PyMVPA Manual

Release 0.4.2

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

June 02, 2009

CONTENTS

1 Introduction **3**
- 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 . . . 4
- 1.5 Acknowledgements . . . 5

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

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

4 Datasets **19**
- 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 **25**
- 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 **35**
- 6.1 Sensitivity Measures . . . 35

7 Feature Selection **39**
- 7.1 Recursive Feature Elimination . . . 40
- 7.2 Incremental Feature Search . . . 41

8 Miscellaneous **43**
- 8.1 Managing (Custom) Configurations . . . 43
- 8.2 Progress Tracking . . . 45
- 8.3 Additional Little Helpers . . . 48
- 8.4 FSL Bindings . . . 48

9 Full Examples **51**
9.1 Example fMRI Dataset . . . 51
9.2 Preprocessing . . . 52
9.3 Analysis . . . 56
9.4 Visualization . . . 73
9.5 Miscellaneous . . . 80

10 PyMVPA for Matlab Users **85**

11 Frequently Asked Questions **87**
11.1 General . . . 87
11.2 Data import, export and storage . . . 88
11.3 Data preprocessing . . . 89
11.4 Data analysis . . . 89

12 Glossary **93**

13 References **95**

14 License **99**

15 PyMVPA Development Changelog **101**
15.1 Releases . . . 101

16 Module Reference **107**
16.1 Global Facilities . . . 107
16.2 Datasets: Input, Output, Storage and Preprocessing . . . 108
16.3 Mappers: Data Transformations . . . 131
16.4 Classifiers and Errors . . . 149
16.5 Measures: Searchlights and Sensitivties . . . 201
16.6 Feature Selection . . . 223
16.7 Additional Algorithms . . . 233
16.8 Common Facilities . . . 235
16.9 Miscellaneous . . . 240

Module Index **275**

Index **277**

The PDF version of the manual is available for download.

CHAPTER

ONE

INTRODUCTION

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

PyMVPA stands for *Multivariate Pattern Analysis* in Python.

1.1 What this Manual is NOT

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

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, University of Magdeburg, Germany
- Yaroslav O. Halchenko, Rutgers University Newark, USA
- Per B. Sederberg, Princeton University, USA
- Emanuele Olivetti, University of Trento, Italy

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

- Greg Detre, Princeton University, USA
- James M. Hughes, Dartmouth College, USA
- Ingo Fründ, University of Magdeburg, Germany
- James Kyle, UCLA, USA
- Scott Gorlins, MIT, USA

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.

1.4.1 Peer-reviewed publications

Hanke, M., Halchenko, Y. O., Sederberg, P. B., Olivetti, E., Fründ, I., Rieger, J. W., Herrmann, C. S., Haxby, J. V., Hanson, S. J. and Pollmann, S. (2009) PyMVPA: a unifying approach to the analysis of neuroscientific data. Frontiers in Neuroinformatics, 3:3.

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

Hanke, M., Halchenko, Y. O., Sederberg, P. B., Hanson, S. J., Haxby, J. V. & Pollmann, S. (2009). PyMVPA: A Python toolbox for multivariate pattern analysis of fMRI data. Neuroinformatics.

First paper introducing fMRI data analysis with PyMVPA.

1.4.2 Posters

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

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

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

First presentation of PyMVPA at the conference *Psychologie und Gehirn* [Psychology and Brain], Magdeburg, 2008. This poster received the poster prize of the *German Society for Psychophysiology and its Application*.

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 a wrapper around the LARS library.

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

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

2.2 Installing Binary Packages

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

2.2.1 Debian

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

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

2.2.2 Debian backports and inofficial Ubuntu packages

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

2.2.3 Windows

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

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

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

Now, you can use the PyMVPA windows installer to install PyMVPA on your system. If done, verify that everything went fine by opening a command 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.

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

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

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

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

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

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

2.2.4 MacOS X

The easiest installation method for OSX is via MacPorts. MacPorts is a package management system for MacOS, which is in some respects very similiar 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>.

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:

> http://www.debian.org/doc/manuals/reference/ch-install.en.html#s-bashconf

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

```
> export PATH=/opt/local/bin:/opt/local/sbin:$PATH
> export DYLD_LIBRARY_PATH=/opt/local/lib:$DYLD_LIBRARY_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:

```
> $ 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:

```
> $ 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:

```
$ 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:

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

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

2.2.5 RPM-based GNU/Linux Distributions

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

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

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

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

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

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

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

For other distributions use the respective package managers (e.g. Yum) to setup the repository URL. The repositories include all core dependencies of PyMVPA (usually Numpy and PyNIfTI), if they are not available from other repositories of the respective distribution. There are two different repository groups, one for Suse and Mandriva-related packages and another one for Fedora, Redhat and CentOS-related packages.

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

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

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

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

2.3.2 Build it (General instructions)

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

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

> **python setup.py build_ext**

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

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

> **python setup.py install**

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

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

2.3.3 Build with enabled LIBSVM bindings

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

PyMVPA needs a patched LIBSVM version, as the original distribution generates a huge amount of debugging messages and therefore makes the console and PyMVPA output almost unusable. Debian (since lenny: 2.84.0-1)

and Ubuntu (since gutsy) already include the patched version. For all other systems a minimal copy of the patched sources is included in the PyMVPA source package (*3rd/libsvm*).

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

```
make 3rd
```

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

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

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

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

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

2.3.4 Alternative build procedure

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

```
export PYTHONPATH=$PWD
```

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

2.3.5 Windows

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

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

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

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

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

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

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

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

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

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

2.3.8 MacOS X

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*.

CHAPTER

THREE

GETTING STARTED

3.1 For the Impatient

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

First import the whole PyMVPA module:

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

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

```
>>> attr = SampleAttributes(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).

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

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

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

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

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

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

If you think that is a good start, take the remaining four minutes to take a look at the examples shipped in the source distribution of PyMVPA (*doc/examples/*; some of them are also listed in *Full Examples* section of this manual). The examples provide a coarse overview of a substantial portion of the functionality provided by PyMVPA, ranging from basic classifier usage, over more sophisticated analysis strategies to simple visualization demos.

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

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

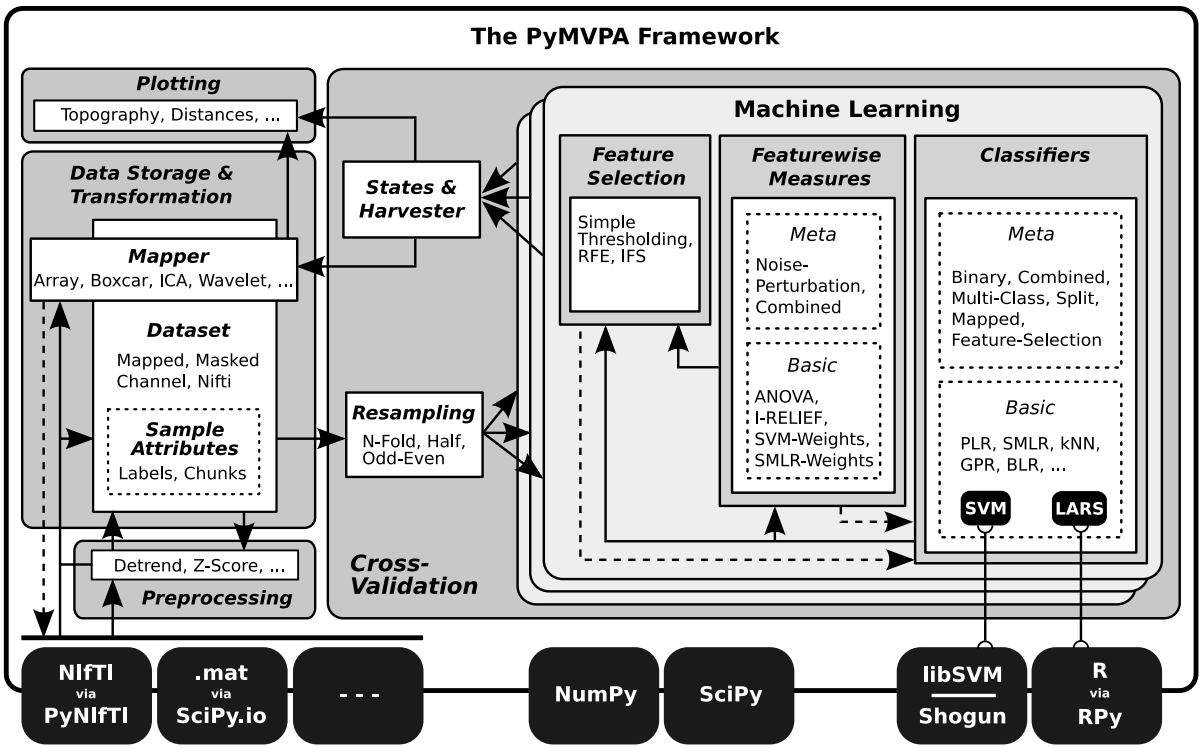
which would run the example shown in the first part of this section.

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

And now for the details ...

3.2 Module Overview

The PyMVPA package consists of three major parts: *Data handling*, *Classifiers* and various algorithms and measures that operate on datasets and classifiers. In the following sections the basic concept of all three parts will be described and examples using certain parts of the PyMVPA package will be given.



The manual does not cover all bits and pieces of PyMVPA. Detailed information about the module layout and additional documentation about all included functionality is available from the *Module Reference* – or the API Reference if you are interested in a more technical document. The main purpose of the manual is to give an idea how the individual parts of PyMVPA can be combined to perform complex analyses – easily.

CHAPTER

FOUR

DATASETS

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

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

4.1 The Basic Concepts

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

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

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

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

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

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

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

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

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

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

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

4.2 Data Mapping

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

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

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

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

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

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

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

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

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

4.3 Data Access Sugaring

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

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

4.4 Data Formats

The concept of mappers in 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:

```
>>> 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:

```
>>> 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.

CHAPTER

FIVE

CLASSIFIERS

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

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

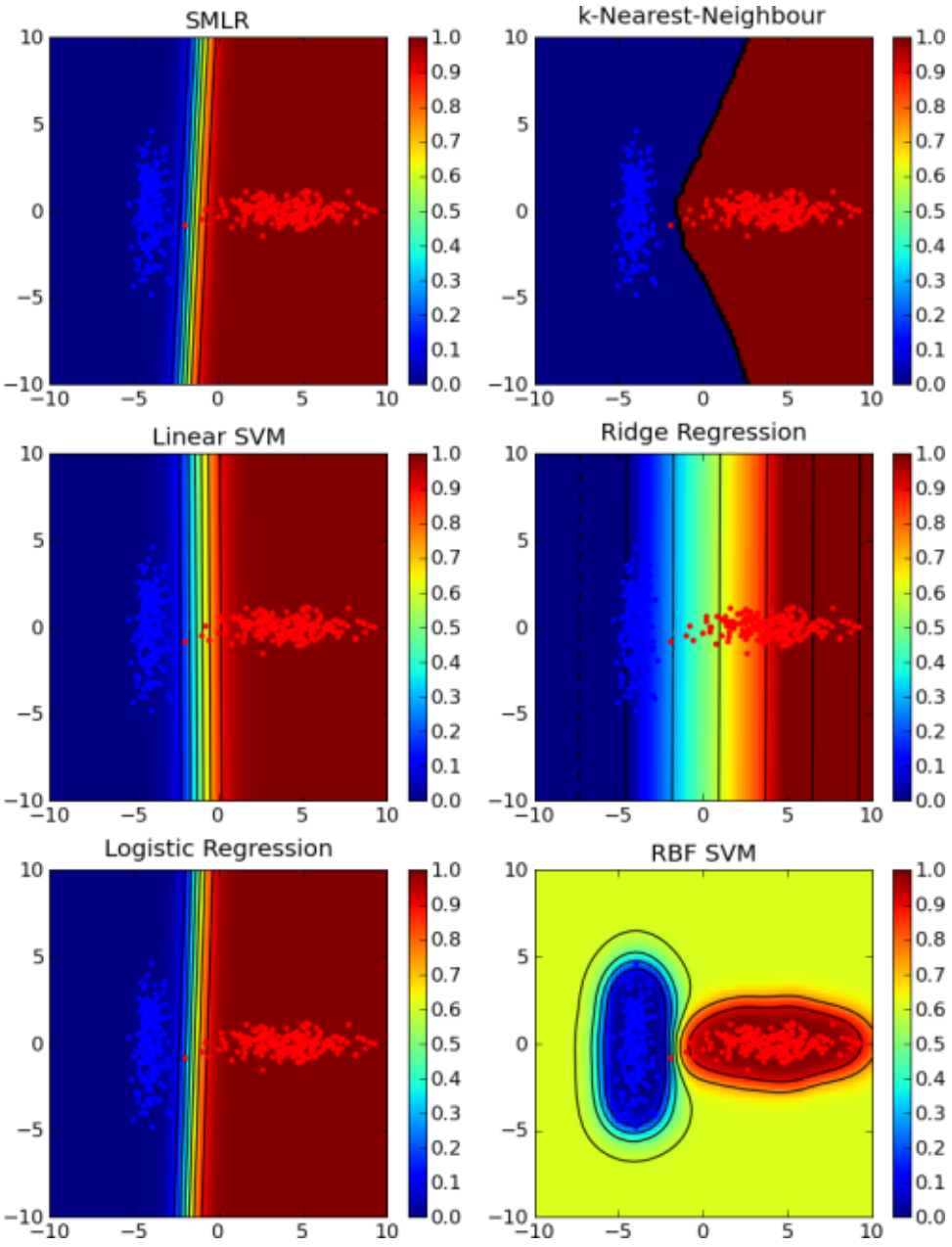
This examples demonstrates the typical daily life of a classifier.

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

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

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

The choice of the classifier in the above example is more or less arbitrary. Any classifier in PyMVPA could be used in place of kNN. This demonstrates another useful feature of PyMVPA's classifiers. Due to the high-level abstraction and the simple interface, almost all classifiers can be combined with most algorithms in PyMVPA. This makes it very easy to test different classifiers on some dataset (see Fig. 1).



A comparison of the behavior of different classifiers (k-Nearest-Neighbour, linear SVM, logistic regression, ridge regression and SVM with radial basis function kernel) on a simple classification problem. The code to generate these figure can be found in the *pylab_2d.py* example in the *Simple Plotting of Classifier Behavior* section.

5.1 Stateful objects

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

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

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

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

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

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

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

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

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

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

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

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

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

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

5.2 Error Calculation

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

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

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

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

5.2.1 Cross-validated Transfer Error

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

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

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

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

5.3 Error Reporting

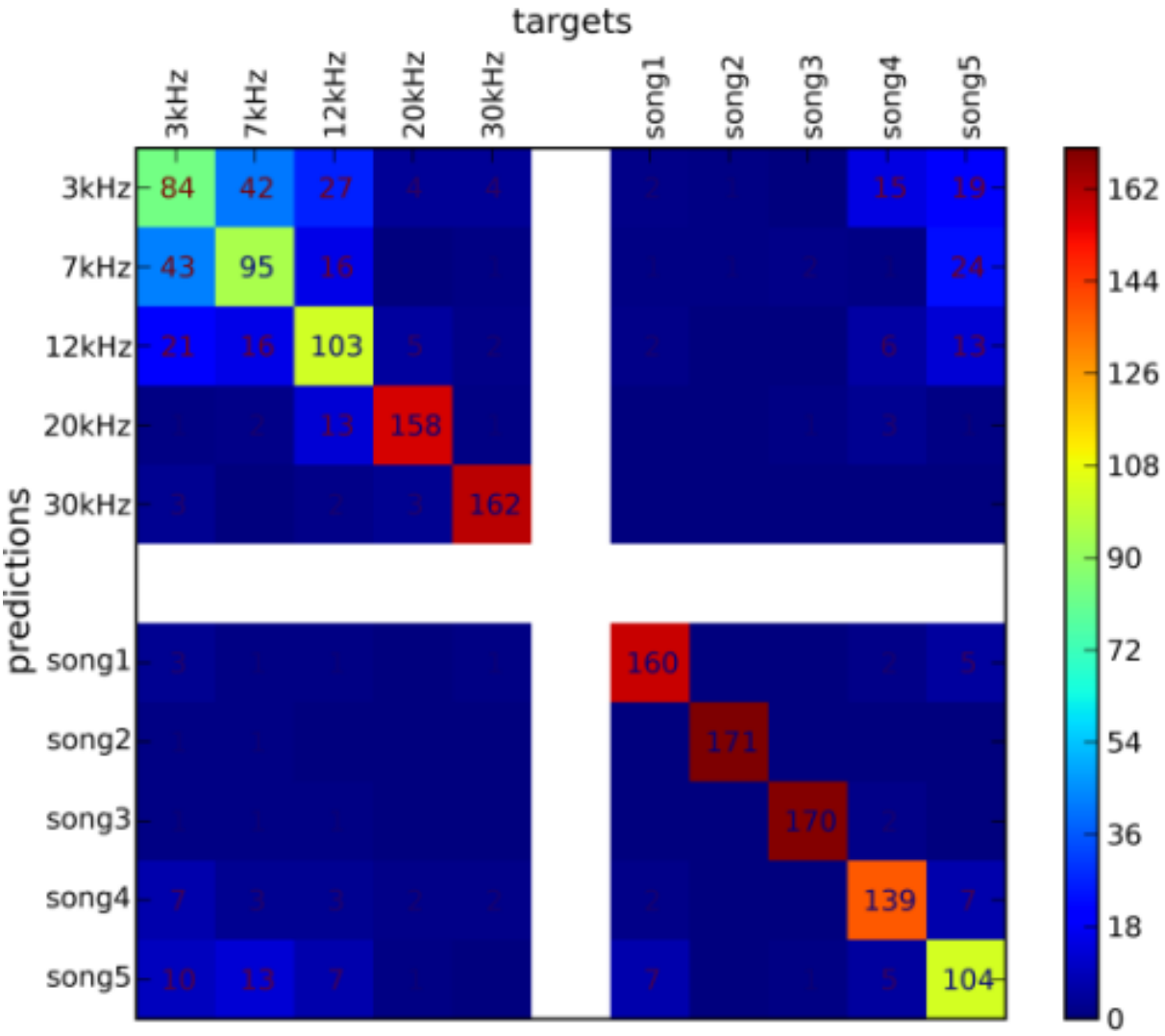
PyMVPA is equipped with easy ways to have a glance overview over the generalization performance of a cross-validated classifier. Such summary is provided by instances of a `ConfusionMatrix` class, and is accompanied by various performance metrics. For example, the 8-fold cross-validation of the dataset 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.asstring(description=True) \
... # doctest: +SKIP
--------.         3kHz  7kHz  12kHz 20kHz 30kHz song1 song2 song3 song4 song5
predict.\targets 38    39    40    41    42    43    44    45    46    47
         `------ ---- ----- ----- ----- ----- ----- ----- ----- ----- -----  P'  N'  FP  FN   PPV
       3kHz / 38 84    42    27     4     4     2     1     0    15    19   198 1351 114 90  0.42
       7kHz / 39 43    94    16     0     1     1     1     2     1    24   183 1331 89  80  0.51
      12kHz / 40 21    16    103    5     2     2     0     0     6    13   168 1312 65  70  0.61
      20kHz / 41  1     2    13    158    1     0     0     1     3     1   180 1202 22  15  0.88
```

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

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

```
>>> import pylab as P
>>> cvterr.confusion.plot() \
... # doctest: +SKIP
>>> P.show() \
... # doctest: +SKIP
```



5.4 Basic Supervised Learning Methods

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

5.4.1 Gaussian Process Regression

GPR (Wikipedia entry about gaussian process regression).

5.4.2 k-Nearest-Neighbour

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

5.4.3 Least Angle Regression

LARS *Efron et al. (2004)*

5.4.4 Penalized Logistic Regression

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

5.4.5 Ridge Regression

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

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

5.4.6 Sparse Multinomial Logistic Regression

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

5.4.7 Support Vector Machines

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

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

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

5.5 Meta-Classifiers

This section has been contributed by James M. Hughes.

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

5.5.1 Examples of Meta-Classifiers

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

`BoostedClassifier`
: typically uses multiple classifiers internally

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

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

CombinedClassifier
: combines predictions using a PredictionsCombiner functor

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

SplitClassifier
: combines a Classifier and an arbitrary Splitter

Furthermore, there are also several ProxyClassifier subclasses:

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

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

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

5.5.2 Implementation Examples

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

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

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

5.6 Retrainable Classifiers

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

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

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

- enable CHECK_RETRAIN debug target while developing the code for analysis. That might guard you against obvious misuses of retraining feature, as well as to spot bugs in the code
- validate on a simple dataset that analysis code provides the same results if classifier was created retrainable or not

5.7 Classifiers "Warehouse"

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

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

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

CHAPTER

SIX

MEASURES

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

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

6.1 Sensitivity Measures

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

Every sensitivity analyzer object computes a one-dimensional array with the respective score for every feature, when called with a `Dataset`. Due to this common 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.libsvmc.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* arguments 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).

```
>>> 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):

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

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

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

7.1 Recursive Feature Elimination

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

- A classifier is trained on a subset of the data and features are ranked according to an arbitrary metric.
- Some amount of those features is either selected or discarded according to a pre-selected rule.
- The classifier is retrained and features are once again ranked; this process continues until some criterion determined textit{a priori} (such as classification error) is reached.
- One or more classifiers trained only on the final set of selected features are used on a generalization dataset and performance is calculated.

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

- We split the training dataset into an arbitrary number of independent datasets and perform RFE on each of these; the sensitivity analysis of features is performed independently for each split and features are selected based on those independent measures.
- We split the training dataset into an arbitrary number of independent datasets (as before), but we average the feature sensitivities and select which features to prune/select based on that one average measure.

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

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

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

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

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

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

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

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

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

7.2 Incremental Feature Search

IFS

(to be written)

CHAPTER

EIGHT

MISCELLANEOUS

8.1 Managing (Custom) Configurations

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

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

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

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

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

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

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

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

```
MVPA_VERBOSE=1
```

will become:

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

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

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

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

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

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

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

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

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

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

# This is a comprehensive list of all settings currently recognized by PyMVPA.
# Users can add arbitrary additional settings, both in new and already existing
# sections.

[general]
#debug =
#verbose =
#seed = 12345

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

[error]
#output =

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

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

[examples]
interactive = yes
```

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

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

[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, depite an
# already present (but possibly outdated) test result
retest = no

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

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

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

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

# verbosity level of the unittest runner
verbosity = 1

[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:

verbose
: regular informative messages about generic actions being performed

debug
: messages about the progress of computation, manipulation on data structures

warning
: messages which are reported by mvpa if something goes a little unexpected but not critical

8.2.1 Redirecting Output

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

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

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

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

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

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

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

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

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

8.2.2 Verbose Messages

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

- command line: you can use opt.verbose for precrafted command line option for to give facility to change it from your script (see examples)
- environment variable **MVPA_VERBOSE**
- code: verbose.level property

The following verbosity levels are supported:

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

8.2.3 Warning Messages

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

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

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

8.2.4 Debug Messages

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

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

tbc
: Concise traceback printout – prefix common with the previous invocation is replaced with `...`

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

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

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

8.3 Additional Little Helpers

8.3.1 Random Number Generation

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

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

8.3.2 Unittests at a Grasp

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

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

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

8.3.3 Others

(to be written)

8.4 FSL Bindings

PyMVPA contains a few little helpers to make interfacing with FSL easier. The purpose of these helpers is to increase the efficiency when doing an analysis by (re)using useful information that is already available from some FSL output. FSL usually stores most interesting information in the NIfTI format. Therefore it can be easily imported into PyMVPA using PyNIfTI. However, some information is stored in 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:

```
>>> 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 additonal regressors
>>> from mvpa.datasets.miscfx import detrend
>>> res = detrend(ds, model='regress', opt_reg=mc.toarray())
>>> # 'res' contains all regressors and their associated weights
```

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

CHAPTER

NINE

FULL EXAMPLES

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

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

9.1 Example fMRI Dataset

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

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

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

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

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

Afterwards the timeseries has been motion-corrected using MCFLIRT:

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

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

bold.nii.gz
: The motion-corrected and skull-stripped 4D timeseries (1452 volumes with 40 x 64 x 64 voxels, corresponding to a 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:

```
>>> 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.

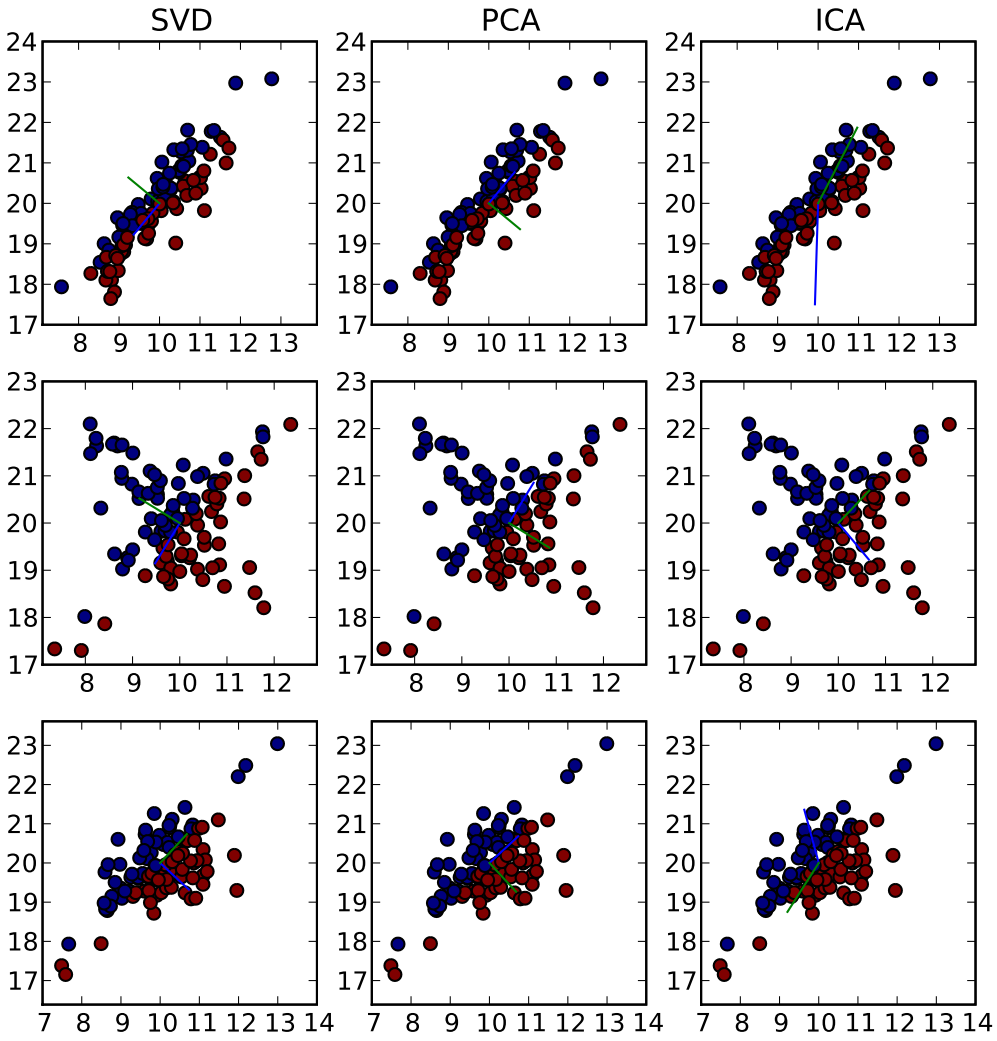
9.2 Preprocessing

9.2.1 Visualization of Data Projection Methods

```
from mvpa.misc.data_generators import noisy_2d_fx
from mvpa.mappers.pca import PCAMapper
from mvpa.mappers.svd import SVDMapper
from mvpa.mappers.ica import ICAMapper
from mvpa import cfg

import pylab as P
import numpy as N
center = [10, 20]
axis_range = 7

def plotProjDir(mproj):
    p = mproj + N.array(center).T

    P.plot([center[0], p[0,0]], [center[1], p[0,1]], hold=True)
    P.plot([center[0], p[1,0]], [center[1], p[1,1]], hold=True)



mappers = {
            'PCA': PCAMapper(),
            'SVD': SVDMapper(),
            'ICA': ICAMapper(),
          }
```

```
datasets = [
    noisy_2d_fx(100, lambda x: x, [lambda x: x],
                center, noise_std=.5),
    noisy_2d_fx(50, lambda x: x, [lambda x: x, lambda x: -x],
                center, noise_std=.5),
    noisy_2d_fx(50, lambda x: x, [lambda x: x, lambda x: 0],
                center, noise_std=.5),
   ]

ndatasets = len(datasets)
nmappers = len(mappers.keys())

P.figure(figsize=(8,8))
fig = 1

for ds in datasets:
    for mname, mapper in mappers.iteritems():
        mapper.train(ds)

        dproj = mapper.forward(ds.samples)
        mproj = mapper.proj
        print mproj

        P.subplot(ndatasets, nmappers, fig)
        if fig <= 3:
            P.title(mname)
        P.axis('equal')

        P.scatter(ds.samples[:, 0],
                  ds.samples[:, 1],
                  s=30, c=(ds.labels) * 200)
        plotProjDir(mproj)
        fig += 1


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

Output of the example:



See Also:

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

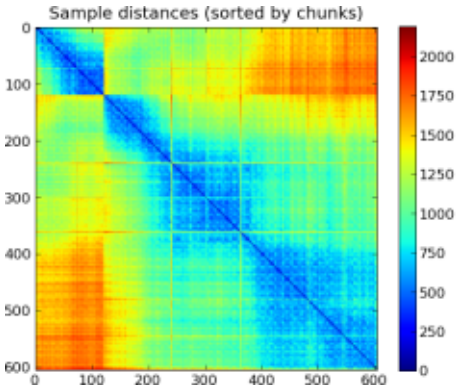
9.2.2 Simple Data-Exploration

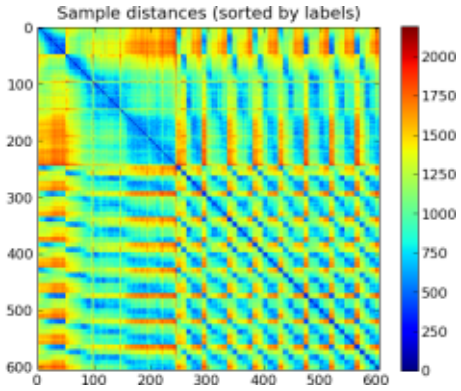
Example showing some possibilities of data exploration (i.e. to 'smell' data).

```
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
```

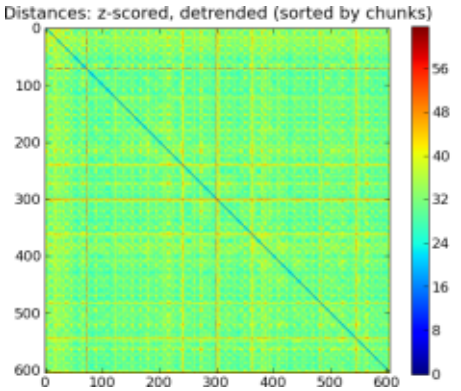
```
attr = SampleAttributes(os.path.join(pymvpa_dataroot, 'attributes.txt'))
ds = NiftiDataset(samples=os.path.join(pymvpa_dataroot, 'bold.nii.gz'),
                  labels=attr.labels,
                  chunks=attr.chunks,
                  mask=os.path.join(pymvpa_dataroot, 'mask.nii.gz'))

# 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 similiarity
# Note, the decreasing similarity with increasing temporal distance
# of the samples
P.subplot(121)
plotSamplesDistance(ds, sortbyattr='chunks')
P.title('Sample distances (sorted by chunks)')

# similar distance plot, but now samples sorted by their
# respective labels, i.e. samples with same labels are plotted
# in adjacent columns/rows.
# Note, that the first and largest group corresponds to the
# 'rest' condition in the dataset
P.subplot(122)
plotSamplesDistance(ds, sortbyattr='labels')
P.title('Sample distances (sorted by labels)')
if cfg.getboolean('examples', 'interactive', True):
    P.show()


# z-score features individually per chunk
print 'Detrending data'
detrend(ds, perchunk=True, model='regress', polyord=2)
print 'Z-Scoring data'
zscore(ds)

P.subplot(121)
plotSamplesDistance(ds, sortbyattr='chunks')
P.title('Distances: z-scored, detrended (sorted by chunks)')
P.subplot(122)
plotSamplesDistance(ds, sortbyattr='labels')
P.title('Distances: z-scored, detrended (sorted by labels)')
if cfg.getboolean('examples', 'interactive', True):
    P.show()

# XXX add some more, maybe show effect of preprocessing
```

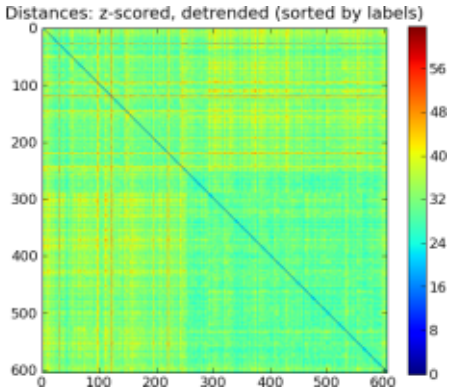
Outputs of the example script. Data prior to preprocessing





Data after minimal preprocessing





See Also:

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

9.3 Analysis

9.3.1 Tiny Example of a Full Cross-Validation

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

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

# load PyMVPA example dataset
attr = SampleAttributes(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 chunkswise linear detrending on dataset
detrend(dataset, perchunk=True, model='linear')

# zscore dataset relative to baseline ('rest') mean
zscore(dataset, perchunk=True, baselinelabels=[0],
       targetdtype='float32')

# select class 1 and 2 for this demo analysis
# would work with full datasets (just a little slower)
dataset = dataset.selectSamples(
                N.array([l in [1, 2] for l in dataset.labels],
                        dtype='bool'))
```

```
# setup cross validation procedure, using SMLR classifier
cv = CrossValidatedTransferError(
            TransferError(SMLR()),
            OddEvenSplitter())
# and run it
error = cv(dataset)

print "Error for %i-fold cross-validation on %i-class problem: %f" \
      % (len(dataset.uniquechunks), len(dataset.uniquelabels), error)
```

See Also:

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

9.3.2 Compare SMLR to Linear SVM Classifier

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

```
from mvpa.suite import *

if __debug__:
    debug.active.append('SMLR_')

# features of sample data
print "Generating samples..."
nfeat = 10000
nsamp = 100
ntrain = 90
goodfeat = 10
offset = .5

# create the sample datasets
samp1 = N.random.randn(nsamp,nfeat)
samp1[:,:goodfeat] += offset

samp2 = N.random.randn(nsamp,nfeat)
samp2[:,:goodfeat] -= offset

# create the pymvpa training dataset from the labeled features
patternsPos = Dataset(samples=samp1[:ntrain,:], labels=1)
patternsNeg = Dataset(samples=samp2[:ntrain,:], labels=0)
trainpat = patternsPos + patternsNeg

# create patters for the testing dataset
patternsPos = Dataset(samples=samp1[ntrain:,:], labels=1)
patternsNeg = Dataset(samples=samp2[ntrain:,:], labels=0)
testpat = patternsPos + patternsNeg

# set up the SMLR classifier
print "Evaluating SMLR classifier..."
smlr = SMLR(fit_all_weights=True)

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

# train with the known points
smlr.train(trainpat)

# run the predictions on the test values
pre = smlr.predict(testpat.samples)
```

```
# calculate the confusion matrix
smlr_confusion = ConfusionMatrix(
    labels=trainpat.uniquelabels, targets=testpat.labels,
    predictions=pre)

# now do the same for a linear SVM
print "Evaluating Linear SVM classifier..."
lsvm = LinearNuSVMC(probability=1)

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

# train with the known points
lsvm.train(trainpat)

# run the predictions on the test values
pre = lsvm.predict(testpat.samples)

# calculate the confusion matrix
lsvm_confusion = ConfusionMatrix(
    labels=trainpat.uniquelabels, targets=testpat.labels,
    predictions=pre)

# now train SVM with selected features
print "Evaluating Linear SVM classifier with SMLR's features..."

keepInd = (N.abs(smlr.weights).mean(axis=1)!=0)
newtrainpat = trainpat.selectFeatures(keepInd, sort=False)
newtestpat = testpat.selectFeatures(keepInd, sort=False)

# train with the known points
lsvm.train(newtrainpat)

# run the predictions on the test values
pre = lsvm.predict(newtestpat.samples)

# calculate the confusion matrix
lsvm_confusion_sparse = ConfusionMatrix(
    labels=newtrainpat.uniquelabels, targets=newtestpat.labels,
    predictions=pre)


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

See Also:

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

9.3.3 Classifier Sweep

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

```
from mvpa.suite import *

# no MVPA warnings during whole testsuite
```

```
warning.handlers = []

def main():

    # fix seed or set to None for new each time
    N.random.seed(44)


    # Load Haxby dataset example
    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, baselinelabels=[rest_label],
           targetdtype='float32')
    haxby8_no0 = haxby8.selectSamples(haxby8.labels != rest_label)

    dummy2 = normalFeatureDataset(perlabel=30, nlabels=2,
                                  nfeatures=100,
                                  nchunks=6, nonbogus_features=[11, 10],
                                  snr=3.0)

    for (dataset, datasetdescr), clfs_ in \
        [
        ((dummy2,
          "Dummy 2-class univariate with 2 useful features out of 100"),
          clfswh[:]),
        ((pureMultivariateSignal(8, 3),
          "Dummy XOR-pattern"),
          clfswh['non-linear']),
        ((haxby8_no0,
          "Haxby 8-cat subject 1"),
          clfswh['multiclass']),
        ]:
        print "%s\n %s" % (datasetdescr, dataset.summary(idhash=False))
        print " Classifier                                   " \
              "%corr  #features\t train predict  full"
        for clf in clfs_:
            print "  %-40s: "  % clf.descr,
            # Lets do splits/train/predict 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 "\n", confusion
            print "%5.1f%%   %-4d\t %.2fs  %.2fs   %.2fs" % \
                  (confusion.percentCorrect, nf, times[0], times[1], tfull)


if __name__ == "__main__":
    main()
```

See Also:

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

9.3.4 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"$\sigma_f=%0.2f$, $length_s=%0.2f$, $\sigma_n=%0.2f$" \
                % (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, "$RMSE=%.3f$" %(accuracy))
            P.text(0.5, -0.95, "$LML=%.3f$" %(g.log_marginal_likelihood))
```

```
        else:
            P.text(0.5, -0.8, "$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 is can be easily performed. This examples shows a minimal draft of a complete analysis.

