PyMVPA Manual

Release 2.3.1

PyMVPA Authors

May 10, 2015

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. Two very good examples are: Pattern Recognition and Machine Learning by Christopher M. Bishop, and *The Elements of Statistical Learning: Data Mining, Inference, and Prediction* by Trevor Hastie, Robert Tibshirani, and Jerome Friedman (PDF was generously made available online free of charge).

There is a growing number of introductory papers about the application of machine learning algorithms to (f)MRI data. A very high-level overview about the basic principles is available in *Mur et al. (2009)*. A more detailed tutorial covering a wide variety of aspects is provided in *Pereira et al. (2009)*. 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 chap_modref. The chap_modref 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 *Example Analyses and Scripts* 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 How to cite PyMVPA

Below is a list of 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, and email use the reference so we could add it to our chap_whoisusingit page.

1.3.1 Peer-reviewed publications

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.
First paper introducing fMRI data analysis with PyMVPA.

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., Haxby, J. V., and Pollmann, S. (2010) Statistical learning analysis in neuroscience: aiming for transparency. Frontiers in Neuroscience. 4,1: 38-43
Focused review article emphasizing the role of transparency to facilitate adoption and evaluation of statistical learning techniques in neuroimaging research.

Haxby, J. V., Guntupalli, J. S., Connolly, A. C., Halchenko, Y. O., Conroy, B. R., Gobbini, M. I., Hanke, M. & Ramadge, P. J. (2011). A Common, High-Dimensional Model of the Representational Space in Human Ventral Temporal Cortex. *Neuron*, *72*, 404–416

The `Hyperalignment` paper demonstrating its application to fMRI data in rich perceptual (movie) and categorization (monkey-dog) experiments.

1.3.2 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*.

1.4 Authors and Contributors

The PyMVPA developers team currently consists of:

- Michael Hanke, University of Magdeburg, Germany
- Yaroslav O. Halchenko, Dartmouth College, USA
- Nikolaas N. Oosterhof, University of Trento, Italy

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

- Florian Baumgartner, University of Magdeburg, Germany
- Sven Buchholz, University of Magdeburg, Germany
- Andrew C. Connolly, Dartmouth College, USA
- Michael W. Cole, Washington University in St. Louis, USA
- Ceyhun Çakar
- Reka Daniel, Princeton University, USA
- Greg Detre, Princeton University, USA
- Matthias Ekman, Donders Institute, Netherlands
- Ingo Fründ, TU Berlin, Germany
- Christoph Gohlke, University of California, Irvine, USA
- Scott Gorlin, MIT, USA
- Satrajit Ghosh, MIT, USA
- Jyothi Swaroop Guntupalli, Dartmouth College, USA
- Valentin Haenel, TU Berlin, Germany
- Stephen José Hanson, Rutgers University, USA
- James V. Haxby, Dartmouth College, USA
- James M. Hughes, Dartmouth College, USA
- James Kyle, UCLA, USA
- Emanuele Olivetti, Fondazione Bruno Kessler, Italy
- Russell Poldrack, University of Texas, USA

- Stefan Pollmann, University of Magdeburg, Germany
- Geethapriya Raghavan, University of Texas Austin, USA
- Rajeev Raizada, Dartmouth College, USA
- Per B. Sederberg, Princeton University, USA
- Tiziano Zito, BCCN, Germany

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, MDP, scikit-learn, 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*.

1.5.1 Grant support

PyMVPA development was supported, in part, by the following research grants. This list includes grants funding development of specific algorithm implementations in PyMVPA, as well as grants supporting individuals to work on PyMVPA:

German Federal Ministry of Education and Research

- BMBF 01GQ11112

German federal state of Saxony-Anhalt

- Project: Center for Behavioral Brain Sciences

German Academic Exchange Service

- PPP-USA D/05/504/7

McDonnel Foundation

US National Institutes of Mental Health

- 5R01MH075706
- F32MH085433-01A1

US National Science Foundation

- NSF 1129764

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.

If you don't want to read this whole document, and you are on a Debian-based system, such as Ubuntu, all you need to know it:

```
sudo aptitude install python-mvpa2
```

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.x
: These days there should be little reason to use anything older than Python 2.7. However, if you are on a tight budget 2.6, or even 2.5 should work – maybe even 2.4 with (at least) ctypes 1.0.1. Python 3.X should work too, but none of the core developers are using it in production (yet), hence it should be considered as less tested.

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 some data transformation algorithms. 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.

NiBabel: access to NIfTI and other neuroimaging file formats
: PyMVPA provides a convenient wrapper for datasets stored in the NIfTI format, that internally uses NiBabel. If you don't need that, NiBabel 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 NiBabel installation.

2.1.3 Suggestions

The following list of software is, again, not required by PyMVPA, but these packages provide 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.

scikit-learn: large parts of its functionality
: PyMVPA can make use of pretty much any algorithm that implements the transformer or estimator and predictor API.

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

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

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

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

2.2 Installing Binary Packages

The easiest way to obtain PyMVPA is to use pre-built binary packages. Currently we provide such packages for the Debian/Ubuntu family, additional installers are provided by contributors (see below). If there are no binary packages for your operating system or platform yet, you can build PyMVPA from source. Please refer to *Building from Source* for more information.

Note:

If you have difficulties deploying PyMVPA or its dependencies, we recommend that you try the NeuroDebian virtual machine. With this virtual appliance you'll be able to deploy a fully functional computing environment, including PyMVPA, in a matter of minutes on any operating system.

2.2.1 Debian

PyMVPA is available as an official Debian package (`python-mvpa2` or `python-mvpa` for the previous stable release). The documentation is provided by the optional `python-mvpa2-doc` package. To install PyMVPA simply do:

```
sudo aptitude install python-mvpa2
```

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 NeuroDebian Repository. Please refer to NeuroDebian for installation instructions.

2.2.3 Windows

There are a few Python distributions for Windows. In theory all of them should work equally well. Christoph Gohlke runs a repository of unofficial Windows binaries for various scientific Python packages, including PyMVPA, that could ease deploying a PyMVPA installation on Windows significantly.

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 mvpa2
>>>
```

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 NiBabel. Download the corresponding installer from the website of the and install it. Verify that it works by importing the `nibabel` module in Python.

```
>>> import nibabel
>>>
```

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>.

Note:

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

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

MacPorts provides a universal binary package installer that is downloadable at http://www.macports.org/install.php

After downloading, simply mount the dmg image and double click `MacPorts.pkg`.

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

> http://www.debian.org/doc/manuals/reference/ch01.en.html#_customizing_bash

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

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

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

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

```
> $ sudo port install py25-pymvpa +scipy +nibabel +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 mvpa2
>>>
```

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

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 Obtain the Sources

Get the sources by cloning the Git repository on GitHub:

`git clone git://github.com/PyMVPA/PyMVPA.git`

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

If you are not familiar with Git or GitHub, visit the GitHub help pages for more information and tutorials.

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.

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 as `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 LIBSVM 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 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 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 mvpa2.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_literal.txt'))
>>> dataset = fmri_dataset(
...                samples=os.path.join(pymvpa_dataroot, 'bold.nii.gz'),
...                targets=attr.targets,
...                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 experiment run individually).

```
>>> poly_detrend(dataset, polyord=1, chunks_attr='chunks')
>>> zscore(dataset, param_est=('targets', ['rest']), dtype='float32')
```

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

```
>>> interesting = np.array([i in ['face', 'house'] for i in dataset.sa.targets])
>>> dataset = dataset[interesting]
```

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

```
>>> cv = CrossValidation(SMLR(), OddEvenPartitioner())
>>> 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 *Example Analyses and Scripts* 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
```

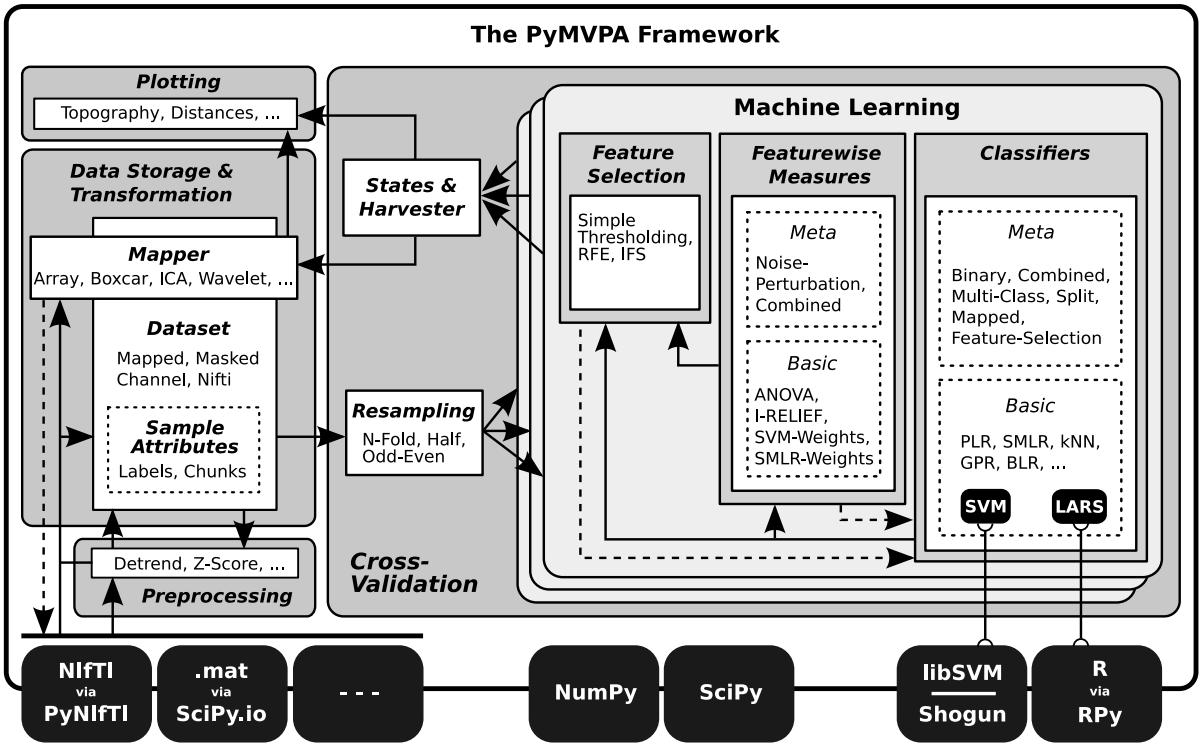
which would run the example shown in the first part of this section.

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

TUTORIAL INTRODUCTION TO PYMVPA

In this tutorial we are going to take a look at all major parts of PyMVPA, introduce the most important concepts, and explore particular functionality in real-life analysis examples. This tutorial also serves as basic course material for workshops on introductions to MVPA. Please contact us, if you are interested in hosting a PyMVPA workshop at your institution.

Please note that this tutorial is only concerned with aspects directly related to PyMVPA. It does **not** teach basic Python programming. If you are new to Python, we recommend that you take a look at the *Tutorial Prerequisites* for information about what you should know and how to obtain that knowledge.

Throughout the tutorial there will be little exercises with tasks that aim to deepen your understanding of a particular problem or to train important skills. However, even without a dedicated exercise you are advised to run the tutorial code interactively and explore code snippets beyond what is touched by the tutorial. Typically, only the most important aspects will be mentioned and each building block in PyMVPA can be used in more flexible ways than what is shown. Enjoy the ride.

Throughout the tutorial we will analyze real BOLD fMRI data. Therefore, to be able to run the code in this tutorial, you need to download the corresponding data from the PyMVPA website. Once downloaded, extract the tarball. On a NeuroDebian-enabled system, the tutorial data is also available from the `python-mvpa2-tutorialdata` package.

The `pymvpa2-tutorial` command (installed with PyMVPA) can be invoked in a console in order to launch a tutorial session. If the tutorial data was downloaded manually it may be necessary to specify the appropriate `--tutorial-data-path` option (see `pymvpa2-tutorial --help` for more information).

Virtually every Python script starts with some `import` statements that load functionality provided elsewhere. Likewise a tutorial session needs to import the PyMVPA packages and some little helpers we are going to use in the tutorial:

```
>>> from mvpa2.tutorial_suite import *
```

If this command succeeds without error, everything is ready to go.

If you want to prevent yourself from re-typing all code snippets into the terminal window, you might want to investigate IPython's `%cpaste` command, or use the **'IPython notebooks'_** provided for each tutorial part.

4.1 Tutorial Prerequisites

The PyMVPA tutorial assumes some basic knowledge about programming in Python. For a short self-assessment of your Python skills, please read the following questions. If you have an approximate answer to each of them, you can safely proceed to the tutorial. Otherwise, it is recommended that you take a look at the Python documentation resources listed under *Recommended Reading and Viewing*.

- Are you using *spaces* or *tabs* for indentation? Why is that important to know?
- What is the difference between `import numpy` and `from numpy import *`?
- What is the difference between a Python list and a tuple?
- What is the difference between a Python list and a Numpy `ndarray`?

- What is the difference between an *iterable* and a *generator* in Python?
- What is a *list comprehension*?
- What is a *callable*?
- What are `*args` and `**kwargs` usually used for?
- When would you use `?` or `??` in IPython?
- What is the difference between a *deep* copy and a *shallow* copy?
- What is a *derived class*?
- Is it always a problem whenever a Python *exception* is raised?

If you could not answer many questions: **Don't panic!** Python is known to be an easy-to-learn language. If you are already proficient in *any* other programming language, you can expect to be able to write fairly complex Python programs after a weekend of training.

4.1.1 What Do I Need To Get Python Running

PyMVPA code is compatible with Python 2.X series (more precisely >= 2.6). Python 3.x is supported as well, but not as widely used (yet), and many 3rd-party Python modules are still lacking Python 3 support. For now, we recommend Python 2.7 for production, but Python 2.6 should work equally well.

Any machine which has Python 2.X available can be used for PyMVPA-based processing (see chap_installation> on how to deploy PyMVPA on your system). Any GNU/Linux distribution already comes with Python by default. The Python website offers installers for Windows and MacOS X.

However, PyMVPA can make use of many additional software packages to enhance its functionality. Therefore it is preferable to use a Python distribution that offers are large variety of scientific Python packages. For Windows, Python(x,y) matches these requirements. For MacOS X, the MacPorts project offers a large variety of Python packages (including PyMVPA).

The *ideal* environment is, however, the Debian operating system. Debian offers the largest selection of free and open-source software in the world, and runs on almost any machine. Moreover, the NeuroDebian project provides Debian packages for a number of popular neuroscience software package, such as AFNI and FSL.

For those who just want to quickly try PyMVPA, or do not want to deal with installing multiple software package we recommend the NeuroDebian Virtual Machine. This is a virtual Debian installation that can be deployed on Linux, Windows, and MacOS X in a matter of minutes. It includes many Python packages, PyMVPA, and other neuroscience software (including AFNI and FSL).

4.1.2 Recommended Reading and Viewing

This section contains a recommended list of useful resources, ranging from basic Python programming to efficient scientific computing with Python.

Tutorial Introductions Into General Python Programming

http://en.wikibooks.org/wiki/Non-Programmer's_Tutorial_for_Python_2.6

> Basic from-scratch introduction into Python. This should give you the basics, even if you had *no* prior programming experience.

http://swaroopch.com/notes/python/

> From the author:
>
> > The aim is that if all you know about computers is how to save text files, then you can learn Python from this book. If you have previous programming experience, then you can also learn Python from this book.

We recommend reading the PDF version that is a lot better formatted.

http://www.diveintopython.net

A famous tutorial that served as the entry-point into Python for many people. However, it has a relatively steep learning curve, and also covers various topics which aren't in the focus of scientific computing.

http://docs.python.org/tutorial/

Written by the creator of Python itself, this is a more comprehensive, but also more compressed tutorial that can serve as a reference. Recommended as resource for people with basic programming experience in *some* language.

Scientific Computing In Python

Python itself is "just" a generic programming language. To employ Python for scientific computing, where a common analysis deals with vast amounts of numerical data, more specialized tools are needed – and are provided by the NumPy package. PyMVPA makes extensive use of NumPy data structures and functions, therefore we recommend you to get familiar with it.

http://www.scipy.org/Tentative_NumPy_Tutorial

Useful for a first glimpse at NumPy – the basis for scientific computing in Python.

http://mathesaurus.sourceforge.net/

Valuable resource for people coming from other languages and environments, such as Matlab. This pages offers cheat sheets with the equivalents of commands and expressions in various languages, including Matlab, R and Python.

http://www.tramy.us/numpybook.pdf

This is *the* comprehensive reference manual of the NumPy package. It gives answers to questions, yet to be asked.

Interactive Python Shell

To make interactive use of Python more enjoyable and productive, we suggest to explore an enhanced interactive environment for Python – IPython.

http://fperez.org/papers/ipython07_pe-gr_cise.pdf

An article from one of the authors of IPython in the *Computing in Science and Engineering* journal, describing goals and basic features of IPython.

http://showmedo.com/videotutorials/series?name=CnluURUTV

Video tutorials from Jeff Rush walking you through basic and advanced features of IPython. While doing that he also exposes basic constructs of Python, so you might like to watch this video whenever you already have basic programming experience with any programming language.

http://ipython.org/documentation.html

IPython documentation page which references additional materials, such as the main IPython documentation which extensively covers features of IPython.

Multivariate Analysis of Neuroimaging Data

There is a constantly growing number of interesting articles related to the field – visit *References* for an extended but not exhaustive list of related publications. For a quick introduction into the topic read *Pereira et. al. 2009*. For the generic reference on machine learning methods we would recommend a great text book *The Elements of Statistical Learning: Data Mining, Inference, and Prediction* by Trevor Hastie, Robert Tibshirani, and Jerome Friedman , PDF of which was generously made available online free of charge. For an overview of recent advances

in computational approaches for modeling and decoding of stimulus and cognitive spaces we recommend video recordings from Neural Computation 2011 Workshop at Dartmouth College.

4.2 Dataset basics and concepts

Note:

This tutorial part is also available for download as an IPython notebook: [ipynb]

A `Dataset` is the basic data container in PyMVPA. It serves as the primary form of data storage, but also as a common container for results returned by most algorithms. In this tutorial part we will take a look at what a dataset consists of, and how it works.

Most datasets in PyMVPA are represented as a two-dimensional array, where the first axis is the *sample*s axis, and the second axis represents the *feature*s of the samples. In the simplest case, a dataset only contains *data* that is a matrix of numerical values.

```
>>> from mvpa2.tutorial_suite import *
>>> data = [[  1,  1, -1],
...         [  2,  0,  0],
...         [  3,  1,  1],
...         [  4,  0, -1]]
>>> ds = Dataset(data)
>>> ds.shape
(4, 3)
>>> len(ds)
4
>>> ds.nfeatures
3
>>> ds.samples
array([[ 1,  1, -1],
       [ 2,  0,  0],
       [ 3,  1,  1],
       [ 4,  0, -1]])
```

In the above example, every row vector in the `data` matrix becomes an observation, a *sample*, in the dataset, and every column vector represents an individual variable, a *feature*. The concepts of samples and features are essential for a dataset, hence we take a closer look.

The dataset assumes that the first axis of the data is to be used to define individual samples. If the dataset is created using a one-dimensional vector it will therefore have as many samples as elements in the vector, and only one feature.

```
>>> one_d = [ 0, 1, 2, 3 ]
>>> one_ds = Dataset(one_d)
>>> one_ds.shape
(4, 1)
```

On the other hand, if a dataset is created from multi-dimensional data, only its second axis represents the features

```
>>> import numpy as np
>>> m_ds = Dataset(np.random.random((3, 4, 2, 3)))
>>> m_ds.shape
(3, 4, 2, 3)
>>> m_ds.nfeatures
4
```

In this case we have a dataset with three samples and four features, where each feature is a 2x3 matrix. In case somebody is wondering now why not simply treat each value in the data array as its own feature (yielding 24 features) – stay tuned, as this is going to be of importance later on.

4.2.1 Attributes

What we have seen so far does not really warrant the use of a dataset over a plain array or a matrix with samples. However, in the MVPA context we often need to know more about each sample than just the value of its features. For example, in order to train a supervised-learning algorithm to discriminate two classes of samples we need per-sample *target* values to label each sample with its respective class. Such information can then be used in order to, for example, split a dataset into specific groups of samples. For this type of auxiliary information a dataset can also contain collections of three types of attributes: a *sample attribute*, a *feature attribute*, and a *dataset attribute*.

For samples

Each *sample* in a dataset can have an arbitrary number of additional attributes. They are stored as vectors of the same length as the number of samples in a collection, and are accessible via the `sa` attribute. A collection is similar to a standard Python `dict`, and hence adding sample attributes works just like adding elements to a dictionary:

```
>>> ds.sa['some_attr'] = [ 0., 1, 1, 3 ]
>>> ds.sa.keys()
['some_attr']
```

However, sample attributes are not directly stored as plain data, but for various reasons as a so-called `Collectable` that in turn embeds a NumPy array with the actual attribute:

```
>>> type(ds.sa['some_attr'])
<class 'mvpa2.base.collections.ArrayCollectable'>
>>> ds.sa['some_attr'].value
array([ 0.,  1.,  1.,  3.])
```

This "complication" is done to be able to extend attributes with additional functionality that is often needed and can offer a significant speed-up of processing. For example, sample attributes carry a list of their unique values. This list is only computed once (upon first request) and can subsequently be accessed directly without repeated and expensive searches:

```
>>> ds.sa['some_attr'].unique
array([ 0.,  1.,  3.])
```

However, for most interactive uses of PyMVPA this type of access to attributes' `.value` is relatively cumbersome (too much typing), therefore collections support direct access by name:

```
>>> ds.sa.some_attr
array([ 0.,  1.,  1.,  3.])
```

Another purpose of the sample attribute collection is to preserve data integrity, by disallowing improper attributes:

```
>>> ds.sa['invalid'] = 4
Traceback (most recent call last):
  File "/usr/lib/python2.6/doctest.py", line 1253, in __run
    compileflags, 1) in test.globs
  File "<doctest tutorial_datasets.rst[20]>", line 1, in <module>
    ds.sa['invalid'] = 4
  File "/home/test/pymvpa/mvpa2/base/collections.py", line 459, in __setitem__
    value = ArrayCollectable(value)
  File "/home/test/pymvpa/mvpa2/base/collections.py", line 171, in __init__
    % self.__class__.__name__)
ValueError: ArrayCollectable only takes sequences as value.
Traceback (most recent call last):
  File "/usr/lib/python2.6/doctest.py", line 1253, in __run
    compileflags, 1) in test.globs
  File "<doctest tutorial_datasets.rst[20]>", line 1, in <module>
    ds.sa['invalid'] = 4
  File "/home/test/pymvpa/mvpa2/base/collections.py", line 459, in __setitem__
    value = ArrayCollectable(value)
```

```
  File "/home/test/pymvpa/mvpa2/base/collections.py", line 171, in __init__
    % self.__class__.__name__)
ValueError: ArrayCollectable only takes sequences as value.
```

```
>>> ds.sa['invalid'] = [ 1, 2, 3, 4, 5, 6 ]
Traceback (most recent call last):
  File "/usr/lib/python2.6/doctest.py", line 1253, in __run
    compileflags, 1) in test.globs
  File "<doctest tutorial_datasets.rst[21]>", line 1, in <module>
    ds.sa['invalid'] = [ 1, 2, 3, 4, 5, 6 ]
  File "/home/test/pymvpa/mvpa2/base/collections.py", line 468, in __setitem__
    str(self)))
ValueError: Collectable 'invalid' with length [6] does not match the required length [4] of colle
Traceback (most recent call last):
  File "/usr/lib/python2.6/doctest.py", line 1253, in __run
    compileflags, 1) in test.globs
  File "<doctest tutorial_datasets.rst[21]>", line 1, in <module>
    ds.sa['invalid'] = [ 1, 2, 3, 4, 5, 6 ]
  File "/home/test/pymvpa/mvpa2/base/collections.py", line 468, in __setitem__
    str(self)))
ValueError: Collectable 'invalid' with length [6] does not match the required length [4] of colle
```

But other than basic plausibility checks, no further constraints on values of samples attributes exist. As long as the length of the attribute vector matches the number of samples in the dataset, and the attributes values can be stored in a NumPy array, any value is allowed. Consequently, it is even possible to have n-dimensional arrays, not just vectors, as attributes – as long as their first axis matched the number of samples in a dataset. Moreover, it is perfectly possible and supported to store literal (non-numerical) attributes. It should also be noted that each attribute may have its own individual data type, hence it is possible to have literal and numeric attributes in the same dataset.

```
>>> ds.sa['literal'] = ['one', 'two', 'three', 'four']
>>> sorted(ds.sa.keys())
['literal', 'some_attr']
>>> for attr in ds.sa:
...     print "%s: %s" % (attr, ds.sa[attr].value.dtype.name)
literal: string40
some_attr: float64
```

For features

*Feature attribute*s are almost identical to *sample attribute*s, the *only* difference is that instead of having one attribute value per sample, feature attributes have one value per (guess what? ...) *feature*. Moreover, they are stored in a separate collection in the dataset that is called `fa`:

```
>>> ds.nfeatures
3
>>> ds.fa['my_fav'] = [0, 1, 0]
>>> ds.fa['responsible'] = ['me', 'you', 'nobody']
>>> sorted(ds.fa.keys())
['my_fav', 'responsible']
```

For the entire dataset

Lastly, there can be also attributes, not per-sample, or per-feature, but for the dataset as a whole: so called *dataset attribute*s. Both assigning such attributes and accessing them later on work in exactly the same way as for the other two types of attributes, except that dataset attributes are stored in their own collection which is accessible via the `a` property of the dataset. However, in contrast to sample and feature attribute, no constraints on the type or size are imposed – anything can be stored. Let's store a list with the names of all files in the current directory, just because we can:

```
>>> from glob import glob
>>> ds.a['pointless'] = glob("*")
>>> 'setup.py' in ds.a.pointless
True
```

4.2.2 Slicing, resampling, feature selection

At this point we can already construct a dataset from simple arrays and enrich it with an arbitrary number of additional attributes. But just having a dataset isn't enough. We often need to be able to select subsets of a dataset for further processing.

Slicing a dataset (i.e. selecting specific subsets) is very similar to slicing a NumPy array. It actually works *almost* identically. A dataset supports Python's `slice` syntax, but also selection by boolean masks and indices. The following three slicing operations result in equivalent output datasets, by always selecting every other samples in the dataset:

```
>>> # original
>>> ds.samples
array([[ 1,  1, -1],
       [ 2,  0,  0],
       [ 3,  1,  1],
       [ 4,  0, -1]])
>>>
>>> # Python-style slicing
>>> ds[::2].samples
array([[ 1,  1, -1],
       [ 3,  1,  1]])
>>>
>>> # Boolean mask array
>>> mask = np.array([True, False, True, False])
>>> ds[mask].samples
array([[ 1,  1, -1],
       [ 3,  1,  1]])
>>>
>>> # Slicing by index -- Python indexing start with 0 !!
>>> ds[[0, 2]].samples
array([[ 1,  1, -1],
       [ 3,  1,  1]])
```

Exercise

Search the NumPy documentation for the difference between "basic slicing" and "advanced indexing". The aspect of memory consumption, especially, applies to dataset slicing as well, and being aware of this fact might help to write more efficient analysis scripts. Which of the three slicing approaches above is the most memory-efficient? Which of the three slicing approaches above might lead to unexpected side-effects if the output dataset gets modified?

All three slicing-styles are equally applicable to the selection of feature subsets within a dataset. Remember, features are represented on the second axis of a dataset.

```
>>> ds[:, [1,2]].samples
array([[ 1, -1],
       [ 0,  0],
       [ 1,  1],
       [ 0, -1]])
```

By applying a selection by indices to the second axis, we can easily get the last two features of our example dataset. Please note that the `:` is supplied for the first axis slicing. This is the Python way to indicate *take everything along this axis*, thus including all samples.

As you can guess, it is also possible to select subsets of samples and features at the same time.

```
>>> subds = ds[[0,1], [0,2]]
>>> subds.samples
array([[ 1, -1],
       [ 2,  0]])
```

If you have prior experience with NumPy you might be confused now. What you might have expected is this:

```
>>> ds.samples[[0,1], [0,2]]
array([1, 0])
```

The above code applies the same slicing directly to the NumPy array of `.samples`, and the result is fundamentally different. For NumPy arrays this style of slicing allows selection of specific elements by their indices on each axis of an array. For PyMVPA's datasets this mode is not very useful, instead we typically want to select rows and columns, i.e. samples and features given by their indices.

Exercise

Try to select samples [0,1] and features [0,2] simultaneously using dataset slicing. Now apply the same slicing to the samples array itself (`ds.samples`) – make sure that the result doesn't surprise you and find a pure NumPy way to achieve similar selection.

One last interesting thing to look at, in the context of dataset slicing, are the attributes. What happens to them when a subset of samples and/or features is chosen? Our original dataset had both samples and feature attributes:

```
>>> print ds.sa.some_attr
[ 0.  1.  1.  3.]
>>> print ds.fa.responsible
['me' 'you' 'nobody']
```

Now let's look at what they became in the subset-dataset we previously created:

```
>>> print subds.sa.some_attr
[ 0.  1.]
>>> print subds.fa.responsible
['me' 'nobody']
```

We see that both attributes are still there and, moreover, also the corresponding subsets have been selected. It makes it convenient to select subsets of the dataset matching specific values of sample or feature attributes, or both:

```
>>> subds = ds[ds.sa.some_attr == 1., ds.fa.responsible == 'me']
>>> print subds.shape
(2, 1)
```

To simplify such selections based on the values of attributes, it is possible to specify the desired selection as a dictionary for either samples of features dimensions, where each key corresponds to an attribute name, and each value specifies a list of desired attribute values. Specifying multiple keys for either dimension can be used to obtain the intersection of matching elements:

```
>>> subds = ds[{'some_attr': [1., 0.], 'literal': ['two']}, {'responsible': ['me', 'you']}]
>>> print subds.sa.some_attr, subds.sa.literal, subds.fa.responsible
[ 1.] ['two'] ['me' 'you']
```

Exercise

Check the documentation of the `select()` method that can also be used to implement such a selection, but provides an additional argument `strict`. Modify the example above to select non-existing elements via `[]`, and compare to the result to the output of `select()` with `strict=False`.

4.2.3 Load fMRI data

Enough theoretical foreplay – let's look at a concrete example of loading an fMRI dataset. PyMVPA has several helper functions to load data from specialized formats, and the one for fMRI data is `fmri_dataset()`. The example dataset we are going to look at is a single subject from Haxby et al. (2001). For more convenience and less typing, we have a short cut for the path of the directory with the fMRI data: `tutorial_data_path`.

In the simplest case, we now let `fmri_dataset` do its job, by just pointing it to the fMRI data file. The data is stored as a NIfTI file that has all volumes of one experiment concatenated into a single file.

```
>>> bold_fname = os.path.join(tutorial_data_path, 'haxby2001', 'sub001',
...                           'BOLD', 'task001_run001', 'bold.nii.gz')
>>> ds = fmri_dataset(bold_fname)
>>> len(ds)
121
>>> ds.nfeatures
163840
>>> ds.shape
(121, 163840)
```

We can notice two things. First – *it worked!* Second, we obtained a two-dimensional dataset with 121 samples (these are volumes in the NIfTI file), and over 160k features (these are voxels in the volume). The voxels are represented as a one-dimensional vector, and it seems that they have lost their association with the 3D-voxel-space. However, this is not the case, as we will see later. PyMVPA represents data in this simple format to make it compatible with a vast range of generic algorithms that expect data to be a simple matrix.

We loaded all data from that NIfTI file, but usually we would be interested in a subset only, i.e. "brain voxels". `fmri_dataset` is capable of performing data masking. We just need to specify a mask image. Such a mask image is generated in pretty much any fMRI analysis pipeline – may it be a full-brain mask computed during skull-stripping, or an activation map from a functional localizer. We are going to use the original GLM-based localizer mask of ventral temporal cortex from Haxby et al. (2001). Let's reload the dataset:

```
>>> mask_fname = os.path.join(tutorial_data_path, 'haxby2001', 'sub001',
...                           'masks', 'orig', 'vt.nii.gz')
>>> ds = fmri_dataset(bold_fname, mask=mask_fname)
>>> len(ds)
121
>>> ds.nfeatures
577
```

As expected, we get the same number of samples, but now only 577 features – voxels corresponding to non-zero elements in the mask image. Now, let's explore this dataset a little further.

Exercise

Explore the dataset attribute collections. What kind of information do they contain?

Besides samples, the dataset offers a number of attributes that enhance the data with information that is present in the NIfTI image file header. Each sample has information about its volume index in the time series and the actual acquisition time (relative to the beginning of the file). Moreover, the original voxel index (sometimes referred to as `ijk`) for each feature is available too. Finally, the dataset also contains information about the dimensionality of the input volumes, voxel size, and any other NIfTI-specific information since it also includes a dump of the full NIfTI image header.

```
>>> ds.sa.time_indices[:5]
array([0, 1, 2, 3, 4])
>>> ds.sa.time_coords[:5]
array([  0. ,   2.5,   5. ,   7.5,  10. ])
>>> ds.fa.voxel_indices[:5]
array([[ 6, 23, 24],
       [ 7, 18, 25],
```

```
       [ 7, 18, 26],
       [ 7, 18, 27],
       [ 7, 19, 25]])
>>> ds.a.voxel_eldim
(3.5, 3.75, 3.75)
>>> ds.a.voxel_dim
(40, 64, 64)
>>> 'imghdr' in ds.a
True
```

In addition to all this information, the dataset also carries a key additional attribute: the *mapper*. A mapper is an important concept in PyMVPA, and hence has its own *tutorial chapter*.

```
>>> print ds.a.mapper
<Chain: <Flatten>-<StaticFeatureSelection>>
```

Having all these attributes being part of a dataset is often a useful thing to have, but in some cases (e.g. when it comes to efficiency, and/or very large datasets) one might want to have a leaner dataset with just the information that is really necessary. One way to achieve this, is to strip all unwanted attributes. The Dataset class' `copy()` method can help with that.

```
>>> stripped = ds.copy(deep=False, sa=['time_coords'], fa=[], a=[])
>>> print stripped
<Dataset: 121x577@int16, <sa: time_coords>>
```

We can see that all attributes besides `time_coords` have been filtered out. Setting the `deep` arguments to `False` causes the copy function to reuse the data from the source dataset to generate the new stripped one, without duplicating all data in memory – meaning both datasets now share the sample data and any change done to `ds` will also affect `stripped`.

4.2.4 Intermediate storage

Some data preprocessing can take a long time. One would rather prevent having to do it over and over again, and instead just store the preprocessed data into a file for subsequent analyses. PyMVPA offers functionality to store a large variety of objects, including datasets, into HDF5 files. A variant of this format is also used by recent versions of Matlab to store data.

For HDF5 support, PyMVPA depends on the h5py package. If it is available, any dataset can be saved to a file by simply calling `save()` with the desired filename.

```
>>> import tempfile, shutil
>>> # create a temporary directory
>>> tempdir = tempfile.mkdtemp()
>>> ds.save(os.path.join(tempdir, 'mydataset.hdf5'))
```

HDF5 is a flexible format that also supports, for example, data compression. To enable it, you can pass additional arguments to `save()` that are supported by h5py's `Group.create_dataset()`. Instead of using `save()` one can also use the `h5save()` function in a similar way. Saving the same dataset with maximum gzip-compression looks like this:

```
>>> ds.save(os.path.join(tempdir, 'mydataset.gzipped.hdf5'), compression=9)
>>> h5save(os.path.join(tempdir, 'mydataset.gzipped.hdf5'), ds, compression=9)
```

Loading datasets from a file is easy too. `h5load()` takes a filename as an argument and returns the stored dataset. Compressed data will be handled transparently.

```
>>> loaded = h5load(os.path.join(tempdir, 'mydataset.hdf5'))
>>> np.all(ds.samples == loaded.samples)
True
>>> # cleanup the temporary directory, and everything it includes
>>> shutil.rmtree(tempdir, ignore_errors=True)
```

Note that this type of dataset storage is not appropriate from long-term archival of data, as it relies on a stable software environment. For long-term storage, use other formats.

4.3 Getting data in shape

Note:

This tutorial part is also available for download as an IPython notebook: [ipynb]

In the tutorial part *Dataset basics and concepts* we have discovered a magic ingredient of datasets: a `Mapper`. Mappers are probably the most powerful concept in PyMVPA, and there is little one would do without them.

In general, a mapper is an algorithm that transforms data. This transformation can be as simple as selecting a subset of data, or as complex as a multi-stage preprocessing pipeline. Some transformations are reversible, others are not. Some are simple one-step computations, others are iterative algorithms that have to be trained on data before they can be used. In PyMVPA, all these transformations are `mappers`.

Note:

If you are an MDP-user you probably have realized the similarity of MDP's nodes and PyMVPA's mappers.

Let's create a dummy dataset (5 samples, 12 features). This time we will use a new method to create the dataset, the `dataset_wizard`. Here it is, fully equivalent to a regular constructor call (i.e. `Dataset`), but we will shortly see some nice convenience aspects.

