) – 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 \textit{samplingrate}. Finally, if \textit{None} the \textit{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.

16.2.3 datasets.eep

Module: datasets.eep

Inheritance diagram for mvpa.datasets.eep:

Dataset that gets its samples from an EEP binary file

EEPDataset

class mvpa.datasets.eep.EEPDataset \((\text{samples}=\text{None}, **\text{kwargs})\)

Bases: mvpa.datasets.channel.ChannelDataset

Dataset using a EEP binary file as source.

EEP files are used by \textit{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
See Also:
Please refer to the documentation of the base class for more information:

ChannelDataset

Initialize EEPDataset.

Parameters

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

16.2.4 datasets.event

Module: datasets.event

Inheritance diagram for mvpa.datasets.event:

Event-based dataset type
EventDataset

class `mvpa.datasets.event.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.

Parameters

- **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.

16.2.5 datasets.mapped

Module: `datasets.mapped`

Inheritance diagram for `mvpa.datasets.mapped:`
Mapped dataset

**MappedDataset**

class `mvpa.datasets.mapped.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 construction 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).

See Also:

Please refer to the documentation of the base class for more information:

**Dataset**

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.

Parameters

- **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.
- **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.

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

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.

Parameters

• 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

16.2.6 datasets.masked

Module: datasets.masked

Inheritance diagram for mvpa.datasets.masked:

Dataset with applied mask

MaskedDataset
class mvpa.datasets.masked.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 checks/data_mangling in the constructor, it might be absorbed inside generic `MappedDataset`

Parameters

- `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.

Parameters

- `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.

16.2.7 datasets.meta

Module: `datasets.meta`

Inheritance diagram for `mvpa.datasets.meta`:

```
datasets.meta.MetaDataset
```

Dataset container

**MetaDataset**

class `mvpa.datasets.meta.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 MetaDataset 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

Parameters

- `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)
  Perform reverse mapping
    Return
    List of results per each used mapper and the corresponding part of the provided 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.

16.2.8 datasets.miscfx

Module: datasets.miscfx

Inheritance diagram for mvpa.datasets.miscfx:

```
 datasets.miscfx.SequenceStats
```

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.
Class

SequenceStats

class mvpa.datasets.miscfx.SequenceStats (seq, order=2)

Bases: dict

Simple helper to provide representation of sequence statistics
Matlab analog: http://cfn.upenn.edu/aguirre/code/matlablib/mseq/mtest.m

WARNING: Experimental – API might change without warning! Current implementation is ugly!

Initialize SequenceStats

Parameters

seq (list or ndarray) – Actual sequence of labels

Keywords

order [int] Maximal order of counter-balancing check. For perfect counterbalancing all matrices should be identical

plot ()

Plot correlation coefficients

Functions

mvpa.datasets.miscfx.aggregateFeatures (dataset, fx=<function mean at 0x2982c80>)

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).

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 grouping into chunks or each sample was independent, thus belonged to its own chunk

Parameters

• 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

mvpa.datasets.miscfx.getSamplesPerChunkLabel (dataset)

Returns an array with the number of samples per label in each chunk.

Array shape is (chunks x labels).

Parameters

dataset (Dataset) – Source dataset.

mvpa.datasets.miscfx.removeInvariantFeatures (dataset)

Returns a new dataset with all invariant features removed.

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.

16.2.9 datasets.miscfx_sp

Module: datasets.miscfx_sp

Misc function performing operations on datasets which are based on scipy

mvpa.datasets.miscfx_sp.detrend(dataset, perchunk=False, model='linear', polyord=None, opt_reg=None)

Given a dataset, detrend the data inplace either entirely or per each chunk

Parameters

• dataset (Dataset) – dataset to operate on

• perchunk (bool) – either to operate on whole dataset at once or on each chunk separately

• model – Type of detrending model to run. If ‘linear’ or ‘constant’, scipy.signal.detrend is used to perform a linear or demeaning detrend. Polynomial detrending is activated when ‘regress’ is used or when polyord or opt_reg are specified.

• polyord (int or list) – Order of the Legendre polynomial to remove from the data. This will remove every polynomial up to and including the provided value. For example, 3 will remove 0th, 1st, 2nd, and 3rd order polynomials from the data. N.B.: The 0th polynomial is the baseline shift, the 1st is the linear trend. If you specify a single int and perchunk is True, then this value is used for each chunk. You can also specify a different polyord value for each chunk by providing a list or ndarray of polyord values the length of the number of chunks.

• opt_reg (ndarray) – Optional ndarray of additional information to regress out from the dataset. One example would be to regress out motion parameters. As with the data, time is on the first axis.

16.2.10 datasets.nifti

Module: datasets.nifti

Inheritance diagram for mvpa.datasets.nifti:
Dataset that gets its samples from a NIfTI file

### Classes

**ERNiftiDataset**

class `mvpa.datasets.nifti.ERNiftiDataset`(samples=None, events=None, mask=None, evconv=False, storeoffset=False, tr=None, enforce_dim=4, scale_data=True, **kwargs)

Bases: `mvpa.datasets.event.EventDataset`

Dataset with event-defined samples from a NIfTI timeseries image.

This is a convenience dataset to facilitate the analysis of event-related fMRI datasets. Boxcar-shaped samples are automatically extracted from the full timeseries using `Event` definition lists. For each event all volumes covering that particular event in time (including partial coverage) are used to form the corresponding sample.

The class supports the conversion of events defined in ‘realtime’ into the discrete temporal space defined by the NIfTI image. Moreover, potentially varying offsets between true event onset and timepoint of the first selected volume can be stored as an additional feature in the dataset.

Additionally, the dataset supports masking. This is done similar to the masking capabilities of `NiftiDataset`. However, the mask can either be of the same shape as a single NIfTI volume, or can be of the same shape as the generated boxcar samples, i.e. a samples consisting of three volumes with 24 slices and 64x64 inplane resolution needs a mask with shape (3, 24, 64, 64). In the former case the mask volume is automatically expanded to be identical in a volumes of the boxcar.

**Parameters**

- `mask` *(str | NiftiImage | ndarray)* – Filename of a NIfTI image or a `NiftiImage` instance or an ndarray of appropriate shape.
- `evconv` *(bool)* – Convert event definitions using `onset` and `duration` in some temporal unit into #sample notation.
• `storeoffset` (bool) – Whether to store temporal offset information when converting Events into discrete time. Only considered when evconv == True.

• `tr` (float) – Temporal distance of two adjacent NIfTI volumes. This can be used to override the corresponding value in the NIfTI header.

• `enforce_dim` (int or None) – If not None, it is the dimensionality of the data to be enforced, commonly 4D for the data, and 3D for the mask in case of fMRI.

• `scale_data` (bool) – NIfTI header specifies scl_slope and scl_inter for scaling and offsetting the data. By default those will get applied to the data (change in behavior post 0.4.6).

`map2Nifti` (data=None)
Maps a data vector into the dataspace and wraps it with a NiftiImage. The header data of this object is used to initialize the new NiftiImage (scl_slope and scl_inter are reset to 1.0 and 0.0 accordingly).

**Note:** Only the features corresponding to voxels are mapped back – not any additional features passed via the Event definitions.

Parameters

- **data** (ndarray or Dataset) – The data to be wrapped into NiftiImage. If None (default), it would wrap samples of the current dataset. If it is a Dataset instance – takes its samples for mapping.

`niftihdr`
Access to the NIfTI header dictionary.

### NiftiDataset

**class** `mvpa.datasets.nifti.NiftiDataset` (samples=None, mask=None, dsattr=None, enforce_dim=4, scale_data=True, **kwargs)

**Bases:** `mvpa.datasets.mapped.MappedDataset`

Dataset loading its samples from a NIfTI image or file.

Samples can be loaded from a NiftiImage instance or directly from a NIfTI file. This class stores all relevant information from the NIfTI file header and provides information about the metrics and neighborhood information of all voxels.

Most importantly it allows to map data back into the original data space and format via `map2Nifti()`.

This class allows for convenient pre-selection of features by providing a mask to the constructor. Only non-zero elements from this mask will be considered as features.


Parameters

- **samples** (str | NiftiImage) – Filename of a NIfTI image or a NiftiImage instance.

- **mask** (str | NiftiImage | ndarray) – Filename of a NIfTI image or a NiftiImage instance or an ndarray of appropriate shape.

- **enforce_dim** (int or None) – If not None, it is the dimensionality of the data to be enforced, commonly 4D for the data, and 3D for the mask in case of fMRI.

- **scale_data** (bool) – NIfTI header specifies scl_slope and scl_inter for scaling and offsetting the data. By default those will get applied to the data (change in behavior post 0.4.6).
dt
Time difference between two samples (in seconds). AKA TR in fMRI world.

getDt()
Return the temporal distance of two samples/volumes.

This method tries to be clever and always returns dt in seconds, by using unit information from the
NIfTI header. If such information is not present the assumed unit will also be seconds.

map2Nifti(data=None)
Maps a data vector into the dataspace and wraps it with a NiftiImage. The header data of this object is
used to initialize the new NiftiImage (scl_slope and scl_inter are reset to 1.0 and 0.0 accordingly).

Parameters
data (ndarray or Dataset) – The data to be wrapped into NiftiImage. If None (default),
it would wrap samples of the current dataset. If it is a Dataset instance – takes its samples
for mapping

niftihdr
Access to the NIfTI header dictionary.

samplingrate
Sampling rate (based on .dt).

Functions

mvpa.datasets.nifti.getNiftiData(nim)
Convenience function to extract the data array from a NiftiImage

This function will make use of advanced features of PyNIfTI to prevent unnecessary copying if a sufficient
version is available.

mvpa.datasets.nifti.getNiftiFromAnySource(src, ensure=False, enforce_dim=None, scale_data=True)
Load/access NIfTI data from files or instances.

Parameters

• src (str | NiftiImage) – Filename of a NIfTI image or a NiftiImage instance.
• ensure (bool) – If True, through ValueError exception if cannot be loaded.
• enforce_dim (int or None) – If not None, it is the dimensionality of the data to be
  enforced, commonly 4D for the data, and 3D for the mask in case of fMRI.
• scale_data (bool) – NIfTI header specifies scl_slope and scl_inter for scaling and off-
  setting the data. By default those will get applied to the data (change in behavior post
  0.4.6).

Return type
NiftiImage | None

Returns
If the source is not supported None is returned.

16.2.11 datasets.splitters

Module: datasets.splitters

Inheritance diagram for mvpa.datasets.splitters:
Collection of dataset splitters.

Module Description

Splitters are destined to split the provided dataset various ways to simplify cross-validation analysis, implement boosting of the estimates, or sample null-space via permutation testing.

Most of the splitters at the moment split 2-ways – conventionally first part is used for training, and 2nd part for testing by CrossValidatedTransferError and SplitClassifier.

Brief Description of Available Splitters

- **NoneSplitter** - just return full dataset as the desired part (training/testing)
- **OddEvenSplitter** - 2 splits: (odd samples, even samples) and (even, odd)
- **HalfSplitter** - 2 splits: (first half, second half) and (second, first)
- **NFoldSplitter** - splits for N-Fold cross validation.

Module Organization

Classes

**CustomSplitter**

```python
class mvpa.datasets.splitters.CustomSplitter(splitrule, **kwargs)
    Bases: mvpa.datasets.splitters.Splitter

Split a dataset using an arbitrary custom rule.

The splitter is configured by passing a custom splitting rule (splitrule) to its constructor. Such a rule is basically a sequence of split definitions. Every single element in this sequence results in exactly 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 attribute 3 and the last dataset contains the samples with attribute 5 and 6.

  CustomSplitter([(None, [0, 1, 2]), ([1, 2], [3], [5, 6])])

See Also:

Please refer to the documentation of the base class for more information:

Splitter
```

16.2. Datasets: Input, Output, Storage and Preprocessing 131
Cheap init.

Parameters

• nperlabel (int or str (or list of them) or float) – Number of dataset samples per label to be included in each split. If given as a float, it must be in [0,1] range and would mean the ratio of selected samples per each label. 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 whose 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

• discard_boundary (None or int or sequence of int) – If not None, how many samples on the boundaries between parts of the split to discard in the training part. If int, then discarded in all parts. If a sequence, numbers to discard are given per part of the split. E.g. if splitter splits only into (training, testing) parts, then ‘discard_boundary’=(2,0) would instruct to discard 2 samples from training which are on the boundary with testing.

• attr (str) – Sample attribute used to determine splits.

• reverse (bool) – If True, the order of datasets in the split is reversed, e.g. instead of (training, testing), (training, testing) will be spit out

HalfSplitter

class mvpa.datasets.splitters.HalfSplitter(**kwargs)
Bases: mvpa.datasets.splitters.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).

See Also:
Please refer to the documentation of the base class for more information:
Splitter

Cheap init.

Parameters

• nperlabel (int or str (or list of them) or float) – Number of dataset samples per label to be included in each split. If given as a float, it must be in [0,1] range and would mean the ratio of selected samples per each label. 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.

• nrunpersplit (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

• discard_boundary (None or int or sequence of int) – If not None, how many samples
on the boundaries between parts of the split to discard in the training part. If int, then
discarded in all parts. If a sequence, numbers to discard are given per part of the split.
E.g. if splitter splits only into (training, testing) parts, then ‘discard_boundary’=(2,0)
would instruct to discard 2 samples from training which are on the boundary with test-
ing.

• attr (str) – Sample attribute used to determine splits.

• reverse (bool) – If True, the order of datasets in the split is reversed, e.g. instead of
(training, testing), (training, testing) will be spit out

NFoldSplitter

class mvpa.datasets.splitters.NFoldSplitter(cvtype=1, **kwargs)
Bases: mvpa.datasets.splitters.Splitter

Generic N-fold data splitter.

Provide folding splitting. Given a dataset with N chunks, with cvtype=1 (which is default), it would generate
N splits, where each chunk sequentially is taken out (with replacement) for cross-validation. Example, if
there is 4 chunks, splits for cvtype=1 are:

[[1, 2, 3], [0]] [[0, 2, 3], [1]] [[0, 1, 3], [2]] [[0, 1, 2], [3]]

If cvtype>1, then all possible combinations of cvtype number of chunks are taken out for testing, so for
cvtype=2 in previous example:

[[2, 3], [0, 1]] [[1, 3], [0, 2]] [[1, 2], [0, 3]] [[0, 3], [1, 2]] [[0, 2], [1, 3]] [[0, 1], [2, 3]]

See Also:
Please refer to the documentation of the base class for more information:

Splitter

Initialize the N-fold splitter.

Parameters

• cvtype (int) – Type of cross-validation: N-(cvtype)

• nperlabel (int or str (or list of them) or float) – Number of dataset samples per label to
be included in each split. If given as a float, it must be in [0,1] range and would mean
the ratio of selected samples per each label. 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

- `discard_boundary (None or int or sequence of int)` – If not None, how many samples
  on the boundaries between parts of the split to discard in the training part. If int, then
  discarded in all parts. If a sequence, numbers to discard are given per part of the split.
  E.g. if splitter splits only into (training, testing) parts, then `discard_boundary`=(2,0)
  would instruct to discard 2 samples from training which are on the boundary with test-

- `attr (str)` – Sample attribute used to determine splits.

- `reverse (bool)` – If True, the order of datasets in the split is reversed, e.g. instead of
  (training, testing), (training, testing) will be spit out

### NGroupSplitter

**class** `mvpa.datasets.splitters.NGroupSplitter (ngroups=4, **kwargs)`

**Bases:** `mvpa.datasets.splitters.Splitter`

Split a dataset into N-groups of the sample attribute.

For example, NGroupSplitter(2) is the same as the HalfSplitter and yields to splits: first (1st half, 2nd half)
and second (2nd half, 1st half).

**See Also:**

Please refer to the documentation of the base class for more information:

**Splitter**

Initialize the N-group splitter.

**Parameters**

- `ngroups (int)` – Number of groups to split the attribute into.

- `nperlabel (int or str (or list of them) or float)` – Number of dataset samples per label to
  be included in each split. If given as a float, it must be in [0,1] range and would mean
  the ratio of selected samples per each label. 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

discard_boundary (None or int or sequence of int) – If not None, how many samples on the boundaries between parts of the split to discard in the training part. If int, then discarded in all parts. If a sequence, numbers to discard are given per part of the split. E.g. if splitter splits only into (training, testing) parts, then ‘discard_boundary’=(2,0) would instruct to discard 2 samples from training which are on the boundary with testing.

attr (str) – Sample attribute used to determine splits.

reverse (bool) – If True, the order of datasets in the split is reversed, e.g. instead of (training, testing), (training, testing) will be spit out

NoneSplitter

class mvpa.datasets.splitters.NoneSplitter (mode='second', **kwargs)

| Bases: | mvpa.datasets.splitters.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’.

See Also:

Please refer to the documentation of the base class for more information: Splitter

Cheap init – nothing special

Parameters

mode – Either ‘first’ or ‘second’ (default) – which output dataset would actually contain samples

nperlabel (int or str (or list of them) or float) – Number of dataset samples per label to be included in each split. If given as a float, it must be in [0,1] range and would mean the ratio of selected samples per each label. 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 whose 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 *count* splits are chosen random (without replacement) *count* splits are chosen equidistant Splits which are equidistant from each other

• **discard_boundary** *(None or int or sequence of int)* – If not None, how many samples on the boundaries between parts of the split to discard in the training part. If int, then discarded in all parts. If a sequence, numbers to discard are given per part of the split. E.g. if splitter splits only into (training, testing) parts, then ‘discard_boundary’=(2,0) would instruct to discard 2 samples from training which are on the boundary with testing.

• **attr** *(str)* – Sample attribute used to determine splits.

• **reverse** *(bool)* – If True, the order of datasets in the split is reversed, e.g. instead of (training, testing), (training, testing) will be spit out

**OddEvenSplitter**

class `mvpa.datasets.splitters.OddEvenSplitter` *(usevalues=False, **kwargs)*

Bases: `mvpa.datasets.splitters.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).

**See Also:**

Please refer to the documentation of the base class for more information:

`Splitter`

Cheap init.

Parameters

• **usevalues** *(bool)* – 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]).

• **nperlabel** *(int or str (or list of them) or float)* – Number of dataset samples per label to be included in each split. If given as a float, it must be in [0,1] range and would mean the ratio of selected samples per each label. 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 *count* splits are chosen random Random (without replacement) *count* splits are chosen equidistant Splits which are equidistant from each other
**discard_boundary** *(None or int or sequence of int) – If not None, how many samples on the boundaries between parts of the split to discard in the training part. If int, then discarded in all parts. If a sequence, numbers to discard are given per part of the split. E.g. if splitter splits only into (training, testing) parts, then ‘discard_boundary’=(2,0) would instruct to discard 2 samples from training which are on the boundary with testing.*

**attr** *(str) – Sample attribute used to determine splits.*

**reverse** *(bool) – If True, the order of datasets in the split is reversed, e.g. instead of (training, testing), (training, testing) will be spit out

**Splitter**

class **mvpa.datasets.splitters.Splitter**(nperlabel='all', nrunspersplit=1, permute=False, count=None, strategy='equidistant', discard_boundary=None, attr='chunks', reverse=False)

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.

Parameters

**nperlabel** *(int or str (or list of them) or float) – Number of dataset samples per label to be included in each split. If given as a float, it must be in [0,1] range and would mean the ratio of selected samples per each label. 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 who's 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

**discard_boundary** *(None or int or sequence of int) – If not None, how many samples on the boundaries between parts of the split to discard in the training part. If int, then discarded in all parts. If a sequence, numbers to discard are given per part of the split. E.g. if splitter splits only into (training, testing) parts, then ‘discard_boundary’=(2,0)
would instruct to discard 2 samples from training which are on the boundary with testing.

• **attr** *(str)* – Sample attribute used to determine splits.

• **reverse** *(bool)* – If True, the order of datasets in the split is reversed, e.g. instead of (training, testing), (training, testing) will be spit out

```python
count = None
Number (max) of splits to output on call

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.

Parameters

• **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
```

### 16.3 Mappers: Data Transformations

#### 16.3.1 mappers.array

**Module**: mappers.array

Inheritance diagram for mvpa.mappers.array:
Data mapper

**DenseArrayMapper**

```python
class mvpa.mappers.array.DenseArrayMapper(mask=mask, metric=metric, distance_function=distance_function, elementsize=elementsize, shape=shape, **kwargs):
```

Bases: `mvpa.mappers.mask.MaskMapper`

Mapper for equally spaced dense arrays.

**See Also:**

Please refer to the documentation of the base class for more information:

`MaskMapper`

Initialize DenseArrayMapper

**Parameters**

- **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 – `DiscreteMetric` 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 `DiscreteMetric` if `metric` is not specified,

- **elementsize** *(list or scalar)* – Determines spacing within `DiscreteMetric`. 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.
16.3.2 mappers.base

Module: mappers.base

Inheritance diagram for mappers.base:

```
+-------------+             +-------------+             +-------------+
| mappers.base |             | mappers.base |             | mappers.base |
| CombinedMapper |             | ProjectionMapper |             | ChainMapper  |
| mappers.base.Mapper |             |               |               |             |
```

Data mapper

Classes

ChainMapper

class mappers.base.ChainMapper(mappers, **kwargs)
    Bases: 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.

    Parameters
    • 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.
        Parameters
        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!
Parameters

- **outId (int)** — Single id of a feature in output space, whose neighbors should be determined.
- **kwargs** (*args,*) — 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.

Parameters

- **data** (array) — data array to be reverse mapped into the original dataspace.

selectOut (outIds)

Remove some elements from the last mapper in the chain.

Parameters

- **outIds** (sequence) — All output feature ids to be selected/kept.

train (dataset)

Trains the last mapper in the chain.

Parameters

- **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).

CombinedMapper

class mvpa.mappers.base.CombinedMapper (mappers, **kwargs)

Bases: mvpa.mappers.base.Mapper

Meta mapper that combines several embedded mappers. This mapper can be used to 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 turn 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).

CombinedMapper fully supports forward and backward mapping, training, runtime selection of a feature subset (in output dataspace) and retrieval of neighborhood information.

Parameters

- **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.

Parameters

- **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.

Parameters

• outId (int) – Single id of a feature in output space, whose neighbors should be determined.

• **kwargs (*args, ) – 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.

Parameters

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

Parameters

outIds (sequence) – All output feature ids to be selected/kept.

train(dataset)
Trains all embedded mappers.

The provided training dataset is split appropriately and the corresponding pieces are passed to the train() method of each embedded mapper.

Parameters

dataset (Dataset or subclass) – A dataset with the number of features matching the outSize of the CombinedMapper.

Mapper
class mvpa.mappers.base.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:
Parameters

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.
Parameters

• inId – id (index) of an element in input dataspace.
• **kwargs (*args,) – 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 = ‘property’

reverse (data)
Reverse map data from OUT space into the IN space.

16.3. Mappers: Data Transformations
**selectOut** (*outIds*)

Limit the OUT space to a certain set of features.

Parameters

* 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.

Parameters

* dataset (Dataset or subclass) –

**Note:** The default behavior of this method is to do nothing.

---

**ProjectionMapper**

```python
class mvpa.mappers.base.ProjectionMapper (selector=None, demean=True)
Bases: mvpa.mappers.base.Mapper

Linear mapping between multidimensional spaces.

This class cannot be used directly. Sub-classes have to implement the _train() method, which has to com-
pute the projection matrix _proj and optionally offset vectors _offset_in and _offset_out (if initialized with
demean=True, which is default) given a dataset (see _train() docstring for more information).

Once the projection matrix is available, this class provides functionality to perform forward and backwards
linear mapping of data, the latter by default using pseudo-inverse (but could be altered in subclasses, like
hermitian (conjugate) transpose in case of SVD). Additionally, ProjectionMapper supports optional selec-
tion 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.

See Also:

Please refer to the documentation of the base class for more information:

Mapper

Initialize the ProjectionMapper

Parameters

* 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.

Parameters

* data (ndarray) – Data array to map

* demean (boolean | None) – Override demean setting for this method call.
```

---

144 Chapter 16. Module Reference
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, *args, **kwargs)
Determine the projection matrix.

Parameters

*dataset (Dataset) – Dataset to operate on
*args – Optional positional arguments to pass to _train of subclass
**kwargs – Optional keyword arguments to pass to _train of subclass

16.3.3 mappers.boxcar

Module: mappers.boxcar

Inheritance diagram for mvpa.mappers.boxcar:

Data mapper

BoxcarMapper

class mvpa.mappers.boxcar.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).

**See Also:**
Please refer to the documentation of the base class for more information:

**Mapper**

Parameters

- **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 available 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

### 16.3.4 mappers.ica

**Module:** mappers.ica

Inheritance diagram for mvpa.mappers.ica:
ICAMapper

```python
class mvpa.mappers.ica.ICAMapper(algorithm='cubica', transpose=False, **kwargs):
    Bases: mvpa.mappers.base.ProjectionMapper

    Mapper to project data onto ICA components estimated from some dataset.
    After the mapper has been instantiated, it has to be train first. The ICA mapper only handles 2D data matrices.

    See Also:
    Please refer to the documentation of the base class for more information:
    ProjectionMapper

    Initialize instance of ICAMapper

    Parameters

    *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()
```

16.3.5 mappers.lle

Module: mappers.lle

Inheritance diagram for mvpa.mappers.lle:
Local Linear Embedding Data mapper.
This is a wrapper class around the corresponding MDP nodes LLE and HLLE (since MDP 2.4).

**LLEMapper**

class mvpa.mappers.lle.LLEMapper (k, algorithm='lle', **kwargs)

   Bases: mvpa.mappers.base.Mapper

Locally linear embeding Mapper.

This mapper performs dimensionality reduction. It wraps two algorithms provided by the Modular Data Processing (MDP) framework.

Locally linear embedding (LLE) approximates the input data with a low-dimensional surface and reduces its dimensionality by learning a mapping to the surface.

This wrapper class provides access to two different LLE algorithms (i.e. the corresponding MDP processing nodes). 1) An algorithm outlined in *An Introduction to Locally Linear Embedding* by L. Saul and S. Roweis, using improvements suggested in *Locally Linear Embedding for Classification* by D. deRidder and R.P.W. Duin (aka LLENode) and 2) Hessian Locally Linear Embedding analysis based on algorithm outlined in *Hessian Eigenmaps: new locally linear embedding techniques for high-dimensional data* by C. Grimes and D. Donoho, 2003.

**Note:** This mapper only provides forward-mapping functionality – no reverse mapping is available.

**See Also:**
http://mdp-toolkit.sourceforge.net

**Parameters**

- `k (int)` – Number of nearest neighbor to be used by the algorithm.
- `algorithm` (`'lle'` | `'hlle'`) – Either use the standard LLE algorithm or Hessian Linear Local Embedding (HLLE).
- `**kwargs` – Additional arguments are passed to the underlying MDP node. Most importantly this is the `output_dim` argument, that determines the number of dimensions to mapper is using as output space.

**forward** *(data)*
Map data from the IN dataspace into OUT space.

**getInSize ()**
Returns the size of the entity in input space
getOutSize()
    Returns the size of the entity in output space

node
    Provide access to the underlying MDP processing node.
    With some care.

reverse(data)
    Reverse map data from OUT space into the IN space.

train(ds)
    Train the mapper.

16.3.6 mappers.mask

Module: mappers.mask

Inheritance diagram for mvpa.mappers.mask:

Data mapper which applies mask to the data

MaskMapper

class mvpa.mappers.mask.MaskMapper(mask, **kwargs)
    Bases: mvpa.mappers.base.Mapper
    Mapper which uses a binary mask to select “Features”
    See Also:
    Please refer to the documentation of the base class for more information:
    Mapper
    Initialize MaskMapper
    Parameters

    • mask (array) – an array in the original dataspace and its nonzero elements are used to
define the features included in the dataset
    • metric (Metric) – Optional metric

convertOutIds2InMask(outIds)
    Returns a boolean mask with all features in outIds selected.
    This method works exactly like Mapper.convertOutIds2OutMask(), but the feature mask is finally
    (reverse) mapped into in-space.
Parameters

`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.

Parameters

`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

Feature/Bug:

• Negative outIds would not raise exception - just would be treated ‘from the tail’

16.3.7 mappers.metric

Module: mappers.metric

Inheritance diagram for mappers.metric:

```
mappers.metric.Metric
 mappers.metric.DescreteMetric
```

Classes and functions to provide sense of distances between sample points

Classes

DescreteMetric

class mva.mappers.metric.DescreteMetric(elementsize=1, distance_function=<function cartesianDistance at 0x66a4e60>, compatmask=None)

Bases: mva.mappers.metric.Metric

Find neighboring points in descretized 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.

For all origin coordinates this class exclusively operates on discretized values, not absolute coordinates (which are e.g. in mm).

Additionally, this metric has the notion of compatible and incompatible dataspace metrics, i.e. the descrete space might contain dimensions for which computing an overall distance is not meaningful. This could, for example, be a combined spatio-temporal space (three spatial dimension, plus the temporal one). This metric allows to define a boolean mask (compatmask) which dimensions share the same dataspace metrics and for which the distance function should be evaluated. If a compatmask is provided, all coordinates are projected into the subspace of the non-zero dimensions and distances are computed within that space.

However, by using a per dimension radius argument for the getNeighbor methods, it is nevertheless possible to define a neighborhood along all dimension. For all non-compatible axes the respective radius is treated as a one-dimensional distance along the respective axis.

Parameters
• **elementsize** *(float | sequence)* – The extent of a dataspace element along all dimensions.

• **distance_function** *(functor)* – The distance measure used to determine distances between dataspace elements.

• **compatmask** *(1D bool array | None)* – A mask where all non-zero elements indicate dimensions with compatible spacemetrics. If None (default) all dimensions are assumed to have compatible spacemetrics.

```python
compactmask
    Return compatmask
```

**Note:** Don’t modify in place since it would need to require to reset __filter_radius whenever changed

```python
elementsize
```

```python
filter_coord
    Lets allow to specify some custom filter to use
```

```python
getNeighbors *(origin, radius=0)*
    Returns coordinates of the neighbors which are within distance from coord.

Parameters

• **origin** *(1D array)* – The center coordinate of the neighborhood.

• **radius** *(scalar | sequence)* – If a scalar, the radius is treated as identical along all dimensions of the dataspace. If a sequence, it defines a per dimension radius, thus has to have the same number of elements as dimensions. Currently, only spherical neighborhoods are supported. Therefore, the radius has to be equal along all dimensions flagged as having compatible dataspace metrics. It is, however, possible to define variant radii for all other dimensions.

**Metric**

```python
class mvpa.mappers.metric.Metric
    Bases: object
```

Abstract class for any metric.

Subclasses abstract a metric of a dataspace with certain properties and can be queried for structural information. Currently, this is limited to neighborhood information, i.e. identifying the surround a some coordinate in the respective dataspace.

At least one of the methods (getNeighbors, getNeighbor) has to be overridden in every derived class. NOTE: derived #2 from derived class #1 has to override all methods which were overridden in class #1

```python
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
```

```python
getNeighbors *(*args, **kwargs)*
    Return the list of coordinates for the neighbors.
    By default it simply constructs the list based on the generator getNeighbor
```
16.3.8 mappers.pca

Module: mappers.pca

Inheritance diagram for mvpa.mappers.pca:

Data mapper

PCAMapper

class mvpa.mappers.pca.PCAMapper(transpose=False, **kwargs)
Bases: mvpa.mappers.base.ProjectionMapper

Mapper to project data onto PCA components estimated from some dataset.

After the mapper has been instantiated, it has to be train first. The PCA mapper only handles 2D data matrices.

See Also:
Please refer to the documentation of the base class for more information:

ProjectionMapper

Initialize instance of PCAMapper

Parameters

• 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()

var
Variances per component
16.3.9 mappers.procrustean

Module: mappers.procrustean

Inheritance diagram for mvpa.mappers.procrustean:

```
  mappers.base.Mapper
     ↓
  mappers.base.ProjectionMapper
     ↓
mappers.procrustean.ProcrusteanMapper
```

Procrustean rotation mapper

**ProcrusteanMapper**

class mvpa.mappers.procrustean.ProcrusteanMapper(scaling=True, reflection=True, reduction=True, oblique=False, oblique_rcond=-1, **kwargs)

Bases: mvpa.mappers.base.ProjectionMapper

Mapper to project from one space to another using Procrustean transformation (shift + scaling + rotation)

See Also:

Please refer to the documentation of the base class for more information:

ProjectionMapper

Initialize the ProcrusteanMapper

Parameters

- **scaling** (bool) – Scale data for the transformation (no longer rigid body transformation)
- **reflection** (bool) – Allow for the data to be reflected (so it might not be a rotation). Effective only for non-oblique transformations
- **reduction** (bool) – If true, it is allowed to map into lower-dimensional space. Forward transformation might be suboptimal then and reverse transformation might not recover all original variance
- **oblique** (bool) – Either to allow non-orthogonal transformation – might heavily overfit the data if there is less samples than dimensions. Use oblique_rcond.
- **oblique_rcond** (float) – Cutoff for ‘small’ singular values to regularize the inverse. See lstsq for more information.
- **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()

## 16.3.10 mappers.samplegroup

**Module:** mappers.samplegroup

Inheritance diagram for `mvpa.mappers.samplegroup`:

```
mappers.base.Mapper
  ↓
mappers.samplegroup.SampleGroupMapper
```

Data mapper

**SampleGroupMapper**