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

:: from mvpa.datasets.masked import MaskedDataset from mvpa.datasets.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.data_generators import normalFeatureDataset

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

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

Now it only takes three lines for a searchlight analysis.

```
# setup measure to be computed in each sphere (cross-validated
# generalization error on odd/even splits)
cv = CrossValidatedTransferError(
         TransferError(LinearCSVMC()),
         OddEvenSplitter())

# setup searchlight with 5 mm radius and measure configured above
sl = Searchlight(cv, radius=5)

# run searchlight on dataset
sl_map = sl(dataset)

print 'Best performing sphere error:', max(sl_map)
```

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

```
.. 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 ressources, we are going to enable some progress output, to entertain us while we are waiting.

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

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

```
#
# load PyMVPA example dataset
#
attr = SampleAttributes(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.

```
# setup plotting
fig = 0
P.figure(figsize=(12,4))


for radius in [1,5,10]:
    # tell which one we are doing
    print "Running searchlight with radius: %i ..." % (radius)

    # setup Searchlight with a custom radius
    # radius has to be in the same unit as the nifti file's pixdim
    # property.
    sl = Searchlight(cv, radius=radius)

    # run searchlight on example dataset and retrieve error map
    sl_map = sl(dataset)

    # map sensitivity map into original dataspace
    orig_sl_map = dataset.mapReverse(N.array(sl_map))
    masked_orig_sl_map = N.ma.masked_array(orig_sl_map,
                                           mask=orig_sl_map == 0)

    # make a new subplot for each classifier
    fig += 1
    P.subplot(1,3,fig)

    P.title('Radius %i' % radius)

    P.imshow(masked_orig_sl_map[0],
             interpolation='nearest',
             aspect=1.25,
             cmap=P.cm.autumn)
    P.clim(0.5, 0.65)
    P.colorbar(shrink=0.6)


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

See Also:

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

9.3.7 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.

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

Create 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)
```

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 = {
    'a) ANOVA': OneWayAnova(transformer=N.abs),
    'b) Linear SVM weights': LinearNuSVMC().getSensitivityAnalyzer(
                                                transformer=N.abs),
    'c) I-RELIEF': IterativeRelief(transformer=N.abs),
    'd) Splitting ANOVA (odd-even)':
        SplitFeaturewiseMeasure(OneWayAnova(transformer=N.abs),
                                OddEvenSplitter()),
    'e) Splitting SVM (odd-even)':
        SplitFeaturewiseMeasure(
            LinearNuSVMC().getSensitivityAnalyzer(transformer=N.abs),
                          OddEvenSplitter()),
    'f) I-RELIEF Online':
        IterativeReliefOnline(transformer=N.abs),
    'g) Splitting ANOVA (nfold)':
        SplitFeaturewiseMeasure(OneWayAnova(transformer=N.abs),
                                NFoldSplitter()),
    'h) Splitting SVM (nfold)':
        SplitFeaturewiseMeasure(
            LinearNuSVMC().getSensitivityAnalyzer(transformer=N.abs),
                          NFoldSplitter()),
       }
```

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

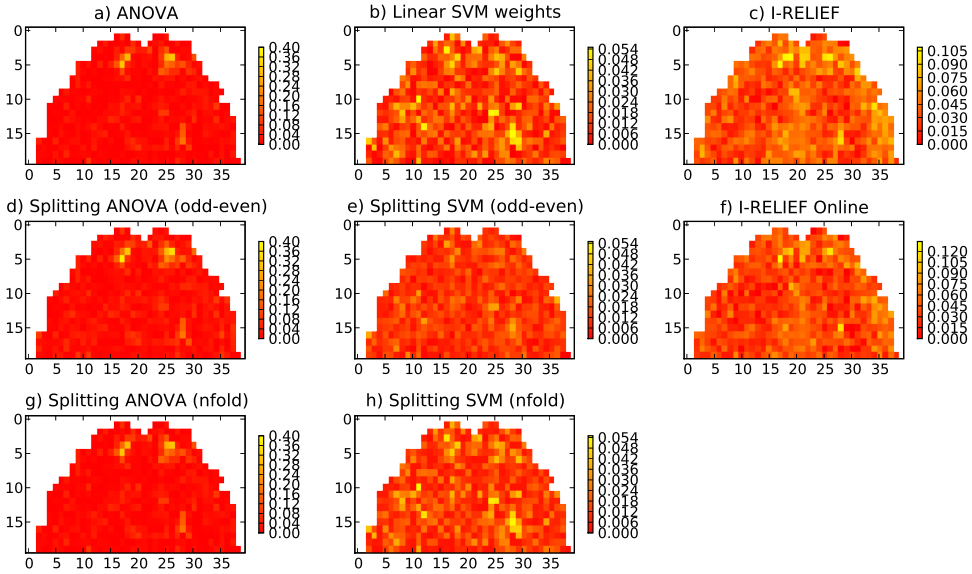
```
# do chunkswise linear detrending on dataset
detrend(dataset, perchunk=True, model='linear')

# only use 'rest', 'shoe' and 'bottle' samples from dataset
dataset = dataset.selectSamples(
               N.array([ l in [0,3,7] for l in dataset.labels],
               dtype='bool'))

# zscore dataset relative to baseline ('rest') mean
zscore(dataset, perchunk=True, baselinelabels=[0], targetdtype='float32')

# remove baseline samples from dataset for final analysis
dataset = dataset.selectSamples(N.array([l != 0 for l in dataset.labels],
                                        dtype='bool'))
```

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

```
fig = 0
P.figure(figsize=(14, 8))
```

```
keys = sensanas.keys()
keys.sort()

for s in keys:
    # tell which one we are doing
    print "Running %s ..." % (s)

    # compute sensitivies
    # I-RELIEF assigns zeros, which corrupts voxel masking for pylab's
    # imshow, so adding some epsilon :)
    smap = sensanas[s](dataset)+0.00001

    # map sensitivity map into original dataspace
    orig_smap = dataset.mapReverse(smap)
    masked_orig_smap = N.ma.masked_array(orig_smap, mask=orig_smap == 0)

    # make a new subplot for each classifier
    fig += 1
    P.subplot(3, 3, fig)

    P.title(s)

    P.imshow(masked_orig_smap[0],
             interpolation='nearest',
             aspect=1.25,
             cmap=P.cm.autumn)

    # uniform scaling per base sensitivity analyzer
    if s.count('ANOVA'):
        P.clim(0, 0.4)
    elif s.count('SVM'):
        P.clim(0, 0.055)
    else:
        pass

    P.colorbar(shrink=0.6)

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

Output of the example analysis:



See Also:

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

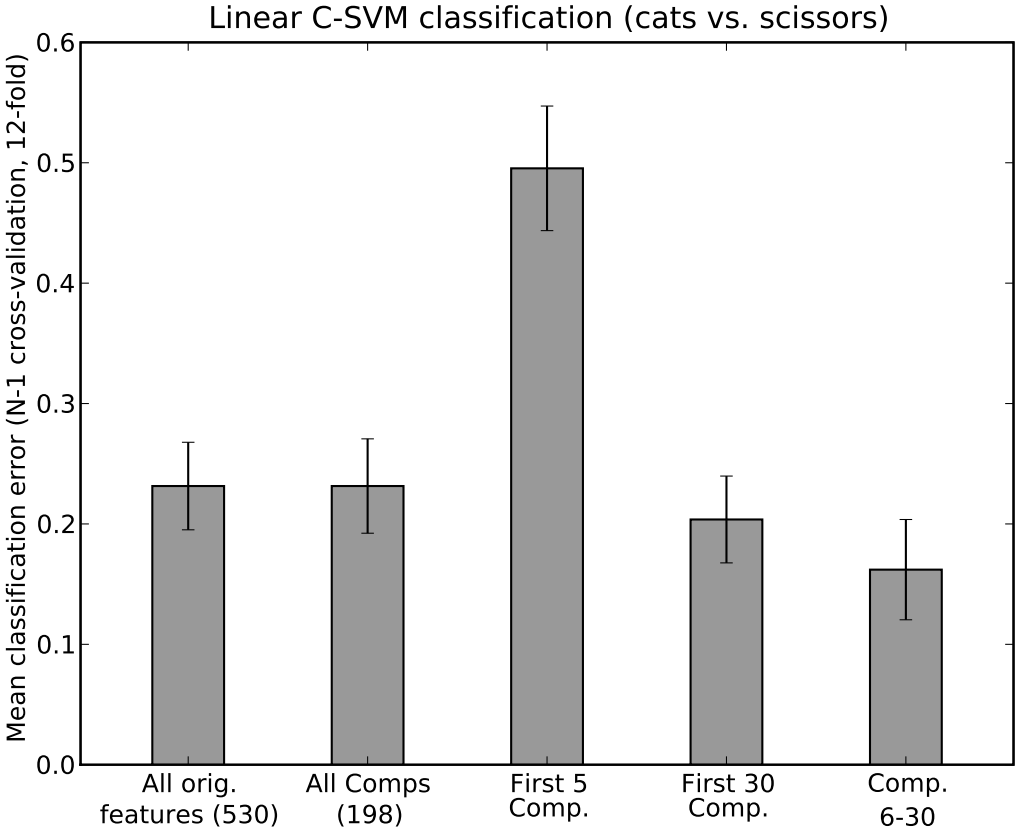
9.3.9 Classification of SVD-mapped Datasets

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

```
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

# define some classifiers: a simple one and several classifiers with
# built-in SVDs
clfs = [('All orig.\nfeatures (%i)' % dataset.nfeatures, Clf()),
        ('All Comps\n(%i)' % (dataset.nsamples \
                 - (dataset.nsamples / len(dataset.uniquechunks)),),
                        MappedClassifier(Clf(), SVDMapper())),
        ('First 5\nComp.', MappedClassifier(Clf(),
                        SVDMapper(selector=range(5)))),
        ('First 30\nComp.', MappedClassifier(Clf(),
                        SVDMapper(selector=range(30)))),
        ('Comp.\n6-30', MappedClassifier(Clf(),
                        SVDMapper(selector=range(5,30))))]


# run and visualize in barplot
results = []
labels = []

for desc, clf in clfs:
    print desc
    cv = CrossValidatedTransferError(
            TransferError(clf),
            NFoldSplitter(),
            enable_states=['results'])
    cv(dataset)

    results.append(cv.results)
    labels.append(desc)

plotBars(results, labels=labels,
         title='Linear C-SVM classification (cats vs. scissors)',
         ylabel='Mean classification error (N-1 cross-validation, 12-fold)',
         distance=0.5)

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

Output of the example analysis:



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 emprical (depending on whether on is looking at performances or errors).

If the family of the distribution is known (e.g. Gaussian/Normal) and provided in *dist_class* parameter of MC-

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

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

# some example data with signal
train = normalFeatureDataset(perlabel=50, nlabels=2, nfeatures=3,
                             nonbogus_features=[0,1], snr=3, nchunks=1)

# define class to estimate NULL distribution of errors
# use left tail of the distribution since we use MeanMatchFx as error
# function and lower is better
# in a real analysis the number of permutations should be MUCH larger
terr = TransferError(clf=SMLR(),
                     null_dist=MCNullDist(permutations=10,
                                          tail='left'))

# compute classifier error on training dataset (should be low :)
err = terr(train, train)
print 'Error on training set:', err

# check that the result is highly significant since we know that the
# data has signal
print 'Corresponding p-value: ',  terr.null_prob
```

See Also:

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

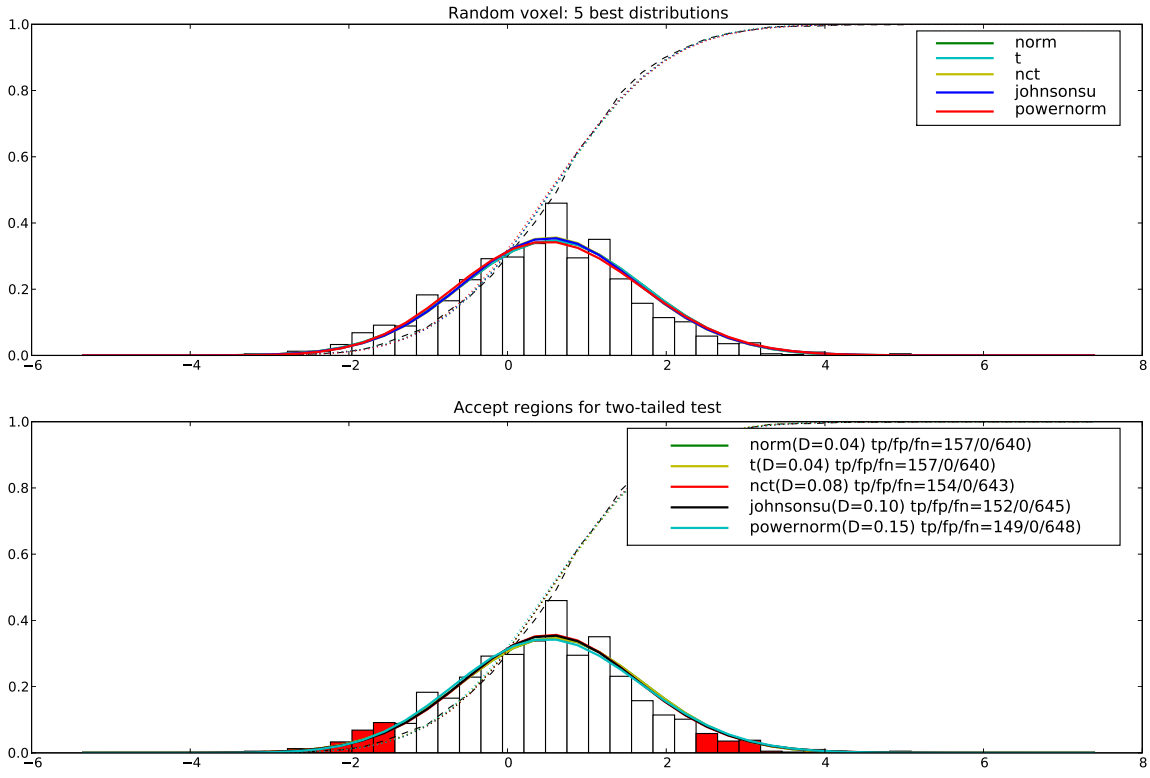
9.3.11 Determine the Distribution of some Variable

This is an example demonstrating discovery of the distribution facility.

```
from mvpa.suite import *

verbose.level = 2
if __debug__:
    # report useful debug information for the example
    debug.active += ['STAT', 'STAT_']

#
# Figure for just normal distribution
#

# generate random signal from normal distribution
verbose(1, "Random signal with normal distribution")
data = N.random.normal(size=(1000,1))

# find matching distributions
# NOTE: since kstest is broken in older versions of scipy
#       p-roc testing is done here, which aims to minimize
#       false positives/negatives while doing H0-testing
test = 'p-roc'
figsize = (15,10)
```

```
verbose(1, "Find matching datasets")
matches = matchDistribution(data, test=test, p=0.05)

P.figure(figsize=figsize)
P.subplot(2,1,1)
plotDistributionMatches(data, matches, legend=1, nbest=5)
P.title('Normal: 5 best distributions')

P.subplot(2,1,2)
plotDistributionMatches(data, matches, nbest=5, p=0.05,
                        tail='any', legend=4)
P.title('Accept regions for two-tailed test')

#
# Figure for fMRI data sample we have
#
verbose(1, "Load sample fMRI dataset")
attr = SampleAttributes(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()*dataset.nfeatures)])

verbose(2, "Minimal preprocessing to remove the bias per each voxel")
detrend(dataset, perchunk=True, model='linear')
zscore(dataset, perchunk=True, baselinelabels=[0],
       targetdtype='float32')

# on all voxels at once, just for the sake of visualization
data = dataset.samples.ravel()
verbose(2, "Find matching distribution")
matches = matchDistribution(data, test=test, p=0.05)

P.figure(figsize=figsize)
P.subplot(2,1,1)
plotDistributionMatches(data, matches, legend=1, nbest=5)
P.title('Random voxel: 5 best distributions')

P.subplot(2,1,2)
plotDistributionMatches(data, matches, nbest=5, p=0.05,
                        tail='any', legend=4)
P.title('Accept regions for two-tailed test')

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

Example output for a random voxel is



See Also:

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

9.4 Visualization

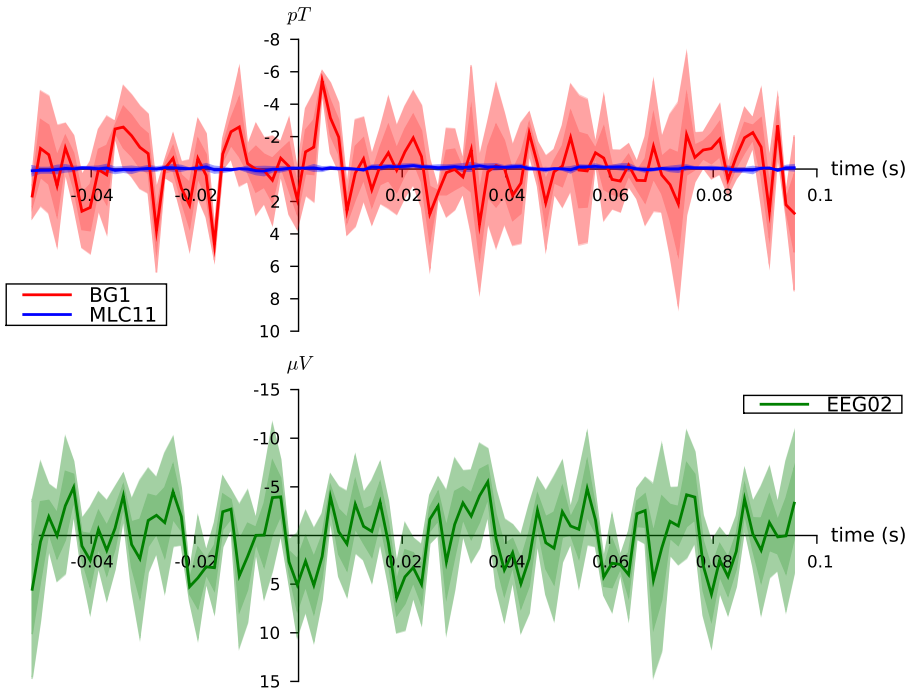
9.4.1 ERP/ERF-Plots

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

```
from mvpa.suite import *

# load data
meg = TuebingenMEG(os.path.join(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',
    'alinewidth' : 1  # assume that we like thin lines
    }

# Create a new figure
fig = P.figure(figsize=(12, 8))

# Following plots are plotted inverted (negative up) for the
# demonstration of this capability and elderly convention for ERP
# plots. That is controlled with ymult (negative gives negative up)


# Plot MEG sensors

# frame_on=False guarantees abent outside rectangular axis with
# labels. plotERP recreates its own axes centered at (0,0)
ax = fig.add_subplot(2, 1, 1, frame_on=False)
plotERPs(plots[:2], ylabel='$pT$', ymult=-1e12, ax=ax, **cargs)

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

# Additional example: plotting a single ERP on an existing plot
# without drawing axis:
#
# plotERP(data=meg.data[:, 0, :], SR=meg.samplingrate, pre=pre,
#         pre_mean=pre, errtype=errtype, ymult=-1.0)

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

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



See Also:

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

9.4.2 Simple Plotting of Classifier Behavior

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

First compose some sample data – no PyMVPA involved.

```
import numpy as N

# set up the labeled data
# two skewed 2-D distributions
num_dat = 200
dist = 4
feat_pos=N.random.randn(2, num_dat)
feat_pos[0, :] *= 2.
feat_pos[1, :] *= .5
feat_pos[0, :] += dist
feat_neg=N.random.randn(2, num_dat)
feat_neg[0, :] *= .5
feat_neg[1, :] *= 2.
feat_neg[0, :] -= dist

# set up the testing features
x1 = N.linspace(-10, 10, 100)
x2 = N.linspace(-10, 10, 100)
x,y = N.meshgrid(x1, x2);
feat_test = N.array((N.ravel(x), N.ravel(y)))
```

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

```
from mvpa.suite import *

# create the pymvpa dataset from the labeled features
patternsPos = Dataset(samples=feat_pos.T, labels=1)
patternsNeg = Dataset(samples=feat_neg.T, labels=0)
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)}
```

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=(6, 6))

    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(2, 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 clasifier
        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 == 'SMLR':
            res = N.asarray(clf.values[:, 1])
        else:
            # get the probabilities from the svm
            res = N.asarray([(q[1][1] - q[1][0] + 1) / 2
                    for q in clf.probabilities])

        # reshape the results
        z = N.asarray(res).reshape((100, 100))

        # plot the predictions
        P.pcolor(x, y, z, shading='interp')
        P.clim(0, 1)
        P.colorbar()
        P.contour(x, y, z, linewidths=1, colors='black', hold=True)

        # add the title
        P.title(c)

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

See Also:

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

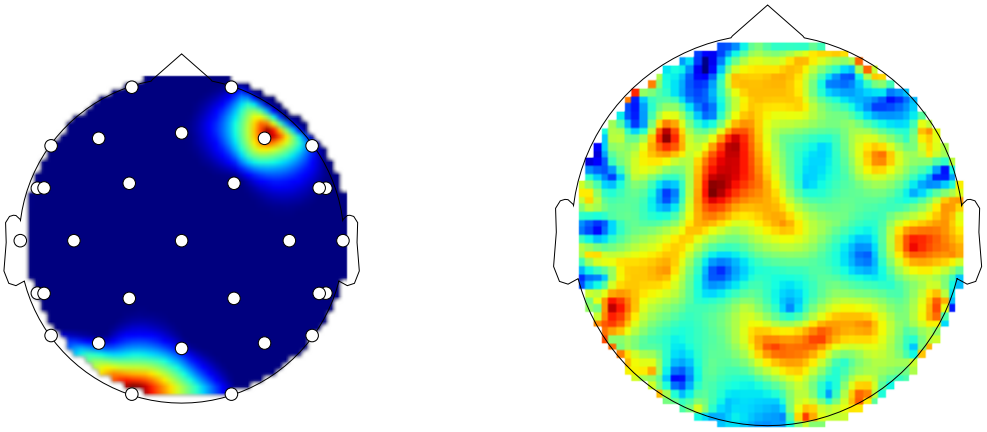
9.4.3 Generating Topography plots

Example demonstrating a topography plot.

```
from mvpa.suite import *

# Sanity check if we have griddata available
externals.exists("griddata", raiseException=True)

# EEG example splot
P.subplot(1, 2, 1)

# load the sensor information from their definition file.
# This file has sensor names, as well as their 3D coordinates
sensors=XAVRSensorLocations(os.path.join(pymvpa_dataroot, 'xavr1010.dat'))

# make up some artifical topography
# 'enable' to channels, all others set to off ;-)
topo = N.zeros(len(sensors.names))
topo[sensors.names.index('O1')] = 1
topo[sensors.names.index('F4')] = 1
```

```
# plot with sensor locations shown
plotHeadTopography(topo, sensors.locations(), plotsensors=True)


# MEG example plot
P.subplot(1, 2, 2)

# load MEG sensor locations
sensors=TuebingenMEGSensorLocations(
            os.path.join(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 ouput of the provided example should look like



See Also:

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

9.4.4 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.

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

First, we define some colors as RGB values from the interval (0,1), i.e. with white being (1, 1, 1) and black being (0, 0, 0). Please note, that a substantial proportion of the defined colors represent variations of 'blue', which are supposed to be represented in more detail in the SOM.

```
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]]

# store the names of the colors for visualization later on
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.

```
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.

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

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

```
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:

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

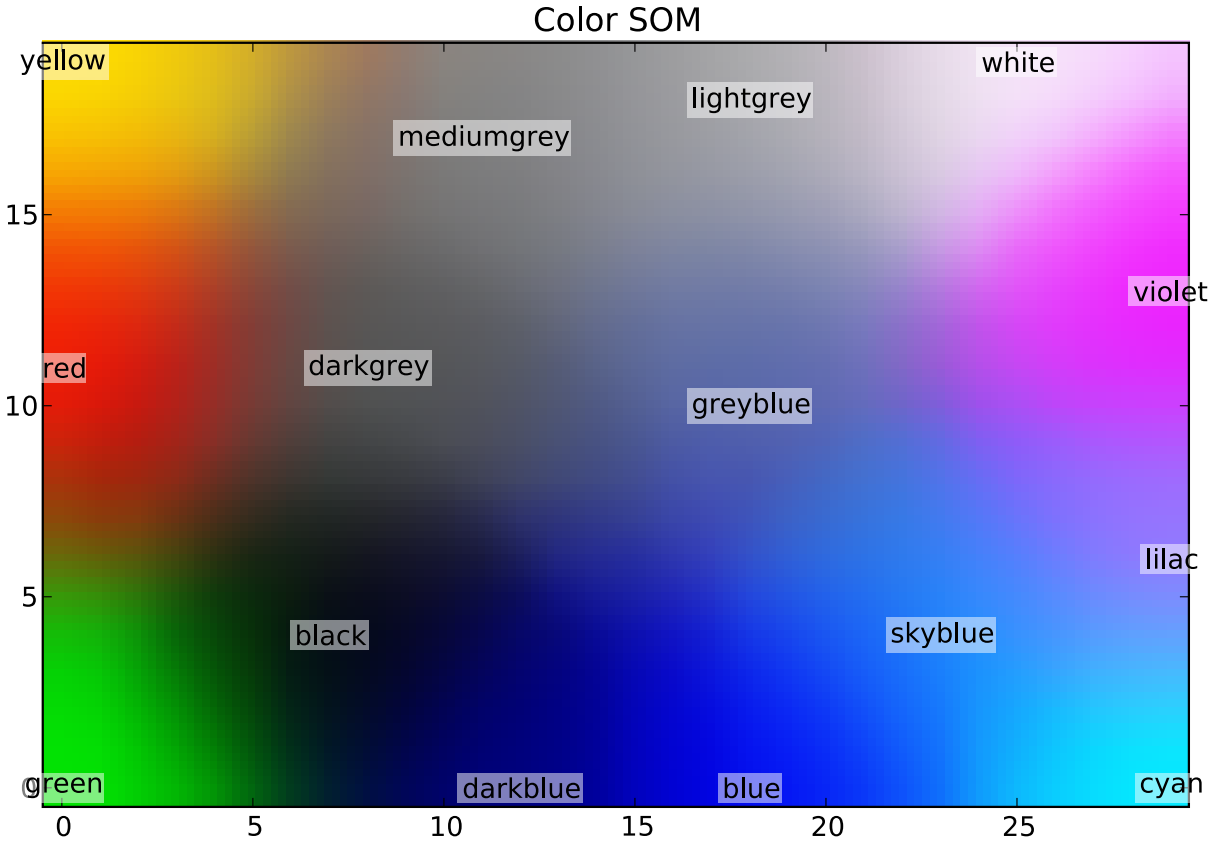
And now, let's take a look onto which coordinates the initial training prototypes were mapped to. The get those coordinates we can simply feed the training data to the mapper and plot the output.

```
mapped = som(colors)

P.title('Color SOM')
# 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.

```
# 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:



See Also:

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

9.5 Miscellaneous

9.5.1 Kernel-Demo

This is an example demonstrating various kernel implementation in PyMVPA.

```
from mvpa.suite import *
from mvpa.clfs.kernel import *
```

```
import pylab as P


# N.random.seed(1)
data = N.random.rand(4, 2)

for kernel_class, kernel_args in (
    (KernelConstant, {'sigma_0':1.0}),
    (KernelConstant, {'sigma_0':1.0}),
    (KernelLinear, {'Sigma_p':N.eye(data.shape[1])}),
    (KernelLinear, {'Sigma_p':N.ones(data.shape[1])}),
    (KernelLinear, {'Sigma_p':2.0}),
    (KernelLinear, {}),
    (KernelExponential, {}),
    (KernelSquaredExponential, {}),
    (KernelMatern_3_2, {}),
    (KernelMatern_5_2, {}),
    (KernelRationalQuadratic, {}),
    ):
    kernel = kernel_class(**kernel_args)
    print kernel
    result = kernel.compute(data)

# In the following we draw some 2D functions at random from the
# distribution N(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/classfication with GPR/GPC.
count = 1
for k in kernel_dictionary.keys():
    P.subplot(3,4,count)
    # X = N.random.rand(size)*12.0-6.0
    # X.sort()
    X = N.arange(-1,1,.02)
    X = X[:,N.newaxis]
    ker = kernel_dictionary[k]()
    K = ker.compute(X,X)
    for i in range(10):
        f = N.random.multivariate_normal(N.zeros(X.shape[0]),K)
        P.plot(X[:,0],f,"b-")

    P.title(k)
    P.axis('tight')
    count += 1

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

See Also:

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

9.5.2 Curve-Fitting

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

First, importing the necessary pieces:

```
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:

```
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)
```

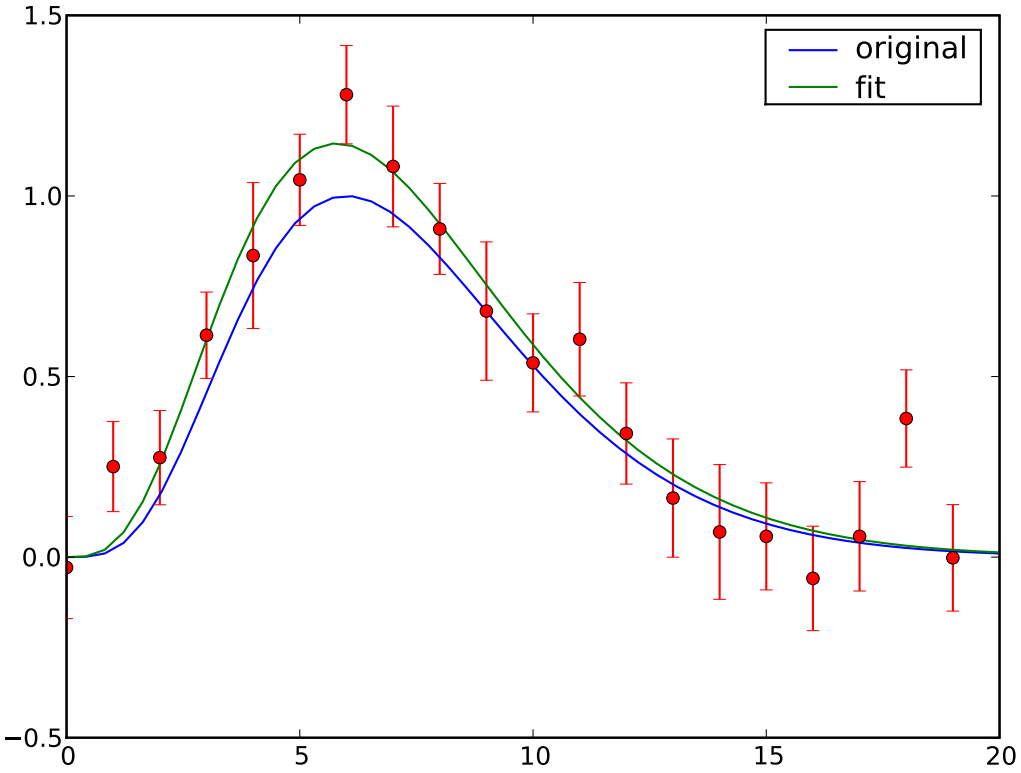
Fitting a gamma function to this data is easy (using resonable seeds for the parameter search (5s time-to-peak, 5s FWHM, and no scaling):

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

Generate high-resultion 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 ouput of the provided example should look like



See Also:

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

CHAPTER

TEN

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"*.

CHAPTER

ELEVEN

FREQUENTLY ASKED QUESTIONS

11.1 General

11.1.1 It is slooooooow. What can I do?

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

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

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

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

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

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

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

However, as always there is a price to pay for this convenience. In contrast to the individual imports there is some 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:

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

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

You can run:

```
git fetch origin
```

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

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

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

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

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

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

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

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

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

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

Writing a manual can be a tricky task if you already know the details and have to imagine what might be the most interesting information for someone who is just starting. If you feel that something is missing which has cost you some time to figure out, please drop us a note and we will add it as soon as possible. If you have developed some code snippets to demonstrate some feature or non-trivial 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:

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

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

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

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

11.4 Data analysis

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

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

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

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

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

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

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

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

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

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

11.4.3 Can PyMVPA deal with literal class labels?

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

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

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

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

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

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

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

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

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

```
>>> 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.

CHAPTER

TWELVE

GLOSSARY

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 abbrevation 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*.

CHAPTER

THIRTEEN

REFERENCES

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.

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

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

Keywords: `feature selection stability`

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

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

This is a review of several classifier benchmark procedures.

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

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

Keywords: `least angle regression`, `LARS`

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

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

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

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

Hanke, M., Halchenko, Y. O., Sederberg, P. B., Hanson, S. J., Haxby, J. V. & Pollmann, S. (2009). PyMVPA: A Python toolbox for multivariate pattern analysis of fMRI data. *Neuroinformatics*, *7*, 37–53.

Introduction into the analysis of fMRI data using PyMVPA.

Keywords: `PyMVPA`, `fMRI`

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

Hanke, M., Halchenko, Y. O., Sederberg, P. B., Olivetti, E., Fründ, I., Rieger, J. W., Herrmann, C. S., Haxby, J. V., Hanson, S. J. & Pollmann, S. (2009). PyMVPA: A Unifying Approach to the Analysis of Neuroscientific Data. *Frontiers in Neuroinformatics*, *3*, 3.

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

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

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

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

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

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

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

Keywords: `split-correlation classifier`

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

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

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

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

Jurica, P. & van Leeuwen, C. (2009). OMPC: an open-source MATLAB-to-Python compiler. *Frontiers in Neuroinformatics*, *3*, 5.

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

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

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

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

Kienzle, W., Franz, M. O., Schölkopf, B. & Wichmann, F. A. (in press). Center-surround patterns emerge as optimal predictors for human saccade targets. *Journal of Vision*.

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

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

Paper introducing the searchlight algorithm.

Keywords: `searchlight`

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

Kriegeskorte, N., Mur, M. & Bandettini, P. (2008). Representational similarity analysis - connecting the branches of systems neuroscience. *Frontiers in Systems Neuroscience*, *2*, 4.

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

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

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

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

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

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

Keywords: `SVM`

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

Mitchell, T., Hutchinson, R., Niculescu, R. S., Pereira, F., Wang, X., Just, M. & Newman, S. (2004). Learning to Decode Cognitive States from Brain Images. *Machine Learning*, *57*, 145–175.
DOI: http://dx.doi.org/10.1023/B:MACH.0000035475.85309.1b

Mur, M., Bandettini, P. A. & Kriegeskorte, N. (2009). Revealing representational content with pattern-information fMRI–an introductory guide. *Social Cognitive and Affective Neuroscience*.
DOI: http://dx.doi.org/10.1093/scan/nsn044

Nichols, T. E. & Holmes, A. P. (2002). Nonparametric permutation tests for functional neuroimaging: a primer with examples. *Human Brain Mapping*, *15*, 1–25.
Overview of standard nonparametric randomization and permutation testing applied to neuroimaging data (e.g. fMRI)
DOI: http://dx.doi.org/10.1002/hbm.1058

Norman, K. A., Polyn, S. M., Detre, G. J. & Haxby, J. V. (2006). Beyond mind-reading: multi-voxel pattern analysis of fMRI data. *Trends in Cognitive Science*, *10*, 424–430.
DOI: http://dx.doi.org/10.1016/j.tics.2006.07.005

O'Toole, A. J., Jiang, F., Abdi, H. & Haxby, J. V. (2005). Partially Distributed Representations of Objects and Faces in Ventral Temporal Cortex . *Journal of Cognitive Neuroscience*, *17*, 580–590.
DOI: http://dx.doi.org/10.1162/0898929053467550

O'Toole, A. J., Jiang, F., Abdi, H., Penard, N., Dunlop, J. P. & Parent, M. A. (2007). Theoretical, statistical, and practical perspectives on pattern-based classification approaches to the analysis of functional neuroimaging data. *Journal of Cognitive Neuroscience*, *19*, 1735–1752.
DOI: http://dx.doi.org/10.1162/jocn.2007.19.11.1735

Pereira, F., Mitchell, T. & Botvinick, M. (in press). Machine learning classifiers and fMRI: A tutorial overview. *Neuroimage*.
DOI: http://dx.doi.org/10.1016/j.neuroimage.2008.11.007

Pessoa, L. & Padmala, S. (2007). Decoding near-threshold perception of fear from distributed single-trial brain activation. *Cerebral Cortex*, *17*, 691–701.
Analysis of slow event-related fMRI data using patter classification techniques.
DOI: http://dx.doi.org/10.1093/cercor/bhk020

Sato, J. R., Mourão-Miranda, J., Martin, M. d. G. M., Amaro, E., Morettin, P. A. & Brammer, M. J. (2008). The impact of functional connectivity changes on support vector machines mapping of fMRI data. *Journal of Neuroscience Methods*, *172*, 94–104.
Discussion of possible scenarios where univariate and multivariate (SVM) sensitivity maps derived from the same dataset could differ. Including the case were univariate methods would assign a substantially larger score to some features.
Keywords: `support vector machine`, `SVM`, `sensitivity`
DOI: http://dx.doi.org/10.1016/j.jneumeth.2008.04.008

Vapnik, V. (1995). The Nature of Statistical Learning Theory. Springer: New York.
Keywords: `support vector machine`, `SVM`

Wang, Z., Childress, A. R., Wang, J. & Detre, J. A. (2007). Support vector machine learning-based fMRI data group analysis. *Neuroimage*, *36*, 1139–51.
Keywords: `support vector machine`, `SVM`, `group analysis`
DOI: http://dx.doi.org/10.1016/j.neuroimage.2007.03.072

Zou, H. & Hastie, T. (2005). Regularization and variable selection via the elastic net. *Journal of the Royal Statistical Society Series B*, *67*, 301–320.
URL: http://www-stat.stanford.edu/%7Ehastie/Papers/B67.2%20(2005)%20301-320%20Zou%20%26%20Hastie.pdf

CHAPTER

FOURTEEN

LICENSE

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.
```

CHAPTER

FIFTEEN

PYMVPA DEVELOPMENT CHANGELOG

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

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

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.

- None yet.

15.1 Releases

- 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 avialable 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 conjuction 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', 'asdict' 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 abitrary lenght (e.g. to build preprocessing pipelines).
  - New `EventDataset` to rapidly extract boxcar-shaped samples from data array using a simple list of `Event` definitions.
  - Removed obsolete MetricMapper class. `Mapper` itself provides the facilities for dealing with metrics.
  - `BoxcarMapper` can now handle data with more than four dimensions/axis and also performs reverse mapping of single boxcar samples.
  - `FslEV3` can now convert EV3 files into a list of `Event` instances.
  - Results of tests for external dependencies are now stored in PyMVPA's config manager (*mvpa.cfg*) and can be stored to a file (not done automatically at the moment). This will significantly decrease the time needed to import the *mvpa* module, as it prevents the repeated and lengthy tests for working externals.
  - Initial support for ROC computing and AUC as an accuracy measure.
  - Weights of LARS are now available via `LARSWeights`.
  - Added an initial list of MVPA-related references to the manual, tagged with keywords and comments as well is DOI or similar URL reference to the original document.

  - Added initial glossary to the manual.
  - New 'Module reference', as a middle-ground between manual and API reference.
  - New manual section about meta-classifiers (contributed by James M. Hughes).
  - New minimal example for a 'getting started' section in the manual.
  - Former **MVPA_QUICKTEST** was renamed to **MVPA_TESTS_QUICK**.
  - Update installation instructions for RPM-based distributions to make use of the OpenSUSE Build Service.
  - Updated install instructions for several RPM-based GNU/Linux distributions.
  - Switch from distutils to numpy.distutils (no change in dependencies).
  - Depend on PyNIfTI >= 0.20081017.1 and gain a smaller memory footprint when accessing NIfTI files via all datasets with NIfTI support.
  - Added workaround to make PyMVPA work with older Shogun releases and those from 0.6.4 on, which introduced backward-incompatible API changes.

- 0.3.1 (Sun, 14 Sep 2008)
  - New manual section about feature selection with a focus on RFE. Contributed by James M. Hughes.
  - New dataset type `ChannelDataset` for data structured in channels. Might be useful for data modalities like EEG and MEG. This dataset includes support for common preprocessing steps like resampling and baseline signal substraction.
  - Plotting of topographies on heads. Thanks to Ingo Fründ for contributing this code. Additionally, a new example shows how to do such plots.
  - New general purpose function for generating barplots and candlestick plots with error bars (`plotBars()`).
  - Dataset supports mapping of string labels onto numerical labels, removing the need to perform this mapping manually in user code. 'clfs_examples.py' is adjusted accordingly to demonstrate the new feature.
  - New `mvpa.clfs.base.Classifier.summary()` method to dump classifier settings.
  - Improved and more flexible `plotERPs()`.
  - New `IterativeRelief` sensitivity analyzer.
  - Added visualization of confusion matrices via `mvpa.clfs.transerror.ConfusionMatrix.plot()` inspired by Ingo Fründ.
  - The PyMVPA version is now globally available in `mvpa.pymvpa_version`.
  - BugFix: `TuebingenMEG` reader failed in some cases.
  - Several improvements (docs and implementation) for building PyMVPA on MacOS X.
  - New convenience accessor methods (`select()`, `where()` and `__getitem__()`) for :class'~mvpa.datasets.base.Dataset'.
  - New `mvpa.seed()` function to configure the random number generators from user code.
  - Added reader for a MEG sensor locations format (`TuebingenMEGSensorLocations`).
  - Initial model selection support for GRP (using openopt).
  - And tons of minor bugfixes, additional tests and improved documentation.

- 0.3.0 (Mon, 18 Aug 2008)
  - Import of binary EEP files (used by EEProbe) and EEPDataset class.
  - Initial version of a meta dataset class (MetaDataset). This is a container for multiple datasets, which behaves like a dataset itself.
  - Regression performance is summarized now within RegressionStatistics.
  - Error functions: CorrErrorPFx, RelativeRMSErrorFx.
  - Measures: CorrCoef.

  - Data generators: chirp, wr1996
  - Few more examples: curvefitting, kerneldemo, smellit, projections
  - Updated kNN classifier. kNN is now able to use custom distance function to determine that nearest neighbors. It also (re)gained the ability to do simple majority or weighted voting.
  - Some initial convenience functions for plotting typical results and data exploration.
  - Unified configuration handling with support for user-specific and analysis-specific config files, as well as the ability to override all config settings via environment variables. The configuration handling is used for PyMVPA internal settings, but can also be easily used for custom (user-)settings.
  - Improved modularity, e.g. SciPy is not required anymore, but still very useful.
  - Initial implementations of ICA and PCA mapper using functionality provided by MDP. These mappers are more or less untested and should be used with great care.
  - Further improved docstrings of some classes, but still a long way to go.
  - New 'boxcar' mapper, which is the similar to the already present transformWithBoxCar() function, but implemented as a mapper.
  - New SampleGroupMapper that can be used for e.g. block averaging of samples. See new FAQ item.
  - Stripped redundant suffixes from module names, e.g. mvpa.datasets.niftidataset -> mvpa.datasets.nifti
  - mvpa.misc.cmdline variables opt* and opts* were groupped within opt and optss class instances. Also names of the options were changed to match 'dest' of the options. Use tools/refactor.py to quickly fix your custom code.
  - Change all references to PyMVPA website to www.pymvpa.org.
  - Make website stylesheet compatible with sphinx 0.4.
  - Several minor improvements of the compatibilty with MacOS.
  - Extended FAQ section of the manual.
  - Bugfix: doubleGammaHRF() ignoring K2 argument.