```
>>> from mvpa2.tutorial_suite import *
>>> ds = dataset_wizard(np.ones((5, 12)))
>>> ds.shape
(5, 12)
```

Some datasets (such as the ones `fmri_dataset()` with a mask) contain mappers as a *dataset attribute* `.a.mapper`. However, not every dataset actually has a mapper. For example, the simple one we have just created doesn't have any:

```
>>> 'mapper' in ds.a
False
```

Now let's look at a very similar dataset that only differs in a tiny but a very important detail:

```
>>> ds = dataset_wizard(np.ones((5, 4, 3)))
>>> ds.shape
(5, 12)
>>> 'mapper' in ds.a
True
>>> print ds.a.mapper
<FlattenMapper>
```

We see that the resulting dataset looks identical to the one above, but this time it got created from a 3D samples array (i.e. five samples, where each is a 4x3 matrix). Somehow this 3D array got transformed into a 2D samples array in the dataset. This magic behavior is unveiled by observing that the dataset's mapper is a `FlattenMapper`.

The purpose of this mapper is precisely what we have just observed: reshaping data arrays into 2D. It does it by preserving the first axis (in PyMVPA datasets this is the axis that separates the samples) and concatenates all other axis into the second one.

Since mappers represent particular transformations they can also be seen as a protocol of what has been done. If we look at the dataset, we know that it had been flattened on the way from its origin to a samples array in a dataset. This feature can become really useful, if the processing become more complex. Let's look at a possible next step – selecting a subset of interesting features:

```
>>> myfavs = [1, 2, 8, 10]
>>> subds = ds[:, myfavs]
>>> subds.shape
(5, 4)
>>> 'mapper' in subds.a
True
>>> print subds.a.mapper
<Chain: <Flatten>-<StaticFeatureSelection>>
```

Now the situation has changed: *two* new mappers appeared in the dataset – a `ChainMapper` and a `StaticFeatureSelection`. The latter describes (and actually performs) the slicing operation we just made, while the former encapsulates the two mappers into a processing pipeline. We can see that the mapper chain represents the processing history of the dataset like a breadcrumb track.

As it has been mentioned, mappers not only can transform a single dataset, but can be fed with other data (as long as it is compatible with the mapper).

```
>>> fwdtest = np.arange(12).reshape(4,3)
>>> print fwdtest
[[ 0  1  2]
 [ 3  4  5]
 [ 6  7  8]
 [ 9 10 11]]
>>> fmapped = subds.a.mapper.forward1(fwdtest)
>>> fmapped.shape
(4,)
>>> print fmapped
[ 1  2  8 10]
```

Although `subds` has less features than our input data, forward mapping applies the same transformation that had been done to the dataset itself also to our test 4x3 array. The procedure yields a feature vector of the same shape as the one in `subds`. By looking at the forward-mapped data, we can verify that the correct features have been chosen.

4.3.1 Load real data

We have pretty much all the pieces to start a first analysis. We know how to load fMRI data from time series images, we know how to add and access attributes in a dataset, we know how to slice datasets, and we know that we can manipulate datasets with mappers.

Now our goal is to combine all these little pieces into the code that produces a dataset like the one used in the seminal work by *Haxby et al. (2001)* – a study were participants passively watched gray scale images of eight object categories in a block-design experiment. From the raw BOLD time series, of which we have the full 12 recording runs of the first subject, they computed:

> A *pattern of activation* for each stimulus category in each half of the data (split by odd vs. even runs; i.e. 16 samples), including the associated *sample attribute*s that are necessary to perform a cross-validated classification analysis of the data.

We have already seen how fMRI data can be loaded from NIfTI images, but this time we need more than just the EPI images. For a classification analysis we also need to associate each sample with a corresponding experimental condition, i.e. a class label, also sometimes called *target* value. Moreover, for a cross-validation procedure we also need to partition the full dataset into, presumably, independent *chunk*s. Independence is critical to achieve an unbiased estimate of the generalization performance of a classifier, i.e. its accuracy in predicting the correct class label for new data, unseen during training. So, where do we get this information from?

Both, target values and chunks are defined by the design of the experiment. In the simplest case the target value for an fMRI volume sample is the experiment condition that has been present/active while the volume has been acquired. However, there are more complicated scenarios which we will look at later on. Chunks of independent data correspond to what fMRI volumes are assumed to be independent. The properties of the MRI acquisition process cause subsequently acquired volumes to be *very* similar, hence they cannot be considered independent.

Ideally, the experiment is split into several acquisition sessions, where the sessions define the corresponding data chunks.

There are many ways to import this information into PyMVPA. The most simple one is to create a two-column text file that has the target value in the first column, and the chunk identifier in the second, with one line per volume in the NIfTI image.

```
>>> # directory that contains the data files
>>> data_path = os.path.join(tutorial_data_path, 'haxby2001')
>>> attr_fname = os.path.join(data_path, 'sub001',
...                           'BOLD', 'task001_run001', 'attributes.txt')
>>> attr = SampleAttributes(attr_fname)
>>> len(attr.targets)
121
>>> print np.unique(attr.targets)
['bottle' 'cat' 'chair' 'face' 'house' 'rest' 'scissors' 'scrambledpix'
 'shoe']
>>> len(attr.chunks)
121
>>> print np.unique(attr.chunks)
[ 0.]
```

`SampleAttributes` allows us to load this type of file, and access its content. We got 121 labels and chunk values, one for each volume. Moreover, we see that there are nine different conditions and all samples are associated with the same chunk. The attributes file for a different scan/run would increment the chunk value.

Now we can load the fMRI data, as we have done before – only loading voxels corresponding to a mask of ventral temporal cortex, and assign the samples attributes to the dataset. `fmri_dataset()` allows us to pass them directly:

```
>>> bold_fname = os.path.join(data_path,
...                           'sub001', 'BOLD', 'task001_run001', 'bold.nii.gz')
>>> mask_fname = os.path.join(tutorial_data_path, 'haxby2001',
...                           'sub001', 'masks', 'orig', 'vt.nii.gz')
>>> fds = fmri_dataset(samples=bold_fname,
...                    targets=attr.targets, chunks=attr.chunks,
...                    mask=mask_fname)
>>> fds.shape
(121, 577)
>>> print fds.sa
<SampleAttributesCollection: chunks,targets,time_coords,time_indices>
```

We got the dataset that we already know from the last part, but this time is also has information about chunks and targets.

4.3.2 More structure, less duplication of work

Although one could craft individual attribute files for each fMRI scan, doing so would be suboptimal. Typically, stimulation is not synchronous with fMRI volume sampling rate, hence timing information would be lost. Moreover, information on stimulation, or experiment design in general, is most likely available already in different form or shape.

To ease working with a broad range of datasets, PyMVPA comes with dedicated support for datasets following the specifications used by the openfmri.org data-sharing platform. These are simple guidelines for file name conventions and design specification that can be easily adopted for your own data.

Exercise

The tutorial data you are working with is following the openfmri.org scheme. Open the dataset folder and inspect the structure and content of the files with meta data. Notice, that it is possible to run a standard analysis using, for example, FSL's FEAT directly on this data in unmodified form.

Accessing such a dataset is done via a handler that simply needs to know where the dataset is stored on disk. This handler offers convenient access to basic information, such as the number of subjects, task descriptions, and other properties.

```
>>> dhandle = OpenFMRIDataset(data_path)
>>> dhandle.get_subj_ids()
[1]
>>> dhandle.get_task_descriptions()
{1: 'object viewing'}
```

More importantly, it supports access to information on experiment design:

```
>>> model = 1
>>> subj = 1
>>> run = 1
>>> events = dhandle.get_bold_run_model(model, subj, run)
>>> for ev in events[:2]:
...     print ev
{'task': 1, 'run': 1, 'onset_idx': 0, 'conset_idx': 0, 'onset': 15.0, 'intensity': 1, 'duration':
{'task': 1, 'run': 1, 'onset_idx': 1, 'conset_idx': 0, 'onset': 52.5, 'intensity': 1, 'duration':
```

As you can see, the stimulus design information is available in a list of standard Python dictionaries for each event. This includes onset and duration of the stimulation, as well as literal condition labels, and task descriptions.

With a utility function it is straightforward to convert such an event list into a sample attribute array like the one we have loaded from a file before. `events2sample_attr()` uses the sample acquisition time information stored in the dataset's `time_coords` sample attribute to match stimulation events to data samples.

```
>>> targets = events2sample_attr(events, fds.sa.time_coords,
...                              noinfolabel='rest', onset_shift=0.0)
>>> print np.unique([attr.targets[i] == targets[i] for i in range(len(targets))])
[ True]
>>> print np.unique(attr.targets)
['bottle' 'cat' 'chair' 'face' 'house' 'rest' 'scissors' 'scrambledpix'
 'shoe']
 >>> print len(fds), len(targets)
 121 121
```

Note, that the conversion of stimulation events to attribute arrays is a rather crude way of labeling fMRI data that only works well with block-design-like experiments. We will see other approaches later in this tutorial.

In addition to experiment design information, the dataset handler also offers convenient access to the actual BOLD fMRI data:

```
>>> task = 1
>>> fds = dhandle.get_bold_run_dataset(subj, task, run, mask=mask_fname)
>>> print fds
<Dataset: 121x577@int16, <sa: run,subj,task,time_coords,time_indices>, <fa: voxel_indices>, <a: in
```

The method `get_bold_run_dataset()` works the same way as `fmri_dataset()`, which we have seen before, and also supports the same arguments. However, instead of giving a custom filename, BOLD data is identified by subject, task, and acquisition run IDs.

Multi-session data

Many fMRI experiments involve multiple runs. Loading such data is best done in a loop. The following code snippet loads all available runs for the object viewing task from our example subject in the dataset.

```
>>> task = 1    # object viewing task
>>> model = 1   # image stimulus category model
>>> subj = 1
>>> run_datasets = []
```

```
>>> for run_id in dhandle.get_task_bold_run_ids(task)[subj]:
...     # load design info for this run
...     run_events = dhandle.get_bold_run_model(model, subj, run_id)
...     # load BOLD data for this run (with masking); add 0-based chunk ID
...     run_ds = dhandle.get_bold_run_dataset(subj, task, run_id,
...                                           chunks=run_id -1,
...                                           mask=mask_fname)
...     # convert event info into a sample attribute and assign as 'targets'
...     run_ds.sa['targets'] = events2sample_attr(
...                 run_events, run_ds.sa.time_coords, noinfolabel='rest')
...     # additional time series preprocessing can go here
...     run_datasets.append(run_ds)
>>> # this is PyMVPA's vstack() for merging samples from multiple datasets
>>> # a=0 indicates that the dataset attributes of the first run should be used
>>> # for the merged dataset
>>> fds = vstack(run_datasets, a=0)
```

Now it is a good time to obtain a `summary()` overview of the dataset: basic statistics, balance in number of samples among targets per chunk, etc.:

```
>>> print fds.summary()
Dataset: 1452x577@int16, <sa: chunks,run,subj,targets,task,time_coords,time_indices>, <fa: voxel_
stats: mean=1656.47 std=342.034 var=116988 min=352 max=2805

Counts of targets in each chunk:
  chunks\targets bottle cat chair face house rest scissors scrambledpix shoe
                   ---  ---  ---   ---  ---   ---     ---         ---      ---
         0           9    9    9     9    9    49       9           9        9
         1           9    9    9     9    9    49       9           9        9
         2           9    9    9     9    9    49       9           9        9
         3           9    9    9     9    9    49       9           9        9
         4           9    9    9     9    9    49       9           9        9
         5           9    9    9     9    9    49       9           9        9
         6           9    9    9     9    9    49       9           9        9
         7           9    9    9     9    9    49       9           9        9
         8           9    9    9     9    9    49       9           9        9
         9           9    9    9     9    9    49       9           9        9
        10           9    9    9     9    9    49       9           9        9
        11           9    9    9     9    9    49       9           9        9

Summary for targets across chunks
    targets  mean std min max #chunks
   bottle       9   0   9   9      12
     cat        9   0   9   9      12
    chair       9   0   9   9      12
    face        9   0   9   9      12
    house       9   0   9   9      12
    rest       49   0   49  49     12
  scissors      9   0   9   9      12
scrambledpix    9   0   9   9      12
    shoe        9   0   9   9      12

Summary for chunks across targets
  chunks mean   std min max #targets
    0    13.4 12.6  9   49      9
    1    13.4 12.6  9   49      9
    2    13.4 12.6  9   49      9
    3    13.4 12.6  9   49      9
    4    13.4 12.6  9   49      9
    5    13.4 12.6  9   49      9
    6    13.4 12.6  9   49      9
    7    13.4 12.6  9   49      9
    8    13.4 12.6  9   49      9
```

```
    9    13.4 12.6  9   49     9
   10    13.4 12.6  9   49     9
   11    13.4 12.6  9   49     9
Sequence statistics for 1452 entries from set ['bottle', 'cat', 'chair', 'face', 'house', 'rest',
Counter-balance table for orders up to 2:
Targets/Order O1                            |  O2                            |
   bottle:    96  0  0  0  0  12  0  0  0  |  84  0  0  0  0  24  0  0  0  |
     cat:      0 96  0  0  0  12  0  0  0  |   0 84  0  0  0  24  0  0  0  |
    chair:     0  0 96  0  0  12  0  0  0  |   0  0 84  0  0  24  0  0  0  |
    face:      0  0  0 96  0  12  0  0  0  |   0  0  0 84  0  24  0  0  0  |
    house:     0  0  0  0 96  12  0  0  0  |   0  0  0  0 84  24  0  0  0  |
    rest:     12 12 12 12 12 491 12 12 12  |  24 24 24 24 24 394 24 24 24  |
  scissors:    0  0  0  0  0  12 96  0  0  |   0  0  0  0  0  24 84  0  0  |
scrambledpix:  0  0  0  0  0  12  0 96  0  |   0  0  0  0  0  24  0 84  0  |
    shoe:      0  0  0  0  0  12  0  0 96  |   0  0  0  0  0  24  0  0 84  |
Correlations: min=-0.19 max=0.88 mean=-0.00069 sum(abs)=77
```

In *Working with OpenFMRI.org data* you can take a look at an example on how the kind of data preparation described above can be perform in an even more compact way.

The next step is to extract the *patterns of activation* from the dataset that we are interested in. But wait! We know that fMRI data is typically contaminated with a lot of noise, or actually *information* that we are not interested in. For example, there are temporal drifts in the data (the signal tends to increase when the scanner is warming up). We also know that the signal is not fully homogeneous throughout the brain.

All these artifacts carry a lot of variance that is (hopefully) unrelated to the experiment design, and we should try to remove it to present the classifier with the cleanest signal possible. There are countless ways to pre-process the data to try to achieve this goal. Some keywords are: high/low/band-pass filtering, de-spiking, motion-correcting, intensity normalization, and so on. In this tutorial, we keep it simple. The data we have just loaded is already motion corrected. For every experiment that is longer than a few minutes, as in this case, temporal trend removal, or detrending, is crucial.

4.3.3 Basic preprocessing

Detrending

PyMVPA provides functionality to remove polynomial trends from the data (other methods are available too), meaning that polynomials are fitted to the time series and only what is not explained by them remains in the dataset. In the case of linear detrending, this means fitting a straight line to the time series of each voxel via linear regression and taking the residuals as the new feature values. Detrending can be seen as a type of data transformation, hence in PyMVPA it is implemented as a mapper.

```
>>> detrender = PolyDetrendMapper(polyord=1, chunks_attr='chunks')
```

What we have just created is a mapper that will perform chunk-wise linear (1st-order polynomial) detrending. Chunk-wise detrending is desirable, since our data stems from 12 different runs, and the assumption of a continuous linear trend across all runs is not appropriate. The mapper is going to use the `chunks` attribute to identify the chunks in the dataset.

We have seen that we could simply forward-map our dataset with this mapper. However, if we want to have the mapper present in the datasets processing history breadcrumb track, we can use its `get_mapped()` method. This method will cause the dataset to map a shallow copy of itself with the given mapper, and return it. Let's try:

```
>>> detrended_fds = fds.get_mapped(detrender)
>>> print detrended_fds.a.mapper
<Chain: <Flatten>-<StaticFeatureSelection>-<PolyDetrend: ord=1>>
```

`detrended_fds` is easily identifiable as a dataset that has been flattened, sliced, and linearly detrended.

Note that detrending doesn't always have to be an explicit step. For example, if you plan on modelling your data with haemodynamic response functions in a general linear model (like it is shown in *Working with OpenFMRI.org data* with NiPy), polynomial detrending can be done simultaneously as part of the modeling.

Normalization

While this will hopefully have solved the problem of temporal drifts in the data, we still have inhomogeneous voxel intensities that can be a problem for some machine learning algorithms. For this tutorial, we are again following a simple approach to address this issue, and perform a feature-wise, chunk-wise Z-scoring of the data. This has many advantages. First, it is going to scale all features into approximately the same range, and also remove their mean. The latter is quite important, since some classifiers are impaired when working with data having large offsets. However, we are not going to perform a very simple Z-scoring removing the global mean, but use the *rest* condition samples of the dataset to estimate mean and standard deviation. Scaling dataset features using these parameters yields a score corresponding to the per time-point voxel intensity difference from the *rest* average.

This type of data normalization is, you guessed it, also implemented as a mapper:

```
>>> zscorer = ZScoreMapper(param_est=('targets', ['rest']))
```

This mapper configuration implements a *chunk*-wise (the default) Z-scoring, while estimating mean and standard deviation from samples targets with 'rest' in the respective chunk of data.

Remember, all mappers return new datasets that only have copies of what has been modified. However, both detrending and Z-scoring have or will modify the samples themselves. That means that the memory consumption will triple! We will have the original data, the detrended data, and the Z-scored data, but typically we are only interested in the final processing stage. To reduce the memory footprint, both mappers have siblings that perform the same processing, but without copying the data. For `PolyDetrendMapper` this is `poly_detrend()`, and for `ZScoreMapper` this is `zscore()`. The following call will do the same as the mapper we have created above, but using less memory:

```
>>> zscore(detrended_fds, param_est=('targets', ['rest']))
>>> fds = detrended_fds
>>> print fds.a.mapper
<Chain: <Flatten>-<StaticFeatureSelection>-<PolyDetrend: ord=1>-<ZScore>>
```

Exercise

Look at the Simple Data-Exploration *example. Using the techniques from this example, explore the dataset we have just created and look at the effect of detrending and Z-scoring.*

The resulting dataset is now both detrended and normalized. The information is nicely presented in the mapper. From this point on we have no use for the samples of the *rest* category anymore, hence we remove them from the dataset:

```
>>> fds = fds[fds.sa.targets != 'rest']
>>> print fds.shape
(864, 577)
```

Computing *Patterns Of Activation*

The last preprocessing step is to compute the actual *patterns of activation*. In the original study, Haxby and colleagues performed a GLM-analysis of odd vs. even runs of the data respectively and used the corresponding contrast statistics (stimulus category vs. rest) as classifier input. In this tutorial, we will use a much simpler shortcut and just compute *mean* samples per condition for both odd and even run independently.

To achieve this, we first add a new sample attribute to assign a corresponding label to each sample in the dataset that indicates which of both run-types it belongs to:

```
>>> rnames = {0: 'even', 1: 'odd'}
>>> fds.sa['runtype'] = [rnames[c % 2] for c in fds.sa.chunks]
```

The rest is trivial. For cases like this – applying a function (i.e. mean) to a set of groups of samples (all combinations of stimulus category and run-type) – PyMVPA has `FxMapper`. it comes with a number of convenience

functions. The one we need here is `mean_group_sample()`. It takes a list of sample attributes, determines all possible combinations of its unique values, selects dataset samples corresponding to these combinations, and averages them. Finally, since this is also a mapper, a new dataset with mean samples is returned:

```
>>> averager = mean_group_sample(['targets', 'runtype'])
>>> type(averager)
<class 'mvpa2.mappers.fx.FxMapper'>
>>> fds = fds.get_mapped(averager)
>>> fds.shape
(16, 577)
>>> print fds.sa.targets
['bottle' 'cat' 'chair' 'face' 'house' 'scissors' 'scrambledpix' 'shoe'
 'bottle' 'cat' 'chair' 'face' 'house' 'scissors' 'scrambledpix' 'shoe']
>>> print fds.sa.chunks
['0+2+4+6+8+10' '0+2+4+6+8+10' '0+2+4+6+8+10' '0+2+4+6+8+10' '0+2+4+6+8+10'
 '0+2+4+6+8+10' '0+2+4+6+8+10' '0+2+4+6+8+10' '1+3+5+7+9+11' '1+3+5+7+9+11'
 '1+3+5+7+9+11' '1+3+5+7+9+11' '1+3+5+7+9+11' '1+3+5+7+9+11' '1+3+5+7+9+11'
 '1+3+5+7+9+11']
```

Here we go! We now have a fully-preprocessed dataset: masked, detrended, normalized, with one sample per stimulus condition that is an average for odd and even runs respectively. Now we could do some serious classification, and this will be shown in *Classifiers – All Alike, Yet Different*, but there is still an important aspect of mappers we have to look at first.

4.3.4 There and back again – a Mapper's tale

Let's take a look back at the simple datasets from the start of the tutorial part.

```
>>> print ds
<Dataset: 5x12@float64, <a: mapper>>
>>> print ds.a.mapper
<FlattenMapper>
```

A very important feature of mappers is that they allow to reverse a transformation, if that is possible. In case of the simple dataset we can ask the mapper to undo the flattening and to put our samples back into the original 3D shape.

```
>>> orig_data = ds.a.mapper.reverse(ds.samples)
>>> orig_data.shape
(5, 4, 3)
```

In interactive scripting sessions this is would be a relatively bulky command to type, although it might be quite frequently used. To make ones fingers suffer less there is a little shortcut that does exactly the same:

```
>>> orig_data = ds.O
>>> orig_data.shape
(5, 4, 3)
```

It is important to realize that reverse-mapping not only works with a single mapper, but also with a `ChainMapper`. Going back to our demo dataset from the beginning we can see how it works:

```
>>> print subds
<Dataset: 5x4@float64, <a: mapper>>
>>> print subds.a.mapper
<Chain: <Flatten>-<StaticFeatureSelection>>
>>> subds.nfeatures
4
>>> revtest = np.arange(subds.nfeatures) + 10
>>> print revtest
[10 11 12 13]
>>> rmapped = subds.a.mapper.reverse1(revtest)
>>> rmapped.shape
(4, 3)
```

```
>>> print rmapped
[[ 0 10 11]
 [ 0  0  0]
 [ 0  0 12]
 [ 0 13  0]]
```

Reverse mapping of a single sample (one-dimensional feature vector) through the mapper chain created a 4x3 array that corresponds to the dimensions of a sample in our original data space. Moreover, we see that each feature value is precisely placed into the position that corresponds to the features selected in the previous dataset slicing operation.

But now let's look at our fMRI dataset again. Here the mapper chain is a little more complex:

```
>>> print fds.a.mapper
<Chain: <Flatten>-<StaticFeatureSelection>-<PolyDetrend: ord=1>-<ZScore>-<Fx: fx=mean>>
```

Initial flattening followed by mask, detrending, Z-scoring and finally averaging. We would reverse mapping do in this case? Let's test:

```
>>> fds.nfeatures
577
>>> revtest = np.arange(100, 100 + fds.nfeatures)
>>> rmapped = fds.a.mapper.reverse1(revtest)
>>> rmapped.shape
(40, 64, 64)
```

What happens is exactly what we expect: The initial one-dimensional vector is passed backwards through the mapper chain. Reverting a group-based averaging doesn't make much sense for a single vector, hence it is ignored. Same happens for Z-Scoring and temporal detrending. However, for all remaining mappers the transformations are reverse. First unmasked, and then reshaped into the original dimensionality – the brain volume.

We can check that this is really the case by only reverse-mapping through the first two mappers in the chain and compare the result:

```
>>> rmapped_partial = fds.a.mapper[:2].reverse1(revtest)
>>> (rmapped == rmapped_partial).all()
True
```

In case you are wondering: The `ChainMapper` behaves like a regular Python list. We have just selected the first two mappers in the list as another `ChainMapper` and used that one for reverse-mapping.

Back To NIfTI

One last interesting aspect in the context of reverse mapping: Whenever it is necessary to export data from PyMVPA, such as results, dataset mappers also play a critical role. For example we can easily export the `revtest` vector into a NIfTI brain volume image. This is possible because the mapper can put it back into 3D space, and because the dataset also stores information about the original source NIfTI image.

```
>>> 'imghdr' in fds.a
True
```

PyMVPA offers `map2nifti()`, a function to combine these two things and convert any vector into the corresponding NIfTI image:

```
>>> nimg = map2nifti(fds, revtest)
```

This image can now be stored as a file (e.g. `nimg.to_filename('mytest.nii.gz')`). In this format it is now compatible with the vast majority of neuroimaging software.

Exercise

Save the NIfTI image to some file, and use an MRI viewer to overlay it on top of the anatomical image in the demo dataset. Does it match our original mask image of ventral temporal cortex?

There are many more mappers in PyMVPA than we could cover in the tutorial part. Some more will be used in other parts, but even more can be found the `mappers` module. Even though they all implement different transformations, they can all be used in the same way, and can all be combined into a chain.

4.4 Classifiers – All Alike, Yet Different

Note:

This tutorial part is also available for download as an IPython notebook: [ipynb]

In this chapter we will continue our work from *Getting data in shape* in order to replicate the study of *Haxby et al. (2001)*. For this tutorial there is a little helper function to yield the dataset we generated manually before:

```
>>> from mvpa2.tutorial_suite import *
>>> ds = get_haxby2001_data()
```

The original study employed a so-called 1-nearest-neighbor classifier, using correlation as a distance measure. In PyMVPA this type of classifier is provided by the `kNN` class, that makes it possible to specify the desired parameters.

```
>>> clf = kNN(k=1, dfx=one_minus_correlation, voting='majority')
```

A k-Nearest-Neighbor classifier performs classification based on the similarity of a sample with respect to each sample in a *training dataset*. The value of `k` specifies the number of neighbors to derive a prediction, `dfx` sets the distance measure that determines the neighbors, and `voting` selects a strategy to choose a single label from the set of targets assigned to these neighbors.

Exercise

Access the built-in help to inspect the `kNN` class regarding additional configuration options.

Now that we have a classifier instance, it can be easily trained by passing the dataset to its `train()` method.

```
>>> clf.train(ds)
```

A trained classifier can subsequently be used to perform classification of unlabeled samples. The classification performance can be assessed by comparing these predictions to the target labels.

```
>>> predictions = clf.predict(ds.samples)
>>> np.mean(predictions == ds.sa.targets)
1.0
```

We see that the classifier performs remarkably well on our dataset – it doesn't make even a single prediction error. However, most of the time we would not be particularly interested in the prediction accuracy of a classifier on the same dataset that it got trained with.

Exercise

Think about why this particular classifier will always perform error-free classification of the training data – regardless of the actual dataset content. If the reason is not immediately obvious, take a look at chapter 13.3 in The Elements of Statistical Learning. *Investigate how the accuracy varies with different values of* `k`. *Why is that?*

Instead, we are interested in the generalizability of the classifier on new, unseen data. This would allow us, in principle, to use it to assign labels to unlabeled data. Because we only have a single dataset, it needs to be split into (at least) two parts to achieve this. In the original study, Haxby and colleagues split the dataset into patterns of activations from odd versus even-numbered runs. Our dataset has this information in the `runtype` sample attribute:

```
>>> print ds.sa.runtype
['even' 'even' 'even' 'even' 'even' 'even' 'even' 'even' 'odd' 'odd' 'odd'
 'odd' 'odd' 'odd' 'odd' 'odd']
```

Using this attribute we can now easily split the dataset in half. PyMVPA datasets can be sliced in similar ways as NumPy's `ndarray`. The following calls select the subset of samples (i.e. rows in the datasets) where the value of the `runtype` attribute is either the string 'even' or 'odd'.

```
>>> ds_split1 = ds[ds.sa.runtype == 'odd']
>>> len(ds_split1)
8
>>> ds_split2 = ds[ds.sa.runtype == 'even']
>>> len(ds_split2)
8
```

Now we could repeat the steps above: call `train()` with one dataset half and `predict()` with the other, and compute the prediction accuracy manually. However, a more convenient way is to let the classifier do this for us. Many objects in PyMVPA support a post-processing step that we can use to compute something from the actual results. The example below computes the *mismatch error* between the classifier predictions and the *target* values stored in our dataset. To make this work, we do not call the classifier's `predict()` method anymore, but "call" the classifier directly with the test dataset. This is a very common usage pattern in PyMVPA that we shall see a lot over the course of this tutorial. Again, please note that we compute an error now, hence lower values represent more accurate classification.

```
>>> clf.set_postproc(BinaryFxNode(mean_mismatch_error, 'targets'))
>>> clf.train(ds_split2)
>>> err = clf(ds_split1)
>>> print np.asscalar(err)
0.125
```

In this case, our choice of which half of the dataset is used for training and which half for testing was completely arbitrary, hence we could also estimate the transfer error after swapping the roles:

```
>>> clf.train(ds_split1)
>>> err = clf(ds_split2)
>>> print np.asscalar(err)
0.0
```

We see that on average the classifier error is really low, and we achieve an accuracy level comparable to the results reported in the original study.

4.4.1 Cross-validation

What we have just done was to manually split the dataset into combinations of training and testing datasets, given a specific sample attribute – in this case whether a *pattern of activation* or *sample* came from *even* or *odd* runs. We ran the classification analysis on each split to estimate the performance of the classifier model. In general, this approach is called *cross-validation*, and involves splitting the dataset into multiple pairs of subsets, choosing

sample groups by some criterion, and estimating the classifier performance by training it on the first dataset in a split and testing against the second dataset from the same split.

PyMVPA provides a way to allow complete cross-validation procedures to run fully automatic, without the need for manual splitting of a dataset. Using the `CrossValidation` class, a cross-validation is set up by specifying what measure should be computed on each dataset split and how dataset splits should be generated. The measure that is usually computed is the transfer error that we already looked at in the previous section. The second element, a generator for datasets, is another very common tool in PyMVPA. The following example uses `HalfPartitioner`, a generator that, when called with a dataset, marks all samples regarding their association with the first or second half of the dataset. This happens based on the values of a specified sample attribute – in this case `runtype` – much like the manual dataset splitting that we have performed earlier. `HalfPartitioner` will make sure to subsequently assign samples to both halves, i.e. samples from the first half in the first generated dataset will be in the second half of the second generated dataset. With these two techniques we can replicate our manual cross-validation easily – reusing our existing classifier, but without the custom post-processing step.

```
>>> # disable post-processing again
>>> clf.set_postproc(None)
>>> # dataset generator
>>> hpart = HalfPartitioner(attr='runtype')
>>> # complete cross-validation facility
>>> cv = CrossValidation(clf, hpart)
```

Exercise

Try calling the `hpart` object with our dataset. What happens? Now try passing the dataset to its `generate()` methods. What happens now? Make yourself familiar with the concept of a Python generator. Investigate what the code snippet `list(xrange(5))` does, and try to adapt it to the `HalfPartitioner`.

Once the `cv` object is created, it can be called with a dataset, just like we did with the classifier before. It will internally perform all the dataset partitioning, split each generated dataset into training and testing sets (based on the partitions), and train and test the classifier repeatedly. Finally, it will return the results of all cross-validation folds.

```
>>> cv_results = cv(ds)
>>> np.mean(cv_results)
0.0625
```

Actually, the cross-validation results are returned as another dataset that has one sample per fold and a single feature with the computed transfer-error per fold.

```
>>> len(cv_results)
2
>>> cv_results.samples
array([[ 0.   ],
       [ 0.125]])
```

4.4.2 Any classifier, really

A short summary of all code for the analysis we developed so far is this:

```
>>> clf = kNN(k=1, dfx=one_minus_correlation, voting='majority')
>>> cvte = CrossValidation(clf, HalfPartitioner(attr='runtype'))
>>> cv_results = cvte(ds)
>>> np.mean(cv_results)
0.0625
```

Looking at this little code snippet we can nicely see the logical parts of a cross-validated classification analysis.

1. Load the data
2. Choose a classifier

3. Set up an error function
4. Evaluate the error in a cross-validation procedure
5. Inspect results

Our previous choice of the classifier was guided by the intention to replicate *Haxby et al. (2001)*, but what if we want to try a different algorithm? In this case another nice feature of PyMVPA comes into play. All classifiers implement a common interface that makes them easily interchangeable without the need to adapt any other part of the analysis code. If, for example, we want to try the popular `support vector machine` (SVM) on our example dataset it looks like this:

```
>>> clf = LinearCSVMC()
>>> cvte = CrossValidation(clf, HalfPartitioner(attr='runtype'))
>>> cv_results = cvte(ds)
>>> np.mean(cv_results)
0.1875
```

Instead of k-nearest-neighbor, we create a linear SVM classifier, internally using the popular LIBSVM library (note that PyMVPA provides additional SVM implementations). The rest of the code remains identical. SVM with its default settings seems to perform slightly worse than the simple kNN-classifier. We'll get back to the classifiers shortly. Let's first look at the remaining part of this analysis.

We already know that `CrossValidation` can be used to compute errors. So far we have only used the mean number of mismatches between actual targets and classifier predictions as the error function (which is the default). However, PyMVPA offers a number of alternative functions in the *`mvpa2.misc.errorfx`* module, but it is also trivial to specify custom ones. For example, if we do not want to have error reported, but instead accuracy, we can do that:

```
>>> cvte = CrossValidation(clf, HalfPartitioner(attr='runtype'),
...                        errorfx=lambda p, t: np.mean(p == t))
>>> cv_results = cvte(ds)
>>> np.mean(cv_results)
0.8125
```

This example reuses the SVM classifier we have create before, and yields exactly what we expect from the previous result.

The details of the cross-validation procedure are also heavily customizable. We have seen that a `Partitioner` is used to generate training and testing dataset for each cross-validation fold. So far we have only used `HalfPartitioner` to divide the dataset into odd and even runs (based on our custom sample attribute `runtype`). However, in general it is more common to perform so called leave-one-out cross-validation, where *one* independent part of a dataset is selected as testing dataset, while the other parts constitute the training dataset. This procedure is repeated till all parts have served as the testing dataset once. In case of our dataset we could consider each of the 12 runs as independent measurements (fMRI data doesn't allow us to consider temporally adjacent data to be considered independent).

To run such an analysis, we first need to redo our dataset preprocessing, as, in the current one, we only have one sample per stimulus category for both odd and even runs. To get a dataset with one sample per stimulus category for each run, we need to modify the averaging step. Using what we have learned from the *last tutorial part* the following code snippet should be plausible:

```
>>> # directory that contains the data files
>>> datapath = os.path.join(tutorial_data_path, 'haxby2001')
>>> # load the raw data
>>> ds = load_tutorial_data(roi='vt')
>>> # pre-process
>>> poly_detrend(ds, polyord=1, chunks_attr='chunks')
>>> zscore(ds, param_est=('targets', ['rest']))
>>> ds = ds[ds.sa.targets != 'rest']
>>> # average
>>> run_averager = mean_group_sample(['targets', 'chunks'])
>>> ds = ds.get_mapped(run_averager)
```

```
>>> ds.shape
(96, 577)
```

Instead of two samples per category in the whole dataset, now we have one sample per category, per experiment run, hence 96 samples in the whole dataset. To set up a 12-fold leave-one-run-out cross-validation, we can make use of `NFoldPartitioner`. By default it is going to select samples from one `chunk` at a time:

```
>>> cvte = CrossValidation(clf, NFoldPartitioner(),
...                        errorfx=lambda p, t: np.mean(p == t))
>>> cv_results = cvte(ds)
>>> np.mean(cv_results)
0.78125
```

We get almost the same prediction accuracy (reusing the SVM classifier and our custom error function). Note that this time we performed the analysis on a lot more samples that were each was computed from just a few fMRI volumes (about nine each).

So far we have just looked at the mean accuracy or error. Let's investigate the results of the cross-validation analysis a bit further.

```
>>> type(cv_results)
<class 'mvpa2.datasets.base.Dataset'>
>>> print cv_results.samples
[[ 0.75 ]
 [ 0.875]
 [ 1.   ]
 [ 0.75 ]
 [ 0.75 ]
 [ 0.875]
 [ 0.75 ]
 [ 0.875]
 [ 0.75 ]
 [ 0.375]
 [ 1.   ]
 [ 0.625]]
```

The returned value is actually a `Dataset` with the results for all cross-validation folds. Since our error function computes only a single scalar value for each fold the dataset only contains a single feature (in this case the accuracy), and a sample per each fold.

4.4.3 We Need To Take A Closer Look

By now we have already done a few cross-validation analyses using two different classifiers and different preprocessing strategies. In all these cases we have just looked at the generalization performance or error. However, error rates hide a lot of interesting information that is very important for an interpretation of results. In our case we analyzed a dataset with eight different categories. An average misclassification rate doesn't tell us much about the contribution of each category to the prediction error. It could be that *half of the samples of each category* get misclassified, but the same average error might be due to *all samples from half of the categories* being completely misclassified, while prediction accuracy for samples from the remaining categories is perfect. These two results would have to be interpreted in totally different ways, despite the same average error rate.

In psychological research this type of results is usually presented as a contingency table or cross tabulation of expected vs. empirical results. Signal detection theory offers a whole range of techniques to characterize such results. From this angle a classification analysis is hardly any different from a psychological experiment where a human observer performs a detection task, hence the same analysis procedures can be applied here as well.

PyMVPA provides convenient access to *confusion matrices*, i.e. contingency tables of targets vs. actual predictions. However, to prevent wasting CPU-time and memory they are not computed by default, but instead have to be enabled explicitly. Optional analysis results like this are available in a dedicated collection of *conditional attribute*s – analogous to `sa` and `fa` in datasets, it is named `ca`. Let's see how it works:

```
>>> cvte = CrossValidation(clf, NFoldPartitioner(),
...                        errorfx=lambda p, t: np.mean(p == t),
...                        enable_ca=['stats'])
>>> cv_results = cvte(ds)
```

Via the `enable_ca` argument we triggered computing confusion tables for all cross-validation folds, but otherwise there is no change in the code. Afterwards the aggregated confusion for the whole cross-validation procedure is available in the `ca` collection. Let's take a look (note that in the printed manual the output is truncated due to page-width constraints – please refer to the HTML-based version full the full matrix).