```python
class mvpa.mappers.samplegroup.SampleGroupMapper(fx=<function FirstAxisMean at 0x4892de8>)
```

Bases: `mvpa.mappers.base.Mapper`

Mapper to apply a mapping function to samples of the same type.

A customizmable 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...)

**See Also:**

Please refer to the documentation of the base class for more information:

**Mapper**

Initialize the PCAMapper

**Parameters**

• **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)

16.3.11 mappers.som

Module: mappers.som

Inheritance diagram for mvpa.mappers.som:

Self-organizing map (SOM) mapper.

SimpleSOMMapper

class mvpa.mappers.som.SimpleSOMMapper (kshape, niter, learning_rate=0.005, iradius=None)  
Bases: mvpa.mappers.base.Mapper  

Mapper using a self-organizing map (SOM) for dimensionality reduction.

This mapper provides a simple, but pretty fast implementation of a self-organizing map using an unsupervised training algorithm. It performs a ND -> 2D mapping, which can for, example, be used for visualization of high-dimensional data.

This SOM implementation uses squared Euclidean distance to determine the best matching Kohonen unit and a Gaussian neighborhood influence kernel.

Parameters

• kshape ((int, int)) – Shape of the internal Kohonen layer. Currently, only 2D Kohonen layers are supported, although the length of an axis might be set to 1.
• niter (int) – Number of iteration during network training.
• learning_rate (float) – Initial learning rate, which will continuously decreased during network training.
\textbf{iradius} (float | None) – Initial radius of the Gaussian neighborhood kernel radius, which will continuously decreased during network training. If None (default) the radius is set equal to the longest edge of the Kohonen layer.

\textbf{K}

Provide access to the Kohonen layer.

With some care.

\textbf{forward} (\textit{data})

Map data from the IN dataspace into OUT space.

Mapping is performed by simple determining the best matching Kohonen unit for each data sample.

\textbf{getInId} (\textit{outId})

Translate a feature id into a coordinate/index in input space.

This is not meaningful in the context of SOMs.

\textbf{getInSize} ()

Returns the size of the entity in input space.

\textbf{getOutSize} ()

Returns the size of the entity in output space.

\textbf{isValidOutId} (\textit{outId})

Validate feature id in OUT space.

\textbf{reverse} (\textit{data})

Reverse map data from OUT space into the IN space.

\textbf{selectOut} (\textit{outIds})

Limit the OUT space to a certain set of features.

This is currently not implemented. Moreover, although it is technically possible to implement this functionality, it is unsure whether it is meaningful in the context of SOMs.

\textbf{train} (\textit{ds})

Perform network training.

Parameters

\textbf{ds} (\textit{Dataset}) – All samples in the dataset will be used for unsupervised training of the SOM.

16.3.12 mappers.svd

\textbf{Module: mappers.svd}

Inheritance diagram for \texttt{mvpa.mappers.svd}:
Singular-value decomposition mapper

**SVDMapper**

class mvpa.mappers.svd.SVDMapper(**kwargs)

    Bases: mvpa.mappers.base.ProjectionMapper

    Mapper to project data onto SVD components estimated from some dataset.

    See Also:
    Please refer to the documentation of the base class for more information:
    ProjectionMapper

    Initialize the SVDMapper

    Parameters

    *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()

    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

16.3.13 mappers.wavelet

Module: mappers.wavelet

Inheritance diagram for mvpa.mappers.wavelet:
Wavelet mappers

Classes

WaveletPacketMapper

class mvpa.mappers.wavelet.WaveletPacketMapper (level=None, **kwargs)
   Bases: mvpa.mappers.wavelet._WaveletMapper

Convert signal into an overcomplete representation using Wavelet packet
Initialize WaveletPacketMapper mapper
Parameters
   level (int or None) – What level to decompose at. If ‘None’ data for all levels is provided,
   but due to different sizes, they are placed in 1D row.

WaveletTransformationMapper

class mvpa.mappers.wavelet.WaveletTransformationMapper (dim=1, wavelet='sym4',
   mode='per', maxlevel=None)
   Bases: mvpa.mappers.wavelet._WaveletMapper

Convert signal into wavelet representation
Initialize _WaveletMapper mapper
Parameters

   *dim (int or tuple of int) – dimensions to work across (for now just scalar value, ie 1D
   transformation) is supported
   *wavelet (basestring) – one from the families available within pywt package
   *mode (basestring) – periodization mode
   *maxlevel (int or None) – number of levels to use. If None - automatically selected by
   pywt
16.3.14 mappers.zscore

Module: mappers.zscore

Inheritance diagram for mvpa.mappers.zscore:

```
  mappers.base.Mapper
     ↓
  mappers.base.ProjectionMapper
     ↓
  mappers.zscore.ZScoreMapper
```

Simple mapper to perform zscoring

**ZScoreMapper**

```python
class mvpa.mappers.zscore.ZScoreMapper(baselinelabels=None, **kwargs):
    Bases: mvpa.mappers.base.ProjectionMapper

    Mapper to project data into standardized values (z-scores).
    After the mapper has been instantiated, it has to be train first.
    Since it tries to reuse ProjectionMapper, invariant features will simply be assigned a std == 1, which would be equivalent to not standardizing them at all. This is similar to not touching them at all, so similar to what zscore function currently does

    See Also:
    Please refer to the documentation of the base class for more information:
    ProjectionMapper

    Initialize ZScoreMapper

    Parameters

    • baselinelabels (None or list of int or string) – Used to compute mean and variance
      TODO: not in effect now and need to be adherent to all ‘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()
```
16.4 Classifiers and Errors

16.4.1 clfs.base

Module: clfs.base

Inheritance diagram for mvpa.clfs.base:

Base class for all classifiers.

At the moment, regressions are treated just as a special case of classifier (or vice versa), so the same base class `Classifier` is utilized for both kinds.

Classes

Classifier

class mvpa.clfs.base.Classifier(**kwargs)
Bases: mvpa.misc.state.ClassWithCollections

Abstract classifier class to be inherited by all classifiers

Note: Available state variables:

- `feature_ids`: Feature IDs which were used for the actual training.
- `predicting_time+`: Time (in seconds) which took classifier to predict
- `predictions+`: Most recent set of predictions
- `trained_dataset`: The dataset it has been trained on
- `trained_labels+`: Set of unique labels it has been trained on
- `trained_nsamples+`: Number of samples it has been trained on
- `training_confusion`: Confusion matrix of learning performance
- `training_time+`: Time (in seconds) which took classifier to train
- `values+`: Internal classifier values the most recent predictions are based on

(States enabled by default are listed with +)

See Also:

Please refer to the documentation of the base class for more information:

ClassWithCollections

Cheap initialization.

Parameters
• **enable_states** *(None or list of basestring)* – Names of the state variables which should be enabled additionally to default ones

• **disable_states** *(None or list of basestring)* – Names of the state variables which should be disabled

**clone**

Create full copy of the classifier.

It might require classifier to be untrained first due to present SWIG bindings.

TODO: think about proper re-implementation, without enrollment of deepcopy

**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 overridden in subclasses unless explicitly 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.

Parameters

• **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.

Parameters

• **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 overridden in subclasses unless explicitly needed to do so

**trained**

Either classifier was already trained
untrain()

Reset trained state

DegenerateInputError

class mvpa.clfs.base.DegenerateInputError
    Bases: mvpa.clfs.base.LearnerError

Exception to be thrown by learners if input data is bogus, i.e. no features or samples

FailedToPredictError

class mvpa.clfs.base.FailedToPredictError
    Bases: mvpa.clfs.base.LearnerError

Exception to be thrown whenever classifier fails to provide predictions. Usually happens if it was trained on degenerate data but without any complaints.

FailedToTrainError

class mvpa.clfs.base.FailedToTrainError
    Bases: mvpa.clfs.base.LearnerError

Exception to be thrown whenever classifier fails to learn for some reason

LearnerError

class mvpa.clfs.base.LearnerError
    Bases: exceptions.Exception

Base class for exceptions thrown by the learners (classifiers, regressions)

16.4.2 clfs.blr

Module: clfs.blr

Inheritance diagram for mvpa.clfs.blr:
Bayesian Linear Regression (BLR).

**BLR**

class mvpa.clfs.blr.BLR(sigma_p=None, sigma_noise=1.0, **kwargs)

Bases: mvpa.clfs.base.Classifier

Bayesian Linear Regression (BLR).

**Note:** Available state variables:

- `feature_ids`: Feature IDS which were used for the actual training.
- `log_marginal_likelihood`: Log Marginal Likelihood
- `predicted_variances`: Variance per each predicted value
- `predicting_time+`: Time (in seconds) which took classifier to predict
- `predictions+`: Most recent set of predictions
- `trained_dataset`: The dataset it has been trained on
- `trained_labels+`: Set of unique labels it has been trained on
- `trained_nsamples+`: Number of samples it has been trained on
- `training_confusion`: Confusion matrix of learning performance
- `training_time+`: Time (in seconds) which took classifier to train
- `values+`: Internal classifier values the most recent predictions are based on

(States enabled by default are listed with +)

**See Also:**

Please refer to the documentation of the base class for more information:

Classifier

Initialize a BLR regression analysis.

Parameters

- `sigma_noise` *(float)* – the standard deviation of the gaussian noise. (Defaults to 0.1)
- `regression` – Either to use ‘regression’ as regression. By default any Classifier- derived class serves as a classifier, so regression does binary classification. (Default: False)
- `enable_states` *(None or list of basestring)* – Names of the state variables which should be enabled additionally to default ones
- `disable_states` *(None or list of basestring)* – Names of the state variables which should be disabled

**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.
16.4.3 clfs.distance

Module: clfs.distance

Distance functions to be used in kernels and elsewhere

Functions

mvpa.clfs.distance.absminDistance(a, b)
Returns distance max(|a-b|) XXX There must be better name! XXX Actually, why is it absmin not absmax?
Useful to select a whole cube of a given “radius”
mvpa.clfs.distance.cartesianDistance(a, b)
Return Cartesian distance between a and b
mvpa.clfs.distance.mahalanobisDistance(x, y=None, w=None)
Calculate Mahalanobis distance of the pairs of points.

Parameters

• 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

\[ w = \text{N.linalg.solve(N.cov(x.T), N.identity(x.shape[1])}) \]

or

\[ w = \text{N.linalg.inv(N.cov(x.T))} \]
mvpa.clfs.distance.manhattanDistance(a, b)
Return Manhattan distance between a and b
mvpa.clfs.distance.oneMinusCorrelation(X, Y)
Return one minus the correlation matrix between the rows of two matrices.

This function computes a matrix of correlations between all pairs of rows of two matrices. Unlike NumPy’s
corrcoef() this function will only consider pairs across matrices and not within, e.g. both elements of a pair
never have the same source matrix as origin.

Both arrays need to have the same number of columns.

Parameters

• X (2D-array) –
• Y (2D-array) –

Example:

```python
>>> X = N.random.rand(20,80)
>>> Y = N.random.rand(5,80)
>>> C = oneMinusCorrelation(X, Y)
>>> print C.shape
(20, 5)
```
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)
\[ \|x - x\|_w = (\text{sum}_{i=1...N} (w_i|x_i - x'_i|)^p)^{1/p} \]

Parameters

- **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 \( p=2 \)

```
mvpa.clfs.distance.squared_euclidean_distance(data1, data2=None, weight=None)
```

Compute weighted euclidean distance matrix between two datasets.

Parameters

- **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)

### 16.4.4 clfs.enet

**Module:** clfs.enet

Inheritance diagram for mvpa.clfs.enet:
Elastic-Net (ENET) regression classifier.

**Classes**

**ENET**

class *mvpa.clfs.enet.ENET* (*lm=1.0, trace=False, normalize=True, intercept=True, max_steps=None, **kwargs*)

Bases: *mvpa.clfs.base.Classifier*

Elastic-Net regression (ENET) *Classifier*.

Elastic-Net is the model selection algorithm from:


Similar to SMLR, it performs a feature selection while performing classification, but instead of starting with all features, it starts with none and adds them in, which is similar to boosting.

Unlike LARS it has both L1 and L2 regularization (instead of just L1). This means that while it tries to sparsify the features it also tries to keep redundant features, which may be very very good for fMRI classification.

In the true nature of the PyMVPA framework, this algorithm was actually implemented in R by Zou and Hastie and wrapped via RPy. To make use of ENET, you must have R and RPy installed as well as both the lars and elasticnet contributed package. You can install the R and RPy with the following command on Debian-based machines:

```
sudo aptitude install python-rpy python-rpy-doc r-base-dev
```

You can then install the lars and elasticnet package by running R as root and calling:

```
install.packages()
```

16.4. Classifiers and Errors
Note: Available state variables:

- `feature_ids`: Feature IDS which were used for the actual training.
- `predicting_time+`: Time (in seconds) which took classifier to predict
- `predictions+`: Most recent set of predictions
- `trained_dataset`: The dataset it has been trained on
- `trained_labels+`: Set of unique labels it has been trained on
- `trained_nsamples+`: Number of samples it has been trained on
- `training_confusion`: Confusion matrix of learning performance
- `training_time+`: Time (in seconds) which took classifier to train
- `values+`: Internal classifier values the most recent predictions are based on

(States enabled by default are listed with +)

See Also:

Please refer to the documentation of the base class for more information:

Classifier

Initialize ENET.

See the help in R for further details on the following parameters:

Parameters

- `lm (float)`: Penalty parameter. 0 will perform LARS with no ridge regression. Default is 1.0.
- `trace (boolean)`: Whether to print progress in R as it works.
- `normalize (boolean)`: Whether to normalize the L2 Norm.
- `intercept (boolean)`: Whether to add a non-penalized intercept to the model.
- `max_steps (None or int)`: If not None, specify the total number of iterations to run. Each iteration adds a feature, but leaving it none will add until convergence.
- `regression`: Either to use ‘regression’ as regression. By default any Classifier-derived class serves as a classifier, so regression does binary classification. (Default: False)
- `enable_states (None or list of basestring)`: Names of the state variables which should be enabled additionally to default ones
- `disable_states (None or list of basestring)`: Names of the state variables which should be disabled

`getSensitivityAnalyzer(**kwargs)`

Returns a sensitivity analyzer for ENET.

weights

ENETWeights

class mvpa.clfs.enet.ENETWeights(clf, force_training=True, **kwargs)
Bases: mvpa.measures.base.Sensitivity

SensitivityAnalyzer that reports the weights ENET trained on a given Dataset.
Note: Available state variables:

- **base_sensitivities**: Stores basic sensitivities if the sensitivity relies on combining multiple ones
- **null_prob+**: State variable
- **null_t**: State variable
- **raw_results**: Computed results before applying any transformation algorithm

(States enabled by default are listed with +)

See Also:

Please refer to the documentation of the base class for more information:

**Sensitivity**

Initialize the analyzer with the classifier it shall use.

Parameters

- **clf** *(Classifier)* – classifier to use.
- **force_training** *(Bool)* – if classifier was already trained – do not retrain
- **enable_states** *(None or list of basestring)* – Names of the state variables which should be enabled additionally to default ones
- **disable_states** *(None or list of basestring)* – Names of the state variables which should be disabled
- **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.
- **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)* – The estimated distribution is used to assign a probability for a certain value of the computed measure.

### 16.4.5 clfs.glmnet

**Module:** clfs.glmnet

Inheritance diagram for mvpa.clfs.glmnet:
GLM-Net (GLMNET) regression classifier.

**Classes**

**GLMNETWeights**

```python
class mvpa.clfs.glmnet.GLMNETWeights(elf, force_training=True, **kwargs):
    Bases: mvpa.measures.base.Sensitivity

    SensitivityAnalyzer that reports the weights GLMNET trained on a given Dataset.
```

**Note:** Available state variables:

- `base_sensitivities`: Stores basic sensitivities if the sensitivity relies on combining multiple ones
- `null_prob+`: State variable
- `null_t`: State variable
- `raw_results`: Computed results before applying any transformation algorithm

(States enabled by default are listed with +)

**See Also:**

Please refer to the documentation of the base class for more information:

*Sensitivity*

Initialize the analyzer with the classifier it shall use.

**Parameters**

- `clf` (*Classifier*) – classifier to use.
- `force_training` (*Bool*) – if classifier was already trained – do not retrain
- `enable_states` (*None or list of basestring*) – Names of the state variables which should be enabled additionally to default ones
• **disable_states** (*None or list of basestring*) – Names of the state variables which should be disabled

• **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.

• **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*) – The estimated distribution is used to assign a probability for a certain value of the computed measure.

### GLMNET_C

```python
class mvpa.clfs.glmnet.GLMNET_C(**kwargs)
    Bases: mvpa.clfs.glmnet._GLMNET

GLM-NET Multinomial Classifier.


See GLMNET_Class for the gaussian regression version.
```

**Note:** Available state variables:

- **feature_ids**: Feature IDS which were used for the actual training.
- **predicting_time+**: Time (in seconds) which took classifier to predict
- **predictions+**: Most recent set of predictions
- **trained_dataset**: The dataset it has been trained on
- **trained_labels+**: Set of unique labels it has been trained on
- **trained_nsamples+**: Number of samples it has been trained on
- **training_confusion**: Confusion matrix of learning performance
- **training_time+**: Time (in seconds) which took classifier to train
- **values+**: Internal classifier values the most recent predictions are based on

(States enabled by default are listed with +)

**See Also:**

Please refer to the documentation of the base class for more information:

```
_GLMLNET
```

Initialize GLM-Net multinomial classifier.

See the help in R for further details on the parameters

**Parameters**

- **family** – Response type of your labels (either ‘gaussian’ for regression or ‘multinomial’ for classification). (Default: gaussian)
• **alpha** – The elastic net mixing parameter. Larger values will give rise to less L2 regularization, with alpha=1.0 as a true LASSO penalty. (Default: 1.0)

• **nlambda** – Maximum number of lambdas to calculate before stopping if not converged. (Default: 100)

• **standardize** – Whether to standardize the variables prior to fitting. (Default: True)

• **thresh** – Convergence threshold for coordinate descent. (Default: 0.0001)

• **pmax** – Limit the maximum number of variables ever to be nonzero. (Default: None)

• **maxit** – Maximum number of outer-loop iterations for ‘multinomial’ families. (Default: 100)

• **model_type** – ‘covariance’ saves all inner-products ever computed and can be much faster than ‘naive’. The latter can be more efficient for nfeatures>>nsamples situations. (Default: covariance)

• **enable_states** (*None or list of basestring*) – Names of the state variables which should be enabled additionally to default ones

• **disable_states** (*None or list of basestring*) – Names of the state variables which should be disabled

---

**GLMNET_R**

```python
class mvpa.clfs.glmnet.GLMNET_R(**kwargs)
    Bases: mvpa.clfs.glmnet._GLMNET
    GLM-NET Gaussian Regression Classifier.
    This is the GLM-NET algorithm from
    parameterized to be a regression.
    See GLMNET_C for the multinomial classifier version.
```

**Note:** Available state variables:

* **feature_ids**: Feature IDS which were used for the actual training.
* **predicting_time+**: Time (in seconds) which took classifier to predict
* **predictions+**: Most recent set of predictions
* **trained_dataset**: The dataset it has been trained on
* **trained_labels+**: Set of unique labels it has been trained on
* **trained_nsamples+**: Number of samples it has been trained on
* **training_confusion**: Confusion matrix of learning performance
* **training_time+**: Time (in seconds) which took classifier to train
* **values+**: Internal classifier values the most recent predictions are based on

(States enabled by default are listed with +)

**See Also:**

Please refer to the documentation of the base class for more information:

```
_GLGLMNET
```

Initialize GLM-Net.
See the help in R for further details on the parameters

Parameters

• **family** – Response type of your labels (either ‘gaussian’ for regression or ‘multinomial’ for classification). (Default: gaussian)

• **alpha** – The elastic net mixing parameter. Larger values will give rise to less L2 regularization, with alpha=1.0 as a true LASSO penalty. (Default: 1.0)

• **nlambda** – Maximum number of lambdas to calculate before stopping if not converged. (Default: 100)

• **standardize** – Whether to standardize the variables prior to fitting. (Default: True)

• **thresh** – Convergence threshold for coordinate descent. (Default: 0.0001)

• **pmax** – Limit the maximum number of variables ever to be nonzero. (Default: None)

• **maxit** – Maximum number of outer-loop iterations for ‘multinomial’ families. (Default: 100)

• **model_type** – ‘covariance’ saves all inner-products ever computed and can be much faster than ‘naive’. The latter can be more efficient for nfeatures>>nsamples situations. (Default: covariance)

• **regression** – Either to use ‘regression’ as regression. By default any Classifier-derived class serves as a classifier, so regression does binary classification. (Default: False)

• **enable_states** (*None or list of basestring*) – Names of the state variables which should be enabled additionally to default ones

• **disable_states** (*None or list of basestring*) – Names of the state variables which should be disabled

16.4.6 clfs.gnb

Module: clfs.gnb

Inheritance diagram for mvpa.clfs.gnb:

```
|--- misc.state.ClassWithCollections
   |--- clfs.base.Classifier
       |--- clfs.gnb.GNB
```

Gaussian Naive Bayes Classifier

EXPERIMENTAL ;) Basic implementation of Gaussian Naive Bayes classifier.
GNB

class mvpa.clfs.gnb.GNB(**kwargs)
   Bases: mvpa.clfs.base.Classifier

Gaussian Naive Bayes Classifier.

GNB is a probabilistic classifier relying on Bayes rule to estimate posterior probabilities of labels given the data. Naive assumption in it is an independence of the features, which allows to combine per-feature likelihoods by a simple product across likelihoods of "independent" features. See http://en.wikipedia.org/wiki/Naive_bayes for more information.

Provided here implementation is "naive" on its own – various aspects could be improved, but has its own advantages:

• implementation is simple and straightforward
• no data copying while considering samples of specific class
• provides alternative ways to assess prior distribution of the classes in the case of unbalanced sets of samples (see parameter prior)
• makes use of NumPy broadcasting mechanism, so should be relatively efficient
• should work for any dimensionality of samples

GNB is listed both as linear and non-linear classifier, since specifics of separating boundary depends on the data and/or parameters: linear separation is achieved whenever samples are balanced (or prior='uniform') and features have the same variance across different classes (i.e. if common_variance=True to enforce this).

Whenever decisions are made based on log-probabilities (parameter logprob=True, which is the default), then state variable values if enabled would also contain log-probabilities. Also mention that normalization by the evidence (P(data)) is disabled by default since it has no impact per se on classification decision. You might like set parameter normalize to True if you want to access properly scaled probabilities in values state variable.

---

Note: Available state variables:

• feature_ids: Feature IDS which were used for the actual training.
• predicting_time+: Time (in seconds) which took classifier to predict
• predictions+: Most recent set of predictions
• trained_dataset: The dataset it has been trained on
• trained_labels+: Set of unique labels it has been trained on
• trained_nsamples+: Number of samples it has been trained on
• training_confusion: Confusion matrix of learning performance
• training_time+: Time (in seconds) which took classifier to train
• values+: Internal classifier values the most recent predictions are based on

(States enabled by default are listed with +)

See Also:

Please refer to the documentation of the base class for more information:

Classifier

Initialize an GNB classifier.

Parameters

• common_variance – Use the same variance across all classes. (Default: False)
• **prior** – How to compute prior distribution. (Default: laplacian_smoothing)

• **logprob** – Operate on log probabilities. Preferable to avoid unneeded exponentiation and loose precision. If set, logprobs are stored in `values`. (Default: True)

• **normalize** – Normalize (log)prob by P(data). Requires probabilities thus for logprob case would require exponentiation of ‘logprob’s, thus disabled by default since does not impact classification output. . (Default: False)

• **enable_states** (*None or list of basestring*) – Names of the state variables which should be enabled additionally to default ones

• **disable_states** (*None or list of basestring*) – Names of the state variables which should be disabled

```python
means = None  # Means of features per class

priors = None  # Class probabilities

ulabels = None  # Labels classifier was trained on

untrain()  # Untrain classifier and reset all learnt params

variances = None  # Variances per class, but “vars” is taken :)
```

### 16.4.7 clfs.gpr

**Module:** clfs.gpr

Inheritance diagram for `mvpa.clfs.gpr`: 
Gaussian Process Regression (GPR).

**Classes**

**GPR**

class mvpa.clfs.gpr.GPR(kernel=None, **kwargs)

    Bases: mvpa.clfs.base.Classifier

    Gaussian Process Regression (GPR).

    **Note:** Available state variables:

    * feature_ids: Feature IDS which were used for the actual training.
    * log_marginal_likelihood: Log Marginal Likelihood
    * log_marginal_likelihood_gradient: Log Marginal Likelihood Gradient
    * predicted_variances: Variance per each predicted value
    * predicting_time+: Time (in seconds) which took classifier to predict
    * predictions+: Most recent set of predictions
    * trained_dataset: The dataset it has been trained on
    * trained_labels+: Set of unique labels it has been trained on
    * trained_nsamples+: Number of samples it has been trained on
    * training_confusion: Confusion matrix of learning performance
    * training_time+: Time (in seconds) which took classifier to train
• values+: Internal classifier values the most recent predictions are based on

(States enabled by default are listed with +)

See Also:
Please refer to the documentation of the base class for more information:

Classifier
Initialize a GPR regression analysis.

Parameters

- **kernel** (*Kernel*) – a kernel object defining the covariance between instances. (Defaults to KernelSquaredExponential if None in arguments)
- **sigma_noise** – the standard deviation of the gaussian noise. (Default: 0.001)
- **lm** – The regularization term lambda. Increase this when the kernel matrix is not positive definite. (Default: 0.0)
- **regression** – Either to use ‘regression’ as regression. By default any Classifier-derived class serves as a classifier, so regression does binary classification. (Default: False)
- **retrainable** – Either to enable retraining for ‘retrainable’ classifier. (Default: False)
- **enable_states** (*None or list of basestring*) – Names of the state variables which should be enabled additionally to default ones
- **disable_states** (*None or list of basestring*) – Names of the state variables which should be disabled

**compute_gradient_log_marginal_likelihood**()
Compute gradient of the log marginal likelihood. This version use a more compact formula provided by Williams and Rasmussen book.

**compute_gradient_log_marginal_likelihood_logscale**()
Compute gradient of the log marginal likelihood when hyperparameters are in logscale. This version use a more compact formula provided by Williams and Rasmussen book.

**compute_log_marginal_likelihood**()
Compute log marginal likelihood using self.train_fv and self.labels.

**getSensitivityAnalyzer**(*flavor='auto', **kwargs*)
Returns a sensitivity analyzer for GPR.

Parameters

- **flavor** (*basestring*) – What sensitivity to provide. Valid values are ‘linear’, ‘model_select’, ‘auto’. In case of ‘auto’ selects ‘linear’ for linear kernel and ‘model_select’ for the rest. ‘linear’ corresponds to GPRLinearWeights and ‘model_select’ to GRPWeights

**kernel**

**set_hyperparameters** (*hyperparameter*)
Set hyperparameters’ values.

Note that ‘hyperparameter’ is a sequence so the order of its values is important. First value must be sigma_noise, then other kernel’s hyperparameters values follow in the exact order the kernel expect them to be.

**untrain**()
GPRLinearWeights

class mvpa.clfs.gpr.GPRLinearWeights(clf, force_training=True, **kwargs)

Bases: mvpa.measures.base.Sensitivity

SensitivityAnalyzer that reports the weights GPR trained on a given Dataset.

In case of KernelLinear compute explicitly the coefficients of the linear regression, together with their variances (if requested).

Note that the intercept is not computed.

Note: Available state variables:

- base_sensitivities: Stores basic sensitivities if the sensitivity relies on combining multiple ones
- null_prob+: State variable
- null_t: State variable
- raw_results: Computed results before applying any transformation algorithm
- variances: Variances of the weights (for KernelLinear)

(States enabled by default are listed with +)

See Also:

Please refer to the documentation of the base class for more information:

Sensitivity

Initialize the analyzer with the classifier it shall use.

Parameters

- clf (Classifier) – classifier to use.
- force_training (Bool) – if classifier was already trained – do not retrain
- enable_states (None or list of basestring) – Names of the state variables which should be enabled additionally to default ones
- disable_states (None or list of basestring) – Names of the state variables which should be disabled
- 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.
- 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) – The estimated distribution is used to assign a probability for a certain value of the computed measure.

16.4.8 clfs.kernel

Module: clfs.kernel

Inheritance diagram for mvpa.clfs.kernel:
Kernels for Gaussian Process Regression and Classification.

**Classes**

**Kernel**

class `mvpa.clfs.kernel.Kernel`

Bases: `object`

Kernel function base class.

- `compute(data1, data2=None)`
- `compute_gradient(alphalphaTK)`
- `compute_lml_gradient(alphalphaT_Kinv, data)`
- `compute_lml_gradient_logscale(alphalphaT_Kinv, data)`
- `reset()`

  Resets the kernel dropping internal variables to the original values

**KernelConstant**

class `mvpa.clfs.kernel.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.

- Parameters

  * `data1` *(numpy.ndarray)* – data
  * `data2` *(numpy.ndarray)* – data (Defaults to None)

- `compute_lml_gradient(alphalphaT_Kinv, data)`
- `compute_lml_gradient_logscale(alphalphaT_Kinv, data)`
- `set_hyperparameters(hyperparameter)`

16.4. Classifiers and Errors
**KernelExponential**

```python
class mvpa.clfs.kernel.KernelExponential

Bases: mvpa.clfs.kernel.Kernel
```

The Exponential kernel class.

Note that it can handle a length scale for each dimension for Automatic Relevance Determination.

Initialize an Exponential kernel instance.

```
Parameters

• 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)
```

```python
compute(data1, data2=None)

Compute kernel matrix.

Parameters

• data1 (numpy.ndarray) – data
• data2 (numpy.ndarray) – data (Defaults to None)
```

```python
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.
```

```python
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.
```

```python
gradient(data1, data2)

Compute gradient of the kernel matrix. A must for fast model selection with high-dimensional data.
```

```python
set_hyperparameters(hyperparameter)

Set hyperaparameters from a vector.

Used by model selection.
```

**KernelLinear**

```python
class mvpa.clfs.kernel.KernelLinear

Bases: mvpa.clfs.kernel.Kernel
```

The linear kernel class.

Initialize the linear kernel instance.

```
Parameters

• 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)
```

```python
Sigma_p
```
compute \((data1, data2=None)\)
Compute kernel matrix. Set Sigma_p to correct dimensions and default value if necessary.

Parameters

- `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 \((hyperparameter)\)

**KernelMatern_3_2**

```python
class mvpa.clfs.kernel.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 \(n_i=3/2\) or \(n_i=5/2\).
Note that it can handle a length scale for each dimension for Automatic Relevance Determination.
Initialize a Squared Exponential kernel instance.

Parameters

- `length_scale` (float OR numpy.ndarray) – the characteristic length-scale (or lengthscales) 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 \(n_i\) 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.

Parameters

- `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 hyperparameters from a vector.
Used by model selection. Note: ‘numerator’ is not considered as an hyperparameter.

**KernelMatern_5_2**

```python
class mvpa.clfs.kernel.KernelMatern_5_2 (**kwargs)
```

Bases: mvpa.clfs.kernel.KernelMatern_3_2

The Matern kernel class for the case \(n_i=5/2\).
This kernel is just KernelMatern_3_2\((nominator=5.0)\).
Initialize a Squared Exponential kernel instance.
Parameters

**length_scale** *(float OR numpy.ndarray)* – the characteristic length-scale (or length-scales) of the phenomenon under investigation. (Defaults to 1.0)

### KernelRationalQuadratic

class `mvpa.clfs.kernel.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 Automatic Relevance Determination.

Initialize a Squared Exponential kernel instance.

Parameters

- **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.

Parameters

- **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 hyperparameters from a vector.

Used by model selection. Note: ‘alpha’ is not considered as an hyperparameter.

### KernelSquaredExponential

class `mvpa.clfs.kernel.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 Automatic Relevance Determination.

Initialize a Squared Exponential kernel instance.

Parameters

- **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.
Parameters

- **data1** (*numpy.ndarray*) – data
- **data2** (*numpy.ndarray*) – data (Defaults to None)

**compute_lml_gradient** (*alphaalphaT_Kinv, data*)

Compute gradient 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 gradient 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).

**length_scale**

**reset**()

**set_hyperparameters** (*hyperparameter*)

Set hyperparameters from a vector.

Used by model selection.

### 16.4.9 clfs.knn

**Module:** clfs.knn

Inheritance diagram for `mvpa.clfs.knn`:

```
<table>
<thead>
<tr>
<th>misc.state.ClassWithCollections</th>
</tr>
</thead>
<tbody>
<tr>
<td>clfs.base.Classifier</td>
</tr>
<tr>
<td>clfs.knn.kNN</td>
</tr>
</tbody>
</table>
```

k-Nearest-Neighbour classifier.

**kNN**

```python
class mvpa.clfs.knn.kNN(k=2, dfx=<function squared_euclidean_distance at 0x6768500>, voting='weighted', **kwargs)
```

Bases: `mvpa.clfs.base.Classifier`

k-Nearest-Neighbour classifier.

This is a simple classifier that bases its decision on the distances between the training dataset samples and the test sample(s). Distances are computed using a customizable distance function. A certain number ($k$)of
nearest neighbors is selected based on the smallest distances and the labels of this neighboring samples are fed into a voting function to determine the labels of the test sample.

Training a kNN classifier is extremely quick, as no actual training is performed as the training dataset is simply stored in the classifier. All computations are done during classifier prediction.

**Note:** If enabled, kNN stores the votes per class in the ‘values’ state after calling predict().

**Note:** Available state variables:
- feature_ids: Feature IDS which were used for the actual training.
- predicting_time+: Time (in seconds) which took classifier to predict
- predictions+: Most recent set of predictions
- trained_dataset: The dataset it has been trained on
- trained_labels+: Set of unique labels it has been trained on
- trained_nsamples+: Number of samples it has been trained on
- training_confusion: Confusion matrix of learning performance
- training_time+: Time (in seconds) which took classifier to train
- values+: Internal classifier values the most recent predictions are based on

(States enabled by default are listed with +)

**See Also:**
Please refer to the documentation of the base class for more information:

**Classifier**

Parameters

- **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).
- **enable_states** (*None or list of basestring*) – Names of the state variables which should be enabled additionally to default ones
- **disable_states** (*None or list of basestring*) – Names of the state variables which should be disabled

**getMajorityVote** (*knn_ids*)
Simple voting by choosing the majority of class neighbors.

**getWeightedVote** (*knn_ids*)
Vote with classes weighted by the number of samples per class.

**untrain**()
Reset trained state
16.4.10 clfs.lars

Module: clfs.lars

Inheritance diagram for mvpa.clfs.lars:

Least angle regression (LARS) classifier.

Classes

LARS

```python
class mvpa.clfs.lars.LARS(model_type='lasso', trace=False, normalize=True, intercept=True,
max_steps=None, use_Gram=False, **kwargs)
```

Bases: mvpa.clfs.base.Classifier

Least angle regression (LARS) Classifier.

LARS is the model selection algorithm from:


Similar to SMLR, it performs a feature selection while performing classification, but instead of starting with all features, it starts with none and adds them in, which is similar to boosting.

This classifier behaves more like a ridge regression in that it returns prediction values and it treats the training labels as continuous.

In the true nature of the PyMVPA framework, this algorithm is actually implemented in R by Trevor Hastie and wrapped via RPy. To make use of LARS, you must have R and RPy installed as well as the LARS contributed package. You can install the R and RPy with the following command on Debian-based machines:
sudo aptitude install python-rpy python-rpy-doc r-base-dev

You can then install the LARS package by running R as root and calling:
install.packages()

Note: Available state variables:

• **feature_ids**: Feature IDs which were used for the actual training.
• **predicting_time+**: Time (in seconds) which took classifier to predict
• **predictions+**: Most recent set of predictions
• **trained_dataset**: The dataset it has been trained on
• **trained_labels+**: Set of unique labels it has been trained on
• **trained_nsamples+**: Number of samples it has been trained on
• **training_confusion**: Confusion matrix of learning performance
• **training_time+**: Time (in seconds) which took classifier to train
• **values+**: Internal classifier values the most recent predictions are based on

(States enabled by default are listed with +)

See Also:
Please refer to the documentation of the base class for more information:

**Classifier**

Initialize LARS.