- 0.2.2 (Tue, 17 Jun 2008)

  - Extended build instructions: Added section on OpenSUSE.
  - Replaced ugly PYMVPA_LIBSVM environment variable to trigger compiling the LIBSVM wrapper with a proper '–with-libsvm' switch in setup.py. Additionally, setup.py now detects if included LIBSVM has been built and enables LIBSVM wrapper automatically in this case.
  - Added proper Makefiles for LIBSVM copy, with configurable compiler flags.
  - Added 'setup.cfg' to remove the need to manually specify swig-opts (Windows specific configuration is in 'setup.cfg.win').

- 0.2.1 (Sun, 15 Jun 2008)

  - Several improvements to make building PyMVPA on Windows systems easy (e.g. added dedicated Makefile.win to build a binary installer).
  - Improved and extended documentation for building and installing PyMVPA.
  - Include a minimal copy of the required (patched) LIBSVM library (currently version 2.85.0) for convenience. This copy is automatically compiled and used for the LIBSVM wrapper when PyMVPA built using the *Make* approach.

- 0.2.0 (Wed, 29 May 2008)

  - New Splitter class (HalfSplitter) to split into first and second half.
  - New Splitter class (CustomSplitter) to allow for splits with an arbitrary number of datasets per split and the ability to specify the association of samples with any of those datasets (not just the validation set).
  - New sparse multinomial logistic regression (SMLR) classifier and associated sensitivity analyzer.
  - New least angle regression classifier (LARS).
  - New gaussian process regression classifier (GPR).

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

- 0.1.0 (Wed, 20 Feb 2008)
  - First public release.

CHAPTER

SIXTEEN

MODULE REFERENCE

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

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

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

16.1 Global Facilities

16.1.1 mvpa

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

group Unittests
: tests

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

requires
: Python 2.4+

version
: 0.4.1

see
: The PyMVPA webpage

see
: GIT Repository Browser

license
: The MIT License <http://www.opensource.org/licenses/mit-license.php>

copyright
: © 2006-2009 Michael Hanke <michael.hanke@gmail.com>

copyright
: © 2007-2009 Yaroslav O. Halchenko <debian@onerussian.com>

newfield contributor
: Contributor, Contributors (Alphabetical Order)

contributor
: Emanuele Olivetti

contributor
: Per B. Sederberg

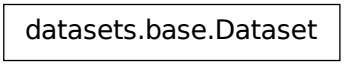
seed(*random_seed*)
: Uniform and combined seeding of all relevant random number generators.

16.2 Datasets: Input, Output, Storage and Preprocessing

16.2.1 datasets.base

Module: `datasets.base`

Inheritance diagram for `mvpa.datasets.base`:



Dataset container

`Dataset`

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

The Dataset.

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

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

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

Initialize dataset instance

There are basically two different way to create a dataset:

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

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

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

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

   labels
   An array or scalar value defining labels for each samples

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

   chunks
   An array or scalar value defining chunks for each sample

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

C
chunks

I
origids

L
labels

S
samples

UC
attrib

UL
attrib

aggregateFeatures (*dataset, fx=<function mean at 0x8768a04>*)
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.

- *featuresmapper* (Mapper) – *Mapper* to somehow transform each sample's features
- *samplesmapper* (Mapper) – *Mapper* to transform each feature across samples
- *train* (bool) – Flag whether to train the mapper with this dataset before applying it.

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

chunks
chunks

coarsenChunks (*source, nchunks=4*)
Change chunking of the dataset

Group chunks into groups to match desired number of chunks. Makes sense if originally there were no strong groupping into chunks or each sample was independent, thus belonged to its own chunk

- *source* (Dataset or list of chunk ids) – dataset or list of chunk ids to operate on. If Dataset, then its chunks get modified
- *nchunks* (int) – desired number of chunks

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

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

Return type
ndarray

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

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

- *mask* (1d ndarray) – Feature mask.

Return type
ndarray

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

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

defineFeatureGroups (*definition*)
Assign *definition* to featuregroups

XXX Feature-groups was not finished to be useful

getLabelsMap ()
Stored labels map (if any)

getNFeatures ()
Number of features per pattern.

getNSamples()
Currently available number of patterns.

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

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

If 'nperlabel' is a list which's length has to match the number of unique label values. In this case 'nperlabel' 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.

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

Array shape is (chunks x labels).

•*dataset* (Dataset) – Source dataset.

idhash
To verify if dataset is in the same state as when smth else was done

Like if classifier was trained on the same dataset as in question

idsbychunks(*x*)
attrib

idsbylabels(*x*)
attrib

idsonboundaries(*prior=0, post=0, attributes_to_track=, ['labels', 'chunks'], affected_labels=None, revert=False*)
Find samples which are on the boundaries of the blocks

Such samples might need to be removed. By default (with prior=0, post=0) ids of the first samples in a 'block' are reported

•*prior* (int) – how many samples prior to transition sample to include

•*post* (int) – how many samples post the transition sample to include

•*attributes_to_track* (list of basestring) – which attributes to track to decide on the boundary condition

•*affected_labels* (list of basestring) – for which labels to perform selection. If None - for all

•*revert* (bool) – either to revert the meaning and provide ids of samples which are found to not to be boundary samples

index(**args, **kwargs*)
Universal indexer to obtain indexes of interesting samples/features. See .select() for more information

Return
tuple of (samples indexes, features indexes). Each item could be also None, if no selection on samples or features was requested (to discriminate between no selected items, and no selections)

labels
labels

labels_map
Stored labels map (if any)

nfeatures
Number of features per pattern.

nsamples
Currently available number of patterns.

origids
origids

permuteLabels (*status, perchunk=True, assure_permute=False*)
Permute the labels.

TODO: rename status into something closer in semantics.

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

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

samples
samples

samplesperchunk
attrib

samplesperlabel
attrib

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

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

Examples:

Mimique plain selectSamples:

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

Mimique plain selectFeatures:

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

Mixed (select features and samples):

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

Select samples matching some attributes:

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

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

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

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

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

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

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

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

Returns a new dataset object containing the selected sample subset.

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

setLabelsMap (*lm*)
Set labels map.

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

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

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

- *uniq* (bool) – Include summary over data attributes which have unique
- *idhash* (bool) – Include idhash value for dataset and samples
- *stats* (bool) – Include some basic statistics (mean, std, var) over dataset samples
- *lstats* (bool) – Include statistics on chunks/labels
- *maxc* (int) – Maximal number of chunks when provide details on labels/chunks
- *maxl* (int) – Maximal number of labels when provide details on labels/chunks

summary_labels (*maxc=30, maxl=20*)
Provide summary statistics over the labels and chunks

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

uniquechunks
attrib

uniquelabels
attrib

where (**args, **kwargs*)
Obtain indexes of interesting samples/features. See select() for more information

XXX somewhat obsoletes idsby...

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*

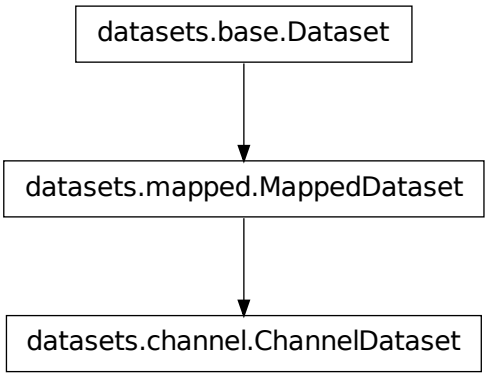
If *perchunk* is True samples within the same chunk are z-scored independent of samples from other chunks, e.i. mean and standard deviation are calculated individually.

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:



Dataset handling data structured in channels.

ChannelDataset

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

Bases: mvpa.datasets.mapped.MappedDataset

Dataset handling data structured into channels.

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

See Also:

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

MappedDataset

Initialize ChannelDataset.

- *samples* (ndarray) – Three-dimensional array: (samples x channels x timepoints).
- *t0* (float) – Reference time of the first timepoint. Can be used to preserve information about the onset of some stimulation. Preferably in seconds.
- *dt* (float) – Temporal distance between two timepoints. Has to be given in seconds. Otherwise *samplingrate* property will not return *Hz*.
- *channelids* (list) – List of channel names.
- *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.

channelids

List of channel IDs

dt

Time difference between two samples (in seconds).

samplingrate
Yeah, sampling rate.

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

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

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

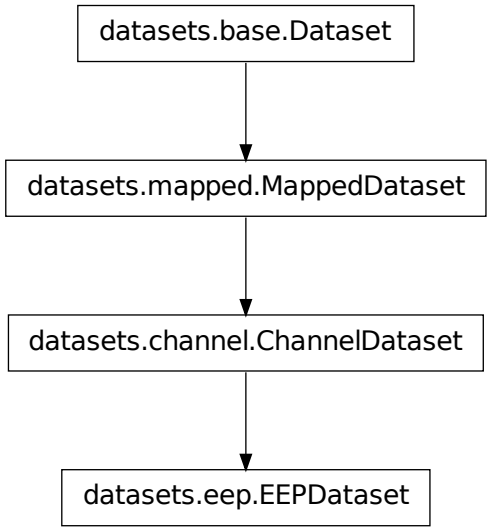
- *t* (int | float | None) – If an integer, *t* denotes the number of timepoints in the from the start of each sample to be used to compute the baseline signal. If a floating point value, *t* is the duration of the baseline window from the start of each sample in whatever unit corresponding to the datasets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

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

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

Dataset using a EEP binary file as source.

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

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

See Also:

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

`ChannelDataset`

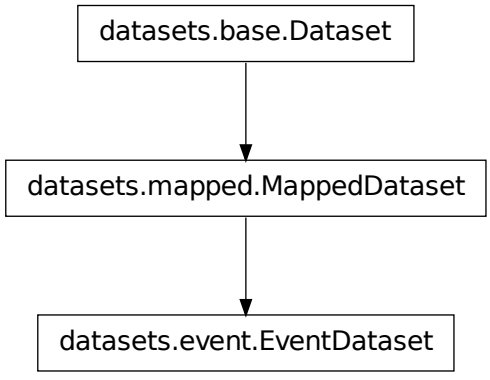
Initialize EEPDataset.

- *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 **`EventDataset`** (*samples=None, events=None, mask=None, bcshape=None, dametric=None, **kwargs*)

Bases: `mvpa.datasets.mapped.MappedDataset`

Event-based dataset

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

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

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

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

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

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

chunk
An optional chunk id.

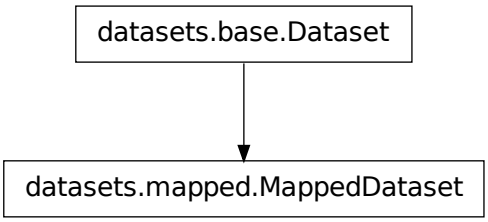
features
A list with an arbitrary number of features values (floats), that will be added to the feature vector of the corresponding sample.

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

16.2.5 datasets.mapped

Module: `datasets.mapped`

Inheritance diagram for `mvpa.datasets.mapped`:



Mapped dataset

MappedDataset

class MappedDataset(*samples=None, mapper=None, dsattr=None, **kwargs*)

Bases: mvpa.datasets.base.Dataset

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

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

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

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

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.

- *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.

O

Return samples in the original shape

mapForward(*data*)

Map data from the original dataspace into featurespace.

mapReverse(*data*)

Reverse map data from featurespace into the original dataspace.

mapSelfReverse()

Reverse samples from featurespace into the original dataspace.

mapper

mapper

samples_original

Return samples in the original shape

selectFeatures(*ids, plain=False, sort=False*)

Select features given their ids.

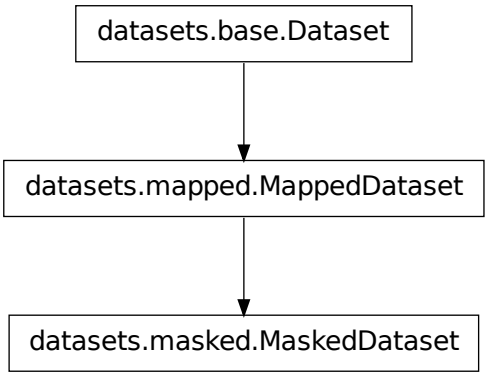
The methods behaves similar to Dataset.selectFeatures(), but additionally takes care of adjusting the embedded mapper appropriately.

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

16.2.6 datasets.masked

Module: datasets.masked

Inheritance diagram for mvpa.datasets.masked:



Dataset with applied mask

MaskedDataset

class MaskedDataset(*samples=None, mask=None, **kwargs*)

Bases: mvpa.datasets.mapped.MappedDataset

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

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

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

selectFeaturesByMask(*mask, plain=False*)

Use a mask array to select features from the current set.

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

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

16.2.7 datasets.meta

Module: datasets.meta

Inheritance diagram for mvpa.datasets.meta:

Dataset container

MetaDataset

class MetaDataset(*datasets*)

Bases: `object`

Dataset container

The class is useful to combine several Datasets with different origin and type and bind them together. Such a combined dataset can then by used to e.g. pass it to a classifier.

MetaDataset does not permanently duplicate data stored in the dataset it contains. The combined samples matrix is build on demand and samples attribute access is redirected to the first dataset in the container.

Currently operations other than samples or feature selection are not fully supported, e.g. passing a 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

- *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

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.

Functions

aggregateFeatures (*dataset, fx=<function mean at 0x8768a04>*)
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).

coarsenChunks (*source, nchunks=4*)
Change chunking of the dataset

Group chunks into groups to match desired number of chunks. Makes sense if originally there were no strong groupping into chunks or each sample was independent, thus belonged to its own chunk

- *source* (Dataset or list of chunk ids) – dataset or list of chunk ids to operate on. If Dataset, then its chunks get modified
- *nchunks* (int) – desired number of chunks

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

Array shape is (chunks x labels).

- *dataset* (Dataset) – Source dataset.

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

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

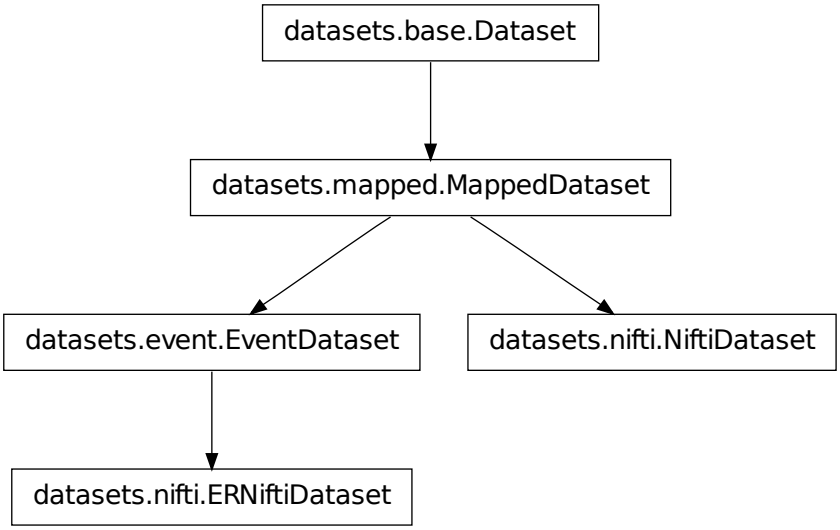
detrend(*dataset, perchunk=False, model='linear', polyord=None, opt_reg=None*)
Given a dataset, detrend the data inplace either entirely or per each chunk

- *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 differnt 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 ERNiftiDataset(*samples=None, events=None, mask=None, evconv=False, storeoffset=False, tr=None, enforce_dim=4, **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 descrete 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.

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 temproal offset information when converting Events into descrete 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.

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.

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

- *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 NiftiDataset (*samples=None, mask=None, dsattr=None, enforce_dim=4, **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.

NIfTI files are accessed via PyNIfTI. See http://niftilib.sourceforge.net/pynifti/ for more information about pynifti.

- *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.

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.

- *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

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 sufficent version is available.

getNiftiFromAnySource(*src, ensure=False, enforce_dim=None*)
Load/access NIfTI data from files or instances.

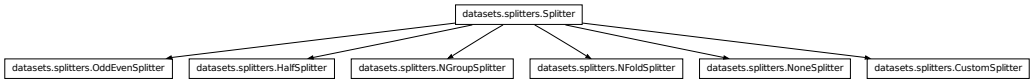
- *src* (str | NiftiImage) – Filename of a NIfTI image or a *NiftiImage* instance.
- *ensure* (bool) – If True, through ValueError exception if cannot be loaded.
- *enforce_dim* (int or None) – If not None, it is the dimensionality of the data to be enforced, commonly 4D for the data, and 3D for the mask in case of fMRI.

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 varous 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

class CustomSplitter(*splitrule, **kwargs*)

Bases: mvpa.datasets.splitters.Splitter

Split a dataset using an arbitrary custom rule.

The splitter is configured by passing a custom spitting rule (*splitrule*) to its constructor. Such a rule is basically a sequence of split definitions. Every single element in this sequence results in excatly one split generated by the Splitter. Each element is another sequence for sequences of sample ids for each dataset that shall be generated in the split.

Example:

- Generate two splits. In the first split the *second* dataset contains all samples with sample attributes corresponding to either 0, 1 or 2. The *first* dataset of the first split contains all samples which are not split into the second dataset.

  The second split yields three datasets. The first with all samples corresponding to sample attributes 1 and 2, the second dataset contains only samples with attrbiute 3 and the last dataset contains the samples with attribute 5 and 6.

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

See Also:

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

Splitter

Cheap init.

- *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.

HalfSplitter

class 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.

- *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.

NFoldSplitter

class 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.

cvtype: Int
Type of cross-validation: N-(cvtype)

kwargs
Additional parameters are passed to the *Splitter* base class.

- *cvtype* (Int) – Type of cross-validation: N-(cvtype)
- *kwargs* – Additional parameters are passed to the *Splitter* base class.
- *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.

NGroupSplitter

class 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.

ngroups: Int
Number of groups to split the attribute into.

kwargs
Additional parameters are passed to the *Splitter* base class.

- *ngroups* (Int) – Number of groups to split the attribute into.
- *kwargs* – Additional parameters are passed to the *Splitter* base class.
- *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.

NoneSplitter

class 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

- *mode* – Either 'first' or 'second' (default) – which output dataset would actually contain the 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 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.

OddEvenSplitter

class 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.

- *usevalues* (Boolean) – If True the values of the attribute used for splitting will be used to determine odd and even samples. If False odd and even chunks are defined by the order of attribute values, i.e. first unique attribute is odd, second is even, despite the corresponding values might indicate the opposite (e.g. in case of [2,3].
- *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.

Splitter

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

Bases: `object`

Base class of dataset splitters.

Each splitter should be initialized with all its necessary parameters. The final splitting is done running the splitter object on a certain Dataset via __call__(). This method has to be implemented like a generator, i.e. it has to return every possible split with a yield() call.

Each split has to be returned as a sequence of Datasets. The properties of the splitted dataset may vary between implementations. It is possible to declare a sequence element as 'None'.

Please note, that even if there is only one Dataset returned it has to be an element in a sequence and not just the Dataset object!

Initialize splitter base.

- *nperlabel* (int or str (or list of them) 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.

setNPerLabel (*value*)

Set the number of samples per label in the split datasets.

'equal' sets sample size to highest possible number of samples that can be provided by each class. 'all' uses all available samples (default).

splitDataset (*dataset, specs*)
Split a dataset by separating the samples where the configured sample attribute matches an element of *specs*.

•*dataset* (Dataset) – This is this source dataset.
•*specs* (sequence of sequences) – Contains ids of a sample attribute that shall be split into the another dataset.

Returns
Tuple of splitted datasets.

splitcfg (*dataset*)
Return splitcfg for a given dataset

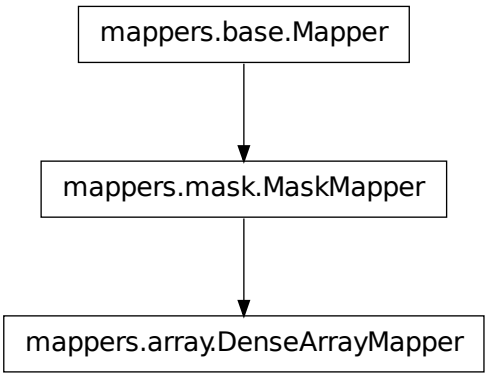
strategy

16.3 Mappers: Data Transformations

16.3.1 mappers.array

Module: mappers.array

Inheritance diagram for mvpa.mappers.array:



Data mapper

DenseArrayMapper

class DenseArrayMapper (*mask=None, metric=None, distance_function=<function cartesianDistance at 0x8c46f0c>, elementsize=None, shape=None, **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

•*mask* (array) – an array in the original dataspace and its nonzero elements are used to define the features included in the dataset. alternatively, the *shape* argument can be used to define the array dimensions.

- *metric* (Metric) – Corresponding metric for the space. No attempt is made to determine whether a certain metric is reasonable for this mapper. If *metric* is None – *DescreteMetric* is constructed that assumes an equal (1) spacing of all mask elements with a *distance_function* given as a parameter listed below.
- *distance_function* (functor) – Distance function to use as the parameter to *DescreteMetric* if *metric* is not specified,
- *elementsize* (list or scalar) – Determines spacing within *DescreteMetric*. If it is given as a scalar, corresponding value is assigned to all dimensions, which are found within *mask*
- *shape* (tuple) – The shape of the array to be mapped. If *shape* is provided instead of *mask*, a full mask (all True) of the desired shape is constructed. If *shape* is specified in addition to *mask*, the provided mask is extended to have the same number of dimensions.

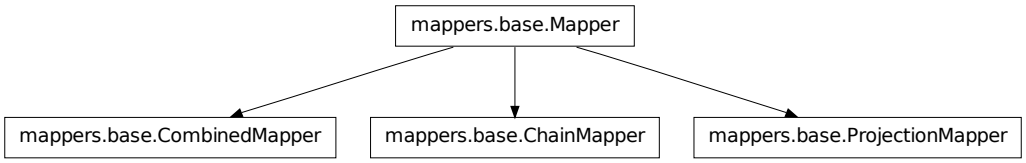
Note

parameters *elementsize* and *distance_function* are relevant only if *metric* is None

16.3.2 mappers.base

Module: `mappers.base`

Inheritance diagram for `mvpa.mappers.base`:



Data mapper

Classes

`ChainMapper`

class `ChainMapper`(*mappers, **kwargs*)

Bases: `mvpa.mappers.base.Mapper`

Meta mapper that embedded a chain of other mappers.

Each mapper in the chain is called successively to perform forward or reverse mapping.

Note: In its current implementation the *ChainMapper* treats all but the last mapper as simple pre-processing (in forward()) or post-processing (in reverse()) steps. All other capabilities, e.g. training and neighbor metrics are provided by or affect *only the last mapper in the chain*.

With respect to neighbor metrics this means that they are determined based on the input space of the *last mapper* in the chain and *not* on the input dataspace of the *ChainMapper* as a whole

- *mappers* (list of Mapper instances) –
- ***kwargs* – All additional arguments are passed to the base-class constructor.

`forward`(*data*)

Calls all mappers in the chain successively.

- *data* – data to be chain-mapped.

getInSize()
Returns the size of the entity in input space

getNeighbor(*outId, *args, **kwargs*)
Get the ids of the neighbors of a single feature in output dataspace.

Note: The neighbors are determined based on the input space of the *last mapper* in the chain and *not* on the input dataspace of the *ChainMapper* as a whole!

- *outId* (int) – Single id of a feature in output space, whos neighbors should be determined.
- **args, **kwargs* – Additional arguments are passed to the metric of the embedded mapper, that is responsible for the corresponding feature.

Returns a list of outIds

getOutSize()
Returns the size of the entity in output space

reverse(*data*)
Calls all mappers in the chain successively, in reversed order.

- *data* (array) – data array to be reverse mapped into the orginal dataspace.

selectOut(*outIds*)
Remove some elements from the *last* mapper in the chain.

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

train(*dataset*)
Trains the *last* mapper in the chain.

- *dataset* (`Dataset` or subclass) – A dataset with the number of features matching the *outSize* of the last mapper in the chain (which is identical to the one of the *ChainMapper* itself).

CombinedMapper

class CombinedMapper(*mappers, **kwargs*)
Bases: `mvpa.mappers.base.Mapper`

Meta mapper that combines several embedded mappers.

This mapper can be used the map from several input dataspaces into a common output dataspace. When `forward()` is called with a sequence of data, each element in that sequence is passed to the corresponding mapper, which in turned forward-maps the data. The output of all mappers is finally stacked (horizontally or column or feature-wise) into a single large 2D matrix (nsamples x nfeatures).

Note: This mapper can only embbed mappers that transform data into a 2D (nsamples x nfeatures) representation. For mappers not supporting this transformation, consider wrapping them in a `ChainMapper` with an appropriate post-processing mapper.

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

- *mappers* (list of Mapper instances) – The order of the mappers in the list is important, as it will define the order in which data snippets have to be passed to `forward()`.
- ***kwargs* – All additional arguments are passed to the base-class constructor.

forward(*data*)
Map data from the IN spaces into to common OUT space.

- *data* (sequence) – Each element in the *data* sequence is passed to the corresponding embedded mapper and is mapped individually by it. The number of elements in *data* has to match the number of embedded mappers. Each element is *data* has to provide the same number of samples (first dimension).

Return type
array

Returns
Horizontally stacked array of all embedded mapper outputs.

getInSize()
Returns the size of the entity in input space

getNeighbor(*outId, *args, **kwargs*)
Get the ids of the neighbors of a single feature in output dataspace.

- *outId* (int) – Single id of a feature in output space, whos neighbors should be determined.
- **args, **kwargs* – Additional arguments are passed to the metric of the embedded mapper, that is responsible for the corresponding feature.

Returns a list of outIds

getOutSize()
Returns the size of the entity in output space

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

- *data* (array) – Single data array to be reverse mapped into a sequence of data snippets in their individual IN spaces.

Return type
list

selectOut(*outIds*)
Remove some elements and leave only ids in 'out'/feature space.

Note: The subset selection is done inplace

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

train(*dataset*)
Trains all embedded mappers.

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

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

Mapper

class Mapper(*metric=None*)
Bases: `object`

Interface to provide mapping between two spaces: IN and OUT. Methods are prefixed correspondingly. forward/reverse operate on the entire dataset. get(In|Out)Id[s] operate per element:

```
     forward
    --------->
IN              OUT
    <---------/
      reverse
```

- *metric* (Metric) – Optional metric

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

getInId(*outId*)
Translate a feature id into a coordinate/index in input space.

Such a translation might not be meaningful or even possible for a particular mapping algorithm and therefore cannot be relied upon.

getInSize()
Returns the size of the entity in input space

getMetric()
To make pylint happy

getNeighbor(*outId, *args, **kwargs*)
Get feature neighbors in input space, given an id in output space.

This method has to be reimplemented whenever a derived class does not provide an implementation for `getInId()`.

getNeighborIn(*inId, *args, **kwargs*)
Return the list of coordinates for the neighbors.

- *inId* – id (index) of an element in input dataspace.
- **args, **kwargs* – Any additional arguments are passed to the embedded metric of the mapper.

XXX See TODO below: what to return – list of arrays or list of tuples?

getNeighbors(*outId, *args, **kwargs*)
Return the list of coordinates for the neighbors.

By default it simply constructs the list based on the generator returned by getNeighbor()

getOutSize()
Returns the size of the entity in output space

isValidInId(*inId*)
Validate id in IN space.

Override if IN space is not simly a 1D vector

isValidOutId(*outId*)
Validate feature id in OUT space.

Override if OUT space is not simly a 1D vector

metric
To make pylint happy

nfeatures
str(object) -> string

Return a nice string representation of the object. If the argument is a string, the return value is the same object.

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

selectOut(*outIds*)
Limit the OUT space to a certain set of features.

- *outIds* (sequence) – Subset of ids of the current feature in OUT space to keep.

setMetric(*metric*)
To make pylint happy

train(*dataset*)
Perform training of the mapper.

This method is called to put the mapper in a state that allows it to perform to intended mapping.

- *dataset* (Dataset or subclass) –

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

ProjectionMapper

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

Mapper using a projection matrix to transform the data.

This class cannot be used directly. Sub-classes have to implement the *_train()* method, which has to compute the projection matrix given a dataset (see *_train()* docstring for more information).

Once the projection matrix is available, this class provides functionality to perform forward and backwards mapping of data, the latter using the hermitian (conjugate) transpose of the projection matrix. Additionally, *ProjectionMapper* supports optional (but done by default) demeaning of the data and selection of arbitrary component (i.e. columns of the projection matrix) of the projection.

Forward and back-projection matrices (a.k.a. *projection* and *reconstruction*) are available via the *proj* and *recon* properties. the latter only after it has been computed (after first call to *reverse*).

See Also:

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

`Mapper`

Initialize the ProjectionMapper

- *selector* (None | list) – Which components (i.e. columns of the projection matrix) should be used for mapping. If *selector* is *None* all components are used. If a list is provided, all list elements are treated as component ids and the respective components are selected (all others are discarded).
- *demean* (bool) – Either data should be demeaned while computing projections and applied back while doing reverse()

forward(*data, demean=None*)
Perform forward projection.

- *data* (ndarray) – Data array to map
- *demean* (boolean | None) – Override demean setting for this method call.

Return type
NumPy array

getInSize()
Returns the number of original features.

getOutSize()
Returns the number of components to project on.

proj
Projection matrix

recon
Backprojection matrix

reverse(*data*)
Reproject (reconstruct) data into the original feature space.

Return type
NumPy array

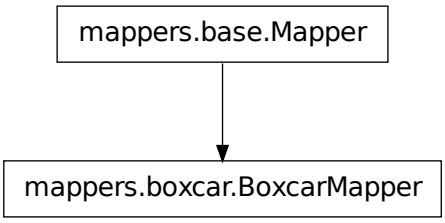
selectOut(*outIds*)
Choose a subset of components (and remove all others).

train(*dataset*)
Determine the projection matrix.

16.3.3 mappers.boxcar

Module: `mappers.boxcar`

Inheritance diagram for `mvpa.mappers.boxcar`:



Data mapper

BoxcarMapper

class BoxcarMapper(*startpoints, boxlength, offset=0, collision_resolution='mean'*)

Bases: mvpa.mappers.base.Mapper

Mapper to combine multiple samples into a single sample.

Note: This mapper is somewhat unconventional since it doesn't preserve number of samples (ie the size of 0-th dimension).

See Also:

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

Mapper

- *startpoints* (sequence) – Index values along the first axis of 'data'.
- *boxlength* (int) – The number of elements after 'startpoint' along the first axis of 'data' to be considered for the boxcar.
- *offset* (int) – The offset between the provided starting point and the actual start of the boxcar.
- *collision_resolution* ('mean') – if a sample belonged to multiple output samples, then on reverse, how to resolve the value

forward(*data*)

Project an ND matrix into N+1D matrix

This method also handles the special of forward mapping a single 'raw' sample. Such a sample is extended (by concatenating clones of itself) to cover a full boxcar. This functionality is only availably after a full data array has been forward mapped once.

Return type
array

getInSize()

Returns the number of original samples which were combined.

getOutSize()

Returns the number of output samples.

isValidInId(*inId*)

Validate if InId is valid

isValidOutId(*outId*)

Validate if OutId is valid

reverse(*data*)

Uncombine features back into original space.

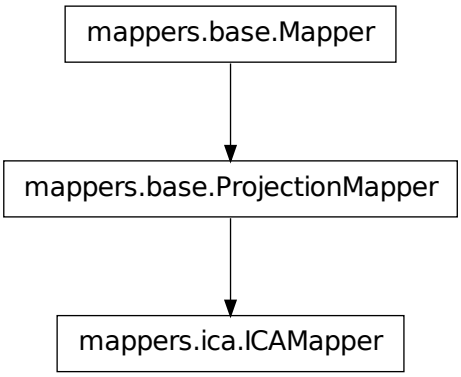
Samples which were not touched by forward will get value 0 assigned

selectOut (*outIds*)
Just complain for now

16.3.4 mappers.ica

Module: mappers.ica

Inheritance diagram for mvpa.mappers.ica:



Data mapper

ICAMapper

class 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

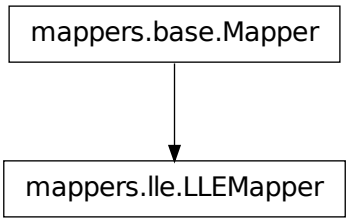
Initialize instance of ICAMapper

- *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 LLEMapper(*k, algorithm='lle', **kwargs*)

Bases: mvpa.mappers.base.Mapper

Locally linear embbeding 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

- *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.

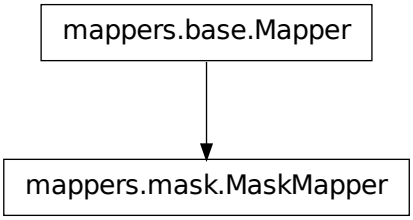
reverse (*data*)
Reverse map data from OUT space into the IN space.

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 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

- *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 *ouIds* selected.

This method works exactly like Mapper.convertOutIds2OutMask(), but the feature mask is finally (reverse) mapped into in-space.

- *outIds* (list or 1d array) – To be selected features ids in out-space.

Return type
ndarray

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

convertOutIds2OutMask (*outIds*)
Returns a boolean mask with all features in *outIds* selected.

- *outIds* (list or 1d array) – To be selected features ids in out-space.

Return type
ndarray

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

discardOut(*outIds*)
Listed outIds would be discarded

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

getInId(*outId*)
Returns a features coordinate in the original data space for a given feature id.

If this method is called with a list of feature ids it returns a 2d-array where the first axis corresponds the dimensions in 'In' dataspace and along the second axis are the coordinates of the features on this dimension (like the output of NumPy.array.nonzero()).

XXX it might become __get_item__ access method

getInIds()
Returns a 2d array where each row contains the coordinate of the feature with the corresponding id.

getInSize()
InShape is a shape of original mask

getMask(*copy=True*)
By default returns a copy of the current mask.

If 'copy' is set to False a reference to the mask is returned instead. This shared mask must not be modified!

getOutId(*coord*)
Translate a feature mask coordinate into a feature ID.

getOutSize()
OutSize is a number of non-0 elements in the mask

isValidInId(*inId*)

mask

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

selectOut(*outIds*)
Only listed outIds would remain.

Function assumes that outIds are sorted. In __debug__ mode selectOut would check if obtained IDs are sorted and would warn the user if they are not.

Note: If you feel strongly that you need to remap features internally (ie to allow Ids with mixed order) please contact developers of mvpa to discuss your use case.

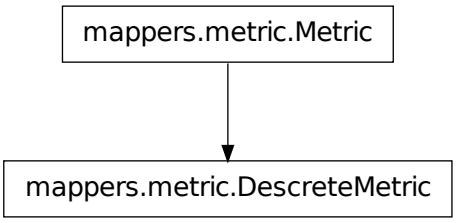
The function used to accept a matrix-mask as the input but now it really has to be a list of IDs

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

16.3.7 mappers.metric

Module: mappers.metric

Inheritance diagram for mvpa.mappers.metric:



Classes and functions to provide sense of distances between sample points

Classes

DescreteMetric

class **DescreteMetric**(*elementsize=1, distance_function=<function cartesianDistance at 0x8c46f0c>, compatmask=None*)

Bases: mvpa.mappers.metric.Metric

Find neighboring points in descretized space

If input space is descretized 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 cordinates 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.

- *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 compatiable spacemetrics. If None (default) all dimensions are assumed to have compatible spacemetrics.

elementsize

filter_coord

Lets allow to specify some custom filter to use

getNeighbors(*origin, radius=0*)

Returns coordinates of the neighbors which are within distance from coord.

- *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

class 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 overriden in every derived class. NOTE: derived #2 from derived class #1 has to override all methods which were overrident in class #1

getNeighbor(**args, **kwargs*)
Generator to return coordinate of the neighbor.

Base class contains the simplest implementation, assuming that getNeighbors returns iterative structure to spit out neighbors 1-by-1

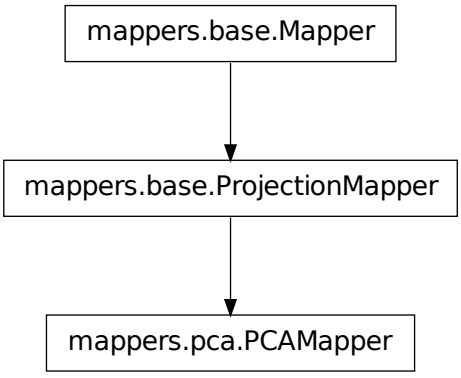
getNeighbors(**args, **kwargs*)
Return the list of coordinates for the neighbors.

By default it simply constracts the list based on the generator getNeighbor

16.3.8 mappers.pca

Module: mappers.pca

Inheritance diagram for mvpa.mappers.pca:



Data mapper

PCAMapper

class 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

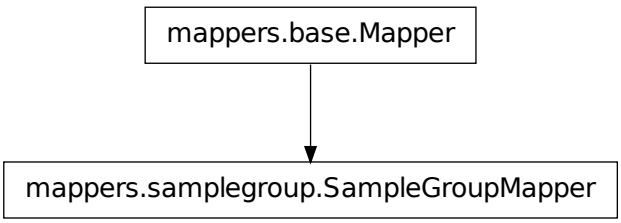
- *selector* (None | list) – Which components (i.e. columns of the projection matrix) should be used for mapping. If *selector* is *None* all components are used. If a list is provided, all list elements are treated as component ids and the respective components are selected (all others are discarded).
- *demean* (bool) – Either data should be demeaned while computing projections and applied back while doing reverse()

var
Variances per component

16.3.9 mappers.samplegroup

Module: `mappers.samplegroup`

Inheritance diagram for `mvpa.mappers.samplegroup`:



Data mapper

`SampleGroupMapper`

class `SampleGroupMapper` (*fx=<function FirstAxisMean at 0x8f6f80c>*)
Bases: `mvpa.mappers.base.Mapper`

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

A customimzable function is applied individually to all samples with the same unique label from the same chunk. This mapper is somewhat unconventional since it doesn't preserve number of samples (ie the size of 0-th dimension...)

See Also:

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

`Mapper`

Initialize the PCAMapper

startpoints: A sequence of index value along the first axis of
'data'.

boxlength: The number of elements after 'startpoint' along the first axis of 'data' to be considered for averaging.

offset: The offset between the starting point and the averaging window (boxcar).

collision_resolution
[string] if a sample belonged to multiple output samples, then on reverse, how to resolve the value (choices: 'mean')

forward(*data*)

getInSize()
Returns the number of original samples which were combined.

getOutSize()
Returns the number of output samples.

reverse(*data*)
This is not implemented.

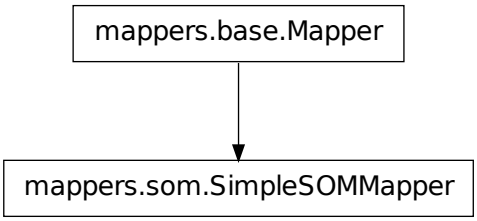
selectOut(*outIds*)
Just complain for now

train(*dataset*)

16.3.10 mappers.som

Module: mappers.som

Inheritance diagram for mvpa.mappers.som:



Self-organizing map (SOM) mapper.

SimpleSOMMapper

class SimpleSOMMapper(*kshape, niter, learning_rate=0.005000000000000001, 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.

- *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.
- *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.

K
Provide access to the Kohonen layer.
With some care.

forward(*data*)
Map data from the IN dataspace into OUT space.
Mapping is performs by simple determining the best matching Kohonen unit for each data sample.

getInId(*outId*)
Translate a feature id into a coordinate/index in input space.
This is not meaningful in the context of SOMs.

getInSize()
Returns the size of the entity in input space

getOutSize()
Returns the size of the entity in output space

isValidOutId(*outId*)
Validate feature id in OUT space.

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

selectOut(*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.

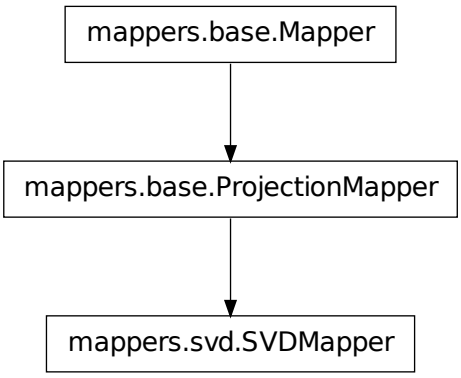
train(*ds*)
Perform network training.

- *ds* (Dataset) – All samples in the dataset will be used for unsupervised training of the SOM.

16.3.11 mappers.svd

Module: mappers.svd

Inheritance diagram for mvpa.mappers.svd:



Singular-value decomposition mapper

SVDMapper

class 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

- *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).