```
>>> print cvte.ca.stats.as_string(description=True)
----------.
predictions\targets     bottle         cat          chair          face         house         scis
             `------  ------------  ------------  ------------  ------------  ------------  -------
       bottle              6             0             3             0             0             5
        cat                0             10            0             0             0             0
       chair               0             0             7             0             0             0
        face               0             2             0             12            0             0
       house               0             0             0             0             12            0
      scissors             2             0             1             0             0             6
    scrambledpix           2             0             1             0             0             0
        shoe               2             0             0             0             0             1
Per target:           ------------  ------------  ------------  ------------  ------------  -------
         P                 12            12            12            12            12            1
         N                 84            84            84            84            84            8
         TP                6             10            7             12            12            6
         TN                69            65            68            63            63            6
Summary \ Means:      ------------  ------------  ------------  ------------  ------------  -------
       CHI^2            442.67       p=2e-58
        ACC              0.78
        ACC%            78.12
     # of sets            12       ACC(i) = 0.87-0.015*i p=0.3 r=-0.33 r^2=0.11

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')
 F1 : F1 score
      F1 = 2TP / (P + P') = 2TP / (2TP + FP + FN)
 AUC: Area under (AUC) curve
 CHI^2: Chi-square of confusion matrix
 LOE(ACC): Linear Order Effect in ACC across sets
 # of sets: number of target/prediction sets which were provided
```

This output is a comprehensive summary of the performed analysis. We can see that the confusion matrix has a

strong diagonal, and confusion happens mostly among small objects. In addition to the plain contingency table there are also a number of useful summary statistics readily available – including average accuracy.

Especially for multi-class datasets the matrix quickly becomes incomprehensible. For these cases the confusion matrix can also be plotted via its `plot()` method. If the confusions shall be used as input for further processing they can also be accessed in pure matrix format:

```
>>> print cvte.ca.stats.matrix
[[ 6  0  3  0  0  5  0  1]
 [ 0 10  0  0  0  0  0  0]
 [ 0  0  7  0  0  0  0  0]
 [ 0  2  0 12  0  0  0  0]
 [ 0  0  0  0 12  0  0  0]
 [ 2  0  1  0  0  6  0  0]
 [ 2  0  1  0  0  0 12  1]
 [ 2  0  0  0  0  1  0 10]]
```

The classifier confusions are just an example of the general mechanism of conditional attribute that is supported by many objects in PyMVPA.

4.5 Looking here and there – Searchlights

Note:

This tutorial part is also available for download as an IPython notebook: [ipynb]

In *Classifiers – All Alike, Yet Different* we have seen how we can implement a classification analysis, but we still have no clue about where in the brain (or our chosen ROIs) our signal of interest is located. And that is despite the fact that we have analyzed the data repeatedly, with different classifiers and investigated error rates and confusion matrices. So what can we do?

Ideally, we would like to have some way of estimating a score for each feature that indicates how important that particular feature (most of the time a voxel) is in the context of a certain classification task. There are various possibilities to get a vector of such per-feature scores in PyMVPA. We could simply compute an ANOVA F-score per each feature, yielding scores that would tell us which features vary significantly between any of the categories in our dataset.

Before we can take a look at the implementation details, let's first recreate our preprocessed demo dataset. The code is very similar to that from *Classifiers – All Alike, Yet Different* and should raise no questions. We get a dataset with one sample per category per run.

```
>>> from mvpa2.tutorial_suite import *
>>> ds = get_haxby2001_data(roi='vt')
>>> ds.shape
(16, 577)
```

4.5.1 Measures

Now that we have the dataset, computing the desired ANOVA F-scores is relatively painless:

```
>>> aov = OneWayAnova()
>>> f = aov(ds)
>>> print f
<Dataset: 1x577@float64, <fa: fprob>>
```

If the code snippet above is of no surprise then you probably got the basic idea. We created an object instance `aov` being a `OneWayAnova`. This instance is subsequently *called* with a dataset and yields the F-scores wrapped into a `Dataset`. Where have we seen this before? Right! This one differs little from a call to `CrossValidation`.

Both are objects that get instantiated (potentially with some custom arguments) and yield the results in a dataset when called with an input dataset. This is called a *processing object* and is a common concept in PyMVPA.

However, there is a difference between the two processing objects. `CrossValidation` returns a dataset with a single feature – the accuracy or error rate, while `OneWayAnova` returns a vector with one value per feature. The latter is called a `FeaturewiseMeasure`. But other than the number of features in the returned dataset there is not much of a difference. All measures in PyMVPA, for example, support an optional post-processing step. During instantiation of a measure an arbitrary mapper can be specified to be called internally to forward-map the results before they are returned. If, for some reason, the F-scores need to be scaled into the interval [0,1], an `FxMapper` can be used to achieve that:

```
>>> aov = OneWayAnova(
...          postproc=FxMapper('features',
...                            lambda x: x / x.max(),
...                            attrfx=None))
>>> f = aov(ds)
>>> print f.samples.max()
1.0
```

Exercise

Map the F-scores back into a brain volume and look at their distribution in the ventral temporal ROI.

Now that we know how to compute feature-wise F-scores we can start worrying about them. Our original goal was to decipher information that is encoded in the multivariate pattern of brain activation. But now we are using an ANOVA, a **univariate** measure, to localize important voxels? There must be something else – and there is!

4.5.2 Searching, searching, searching, ...

Kriegeskorte et al. (2006) suggested an algorithm that takes a small, sphere-shaped neighborhood of brain voxels and computes a multivariate measure to quantify the amount of information encoded in its pattern (e.g. mutual information). Later on this searchlight approach has been extended to run a full classifier cross-validation in every possible sphere in the brain. Since that, numerous studies have employed this approach to localize relevant information in a locally constraint fashion.

We know almost all pieces to implement a searchlight analysis in PyMVPA. We can load and preprocess datasets, we can set up a cross-validation procedure.

```
>>> clf = kNN(k=1, dfx=one_minus_correlation, voting='majority')
>>> cv = CrossValidation(clf, HalfPartitioner())
```

The only thing left to do is that we have to split the dataset into all possible sphere neighborhoods that intersect with the brain. To achieve this, we can use `sphere_searchlight()`:

```
>>> sl = sphere_searchlight(cv, radius=3, postproc=mean_sample())
```

This single line configures a searchlight analysis that runs a full cross-validation in every possible sphere in the dataset. Each sphere has a radius of three voxels. The algorithm uses the coordinates (by default `voxel_indices`) stored in a feature attribute of the input dataset to determine local neighborhoods. From the `postproc` argument you might have guessed that this object is also a measure – and your are right. This measure returns whatever value is computed by the basic measure (here this is a cross-validation) and assigns it to the feature representing the center of the sphere in the output dataset. For this initial example we are not interested in the full cross-validation output (error per each fold), but only in the mean error, hence we are using an appropriate mapper for post-processing. As with any other *processing object* we have to call it with a dataset to run the actual analysis:

```
>>> res = sl(ds)
>>> print res
<Dataset: 1x577@float64, <sa: cvfolds>, <fa: center_ids>, <a: mapper>>
```

That was it. However, this was just a toy example with only our ventral temporal ROI. Let's now run it on a much larger volume, so we can actually localize something (even loading and preprocessing will take a few seconds). We will reuse the same searchlight setup and run it on this data as well. Due to the size of the data it might take a few minutes to compute the results, depending on the number of CPUs in the system.

```
>>> ds = get_haxby2001_data_alternative(roi=0)
>>> print ds.nfeatures
34888
>>> res = sl(ds)
```

Now let's see what we got. Since a vector with 35k elements is a little hard to comprehend we have to resort to some statistics.

```
>>> sphere_errors = res.samples[0]
>>> res_mean = np.mean(res)
>>> res_std = np.std(res)
>>> # we deal with errors here, hence 1.0 minus
>>> chance_level = 1.0 - (1.0 / len(ds.uniquetargets))
```

As you'll see, the mean empirical error is just barely below the chance level. However, we would not expect a signal for perfect classification performance in all spheres anyway. Let's see for how many spheres the error is more the two standard deviations lower than chance.

```
>>> frac_lower = np.round(np.mean(sphere_errors < chance_level - 2 * res_std), 3)
```

So in almost 10% of all spheres the error is substantially lower than what we would expect for random guessing of the classifier – that is more than 3000 spheres!

Exercise

Look at the distribution of the errors (hint: `hist(sphere_errors, bins=np.linspace(0, 1, 18))`. In how many spheres do you think the classifier actually picked up real signal? What would be a good value to threshold the errors to distinguish false from true positives? Think of it in the context of statistical testing of fMRI data results. What problems are we facing here?
Once you are done thinking about that – and only after *you're done, project the sphere error map back into the fMRI volume and look at it as a brain overlay in your favorite viewer (hint: you might want to store accuracies instead of errors, if your viewer cannot visualize the lower tail of the distribution: `map2nifti(ds, 1.0 - sphere_errors).to_filename('sl.nii.gz')`). Did looking at the image change your mind?*

4.5.3 For real!

Now that we have an idea of what can happen in a searchlight analysis, let's do another one, but this time on a more familiar ROI – the full brain.

Exercise

Load the dataset with `get_haxby2001_data_alternative(roi='brain')` this will apply any required preprocessing for you. Now run a searchlight analysis for radii 0, 1 and 3. For each resulting error map look at the distribution of values, project them back into the fMRI volume and compare them. How does the distribution change with radius and how does it compare to results of the previous exercise? What would be a good choice for the threshold in this case?

You have now performed a number of searchlight analyses, investigated the results and probably tried to interpret them. What conclusions did you draw from these analyses in terms of the neuroscientific aspects? What have you learned about object representation in the brain? In this case we have run 8-way classification analyses and have looked at the average error rate across all conditions in thousands of sphere-shaped ROIs in the brain. In some spheres the classifier could perform well, i.e. it could predict all samples equally well. However, this only

applies to a handful of over 30k spheres we have tested, and does not reveal whether the classifier was capable of classifying *all* of the conditions or just some. For the vast majority we observe errors somewhere between the theoretical chance level and zero and we don't know what caused the error to decrease. We don't even know which samples get misclassified.

From *Classifiers – All Alike, Yet Different* we know that there is a way out of this dilemma. We can look at the confusion matrix of a classifier to get a lot more information that is otherwise hidden. However, we cannot reasonably do this for thousands of searchlight spheres (Note that this is not completely true. See e.g. *Connolly et al., 2012* for some creative use-cases for searchlights). It becomes obvious that a searchlight analysis is probably not the end of a data exploration but rather a crude take off, as it raises more questions than it answers.

Moreover, a searchlight cannot detect signals that extend beyond a small local neighborhood. This property effectively limits the scope of analyses that can employ this strategy. A study looking a global brain circuitry will hardly restrict the analysis to patches of a few cubic millimeters of brain tissue. As we have seen before, searchlights also have another nasty aspect. Although they provide us with a multivariate localization measure, they also inherit the curse of univariate fMRI data analysis – multiple comparisons. PyMVPA comes with an algorithm that can help to cope with the problem in the context of group analyses: `GroupClusterThreshold`.

Despite these limitations a searchlight analysis can be a valuable exploratory tool if used appropriately. The capabilities of PyMVPA's searchlight implementation go beyond what we looked at in this tutorial. It is not only possible to run *spatial* searchlights, but multiple spaces can be considered simultaneously. This is further illustrated in *Multi-dimensional Searchlights*.

4.6 Classifiers that do more – Meta Classifiers

Note:

This tutorial part is also available for download as an IPython notebook: [ipynb]

In *Classifiers – All Alike, Yet Different* we saw that it is possible to encapsulate a whole cross-validation analysis into a single object that can be called with any dataset to produce the desired results. We also saw that despite this encapsulation we can still get a fair amount of information about the performed analysis. However, what happens if we want to do some further processing of the data **within** the cross-validation analysis. That seems to be difficult, since we feed a whole dataset into the analysis, and only internally does it get split into the respective pieces.

Of course there is a solution to this problem – a *meta-classifier*. This is a classifier that doesn't implement a classification algorithm on its own, but uses another classifier to do the actual work. In addition, the meta-classifier adds another processing step that is performed before the actual base-classifier sees the data.

An example of such a meta-classifier is `MappedClassifier`. Its purpose is simple: Apply a mapper to both training and testing data before it is passed on to the internal base-classifier. With this technique it is possible to implement arbitrary pre-processing within a cross-validation analysis.

Before we get into that, let's reproduce the dataset from *Classifiers – All Alike, Yet Different*:

```
>>> from mvpa2.tutorial_suite import *
>>> # directory that contains the data files
>>> datapath = os.path.join(tutorial_data_path, 'data')
>>> # load the raw data
>>> ds = load_tutorial_data(roi='vt')
>>> # pre-process
>>> poly_detrend(ds, polyord=1, chunks_attr='chunks')
>>> zscore(ds, param_est=('targets', ['rest']))
>>> ds = ds[ds.sa.targets != 'rest']
>>> # average
>>> run_averager = mean_group_sample(['targets', 'chunks'])
>>> ds = ds.get_mapped(run_averager)
>>> ds.shape
(96, 577)
```

Now, suppose we want to perform the classification not on voxel intensities themselves, but on the same samples in the space spanned by the singular vectors of the training data, it would look like this:

```
>>> baseclf = LinearCSVMC()
>>> metaclf = MappedClassifier(baseclf, SVDMapper())
>>> cvte = CrossValidation(metaclf, NFoldPartitioner())
>>> cv_results = cvte(ds)
>>> print np.mean(cv_results)
0.15625
```

First we notice that little has been changed in the code and the results – the error is slightly reduced, but still comparable. The critical line is the second, where we create the `MappedClassifier` from the SVM classifier instance, and a `SVDMapper` that implements singular value decomposition as a mapper.

Exercise

What might be the reasons for the error decrease in comparison to the results on the dataset with voxel intensities?

We know that mappers can be combined into complex processing pipelines, and since `MappedClassifier` takes any mapper as argument, we can implement arbitrary preprocessing steps within the cross-validation procedure. Let's say we have heard rumors that only the first two dimensions of the space spanned by the SVD vectors cover the "interesting" variance and the rest is noise. We can easily check that with an appropriate mapper:

```
>>> mapper = ChainMapper([SVDMapper(), StaticFeatureSelection(slice(None, 2))])
>>> metaclf = MappedClassifier(baseclf, mapper)
>>> cvte = CrossValidation(metaclf, NFoldPartitioner())
>>> cv_results = cvte(ds)
>>> svm_err = np.mean(cv_results)
>>> print round(svm_err, 2)
0.57
```

Well, obviously the discarded components cannot only be noise, since the error is substantially increased. But maybe it is the classifier that cannot deal with the data. Since nothing in this code is specific to the actual classification algorithm we can easily go back to the kNN classifier that has served us well in the past.

```
>>> baseclf = kNN(k=1, dfx=one_minus_correlation, voting='majority')
>>> mapper = ChainMapper([SVDMapper(), StaticFeatureSelection(slice(None, 2))])
>>> metaclf = MappedClassifier(baseclf, mapper)
>>> cvte = CrossValidation(metaclf, NFoldPartitioner())
>>> cv_results = cvte(ds)
>>> np.mean(cv_results) < svm_err
False
```

Oh, that was even worse. We would have to take a closer look at the data to figure out what is happening here.

Exercise

Inspect the confusion matrix of this analysis for both classifiers. What information is represented in the first two SVD components and what is not? Plot the samples of the full dataset after they have been mapped onto the first two SVD components. Why does the kNN classifier perform so bad in comparison to the SVM (hint: think about the distance function)?

In this tutorial part we took a look at classifiers. We have seen that, regardless of the actual algorithm, all classifiers are implementing the same interface. Because of this, they can be replaced by another classifier without having to change any other part of the analysis code. Moreover, we have seen that it is possible to enable and access optional information that is offered by particular parts of the processing pipeline.

4.7 Classification Model Parameters – Sensitivity Analysis

Note:

This tutorial part is also available for download as an IPython notebook: [ipynb]

In the *Looking here and there – Searchlights* we made a first attempt at localizing information in the brain that is relevant to a particular classification analyses. While we were relatively successful, we experienced some problems and also had to wait quite a bit. Here we want to look at another approach to localization. To get started, we pre-process the data as we have done before and perform volume averaging to get a single sample per stimulus category and original experiment session.

```
>>> from mvpa2.tutorial_suite import *
>>> ds = get_raw_haxby2001_data(roi='brain')
>>> print ds.shape
(1452, 39912)
>>> # pre-process
>>> poly_detrend(ds, polyord=1, chunks_attr='chunks')
>>> zscore(ds, param_est=('targets', ['rest']))
>>> ds = ds[ds.sa.targets != 'rest']
>>> # average
>>> run_averager = mean_group_sample(['targets', 'chunks'])
>>> ds = ds.get_mapped(run_averager)
>>> print ds.shape
(96, 39912)
```

A searchlight analysis on this dataset would look exactly as we have seen in *Looking here and there – Searchlights*, but it would take a bit longer due to a higher number of samples. The error map that is the result of a searchlight analysis only offers an approximate localization. First, it is smeared by the overlapping spheres and second the sphere-shaped ROIs probably do not reflect the true shape and extent of functional subregions in the brain. Therefore, it mixes and matches things that might not belong together. This can be mitigated to some degree by using more clever searchlight algorithms (see *Surface-based searchlight on fMRI data*). But it would also be much nicer if we were able to obtain a per-feature measure, where each value can really be attributed to the respective feature and not just to an area surrounding it.

4.7.1 It's A Kind Of Magic

One way to get such a measure is to inspect the classifier itself. Each classifier creates a model to map from the training data onto the respective target values. In this model, classifiers typically associate some sort of weight with each feature that is an indication of its impact on the classifiers decision. How to get this information from a classifier will be the topic of this tutorial.

However, if we want to inspect a trained classifier, we first have to train one. But hey, we have a full brain dataset here with almost 40k features. Will we be able to do that? Well, let's try (and hope that there is still a warranty on the machine you are running this on...).

We will use a simple cross-validation procedure with a linear support vector machine. We will also be interested in summary statistics of the classification, a confusion matrix in our case of classification:

```
>>> clf = LinearCSVMC()
>>> cvte = CrossValidation(clf, NFoldPartitioner(),
...                        enable_ca=['stats'])
```

Ready, set, go!

```
>>> results = cvte(ds)
```

That was surprisingly quick, wasn't it? But was it any good?

```
>>> print np.round(cvte.ca.stats.stats['ACC%'], 1)
26.0
>>> print cvte.ca.stats.matrix
[[1 1 2 3 0 1 1 1]
 [1 2 2 0 2 3 3 1]
 [5 3 3 0 4 3 0 2]
 [3 2 0 5 0 0 0 1]
 [0 3 1 0 3 2 0 0]
 [0 0 0 0 0 0 1 0]
 [0 1 4 3 2 1 7 3]
 [2 0 0 1 1 2 0 4]]
```

Well, the accuracy is not exactly at a chance level, but the confusion matrix doesn't seem to have any prominent diagonal. It looks like, although we can easily train a support vector machine on the full brain dataset, it cannot construct a reliably predicting model. At least we are in the lucky situation to already know that there is some signal in the data, hence we can attribute this failure to the classifier. In most situations it would be as likely that there is actually no signal in the data...

Often people claim that classification performance improves with *feature selection*. If we can reduce the dataset to the important ones, the classifier wouldn't have to deal with all the noise anymore. A simple approach would be to compute a full-brain ANOVA and only go with the voxels that show some level of variance between categories. From the *Looking here and there – Searchlights* we know how to compute the desired F-scores and we could use them to manually select features with some threshold. However, PyMVPA offers a more convenient way – feature selectors:

```
>>> fsel = SensitivityBasedFeatureSelection(
...             OneWayAnova(),
...             FixedNElementTailSelector(500, mode='select', tail='upper'))
```

The code snippet above configures such a selector. It uses an ANOVA measure to select 500 features with the highest F-scores. There are a lot more ways to perform the selection, but we will go with this one for now. The `SensitivityBasedFeatureSelection` instance is yet another *processing object* that can be called with a dataset to perform the feature selection:

```
>>> fsel.train(ds)
>>> ds_p = fsel(ds)
>>> print ds_p.shape
(96, 500)
```

This is the dataset we wanted, so we can rerun the cross-validation and see if it helped. But first, take a step back and look at this code snippet again. There is an object that gets called with a dataset and returns a dataset. You cannot prevent noticing the striking similarity between a measure in PyMVPA or a mapper. And yes, feature selection procedures are also *processing object*s and work just like measures or mappers. Now back to the analysis:

```
>>> results = cvte(ds_p)
>>> print np.round(cvte.ca.stats.stats['ACC%'], 1)
79.2
>>> print cvte.ca.stats.matrix
[[ 5  0  3  0  0  3  0  2]
 [ 0 11  0  0  0  0  0  0]
 [ 0  0  7  0  0  1  0  0]
 [ 2  1  0 12  0  0  0  0]
 [ 0  0  0  0 12  0  0  0]
 [ 2  0  1  0  0  8  0  0]
 [ 0  0  1  0  0  0 12  1]
 [ 3  0  0  0  0  0  0  9]]
```

Yes! We did it. Almost 80% correct classification for an 8-way classification and the confusion matrix has a strong diagonal. Apparently, the ANOVA-selected features were the right ones.

Exercise

If you are not yet screaming or haven't started composing an email to the PyMVPA mailing list pointing to a major problem in the tutorial, you need to reconsider what we have just done. Why is this wrong?

Let's repeat this analysis on a subset of the data. We select only `bottle` and `shoe` samples. In the analysis we just did, they are relatively often confused by the classifier. Let's see how the full brain SVM performs on this binary problem

```
>>> bin_demo = ds[np.array([i in ['bottle', 'shoe'] for i in ds.sa.targets])]
>>> results = cvte(bin_demo)
>>> print np.round(cvte.ca.stats.stats['ACC%'], 1)
62.5
```

Not much, but that doesn't surprise. Let's see what effect our ANOVA-based feature selection has

```
>>> fsel.train(bin_demo)
>>> bin_demo_p = fsel(bin_demo)
>>> results = cvte(bin_demo_p)
>>> print cvte.ca.stats.stats["ACC%"]
100.0
```

Wow, that is a jump. Perfect classification performance, even though the same categories couldn't be distinguished by the same classifier, when trained on all eight categories. I guess, it is obvious that our way of selecting features is somewhat fishy – if not illegal. The ANOVA measure uses the full dataset to compute the F-scores, hence it determines which features show category differences in the whole dataset, including our supposed-to-be independent testing data. Once we have found these differences, we are trying to rediscover them with a classifier. Being able to do that is not surprising, and precisely constitutes the *double-dipping* procedure. As a result, both the obtained prediction accuracy and the created model are potentially completely meaningless.

4.7.2 Thanks For The Fish

To implement an ANOVA-based feature selection *properly* we have to do it on the training dataset **only**. The PyMVPA way of doing this is via a `FeatureSelectionClassifier`:

```
>>> fclf = FeatureSelectionClassifier(clf, fsel)
```

This is a *meta-classifier* and it just needs two things: A basic classifier to do the actual classification work and a feature selection object. We can simply re-use the object instances we already had. Now we got a meta-classifier that can be used just as any other classifier. Most importantly we can plug it into a cross-validation procedure (almost identical to the one we had in the beginning).

```
>>> cvte = CrossValidation(fclf, NFoldPartitioner(),
...                        enable_ca=['stats'])
>>> results = cvte(bin_demo)
>>> print np.round(cvte.ca.stats.stats['ACC%'], 1)
70.8
```

This is a lot worse and a lot closer to the truth – or a so-called unbiased estimate of the generalizability of the classifier model. Now we can also run this improved procedure on our original 8-category dataset.

```
>>> results = cvte(ds)
>>> print np.round(cvte.ca.stats.stats['ACC%'], 1)
78.1
>>> print cvte.ca.stats.matrix
[[ 5  0  2  0  0  4  0  2]
 [ 0 10  0  0  0  0  0  0]
 [ 0  0  8  0  0  1  0  0]
 [ 2  2  0 12  0  0  0  0]
 [ 0  0  0  0 12  0  0  0]
```

```
 [ 1  0  1  0  0  7  0  0]
 [ 0  0  1  0  0  0 12  1]
 [ 4  0  0  0  0  0  0  9]]
```

That is still a respectable accuracy for an 8-way classification and the confusion table also confirms this.

4.7.3 Dissect The Classifier

But now back to our original goal: getting the classifier's opinion about the importance of features in the dataset. With the approach we have used above, the classifier is trained on 500 features. We can only have its opinion about those. Although this is just few times larger than a typical searchlight sphere, we have lifted the spatial constraint of searchlights – these features can come from all over an ROI.

However, we still want to consider more features, so we are changing the feature selection to retain more.

```
>>> fsel = SensitivityBasedFeatureSelection(
...             OneWayAnova(),
...             FractionTailSelector(0.05, mode='select', tail='upper'))
>>> fclf = FeatureSelectionClassifier(clf, fsel)
>>> cvte = CrossValidation(fclf, NFoldPartitioner(),
...                        enable_ca=['stats'])
>>> results = cvte(ds)
>>> print cvte.ca.stats.stats['ACC%'] >= 78.1
False
>>> print np.round(cvte.ca.stats.stats['ACC%'], 1)
69.8
```

A drop of 8% in accuracy on about 4 times the number of features. This time we asked for the top 5% of F-scores.

But how do we get the weights, finally? In PyMVPA many classifiers are accompanied with so-called sensitivity analyzers. This is an object that knows how to get them from a particular classifier type (since each classification algorithm hides them in different places). To create this *analyzer* we can simply ask the classifier to do it:

```
>>> sensana = fclf.get_sensitivity_analyzer()
>>> type(sensana)
<class 'mvpa2.measures.base.MappedClassifierSensitivityAnalyzer'>
```

As you can see, this even works for our meta-classifier. And again this analyzer is a *processing object* that returns the desired sensitivity when called with a dataset.

```
>>> sens = sensana(ds)
>>> type(sens)
<class 'mvpa2.datasets.base.Dataset'>
>>> print sens.shape
(28, 39912)
```

Why do we get 28 sensitivity maps from the classifier? The support vector machine constructs a model for binary classification problems. To be able to deal with this 8-category dataset, the data is internally split into all possible binary problems (there are exactly 28 of them). The sensitivities are extracted for all these partial problems.

Exercise

Figure out which sensitivity map belongs to which combination of categories.

If you are not interested in this level of detail, we can combine the maps into one, as we have done with dataset samples before. A feasible algorithm might be to take the per feature maximum of absolute sensitivities in any of the maps. The resulting map will be an indication of the importance of feature for *some* partial classification.

```
>>> sens_comb = sens.get_mapped(maxofabs_sample())
```

Exercise

Project this sensitivity map back into the fMRI volume and compare it to the searchlight maps of different radii from the previous tutorial part.

You might have noticed some imperfection in our recent approach to computing a full-brain sensitivity map. We derived it from the full dataset, and not from cross-validation splits of the data. Rectifying this is easy with a meta-measure. A meta-measure is analogous to a meta-classifier: a measure that takes a basic measure, adds a processing step to it and behaves like a measure itself. The meta-measure we want to use is `RepeatedMeasure`.

```
>>> sensana = fclf.get_sensitivity_analyzer(postproc=maxofabs_sample())
>>> cv_sensana = RepeatedMeasure(sensana,
...                              ChainNode((NFoldPartitioner(),
...                                         Splitter('partitions',
...                                                  attr_values=(1,)))))
>>> sens = cv_sensana(ds)
>>> print sens.shape
(12, 39912)
```

We re-create our basic sensitivity analyzer, this time automatically applying the post-processing step that combines the sensitivity maps for all partial classifications. Finally, we plug it into the meta-measure that uses the partitions generated by `NFoldPartitioner` to extract the training portions of the dataset for each fold. Afterwards, we can run the analyzer and we get another dataset, this time with a sensitivity map per each cross-validation split.

We could combine these maps in a similar way as before, but let's look at the stability of the ANOVA feature selection instead.

```
>>> ov = MapOverlap()
>>> overlap_fraction = ov(sens.samples > 0)
```

With the *`MapOverlap`* helper we can easily compute the fraction of features that have non-zero sensitivities in all dataset splits.

Exercise

Inspect the `ov` object. Access that statistics map with the fraction of per-feature selections across all splits and project them back into the fMRI volume to investigate them.

This could be the end of the data processing. However, by using the meta measure to compute the sensitivity maps we have lost a convenient way to access the total performance of the underlying classifier. To again gain access to it, and get the sensitivities at the same time, we can twist the processing pipeline a bit.

```
>>> sclf = SplitClassifier(fclf, enable_ca=['stats'])
>>> cv_sensana = sclf.get_sensitivity_analyzer()
>>> sens = cv_sensana(ds)
>>> print sens.shape
(336, 39912)
>>> print cv_sensana.clf.ca.stats.matrix
[[ 5  0  3  0  0  3  0  2]
 [ 0  9  0  0  0  0  0  0]
 [ 0  2  4  0  0  1  0  0]
 [ 2  1  0 12  0  0  0  0]
 [ 0  0  0  0 12  0  0  0]
 [ 3  0  4  0  0  6  2  1]
 [ 0  0  1  0  0  0 10  0]
 [ 2  0  0  0  0  2  0  9]]
```

I guess that deserves some explanation. We wrap our `FeatureSelectionClassifier` with a new thing, a `SplitClassifier`. This is another meta classifier that performs splitting of a dataset and runs training (and prediction) on each of the dataset splits separately. It can effectively perform a cross-validation analysis

internally, and we ask it to compute a confusion matrix of it. The next step is to get a sensitivity analyzer for this meta meta classifier (this time no post-processing). Once we have got that, we can run the analysis and obtain sensitivity maps from all internally trained classifiers. Moreover, the meta sensitivity analyzer also allows access to its internal meta meta classifier that provides us with the confusion statistics. Yeah!

While we are at it, it is worth mentioning that the scenario above can be further extended. We could add more selection or pre-processing steps into the classifier, like projecting the data onto PCA components and limit the classifier to the first 10 components – for each split. PyMVPA offers even more complex meta classifiers (e.g. `TreeClassifier`) that might be very helpful in some analysis scenarios.

4.7.4 Closing Words

We have seen that sensitivity analyses are a useful approach to localize information that is less constrained and less demanding than a searchlight analysis. Specifically, we can use it to discover signals that are distributed throughout the whole set of features (e.g. the full brain), but we could also perform an ROI-based analysis with it. It is less computationally demanding as we only train the classifier on one set of features and not thousands, which results in a significant reduction of required CPU time.

However, there are also caveats. While sensitivities are a much more direct measure of feature importance in the constructed model, being close to the bare metal of classifiers also has problems. Depending on the actual classification algorithm and data preprocessing sensitivities might mean something completely different when compared across classifiers. For example, the popular SVM algorithm solves the classification problem by identifying the data samples that are *most tricky* to model. The extracted sensitivities reflect this property. Other algorithms, such as "Gaussian Naive Bayes" (`GNB`) make assumptions about the distribution of the samples in each category. GNB sensitivities *might* look completely different, even if GNB and SVM classifiers both perform at comparable accuracy levels. Note, however, that these properties can also be used to address related research questions.

It should also be noted that sensitivities can not be directly compared to each other, even if they stem from the same algorithm and are just computed on different dataset splits. In an analysis one would have to normalize them first. PyMVPA offers, for example, `l1_normed()` and `l2_normed()` that can be used in conjunction with `FxMapper` to do that as a post-processing step.

In this tutorial part we also touched the surface of another important topic: *feature selection*. We performed an ANOVA-based feature selection prior to classification to help SVM achieve acceptable performance. One might wonder if that was a clever idea, since a *univariate* feature selection step prior to a *multivariate* analysis somewhat contradicts the goal to identify *multivariate* signals. Only features will be retained that show some signal on their own. If that turns out to be a problem for a particular analysis, PyMVPA offers a number of multivariate alternatives for features selection. There is an implementation of recursive feature selection (`RFE`), and also all classifier sensitivities can be used to select features. For classifiers where sensitivities cannot easily be extracted PyMVPA provides a noise perturbation measure (`NoisePerturbationSensitivity`; see *Hanson et al. (2004)* for an example application).

With these building blocks it is possible to run fairly complex analyses. However, interpreting the results might not always be straight-forward. In the *next tutorial part* we will set out to take away another constraint of all our previously performed analyses. We are going to go beyond spatial analyses and explore the time dimension.

4.8 Event-related Data Analysis

Note:

This tutorial part is also available for download as an IPython notebook: [ipynb]

In all previous tutorial parts we have analyzed the same fMRI data. We analyzed it using a number of different strategies, but they all had one thing in common: A sample in each dataset was always a single volume from an fMRI time series. Sometimes, we have limited ourselves to just a specific temporal windows of interest, sometimes we averaged many fMRI volumes into a single one. In all cases, however, a feature always corresponded to a voxel in the fMRI volume and appeared only once in the dataset.

In this part we are going to extend the analysis beyond the spatial dimensions and will consider *time* as another aspect of our data. We will demonstrate two different approaches: 1) modeling of experimental conditions and proceed with an analysis of model parameter estimates, and 2) the extraction of spatio-temporal data samples. The latter approach is common, for example, in ERP-analyses of EEG data.

Let's start with our well-known example dataset – this time selecting a subset of ventral temporal regions.

```
>>> from mvpa2.tutorial_suite import *
>>> ds = get_raw_haxby2001_data(roi=(36,38,39,40))
```

As we know, this dataset consists of 12 concatenated experiment sessions. Every session had a stimulation block spanning multiple fMRI volumes for each of the eight stimulus categories. Stimulation blocks were separated by rest periods.

4.8.1 Event-related Pre-processing Is Not Event-related

For an event-related analysis, most of the processing is done on data samples that are somehow derived from a set of events. The rest of the data could be considered irrelevant. However, some preprocessing is only meaningful when performed on the full time series and not on the segmented event samples. An example is the detrending that typically needs to be done on the original, continuous time series.

In its current shape our datasets consists of samples that represent contiguous fMRI volumes. At this stage we can easily perform linear detrending.

```
>>> poly_detrend(ds, polyord=1, chunks_attr='chunks')
```

Let's make a copy of the de-trended dataset that we can use later on for some visualization.

```
>>> orig_ds = ds.copy()
```

4.8.2 Design Specification

For any event-related analysis we need some information on the experiment design: when was stimulated with what for how long (and maybe with what intensity). In PyMVPA this is done by compiling a list of event definitions. In many cases, an event is defined by *onset*, *duration* and potentially a number of additional properties, such as stimulus condition or recording session number.

To see how such events definitions look like, we will simply convert the block-design setup defined by the samples attributes of our dataset into a list of events. With `find_events()`, PyMVPA provides a function to convert sequential attributes into event lists. In our dataset, we have the stimulus conditions of each volume sample available in the `targets` sample attribute.

```
>>> events = find_events(targets=ds.sa.targets, chunks=ds.sa.chunks)
>>> print len(events)
204
>>> for e in events[:4]:
...     print e
{'chunks': 0, 'duration': 6, 'onset': 0, 'targets': 'rest'}
{'chunks': 0, 'duration': 9, 'onset': 6, 'targets': 'scissors'}
{'chunks': 0, 'duration': 6, 'onset': 15, 'targets': 'rest'}
{'chunks': 0, 'duration': 9, 'onset': 21, 'targets': 'face'}
```

We are feeding not only the `targets` to the function, but also the `chunks` attribute, since we do not want to have events spanning multiple recording sessions. `find_events()` sequentially parses all provided attributes and records an event whenever the value in *any* of the attributes changes. The generated event definition is a dictionary that contains:

1. Onset of the event as an index in the sequence (in this example this is a volume id)
2. Duration of the event in "number of sequence elements" (i.e. number of volumes). The duration is determined by counting the number of identical attribute combinations following an event onset.

3. Attribute combination of this event, i.e. the actual values of all given attributes at the particular position.

Let's limit ourselves to `face` and `house` stimulation blocks for now. We can easily filter out all other events.

```
>>> events = [ev for ev in events if ev['targets'] in ['house', 'face']]
>>> print len(events)
24
>>> for e in events[:4]:
...     print e
{'chunks': 0, 'duration': 9, 'onset': 21, 'targets': 'face'}
{'chunks': 0, 'duration': 9, 'onset': 63, 'targets': 'house'}
{'chunks': 1, 'duration': 9, 'onset': 127, 'targets': 'face'}
{'chunks': 1, 'duration': 9, 'onset': 213, 'targets': 'house'}
```

4.8.3 Response Modeling

Whenever we have to deal with data were multiple concurrent signals are overlapping in time, such as in fast event-related fMRI studies, it often makes sense to fit an appropriate model to the data and proceed with an analysis of model parameter estimates, instead of the raw data.

PyMVPA can make use of NiPy's GLM modeling capabilities. It expects information on stimulation events to be given as actual time stamps and not data sample indices, hence we have to convert our event list.

```
>>> # temporal distance between samples/volume is the volume repetition time
>>> TR = np.median(np.diff(ds.sa.time_coords))
>>> # convert onsets and durations into timestamps
>>> for ev in events:
...     ev['onset'] = (ev['onset'] * TR)
...     ev['duration'] = ev['duration'] * TR
```

Now we can fit a model of the hemodynamic response to all relevant stimulus conditions. The function `eventrelated_dataset()` does everything for us. For a given input dataset we need to provide a list of events, the name of an attribute with a time stamp for each sample, and information on what conditions we would like to have modeled. The latter is specified to the `condition_attr` argument. This can be a single attribute name in which case all unique values will be used as conditions. It can also be a sequence of multiple attribute names, and all combinations of unique values of the attributes will be used as conditions. In the following example `('targets', 'chunks')` indicates that we want a separate model for each stimulation condition (`targets`) for each run of our example dataset (`chunks`).