See the help in R for further details on the following parameters:

Parameters

• **model_type** (string) – Type of LARS to run. Can be one of ('lasso', 'lar', 'forward.stagewise', 'stepwise').
• **trace** (boolean) – Whether to print progress in R as it works.
• **normalize** (boolean) – Whether to normalize the L2 Norm.
• **intercept** (boolean) – Whether to add a non-penalized intercept to the model.
• **max_steps** (None or int) – If not None, specify the total number of iterations to run. Each iteration adds a feature, but leaving it none will add until convergence.
• **use_Gram** (boolean) – Whether to compute the Gram matrix (this should be false if you have more features than samples.)
• **regression** – Either to use ‘regression’ as regression. By default any Classifier-derived class serves as a classifier, so regression does binary classification. (Default: False)
• **enable_states** (None or list of basestring) – Names of the state variables which should be enabled additionally to default ones
• **disable_states** (None or list of basestring) – Names of the state variables which should be disabled

getSensitivityAnalyzer(**kwargs)

Returns a sensitivity analyzer for LARS.

weights
class mvpa.clfs.lars.LARSWeights(clf, force_training=True, **kwargs)

Bases: mvpa.measures.base.Sensitivity

SensitivityAnalyzer that reports the weights LARS trained on a given Dataset.

Note: Available state variables:

• base_sensitivities: Stores basic sensitivities if the sensitivity relies on combining multiple ones
• null_prob+: State variable
• null_v: State variable
• raw_results: Computed results before applying any transformation algorithm

(States enabled by default are listed with +)

See Also:
Please refer to the documentation of the base class for more information:

Sensitivity

Initialize the analyzer with the classifier it shall use.

Parameters

• clf (Classifier) – classifier to use.
• force_training (Bool) – if classifier was already trained – do not retrain
• enable_states (None or list of basestring) – Names of the state variables which should be enabled additionally to default ones
• disable_states (None or list of basestring) – Names of the state variables which should be disabled
• 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.
• 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) – The estimated distribution is used to assign a probability for a certain value of the computed measure.

16.4.11 clfs.libsmmlrc

Module: clfs.libsmmlrc

Wraper for the stepwise_regression function for SMLR.

canvas.png

mvpa.clfs.libsmmlrc.stepwise_regression(*args)

16.4.12 clfs.libsmmlrc.ctypes_helper

Module: clfs.libsmmlrc.ctypes_helper

Helpers for wrapping C libraries with ctypes.
Functions

mvpa.clfs.libsmlrc.ctypes_helper.extend_args(*args)
   Turn ndarray arguments into dims and arrays.
mvpa.clfs.libsmlrc.ctypes_helper.get_argtypes(*args)
mvpa.clfs.libsmlrc.ctypes_helper.process_args(*args)
   Turn ndarray arguments into dims and array pointers for calling a ctypes-wrapped function.

16.4.13 clfs.libsvmc.sens

Module: clfs.libsvmc.sens

Inheritance diagram for mvpa.clfs.libsvmc.sens:

```
+-------------------+
| misc.state.ClassWithCollections |
|                    +-------------------+-------------------+
| measures.base.DatasetMeasure        | measures.base.DatasetMeasure |
|                                        +-------------------+-------------------+-------------------+
| measures.base.FeaturewiseDatasetMeasure                      | measures.base.Sensitivity |
|                                                            +-------------------+-------------------+-------------------+
| libsvmc.sens.LinearSVMWeights |
```

Provide sensitivity measures for libsvm’s SVM.

**LinearSVMWeights**

class mvpa.clfs.libsvmc.sens.LinearSVMWeights(clf, **kwargs)
   Bases: mvpa.measures.base.Sensitivity

SensitivityAnalyzer for the LIBSVM implementation of a linear SVM.

**Note:** Available state variables:

- **base_sensitivities**: Stores basic sensitivities if the sensitivity relies on combining multiple ones
- **biases+**: Offsets of separating hyperplanes
- **null_prob+**: State variable
• *null_t*: State variable
• *raw_results*: Computed results before applying any transformation algorithm

(States enabled by default are listed with +)

See Also:

Please refer to the documentation of the base class for more information:

Sensitivity

Initialize the analyzer with the classifier it shall use.

Parameters

• **clf** (*LinearSVM*) – classifier to use. Only classifiers sub-classed from *LinearSVM* may be used.

• **split_weights** – If binary classification either to sum SVs per each class separately. (Default: False)

• **enable_states** (*None or list of basestring*) – Names of the state variables which should be enabled additionally to default ones

• **disable_states** (*None or list of basestring*) – Names of the state variables which should be disabled

• **force_training** (*Bool*) – if classifier was already trained – do not retrain

• **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.

• **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*) – The estimated distribution is used to assign a probability for a certain value of the computed measure.

16.4.14 clfs.libsvmc.svm

**Module:** clfs.libsvmc.svm

Inheritance diagram for mvpa.clfs.libsvmc.svm:
Wrap the libsvm package into a very simple class interface.

**SVM**

```python
class mvpa.clfs.libsvmc.svm.SVM(kernel_type='linear', **kwargs):
    Bases: mvpa.clfs._svmbase._SVM

    Support Vector Machine Classifier.
    This is a simple interface to the libSVM package.
```

**Note:** Available state variables:

- `feature_ids`: Feature IDS which were used for the actual training.
- `predicting_time+`: Time (in seconds) which took classifier to predict
- `predictions+`: Most recent set of predictions
- `probabilities`: Estimates of samples probabilities as provided by LibSVM
- `trained_dataset`: The dataset it has been trained on
- `trained_labels+`: Set of unique labels it has been trained on
- `trained_nsamples+`: Number of samples it has been trained on
- `training_confusion`: Confusion matrix of learning performance
- `training_time+`: Time (in seconds) which took classifier to train
- `values+`: Internal classifier values the most recent predictions are based on

(States enabled by default are listed with +)

**See Also:**

Please refer to the documentation of the base class for more information:

_SVM

Interface class to LIBSVM classifiers and regressions.
Default implementation (C/nu/epsilon SVM) is chosen depending on the given parameters (C/nu/tube_epsilon).

SVM/SVR definition is dependent on specifying kernel, implementation type, and parameters for each of them which vary depending on the choices made.

Desired implementation is specified in `svm_impl` argument. Here is the list if implementations known to this class, along with specific to them parameters (described below among the rest of parameters), and what tasks it is capable to deal with (e.g. regression, binary and/or multiclass classification).

Implementations

- **ONE_CLASS**
  - [one-class-SVM] Capabilities: oneclass

- **C_SVC**
  - [C-SVM classification] Parameters: C
  - Capabilities: binary, multiclass

- **NU_SVR**
  - [nu-SVM regression] Parameters: nu, tube_epsilon
  - Capabilities: regression

- **NU_SVC**
  - [nu-SVM classification] Parameters: nu
  - Capabilities: binary, multiclass

- **EPSILON_SVR**
  - [epsilon-SVM regression] Parameters: C, tube_epsilon
  - Capabilities: regression

Kernel choice is specified as a string argument `kernel_type` and it can be specialized with additional arguments to this constructor function. Some kernels might allow computation of per feature sensitivity.

Kernels

- rbf gamma
- linear
  - [provides sensitivity] No parameters
- poly
  - coef0, degree, gamma
- sigmoid
  - coef0, gamma

Parameters

- **tube_epsilon** – Epsilon in epsilon-insensitive loss function of epsilon-SVM regression (SVR). (Default: 0.01)
- **C** – Trade-off parameter between width of the margin and number of support vectors. Higher C – more rigid margin SVM. In linear kernel, negative values provide automatic scaling of their value according to the norm of the data. (Default: -1.0)
- **probability** – Flag to signal either probability estimate is obtained within LIBSVM. (Default: 0)
- **degree** – Degree of polynomial kernel. (Default: 3)
- **shrinking** – Either shrinking is to be conducted. (Default: 1)
• **weight_label** – To be used in conjunction with weight for custom per-label weight. (Default: [])
• **weight** – Custom weights per label. (Default: [])
• **epsilon** – Tolerance of termination criteria. (For nu-SVM default is 0.001). (Default: 5e-05)
• **cache_size** – Size of the kernel cache, specified in megabytes. (Default: 100)
• **coef0** – Offset coefficient in polynomial and sigmoid kernels. (Default: 0.5)
• **nu** – Fraction of datapoints within the margin. (Default: 0.5)
• **gamma** – Scaling (width in RBF) within non-linear kernels. (Default: 0)
• **enable_states** (*None or list of basestring*) – Names of the state variables which should be enabled additionally to default ones
• **disable_states** (*None or list of basestring*) – Names of the state variables which should be disabled
• **kernel_type** (*basestr*) – String must be a valid key for cls._KERNELS

```python
def model
    Access to the SVM model.

def summary()
    Provide quick summary over the SVM classifier

def untrain()
    Untrain libsvm’s SVM: forget the model
```

## 16.4.15 clfs.libsvmc.svmc

**Module:** clfs.libsvmc.svmc

Inheritance diagram for mvpa.clfs.libsvmc.svmc:

```
libsvmc.svmc.svm_problem          libsvmc.svmc.svm_model          libsvmc.svmc.svm_parameter
```

### Classes

```python
class svm_model
    Bases: object
```

```python
    SV
    free_sv
    l
    label
    nSV
    nr_class
```
param
probA
probB
rho
sv_coef

**svm_parameter**

```python
class mvpa.clfs.libsvm.svmc.svm_parameter
    Bases: object
    C
cache_size
coeff0
degree
eps
gamma
kernel_type
nr_weight
nu
P
probability
shrinking
svm_type
weight
weight_label
```

**svm_problem**

```python
class mvpa.clfs.libsvm.svmc.svm_problem
    Bases: object
    l
    x
    y
```

**Functions**

```python
mvpa.clfs.libsvm.svmc.delete_double()
mvpa.clfs.libsvm.svmc.delete_int()
mvpa.clfs.libsvm.svmc.double_getitem()
mvpa.clfs.libsvm.svmc.double_setitem()
mvpa.clfs.libsvm.svmc.doubleppcarray2numpy_array()
mvpa.clfs.libsvm.svmc.int_getitem()
```
Classes for meta classifiers – classifiers which use other classifiers

Meta Classifiers can be grouped according to their function as
group BoostedClassifiers
   CombinedClassifier MulticlassClassifier SplitClassifier

group ProxyClassifiers
   ProxyClassifier BinaryClassifier MappedClassifier FeatureSelectionClassifier

group PredictionsCombiners for CombinedClassifier
   PredictionsCombiner MaximalVote MeanPrediction

**Classes**

**BinaryClassifier**

class :class:`mvpa.clfs.meta.BinaryClassifier`
   (clf, poslabels, neglabels, **kwargs)
   Bases: mvpa.clfs.meta.ProxyClassifier

   `ProxyClassifier` which maps set of two labels into +1 and -1

   **Note:** Available state variables:
   
   * `feature_ids`: Feature IDS which were used for the actual training.
   * `predicting_time+`: Time (in seconds) which took classifier to predict
   * `predictions+`: Most recent set of predictions
   * `trained_dataset`: The dataset it has been trained on
   * `trained_labels+`: Set of unique labels it has been trained on
   * `trained_nsamples+`: Number of samples it has been trained on
   * `training_confusion`: Confusion matrix of learning performance
   * `training_time+`: Time (in seconds) which took classifier to train
   * `values+`: Internal classifier values the most recent predictions are based on

   (States enabled by default are listed with +)

   **See Also:**

   Please refer to the documentation of the base class for more information:

   `ProxyClassifier`

   **Parameters**

   * `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
   * `enable_states` *(None or list of basestring)* – Names of the state variables which should be enabled additionally to default ones
   * `disable_states` *(None or list of basestring)* – Names of the state variables which should be disabled

---

**16.4. Classifiers and Errors**

195
**BoostedClassifier**

```python
class mvpa.clfs.meta.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

**Note:** Available state variables:

- `feature_ids`: Feature IDS which were used for the actual training.
- `harvested`: Store specified attributes of classifiers at each split
- `predicting_time+`: Time (in seconds) which took classifier to predict
- `predictions+`: Most recent set of predictions
- `raw_predictions`: Predictions obtained from each classifier
- `raw_values`: Values obtained from each classifier
- `trained_dataset`: The dataset it has been trained on
- `trained_labels+`: Set of unique labels it has been trained on
- `trained_nsamples+`: Number of samples it has been trained on
- `training_confusion`: Confusion matrix of learning performance
- `training_time+`: Time (in seconds) which took classifier to train
- `values+`: Internal classifier values the most recent predictions are based on

(States enabled by default are listed with +)

**See Also:**

Please refer to the documentation of the base classes for more information:

`Classifier`, `Harvestable`

Initialize the instance.

Parameters

- `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
- `enable_states (None or list of basestring)` – Names of the state variables which should be enabled additionally to default ones
- `disable_states (None or list of basestring)` – Names of the state variables which should be disabled
- `harvest_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
clfs
Used classifiers

getSensitivityAnalyzer(**kwargs)
Return an appropriate SensitivityAnalyzer

untrain()
Untrain BoostedClassifier
Has to untrain any known classifier

ClassifierCombiner

class mvpa.clfs.meta.ClassifierCombiner(clf, variables=None)
Bases: mvpa.clfs.meta.PredictionsCombiner
Provides a decision using training a classifier on predictions/values
TODO: implement

Note: Available state variables:
* predictions+: Trained predictions

(States enabled by default are listed with +)

See Also:
Please refer to the documentation of the base class for more information:
PredictionsCombiner

Initialize ClassifierCombiner

Parameters

• 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
• enable_states (None or list of basestring) – Names of the state variables which should be enabled additionally to default ones
• disable_states (None or list of basestring) – Names of the state variables which should be disabled

untrain()
It might be needed to untrain used classifier

CombinedClassifier

class mvpa.clfs.meta.CombinedClassifier(clfs=None, combiner=None, **kwargs)
Bases: mvpa.clfs.meta.BoostedClassifier
BoostedClassifier which combines predictions using some PredictionsCombiner functor.

Note: Available state variables:
* feature_ids: Feature IDS which were used for the actual training.
* harvested: Store specified attributes of classifiers at each split
* predicting_time+: Time (in seconds) which took classifier to predict
• _predictions+: Most recent set of predictions
• _raw_predictions_: Predictions obtained from each classifier
• _raw_values_: Values obtained from each classifier
• _trained_dataset_: The dataset it has been trained on
• _trained_labels+: Set of unique labels it has been trained on
• _trained_nsamples+: Number of samples it has been trained on
• _training_confusion_: Confusion matrix of learning performance
• _training_time+: Time (in seconds) which took classifier to train
• _values+: Internal classifier values the most recent predictions are based on

(States enabled by default are listed with +)

See Also:
Please refer to the documentation of the base class for more information:

**BoostedClassifier**

Initialize the instance.

Parameters

• _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)
• _enable_states_ (None or list of basestring) – Names of the state variables which should be enabled additionally to default ones
• _disable_states_ (None or list of basestring) – Names of the state variables which should be disabled
• _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 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

NB: _combiner_ might need to operate not on ‘predictions’ discrete 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()**

Provide summary for the CombinedClassifier.

**untrain()**

Untrain CombinedClassifier
FeatureSelectionClassifier

**class** `mvpa.clfs.meta.FeatureSelectionClassifier(clf, feature_selection, test_dataset=None, **kwargs)`

**Bases:** `mvpa.clfs.meta.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 explicitly use isTrained here if we want... need to think more

---

**Note:** Available state variables:

- *feature_ids*: Feature IDS which were used for the actual training.
- *predicting_time+*: Time (in seconds) which took classifier to predict
- *predictions+*: Most recent set of predictions
- *trained_dataset*: The dataset it has been trained on
- *trained_labels+*: Set of unique labels it has been trained on
- *trained_nsamples+*: Number of samples it has been trained on
- *training_confusion*: Confusion matrix of learning performance
- *training_time+*: Time (in seconds) which took classifier to train
- *values+*: Internal classifier values the most recent predictions are based on

(States enabled by default are listed with +)

**See Also:**

Please refer to the documentation of the base class for more information:

*ProxyClassifier*

Initialize the instance

**Parameters**

- **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
- **enable_states** (*None or list of basestring*) – Names of the state variables which should be enabled additionally to default ones
- **disable_states** (*None or list of basestring*) – Names of the state variables which should be disabled

**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

MappedClassifier

class mvpa.clfs.meta.MappedClassifier(clf, mapper, **kwargs)
    Bases: mvpa.clfs.meta.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.

    Note: Available state variables:
    • feature_ids: Feature IDS which were used for the actual training.
    • predicting_time+: Time (in seconds) which took classifier to predict
    • predictions+: Most recent set of predictions
    • trained_dataset: The dataset it has been trained on
    • trained_labels+: Set of unique labels it has been trained on
    • trained_nsamples+: Number of samples it has been trained on
    • training_confusion: Confusion matrix of learning performance
    • training_time+: Time (in seconds) which took classifier to train
    • values+: Internal classifier values the most recent predictions are based on

    (States enabled by default are listed with +)

    See Also:
    Please refer to the documentation of the base class for more information:

    ProxyClassifier
    Initialize the instance

    Parameters

    • clf (Classifier) – classifier based on which mask classifiers is created
    • mapper – whatever Mapper comes handy
    • enable_states (None or list of basestring) – Names of the state variables which should be enabled additionally to default ones
    • disable_states (None or list of basestring) – Names of the state variables which should be disabled

    getSensitivityAnalyzer (*args_, **kwargs_)
mapper

Used mapper

MaximalVote

class mvpa.clfs.meta.MaximalVote
    Bases: mvpa.clfs.meta.PredictionsCombiner
    Provides a decision using maximal vote rule

    Note: Available state variables:
        • *all_label_counts*: Counts across classifiers for each label/sample
        • *predictions*: Voted predictions

    (States enabled by default are listed with +)

    See Also:
    Please refer to the documentation of the base class for more information:
    PredictionsCombiner

XXX Might get a parameter to use raw decision values if voting is not unambiguous (ie two classes have equal number of votes

Parameters

*enable_states* (*None or list of basestring*) – Names of the state variables which should be enabled additionally to default ones

*disable_states* (*None or list of basestring*) – Names of the state variables which should be disabled

MeanPrediction

class mvpa.clfs.meta.MeanPrediction (descr=None, **kwargs)
    Bases: mvpa.clfs.meta.PredictionsCombiner
    Provides a decision by taking mean of the results

    Note: Available state variables:
        • *predictions*: Mean predictions

    (States enabled by default are listed with +)

    See Also:
    Please refer to the documentation of the base class for more information:
    PredictionsCombiner

MulticlassClassifier

class mvpa.clfs.meta.MulticlassClassifier (clf, bclf_type='1-vs-1', **kwargs)
    Bases: mvpa.clfs.meta.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)
Note: Available state variables:

- **feature_ids**: Feature IDS which were used for the actual training.
- **harvested**: Store specified attributes of classifiers at each split
- **predicting_time+**: Time (in seconds) which took classifier to predict
- **predictions+**: Most recent set of predictions
- **raw_predictions**: Predictions obtained from each classifier
- **raw_values**: Values obtained from each classifier
- **trained_dataset**: The dataset it has been trained on
- **trained_labels+**: Set of unique labels it has been trained on
- **trained_nsamples+**: Number of samples it has been trained on
- **training_confusion**: Confusion matrix of learning performance
- **training_time+**: Time (in seconds) which took classifier to train
- **values+**: Internal classifier values the most recent predictions are based on

(States enabled by default are listed with +)

See Also:
Please refer to the documentation of the base class for more information:

**CombinedClassifier**

Initialize the instance

Parameters

- **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
- **enable_states** (None or list of basestring) – Names of the state variables which should be enabled additionally to default ones
- **disable_states** (None or list of basestring) – Names of the state variables which should be disabled
- **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))
- **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 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
PredictionsCombiner

class mvpa.clfs.meta.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

Parameters

*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

ProxyClassifier

class mvpa.clfs.meta.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?

Note: Available state variables:

*feature_ids: Feature IDS which were used for the actual training.
*predicting_time+: Time (in seconds) which took classifier to predict
*predictions+: Most recent set of predictions
*trained_dataset: The dataset it has been trained on
*trained_labels+: Set of unique labels it has been trained on
*trained_nsamples+: Number of samples it has been trained on
*training_confusion: Confusion matrix of learning performance
*training_time+: Time (in seconds) which took classifier to train
*values+: Internal classifier values the most recent predictions are based on

(States enabled by default are listed with +)

See Also:
Please refer to the documentation of the base class for more information:

Classifier
Initialize the instance

Parameters

*clf (Classifier) – classifier based on which mask classifiers is created
*enable_states (None or list of basestring) – Names of the state variables which should be enabled additionally to default ones
• **disable_states** *(None or list of basestring)* – Names of the state variables which should be disabled

```python
clf
Used Classifier
getSensitivityAnalyzer(*args_, **kwargs_)
summary()
untrain()
Untrain ProxyClassifier
```

### SplitClassifier

class `mvpa.clfs.meta.SplitClassifier` *(clf, splitter=<mvpa.datasets.splitters.NFoldSplitter object at 0x4869450>, **kwargs)*

**Bases:** `mvpa.clfs.meta.CombinedClassifier`

**BoostedClassifier** to work on splits of the data

**Note:** Available state variables:

- **confusion:** Resultant confusion whenever classifier trained on 1 part and tested on 2nd part of each split
- **feature_ids:** Feature IDS which were used for the actual training.
- **harvested:** Store specified attributes of classifiers at each split
- **predicting_time+:** Time (in seconds) which took classifier to predict
- **predictions+:** Most recent set of predictions
- **raw_predictions:** Predictions obtained from each classifier
- **raw_values:** Values obtained from each classifier
- **splits:** Store the actual splits of the data. Can be memory expensive
- **trained_dataset:** The dataset it has been trained on
- **trained_labels+:** Set of unique labels it has been trained on
- **trained_nsamples+:** Number of samples it has been trained on
- **training_confusion:** Confusion matrix of learning performance
- **training_time+:** Time (in seconds) which took classifier to train
- **values+:** Internal classifier values the most recent predictions are based on

(States enabled by default are listed with +)

**See Also:**

Please refer to the documentation of the base class for more information:

`CombinedClassifier`

Initialize the instance

**Parameters**

- **clf** *(Classifier)* – classifier based on which multiple classifiers are created for multiclass
- **splitter** *(Splitter)* – *Splitter* to use to split the dataset prior training
- **enable_states** (*None or list of basestring*) – Names of the state variables which should be enabled additionally to default ones
- **disable_states** (*None or list of basestring*) – Names of the state variables which should be disabled
- **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)
- **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 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

`getSensitivityAnalyzer(*args_**, **kwargs_)`

**splitter**
  Splitter user by SplitClassifier

**TreeClassifier**

class `mvpa.clfs.meta.TreeClassifier(clf, groups, **kwargs)`

Bases: `mvpa.clfs.meta.ProxyClassifier`

*TreeClassifier* which allows to create hierarchy of classifiers

Functions by grouping some labels into a single “meta-label” and training classifier first to separate between meta-labels. Then each group further proceeds with classification within each group.

Possible scenarios:

```
TreeClassifier(SVM(),
{'animate': ((1,2,3,4),
  TreeClassifier(SVM(),
    {'human': (('male', 'female'), SVM()),
     'animals': (((monkey', 'dog'), SMLR())))),
  'inanimate': ((5,6,7,8), SMLR()))})
```

would create classifier which would first do binary classification to separate animate from inanimate, then for animate result it would separate to classify human vs animal and so on:

```
SVM / animate inanimate
 /
 / SVM SMLR
 | human animal 5 6 7 8
| | | | | |
SVM SVM
/ / male female monkey dog
1 2 3 4
```

If it is desired to have a trailing node with a single label and thus without any classification, such as in

```
SVM
/ g1 g2
```

16.4. Classifiers and Errors
then just specify None as the classifier to use:

```
TreeClassifier(SVM(),
    {'g1': ((1,), None),
     'g2': ((1, 2, 3, 4), SVM())})
```

**Note:** Available state variables:

- `feature_ids`: Feature IDS which were used for the actual training.
- `predicting_time+`: Time (in seconds) which took classifier to predict
- `predictions+`: Most recent set of predictions
- `trained_dataset`: The dataset it has been trained on
- `trained_labels+`: Set of unique labels it has been trained on
- `trained_nsamples+`: Number of samples it has been trained on
- `training_confusion`: Confusion matrix of learning performance
- `training_time+`: Time (in seconds) which took classifier to train
- `values+`: Internal classifier values the most recent predictions are based on

(States enabled by default are listed with +)

**See Also:**

Please refer to the documentation of the base class for more information:

- `ProxyClassifier`

Initialize TreeClassifier

Parameters

- `clf (Classifier)` – Classifier to separate between the groups
- `groups (dict of meta-label: tuple of (tuple of labels, classifier))` – Defines the groups of labels and their classifiers. See `TreeClassifier` for example
- `enable_states (None or list of basestring)` – Names of the state variables which should be enabled additionally to default ones
- `disable_states (None or list of basestring)` – Names of the state variables which should be disabled

```python
clfs = None
```

Dictionary of classifiers used by the groups

```python
summary()
```

Provide summary for the `TreeClassifier`.

```python
untrain()
```

Untrain `TreeClassifier`

### 16.4.17 `clfs.model_selector`

**Module:** `clfs.model_selector`

Inheritance diagram for `mvpa.clfs.model_selector:`
Model selection.

**ModelSelector**

class `mvpa.clfs.model_selector.ModelSelector`(*parametric_model, dataset*)

Bases: `object`

Model selection facility.

Select a model among multiple models (i.e., a parametric model, parametrized by a set of hyperparameters).

max_log_marginal_likelihood(hyp_initial_guess, maxiter=1, optimization_algorithm='scipy_cg', ftol=0.001, fixedHypers=None, use_gradient=False, logscale=False)

Set up the optimization problem in order to maximize the log_marginal_likelihood.

Parameters

*parametric_model* (*Classifier*) – the actual parameteric model to be optimized.

*hyp_initial_guess* (*ndarray*) – set of hyperparameters’ initial values where to start optimization.

*optimization_algorithm* (*string*) – actual name of the optimization algorithm. See http://scipy.org/scipy/scikits/wiki/NLP for a comprehensive/updated list of available NLP solvers. (Defaults to ‘ralg’)

*ftol* (*float*) – threshold for the stopping criterion of the solver, which is mapped in OpenOpt NLP.ftol (Defaults to 1.0e-3)

*fixedHypers* (*ndarray (boolean array]*) – boolean vector of the same size of hyp_initial_guess; ‘False’ means that the corresponding hyperparameter must be kept fixed (so not optimized). (Defaults to None, which during means all True)

NOTE: the maximization of log_marginal_likelihood is a non-linear optimization problem (NLP). This fact is confirmed by Dmitrey, author of OpenOpt.

solve(problem=) solve the maximization problem, check outcome and collect results.

16.4.18 clfs.plr

Module: clfs.plr

Inheritance diagram for `mvpa.clfs.plr`: 

16.4. Classifiers and Errors 207
Penalized logistic regression classifier.

**PLR**

class `mvpa.clfs.plr.PLR`(lm=1, criterion=1, reduced=0.0, maxiter=20, **kwargs)

Bases: `mvpa.clfs.base.Classifier`

Penalized logistic regression `Classifier`.

Note: Available state variables:

- `feature_ids`: Feature IDS which were used for the actual training.
- `predicting_time+`: Time (in seconds) which took classifier to predict
- `predictions+`: Most recent set of predictions
- `trained_dataset`: The dataset it has been trained on
- `trained_labels+`: Set of unique labels it has been trained on
- `trained_nsamples+`: Number of samples it has been trained on
- `training_confusion`: Confusion matrix of learning performance
- `training_time+`: Time (in seconds) which took classifier to train
- `values+`: Internal classifier values the most recent predictions are based on

(States enabled by default are listed with +)

See Also:

Please refer to the documentation of the base class for more information:

`Classifier`

Initialize a penalized logistic regression analysis

Parameters

- `lm (int)` – the penalty term lambda.
- `criterion (int)` – the criterion applied to judge convergence.
• **reduced** (*float*) – if not 0, 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.

• **enable_states** (*None or list of basestring*) – Names of the state variables which should be enabled additionally to default ones

• **disable_states** (*None or list of basestring*) – Names of the state variables which should be disabled

### 16.4.19 clfs.ridge

**Module:** clfs.ridge

Inheritance diagram for mvpa.clfs.ridge:

```
          misc.state.ClassWithCollections
             ↓
        clfs.base.Classifier
             ↓
   clfs.ridge.RidgeReg
```

Ridge regression classifier.

**RidgeReg**

```python
class mvpa.clfs.ridge.RidgeReg(lm=None, **kwargs)
    Bases: mvpa.clfs.base.Classifier
```

Ridge regression Classifier.

This ridge regression adds an intercept term so your labels do not have to be zero-centered.

**Note:** Available state variables:

• **feature_ids**: Feature IDS which were used for the actual training.

• **predicting_time+**: Time (in seconds) which took classifier to predict

• **predictions+**: Most recent set of predictions

• **trained_dataset**: The dataset it has been trained on

• **trained_labels+**: Set of unique labels it has been trained on

• **trained_nsamples+**: Number of samples it has been trained on
• *training_confusion*: Confusion matrix of learning performance
• *training_time*: Time (in seconds) which took classifier to train
• *values*: Internal classifier values the most recent predictions are based on

(States enabled by default are listed with +)

**See Also:**

Please refer to the documentation of the base class for more information:

**Classifier**

Initialize a ridge regression analysis.

**Parameters**

• **lm** (*float*) – the penalty term lambda. (Defaults to .05*nFeatures)
• **regression** – Either to use ‘regression’ as regression. By default any Classifier-derived class serves as a classifier, so regression does binary classification. (Default: False)
• **enable_states** (*None or list of basestring*) – Names of the state variables which should be enabled additionally to default ones
• **disable_states** (*None or list of basestring*) – Names of the state variables which should be disabled

### 16.4.20 clfs.sg.sens

**Module:** clfs.sg.sens

Inheritance diagram for `mvpa.clfs.sg.sens`:
Provide sensitivity measures for sg’s SVM.

**LinearSVMWeights**

```python
class mvpa.clfs.sg.sens.LinearSVMWeights(clf, **kwargs)

Bases: mvpa.measures.base.Sensitivity

Sensitivity that reports the weights of a linear SVM trained on a given Dataset.
```

**Note:** Available state variables:

- *base_sensitivities*: Stores basic sensitivities if the sensitivity relies on combining multiple ones
- *biases*: Offsets of separating hyperplanes
- *null_prob*: State variable
- *null_f*: State variable
- *raw_results*: Computed results before applying any transformation algorithm

(States enabled by default are listed with +)

**See Also:**

Please refer to the documentation of the base class for more information:

**Sensitivity**

Initialize the analyzer with the classifier it shall use.

**Parameters**
• **clf** (*LinearSVM*) – classifier to use. Only classifiers sub-classed from *LinearSVM* may be used.

• **enable_states** (*None or list of basestring*) – Names of the state variables which should be enabled additionally to default ones.

• **disable_states** (*None or list of basestring*) – Names of the state variables which should be disabled.

• **force_training** (*Bool*) – if classifier was already trained – do not retrain.

• **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.

• **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*) – The estimated distribution is used to assign a probability for a certain value of the computed measure.

### 16.4.21 clfs.sg.svm

**Module:** clfs.sg.svm

Inheritance diagram for `mvpa.clfs.sg.svm`:

```
misc.state.ClassWithCollections
    ↓
clfs.base.Classifier
    ↓
clfs._svmbase._SVM
    ↓
sg.svm.SVM
```

Wrap the libsvm package into a very simple class interface.

**SVM**

```python
class mvpa.clfs.sg.svm.SVM(kernel_type='linear', **kwargs):
    Bases: mvpa.clfs._svmbase._SVM
    Support Vector Machine Classifier(s) based on Shogun
    This is a simple base interface
```

212 Chapter 16. Module Reference
Note: Available state variables:

• *feature_ids*: Feature IDS which were used for the actual training.
• *predicting_time*: Time (in seconds) which took classifier to predict
• *predictions*: Most recent set of predictions
• *trained_dataset*: The dataset it has been trained on
• *trained_labels*: Set of unique labels it has been trained on
• *trained_nsamples*: Number of samples it has been trained on
• *training_confusion*: Confusion matrix of learning performance
• *training_time*: Time (in seconds) which took classifier to train
• *values*: Internal classifier values the most recent predictions are based on

(States enabled by default are listed with +)

See Also:

Please refer to the documentation of the base class for more information:

_SVM_

Interface class to Shogun’s classifiers and regressions.

Default implementation is ‘libsvm’.

SVM/SVR definition is dependent on specifying kernel, implementation type, and parameters for each of them which vary depending on the choices made.

Desired implementation is specified in _svm_impl_ argument. Here is the list if implementations known to this class, along with specific to them parameters (described below among the rest of parameters), and what tasks it is capable to deal with (e.g. regression, binary and/or multiclass classification).

Implementations

libsvr
  [LIBSVM’s epsilon-SVR] Parameters: C, tube_epsilon
  Capabilities: regression

gnpp
  [Generalized Nearest Point Problem SVM] Parameters: C
  Capabilities: binary

libsvm
  [LIBSVM’s C-SVM (L2 soft-margin SVM)] Parameters: C
  Capabilities: binary, multiclass

gmnp
  [Generalized Nearest Point Problem SVM] Parameters: C
  Capabilities: binary, multiclass

gpbt
  [Gradient Projection Decomposition Technique for large-scale SVM problems] Parameters: C
  Capabilities: binary

Kernel choice is specified as a string argument _kernel_type_ and it can be specialized with additional arguments to this constructor function. Some kernels might allow computation of per feature sensitivity.
Kernels

rbf gamma
rbfshift
  gamma, max_shift, shift_step
linear
  [provides sensitivity] No parameters
sigmoid
  cache_size, coef0, gamma

Parameters

• **tube_epsilon** – Epsilon in epsilon-insensitive loss function of epsilon-SVM regression (SVR). (Default: 0.01)
• **C** – Trade-off parameter between width of the margin and number of support vectors. Higher C – more rigid margin SVM. In linear kernel, negative values provide automatic scaling of their value according to the norm of the data. (Default: -1.0)
• **shift_step** – Shift step for SGs GaussianShiftKernel. (Default: 1)
• **max_shift** – Maximal shift for SGs GaussianShiftKernel. (Default: 10)
• **epsilon** – Tolerance of termination criteria. (For nu-SVM default is 0.001). (Default: 5e-05)
• **cache_size** – Size of the kernel cache, specified in megabytes. (Default: 100)
• **coef0** – Offset coefficient in polynomial and sigmoid kernels. (Default: 0.5)
• **gamma** – Scaling (width in RBF) within non-linear kernels. (Default: 0)
• **num_threads** – Number of threads to utilize. (Default: 1)
• **retrainable** – Either to enable retraining for ‘retrainable’ classifier. (Default: False)
• **enable_states** (*None or list of basestring*) – Names of the state variables which should be enabled additionally to default ones
• **disable_states** (*None or list of basestring*) – Names of the state variables which should be disabled
• **kernel_type** (*basestr*) – String must be a valid key for cls._KERNELS

**svm**
  Access to the SVM model.

**traindataset**
  Dataset which was used for training
  TODO – might better become state variable I guess

**untrain()**

16.4.22 clfs.smlr

Module: clfs.smlr

Inheritance diagram for mvpa.clfs.smlr:
Sparse Multinomial Logistic Regression classifier.

**Classes**

**SMLR**

class mvpa.clfs.smlr.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* (2005, IEEE Transactions on Pattern Analysis and Machine Intelligence). Be sure to cite that article if you use this classifier for your work.

**Note:** Available state variables:

- *feature_ids*: Feature IDS which were used for the actual training.
- *predicting_time+*: Time (in seconds) which took classifier to predict
- *predictions+*: Most recent set of predictions
- *trained_dataset*: The dataset it has been trained on
- *trained_labels+*: Set of unique labels it has been trained on
- *trained_nsamples+*: Number of samples it has been trained on
- *training_confusion*: Confusion matrix of learning performance
- *training_time+*: Time (in seconds) which took classifier to train
- *values+*: Internal classifier values the most recent predictions are based on
Initialize an SMLR classifier.

Parameters

- **lm** – The penalty term lambda. Larger values will give rise to more sparsification. (Default: 0.1)
- **convergence_tol** – When the weight change for each cycle drops below this value the regression is considered converged. Smaller values lead to tighter convergence. (Default: 0.001)
- **resamp_decay** – Decay rate in the probability of resampling a zero weight. 1.0 will immediately decrease to the min_resamp from 1.0, 0.0 will never decrease from 1.0. (Default: 0.5)
- **min_resamp** – Minimum resampling probability for zeroed weights. (Default: 0.001)
- **maxiter** – Maximum number of iterations before stopping if not converged. (Default: 10000)
- **has_bias** – Whether to add a bias term to allow fits to data not through zero. (Default: True)
- **fit_all_weights** – Whether to fit weights for all classes or to the number of classes minus one. Both should give nearly identical results, but if you set fit_all_weights to True it will take a little longer and yield weights that are fully analyzable for each class. Also, note that the convergence rate may be different, but convergence point is the same. (Default: True)
- **implementation** – Use C or Python as the implementation of stepwise_regression. C version brings significant speedup thus is the default one. (Default: C)
- **seed** – Seed to be used to initialize random generator, might be used to replicate the run. (Default: None)
- **unsparsify** – *EXPERIMENTAL* Whether to unsparsify the weights via regression. Note that it likely leads to worse classifier performance, but more interpretable weights. (Default: False)
- **std_to_keep** – Standard deviation threshold of weights to keep when unsparsifying. (Default: 2.0)
- **enable_states** (None or list of basestring) – Names of the state variables which should be enabled additionally to default ones
- **disable_states** (None or list of basestring) – Names of the state variables which should be disabled

**biases**

**getSensitivityAnalyzer(**kwargs)**

Returns a sensitivity analyzer for SMLR.

**weights**
SMLRWeights

class mvpa.clfs.smlr.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 customize this behavior.

Note: Available state variables:

*base_sensitivities: Stores basic sensitivities if the sensitivity relies on combining multiple ones
*biases+: A 1-d ndarray of biases
*null_prob+: State variable
*null_t: State variable
*raw_results: Computed results before applying any transformation algorithm

(States enabled by default are listed with +)

See Also:

Please refer to the documentation of the base class for more information:

Sensitivity

Initialize the analyzer with the classifier it shall use.

Parameters

•clf (Classifier) – classifier to use.(218,689),(353,704)

*force_training (Bool) – if classifier was already trained – do not retrain

*enable_states (None or list of basestring) – Names of the state variables which should be enabled additionally to default ones

*disable_states (None or list of basestring) – Names of the state variables which should be disabled

*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.

*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) – The estimated distribution is used to assign a probability for a certain value of the computed measure

16.4.23 clfs.stats

Module: clfs.stats

Inheritance diagram for mvpa.clfs.stats:
Estimator for classifier error distributions.

**Classes**

**AdaptiveNormal**