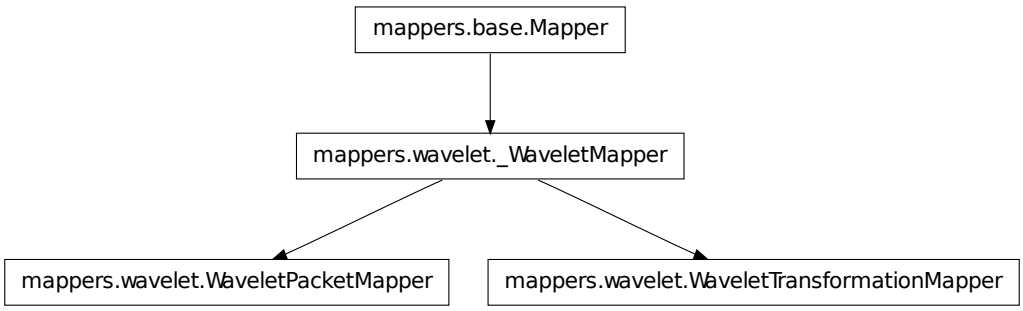
sv

Singular values

16.3.12 mappers.wavelet

Module: mappers.wavelet

Inheritance diagram for mvpa.mappers.wavelet:



Wavelet mappers

Classes

WaveletPacketMapper

class WaveletPacketMapper (*level=None, **kwargs*)

Bases: mvpa.mappers.wavelet._WaveletMapper

Convert signal into an overcomplete representaion using Wavelet packet

Initialize WaveletPacketMapper mapper

- *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 WaveletTransformationMapper (*dim=1, wavelet='sym4', mode='per', maxlevel=None*)

Bases: mvpa.mappers.wavelet._WaveletMapper

Convert signal into wavelet representaion

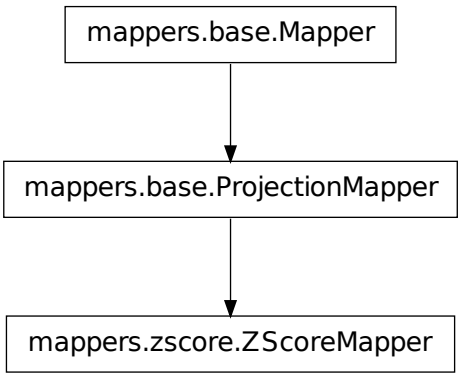
Initialize _WaveletMapper mapper

- *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 withing pywt package
- *mode* (basestring) – periodization mode
- *maxlevel* (int or None) – number of levels to use. If None - automatically selected by pywt

16.3.13 mappers.zscore

Module: mappers.zscore

Inheritance diagram for mvpa.mappers.zscore:



Simple mapper to perform zscoring

ZScoreMapper

class 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

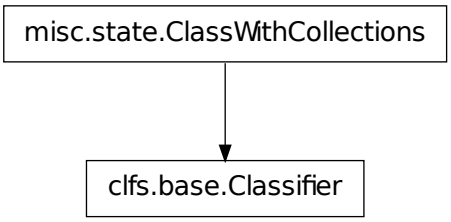
- *baselinelabels* (None or list of int or string) – Used to compute mean and variance TODO: not in effect now and need to be adherent to all 'ProjectionMapper's
- *selector* (None | list) – Which components (i.e. columns of the projection matrix) should be used for mapping. If *selector* is *None* all 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 vise verse), so the same base class *Classifier* is utilized for both kinds.

`Classifier`

class `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
- *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.

- *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.

- *data* – data which is conventionally given to predict
- *kwargs* – that is what _changedData gets updated with. So, smth like `(params=['C'], labels=True)` if parameter C and labels got changed

retrain(*dataset, **kwargs*)
Helper to avoid check if data was changed actually changed

Useful if just some aspects of classifier were changed since its previous training. For instance if dataset wasn't changed but only classifier parameters, then kernel matrix does not have to be computed.

Words of caution: classifier must be previously trained, results always should first be compared to the results on not 'retrainable' classifier (without calling retrain). Some additional checks are enabled if debug id 'CHECK_RETRAIN' is enabled, to guard against obvious mistakes.

- *kwargs* – that is what _changedData gets updated with. So, smth like `(params=['C'], labels=True)` if parameter C and labels got changed

summary()
Providing summary over the classifier

train(*dataset*)
Train classifier on a dataset

Shouldn't be overridden in subclasses unless explicitly needed to do so

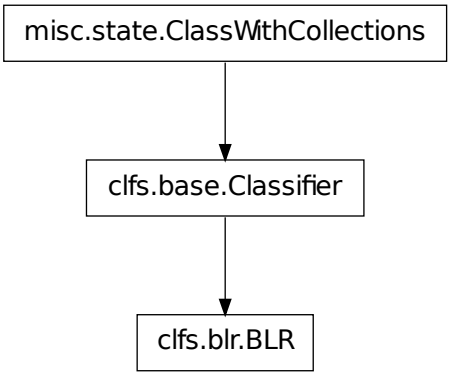
trained
Either classifier was already trained

untrain()
Reset trained state

16.4.2 clfs.blr

Module: clfs.blr

Inheritance diagram for mvpa.clfs.blr:



Bayesian Linear Regression (BLR).

BLR

class 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
- *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.

- *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

absminDistance(*a, b*)
Returns dinstance 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"

cartesianDistance(*a, b*)
Return Cartesian distance between a and b

mahalanobisDistance(*x, y=None, w=None*)
Calculate Mahalanobis distance of the pairs of points.

- *x* – first list of points. Rows are samples, columns are features.
- *y* – second list of points (optional)
- *w* (N.ndarray) – optional inverse covariance matrix between the points. It is computed if not given

Inverse covariance matrix can be calculated with the following

```
w = N.linalg.solve(N.cov(x.T), N.identity(x.shape[1]))
```

or

```
w = N.linalg.inv(N.cov(x.T))
```

manhattenDistance(*a, b*)
Return Manhatten distance between a and b

oneMinusCorrelation(*X, Y*)
Return one minus the correlation matrix between the rows of two matrices.

This functions computes a matrix of correlations between all pairs of rows of two matrices. Unlike NumPy's corrcoef() this function will only considers 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.

- *X* (2D-array) –
- *Y* (2D-array) –

Example:

```
>>> X = N.random.rand(20,80)
>>> Y = N.random.rand(5,80)
>>> C = oneMinusCorrelation(X, Y)
>>> print C.shape
(20, 5)
```

pnorm_w_python(*data1, data2=None, weight=None, p=2, heuristic='auto', use_sq_euclidean=True*)
Weighted p-norm between two datasets (pure Python implementation)

||x - x'||_w = (sum_{i=1...N} (w_i*|x_i - x'_i|)**p)**(1/p)

- *data1* (N.ndarray) – First dataset
- *data2* (N.ndarray or None) – Optional second dataset
- *weight* (N.ndarray or None) – Optional weights per 2nd dimension (features)
- *p* – Power
- *heuristic* (basestring) – Which heuristic to use: * 'samples' – python sweep over 0th dim * 'features' – python sweep over 1st dim * 'auto' decides automatically. If # of features (shape[1]) is much larger than # of samples (shape[0]) – use 'samples', and use 'features' otherwise
- *use_sq_euclidean* (bool) – Either to use squared_euclidean_distance_matrix for computation if p==2

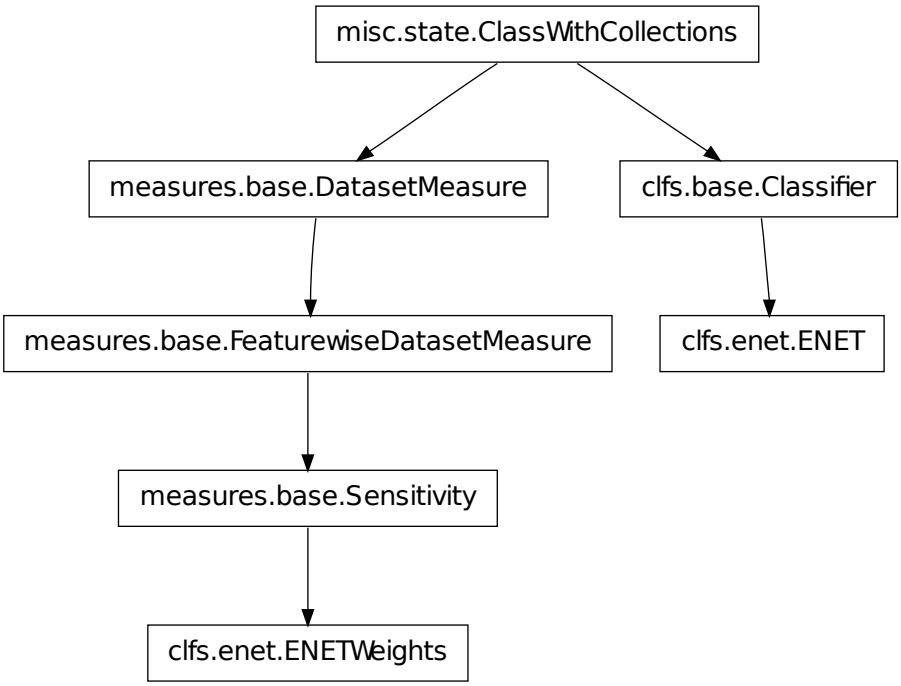
squared_euclidean_distance(*data1, data2=None, weight=None*)
Compute weighted euclidean distance matrix between two datasets.

- *data1* (N.ndarray) – first dataset
- *data2* (N.ndarray) – second dataset. If None, compute the euclidean distance between the first dataset versus itself. (Defaults to None)
- *weight* (N.ndarray) – vector of weights, each one associated to each dimension of the dataset (Defaults to None)

16.4.4 clfs.enet

Module: `clfs.enet`

Inheritance diagram for `mvpa.clfs.enet`:



Elastic-Net (ENET) regression classifier.

Classes

ENET

class 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:

Zou and Hastie (2005) 'Regularization and Variable Selection via the Elastic Net' Journal of the Royal Statistical Society, Series B, 67, 301-320.

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()

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

•*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:

•*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 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_result*: 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`

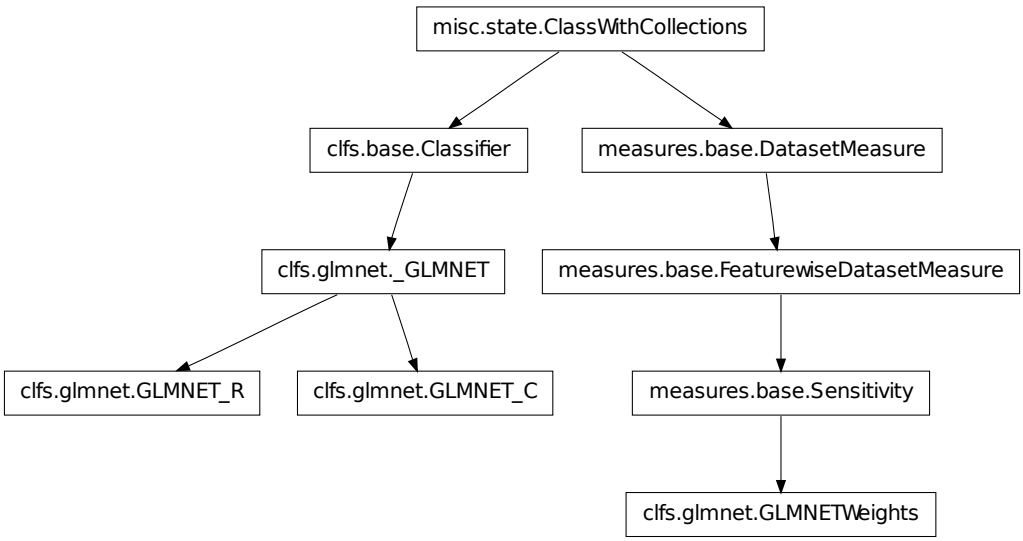
Initialize the analyzer with the classifier it shall use.

- *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.

16.4.5 clfs.glmnet

Module: `clfs.glmnet`

Inheritance diagram for `mvpa.clfs.glmnet`:



GLM-Net (GLMNET) regression classifier.

Classes

`GLMNETWeights`

class `GLMNETWeights`(*clf*, *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_result*: 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.

- *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.

GLMNET_C

class GLMNET_C (***kwargs*)

Bases: `mvpa.clfs.glmnet._GLMNET`

GLM-NET Multinomial Classifier.

This is the GLM-NET algorithm from

Friedman, J., Hastie, T. and Tibshirani, R. (2008) Regularization Paths for Generalized Linear Models via Coordinate Descent. http://www-stat.stanford.edu/~hastie/Papers/glmnet.pdf

parameterized to be a 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
- *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:

`_GLMNET`

Initialize GLM-Net multinomial classifier.

See the help in R for further details on the parameters

- *family* – Response type of your labels (either regression or 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

class GLMNET_R (***kwargs*)

Bases: `mvpa.clfs.glmnet._GLMNET`

GLM-NET Gaussian Regression Classifier.

This is the GLM-NET algorithm from

Friedman, J., Hastie, T. and Tibshirani, R. (2008) Regularization Paths for Generalized Linear Models via Coordinate Descent. http://www-stat.stanford.edu/~hastie/Papers/glmnet.pdf

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
- *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:

`_GLMNET`

Initialize GLM-Net.

See the help in R for further details on the parameters

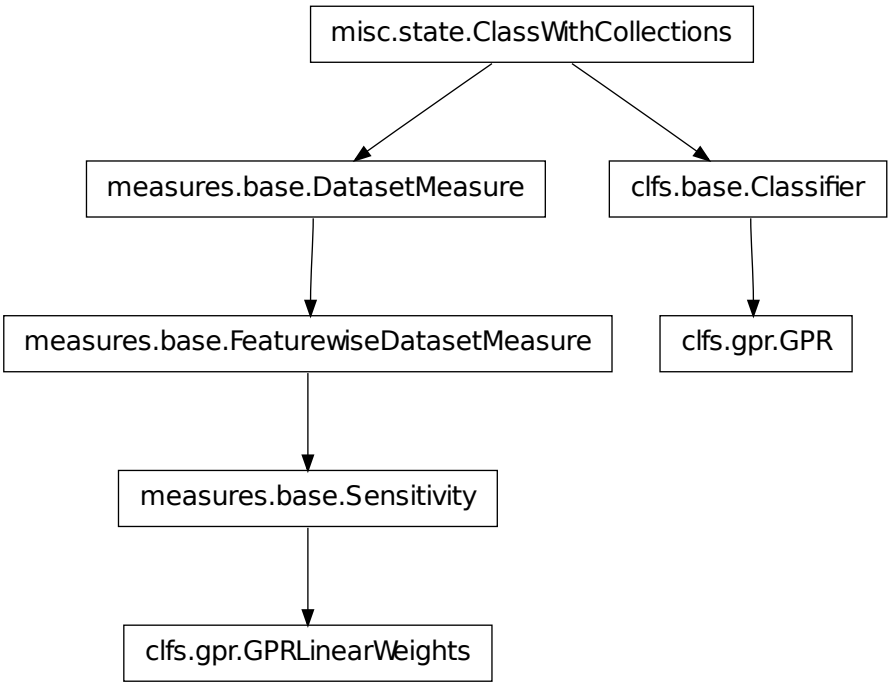
- *family* – Response type of your labels (either regression or classification. (Default: gaussian)
- *alpha* – The elastic net mixing parameter. Larger values will give rise to less L2 regularization, with alpha=1.0 as a true lasso penalty. (Default: 1.0)
- *nlambda* – Maximum number of lambdas to calculate 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.gpr

Module: `clfs.gpr`

Inheritance diagram for `mvpa.clfs.gpr`:



Gaussian Process Regression (GPR).

Classes

GPR

class `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

•*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.

- *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.

- *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 `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_result*: 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`

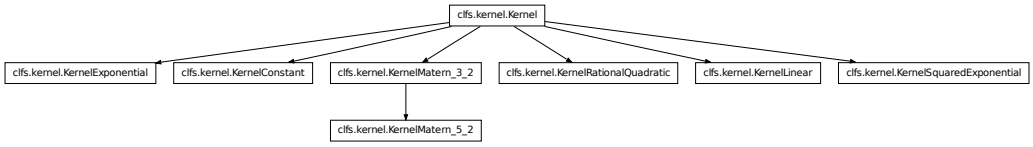
Initialize the analyzer with the classifier it shall use.

- *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.

16.4.7 clfs.kernel

Module: `clfs.kernel`

Inheritance diagram for `mvpa.clfs.kernel`:



Kernels for Gaussian Process Regression and Classification.

Classes

Kernel

class Kernel()
Bases: object

Kernel function base class.

compute(*data1, data2=None*)

compute_gradient(*alphaalphaTK*)

compute_lml_gradient(*alphaalphaT_Kinv, data*)

compute_lml_gradient_logscale(*alphaalphaT_Kinv, data*)

reset()
Resets the kernel dropping internal variables to the original values

KernelConstant

class KernelConstant(*sigma_0=1.0, **kwargs*)
Bases: mvpa.clfs.kernel.Kernel

The constant kernel class.

Initialize the constant kernel instance.

- *sigma_0* (float) – standard deviation of the Gaussian prior probability N(0,sigma_0**2) of the intercept of the constant regression. (Defaults to 1.0)

compute(*data1, data2=None*)
Compute kernel matrix.

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

compute_lml_gradient(*alphaalphaT_Kinv, data*)

compute_lml_gradient_logscale(*alphaalphaT_Kinv, data*)

set_hyperparameters(*hyperparameter*)

KernelExponential

class KernelExponential(*length_scale=1.0, sigma_f=1.0, **kwargs*)
Bases: mvpa.clfs.kernel.Kernel

The Exponential kernel class.

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

Initialize an Exponential kernel instance.

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

compute(*data1, data2=None*)
Compute kernel matrix.

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

compute_lml_gradient(*alphaalphaT_Kinv, data*)
Compute grandient of the kernel and return the portion of log marginal likelihood gradient due to the kernel. Shorter formula. Allows vector of lengthscales (ARD) BUT THIS LAST OPTION SEEMS NOT TO WORK FOR (CURRENTLY) UNKNOWN REASONS.

compute_lml_gradient_logscale(*alphaalphaT_Kinv, data*)
Compute grandient of the kernel and return the portion of log marginal likelihood gradient due to the kernel. Shorter formula. Allows vector of lengthscales (ARD). BUT THIS LAST OPTION SEEMS NOT TO WORK FOR (CURRENTLY) UNKNOWN REASONS.

gradient(*data1, data2*)
Compute gradient of the kernel matrix. A must for fast model selection with high-dimensional data.

set_hyperparameters(*hyperparameter*)
Set hyperaparmeters from a vector.
Used by model selection.

KernelLinear

class KernelLinear(*Sigma_p=None, sigma_0=1.0, **kwargs*)
Bases: mvpa.clfs.kernel.Kernel

The linear kernel class.

Initialize the linear kernel instance.

- *Sigma_p* (numpy.ndarray) – Covariance matrix of the Gaussian prior probability N(0,Sigma_p) on the weights of the linear regression. (Defaults to None)
- *sigma_0* (float) – the standard deviation of the Gaussian prior N(0,sigma_0**2) of the intercept of the linear regression. (Deafults to 1.0)

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

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

compute_lml_gradient(*alphaalphaT_Kinv, data*)

compute_lml_gradient_logscale(*alphaalphaT_Kinv, data*)

reset()

set_hyperparameters(*hyperparameter*)

KernelMatern_3_2

class KernelMatern_3_2(*length_scale=1.0, sigma_f=1.0, numerator=3.0, **kwargs*)
Bases: mvpa.clfs.kernel.Kernel

The Matern kernel class for the case ni=3/2 or ni=5/2.

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

Initialize a Squared Exponential kernel instance.

- *length_scale* (float OR numpy.ndarray) – the characteristic length-scale (or length-scales) of the phenomenon under investigation. (Defaults to 1.0)
- *sigma_f* (float) – Signal standard deviation. (Defaults to 1.0)
- *numerator* (float) – the numerator of parameter ni of Matern covariance functions. Currently only numerator=3.0 and numerator=5.0 are implemented. (Defaults to 3.0)

compute(*data1, data2=None*)
Compute kernel matrix.

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

gradient(*data1, data2*)
Compute gradient of the kernel matrix. A must for fast model selection with high-dimensional data.

set_hyperparameters(*hyperparameter*)
Set hyperaparmeters from a vector.
Used by model selection. Note: 'numerator' is not considered as an hyperparameter.

KernelMatern_5_2

class KernelMatern_5_2 (***kwargs*)

Bases: mvpa.clfs.kernel.KernelMatern_3_2

The Matern kernel class for the case ni=5/2.

This kernel is just KernelMatern_3_2(numerator=5.0).

Initialize a Squared Exponential kernel instance.

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

KernelRationalQuadratic

class KernelRationalQuadratic (*length_scale=1.0, sigma_f=1.0, alpha=0.5, **kwargs*)

Bases: mvpa.clfs.kernel.Kernel

The Rational Quadratic (RQ) kernel class.

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

Initialize a Squared Exponential kernel instance.

- *length_scale* (float OR numpy.ndarray) – the characteristic length-scale (or length-scales) of the phenomenon under investigation. (Defaults to 1.0)
- *sigma_f* (float) – Signal standard deviation. (Defaults to 1.0)
- *alpha* (float) – The parameter of the RQ functions family. (Defaults to 2.0)

compute (*data1, data2=None*)

Compute kernel matrix.

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

gradient (*data1, data2*)

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

set_hyperparameters (*hyperparameter*)

Set hyperaparmeters from a vector.

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

KernelSquaredExponential

class KernelSquaredExponential (*length_scale=1.0, sigma_f=1.0, **kwargs*)

Bases: mvpa.clfs.kernel.Kernel

The Squared Exponential kernel class.

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

Initialize a Squared Exponential kernel instance.

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

compute (*data1, data2=None*)

Compute kernel matrix.

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

compute_lml_gradient (*alphaalphaT_Kinv, data*)

Compute grandient of the kernel and return the portion of log marginal likelihood gradient due to the kernel. Shorter formula. Allows vector of lengthscales (ARD).

compute_lml_gradient_logscale(*alphaalphaT_Kinv, data*)
Compute grandient of the kernel and return the portion of log marginal likelihood gradient due to the kernel. Hyperparameters are in log scale which is sometimes more stable. Shorter formula. Allows vector of lengthscales (ARD).

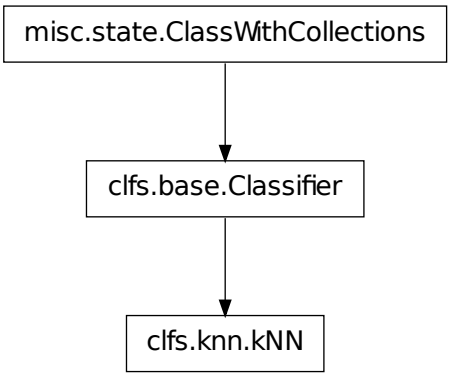
reset()

set_hyperparameters(*hyperparameter*)
Set hyperaparmeters from a vector.
Used by model selection.

16.4.8 clfs.knn

Module: clfs.knn

Inheritance diagram for mvpa.clfs.knn:



This module provides a k-Nearest-Neighbour classifier.

kNN

class kNN(*k=2, dfx=<function squared_euclidean_distance at 0x8c46e64>, 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 actuall 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
- *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`

- *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 neighbours.

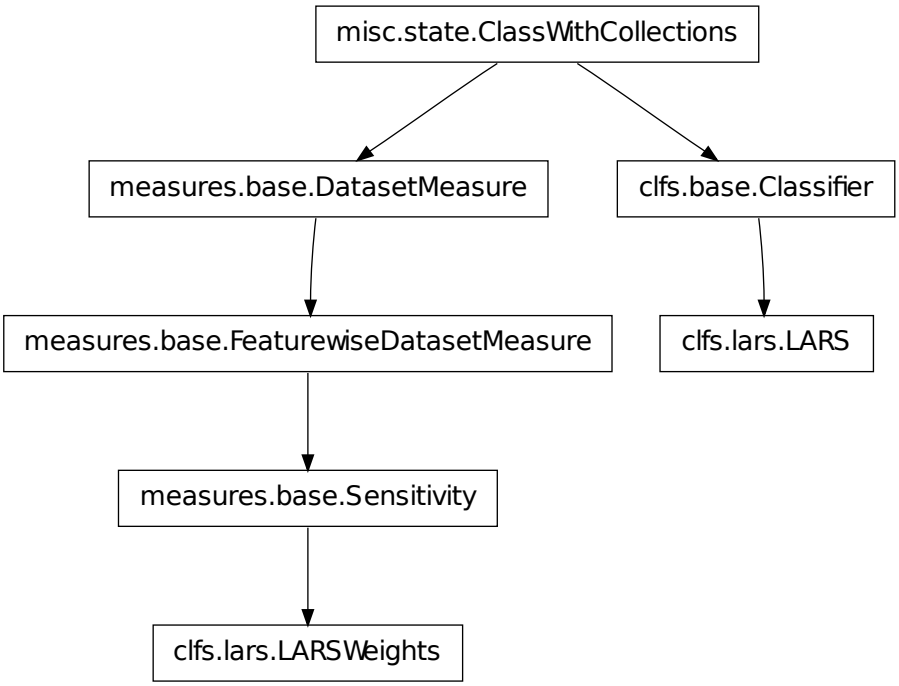
getWeightedVote(*knn_ids*)
Vote with classes weighted by the number of samples per class.

untrain()
Reset trained state

16.4.9 clfs.lars

Module: `clfs.lars`

Inheritance diagram for `mvpa.clfs.lars`:



Least angle regression (LARS) classifier.

Classes

LARS

class 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:

Bradley Efron, Trevor Hastie, Iain Johnstone and Robert Tibshirani, Least Angle Regression Annals of Statistics (with discussion) (2004) 32(2), 407-499. A new method for variable subset selection, with the lasso and 'epsilon' forward stagewise methods as special cases.

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
- *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:

- *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

LARSWeights

class 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_t*: State variable
- *raw_result*: 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.

- *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.

16.4.10 clfs.libsmlrc

Module: `clfs.libsmlrc`

Wraper for the stepwise_regression function for SMLR.

stepwise_regression(**args*)

16.4.11 clfs.libsmlrc.ctypes_helper

Module: `clfs.libsmlrc.ctypes_helper`

Helpers for wrapping C libraries with ctypes.

Functions

extend_args(**args*)
Turn ndarray arguments into dims and arrays.

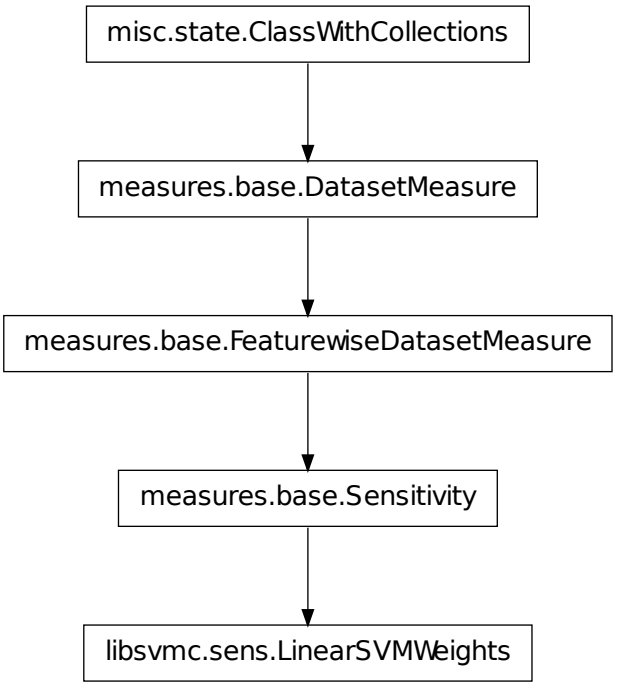
get_argtypes(**args*)

process_args(**args*)
Turn ndarray arguments into dims and array pointers for calling a ctypes-wrapped function.

16.4.12 clfs.libsvmc.sens

Module: `clfs.libsvmc.sens`

Inheritance diagram for `mvpa.clfs.libsvmc.sens`:



Provide sensitivity measures for libsvm's SVM.

LinearSVMWeights

class 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_result*: 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.

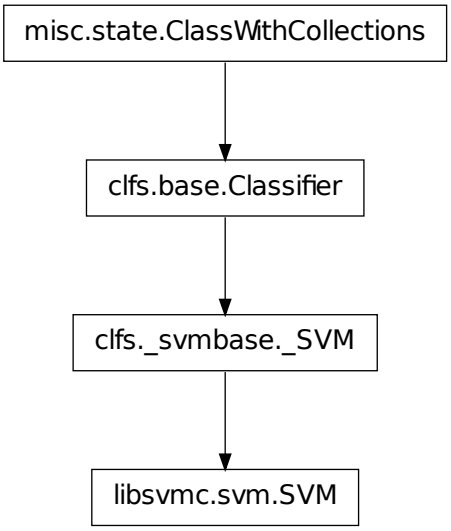
- *clf* (LinearSVM) – classifier to use. Only classifiers sub-classed from *LinearSVM* may be used.
- *split_weights* – If binary classification either to sum SVs per each class separately. (Default: False)
- *enable_states* (None or list of basestring) – Names of the state variables which should be 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.

16.4.13 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

class **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
- *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).

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.

rbf gamma

linear
: [provides sensitivity] No parameters

poly
: coef0, degree, gamma

sigmoid
: coef0, gamma

- *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

model

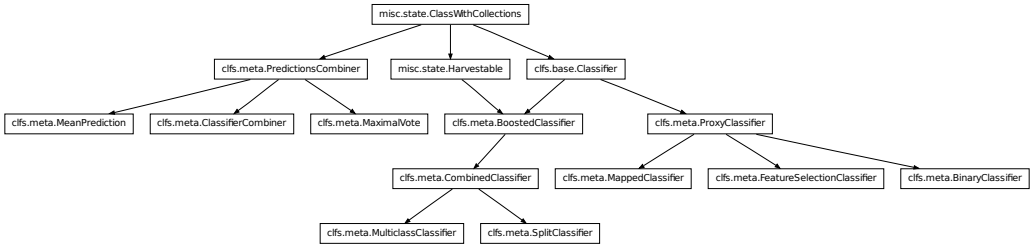
summary()
Provide quick summary over the SVM classifier

untrain()
Untrain libsvm's SVM: forget the model

16.4.14 clfs.meta

Module: clfs.meta

Inheritance diagram for mvpa.clfs.meta:



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 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
- *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`

- *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

BoostedClassifier

class BoostedClassifier(*clfs=None, propagate_states=True, harvest_attribs=None, copy_attribs='copy', **kwargs*)

Bases: `mvpa.clfs.base.Classifier`, `mvpa.misc.state.Harvestable`

Classifier containing the farm of other classifiers.

Should rarely be used directly. Use one of its childs instead

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
- *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.

- *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 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*

- *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 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

•*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.

- *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' descrete
labels but rather on raw 'class' values classifiers estimate (which is pretty much what is stored under *values*

combiner
Used combiner to derive a single result

summary()
Provide summary for the *CombinedClassifier*.

untrain()
Untrain *CombinedClassifier*

FeatureSelectionClassifier

class FeatureSelectionClassifier(*clf, feature_selection, testdataset=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 expclitely 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
- *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

- *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 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
- *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

- *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 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 unambigous (ie two classes have equal number of votes

- *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 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 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
- *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

- *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 PredictionsCombiner(*descr=None, **kwargs*)

Bases: mvpa.misc.state.ClassWithCollections

Base class for combining decisions of multiple classifiers

train(*clfs, dataset*)

PredictionsCombiner might need to be trained

- *clfs* (list of Classifier) – List of classifiers to combine. Has to be classifiers (not pure predictions), since combiner might use some other state variables (value's) instead of pure prediction's
- *dataset* (Dataset) – training data in this case

ProxyClassifier

class ProxyClassifier(*clf, **kwargs*)

Bases: mvpa.clfs.base.Classifier

Classifier which decorates another classifier

Possible uses:

- modify data somehow prior training/testing: * normalization * feature selection * modification
- optimized classifier?

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
- *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

- *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

clf
Used *Classifier*

getSensitivityAnalyzer(**args_, **kwargs_*)

summary()

untrain()
Untrain ProxyClassifier

SplitClassifier

class SplitClassifier(*clf, splitter=<mvpa.datasets.splitters.NFoldSplitter object at 0x8cd3dcc>, **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
- *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

- *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

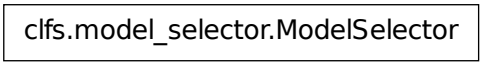
getSensitivityAnalyzer(**args_, **kwargs_*)

splitter
Splitter user by SplitClassifier

16.4.15 clfs.model_selector

Module: clfs.model_selector

Inheritance diagram for mvpa.clfs.model_selector:



Model selction.

ModelSelector

class 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 hyperparamenters).

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.

- *parametric_model* (Classifier) – the actual parameteric model to be optimized.
- *hyp_initial_guess* (numpy.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* (numpy.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)

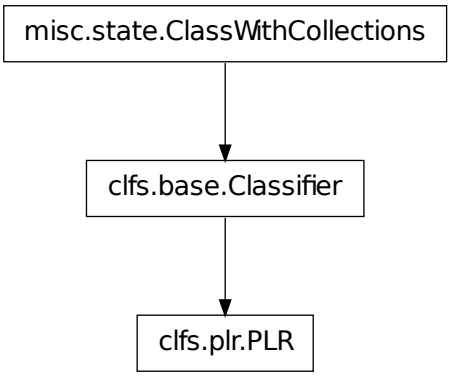
NOTE: the maximization of log_marginal_likelihood is a non-linear optimization problem (NLP). This fact is confirmed by Dmitrey, author of OpenOpt.

solve(*problem=None*)
Solve the maximization problem, check outcome and collect results.

16.4.16 clfs.plr

Module: clfs.plr

Inheritance diagram for mvpa.clfs.plr:



Penalized logistic regression classifier.

PLR

class PLR(*lm=1, criterion=1, reduced=False, 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
- *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

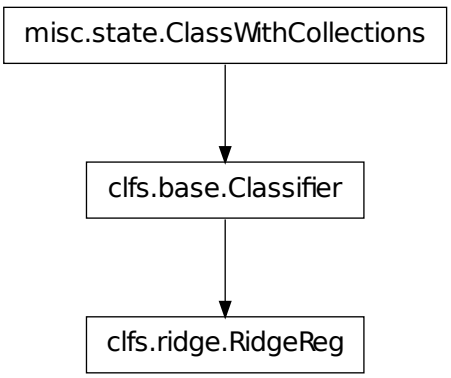
- *lm* (int) – the penalty term lambda.
- *criterion* (int) – the criterion applied to judge convergence.
- *reduced* (Bool) – if not False, the rank of the data is reduced before performing the calculations. In that case, reduce is taken as the fraction of the first singular value, at which a dimension is not considered significant anymore. A reasonable criterion is reduced=0.01

- *maxiter* (int) – maximum number of iterations. If no convergence occurs after this number of iterations, an exception is raised.
- *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

16.4.17 clfs.ridge

Module: `clfs.ridge`

Inheritance diagram for `mvpa.clfs.ridge`:



Ridge regression classifier.

RidgeReg

class 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
- *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

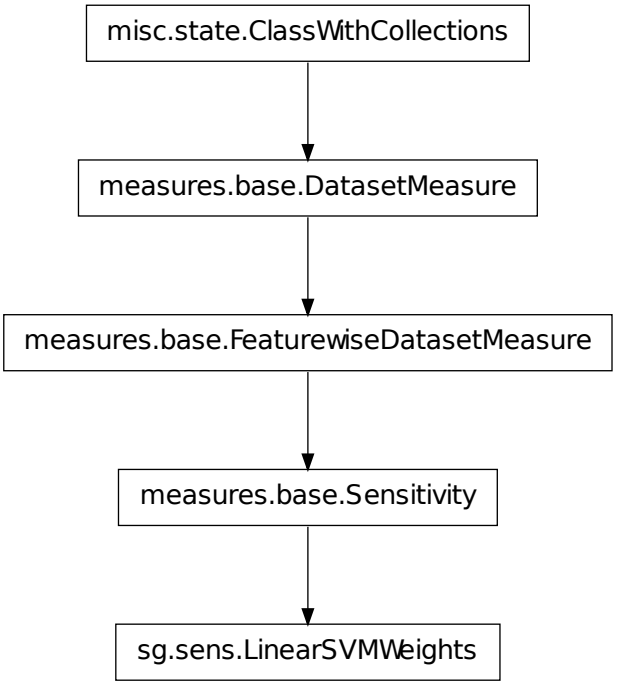
Initialize a ridge regression analysis.

- *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

16.4.18 clfs.sg.sens

Module: clfs.sg.sens

Inheritance diagram for mvpa.clfs.sg.sens:



Provide sensitivity measures for sg's SVM.

LinearSVMWeights

class 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_t*: State variable
- *raw_result*: 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`

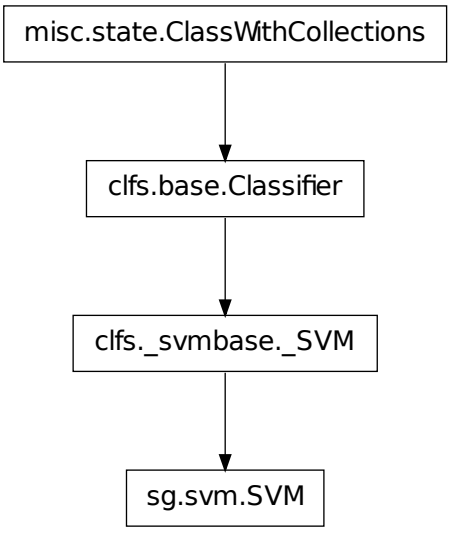
Initialize the analyzer with the classifier it shall use.

- *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.

16.4.19 clfs.sg.svm

Module: `clfs.sg.svm`

Inheritance diagram for `mvpa.clfs.sg.svm`:



Wrap the libsvm package into a very simple class interface.

SVM

class SVM(*kernel_type='linear', **kwargs*)

Bases: mvpa.clfs._svmbase._SVM

Support Vector Machine Classifier(s) based on Shogun

This is a simple base interface

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
- *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

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

- *epsilon* – Tolerance of termination criteria. (For nu-SVM default is 0.001). (Default: 5e-05)
- *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

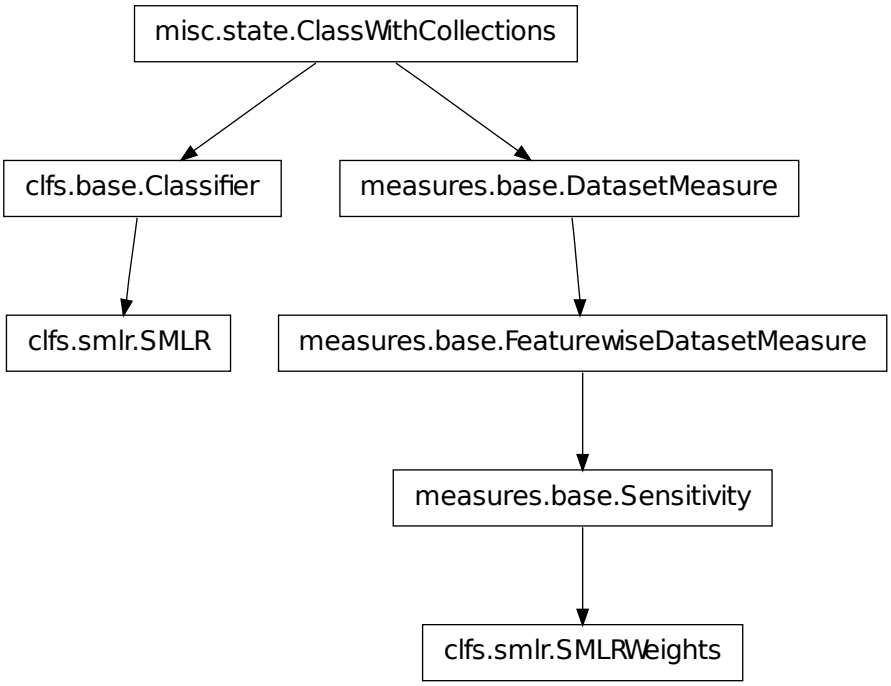
traindataset

untrain()

16.4.20 clfs.smlr

Module: `clfs.smlr`

Inheritance diagram for `mvpa.clfs.smlr`:



Sparse Multinomial Logistic Regression classifier.

Classes

SMLR

class SMLR (***kwargs*)

Bases: `mvpa.clfs.base.Classifier`

Sparse Multinomial Logistic Regression *Classifier*.

This is an implementation of the SMLR algorithm published in *Krishnapuram et al., 2005* (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
- *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 SMLR classifier.

- *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 SMLRWeights(*clf, force_training=True, **kwargs*)
Bases: `mvpa.measures.base.Sensitivity`

SensitivityAnalyzer that reports the weights SMLR trained on a given *Dataset*.

By default SMLR provides multiple weights per feature (one per label in training dataset). By default, all weights are combined into a single sensitivity value. Please, see the *FeaturewiseDatasetMeasure* constructor arguments how to custmize this behavior.