```
>>> evds = fit_event_hrf_model(ds,
...                            events,
...                            time_attr='time_coords',
...                            condition_attr=('targets', 'chunks'))
>>> print len(evds)
24
```

This yields one parameter estimate sample for each target value for each chunks.

Exercise

Explore the `evds` dataset. It contains the generated HRF model. Find and plot (some of) them. Take a look at the parameter estimate samples themselves – can you spot a pattern?

Before we can run a classification analysis we still need to normalize each feature (GLM parameters estimates for each voxel at this point).

```
>>> zscore(evds, chunks_attr=None)
```

The rest is straight-forward: we set up a cross-validation analysis with a chosen classifier and run it:

```
>>> clf = kNN(k=1, dfx=one_minus_correlation, voting='majority')
>>> cv = CrossValidation(clf, NFoldPartitioner(attr='chunks'))
>>> cv_glm = cv(evds)
>>> print '%.2f' % np.mean(cv_glm)
0.04
```

Not bad! Let's compare that to a simpler approach that is also suitable for block-design experiments like this one.

```
>>> zscore(ds, param_est=('targets', ['rest']))
>>> avgds = ds.get_mapped(mean_group_sample(['targets', 'chunks']))
>>> avgds = avgds[np.array([t in ['face', 'house'] for t in avgds.sa.targets])]
```

We normalize all voxels with respect to the `rest` condition. This yields some crude kind of "activation" score for all stimulation conditions. Subsequently, we average all sample of a condition in each run. This yield a dataset of the same size as from the GLM modeling. We can re-use the cross-validation setup.

```
>>> cv_avg = cv(avgds)
>>> print '%.2f' % np.mean(cv_avg)
0.04
```

Not bad either. However, it is worth repeating that this simple average-sample approach is limited to block-designs with a clear temporal separation of all signals of interest, whereas the HRF modeling is more suitable for experiments with fast stimulation alternation.

Exercise

Think about what need to be done to perform odd/even run GLM modeling.

4.8.4 From Timeseries To Spatio-temporal Samples

Now we want to try something different. Instead of compressing all temporal information into a single model parameter estimate, we can also consider the entire spatio-temporal signal across our region of interest and the full duration of the stimulation blocks. In other words, we can perform a sensitivity analysis (see *Classification Model Parameters – Sensitivity Analysis*) revealing the spatio-temporal distribution of classification-relevant information.

Before we start with our event-extraction, we want to normalize each feature (i.e. a voxel at this point). In this case we are, again, going to Z-score them, using the mean and standard deviation from the experiment's rest condition, and the resulting values might be interpreted as "activation scores".

```
>>> zscore(ds, chunks_attr='chunks', param_est=('targets', 'rest'))
```

For this analysis we do not have to convert event onset information into time-stamp, but can operate on sample indices, hence we start with the original event list again.

```
>>> events = find_events(targets=ds.sa.targets, chunks=ds.sa.chunks)
>>> events = [ev for ev in events if ev['targets'] in ['house', 'face']]
```

All of our events are of the same length, 9 consecutive fMRI volumes. Later on we would like to view the temporal sensitivity profile from *before* until *after* the stimulation block, hence we should extend the duration of the events a bit.

```
>>> event_duration = 13
>>> for ev in events:
...     ev['onset'] -= 2
...     ev['duration'] = event_duration
```

The next and most important step is to actually segment the original time series dataset into event-related samples. PyMVPA offers `eventrelated_dataset()` as a function to perform this conversion. Let's just do it, it only needs the original dataset and our list of events.

```
>>> evds = eventrelated_dataset(ds, events=events)
>>> len(evds) == len(events)
True
>>> evds.nfeatures == ds.nfeatures * event_duration
True
```

Exercise

Inspect the `evds` dataset. It has a fairly large number of attributes – both for samples and for features. Look at each of them and think about what it could be useful for.

At this point it is worth looking at the dataset's mapper – in particular at the last two items in the chain mapper that have been added during the conversion into events.

```
>>> print evds.a.mapper[-2:]
<Chain: <Boxcar: bl=13>-<Flatten>>
```

Exercise

Reverse-map a single sample through the last two items in the chain mapper. Inspect the result and make sure it doesn't surprise. Now, reverse-map multiple samples at once and compare the result. Is this what you would expect?

The rest of our analysis is business as usual and is quickly done. We want to perform a cross-validation analysis of an SVM classifier. We are not primarily interested in its performance, but in the weights it assigns to the features. Remember, each feature is now voxel-at-time-point, so we get a chance of looking at the spatio-temporal profile of classification-relevant information in the data. We will nevertheless enable computing of a confusion matrix, so we can assure ourselves that the classifier is performing reasonably well, because only a generalizing model is worth inspecting, as otherwise it overfits and the assigned weights could be meaningless.

```
>>> sclf = SplitClassifier(LinearCSVMC(),
...                        enable_ca=['stats'])
>>> sensana = sclf.get_sensitivity_analyzer()
>>> sens = sensana(evds)
```

Exercise

Check that the classifier achieves an acceptable accuracy. Is it enough above chance level to allow for an interpretation of the sensitivities?

Exercise

Using what you have learned in the last tutorial part: Combine the sensitivity maps for all splits into a single map. Project this map into the original dataspace. What is the shape of that space? Store the projected map into a NIfTI file and inspect it using an MRI viewer. Viewer needs to be capable of visualizing time series (hint: for FSLView the time series image has to be opened first)!

4.8.5 A Plotting Example

We have inspected the spatio-temporal profile of the sensitivities using some MRI viewer application, but we can also assemble an informative figure right here. Let's compose a figure that shows the original peri-stimulus time series, the effect of normalization, as well as the corresponding sensitivity profile of the trained SVM classifier.

We are going to do that for two example voxels, whose coordinates we might have derived from inspecting the full map.

```
>>> example_voxels = [(28,25,25), (28,23,25)]
```

The plotting will be done by the popular matplotlib package.

First, we plot the original signal after initial detrending. To do this, we apply the same time series segmentation to the original detrended dataset and plot the mean signal for all face and house events for both of our example voxels. The code below will create the plot using matplotlib's `pylab` interface (imported as `pl`). If you are familiar with Matlab's plotting facilities, this shouldn't be hard to read.

Note:

`_` = is used in the examples below simply to absorb output of plotting functions. You do not have to swallow output in your interactive sessions.

```
>>> # linestyles and colors for plotting
>>> vx_lty = ['-', '--']
>>> t_col = ['b', 'r']
>>>
>>> # for each of the example voxels
>>> for i, v in enumerate(example_voxels):
...     # get a slicing array matching just to current example voxel
...     slicer = np.array([tuple(idx) == v for idx in ds.fa.voxel_indices])
...     # perform the timeseries segmentation just for this voxel
...     evds_detrend = eventrelated_dataset(orig_ds[:, slicer], events=events)
...     # now plot the mean timeseries and standard error
...     for j, t in enumerate(evds.uniquetargets):
...         l = plot_err_line(evds_detrend[evds_detrend.sa.targets == t].samples,
...                           fmt=t_col[j], linestyle=vx_lty[i])
...         # label this plot for automatic legend generation
...         l[0][0].set_label('Voxel %i: %s' % (i, t))
>>> # y-axis caption
>>> _ = pl.ylabel('Detrended signal')
>>> # visualize zero-level
>>> _ = pl.axhline(linestyle='--', color='0.6')
>>> # put automatic legend
>>> _ = pl.legend()
>>> _ = pl.xlim((0,12))
```

In the next figure we do exactly the same again, but this time for the normalized data.

```
>>> for i, v in enumerate(example_voxels):
...     slicer = np.array([tuple(idx) == v for idx in ds.fa.voxel_indices])
...     evds_norm = eventrelated_dataset(ds[:, slicer], events=events)
...     for j, t in enumerate(evds.uniquetargets):
...         l = plot_err_line(evds_norm[evds_norm.sa.targets == t].samples,
...                           fmt=t_col[j], linestyle=vx_lty[i])
...         l[0][0].set_label('Voxel %i: %s' % (i, t))
>>> _ = pl.ylabel('Normalized signal')
>>> _ = pl.axhline(linestyle='--', color='0.6')
>>> _ = pl.xlim((0,12))
```

Finally, we plot the associated SVM weight profile for each peri-stimulus time-point of both voxels. For easier selection we do a little trick and reverse-map the sensitivity profile through the last mapper in the dataset's chain mapper (look at `evds.a.mapper` for the whole chain). This will reshape the sensitivities into `cross-validation fold x volume x voxel features`.

```
>>> # L1 normalization of sensitivity maps per split to make them
>>> # comparable
>>> normed = sens.get_mapped(FxMapper(axis='features', fx=l1_normed))
>>> smaps = evds.a.mapper[-1].reverse(normed)
```

```
>>>
>>> for i, v in enumerate(example_voxels):
...     slicer = np.array([tuple(idx) == v for idx in ds.fa.voxel_indices])
...     smap = smaps.samples[:,:,slicer].squeeze()
...     l = plot_err_line(smap, fmt='ko', linestyle=vx_lty[i], errtype='std')
>>> _ = pl.xlim((0,12))
>>> _ = pl.ylabel('Sensitivity')
>>> _ = pl.axhline(linestyle='--', color='0.6')
>>> _ = pl.xlabel('Peristimulus volumes')
```

That was it. Perhaps you are scared by the amount of code. Please note that it could have been done shorter, but this way allows for plotting any other voxel coordinate combination as well. matplotlib also allows for saving this figure in SVG format, allowing for convenient post-processing in Inkscape – a publication quality figure is only minutes away.

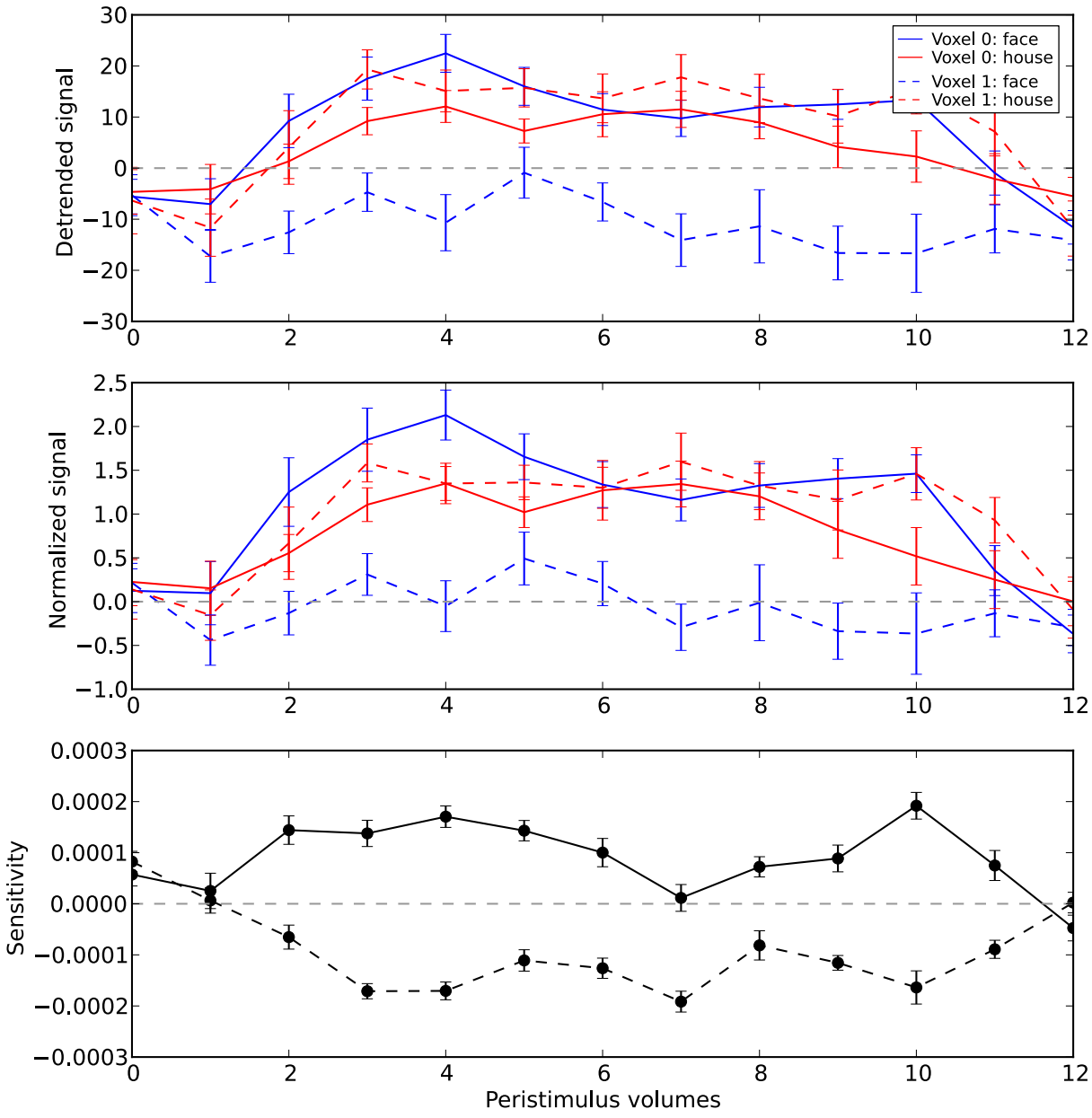


Fig. 4.1: Sensitivity profile for two example voxels for *face* vs. *house* classification on event-related fMRI data from ventral temporal cortex.

Exercise

What can we say about the properties of the example voxel's signal from the peri-stimulus plot?

This demo showed an event-related data analysis. Although we have performed it on fMRI data, an analogous analysis can be done for any time series based data in an almost identical fashion. Moreover, if a dataset has information about acquisition time (e.g. like the ones created by `fmri_dataset()`) `eventrelated_dataset()` can also convert event-definition in real time, making it relatively easy to "convert" experiment design logfiles into event lists. In this case there would be no need to run a function like `find_events()`, but instead they could be directly specified and passed to `eventrelated_dataset()`.

4.9 Multi-dimensional Searchlights

Note:

This tutorial part is also available for download as an IPython notebook: [ipynb]

This is a little addendum to *Event-related Data Analysis* where we want to combine what we have learned about event-related data analysis and, at the same time, take a little glimpse on the power of PyMVPA for "multi-space" analysis.

First let's re-create the dataset with the spatio-temporal features from *Event-related Data Analysis*:

```
>>> from mvpa2.tutorial_suite import *
>>> ds = get_raw_haxby2001_data(roi=(36,38,39,40))
>>> poly_detrend(ds, polyord=1, chunks_attr='chunks')
>>> zscore(ds, chunks_attr='chunks', param_est=('targets', 'rest'))
>>> events = find_events(targets=ds.sa.targets, chunks=ds.sa.chunks)
>>> events = [ev for ev in events if ev['targets'] in ['house', 'face']]
>>> event_duration = 13
>>> for ev in events:
...     ev['onset'] -= 2
...     ev['duration'] = event_duration
>>> evds = eventrelated_dataset(ds, events=events)
```

From the *Looking here and there – Searchlights* we know how to do searchlight analyses and it was promised that there is more to it than what we already saw. And here it is:

```
>>> cvte = CrossValidation(GNB(), NFoldPartitioner(),
...                        postproc=mean_sample())
>>> sl = Searchlight(cvte,
...                  IndexQueryEngine(voxel_indices=Sphere(1),
...                                   event_offsetidx=Sphere(2)),
...                  postproc=mean_sample())
>>> res = sl(evds)
```

Have you been able to deduce what this analysis will do? Clearly, it is some sort of searchlight, but it doesn't use `sphere_searchlight()`. Instead, it utilizes `Searchlight`. Yes, you are correct this is a spatio-temporal searchlight. The searchlight focus travels along all possible locations in our ventral temporal ROI, but at the same time also along the peristimulus time segment covered by the events. The spatial searchlight extent is the center voxel and its immediate neighbors and the temporal dimension comprises of two additional time-points in each direction. The result is again a dataset. Its shape is compatible with the mapper of `evds`, hence it can also be back-projected into the original 4D fMRI brain space.

`Searchlight` is a powerful class that allows for complex runtime ROI generation. In this case it uses an *`IndexQueryEngine`* to look at certain feature attributes in the dataset to compose sphere-shaped ROIs in two spaces at the same time. This approach is very flexible and can be extended with additional query engines to algorithms of almost arbitrary complexity.

```
>>> ts = res.a.mapper.reverse1(1 - res.samples[0])
>>> ni = nb.Nifti1Image(ts, ds.a.imgaffine).to_filename('ersl.nii')
```

After you are done and want to tidy up after yourself, you can easily remove unneeded generated files from within Python:

```
>>> os.unlink('ersl.nii')
```

4.10 Working with OpenFMRI.org data

Note:

This tutorial part is also available for download as an IPython notebook: [ipynb]

Working with data from other researchers can be hard. There are lots of ways to collect data, and even more ways to store it on a hard drive. This variability turns discovering the structure of a "foreign" dataset into a research project of its own.

Standardization is one way to make this easier and the OpenFMRI project has proposed a scheme for structuring (task) fMRI dataset in order to facilitate automated analysis. While there are other approaches to standardization, the layout proposed by OpenFMRI is appealing, because it offers a good balance between the level of standardization and the required effort to achieve it.

PyMVPA offers convenient tools to work with dataset that are (somewhat) compliant with the OpenFMRI structure. So independent of whether you plan on sharing your data or not, it may make sense to adopt these conventions, when working with PyMVPA. Take a look at this tutorial and make up your mind whether there is something about this convenience that you like. As a bonus, if you have your dataset formated for OpenFMRI already, it becomes technically trivial to share it on openfmri.org later on – for free. Here is how it looks like to work with an OpenFMRI dataset, starting with the bare necessities:

```
>>> from os.path import join as opj
>>> import mvpa2
>>> from mvpa2.datasets.sources import OpenFMRIDataset
```

Assuming you downloaded and extracted a dataset from OpenFMRI.org into the current directory, you will have a sub-directory (for example `ds105` if you picked the Haxby et al, (2001) data) that contains all files of the data release. In order to have PyMVPA access this data, we simply have to create a handler that is pointed to this sub-directory. In order to spare you the 2GB download just to run this tutorial, we are using a minified version of that dataset in this demo which already comes with PyMVPA.

```
>>> path = opj(mvpa2.pymvpa_dataroot , 'haxby2001')
>>> of = OpenFMRIDataset(path)
```

Through this handler we can access lots of information about this dataset. Let's start with what this dataset is all about.

```
>>> print of.get_task_descriptions()
{1: 'object viewing'}
```

We can immediately see that the dataset is concerned with a single task *object viewing* The descriptions are always returned as a dictionary that maps the task ID (an integer number) to a verbal description. This is done, because a dataset can contain data for more than one task.

Other descriptive information, such as the number and IDs of the subjects in the dataset, as well as other supporting information specified in the `scan_key.txt` meta data file are also available:

```
>>> print of.get_subj_ids()
[1, 'phantom']
>>> of.get_scan_properties()
{'TR': '2.5'}
```

As you can see, subject IDs don't have to be numerical.

So far, the information we retrieved was rather simple and the advantages of being able to access them through an API will not become obvious until one starts working with a lot of datasets simultaneously. So let's take a look at some functionality that is more useful in the context of a single dataset.

For task fMRI, we are almost always interested in information about the stimulation model, i.e. when was any particular subject exposed to which experiment conditions. All this information is readily available. Here is how you get the number and IDs of all contained model specifications:

```
>>> of.get_model_ids()
[1]
>>> of.get_model_descriptions()
{1: 'visual object categories'}
```

This particular dataset contains a single model specification. With its numerical ID we can query more information about the model:

```
>>> conditions = of.get_model_conditions(1)
>>> print conditions
[{'task': 1, 'id': 1, 'name': 'house'}, {'task': 1, 'id': 2, 'name': 'scrambledpix'}, {'task': 1,
>>> # that was not human readable -> make prettier
>>> print [c['name'] for c in conditions]
['house', 'scrambledpix', 'cat', 'shoe', 'bottle', 'scissors', 'chair', 'face']
```

We can easily get a list of the condition names and their association with a particular task. And with the task ID we can query the dataset for the number (and IDs) of all related BOLD run fMRI images.

```
>>> print of.get_task_bold_run_ids(1)
{1: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12]}
```

If there would be actual data available for the `phantom` subject, we would see it in the output too.

With this information we can access almost any item in this dataset that is related to task fMRI. Take a look at `get_bold_run_image()`, `get_bold_run_dataset()`, and the other methods in order to explore the possibilities. After looking at all the raw information available in a dataset, let's take a look at some high-level functionality that is more interesting when actually working with a task fMRI dataset.

For any supervised analysis strategy, for example a classification analysis, it is necessary to assign labels to data points. In PyMVPA, this is done by creating a dataset with (at least) one sample attribute containing the labels – one for each sample in the dataset. The `get_model_bold_dataset()` method is a convenient way of generating such a dataset directly from the OpenFMRI specification. As you'll see in a second, this methods uses any relevant information contained in the OpenFMRI specification and we only need to fill in the details of how exactly we want the PyMVPA dataset to be created. So here is a complete example:

```
>>> from mvpa2.datasets.eventrelated import fit_event_hrf_model
>>> ds = of.get_model_bold_dataset(
...           model_id=1,
...           subj_id=1,
...           flavor='25mm',
...           mask=opj(path, 'sub001', 'masks', '25mm', 'brain.nii.gz'),
...           modelfx=fit_event_hrf_model,
...           time_attr='time_coords',
...           condition_attr='condition')
```

So let's take this bit of code apart in order to understand what it is doing. When calling `get_model_bold_dataset()`, we specify the model ID and subject ID, as well as the "flavor" of data we are interested in. Think of the flavor as different variants of the same raw fMRI time series (e.g. different set of applied preprocessing steps). We are using the "25mm" flavor, which is our minified variant of the original dataset, down-sampled to voxels with 25 mm edge length. Based on this information, the relevant stimulus model specifications are discovered and data files for the associated subject are loaded. This method could be called in a loop to, subsequently, load data for all available subjects. In addition, we specify a mask image file to exclude non-brain voxels. Often these masks do not come with a data release and have to be created first.

Now for the important bits: The `modelfx` argument takes a, so-called, factory method that can transform a time series dataset (each sample in the dataset is a time point at that stage) into the desired type of sample (or observation). In this example, we have used `fit_event_hrf_model()` that is designed to perform modeling of each stimulation event contained in the OpenFMRI specification. PyMVPA ships with three principal transformation methods that can be used here: `fit_event_hrf_model()`, `extract_boxcar_event_samples()` and `assign_conditionlabels()`. The difference between the three is that the latter simply assignes conditions labels to the time point samples of a time series dataset, whereas the former two can do more complex transformations, such as temporal compression, or model fitting. Note, that is is possible to implement custom transformation functions for `modelfx`, but all common use cases should be supported by the three functions that already come with PyMVPA.

All subsequent argument are passed on to the `modelfx`. In this example, we requested all events of the same condition to be modeled by a regressor that is based on a canonical hemodynamic response function (this requires the specification of a dataset attribute that encodes the timing of a time series samples; `time_attr`).

```
>>> print ds
<Dataset: 96x129@float64, <sa: chunks,condition,regressors,run,subj>, <fa: voxel_indices>, <a: ad
```

This all led to an output dataset with 96 samples, one sample per each of the eight condition in each of the 12 runs.

```
>>> print ds.sa.condition
['bottle' 'cat' 'chair' 'face' 'house' 'scissors' 'scrambledpix' 'shoe'
 'bottle' 'cat' 'chair' 'face' 'house' 'scissors' 'scrambledpix' 'shoe'
 'bottle' 'cat' 'chair' 'face' 'house' 'scissors' 'scrambledpix' 'shoe'
 'bottle' 'cat' 'chair' 'face' 'house' 'scissors' 'scrambledpix' 'shoe'
 'bottle' 'cat' 'chair' 'face' 'house' 'scissors' 'scrambledpix' 'shoe'
 'bottle' 'cat' 'chair' 'face' 'house' 'scissors' 'scrambledpix' 'shoe'
 'bottle' 'cat' 'chair' 'face' 'house' 'scissors' 'scrambledpix' 'shoe'
 'bottle' 'cat' 'chair' 'face' 'house' 'scissors' 'scrambledpix' 'shoe'
 'bottle' 'cat' 'chair' 'face' 'house' 'scissors' 'scrambledpix' 'shoe'
 'bottle' 'cat' 'chair' 'face' 'house' 'scissors' 'scrambledpix' 'shoe'
 'bottle' 'cat' 'chair' 'face' 'house' 'scissors' 'scrambledpix' 'shoe'
 'bottle' 'cat' 'chair' 'face' 'house' 'scissors' 'scrambledpix' 'shoe']
>>> print ds.sa.chunks
[ 0  0  0  0  0  0  0  0  1  1  1  1  1  1  1  1  2  2  2  2  2  2  2  2  3
  3  3  3  3  3  3  3  4  4  4  4  4  4  4  4  5  5  5  5  5  5  5  5  6  6
  6  6  6  6  6  6  7  7  7  7  7  7  7  7  8  8  8  8  8  8  8  8  9  9  9
  9  9  9  9  9 10 10 10 10 10 10 10 10 11 11 11 11 11 11 11 11]
```

Each value in the sample matrix corresponds to the estimated model parameter (or weight) for the associated voxel. Model fitting is performed individually per each run. The model regressors, as well as numerous other bits of information are available in the returned dataset.

Depending on the type of preprocessing that was applied to this data flavor, the dataset `ds` may be ready for immediate analysis, for example in a cross-validated classification analysis. If further preprocessing steps are desired, the `preproc_ds` argument of `get_model_bold_dataset()` provides an interface for applying additional transformations, such as temporal filtering, to the time series data of each individual BOLD fMRI run.

4.11 WiP: The Earth Is Round – Significance Testing

Note:

This tutorial part is also available for download as an IPython notebook: [ipynb]

After performing a classification analysis one is usually interested in an evaluation of the results with respect to its statistical uncertainty. In the following we will take at a few possible approaches to get this from PyMVPA.

Let's look at a typical setup for a cross-validated classification. We start by generating a dataset with 200 samples and 3 features of which only two carry some relevant signal. Afterwards we set up a standard leave-one-chunk-out

cross-validation procedure for an SVM classifier. At this point we have seen this numerous times, and the code should be easy to read:

```
>>> # lazy import
>>> from mvpa2.suite import *
>>> # some example data with signal
>>> ds = normal_feature_dataset(perlabel=100, nlabels=2, nfeatures=3,
...                             nonbogus_features=[0,1], snr=0.3, nchunks=2)
>>> # classifier
>>> clf = LinearCSVMC()
>>> # data folding
>>> partitioner = NFoldPartitioner()
```

```
>>> # complete cross-validation setup
>>> cv = CrossValidation(clf,
...                      partitioner,
...                      errorfx=mean_match_accuracy,
...                      postproc=mean_sample())
>>> acc = cv(ds)
```

Exercise

Take a look at the performance statistics of the classifier. Explore how it changes with different values of the signal-to-noise (`snr`) parameter of the dataset generator function.

The simplest way to get a quick assessment of the statistical uncertainty of the classification accuracy is to look at the standard deviation of the accuracies across cross-validation folds. This can be achieved by removing the `postproc` argument of `CrossValidation`.

Another, slightly more informative, approach is to compute confidence intervals for the classification accuracy. We can do this by treating each prediction of the classifier as a Bernoulli trial with some success probability. If we further assume statistical independence of these prediction outcomes we can compute binomial proportion confidence intervals using a variety of methods. To implement this calculation we only have to modify the error function and the post processing of our previous analysis setup.

```
>>> # complete cross-validation setup
>>> cv = CrossValidation(
...         clf,
...         partitioner,
...         errorfx=prediction_target_matches,
...         postproc=BinomialProportionCI(width=.95, meth='jeffreys'))
>>> ci_result = cv(ds)
>>> ci = ci_result.samples[:, 0]
>>> ci[0] < np.asscalar(acc) < ci[1]
True
```

Instead of computing accuracies we use an error function that returns a boolean vector of prediction success for each sample. In the post processing this information is then used to compute the confidence intervals. We can see that the previously computed accuracy lies within the confidence interval. If the assumption of statistical independence of the classifier prediction success holds we can be 95% certain that the true accuracy is within this interval.

Exercise

Think about situations in which we cannot reasonably assume statistical independence of classifier prediction outcomes. Hint: What if the data in the testing dataset shows strong auto-correlation?

4.11.1 *Null* hypothesis testing

Another way of making statements like *"Performance is significantly above chance-level"* is *Null* hypothesis (aka *H0*) testing that PyMVPA supports for any `Measure`.

However, as with other applications of statistics in classifier-based analyses, there is the problem that we typically do not know the distribution of a variable like error or performance under the *Null* hypothesis (i.e. the probability of a result given that there is no signal), hence we cannot easily assign the adored p-values. 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, total number of labels, as well as the peculiarities of "independence" of training and testing data – especially in the neuroimaging domain.

Monte Carlo – here I come!

One approach to deal with this situation is to *estimate* the *Null* distribution using permutation testing. The *Null* distribution is estimated by computing the measure of interest multiple times using the original data samples but with permuted targets, presumably scrambling or destroying the signal of interest. Since quite often the exploration of all permutations is unfeasible, Monte-Carlo testing (see *Nichols et al. (2002)*) allows us to obtain a stable estimate with only a limited number of random permutations.

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

Here is our previous cross-validation set up:

```
>>> cv = CrossValidation(clf,
...                      partitioner,
...                      postproc=mean_sample(),
...                      enable_ca=['stats'])
>>> err = cv(ds)
```

Now we want to run this analysis again, repeatedly and with a fresh permutation of the targets for each run. We need two pieces for the Monte Carlo shuffling. The first is an instance of an `AttributePermutator` that will permute the target attribute of the dataset for each iteration. We will instruct it to perform 200 permutations. In a real analysis, the number of permutations will often be more than that.

```
>>> permutator = AttributePermutator('targets', count=200)
```

Exercise

The `permutator` is a generator. Try generating all 200 permuted datasets.

The second necessary component for a Monte-Carlo-style estimation of the *Null* distribution is the actual "estimator". `MCNullDist` will use the already created `permutator` to shuffle the targets and later on report the p-value from the left tail of the *Null* distribution, because we are going to compute errors and we are interested in them being *lower* than chance. Finally, we also ask for all results from Monte-Carlo shuffling to be stored for subsequent visualization of the distribution.

```
>>> distr_est = MCNullDist(permutator, tail='left', enable_ca=['dist_samples'])
```

The rest is easy. Measures take an optional constructor argument `null_dist` that can be used to provide an instance of some `NullDist` estimator – and we have just created one! Because a cross-validation is nothing but a measure, we can assign it our *Null* distribution estimator, and it will also perform permutation testing, in addition to the regular classification analysis on the "real" dataset. Consequently, the code hasn't changed much:

```
>>> cv_mc = CrossValidation(clf,
...                         partitioner,
...                         postproc=mean_sample(),
```

```
...                             null_dist=distr_est,
...                             enable_ca=['stats'])
>>> err = cv_mc(ds)
>>> cv.ca.stats.stats['ACC'] == cv_mc.ca.stats.stats['ACC']
True
```

Other than it taking a bit longer to compute, the performance did not change. But the additional waiting wasn't in vain, as we get the results of the statistical evaluation. The `cv_mc` *conditional attribute* `null_prob` has a dataset that contains the p-values representing the likelihood of an empirical value (i.e. the result from analysing the original dataset) being equal or lower to one under the *Null* hypothesis, i.e. no actual relevant signal in the data. Or in more concrete terms, the p-value is the fraction of permutation results less than or equal to the empirical result.

```
>>> p = cv_mc.ca.null_prob
>>> # should be exactly one p-value
>>> p.shape
(1, 1)
>>> np.asscalar(p) < 0.1
True
```

Exercise

How many cross-validation analyses were computed when running `cv_mc`? Make sure you are not surprised that it is more than 200. What is the minimum p-value that we can get from 200 permutations?

Let's practise our visualization skills a bit and create a quick plot to show the *Null* distribution and how "significant" our empirical result is. And let's make a function for plotting to show off our Python-foo!

```
>>> def make_null_dist_plot(dist_samples, empirical):
...     pl.hist(dist_samples, bins=20, normed=True, alpha=0.8)
...     pl.axvline(empirical, color='red')
...     # chance-level for a binary classification with balanced samples
...     pl.axvline(0.5, color='black', ls='--')
...     # scale x-axis to full range of possible error values
...     pl.xlim(0,1)
...     pl.xlabel('Average cross-validated classification error')
>>>
>>> # make new figure ('_ =' is only used to swallow unnecessary output)
>>> _ = pl.figure()
>>> make_null_dist_plot(np.ravel(cv_mc.null_dist.ca.dist_samples),
...                     np.asscalar(err))
>>> # run pl.show() if the figure doesn't appear automatically
```

You can see that we have created a histogram of the "distribution samples" stored in the *Null* distribution (because we asked for it previously). We can also see that the *Null* or chance distribution is centered around the expected chance-level and the empirical error value is in the far left tail, thus relatively unlikely to be a *Null* result, hence the low-ish p-value.

This wasn't too bad, right? We could stop here. But there is this smell....

Exercise

The p-value that we have just computed and the Null *distribution we looked at are, unfortunately,* ***invalid*** *– at least if we want to know how likely it is to obtain our* ***empirical*** *result under a no-signal condition. Can you figure out why?*
PS: The answer is obviously in the next section, so do not spoil your learning experience by reading it before you have thought about this issue!

Avoiding the trap OR Advanced magic 101

Here is what went wrong: The dataset's class labels (aka targets) were shuffled repeatedly, and for each iteration a full cross-validation of classification error was computed. However, the shuffling was done on the *full* dataset, hence target values were permuted in both training *and* testing dataset portions in each CV-fold. This basically means that for each Monte Carlo iteration the classifier was **tested** on new data/signal. However, we are actually interested in what the classifier has to say about the *actual* data, but when it was **trained** on randomly permuted data.

Doing a whole-dataset permutation is a common mistake with very beneficial side-effects – as you will see in a bit. Sadly, doing the permuting correctly (i.e. in the training portion of the dataset only) is a bit more complicated due to the data-folding scheme that we have to deal with. Here is how it goes:

```
>>> repeater = Repeater(count=200)
```

A `repeater` is a simple node that returns any given dataset a configurable number of times. We use this helper to configure the number of Monte Carlo iterations.

Exercise

A `Repeater` is also a generator. Try calling it with our dataset. What does it do? How can you get it to produce the 200 datasets?

The new `permutator` is again configured to shuffle the `targets` attribute. But this time only *once* and only for samples that were labeled as being part of the training set in a particular CV-fold. The `partitions` sample attribute is created by the NFoldPartitioner that we have already configured earlier (or any other partitioner in PyMVPA for that matter).

```
>>> permutator = AttributePermutator('targets',
...                                  limit={'partitions': 1},
...                                  count=1)
```

The most significant difference is that we are now going to use a dedicate measure to estimate the *Null* distribution. That measure is very similar to the cross-validation we have used before, but differs in an important twist: we use a chained generator to perform the data-folding. This chain comprises of our typical partitioner (marks one chunk as testing data and the rest as training, for all chunks) and the new one-time permutator. This chain-generator causes the cross-validation procedure to permute the training data only for each data-fold and leave the testing data untouched. Note, that we make the chain use the `space` of the partitioner, to let the `CrossValidation` know which samples attribute defines training and testing partitions.

```
>>> null_cv = CrossValidation(
...             clf,
...             ChainNode(
...                  [partitioner, permutator],
...                  space=partitioner.get_space()),
...             postproc=mean_sample())
```

Exercise

Create a separate chain-generator and explore what it does. Remember: it is just a generator.

Now we create our new and improved distribution estimator. This looks similar to what we did before, but we now use our dedicated *Null* cross-validation measure, and run it as often as `repeater` is configured to estimate the *Null* performance.

```
>>> distr_est = MCNullDist(repeater, tail='left',
...                        measure=null_cv,
...                        enable_ca=['dist_samples'])
```

On the "outside" the cross-validation measure for computing the empricial performance estimate is 100% identical to what we have used before. All the magic happens inside the distribution estimator.

```
>>> cv_mc_corr = CrossValidation(clf,
...                              partitioner,
...                              postproc=mean_sample(),
...                              null_dist=distr_est,
...                              enable_ca=['stats'])
>>> err = cv_mc_corr(ds)
>>> cv_mc_corr.ca.stats.stats['ACC'] == cv_mc.ca.stats.stats['ACC']
True
>>> cv_mc.ca.null_prob.samples <  cv_mc_corr.ca.null_prob.samples
array([[ True]], dtype=bool)
```

After running it we see that there is no change in the empirical performance (great!), but our significance did suffer (poor thing!). We can take a look at the whole picture by plotting our previous *Null* distribution estimate and the new, improved one as an overlay.