```python
class mvpa.clfs.stats.AdaptiveNormal(dist, **kwargs)
    Bases: mvpa.clfs.stats.AdaptiveNullDist

Adaptive Normal Distribution: params are (0, sqrt(1/nfeatures))
```

**Note:** Available state variables:

- 

(States enabled by default are listed with +)

**See Also:**

Please refer to the documentation of the base class for more information:

**AdaptiveNullDist**

**Parameters**

- **dist (distribution object)** – This can be any object the has a `cdf()` method to report the cumulative distribution function values.

- **enable_states (None or list of basestring)** – Names of the state variables which should be enabled additionally to default ones

- **disable_states (None or list of basestring)** – Names of the state variables which should be disabled

- **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.
AdaptiveNullDist

class mvpa.clfs.stats.AdaptiveNullDist(dist, **kwargs)

Bases: mvpa.clfs.stats.FixedNullDist

Adaptive distribution which adjusts parameters according to the data

WiP: internal implementation might change

**Note:** Available state variables:

- 

(States enabled by default are listed with +)

**See Also:**

Please refer to the documentation of the base class for more information:

FixedNullDist

**Parameters**

- **dist** *(distribution object)* – This can be any object the has a cdf() method to report the cumulative distribution function values.

- **enable_states** *(None or list of basestring)* – Names of the state variables which should be enabled additionally to default ones

- **disable_states** *(None or list of basestring)* – Names of the state variables which should be disabled

- **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.

**fit** *(measure, wdata, vdata=None)*

Cares about dimensionality of the feature space in measure

AdaptiveRDist

class mvpa.clfs.stats.AdaptiveRDist(dist, **kwargs)

Bases: mvpa.clfs.stats.AdaptiveNullDist

Adaptive rdist: params are (nfeatures-1, 0, 1)

**Note:** Available state variables:

- 

(States enabled by default are listed with +)

**See Also:**

Please refer to the documentation of the base class for more information:

AdaptiveNullDist

**Parameters**

- **dist** *(distribution object)* – This can be any object the has a cdf() method to report the cumulative distribution function values.
•`enable_states` *(None or list of basestring)* – Names of the state variables which should be enabled additionally to default ones

•`disable_states` *(None or list of basestring)* – Names of the state variables which should be disabled

•`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.

```python
cdf(x)
```

**FixedNullDist**

class `mvpa.clfs.stats.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.

```python
>>> 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])
```

**Note:** Available state variables:

- *(States enabled by default are listed with +)*

**See Also:**

Please refer to the documentation of the base class for more information:

**NullDist**

Parameters

•`dist` *(distribution object)* – This can be any object the has a `cdf()` method to report the cumulative distribution function values.

•`enable_states` *(None or list of basestring)* – Names of the state variables which should be enabled additionally to default ones

•`disable_states` *(None or list of basestring)* – Names of the state variables which should be disabled

•`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)*  
Return value of the cumulative distribution function at *x*.

**fit** *(measure, wdata, vdata=None)*  
Does nothing since the distribution is already fixed.

### MCNullDist

**class** `mvpa.clfs.stats.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.

**Note:** Available state variables:

*• dist_samples: Samples obtained for each permutation

(States enabled by default are listed with +)

**See Also:**

Please refer to the documentation of the base class for more information:

**NullDist**

Initialize Monte-Carlo Permutation Null-hypothesis testing

**Parameters**

*• 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.

• enable_states (None or list of basestring) – Names of the state variables which should be enabled additionally to default ones

• disable_states (None or list of basestring) – Names of the state variables which should be disabled

• 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)*  
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.

```python
fit(measure, wdata, vdata=None)
```

Fit the distribution by performing multiple cycles which repeatedly permuted labels in the training dataset.

**Parameters**

- `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.

## Nonparametric

```python
class mvpa.clfs.stats.Nonparametric(dist_samples, correction='clip')
```

Non-parametric 1d distribution – derives cdf based on stored values.

Introduced to complement parametric distributions present in scipy.stats.

**Parameters**

- `dist_samples (ndarray) – Samples to be used to assess the distribution.
- `correction ({'clip'} or None, optional) – Determines the behavior when .cdf is queried. If None – no correction is made. If ‘clip’ – values are clipped to lie in the range \([1/(N+2), (N+1)/(N+2)]\) (simply because non-parametric assessment lacks the power to resolve with higher precision in the tails, so ‘imagery’ samples are placed in each of the two tails).

```python
cdf(x)
```

Returns the cdf value at `x`.

```python
static fit(dist_samples)
```

## NullDist

```python
class mvpa.clfs.stats.NullDist(tail='both', **kwargs)
```

Base class for null-hypothesis testing.

**Note:** Available state variables:

- `.

(States enabled by default are listed with +)

**See Also:**

Please refer to the documentation of the base class for more information:

```python
ClassWithCollections
Cheap initialization.
```
Parameters

- **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.

- **enable_states** *(None or list of basestring)* – Names of the state variables which should be enabled additionally to default ones

- **disable_states** *(None or list of basestring)* – Names of the state variables which should be disabled

**cdf** *(x)*
Implementations return the value of the cumulative distribution function (left or right tail depending 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**

**Functions**

- **mvpa.clfs.stats.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

- **mvpa.clfs.stats.nanmean** *(x, axis=0)*
Compute the mean over the given axis ignoring nans.

Parameters

- **x** *(ndarray)* – input array

Results

- **m** * [float] the mean.

16.4.24 clfs.transerror

**Module:** clfs.transerror

Inheritance diagram for `mvpa.clfs.transerror`:
Utility class to compute the transfer error of classifiers.

**Classes**

**ClassifierError**

```python
class mvpa.clfs.transerror.ClassifierError(clf, labels=None, train=True, **kwargs)
```

*Compute (or return) some error of a (trained) classifier on a dataset.*

**See Also:**

Please refer to the documentation of the base class for more information:

**ClassWithCollections**

---

**Note:** Available state variables:

- `confusion`: State variable
- `training_confusion`: Proxy training_confusion from underlying classifier.

(States enabled by default are listed with `+`)  

**See Also:**

Please refer to the documentation of the base class for more information:

**ClassWithCollections**

Initialization.

**Parameters**

- `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__`
- `enable_states (None or list of basestring)` – Names of the state variables which should be enabled additionally to default ones
- `disable_states (None or list of basestring)` – Names of the state variables which should be disabled

`clf`
confusion = None

TODO Think that labels might be also symbolic thus can’t directly be indices of the array

labels

untrain()

Untrain the Error which relies on the classifier

ConfusionBasedError

class mvpa.clfs.transerror.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

See Also:

Please refer to the documentation of the base class for more information:

ClassifierError

Note: Available state variables:

*confusion: State variable
*training_confusion: Proxy training_confusion from underlying classifier.

(States enabled by default are listed with +)

See Also:

Please refer to the documentation of the base class for more information:

ClassifierError

Initialization.

Parameters

•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
•train (bool) – unless train=False, classifier gets trained if trainingdata provided to __call__
•enable_states (None or list of basestring) – Names of the state variables which should be enabled additionally to default ones
•disable_states (None or list of basestring) – Names of the state variables which should be disabled
•enable_states – Names of the state variables which should be enabled additionally to default ones
•disable_states – Names of the state variables which should be disabled

16.4. Classifiers and Errors 225
ConfusionMatrix

class mvpa.clfs.transerror.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

Parameters

• **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

Parameters

• **short** (bool) – if True, ignores the rest of the parameters and provides concise 1 line summary
• **header** (bool) – print header of the table
• **summary** (bool) – print summary (accuracy)
• **description** (bool) – print verbose description of presented statistics

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

Parameters

• **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 (beneficial 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)

**ROCCurve**

class mvpa.clfs.transerror.ROCCurve (labels, sets=None)

Bases: object

Generic class for ROC curve computation and plotting

Parameters

• **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?

**RegressionStatistics**

class mvpa.clfs.transerror.RegressionStatistics(**kwargs)

Bases: mvpa.clfs.transerror.SummaryStatistics

Class to contain information and display on regression results.

Initialize RegressionStatistics

Parameters

• **targets** – Optional set of targets

• **predictions** – Optional set of predictions

**asstring** (short=False, header=True, summary=True, description=False)

‘Pretty print’ the statistics

error

16.4. Classifiers and Errors
```python
plot(plot=True, plot_stats=True, splot=True)
```

Provide presentation of regression performance in image

Parameters

- `plot (bool)` – Plot regular plot of values (targets/predictions)
- `plot_stats (bool)` – Print basic statistics in the title
- `splot (bool)` – Plot scatter plot

Return type

( fig, im, cb ) – figure, imshow, colorbar

**SummaryStatistics**

class mvpa.clfs.transerror.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)

Parameters

- `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

Parameters

- `short (bool)` – if True, ignores the rest of the parameters and provides concise 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

TransferError

class mvpa.clfs.transerror.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.

See Also:

Please refer to the documentation of the base class for more information:

ClassifierError

Note: Available state variables:

*confusion: State variable
*null_prob+: Stores the probability of an error result under the NULL hypothesis
*samples_error: Per sample errors computed by invoking the error function for each sample individually. Errors are available in a dictionary with each samples origid as key.
*training_confusion: Proxy training_confusion from underlying classifier.

(States enabled by default are listed with +)

See Also:

Please refer to the documentation of the base class for more information:

ClassifierError

Initialization.

Parameters

*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) –
*train (bool) – unless train=False, classifier gets trained if trainingdata provided to __call__
*enable_states (None or list of basestring) – Names of the state variables which should be enabled additionally to default ones
*disable_states (None or list of basestring) – Names of the state variables which should be disabled
*enable_states – Names of the state variables which should be enabled additionally to default ones
*disable_states – Names of the state variables which should be disabled

errorfx
null_dist

16.4.25 clfs.warehouse

Module: clfs.warehouse

Inheritance diagram for `mvpa.clfs.warehouse`:

```
clfs.warehouse.Warehouse
```

Collection of classifiers to ease the exploration.

**Warehouse**

```python
class mvpa.clfs.warehouse.Warehouse(known_tags=None, matches=None)
    Bases: object
    
    Class to keep known instantiated classifiers

    Should provide easy ways to select classifiers of needed kind: clfswh[‘linear’, ‘svm’] should return all
    linear SVMs clfswh[‘linear’, ‘multiclass’] should return all linear classifiers capable of doing multiclass
    classification

    Initialize warehouse
    Parameters

    • 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
    Known internal tags of the classifiers

    items
    Registered items

    listing()
    Listing (description + internals) of registered items
```

16.5 Measures: Searchlights and Sensitivities

16.5.1 measures.anova

Module: measures.anova

Inheritance diagram for `mvpa.measures.anova`:
FeaturewiseDatasetMeasure performing a univariate ANOVA.

**Classes**

**CompoundOneWayAnova**

```python
class mvpa.measures.anova.CompoundOneWayAnova (combiner=<function SecondAxisSumOfAbs at 0x4892f50>, **kwargs)
Bases: mvpa.measures.anova.OneWayAnova
```

Compound comparisons via univariate ANOVA.
Provides F-scores per each label if compared to the other labels.

**Note:** Available state variables:

- *base_sensitivities*: Stores basic sensitivities if the sensitivity relies on combining multiple ones
- *null_prob+*: State variable
- *null_t*: State variable
- *raw_results*: Computed results before applying any transformation algorithm

(States enabled by default are listed with +)

**See Also:**

Please refer to the documentation of the base class for more information:

OneWayAnova

Initialize
Parameters

• **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.

• **enable_states** (*None or list of basestring*) – Names of the state variables which should be enabled additionally to default ones

• **disable_states** (*None or list of basestring*) – Names of the state variables which should be disabled

• **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*) – The estimated distribution is used to assign a probability for a certain value of the computed measure.

### OneWayAnova

```python
class mvpa.measures.anova.OneWayAnova (combiner=<function SecondAxisSumOfAbs at 0x4892f50>, **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.

**Note:** Available state variables:

• **base_sensitivities**: Stores basic sensitivities if the sensitivity relies on combining multiple ones

• **null_prob+**: State variable

• **null_t**: State variable

• **raw_results**: Computed results before applying any transformation algorithm

(States enabled by default are listed with +)

**See Also:**

Please refer to the documentation of the base class for more information:

FeaturewiseDatasetMeasure

```
Initialize

Parameters
```

• **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.

• **enable_states** (*None or list of basestring*) – Names of the state variables which should be enabled additionally to default ones

• **disable_states** (*None or list of basestring*) – Names of the state variables which should be disabled
• **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*) – The estimated distribution is used to assign a probability for a certain value of the computed measure.

### 16.5.2 measures.base

**Module:** measures.base

Inheritance diagram for `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.

**Classes**

**BoostedClassifierSensitivityAnalyzer**

```python
class mvpa.measures.base.BoostedClassifierSensitivityAnalyzer(*args_, **kwargs_):
    Bases: mvpa.measures.base.Sensitivity
    Set sensitivity analyzers to be merged into a single output
```

**Note:** Available state variables:

- **base_sensitivities**: Stores basic sensitivities if the sensitivity relies on combining multiple ones
- **null_prob+**: State variable
- **null_t**: State variable
- **raw_results**: Computed results before applying any transformation algorithm

(States enabled by default are listed with +)

**See Also:**
Please refer to the documentation of the base class for more information:

**Sensitivity**

Initialize instance of BoostedClassifierSensitivityAnalyzer

Parameters

- `enable_states` *(None or list of basestring)* – Names of the state variables which should be enabled additionally to default ones
- `disable_states` *(None or list of basestring)* – Names of the state variables which should be disabled

combined_analyzer

untrain()

Untrain BoostedClassifierSensitivityAnalyzer

---

**CombinedFeaturewiseDatasetMeasure**

class mvpa.measures.base.CombinedFeaturewiseDatasetMeasure (analyzers=None, combiner=None, **kwargs)

Bases: mvpa.measures.base.FeaturewiseDatasetMeasure

Set sensitivity analyzers to be merged into a single output

---

**Note:** Available state variables:

- `base_sensitivities`: Stores basic sensitivities if the sensitivity relies on combining multiple ones
- `null_prob+`: State variable
- `null_t`: State variable
- `raw_results`: Computed results before applying any transformation algorithm
- `sensitivities`: Sensitivities produced by each analyzer

(States enabled by default are listed with +)

**See Also:**

Please refer to the documentation of the base class for more information:

**FeaturewiseDatasetMeasure**

Initialize CombinedFeaturewiseDatasetMeasure

Parameters

- `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
- `enable_states` *(None or list of basestring)* – Names of the state variables which should be enabled additionally to default ones
- `disable_states` *(None or list of basestring)* – Names of the state variables which should be disabled
- `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.
• **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*) – The estimated distribution is used to assign a probability for a certain value of the computed measure.

```python
def untrain():
    Untrain CombinedFDM
```

### DatasetMeasure

```python
class mvpa.measures.base.DatasetMeasure(
    transformer=None, null_dist=None, **kwargs)
```

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_results` state variable.

Additionally all dataset measures support the estimation of the probability 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.

**Note:** For developers: 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.

**See Also:**

Please refer to the documentation of the base class for more information:

`ClassWithCollections`

**Note:** Available state variables:

- **null_prob**+: State variable
- **null_t**: State variable
- **raw_results**: Computed results before applying any transformation algorithm

(States enabled by default are listed with +)

**See Also:**

Please refer to the documentation of the base class for more information:

`ClassWithCollections`

Does nothing special.

**Parameters**

- **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*) – The estimated distribution is used to assign a probability for a certain value of the computed measure.
• **enable_states** *(None or list of basestring)* – Names of the state variables which should be enabled additionally to default ones

• **disable_states** *(None or list of basestring)* – Names of the state variables which should be disabled

`null_dist`

Return Null Distribution estimator

`null_prob = None`

Stores the probability of a measure under the NULL hypothesis

`null_t = None`

Stores the t-score corresponding to null_prob under assumption of Normal distribution

`transformer`

Return transformer

`untrain()`

‘Untraining’ Measure

Some derived classes might used classifiers, so we need to untrain those

**FeatureSelectionClassifierSensitivityAnalyzer**

```python
class mvpa.measures.base.FeatureSelectionClassifierSensitivityAnalyzer(*args_, **kwargs_)
```

Bases: `mvpa.measures.base.ProxyClassifierSensitivityAnalyzer`

Set sensitivity analyzer output be reverse mapped using mapper of the slave classifier

**Note:** Available state variables:

- **base_sensitivities**: Stores basic sensitivities if the sensitivity relies on combining multiple ones
- **clf_sensitivities**: Stores sensitivities of the proxied classifier
- **null_prob+**: State variable
- **null_t**: State variable
- **raw_results**: Computed results before applying any transformation algorithm

(States enabled by default are listed with +)

**See Also:**

Please refer to the documentation of the base class for more information:

ProxyClassifierSensitivityAnalyzer

Initialize instance of ProxyClassifierSensitivityAnalyzer

Parameters

• **enable_states** *(None or list of basestring)* – Names of the state variables which should be enabled additionally to default ones

• **disable_states** *(None or list of basestring)* – Names of the state variables which should be disabled
FeaturewiseDatasetMeasure

class mvpa.measures.base.FeaturewiseDatasetMeasure (combiner=<function SecondAxisSumOfAbs at 0x4892f50>, **kwargs)

Bases: mvpa.measures.base.DatasetMeasure

A per-feature-measure computed from a Dataset (base class).
Should behave like a DatasetMeasure.

Note: Available state variables:
• base_sensitivities: Stores basic sensitivities if the sensitivity relies on combining multiple ones
• null_prob+: State variable
• null_t: State variable
• raw_results: Computed results before applying any transformation algorithm

(States enabled by default are listed with +)

See Also:
Please refer to the documentation of the base class for more information:
DatasetMeasure

Initialize

Parameters

• 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.

• enable_states (None or list of basestring) – Names of the state variables which should be enabled additionally to default ones

• disable_states (None or list of basestring) – Names of the state variables which should be disabled

• 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) – The estimated distribution is used to assign a probability for a certain value of the computed measure.

combiner

Return combiner

MappedClassifierSensitivityAnalyzer

class mvpa.measures.base.MappedClassifierSensitivityAnalyzer (*args_, **kwargs_)

Bases: mvpa.measures.base.ProxyClassifierSensitivityAnalyzer

Set sensitivity analyzer output be reverse mapped using mapper of the slave classifier

Note: Available state variables:
• base_sensitivities: Stores basic sensitivities if the sensitivity relies on combining multiple ones
• **clf_sensitivities**: Stores sensitivities of the proxied classifier
• **null_prob+**: State variable
• **null_t**: State variable
• **raw_results**: Computed results before applying any transformation algorithm

(States enabled by default are listed with +)

**See Also:**
Please refer to the documentation of the base class for more information:

*ProxyClassifierSensitivityAnalyzer*

Initialize instance of *ProxyClassifierSensitivityAnalyzer*

**Parameters**

• **enable_states** (*None or list of basestring*) – Names of the state variables which should be enabled additionally to default ones
• **disable_states** (*None or list of basestring*) – Names of the state variables which should be disabled

*ProxyClassifierSensitivityAnalyzer*

class `mvpa.measures.base.ProxyClassifierSensitivityAnalyzer(*args_, **kwargs_)`
Bases: `mvpa.measures.base.Sensitivity`

Set sensitivity analyzer output just to pass through

**Note:** Available state variables:

• **base_sensitivities**: Stores basic sensitivities if the sensitivity relies on combining multiple ones
• **clf_sensitivities**: Stores sensitivities of the proxied classifier
• **null_prob+**: State variable
• **null_t**: State variable
• **raw_results**: Computed results before applying any transformation algorithm

(States enabled by default are listed with +)

**See Also:**
Please refer to the documentation of the base class for more information:

*ProxyClassifierSensitivityAnalyzer*

Initialize instance of *ProxyClassifierSensitivityAnalyzer*

**Parameters**

• **enable_states** (*None or list of basestring*) – Names of the state variables which should be enabled additionally to default ones
• **disable_states** (*None or list of basestring*) – Names of the state variables which should be disabled

*ProxyClassifierSensitivityAnalyzer*

untrain()
**Sensitivity**

```python
class mvpa.measures.base.Sensitivity(clf, force_training=True, **kwargs)
Bases: mvpa.measures.base.FeaturewiseDatasetMeasure
```

No documentation found. Sorry!

**Note:** Available state variables:

- `base_sensitivities`: Stores basic sensitivities if the sensitivity relies on combining multiple ones
- `null_prob+`: State variable
- `null_t`: State variable
- `raw_results`: Computed results before applying any transformation algorithm

(States enabled by default are listed with +)

**See Also:**

Please refer to the documentation of the base class for more information:

`FeaturewiseDatasetMeasure`

Initialize the analyzer with the classifier it shall use.

**Parameters**

- `clf` (Classifier) – classifier to use.
- `force_training` (Bool) – if classifier was already trained – do not retrain
- `enable_states` (None or list of basestring) – Names of the state variables which should be enabled additionally to default ones
- `disable_states` (None or list of basestring) – Names of the state variables which should be disabled
- `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.
- `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) – The estimated distribution is used to assign a probability for a certain value of the computed measure.

```python
clf
feature_ids
    Return feature_ids used by the underlying classifier
untrain()
    Untrain corresponding classifier for Sensitivity
```

**SplitFeaturewiseDatasetMeasure**

```python
class mvpa.measures.base.SplitFeaturewiseDatasetMeasure(splitter, analyzer, in-split_index=0, combiner=None, **kwargs)
Bases: mvpa.measures.base.FeaturewiseDatasetMeasure
```

Compute measures across splits for a specific analyzer
Note: Available state variables:

- `base_sensitivities`: Stores basic sensitivities if the sensitivity relies on combining multiple ones
- `null_prob+`: State variable
- `null_t`: State variable
- `raw_results`: Computed results before applying any transformation algorithm
- `sensitivities`: Sensitivities produced for each split
- `splits`: Store the actual splits of the data. Can be memory expensive

(States enabled by default are listed with +)

See Also:

Please refer to the documentation of the base class for more information:

`FeaturewiseDatasetMeasure`

Initialize `SplitFeaturewiseDatasetMeasure`

Parameters

- `splitter` (`Splitter`) – Splitter to use to split the dataset
- `analyzer` (`DatasetMeasure`) – Measure to be used. Could be analyzer as well (XXX)
- `insplit_index` (int) – splitter generates tuples of dataset on each iteration (usually 0th for training, 1st for testing). On what split index in that tuple to operate.
- `enable_states` (`None or list of basestring`) – Names of the state variables which should be enabled additionally to default ones
- `disable_states` (`None or list of basestring`) – Names of the state variables which should be disabled
- `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.
- `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`) – The estimated distribution is used to assign a probability for a certain value of the computed measure.

`untrain()`

Untrain `SplitFeaturewiseDatasetMeasure`

---

`StaticDatasetMeasure`

```python
class mvpa.measures.base.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

Note: Available state variables:

- `null_prob+`: State variable
null_t: State variable

raw_results: Computed results before applying any transformation algorithm

(States enabled by default are listed with +)

See Also:

Please refer to the documentation of the base class for more information:

DatasetMeasure

Initialize.

Parameters

- measure – actual sensitivity to be returned
- bias – optionally available bias
- enable_states (None or list of basestring) – Names of the state variables which should be enabled additionally to default ones
- disable_states (None or list of basestring) – Names of the state variables which should be disabled
- 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) – The estimated distribution is used to assign a probability for a certain value of the computed measure.

16.5.3 measures.corrcoef

Module: measures.corrcoef

Inheritance diagram for mvpa.measures.corrcoef:
FeaturewiseDatasetMeasure of correlation with the labels.

**CorrCoef**

class `mvpa.measures.corrcoef.CorrCoef(pvalue=False, attr='labels', **kwargs)`

Bases: `mvpa.measures.base.FeaturewiseDatasetMeasure`

FeaturewiseDatasetMeasure that performs correlation with labels

XXX: Explain me!

Note: Available state variables:

- `base_sensitivities`: Stores basic sensitivities if the sensitivity relies on combining multiple ones
- `null_prob+`: State variable
- `null_t`: State variable
- `raw_results`: Computed results before applying any transformation algorithm

(States enabled by default are listed with+)

See Also:

Please refer to the documentation of the base class for more information:

FeaturewiseDatasetMeasure

Initialize

Parameters

- `pvalue (bool)` – Either to report p-value of pearsons correlation coefficient instead of pure correlation coefficient
- `attr (basestring)` – What attribut to correlate with
- `enable_states (None or list of basestring)` – Names of the state variables which should be enabled additionally to default ones
- `disable_states (None or list of basestring)` – Names of the state variables which should be disabled
- `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.
- `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)` – The estimated distribution is used to assign a probability for a certain value of the computed measure.

**16.5.4 measures.corrstability**

Module: `measures.corrstability`

Inheritance diagram for `mvpa.measures.corrstability:`
FeaturewiseDatasetMeasure of stability of labels across chunks based on correlation.

**CorrStability**

class mvpa.measures.corrstability.CorrStability(attr='labels', **kwargs)

Bases: mvpa.measures.base.FeaturewiseDatasetMeasure

FeaturewiseDatasetMeasure that assesses feature stability across runs for each unique label by correlating label activity for pairwise combinations of the chunks.

If there are multiple samples with the same label in a single chunk (as is typically the case) this algorithm will take the featurewise average of the sample activations to get a single value per label, per chunk.

**Note:** Available state variables:

- `base_sensitivities`: Stores basic sensitivities if the sensitivity relies on combining multiple ones
- `null_prob+`: State variable
- `null_t`: State variable
- `raw_results`: Computed results before applying any transformation algorithm

(States enabled by default are listed with+)

**See Also:**

Please refer to the documentation of the base class for more information:

FeaturewiseDatasetMeasure

Initialize

Parameters

- `attr (basestring)` – Attribute to correlate across chunks.
- `enable_states (None or list of basestring)` – Names of the state variables which should be enabled additionally to default ones
**disable_states** *(None or list of basestring)* – Names of the state variables which should be disabled

**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.

**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)* – The estimated distribution is used to assign a probability for a certain value of the computed measure.

### 16.5.5 measures.ds

**Module:** measures.ds

Inheritance diagram for `mvpa.measures.ds`:

```
    misc.state.ClassWithCollections
          |                         |
          v                         v
measures.base.DatasetMeasure
          |                         |
          v                         v
measures.ds.DSMDatasetMeasure
```

Dissimilarity measure.

**DSMDatasetMeasure**

**class** `mvpa.measures.ds.DSMDatasetMeasure` *(dsmatrix, dset_metric, output_metric='spearman')*

Bases: `mvpa.measures.base.DatasetMeasure`

DSMDatasetMeasure creates a DatasetMeasure object where metric can be one of ‘euclidean’, ‘spearman’, ‘pearson’ or ‘confusion’

**Note:** Available state variables:

- **null_prob**: State variable
- **null_t**: State variable
- **raw_results**: Computed results before applying any transformation algorithm

(States enabled by default are listed with +)

**See Also:**
Please refer to the documentation of the base class for more information:

**DatasetMeasure**

Initialize instance of DSMDatasetMeasure

Parameters

- **enable_states** (None or list of basestring) – Names of the state variables which should be enabled additionally to default ones
- **disable_states** (None or list of basestring) – Names of the state variables which should be disabled

### 16.5.6 measures.glm

**Module:** measures.glm

Inheritance diagram for `mvpa.measures.glm`:

```
+---------------------+          +---------------------+
| misc.state.ClassWithCollections |         | measures.base.DatasetMeasure |
|                                 |          +---------------------+
| measures.base.DatasetMeasure  |          | measures.base.FeaturewiseDatasetMeasure |
|                                 |          +---------------------+
| measures.glm.GLM              |          |
```

The general linear model (GLM).

**GLM**

```
class mvpa.measures.glm.GLM(design, voi='pe', **kwargs)
```

**Bases:** `mvpa.measures.base.FeaturewiseDatasetMeasure`

General linear model (GLM).

Regressors can be defined in a design matrix and a linear fit of the data is computed univariately (i.e. independently for each feature). This measure can report ‘raw’ parameter estimates (i.e. beta weights) of the linear model, as well as standardized parameters (z-stat) using an ordinary least squares (aka fixed-effects) approach to estimate the parameter estimate.

The measure is reported in a (nfeatures x nregressors)-shaped array.
Note: Available state variables:

• `base_sensitivities`: Stores basic sensitivities if the sensitivity relies on combining multiple ones
• `null_prob+`: State variable
• `null_t`: State variable
• `pe`: Parameter estimates (nfeatures x nparameters).
• `raw_results`: Computed results before applying any transformation algorithm
• `zstat`: Standardized parameter estimates (nfeatures x nparameters).

(States enabled by default are listed with+)

See Also:

Please refer to the documentation of the base class for more information:

`FeaturewiseDatasetMeasure`

Parameters

• `design` (array(nsamples x nregressors)) – GLM design matrix.
• `voi` (‘pe’ | ‘zstat’) – Variable of interest that should be reported as feature-wise measure. ‘beta’ are the parameter estimates and ‘zstat’ returns standardized parameter estimates.
• `enable_states` (None or list of basestring) – Names of the state variables which should be enabled additionally to default ones
• `disable_states` (None or list of basestring) – Names of the state variables which should be disabled
• `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.
• `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) – The estimated distribution is used to assign a probability for a certain value of the computed measure.

16.5.7 measures.irelief

Module: measures.irelief

Inheritance diagram for mvpa.measures.irelief:

**Classes**

**IterativeRelief**

```python
class mvpa.measures.irelief.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.


Note that current implementation allows to use only exponential-like kernels. Support for linear kernel will be added later.

**Note:** Available state variables:

- **base_sensitivities**: Stores basic sensitivities if the sensitivity relies on combining multiple ones
- **null_prob+**: State variable
- **null_t**: State variable
- **raw_results**: Computed results before applying any transformation algorithm

(States enabled by default are listed with +)

16.5. Measures: Searchlights and Sensitivities 247
See Also:
Please refer to the documentation of the base class for more information:

`FeaturewiseDatasetMeasure`

Constructor of the IRELIEF class.

Parameters

- `enable_states` (*None or list of basestring*) – Names of the state variables which should be enabled additionally to default ones
- `disable_states` (*None or list of basestring*) – Names of the state variables which should be disabled
- `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.
- `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*) – The estimated distribution is used to assign a probability for a certain value of the computed measure.

`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.

`IterativeReliefOnline`

class `mvpa.measures.irelief.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.

**Note:** Available state variables:

- `base_sensitivities`: Stores basic sensitivities if the sensitivity relies on combining multiple ones
- `null_prob+`: State variable
- `null_t`: State variable
- `raw_results`: Computed results before applying any transformation algorithm

(States enabled by default are listed with +)

See Also:
Please refer to the documentation of the base class for more information:

`IterativeRelief`
Constructor of the IRELIEF class.

Parameters

• **enable_states** (*None or list of basestring*) – Names of the state variables which should be enabled additionally to default ones

• **disable_states** (*None or list of basestring*) – Names of the state variables which should be disabled

• **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.

• **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*) – The estimated distribution is used to assign a probability for a certain value of the computed measure.

**IterativeReliefOnline_Devel**

class mvpa.measures.irelief.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.


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.

Note: Available state variables:

• **base_sensitivities**: Stores basic sensitivities if the sensitivity relies on combining multiple ones

• **null_prob+**: State variable

• **null_t**: State variable

• **raw_results**: Computed results before applying any transformation algorithm

(States enabled by default are listed with +)

See Also:

Please refer to the documentation of the base class for more information:

**IterativeRelief_Devel**

Constructor of the IRELIEF class.