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_result*: 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`

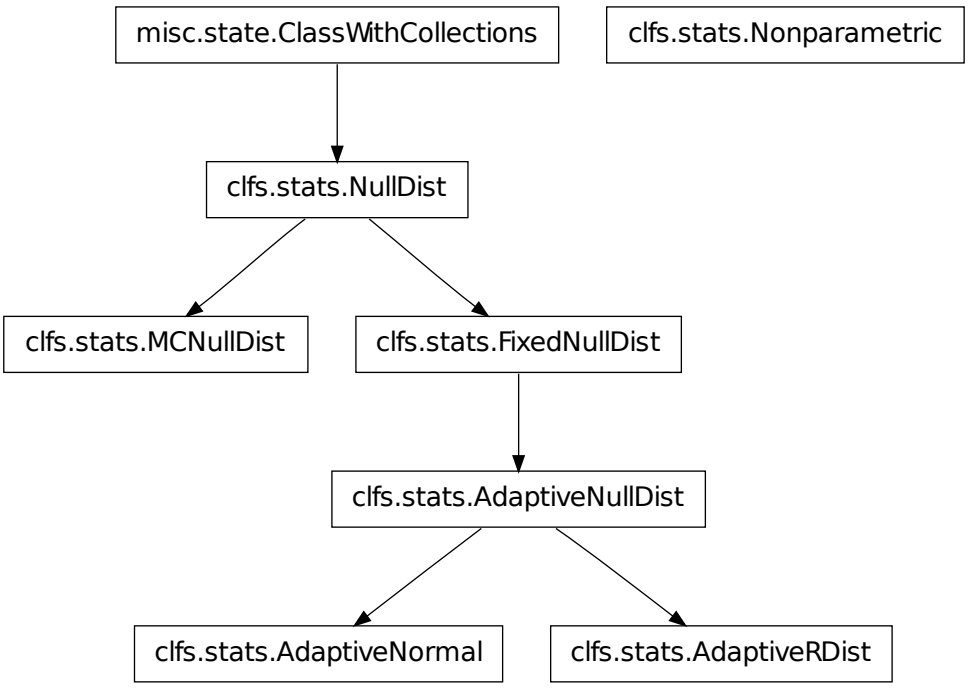
Initialize the analyzer with the classifier it shall use.

- *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.

16.4.21 clfs.stats

Module: `clfs.stats`

Inheritance diagram for `mvpa.clfs.stats`:



Estimator for classifier error distributions.

Classes

AdaptiveNormal

class 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

dist: distribution object
This can be any object the has a *cdf()* method to report the cumulative distribition function values.

- *dist* (distribution object) – This can be any object the has a *cdf()* method to report the cumulative distribition 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

AdaptiveNullDist

class 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

dist: distribution object
This can be any object the has a *cdf()* method to report the cumulative distribition function values.

- *dist* (distribution object) – This can be any object the has a *cdf()* method to report the cumulative distribition 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

fit (*measure, wdata, vdata=None*)

Cares about dimensionality of the feature space in measure

AdaptiveRDist

class 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

dist: distribution object
This can be any object the has a *cdf()* method to report the cumulative distribition function values.

- *dist* (distribution object) – This can be any object the has a *cdf()* method to report the cumulative distribition 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

cdf (*x*)

FixedNullDist

class FixedNullDist (*dist, **kwargs*)

Bases: mvpa.clfs.stats.NullDist

Proxy/Adaptor class for SciPy distributions.

All distributions from SciPy's 'stats' module can be used with this class.

```
>>> import numpy as N
>>> from scipy import stats
>>> from mvpa.clfs.stats import FixedNullDist
>>>
>>> dist = FixedNullDist(stats.norm(loc=2, scale=4))
>>> dist.p(2)
0.5
>>>
>>> dist.cdf(N.arange(5))
array([ 0.30853754,  0.40129367,  0.5       ,  0.59870633,  0.69146246])
>>>
>>> dist = FixedNullDist(stats.norm(loc=2, scale=4), tail='right')
>>> dist.p(N.arange(5))
array([ 0.69146246,  0.59870633,  0.5       ,  0.40129367,  0.30853754])
```

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

dist: distribution object
This can be any object the has a *cdf()* method to report the cumulative distribition function values.

•*dist* (distribution object) – This can be any object the has a *cdf()* method to report the cumulative distribition 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

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 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

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.

•*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

cdf (*x*)
Return value of the cumulative distribution function at *x*.

clean ()
Clean stored distributions

Storing all of the distributions might be too expensive (e.g. in case of Nonparametric), and the scope of the object might be too broad to wait for it to be destroyed. Clean would bind dist_samples to empty list to let gc revoke the memory.

fit (*measure, wdata, vdata=None*)
Fit the distribution by performing multiple cycles which repeatedly permuted labels in the training dataset.

•*measure* ((*Featurewise*)'DatasetMeasure' | *TransferError*) – TransferError instance used to compute all errors.

•*wdata* (*Dataset* which gets permuted and used to compute the) – measure/transfer error multiple times.

•*vdata* (*Dataset* used for validation.) – If provided measure is assumed to be a *TransferError* and working and validation dataset are passed onto it.

Nonparametric

class Nonparametric (*dist_samples*)
Bases: `object`

Non-parametric 1d distribution – derives cdf based on stored values.

Introduced to complement parametric distributions present in scipy.stats.

cdf (*x*)
Returns the cdf value at *x*.

fit

NullDist

class NullDist (*tail='both', **kwargs*)
Bases: `mvpa.misc.state.ClassWithCollections`

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:

`ClassWithCollections`

Cheap initialization.

tail: str ('left', 'right', 'any', 'both')
Which tail of the distribution to report. For 'any' and 'both' it chooses the tail it belongs to based on the comparison to p=0.5. In the case of 'any' significance is taken like in a one-tailed test.

•*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 dpending on the setting).

fit (*measure, wdata, vdata=None*)
Implement to fit the distribution to the data.

p (*x, **kwargs*)
Returns the p-value for values of *x*. Returned values are determined left, right, or from any tail depending on the constructor setting.
In case a *FeaturewiseDatasetMeasure* was used to estimate the distribution the method returns an array. In that case *x* can be a scalar value or an array of a matching shape.

tail

Functions

autoNullDist (*dist*)
Cheater for human beings – wraps *dist* if needed with some NullDist

tail and other arguments are assumed to be default as in NullDist/MCNullDist

nanmean (*x, axis=0*)
Compute the mean over the given axis ignoring nans.

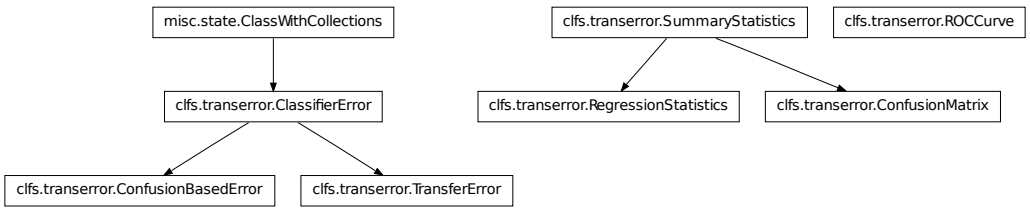
- *x* (ndarray) – input array
- *axis* (int) – axis along which the mean is computed.

m [float] the mean.

16.4.22 clfs.transerror

Module: clfs.transerror

Inheritance diagram for mvpa.clfs.transerror:



Utility class to compute the transfer error of classifiers.

Classes

ClassifierError

class ClassifierError (*clf, labels=None, train=True, **kwargs*)
Bases: mvpa.misc.state.ClassWithCollections

Compute (or return) some error of a (trained) classifier on a dataset.

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.

- *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

labels

ConfusionBasedError

class ConfusionBasedError(*clf, labels=None, confusion_state='training_confusion', **kwargs*)

Bases: mvpa.clfs.transerror.ClassifierError

For a given classifier report an error based on internally computed error measure (given by some *ConfusionMatrix* 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.

- *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* – Names of the state variables which should be enabled additionally to default ones
- *disable_states* – Names of the state variables which should be disabled
- *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

ConfusionMatrix

class ConfusionMatrix(*labels=None, labels_map=None, **kwargs*)

Bases: mvpa.clfs.transerror.SummaryStatistics

Class to contain information and display confusion matrix.

Implementation of the *SummaryStatistics* in the case of classification problem. Actual computation of confusion matrix is delayed until all data is acquired (to figure out complete set of labels). If testing data doesn't have a complete set of labels, but you like to include all labels, provide them as a parameter to the constructor.

Confusion matrix provides a set of performance statistics (use asstring(description=True) for the description of abbreviations), as well ROC curve (http://en.wikipedia.org/wiki/ROC_curve) plotting and analysis (AUC) in the limited set of problems: binary, multiclass 1-vs-all.

Initialize ConfusionMatrix with optional list of *labels*

- *labels* (list) – Optional set of labels to include in the matrix
- *labels_map* (None or dict) – Dictionary from original dataset to show mapping into numerical labels
- *targets* – Optional set of targets
- *predictions* – Optional set of predictions

asstring(*short=False, header=True, summary=True, description=False*)

'Pretty print' the matrix

- *short* (bool) – if True, ignores the rest of the parameters and provides consise 1 line summary
- *header* (bool) – print header of the table
- *summary* (bool) – print summary (accuracy)
- *description* (bool) – print verbose description of presented statistics

error

getLabels_map()

labels

labels_map

matrices

Return a list of separate confusion matrix per each stored set

matrix

percentCorrect

plot(*labels=None, numbers=False, origin='upper', numbers_alpha=None, xlabels_vertical=True, numbers_kwargs={}, **kwargs*)

Provide presentation of confusion matrix in image

- *labels* (list of int or basestring) – Optionally provided labels guarantee the order of presentation. Also value of None places empty column/row, thus provides visual grouping of labels (Thanks Ingo)

- *numbers* (bool) – Place values inside of confusion matrix elements
- *numbers_alpha* (None or float) – Controls textual output of numbers. If None – all numbers are plotted in the same intensity. If some float – it controls alpha level – higher value would give higher contrast. (good value is 2)
- *origin* (basestring) – Which left corner diagonal should start
- *xlabels_vertical* (bool) – Either to plot xlabels vertical (benefitial if number of labels is large)
- *numbers_kwargs* (dict) – Additional keyword parameters to be added to numbers (if numbers is True)
- ***kwargs* – Additional arguments given to imshow (eg me cmap)

Return type
(fig, im, cb) – figure, imshow, colorbar

setLabels_map(*val*)

ROCCurve

class ROCCurve(*labels, sets=None*)
Bases: object

Generic class for ROC curve computation and plotting

- *labels* (list) – labels which were used (in order of values if multiclass, or 1 per class for binary problems (e.g. in SMLR))
- *sets* (list of tuples) – list of sets for the analysis

ROCs

aucs
Compute and return set of AUC values 1 per label

plot(*label_index=0*)
TODO: make it friendly to labels given by values?
should we also treat labels_map?

RegressionStatistics

class RegressionStatistics(***kwargs*)
Bases: mvpa.clfs.transerror.SummaryStatistics

Class to contain information and display on regression results.

Initialize RegressionStatistics

- *targets* – Optional set of targets
- *predictions* – Optional set of predictions

asstring(*short=False, header=True, summary=True, description=False*)
'Pretty print' the statistics

error

plot(*plot=True, plot_stats=True, splot=True*)
Provide presentation of regression performance in image

- *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 SummaryStatistics(*targets=None, predictions=None, values=None, sets=None*)
Bases: object

Basic class to collect targets/predictions and report summary statistics

It takes care about collecting the sets, which are just tuples (targets, predictions, values). While 'computing' the matrix, all sets are considered together. Children of the class are responsible for computation and display.

Initialize SummaryStatistics

targets or predictions cannot be provided alone (ie targets without predictions)

- *targets* – Optional set of targets
- *predictions* – Optional set of predictions
- *values* – Optional set of values (which served for prediction)
- *sets* – Optional list of sets

add(*targets, predictions, values=None*)
Add new results to the set of known results

asstring(*short=False, header=True, summary=True, description=False*)
'Pretty print' the matrix

- *short* (bool) – if True, ignores the rest of the parameters and provides consise 1 line summary
- *header* (bool) – print header of the table
- *summary* (bool) – print summary (accuracy)
- *description* (bool) – print verbose description of presented statistics

compute()
Actually compute the confusion matrix based on all the sets

error

reset()
Cleans summary – all data/sets are wiped out

sets

stats

summaries
Return a list of separate summaries per each stored set

TransferError

class TransferError(*clf, errorfx=MeanMismatchErrorFx(), labels=None, null_dist=None, **kwargs*)
Bases: mvpa.clfs.transerror.ClassifierError

Compute the transfer error of a (trained) classifier on a dataset.

The actual error value is computed using a customizable error function. Optionally the classifier can be trained by passing an additional training dataset to the __call__() method.

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.

- *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* – Names of the state variables which should be enabled additionally to default ones
- *disable_states* – Names of the state variables which should be disabled
- *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

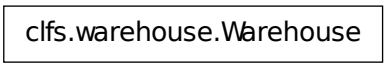
errorfx

null_dist

16.4.23 clfs.warehouse

Module: `clfs.warehouse`

Inheritance diagram for `mvpa.clfs.warehouse`:



Collection of classifiers to ease the exploration.

Warehouse

class 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

- *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

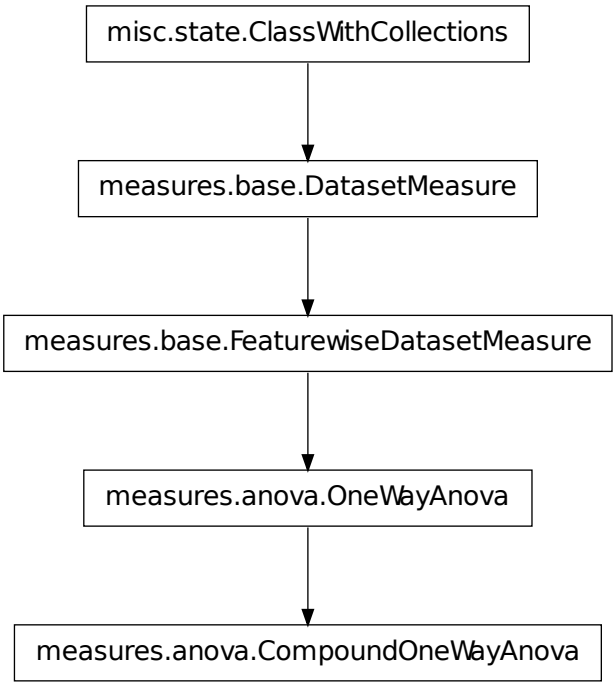
listing()
Listing (description + internals) of registered items

16.5 Measures: Searchlights and Sensitivties

16.5.1 measures.anova

Module: measures.anova

Inheritance diagram for mvpa.measures.anova:



FeaturewiseDatasetMeasure performing a univariate ANOVA.

Classes

CompoundOneWayAnova

class CompoundOneWayAnova(*combiner=<function SecondAxisSumOfAbs at 0x8f6f8b4>*, ***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_result*: 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

- *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

OneWayAnova

class OneWayAnova (*combiner=<function SecondAxisSumOfAbs at 0x8f6f8b4>, **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_result*: 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`

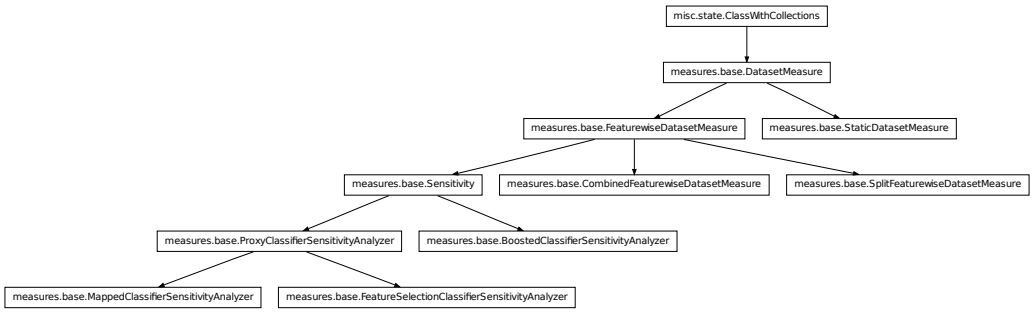
Initialize

- *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

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

class 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_result*: 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

- *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

CombinedFeaturewiseDatasetMeasure

class 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_result*: 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

- *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.

analyzers

Used analyzers

DatasetMeasure

class DatasetMeasure(*transformer=None, null_dist=None, **kwargs*)

Bases: mvpa.misc.state.ClassWithCollections

A measure computed from a *Dataset*

All dataset measures support arbitrary transformation of the measure after it has been computed. Transformation are done by processing the measure with a functor that is specified via the *transformer* keyword argument of the constructor. Upon request, the raw measure (before transformations are applied) is stored in the *raw_result* state variable.

Additionally all dataset measures support the estimation of the probabilit(y,ies) of a measure under some distribution. Typically this will be the NULL distribution (no signal), that can be estimated with permutation tests. If a distribution estimator instance is passed to the *null_dist* keyword argument of the constructor the respective probabilities are automatically computed and stored in the *null_prob* state variable.

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_result*: 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.

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.

- *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

transformer
: Return transformer

FeatureSelectionClassifierSensitivityAnalyzer

class 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_result*: 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

- *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 FeaturewiseDatasetMeasure(*combiner=<function SecondAxisSumOfAbs at 0x8f6f8b4>*, ***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_result*: 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

- *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

combiner

Return combiner

MappedClassifierSensitivityAnalyzer

class 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_result*: 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

- *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 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_result*: 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 ProxyClassifierSensitivityAnalyzer

- *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

analyzer

Sensitivity

class 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_result*: 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.

- *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.

clf

feature_ids

Return feature_ids used by the underlying classifier

SplitFeaturewiseDatasetMeasure

class SplitFeaturewiseDatasetMeasure(*splitter, analyzer, insplit_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_result*: 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

- *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.

StaticDatasetMeasure

class StaticDatasetMeasure(*measure=None, bias=None, *args, **kwargs*)

Bases: mvpa.measures.base.DatasetMeasure

A static (assigned) sensitivity measure.

Since implementation is generic it might be per feature or per whole dataset

Note: Available state variables:

- *null_prob*+: State variable

- *null_t*: State variable
- *raw_result*: 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.

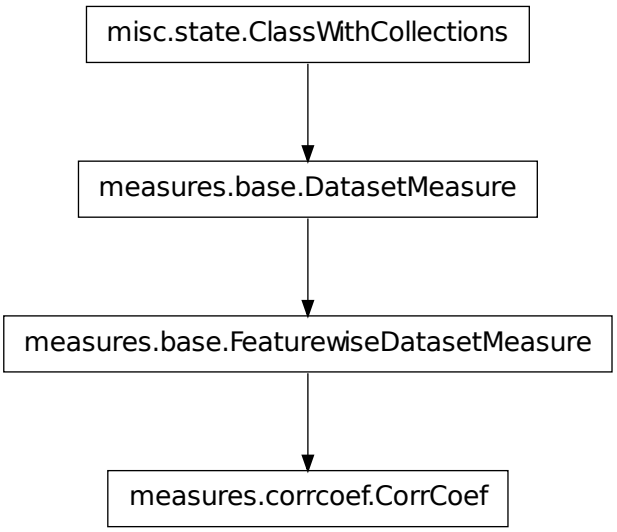
- *measure* – actual sensitivity to be returned
- *bias* – optionally available bias
- *enable_states* (None or list of basestring) – Names of the state variables which should be enabled additionally to default ones
- *disable_states* (None or list of basestring) – Names of the state variables which should be disabled

bias

16.5.3 measures.corrcoef

Module: `measures.corrcoef`

Inheritance diagram for `mvpa.measures.corrcoef`:



FeaturewiseDatasetMeasure of correlation with the labels.

`CorrCoef`

class `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_result*: 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`

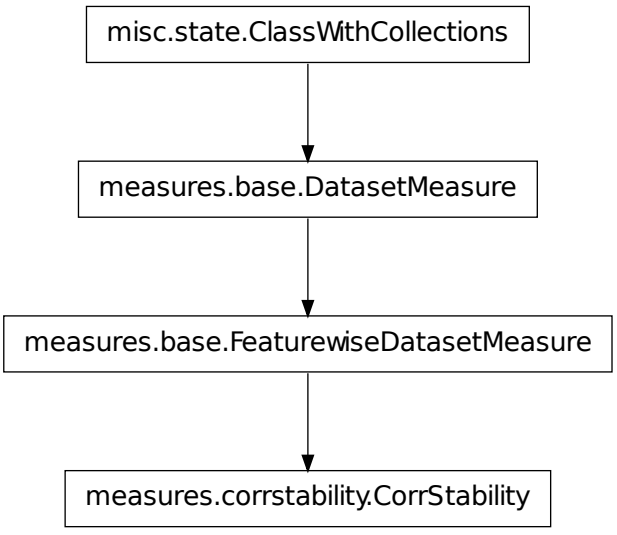
Initialize

- *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.

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 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_result*: 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

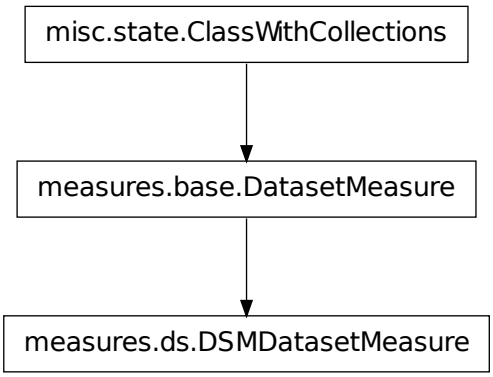
Initialize

- *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.

16.5.5 measures.ds

Module: measures.ds

Inheritance diagram for mvpa.measures.ds:



Dissimilarity measure.

DSMDatasetMeasure

class 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_result*: 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

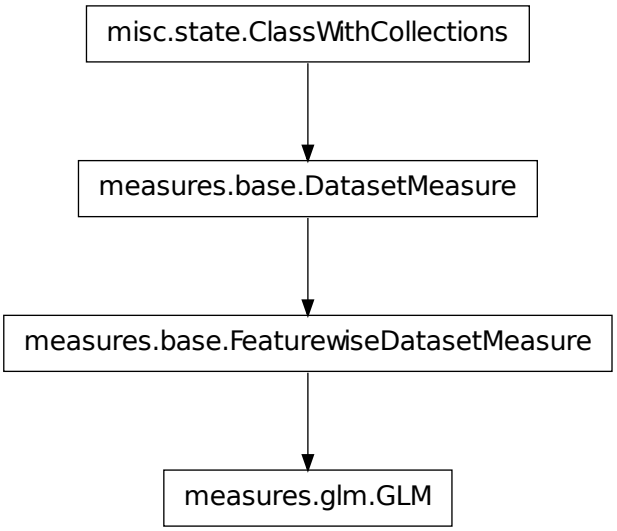
Initialize instance of DSMDatasetMeasure

- *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:



The general linear model (GLM).

GLM

class 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. indepently 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_result*: 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:

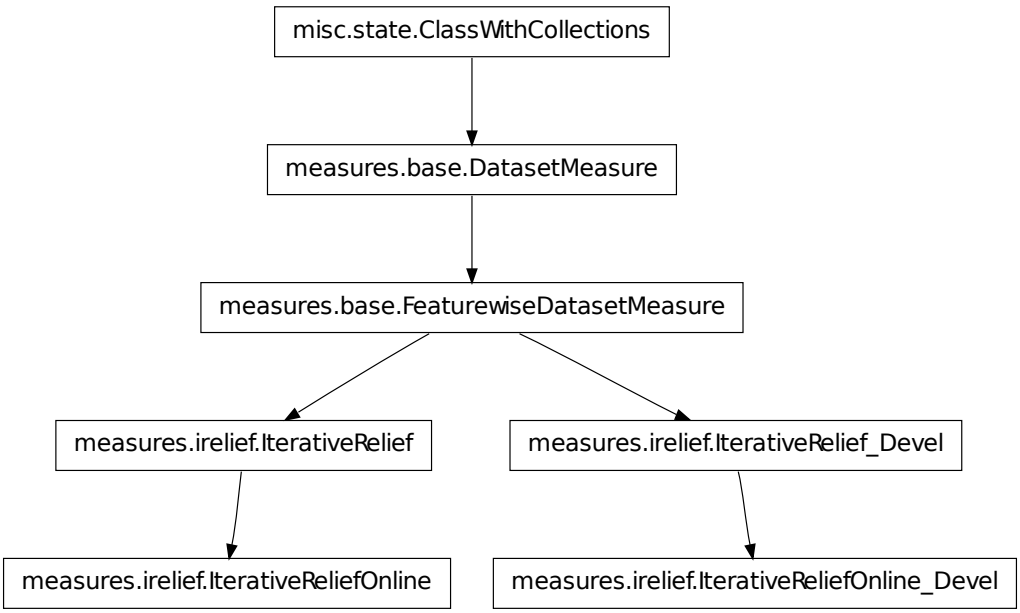
FeaturewiseDatasetMeasure

- *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.

16.5.7 measures.irelief

Module: measures.irelief

Inheritance diagram for mvpa.measures.irelief:



FeaturewiseDatasetMeasure performing multivariate Iterative RELIEF (I-RELIEF) algorithm. See : Y. Sun, Iterative RELIEF for Feature Weighting: Algorithms, Theories, and Applications, IEEE Trans. on Pattern Analysis and Machine Intelligence (TPAMI), vol. 29, no. 6, pp. 1035-1051, June 2007.

Classes

IterativeRelief

class IterativeRelief (*threshold=0.01, kernel_width=1.0, w_guess=None, **kwargs*)

Bases: mvpa.measures.base.FeaturewiseDatasetMeasure

FeaturewiseDatasetMeasure that performs multivariate I-RELIEF algorithm. Batch version.

Batch I-RELIEF-2 feature weighting algorithm. Works for binary or multiclass class-labels. Batch version with complexity O(T*N^2*I), where T is the number of iterations, N the number of instances, I the number of features.

See: Y. Sun, Iterative RELIEF for Feature Weighting: Algorithms, Theories, and Applications, IEEE Trans. on Pattern Analysis and Machine Intelligence (TPAMI), vol. 29, no. 6, pp. 1035-1051, June 2007. http://plaza.ufl.edu/sunyijun/Paper/PAMI_1.pdf

Note that current implementation allows to use only exponential-like kernels. Support for linear kernel will be added later.

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_result*: 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.

- *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.

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 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_result*: 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.

- *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.

IterativeReliefOnline_Devel

class IterativeReliefOnline_Devel(*a=5.0, permute=True, max_iter=3, **kwargs*)

Bases: mvpa.measures.irelief.IterativeRelief_Devel

FeaturewiseDatasetMeasure that performs multivariate I-RELIEF algorithm. Online version.

UNDER DEVELOPMENT

Online version with complexity O(T*N*I), where N is the number of instances and I the number of features.

See: Y. Sun, Iterative RELIEF for Feature Weighting: Algorithms, Theories, and Applications, IEEE Trans. on Pattern Analysis and Machine Intelligence (TPAMI), vol. 29, no. 6, pp. 1035-1051, June 2007. http://plaza.ufl.edu/sunyijun/Paper/PAMI_1.pdf

Note that this implementation is not fully online, since hit and miss dictionaries (H,M) are computed once at the beginning using full access to all labels. This can be easily corrected to a full online implementation. But this is not mandatory now since the major goal of this current online implementation is reduction of computational complexity.

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_result*: 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.

- *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.

IterativeRelief_Devel

class IterativeRelief_Devel(*threshold=0.01, kernel=None, kernel_width=1.0, w_guess=None, **kwargs*)

Bases: mvpa.measures.base.FeaturewiseDatasetMeasure

FeaturewiseDatasetMeasure that performs multivariate I-RELIEF algorithm. Batch version allowing various kernels.

UNDER DEVELOPEMNT.

Batch I-RELIEF-2 feature weighting algorithm. Works for binary or multiclass class-labels. Batch version with complexity O(T*N^2*I), where T is the number of iterations, N the number of instances, I the number of features.

See: Y. Sun, Iterative RELIEF for Feature Weighting: Algorithms, Theories, and Applications, IEEE Trans. on Pattern Analysis and Machine Intelligence (TPAMI), vol. 29, no. 6, pp. 1035-1051, June 2007. http://plaza.ufl.edu/sunyijun/Paper/PAMI_1.pdf

Note that current implementation allows to use only exponential-like kernels. Support for linear kernel will be added later.

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_result*: 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.

- *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.

compute_M_H(*label*)

Compute hit/miss dictionaries.

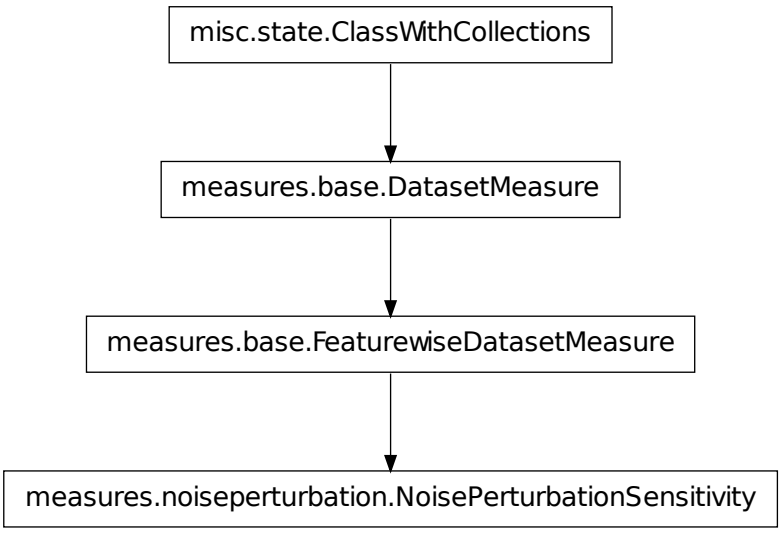
For each instance compute the set of indices having the same class label and different class label.

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 NoisePerturbationSensitivity(*datameasure*, *noise=<built-in method normal of mtrand.RandomState object at 0x401ed330>*)

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_result*: 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.

datameasure: *Datameasure* that is used to quantify the effect of noise perturbation.

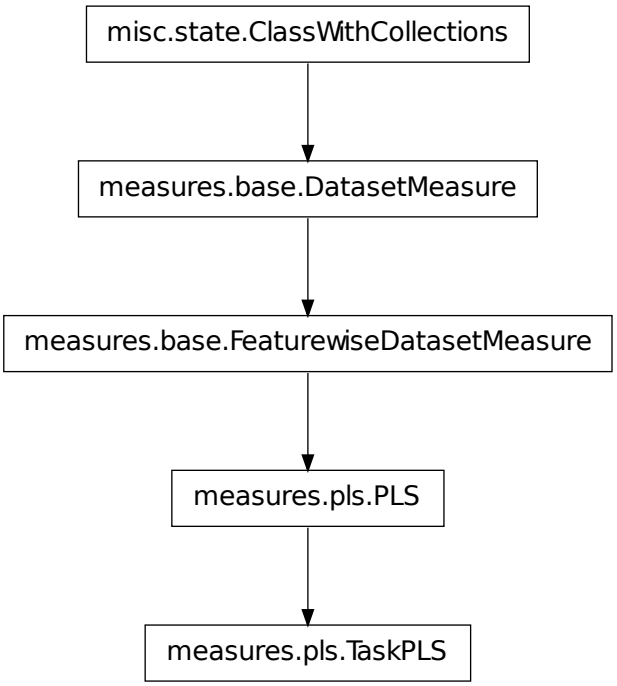
noise: Functor to generate noise. The noise generator has to return an 1d array of n values when called the *size=n* keyword argument. This is the default interface of the random number generators in NumPy's *random* module.

- *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 PLS (*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_result*: 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

- *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.

TaskPLS

class 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_result*: 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

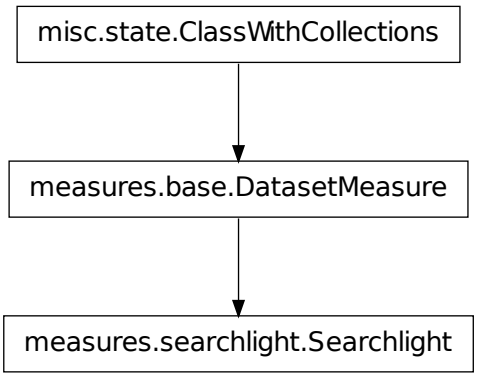
Initialize instance of PLS

- *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.

16.5.10 measures.searchlight

Module: measures.searchlight

Inheritance diagram for mvpa.measures.searchlight:



Implementation of the Searchlight algorithm

Searchlight

class Searchlight (*datameasure, radius=1.0, center_ids=None, **kwargs*)

Bases: mvpa.measures.base.DatasetMeasure

Runs a scalar *DatasetMeasure* on all possible spheres of a certain size within a dataset.

The idea for a searchlight algorithm stems from a paper by *Kriegeskorte et al. (2006)*.

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_result*: 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

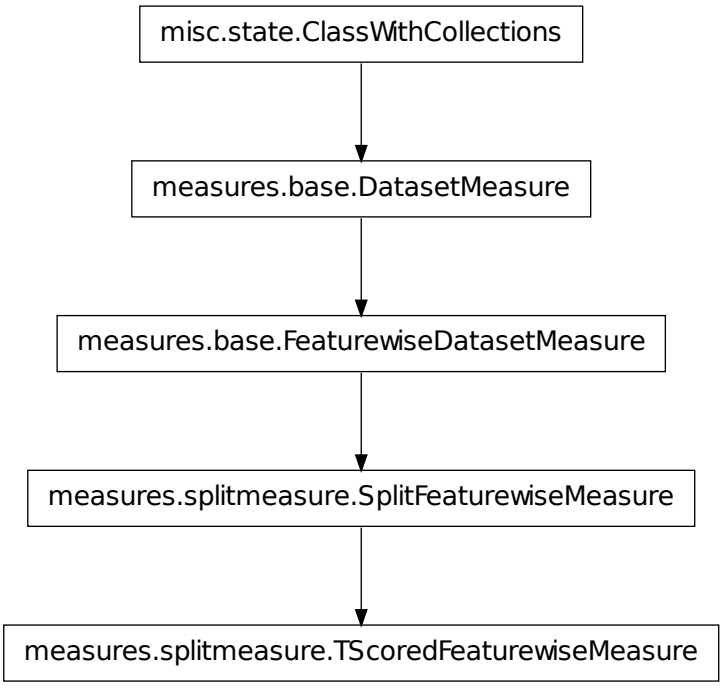
- *datameasure* (callable) – Any object that takes a Dataset and returns some measure when called.
- *radius* (float) – All features within the radius around the center will be part of a sphere. Provided dataset should have a metric assigned (for NiftiDataset, voxel size is used to provide such a metric, 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.
- *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
- *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

Note: If *Searchlight* is used as *SensitivityAnalyzer* one has to make sure that the specified scalar *DatasetMeasure* 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 sensitivites 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 differents splits of a *Dataset*.

Classes

SplitFeaturewiseMeasure

class SplitFeaturewiseMeasure(*sensana, splitter=<class 'mvpa.datasets.splitters.NoneSplitter'>, combiner=<function FirstAxisMean at 0x8f6f80c>, **kwargs*)

Bases: mvpa.measures.base.FeaturewiseDatasetMeasure

This is a *FeaturewiseDatasetMeasure* that uses another *FeaturewiseDatasetMeasure* and runs it multiple times on differents splits of a *Dataset*.

When called with a *Dataset* it returns the mean sensitivity maps of all data splits.

Additonally this class supports the *State* interface. Several postprocessing functions can be specififed 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_result*: 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.

- *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

TScoredFeaturewiseMeasure

class 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_result*: 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.

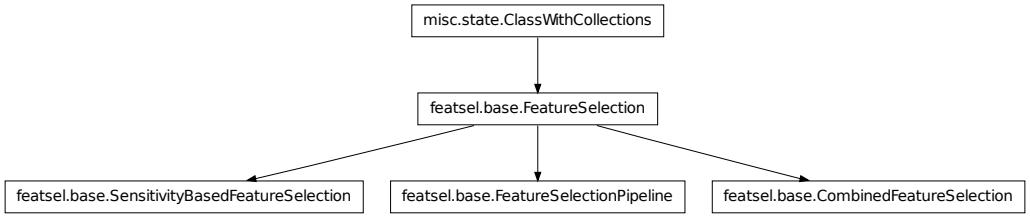
- *sensana* (SensitivityAnalyzer) – that shall be run on the *Dataset* splits.
- *splitter* (Splitter) – used to split the *Dataset*. By convention the first dataset in the tuple returned by the splitter on each iteration is used to compute the sensitivity map.
- *noise_level* (float) – Theoretical output 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 an 1d ndarray.

16.6 Feature Selection

16.6.1 featsel.base

Module: featsel.base

Inheritance diagram for mvpa.featsel.base:



Feature selection base class and related stuff base classes and helpers.

Classes

CombinedFeatureSelection

class 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

- *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*

FeatureSelection

class FeatureSelection (*kwargs*)

Bases: mvpa.misc.state.ClassWithCollections

Base class for any feature selection

Base class for Functors which implement feature selection on the datasets.

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

- *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

FeatureSelectionPipeline

class FeatureSelectionPipeline (*feature_selections, **kwargs*)

Bases: `mvpa.featsel.base.FeatureSelection`

Feature elimination through the list of FeatureSelection's.

Given as list of FeatureSelections it applies them in turn.

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

- *feature_selections* (lisf 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*

SensitivityBasedFeatureSelection

class SensitivityBasedFeatureSelection (*sensitivity_analyzer, feature_selector=FractionTailSelector(fraction=0.050000, **kwargs*)

Bases: `mvpa.featsel.base.FeatureSelection`

Feature elimination.

A *FeaturewiseDatasetMeasure* is used to compute sensitivity maps given a certain dataset. These sensitivity maps are in turn used to discard unimportant features.

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

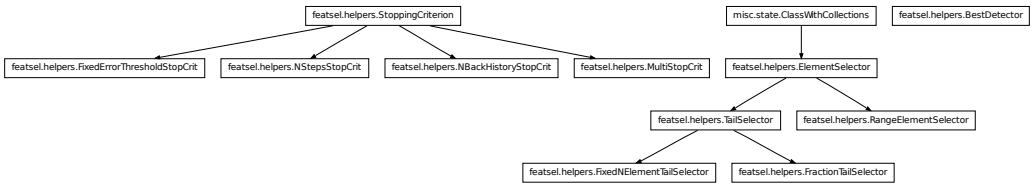
- *sensitivity_analyzer* (FeaturewiseDatasetMeasure) – sensitivity analyzer to come up with sensitivity
- *feature_selector* (Functor) – Given a sensitivity map it has to return the ids of those features that should be kept.
- *enable_states* (None or list of basestring) – Names of the state variables 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

16.6.2 featsel.helpers

Module: `featsel.helpers`

Inheritance diagram for `mvpa.featsel.helpers`:



Classes

`BestDetector`

class `BestDetector` (*func=<built-in function min>*, *lastminimum=False*)
Bases: `object`

Determine whether the last value in a sequence is the best one given some criterion.

Initialize with number of steps

- *fun* (functor) – Functor to select the best results. Defaults to min
- *lastminimum* (bool) – Toggle whether the latest or the earliest minimum is used as optimal value to determine the stopping criterion.

bestindex

ElementSelector

class ElementSelector (*mode='discard', **kwargs*)

Bases: mvpa.misc.state.ClassWithCollections

Base class to implement functors to select some elements based on a sequence of values.

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:

ClassWithCollections

Cheap initialization.

- *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

mode

FixedErrorThresholdStopCrit

class FixedErrorThresholdStopCrit (*threshold*)

Bases: mvpa.featsel.helpers.StoppingCriterion

Stop computation if the latest error drops below a certain threshold.

Initialize with threshold.

- *threshold* (float [0,1]) – Error threshold.

threshold

FixedNElementTailSelector

class FixedNElementTailSelector (*nelements, **kwargs*)

Bases: mvpa.featsel.helpers.TailSelector

Given a sequence, provide set of IDs for a fixed number of to be selected elements.

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.

- *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 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.

- *felements* (float (0,1.0]) – Fraction of elements to select/discard. Note: Even when 0.0 is specified at least one element will be selected.
- *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 MultiStopCrit (*crits, mode='or'*)

Bases: mvpa.featsel.helpers.StoppingCriterion

Stop computation if the latest error drops below a certain threshold.

- *crits* (list of StoppingCriterion instances) – For each call to MultiStopCrit all of these criterions will be evaluated.
- *mode* (any of ('and', 'or')) – Logical function to determine the multi criterion from the set of base criteria.

NBackHistoryStopCrit

class NBackHistoryStopCrit (*bestdetector=<mvpa.featsel.helpers.BestDetector object at 0x9133aac>, steps=10*)

Bases: mvpa.featsel.helpers.StoppingCriterion

Stop computation if for a number of steps error was increasing

Initialize with number of steps

- *bestdetector* (BestDetector instance) – used to determine where the best error is located.
- *steps* (int) – How many steps to check after optimal value.

steps

NStepsStopCrit

class NStepsStopCrit (*steps*)

Bases: mvpa.featsel.helpers.StoppingCriterion

Stop computation after a certain number of steps.

Initialize with number of steps.