```
>>> make_null_dist_plot(cv_mc.null_dist.ca.dist_samples, np.asscalar(err))
>>> make_null_dist_plot(cv_mc_corr.null_dist.ca.dist_samples, np.asscalar(err))
>>> # run pl.show() if the figure doesn't appear automatically
```

It should be obvious that there is a substantial difference in the two estimates, but only the latter/wider distribution is valid!

Exercise

Keep it in mind. Keep it in mind. Keep it in mind.

4.11.2 The following content is incomplete and experimental

If you have a clue

There a many ways to further tweak the statistical evaluation. For example, if the family of the distribution is known (e.g. Gaussian/Normal) and provided via the `dist_class` parameter of `MCNullDist`, then permutation tests samples will be used to fit this particular distribution and estimate distribution parameters. This could yield enormous speed-ups. Under the (strong) assumption of Gaussian distribution, 20-30 permutations should be sufficient to get sensible estimates of the distribution parameters. Fitting a normal distribution would look like this. Actually, only a single modification is necessary (the `dist_class` argument), but we will also reduce the number permutations.

```
>>> distr_est = MCNullDist(Repeater(count=200),
...                        dist_class=scipy.stats.norm,
...                        tail='left',
...                        measure=null_cv,
...                        enable_ca=['dist_samples'])
>>> cv_mc_norm = CrossValidation(clf,
...                              partitioner,
...                              postproc=mean_sample(),
...                              null_dist=distr_est,
...                              enable_ca=['stats'])
>>> err = cv_mc_norm(ds)
>>> distr = cv_mc_norm.null_dist.dists()[0]
>>> make_null_dist_plot(cv_mc_norm.null_dist.ca.dist_samples,
...                     np.asscalar(err))
>>> x = np.linspace(0,1,100)
>>> _ = pl.plot(x, distr.pdf(x), color='black', lw=2)
```

Family-friendly

When going through this chapter you might have thought: "Jeez, why do they need to return a single p-value in a freaking dataset?" But there is a good reason for this. Lets set up another cross-validation procedure. This one is basically identical to the last one, except for not averaging classifier performances across data-folds (i.e. `postproc=mean_sample()`).

```
>>> cvf = CrossValidation(
...         clf,
...         partitioner,
...         null_dist=MCNullDist(
...                     repeater,
...                     tail='left',
...                     measure=CrossValidation(
...                                 clf,
...                                 ChainNode([partitioner, permutator],
...                                           space=partitioner.get_space()))
...                     )
...         )
```

If we run this on our dataset, we no longer get a single performance value, but one per data-fold (chunk) instead:

```
>>> err = cvf(ds)
>>> len(err) == len(np.unique(ds.sa.chunks))
True
```

But here comes the interesting bit:

```
>>> len(cvf.ca.null_prob) == len(err)
True
```

So we get one p-value for each element in the datasets returned by the cross-validation run. More generally speaking, the distribution estimation happens independently for each value returned by a measure – may this be multiple samples, or multiple features, or both. Consequently, it is possible to test a large variety of measure with this facility.

4.11.3 Evaluating multi-class classifications

So far we have mostly looked at the situation where a classifier is trying to discriminate data from two possible classes. In many cases we can assume that a classifier that *cannot* discriminate these two classes would perform at a chance-level of 0.5 (ACC). If it does that we would conclude that there is no signal of interest in the data, or our classifier of choice cannot pick it up. However, there is a whole universe of classification problems where it is not that simple.

Let's revisit the classification problem from *the chapter on classifiers*.

```
>>> from mvpa2.tutorial_suite import *
>>> ds = get_haxby2001_data_alternative(roi='vt', grp_avg=False)
>>> print ds.sa['targets'].unique
['bottle' 'cat' 'chair' 'face' 'house' 'scissors' 'scrambledpix' 'shoe']
>>> clf = kNN(k=1, dfx=one_minus_correlation, voting='majority')
>>> cv = CrossValidation(clf, NFoldPartitioner(), errorfx=mean_mismatch_error,
...                      enable_ca=['stats'])
>>> cv_results = cv(ds)
>>> print '%.2f' % np.mean(cv_results)
0.53
```

So here we have an 8-way classification problem, and during the cross-validation procedure the chosen classifier makes correct predictions for approximately half of the data points. The big question is now: **What does that tell us?**

There are many scenarios that could lead to this prediction performance. It could be that the fitted classifier model is very good, but only captures the data variance for half of the data categories/classes. It could also be that the

classifier model quality is relatively poor and makes an equal amount of errors for all classes. In both cases the average accuracy will be around 50%, and most likely **highly significant**, given a chance performance of 1/8. We could now spend some time testing this significance with expensive permutation tests, or making assumptions on the underlying distribution. However, that would only give us a number telling us that the average accuracy is really different from chance, but it doesn't help with the problem that the accuracy really doesn't tell us much about what we are interested in.

Interesting hypotheses in the context of this dataset could be whether the data carry a signal that can be used to distinguish brain response patterns from animate vs. inanimate stimulus categories, or whether data from object-like stimuli are all alike and can only be distinguished from random noise, etc. One can imagine running such an analysis on data from different parts of the brain and the results changing – without necessarily having a big impact on the overall classification accuracy.

A lot more interesting information is available from the confusion matrix, a contingency table showing prediction targets vs. actual predictions.

```
>>> print cv.ca.stats.matrix
[[36  7 18  4  1 18 15 18]
 [ 3 56  6 18  0  3  7  5]
 [ 2  2 21  0  4  0  3  1]
 [ 3 16  0 76  4  5  3  1]
 [ 1  1  6  1 97  1  4  0]
 [20  5 15  4  0 29 15 11]
 [ 0  1  0  0  0  2 19  0]
 [43 20 42  5  2 50 42 72]]
```

We can see a strong diagonal, but also block-like structure, and have to realize that simply staring at the matrix doesn't help us to easily assess the likelihood of any of our hypotheses being true or false. It is trivial to do a Chi-square test of the confusion table...

```
>>> print 'Chi^2: %.3f (p=%.3f)' % cv.ca.stats.stats["CHI^2"]
Chi^2: 1942.519 (p=0.000)
```

... but, again, it doesn't tell us anything other than that the classifier is not just doing random guesses. It would be much more useful if we could estimate how likely it is, given the observed confusion matrix, that the employed classifier is able to discriminate *all* stimulus classes from each other, and not just a subset. Even more useful would be if we could relate this probability to specific alternative hypotheses, such as an animate/inanimate-only distinction.

Olivetti et al. (2012) have devised a method that allows for doing exactly that. The confusion matrix is analyzed in a Bayesian framework regarding the statistical dependency of observed and predicted class labels. Confusions within a set of classes that cannot be discriminated should be independently distributed, while there should be a statistical dependency of confusion patterns within any set of classes that can all be discriminated from each other.

This algorithm is available in the `BayesConfusionHypothesis` node.

```
>>> cv = CrossValidation(clf, NFoldPartitioner(),
...                      errorfx=None,
...                      postproc=ChainNode((Confusion(labels=ds.UT),
...                                          BayesConfusionHypothesis())))
>>> cv_results = cv(ds)
>>> print cv_results.fa.stat
['log(p(C|H))' 'log(p(H|C))']
```

Most likely hypothesis to explain this confusion matrix:

```
>>> print cv_results.sa.hypothesis[np.argsort(cv_results.samples[:,1])[-1]]
[['bottle'], ['cat'], ['chair'], ['face'], ['house'], ['scissors'], ['scrambledpix'], ['shoe']]
```

4.11.4 Previously in part 8

Previously, *while looking at classification* we have observed that classification error depends on the chosen classification method, data preprocessing, and how the error was obtained – training error vs generalization estimates

using different data splitting strategies. Moreover in *attempts to localize activity using searchlight* we saw that generalization error can reach relatively small values even when processing random data which (should) have no true signal. So, the value of the error alone does not provide sufficient evidence to state that our classifier or any other method actually learnt the mapping from the data into variables of interest. So, how do we decide what estimate of error can provide us sufficient evidence that constructed mapping reflects the underlying phenomenon or that our data carried the signal of interest?

Researchers interested in developing statistical learning methods usually aim at achieving as high generalization performance as possible. Newly published methods often stipulate their advantage over existing ones by comparing their generalization performance on publicly available datasets with known characteristics (number of classes, independence of samples, actual presence of information of interest, etc.). Therefore, generalization performances presented in statistical learning publications are usually high enough to obliterate even a slight chance that they could have been obtained simply by chance. For example, those classifiers trained on MNIST dataset of handwritten digits were worth reporting whenever they demonstrated average **errors of only 1-2%** while doing classification among samples of 10 different digits (the largest error reported was 12% using the simplest classification approach).

The situation is substantially different in the domain of neural data analysis. There classification is most often used not to construct a reliable mapping from data into behavioral variable(s) with as small error as possible, but rather to show that learnt mapping is good enough to claim that such mapping exists and data carries the effects caused by the corresponding experiment. Such an existence claim is conventionally verified with a classical methodology of null-hypothesis (H0) significance testing (NHST), whenever the achievement of generalization performance with *statistically significant* excursion away from the *chance-level* is taken as the proof that data carries effects of interest.

The main conceptual problem with NHST is a widespread belief that having observed the data, the level of significance at which H0 could be rejected is equivalent to the probability of the H0 being true. I.e. if it is unlikely that data comes from H0, it is as unlikely for H0 being true. Such assumptions were shown to be generally wrong using *deductive and Bayesian reasoning* since P(D|H0) not equal P(H0|D) (unless P(D)==P(H0)). Moreover, *statistical significance* alone, taken without accompanying support on viability and reproducibility of a given finding, was argued *more likely to be incorrect*.

What differs multivariate analysis from univariate is that it

- avoids **multiple comparisons** problem in NHST
- has higher **flexibility**, thus lower **stability**

Multivariate methods became very popular in the last decade of neuroimaging research partially due to their inherent ability to avoid multiple comparisons issue, which is a flagman of difficulties while going for a *fishing expedition* with univariate methods. Performing cross-validation on entire ROI or even full-brain allowed people to state presence of so desired effects without defending chosen critical value against multiple-comparisons. Unfortunately, as there is no such thing as *free lunch*, ability to work with all observable data at once came at a price for multivariate methods.

The second peculiarity of the application of statistical learning in psychological research is the actual neural data which researchers are doomed to analyze. As we have already seen from previous tutorial parts, typical fMRI data has

- relatively **low number of samples** (up to few thousands in total)
- relatively **large dimensionality** (tens of thousands)
- **small signal-to-noise ratio**
- **non-independent measurements**
- **unknown ground-truth** (either there is an effect at all, or if there is – what is inherent bias/error)
- **unknown nature of the signal**, since BOLD effect is not entirely understood.

In the following part of the tutorial we will investigate the effects of some of those factors on classification performance with simple (or not so) examples. But first lets overview the tools and methodologies for NHST commonly employed.

4.11.5 Statistical Tools in Python

`scipy` Python module is an umbrella project to cover the majority of core functionality for scientific computing in Python. In turn, `stats` submodule covers a wide range of continuous and discrete distributions and statistical functions.

Exercise

Glance over the `scipy.stats` documentation for what statistical functions and distributions families it provides. If you feel challenged, try to figure out what is the meaning/application of `rdist()`.

The most popular distribution employed for NHST in the context of statistical learning, is `binom` for testing either generalization performance of the classifier on independent data could provide evidence that the data contains the effects of interest.

Note:

`scipy.stats` provides function `binom_test()`, but that one was devised only for doing two-sides tests, thus is not directly applicable for testing generalization performance where we aim at the tail with lower than chance performance values.

Exercise

Think about scenarios when could you achieve strong and very significant mis-classification performance, i.e. when, for instance, binary classifier tends to generalize into the other category. What could it mean?

`binom` whenever instantiated with the parameters of the distribution (which are number of trials, probability of success on each trial), it provides you ability to easily compute a variety of statistics of that distribution. For instance, if we want to know, what would be the probability of having achieved 57 of more correct responses out of 100 trials, we need to use a survival function (1-cdf) to obtain the *weight* of the right tail including 57 (i.e. query for survival function of 56):

```
>>> from scipy.stats import binom
>>> binom100 = binom(100, 1./2)
>>> print '%.3g' % binom100.sf(56)
0.0967
```

Apparently obtaining 57 correct out 100 cannot be considered significantly good performance by anyone. Lets investigate how many correct responses we need to reach the level of 'significance' and use *inverse survival function*:

```
>>> binom100.isf(0.05) + 1
59.0
>>> binom100.isf(0.01) + 1
63.0
```

So, depending on your believe and prior support for your hypothesis and data you should get at least 59-63 correct responses from a 100 trials to claim the existence of the effects. Someone could rephrase above observation that to achieve significant performance you needed an effect size of 9-13 correspondingly for those two levels of significance.

Exercise

Plot a curve of effect sizes *(number of correct predictions above chance-level) vs a number of trials at significance level of 0.05 for a range of trial numbers from 4 to 1000. Plot %-accuracy vs number of trials for the same range in a separate plot. TODO*

4.11.6 Dataset Exploration for Confounds

"Randomization is a crucial aspect of experimental design... In the absence of random allocation, unforeseen factors may bias the results.".

Unfortunately it is impossible to detect and warn about all possible sources of confounds which would invalidate NHST based on a simple parametric binomial test. As a first step, it is always useful to inspect your data for possible sources of samples non-independence, especially if your results are not strikingly convincing or too provocative. Possible obvious problems could be:

- dis-balanced testing sets (usually non-equal number of samples for each label in any given chunk of data)
- order effects: either preference of having samples of particular target in a specific location or the actual order of targets

To allow for easy inspection of dataset to prevent such obvious confounds, `summary()` function (also a method of any `Dataset`) was constructed. Lets have yet another look at our 8-categories dataset:

```
>>> from mvpa2.tutorial_suite import *
>>> ds = get_haxby2001_data(roi='vt')
>>> print ds.summary()
Dataset: 16x577@float64, <sa: chunks,run,runtype,subj,targets,task,time_coords,time_indices>, <fa
stats: mean=11.5788 std=13.7772 var=189.811 min=-49.5554 max=97.292

Counts of targets in each chunk:
  chunks\targets bottle cat chair face house scissors scrambledpix shoe
                   ---  ---  ---   ---  ---     ---         ---      ---
  0+2+4+6+8+10       1    1    1     1    1       1           1        1
  1+3+5+7+9+11       1    1    1     1    1       1           1        1

Summary for targets across chunks
    targets  mean std min max #chunks
   bottle       1   0   1   1      2
     cat        1   0   1   1      2
    chair       1   0   1   1      2
    face        1   0   1   1      2
    house       1   0   1   1      2
  scissors      1   0   1   1      2
scrambledpix    1   0   1   1      2
    shoe        1   0   1   1      2

Summary for chunks across targets
    chunks   mean std min max #targets
0+2+4+6+8+10   1   0   1   1      8
1+3+5+7+9+11   1   0   1   1      8
Sequence statistics for 16 entries from set ['bottle', 'cat', 'chair', 'face', 'house', 'scissors
Counter-balance table for orders up to 2:
Targets/Order O1                |  O2                |
   bottle:     0 2 0 0 0 0 0 0  |   0 0 2 0 0 0 0 0  |
     cat:      0 0 2 0 0 0 0 0  |   0 0 0 2 0 0 0 0  |
    chair:     0 0 0 2 0 0 0 0  |   0 0 0 0 2 0 0 0  |
    face:      0 0 0 0 2 0 0 0  |   0 0 0 0 0 2 0 0  |
    house:     0 0 0 0 0 2 0 0  |   0 0 0 0 0 0 2 0  |
  scissors:    0 0 0 0 0 0 2 0  |   0 0 0 0 0 0 0 2  |
scrambledpix:  0 0 0 0 0 0 0 2  |   1 0 0 0 0 0 0 0  |
    shoe:      1 0 0 0 0 0 0 0  |   0 1 0 0 0 0 0 0  |
Correlations: min=-0.52 max=1 mean=-0.067 sum(abs)=5.7
```

You can see that labels were balanced across chunks – i.e. that each chunk has an equal number of samples of each target label, and that samples of different labels are evenly distributed across chunks. TODO...

Counter-balance table shows either there were any order effects among conditions. In this case we had only two instances of each label in the dataset due to the averaging of samples across blocks, so it would be more informative to look at the original sequence. To do so avoiding loading a complete dataset we would simply provide the stimuli

sequence to `SequenceStats` for the analysis:

```
>>> attributes_filename = os.path.join(pymvpa_dataroot, 'attributes_literal.txt')
>>> attr = SampleAttributes(attributes_filename)
>>> targets = np.array(attr.targets)
>>> ss = SequenceStats(attr.targets)
>>> print ss
Sequence statistics for 1452 entries from set ['bottle', 'cat', 'chair', 'face', 'house', 'rest',
Counter-balance table for orders up to 2:
Targets/Order O1                          |  O2                                       |
   bottle:    96  0  0  0  0  12  0  0  0  |  84  0  0  0  0  24  0  0  0  |
     cat:      0 96  0  0  0  12  0  0  0  |   0 84  0  0  0  24  0  0  0  |
    chair:     0  0 96  0  0  12  0  0  0  |   0  0 84  0  0  24  0  0  0  |
    face:      0  0  0 96  0  12  0  0  0  |   0  0  0 84  0  24  0  0  0  |
    house:     0  0  0  0 96  12  0  0  0  |   0  0  0  0 84  24  0  0  0  |
    rest:     12 12 12 12 12 491 12 12 12  |  24 24 24 24 24 394 24 24 24  |
  scissors:    0  0  0  0  0  12 96  0  0  |   0  0  0  0  0  24 84  0  0  |
scrambledpix:  0  0  0  0  0  12  0 96  0  |   0  0  0  0  0  24  0 84  0  |
    shoe:      0  0  0  0  0  12  0  0 96  |   0  0  0  0  0  24  0  0 84  |
Correlations: min=-0.19 max=0.88 mean=-0.00069 sum(abs)=77
```

Order statistics look funky at first, but they would not surprise you if you recall the original design of the experiment – blocks of 8 TRs per each category, interleaved with 6 TRs of rest condition. Since samples from two adjacent blocks are far apart enough not to contribute to 2-back table (O2 table on the right), it is worth inspecting if there was any dis-balance in the order of the picture conditions blocks. It would be easy to check if we simply drop the 'rest' condition from consideration:

```
>>> print SequenceStats(targets[targets != 'rest'])
Sequence statistics for 864 entries from set ['bottle', 'cat', 'chair', 'face', 'house', 'scissors
Counter-balance table for orders up to 2:
Targets/Order O1                        |  O2                                 |
   bottle:    96  2  1  2  2  3  0  2  |  84  4  2  4  4  6  0  4  |
     cat:      2 96  1  1  1  1  4  2  |   4 84  2  2  2  2  8  4  |
    chair:     2  3 96  1  1  2  1  2  |   4  6 84  2  2  4  2  4  |
    face:      0  3  3 96  1  1  2  2  |   0  6  6 84  2  2  4  4  |
    house:     0  1  2  2 96  2  4  1  |   0  2  4  4 84  4  8  2  |
  scissors:    3  0  2  3  1 96  0  2  |   6  0  4  6  2 84  0  4  |
scrambledpix:  2  1  1  2  3  2 96  1  |   4  2  2  4  6  4 84  2  |
    shoe:      3  2  2  1  3  0  1 96  |   6  4  4  2  6  0  2 84  |
Correlations: min=-0.3 max=0.87 mean=-0.0012 sum(abs)=59
```

TODO

Exercise

Generate few 'designs' consisting of varying condition sequences and assess their counter-balance. Generate some random designs using random number generators or permutation functions provided in `numpy.random` *and assess their counter-balance.*

Some sources of confounds might be hard to detect or to eliminate:

- dependent variable is assessed after data has been collected (RT, ACC, etc) so it might be hard to guarantee equal sampling across different splits of the data.
- motion effects, if motion is correlated with the design, might introduce major confounds into the signal. With multivariate analysis the problem becomes even more sever due to the high sensitivity of multivariate methods and the fact that motion effects might be impossible to eliminate entirely since they are strongly non-linear. So, even if you regress out whatever number of descriptors describing motion (mean displacement, angles, shifts, etc.) you would not be able to eliminate motion effects entirely. And that residual variance from motion spread through the entire volume might contribute to your *generalization performance*.

Exercise

Inspect the arguments of generic interface of all splitters `Splitter` *for a possible workaround in the case of dis-balanced targets.*

Therefore, before the analysis on the actual fMRI data, it might be worth inspecting what kind of *generalization* performance you might obtain if you operate simply on the confounds (e.g. motion parameters and effects).

4.11.7 Hypothesis Testing

Note:

When thinking about what critical value to choose for NHST keep such guidelines from NHST inventor, Dr.Fisher *in mind. For significance range '0.2 - 0.5' he says: "judged significant, though barely so; ... these data do not, however, demonstrate the point beyond possibility of doubt".*

Ways to assess *by-chance* null-hypothesis distribution of measures range from fixed, to estimated parametric, to non-parametric permutation testing. Unfortunately not a single way provides an ultimate testing facility to be applied blindly to any chosen problem without investigating the appropriateness of the data at hand (see previous section). Every kind of `Measure` provides an easy way to trigger assessment of *statistical significance* by specifying `null_dist` parameter with a distribution estimator. After a given measure is computed, the corresponding p-value(s) for the returned value(s) could be accessed at `ca.null_prob`.

"Applications of permutation testing methods to single subject fMRI require modelling the temporal autocorrelation in the time series."

Exercise

Try to assess significance of the finding on two problematic categories from 8-categories dataset without averaging the samples within the blocks of the same target. Even non-parametric test should be overly optimistic (forgotten ***exchangeability*** *requirement for parametric testing, such as multiple samples within a block for a block design)... TODO*

Independent Samples

Since "voodoo correlations" paper, most of the literature in brain imaging is seems to became more careful in avoiding "double-dipping" and keeping their testing data independent from training data, which is one of the major concerns for doing valid hypothesis testing later on. Not much attention is given though to independence of samples aspect – i.e. not only samples in testing set should be independent from training ones, but, to make binomial distribution testing valid, testing samples should be independent from each other as well. The reason is simple – number of the testing samples defines the width of the null-chance distribution, but consider the limiting case where all testing samples are heavily non-independent, consider them to be a 1000 instances of the same sample. Canonical binomial distribution would be very narrow, although effectively it is just 1 independent sample being tested, thus ... TODO

4.11.8 Statistical Treatment of Sensitivities

Note:

Statistical learning is about constructing reliable models to describe the data, and not really to reason either data is noise.

Note:

How do we decide to threshold sensitivities, remind them searchlight results with strong bimodal distributions, distribution outside of the brain as a true by-chance. May be reiterate that sensitivities of bogus model are bogus

Moreover, constructed mapping with barely *above-chance* performance is often further analyzed for its *sensitivity to the input variables*.

4.11.9 References

Cohen, J. (1994)
Classical critic of null hypothesis significance testing

Fisher, R. A. (1925)
One of the 20th century's most influential books on statistical methods, which coined the term 'Test of significance'.

Ioannidis, J. (2005)
Simulation study speculating that it is more likely for a research claim to be false than true. Along the way the paper highlights aspects to keep in mind while assessing the 'scientific significance' of any given study, such as, viability, reproducibility, and results.

Nichols et al. (2002)
Overview of standard nonparametric randomization and permutation testing applied to neuroimaging data (e.g. fMRI)

Wright, D. (2009)
Historical excurse into the life of 10 prominent statisticians of XXth century and their scientific contributions.

CHAPTER

FIVE

MISCELLANEOUS

Import helper for PyMVPA misc modules

5.1 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 `mvpa2` module is imported. It can be used from any script like this:

```
>>> from mvpa2 import cfg
```

By default the config manager reads settings from two config files (if any of them exists). The first is a file named `pymvpa2.cfg` and located in the user's home directory. The second is `pymvpa2.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/pymvpa2.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)
True
```

queries the config manager whether warnings should be suppressed (i.e. if there is a variable `suppress` in section `warnings`). In case, there is no such setting, i.e. neither config files nor environment variables defined it, the `default` values is returned. Please see the documentation of ConfigManager for its full functionality.

The source tarballs includes an example configuration file (`doc/examples/pymvpa2.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]
# causes the output of __str__ to be truncated to the given number of
# characters
truncate str = 200
# causes the output of __repr__ to be truncated to the given number of
# characters
truncate repr = 200

#
# XXX The previous verbose section and verbose() should better be called info()
#     to not collide with the concept of verbosity. We would have messages of
#     types: debug, info, warning, error (familiar concepts that all can be
#     subsumed as verbosity.


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

[error]
#output =

[warnings]
# integer
#bt =
# integer
#count =
# comma-separated list of handlers, e.g. stdout
#output =
# Boolean (former: MVPA_NO_WARNINGS)
```

```
suppress = no

[debug]
# comma-separated list of handlers, e.g. stdout
#output =
#metrics =
# either to use custom (improved) exception handler to report
# information about pymvpa useful during bug reporting
#wtf = no
#cmdline = no

[examples]
interactive = yes

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

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

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

# Control over warnings spit out by R modules.  From help(options) If
#  'warn' is negative all warnings are ignored.  If 'warn' is zero
#  (the default) warnings are stored until the top-level function
#  returns. ... If 'warn' is one, warnings are printed as they occur.
#  If 'warn' is two or larger all warnings are turned into errors.
# By default we want no warnings
warn = -1

[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

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

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

[data]
# root directory where datasets from pymvpa.org reside. By default this is going
# to be a directory 'data' in the installation path of PyMVPA
#root =

[datasets]
# repr by default prints a complete content of the Dataset so it could
# be inspected or stored as a string.  For large datasets it might be
# an overwhelming amount of textual information, so possible options are possible
# full -- default, entire content; str -- use __str__ for __repr__.
# Option is in effect at import time, i.e. change of it wouldn't effect after dataset
# has already being loaded
repr = full
```

5.2 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

5.2.1 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 mvpa2.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
>>>
```

5.2.2 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

5.2.3 Warning Messages

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

- `MVPA_WARNINGS_COUNT` = controls 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` = 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.

5.2.4 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.

5.2.5 PyMVPA Status Summary

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

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

5.3 Additional Little Helpers

5.3.1 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 =`. 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
...
```

5.3.2 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'.

5.4 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 np
>>>
>>> # some dummy dataset
>>> from mvpa2.datasets import Dataset
>>> ds = Dataset(samples=np.random.normal(size=(19, 3)))
>>>
>>> # load motion correction output
>>> from mvpa2.misc.fsl.base import McFlirtParams
>>> mc = McFlirtParams(path.join('mvpa2', 'data', 'bold_mc.par'))
>>>
>>> # simple plot using pylab (use pylab.show() or pylab.savefig()
>>> # afterwards)
>>> mc.plot()
>>>
>>> # merge the correction parameters into the dataset itself
>>> for param in mc:
...     ds.sa['mc_' + param] = mc[param]
```

```
>>>
>>> # detrend some dataset with mc params as additonal regressors
>>> from mvpa2.mappers.detrend import poly_detrend
>>> res = poly_detrend(ds, opt_regs=['mc_x', 'mc_y', 'mc_z',
...                                  'mc_rot1', 'mc_rot2', 'mc_rot3'])
>>> # 'res' contains all regressors and their associated weights
```

All FSL bindings are located in the mvpa2.misc.fsl module.

CHAPTER

SIX

EXAMPLE ANALYSES AND SCRIPTS

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.

All examples listed here utilize the Python API of PyMVPA. Additional examples that demonstrate the command line interface are also available.

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.py`). Alternatively, one can download a full example dataset.

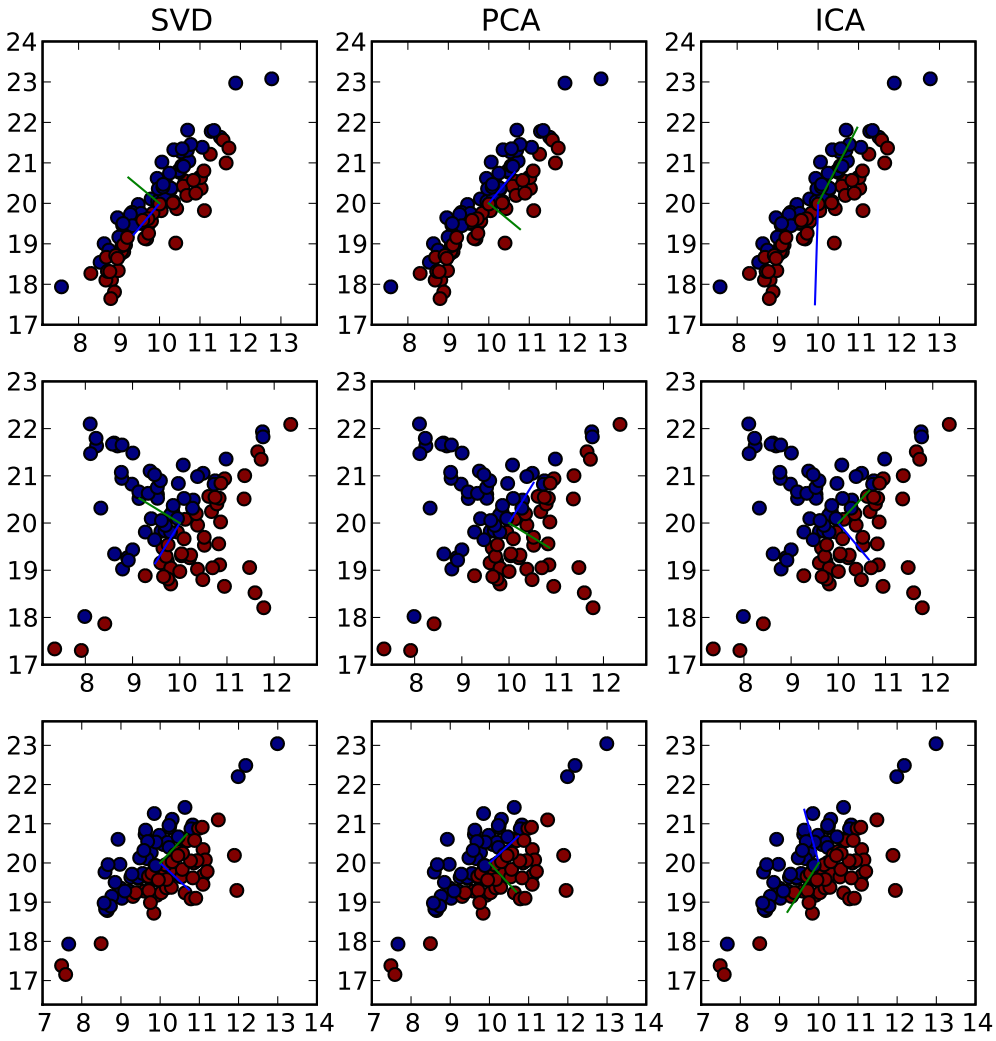
6.1 Preprocessing

6.1.1 Visualization of Data Projection Methods

```
from mvpa2.support.pylab import pl
from mvpa2.misc.data_generators import noisy_2d_fx
from mvpa2.mappers.svd import SVDMapper
from mvpa2.mappers.mdp_adaptor import ICAMapper, PCAMapper
from mvpa2 import cfg

center = [10, 20]
axis_range = 7

##REF: Name was automagically refactored
def plot_proj_dir(p):
    pl.plot([0, p[0,0]], [0, p[0,1]],
            linewidth=3, hold=True, color='y')
    pl.plot([0, p[1,0]], [0, p[1,1]],
            linewidth=3, hold=True, color='k')

mappers = {
             'PCA': PCAMapper(),
             'SVD': SVDMapper(),
             'ICA': ICAMapper(alg='CuBICA'),
           }
datasets = [
    noisy_2d_fx(100, lambda x: x, [lambda x: x],
                center, noise_std=0.5),
    noisy_2d_fx(50, lambda x: x, [lambda x: x, lambda x: -x],
                center, noise_std=0.5),
    noisy_2d_fx(50, lambda x: x, [lambda x: x, lambda x: 0],
                center, noise_std=0.5),
   ]

ndatasets = len(datasets)
```

```
nmappers = len(mappers.keys())

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

        pl.scatter(ds.samples[:, 0] - center[0],
                   ds.samples[:, 1] - center[1],
                   s=30, c=(ds.sa.targets) * 200)
        plot_proj_dir(mproj)
        fig += 1
```

Output of the example:



See also:

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

6.1.2 Simple Data-Exploration

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

```
from mvpa2.suite import *

# load example fmri dataset
ds = load_example_fmri_dataset()

# only use the first 5 chunks to save some cpu-cycles
ds = ds[ds.chunks < 5]
```

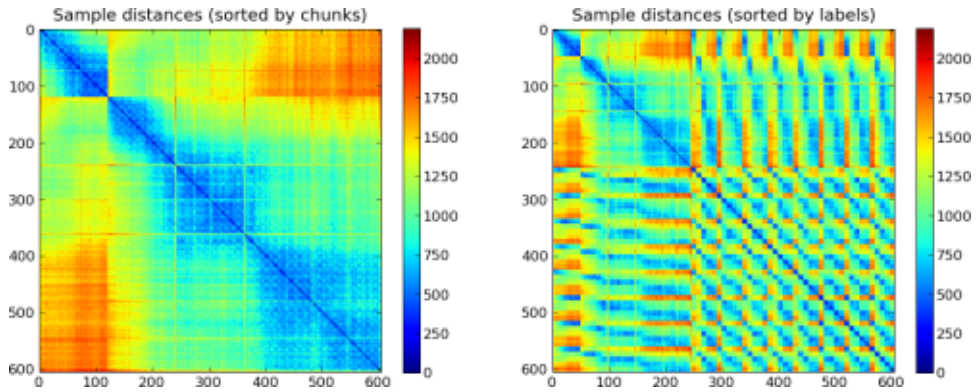
It is always useful to have a quick look at the summary of the dataset and verify that statistics (mean, standard deviation) are in the expected range, that there is balance among targets/chunks, and that order is balanced (where appropriate).

```
print ds.summary()
```

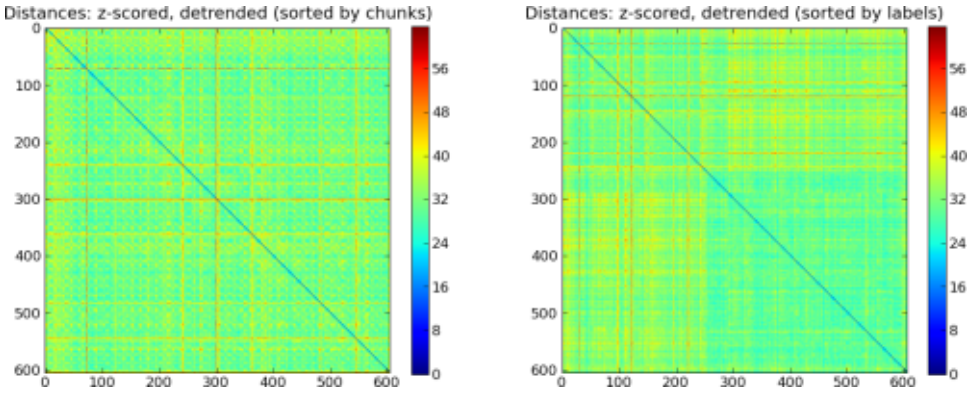
Now we can take a look at the distribution of the feature values in all sample categories and chunks.

```
pl.figure(figsize=(14, 14)) # larger figure
hist(ds, xgroup_attr='chunks', ygroup_attr='targets', noticks=None,
     bins=20, normed=True)

# next only works with floating point data
ds.samples = ds.samples.astype('float')

# look at sample similarity
# Note, the decreasing similarity with increasing temporal distance
# of the samples
pl.figure(figsize=(14, 6))
pl.subplot(121)
plot_samples_distance(ds, sortbyattr='chunks')
pl.title('Sample distances (sorted by chunks)')

# similar distance plot, but now samples sorted by their
# respective targets, i.e. samples with same targets are plotted
# in adjacent columns/rows.
# Note, that the first and largest group corresponds to the
# 'rest' condition in the dataset
pl.subplot(122)
plot_samples_distance(ds, sortbyattr='targets')
pl.title('Sample distances (sorted by targets)')

# z-score features individually per chunk
print 'Detrending data'
poly_detrend(ds, polyord=2, chunks_attr='chunks')
print 'Z-Scoring data'
zscore(ds)

pl.figure(figsize=(14, 6))
pl.subplot(121)
plot_samples_distance(ds, sortbyattr='chunks')
pl.title('Distances: z-scored, detrended (sorted by chunks)')
pl.subplot(122)
plot_samples_distance(ds, sortbyattr='targets')
pl.title('Distances: z-scored, detrended (sorted by targets)');
# 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`).

6.2 Analysis strategies and Background

6.2.1 A simple start

Here we show how to perform a simple cross-validated classification analysis with PyMVPA. This script is the exact equivalent of the example_cmdline_start_easy example, but using the Python API instead of the command line interface.

First, we import the PyMVPA suite to enable all PyMVPA building blocks

```
from mvpa2.suite import *
```

Now we load an example fMRI dataset that comes with PyMVPA. It has some attributes associated with each volume, and is masked to exclude voxels outside of the brain.

```
# load PyMVPA example dataset with literal labels
dataset = load_example_fmri_dataset(literal=True)
```

Next we remove linear trends by polynomial regression for each voxel and each chunk (recording run) of the dataset individually.

```
poly_detrend(dataset, polyord=1, chunks_attr='chunks')
```

For this example we are only interested in data samples that correspond to the `face` or to the `house` condition.

```
dataset = dataset[np.array([l in ['face', 'house'] for l in dataset.sa.targets],
                           dtype='bool')]
```

The setup for our cross-validation analysis include the selection of a classifier, and a partitioning scheme, and an error function to convert literal predictions into a quantitative performance metric.

```
cv = CrossValidation(SMLR(), OddEvenPartitioner(), errorfx=mean_mismatch_error)
error = cv(dataset)
```

The resulting dataset contains the computed accuracy.