Parameters
• **enable_states** *(None or list of basestring)* – Names of the state variables which should be enabled additionally to default ones

• **disable_states** *(None or list of basestring)* – Names of the state variables which should be disabled

• **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.

• **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)* – The estimated distribution is used to assign a probability for a certain value of the computed measure.

### IterativeRelief_Devel

```python
class mvpa.measures.irelief.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 DEVELOPMENT.**

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.


Note that current implementation allows to use only exponential-like kernels. Support for linear kernel will be added later.

**Note:** Available state variables:

• **base_sensitivities**: Stores basic sensitivities if the sensitivity relies on combining multiple ones

• **null_prob+**: State variable

• **null_t**: State variable

• **raw_results**: Computed results before applying any transformation algorithm

(States enabled by default are listed with +)

**See Also:**

Please refer to the documentation of the base class for more information:

*FeaturewiseDatasetMeasure*

Constructor of the IRELIEF class.

Parameters

• **enable_states** *(None or list of basestring)* – Names of the state variables which should be enabled additionally to default ones
• **disable states** *(None or list of basestring)* – Names of the state variables which should be disabled

• **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.

• **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)* – The estimated distribution is used to assign a probability for a certain value of the computed measure.

```python
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.

### 16.5.8 measures.noiseperturbation

**Module:** measures.noiseperturbation

Inheritance diagram for `mvpa.measures.noiseperturbation`:

This is a `FeaturewiseDatasetMeasure` that uses a scalar `DatasetMeasure` and selective noise perturbation to compute a sensitivity map.
NoisePerturbationSensitivity

class mvpa.measures.noiseperturbation.NoisePerturbationSensitivity (datameasure, noise=<built-in method normal of mtrand.RandomState object at 0x2aee282a86a8>)

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!

Note: Available state variables:

• base_sensitivities: Stores basic sensitivities if the sensitivity relies on combining multiple ones
• null_prob+: State variable
• null_t: State variable
• raw_results: Computed results before applying any transformation algorithm

(States enabled by default are listed with +)

See Also:

Please refer to the documentation of the base class for more information:
FeaturewiseDatasetMeasure

Cheap initialization.

Parameters

• 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.
• enable_states (None or list of basestring) – Names of the state variables which should be enabled additionally to default ones
• disable_states (None or list of basestring) – Names of the state variables which should be disabled

16.5.9 measures.pls

Module: measures.pls

Inheritance diagram for mvpa.measures.pls:
Classes

**PLS**

class mvpa.measures.pls.PL5 (num_permutations=200, num_bootstraps=100, **kwargs)

Bases: mvpa.measures.base.FeaturewiseDatasetMeasure

No documentation found. Sorry!

**Note:** Available state variables:

- *base_sensitivities:* Stores basic sensitivities if the sensitivity relies on combining multiple ones
- *null_prob+:* State variable
- *null_t:* State variable
- *raw_results:* Computed results before applying any transformation algorithm

(States enabled by default are listed with +)

**See Also:**

Please refer to the documentation of the base class for more information:

FeaturewiseDatasetMeasure

Initialize instance of PLS

Parameters

- **enable_states** (*None or list of basestring*) – Names of the state variables which should be enabled additionally to default ones
• **disable_states** (*None or list of basestring*) – Names of the state variables which should be disabled

• **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.

• **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*) – The estimated distribution is used to assign a probability for a certain value of the computed measure.

**TaskPLS**

class `mvpa.measures.pls.TaskPLS` (num_permutations=200, num_bootstraps=100, **kwargs)

Bases: `mvpa.measures.pls.PLS`

No documentation found. Sorry!

**Note:** Available state variables:

* **base_sensitivities**: Stores basic sensitivities if the sensitivity relies on combining multiple ones
* **null_prob+**: State variable
* **null_t**: State variable
* **raw_results**: Computed results before applying any transformation algorithm

(States enabled by default are listed with +)

**See Also:**

Please refer to the documentation of the base class for more information:

**PLS**

Initialize instance of PLS

Parameters

• **enable_states** (*None or list of basestring*) – Names of the state variables which should be enabled additionally to default ones

• **disable_states** (*None or list of basestring*) – Names of the state variables which should be disabled

• **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.

• **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*) – The estimated distribution is used to assign a probability for a certain value of the computed measure.
16.5.10 measures.searchlight

Module: measures.searchlight

Inheritance diagram for mvpa.measures.searchlight:

```
+--------------------------
| misc.state.ClassWthCollections
+--------------------------
       ^
       | measures.base.DatasetMeasure
+--------------------------
| measures.searchlight.Searchlight
```

Implementation of the Searchlight algorithm

**Searchlight**

class mvpa.measures.searchlight.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).

See Also:

Please refer to the documentation of the base class for more information:

DatasetMeasure

---

**Note:** Available state variables:

- *null_prob*: State variable
- *null_t*: State variable
- *raw_results*: Computed results before applying any transformation algorithm
- *spheresizes*: Number of features in each sphere.

(States enabled by default are listed with +)

See Also:

Please refer to the documentation of the base class for more information:

DatasetMeasure

---

Parameters
• **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. Provided dataset should have a metric assigned (for NiftiDataset, voxel size is used to provide such a metric, hence radius should be specified in mm).

• **center_ids** (*list(int)*) — List of feature ids (not coordinates) the shall serve as sphere centers. By default all features will be used.

• **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*) — The estimated distribution is used to assign a probability for a certain value of the computed measure.

• **enable_states** (*None or list of basestring*) — Names of the state variables which should be enabled additionally to default ones.

• **disable_states** (*None or list of basestring*) — Names of the state variables which should be disabled.

• **enable_states** — Names of the state variables which should be enabled additionally to default ones.

• **disable_states** — Names of the state variables which should be disabled.

**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 sensitivities and this conflicts with the intended behavior of a SensitivityAnalyzer.

### 16.5.11 measures.splitmeasure

**Module:** measures.splitmeasure

Inheritance diagram for `mvpa.measures.splitmeasure`:
This is a FeaturewiseDatasetMeasure that uses another FeaturewiseDatasetMeasure and runs it multiple times on different splits of a Dataset.

Classes

SplitFeaturewiseMeasure

class mvpa.measures.splitmeasure.SplitFeaturewiseMeasure(sensana, splitter=<class 'mvpa.datasets.splitters.NoneSplitter'>, combiner=<function FirstAxisMean at 0x4892de8>, **kwargs)

Bases: mvpa.measures.base.FeaturewiseDatasetMeasure

This is a FeaturewiseDatasetMeasure that uses another FeaturewiseDatasetMeasure and runs it multiple times on different 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 specified to the constructor. The results of the functions specified in the postproc dictionary will be available via their respective keywords.

Note: Available state variables:

• base_sensitivities: Stores basic sensitivities if the sensitivity relies on combining multiple ones
• maps: To store maps per each split
• null_prob+: State variable
• null_t: State variable
**raw_results**: Computed results before applying any transformation algorithm

(States enabled by default are listed with +)

**See Also:**

Please refer to the documentation of the base class for more information:

`FeaturewiseDatasetMeasure`

Cheap initialization.

**Parameters**

- **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.
- **enable_states** ([`None or list of basestring`]) – Names of the state variables which should be enabled additionally to default ones
- **disable_states** ([`None or list of basestring`]) – Names of the state variables which should be disabled
- **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]) – The estimated distribution is used to assign a probability for a certain value of the computed measure.

`TScoredFeaturewiseMeasure`

**class** `mvpa.measures.splitmeasure.TScoredFeaturewiseMeasure`(`sensana`, `splitter`,
`noise_level=0.0`, **kwargs)

**Bases**: `mvpa.measures.splitmeasure.SplitFeaturewiseMeasure`

`SplitFeaturewiseMeasure` computing featurewise t-score of sensitivities across splits.

**Note**: Available state variables:

- **base_sensitivities**: Stores basic sensitivities if the sensitivity relies on combining multiple ones
- **maps**: To store maps per each split
- **null_prob+**: State variable
- **null_t**: State variable
- **raw_results**: Computed results before applying any transformation algorithm

(States enabled by default are listed with +)

**See Also:**

Please refer to the documentation of the base class for more information:

`SplitFeaturewiseMeasure`

Cheap initialization.

**Parameters**
• **sensana** (*SensitivityAnalyzer*) – 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.

• **noise_level** (*float*) – Theoretical output of the respective *SensitivityAnalyzer* for a pure noise pattern. For most algorithms this is probably zero, hence the default.

• **enable_states** (*None or list of basestring*) – Names of the state variables which should be enabled additionally to default ones

• **disable_states** (*None or list of basestring*) – Names of the state variables which should be disabled

• **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 a 1d ndarray.

• **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*) – The estimated distribution is used to assign a probability for a certain value of the computed measure.

## 16.6 Feature Selection

### 16.6.1 featsel.base

**Module: featsel.base**

Feature selection base class and related stuff base classes and helpers.

**Classes**

**CombinedFeatureSelection**

```python
class mvpa.featsel.base.CombinedFeatureSelection
    (feature_selections, combiner, **kwargs)
```

**Bases:** mvpa.featsel.base.FeatureSelection

Meta feature selection utilizing several embedded selection methods.

Each embedded feature selection method is computed individually. Afterwards all feature sets are combined by either taking the union or intersection of all sets.

The individual feature sets of all embedded methods are optionally avialable from the *selections_ids* state variable.
Note: Available state variables:

- `selected_ids`: State variable
- `selections_ids+`: List of feature id sets for each performed method.

(States enabled by default are listed with +)

See Also:
Please refer to the documentation of the base class for more information:

`FeatureSelection`

Parameters

- `feature_selections` *(list)* – FeatureSelection instances to run. Order is not important.
- `combiner` *(‘union’, ‘intersection’)* – which method to be used to combine the feature selection set of all computed methods.
- `enable_states` *(None or list of basestring)* – Names of the state variables which should be enabled additionally to default ones
- `disable_states` *(None or list of basestring)* – Names of the state variables which should be disabled

`combiner`
Selection set combination method.

`feature_selections`
List of FeatureSelections

`untrain` ()

`FeatureSelection`

class `mvpa.featsel.base.FeatureSelection(**kwargs)`

Bases: `mvpa.misc.state.ClassWithCollections`

Base class for any feature selection

Base class for Functors which implement feature selection on the datasets.

Note: Available state variables:

- `selected_ids`: State variable

(States enabled by default are listed with +)

See Also:
Please refer to the documentation of the base class for more information:

`ClassWithCollections`

Initialize instance of FeatureSelection

Parameters

- `enable_states` *(None or list of basestring)* – Names of the state variables which should be enabled additionally to default ones
•**disable_states** *(None or list of basestring)* – Names of the state variables which should be disabled

```python
untrain()
```

‘Untrain’ feature selection

Necessary for full ‘untraining’ of the classifiers. By default does nothing, needs to be overridden in corresponding feature selections to pass to the sensitivities

### FeatureSelectionPipeline

```python
class mvpa.featsel.base.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.

**Note:** Available state variables:

•*nfeatures*: Number of features before each step in pipeline

•*selected_ids*: State variable

(States enabled by default are listed with +)

**See Also:**

Please refer to the documentation of the base class for more information:

#### FeatureSelection

Initialize feature selection pipeline

**Parameters**

•**feature_selections** *(list of FeatureSelection)* – selections which to use. Order matters

•**enable_states** *(None or list of basestring)* – Names of the state variables which should be enabled additionally to default ones

•**disable_states** *(None or list of basestring)* – Names of the state variables which should be disabled

**feature_selections**

List of FeatureSelections

```python
untrain()
```

### SensitivityBasedFeatureSelection

```python
class mvpa.featsel.base.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.

**Note:** Available state variables:
• selected_ids: State variable
• sensitivity: State variable

(States enabled by default are listed with +)

See Also:
Please refer to the documentation of the base class for more information:

FeatureSelection
Initialize feature selection

Parameters

• 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.

• enable_states (None or list of basestring) – Names of the state variables which should be enabled additionally to default ones

• disable_states (None or list of basestring) – Names of the state variables which should be disabled

sensitivity_analyzer
Measure which was used to do selection

untrain()

16.6.2 featsel.helpers

Module: featsel.helpers

Inheritance diagram for mvpa.featsel.helpers:

Classes

BestDetector

class mvpa.featsel.helpers.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
Parameters

- **fun (functor)** – Functor to select the best results. Defaults to \texttt{min}.
- **lastminimum (bool)** – Toggle whether the latest or the earliest minimum is used as optimal value to determine the stopping criterion.

**ElementSelector**

class \texttt{mvpa.featsel.helpers.ElementSelector (mode='discard', \*\*kwargs)}

Bases: \texttt{mvpa.misc.state.ClassWithCollections}

Base class to implement functors to select some elements based on a sequence of values.

**Note:** Available state variables:

- \texttt{True+}: Store number of discarded elements.

(States enabled by default are listed with +)

**See Also:**

Please refer to the documentation of the base class for more information:

\texttt{ClassWithCollections}

Cheap initialization.

Parameters

- **mode ([‘discard’, ‘select’])** – Decides whether to select or to discard features.
- **enable\_states (None or list of basestring)** – Names of the state variables which should be enabled additionally to default ones
- **disable\_states (None or list of basestring)** – Names of the state variables which should be disabled

**FixedErrorThresholdStopCrit**

class \texttt{mvpa.featsel.helpers.FixedErrorThresholdStopCrit (threshold)}

Bases: \texttt{mvpa.featsel.helpers.StoppingCriterion}

Stop computation if the latest error drops below a certain threshold.

Initialize with threshold.

Parameters

- **threshold (float \([0,1]\))** – Error threshold.

**FixedNElementTailSelector**

class \texttt{mvpa.featsel.helpers.FixedNElementTailSelector (nelements, \*\*kwargs)}

Bases: \texttt{mvpa.featsel.helpers.TailSelector}

Given a sequence, provide set of IDs for a fixed number of to be selected elements.
Note: Available state variables:

*True+: Store number of discarded elements.

(States enabled by default are listed with +)

See Also:
Please refer to the documentation of the base class for more information:

TailSelector

Cheap initialization.

Parameters

*nelements (int) – Number of elements to select/discard.

*enable_states (None or list of basestring) – Names of the state variables which should be enabled additionally to default ones

*disable_states (None or list of basestring) – Names of the state variables which should be disabled

*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.

*mode (['discard', 'select']) – Decides whether to select or to discard features.

nelements

FractionTailSelector

class mvpa.featsel.helpers.FractionTailSelector (felements, **kwargs)

Bases: mvpa.featsel.helpers.TailSelector

Given a sequence, provide IDs for a fraction of elements

Note: Available state variables:

*True+: Store number of discarded elements.

(States enabled by default are listed with +)

See Also:
Please refer to the documentation of the base class for more information:

TailSelector

Cheap initialization.

Parameters

*felements (float (0.1,0.1)) – Fraction of elements to select/discard. Note: Even when 0.0 is specified at least one element will be selected.

*enable_states (None or list of basestring) – Names of the state variables which should be enabled additionally to default ones

*disable_states (None or list of basestring) – Names of the state variables which should be disabled
• `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.
• `mode` ("discard", "select") – Decides whether to select or to discard features.

felements

**MultiStopCrit**

class mvpa.featsel.helpers.MultiStopCrit (crits, mode='or')

Bases: mvpa.featsel.helpers.StoppingCriterion

Stop computation if the latest error drops below a certain threshold.

Parameters

• `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.

**NBackHistoryStopCrit**

class mvpa.featsel.helpers.NBackHistoryStopCrit (bestdetector=<mvpa.featsel.helpers.BestDetector object at 0x88b1ed0>, steps=10)

Bases: mvpa.featsel.helpers.StoppingCriterion

Stop computation if for a number of steps error was increasing

Initialize with number of steps

Parameters

• `bestdetector` (BestDetector instance) – used to determine where the best error is located.
• `steps` (int) – How many steps to check after optimal value.

**NStepsStopCrit**

class mvpa.featsel.helpers.NStepsStopCrit (steps)

Bases: mvpa.featsel.helpers.StoppingCriterion

Stop computation after a certain number of steps.

Initialize with number of steps.

Parameters

• `steps` (int) – Number of steps after which to stop.
RangeElementSelector

class mvpa.featsel.helpers.RangeElementSelector(lower=None, upper=None, inclusive=False, mode='select', **kwargs)

Bases: mvpa.featsel.helpers.ElementSelector

Select elements based on specified range of values

Note: Available state variables:

• True+: Store number of discarded elements.

(States enabled by default are listed with +)

See Also:

Please refer to the documentation of the base class for more information:

ElementSelector

Initialization RangeElementSelector

Parameters

• 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??
• enable_states (None or list of basestring) – Names of the state variables which should be enabled additionally to default ones
• disable_states (None or list of basestring) – Names of the state variables which should be disabled

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

StoppingCriterion

class mvpa.featsel.helpers.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

TailSelector

class mvpa.featsel.helpers.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.

Note: Available state variables:
•**True+:** Store number of discarded elements.

(States enabled by default are listed with +)

See Also:
Please refer to the documentation of the base class for more information:

ElementSelector

Initialize TailSelector

Parameters

- **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.
- **enable_states** (None or list of basestring) – Names of the state variables which should be enabled additionally to default ones
- **disable_states** (None or list of basestring) – Names of the state variables which should be disabled
- **mode** (["discard", "select"]) – Decides whether to select or to discard features.

16.6.3 featsel.ifs

Module: featsel.ifs

Inheritance diagram for mvpa.featsel.ifs:

```
  misc.state.ClassWithCollections
    
  featsel.base.FeatureSelection
    
  featsel.ifs.IFS
```

Incremental feature search (IFS).

Very similar to Recursive feature elimination (RFE), but instead of beginning with all features and stripping some sequentially, start with an empty feature set and include important features successively.
**IFS**

**class** mvpa.featsel.ifsf.**IFS** *(data_measure, transfer_error, bestdetector=<mvpa.featsel.helpers.BestDetector object at 0x99868d0>, stopping_criterion=<mvpa.featsel.helpers.NBackHistoryStopCrit object at 0x9986e90>, feature_selector=FixedNElementTailSelector(), number=1.000000, **kwargs)*

**Bases:** mvpa.featsel.base.**FeatureSelection**

**Incremental feature search.**

A scalar `DatasetMeasure` is computed multiple times on variations of a certain dataset. These measures are then 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 testdataset is computed. This procedure is repeated until a given `StoppingCriterion` is reached.

**Note:** Available state variables:

- `errors+`: State variable
- `selected_ids`: State variable

(States enabled by default are listed with +)

**See Also:**

Please refer to the documentation of the base class for more information:

**FeatureSelection**

**Initialize incremental feature search**

**Parameters**

- **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.
- **enable_states** *(None or list of basestring)* – Names of the state variables which should be enabled additionally to default ones
- **disable_states** *(None or list of basestring)* – Names of the state variables which should be disabled

**16.6.4 featsel.rfe**

**Module:** featsel.rfe

Inheritance diagram for mvpa.featsel.rfe:
Recursive feature elimination.

**RFE**

```python
class mvpa.featsel.rfe.RFE(sensitivity_analyzer, transfer_error, feature_selector=FractionTailSelector(fraction=0.050000), bestdetector=<mvpa.featsel.helpers.BestDetector object at 0x8d7c910>, stopping_criterion=<mvpa.featsel.helpers.NBackHistoryStopCrit object at 0x8d7cf50>, 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 testdataset is computed. This procedure is repeated until a given StoppingCriterion is reached.

Such strategy after


was applied to SVM-based analysis of fMRI data in


**Note:** Available state variables:

- `errors+`: State variable
- `history+`: State variable
- `nfeatures+`: State variable
- `selected_ids`: State variable
- `sensitivities`: State variable

(States enabled by default are listed with +)

**See Also:**

Please refer to the documentation of the base class for more information:

FeatureSelection
Initialize recursive feature elimination

Parameters

- **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 sensitivities are recomputed at each selection step.

- **enable_states** (*None or list of basestring*) – Names of the state variables which should be enabled additionally to default ones.

- **disable_states** (*None or list of basestring*) – Names of the state variables which should be disabled.

## 16.7 Additional Algorithms

### 16.7.1 algorithms.cvtranserror

**Module:** algorithms.cvtranserror

Inheritance diagram for mvpa.algorithms.cvtranserror:
Cross-validate a classifier on a dataset

**CrossValidatedTransferError**

```python
class mvpa.algorithms.cvtranserror.CrossValidatedTransferError (transerror, splitter=None, combiner='mean', expose_testdataset=False, harvest_attribs=None, copy_attribs='copy', **kwargs)
```

**Bases:** `mvpa.measures.base.DatasetMeasure`, `mvpa.misc.state.Harvestable`

Classifier cross-validation.

This class provides a simple interface to cross-validate a classifier on datasets generated by a splitter from a single 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.

**Note:** Available state variables:

- `confusion`: Store total confusion matrix (if available)
- `harvested`: Store specified attributes of classifiers at each split
- `null_prob+`: State variable
- `null_t`: State variable
- `raw_results`: Computed results before applying any transformation algorithm
- `results`: Store individual results in the state
- `samples_error`: Per sample errors.
- `splits`: Store the actual splits of the data. Can be memory expensive
- `training_confusion`: Store total training confusion matrix (if available)
transerrors: Store copies of transerrors at each step. If enabled - operates on clones of transerror, but for the last split original transerror is used.

(States enabled by default are listed with +)

See Also:
Please refer to the documentation of the base classes for more information:

DatasetMeasure, Harvestable

Parameters

- **transerror** *(TransferError instance)* – Provides the classifier used for cross-validation.
- **splitter** *(Splitter | None)* – 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. If None (default) an instance of NoneSplitter is used.
- **combiner** *(Functor | ‘mean’)* – Used to aggregate the error values of all cross-validation folds. If ‘mean’ (default) the grand mean of the transfer errors is computed.
- **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 might 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 | basestr)* – Force copying values of attributes on harvesting
- **enable_states** *(None or list of basestring)* – Names of the state variables which should be enabled additionally to default ones
- **disable_states** *(None or list of basestring)* – Names of the state variables which should be disabled
- **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)* – The estimated distribution is used to assign a probability for a certain value of the computed measure.

**combiner**
Access to the configured combiner.

**splitter**
Access to the Splitter instance.

**transerror**
Access to the TransferError instance.

16.7.2 algorithms.hyperalignment

**Module**: algorithms.hyperalignment

Inheritance diagram for mvpa.algorithms.hyperalignment:
Hyperalignment of functional data to the common space

References: TODO...

see SMLR code for example on how to embed the reference so in future it gets properly referenced...

**Hyperalignment**

class mvpa.algorithms.hyperalignment.Hyperalignment (alignment=None, levels=3, combiner1='mean', combiner2='mean', **kwargs)

Bases: mvpa.misc.state.ClassWithCollections

... Given a set of datasets (may be just data) provide mapping of features into a common space

**Note:** Available state variables:

*who_knows_maybe_something_to_store_optionally: ....

(States enabled by default are listed with +)

**See Also:**

Please refer to the documentation of the base class for more information:

ClassWithCollections

Initialize instance of Hyperalignment

**Parameters**

- **alignment** – ... XXX If *None* (default) an instance of ProcrusteanMapper is used. (Default: *None*)
- **levels** – Number of levels ...XXX. (Default: 3)
- **combiner1** – XXX. (Default: mean)
- **combiner2** – XXX. (Default: mean)
- **enable_states** (*None or list of basestring*) – Names of the state variables which should be enabled additionally to default ones
- **disable_states** (*None or list of basestring*) – Names of the state variables which should be disabled
16.8 Common Facilities

16.8.1 base

Module: base

Inheritance diagram for mvpa.base:

```
mvpa.base.__Singleton  base.verbosity.Logger  mvpa.base.__SingletonType
                   base.verbosity.OnceLogger
                                     mvpa.base.WarningLog
```

Base functionality of PyMVPA

Module Organization

mvpa.base module contains various modules which are used throughout PyMVPA code, and are generic building blocks

```
    group Basic
        externals, config, verbosity, dochelpers
```

WarningLog

class mvpa.base.WarningLog (btlevels=10, btdefault=False, maxcount=1, *args, **kwargs)

Logging class of messages to be printed just once per each message

Define Warning logger.

It is defined by

```
    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
```

mvpa.base.error (msg, critical=True)

Helper function to output errors in a consistent way.
Parameters

- **msg** *(string)* – Actual error message (will be prefixed with ERROR:)
- **critical** *(bool)* – If critical error – exit with

### 16.8.2 base.config

**Module:** base.config

Inheritance diagram for `mvpa.base.config`:

```
ConfigParser.RawConfigParser
  ↓
ConfigParser.ConfigParser
  ↓
ConfigParser.SafeConfigParser
  ↓
base.config.ConfigManager
```

Registry-like monster

**ConfigManager**

class `mvpa.base.config.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. Moreover, 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 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 length 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.

**Parameters**

- **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.

### 16.8.3 base.dochelpers

**Module:** `base.dochelpers`

Various helpers to improve docstrings and textual output

**Functions**

- `mvpa.base.dochelpers.enhancedDocString(item, *args, **kwargs)`

  Generate enhanced doc strings for various items.
Parameters

- **item** (*basestring or class*) – What object requires enhancing of documentation
- **args** (*list*) – Includes base classes to look for parameters, as well, first item must be a dictionary of locals if item is given by a string
- **force_extend** (*bool*) – Either to force looking for the documentation in the parents. By default force_extend = False, and lookup happens only if kwargs is one of the arguments to the respective function (e.g. item.__init__)
- **skip_params** (*list of basestring*) – List of parameters (in addition to [kwargs]) which should not be added to the documentation of the class.

It is to be used from a collector, ie whenever class is already created

```python
mvpa.base.dochelpers.handleDocString(text, polite=True)
```
Take care of empty and non existing doc strings.

```python
mvpa.base.dochelpers.rstUnderline(text, markup)
```
Add and underline RsT string matching the length of the given string.

```python
mvpa.base.dochelpers.singleOrPlural(single, plural, n)
```
Little helper to spit out single or plural version of a word.

```python
mvpa.base.dochelpers.table2string(table, out=None)
```
Given list of lists figure out their common widths and print to out

Parameters

- **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

16.8.4 base.externals

**Module:** base.externals

Helper to verify presence of external libraries and modules

**Functions**

```python
mvpa.base.externals.exists(dep, force=False, raiseException=False, issueWarning=None)
```
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.

Parameters

- **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 or ‘always’*) – Whether to raise RuntimeError if dependency is missing. If True, it is still conditioned on the global setting MVPA_EXTERNALS_RAISE_EXCEPTION, while would raise exception if missing despite the configuration if ‘always’.
**issueWarning** *(string or None or True)* – If string, warning with given message would be thrown. If True, standard message would be used for the warning text.

```python
mvpa.base.externals.testAllDependencies(force=False)
```
Test for all known dependencies.

Parameters

- **force** *(boolean)* – Whether to force the test even if it has already been performed.

### 16.8.5 base.info

**Module:** base.info

Inheritance diagram for `mvpa.base.info`:

#### WTF

```python
class mvpa.base.info.WTF
```
Bases: object

Convenience class to contain information about PyMVPA and OS

TODO: refactor to actually not contain just string representation but rather a dictionary (of dictionaries)

```python
mvpa.base.info.wtf(filename=None)
```
Report summary about PyMVPA and the system

Keywords

- **filename** *(None or string)* If provided, information will be stored in a file, not printed to the screen

### 16.8.6 base.report

**Module:** base.report

Inheritance diagram for `mvpa.base.report`:

#### Report

Creating simple PDF reports using reportlab
Report

class mvpa.base.report.Report (name='report', title=None, path=None, author=None, style='Normal', fig_ext=None, font='Helvetica', page-size=None)

Bases: object

Simple PDF reports using reportlab

Named report ‘report’ generates ‘report.pdf’ and directory ‘report/’ with images which were requested to be included in the report

You can attach report to the existing ‘verbose’ with

    report = Report() verbose.handlers += [report]

and then all verbose messages present on the screen will also be recorded in the report. Use

    report.text("string") to add arbitrary text
    report.xml("<H1>skajdsf</H1>") to add XML snippet

    or    report.figure() to add the current figure to the report. report.figures() to add existing figures to the report, but they

        might not be properly interleaved with verbose messages if there were any between the creations of the figures.

Inspired by Andy Connolly

Initialize report

Parameters

• name (string) – Name of the report
• title (string or None) – Title to start the report, if None, name will be used
• path (string) – Top directory where named report will be stored. Has to be set now to have correct path for storing image renderings. Default: current directory
• author (string or None) – Optional author identity to be printed
• style (string) – Default Paragraph to be used. Must be the one of the known to reportlab styles, e.g. Normal
• fig_ext (string) – What extension to use for figures by default. If None, a default will be used. Since versions prior 2.2 of reportlab might do not support pdf, ‘png’ is default for those, ‘pdf’ otherwise
• font (string) – Name of the font to use
• pagesize (tuple of floats) – Optional page size if not to be default

clear ()

Clear the report

figure (fig=None, name=None, savefig_kwargs={}, **kwargs)

Add a figure to the report

Parameters

• fig (None or string or figure.Figure) – Figure to place into report string : treat as a filename Figure : stores it into a file under directory and embedds into the report None : takes the current figure
• savefig_kwargs (dict) – Additional keyword arguments to provide savefig with (e.g. dpi)
**kwargs – Passed to `reportlab.platypus.Image` constructor

`figures(*args, **kwargs)`

Adds all present figures at once

If called twice, it might add the same figure multiple times, so make sure to close all previous figures if you use `figures()` multiple times

`flowbreak()`

Just a marker for the break of the flow

`save(add_preamble=True)`

Saves PDF

Parameters

`add_preamble (bool)` – Either to add preamble containing title/date/author information

`text(line, **kwargs)`

Add a text string to the report

`write(line, **kwargs)`

Just an alias for `.text`, so we could simply provide report as a handler for verbose

`xml(line, style=None)`

Adding XML string to the report

`mvpa.base.report.escapeXML(s)`

### 16.8.7 base.report_dummy

**Module:** `base.report_dummy`

Inheritance diagram for `mvpa.base.report_dummy`:

```
base.report_dummy.Report
```

Dummy report class, to just be there in case if `reportlab` is not available.

**Report**

```
class mvpa.base.report_dummy.Report(*args, **kwargs):
    Bases: object

    Dummy report class which does nothing but complains if used
    Initialize dummy report
```

### 16.8.8 base.verbosity

**Module:** `base.verbosity`

Inheritance diagram for `mvpa.base.verbosity`:
Verbose output and debugging facility

Examples: from verbosity import verbose, debug; debug.active = [1,2,3]; debug(1, “blah”)

Classes

LevelLogger

class mvpa.base.verbosity.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

Logger

class mvpa.base.verbosity.Logger (handlers=None)
    Bases: object
    Base class to provide logging
    Initialize the logger with a set of handlers to use for output
    Each handler must have write() method implemented

    handlers
        Return active handlers

    lfprev

OnceLogger

class mvpa.base.verbosity.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.
SetLogger

class mvpa.base.verbosity.SetLogger(\(\text{register=None, active=None, printsetid=True, *args, **kwargs}\))

Bases: mvpa.base.verbosity.Logger

Logger which prints based on defined sets identified by Id.

active

print_registered\(\text{detailed=True}\)

printsetid

register \(\text{setid, description}\)

"Register" a new setid with a given description for easy finding

registered

setActiveFromString\(\text{value}\)

Given a string listing registered(?) setids, make them active

16.9 Miscellaneous

16.9.1 misc.args

Module: misc.args

Helpers for arguments handling.

Functions

mvpa.misc.args.group_kwargs\(\text{prefixes, assign=False, passthrough=False}\)

Decorator function to join parts of kwargs together

Parameters

*prefixes \(\text{list of basestrings}\) – Prefixes to split based on. See split_kwargs

*assign \(\text{bool}\) – Flag to assign the obtained arguments to self._<prefix>_kwargs

*passthrough \(\text{bool}\) – 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<underscore>’ together under slave_kwargs parameter

mvpa.misc.args.split_kwargs\(\text{kwargs, prefixes=[]}\)

Helper to separate kwargs into multiple groups

Parameters

prefixes \(\text{list of basestrings}\) – 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
16.9.2 misc.attributes

Module: misc.attributes

Inheritance diagram for mvpa.misc.attributes:

```plaintext
+------------------+
| misc.attributes.CollectableAttribute |
+------------------+
| misc.attributes.AttributeWithUnique |
| misc.attributes.StateVariable |
+------------------+
| misc.attributes.FeatureAttribute |
| misc.attributes.SampleAttribute |
| misc.attributes.DatasetAttribute |
```

Module with some special objects to be used as magic attributes with dedicated containers aka. Collections.

Classes

AttributeWithUnique

class mvpa.misc.attributes.AttributeWithUnique (name=None, hasunique=True, doc='Attribute with unique')

Bases: mvpa.misc.attributes.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).

Pros:

- don’t need to mess with getattr since it would become just another attribute

Cons:

- 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

CollectableAttribute

class mvpa.misc.attributes.CollectableAttribute (name=None, doc=None, index=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 grouped into corresponding collections for each class by statecollector metaclass, ie it would be done on a class creation (ie not per each object)

```python
isSet
name
reset()
    Simply reset the flag
value
```

### DatasetAttribute

```python
class mvpa.misc.attributes.DatasetAttribute(name=None, hasunique=True, doc='Attribute with unique')
Bases: mvpa.misc.attributes.AttributeWithUnique
```

### FeatureAttribute

```python
class mvpa.misc.attributes.FeatureAttribute(name=None, hasunique=True, doc='Attribute with unique')
Bases: mvpa.misc.attributes.AttributeWithUnique
```

### SampleAttribute

```python
class mvpa.misc.attributes.SampleAttribute(name=None, hasunique=True, doc='Attribute with unique')
Bases: mvpa.misc.attributes.AttributeWithUnique
```

### StateVariable

```python
class mvpa.misc.attributes.StateVariable(name=None, enabled=True, doc='State variable')
Bases: mvpa.misc.attributes.CollectableAttribute
Simple container intended to conditionally store the value
enable(value=False)
isEnabled
reset()
    Simply detach the value, and reset the flag
```

## 16.9.3 misc.bv.base

### Module: misc.bv.base

Inheritance diagram for `mvpa.misc.bv.base`: 

---

284 Chapter 16. Module Reference
Tiny snippets to interface with FSL easily.

**BrainVoyagerRTC**

```python
class mvpa.misc.bv.base.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

Data is always read as float and header is actually ignored

Read and write BrainVoyager RTC files.

Parameters

source (filename of an RTC file) –

toarray ()

Returns the data as an array

### 16.9.4 misc.cmdline

**Module:** misc.cmdline

Inheritance diagram for mvpa.misc.cmdline:

```
misc.cmdline.Options  misc.cmdline.OptionGroups
```

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.