- *steps* (int) – Number of steps after which to stop.

steps

RangeElementSelector

class RangeElementSelector (*lower=None, upper=None, inclusive=False, mode='select', **kwargs*)

Bases: mvpa.featsel.helpers.ElementSelector

Select elements based on specified range of values

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*

- *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 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 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

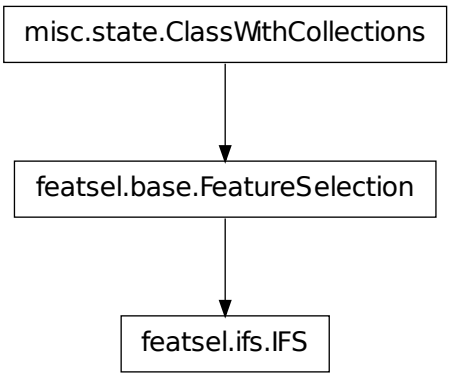
Initialize TailSelector

- *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:



Incremental feature search (IFS).

Very similar to Recursive feature elimination (RFE), but instead of begining with all features and stripping some sequentially, start with an empty feature set and include important features successively.

IFS

class IFS(*data_measure, transfer_error, bestdetector=<mvpa.featsel.helpers.BestDetector object at 0x8cb5d2c>, stopping_criterion=<mvpa.featsel.helpers.NBackHistoryStopCrit object at 0x8b633cc>, 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 in turn used to incrementally select important features. Starting with an empty feature set the dataset measure is first computed for each single feature. A number of features is selected based on the resulting data measure map (using an *ElementSelector*).

Next the dataset measure is computed again using each feature in addition to the already selected feature set. Again the *ElementSelector* is used to select more features.

For each feature selection the transfer error on some testdatset is computed. This procedure is repeated until a given *StoppingCriterion* is reached.

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

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.

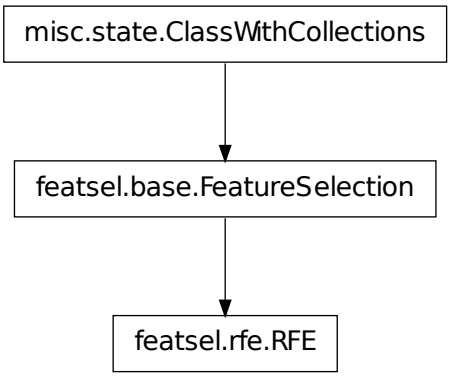
stopping_criterion
: [Functor] Given a list of error values it has to return whether the criterion is fulfilled.

- *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

class RFE (*sensitivity_analyzer, transfer_error, feature_selector=FractionTailSelector() fraction=0.050000, bestdetector=<mvpa.featsel.helpers.BestDetector object at 0x929dc0c>, stopping_criterion=<mvpa.featsel.helpers.NBackHistoryStopCrit object at 0x929daec>, train_clf=None, update_sensitivity=True, **kargs*)

Bases: `mvpa.featsel.base.FeatureSelection`

Recursive feature elimination.

A *FeaturewiseDatasetMeasure* is used to compute sensitivity maps given a certain dataset. These sensitivity maps are in turn used to discard unimportant features. For each feature selection the transfer error on some testdatset is computed. This procedure is repeated until a given *StoppingCriterion* is reached.

Such strategy after

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

was applied to SVM-based analysis of fMRI data in

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

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

- *sensitivity_analyzer* (FeaturewiseDatasetMeasure object) –

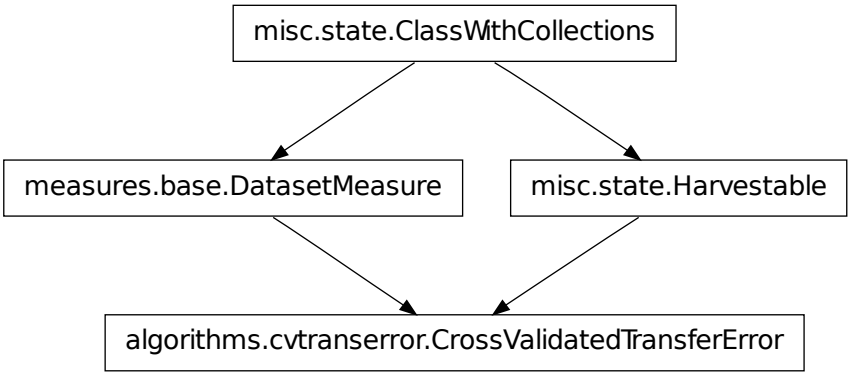
- *transfer_error* (TransferError object) – used to compute the transfer error of a classifier based on a certain feature set on the test dataset. NOTE: If sensitivity analyzer is based on the same classifier as transfer_error is using, make sure you initialize transfer_error with train=False, otherwise it would train classifier twice without any necessity.
- *feature_selector* (Functor) – Given a sensitivity map it has to return the ids of those features that should be kept.
- *bestdetector* (Functor) – Given a list of error values it has to return a boolean that signals whether the latest error value is the total minimum.
- *stopping_criterion* (Functor) – Given a list of error values it has to return whether the criterion is fulfilled.
- *train_clf* (bool) – Flag whether the classifier in *transfer_error* should be trained before computing the error. In general this is required, but if the *sensitivity_analyzer* and *transfer_error* share and make use of the same classifier it can be switched off to save CPU cycles. Default *None* checks if sensitivity_analyzer is based on a classifier and doesn't train if so.
- *update_sensitivity* (bool) – If False the sensitivity map is only computed once and reused for each iteration. Otherwise the senstitivities are recomputed at each selection step.
- *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

class 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_result*: 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

(States enabled by default are listed with +)

See Also:

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

DatasetMeasure, Harvestable

- *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 migth wanted to be enabled (provide testdataset for RFE to determine stopping point).
- *harvest_attribs* (list of basestr) – What attributes of call to store and return within harvested state variable
- *copy_attribs* (None | 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

combiner
Access to the configured combiner.

splitter
Access to the Splitter instance.

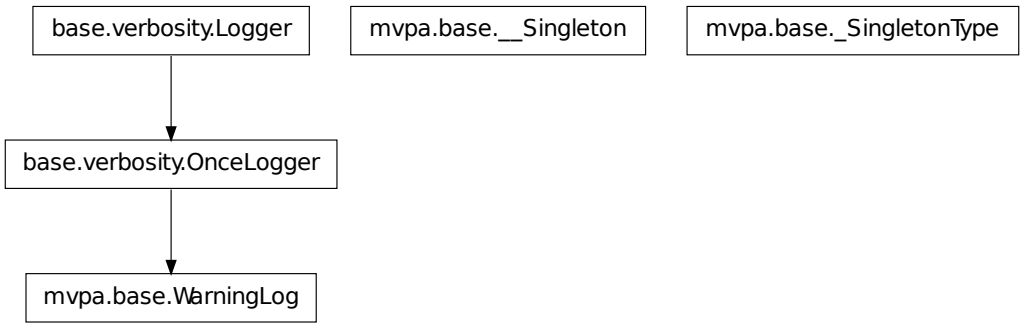
transerror
Access to the TransferError instance.

16.8 Common Facilities

16.8.1 base

Module: base

Inheritance diagram for mvpa.base:



Base functionality of PyMVPA

Module Organization

mvpa.base module contains various modules which are used through out PyMVPA code, and are generic building blocks

group Basic
: externals, config, verbosity, dochelpers

WarningLog

class WarningLog(*btlevels=10, btdefault=False, maxcount=1, *args, **kwargs*)

Bases: mvpa.base.verbosity.OnceLogger

Logging class of messsages to be printed just once per each message

Define Warning logger.

btlevels
: [int] how many levels of backtrack to print to give a hint on WTF

btdefault
: [bool] if to print backtrace for all warnings at all

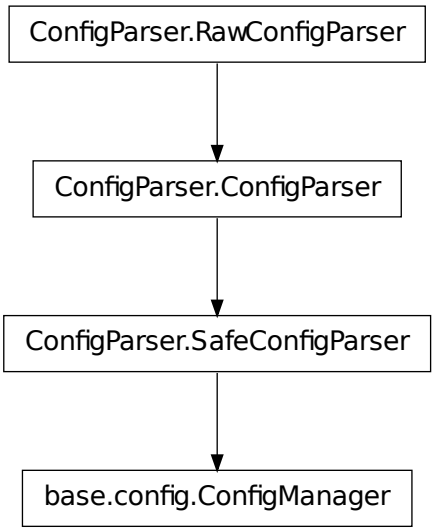
maxcount
: [int] how many times to print each warning

maxcount

16.8.2 base.config

Module: base.config

Inheritance diagram for mvpa.base.config:



Registry-like monster

ConfigManager

class ConfigManager (*filenames=None*)

Bases: ConfigParser.SafeConfigParser

Central configuration registry for PyMVPA.

The purpose of this class is to collect all configurable settings used by various parts of PyMVPA. It is fairly simple and does only little more than the standard Python ConfigParser. Like ConfigParser it is blind to the data that it stores, i.e. not type checking is performed.

Configuration files (INI syntax) in multiple location are passed when the class is instanciated or whenever *Config.reload()* is called later on. By default it looks for a config file named *pymvpa.cfg* in the current directory and *.pymvpa.cfg* in the user's home directory. Morever, the constructor takes an optional argument with a list of additional file names to parse.

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

```
MVPA_VERBOSE=1
```

will become:

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

However, *MVPA_VERBOSE_OUTPUT=stdout* becomes:

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

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

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

Settings from custom configuration files (specified by the constructor argument) have the highest priority and override settings found in the current directory. They in turn override user-specific settings and finally the content of any *MVPA_** environment variables overrides all settings read from any file.

Initialization reads settings from config files and env. variables.

•*filenames* (list of filenames) –

get(*section, option, default=None, **kwargs*)
Wrapper around SafeConfigParser.get() with a custom default value.

This method simply wraps the base class method, but adds a *default* keyword argument. The value of *default* is returned whenever the config parser does not have the requested option and/or section.

getAsDType(*section, option, dtype, default=None*)
Convenience method to query options with a custom default and type

This method simply wraps the base class method, but adds a *default* keyword argument. The value of *default* is returned whenever the config parser does not have the requested option and/or section.

In addition, the returned value is converted into the specified *dtype*.

getboolean(*section, option, default=None*)
Wrapper around SafeConfigParser.getboolean() with a custom default.

This method simply wraps the base class method, but adds a *default* keyword argument. The value of *default* is returned whenever the config parser does not have the requested option and/or section.

reload()
Re-read settings from all configured locations.

save(*filename*)
Write current configuration to a file.

16.8.3 base.dochelpers

Module: base.dochelpers

Various helpers to improve docstrings and textual output

Functions

enhancedDocString(*item, *args, **kwargs*)
Generate enhanced doc strings for various items.

•*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

handleDocString(*text, polite=True*)
Take care of empty and non existing doc strings.

rstUnderline(*text, markup*)
Add and underline RsT string matching the length of the given string.

singleOrPlural (*single, plural, n*)
Little helper to spit out single or plural version of a word.

table2string (*table, out=None*)
Given list of lists figure out their common widths and print to out

- *table* (list of lists of strings) – What is aimed to be printed
- *out* (None or stream) – Where to print. If None – will print and return string

Return type
string if out was None

16.8.4 base.externals

Module: base.externals

Helper to verify presence of external libraries and modules

Functions

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.

- *dep* (string or list of string) – The dependency key(s) to test.
- *force* (boolean) – Whether to force the test even if it has already been performed.
- *raiseException* (boolean) – Whether to raise RuntimeError if dependency is missing.
- *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.

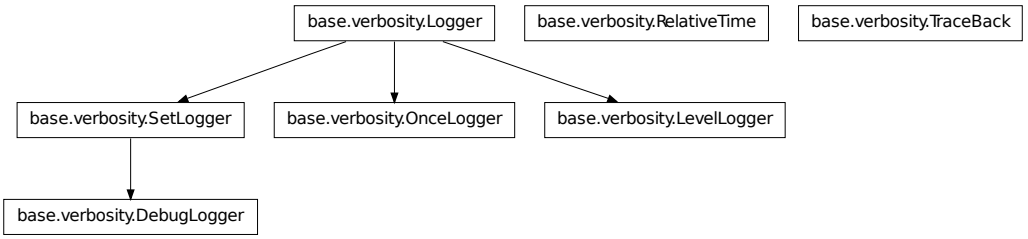
testAllDependencies (*force=False*)
Test for all known dependencies.

- *force* (boolean) – Whether to force the test even if it has already been performed.

16.8.5 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 **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 **Logger**(*handlers=None*)
Bases: object

Base class to provide logging

Initialize the logger with a set of handlers to use for output

Each hanlder must have write() method implemented

handlers
Return active handlers

lfprev

OnceLogger

class **OnceLogger**(**args, **kwargs*)
Bases: mvpa.base.verbosity.Logger

Logger which prints a message for a given ID just once.

It could be used for one-time warning to don't overfill the output with useless repeatative messages

Define once logger.

SetLogger

class **SetLogger**(*register=None, active=None, printsetid=True, *args, **kwargs*)
Bases: mvpa.base.verbosity.Logger

Logger which prints based on defined sets identified by Id.

active

print_registered(*detailed=True*)

printsetid

register(*setid, description*)
"Register" a new setid with a given description for easy finding

registered

setActiveFromString(*value*)
Given a string listing registered(?) setids, make then active

16.9 Miscellaneous

16.9.1 misc.args

Module: misc.args

Helpers for arguments handling.

Functions

group_kwargs (*prefixes, assign=False, passthrough=False*)
Decorator function to join parts of kwargs together

- *prefixes* (list of basestrings) – Prefixes to split based on. See *split_kwargs*
- *assign* (bool) – Flag to assign the obtained arguments to self._<prefix>_kwargs
- *passthrough* (bool) – Flag to pass joined arguments as <prefix>_kwargs argument. Usually it is sufficient to have either assign or passthrough. If none of those is True, decorator simply filters out mentioned groups from being passed to the method

Example: if needed to join all args which start with 'slave<underscore>' together under slave_kwargs parameter

split_kwargs (*kwargs, prefixes=, []*)
Helper to separate kwargs into multiple groups

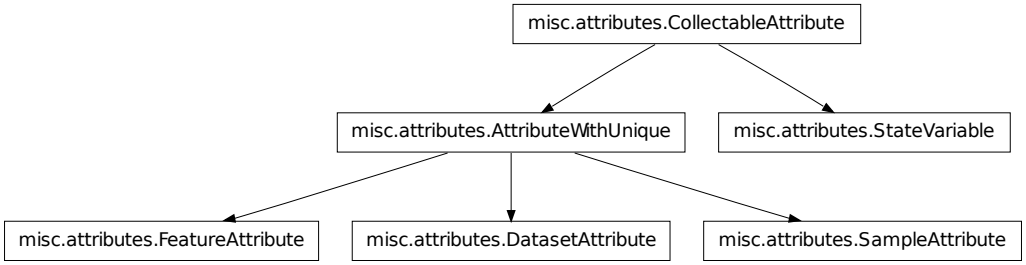
- *prefixes* (list of basestrings) – Each entry sets a prefix which puts entry with key starting with it into a separate group. Group '' corresponds to 'leftovers'

Output
dictionary with keys == *prefixes*

16.9.2 misc.attributes

Module: misc.attributes

Inheritance diagram for mvpa.misc.attributes:



Module with some special objects to be used as magic attributes with dedicated containers aka. *Collections*.

Classes

AttributeWithUnique

class 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).

- don't need to mess with getattr since it would become just another attribute
- might be worse design in terms of comprehension
- take care about _set, since we shouldn't allow change it externally

For now lets do it within a single class and tune up getattr

hasunique

reset()

uniqueValues

CollectableAttribute

class 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 groupped into corresponding collections for each class by statecollector metaclass, ie it would be done on a class creation (ie not per each object)

isSet

name

reset()
Simply reset the flag

value

DatasetAttribute

class DatasetAttribute (*name=None, hasunique=True, doc='Attribute with unique'*)
Bases: mvpa.misc.attributes.AttributeWithUnique

FeatureAttribute

class FeatureAttribute (*name=None, hasunique=True, doc='Attribute with unique'*)
Bases: mvpa.misc.attributes.AttributeWithUnique

SampleAttribute

class SampleAttribute(*name=None, hasunique=True, doc='Attribute with unique'*)
Bases: mvpa.misc.attributes.AttributeWithUnique

StateVariable

class StateVariable(*name=None, enabled=True, doc='State variable'*)
Bases: mvpa.misc.attributes.CollectableAttribute

Simple container intended to conditionally store the value

enable(*value=False*)

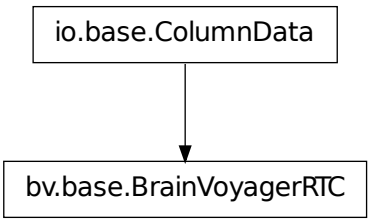
isEnabled

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:



Tiny snippets to interface with FSL easily.

BrainVoyagerRTC

class BrainVoyagerRTC(*source*)
Bases: mvpa.misc.io.base.ColumnData

IO helper to read BrainVoyager RTC files.

This is a textfile format that is used to specify stimulation protocols for data analysis in BrainVoyager. It looks like

FileVersion: 2 Type: DesignMatrix NrOfPredictors: 4 NrOfDataPoints: 147

"fm_l_60dB" "fm_r_60dB" "fm_l_80dB" "fm_r_80dB" 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000

Data is always read as *float* and header is actually ignored

Read and write BrainVoyager RTC files.

•*source* (filename of an RTC file) –

toarray()
Returns the data as an array

16.9.4 misc.cmdline

Module: misc.cmdline

Inheritance diagram for mvpa.misc.cmdline:

Common functions and options definitions for command line

__docformat__ = 'restructuredtext'

Conventions: Every option (instance of optparse.Option) has prefix "opt". Lists of options has prefix opts (e.g. *opts.common*).

Option name should be camelbacked version of .dest for the option.

Classes

OptionGroups

class OptionGroups(*parser*)

Bases: object

Group creation is delayed until instance is requested.

This allows to overcome the problem of poluting handled cmdline options

add(*name, l, doc*)

Options

class 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

chirpLinear(*n_instances, n_features=4, n_nonbogus_features=2, data_noise=0.40000000000000002, noise=0.10000000000000001*)

Generates simple dataset for linear regressions

Generates chirp signal, populates n_nonbogus_features out of n_features with it with different noise level and then provides signal itself with additional noise as labels

dumbFeatureBinaryDataset()

Very simple binary (2 labels) dataset

dumbFeatureDataset()
Create a very simple dataset with 2 features and 3 labels

getMVPattern(*s2n*)
Simple multivariate dataset

linear_awgn(*size=10, intercept=0.0, slope=0.40000000000000002, noise_std=0.01, flat=False*)
Generate a dataset from a linear function with Added White Gaussian Noise (AWGN). It can be multidimensional if 'slope' is a vector. If flat is True (in 1 dimesion) generate equally spaces samples instead of random ones. This is useful for the test phase.

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

noisy_2d_fx(*size_per_fx, dfx, sfx, center, noise_std=1*)

normalFeatureDataset(*perlabel=50, nlabels=2, nfeatures=4, nchunks=5, means=None, nonbogus_features=None, snr=1.0*)
Generate a dataset where each label is some normally distributed beastie around specified mean (0 if None).

snr is assuming that signal has std 1.0 so we just divide noise by snr

Probably it is a generalization of pureMultivariateSignal where means=[[0,1], [1,0]]

Specify either means or nonbogus_features so means get assigned accordingly

normalFeatureDataset__(*dataset=None, labels=None, nchunks=None, perlabel=50, activation_probability_steps=1, randomseed=None, randomvoxels=False*)
NOT FINISHED

pureMultivariateSignal(*patterns, signal2noise=1.5, chunks=None*)
Create a 2d dataset with a clear multivariate signal, but no univariate information.

```
%%%%%%%%%
% O % X %
%%%%%%%%%
% X % O %
%%%%%%%%%
```

sinModulated(*n_instances, n_features, flat=False, noise=0.40000000000000002*)
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

wr1996(*size=200*)
Generate '6d robot arm' dataset (Williams and Rasmussen 1996)

Was originally created in order to test the correctness of the implementation of kernel ARD. For full details see: http://www.gaussianprocess.org/gpml/code/matlab/doc/regression.html#ard

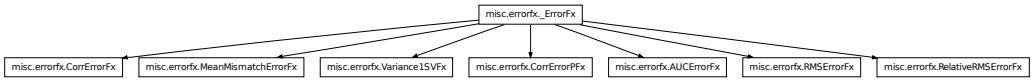
x_1 picked randomly in [-1.932, -0.453] x_2 picked randomly in [0.534, 3.142] r_1 = 2.0 r_2 = 1.3 f(x_1,x_2) = r_1 cos (x_1) + r_2 cos(x_1 + x_2) + N(0,0.0025) etc.

Expected relevances: ell_1 1.804377 ell_2 1.963956 ell_3 8.884361 ell_4 34.417657 ell_5 1081.610451 ell_6 375.445823 sigma_f 2.379139 sigma_n 0.050835

16.9.6 misc.errorfx

Module: misc.errorfx

Inheritance diagram for mvpa.misc.errorfx:



Error functions helpers.

PyMVPA can use arbitrary function which takes 2 arguments: predictions and targets and spits out a scalar value. Functions below are for the convinience, and they confirm the agreement that 'smaller' is 'better'

Classes

AUCErrorFx

class AUCErrorFx()
Bases: mvpa.misc.errorfx._ErrorFx

Computes the area under the ROC for the given the target and predicted to make the prediction.

MeanMismatchErrorFx

class MeanMismatchErrorFx()
Bases: mvpa.misc.errorfx._ErrorFx

Computes the percentage of mismatches between some target and some predicted values.

RMSErrorFx

class RMSErrorFx()
Bases: mvpa.misc.errorfx._ErrorFx

Computes the root mean squared error of some target and some predicted values.

RelativeRMSErrorFx

class RelativeRMSErrorFx()
Bases: mvpa.misc.errorfx._ErrorFx

Ratio between RMSE and root mean power of target output.

So it can be considered as a scaled RMSE – perfect reconstruction has values near 0, while no reconstruction has values around 1.0. Word of caution – it is not commutative, ie exchange of predicted and target might lead to completely different answers

Variance1SVFx

class Variance1SVFx()
Bases: mvpa.misc.errorfx._ErrorFx

Ratio of variance described by the first singular value component.

Of limited use – left for the sake of not wasting it

Functions

meanPowerFx(*data*)
Returns mean power

Similar to var but without demeaning

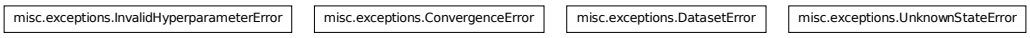
rootMeanPowerFx(*data*)
Returns root mean power

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

class ConvergenceError()
Bases: exceptions.Exception

Thrown if some algorithm does not converge to a solution.

DatasetError

class DatasetError(*msg=''*)
Bases: exceptions.Exception

Thrown if there is an internal problem with a Dataset.

ValueError exception is too generic to be used for any needed case, thus this one is created

InvalidHyperparameterError

class InvalidHyperparameterError()
Bases: exceptions.Exception

Generic exception to be raised when setting improper values as hyperparameters.

UnknownStateError

class UnknownStateError(*msg=''*)
Bases: exceptions.Exception

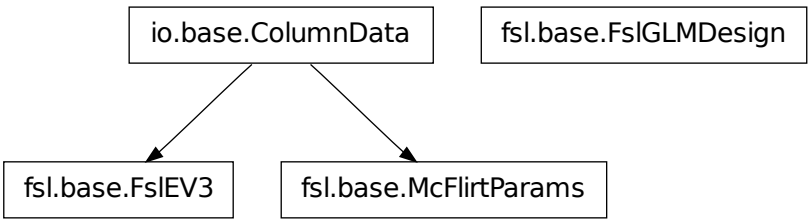
Thrown if the internal state of the class is not yet defined.

Classifiers and Algorithms classes might have properties, which are not defined prior to training or invocation has happened.

16.9.8 misc.fsl.base

Module: `misc.fsl.base`

Inheritance diagram for `mvpa.misc.fsl.base`:



Tiny snippets to interface with FSL easily.

Classes

`FslEV3`

class `FslEV3` (*source*)

Bases: `mvpa.misc.io.base.ColumnData`

IO helper to read FSL's EV3 files.

This is a three-column textfile format that is used to specify stimulation protocols for fMRI data analysis in FSL's FEAT module.

Data is always read as *float*.

Read and write FSL EV3 files.

- *source* (filename of an EV3 file) –

`durations`
durations

`getEV` (*evid*)
Returns a tuple of (onset time, simulus duration, intensity) for a certain EV.

`getNEVs` ()
Returns the number of EVs in the file.

`intensities`
intensities

`nevs`
Returns the number of EVs in the file.

`onsets`
onsets

`toEvents` (***kwargs*)
Convert into a list of *Event* instances.

- *kwargs* – Any keyword arugment 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 FslGLMDesign (*source*)
Bases: object

Load FSL GLM design matrices from file.

Be aware that such a desig matrix has its regressors in columns and the samples in its rows.

•*source* (filename) – Compressed files will be read as well, if their filename ends with '.gz'.

plot (*style='lines', **kwargs*)
Visualize the design matrix.

•*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 McFlirtParams (*source*)
Bases: mvpa.misc.io.base.ColumnData

Read and write McFlirt's motion estimation parameters from and to text files.

•*source* (str) – Filename of a parameter file.

plot ()
Produce a simple plot of the estimated translation and rotation parameters using.
You still need to can pylab.show() or pylab.savefig() if you want to see/get anything.

toarray ()
Returns the data as an array with six columns (same order as in file).

tofile (*filename*)
Write motion parameters to file.

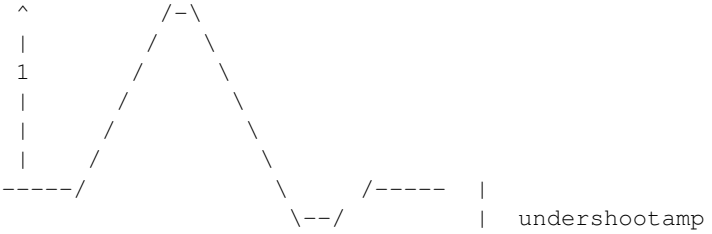
16.9.9 misc.fsl.flobs

Module: misc.fsl.flobs

Wrapper around FSLs halfcosbasis to generate HRF kernels

makeFlobs (*pre=0, rise=5, fall=5, undershoot=5, undershootamp=0.29999999999999999, nsamples=1, resolution=0.050000000000000003, 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.



```
|    |      |     |         |
|    |      |     |         |

 pre   rise  fall  undershoot
```

Parameters ‘pre’, ‘rise’, ‘fall’, ‘undershoot’ and ‘undershootamp’ can be specified as 2-tuples (min-max range for sampling) and single value (setting exact values – no sampling).

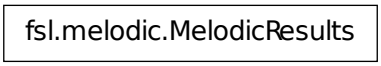
If ‘nsec’ is negative, the length of the samples is determined automatically to include the whole kernel function (until it returns to baseline). ‘nsec’ has to be an integer value and is set to the next greater integer value if it is not.

All parameters except for ‘nsamples’ and ‘nbasisfns’ are in seconds.

16.9.10 misc.fsl.melodic

Module: misc.fsl.melodic

Inheritance diagram for mvpa.misc.fsl.melodic:



Wrapper around the output of MELODIC (part of FSL)

MelodicResults

class 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

doubleGammaHRF(*t, A1=5.4000000000000004, W1=5.2000000000000002, K1=1.0, A2=10.800000000000001, W2=7.3499999999999996, K2=0.34999999999999998*)

Hemodynamic response function model.

The version is using two gamma functions (also see singleGammaHRF()).

- *t* (float) – Time.
- *A* (float) – Time to peak.
- *W* (float) – Full-width at half-maximum.
- *K* (float) – Scaling factor.

Parameters A, W and K exists individually for each of both gamma functions.

leastSqFit(*fx, params, y, x=None, **kwargs*)

Simple convenience wrapper around SciPy's optimize.leastsq.

The advantage of using this wrapper instead of optimize.leastsq directly is, that it automatically constructs an appropriate error function and easily deals with 2d data arrays, i.e. each column with multiple values for the same function argument (*x*-value).

- *fx* (functor) – Function to be fitted to the data. It has to take a vector with function arguments (*x*-values) as the first argument, followed by an arbitrary number of (to be fitted) parameters.
- *params* (sequence) – Sequence of start values for all to be fitted parameters. During fitting all parameters in this sequences are passed to the function in the order in which they appear in this sequence.
- *y* (1d or 2d array) – The data the function is fitted to. In the case of a 2d array, each column in the array is considered to be multiple observations or measurements of function values for the same *x*-value.
- *x* (Corresponding function arguments (*x*-values) for each datapoint, i.e.) – element in *y* or columns in *y', in the case of 'y* being a 2d array. If *x* is not provided it will be generated by *N.arange(m)*, where *m* is either the length of *y* or the number of columns in *y*, if *y* is a 2d array.
- ***kwargs* – All additonal keyword arguments are passed to *fx*.

Return type
: tuple

Returns
: i.e. 2-tuple with list of final (fitted) parameters of *fx* and an integer value indicating success or failure of the fitting procedure (see leastsq docs for more information).

singleGammaHRF(*t, A=5.4000000000000004, W=5.2000000000000002, K=1.0*)

Hemodynamic response function model.

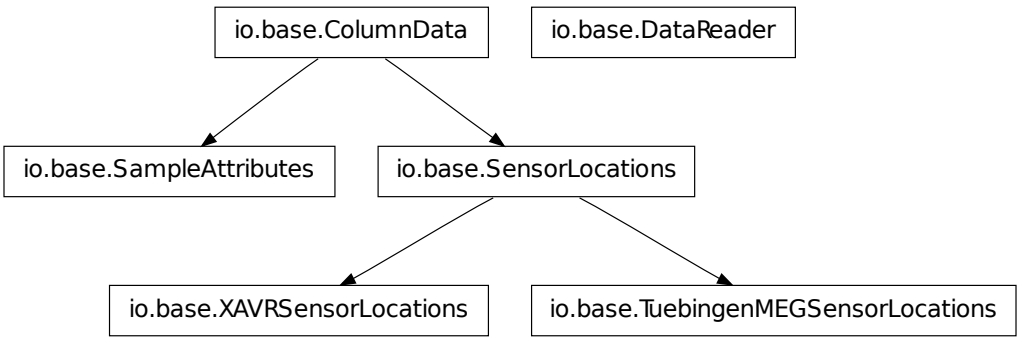
The version consists of a single gamma function (also see doubleGammaHRF()).

- *t* (float) – Time.
- *A* (float) – Time to peak.
- *W* (float) – Full-width at half-maximum.
- *K* (float) – Scaling factor.

16.9.12 misc.io.base

Module: `misc.io.base`

Inheritance diagram for `mvpa.misc.io.base`:



Some little helper for reading (and writing) common formats from and to disk.

Classes

ColumnData

class ColumnData (*source, header=True, sep=None, headersep=None, dtype=<type 'float'>, skiplines=0*)

Bases: dict

Read data that is stored in columns of text files.

All read data is available via a dictionary-like interface. If column headers are available, the column names serve as dictionary keys. If no header exists an articfical key is generated: str(number_of_column).

Splitting of text file lines is performed by the standard split() function (which gets passed the *sep* argument as separator string) and each element is converted into the desired datatype.

Because data is read into a dictionary no two columns can have the same name in the header! Each column is stored as a list in the dictionary.

Read data from file into a dictionary.

- *source* (basestring or dict) – If values is given as a string all data is read from the file and additonal keyword arguments can be sued to customize the read procedure. If a dictionary is passed a deepcopy is performed.
- *header* (bool or list of basestring) – Indicates whether the column names should be read from the first line (*header=True*). If *header=False* unique column names will be generated (see class docs). If *header* is a python list, it's content is used as column header names and its length has to match the number of columns in the file.
- *sep* (basestring or None) – Separator string. The actual meaning depends on the output format (see class docs).
- *headersep* (basestring or None) – Separator string used in the header. The actual meaning depends on the output format (see class docs).
- *dtype* (type or list(types)) – Desired datatype(s). Datatype per column get be specified by passing a list of types.
- *skiplines* (int) – Number of lines to skip at the beginning of the file.

getNColumns()
Returns the number of columns.

getNRows()
Returns the number of rows.

ncolumns
Returns the number of columns.

nrows
Returns the number of rows.

selectSamples(*selection*)
Return new ColumnData with selected samples

tofile(*filename, header=True, header_order=None, sep=' '*)
Write column data to a text file.

- *filename* (Think about it!) –
- *header* (If *True* a column header is written, using the column) – keys. If *False* no header is written.
- *header_order* (If it is a list of strings, they will be used instead) – of simply asking for the dictionary keys. However these strings must match the dictionary keys in number and identity. This argument type can be used to determine the order of the columns in the output file. The default value is *None*. In this case the columns will be in an arbitrary order.
- *sep* (String that is written as a separator between to data columns.) –

DataReader

class DataReader()
Bases: `object`

Base class for data readers.

Every subclass has to put all information into to variable:

self._data: ndarray
The data array has to have the samples separating dimension along the first axis.

self._props: dict
All other meaningful information has to be stored in a dictionary.

This class provides two methods (and associated properties) to retrieve this information.

Cheap init.

data
Data array

getData()
Return the data array.

getPropsAsDict()
Return the dictionary with the data properties.

props
Property dict

SampleAttributes

class SampleAttributes(*source, literallabels=False*)
Bases: `mvpa.misc.io.base.ColumnData`

Read and write PyMVPA sample attribute definitions from and to text files.

Read PyMVPA sample attributes from disk.

- *source* (filename of an atrribute file) –

getNSamples()
Returns the number of samples in the file.

nsamples
Returns the number of samples in the file.

tofile(*filename*)
Write sample attributes to a text file.

SensorLocations

class SensorLocations(**args, **kwargs*)

Bases: mvpa.misc.io.base.ColumnData

Base class for sensor location readers.

Each subclass should provide x, y, z coordinates via the *pos_x*, *pos_y*, and *pos_z* attrbibutes.

Axes should follow the following convention:

x-axis: left -> right y-axis: anterior -> posterior z-axis: superior -> inferior

Pass arguments to ColumnData.

locations()

Get the sensor locations as an array.

Return type
(nchannels x 3) array with coordinates in (x, y, z)

TuebingenMEGSensorLocations

class TuebingenMEGSensorLocations(*source*)

Bases: mvpa.misc.io.base.SensorLocations

Read sensor location definitions from a specific text file format.

File layout is assumed to be 7 columns:

1: sensor name 2: position on y-axis 3: position on x-axis 4: position on z-axis 5-7: same as 2-4, but for some outer surface thingie.

Note that x and y seem to be swapped, ie. y as defined by SensorLocations conventions seems to be first axis and followed by x.

Only inner surface coordinates are reported by *locations()*.

Read sensor locations from file.

•*source* (filename of an attribute file) –

XAVRSensorLocations

class XAVRSensorLocations(*source*)

Bases: mvpa.misc.io.base.SensorLocations

Read sensor location definitions from a specific text file format.

File layout is assumed to be 5 columns:

1.sensor name

2.some useless integer

3.position on x-axis

4.position on y-axis

5.position on z-axis

Read sensor locations from file.

•*source* (filename of an attribute file) –

Functions

design2labels (*columndata, baseline_label=0, func=<function <lambda> at 0x92c1764>*)

Helper to convert design matrix into a list of labels

Given a design, assign a single label to any given sample

TODO: fix description/naming

- *columndata* (ColumnData) – Attributes where each known will be considered as a separate explanatory variable (EV) in the design.
- *baseline_label* – What label to assign for samples where none of EVs was given a value
- *func* (functor) – Function which decides either a value should be considered

Output

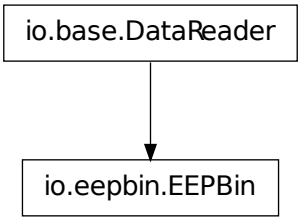
list of labels which are taken from column names in ColumnData and baseline_label

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

class EEPBin (*source*)

Bases: mvpa.misc.io.base.DataReader

Read-access to binary EEP files.

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

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

EEP files consist of a plain text header and a binary data block in a single file. The header starts with a line of the form

';%d %d %g %g' % (Nchannels, Nsamples, Ntrials, t0, dt)

where Nchannels, Nsamples, Ntrials are the numbers of channels, samples per trial and trials respectively. t0 is the time of the first sample of a trial relative to the stimulus onset and dt is the sampling interval.

The binary data block consists of single precision floats arranged in the following way:

```
<trial1,channel1,sample1>,<trial1,channel1,sample2>,...
<trial1,channel2,sample1>,<trial1,channel2,sample2>,...
.
<trial2,channel1,sample1>,<trial2,channel1,sample2>,...
<trial2,channel2,sample1>,<trial2,channel2,sample2>,...
```

Read EEP file and store header and data.