```
# UC: unique chunks, UT: unique targets
print "Error for %i-fold cross-validation on %i-class problem: %f" \
      % (len(dataset.UC), len(dataset.UT), np.mean(error))
```

See also:

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

6.2.2 Separating hyperplane tutorial

This is a very introductory tutorial, showing how a classification task (in this case, deciding whether people are sumo wrestlers or basketball players, based on their height and weight) can be viewed as drawing a decision boundary in a feature space. It shows how to plot the data, calculate the weights of a simple linear classifier, and see how the resulting classifier carves up the feature space into two categories.

Note:

This tutorial was originally written by Rajeev Raizada for Matlab and was ported to Python by the PyMVPA authors. The original Matlab code is available from: http://www.dartmouth.edu/~raj/Matlab/fMRI/classification_plane_tutorial.m

Let's look at a toy example: classifying people as either sumo wrestlers or basketball players, depending on their height and weight. Let's call the x-axis height and the y-axis weight

```
sumo_wrestlers_height = [ 4, 2, 2, 3, 4 ]
sumo_wrestlers_weight = [ 8, 6, 2, 5, 7 ]
basketball_players_height = [ 3, 4, 5, 5, 3 ]
basketball_players_weight = [ 2, 5, 3, 7, 3 ]
```

Let's plot this.

```
import pylab as pl
pl.plot(sumo_wrestlers_height, sumo_wrestlers_weight, 'ro',
       linewidth=2, label="Sumo wrestlers")
pl.plot(basketball_players_height, basketball_players_weight, 'bx',
       linewidth=2, label="Basketball players")
pl.xlim(0, 6)
pl.ylim(0, 10)
pl.xlabel('Height')
pl.ylabel('Weight')
pl.legend()
```

Let's stack up the sumo data on top of the basketball players data.

```
import numpy as np

# transpose to have observations along the first axis
sumo_data = np.vstack((sumo_wrestlers_height,
                       sumo_wrestlers_weight)).T
# same for the baskball data
basketball_data = np.vstack((basketball_players_height,
                             basketball_players_weight)).T
# now stack them all together
all_data = np.vstack((sumo_data, basketball_data))
```

In order to be able to train a classifier on the input vectors, we need to know what the desired output categories are for each one. Let's set this to be `+1` for sumo wrestlers, and `-1` for basketball players.

```
# creates: [  1,  1,  1,  1,  1, -1, -1, -1, -1, -1]
all_desired_output = np.repeat([1, -1], 5)
```

We want to find a linear decision boundary, i.e. simply a straight line, such that all the data points on one side of the line get classified as sumo wrestlers, i.e. get mapped onto the desired output of `+1`, and all the data points on the other side get classified as basketball players, i.e. get mapped onto the desired output of `-1`.

The equation for a straight line has this form:

$$\vec{w}\mathbf{D} - \vec{b} = 0$$

were $\vec{w}$ is a weight vector, $\mathbf{D}$ is the data matrix, and $\vec{b}$ is the offset of the dataset form the origin, or the bias. We're not so interested for now in $\vec{b}$, so we can get rid of that by subtracting the mean from our data to get $\mathbf{D_C}$ the per-column (i.e. variable) demeaned data that is now centered around the origin.

```
zero_meaned_data = all_data - all_data.mean(axis=0)
```

Now, having gotten rid of that annoying bias term, we want to find a weight vector which gives us the best solution that we can find to this equation:

$$\mathbf{D_C}\vec{w} = \vec{o}$$

were $\vec{o}$ is the desired output, or the class labels. But, there is no such perfect set of weights. We can only get a best fit, such that

$$\mathbf{D_C}\vec{w} = \vec{o} + \vec{e}$$

where the error term $\vec{e}$ is as small as possible.

Note that our equation

$$\mathbf{D_C}\vec{w} = \vec{o}$$

has exactly the same form as the equation from the tutorial code in http://www.dartmouth.edu/~raj/Matlab/fMRI/design_matrix_tutorial.m which is:

$$\mathbf{X}\vec{\beta} = \vec{y}$$

where $\mathbf{X}$ was the design matrix, $\vec{\beta}$ the sensitivity vector, and $\vec{y}$ the voxel response.

The way we solve the equation is exactly the same, too. If we could find a matrix-inverse of the data matrix, then we could pre-multiply both sides by that inverse, and that would give us the weights:

$$\mathbf{D_C^{-1}}\mathbf{D_C}\vec{w} = \mathbf{D_C^{-1}}\vec{o}$$

The $\mathbf{D_C^{-1}}$ and $\mathbf{D_C}$ terms on the left would cancel each other out, and we would be left with:

$$\vec{w} = \mathbf{D_C^{-1}}\vec{o}$$

However, unfortunately there will in general not exist any matrix-inverse of the data matrix $\mathbf{D_C}$. Only square matrices have inverses, and not even all of them do. Luckily, however, we can use something that plays a similar role, called a pseudo-inverse. In Numpy, this is given by the command `pinv`. The pseudo-inverse won't give us a perfect solution to the above equation but it will give us the best approximate solution, which is what we want.

So, instead of

$$\vec{w} = \mathbf{D_C^{-1}}\vec{o}$$

we have this equation:

```
# compute pseudo-inverse as a matrix
pinv = np.linalg.pinv(np.mat(zero_meaned_data))
# column-vector of observations
y = all_desired_output[np.newaxis].T

weights = pinv * y
```

Let's have a look at how these weights carve up the input space A useful command for making grids of points which span a particular 2D space is called "meshgrid"

```
gridspec = np.linspace(-4, 4, 20)
input_space_X, input_space_Y = np.meshgrid(gridspec, gridspec)

# for the rest it is easier to have `weights` as a simple array, instead
# of a matrix
weights = weights.A

weighted_output_Z = input_space_X * weights[0] + input_space_Y * weights[1]
```

The weighted output gets turned into the category-decision `+1` if it is greater than 0, and `-1` if it is less than zero. Let's plot the decision surface color-coded and then plot the zero-meaned sumo and basketball data on top.

```
pl.figure()
pl.pcolor(input_space_X, input_space_Y, weighted_output_Z,
          cmap=pl.cm.Spectral)
pl.plot(zero_meaned_data[all_desired_output == 1, 0],
        zero_meaned_data[all_desired_output == 1, 1],
        'ro', linewidth=2, label="Sumo wrestlers")
pl.plot(zero_meaned_data[all_desired_output == -1, 0],
        zero_meaned_data[all_desired_output == -1, 1],
        'bx', linewidth=2, label="Basketball players")
pl.xlim(-4, 4)
pl.ylim(-4, 4)
pl.colorbar()
pl.xlabel('Demeaned height')
pl.ylabel('Demeaned weight')
pl.title('Decision output')
pl.legend()


from mvpa2.base import cfg
```

See also:

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

6.2.3 Minimal Searchlight Example

The term `Searchlight` refers to an algorithm that runs a scalar `Measure` 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-validation of a classifier performance (see *CrossValidation* section in the tutorial). 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.

```
import numpy as np

from mvpa2.generators.partition import OddEvenPartitioner
from mvpa2.clfs.svm import LinearCSVMC
from mvpa2.measures.base import CrossValidation
from mvpa2.measures.searchlight import sphere_searchlight
from mvpa2.testing.datasets import datasets
from mvpa2.mappers.fx import mean_sample
```

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

```
# Lets just use our tiny 4D dataset from testing battery
dataset = datasets['3dlarge']
```

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 = CrossValidation(LinearCSVMC(), OddEvenPartitioner())

# setup searchlight with 2 voxels radius and measure configured above
sl = sphere_searchlight(cv, radius=2, space='myspace',
                        postproc=mean_sample())

# run searchlight on dataset
sl_map = sl(dataset)
```

```
print 'Best performing sphere error:', np.min(sl_map.samples)
```

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

See also:

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

6.2.4 Searchlight on fMRI data

The original idea of a spatial searchlight algorithm stems from a paper by *Kriegeskorte et al. (2006)*, and has subsequently been used in a number of studies. The most common use for a searchlight is to compute a full cross-validation analysis in each spherical region of interest (ROI) in the brain. This analysis yields a map of (typically) classification accuracies that are often interpreted or post-processed similar to a GLM statistics output map (e.g. subsequent analysis with inferential statistics). In this example we look at how this type of analysis can be conducted in PyMVPA.

As always, we first have to import PyMVPA.

```
from mvpa2.suite import *
```

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

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

The next few calls load an fMRI dataset, while assigning associated class targets and chunks (experiment runs) to each volume in the 4D timeseries. One aspect is worth mentioning. When loading the fMRI data with `fmri_dataset()` additional feature attributes can be added, by providing a dictionary with names and source pairs to the `add_fa` arguments. In this case we are loading a thresholded zstat-map of a category selectivity contrast for voxels ventral temporal cortex.

```
# data path
datapath = os.path.join(mvpa2.cfg.get('location', 'tutorial data'), 'haxby2001')
dataset = load_tutorial_data(
        roi='brain',
        add_fa={'vt_thr_glm': os.path.join(datapath, 'sub001', 'masks',
                                           'orig', 'vt.nii.gz')})
```

The dataset is now loaded and contains all brain voxels as features, and all volumes as samples. To precondition this data for the intended analysis we have to perform a few preprocessing steps (please note that the data was already motion-corrected). The first step is a chunk-wise (run-wise) removal of linear trends, typically caused by the acquisition equipment.

```
poly_detrend(dataset, polyord=1, chunks_attr='chunks')
```

Now that the detrending is done, we can remove parts of the timeseries we are not interested in. For this example we are only considering volumes acquired during a stimulation block with images of houses and scrambled pictures, as well as rest periods (for now). It is important to perform the detrending before this selection, as otherwise the equal spacing of fMRI volumes is no longer guaranteed.

```
dataset = dataset[np.array([l in ['rest', 'house', 'scrambledpix']
                            for l in dataset.targets], dtype='bool')]
```

The final preprocessing step is data-normalization. This is a required step for many classification algorithms. It scales all features (voxels) into approximately the same range and removes the mean. In this example, we perform a chunk-wise normalization and compute standard deviation and mean for z-scoring based on the volumes

corresponding to rest periods in the experiment. The resulting features could be interpreted as being voxel salience relative to 'rest'.

```
zscore(dataset, chunks_attr='chunks', param_est=('targets', ['rest']), dtype='float32')
```

After normalization is completed, we no longer need the 'rest'-samples and remove them.

```
dataset = dataset[dataset.sa.targets != 'rest']
```

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.

```
# choose classifier
clf = LinearNuSVMC()

# setup measure to be computed by Searchlight
# cross-validated mean transfer using an N-fold dataset splitter
cv = CrossValidation(clf, NFoldPartitioner())
```

In this example, we do not want to compute full-brain accuracy maps, but instead limit ourselves to a specific subset of voxels. We'll select all voxel that have a non-zero z-stats value in the localizer mask we loaded above, as center coordinates for a searchlight sphere. These spheres will still include voxels that did not pass the threshold. the localizer merely define the location of all to be processed spheres.

```
# get ids of features that have a nonzero value
center_ids = dataset.fa.vt_thr_glm.nonzero()[0]
```

Finally, we can run the searchlight. We'll perform the analysis for three different radii, each time computing an error for each sphere. To achieve this, we simply use the `sphere_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 `sphere_searchlight()` object will do nothing more than generate small datasets for each sphere, feeding them to the processing object, and storing the result.

```
# setup plotting parameters (not essential for the analysis itself)
plot_args = {
    'background' : os.path.join(datapath, 'sub001', 'anatomy', 'highres001.nii.gz'),
    'background_mask' : os.path.join(datapath, 'sub001', 'masks', 'orig', 'brain.nii.gz'),
    'overlay_mask' : os.path.join(datapath, 'sub001', 'masks', 'orig', 'vt.nii.gz'),
    'do_stretch_colors' : False,
    'cmap_bg' : 'gray',
    'cmap_overlay' : 'autumn', # YlOrRd_r # pl.cm.autumn
    'interactive' : cfg.getboolean('examples', 'interactive', True),
    }

for radius in [0, 1, 3]:
    # tell which one we are doing
    print "Running searchlight with radius: %i ..." % (radius)
```

Here we actually setup the spherical searchlight by configuring the radius, and our selection of sphere center coordinates. Moreover, via the `space` argument we can instruct the searchlight which feature attribute shall be used to determine the voxel neighborhood. By default, `fmri_dataset()` creates a corresponding attribute called `voxel_indices`. Using the `mapper` argument it is possible to post-process the results computed for each sphere. Cross-validation will compute an error value per each fold, but here we are only interested in the mean error across all folds. Finally, on multi-core machines `nproc` can be used to enabled parallelization by setting it to the number of processes utilized by the searchlight (default value of `nproc`='`None` utilizes all available local cores).

```
»    sl = sphere_searchlight(cv, radius=radius, space='voxel_indices',
                             center_ids=center_ids,
                             postproc=mean_sample())
```

Since we care about efficiency, we are stripping all attributes from the dataset that are not required for the searchlight analysis. This will offers some speedup, since it reduces the time that is spent on dataset slicing.

```
»    ds = dataset.copy(deep=False,
                       sa=['targets', 'chunks'],
                       fa=['voxel_indices'],
                       a=['mapper'])
```

Finally, we actually run the analysis. The result is returned as a dataset. For the upcoming plots, we are transforming the returned error maps into accuracies.

```
»    sl_map = sl(ds)
     sl_map.samples *= -1
     sl_map.samples += 1
```

The result dataset is fully aware of the original dataspace. Using this information we can map the 1D accuracy maps back into "brain-space" (using NIfTI image header information from the original input timeseries.

```
»    niftiresults = map2nifti(sl_map, imghdr=dataset.a.imghdr)
```

PyMVPA comes with a convenient plotting function to visualize the searchlight maps. We are only looking at fMRI slices that are covered by the mask of ventral temproal cortex.

```
»    fig = pl.figure(figsize=(12, 4), facecolor='white')
     subfig = plot_lightbox(overlay=niftiresults,
                            vlim=(0.5, None), slices=range(23,31),
                            fig=fig, **plot_args)
     pl.title('Accuracy distribution for radius %i' % radius)
```

The following figures show the resulting accuracy maps for the slices covered by the ventral temporal cortex mask. Note that each voxel value represents the accuracy of a sphere centered around this voxel.

With radius 0 (only the center voxel is part of the part the sphere) there is a clear distinction between two distributions. The *chance distribution*, relatively symetric and centered around the expected chance-performance at 50%. The second distribution, presumambly of voxels with univariate signal, is nicely segregated from that. Increasing the searchlight size significantly blurrs the accuracy map, but also lead to an increase in classification accuracy.

See also:

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

6.2.5 Surface-based searchlight on fMRI data

This example employs a surface-based searchlight as described in *Oosterhof et al. (2011)* (with a minor difference that distances are currently computed using a Dijkstra distance metric rather than a geodesic one). For more details, see the Surfing website.

Surfaces used in this example are available in the tutorial dataset files; either the tutorial_data_surf_minimal or tutorial_data_surf_complete version. The surfaces were reconstructed using FreeSurfer and subsequently preprocessed with AFNI and SUMA using the pymvpa2-prep-afni-surf wrapper script in PyMVPA's 'bin' directory, which resamples the surfaces to standard topologies (with different resolutions) using MapIcosehedron, aligns surfaces to a reference functional volume, and merges left and right hemispheres into single surface files. A more detailed description of the steps that this script takes is provided in the documentation on the Surfing website.

If you use the surface-based searchlight code for a publication, please cite both *PyMVPA (2009)* and *Oosterhof et al. (2011)*.

As always, we first have to import PyMVPA.

```
from mvpa2.suite import *
from mvpa2.clfs.svm import LinearCSVMC
from mvpa2.base.hdf5 import h5save, h5load
```

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

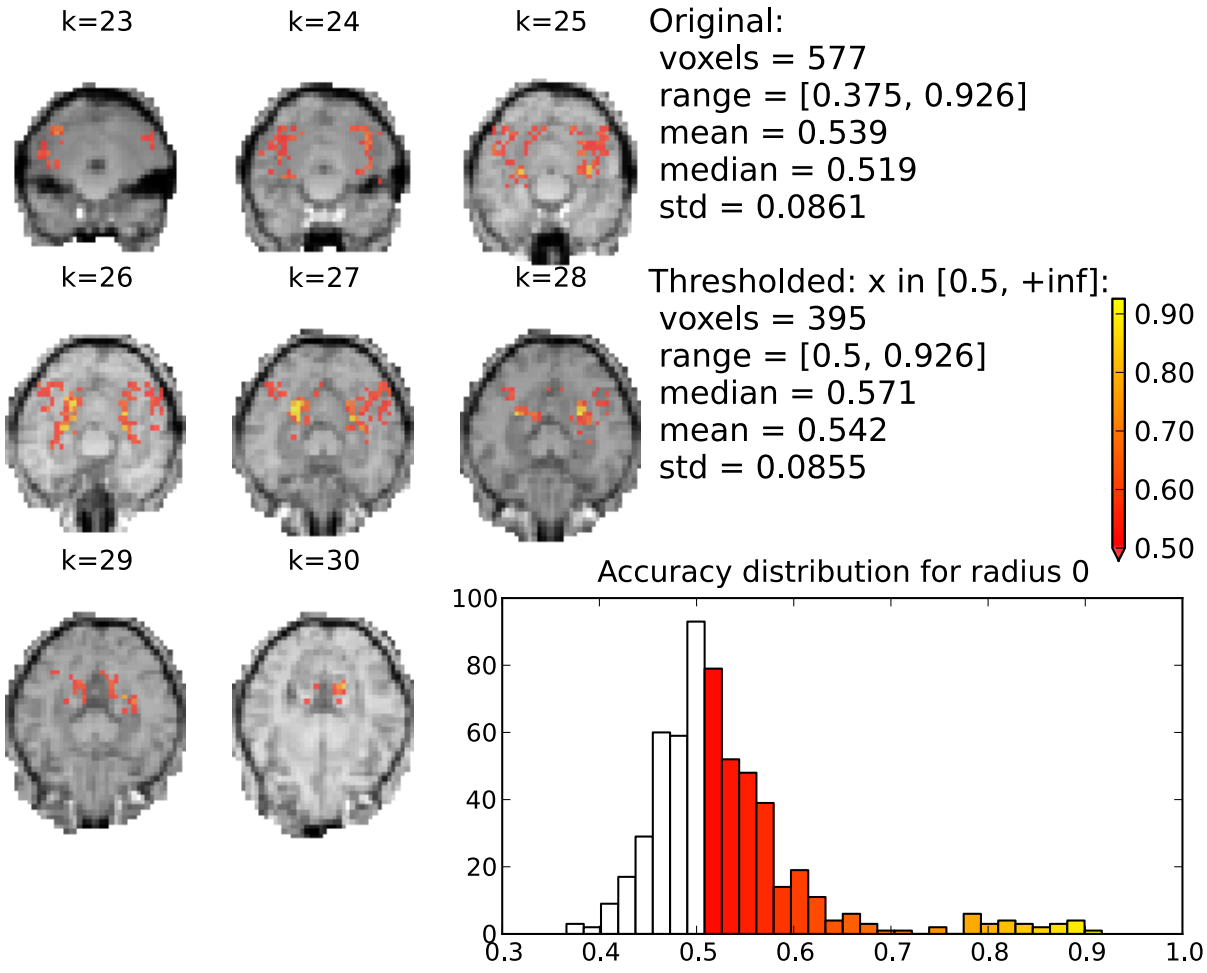


Fig. 6.1: Searchlight (single element; univariate) accuracy maps for binary classification *house* vs. *scrambledpix*.

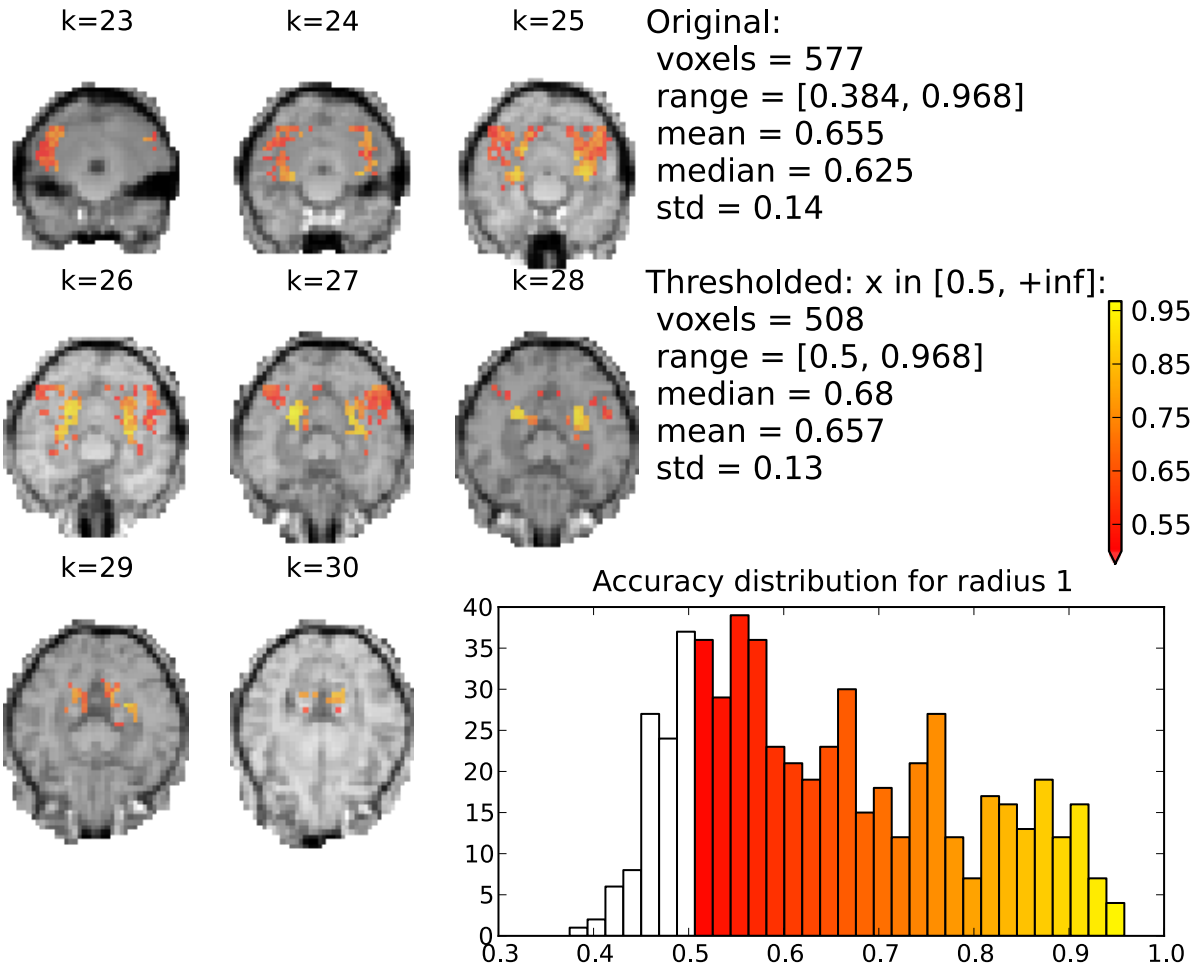


Fig. 6.2: Searchlight (sphere of neighboring voxels; 9 elements) accuracy maps for binary classification *house* vs. *scrambledpix*.

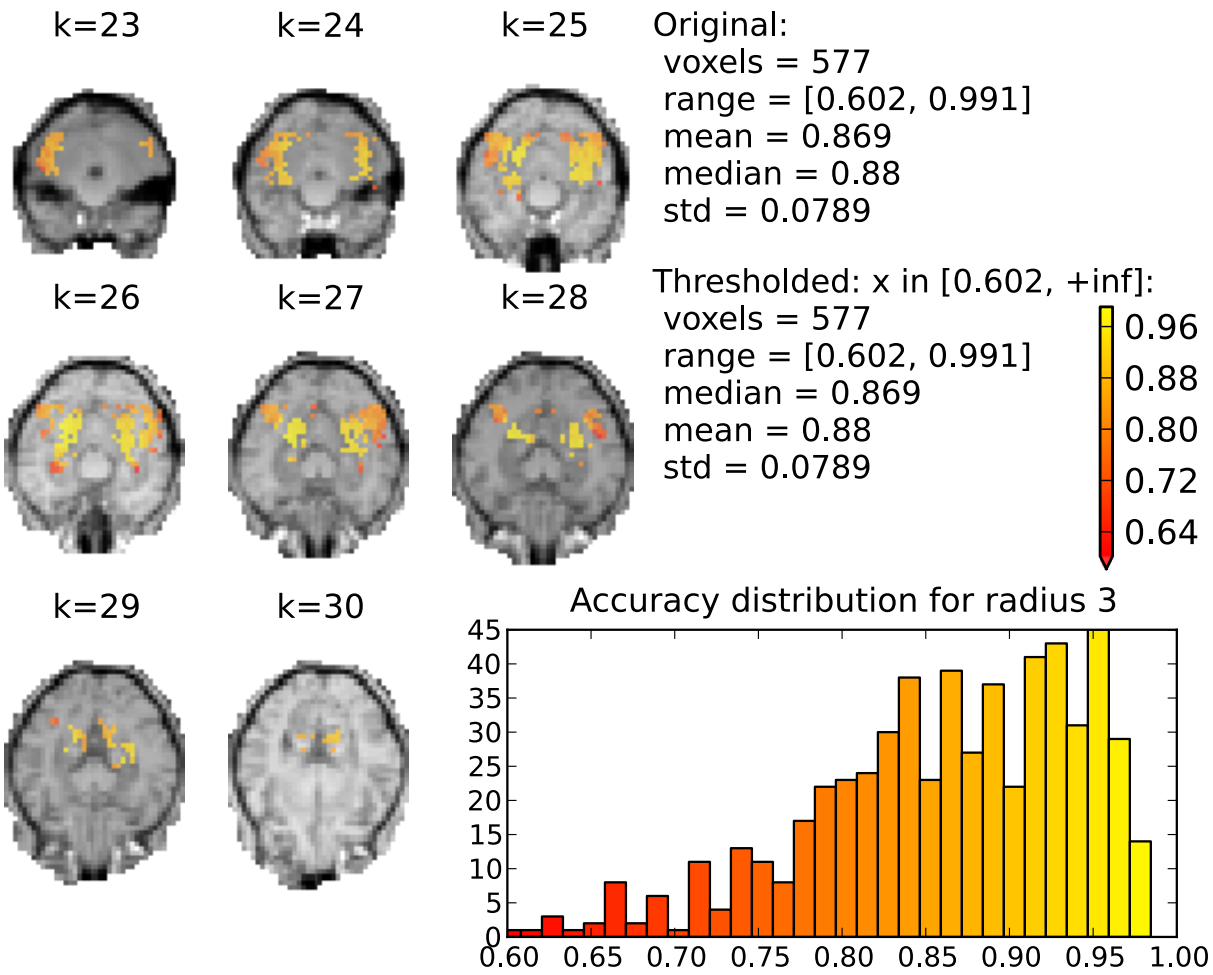


Fig. 6.3: Searchlight (radius 3 elements; 123 voxels) accuracy maps for binary classification *house* vs. *scrambledpix*.

```
if __debug__:
    from mvpa2.base import debug
    debug.active += ["SVS", "SLC"]
```

Define surface and volume data paths:

```
rootpath = os.path.join(pymvpa_datadbroot,
                        'tutorial_data', 'tutorial_data')

datapath = os.path.join(rootpath, 'data')
surfpath = os.path.join(rootpath, 'suma_surfaces')
```

Define functional data volume filename:

```
epi_fn = os.path.join(datapath, 'sub001', 'BOLD', 'task001_run001', 'bold.nii.gz')
```

In this example we are concerned with the left hemisphere only. (Other possible values are 'r' for the right hemisphere and 'm' for merged hemispheres; the latter contains the nodes from the left and right hemispheres in a single file. Both the 'r' and 'm' options require the tutorial_data_surf_complete tutorial data.)

```
hemi = 'l'
```

Define the surfaces that enclose the grey matter, which are used to delineate the grey matter. The pial surface is the 'outside' border of the grey matter; the white surface is the 'inside' border.

The surfaces in this example were resampled using AFNI's MapIcosahedron (for more details, see the top of this script). ld refers to the number of linear divisions of the twenty 'large' triangles of the original icosahedron; ld=x means there are 10*x**2+2 nodes (a.k.a. vertices) and 20*x**2 triangles (a.k.a. faces).

```
highres_ld = 128 # 64 or 128 is reasonable

pial_surf_fn = os.path.join(surfpath, "ico%d_%sh.pial_al.asc"
                                      % (highres_ld, hemi))
white_surf_fn = os.path.join(surfpath, "ico%d_%sh.smoothwm_al.asc"
                                       % (highres_ld, hemi))
```

Define the surface on which the nodes are centers of the searchlight. This surface should be an 'intermediate' surface, which is formed by the node-wise average spatial coordinates of the inner (white) and outer (pial) surfaces.

In this example a surface coarser (fewer nodes) than the grey matter-enclosing surfaces is employed. This reduces the number of searchlights and therefore the script's execution time. Of course one could also use a surface that has the same number of nodes as the grey-matter enclosing surfaces; this is actually the default and used when souce_surf_fn (assigned below) is set to None.

It is required that highres_ld is an integer multiple of lowres_ld, so that all nodes in the low-res surface have a corresponding node (i.e., with the same, or almost the same, spatial coordinate) on the high-res surface.

Choice of lowres_ld and highres_ld is somewhat arbitrary and a trade-off between spatial specificity and execution speed. For highres_ld a value of at least 64 is be advisable as this ensures enough anatomical detail is available to select voxels in the grey matter accurately. Typical values for lowres_ld range from 8 to 64.

Note that the data in tutorial_data_surf_minimal only contains all necessary surfaces for visualization for lowres_ld=16. For other values of lowres_ld (4, 8, 32, 64 and 128) the surfaces in tutorial_data_surf_complete are required.

```
lowres_ld = 16 # 16, 32 or 64 is reasonable. 4 and 8 are really fast

source_surf_fn = os.path.join(surfpath, "ico%d_%sh.intermediate_al.asc"
                                        % (lowres_ld, hemi))
```

Radius is specified as either an int (referring to a fixed number of voxels across searchlights, with a variable radius in millimeters (or whatever unit is used in the files that define the surfaces), or a float (referring to the radius in millimeters, with a variable number of voxels).

Note that "a fixed number of voxels" in this context actually means an approximation, in that on average that number of voxels is selected but the actual number will vary slightly (typically in the range +/- 2 voxels)

```
radius = 100
```

We're all set to go to create a query engine that performs 'voxel selection', that is determines, for each node, which voxels are near it (that is, in the corresponding searchlight disc).

As a reminder, the only essential values we have set so far are the filenames of three surfaces (high-res inner and outer, and low-res source surface), the functional volume, and the searchlight radius.

Note that if the functional data was preprocessed and subsequently masked, voxel selection should take into account this mask. To do so, the instantiation of the query engine below takes an optional argument 'volume_mask' (which can be a PyMVPA dataset, a numpy array, a Nifti volume, or a string representing the file name of a Nifti volume). It is, however, recommended to *not* mask the functional data prior to voxel selection, because the voxel selection uses (implicitly) a mask based on the grey-matter enclosing surfaces already, and this mask is assumed to be more precise than typical volume-based masking implementations.

Also note that, as described above, the argument defining the low-res source surface can be omitted, in which case it is computed as the node-wise average of the white and pial surface.)

```
qe = disc_surface_queryengine(radius, epi_fn,
                              white_surf_fn, pial_surf_fn,
                              source_surf_fn)
```

Voxel selection is now completed; each node has been assigned a list of linear voxel indices in the searchlight. These result are stored in 'qe.voxsel' and can be saved with h5save for later re-use.

(Linear voxel indices mean that each voxel is indexed by a value between 0 (inclusive) and N (exclusive), where N is the number of voxels in the volume (N = NX * NY * NZ, where NX, NY and NZ are the number of voxels in the three spatial dimensions). For certain analyses one may want to index voxels by 'sub indices' (triples (i,j,k) with 0<=i<NX, 0<=j<=NY, and 0<=k<NZ) or spatial coordinates; conversions amongst linear and sub indices and spatial coordinates is provided by functions in the VolGeom (volume geometry) instance stored in 'qe.voxsel.volgeom'.)

From now on we follow the example as in doc/examples/searchlight.py.

First, cross-validation is defined using a (SVM) classifier.

```
clf = LinearCSVMC()

cv = CrossValidation(clf, NFoldPartitioner(),
                     errorfx=lambda p, t: np.mean(p == t),
                     enable_ca=['stats'])
```

Set the roi_ids, that is the node indices that serve as searchlight center. In this example it is set to None, meaning that all nodes are used as a searchlight center. It is also possible to restrict the nodes that serve as a searchlight center: setting roi_ids=np.arange(400,800) means that only nodes in the range from 400 (inclusive) to 800 (exclusive) are used as a searchlight center, and the result would be a partial brain map.

```
roi_ids = None
```

Combining the query-engine and the cross-validation defines the searchlight. The postproc-step averages the classification accuracies in each cross-validation fold to a single overall classification accuracy.

Because roi_ids is None is this example it could be omitted - it is only included for instructive purposes.

```
sl = Searchlight(cv, queryengine=qe, postproc=mean_sample(), roi_ids=roi_ids)
```

In the next step the functional data is loaded. We can reduce memory requirements significantly by considering which voxels to load: since the searchlight analysis will only use data from voxels that were selected (at least once) by the voxel selection step, a mask is derived from the voxel selection results and used when loading the functional data.

```
mask = qe.voxsel.get_mask()
```

Load the functional data for subject 1 and the condition model 1 in the dataset (object viewing with 8 object categories). Note that we use the mask that came from the voxel selection.

```
model = 1
subject = 1
of = OpenFMRIDataset(datapath)
dataset = of.get_model_bold_dataset(model, subject,
                                    noinfolabel='rest', mask=mask)
```

Apply some typical preprocessing steps

```
poly_detrend(dataset, polyord=1, chunks_attr='chunks')

dataset = dataset[np.array([l in ['rest', 'house', 'scrambledpix']
                            for l in dataset.targets], dtype='bool')]

zscore(dataset, chunks_attr='chunks', param_est=('targets', ['rest']),
       dtype='float32')

dataset = dataset[dataset.sa.targets != 'rest']
```

Run the searchlight on the dataset.

```
sl_dset = sl(dataset)
```

Searchlight results are now stored in sl_dset. As sl_dset is just like any other PyMVPA dataset, it can be stored with h5save for future use.

The remainder of this example provides a data file that can be visualized using AFNI's SUMA. This is achieved by storing the dataset as an NIML (NeuroImaging Markup Language) dataset that can be viewed by AFNI's SUMA. sl_dset contains a feature attribute 'center_ids' that is automagically used to define the node indices of the searchlight centers in this NIML dataset.

Note that this conversion will not preserve all information in sl_dset but only the samples and (feature, sample, dataset) attributes that behave like arrays or strings or scalars. For example, in this example sl_dset has a dataset attribute 'mapper' which is not stored in the NIML dataset (and a warning message is printed during the conversion, which can be ignored savely). As mentioned above, using h5save will preserve this information (but its output cannot be viewed in SUMA).

Before saving the dataset, first the labels are set for each sample (in this case, only one) so that they show up in SUMA.

```
sl_dset.sa['labels'] = ['HOUSvsSCRM']
```

Set the filename for output. Searchlight results are stored in the surface directory for easy visualization. Finally print an informative message on how the generated data can be visualized using SUMA.

```
fn = 'ico%d-%d_%sh_%dvx.niml.dset' % (lowres_ld, highres_ld, hemi, radius)
path_fn = os.path.join(surfpath, fn)

niml.write(path_fn, sl_dset)

print ("To view results in SUMA, cd to '%s', run 'suma -spec "
       "%sh_ico%d_al.spec', press ctrl+s, "
        "click on 'Load Dset', and select %s" %
        (surfpath, hemi, lowres_ld, fn))
```

See also:

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

6.2.6 Representational similarity analysis (RSA) on fMRI data

In this example we are going to take a look at representational similarity analysis (RSA). This term was coined by *Kriegeskorte et al. (2008)* and refers to a technique were data samples are converted into a self-referential distance space, in order to aid comparsion across domains. The premise is that whenever no appropriate transformation is known to directly compare two types of data directly (1st-level), it is still useful to compare similarities computed in individual domains (2nd-level). For example, this analysis technique has been used to identify inter-species commonalities in brain response pattern variations during stimulation with visual objects (single-cell recordings in monkeys compared to human fMRI, Krigeskorte et al., 2008), and to relate brain response pattern similarities to predictions of computational models of perception (Connolly et al., 2012).

```
import numpy as np
import pylab as pl
from os.path import join as pjoin
from mvpa2 import cfg
```

In this example we use a dataset from *Haxby et al. (2001)* were participants watched pictures of eight different visual objects, while fMRI was recorded. The following snippet load a portion of this dataset (single subject) from regions on the ventral and occipital surface of the brain.

```
# load dataset -- ventral and occipital ROIs
from mvpa2.datasets.sources.native import load_tutorial_data
datapath = pjoin(cfg.get('location', 'tutorial data'), 'haxby2001')
ds = load_tutorial_data(roi=(15, 16, 23, 24, 36, 38, 39, 40, 48))
```

We only do minimal pre-processing: linear trend removal and Z-scoring all voxel time-series with respect to the mean and standard deviation of the "rest" condition.

```
# only minial detrending
from mvpa2.mappers.detrend import poly_detrend
poly_detrend(ds, polyord=1, chunks_attr='chunks')
# z-scoring with respect to the 'rest' condition
from mvpa2.mappers.zscore import zscore
zscore(ds, chunks_attr='chunks', param_est=('targets', 'rest'))
# now remove 'rest' samples
ds = ds[ds.sa.targets != 'rest']
```

RSA is all about so-called dissimilarity matrices: square, symetric matrices with a zero diagonal that encode the (dis)similarity between all pairs of data samples or conditions in a dataset. We compose a little helper function to plot such matrices, including a color-scale and proper labeling of matrix rows and columns.

```
# little helper function to plot dissimilarity matrices
def plot_mtx(mtx, labels, title):
    pl.figure()
    pl.imshow(mtx, interpolation='nearest')
    pl.xticks(range(len(mtx)), labels, rotation=-45)
    pl.yticks(range(len(mtx)), labels)
    pl.title(title)
    pl.clim((0,1))
    pl.colorbar()
```

As a start, we want to inspect the dissimilarity structure of the stimulation conditions in the entire ROI. For this purpose, we average all samples of each conditions into a single examplar, using an FxMapper() instance.

```
# compute a dataset with the mean samples for all conditions
from mvpa2.mappers.fx import mean_group_sample
mtgs = mean_group_sample(['targets'])
mtds = mtgs(ds)
```

After these preparations we can use the PDist() measure to compute the desired distance matrix – by default using correlation distance as a metric. The `square` argument will cause a ful square matrix to be produced, instead of a leaner upper-triangular matrix in vector form.

```
# basic ROI RSA -- dissimilarity matrix for the entire ROI
from mvpa2.measures import rsa
dsm = rsa.PDist(square=True)
res = dsm(mtds)
plot_mtx(res, mtds.sa.targets, 'ROI pattern correlation distances')
```

Inspecting the figure we can see that there is not much structure in the matrix, except for the face and the house condition being slightly more dissimilar than others.