**Classes**

**OptionGroups**

```python
class mvpa.misc.cmdline.OptionGroups(parser)
    Bases: object
    Group creation is delayed until instance is requested.
    This allows to overcome the problem of polluting handled cmdline options
    add(name, l, doc)
```

**Options**

```python
class mvpa.misc.cmdline.Options
    Bases: object
    Just a convinience placeholder for all available options
```

### 16.9.5 misc.data_generators

**Module:** `misc.data_generators`

Miscelaneous data generators for unittests and demos

**Functions**

```python
mvpa.misc.data_generators.chirpLinear(n_instances, n_features=4, n_nonbogus_features=2, data_noise=0.4, noise=0.1)
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
```

```python
mvpa.misc.data_generators.dumbFeatureBinaryDataset()
Very simple binary (2 labels) dataset
```

```python
mvpa.misc.data_generators.dumbFeatureDataset()
Create a very simple dataset with 2 features and 3 labels
```

```python
mvpa.misc.data_generators.getMVPattern(s2n)
Simple multivariate dataset
```

```python
mvpa.misc.data_generators.linear1d_gaussian_noise(size=100, slope=0.5, intercept=1.0, x_min=-2.0, x_max=3.0, sigma=0.2)
A straight line with some Gaussian noise.
```

```python
mvpa.misc.data_generators.linear_awgn(size=10, intercept=0.0, slope=0.4, noise_std=0.01, flat=False)
Generate a dataset from a linear function with AWGN (Added White Gaussian Noise).
It can be multidimensional if ‘slope’ is a vector. If flat is True (in 1 dimension) generate equally spaces
samples instead of random ones. This is useful for the test phase.
```
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

mvpa.misc.data_generators.noisy_2d_fx(size_per_fx, dfx, sfx, center, noise_std=1)

mvpa.misc.data_generators.normalFeatureDataset(perlabel=50, nlabels=2, nfeatures=4, nchunks=5, means=None, nonbogus_features=None, snr=3.0)

Generate a univariate dataset with normal noise and specified means.

Keywords

perlabel
    [int] Number of samples per each label

nlabels
    [int] Number of labels in the dataset

nfeatures
    [int] Total number of features (including bogus features which carry no label-related signal)

nchunks
    [int] Number of chunks (perlabel should be multiple of nchunks)

means
    [None or list of float or ndarray] Specified means for each of features among nfeatures.

nonbogus_features
    [None or list of int] Indexes of non-bogus features (1 per label)

snr [float] Signal-to-noise ration assuming that signal has std 1.0 so we just divide random normal 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

mvpa.misc.data_generators.normalFeatureDataset__(dataset=None, labels=None, nchunks=None, perlabel=50, activation_probability_steps=1, randomseed=None, randomvoxels=False)

NOT FINISHED

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\%
\$

mvpa.misc.data_generators.sinModulated(n_instances, n_features, flat=False, noise=0.4)

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

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: \(\ell_1 = 1.804377\) \(\ell_2 = 1.963956\) \(\ell_3 = 8.884361\) \(\ell_4 = 34.417657\) \(\ell_5 = 1081.610451\) \(\ell_6 = 375.445823\) \(\sigma_{\text{f}} = 2.379139\) \(\sigma_{\text{n}} = 0.050835\)

### 16.9.6 misc.errorfx

**Module: misc.errorfx**

Inheritance diagram for `mvpa.misc.errorfx`:

```
  misc.errorfx.MeanMismatchErrorFx
  |______________________________|
  |______________________________|
  |misc.errorfx._ErrorFx         |
  v                                v
misc.errorfx.CorrErrorPFx
  |______________________________|
  | misc.errorfx.Variance1SVFx   |
  v                                v
misc.errorfx.CorrErrorFx
  |______________________________|
  | misc.errorfx.AUCErrorFx      |
  v                                v
misc.errorfx.RelativeRMSErrorFx
  |______________________________|
  | misc.errorfx.RMSErrorFx       |
```

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’

#### Classes

**AUCErrorFx**

```python
class mvpa.misc.errorfx.AUCErrorFx
    Bases: mvpa.misc.errorfx._ErrorFx

    Computes the area under the ROC for the given the target and predicted to make the prediction.
```

**MeanMismatchErrorFx**

```python
class mvpa.misc.errorfx.MeanMismatchErrorFx
    Bases: mvpa.misc.errorfx._ErrorFx

    Computes the percentage of mismatches between some target and some predicted values.
```

**RMSErrorFx**

```python
class mvpa.misc.errorfx.RMSErrorFx
    Bases: mvpa.misc.errorfx._ErrorFx

    Computes the root mean squared error of some target and some predicted values.
```

**RelativeRMSErrorFx**

```python
class mvpa.misc.errorfx.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.

**Variance1SVFx**

```python
class mvpa.misc.errorfx.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
```

**Functions**

```python
mvpa.misc.errorfx.meanPowerFx(data)
    Returns mean power

    Similar to var but without demeaning

mvpa.misc.errorfx.rootMeanPowerFx(data)
    Returns root mean power

    to be comparable against RMSE
```

### 16.9.7 misc.exceptions

**Module:** `misc.exceptions`

Inheritance diagram for `mvpa.misc.exceptions`:

Exception classes which might get thrown

**Classes**

**ConvergenceError**

```python
class mvpa.misc.exceptions.ConvergenceError
    Bases: exceptions.Exception

    Thrown if some algorithm does not converge to a solution.
```

**DatasetError**

```python
class mvpa.misc.exceptions.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
```
InvalidHyperparameterError

class mvpa.misc.exceptions.InvalidHyperparameterError
   Bases: exceptions.Exception
       Generic exception to be raised when setting improper values as hyperparameters.

UnknownStateError

class mvpa.misc.exceptions.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.

16.9.8 misc.fsl.base

Module: misc.fsl.base

Inheritance diagram for mvpa.misc.fsl.base:

```
    io.base.ColumnData   fsl.base.FslGLMDesign
                  /                /
         |                  |
 fsl.base.McFlirtParams fsl.base.FslEV3
```

Tiny snippets to interface with FSL easily.

Classes

FslEV3

class mvpa.misc.fsl.base.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.
    Parameters
       source (filename of an EV3 file) –
       durations
getEV (evid)
    Returns a tuple of (onset time, stimulus duration, intensity) for a certain EV.

getNEVs ()
    Returns the number of EVs in the file.

intensities

nevs
    Returns the number of EVs in the file.

onsets
toEvents (**kwargs)
    Convert into a list of Event instances.

    Parameters
    \n    **kwargs – Any keyword argument provided would be replicated, through all the entries.
    Useful to specify label or even a chunk

tofile (filename)
    Write data to a FSL EV3 file.

FslGLMDesign
class mvpa.misc.fsl.base.FslGLMDesign (source)
    Bases: object
    Load FSL GLM design matrices from file.
    Be aware that such a design matrix has its regressors in columns and the samples in its rows.

    Parameters
    \n    source (filename) – Compressed files will be read as well, if their filename ends with ‘.gz’.

plot (style=’lines’, **kwargs)
    Visualize the design matrix.

    Parameters

    *style (‘lines’, ‘matrix’) –
    **kwargs – Additional arguments will be passed to the corresponding matplotlib plotting functions ‘plot()’ and ‘pcolor()’ for ‘lines’ and ‘matrix’ plots respectively.

McFlirtParams
class mvpa.misc.fsl.base.McFlirtParams (source)
    Bases: mvpa.misc.io.base.ColumnData
    Read and write McFlirt’s motion estimation parameters from and to text files.

    Initialize McFlirtParams

    Parameters

    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.

Function

mvpa.misc.fsl.base.read_fsl_design(fsf_file)
    Reads an FSL FEAT design.fsf file and return the content as a dictionary.

Parameters

    fsf_file (filename, file-like) –

16.9.9 misc.fsl.flobs

Module: misc.fsl.flobs

Wrapper around FSL’s halfcosbasis to generate HRF kernels

mvpa.misc.fsl.flobs.makeFlobs(pre=0, rise=5, fall=5, undershoot=5, undershootamp=0.3, nsamples=1, resolution=0.05, 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.

```
^          /-
 |          /  \                  | undershootamp
 |          /    \                  |
 |          /      \               |
 |          /  -----\             |
 |          \         |             |
 |          \    undershootamp   |
 |          \              |
 |          \              |
```

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.

16.9.10 misc.fsl.melodic

Module: misc.fsl.melodic

Inheritance diagram for mvpa.misc.fsl.melodic:

```python
fsl.melodic.MelodicResults
```
Wrapper around the output of MELODIC (part of FSL)

**MelodicResults**

```python
class mvpa.misc.fsl.melodic.MelodicResults(path)

Bases: object
```

Easy access to MELODIC output.

Only important information is available (important as judged by the author).

Reads all information from the given MELODIC output path.

- `funcdata`
- `ic`
- `icastats`
- `nic`
- `path`
- `relvar_per_ic`
- `smodes`
- `tmodes`
- `tr`
- `truevar_per_ic`

### 16.9.11 misc.fx

**Module:** misc.fx

Misc. functions (in the mathematical sense)

#### Functions

- `mvpa.misc.fx.doubleGammaHRF(t, A1=5.4, W1=5.2, K1=1.0, A2=10.8, W2=7.35, K2=0.35)`

  Hemodynamic response function model.

  The version is using two gamma functions (also see singleGammaHRF()).

  Parameters

  - `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.

- `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).
Parameters

• **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 additional 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).

```python
mvpa.misc.fx.singleGammaHRF(t, A=5.4, W=5.2, K=1.0)
```

Hemodynamic response function model.

The version consists of a single gamma function (also see `doubleGammaHRF()`).

Parameters

• **t** (*float*) – Time.

• **A** (*float*) – Time to peak.

• **W** (*float*) – Full-width at half-maximum.

• **K** (*float*) – Scaling factor.

16.9.12 misc.io.base

Module: **misc.io.base**

Inheritance diagram for `mvpa.misc.io.base`:  

---

294 Chapter 16. Module Reference
Some little helper for reading (and writing) common formats from and to disk.

Classes

ColumnData

class mvpa.misc.io.base.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 artifical 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.

Parameters

- **source** (basestring or dict) – If values is given as a string all data is read from the file and additional 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.

Parameters

• filename (basestring) – Target filename
• header (bool) – If True a column header is written, using the column keys. If False no header is written.
• header_order (None or list of basestring) – 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 (basestring) – String that is written as a separator between to data columns.

DataReader

class mvpa.misc.io.base.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

SampleAttributes

class mvpa.misc.io.base.SampleAttributes(source, literallabels=False, header=None)  
Bases: mvpa.misc.io.base.ColumnData
Read and write PyMVPA sample attribute definitions from and to text files.

Read PyMVPA sample attributes from disk.

Parameters

*source* *(basestring)* — Filename of an attribute file

*literallabels* *(bool)* — Either labels are given as literal strings

*header* *(None or bool or list of str)* — If None, [‘labels’, ‘chunks’] is assumed. Otherwise the same behavior as of ColumnData

getNSamples ()

Returns the number of samples in the file.

nsamples

Returns the number of samples in the file.

toEvents (**kwargs)**

Convert into a list of Event instances.

Each change in the label or chunks value is taken as a new event onset. The length of an event is determined by the number of identical consecutive label-chunk combinations. Since the attributes list has no sense of absolute timing, both onset and duration are determined and stored in #samples units.

Parameters

**kwargs** — Any keyword argument provided would be replicated, through all the entries.

tofile *(filename)*

Write sample attributes to a text file.

SensorLocations

class mvpa.misc.io.base.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* attributes.

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)

TuebingenMEGSensorLocations

class mvpa.misc.io.base.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, i.e. 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.

Parameters

- `source (filename of an attribute file)` –

**XAVRSensorLocations**

```python
class mvpa.misc.io.base.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.

Parameters

- `source (filename of an attribute file)` –

**Functions**

```python
def design2labels(columndata, baseline_label=0, func=<function <lambda> at 0x86b8668>)
```

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

Parameters

- `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

```python
def labels2chunks(labels, method='alllabels', ignore_labels=None)
```

**16.9.13 misc.io.eepbin**

Module: `misc.io.eepbin`

Inheritance diagram for `mvpa.misc.io.eepbin`: 
Reader for binary EEP files.

**EEPBin**

```python
class mvpa.misc.io.eepbin.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>,...
```

Read EEP file and store header and data.

- **Parameters**
  - `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
16.9.14 misc.io.hamster

Module: misc.io.hamster

Inheritance diagram for mvpa.misc.io.hamster:

```
io.hamster.Hamster
```

Helper for simple storage facility via cPickle and optionally zlib

**Hamster**

class mvpa.misc.io.hamster.Hamster(*args, **kwargs)

Bases: object

Simple container class with basic IO capabilities.

It is capable of storing itself in a file, or loading from a file using cPickle (optionally via zlib from compressed files). Any serializable object can be bound to a hamster to be stored.

To undig buried hamster use Hamster(filename). Here is an example:

```python
>>> h = Hamster(bla='blai')
>>> h.boo = N.arange(5)
... h.dump(filename)
...
>>> h = Hamster(filename)
```

Since Hamster introduces methods *dump, asdict* and property ‘registered’, those names cannot be used to assign an attribute, nor provided in among constructor arguments.

Initialize Hamster.

Providing a single parameter string would treat it as a filename from which to undig the data. Otherwise all keyword parameters are assigned into the attributes of the object.

**asdict** ()

Return registered data as dictionary

**dump** (filename, compresslevel='auto')

Bury the hamster into the file

Parameters

- **filename** (str) – Name of the target file. When writing to a compressed file the filename gets a ‘.gz’ extension if not already specified. This is necessary as the constructor uses the extension to decide whether it loads from a compressed or uncompressed file.

- **compresslevel** (‘auto’ or int) – Compression level setting passed to gzip. When set to ‘auto’, if filename ends with ‘.gz’ compresslevel is set to 5, 0 otherwise. However, when compresslevel is set to 0 gzip is bypassed completely and everything is written to an uncompressed file.

**registered**

List registered attributes.
16.9.15 misc.io.meg

**Module:** misc.io.meg

Inheritance diagram for mvpa.misc.io.meg:

```
io.meg.TuebingenMEG
```

IO helper for MEG datasets.

**TuebingenMEG**

```python
class mvpa.misc.io.meg.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 unknown 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 textfiles or file-like objects.

**Parameters**

- `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.

16.9.16 misc.param

**Module:** misc.param

Inheritance diagram for mvpa.misc.param:
Parameter representation

**Classes**

**KernelParameter**

```python
class mvpa.misc.param.KernelParameter (default, name=None, doc=None, index=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

**Parameter**

```python
class mvpa.misc.param.Parameter (default, name=None, doc=None, index=None, **kwargs)
```

Bases: `mvpa.misc.attributes.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

calss.parameter1[1] = ...

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**
Docstring for the parameter to be used in lists of parameters

**Return type**

- string or list of strings (if `indent` is None)

**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**

### 16.9.17 misc.plot.base

**Module:** misc.plot.base

Misc. plotting helpers.

#### Functions

- `mvpa.misc.plot.base.inverseCmap(cmap_name)`
  
  Create a new colormap from the named colormap, where it got reversed

- `mvpa.misc.plot.base.plotBars(data, labels=None, title=None, ylim=None, ylabel=None, width=0.2, offset=0.2, color='0.6', distance=1.0, yerr='ste', **kwargs)`
  
  Make bar plots with automatically computed error bars.

  Candlestick plot (multiple interleaved barplots) can be done, by calling this function multiple time with appropriately modified `offset` argument.

  **Parameters**

  - `data` *(array (nbars x nobservations) | other sequence type)* – Source data for the barplot. Error measure is computed along the second axis.
  - `labels` *(list | None)* – If not None, a label from this list is placed on each bar.
  - `title` *(str)* – An optional title of the barplot.
  - `ylim` *(2-tuple)* – Y-axis range.
  - `ylabel` *(str)* – An optional label for the y-axis.
  - `width` *(float)* – Width of a bar. The value should be in a reasonable relation to `distance`.
  - `offset` *(float)* – Constant offset of all bar along the x-axis. Can be used to create candlestick plots.
  - `color` *(matplotlib color spec)* – Color of the bars.
  - `distance` *(float)* – Distance of two adjacent bars.
  - `yerr` *(‘ste’ | ‘std’ | None)* – Type of error for the errorbars. If `None` no errorbars are plotted.
  - `**kwargs` – Any additional arguments are passed to matplotlib’s `bar()` function.
mvpa.misc.plot.base.plotDatasetChunks(ds, clf_labels=None)
    Quick plot to see chunk structure in dataset with 2 features
    if clf_labels is provided for the predicted labels, then incorrectly labeled samples will have ‘x’ in them

mvpa.misc.plot.base.plotErrLine(data, x=None, errtype='ste', curves=None, linestyle='-',
    fmt='o', perc_sigchg=False, baseline=None)
    Make a line plot with errorbars on the data points.
    Parameters

    • data (sequence of sequences) – First axis separates samples and second axis will appear as x-axis in the plot.
    • x (sequence) – Value to be used as ‘x-values’ corresponding to the elements of the 2nd axis id data. If None, a sequence of ascending integers will be generated.
    • errtype (‘ste’ | ‘std’) – Type of error value to be computed per datapoint. ‘ste’: standard error of the mean ‘std’: standard deviation
    • curves (None | list of tuple(x, y)) – Each tuple represents an additional curve, with x and y coordinates of each point on the curve.
    • linestyle (str) – matplotlib linestyle argument. Applied to either the additional curve or a the line connecting the datapoints. Set to ‘None’ to disable the line completely.
    • fmt (str) – matplotlib plot style argument to be applied to the data points and errorbars.
    • perc_sigchg (bool) – If True the plot will show percent signal changes relative to a baseline.
    • baseline (float | None) – Baseline used for converting values into percent signal changes. If None and perc_sigchg is True, the absolute of the mean of the first feature (i.e. [:,0]) will be used as a baseline.

Make dataset with 20 samples from a full sinus wave period, computed 100 times with individual noise pattern.
>>> x = N.linspace(0, N.pi * 2, 20)
>>> data = N.vstack([N.sin(x)] * 30)
>>> data += N.random.normal(size=data.shape)
Now, plot mean data points with error bars, plus a high-res version of the original sinus wave.
>>> x = N.linspace(0, N.pi * 2, 200)
>>> plotErrLine(data, curves=[(x, N.sin(x))])
>>> #P.show()

mvpa.misc.plot.base.plotFeatureHist(dataset, xlim=None, noticks=True, perchunk=False,
    **kwargs)
    Plot histograms of feature values for each labels.
    Parameters

    • dataset (Dataset) –
    • xlim (None | 2-tuple) – Common x-axis limits for all histograms.
    • noticks (boolean) – If True, no axis ticks will be plotted. This is useful to save space in large plots.
    • perchunk (boolean) – If True, one histogramm will be plotted per each label and each chunk, resulting is a histogram grid (labels x chunks).
    • **kwargs – Any additional arguments are passed to matplotlib’s hist().

mvpa.misc.plot.base.plotSamplesDistance(dataset, sortbyattr=None)
    Plot the euclidean distances between all samples of a dataset.
Parameters

- **dataset** *(Dataset)* – Providing the samples.
- **sortbyattr** *(None | str)* – If None, the samples distances will be in the same order as their appearance in the dataset. Alternatively, the name of a samples attribute can be given, which will then be used to sort/group the samples, e.g. to investigate the similarity samples by label or by chunks.

### 16.9.18 misc.plot.erp

**Module: misc.plot.erp**

Basic ERP (here ERP = Event Related Plot ;-) ) plotting

Can be used for plotting not only ERP but any event-locked data

**Functions**

**mvpa.misc.plot.erp.plotERP** *(data, SR=500, onsets=None, pre=0.2, pre_onset=None, post=None, pre_mean=None, color='r', errorcolor=None, errtype=None, ax=<module 'pylab' from '/usr/lib/pymodules/python2.7/pylab.pyc'>, ymult=1.0, *args, **kwargs)*

Plot single ERP on existing canvas

**Parameters**

- **data** *(1D or 2D ndarray)* – The data array can either be 1D (samples over time) or 2D (trials x samples). In the first case a boxcar mapper is used to extract the respective trial timecourses given a list of trial onsets. In the latter case, each row of the data array is taken as the EEG signal timecourse of a particular trial.
- **onsets** *(list(int))* – List of onsets (in samples not in seconds).
- **SR** *(int)* – Sampling rate (1/s) of the signal.
- **pre** *(float)* – Duration (in seconds) to be plotted prior to onset.
- **pre_onset** *(float or None)* – If data is already in epochs (2D) then pre_onset provides information on how many seconds pre-stimulus were used to generate them. If None, then pre_onset = pre
- **post** *(float)* – Duration (in seconds) to be plotted after the onset.
- **pre_mean** *(float)* – Duration (in seconds) at the beginning of the window which is used for deriving the mean of the signal. If None, pre_mean = pre
- **errtype** *(None | ‘ste’ | ‘std’ | ‘ci95’ | list of previous three)* – Type of error value to be computed per datapoint. ‘ste’: standard error of the mean ‘std’: standard deviation ‘ci95’: 95% confidence interval (1.96 * ste) None: no error margin is plotted (default) Optionally, multiple error types can be specified in a list. In that case all of them will be plotted.
- **color** *(matplotlib color code)* – Color to be used for plotting the mean signal timecourse.
- **errorcolor** *(matplotlib color code)* – Color to be used for plotting the error margin. If None, use main color but with weak alpha level
- **ax** – Target where to draw.
- **ymult** *(float)* – Multiplier for the values. E.g. if negative-up ERP plot is needed: provide ymult=-1.0
**kwargs (*args) – Additional arguments to plot().

Return type
array

Returns
Mean ERP timeseries.

`mvpa.misc.plot.erp.plotERPs` `erps`, `data=None`, `ax=None`, `pre=0.2`, `post=None`, `pre_onset=None`, `xlabel='time (s)'`, `ylabel='\mu V$', `ylim=None`, `ymult=1.0`, `legend=None`, `xformat='%4g', `yformat='\%4g', `loffset=10`, `alinewidth=2`, **kwargs)

Plot multiple ERPs on a new figure.

Parameters

- **erps** (list of tuples) – List of definitions of ERPs. Each tuple should consist of (label, color, onsets) or a dictionary which defines, label, color, onsets, data. Data provided in dictionary overrides ‘common’ data provided in the next argument `data`
- **data** – Data for ERPs to be derived from 1D (samples)
- **ax** – Where to draw (e.g. subplot instance). If None, new figure is created
- **pre** (float) – Duration (seconds) to be plotted prior to onset
- **pre_onset** (float or None) – If data is already in epochs (2D) then pre_onset provides information on how many seconds pre-stimulus were used to generate them. If None, then pre_onset = pre
- **post** (float or None) – Duration (seconds) to be plotted after the onset. If any data is provided with onsets, it can’t be None. If None – plots all time points after onsets
- **ymult** (float) – Multiplier for the values. E.g. if negative-up ERP plot is needed: provide ymult=-1.0
- **xformat** (basestring) – Format of the x ticks
- **yformat** (basestring) – Format of the y ticks
- **legend** (basestring or None) – If not None, legend will be plotted with position argument provided in this argument
- **loffset** (int) – Offset in voxels for axes and tick labels. Different matplotlib frontends might have different opinions, thus offset value might need to be tuned specifically per frontend
- **alinewidth** (int) – Axis and ticks line width
- **kwargs** – Additional arguments provided to plotERP()

```python
kwargs = {'SR' : eeg.SR, 'pre_mean' : 0.2}
fig = plotERPs(((‘60db’, ‘b’, eeg.erp_onsets[‘60db’]),
               (‘80db’, ‘r’, eeg.erp_onsets[‘80db’])),
data[:, eeg.sensor_mapping[‘Cz’]],
ax=fig.add_subplot(1,1,1,frame_on=False), pre=0.2,
post=0.6, **kwargs)
or
fig = plotERPs(((‘60db’, ‘b’, eeg.erp_onsets[‘60db’]),
               (‘80db’, ‘r’, eeg.erp_onsets[‘80db’]),
               (‘onsets’: eeg.erp_onsets[‘80db’],
               ‘data’ : data[:, eeg.sensor_mapping[‘Cz’]])
               ),
data[:, eeg.sensor_mapping[‘Cz’]],
ax=fig.add_subplot(1,1,1,frame_on=False), pre=0.2,
post=0.6, **kwargs)
```
Returns
current fig handler

16.9.19 misc.plot.mri

Module: misc.plot.mri

Basic (f)MRI plotting with ability to interactively perform thresholding

```
mvpa.misc.plot.mri.plotMRI (background=None, background_mask=None, cmap_bg='gray',
    overlay=None, overlay_mask=None, cmap_overlay='autumn',
    vlim=(0.0, None), vlim_type=None, do_stretch_colors=False,
    add_info=True, add_hist=True, add_colorbar=True, fig=None,
    interactive=None, nrows=None, ncolumns=None)
```

Very basic plotting of 3D data with interactive thresholding.

Background/overlay could be nifti files names or NiftiImage objects, or 3D ndarrays. if no mask provided,
only non-0 elements are plotted

Parameters

- **do_stretch_colors** (bool) – Stretch color range to the data (not just to visible data)

- **vlim** – 2 element tuple of low/upper bounds of values to plot

- **vlim_type** (None or ‘symneg_z’) – If not None, then vlim would be treated accordingly:
symneg_z z-score values of symmetric normal around 0, estimated by symmetrizing
negative part of the distribution, which often could be assumed when total distribution
is a mixture of by-chance performance normal around 0, and some other in the positive
tail

- **ncolumns** (int or None) – Explicit starting number of columns into which position the
slice renderings. If None, square arrangement would be used

- **nrows** (int or None) – Explicit starting number of rows into which position the slice
renderings. If None, square arrangement would be used

- **add_hist** (bool or tuple (int, int)) – If True, add histogram and position automagically.
If a tuple – use as (row, column)

- **add_info** (bool or tuple (int, int)) – If True, add information and position automagically.
If a tuple – use as (row, column).

Available colormaps are presented nicely on
http://www.scipy.org/Cookbook/Matplotlib/Show_colormaps

TODO:

- Make interface more attractive/usable
- allow multiple overlays... or just unify for them all to be just a list of entries
- handle cases properly when there is only one - background/overlay

16.9.20 misc.plot.topo

Module: misc.plot.topo

Plot parameter distributions on a head surface (topography plots).
**Functions**

**mvpa.misc.plot.topo.plotHeadOutline**(scale=1, shift=(0, 0), color='k', linewidth='5', **kwargs)

Plots a simple outline of a head viewed from the top. The plot contains schematic representations of the nose and ears. The size of the head is basically a unit circle for nose and ears attached to it.

**Parameters**

- **scale** *(float)* – Factor to scale the size of the head.
- **shift** *(2-tuple of floats)* – Shift the center of the head circle by these values.
- **color** *(matplotlib color spec)* – The color the outline should be plotted in.
- **linewidth** *(int)* – Linewidth of the head outline.
- **kwargs** – All additional arguments are passed to *pylab.plot()*.

**Return type**
Matplotlib lines2D object

**Returns**
can be used to tweak the look of the head outline.

**mvpa.misc.plot.topo.plotHeadTopography**(topography, sensorlocations, plotsensors=False, resolution=51, masked=True, plothead=True, plothead_kwargs=None, **kwargs)

Plot distribution to a head surface, derived from some sensor locations. The sensor locations are first projected onto the best fitting sphere and finally projected onto a circle (by simply ignoring the z-axis).

**Parameters**

- **topography** *(array)* – A vector of some values corresponding to each sensor.
- **sensorlocations** *((nsensors x 3) array)* – 3D coordinates of each sensor. The order of the sensors has to match with the *topography* vector.
- **plotsensors** *(bool)* – If True, sensor will be plotted on their projected coordinates. No sensor are shown otherwise.
- **plothead** *(bool)* – If True, a head outline is plotted.
- **plothead_kwargs** *(dict)* – Additional keyword arguments passed to *plotHeadOutline()*.
- **resolution** *(int)* – Number of surface samples along both x and y-axis.
- **masked** *(bool)* – If True, all surface sample extending to head outline will be masked.
- **kwargs** – All additional arguments will be passed to *pylab.imshow()*.

**Return type**
*(map, head, sensors)*

**Returns**
The corresponding matplotlib objects are returned if plotted, i.e. if plothead is set to *False*, *head* will be *None*. *map* The colormap that makes the actual plot, a matplotlib.imageAxesImage instance. *head* What is returned by *plotHeadOutline()*.

* sensors The dots marking the electrodes, a matplotlib.linesLine2d instance.
16.9.21 misc.state

Module: misc.state

Inheritance diagram for mvpa.misc.state:

Classes to control and store state information.
It was devised to provide conditional storage

Classes

AttributesCollector

class mvpa.misc.state.AttributesCollector (name, bases, dict)

Bases: type

Intended to collect and compose StateCollection for any child class of this metaclass

ClassWithCollections

class mvpa.misc.state.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()

Collection

class mvpa.misc.state.Collection (items=None, owner=None, name=None)

Bases: object

Container of some CollectableAttributes.

Groups

• 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

Parameters

- \*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

Parameters

- 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.

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

Parameters

- index (None or basestring or list of basestring) – What items to check if they were set in the collection

items
listing = ‘property’
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

whichSet()
Return list of indexes which were set

Harvestable

class mvpamiscstate\_Harvestable (harvest\_attribs=None, copy\_attribs=’copy’, **kwargs)
Bases: mvpamiscstate\_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 intermediate possibly interesting results (like in case of CrossValidatedTransferError and states of the trained classifier or TransferError).

**Note:** Available state variables:

- *harvested*: Store specified attributes of classifiers at each split

(Sources enabled by default are listed with `+`)

**See Also:**
Please refer to the documentation of the base class for more information:

ClassWithCollections

Initialize state of harvestable

Parameters

- **harvest_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
- **enable_states** *(None or list of basestring)* – Names of the state variables which should be enabled additionally to default ones
- **disable_states** *(None or list of basestring)* – Names of the state variables which should be disabled

**harvest_attribs**

**ParameterCollection**

```python
class mvpa.misc.state.ParameterCollection(items=None, owner=None, name=None)
    Bases: mvpa.misc.state.Collection

    Container of Parameters for a stateful object.
    Initialize the Collection
    Parameters
```

- **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

**SampleAttributesCollection**

```python
class mvpa.misc.state.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
```

16.9. Miscellaneous
Parameters

• *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’)

**StateCollection**

```python
class mvpa.misc.state.StateCollection(items=None, owner=None)
Bases: mvpa.misc.state.Collection
```

Container of StateVariables for a stateful object.

Groups

• **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

Parameters

• *items* (*dict*) – dictionary of states
• *owner* (*ClassWithCollections*) – 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

Parameters

• *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

### 16.9.22 misc.stats

**Module**: `misc.stats`

Inheritance diagram for `mvpa.misc.stats`:
Little statistics helper

**DSMatrix**

class `mvpa.misc.stats.DSMatrix(data_vectors, metric='spearman')`

Bases: `object`

DSMatrix allows for the creation of dissimilarity matrices using arbitrary distance metrics.

Initialize DSMatrix

Parameters

- **data_vectors** *(ndarray)* – m x n collection of vectors, where m is the number of exemplars and n is the number of features per exemplar

- **metric** *(string)* – Distance metric to use (e.g., ‘euclidean’, ‘spearman’, ‘pearson’, ‘confusion’)

getFullMatrix()

getMetric()

getTriangle()

getVectorForm()

`mvpa.misc.stats.chisquare(obs, exp=None)`

Compute the chisquare value of a contingency table with arbitrary dimensions.

If no expected frequencies are supplied, the total N is assumed to be equally distributed across all cells.

Returns: chisquare-stats, associated p-value (upper tail)

---

**16.9.23 misc.support**

**Module:** `misc.support`

Inheritance diagram for `mvpa.misc.support`:

```
      misc.support.SmartVersion    misc.support.MapOverlap    misc.support.Event
        misc.support.Harvester
```

Support function – little helpers in everyday life
Classes

Event

```python
class mvpa.misc.support.Event(**kwargs):
    Bases: dict
    Simple class to define properties of an event.
    The class is basically a dictionary. Any properties can be passed 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.

    asDiscreteTime(dt, storeoffset=False)
    Convert onset and duration information into discrete timepoints.
    Parameters

    • 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 discrete 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!
```

Harvester

```python
class mvpa.misc.support.Harvester(source, calls, simplify_results=True)
    Bases: object
    World domination helper: do whatever it is asked and accumulate results
    XXX Thinks about:

    • Might we need to deepcopy attributes values?
    • Might we need to specify what attribs to copy and which just to bind?

    Initialize
```
Parameters

- **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 appearance in the sequence.
- **simplify_results** *(bool)* – Remove unnecessary overhead in results if possible (nested lists and dictionaries).

**HarvesterCall**

class mvpa.misc.support.HarvesterCall(call, attribs=None, argfilter=None, expand_args=True, copy_attribs=True)

Initialize

Parameters

- **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

**call** = None

Call which gets called in the harvester.

**MapOverlap**

class mvpa.misc.support.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.

**SmartVersion**

class mvpa.misc.support.SmartVersion(vstring=None)

Bases: distutils.version.Version

A bit evolved comparison of versions
The reason for not using python’s distutil.version is that it seems to have no clue about somewhat common conventions of using ‘-dev’ or ‘dev’ or ‘rc’ suffixes for upcoming releases (so major version does contain upcoming release already).

So here is an ad-hoc and not as nice implementation

\[
\text{parse(vstring)}
\]

### Functions

- **mvpa.misc.support.RFEHistory2maps(history)**
  Convert history generated by RFE into the array of binary maps
  
  Example:
  
  \[
  \text{history2maps(N.array([ 3,2,1,0 ]))}
  \]
  
  results in

  \[
  \text{array([[ 1., 1., 1., 1.], [ 1., 1., 1., 0.], [ 1., 1., 0., 0.], [ 1., 0., 0., 0.]])}
  \]

- **mvpa.misc.support.getBreakPoints(items, contiguous=True)**
  Return a list of break points.

  Parameters

  - **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

- **mvpa.misc.support.getUniqueLengthNCombinations(L, n=None, sort=True)**
  Find all subsets of data

  Parameters

  - **L (list)** – list of unique ids
  - **n (None or int)** – If None, all possible subsets are returned. If n is specified (int), then only the ones of the length n are returned

  TODO: work out single stable implementation – probably just by fixing _getUniqueLengthNCombinations_l13

- **mvpa.misc.support.idhash(val)**
  Craft unique id+hash for an object

- **mvpa.misc.support.indentDoc(v)**
  Given a value returns a string where each line is indented

  Needed for a cleaner __repr__ output v - arbitrary

- **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
mvpa.misc.support.isSorted(items)
    Check if listed items are in sorted order.

    Parameters
    items (iterable container) –

    Returns
    True if were sorted. Otherwise False + Warning

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

    Parameters
    force (bool) – if True, force it even if the file2 starts with /

mvpa.misc.support.transformWithBoxcar(data, startpoints, boxlength, offset=0,
                                       fx=<function mean at 0x2982c80>)
    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).