•*source* (str) – Filename.

channels
List of channel names

dt
Time difference between two adjacent samples

nchannels
Number of channels

nsamples
Number of trials/samples

ntimepoints
Number of data timepoints

t0
Relative start time of sampling interval

16.9.14 misc.io.hamster

Module: misc.io.hamster

Inheritance diagram for mvpa.misc.io.hamster:

Helper for simple storage facility via cPickle and optionally zlib

Hamster

class 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 burried hamster use Hamster(filename). Here is an example:

```
>>> 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

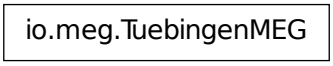
- *filename* (str) – Name of the target file. When writing to a compressed file the filename gets a '.gz' extension if not already specified. This is necessary as the constructor uses the extension to decide whether it loads from a compressed or uncompressed file.
- *compresslevel* ('auto' or int) – 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 helper for MEG datasets.

TuebingenMEG

class TuebingenMEG(*source*)
Bases: object

Reader for MEG data from line-based textfile format.

This class reads segmented MEG data from a textfile, which is created by converting the proprietary binary output files of a MEG device in Tuebingen (Germany) with an unkown tool.

The file format is line-based, i.e. all timepoints for all samples/trials are written in a single line. Each line is prefixed with an identifier (using a colon as the delimiter between identifier and data). Two lines have a special purpose. The first 'Sample Number' is a list of timepoint ids, similar to *range(ntimepoints)* for each sample/trial (all concatenated into one line. The second 'Time' contains the timing information for each timepoint (relative to stimulus onset), again for all trials concatenated into a single line.

All other lines contain various information (channels) recorded during the experiment. The meaning of some channels is unknown. Known ones are:

M*: MEG channels EEG*: EEG channels ADC*: Analog to digital converter output

Dataset properties are available from various class attributes. The *data* member provides all data from all channels (except for 'Sample Number' and 'Time') in a NumPy array (nsamples x nchannels x ntimepoints).

The reader supports uncompressed as well as gzipped input files (or other file-like objects).

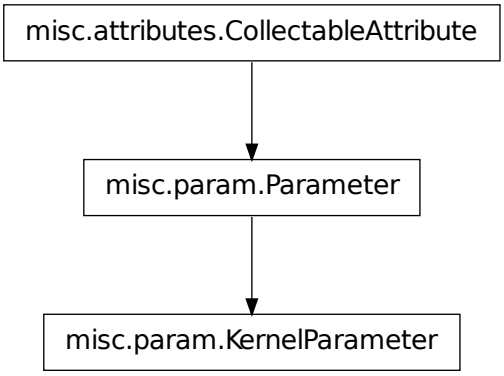
Reader MEG data from texfiles or file-like objects.

- *source* (str | file-like) – Strings are assumed to be filenames (with *.gz* suffix compressed), while all other object types are treated as file-like objects.

16.9.16 misc.param

Module: misc.param

Inheritance diagram for mvpa.misc.param:



Parameter representation

Classes

KernelParameter

class 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

class 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

cls.parameter1[:] = ...

or we wouldn't know that it was changed

Here is a list of possible property names:

min - minimum value max - maximum value step - increment/decrement stepsize

Specify a parameter by its default value and optionally an arbitrary number of additional parameters.

TODO: :Parameters: for Parameter

default

doc(*indent=' ', width=70*)
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

Functions

16.9.18 misc.plot.erp

Module: misc.plot.erp

Functions

16.9.19 misc.plot.mri

Module: misc.plot.mri

16.9.20 misc.plot.topo

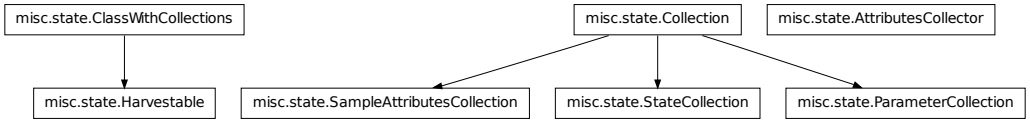
Module: misc.plot.topo

Functions

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 AttributesCollector(*name, bases, dict*)

Bases: `type`

Intended to collect and compose StateCollection for any child class of this metaclass

ClassWithCollections

class ClassWithCollections(*descr=None, **kwargs*)

Bases: `object`

Base class for objects which contain any known collection

Classes inherited from this class gain ability to access collections and their items as simple attributes. Access to collection items "internals" is done via <collection_name> attribute and interface of a corresponding *Collection*.

descr

Description of the object if any

reset()

Collection

class Collection(*items=None, owner=None, name=None*)

Bases: `object`

Container of some CollectableAttributes.

•*Public Access Functions*: *isKnown*

•*Access Implementors*: *_getListing*, *_getNames*

•*Mutators*: *__init__*

•*R/O Properties*: *listing*, *names*, *items*

XXX Seems to be not used and duplicating functionality: *_getListing* (thus *listing* property)

Initialize the Collection

•*items* (dict of CollectableAttribute's) – items to initialize with

•*owner* (object) – an object to which collection belongs

•*name* (basestring) – name of the collection (as seen in the owner, e.g. 'states')

add(*item*)

Add a new CollectableAttribute to the collection

•*item* (CollectableAttribute) – or of derived class. Must have 'name' assigned

TODO: we should make it stricter to don't add smth of
wrong type into Collection since it might lead to problems
Also we might convert to __setitem__

get (*index, default*)
Access the value by a given index.

Mimiquing regular dictionary behavior, if value cannot be obtained (i.e. if any exception is caught) return default value.

getvalue (*index*)
Returns the value by index

isKnown (*index*)
Returns *True* if state *index* is known at all

isSet (*index=None*)
If item (or any in the present or listed) was set

•*index* (None or basestring or list of basestring) – What items to check if they were set in the collection

items

listing
str(object) -> string

Return a nice string representation of the object. If the argument is a string, the return value is the same object.

name

names
Return ids for all registered state variables

owner

remove (*index*)
Remove item from the collection

reset (*index=None*)
Reset the state variable defined by *index*

setvalue (*index, value*)
Sets the value by index

whichSet ()
Return list of indexes which were set

Harvestable

class Harvestable (*harvest_attribs=None, copy_attribs='copy', **kwargs*)
Bases: mvpa.misc.state.ClassWithCollections

Classes inherited from this class intend to collect attributes within internal processing.

Subclassing Harvestable we gain ability to collect any internal data from the processing which is especially important if an object performs something in loop and discards some intermidiate possibly interesting results (like in case of CrossValidatedTransferError and states of the trained classifier or TransferError).

Note: Available state variables:

•*harvested*: Store specified attributes of classifiers at each split

(States 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

- *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

class ParameterCollection(*items=None, owner=None, name=None*)

Bases: mvpa.misc.state.Collection

Container of Parameters for a stateful object.

Initialize the Collection

- *items* (dict of CollectableAttribute's) – items to initialize with
- *owner* (object) – an object to which collection belongs
- *name* (basestring) – name of the collection (as seen in the owner, e.g. 'states')

resetvalue(*index, missingok=False*)

Reset all parameters to default values

SampleAttributesCollection

class SampleAttributesCollection(*items=None, owner=None, name=None*)

Bases: mvpa.misc.state.Collection

Container for data and attributes of samples (ie data/labels/chunks/...)

Initialize the Collection

- *items* (dict of CollectableAttribute's) – items to initialize with
- *owner* (object) – an object to which collection belongs
- *name* (basestring) – name of the collection (as seen in the owner, e.g. 'states')

StateCollection

class StateCollection(*items=None, owner=None*)

Bases: mvpa.misc.state.Collection

Container of StateVariables for a stateful object.

- *Public Access Functions*: *isKnown*, *isEnabled*, *isActive*
- *Access Implementors*: *_getListing*, *_getNames*, *_getEnabled*
- *Mutators*: *__init__*, *enable*, *disable*, *_setEnabled*
- *R/O Properties*: *listing*, *names*, *items*
- *R/W Properties*: *enabled*

Initialize the state variables of a derived class

- *items* (dict) – dictionary of states
- *owner* (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

- *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 DSMatrix(*data_vectors, metric='spearman'*)
Bases: object

DSMatrix allows for the creation of dissilimarity matrices using arbitrary distance metrics.

Initialize DSMatrix

- *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()

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:

Support function – little helpers in everyday life

Classes

Event

class Event (***kwargs*)

Bases: dict

Simple class to define properties of an event.

The class is basically a dictionary. Any properties can be 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.

asDescreteTime (*dt, storeoffset=False*)

Convert *onset* and *duration* information into descrete timepoints.

- *dt* (float) – Temporal distance between two timepoints in the same unit as *onset* and *duration*.
- *storeoffset* (bool) – If True, the temporal offset between original *onset* and descretized *onset* is stored as an additional item in *features*.

Return
A copy of the original *Event* with *onset* and optionally *duration* replaced by their corresponding descrete timepoint. The new onset will correspond to the timepoint just before or exactly at the original onset. The new duration will be the number of timepoints covering the event from the computed onset timepoint till the timepoint exactly at the end, or just after the event.
Note again, that the new values are expressed as #timepoint and not in their original unit!

Harvester

class Harvester (*source, calls, simplify_results=True*)
Bases: object

World domination helper: do whatever it is asked and accumulate results

- Might we need to deepcopy attributes values?
- Might we need to specify what attribs to copy and which just to bind?

Initialize

- *source* – Generator which produce food for the calls.
- *calls* (sequence of HarvesterCall instances) – Calls which are processed in the loop. All calls are processed in order of apperance in the sequence.
- *simplify_results* (bool) – Remove unecessary overhead in results if possible (nested lists and dictionaries).

HarvesterCall

class HarvesterCall (*call, attribs=None, argfilter=None, expand_args=True, copy_attribs=True*)
Bases: object

Initialize

- *expand_args* (bool) – Either to expand the output of looper into a list of arguments for call
- *attribs* (list of basestr) – What attributes of call to store and return later on?
- *copy_attribs* (bool) – Force copying values of attributes

MapOverlap

class MapOverlap (*overlap_threshold=1.0*)
Bases: object

Compute some overlap stats from a sequence of binary maps.

When called with a sequence of binary maps (e.g. lists or arrays) the fraction of mask elements that are non-zero in a customizable proportion of the maps is returned. By default this threshold is set to 1.0, i.e. such an element has to be non-zero in *all* maps.

Three additional maps (same size as original) are computed:

- overlap_map: binary map which is non-zero for each overlapping element.

  spread_map: binary map which is non-zero for each element that is non-zero in any map, but does not exceed the overlap threshold.
- ovstats_map: map of float with the raw elementwise fraction of overlap.

All maps are available via class members.

Nothing to be seen here.

Functions

RFEHistory2maps (*history*)
Convert history generated by RFE into the array of binary maps

Example:
history2maps(N.array([3,2,1,0]))

array([[1., 1., 1., 1.],
[1., 1., 1., 0.], [1., 1., 0., 0.], [1., 0., 0., 0.]])

getBreakPoints(*items, contiguous=True*)
Return a list of break points.

- *items* (iterable) – list of items, such as chunks
- *contiguous* (bool) – if *True* (default) then raise Value Error if items are not contiguous, i.e. a label occur in multiple contiguous sets

Raises
ValueError

Returns
list of indexes for every new set of items

getUniqueLengthNCombinations(*L, n=None, sort=True*)
Find all subsets of data

- *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_lt3

getUniqueLengthNCombinations(*L, n=None, sort=True*)
Find all subsets of data

- *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_lt3

idhash(*val*)
Craft unique id+hash for an object

indentDoc(*v*)
Given a *value* returns a string where each line is indented

Needed for a cleaner __repr__ output *v* - arbitrary

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

isSorted(*items*)
Check if listed items are in sorted order.

- *items* (iterable container) –

Returns
True if were sorted. Otherwise *False* + Warning

reuseAbsolutePath(*file1, file2, force=False*)
Use path to file1 as the path to file2 is no absolute path is given for file2

- *force* (bool) – if True, force it even if the file2 starts with /

transformWithBoxcar(*data, startpoints, boxlength, offset=0, fx=<function mean at 0x8768a04>*)
This function extracts boxcar windows from an array. Such a boxcar is defined by a starting point and the size of the window along the first axis of the array (*boxlength*). Afterwards a customizable function is applied to each boxcar individually (Default: averaging).

- *data* (array) – An array with an arbitrary number of dimensions.

- *startpoints* (sequence) – Boxcar startpoints as index along the first array axis
- *boxlength* (int) – Length of the boxcar window in #array elements
- *offset* (int) – Optional offset between the configured starting point and the actual beginning of the boxcar window.

Return type
array (len(startpoints) x data.shape[1:])

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 :)

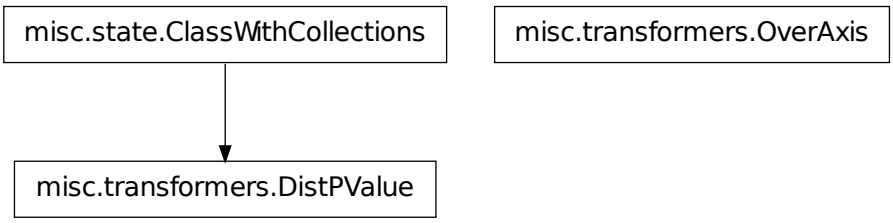
- *L* (list) – list of unique ids
- *n* (int) – grouping size

Adopted from Li Daobing http://code.activestate.com/recipes/190465/ (MIT license, according to activestate.com's policy)

16.9.24 misc.transformers

Module: misc.transformers

Inheritance diagram for mvpa.misc.transformers:



Simply functors that transform something.

Classes

DistPValue

class 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.

- *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 OverAxis (*transformer, axis=None*)

Bases: `object`

Helper to apply transformer over specific axis

Initialize transformer wrapper with an axis.

- *transformer* – A callable to be used
- *axis* (None or int) – If None – apply transformer across all the data. If some int – over that axis

Functions

Absolute (*x*)

Returns the elementwise absolute of any argument.

- *x* (scalar | sequence) –

FirstAxisMean (*x*)

Mean computed along the first axis.

FirstAxisSumNotZero (*x*)

Sum computed over first axis of whether the values are not equal to zero.

GrandMean (*x*)

Just what the name suggests.

Identity (*x*)

Return whatever it was called with.

L1Normed (*x, norm=1.0, reverse=False*)

Norm the values so that L_1 norm (sum|x|) becomes *norm*

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*.

OneMinus (*x*)

Returns elementwise '1 - x' of any argument.

RankOrder (*x, reverse=False*)

Rank-order by value. Highest gets 0

ReverseRankOrder (*x*)

Convinience functor

SecondAxisMaxOfAbs (*x*)
Max of absolute values along the 2nd axis

SecondAxisMean (*x*)
Mean across 2nd axis

•to combine multiple sensitivities to get sense about mean sensitivity across splits

SecondAxisSumOfAbs (*x*)
Sum of absolute values along the 2nd axis

•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

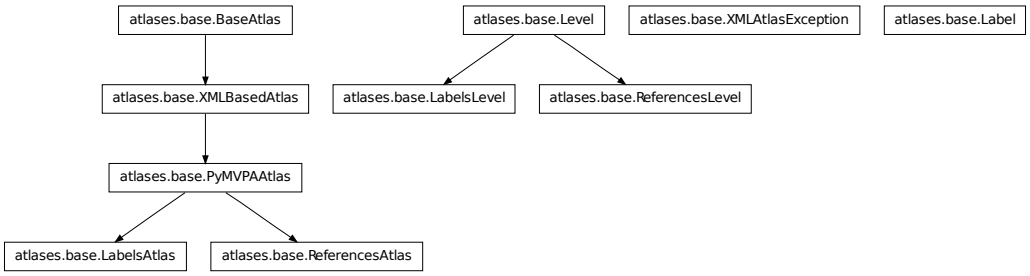
class VProperty (*fget=None, fset=None, fdel=None, doc=''*)
Bases: object

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 BaseAtlas()
Bases: object

Base class for the atlases.

Create an atlas object based on the... XXX

Label

class Label(*text, abbr=None, coord=(None, None, None), count=0, index=0*)
Bases: object

Represents a label. Just to bring all relevant information together

- *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

generateFromXML

index

text

LabelsAtlas

class LabelsAtlas(**args, **kwargs*)
Bases: mvpa.atlases.base.PyMVPAAtlas

Atlas which provides labels for the given coordinate

labelVoxel(*c, levels=None*)
Return labels for the given voxel at specified levels specified by index

LabelsLevel

class LabelsLevel (*description, index=None, labels=, []*)

Bases: mvpa.atlases.base.Level

Level of labels.

XXX extend

generateFromXML

index

labels

Level

class Level (*description*)

Bases: object

Represents a level. Just to bring all relevant information together

generateFromXML

Simple factory of levels

levelType

PyMVPAAtlas

class PyMVPAAtlas (**args, **kwargs*)

Bases: mvpa.atlases.base.XMLBasedAtlas

Base class for PyMVPA atlases, such as LabelsAtlas and ReferenceAtlas

Nlevels

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.

Initialize *ReferencesAtlas*

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 ReferencesLevel (*description, indexes=, []*)
Bases: mvpa.atlases.base.Level

Level which carries reference points

generateFromXML

indexes

XMLAtlasException

class XMLAtlasException (*msg=''*)
Bases: exceptions.Exception

Exception to be thrown if smth goes wrong dealing with XML based atlas

XMLBasedAtlas

class XMLBasedAtlas (*filename=None, resolution=None, query_voxel=False, coordT=None, levels=None*)
Bases: mvpa.atlases.base.BaseAtlas

- *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
- *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

so we first transform point into the voxel space

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

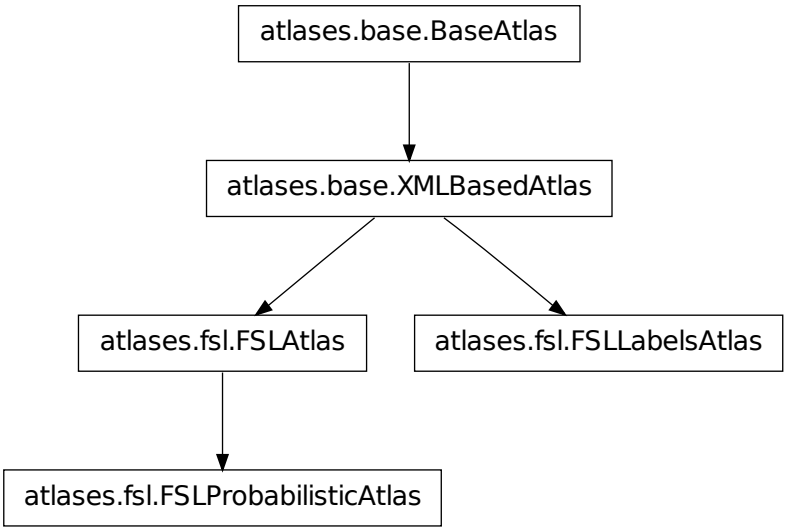
Function

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 FSLAtlas (**args, **kwargs*)

Bases: mvpa.atlases.base.XMLBasedAtlas

Base class for FSL atlases

- *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

FSLLabelsAtlas

class FSLLabelsAtlas (**args, **kwargs*)

Bases: mvpa.atlases.base.XMLBasedAtlas

FSLProbabilisticAtlas

class FSLProbabilisticAtlas (*thr=0.0, strategy='all', sort=True, *args, **kwargs*)

Bases: mvpa.atlases.fsl.FSLAtlas

- *thr* (float) – Value to threshold at

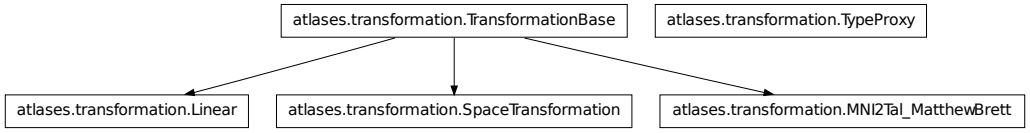
- *strategy* (basestring) – Possible values all - all entries above thr max - entry with maximal value
- *sort* (bool) – Either to sort entries for 'all' strategy according to probability

labelVoxel(*c, levels=None*)

16.9.28 atlases.transformation

Module: atlases.transformation

Inheritance diagram for mvpa.atlases.transformation:



Coordinate transformations

Classes

Linear

class Linear(transf=array([[1., 0., 0., 0.],()
class [0., 1., 0., 0.],()
class [0., 0., 1., 0.],()
class [0., 0., 0., 1.]), **kwargs)()

Bases: mvpa.atlases.transformation.TransformationBase

Simple linear transformation defined by a matrix

apply(*coord*)

MNI2Tal_MatthewBrett

class 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 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 TransformationBase (*previous=None*)
Basic class to describe a transformation. Pretty much an interface

apply (*coord*)

TypeProxy

class 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

MNI2Tal_Lancaster07FSL (**args, **kwargs*)

MNI2Tal_Lancaster07pooled (**args, **kwargs*)

MNI2Tal_MeyerLindenberg98 (**args, **kwargs*)
Due to Andreas Meyer-Lindenberg Taken from http://imaging.mrc-cbu.cam.ac.uk/imaging/MniTalairach

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 -finesearch 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 -finesearch 1 -v convert_xfm -inverse -omat mni2talairach.mat talairach2mni_fine1.mat

Tal2MNI_Lancaster07FSL (**args, **kwargs*)

Tal2MNI_Lancaster07pooled (**args, **kwargs*)

Tal2MNI_YOHflirt (**args, **kwargs*)
See MNI2Tal_YOHflirt doc

16.9.29 atlases.warehouse

Module: atlases.warehouse

Collection of the known atlases

Atlas (*filename=None, name=None, *args, **kwargs*)
A convinience factory for the atlases

MODULE INDEX

M

mvpa, 107
mvpa.algorithms.cvtranserror, 233
mvpa.atlases.base, 268
mvpa.atlases.fsl, 272
mvpa.atlases.transformation, 273
mvpa.atlases.warehouse, 274
mvpa.base, 235
mvpa.base.config, 236
mvpa.base.dochelpers, 237
mvpa.base.externals, 238
mvpa.base.verbosity, 238
mvpa.clfs.base, 149
mvpa.clfs.blr, 151
mvpa.clfs.distance, 152
mvpa.clfs.enet, 154
mvpa.clfs.glmnet, 156
mvpa.clfs.gpr, 159
mvpa.clfs.kernel, 161
mvpa.clfs.knn, 165
mvpa.clfs.lars, 167
mvpa.clfs.libsmlrc, 169
mvpa.clfs.libsmlrc.ctypes_helper, 169
mvpa.clfs.libsvmc.sens, 170
mvpa.clfs.libsvmc.svm, 171
mvpa.clfs.meta, 173
mvpa.clfs.model_selector, 182
mvpa.clfs.plr, 183
mvpa.clfs.ridge, 184
mvpa.clfs.sg.sens, 185
mvpa.clfs.sg.svm, 186
mvpa.clfs.smlr, 188
mvpa.clfs.stats, 190
mvpa.clfs.transerror, 195
mvpa.clfs.warehouse, 200
mvpa.datasets.base, 108
mvpa.datasets.channel, 114
mvpa.datasets.eep, 115
mvpa.datasets.event, 116
mvpa.datasets.mapped, 117
mvpa.datasets.masked, 119
mvpa.datasets.meta, 120
mvpa.datasets.miscfx, 121
mvpa.datasets.miscfx_sp, 121
mvpa.datasets.nifti, 122
mvpa.datasets.splitters, 124
mvpa.featsel.base, 224
mvpa.featsel.helpers, 226
mvpa.featsel.ifs, 230
mvpa.featsel.rfe, 232
mvpa.mappers.array, 131
mvpa.mappers.base, 132
mvpa.mappers.boxcar, 137
mvpa.mappers.ica, 138
mvpa.mappers.lle, 139
mvpa.mappers.mask, 140
mvpa.mappers.metric, 142
mvpa.mappers.pca, 143
mvpa.mappers.samplegroup, 144
mvpa.mappers.som, 145
mvpa.mappers.svd, 146
mvpa.mappers.wavelet, 147
mvpa.mappers.zscore, 148
mvpa.measures.anova, 201
mvpa.measures.base, 203
mvpa.measures.corrcoef, 209
mvpa.measures.corrstability, 210
mvpa.measures.ds, 211
mvpa.measures.glm, 212
mvpa.measures.irelief, 214
mvpa.measures.noiseperturbation, 217
mvpa.measures.pls, 219
mvpa.measures.searchlight, 220
mvpa.measures.splitmeasure, 222
mvpa.misc.args, 240
mvpa.misc.attributes, 240
mvpa.misc.bv.base, 242
mvpa.misc.cmdline, 243
mvpa.misc.data_generators, 243
mvpa.misc.errorfx, 245
mvpa.misc.exceptions, 246
mvpa.misc.fsl.base, 247
mvpa.misc.fsl.flobs, 248
mvpa.misc.fsl.melodic, 249
mvpa.misc.fx, 249
mvpa.misc.io.base, 251
mvpa.misc.io.eepbin, 254
mvpa.misc.io.hamster, 255
mvpa.misc.io.meg, 256
mvpa.misc.param, 257
mvpa.misc.state, 259

mvpa.misc.stats, 262
mvpa.misc.support, 263
mvpa.misc.transformers, 266
mvpa.misc.vproperty, 268

INDEX

A

abbr (mvpa.atlases.base.Label attribute), 269
absminDistance() (in module mvpa.clfs.distance), 152
Absolute() (in module mvpa.misc.transformers), 267
active (mvpa.base.verbosity.SetLogger attribute), 239
AdaptiveNormal (class in mvpa.clfs.stats), 191
AdaptiveNullDist (class in mvpa.clfs.stats), 191
AdaptiveRDist (class in mvpa.clfs.stats), 192
add() (mvpa.clfs.transerror.SummaryStatistics method), 199
add() (mvpa.misc.cmdline.OptionGroups method), 243
add() (mvpa.misc.state.Collection method), 259
AFNI, 8
aggregateFeatures() (in module mvpa.datasets.miscfx), 121
aggregateFeatures() (mvpa.datasets.base.Dataset method), 110
alternative build procedure, 13
analyzer (mvpa.measures.base.ProxyClassifierSensitivityAnalyzer attribute), 207
analyzers (mvpa.measures.base.CombinedFeaturewiseDatasetMeasure attribute), 204
anova, 35
API reference, 3
apply() (mvpa.atlases.transformation.Linear method), 273
apply() (mvpa.atlases.transformation.MNI2Tal_MatthewBrett method), 273
apply() (mvpa.atlases.transformation.TransformationBase method), 274
applyMapper() (mvpa.datasets.base.Dataset method), 110
applyMapper() (mvpa.datasets.meta.MetaDataset method), 120
asDescreteTime() (mvpa.misc.support.Event method), 263
asdict() (mvpa.misc.io.hamster.Hamster method), 256
asstring() (mvpa.clfs.transerror.ConfusionMatrix method), 197
asstring() (mvpa.clfs.transerror.RegressionStatistics method), 198
asstring() (mvpa.clfs.transerror.SummaryStatistics method), 199
Atlas() (in module mvpa.atlases.warehouse), 274
AttributesCollector (class in mvpa.misc.state), 259
AttributeWithUnique (class in mvpa.misc.attributes), 241
AUCErrorFx (class in mvpa.misc.errorfx), 245
aucs (mvpa.clfs.transerror.ROCCurve attribute), 198
autoNullDist() (in module mvpa.clfs.stats), 195

B

backports, 9
BaseAtlas (class in mvpa.atlases.base), 269
BestDetector (class in mvpa.featsel.helpers), 226
bestindex (mvpa.featsel.helpers.BestDetector attribute), 226
bias (mvpa.measures.base.StaticDatasetMeasure attribute), 209
biases (mvpa.clfs.smlr.SMLR attribute), 189
binary packages, 9
BinaryClassifier (class in mvpa.clfs.meta), 173
Block-averaging, 93
block-averaging, 89
BLR (class in mvpa.clfs.blr), 151
BoostedClassifier (class in mvpa.clfs.meta), 174
BoostedClassifierSensitivityAnalyzer (class in mvpa.measures.base), 203
BoxcarMapper (class in mvpa.mappers.boxcar), 137
BrainVoyagerRTC (class in mvpa.misc.bv.base), 242
build instructions, 12
building from source, 11
building on Windows, 13

C

C (mvpa.datasets.base.Dataset attribute), 109
cartesianDistance() (in module mvpa.clfs.distance), 152
cdf() (mvpa.clfs.stats.AdaptiveRDist method), 192
cdf() (mvpa.clfs.stats.FixedNullDist method), 193
cdf() (mvpa.clfs.stats.MCNullDist method), 194
cdf() (mvpa.clfs.stats.Nonparametric method), 194
cdf() (mvpa.clfs.stats.NullDist method), 195
cfg, 43
ChainMapper (class in mvpa.mappers.base), 132
changelog, 99
ChannelDataset (class in mvpa.datasets.channel), 114
channelids (mvpa.datasets.channel.ChannelDataset attribute), 114
channels (mvpa.misc.io.eepbin.EEPBin attribute), 255
checkRange() (in module mvpa.atlases.base), 271

chirpLinear() (in module mvpa.misc.data_generators), 243
chisquare() (in module mvpa.misc.stats), 262
Chunk, 93
chunks, 19
chunks (mvpa.datasets.base.Dataset attribute), 110
citation, 4
classifier, 23
Classifier (class in mvpa.clfs.base), 149
classifier error, 27
classifier weights, 35
ClassifierCombiner (class in mvpa.clfs.meta), 175
ClassifierError (class in mvpa.clfs.transerror), 195
ClassWithCollections (class in mvpa.misc.state), 259
clean() (mvpa.clfs.stats.MCNullDist method), 194
clf (mvpa.clfs.meta.ProxyClassifier attribute), 181
clf (mvpa.clfs.transerror.ClassifierError attribute), 196
clf (mvpa.measures.base.Sensitivity attribute), 208
clfs (mvpa.clfs.meta.BoostedClassifier attribute), 175
clone() (mvpa.clfs.base.Classifier method), 150
coarsenChunks() (in module mvpa.datasets.miscfx), 121
coarsenChunks() (mvpa.datasets.base.Dataset method), 110
CollectableAttribute (class in mvpa.misc.attributes), 241
Collection (class in mvpa.misc.state), 259
ColumnData (class in mvpa.misc.io.base), 251
combined_analyzer (mvpa.measures.base.BoostedClassifierSensitivityAnalyzer attribute), 203
CombinedClassifier (class in mvpa.clfs.meta), 175
CombinedFeatureSelection (class in mvpa.featsel.base), 224
CombinedFeaturewiseDatasetMeasure (class in mvpa.measures.base), 204
CombinedMapper (class in mvpa.mappers.base), 133
combiner (mvpa.algorithms.cvtranserror.CrossValidatedTransferError attribute), 234
combiner (mvpa.clfs.meta.CombinedClassifier attribute), 176
combiner (mvpa.featsel.base.CombinedFeatureSelection attribute), 224
combiner (mvpa.measures.base.FeaturewiseDatasetMeasure attribute), 206
CompoundOneWayAnova (class in mvpa.measures.anova), 201
compute() (mvpa.clfs.kernel.Kernel method), 162
compute() (mvpa.clfs.kernel.KernelConstant method), 162
compute() (mvpa.clfs.kernel.KernelExponential method), 162
compute() (mvpa.clfs.kernel.KernelLinear method), 163
compute() (mvpa.clfs.kernel.KernelMatern_3_2 method), 163
compute() (mvpa.clfs.kernel.KernelRationalQuadratic method), 164
compute() (mvpa.clfs.kernel.KernelSquaredExponential method), 164
compute() (mvpa.clfs.transerror.SummaryStatistics method), 199
compute_gradient() (mvpa.clfs.kernel.Kernel method), 162
compute_gradient_log_marginal_likelihood() (mvpa.clfs.gpr.GPR method), 160
compute_gradient_log_marginal_likelihood_logscale() (mvpa.clfs.gpr.GPR method), 160
compute_lml_gradient() (mvpa.clfs.kernel.Kernel method), 162
compute_lml_gradient() (mvpa.clfs.kernel.KernelConstant method), 162
compute_lml_gradient() (mvpa.clfs.kernel.KernelExponential method), 162
compute_lml_gradient() (mvpa.clfs.kernel.KernelLinear method), 163
compute_lml_gradient() (mvpa.clfs.kernel.KernelSquaredExponential method), 164
compute_lml_gradient_logscale() (mvpa.clfs.kernel.Kernel method), 162
compute_lml_gradient_logscale() (mvpa.clfs.kernel.KernelConstant method), 162
compute_lml_gradient_logscale() (mvpa.clfs.kernel.KernelExponential method), 163
compute_lml_gradient_logscale() (mvpa.clfs.kernel.KernelLinear method), 163
compute_lml_gradient_logscale() (mvpa.clfs.kernel.KernelSquaredExponential method), 165
compute_log_marginal_likelihood() (mvpa.clfs.blr.BLR method), 152
compute_log_marginal_likelihood() (mvpa.clfs.gpr.GPR method), 160
compute_M_H() (mvpa.measures.irelief.IterativeRelief method), 215
compute_M_H() (mvpa.measures.irelief.IterativeRelief_Devel method), 217
config file, 44
ConfigManager (class in mvpa.base.config), 236
configuration, 43
ConfusionBasedError (class in mvpa.clfs.transerror), 196
ConfusionMatrix (class in mvpa.clfs.transerror), 197
ConvergenceError (class in mvpa.misc.exceptions), 246
convertFeatureIds2FeatureMask() (mvpa.datasets.base.Dataset method), 110
convertFeatureMask2FeatureIds() (mvpa.datasets.base.Dataset method), 110

convertOutIds2InMask() (mvpa.mappers.mask.MaskMapper method), 140
convertOutIds2OutMask() (mvpa.mappers.mask.MaskMapper method), 140
coord (mvpa.atlases.base.Label attribute), 269
coordT (mvpa.atlases.base.XMLBasedAtlas attribute), 271
copy() (mvpa.datasets.base.Dataset method), 110
CorrCoef (class in mvpa.measures.corrcoef), 209
CorrStability (class in mvpa.measures.corrstability), 211
count (mvpa.atlases.base.Label attribute), 269
cross-validation, 28, 62, 65, 90
CrossValidatedTransferError (class in mvpa.algorithms.cvtranserror), 234
CustomSplitter (class in mvpa.datasets.splitters), 125

D

data (mvpa.misc.io.base.DataReader attribute), 252
data formats, 21
data splitting, 22
DataReader (class in mvpa.misc.io.base), 252
Dataset, 93
dataset, 18
Dataset (class in mvpa.datasets.base), 108
dataset attribute, 18
DatasetAttribute (class in mvpa.misc.attributes), 241
DatasetError (class in mvpa.misc.exceptions), 246
DatasetMeasure (class in mvpa.measures.base), 204
datasetmethod() (in module mvpa.datasets.base), 113
datasets (mvpa.datasets.meta.MetaDataset attribute), 120
Debian, 9
debug, 45, 47
Decoding, 93
default (mvpa.misc.param.Parameter attribute), 258
defineFeatureGroups() (mvpa.datasets.base.Dataset method), 110
DenseArrayMapper (class in mvpa.mappers.array), 131
descr (mvpa.misc.state.ClassWithCollections attribute), 259
DescreteMetric (class in mvpa.mappers.metric), 142
design2labels() (in module mvpa.misc.io.base), 254
detrend() (in module mvpa.datasets.miscfx_sp), 121
detrending, 48
development, 87
development snapshot, 12
disable() (mvpa.misc.state.StateCollection method), 262
discardOut() (mvpa.mappers.mask.MaskMapper method), 141
dissimilarity matrix, 65
distance (mvpa.atlases.base.ReferencesAtlas attribute), 270
DistPValue (class in mvpa.misc.transformers), 266
doc() (mvpa.misc.param.Parameter method), 258
doubleGammaHRF() (in module mvpa.misc.fx), 250
DSMatrix (class in mvpa.misc.stats), 262
DSMDatasetMeasure (class in mvpa.measures.ds), 212
dt (mvpa.datasets.channel.ChannelDataset attribute), 114
dt (mvpa.datasets.nifti.NiftiDataset attribute), 124
dt (mvpa.misc.io.eepbin.EEPBin attribute), 255
dumbFeatureBinaryDataset() (in module mvpa.misc.data_generators), 243
dumbFeatureDataset() (in module mvpa.misc.data_generators), 244
dump() (mvpa.misc.io.hamster.Hamster method), 256
durations (mvpa.misc.fsl.base.FslEV3 attribute), 247

E

EEPBin (class in mvpa.misc.io.eepbin), 254
EEPDataset (class in mvpa.datasets.eep), 115
ElementSelector (class in mvpa.featsel.helpers), 227
elementsize (mvpa.mappers.metric.DescreteMetric attribute), 142
enable() (mvpa.misc.attributes.StateVariable method), 242
enable() (mvpa.misc.state.StateCollection method), 262
enabled (mvpa.misc.state.StateCollection attribute), 262
ENET (class in mvpa.clfs.enet), 154
ENETWeights (class in mvpa.clfs.enet), 155
enhancedDocString() (in module mvpa.base.dochelpers), 237
environment variable
  MVPA_DEBUG, 47
  MVPA_DEBUG_METRICS, 48
  MVPA_MATPLOTLIB_BACKEND, 101
  MVPA_QUICKTEST, 103
  MVPA_SEED, 48
  MVPA_SVM_BACKEND, 102
  MVPA_TESTS_LABILE, 48, 102
  MVPA_TESTS_QUICK, 48, 103
  MVPA_VERBOSE, 46
  MVPA_VERBOSE_OUTPUT, 43
  MVPA_WARNINGS_BT, 47
  MVPA_WARNINGS_COUNT, 47
  MVPA_WARNINGS_SUPPRESS, 47
Epoch, 93
equalDefault (mvpa.misc.param.Parameter attribute), 258
ERNiftiDataset (class in mvpa.datasets.nifti), 122
error, 27
error (mvpa.clfs.transerror.ConfusionMatrix attribute), 197
error (mvpa.clfs.transerror.RegressionStatistics attribute), 198
error (mvpa.clfs.transerror.SummaryStatistics attribute), 199
errorfx (mvpa.clfs.transerror.TransferError attribute), 200
Event (class in mvpa.misc.support), 263
EventDataset (class in mvpa.datasets.event), 116

Example, 93
example, 49
example fMRI dataset, 51
examples, 3
exists() (in module mvpa.base.externals), 238
extend_args() (in module mvpa.clfs.libsmlrc.ctypes_helper), 169
extent (mvpa.atlases.base.XMLBasedAtlas attribute), 271

F

F-score, 35
Feature, 93
feature, 19, 20
feature selection, 21, 35, 37, 89
feature_ids, 89
feature_ids (mvpa.measures.base.Sensitivity attribute), 208
feature_selection (mvpa.clfs.meta.FeatureSelectionClassifier attribute), 177
feature_selections (mvpa.featsel.base.CombinedFeatureSelection attribute), 224
feature_selections (mvpa.featsel.base.FeatureSelectionPipeline attribute), 225
FeatureAttribute (class in mvpa.misc.attributes), 241
FeatureSelection, 39
FeatureSelection (class in mvpa.featsel.base), 224
FeatureSelectionClassifier, 39
FeatureSelectionClassifier (class in mvpa.clfs.meta), 176
FeatureSelectionClassifierSensitivityAnalyzer (class in mvpa.measures.base), 205
FeatureSelectionPipeline (class in mvpa.featsel.base), 225
FeaturewiseDatasetMeasure (class in mvpa.measures.base), 206
Fedora, 15
felements (mvpa.featsel.helpers.FractionTailSelector attribute), 228
filter_coord (mvpa.mappers.metric.DescreteMetric attribute), 142
FirstAxisMean() (in module mvpa.misc.transformers), 267
FirstAxisSumNotZero() (in module mvpa.misc.transformers), 267
fit (mvpa.clfs.stats.Nonparametric attribute), 194
fit() (mvpa.clfs.stats.AdaptiveNullDist method), 191
fit() (mvpa.clfs.stats.FixedNullDist method), 193
fit() (mvpa.clfs.stats.MCNullDist method), 194
fit() (mvpa.clfs.stats.NullDist method), 195
FixedErrorThresholdStopCrit (class in mvpa.featsel.helpers), 227
FixedNElementTailSelector (class in mvpa.featsel.helpers), 227
FixedNullDist (class in mvpa.clfs.stats), 192
fMRI, 93
forward mapping, 20
forward() (mvpa.mappers.base.ChainMapper method), 132
forward() (mvpa.mappers.base.CombinedMapper method), 133
forward() (mvpa.mappers.base.Mapper method), 134
forward() (mvpa.mappers.base.ProjectionMapper method), 136
forward() (mvpa.mappers.boxcar.BoxcarMapper method), 137
forward() (mvpa.mappers.lle.LLEMapper method), 139
forward() (mvpa.mappers.mask.MaskMapper method), 141
forward() (mvpa.mappers.samplegroup.SampleGroupMapper method), 145
forward() (mvpa.mappers.som.SimpleSOMMapper method), 146
FractionTailSelector (class in mvpa.featsel.helpers), 228
free software, 3
FSL, 8, 48
FSLAtlas (class in mvpa.atlases.fsl), 272
FslEV3 (class in mvpa.misc.fsl.base), 247
FslGLMDesign (class in mvpa.misc.fsl.base), 248
FSLLabelsAtlas (class in mvpa.atlases.fsl), 272
FSLProbabilisticAtlas (class in mvpa.atlases.fsl), 272
funcdata (mvpa.misc.fsl.melodic.MelodicResults attribute), 249

G

gaussian process regression, 30
generateFromXML (mvpa.atlases.base.Label attribute), 269
generateFromXML (mvpa.atlases.base.LabelsLevel attribute), 270
generateFromXML (mvpa.atlases.base.Level attribute), 270
generateFromXML (mvpa.atlases.base.ReferencesLevel attribute), 271