Now, let's take a look at the variation of similarity structure through the brain. We can plug the PDist() measure into a searchlight to quickly scan the brain and harvest this information.

```
# same as above, but done in a searchlight fashion
from mvpa2.measures.searchlight import sphere_searchlight
dsm = rsa.PDist(square=False)
sl = sphere_searchlight(dsm, 2)
slres = sl(mtds)
```

The result is a compact distance matrix in vector form for each searchlight location. We can now try to score each matrix. Let's find the distance matrix with the largest overall distances across all stimulation conditions, i.e. the location in the brain where brain response patterns are most dissimilar.

```
# score each searchlight sphere result wrt global pattern dissimilarity
distinctiveness = np.sum(np.abs(slres), axis=0)
print 'Most dissimilar patterns around', \
        mtds.fa.voxel_indices[distinctiveness.argmax()]
# take a look at the this dissimilarity structure
from scipy.spatial.distance import squareform
plot_mtx(squareform(slres.samples[:, distinctiveness.argmax()]),
         mtds.sa.targets,
         'Maximum distinctive searchlight pattern correlation distances')
```

That looks confusing. But how do we know that this is not just noise (it probably is)? One way would be to look at how stable a distance matrix is, when computed for different portions of a dataset.

To perform this analysis, we use another FxMapper() instance that averages all data into a single sample per stimulation conditions, per `chunk`. A chunk in this context indicates a complete fMRI recording run.

```
# more interesting: let's look at the stability of similarity sturctures
# across experiment runs
# mean condition samples, as before, but now individually for each run
mtcgs = mean_group_sample(['targets', 'chunks'])
mtcds = mtcgs(ds)
```

With this dataset we can use PDistConsistency() to compute the similarity of dissimilarity matrices computes from different chunks. And, of course, it can be done in a searchlight.

```
# searchlight consistency measure -- how correlated are the structures
# across runs
dscm = rsa.PDistConsistency()
sl_cons = sphere_searchlight(dscm, 2)
slres_cons = sl_cons(mtcds)
```

Now we can determine the most brain location with the most stable dissimilarity matrix.

```
# mean correlation
mean_consistency = np.mean(slres_cons, axis=0)
print 'Most stable dissimilarity patterns around', \
        mtds.fa.voxel_indices[mean_consistency.argmax()]
# Look at this pattern
plot_mtx(squareform(slres.samples[:, mean_consistency.argmax()]),
         mtds.sa.targets,
         'Most consistent searchlight pattern correlation distances')
```

It is all in the face!

It would be interesting to know where in the brain dissimilarity structures can be found that are similar to this one. PDistTargetSimilarity() can be used to discover this kind of information with any kind of target dissimilarity structure. We need to transpose the result for aggregation into a searchlight map, as PDistTargetSimilarity can return more features than just the correlation coefficient.

```
# let's see where in the brain we find dissimilarity structures that are
# similar to our most stable one
tdsm = rsa.PDistTargetSimilarity(
            slres.samples[:, mean_consistency.argmax()])
# using a searchlight
from mvpa2.base.learner import ChainLearner
from mvpa2.mappers.shape import TransposeMapper
sl_tdsm = sphere_searchlight(ChainLearner([tdsm, TransposeMapper()]), 2)
slres_tdsm = sl_tdsm(mtds)
```

Lastly, we can map this result back onto the 3D voxel grid, and overlay it onto the brain anatomy.

```
# plot the spatial distribution using NiPy
vol = ds.a.mapper.reverse1(slres_tdsm.samples[0])
import nibabel as nb
anat = nb.load(pjoin(datapath, 'sub001', 'anatomy', 'highres001.nii.gz'))

from nipy.labs.viz_tools.activation_maps import plot_map
pl.figure(figsize=(15,4))
sp = pl.subplot(121)
pl.title('Distribution of target similarity structure correlation')
slices = plot_map(
            vol,
            ds.a.imgaffine,
            cut_coords=np.array((12,-42,-20)),
            threshold=.5,
            cmap="bwr",
            vmin=0,
            vmax=1.,
            axes=sp,
            anat=anat.get_data(),
            anat_affine=anat.get_affine(),
         )
img = pl.gca().get_images()[1]
cax = pl.axes([.05, .05, .05, .9])
pl.colorbar(img, cax=cax)

sp = pl.subplot(122)
pl.hist(slres_tdsm.samples[0],
        #range=(0,410),
        normed=False,
        bins=30,
        color='0.6')
pl.ylabel("Number of voxels")
pl.xlabel("Target similarity structure correlation")
```

See also:

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

6.2.7 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 mvpa2.suite import *

# load PyMVPA example dataset with literal labels
dataset = load_example_fmri_dataset(literal=True)
print dataset.a
```

As with classifiers it is easy to define a bunch of sensitivity analyzers. It is usually possible to simply call `get_sensitivity_analyzer()` 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(postproc=absolute_features()),
    'b) Linear SVM weights': LinearNuSVMC().get_sensitivity_analyzer(
                                                postproc=absolute_features()),
    'c) I-RELIEF': IterativeRelief(postproc=absolute_features()),
    'd) Splitting ANOVA (odd-even)':
        RepeatedMeasure(
            OneWayAnova(postproc=absolute_features()),
            OddEvenPartitioner()),
    'e) Splitting SVM (odd-even)':
        RepeatedMeasure(
            LinearNuSVMC().get_sensitivity_analyzer(postproc=absolute_features()),
            OddEvenPartitioner()),
    'f) I-RELIEF Online':
        IterativeReliefOnline(postproc=absolute_features()),
    'g) Splitting ANOVA (nfold)':
        RepeatedMeasure(
            OneWayAnova(postproc=absolute_features()),
            NFoldPartitioner()),
    'h) Splitting SVM (nfold)':
        RepeatedMeasure(
            LinearNuSVMC().get_sensitivity_analyzer(postproc=absolute_features()),
            NFoldPartitioner()),
    'i) GNB Searchlight':
        sphere_gnbsearchlight(GNB(), NFoldPartitioner(cvtype=1),
                              radius=0, errorfx=mean_match_accuracy)
       }
```

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.

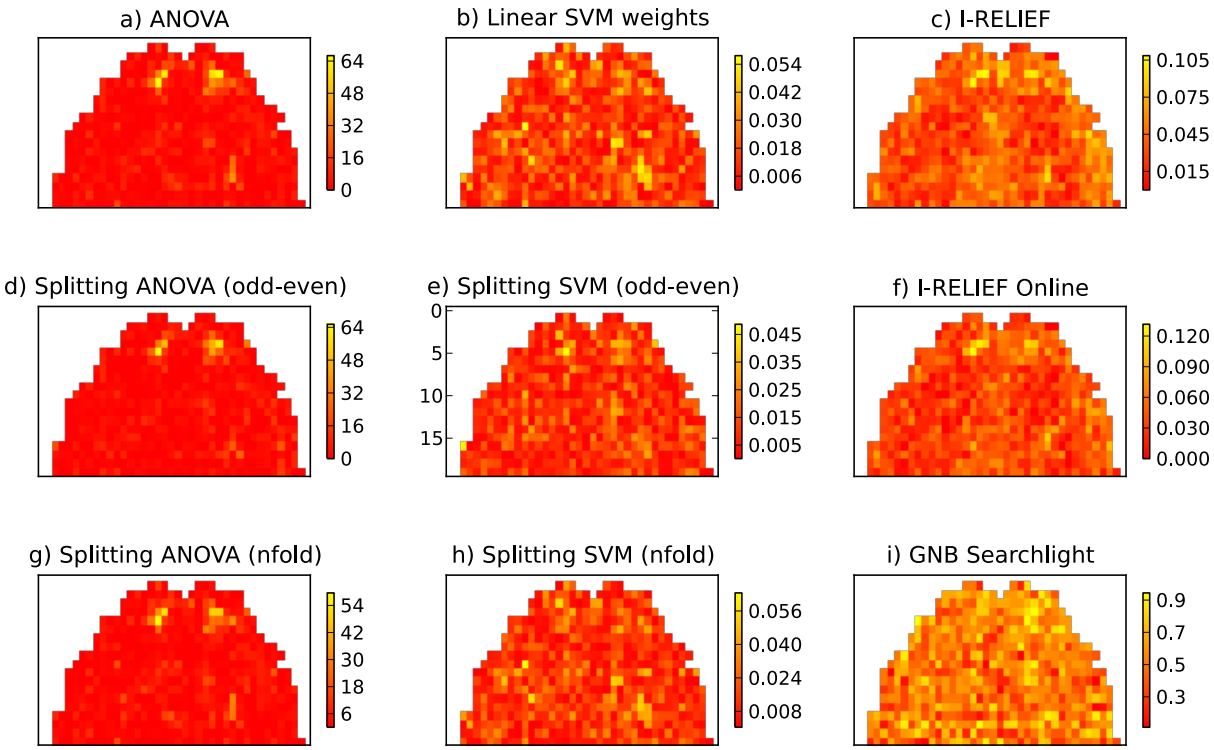
```
# do chunkswise linear detrending on dataset
poly_detrend(dataset, polyord=1, chunks_attr='chunks')

# only use 'rest', 'shoe' and 'bottle' samples from dataset
dataset = dataset[np.array([l in ['rest', 'shoe', 'bottle']
                    for l in dataset.sa.targets], dtype='bool')]

# zscore dataset relative to baseline ('rest') mean
zscore(dataset, chunks_attr='chunks',
       param_est=('targets', ['rest']), dtype='float32')

# remove baseline samples from dataset for final analysis
dataset = dataset[dataset.sa.targets != 'rest']
```

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
pl.figure(figsize=(14, 8))

keys = sensanas.keys()
```

```
keys.sort()

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

    sana = sensanas[s]
    # compute sensitivies
    sens = sana(dataset)
    # I-RELIEF assigns zeros, which corrupts voxel masking for pylab's
    # imshow, so adding some epsilon :)
    smap = sens.samples[0] + 0.00001

    # map sensitivity map into original dataspace
    orig_smap = dataset.mapper.reverse1(smap)
    masked_orig_smap = np.ma.masked_array(orig_smap, mask=orig_smap == 0)

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

    pl.title(s)

    pl.imshow(masked_orig_smap[..., 0].T,
              interpolation='nearest',
              aspect=1.25,
              cmap=pl.cm.autumn)

    # uniform scaling per base sensitivity analyzer
    ## if s.count('ANOVA'):
    ##     pl.clim(0, 30)
    ## elif s.count('SVM'):
    ##     pl.clim(0, 0.055)
    ## else:
    ##     pass

    pl.colorbar(shrink=0.6)
```

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`).

6.2.8 Classification of SVD-mapped Datasets

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

```
from mvpa2.suite import *

if __debug__:
    debug.active += ["REPM"]

#
# load PyMVPA example dataset
#
dataset = load_example_fmri_dataset(literal=True)

#
# preprocessing
#

# do chunkswise linear detrending on dataset
poly_detrend(dataset, polyord=1, chunks_attr='chunks')

# only use 'rest', 'cats' and 'scissors' samples from dataset
dataset = dataset[np.array([ l in ['rest', 'cat', 'scissors']
                    for l in dataset.targets], dtype='bool')]

# zscore dataset relative to baseline ('rest') mean
zscore(dataset, chunks_attr='chunks', param_est=('targets', ['rest']), dtype='float32')

# remove baseline samples from dataset for final analysis
dataset = dataset[dataset.sa.targets != 'rest']
```

```
# Specify the class of a base classifier to be used
Clf = LinearCSVMC
# And create the instance of SVDMapper to be reused
svdmapper = SVDMapper()
```

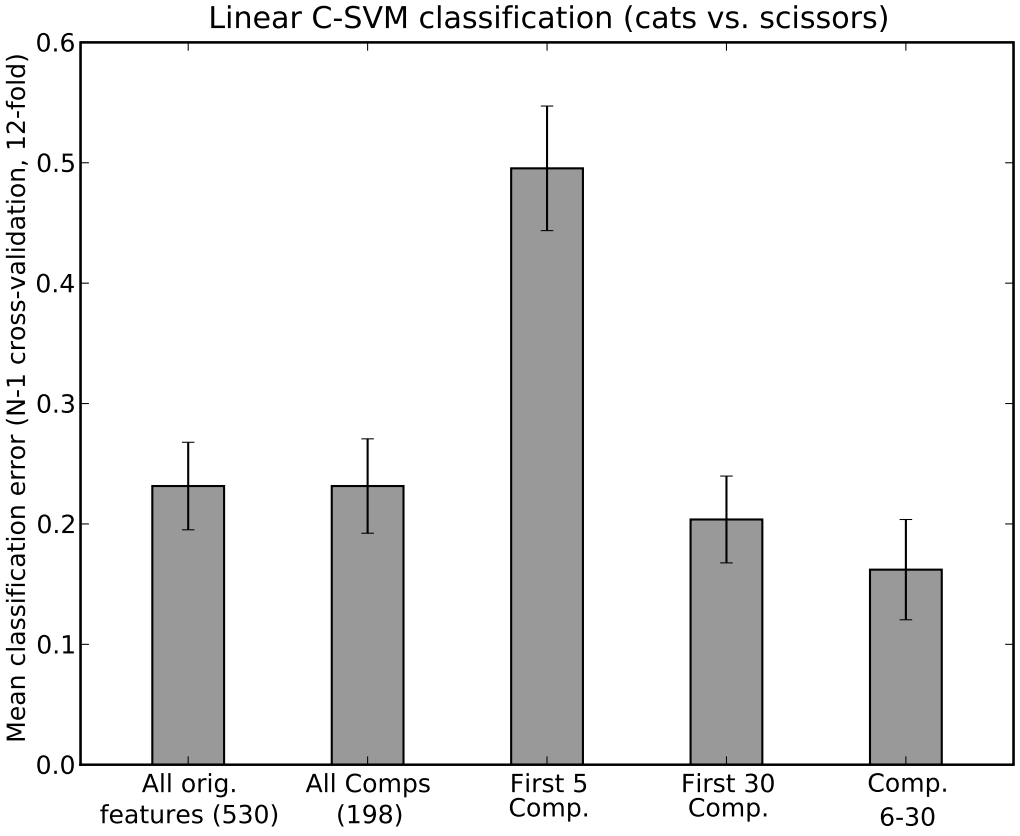
Lets create a generator of a `ChainMapper` which would first perform SVD and then subselect the desired range of components.

```
get_SVD_sliced = lambda x: ChainMapper([svdmapper,
                                        StaticFeatureSelection(x)])
```

Now we can define a list of some classifiers: a simple one and several classifiers with built-in SVD transformation and selection of corresponding SVD subspaces

```
clfs = [('All orig.\nfeatures (%i)' % dataset.nfeatures, Clf()),
        ('All Comps\n(%i)' % (dataset.nsamples \
                 - (dataset.nsamples / len(dataset.UC)),),
                      MappedClassifier(Clf(), svdmapper)),
        ('First 5\nComp.', MappedClassifier(Clf(),
                      get_SVD_sliced(slice(0, 5)))),
        ('First 30\nComp.', MappedClassifier(Clf(),
                      get_SVD_sliced(slice(0, 30)))),
        ('Comp.\n6-30', MappedClassifier(Clf(),
                      get_SVD_sliced(slice(5, 30))))]


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

for desc, clf in clfs:
    print desc.replace('\n', ' ')
    cv = CrossValidation(clf, NFoldPartitioner())
    res = cv(dataset)
    # there is only one 'feature' i.e. the error in the returned
    # dataset
    results.append(res.samples[:,0])
    labels.append(desc)

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

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`).

6.2.9 Monte-Carlo testing of Classifier-based Analyses

It is often desirable to be able to make statements like *"Performance is significantly above chance-level"* and to help with that PyMVPA supports *Null* hypothesis (aka *H0*) testing for any `Measure`. Measures take an optional constructor argument `null_dist` that can be used to provide an instance of some `NullDist` estimator. If the properties of the expected *Null* distribution are known a-priori, it is possible to use any distribution specified in SciPy's `stats` module for this purpose (see e.g. `FixedNullDist`).

However, as with other applications of statistics in classifier-based analyses there is the problem that we typically do not know the distribution of a variable like error or performance under the *Null* hypothesis (i.e. the probability of a result given that there is no signal), hence we cannot easily assign the adored p-values. 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 using permutation testing. The *Null* distribution is then estimated by computing the measure of interest multiple times using original data samples but with permuted targets. Since quite often the exploration of all permutations is unfeasible, Monte-Carlo testing (see *Nichols et al. (2006)*) allows to obtain stable estimate with only a limited number of random permutations.

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

Here is how this looks for a simple cross-validated classification in PyMVPA. We start by generated a dataset with 200 samples and 3 features of which 2 carry some relevant signal.

```
# lazy import
from mvpa2.suite import *

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

# some example data with signal
ds = normal_feature_dataset(perlabel=100, nlabels=2, nfeatures=3,
                            nonbogus_features=[0,1], snr=0.3, nchunks=2)
```

Now we can start collecting the pieces that play a role in this analysis. We need a classifier.

```
clf = LinearCSVMC()
```

We need a generator than will produce partitioned datasets, one for each fold of the cross-validation. A partitioned dataset is basically the same as the original dataset, but has an additional samples attribute that indicates whether particular samples will be the *part* of the data that is used for training the classifier, or for testing it. By default, the `NFoldPartitioner` will create a sample attribute `partitions` that will label one *chunk* in each fold differently from all others (hence mark it as taken-out for testing).

```
partitioner = NFoldPartitioner()
```

We need two pieces for the Monte Carlo shuffling. The first of them is an instance of an `AttributePermutator` that will permute the target attribute of the dataset for each iteration. We will instruct it to perform 200 permutations. In a real analysis the number of permutations should be larger to get stable estimates.

```
permutator = AttributePermutator('targets', count=200)
```

The second mandatory piece for a Monte-Carlo-style estimation of the *Null* distribution is the actual "estimator". `MCNullDist` will use the constructed `permutator` to shuffle the targets and later on report p-value from the left tail of the *Null* distribution, because we are going to compute errors and are interested in them being *lower* than chance. Finally we also ask for all results from Monte-Carlo shuffling to be stored for subsequent visualization of the distribution.

```
distr_est = MCNullDist(permutator, tail='left', enable_ca=['dist_samples'])
```

Now we have all pieces and can conduct the actual cross-validation. We assign a post-processing node `mean_sample` that will take care of averaging error values across all cross-validation fold. Consequently, the *Null* distribution of *average cross-validated classification error* will be estimated and used for statistical evaluation.

```
cv = CrossValidation(clf, partitioner,
                     errorfx=mean_mismatch_error,
                     postproc=mean_sample(),
                     null_dist=distr_est,
                     enable_ca=['stats'])
# run
err = cv(ds)
```

Now we have a usual cross-validation error and `cv` stores conditional attribute's such as confusion matrices:

```
print 'CV-error:', 1 - cv.ca.stats.stats['ACC']
```

However, in addition it also provides the results of the statistical evaluation. The *conditional attribute* `null_prob` has a dataset that contains the p-values representing the likelihood of an error equal or lower to the output one under the *Null* hypothesis, i.e. no actual relevant signal in the data. For a reason that will appear sensible later on, the p-value is contained in a dataset.

```
p = cv.ca.null_prob
# should be exactly one p-value
assert(p.shape == (1,1))
print 'Corresponding p-value:',  np.asscalar(p)
```

We can now look at the distribution of the errors under *H0* and plot the expected chance level as well as the empirical error.

```
# make new figure
pl.figure()
# histogram of all computed errors from permuted data
pl.hist(np.ravel(cv.null_dist.ca.dist_samples), bins=20)
# empirical error
pl.axvline(np.asscalar(err), color='red')
# chance-level for a binary classification with balanced samples
pl.axvline(0.5, color='black', ls='--')
# scale x-axis to full range of possible error values
pl.xlim(0,1)
pl.xlabel('Average cross-validated classification error')
```

We can see that the *Null* or chance distribution is centered around the expected chance-level and the empirical error value is in the far left tail, thus unlikely to belong to *Null* distribution, and hence the low p-value.

This could be the end, but sometimes one needs to have a closer look. Let's say your data is not that homogeneous. Let's say that some *chunk* may be very different from others. You might want to look at the error value probability for specific cross-validation folds. Sounds complicated? Luckily it is very simple. It only needs a tiny change in the cross-validation setup – the removal of the `mean_sample` post-processing node.

```
cv = CrossValidation(clf, partitioner,
                     errorfx=mean_mismatch_error,
                     null_dist=distr_est,
                     enable_ca=['stats'])
# run
err = cv(ds)

assert (err.shape == (2,1))
print 'CV-errors:', np.ravel(err)
```

Now we get two errors – one for each cross-validation fold and most importantly, we also get the two associated p-values.

```
p = cv.ca.null_prob
assert(p.shape == (2,1))
print 'Corresponding p-values:',  np.ravel(p)
```

What happened is that a dedicated *Null* distribution has been estimated for each element in the measure results. Without `mean_sample` an error is reported for each CV-fold, hence a separate distributions are estimated for each CV-fold too. And because we have also asked for all distribution samples to be reported, we can now plot both distribution and both empirical errors. But how do we figure out with value is which?

As mentioned earlier all results are returned in Datasets. All datasets have compatible sample and feature axes, hence corresponding elements.

```
assert(err.shape == p.shape == cv.null_dist.ca.dist_samples.shape[:2])

# let's make a function this time
def plot_cv_results(cv, err, title):
    # make new figure
    pl.figure()
    colors = ['green', 'blue']
    # null distribution samples
    dist_samples = np.asarray(cv.null_dist.ca.dist_samples)
    for i in range(len(err)):
        # histogram of all computed errors from permuted data per CV-fold
        pl.hist(np.ravel(dist_samples[i]), bins=20, color=colors[i],
                label='CV-fold %i' %i, alpha=0.5,
                range=(dist_samples.min(), dist_samples.max()))
        # empirical error
        pl.axvline(np.asscalar(err[i]), color=colors[i])
```

```
    # chance-level for a binary classification with balanced samples
    pl.axvline(0.5, color='black', ls='--')
    # scale x-axis to full range of possible error values
    pl.xlim(0,1)
    pl.xlabel(title)

plot_cv_results(cv, err, 'Per CV-fold classification error')
```

We have already seen that the statistical evaluation is pretty flexible. However, we haven't yet seen whether it is flexible enough. To illustrate that think about what was done in the above Monte Carlo analyses.

A dataset was shuffled repeatedly, and for each iteration a full cross-validation of classification error was performed. However, the shuffling was done on the *full* dataset, hence target were permuted in both training *and* testing dataset portions in each CV-fold. This basically means that for each Monte Carlo iteration the classifier was tested on a new data/signal. However, we may be more interested in what the classifier has to say on the *actual* data, but when it was trained on randomly permuted data.

As you can guess this is possible too and goes like this. The most important difference is that we are going to use now a dedicate measure to estimate the *Null* distribution. That measure is very similar to the cross-validation we have used before, but differs in an important bit. Let's look at the pieces.

```
# how often do we want to shuffle the data
repeater = Repeater(count=200)
# permute the training part of a dataset exactly ONCE
permutator = AttributePermutator('targets', limit={'partitions': 1}, count=1)
# CV with null-distribution estimation that permutes the training data for
# each fold independently
null_cv = CrossValidation(
            clf,
            ChainNode([partitioner, permutator], space=partitioner.get_space()),
            errorfx=mean_mismatch_error)
# Monte Carlo distribution estimator
distr_est = MCNullDist(repeater, tail='left', measure=null_cv,
                       enable_ca=['dist_samples'])
# actual CV with H0 distribution estimation
cv = CrossValidation(clf, partitioner, errorfx=mean_mismatch_error,
                     null_dist=distr_est, enable_ca=['stats'])
```

The `repeater` is a simple node that returns any given dataset a configurable number of times. We use the helper to configure the number of Monte Carlo iterations. The new `permutator` is again configured to shuffle the `targets` attribute, but only *once* and only for samples that were labeled as being part of the training set in a particular CV-fold (the `partitions` sample attribute will be created by the NFoldPartitioner that we have configured earlier).

The most important difference is a new dedicated measure that will be used to perform a cross-validation analysis under the *H0* hypotheses. To this end we set up a standard CV procedure with a twist: we use a chained generator (comprising of the typical partitioner and the new one-time permutator). This will cause the CV to permute the training set for each CV-fold internally (and that is what we wanted).

Now we assign the *H0* cross-validation procedure to the distribution estimator and use the `repeater` to set the number of iterations. Lastly, we plug everything into a standard CV analysis with, again, a non-permuting `partitioner` and the pimped *Null* distribution estimator.

Now we just need to run it, and plot the results the same way we did before.

```
err = cv(ds)
print 'CV-errors:', np.ravel(err)
p = cv.ca.null_prob
print 'Corresponding p-values:',  np.ravel(p)
# plot
plot_cv_results(cv, err,
                'Per CV-fold classification error (only training permutation)')
```

There a many ways to futher tweak the statistical evaluation. For example, if the family of the distribution is known (e.g. Gaussian/Normal) and provided with the `dist_class` parameter of `MCNullDist`, then permutation tests done by `MCNullDist` allow determining the distribution parameters. Under the (strong) assumption of Gaussian distribution, 20-30 permutations should be sufficient to get sensible estimates of the distribution parameters.

But that would be another story...

See also:

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

6.2.10 Nested Cross-Validation

Often it is desired to explore multiple models (classifiers, parameterizations) but it becomes an easy trap for introducing an optimistic bias into generalization estimate. The easiest but computationally intensive solution to overcome such a bias is to carry model selection by estimating the same (or different) performance characteristic while operating only on training data. If such performance is a cross-validation, then it leads to the so called "nested cross-validation" procedure.

This example will demonstrate on how to implement such nested cross-validation while selecting the best performing classifier from the warehouse of available within PyMVPA.

```
from mvpa2.suite import *
# increase verbosity a bit for now
verbose.level = 3
# pre-seed RNG if you want to investigate the effects, thus
# needing reproducible results
#mvpa2.seed(3)
```

For this simple example lets generate some fresh random data with 2 relevant features and low SNR.

```
dataset = normal_feature_dataset(perlabel=24, nlabels=2, nchunks=3,
                                 nonbogus_features=[0, 1],
                                 nfeatures=100, snr=3.0)
```

For the demonstration of model selection benefit, lets first compute cross-validated error using simple and popular kNN.

```
clf_sample = kNN()
cv_sample = CrossValidation(clf_sample, NFoldPartitioner())

verbose(1, "Estimating error using a sample classifier")
error_sample = np.mean(cv_sample(dataset))
```

For the convenience lets define a helpful function which we will use twice – once within cross-validation, and once on the whole dataset

```
def select_best_clf(dataset_, clfs):
    """Select best model according to CVTE

    Helper function which we will use twice -- once for proper nested
    cross-validation, and once to see how big an optimistic bias due
    to model selection could be if we simply provide an entire dataset.

    Parameters
    ----------
    dataset_ : Dataset
    clfs : list of Classifiers
      Which classifiers to explore

    Returns
    -------
```

```
    best_clf, best_error
    """
    best_error = None
    for clf in clfs:
        cv = CrossValidation(clf, NFoldPartitioner())
        # unfortunately we don't have ability to reassign clf atm
        # cv.transerror.clf = clf
        try:
            error = np.mean(cv(dataset_))
        except LearnerError, e:
            # skip the classifier if data was not appropriate and it
            # failed to learn/predict at all
            continue
        if best_error is None or error < best_error:
            best_clf = clf
            best_error = error
        verbose(4, "Classifier %s cv error=%.2f" % (clf.descr, error))
    verbose(3, "Selected the best out of %i classifiers %s with error %.2f"
            % (len(clfs), best_clf.descr, best_error))
    return best_clf, best_error
```

First lets select a classifier within cross-validation, thus eliminating model-selection bias

```
best_clfs = {}
confusion = ConfusionMatrix()
verbose(1, "Estimating error using nested CV for model selection")
partitioner = NFoldPartitioner()
splitter = Splitter('partitions')
for isplit, partitions in enumerate(partitioner.generate(dataset)):
    verbose(2, "Processing split #%i" % isplit)
    dstrain, dstest = list(splitter.generate(partitions))
    best_clf, best_error = select_best_clf(dstrain, clfswh['!gnpp'])
    best_clfs[best_clf.descr] = best_clfs.get(best_clf.descr, 0) + 1
    # now that we have the best classifier, lets assess its transfer
    # to the testing dataset while training on entire training
    tm = TransferMeasure(best_clf, splitter,
                         postproc=BinaryFxNode(mean_mismatch_error,
                                               space='targets'),
                         enable_ca=['stats'])
    tm(partitions)
    confusion += tm.ca.stats
```

And for comparison, lets assess what would be the best performance if we simply explore all available classifiers, providing all the data at once

```
verbose(1, "Estimating error via fishing expedition (best clf on entire dataset)")
cheating_clf, cheating_error = select_best_clf(dataset, clfswh['!gnpp'])

print """Errors:
 sample classifier (kNN): %.2f
 model selection within cross-validation: %.2f
 model selection via fishing expedition: %.2f with %s
 """ % (error_sample, 1 - confusion.stats['ACC'],
        cheating_error, cheating_clf.descr)

print "# of times following classifiers were selected within " \
      "nested cross-validation:"
for c, count in sorted(best_clfs.items(), key=lambda x:x[1], reverse=True):
    print " %i times %s" % (count, c)

print "\nConfusion table for the nested cross-validation results:"
print confusion
```

See also:

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

6.2.11 Determine the Distribution of some Variable

This is an example demonstrating discovery of the distribution facility.

```
from mvpa2.suite import *

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

While doing distribution matching, this example also demonstrates infrastructure within PyMVPA to log a progress report not only on the screen, but also into external files, such as

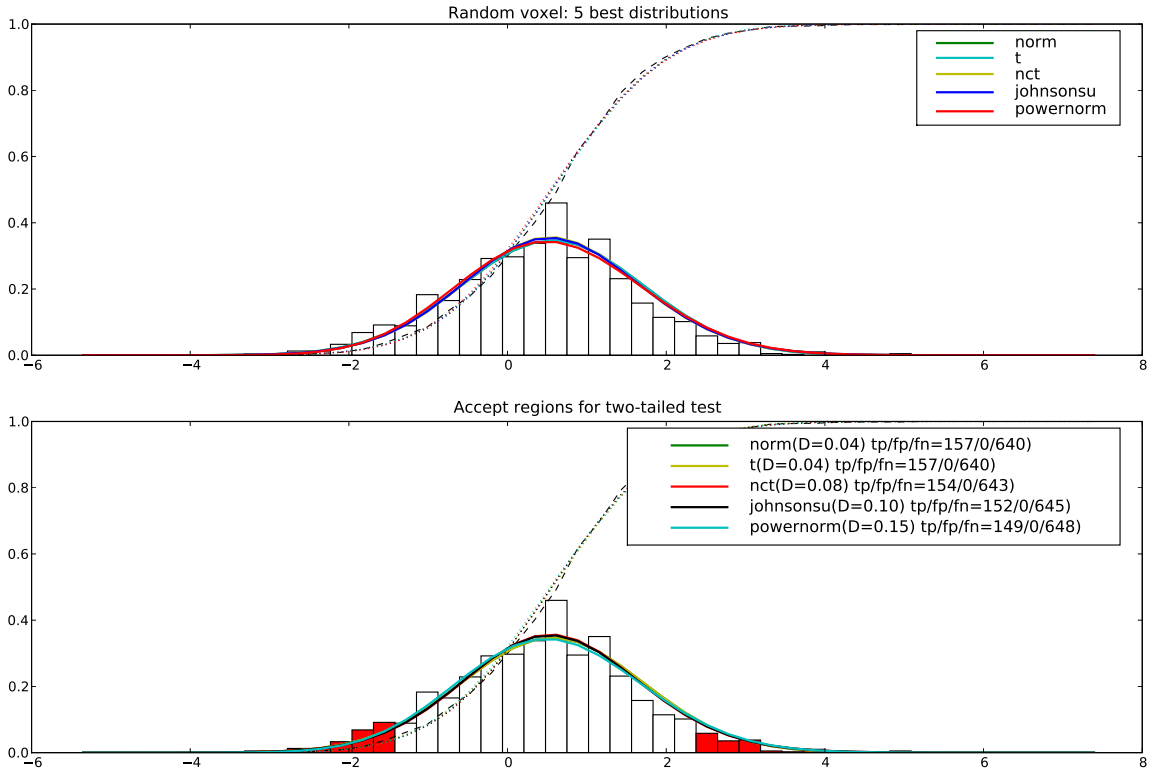
- simple text file,
- PDF file including all text messages and pictures which were rendered.

For PDF report you need to have `reportlab` external available.

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

# Also append verbose output into a log file we care about
verbose.handlers += [open('match_distribution_report.log', 'a')]

#
# Figure for just normal distribution
#

# generate random signal from normal distribution
verbose(1, "Random signal with normal distribution")
data = np.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 = match_distribution(data, test=test, p=0.05)

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

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

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

#
# Figure for fMRI data sample we have
```

```
#
verbose(1, "Load sample fMRI dataset")
dataset = load_example_fmri_dataset()
# select random voxel
dataset = dataset[:, int(np.random.uniform()*dataset.nfeatures)]

verbose(2, "Minimal preprocessing to remove the bias per each voxel")
poly_detrend(dataset, chunks_attr='chunks', polyord=1)
zscore(dataset, chunks_attr='chunks', param_est=('targets', ['0']),
       dtype='float32')

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

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

pl.subplot(2, 1, 2)
plot_distribution_matches(data, matches, nbest=5, p=0.05,
                          tail='any', legend=4)
pl.title('Accept regions for two-tailed test')
report.figure()
```

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`).

6.2.12 Spatio-temporal Analysis of event-related fMRI data

In this example we are going to take a look at an event-related analysis of timeseries data. We will do this on fMRI data, implementing a spatio-temporal analysis of multi-volume samples. It starts as usual by loading PyMVPA and the fMRI dataset.

```
from mvpa2.suite import *

ds = load_tutorial_data(roi=(36,38,39,40))
```

The dataset we have just loaded is the full timeseries of voxels in the ventral temporal cortex for 12 concatenated experiment runs. Although originally a block-design experiment, we'll analyze it in an event-related fashion, where each stimulation block will be considered as an individual event.

For an event-related analysis most of the processing is done on data samples that are somehow derived from a set of events. The rest of the data could be considered irrelevant. However, some preprocessing is only meaningful when performed on the full timeseries and not on the segmented event samples. An example is the detrending that typically needs to be done on the original, continuous timeseries.

In its current shape our dataset consists of 1452 samples that represent contiguous fMRI volumes. At this stage we can easily perform linear detrending. We are going to do it per each experiment run (the dataset has to runs encoded in the `chunk` sample attribute), since we do not assume a contiguous linear trend throughout the whole recording session.

```
# detrend on full timeseries
poly_detrend(ds, polyord=1, chunks_attr='chunks')
```

Let's make a copy of the detrended dataset that we can later on use for some visualization.

```
orig_ds = ds.copy()
```

We still need to normalize each feature (i.e. a voxel at this point). In this case we are going to Z-score them, using the mean and standard deviation from the experiment's rest condition. The resulting values might be interpreted as "activation scores". We are again doing it per each run.

```
zscore(ds, chunks_attr='chunks', param_est=('targets', 'rest'))
```

After detrending and normalization, we can now segment the timeseries into a set of events. To achieve this we have to compile a list of event definitions first. In this example we will simply convert the block-design setup defined by the samples attributes into events, so that each stimulation block becomes an event with an associated onset and duration. The events are defined by a change in any of the provided attributes, hence we get an event for starting stimulation block and any start of a run in the experiment.

```
events = find_events(targets=ds.sa.targets, chunks=ds.sa.chunks)
```

`events` is a simple list of event definitions (each one being a dictionary) that can easily inspected for startpoints and duration of events. Later on we want to look at the sensitivity profile ranging from just before until a little after each block. Therefore we are slightly moving the event onsets prior the block start and request to extract a set of 13 consecutive volumes a as sample for each event. Finally, in this example we are only interested in `face` or `house` blocks.

```
# filter out events
events = [ev for ev in events if ev['targets'] in ['house', 'face']]

# modify event start and set uniform duration
for ev in events:
    ev['onset'] -= 2
    ev['duration'] = 13
```

Now we get to the core of an event-related analysis. We turn our existing timeseries datasets into one with samples of timeseries segments.