    Parameters

    • 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 begin-
      ning of the boxcar window.

    Return type
    array (len(startpoints) x data.shape[1:])

mvpa.misc.support.version_to_tuple(v)
    Convert literal string into a tuple, if possible of ints

    Tuple of integers constructed by splitting at ‘.’ or interleaves of numerics and alpha numbers

mvpa.misc.support.xuniqueCombinations(L, n)
    Generator of unique combinations form a list L of objects in groups of size n.

    # XXX EO: I guess they are already sorted. # XXX EO: It seems to work well for n>20 :)

    Parameters

    • L (list) – list of unique ids
    • n (int) – grouping size

    Adopted from Li Daobing http://code.activestate.com/recipes/190465/ (MIT license, according to actives-
    tate.com’s policy)

16.9.24 misc.transformers

Module: misc.transformers

Inheritance diagram for mvpa.misc.transformers:
Simply functors that transform something.

**Classes**

**DistPValue**

```python
class mvpa.misc.transformers.DistPValue(sd=0, distribution='rdist', fpp=None, nbins=400, **kwargs)
```

Bases: `mvpa.misc.state.ClassWithCollections`

Converts values into p-values under vague and non-scientific assumptions

**Note:** Available state variables:

- `nulldist_number+`: Number of features within the estimated null-distribution
- `positives_recovered+`: Number of features considered to be positives and which were recovered

(States enabled by default are listed with +)

**See Also:**

Please refer to the documentation of the base class for more information:

*ClassWithCollections*

L2-Norm the values, convert them to p-values of a given distribution.

**Parameters**

- `sd (int)` – Samples dimension (if len(x.shape)>1) on which to operate
- `distribution (string)` – Which distribution to use. Known are: ‘rdist’ (later normal should be there as well)
- `fpp (float)` – At what p-value (both tails) if not None, to control for false positives. It would iteratively prune the tails (tentative real positives) until empirical p-value becomes less or equal to numerical.
- `nbins (int)` – Number of bins for the iterative pruning of positives
- `enable_states (None or list of basestring)` – Names of the state variables which should be enabled additionally to default ones
- `disable_states (None or list of basestring)` – Names of the state variables which should be disabled

**WARNING:** Highly experimental/slow/etc: no theoretical grounds have been presented in any paper, nor proven
OverAxis

class mvpa.misc.transformers.OverAxis(transformer, axis=None)

    Bases: object

    Helper to apply transformer over specific axis

    Initialize transformer wrapper with an axis.

    Parameters

    *transformer – A callable to be used

    *axis (None or int) – If None – apply transformer across all the data. If some int – over that axis

Functions

mvpa.misc.transformers.Absolute(x)

    Returns the elementwise absolute of any argument.

    Parameters

    x (scalar | sequence) –

mvpa.misc.transformers.FirstAxisMean(x)

    Mean computed along the first axis.

mvpa.misc.transformers.FirstAxisSumNotZero(x)

    Sum computed over first axis of whether the values are not equal to zero.

mvpa.misc.transformers.GrandMean(x)

    Just what the name suggests.

mvpa.misc.transformers.Identity(x)

    Return whatever it was called with.

mvpa.misc.transformers.L1Normed(x, norm=1.0, reverse=False)

    Norm the values so that L_1 norm (sum|xl) becomes norm

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.

mvpa.misc.transformers.OneMinus(x)

    Returns elementwise ‘1 - x’ of any argument.

mvpa.misc.transformers.RankOrder(x, reverse=False)

    Rank-order by value. Highest gets 0

mvpa.misc.transformers.ReverseRankOrder(x)

    Convenience functor

mvpa.misc.transformers.SecondAxisMaxOfAbs(x)

    Max of absolute values along the 2nd axis

mvpa.misc.transformers.SecondAxisMean(x)

    Mean across 2nd axis

Use cases:

    *to combine multiple sensitivities to get sense about mean sensitivity across splits

mvpa.misc.transformers.SecondAxisSumOfAbs(x)

    Sum of absolute values along the 2nd axis

16.9. Miscellaneous 319
Use cases:

• to combine multiple sensitivities to get sense about what features are most influential

16.9.25 misc.vproperty

Module: misc.vproperty

Inheritance diagram for mvpa.misc.vproperty:

C++-like virtual properties

VProperty

class mvpa.misc.vproperty.VProperty (fget=None, fset=None, fdel=None, doc='')
Bases: object

Provides “virtual” property: uses derived class’s method

16.9.26 atlases.base

Module: atlases.base

Inheritance diagram for mvpa.atlases.base:

Base classes for Anatomy atlases support

TODOs:

• major optimization. Now code is sloppy and slow – plenty of checks etc
Module Organization

mvpa.atlases.base module contains support for various atlases

  group Base
  BaseAtlas XMLBasedAtlas Label Level LabelsLevel
  group Talairach
  PyMVPAAtlas LabelsAtlas ReferencesAtlas
  group Exceptions
  XMLAtlasException

Classes

BaseAtlas

class mvpa.atlases.base.BaseAtlas
  Bases: object
  Base class for the atlases.
  Create an atlas object based on the... XXX

Label

class mvpa.atlases.base.Label (text, abbr=None, coord=(None, None, None), count=0, index=0)
  Bases: object
  Represents a label. Just to bring all relevant information together
  Parameters
    •text (basestring) – fullname of the label
    •abbr (basestring) – abbreviated name (optional)
    •coord (tuple of float) – coordinates (optional)
    •count (int) – count of those labels in the atlas (optional)

  abbr
    Returns abbreviated version if such is available
  coord
  count
  static generateFromXML (Elabel)
  index
  text

LabelsAtlas

class mvpa.atlases.base.LabelsAtlas (*args, **kwargs)
  Bases: mvpa.atlases.base.PyMVPAAtlas
  Atlas which provides labels for the given coordinate
  See Also:
  Please refer to the documentation of the base class for more information:
PyMVPAAtlas

Initialize instance of PyMVPAAtlas

Parameters

• `filename (string)` – Filename for the xml definition of the atlas

• `resolution (None or float)` – Some atlases link to multiple images at different resolutions. if None – best resolution is selected using 0th dimension resolution

• `image_file (None or str)` – If None, overrides filename for the used imagefile, so it could load a custom (re-registered) atlas maps

• `query_voxel (bool)` – By default [x,y,z] assumes coordinates in space, but if query_voxel is True, they are assumed to be voxel coordinates

• `coordT` – Optional transformation to apply first

• `levels (None or slice or list of int)` – What levels by default to operate on

`labelVoxel (c, levels=None)`

Return labels for the given voxel at specified levels specified by index

LabelsLevel

class mvpa.atlases.base.LabelsLevel (description, index=None, labels=[])  
Bases: mvpa.atlases.base.Level

Level of labels.

XXX extend

`find (target, unique=True)`

Return labels descr of which matches the string

Parameters

• `target (str or re_pattern_type)` – Substring in abbreviation to be searched for, or compiled regular expression to be searched or matched if anchored.

• `unique (bool)` – If True, raise exception if none or more than 1 was found. Return just a single item if found (not list).

static `generateFromXML (Elevel, levelIndex=[0])`  

index

labels

Level

class mvpa.atlases.base.Level (description)  
Bases: object

Represents a level. Just to bring all relevant information together

static `generateFromXML (Elevel, levelType=None)`  

Simple factory of levels

levelType
class PyMVPAAtlas(*args, **kwargs)
    Bases: mvpa.atlases.base.XMLBasedAtlas
    Base class for PyMVPA atlases, such as LabelsAtlas and ReferenceAtlas

See Also:
    Please refer to the documentation of the base class for more information:
XMLBasedAtlas

Initialize instance of PyMVPAAtlas

Parameters

• filename (string) – Filename for the xml definition of the atlas
• resolution (None or float) – Some atlases link to multiple images at different resolutions. if None – best resolution is selected using 0th dimension resolution
• image_file (None or str) – If None, overrides filename for the used imagefile, so it could load a custom (re-registered) atlas maps
• query_voxel (bool) – By default [x,y,z] assumes coordinates in space, but if query_voxel is True, they are assumed to be voxel coordinates
• coordT – Optional transformation to apply first
• levels (None or slice or list of int) – What levels by default to operate on

Nlevels

source = ‘PyMVPA’
space
spaceFlavor

ReferencesAtlas

class ReferencesAtlas(distance=0, *args, **kwargs)
    Bases: mvpa.atlases.base.PyMVPAAtlas
    Atlas which provides references to the other atlases.
    Example: the atlas which has references to the closest points (closest Gray, etc) in another atlas.

See Also:
    Please refer to the documentation of the base class for more information:
PyMVPAAtlas

Initialize ReferencesAtlas

Parameters

• filename (string) – Filename for the xml definition of the atlas
• resolution (None or float) – Some atlases link to multiple images at different resolutions. if None – best resolution is selected using 0th dimension resolution
• image_file (None or str) – If None, overrides filename for the used imagefile, so it could load a custom (re-registered) atlas maps

16.9. Miscellaneous
• **query_voxel** *(bool)* – By default \([x,y,z]\) assumes coordinates in space, but if query_voxel is True, they are assumed to be voxel coordinates
• **coordT** – Optional transformation to apply first
• **levels** *(None or slice or list of int)* – What levels by default to operate on

```
distance
```

```
labelVoxel(c, levels=None)
```

```
levelsListing()
```

```
setDistance(distance)
  Set desired maximal distance for the reference
```

```
setReferenceLevel(level)
  Set the level which will be queried
```

ReferencesLevel

```
class mvpa.atlases.base.ReferencesLevel(description, indexes=[])
    Bases: mvpa.atlases.base.Level
    Level which carries reference points
    static generateFromXML(Elevel)
    indexes
```

XMLAtlasException

```
class mvpa.atlases.base.XMLAtlasException(msg='')
    Bases: exceptions.Exception
    Exception to be thrown if smth goes wrong dealing with XML based atlas
```

XMLBasedAtlas

```
class mvpa.atlases.base.XMLBasedAtlas(filename=None, resolution=None, image_file=None, query_voxel=False, coordT=None, levels=None)
    Bases: mvpa.atlases.base.BaseAtlas
    Parameters
```

• **filename** *(string)* – Filename for the xml definition of the atlas
• **resolution** *(None or float)* – Some atlases link to multiple images at different resolutions. if None – best resolution is selected using 0th dimension resolution
• **image_file** *(None or str)* – If None, overrides filename for the used imagefile, so it could load a custom (re-registered) atlas maps
• **query_voxel** *(bool)* – By default \([x,y,z]\) assumes coordinates in space, but if query_voxel is True, they are assumed to be voxel coordinates
• **coordT** – Optional transformation to apply first
• **levels** *(None or slice or list of int)* – What levels by default to operate on

coordT

extent
**labelPoint** *(coord, levels=None)*  
Return labels for the given spatial point at specified levels  
Function takes care about first transforming the point into the voxel space  
Parameters  

- **coord** *(tuple)* – Coordinates of the point (xyz)  
- **levels** *(None or list of int)* – At what levels to return the results

```python
levelsListing()
levels_dict
loadAtlas(filename)
origin
setCoordT(coordT)
  Set coordT transformation.
  spaceT needs to be adjusted since we glob those two transformations together

spaceT
version
voxdim
```

**Function**

```python
mvpa.atlases.base.checkRange(coord, range)
  Check if coordinates are within range (0,0,0) - (range) Return True on success
```

### 16.9.27 atlases.fsl

**Module:** `atlases.fsl`

Inheritance diagram for `mvpa.atlases.fsl:`
FSL atlases interfaces

**Classes**

**FSLAtlas**

class `mvpa.atlases.fsl.FSLAtlas`(*args, **kwargs)
    Bases: `mvpa.atlases.base.XMLBasedAtlas`

Base class for FSL atlases

See Also:

Please refer to the documentation of the base class for more information:

XMLBasedAtlas

Parameters

- `filename` *(string)* – Filename for the xml definition of the atlas
- `resolution` *(None or float)* – Some atlases link to multiple images at different resolutions. if None – best resolution is selected using 0th dimension resolution
- `image_file` *(None or str)* – If None, overrides filename for the used imagefile, so it could load a custom (re-registered) atlas maps
- `query_voxel` *(bool)* – By default [x,y,z] assumes coordinates in space, but if query_voxel is True, they are assumed to be voxel coordinates
- `coordT` – Optional transformation to apply first
- `levels` *(None or slice or list of int)* – What levels by default to operate on

source = ‘FSL’
FSLLabelsAtlas

class mvpa.atlases.fsl.FSLLabelsAtlas(*args, **kwargs)
    Bases: mvpa.atlases.base.XMLBasedAtlas
    Not sure what this one was for
    not implemented

FSLProbabilisticAtlas

class mvpa.atlases.fsl.FSLProbabilisticAtlas(thr=0.0, strategy='all', sort=True, *args, **kwargs)
    Bases: mvpa.atlases.fsl.FSLAtlas
    Probabilistic FSL atlases
    See Also:
    Please refer to the documentation of the base class for more information:

FSLAtlas
    Parameters

•thr (float) – Value to threshold at
•strategy (basestring) – Possible values all - all entries above thr max - entry with maximal value
•sort (bool) – Either to sort entries for ‘all’ strategy according to probability
•filename (string) – Filename for the xml definition of the atlas
•resolution (None or float) – Some atlases link to multiple images at different resolutions. if None – best resolution is selected using 0th dimension resolution
•image_file (None or str) – If None, overrides filename for the used imagefile, so it could load a custom (re-registered) atlas maps
•query_voxel (bool) – By default [x,y,z] assumes coordinates in space, but if query_voxel is True, they are assumed to be voxel coordinates
•coordT – Optional transformation to apply first
•levels (None or slice or list of int) – What levels by default to operate on

find(*args, **kwargs)
    Just a shortcut to the only level.
    See find for more info

getMap(target, strategy='unique')
    Return a probability map
    Parameters

•target (int or str or re._pattern_type) – If int, map for given index is returned. Otherwise, .find is called with unique=True to find matching area
•strategy (str in (‘unique’, ‘max’)) – If ‘unique’, then if multiple areas match, exception would be raised. In case of ‘max’, each voxel would get maximal value of probabilities from all matching areas

getMaps(target)
    Return a list of probability maps for the target
Parameters

**target** *(str or re._pattern_type)* — find is called with a target and unique=False to find all matches

**labelVoxel** *(c, levels=None)*

Return labels for the voxel

Parameters

• *c (-)*
• *levels (-)*

### 16.9.28 atlases.transformation

**Module:** `atlases.transformation`

Inheritance diagram for `mvpa.atlases.transformation`:

Coordinate transformations

#### Classes

**Linear**

**Linear** *(transf=\[
[ 1., 0., 0., 0.],
[ 0., 1., 0., 0.],
[ 0., 0., 1., 0.],
[ 0., 0., 0., 1.]
\]), **kwargs)*

Bases: `mvpa.atlases.transformation.TransformationBase`

Simple linear transformation defined by a matrix

Linear`.apply`(coord)

**MNI2Tal_MatthewBrett**

**class** `mvpa.atlases.transformation.MNI2Tal_MatthewBrett` *(\*args, \**kwargs)*

Bases: `mvpa.atlases.transformation.TransformationBase`

Transformation to bring MNI coordinates into MNI space

Apparently it is due to Matthew Brett http://imaging.mrc-cbu.cam.ac.uk/imaging/MniTalairach

apply`(coord)`
SpaceTransformation

class mvpa.atlases.transformation.SpaceTransformation (voxelSize=None, origin=None, toRealSpace=True, *args, **kwargs)

Bases: mvpa.atlases.transformation.TransformationBase

To perform transformation from Voxel into Real Space. Simple one – would subtract the origin and multiply by voxelSize. if toRealSpace is True then on call/getitem converts to RealSpace

toRealSpace (coord)
toVoxelSpace (coord)

TransformationBase

class mvpa.atlases.transformation.TransformationBase (previous=None)

Basic class to describe a transformation. Pretty much an interface

apply (coord)

TypeProxy

class mvpa.atlases.transformation.TypeProxy (value, toType=<built-in function array>)

Simple class to convert from and then back to original type working with list, tuple, ndarray and having

XXX Obsolete functionality ??

Functions

mvpa.atlases.transformation.MNI2Tal_Lancaster07FSL (*args, **kwargs)
mvpa.atlases.transformation.MNI2Tal_Lancaster07pooled (*args, **kwargs)
mvpa.atlases.transformation.MNI2Tal_MeyerLindenberg98 (*args, **kwargs)

Due to Andreas Meyer-Lindenberg Taken from http://imaging.mrc-cbu.cam.ac.uk/imaging/MniTalairach

mvpa.atlases.transformation.MNI2Tal_YOHflirt (*args, **kwargs)

Transformations obtained using flirt from Talairach to Standard

Transformations were obtained by registration of grey/white matter image from talairach atlas to FSL’s standard volume. Following sequence of commands was used:

fslroi /usr/share/rumba/atlases/data/talairach_atlas.nii.gz talairach_graywhite.nii.gz 3 1 flirt -in talairach_graywhite.nii.gz -ref /usr/apps/fsl.4.1/data/standard/MNI152_T1_1mm_brain.nii.gz -out talairach2mni.nii.gz -omat talairach2mni.mat -searchrx -20 20 -searchry -20 20 -searchrz -20 20 -coarsesearch 10 -finsearch 6 -v flirt -datatype float -in talairach_graywhite.nii.gz -init talairach2mni.mat -ref /usr/apps/fsl.4.1/data/standard/MNI152_T1_1mm_brain.nii.gz -out talairach2mni_fine1.nii.gz -omat talairach2mni_fine1.mat -searchrx -10 10 -searchry -10 10 -searchrz -10 10 -coarsesearch 5 -finsearch 1 -v convert_xfm -inverse -omat mni2talairach.mat talairach2mni_fine1.mat

mvpa.atlases.transformation.Tal2MNI_Lancaster07FSL (*args, **kwargs)
mvpa.atlases.transformation.Tal2MNI_Lancaster07pooled (*args, **kwargs)
mvpa.atlases.transformation.Tal2MNI_YOHflirt (*args, **kwargs)

See MNI2Tal_YOHflirt doc
16.9.29 `atlases.warehouse`

**Module: `atlases.warehouse`**

Collection of the known atlases

```python
mvpa.atlases.warehouse.Atlas(filename=None, name=None, *args, **kwargs)
```

A convinience factory for the atlases
m
mvpa, 109
mvpa.algorithms.cvtranserror, 271
mvpa.algorithms.hyperalignment, 273
mvpa.atlases.base, 320
mvpa.atlases.fsl, 326
mvpa.atlases.transformation, 328
mvpa.atlases.warehouse, 330
mvpa.base, 274
mvpa.base.config, 275
mvpa.base.docheplers, 276
mvpa.base.externals, 277
mvpa.base.info, 278
mvpa.base.report, 278
mvpa.base.report_dummy, 280
mvpa.base.verbosity, 281
mvpa.clfs.base, 161
mvpa.clfs.blr, 163
mvpa.clfs.distance, 165
mvpa.clfs.enet, 167
mvpa.clfs.glmnet, 170
mvpa.clfs.gnb, 173
mvpa.clfs.gpr, 176
mvpa.clfs.kernel, 179
mvpa.clfs.knn, 183
mvpa.clfs.lars, 185
mvpa.clfs.libsmrlc, 187
mvpa.clfs.libsmrlc.ctypes_helper, 187
mvpa.clfs.libsvmc.sens, 188
mvpa.clfs.libsvmc.svm, 190
mvpa.clfs.libsvmc.svmc, 192
mvpa.clfs.meta, 194
mvpa.clfs.model_selector, 207
mvpa.clfs.plr, 208
mvpa.clfs.ridge, 209
mvpa.clfs.sg.sens, 211
mvpa.clfs.sg.svm, 212
mvpa.clfs.smlr, 215
mvpa.clfs.stats, 218
mvpa.clfs.transfaror, 224
mvpa.clfs.warehouse, 230
mvpa.datasets.base, 110
mvpa.datasets.channel, 117
mvpa.datasets.eep, 119
mvpa.datasets.event, 120
mvpa.datasets.mapped, 122
mvpa.datasets.masked, 123
mvpa.datasets.meta, 124
mvpa.datasets.miscfx, 125
mvpa.datasets.miscfx_sp, 127
mvpa.datasets.nifti, 128
mvpa.datasets.splitter, 131
mvpa.feature.base, 259
mvpa.featurehelpers, 262
mvpa.featureifs, 267
mvpa.feature.rf, 269
mvpa.mappers.array, 139
mvpa.mappers.base, 140
mvpa.mappers.boxcar, 145
mvpa.mappers.ica, 147
mvpa.mappers.lle, 148
mvpa.mappers.mask, 149
mvpa.mappers.metric, 151
mvpa.mappers.pca, 153
mvpa.mappers.procrustean, 154
mvpa.mappers.samplegroup, 155
mvpa.mappers.som, 156
mvpa.mappers.svd, 158
mvpa.mappers.wavelet, 159
mvpa.mappers.zscore, 160
mvpa.measures.anova, 231
mvpa.measures.base, 233
mvpa.measures.corrcoef, 241
mvpa.measures.coefficient, 243
mvpa.measures.ds, 244
mvpa.measures.glm, 245
mvpa.measures.irelief, 247
mvpa.measures.noiseperturbation, 251
mvpa.measures.pls, 253
mvpa.measures.searchlight, 255
mvpa.measures.splitmeasure, 257
mvpa.misc.args, 282
mvpa.misc.attributes, 283
mvpa.misc.bv.base, 285
mvpa.misc.cmdline, 285
mvpa.misc.data_generators, 286
mvpa.misc.errorfx, 288
mvpa.misc.exceptions, 289
mvpa.misc.fsl.base, 290
mvpa.misc.fsl.flobs, 292
mvpa.misc.fsl.melodic, 292
INDEX

A
abbr (mvpa.atlases.base.Label attribute), 321
absminDistance() (in module mvpa.clfs.distance), 165
Absolute() (in module mvpa.misc.transformers), 319
active (mvpa.base.verbosity.SetLogger attribute), 282
AdaptiveNormal (class in mvpa.clfs.stats), 218
AdaptiveNullDist (class in mvpa.clfs.stats), 219
AdaptiveRDist (class in mvpa.clfs.stats), 219
add() (mvpa.clfs.transerror.SummaryStatistics method), 228
add() (mvpa.misc.cmdline.OptionGroups method), 286
add() (mvpa.misc.state.Collection method), 310
AFNI, 8
aggregateFeatures() (in module mvpa.datasets.miscfx), 126
aggregateFeatures() (mvpa.datasets.base.Dataset method), 112
alternative build procedure, 13
analyzer (mvpa.measures.base.ProxyClassifierSensitivityAnalyzer attribute), 238
analyzers (mvpa.measures.base.CombinedFeaturewiseDatasetMeasure attribute), 235
anova, 35
API reference, 3
apply() (mvpa.atlases.transformation.Linear method), 328
apply() (mvpa.atlases.transformation.MNI2Tal_MatthewBrett method), 328
apply() (mvpa.atlases.transformation.TransformationBase method), 329
applyMapper() (mvpa.datasets.base.Dataset method), 112
applyMapper() (mvpa.datasets.meta.MetaDataset method), 124
asDiscreteTime() (mvpa.misc.support.Event method), 314
asdict() (mvpa.misc.io.hamster.Hamster method), 300
asstring() (mvpa.clfs.transerror.ConfusionMatrix method), 226
asstring() (mvpa.clfs.transerror.RegressionStatistics method), 227
asstring() (mvpa.clfs.transerror.SummaryStatistics method), 228
Atlas() (in module mvpa.atlases.warehouse), 330
AttributesCollector (class in mvpa.misc.state), 309
AttributeWithUnique (class in mvpa.misc.attributes), 283
AUCErrorFx (class in mvpa.misc.errorfx), 288
aucs (mvpa.clfs.transerror.ROCCurve attribute), 227
autoNullDist() (in module mvpa.clfs.stats), 223
B
backports, 9
BaseAtlas (class in mvpa.atlases.base), 321
BestDetector (class in mvpa.featsel.helpers), 262
bestindex (mvpa.featsel.helpers.BestDetector attribute), 263
bias (mvpa.measures.base.StaticDatasetMeasure attribute), 241
biases (mvpa.clfs.smlr.SMLR attribute), 216
binary packages, 9
BinaryClassifier (class in mvpa.clfs.meta), 195
Block-averaging, 93
Block averaging, 89
BLR (class in mvpa.clfs.blr), 164
BoostedClassifier (class in mvpa.clfs.meta), 196
BoostedClassifierSensitivityAnalyzer (class in mvpa.measures.base), 233
BoxcarMapper (class in mvpa.mappers.boxcar), 145
BrainVoyagerRTC (class in mvpa.misc.bv.base), 285
build instructions, 12
building from source, 12
building on Windows, 13
C
C (mvpa.clfs.libsvm.svmc.svm_parameter attribute), 193
C (mvpa.datasets.base.Dataset attribute), 112
cache_size (mvpa.clfs.libsvm.svmc.svm_parameter attribute), 193
call (mvpa.misc.support.HarvesterCall attribute), 315
cartesianDistance() (in module mvpa.clfs.distance), 165
cdf() (mvpa.clfs.stats.AdaptiveRDist method), 220
cdf() (mvpa.clfs.stats.FixedNullDist method), 220
cdf() (mvpa.clfs.stats.MCNullDist method), 221
cdf() (mvpa.clfs.stats.Nonparametric method), 222
cdf() (mvpa.clfs.stats.NullDist method), 223
cfg, 43
ChainMapper (class in mvpa.mappers.base), 140
changelog, 99
ChannelDataset (class in mvpa.datasets.channel), 117
configuration, 43
confusion (mvpa.clfs.transerror.ClassifierError attribute), 224
ConfusionBasedError (class in mvpa.clfs.transerror), 225
ConfusionMatrix (class in mvpa.clfs.transerror), 226
ConvergenceError (class in mvpa.misc.exceptions), 289
convertFeatureIds2FeatureMask() (mvpa.datasets.base.Dataset method), 112
class (mvpa.datasets.base.Dataset method), 113
class (mvpa.mappers.mask.MaskMapper method), 149
class (mvpa.mappers.mask.MaskMapper method), 150
cor (mvpa.datasets.base.Label attribute), 321
coordT (mvpa.atlases.base.XMLBasedAtlas attribute), 324
copy() (mvpa.datasets.base.Dataset method), 113
CorrCoef (class in mvpa.measures.corrcoef), 242
CorrStability (class in mvpa.measures.corrstability), 243
count (mvpa.atlases.base.Label attribute), 321
count (mvpa.datasets.splitters.Splitter attribute), 138
cross-validation, 28, 62, 64, 90
CrossValidatedTransferError (class in mvpa.algorithms.cvtranserror), 271
CustomSplitter (class in mvpa.datasets.splitters), 131

D
data (mvpa.misc.io.base.DataReader attribute), 296
data formats, 21
data splitting, 22
DataReader (class in mvpa.misc.io.base), 296
Dataset, 93
dataset, 18
Dataset (class in mvpa.datasets.base), 111
dataset attribute, 18
DatasetAttribute (class in mvpa.misc.attributes), 284
DatasetError (class in mvpa.misc.exceptions), 289
DatasetMeasure (class in mvpa.measures.base), 235
datasetmethod() (in module mvpa.datasets.base), 117
datasets (mvpa.datasets.meta.MetaDataset attribute), 125
Debian, 9
debug, 45, 47
Decoding, 93
default (mvpa.misc.param.Parameter attribute), 302
defineFeatureGroups() (mvpa.datasets.base.Dataset method), 113
DegenerateInputError (class in mvpa.clfs.base), 163
degree (mvpa.clfs.libsvmcm.svmc.svm_parameter attribute), 193
delete_double() (in module mvpa.clfs.libsvmcm.svmc), 193
delete_int() (in module mvpa.clfs.libsvmcm.svmc), 193
DenseArrayMapper (class in mvpa.mappers.array), 139
descr (mvpa.misc.state.ClassWithCollections attribute), 309
DescreteMetric (class in mvpa.mappers.metric), 151
design2labels() (in module mvpa.misc.io.base), 298
detrend() (in module mvpa.datasets.mcsfx_sp), 127
detrend() (mvpa.datasets.base.Dataset method), 113
detrending, 48
development, 87
development snapshot, 12
disable() (mvpa.misc.state.StateCollection method), 312
discardOut() (mvpa.mappers.mask.MaskMapper method), 150
dissimilarity matrix, 64
distance (mvpa.atlases.base.ReferencesAtlas attribute), 324
DistPValue (class in mvpa.misc.transformers), 318
doc() (mvpa.misc.param.Parameter method), 302
double_getitem() (in module mvpa.clfs.libsvmcm.svmc), 193
double_setitem() (in module mvpa.clfs.libsvmcm.svmc), 193
doubleGammaHRF() (in module mvpa.misc.fx), 293
doublepccarray2numpy_array() (in module mvpa.clfs.libsvmcm.svmc), 193
DSMatrix (class in mvpa.misc.stats), 313
DSMDatasetMeasure (class in mvpa.measures.ds), 244
dt (mvpa.datasets.channel.ChannelDataset attribute), 118
dt (mvpa.datasets.nifti.NiftiDataset attribute), 129
dt (mvpa.misc.io.eepbin.EEPBin attribute), 299
dumbFeatureBinaryDataset() (in module mvpa.misc.data_generators), 286
dumbFeatureDataset() (in module mvpa.misc.data_generators), 286
dump() (mvpa.misc.io.hamster.Hamster method), 300
durations (mvpa.misc.fsl.base.FslEV3 attribute), 290

E
EEPBin (class in mvpa.misc.io.eepbin), 299
EEPDataset (class in mvpa.datasets.eep), 119
ElementSelector (class in mvpa.featsel.helpers), 263
elementsizer (mvpa.mappers.metric.DescreteMetric attribute), 152
enable() (mvpa.misc.attributes.StateVariable method), 284
enable() (mvpa.misc.state.StateCollection method), 312
enabled (mvpa.misc.state.StateCollection attribute), 312
ENET (class in mvpa.clfs.enet), 167
ENETWeights (class in mvpa.clfs.enet), 168
eenhancedDocString() (in module mvpa.base.dochelpers), 276
environment variable
MVPA_DEBUG, 47
MVPA_DEBUG_METRICS, 48
MVPA_DEBUG_WTF, 48
MVPA_MATPLOTLIB_BACKEND, 104
MVPA_QUICKTEST, 105
MVPA_SEED, 48
MVPA_SVM_BACKEND, 105
MVPA_TESTS_LABILE, 48, 105
MVPA_TESTS_QUICK, 48, 105
MVPA_VERBOSE, 46
MVPA_VERBOSE_OUTPUT, 43
MVPA_WARNINGS_BT, 47
MVPA_WARNINGS_COUNT, 47
MVPA_WARNINGS_SUPPRESS, 47

Epoch, 93
eps (mvpa.clfs.libsvm.svmc.svm_parameter attribute), 193
equalDefault (mvpa.misc.param.Parameter attribute), 303
ERNiftiDataset (class in mvpa.datasets.nifti), 128
error, 27
error (mvpa.clfs.transerror.ConfusionMatrix attribute), 226
error (mvpa.clfs.transerror.RegressionStatistics attribute), 227
error (mvpa.clfs.transerror.SummaryStatistics attribute), 228
error() (in module mvpa.base), 274
errorfx (mvpa.clfs.transerror.TransferError attribute), 229
escapeXML() (in module mvpa.base.report), 280
Event (class in mvpa.misc.support), 314
EventDataset (class in mvpa.datasets.event), 121
Example, 93
example, 49
event-related fMRI, 72
EventDataset (class in mvpa.datasets.event), 121

F
F-score, 35
FailedToPredictError (class in mvpa.clfs.base), 163
FailedToTrainError (class in mvpa.clfs.base), 163
Feature, 93
feature, 19, 20
feature selection, 21, 35, 37, 89
feature_ids, 89
feature_ids (mvpa.measures.base.Sensitivity attribute), 239
feature_selection (mvpa.clfs.meta.FeatureSelectionClassifier attribute), 199
feature_selections (mvpa.featsel.base.CombinedFeatureSelection attribute), 260
feature_selections (mvpa.featsel.base.FeatureSelectionPipeline attribute), 261
FeatureAttribute (class in mvpa.misc.attributes), 284
FeatureSelection, 39
FeatureSelection (class in mvpa.featsel.base), 260
FeatureSelectionClassifier, 39
FeatureSelectionClassifier (class in mvpa.clfs.meta), 199
FeatureSelectionClassifierSensitivityAnalyzer (class in mvpa.measures.base), 236
FeatureSelectionPipeline (class in mvpa.featsel.base), 261
FeaturewisePipeline (class in mvpa.featsel.base), 237
Fedora, 15
felements (mvpa.featsel.helpers.FractionTailSelector attribute), 265
figure() (mvpa.base.report.Report method), 279
figures() (mvpa.base.report.Report method), 280
filter_coord (mvpa.mappers.metric.DiscreteMetric attribute), 327
find() (mvpa.atlases.base.LevelsLevel method), 322
find() (mvpa.atlases.fsl.FSLProbabilisticAtlas method), 327
FirstAxisMean() (in module mvpa.misc.transformers), 319
FirstAxisSumNotZero() (in module mvpa.misc.transformers), 319
FixedNullDist (class in mvpa.clfs.stats), 220
flowbreak() (mvpa.base.report.Report method), 280
fMRI, 93
forward mapping, 20
forward() (mvpa.mappers.base.ChainMapper method), 140
forward() (mvpa.mappers.base.CombinedMapper method), 144
forward() (mvpa.mappers.boxcar.BoxcarMapper method), 146
forward() (mvpa.mappers.lle.LLEMapper method), 148
forward() (mvpa.mappers.mask.MaskMapper method), 150
forward() (mvpa.mappers.samplegroup.SampleGroupMapper method), 155
forward() (mvpa.mappers.som.SimpleSOMMapper method), 157
FractionTailSelector (class in mvpa.featsel.helpers), 264
free software, 3
free_sv (mvpa.clfs.libsvm.svmc.svm_model attribute), 192
FSL, 8, 48
FSLAtlas (class in mvpa.atlases.fsl), 326
FslEV3 (class in mvpa.misc.fsl.base), 290
FslGLMDesign (class in mvpa.misc.fsl.base), 291
FSSLLabelsAtlas (class in mvpa.atlases.fsl), 327
FSLProbabilisticAtlas (class in mvpa.atlases.fsl), 327
funcdata (mvpa.misc.fsl.melodic.MelodicResults attribute), 293