get() (mvpa.base.config.ConfigManager method), 237
get() (mvpa.misc.state.Collection method), 260
get_argtypes() (in module mvpa.clfs.libsmlrc.ctypes_helper), 169
getAsDType() (mvpa.base.config.ConfigManager method), 237
getboolean() (mvpa.base.config.ConfigManager method), 237
getBreakPoints() (in module mvpa.misc.support), 265
getData() (mvpa.misc.io.base.DataReader method), 252
getDt() (mvpa.datasets.nifti.NiftiDataset method), 124
getEV() (mvpa.misc.fsl.base.FslEV3 method), 247
getFullMatrix() (mvpa.misc.stats.DSMatrix method), 262
getInId() (mvpa.mappers.base.Mapper method), 134
getInId() (mvpa.mappers.mask.MaskMapper method), 141
getInId() (mvpa.mappers.som.SimpleSOMMapper method), 146

getInIds() (mvpa.mappers.mask.MaskMapper method), 141
getInSize() (mvpa.mappers.base.ChainMapper method), 132
getInSize() (mvpa.mappers.base.CombinedMapper method), 134
getInSize() (mvpa.mappers.base.Mapper method), 134
getInSize() (mvpa.mappers.base.ProjectionMapper method), 136
getInSize() (mvpa.mappers.boxcar.BoxcarMapper method), 137
getInSize() (mvpa.mappers.lle.LLEMapper method), 139
getInSize() (mvpa.mappers.mask.MaskMapper method), 141
getInSize() (mvpa.mappers.samplegroup.SampleGroupMapper method), 145
getInSize() (mvpa.mappers.som.SimpleSOMMapper method), 146
getLabels_map() (mvpa.clfs.transerror.ConfusionMatrix method), 197
getLabelsMap() (mvpa.datasets.base.Dataset method), 110
getMajorityVote() (mvpa.clfs.knn.kNN method), 166
getMask() (mvpa.mappers.mask.MaskMapper method), 141
getMetric() (mvpa.mappers.base.Mapper method), 135
getMetric() (mvpa.misc.stats.DSMatrix method), 262
getMVPattern() (in module mvpa.misc.data_generators), 244
getNColumns() (mvpa.misc.io.base.ColumnData method), 251
getNeighbor() (mvpa.mappers.base.ChainMapper method), 133
getNeighbor() (mvpa.mappers.base.CombinedMapper method), 134
getNeighbor() (mvpa.mappers.base.Mapper method), 135
getNeighbor() (mvpa.mappers.metric.Metric method), 143
getNeighborIn() (mvpa.mappers.base.Mapper method), 135
getNeighbors() (mvpa.mappers.base.Mapper method), 135
getNeighbors() (mvpa.mappers.metric.DescreteMetric method), 142
getNeighbors() (mvpa.mappers.metric.Metric method), 143
getNEVs() (mvpa.misc.fsl.base.FslEV3 method), 247
getNFeatures() (mvpa.datasets.base.Dataset method), 110
getNFeatures() (mvpa.datasets.meta.MetaDataset method), 120
getNiftiData() (in module mvpa.datasets.nifti), 124
getNiftiFromAnySource() (in module mvpa.datasets.nifti), 124
getNRows() (mvpa.misc.io.base.ColumnData method), 251
getNSamples() (mvpa.datasets.base.Dataset method), 110
getNSamples() (mvpa.datasets.meta.MetaDataset method), 120
getNSamples() (mvpa.misc.io.base.SampleAttributes method), 252
getOutId() (mvpa.mappers.mask.MaskMapper method), 141
getOutSize() (mvpa.mappers.base.ChainMapper method), 133
getOutSize() (mvpa.mappers.base.CombinedMapper method), 134
getOutSize() (mvpa.mappers.base.Mapper method), 135
getOutSize() (mvpa.mappers.base.ProjectionMapper method), 136
getOutSize() (mvpa.mappers.boxcar.BoxcarMapper method), 137
getOutSize() (mvpa.mappers.lle.LLEMapper method), 139
getOutSize() (mvpa.mappers.mask.MaskMapper method), 141
getOutSize() (mvpa.mappers.samplegroup.SampleGroupMapper method), 145
getOutSize() (mvpa.mappers.som.SimpleSOMMapper method), 146
getPropsAsDict() (mvpa.misc.io.base.DataReader method), 252
getRandomSamples() (mvpa.datasets.base.Dataset method), 111
getRandomSamples() (mvpa.datasets.meta.MetaDataset method), 120
getSamplesPerChunkLabel() (in module mvpa.datasets.miscfx), 121
getSamplesPerChunkLabel() (mvpa.datasets.base.Dataset method), 111
getSensitivityAnalyzer() (mvpa.clfs.base.Classifier method), 150
getSensitivityAnalyzer() (mvpa.clfs.enet.ENET method), 155
getSensitivityAnalyzer() (mvpa.clfs.gpr.GPR method), 160
getSensitivityAnalyzer() (mvpa.clfs.lars.LARS method), 168
getSensitivityAnalyzer() (mvpa.clfs.meta.BoostedClassifier method), 175
getSensitivityAnalyzer() (mvpa.clfs.meta.FeatureSelectionClassifier method), 177
getSensitivityAnalyzer() (mvpa.clfs.meta.MappedClassifier method), 178
getSensitivityAnalyzer() (mvpa.clfs.meta.ProxyClassifier method), 181
getSensitivityAnalyzer() (mvpa.clfs.meta.SplitClassifier method),

182
getSensitivityAnalyzer() (mvpa.clfs.smlr.SMLR method), 189
getTriangle() (mvpa.misc.stats.DSMatrix method), 262
getUniqueLengthNCombinations() (in module mvpa.misc.support), 265
getvalue() (mvpa.misc.state.Collection method), 260
getVectorForm() (mvpa.misc.stats.DSMatrix method), 262
getWeightedVote() (mvpa.clfs.knn.kNN method), 166
Git, 12, 87
Git repository, 12
GLM (class in mvpa.measures.glm), 213
GLMNET_C (class in mvpa.clfs.glmnet), 157
GLMNET_R (class in mvpa.clfs.glmnet), 158
GLMNETWeights (class in mvpa.clfs.glmnet), 156
GPR, 30, 60
GPR (class in mvpa.clfs.gpr), 159
GPRLinearWeights (class in mvpa.clfs.gpr), 161
gradient() (mvpa.clfs.kernel.KernelExponential method), 163
gradient() (mvpa.clfs.kernel.KernelMatern_3_2 method), 163
gradient() (mvpa.clfs.kernel.KernelRationalQuadratic method), 164
GrandMean() (in module mvpa.misc.transformers), 267
group_kwargs() (in module mvpa.misc.args), 240

H

HalfSplitter (class in mvpa.datasets.splitters), 126
Hamster (class in mvpa.misc.io.hamster), 255
handleDocString() (in module mvpa.base.dochelpers), 237
handlers (mvpa.base.verbosity.Logger attribute), 239
harvest_attribs (mvpa.misc.state.Harvestable attribute), 261
Harvestable (class in mvpa.misc.state), 260
Harvester (class in mvpa.misc.support), 264
HarvesterCall (class in mvpa.misc.support), 264
hasunique (mvpa.misc.attributes.AttributeWithUnique attribute), 241
history, 3
hlcuster, 8

I

I (mvpa.datasets.base.Dataset attribute), 109
ic (mvpa.misc.fsl.melodic.MelodicResults attribute), 249
ICAMapper (class in mvpa.mappers.ica), 138
icastats (mvpa.misc.fsl.melodic.MelodicResults attribute), 249
Identity() (in module mvpa.misc.transformers), 267
idhash (mvpa.datasets.base.Dataset attribute), 111
idhash() (in module mvpa.misc.support), 265
idsbychunks() (mvpa.datasets.base.Dataset method), 111
idsbylabels() (mvpa.datasets.base.Dataset method), 111
idsonboundaries() (mvpa.datasets.base.Dataset method), 111
IFS, 41
IFS (class in mvpa.featsel.ifs), 231
incremental feature search, 41
indent (mvpa.base.verbosity.LevelLogger attribute), 239
indentDoc() (in module mvpa.misc.support), 265
index (mvpa.atlases.base.Label attribute), 269
index (mvpa.atlases.base.LabelsLevel attribute), 270
index() (mvpa.datasets.base.Dataset method), 111
indexes (mvpa.atlases.base.ReferencesLevel attribute), 271
installation, 9
intensities (mvpa.misc.fsl.base.FslEV3 attribute), 247
internals (mvpa.clfs.warehouse.Warehouse attribute), 201
introduction, 18
InvalidHyperparameterError (class in mvpa.misc.exceptions), 246
invariant features, 89
IPython, 8
isActive() (mvpa.misc.state.StateCollection method), 262
isDefault (mvpa.misc.param.Parameter attribute), 258
isEnabled (mvpa.misc.attributes.StateVariable attribute), 242
isEnabled() (mvpa.misc.state.StateCollection method), 262
isInVolume() (in module mvpa.misc.support), 265
isKnown() (mvpa.misc.state.Collection method), 260
isSet (mvpa.misc.attributes.CollectableAttribute attribute), 241
isSet() (mvpa.misc.state.Collection method), 260
isSorted() (in module mvpa.misc.support), 265
isTrained() (mvpa.clfs.base.Classifier method), 150
isValidInId() (mvpa.mappers.base.Mapper method), 135
isValidInId() (mvpa.mappers.boxcar.BoxcarMapper method), 137
isValidInId() (mvpa.mappers.mask.MaskMapper method), 141
isValidOutId() (mvpa.mappers.base.Mapper method), 135
isValidOutId() (mvpa.mappers.boxcar.BoxcarMapper method), 137
isValidOutId() (mvpa.mappers.som.SimpleSOMMapper method), 146
items (mvpa.clfs.warehouse.Warehouse attribute), 201
items (mvpa.misc.state.Collection attribute), 260
IterativeRelief (class in mvpa.measures.irelief), 214
IterativeRelief_Devel (class in mvpa.measures.irelief), 216
IterativeReliefOnline (class in mvpa.measures.irelief), 215
IterativeReliefOnline_Devel (class in mvpa.measures.irelief), 216

K

K (mvpa.mappers.som.SimpleSOMMapper attribute), 146
k() (mvpa.measures.irelief.IterativeRelief method), 215
k-nearest-neighbour, 30
Kernel (class in mvpa.clfs.kernel), 162
kernel (mvpa.clfs.gpr.GPR attribute), 160
KernelConstant (class in mvpa.clfs.kernel), 162
KernelExponential (class in mvpa.clfs.kernel), 162
KernelLinear (class in mvpa.clfs.kernel), 163
KernelMatern_3_2 (class in mvpa.clfs.kernel), 163
KernelMatern_5_2 (class in mvpa.clfs.kernel), 164
KernelParameter (class in mvpa.misc.param), 257
KernelRationalQuadratic (class in mvpa.clfs.kernel), 164
KernelSquaredExponential (class in mvpa.clfs.kernel), 164
kNN, 30
kNN (class in mvpa.clfs.knn), 165

L

L (mvpa.datasets.base.Dataset attribute), 109
L1Normed() (in module mvpa.misc.transformers), 267
L2Normed() (in module mvpa.misc.transformers), 267
Label, 93
Label (class in mvpa.atlases.base), 269
labelPoint() (mvpa.atlases.base.XMLBasedAtlas method), 271
labels, 19
labels (mvpa.atlases.base.LabelsLevel attribute), 270
labels (mvpa.clfs.transerror.ClassifierError attribute), 196
labels (mvpa.clfs.transerror.ConfusionMatrix attribute), 197
labels (mvpa.datasets.base.Dataset attribute), 111
labels2chunks() (in module mvpa.misc.io.base), 254
labels_map (mvpa.clfs.transerror.ConfusionMatrix attribute), 197
labels_map (mvpa.datasets.base.Dataset attribute), 111
LabelsAtlas (class in mvpa.atlases.base), 269
LabelsLevel (class in mvpa.atlases.base), 270
labelVoxel() (mvpa.atlases.base.LabelsAtlas method), 269
labelVoxel() (mvpa.atlases.base.ReferencesAtlas method), 270
labelVoxel() (mvpa.atlases.fsl.FSLProbabilisticAtlas method), 273
LARS, 30
LARS (class in mvpa.clfs.lars), 167
LARSWeights (class in mvpa.clfs.lars), 168
least angle regression, 30
leastSqFit() (in module mvpa.misc.fx), 250
leave-one-out, 22
Level (class in mvpa.atlases.base), 270
level (mvpa.base.verbosity.LevelLogger attribute), 239
LevelLogger (class in mvpa.base.verbosity), 239
levels_dict (mvpa.atlases.base.XMLBasedAtlas attribute), 271
levelsListing() (mvpa.atlases.base.ReferencesAtlas method), 270
levelsListing() (mvpa.atlases.base.XMLBasedAtlas method), 271
levelType (mvpa.atlases.base.Level attribute), 270
lfprev (mvpa.base.verbosity.Logger attribute), 239
LIBSVM, 8, 12
license, 3
linear_awgn() (in module mvpa.misc.data_generators), 244
LinearSVMWeights (class in mvpa.clfs.libsvmc.sens), 170
LinearSVMWeights (class in mvpa.clfs.sg.sens), 185
listing (mvpa.misc.state.Collection attribute), 260
listing() (mvpa.clfs.warehouse.Warehouse method), 201
LLEMapper (class in mvpa.mappers.lle), 139
loadAtlas() (mvpa.atlases.base.XMLBasedAtlas method), 271
locations() (mvpa.misc.io.base.SensorLocations method), 253
Logger (class in mvpa.base.verbosity), 239
logistic regression, 30

M

MacOS X, 10, 11, 15
mahalanobisDistance() (in module mvpa.clfs.distance), 152
makeFlobs() (in module mvpa.misc.fsl.flobs), 248
manhattenDistance() (in module mvpa.clfs.distance), 152
map2Nifti() (mvpa.datasets.nifti.ERNiftiDataset method), 123
map2Nifti() (mvpa.datasets.nifti.NiftiDataset method), 124
mapForward() (mvpa.datasets.mapped.MappedDataset method), 118
MapOverlap (class in mvpa.misc.support), 264
MappedClassifier, 68
MappedClassifier (class in mvpa.clfs.meta), 177
MappedClassifierSensitivityAnalyzer (class in mvpa.measures.base), 206
MappedDataset (class in mvpa.datasets.mapped), 118
mapper, 20, 68, 78
Mapper (class in mvpa.mappers.base), 134
mapper (mvpa.clfs.meta.MappedClassifier attribute), 178
mapper (mvpa.datasets.mapped.MappedDataset attribute), 118
mapReverse() (mvpa.datasets.mapped.MappedDataset method), 118
mapReverse() (mvpa.datasets.meta.MetaDataset method), 120
mapSelfReverse() (mvpa.datasets.mapped.MappedDataset method), 118
mask (mvpa.mappers.mask.MaskMapper attribute), 141

maskclf (mvpa.clfs.meta.FeatureSelectionClassifier attribute), 177
MaskedDataset, 20
MaskedDataset (class in mvpa.datasets.masked), 119
MaskMapper (class in mvpa.mappers.mask), 140
Matlab, 83
matplotlib, 8
matrices (mvpa.clfs.transerror.ConfusionMatrix attribute), 197
matrix (mvpa.clfs.transerror.ConfusionMatrix attribute), 197
max_log_marginal_likelihood() (mvpa.clfs.model_selector.ModelSelector method), 182
maxcount (mvpa.base.WarningLog attribute), 235
MaximalVote (class in mvpa.clfs.meta), 178
McFlirtParams (class in mvpa.misc.fsl.base), 248
MCNullDist (class in mvpa.clfs.stats), 193
MeanMismatchErrorFx (class in mvpa.misc.errorfx), 245
meanPowerFx() (in module mvpa.misc.errorfx), 246
MeanPrediction (class in mvpa.clfs.meta), 179
measure, 33, 35–37
MelodicResults (class in mvpa.misc.fsl.melodic), 249
meta measures, 36
MetaDataset (class in mvpa.datasets.meta), 120
Metric (class in mvpa.mappers.metric), 143
metric (mvpa.mappers.base.Mapper attribute), 135
misc, 41
MNI2Tal_Lancaster07FSL() (in module mvpa.atlases.transformation), 274
MNI2Tal_Lancaster07pooled() (in module mvpa.atlases.transformation), 274
MNI2Tal_MatthewBrett (class in mvpa.atlases.transformation), 273
MNI2Tal_MeyerLindenberg98() (in module mvpa.atlases.transformation), 274
MNI2Tal_YOHflirt() (in module mvpa.atlases.transformation), 274
mode (mvpa.featsel.helpers.ElementSelector attribute), 227
model (mvpa.clfs.libsvmc.svm.SVM attribute), 173
ModelSelector (class in mvpa.clfs.model_selector), 182
modular architecture, 18
monte-carlo, 70
motion correction, 48
MulticlassClassifier (class in mvpa.clfs.meta), 179
multipleChunks() (in module mvpa.misc.data_generators), 244
MultiStopCrit (class in mvpa.featsel.helpers), 228
MVPA, 3, 93
mvpa (module), 107
MVPA toolbox for Matlab, 3, 85
mvpa.algorithms.cvtranserror (module), 233
mvpa.atlases.base (module), 268
mvpa.atlases.fsl (module), 272
mvpa.atlases.transformation (module), 273
mvpa.atlases.warehouse (module), 274
mvpa.base (module), 235
mvpa.base.config (module), 236
mvpa.base.dochelpers (module), 237
mvpa.base.externals (module), 238
mvpa.base.verbosity (module), 238
mvpa.clfs.base (module), 149
mvpa.clfs.blr (module), 151
mvpa.clfs.distance (module), 152
mvpa.clfs.enet (module), 154
mvpa.clfs.glmnet (module), 156
mvpa.clfs.gpr (module), 159
mvpa.clfs.kernel (module), 161
mvpa.clfs.knn (module), 165
mvpa.clfs.lars (module), 167
mvpa.clfs.libsmlrc (module), 169
mvpa.clfs.libsmlrc.ctypes_helper (module), 169
mvpa.clfs.libsvmc.sens (module), 170
mvpa.clfs.libsvmc.svm (module), 171
mvpa.clfs.meta (module), 173
mvpa.clfs.model_selector (module), 182
mvpa.clfs.plr (module), 183
mvpa.clfs.ridge (module), 184
mvpa.clfs.sg.sens (module), 185
mvpa.clfs.sg.svm (module), 186
mvpa.clfs.smlr (module), 188
mvpa.clfs.stats (module), 190
mvpa.clfs.transerror (module), 195
mvpa.clfs.warehouse (module), 200
mvpa.datasets.base (module), 108
mvpa.datasets.channel (module), 114
mvpa.datasets.eep (module), 115
mvpa.datasets.event (module), 116
mvpa.datasets.mapped (module), 117
mvpa.datasets.masked (module), 119
mvpa.datasets.meta (module), 120
mvpa.datasets.miscfx (module), 121
mvpa.datasets.miscfx_sp (module), 121
mvpa.datasets.nifti (module), 122
mvpa.datasets.splitters (module), 124
mvpa.featsel.base (module), 224
mvpa.featsel.helpers (module), 226
mvpa.featsel.ifs (module), 230
mvpa.featsel.rfe (module), 232
mvpa.mappers.array (module), 131
mvpa.mappers.base (module), 132
mvpa.mappers.boxcar (module), 137
mvpa.mappers.ica (module), 138
mvpa.mappers.lle (module), 139
mvpa.mappers.mask (module), 140
mvpa.mappers.metric (module), 142
mvpa.mappers.pca (module), 143
mvpa.mappers.samplegroup (module), 144
mvpa.mappers.som (module), 145
mvpa.mappers.svd (module), 146
mvpa.mappers.wavelet (module), 147
mvpa.mappers.zscore (module), 148
mvpa.measures.anova (module), 201
mvpa.measures.base (module), 203

mvpa.measures.corrcoef (module), 209
mvpa.measures.corrstability (module), 210
mvpa.measures.ds (module), 211
mvpa.measures.glm (module), 212
mvpa.measures.irelief (module), 214
mvpa.measures.noiseperturbation (module), 217
mvpa.measures.pls (module), 219
mvpa.measures.searchlight (module), 220
mvpa.measures.splitmeasure (module), 222
mvpa.misc.args (module), 240
mvpa.misc.attributes (module), 240
mvpa.misc.bv.base (module), 242
mvpa.misc.cmdline (module), 243
mvpa.misc.data_generators (module), 243
mvpa.misc.errorfx (module), 245
mvpa.misc.exceptions (module), 246
mvpa.misc.fsl.base (module), 247
mvpa.misc.fsl.flobs (module), 248
mvpa.misc.fsl.melodic (module), 249
mvpa.misc.fx (module), 249
mvpa.misc.io.base (module), 251
mvpa.misc.io.eepbin (module), 254
mvpa.misc.io.hamster (module), 255
mvpa.misc.io.meg (module), 256
mvpa.misc.param (module), 257
mvpa.misc.state (module), 259
mvpa.misc.stats (module), 262
mvpa.misc.support (module), 263
mvpa.misc.transformers (module), 266
mvpa.misc.vproperty (module), 268
MVPA_DEBUG, 47
MVPA_DEBUG_METRICS, 48
MVPA_MATPLOTLIB_BACKEND, 101
MVPA_QUICKTEST, 103
MVPA_SEED, 48
MVPA_SVM_BACKEND, 102
MVPA_TESTS_LABILE, 48, 102
MVPA_TESTS_QUICK, 48, 103
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), 241
name (mvpa.misc.state.Collection attribute), 260
names (mvpa.misc.state.Collection attribute), 260
nanmean() (in module mvpa.clfs.stats), 195
NBackHistoryStopCrit (class in mvpa.featsel.helpers), 228
nchannels (mvpa.misc.io.eepbin.EEPBin attribute), 255
ncolumns (mvpa.misc.io.base.ColumnData attribute), 251
nelements (mvpa.featsel.helpers.FixedNElementTailSelector attribute), 228
nevs (mvpa.misc.fsl.base.FslEV3 attribute), 247
nfeatures (mvpa.datasets.base.Dataset attribute), 111
nfeatures (mvpa.datasets.meta.MetaDataset attribute), 120
nfeatures (mvpa.mappers.base.Mapper attribute), 135
NFoldSplitter (class in mvpa.datasets.splitters), 127
NGroupSplitter (class in mvpa.datasets.splitters), 127
nic (mvpa.misc.fsl.melodic.MelodicResults attribute), 249
NIfTI, 4, 63
NiftiDataset (class in mvpa.datasets.nifti), 123
niftihdr (mvpa.datasets.nifti.ERNiftiDataset attribute), 123
niftihdr (mvpa.datasets.nifti.NiftiDataset attribute), 124
Nlevels (mvpa.atlases.base.PyMVPAAtlas attribute), 270
node (mvpa.mappers.lle.LLEMapper attribute), 139
noise perturbation, 36
NoisePerturbationSensitivity (class in mvpa.measures.noiseperturbation), 218
noisy_2d_fx() (in module mvpa.misc.data_generators), 244
NoneSplitter (class in mvpa.datasets.splitters), 128
Nonparametric (class in mvpa.clfs.stats), 194
normalFeatureDataset() (in module mvpa.misc.data_generators), 244
normalFeatureDataset__() (in module mvpa.misc.data_generators), 244
nrows (mvpa.misc.io.base.ColumnData attribute), 251
nsamples (mvpa.datasets.base.Dataset attribute), 111
nsamples (mvpa.datasets.meta.MetaDataset attribute), 120
nsamples (mvpa.misc.io.base.SampleAttributes attribute), 252
nsamples (mvpa.misc.io.eepbin.EEPBin attribute), 255
NStepsStopCrit (class in mvpa.featsel.helpers), 229
ntimepoints (mvpa.misc.io.eepbin.EEPBin attribute), 255
null_dist (mvpa.clfs.transerror.TransferError attribute), 200
null_dist (mvpa.measures.base.DatasetMeasure attribute), 205
NullDist (class in mvpa.clfs.stats), 194
NumPy, 8

O

O (mvpa.datasets.mapped.MappedDataset attribute), 118
OddEvenSplitter (class in mvpa.datasets.splitters), 129
OnceLogger (class in mvpa.base.verbosity), 239
OneMinus() (in module mvpa.misc.transformers), 267
oneMinusCorrelation() (in module mvpa.clfs.distance), 152
OneWayAnova (class in mvpa.measures.anova), 202
onsets (mvpa.misc.fsl.base.FslEV3 attribute), 247
OpenSUSE, 11, 14
optimization, 87
OptionGroups (class in mvpa.misc.cmdline), 243
Options (class in mvpa.misc.cmdline), 243

origids (mvpa.datasets.base.Dataset attribute), 111
origin (mvpa.atlases.base.XMLBasedAtlas attribute), 271
OverAxis (class in mvpa.misc.transformers), 267
owner (mvpa.misc.state.Collection attribute), 260

P

p() (mvpa.clfs.stats.NullDist method), 195
Parameter (class in mvpa.misc.param), 257
ParameterCollection (class in mvpa.misc.state), 261
path (mvpa.misc.fsl.melodic.MelodicResults attribute), 249
PCAMapper (class in mvpa.mappers.pca), 143
penalized logistic regression, 30
percentCorrect (mvpa.clfs.transerror.ConfusionMatrix attribute), 197
permutation, 70
permuteLabels() (mvpa.datasets.base.Dataset method), 111
permuteLabels() (mvpa.datasets.meta.MetaDataset method), 120
plot() (mvpa.clfs.transerror.ConfusionMatrix method), 197
plot() (mvpa.clfs.transerror.RegressionStatistics method), 198
plot() (mvpa.clfs.transerror.ROCCurve method), 198
plot() (mvpa.misc.fsl.base.FslGLMDesign method), 248
plot() (mvpa.misc.fsl.base.McFlirtParams method), 248
plotting example, 75
PLR (class in mvpa.clfs.plr), 183
PLS (class in mvpa.measures.pls), 219
pnorm_w_python() (in module mvpa.clfs.distance), 153
predict() (mvpa.clfs.base.Classifier method), 150
PredictionsCombiner (class in mvpa.clfs.meta), 180
print_registered() (mvpa.base.verbosity.SetLogger method), 239
printsetid (mvpa.base.verbosity.SetLogger attribute), 239
process_args() (in module mvpa.clfs.libsmlrc.ctypes_helper), 169
Processing object, 94
processing object, 23, 35
progress tracking, 45
proj (mvpa.mappers.base.ProjectionMapper attribute), 136
ProjectionMapper (class in mvpa.mappers.base), 135
props (mvpa.misc.io.base.DataReader attribute), 252
ProxyClassifier (class in mvpa.clfs.meta), 180
ProxyClassifierSensitivityAnalyzer (class in mvpa.measures.base), 207
pureMultivariateSignal() (in module mvpa.misc.data_generators), 244
PyMatlab, 4
PyMVPA poster, 4
PyMVPAAtlas (class in mvpa.atlases.base), 270
PyNIfTI, 8

R

R, 8
random number generation, 48
RangeElementSelector (class in mvpa.featsel.helpers), 229
RankOrder() (in module mvpa.misc.transformers), 267
rebuildSamples() (mvpa.datasets.meta.MetaDataset method), 120
recommended software, 8
recon (mvpa.mappers.base.ProjectionMapper attribute), 136
recursive feature selection, 39
redirecting output, 45
references, 94
ReferencesAtlas (class in mvpa.atlases.base), 270
ReferencesLevel (class in mvpa.atlases.base), 271
register() (mvpa.base.verbosity.SetLogger method), 239
registered (mvpa.base.verbosity.SetLogger attribute), 239
registered (mvpa.misc.io.hamster.Hamster attribute), 256
RegressionStatistics (class in mvpa.clfs.transerror), 198
RelativeRMSErrorFx (class in mvpa.misc.errorfx), 245
releases, 12
reload() (mvpa.base.config.ConfigManager method), 237
relvar_per_ic (mvpa.misc.fsl.melodic.MelodicResults attribute), 249
remove() (mvpa.misc.state.Collection method), 260
removeInvariantFeatures() (in module mvpa.datasets.miscfx), 121
removeInvariantFeatures() (mvpa.datasets.base.Dataset method), 112
repredict() (mvpa.clfs.base.Classifier method), 150
required software, 8
reset() (mvpa.clfs.kernel.Kernel method), 162
reset() (mvpa.clfs.kernel.KernelLinear method), 163
reset() (mvpa.clfs.kernel.KernelSquaredExponential method), 165
reset() (mvpa.clfs.transerror.SummaryStatistics method), 199
reset() (mvpa.misc.attributes.AttributeWithUnique method), 241
reset() (mvpa.misc.attributes.CollectableAttribute method), 241
reset() (mvpa.misc.attributes.StateVariable method), 242
reset() (mvpa.misc.state.ClassWithCollections method), 259
reset() (mvpa.misc.state.Collection method), 260
resetvalue() (mvpa.misc.param.Parameter method), 258
resetvalue() (mvpa.misc.state.ParameterCollection method), 261
retrain() (mvpa.clfs.base.Classifier method), 150
reuseAbsolutePath() (in module mvpa.misc.support), 265
reverse mapping, 20

reverse() (mvpa.mappers.base.ChainMapper method), 133
reverse() (mvpa.mappers.base.CombinedMapper method), 134
reverse() (mvpa.mappers.base.Mapper method), 135
reverse() (mvpa.mappers.base.ProjectionMapper method), 136
reverse() (mvpa.mappers.boxcar.BoxcarMapper method), 137
reverse() (mvpa.mappers.lle.LLEMapper method), 139
reverse() (mvpa.mappers.mask.MaskMapper method), 141
reverse() (mvpa.mappers.samplegroup.SampleGroupMapper method), 145
reverse() (mvpa.mappers.som.SimpleSOMMapper method), 146
ReverseRankOrder() (in module mvpa.misc.transformers), 267
review, 3
RFE, 39
RFE (class in mvpa.featsel.rfe), 232
RFEHistory2maps() (in module mvpa.misc.support), 264
ridge regression, 31
RidgeReg (class in mvpa.clfs.ridge), 184
RMSErrorFx (class in mvpa.misc.errorfx), 245
RNG, 48
ROCCurve (class in mvpa.clfs.transerror), 198
ROCs (mvpa.clfs.transerror.ROCCurve attribute), 198
rootMeanPowerFx() (in module mvpa.misc.errorfx), 246
RPy, 4, 8
rstUnderline() (in module mvpa.base.dochelpers), 237

S

S (mvpa.datasets.base.Dataset attribute), 109
Sample, 94
sample, 19, 20
sample attribute, 18
SampleAttribute (class in mvpa.misc.attributes), 242
SampleAttributes (class in mvpa.misc.io.base), 252
SampleAttributesCollection (class in mvpa.misc.state), 261
SampleGroupMapper (class in mvpa.mappers.samplegroup), 144
samples (mvpa.datasets.base.Dataset attribute), 112
samples_original (mvpa.datasets.mapped.MappedDataset attribute), 118
samplesperchunk (mvpa.datasets.base.Dataset attribute), 112
samplesperlabel (mvpa.datasets.base.Dataset attribute), 112
samplingrate (mvpa.datasets.channel.ChannelDataset attribute), 114
samplingrate (mvpa.datasets.nifti.NiftiDataset attribute), 124
save() (mvpa.base.config.ConfigManager method), 237
SciPy, 8
searchlight, 62, 63, 65
Searchlight (class in mvpa.measures.searchlight), 221
SecondAxisMaxOfAbs() (in module mvpa.misc.transformers), 267
SecondAxisMean() (in module mvpa.misc.transformers), 268
SecondAxisSumOfAbs() (in module mvpa.misc.transformers), 268
seed() (in module mvpa), 108
select() (mvpa.datasets.base.Dataset method), 112
selectFeatures() (mvpa.datasets.base.Dataset method), 112
selectFeatures() (mvpa.datasets.mapped.MappedDataset method), 118
selectFeatures() (mvpa.datasets.meta.MetaDataset method), 120
selectFeaturesByMask() (mvpa.datasets.masked.MaskedDataset method), 119
selectOut() (mvpa.mappers.base.ChainMapper method), 133
selectOut() (mvpa.mappers.base.CombinedMapper method), 134
selectOut() (mvpa.mappers.base.Mapper method), 135
selectOut() (mvpa.mappers.base.ProjectionMapper method), 136
selectOut() (mvpa.mappers.boxcar.BoxcarMapper method), 137
selectOut() (mvpa.mappers.mask.MaskMapper method), 141
selectOut() (mvpa.mappers.samplegroup.SampleGroupMapper method), 145
selectOut() (mvpa.mappers.som.SimpleSOMMapper method), 146
selectOut() (mvpa.mappers.svd.SVDMapper method), 147
selectSamples() (mvpa.datasets.base.Dataset method), 113
selectSamples() (mvpa.datasets.meta.MetaDataset method), 120
selectSamples() (mvpa.misc.io.base.ColumnData method), 252
self-organizing map, 78
Sensitivity, 94
sensitivity, 33, 35, 65, 90
Sensitivity (class in mvpa.measures.base), 207
Sensitivity Map, 94
sensitivity_analyzer (mvpa.featsel.base.SensitivityBasedFeatureSelection attribute), 226
SensitivityBasedFeatureSelection (class in mvpa.featsel.base), 225
SensorLocations (class in mvpa.misc.io.base), 253
set_hyperparameters() (mvpa.clfs.blr.BLR method), 152
set_hyperparameters() (mvpa.clfs.gpr.GPR method), 160
set_hyperparameters() (mvpa.clfs.kernel.KernelConstant method), 162

set_hyperparameters() (mvpa.clfs.kernel.KernelExponential method), 163
set_hyperparameters() (mvpa.clfs.kernel.KernelLinear method), 163
set_hyperparameters() (mvpa.clfs.kernel.KernelMatern_3_2 method), 163
set_hyperparameters() (mvpa.clfs.kernel.KernelRationalQuadratic method), 164
set_hyperparameters() (mvpa.clfs.kernel.KernelSquaredExponential method), 165
setActiveFromString() (mvpa.base.verbosity.SetLogger method), 239
setCoordT() (mvpa.atlases.base.XMLBasedAtlas method), 271
setDefault() (mvpa.misc.param.Parameter method), 258
setDistance() (mvpa.atlases.base.ReferencesAtlas method), 270
setLabels_map() (mvpa.clfs.transerror.ConfusionMatrix method), 198
setLabelsMap() (mvpa.datasets.base.Dataset method), 113
SetLogger (class in mvpa.base.verbosity), 239
setMetric() (mvpa.mappers.base.Mapper method), 135
setNPerLabel() (mvpa.datasets.splitters.Splitter method), 130
setReferenceLevel() (mvpa.atlases.base.ReferencesAtlas method), 270
sets (mvpa.clfs.transerror.SummaryStatistics attribute), 199
setSamplesDType() (mvpa.datasets.base.Dataset method), 113
setSamplesDType() (mvpa.datasets.meta.MetaDataset method), 121
setTestDataset() (mvpa.clfs.meta.FeatureSelectionClassifier method), 177
settings, 43
setvalue() (mvpa.misc.state.Collection method), 260
Shogun, 8
SimpleSOMMapper, 78
SimpleSOMMapper (class in mvpa.mappers.som), 145
singleGammaHRF() (in module mvpa.misc.fx), 250
singleOrPlural() (in module mvpa.base.dochelpers), 237
sinModulated() (in module mvpa.misc.data_generators), 244
SMLR, 31, 57
SMLR (class in mvpa.clfs.smlr), 188
SMLRWeights (class in mvpa.clfs.smlr), 189
smodes (mvpa.misc.fsl.melodic.MelodicResults attribute), 249
solve() (mvpa.clfs.model_selector.ModelSelector method), 182
SOM, 78
source package, 11
space (mvpa.atlases.base.PyMVPAAtlas attribute), 270
spaceFlavor (mvpa.atlases.base.PyMVPAAtlas attribute), 270
spaceT (mvpa.atlases.base.XMLBasedAtlas attribute), 271
SpaceTransformation (class in mvpa.atlases.transformation), 273
Sparse multinomial logistic regression, 31
Spatial Discrimination Map (SDM), 94
split_kwargs() (in module mvpa.misc.args), 240
splitcfg() (mvpa.datasets.splitters.Splitter method), 131
SplitClassifier, 40
SplitClassifier (class in mvpa.clfs.meta), 181
splitDataset() (mvpa.datasets.splitters.Splitter method), 130
SplitFeaturewiseDatasetMeasure (class in mvpa.measures.base), 208
SplitFeaturewiseMeasure (class in mvpa.measures.splitmeasure), 222
splitter, 22
Splitter (class in mvpa.datasets.splitters), 130
splitter (mvpa.algorithms.cvtranserror.CrossValidatedTransferError attribute), 234
splitter (mvpa.clfs.meta.SplitClassifier attribute), 182
splitting measures, 37
squared_euclidean_distance() (in module mvpa.clfs.distance), 153
StateCollection (class in mvpa.misc.state), 261
states, 26
StateVariable (class in mvpa.misc.attributes), 242
StaticDatasetMeasure (class in mvpa.measures.base), 208
Statistical Discrimination Map (SDM), 94
statistical testing, 70
stats (mvpa.clfs.transerror.SummaryStatistics attribute), 199
steps (mvpa.featsel.helpers.NBackHistoryStopCrit attribute), 229
steps (mvpa.featsel.helpers.NStepsStopCrit attribute), 229
stepwise_regression() (in module mvpa.clfs.libsmlrc), 169
StoppingCriterion (class in mvpa.featsel.helpers), 229
strategy (mvpa.datasets.splitters.Splitter attribute), 131
substractBaseline() (mvpa.datasets.channel.ChannelDataset method), 115
suggested software, 8
summaries (mvpa.clfs.transerror.SummaryStatistics attribute), 199
summary() (mvpa.clfs.base.Classifier method), 150
summary() (mvpa.clfs.libsvmc.svm.SVM method), 173
summary() (mvpa.clfs.meta.CombinedClassifier method), 176
summary() (mvpa.clfs.meta.ProxyClassifier method), 181
summary() (mvpa.datasets.base.Dataset method), 113
summary_labels() (mvpa.datasets.base.Dataset method), 113
SummaryStatistics (class in mvpa.clfs.transerror), 199
support vector machine, 31
sv (mvpa.mappers.svd.SVDMapper attribute), 147

SVD, 68
SVDMapper (class in mvpa.mappers.svd), 147
SVM, 31, 35, 57
SVM (class in mvpa.clfs.libsvmc.svm), 171
SVM (class in mvpa.clfs.sg.svm), 187
svm (mvpa.clfs.sg.svm.SVM attribute), 187
SWIG, 12
syntactic sugaring, 21

T

t0 (mvpa.datasets.channel.ChannelDataset attribute), 115
t0 (mvpa.misc.io.eepbin.EEPBin attribute), 255
table2string() (in module mvpa.base.dochelpers), 238
tail (mvpa.clfs.stats.NullDist attribute), 195
TailSelector (class in mvpa.featsel.helpers), 230
Tal2MNI_Lancaster07FSL() (in module mvpa.atlases.transformation), 274
Tal2MNI_Lancaster07pooled() (in module mvpa.atlases.transformation), 274
Tal2MNI_YOHflirt() (in module mvpa.atlases.transformation), 274
TaskPLS (class in mvpa.measures.pls), 220
testAllDependencies() (in module mvpa.base.externals), 238
testdataset (mvpa.clfs.meta.FeatureSelectionClassifier attribute), 177
text (mvpa.atlases.base.Label attribute), 269
textbook, 3
threshold (mvpa.featsel.helpers.FixedErrorThresholdStopCrit attribute), 227
Time-compression, 94
tmodes (mvpa.misc.fsl.melodic.MelodicResults attribute), 249
toarray() (mvpa.misc.bv.base.BrainVoyagerRTC method), 242
toarray() (mvpa.misc.fsl.base.McFlirtParams method), 248
toEvents() (mvpa.misc.fsl.base.FslEV3 method), 247
tofile() (mvpa.misc.fsl.base.FslEV3 method), 248
tofile() (mvpa.misc.fsl.base.McFlirtParams method), 248
tofile() (mvpa.misc.io.base.ColumnData method), 252
tofile() (mvpa.misc.io.base.SampleAttributes method), 252
toRealSpace() (mvpa.atlases.transformation.SpaceTransformation method), 273
toVoxelSpace() (mvpa.atlases.transformation.SpaceTransformation method), 273
tr (mvpa.misc.fsl.melodic.MelodicResults attribute), 249
train() (mvpa.clfs.base.Classifier method), 150
train() (mvpa.clfs.meta.PredictionsCombiner method), 180
train() (mvpa.mappers.base.ChainMapper method), 133
train() (mvpa.mappers.base.CombinedMapper method), 134
train() (mvpa.mappers.base.Mapper method), 135
train() (mvpa.mappers.base.ProjectionMapper method), 136
train() (mvpa.mappers.lle.LLEMapper method), 140
train() (mvpa.mappers.samplegroup.SampleGroupMapper method), 145
train() (mvpa.mappers.som.SimpleSOMMapper method), 146
traindataset (mvpa.clfs.sg.svm.SVM attribute), 187
trained (mvpa.clfs.base.Classifier attribute), 150
transerror (mvpa.algorithms.cvtranserror.CrossValidatedTransferError attribute), 234
transfer error, 27
TransferError (class in mvpa.clfs.transerror), 199
TransformationBase (class in mvpa.atlases.transformation), 274
transformer (mvpa.measures.base.DatasetMeasure attribute), 205
transformWithBoxcar() (in module mvpa.misc.support), 265
truevar_per_ic (mvpa.misc.fsl.melodic.MelodicResults attribute), 249
TScoredFeaturewiseMeasure (class in mvpa.measures.splitmeasure), 223
TuebingenMEG (class in mvpa.misc.io.meg), 256
TuebingenMEGSensorLocations (class in mvpa.misc.io.base), 253
TypeProxy (class in mvpa.atlases.transformation), 274

U

Ubuntu, 9
UC (mvpa.datasets.base.Dataset attribute), 109
UL (mvpa.datasets.base.Dataset attribute), 109
uniquechunks (mvpa.datasets.base.Dataset attribute), 113
uniquelabels (mvpa.datasets.base.Dataset attribute), 113
uniqueValues (mvpa.misc.attributes.AttributeWithUnique attribute), 241
unittests, 48
univariate, 35
UnknownStateError (class in mvpa.misc.exceptions), 246
untrain() (mvpa.clfs.base.Classifier method), 151
untrain() (mvpa.clfs.gpr.GPR method), 160
untrain() (mvpa.clfs.knn.kNN method), 166
untrain() (mvpa.clfs.libsvmc.svm.SVM method), 173
untrain() (mvpa.clfs.meta.BoostedClassifier method), 175
untrain() (mvpa.clfs.meta.ClassifierCombiner method), 175
untrain() (mvpa.clfs.meta.CombinedClassifier method), 176
untrain() (mvpa.clfs.meta.FeatureSelectionClassifier method), 177
untrain() (mvpa.clfs.meta.ProxyClassifier method), 181
untrain() (mvpa.clfs.sg.svm.SVM method), 187

V

validation data, 22
value (mvpa.misc.attributes.CollectableAttribute attribute), 241
value (mvpa.misc.param.Parameter attribute), 258
var (mvpa.mappers.pca.PCAMapper attribute), 144
Variance1SVFx (class in mvpa.misc.errorfx), 245
verbosity, 45, 46
version (mvpa.atlases.base.XMLBasedAtlas attribute), 271
voxdim (mvpa.atlases.base.XMLBasedAtlas attribute), 271
VProperty (class in mvpa.misc.vproperty), 268

W

Warehouse (class in mvpa.clfs.warehouse), 200
warning, 45, 47
WarningLog (class in mvpa.base), 235
WaveletPacketMapper (class in mvpa.mappers.wavelet), 147
WaveletTransformationMapper (class in mvpa.mappers.wavelet), 148
Weight Vector, 94
weights, 35
weights (mvpa.clfs.enet.ENET attribute), 155
weights (mvpa.clfs.lars.LARS attribute), 168
weights (mvpa.clfs.smlr.SMLR attribute), 189
where() (mvpa.datasets.base.Dataset method), 113
whichSet() (mvpa.misc.state.Collection method), 260
Windows, 9
Windows installer, 9
working data, 22
wr1996() (in module mvpa.misc.data_generators), 244

X

XAVRSensorLocations (class in mvpa.misc.io.base), 253
XMLAtlasException (class in mvpa.atlases.base), 271
XMLBasedAtlas (class in mvpa.atlases.base), 271
xuniqueCombinations() (in module mvpa.misc.support), 266

Z

zscore() (in module mvpa.datasets.miscfx), 121
zscore() (mvpa.datasets.base.Dataset method), 113
ZScoreMapper (class in mvpa.mappers.zscore), 148