G
gamma (mvpa.clfs.libsvm.svmc.svm_parameter attribute), 193
Gaussian process regression, 30
generateFromXML() (mvpa.atlases.base.Label static method), 321
generateFromXML() (mvpa.atlases.base.LabelsLevel static method), 322
generateFromXML() (mvpa.atlases.base.Level static method), 322
generateFromXML() (mvpa.atlases.base.ReferencesLevel static method), 324
get() (mvpa.base.config.ConfigManager method), 276
get() (mvpa.misc.state.Collection method), 310
get_argtypes() (in module mvpa.clfs.libsmlrc.ctypes_helper), 188
getAsDType() (mvpa.base.config.ConfigManager method), 276
getboolean() (mvpa.base.config.ConfigManager method), 276
getBreakPoints() (in module mvpa.misc.support), 316
getData() (mvpa.misc.io.base.DataReader method), 296
getDi() (mvpa.datasets.nifti.NiftiDataset method), 130
getEV() (mvpa.misc.fsl.base.FslEV3 method), 290
getFullMatrix() (mvpa.misc.stats.DSMatrix method), 313
getInId() (mvpa.mappers.base.Mapper method), 143
getInId() (mvpa.mappers.mask.MaskMapper method), 150
getInId() (mvpa.mappers.som.SimpleSOMMapper method), 157
getInIds() (mvpa.mappers.mask.MaskMapper method), 150
getInSize() (mvpa.mappers.samplegroup.SampleGroupMapper method), 155
getInSize() (mvpa.mappers.som.SimpleSOMMapper method), 157
getLabels_map() (mvpa.clfs.transerror.ConfusionMatrix method), 226
getLabelsMap() (mvpa.datasets.base.Datset method), 113
getMajorityVote() (mvpa.clfs.knn.kNN method), 184
getMap() (mvpa.atlases.fsl.FSLProbabilisticAtlas method), 327
getMaps() (mvpa.atlases.fsl.FSLProbabilisticAtlas method), 327
getMask() (mvpa.mappers.mask.MaskMapper method), 150
getMetric() (mvpa.mappers.base.Mapper method), 143
getMetric() (mvpa.misc.stats.DSMatrix method), 313
getMVPattern() (in module mvpa.misc.data_generators), 286
getNColumns() (mvpa.misc.io.base.ColumnData method), 295
getNeighbor() (mvpa.mappers.base.ChainMapper method), 140
getNeighbor() (mvpa.mappers.base.CombinedMapper method), 142
getNeighbor() (mvpa.mappers.base.Mapper method), 143
getNeighbor() (mvpa.mappers.metric.Metric method), 152
getNeighbor() (mvpa.mappers.metric.Descrmetric method), 152
getNeighbor() (mvpa.mappers.metric.DiscreteMetric method), 152
getNEVs() (mvpa.misc.fsl.base.FslEV3 method), 291
getNFeatures() (mvpa.datasets.base.Dataset method), 113
getNFeatures() (mvpa.datasets.meta.MetaDataset method), 125
getNifitiData() (in module mvpa.datasets.nifti), 130
getNifitiFromAnySource() (in module mvpa.datasets.nifti), 130
getNRows() (mvpa.misc.io.base.ColumnData method), 295
getNSamples() (mvpa.datasets.base.Datset method), 114
getNSamples() (mvpa.datasets.meta.MetaDataset method), 125
getNSamples() (mvpa.misc.io.base.SampleAttributes method), 297
getOutId() (mvpa.mappers.mask.MaskMapper method), 150
getOutSize() (mvpa.mappers.mask.MaskMapper method), 150

Index 337
Level (class in mvpa.atlases.base), 322
level (mvpa.base.verbosity.LevelLogger attribute), 281
levels_dict (mvpa.atlases.base.XMLBasedAtlas attribute), 325
levelsListing() (mvpa.atlases.base.ReferencesAtlas method), 324
levelsListing() (mvpa.atlases.base.XMLBasedAtlas method), 325
levelType (mvpa.atlases.base.Level attribute), 322
lfprev (mvpa.base.verbosity.Logger attribute), 281
LIBSVM, 8, 12
license, 3
linear1d_gaussian_noise() (in module mvpa.misc.data_generators), 286
linear_awgn() (in module mvpa.misc.data_generators), 286
LinearSVMWeights (class in mvpa.clfs.libsvmc.sens), 188
LinearSVMWeights (class in mvpa.clfs.sg.sens), 211
listing (mvpa.misc.state.Collection attribute), 310
listing() (mvpa.clfs.warehouse.Warehouse method), 230
LLEMapper (class in mvpa.mappers.lle), 148
loadAtlas() (mvpa.atlases.base.XMLBasedAtlas method), 325
locations() (mvpa.misc.io.base.SensorLocations method), 297
Logger (class in mvpa.base.verbosity), 281
logistic regression, 30
MacOS X, 10, 12, 15
mahalanobisDistance() (in module mvpa.clfs.distance), 165
makeFlobs() (in module mvpa.misc.fsl.flobs), 292
manhattenDistance() (in module mvpa.clfs.distance), 165
map2Nifti() (mvpa.datasets.nifti.ERNiftiDataset method), 129
map2Nifti() (mvpa.datasets.nifti.NiftiDataset method), 130
mapForward() (mvpa.datasets.mapped.MappedDataset method), 122
MapOverlap (class in mvpa.misc.support), 315
MappedClassifier, 67
MappedClassifier (class in mvpa.clfs.meta), 200
MappedClassifierSensitivityAnalyzer (class in mvpa.measures.base), 237
MappedDataset (class in mvpa.datasets.mapped), 122
mapper, 20, 67, 80
Mapper (class in mvpa.mappers.base), 142
map (mvpa.clfs.meta.MappedClassifier attribute), 200
map (mvpa.datasets.mapped.MappedDataset attribute), 123
mapReverse() (mvpa.datasets.meta.MetaDataset method), 125
mapSelReverse() (mvpa.datasets.mapped.MappedDataset method), 123
mask (mvpa.mappers.mask.MaskMapper attribute), 150
maskclf (mvpa.clfs.meta.FeatureSelectionClassifier attribute), 199
MaskedDataset, 20
MaskedDataset (class in mvpa.datasets.masked), 123
MaskMapper (class in mvpa.mappers.mask), 149
Matlab, 84
maptoolib, 8
matrices (mvpa.clfs.transerror.ConfusionMatrix attribute), 226
matrix (mvpa.clfs.transerror.ConfusionMatrix attribute), 226
max_log_marginal_likelihood() (mvpa.clfs.model_selector.ModelSelector method), 207
maxcount (mvpa.base.WarningLog attribute), 274
MaximalVote (class in mvpa.clfs.meta), 201
McFlirtParams (class in mvpa.misc.fsl.fslbase), 291
MCNullDist (class in mvpa.clfs.stats), 221
MeanMismatchErrorFx (class in mvpa.misc.errorfx), 288
meanPowerFx() (in module mvpa.misc.errorfx), 289
MeanPrediction (class in mvpa.clfs.stats), 201
means (mvpa.clfs.gnb.GNB attribute), 175
measure, 33, 35–37
MelodicResults (class in mvpa.misc.fsl.melodic), 293
meta measures, 37
MetaDataset (class in mvpa.datasets.meta), 124
Metric (class in mvpa.mappers.metric), 152
metric (mvpa.mappers.base.Mapper attribute), 143
misc, 41
MN2Tal_Lancaster07FSL() (in module mvpa.atlases.transformation), 329
MN2Tal_Lancaster07pooled() (in module mvpa.atlases.transformation), 329
MN2Tal_MatthewBrett (class in mvpa.atlases.transformation), 328
MN2Tal_MeyerLindenberg98() (in module mvpa.atlases.transformation), 329
MN2Tal_YOHflirt() (in module mvpa.atlases.transformation), 329
mode (mvpa.featsel.helpers.ElementSelector attribute), 263
model (mvpa.clfs.libsvmc.svm.SVM attribute), 192
ModelSelector (class in mvpa.clfs.model_selector), 207
modular architecture, 18
monte-carlo, 69
motion correction, 48
MulticlassClassifier (class in mvpa.clfs.meta), 201
multipleChunks() (in module mvpa.misc.data_generators), 286
MultiStopCrit (class in mvpa.featsel.helpers), 265
MVPA, 3, 94
mvpa (module), 109
MVPA toolbox for Matlab, 3, 85
mvpa.algorithms.cvtranserror (module), 271
mvpa.algorithms.hyperalignment (module), 273
mvpa.atlases.base (module), 320
mvpa.atlases.fsl (module), 326
mvpa.atlases.transformation (module), 328
mvpa.atlases.warehouse (module), 330
mvpa.base (module), 274
mvpa.base.config (module), 275
mvpa.base.docheaders (module), 276
mvpa.base.externals (module), 277
mvpa.base.info (module), 278
mvpa.base.report (module), 278
mvpa.base.report_dummy (module), 280
mvpa.base.verbosity (module), 281
mvpa.clfs.base (module), 161
mvpa.clfs.blr (module), 163
mvpa.clfs.distance (module), 165
mvpa.clfs.enet (module), 167
mvpa.clfs.glmmnet (module), 170
mvpa.clfs.gnb (module), 173
mvpa.clfs.gpr (module), 176
mvpa.clfs.kernel (module), 179
mvpa.clfs.knn (module), 183
mvpa.clfs.lars (module), 185
mvpa.clfs.libsmmr (module), 187
mvpa.clfs.libsmrctypes_helper (module), 187
mvpa.clfs.libsvmc.sens (module), 188
mvpa.clfs.libsvmc.svm (module), 190
mvpa.clfs.libsvmc.svnc (module), 192
mvpa.clfs.meta (module), 194
mvpa.clfs.model_selector (module), 207
mvpa.clfs.plr (module), 208
mvpa.clfs.ridge (module), 209
mvpa.clfs.sg.sens (module), 211
mvpa.clfs.sg.svm (module), 212
mvpa.clfs.smlr (module), 215
mvpa.clfs.stats (module), 218
mvpa.clfs.transerror (module), 224
mvpa.clfs.warehouse (module), 230
mvpa.datasets.base (module), 110
mvpa.datasets.channel (module), 117
mvpa.datasets.eep (module), 119
mvpa.datasets.event (module), 120
mvpa.datasets.mapped (module), 122
mvpa.datasets.masked (module), 123
mvpa.datasets.meta (module), 124
mvpa.datasets.miscfx (module), 125
mvpa.datasets.miscfx_sp (module), 127
mvpa.datasets.nifti (module), 128
mvpa.datasets.splitizers (module), 131
mvpa.featsel.base (module), 259
mvpa.featsel.helpers (module), 262
mvpa.featsel.lfs (module), 267
mvpa.featsel.rfe (module), 269
mvpa.mappers.array (module), 139
mvpa.mappers.base (module), 140
mvpa.mappers.boxcar (module), 145
mvpa.mappers.ica (module), 147
mvpa.mappers.lle (module), 148
mvpa.mappers.mask (module), 149
mvpa.mappers.metric (module), 151
mvpa.mappers.pca (module), 153
mvpa.mappers.procrustean (module), 154
mvpa.mappers.samplegroup (module), 155
mvpa.mappers.som (module), 156
mvpa.mappers.svd (module), 158
mvpa.mappers.wavelet (module), 159
mvpa.mappers.zscore (module), 160
mvpa.measures.anova (module), 231
mvpa.measures.base (module), 233
mvpa.measures.cor (module), 241
mvpa.measures.correlation (module), 243
mvpa.measures.discriminant (module), 247
mvpa.measures.feature (module), 247
mvpa.measures.noise (module), 251
mvpa.measures.pnl (module), 253
mvpa.measures.spatial (module), 255
mvpa.measures.split (module), 257
mvpa.misc.args (module), 282
mvpa.misc.attributes (module), 283
mvpa.misc.bv.base (module), 285
mvpa.misc.cmdline (module), 285
mvpa.misc.data (module), 286
mvpa.misc.errorfx (module), 288
mvpa.misc.exceptions (module), 289
mvpa.misc.fsl_base (module), 290
mvpa.misc.fsl.flobs (module), 292
mvpa.misc.fsl.melodic (module), 292
mvpa.misc.fx (module), 293
mvpa.misc.io (module), 295
mvpa.misc.io.eepbin (module), 299
mvpa.misc.io.hamster (module), 300
mvpa.misc.io.sig (module), 301
mvpa.misc.param (module), 302
mvpa.misc.plot.base (module), 303
mvpa.misc.plot.erp (module), 305
mvpa.misc.plot.mri (module), 307
mvpa.misc.plot.topo (module), 307
mvpa.misc.state (module), 309
mvpa.misc.stats (module), 313
mvpa.misc.support (module), 313
mvpa.misc.transformers (module), 318
mvpa.misc.vproperty (module), 320
MVPA_DEBUG, 47
MVPA_DEBUG_METRICS, 48
MVPA_DEBUG_WTF, 48
MVPA_MATPLOTLIB_BACKEND, 104
MVPA_DEBUGTEST, 105
MVPA_SEED, 48
MVPA_SVM_BACKEND, 105
MVPA_TESTS_LABILE, 48, 105
MVPA_TESTS_QUICK, 48, 105
MVPA_VERBOSE, 46
MVPA VERBOSE OUTPUT, 43
MVPA WARNINGS BT, 47
MVPA WARNINGS COUNT, 47
MVPA WARNINGS SUPPRESS, 47

N
name (mvpa.misc.attributes.CollectableAttribute attribute), 284
name (mvpa.misc.state.Collection attribute), 310
names (mvpa.misc.state.Collection attribute), 310
nanmean() (in module mvpa.clfs.stats), 223
NBackHistoryStopCrit (class in mvpa.featsel.helpers), 265
nchannels (mvpa.misc.io.eepbin.EEPBin attribute), 299
columns (mvpa.misc.io.base.ColumnData attribute), 296
nelements (mvpa.featsel.helpers.FixedNElementTailSelector attribute), 264
nevs (mvpa.misc.fsl.base.FslEV3 attribute), 291
new_double() (in module mvpa.clfs.libsvmc.svmc), 194
new_int() (in module mvpa.clfs.libsvmc.svmc), 194
nfeatures (mvpa.datasets.base.Dataset attribute), 115
nfeatures (mvpa.datasets.meta.MetaDataset attribute), 125
nfeatures (mvpa.mappers.base.Mapper attribute), 143
NFoldSplitter (class in mvpa.datasets.splitters), 133
NGroupSplitter (class in mvpa.datasets.splitters), 134
nic (mvpa.misc.fsl.melodic.MelodicResults attribute), 293
NIFTI, 4, 63
NiftiDataset (class in mvpa.datasets.nifti), 129
niftihdr (mvpa.datasets.nifti.ERNiftiDataset attribute), 129
niftihdr (mvpa.datasets.nifti.NiftiDataset attribute), 130
Nlevels (mvpa.atlases.base.PyMVPAAtlas attribute), 323
node (mvpa.mappers.lle.LLEMapper attribute), 149
noise perturbation, 36
NoisePerturbationSensitivity (class in mvpa.measures.noiseperturbation), 252
noisy_2d_fx() (in module mvpa.misc.data_generators), 287
NoneSplitter (class in mvpa.datasets.splitters), 135
Nonparametric (class in mvpa.clfs.stats), 222
normalFeatureDataset() (in module mvpa.clfs.libsvmc.svmc), 287
normalFeatureDataset__() (in module mvpa.clfs.libsvmc.svmc), 287
nr_class (mvpa.clfs.libsvmc.svmc.svm_model attribute), 192
nr_weight (mvpa.clfs.libsvmc.svmc.svm_model attribute), 193
nrows (mvpa.misc.io.base.ColumnData attribute), 296
nsamples (mvpa.misc.io.base.Dataset attribute), 115
nsamples (mvpa.datasets.meta.MetaDataset attribute), 125
nsamples (mvpa.misc.io.base.SampleAttributes attribute), 297
nsamples (mvpa.misc.io.eepbin.EEPBin attribute), 299
NStepsStopCrit (class in mvpa.featsel.helpers), 265
nSV (mvpa.clfs.libsvmc.svmc.svm_model attribute), 192
timepoints (mvpa.misc.io.eepbin.EEPBin attribute), 299
nu (mvpa.clfs.libsvmc.svmc.svm_parameter attribute), 193
null_dist (mvpa.clfs.transerror.TransferError attribute), 229
null_dist (mvpa.measures.base.DatasetMeasure attribute), 236
null_prob (mvpa.measures.base.DatasetMeasure attribute), 236
NullDist (class in mvpa.clfs.stats), 222
NumPy, 8

O
O (mvpa.datasets.mapped.MappedDataset attribute), 122
OddEvenSplitter (class in mvpa.datasets.splitters), 136
OnceLogger (class in mvpa.base.verbosity), 281
OneMinus() (in module mvpa.misc.transforms), 319
oneMinusCorrelation() (in module mvpa.clfs.distance), 165
OneWayAnova (class in mvpa.measures.anova), 232
onsets (mvpa.misc.fsl.base.FslEV3 attribute), 291
OpenSUSE, 11, 14
optimization, 87
OptionGroups (class in mvpa.misc.cmdline), 286
Options (class in mvpa.misc.cmdline), 286
origids (mvpa.datasets.base.Dataset attribute), 115
origin (mvpa.atlases.base.XMLBasedAtlas attribute), 325
OverAxis (class in mvpa.misc.transforms), 319
owner (mvpa.misc.state.Collection attribute), 310

P
p (mvpa.clfs.libsvmc.svmc.svm_parameter attribute), 193
pt() (mvpa.clfs.stats.NullDist method), 223
param (mvpa.clfs.libsvmc.svmc.svm_model attribute), 192
Parameter (class in mvpa.misc.param), 302
ParameterCollection (class in mvpa.misc.state), 311
parse() (mvpa.misc.support.SmartVersion method), 316
path (mvpa.misc.fsl.melodic.MelodicResults attribute), 293
PCAMapper (class in mvpa.mappers.pca), 153
penalized logistic regression, 30
percentCorrect (mvpa.clfs.transerror.ConfusionMatrix attribute), 226
permutation, 69

Index
permuteLabels() (mvpa.datasets.base.Dataset method), 115
permuteLabels() (mvpa.datasets.meta.MetaDataset method), 125
plot() (mvpa.clfs.transerror.ConfusionMatrix method), 226
plot() (mvpa.clfs.transerror.RegressionStatistics method), 227
plot() (mvpa.clfs.transerror.ROCCurve method), 227
plot() (mvpa.datasets.miscfx.SequenceStats method), 126
plot() (mvpa.misc.fsl.base.FslGLMDesign method), 291
plot() (mvpa.misc.fsl.base.McFlirtParams method), 291
plotBars() (in module mvpa.misc.plot.base), 303
plotDatasetChunks() (in module mvpa.misc.plot.base), 303
plotERP() (in module mvpa.misc.plot.erp), 305
plotERPs() (in module mvpa.misc.plot.erp), 306
plotErrLine() (in module mvpa.misc.plot.base), 304
plotFeatureHist() (in module mvpa.misc.plot.base), 304
plotHeadOutline() (in module mvpa.misc.plot.topo), 308
plotHeadTopography() (in module mvpa.misc.plot.topo), 308
plotMRI() (in module mvpa.misc.plot.mri), 307
plotSamplesDistance() (in module mvpa.misc.plot.base), 304
plotting example, 76
PLR (class in mvpa.clfs.plr), 208
PLS (class in mvpa.measures.pls), 253
pnorm_w_python() (in module mvpa.clfs.distance), 165
predict() (mvpa.clfs.base.Classifier method), 162
PredictionsCombiner (class in mvpa.clfs.meta), 203
print_registered() (mvpa.base.verbosity.SetLogger method), 282
printsetid (mvpa.base.verbosity.SetLogger attribute), 282
priors (mvpa.clfs.gnb.GNB attribute), 175
probA (mvpa.clfs.libsvm.svmc.svm_model attribute), 193
probability (mvpa.clfs.libsvm.svmc.svm_model attribute), 193
probB (mvpa.clfs.libsvm.svmc.svm_model attribute), 193
process_args() (in module mvpa.clfs.libsmlrc.ctypes_helper), 188
Processing object, 94
processing object, 23, 35
ProcrusteanMapper (class in mvpa.mappers.procrustean), 154
progress tracking, 45
proj (mvpa.mappers.base.ProjectionMapper attribute), 145
ProjectionMapper (class in mvpa.mappers.base), 144
props (mvpa.misc.io.base.DataReader attribute), 296
ProxyClassifier (class in mvpa.clfs.meta), 203
ProxyClassifierSensitivityAnalyzer (class in mvpa.measures.base), 238
pureMultivariateSignal() (in module mvpa.misc.data_generators), 287
PyMatlab, 4
PyMVPA poster, 4
PyMVPAAtlas (class in mvpa.atlases.base), 323
PyNIfTI, 8
R
R, 8
random number generation, 48
RandomElementSelector (class in mvpa.featsel.helpers), 266
RankOrder() (in module mvpa.misc.transformers), 319
read_fsl_design() (in module mvpa.misc.fsl.base), 292
rebuildSamples() (mvpa.datasets.meta.MetaDataset method), 125
recommended software, 8
recon (mvpa.mappers.base.ProjectionMapper attribute), 145
recursive feature selection, 39
redirecting output, 46
references, 94
ReferencesAtlas (class in mvpa.atlases.base), 323
ReferencesLevel (class in mvpa.atlases.base), 324
register() (mvpa.base.verbosity.SetLogger method), 282
registered (mvpa.base.verbosity.SetLogger attribute), 282
registered (mvpa.misc.io.hamster.Hamster attribute), 300
RegressionStatistics (class in mvpa.clfs.transerror), 227
RelativeRMSErrorFx (class in mvpa.clfs.errorfx), 288
releases, 12
reload() (mvpa.base.config.ConfigManager method), 276
relvar_per_ic (mvpa.misc.fsl.melodic.MelodicResults attribute), 293
remove() (mvpa.misc.state.Collection method), 310
removeInvariantFeatures() (in module mvpa.datasets.miscfx), 126
removeInvariantFeatures() (mvpa.datasets.base.Dataset method), 115
Report (class in mvpa.base.report), 279
Report (class in mvpa.base.report_dummy), 280
repredict() (mvpa.clfs.base.Classifier method), 162
reloaded software, 8
resample() (mvpa.datasets.channel.ChannelDataset method), 118
reset() (mvpa.clfs.kernel.Kernel method), 179
reset() (mvpa.clfs.kernel.KernelLinear method), 181
reset() (mvpa.clfs.kernel.KernelSquaredExponential method), 183
reset() (mvpa.clfs.transerror.SummaryStatistics method), 228
reset() (mvpa.misc.attributes.AttributeWithUnique method), 283
reset() (mvpa.misc.attributes.CollectableAttribute method), 284
reset() (mvpa.misc.attributes.StateVariable method), 284
reset() (mvpa.misc.state.ClassWithCollections method), 309
reset() (mvpa.misc.state.Collection method), 310
resetvalue() (mvpa.misc.param.Parameter method), 303
resetvalue() (mvpa.misc.state.ParameterCollection method), 311
retrain() (mvpa.clfs.base.Classifier method), 162
reuseAbsolutePath() (in module mvpa.misc.support), 317
reverse mapping, 20
reverse() (mvpa.mappers.base.ChainMapper method), 141
reverse() (mvpa.mappers.base.CombinedMapper method), 142
reverse() (mvpa.mappers.base.Mapper method), 143
reverse() (mvpa.mappers.base.ProjectionMapper method), 145
reverse() (mvpa.mappers.boxcar.BoxcarMapper method), 146
reverse() (mvpa.mappers.lle.LLEMapper method), 149
reverse() (mvpa.mappers.mask.MaskMapper method), 150
reverse() (mvpa.mappers.samplegroup.SampleGroupMapper method), 156
reverse() (mvpa.mappers.som.SimpleSOMMapper method), 157
ReverseRankOrder() (in module mvpa.misc.transformers), 319
review, 3
RFE, 39
RFE (class in mvpa.featsel.rfe), 269
RFEHistory2maps() (in module mvpa.misc.support), 316
rho (mvpa.clfs.libsvm.vmc.svmc.svm_model attribute), 193
ridge regression, 31
RidgeReg (class in mvpa.clfs.ridge), 209
RMSEErrorFx (class in mvpa.misc.errorfx), 288
RNG, 48
ROC Curve (class in mvpa.clfs.transerror), 227
ROCs (mvpa.clfs.transerror.ROCCurve attribute), 227
rootMeanPowerFx() (in module mvpa.misc.errorfx), 289
RPy, 4, 8
rstUnderline() (in module mvpa.base.dochelpers), 277
S
S (mvpa.datasets.base.Dataset attribute), 112
Sample, 94
sample, 19, 20
sample attribute, 18
SampleAttribute (class in mvpa.misc.attributes), 284
SampleAttributes (class in mvpa.misc.attributes), 284
SampleAttributesCollection (class in mvpa.misc.state), 311
SampleGroupMapper (class in mvpa.mappers.samplegroup), 155
samples (mvpa.datasets.base.Dataset attribute), 115
samples_original (mvpa.datasets.mapped.MappedDataset attribute), 123
samplesperchunk (mvpa.datasets.base.Dataset attribute), 115
samplesperlabel (mvpa.datasets.base.Dataset attribute), 115
samplingrate (mvpa.datasets.channel.ChannelDataset attribute), 118
samplingrate (mvpa.datasets.nifti.NiftiDataset attribute), 130
save() (mvpa.base.config.ConfigManager method), 276
save() (mvpa.base.report.Report method), 280
SciPy, 8
searchlight, 62–64
Searchlight (class in mvpa.measures.searchlight), 255
SecondAxisMaxOfAbs() (in module mvpa.misc.transformers), 319
SecondAxisMean() (in module mvpa.misc.transformers), 319
SecondAxisSumOfAbs() (in module mvpa.misc.transformers), 319
select() (mvpa.datasets.base.Dataset method), 115
selectFeatures() (mvpa.datasets.base.Dataset method), 116
selectFeatures() (mvpa.datasets.mapped.MappedDataset method), 123
selectFeatures() (mvpa.datasets.meta.MetaDataset method), 125
selectFeaturesByMask() (mvpa.datasets.masked.MaskedDataset method), 124
selectOut() (mvpa.mappers.base.ChainMapper method), 141
selectOut() (mvpa.mappers.base.CombinedMapper method), 142
selectOut() (mvpa.mappers.base.Mapper method), 143
selectOut() (mvpa.mappers.base.ProjectionMapper method), 145
selectOut() (mvpa.mappers.boxcar.BoxcarMapper method), 146
selectOut() (mvpa.mappers.mask.MaskMapper method), 150
selectOut() (mvpa.mappers.samplegroup.SampleGroupMapper method), 156
selectOut() (mvpa.mappers.som.SimpleSOMMapper method), 157
selectOut() (mvpa.mappers.svd.SVDMapper method), 158
selectSamples() (mvpa.datasets.base.Dataset method), 116
selectSamples() (mvpa.datasets.meta.MetaDataset method), 125
selectSamples() (mvpa.misc.io.base.ColumnData method), 296
self-organizing map, 80
Sensitivity, 94
sensitivity, 33, 35, 65, 90
Sensitivity (class in mvpa.measures.base), 239
Sensitivity Map, 94
sensitivity_analyzer (mvpa.featsel.base.SensitivityBasedFeatureSelection attribute), 262
SensitivityBasedFeatureSelection (class in mvpa.featsel.base), 261
SensorLocations (class in mvpa.misc.io.base), 297
SequenceStats (class in mvpa.datasets.miscfx), 126
set_hyperparameters() (mvpa.clfs.blr.BLRL method), 164
set_hyperparameters() (mvpa.clfs.gpr.GPR method), 177
set_hyperparameters() (mvpa.clfs.kernel.KernelConstant method), 179
set_hyperparameters() (mvpa.clfs.kernel.KernelExponential method), 180
set_hyperparameters() (mvpa.clfs.kernel.KernelLinear method), 181
set_hyperparameters() (mvpa.clfs.kernel.KernelMatern_3_2 method), 181
set_hyperparameters() (mvpa.clfs.kernel.KernelRationalQuadratic method), 182
set_hyperparameters() (mvpa.clfs.kernel.KernelSquaredExponential method), 183
setActiveFromString() (mvpa.base.verbosity.SetLogger method), 282
setCoordT() (mvpa.atlases.base.XMLBasedAtlas method), 325
setDefault() (mvpa.misc.param.Parameter method), 303
setDistance() (mvpa.atlases.base.ReferencesAtlas method), 324
setLabels_map() (mvpa.clfs.blr.BLRL method), 227
setLabelsMap() (mvpa.datasets.base.Dataset method), 116
setNPerLabel() (mvpa.datasets.splitters.Splitter method), 138
setReferenceLevel() (mvpa.atlases.base.ReferencesAtlas method), 324
sets (mvpa.clfs.blr.BLRL method), 228
setSamplesDType() (mvpa.datasets.base.Dataset method), 116
setSamplesDType() (mvpa.datasets.meta.MetaDataset method), 125
setTestDataset() (mvpa.clfs.meta.FeatureSelectionClassifier method), 199
settings, 43
Shogun, 8
shrinking (mvpa.clfs.libsvmcsvmc.svmc.svm_parameter attribute), 193
Sigma_p (mvpa.clfs.kernel.KernelLinear attribute), 180
SimpleSOMMapper, 80
SimpleSOMMapper (class in mvpa.mappers.som), 156
singleGammaHRF() (in module mvpa.misc.fx), 294
singleOrPlural() (in module mvpa.base.dochelpers), 277
sinModulated() (in module mvpa.misc.data_generators), 287
split_kwargs() (in module mvpa.misc.args), 282
splitcfg() (mvpa.datasets.splitters.Splitter method), 138
SplitClassifier, 40
SplitClassifier (class in mvpa.clfs.meta), 204
splitDataset() (mvpa.datasets.splitters.Splitter method), 138
SplitFeaturewiseDatasetMeasure (class in mvpa.measures.base), 239
SplitFeaturewiseMeasure (class in mvpa.measures.base), 257
splitter, 22
Splitter (class in mvpa.datasets.splitters), 137
splitter (mvpa.algorithms.cvtranserror.CrossValidatedTransferError attribute), 272
splitter (mvpa.clfs.meta.SplitClassifier attribute), 205
splitting measures, 37
squared_euclidean_distance() (in module mvpa.clfs.distance), 166
StateCollection (class in mvpa.misc.state), 312
states, 26
StateVariable (class in mvpa.misc.attributes), 284
StaticDatasetMeasure (class in mvpa.measures.base), 240
Statistical Discrimination Map (SDM), 94
statistical testing, 69
stats (mvpa.clfs.transerror.SummaryStatistics attribute), 228
steps (mvpa.featsel.helpers.NBackHistoryStopCrit attribute), 265
steps (mvpa.featsel.helpers.NStepsStopCrit attribute), 265
stepwise_regression() (in module mvpa.clfs.libsmrlc), 187
StoppingCriterion (class in mvpa.featsel.helpers), 266
strategy (mvpa.datasets.splitter.Splitter attribute), 138
substractBaseline() (mvpa.datasets.channel.ChannelDataset method), 118
suggested software, 8
summaries (mvpa.clfs.transerror.SummaryStatistics attribute), 228
summary() (mvpa.clfs.base.Classifier method), 162
summary() (mvpa.clfs.meta.CombinedClassifier method), 198
summary() (mvpa.clfs.meta.ProxyClassifier method), 204
summary() (mvpa.clfs.meta.TreeClassifier method), 206
summary() (mvpa.datasets.base.Dataset method), 116
SummaryStatistics (class in mvpa.clfs.transerror), 228
SV (mvpa.clfs.libsvmc.svmc.svm_model attribute), 192
sv (mvpa.mappers.svd.SVDMapper attribute), 158
sv_coef (mvpa.clfs.libsvmc.svmc.svm_model attribute), 193
SVD, 67
SVDMapper (class in mvpa.mappers.svd), 158
SVM, 31, 35, 57
SVM (class in mvpa.clfs.libsvmc.svm), 190
SVM (class in mvpa.clfs.sg.svm), 212
svm_check_parameter() (in module mvpa.clfs.libsvmc.svmc), 194
svm_check_probability_model() (in module mvpa.clfs.libsvmc.svmc), 194
svm_cross_validation() (in module mvpa.clfs.libsvmc.svmc), 194
svm_get_labels() (in module mvpa.clfs.libsvmc.svmc), 194
t0 (mvpa.datasets.channel.ChannelDataset attribute), 119
t0 (mvpa.misc.io.eepbin.EEPBin attribute), 299
table2string() (in module mvpa.base.docheelpers), 277
tail (mvpa.clfs.stats.NullDist attribute), 223
TailSelector (class in mvpa.featsel.helpers), 266
Tal2MNI_Lancaster07FSL() (in module mvpa.atlases.transformation), 329
Tal2MNI_Lancaster07pooled() (in module mvpa.atlases.transformation), 329
Tal2MNI_YOH flirt() (in module mvpa.atlases.transformation), 329
TaskPLS (class in mvpa.measures.pls), 254
testAllDependencies() (in module mvpa.base.externals), 278
testdataset (mvpa.clfs.meta.FeatureSelectionClassifier attribute), 200
text (mvpa.atlases.base.Label attribute), 321
text() (mvpa.base.report.Report method), 280
textbook, 3
threshold (mvpa.featsel.helpers.FixedErrorThresholdStopCrit attribute), 263
Time-compression, 94
tmodes (mvpa.misc.fsl.melodic.MelodicResults attribute), 293
toarray() (mvpa.misc.bv.base.BrainVoyagerRTC method), 285
Index
version (mvpa.atlases.base.XMLBasedAtlas attribute), 325
version_to_tuple() (in module mvpa.misc.support), 317
voxdim (mvpa.atlases.base.XMLBasedAtlas attribute), 325
VProperty (class in mvpa.misc.vproperty), 320

W
Warehouse (class in mvpa.clfs.warehouse), 230
warning, 45, 47
WarningLog (class in mvpa.base), 274
WaveletPacketMapper (class in mvpa.mappers.wavelet), 159
WaveletTransformationMapper (class in mvpa.mappers.wavelet), 159
weight (mvpa.clfs.libsvmc.svmc.svm_parameter attribute), 193
Weight Vector, 94
weight_label (mvpa.clfs.libsvmc.svmc.svm_parameter attribute), 193
weights, 35
weights (mvpa.clfs.enet.ENET attribute), 168
weights (mvpa.clfs.lars.LARS attribute), 186
weights (mvpa.clfs.smlr.SMLR attribute), 216
where() (mvpa.datasets.base.Dataset method), 117
whichSet() (mvpa.misc.state.Collection method), 310
Windows, 9
Windows installer, 9
working data, 22
wr1996() (in module mvpa.misc.data_generators), 287
write() (mvpa.base.report.Report method), 280
WTF (class in mvpa.base.info), 278
wtf() (in module mvpa.base.info), 278

X
x (mvpa.clfs.libsvmc.svmc.svm_problem attribute), 193
XAVRSensorLocations (class in mvpa.misc.io.base), 298
xml() (mvpa.base.report.Report method), 280
XMLAtlasException (class in mvpa.atlases.base), 324
XMLBasedAtlas (class in mvpa.atlases.base), 324
xuniqueCombinations() (in module mvpa.misc.support), 317

Y
y (mvpa.clfs.libsvmc.svmc.svm_problem attribute), 193

Z
zscore() (in module mvpa.datasets.miscfx), 126
zscore() (mvpa.datasets.base.Dataset method), 117
ZScoreMapper (class in mvpa.mappers.zscore), 160