PyMVPA offers `eventrelated_dataset()` to perform this conversion – given a list of events and a dataset with samples that are sorted by time. If a dataset has information about acquisition time

`eventrelated_dataset()` can also convert event-definition in real time.

```
evds = eventrelated_dataset(ds, events=events)
```

Now we have our final dataset with spatio-temporal fMRI samples. Look at the attributes of the dataset to see what information is available about each event. The rest is pretty much standard.

We want to perform a cross-validation analysis of a SVM classifier. We are not primarily interested in its performance, but in the weights it assigns to the features. Remember, each feature is now voxel-timepoint, so we get a chance of looking at the spatio-temporal profile of classification relevant information in the data. We will nevertheless enable computing a confusion matrix, so we can assure ourselves that the classifier is performing reasonably well, since only a generalizing classifier model is worth inspecting, as otherwise the assigned weights are meaningless.

```
clf = LinearCSVMC()
sclf = SplitClassifier(clf, enable_ca=['stats'])

# Compute sensitivity, which internally trains the classifier
analyzer = sclf.get_sensitivity_analyzer()
sensitivities = analyzer(evds)
```

Now let's look at the confusion matrix – it turns out that the classifier performs excellent.

```
print sclf.ca.stats
```

We could now convert the computed sensitivities back into a 4D fMRI image to look at the spatio-temporal sensitivity profile using the datasets mapper. However, in this example we are going to plot it for two example voxels and compare it to the actual signal timecourse prior and after normalization.

```
# example voxel coordinates
example_voxels = [(28,25,25), (28,23,25)]
```

First we plot the orginal signal after initial detrending. To do this, we apply the timeseries segmentation to the original detrended dataset and plot to mean signal for all face and house events for both of our example voxels.

```
vx_lty = ['-', '--']
t_col = ['b', 'r']

pl.subplot(311)
for i, v in enumerate(example_voxels):
    slicer = np.array([tuple(idx) == v for idx in ds.fa.voxel_indices])
    evds_detrend = eventrelated_dataset(orig_ds[:, slicer], events=events)
    for j, t in enumerate(evds.uniquetargets):
        pl.plot(np.mean(evds_detrend[evds_detrend.sa.targets == t], axis=0),
                t_col[j] + vx_lty[i],
                label='Voxel %i: %s' % (i, t))
pl.ylabel('Detrended signal')
pl.axhline(linestyle='--', color='0.6')
pl.legend()
```

In the next step we do exactly the same again, but this time for the normalized data.

```
pl.subplot(312)
for i, v in enumerate(example_voxels):
    slicer = np.array([tuple(idx) == v for idx in ds.fa.voxel_indices])
    evds_norm = eventrelated_dataset(ds[:, slicer], events=events)
    for j, t in enumerate(evds.uniquetargets):
        pl.plot(np.mean(evds_norm[evds_norm.sa.targets == t], axis=0),
                t_col[j] + vx_lty[i])
pl.ylabel('Normalized signal')
pl.axhline(linestyle='--', color='0.6')
```

Finally, we plot the associated SVM weight profile for each peristimulus timepoint of both voxels. For easier selection we do a little trick and reverse-map the sensitivity profile through the last mapper in the dataset's chain mapper

(look at `evds.a.mapper` for the whole chain). This will reshape the sensitivities into `cross-validation fold x volume x voxel features`.

```
pl.subplot(313)
smaps = evds.a.mapper[-1].reverse(sensitivities)

for i, v in enumerate(example_voxels):
    slicer = np.array([tuple(idx) == v for idx in ds.fa.voxel_indices])
    smap = smaps.samples[:,:,slicer].squeeze()
    plot_err_line(smap, fmt='ko', linestyle=vx_lty[i])
pl.xlim((0,12))
pl.ylabel('Sensitivity')
pl.axhline(linestyle='--', color='0.6')
pl.xlabel('Peristimulus volumes')
```

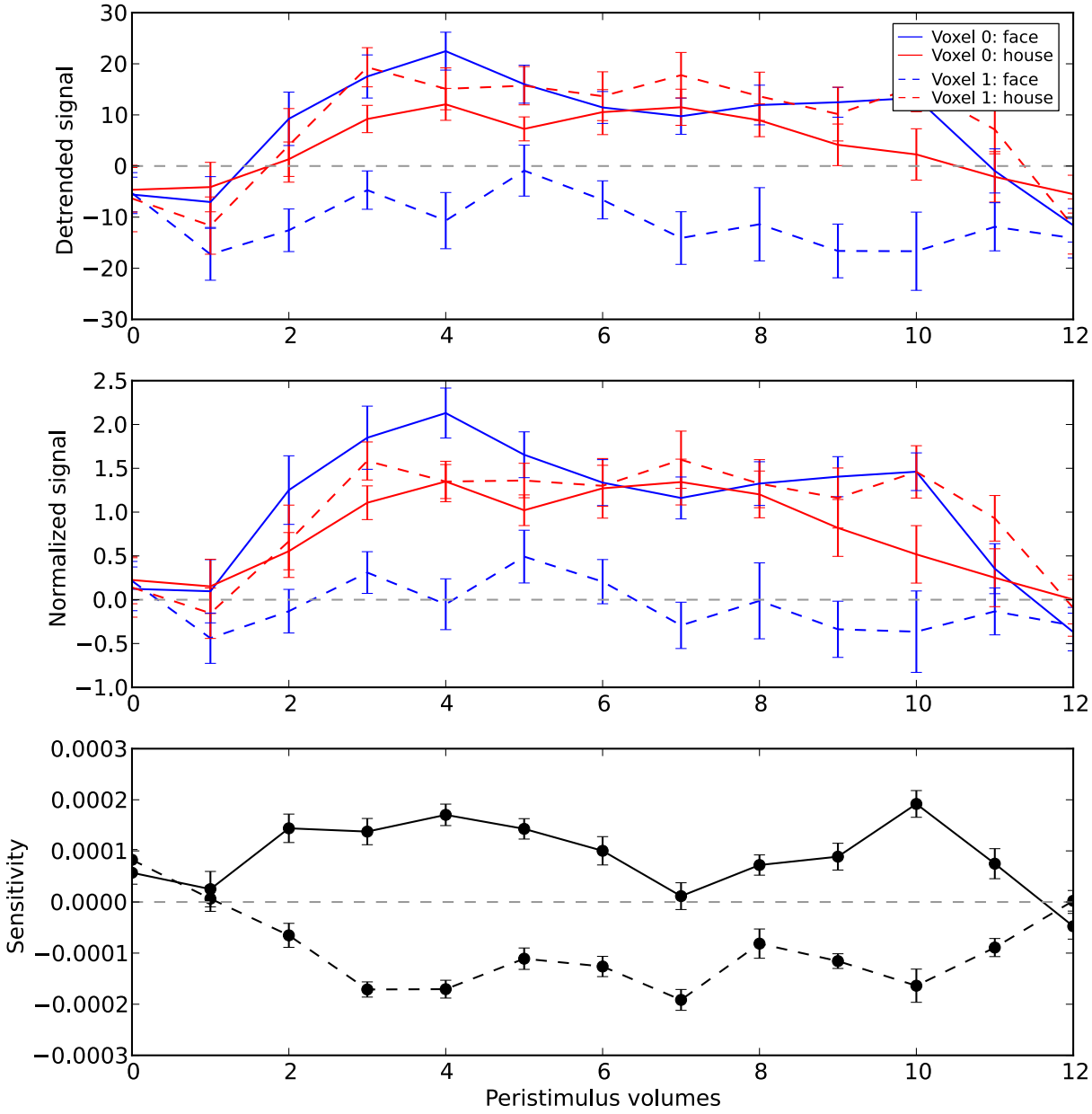


Fig. 6.4: Sensitivity profile for two example voxels for *face* vs. *house* classification on event-related fMRI data from ventral temporal cortex.

This demo showed an event-related data analysis. Although we have performed it on fMRI data, an analogous analysis can be done for any timeseries-based data in an almost identical fashion.

See also:

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

6.2.13 Hyperalignment for between-subject analysis

Multivariate pattern analysis (MVPA) reveals how the brain represents fine-scale information. Its power lies in its sensitivity to subtle pattern variations that encode this fine-scale information but that also presents a hurdle for group analyses due to between-subject variability of both anatomical & functional architectures. *Haxby et al. (2011)* recently proposed a method of aligning subjects' brain data in a high-dimensional functional space and showed how to build a common model of ventral temporal cortex that captures visual object category information. They tested their model by successfully performing between-subject classification of category information. Moreover, when they built the model using a complex naturalistic stimulation (a feature film), it even generalized to other independent experiments even after removing any occurrences of the experimental stimuli from the movie data.

In this example we show how to perform Hyperalignment within a single experiment. We will compare between-subject classification after hyperalignment to between-subject classification on anatomically aligned data (currently the most typical approach), and within-subject classification performance.

Analysis setup

```
from mvpa2.suite import *

verbose.level = 2
```

We start by loading preprocessed datasets of 10 subjects with BOLD-responses of stimulation with face and object images (*Haxby et al., 2011*). Each dataset, after preprocessing, has one sample per category and run for each of the eight runs and seven stimulus categories. Individual subject brains have been aligned anatomically using a 12 dof linear transformation.

```
verbose(1, "Loading data...")
filepath = os.path.join(cfg.get('location', 'tutorial data'),
                        'hyperalignment_tutorial_data.hdf5.gz')
ds_all = h5load(filepath)
# zscore all datasets individually
_ = [zscore(ds) for ds in ds_all]
# inject the subject ID into all datasets
for i,sd in enumerate(ds_all):
    sd.sa['subject'] = np.repeat(i, len(sd))
# number of subjects
nsubjs = len(ds_all)
# number of categories
ncats = len(ds_all[0].UT)
# number of run
nruns = len(ds_all[0].UC)
verbose(2, "%d subjects" % len(ds_all))
verbose(2, "Per-subject dataset: %i samples with %i features" % ds_all[0].shape)
verbose(2, "Stimulus categories: %s" % ', '.join(ds_all[0].UT))
```

Now we'll create a couple of building blocks for the intended analyses. We'll use a linear SVM classifier, and perform feature selection with a simple one-way ANOVA selecting the `nf` highest scoring features.

```
# use same classifier
clf = LinearCSVMC()

# feature selection helpers
nf = 100
fselector = FixedNElementTailSelector(nf, tail='upper',
                                      mode='select',sort=False)
```

```
sbfs = SensitivityBasedFeatureSelection(OneWayAnova(), fselector,
                                        enable_ca=['sensitivities'])
# create classifier with automatic feature selection
fsclf = FeatureSelectionClassifier(clf, sbfs)
```

Within-subject classification

We start off by running a cross-validated classification analysis for every subject's dataset individually. Data folding will be performed by leaving out one run to serve as the testing dataset. ANOVA-based features selection will be performed automatically on training dataset and applied to testing dataset.

```
verbose(1, "Performing classification analyses...")
verbose(2, "within-subject...", cr=False, lf=False)
wsc_start_time = time.time()
cv = CrossValidation(fsclf,
                     NFoldPartitioner(attr='chunks'),
                     errorfx=mean_match_accuracy)
# store results in a sequence
wsc_results = [cv(sd) for sd in ds_all]
wsc_results = vstack(wsc_results)
verbose(2, " done in %.1f seconds" % (time.time() - wsc_start_time,))
```

Between-subject classification using anatomically aligned data

For between-subject classification with MNI-aligned voxels, we can stack up all individual datasets into a single one, as (anatomical!) feature correspondence is given. The crossvalidation analysis using the feature selection classifier will automatically perform the desired ANOVA-based feature selection on every training dataset partition. However, data folding will now be done by leaving out a complete subject for testing.

```
verbose(2, "between-subject (anatomically aligned)...", cr=False, lf=False)
ds_mni = vstack(ds_all)
mni_start_time = time.time()
cv = CrossValidation(fsclf,
                     NFoldPartitioner(attr='subject'),
                     errorfx=mean_match_accuracy)
bsc_mni_results = cv(ds_mni)
verbose(2, "done in %.1f seconds" % (time.time() - mni_start_time,))
```

Between-subject classification with Hyperalignment(TM)

Between-subject classification using Hyperalignment is very similar to the previous analysis. However, now we no longer assume feature correspondence (or aren't satisfied with anatomical alignment anymore). Consequently, we have to transform the individual datasets into a common space before performing the classification analysis. To avoid introducing circularity problems to the analysis, we perform leave-one-run-out data folding manually, and train Hyperalignment only on the training dataset partitions. Subsequently, we will apply the derived transformation to the full datasets, stack them up across individual subjects, as before, and run the classification analysis. ANOVA-based feature selection is done in the same way as before (but also manually).

```
verbose(2, "between-subject (hyperaligned)...", cr=False, lf=False)
hyper_start_time = time.time()
bsc_hyper_results = []
# same cross-validation over subjects as before
cv = CrossValidation(clf, NFoldPartitioner(attr='subject'),
                     errorfx=mean_match_accuracy)

# leave-one-run-out for hyperalignment training
for test_run in range(nruns):
    # split in training and testing set
```

```
    ds_train = [sd[sd.sa.chunks != test_run,:] for sd in ds_all]
    ds_test = [sd[sd.sa.chunks == test_run,:] for sd in ds_all]

    # manual feature selection for every individual dataset in the list
    anova = OneWayAnova()
    fscores = [anova(sd) for sd in ds_train]
    featsels = [StaticFeatureSelection(fselector(fscore)) for fscore in fscores]
    ds_train_fs = [featsels[i].forward(sd) for i, sd in enumerate(ds_train)]


    # Perform hyperalignment on the training data with default parameters.
    # Computing hyperalignment parameters is as simple as calling the
    # hyperalignment object with a list of datasets. All datasets must have the
    # same number of samples and time-locked responses are assumed.
    # Hyperalignment returns a list of mappers corresponding to subjects in the
    # same order as the list of datasets we passed in.


    hyper = Hyperalignment()
    hypmaps = hyper(ds_train_fs)

    # Applying hyperalignment parameters is similar to applying any mapper in
    # PyMVPA. We start by selecting the voxels that we used to derive the
    # hyperalignment parameters. And then apply the hyperalignment parameters
    # by running the test dataset through the forward() function of the mapper.

    ds_test_fs = [featsels[i].forward(sd) for i, sd in enumerate(ds_test)]
    ds_hyper = [ hypmaps[i].forward(sd) for i, sd in enumerate(ds_test_fs)]

    # Now, we have a list of datasets with feature correspondence in a common
    # space derived from the training data. Just as in the between-subject
    # analyses of anatomically aligned data we can stack them all up and run the
    # crossvalidation analysis.

    ds_hyper = vstack(ds_hyper)
    # zscore each subject individually after transformation for optimal
    # performance
    zscore(ds_hyper, chunks_attr='subject')
    res_cv = cv(ds_hyper)
    bsc_hyper_results.append(res_cv)

bsc_hyper_results = hstack(bsc_hyper_results)
verbose(2, "done in %.1f seconds" % (time.time() - hyper_start_time,))
```

Comparing the results

Performance

First we take a look at the classification performance (or accuracy) of all three analysis approaches.

```
verbose(1, "Average classification accuracies:")
verbose(2, "within-subject: %.2f +/-%.3f"
        % (np.mean(wsc_results),
           np.std(wsc_results) / np.sqrt(nsubjs - 1)))
verbose(2, "between-subject (anatomically aligned): %.2f +/-%.3f"
        % (np.mean(bsc_mni_results),
           np.std(np.mean(bsc_mni_results, axis=1)) / np.sqrt(nsubjs - 1)))
verbose(2, "between-subject (hyperaligned): %.2f +/-%.3f" \
        % (np.mean(bsc_hyper_results),
           np.std(np.mean(bsc_hyper_results, axis=1)) / np.sqrt(nsubjs - 1)))
```

The output of this demo looks like this:

```
Loading data...
 10 subjects
 Per-subject dataset: 56 samples with 3509 features
 Stimulus categories: Chair, DogFace, FemaleFace, House, MaleFace, MonkeyFace, Shoe
Performing classification analyses...
 within-subject... done in 4.3 seconds
 between-subject (anatomically aligned)...done after 3.2 seconds
 between-subject (hyperaligned)...done in 10.5 seconds
Average classification accuracies:
 within-subject: 0.57 +/-0.063
 between-subject (anatomically aligned): 0.42 +/-0.035
 between-subject (hyperaligned): 0.62 +/-0.046
```

It is obvious that the between-subject classification using anatomically aligned data has significantly worse performance when compared to within-subject classification. Clearly the group classification model is inferior to individual classifiers fitted to a particular subject's data. However, a group classifier trained on hyperaligned data is performing at least as good as the within-subject classifiers – possibly even slightly better due to the increased size of the training dataset.

Similarity structures

To get a better understanding of how hyperalignment transforms the structure of the data, we compare the similarity structures of the corresponding input datasets of all three analysis above (and one in addition).

These are respectively:

1. Average similarity structure of the individual data.
2. Similarity structure of the averaged hyperaligned data.
3. Average similarity structure of the individual data after hyperalignment.
4. Similarity structure of the averaged anatomically-aligned data.

Similarity structure in this case is the correlation matrix of multivariate response patterns for all seven stimulus categories in the datasets. For the sake of simplicity, all similarity structures are computed on the full dataset without data folding.

```
# feature selection as above
anova = OneWayAnova()
fscores = [anova(sd) for sd in ds_all]
fscores = np.mean(np.asarray(vstack(fscores)), axis=0)
# apply to full datasets
ds_fs = [sd[:,fselector(fscores)] for i,sd in enumerate(ds_all)]
#run hyperalignment on full datasets
hyper = Hyperalignment()
mappers = hyper(ds_fs)
ds_hyper = [ mappers[i].forward(ds_) for i,ds_ in enumerate(ds_fs)]
# similarity of original data samples
sm_orig = [np.corrcoef(
                sd.get_mapped(
                    mean_group_sample(['targets'])).samples)
                        for sd in ds_fs]
# mean across subjects
sm_orig_mean = np.mean(sm_orig, axis=0)
# same individual average but this time for hyperaligned data
sm_hyper_mean = np.mean(
                    [np.corrcoef(
                        sd.get_mapped(mean_group_sample(['targets'])).samples)
                            for sd in ds_hyper], axis=0)
# similarity for averaged hyperaligned data
ds_hyper = vstack(ds_hyper)
```

```
sm_hyper = np.corrcoef(ds_hyper.get_mapped(mean_group_sample(['targets'])))
# similarity for averaged anatomically aligned data
ds_fs = vstack(ds_fs)
sm_anat = np.corrcoef(ds_fs.get_mapped(mean_group_sample(['targets'])))
```

We then plot the respective similarity strucures.

```
# class labels should be in more meaningful order for visualization
# (human faces, animals faces, objects)
intended_label_order = [2,4,1,5,3,0,6]
labels = ds_all[0].UT
labels = labels[intended_label_order]

pl.figure(figsize=(6,6))
# plot all three similarity structures
for i, sm_t in enumerate((
    (sm_orig_mean, "Average within-subject\nsimilarity"),
    (sm_anat, "Similarity of group average\ndata (anatomically aligned)"),
    (sm_hyper_mean, "Average within-subject\nsimilarity (hyperaligned data)"),
    (sm_hyper, "Similarity of group average\ndata (hyperaligned)"),
                      )):
    sm, title = sm_t
    # reorder matrix columns to match label order
    sm = sm[intended_label_order][:,intended_label_order]
    pl.subplot(2, 2, i+1)
    pl.imshow(sm, vmin=-1.0, vmax=1.0, interpolation='nearest')
    pl.colorbar(shrink=.4, ticks=[-1,0,1])
    pl.title(title, size=12)
    ylim = pl.ylim()
    pl.xticks(range(ncats), labels, size='small', stretch='ultra-condensed',
              rotation=45)
    pl.yticks(range(ncats), labels, size='small', stretch='ultra-condensed',
              rotation=45)
    pl.ylim(ylim)
```

We can clearly see that averaging anatomically aligned data has a negative effect on the similarity structure, as the fine category structure is diminished and only the coarse structure (faces vs. objects) is preserved. Moreover, we can see that after hyperalignment the average similarity structure of individual data is essentially identical to the similarity structure of averaged data – reflecting the feature correspondence in the common high-dimensional space.

Regularized Hyperalignment

According to *Xu et al. 2012*, Hyperalignment can be reformulated to a regularized algorithm that can span the whole continuum between canonical correlation analysis (CCA) and regular hyperalignment by varying a regularization parameter (alpha). Here, we repeat the above between-subject hyperalignment and classification analyses with varying values of alpha from 0 (CCA) to 1.0 (regular hyperalignment).

The following code is essentially identical to the implementation of between-subject classification shown above. The only difference is an addition `for` loop doing the alpha value sweep for each cross-validation fold.

```
alpha_levels = np.concatenate(
                    (np.linspace(0.0, 0.7, 8),
                     np.linspace(0.8, 1.0, 9)))
# to collect the results for later visualization
bsc_hyper_results = np.zeros((nsubjs, len(alpha_levels), nruns))
# same cross-validation over subjects as before
cv = CrossValidation(clf, NFoldPartitioner(attr='subject'),
                     errorfx=mean_match_accuracy)

# leave-one-run-out for hyperalignment training
for test_run in range(nruns):
```

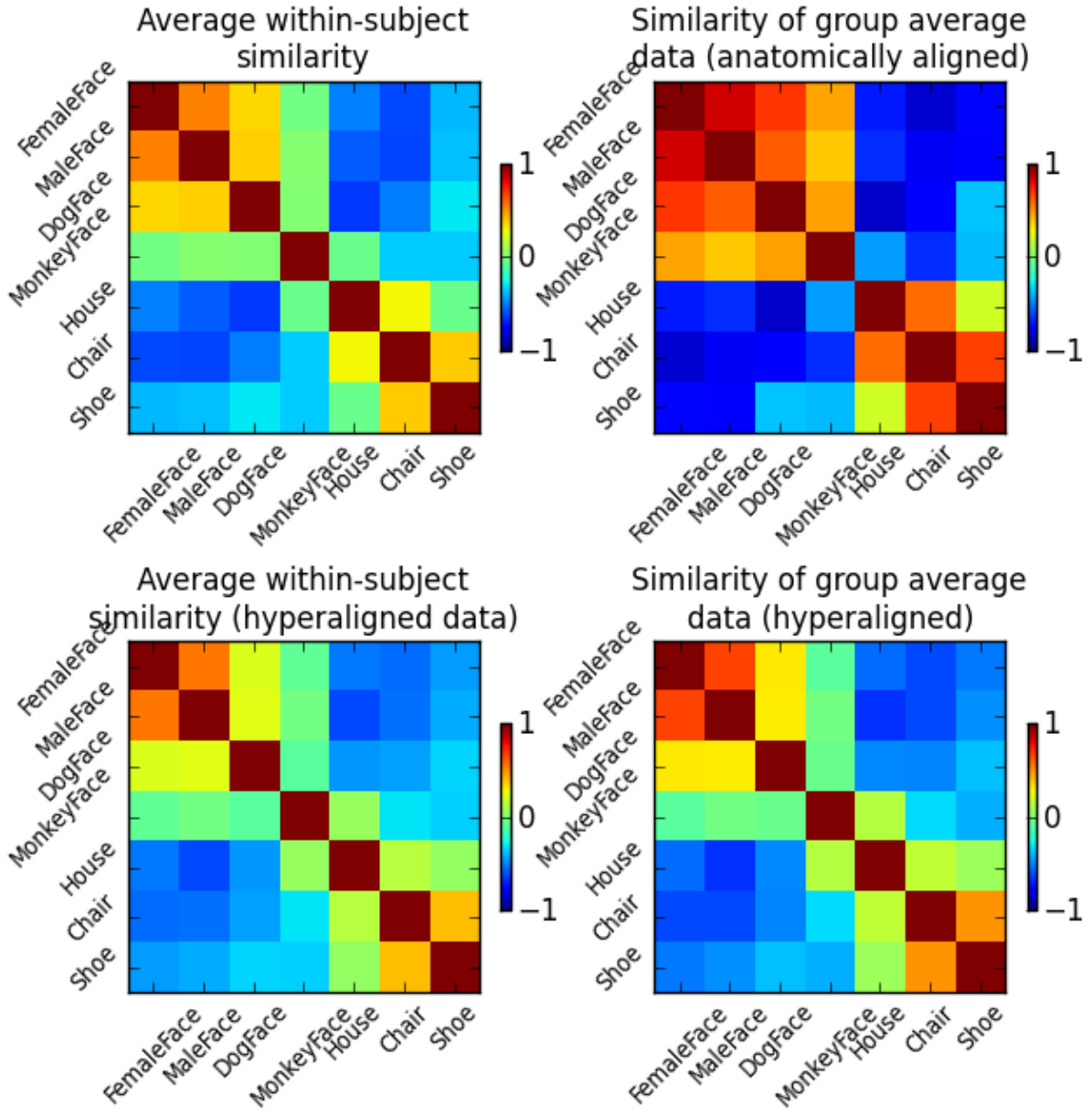


Fig. 6.5: Fig. 1: Correlation of category-specific response patterns using the 100 most informative voxels (based on ANOVA F-score ranking).

```
    # split in training and testing set
    ds_train = [sd[sd.sa.chunks != test_run,:] for sd in ds_all]
    ds_test = [sd[sd.sa.chunks == test_run,:] for sd in ds_all]

    # manual feature selection for every individual dataset in the list
    anova = OneWayAnova()
    fscores = [anova(sd) for sd in ds_train]
    featsels = [StaticFeatureSelection(fselector(fscore)) for fscore in fscores]
    ds_train_fs = [featsels[i].forward(sd) for i, sd in enumerate(ds_train)]

    for alpha_level, alpha in enumerate(alpha_levels):
        hyper = Hyperalignment(alignment=ProcrusteanMapper(svd='dgesvd',
                                                           space='commonspace'),
                               alpha=alpha)
        hypmaps = hyper(ds_train_fs)
        ds_test_fs = [featsels[i].forward(sd) for i, sd in enumerate(ds_test)]
        ds_hyper = [ hypmaps[i].forward(sd) for i, sd in enumerate(ds_test_fs)]
        ds_hyper = vstack(ds_hyper)
        zscore(ds_hyper, chunks_attr='subject')
        res_cv = cv(ds_hyper)
        bsc_hyper_results[:, alpha_level, test_run] = res_cv.samples.T
```

Now we can plot the classification accuracy as a function of regularization intensity.

```
bsc_hyper_results = np.mean(bsc_hyper_results, axis=2)
pl.figure()
plot_err_line(bsc_hyper_results, alpha_levels)
pl.xlabel('Regularization parameter: alpha')
pl.ylabel('Average BSC using hyperalignment +/- SEM')
pl.title('Using regularized hyperalignment with varying alpha values')
```

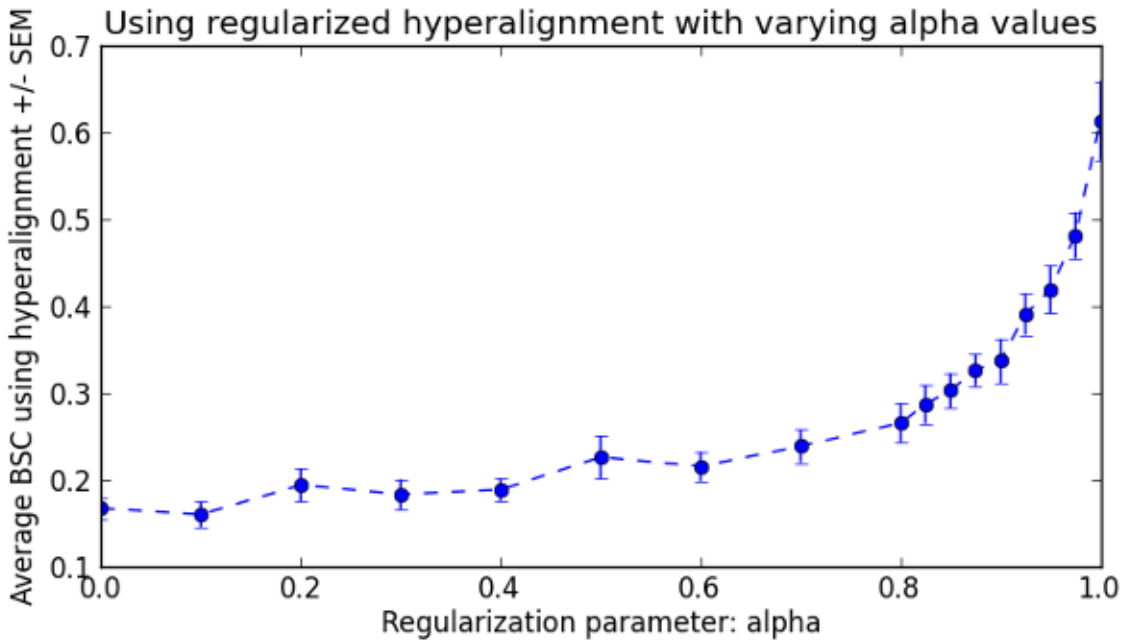


Fig. 6.6: Fig. 2: Mean between-subject classification accuracies using regularized hyperalignment with alpha value ranging from 0 (CCA) to 1 (vanilla hyperalignment).

We can clearly see that the regular hyperalignment performs best for this dataset. However, please refer to *Xu et al. 2012* for an example showing that this is not always the case.

See also:

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

6.2.14 Analysis of eye movement patterns

In this example we are going to look at a classification analysis of eye movement patterns. Although complex preprocessing steps can be performed to extract higher-order features from the raw coordinate timeseries provided by an eye-tracker, we are keeping it simple.

Right after importing the PyMVPA suite, we load the data from a textfile. It contains coordinate timeseries of 144 trials (recorded with 350 Hz), where subjects either looked at upright or inverted images of human faces. Each timeseries snippet covers 3 seconds. This data has been pre-processed to remove eyeblink artefacts.

In addition to the coordinates we also load trial attributes from a second textfile. These attributes indicate which image was shown, whether it was showing a male or female face, and wether it was upright or inverted.

```
from mvpa2.suite import *

# where is the data
datapath = os.path.join(pymvpa_datadbroot,
                        'face_inversion_demo', 'face_inversion_demo')
# (X, Y, trial id) for all timepoints
data = np.loadtxt(os.path.join(datapath, 'gaze_coords.txt'))
# (orientation, gender, image id) for each trial
attribs = np.loadtxt(os.path.join(datapath, 'trial_attrs.txt'))
```

As a first step we put the coordinate timeseries into a dataset, and labels each timepoint with its associated trial ID. We also label the two features accordingly.

```
raw_ds = Dataset(data[:,:2],
                 sa = {'trial': data[:,2]},
                 fa = {'fid': ['rawX', 'rawY']})
```

The second step is down-sampling the data to about 33 Hz, resampling each trial timeseries individually (using the trial ID attribute to define dataset chunks).

```
ds = fft_resample(raw_ds, 100, window='hann',
                  chunks_attr='trial', attr_strategy='sample')
```

Now we can use a `BoxcarMapper` to turn each trial-timeseries into an individual sample. We know that each sample consists of 100 timepoints. After the dataset is mapped we can add all per-trial attributes into the sample attribute collection.

```
bm = BoxcarMapper(np.arange(len(ds.sa['trial'].unique)) * 100,
                  boxlength=100)
bm.train(ds)
ds=ds.get_mapped(bm)

ds.sa.update({'orient': attribs[:,0].astype(int),
              'gender': attribs[:,1].astype(int),
              'img_id': attribs[:,1].astype(int)})
```

In comparison with upright faces, inverted ones had prominent features at very different locations on the screen. Most notably, the eyes were flipped to the bottom half. To prevent the classifier from using such differences, we flip the Y-coordinates for trials with inverted to align the with the upright condition.

```
ds.samples[ds.sa.orient == 1, :, 1] = \
        -1 * (ds.samples[ds.sa.orient == 1, :, 1] - 512) + 512
```

The current dataset has 100 two-dimensional features, the X and Y coordinate for each of the hundred timepoints. We use a `FlattenMapper` to convert each sample into a one-dimensionl vector (of length 200). However, we also keep the original dataset, because it will allow us to perform some plotting much easier.

```
fm = FlattenMapper()
fm.train(ds)
# want to make a copy to keep the original pristine for later plotting
fds = ds.copy().get_mapped(fm)

# simplify the trial attribute
fds.sa['trial'] = [t[0] for t in ds.sa.trial]
```

The last steps of preprocessing are Z-scoring all features (coordinate-timepoints) and dividing the dataset into 8 chunks – to simplify a cross-validation analysis.

```
zscore(fds, chunks_attr=None)

# for classification divide the data into chunks
nchunks =  8
chunks = np.zeros(len(fds), dtype='int')
for o in fds.sa['orient'].unique:
    chunks[fds.sa.orient == o] = np.arange(len(fds.sa.orient == o)) % nchunks
fds.sa['chunks'] = chunks
```

Now everything is set and we can proceed to the classification analysis. We are using a support vector machine that is going to be trained on the `orient` attribute, indicating trials with upright and inverted faces. We are going to perform the analysis with a `SplitClassifier`, because we are also interested in the temporal sensitivity profile. That one is easily accessible via the corresponding sensitivity analyzer.

```
clf = SVM(space='orient')
mclf = SplitClassifier(clf, space='orient',
                       enable_ca=['confusion'])
sensana = mclf.get_sensitivity_analyzer()
sens = sensana(fds)
print mclf.ca.confusion
```

The 8-fold cross-validation shows a trial-wise classification accuracy of over 80%. Now we can take a look at the sensitivity. We use the `FlattenMapper` that is stored in the dataset to unmangle X and Y coordinate vectors in the sensitivity array.

```
# split mean sensitivities into X and Y coordinate parts by reversing through
# the flatten mapper
xy_sens = fds.a.mapper[-2].reverse(sens).samples
```

Plotting the results

The analysis is done and we can compile a figure to visualize the results. After some inital preparations, we plot an example image of a face that was used in this experiment. We align the image coordinates with the original on-screen coordinates to match them to the gaze track, and overlay the image with the mean gaze track across all trials for each condition.

```
# descriptive plots
pl.figure()
# original screen size was
axes = ('x', 'y')
screen_size = np.array((1280, 1024))
screen_center = screen_size / 2
colors = ('r','b')
fig = 1

pl.subplot(2, 2, fig)
pl.title('Mean Gaze Track')
face_img = pl.imread(os.path.join(datapath, 'demo_face.png'))
# determine the extend of the image in original screen coordinates
# to match with gaze position
orig_img_extent=(screen_center[0] - face_img.shape[1]/2,
```

```
                  screen_center[0] + face_img.shape[1]/2,
                  screen_center[1] + face_img.shape[0]/2,
                  screen_center[1] - face_img.shape[0]/2)
# show face image and put it with original pixel coordinates
pl.imshow(face_img,
          extent=orig_img_extent,
          cmap=pl.cm.gray)
pl.plot(np.mean(ds.samples[ds.sa.orient == 1,:,0], axis=0),
        np.mean(ds.samples[ds.sa.orient == 1,:,1], axis=0),
        colors[0], label='inverted')
pl.plot(np.mean(ds.samples[ds.sa.orient == 2,:,0], axis=0),
        np.mean(ds.samples[ds.sa.orient == 2,:,1], axis=0),
        colors[1], label='upright')
pl.axis(orig_img_extent)
pl.legend()
fig += 1
```

The next two subplot contain the gaze coordinate over the peri-stimulus time for both, X and Y axis respectively.

```
pl.subplot(2, 2, fig)
pl.title('Gaze Position X-Coordinate')
plot_erp(ds.samples[ds.sa.orient == 1,:,1], pre=0, errtype = 'std',
         color=colors[0], SR=100./3.)
plot_erp(ds.samples[ds.sa.orient == 2,:,1], pre=0, errtype = 'std',
         color=colors[1], SR=100./3.)
pl.ylim(orig_img_extent[2:])
pl.xlabel('Peristimulus Time')
fig += 1

pl.subplot(2, 2, fig)
pl.title('Gaze Position Y-Coordinate')
plot_erp(ds.samples[ds.sa.orient == 1,:,0], pre=0, errtype = 'std',
         color=colors[0], SR=100./3.)
plot_erp(ds.samples[ds.sa.orient == 2,:,0], pre=0, errtype = 'std',
         color=colors[1], SR=100./3.)
pl.ylim(orig_img_extent[:2])
pl.xlabel('Peristimulus Time')
fig += 1
```

The last panel has the associated sensitivity profile for both coordinate axes.

```
pl.subplot(2, 2, fig)
pl.title('SVM-Sensitivity Profiles')
lines = plot_err_line(xy_sens[..., 0], linestyle='-', fmt='ko', errtype='std')
lines[0][0].set_label('X')
lines = plot_err_line(xy_sens[..., 1], linestyle='-', fmt='go', errtype='std')
lines[0][0].set_label('Y')
pl.legend()
pl.ylim((-0.1, 0.1))
pl.xlim(0,100)
pl.axhline(y=0, color='0.6', ls='--')
pl.xlabel('Timepoints')

from mvpa2.base import cfg
```

The following figure is not exactly identical to the product of this code, but rather shows the result of a few minutes of beautifications in Inkscape.

See also:

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

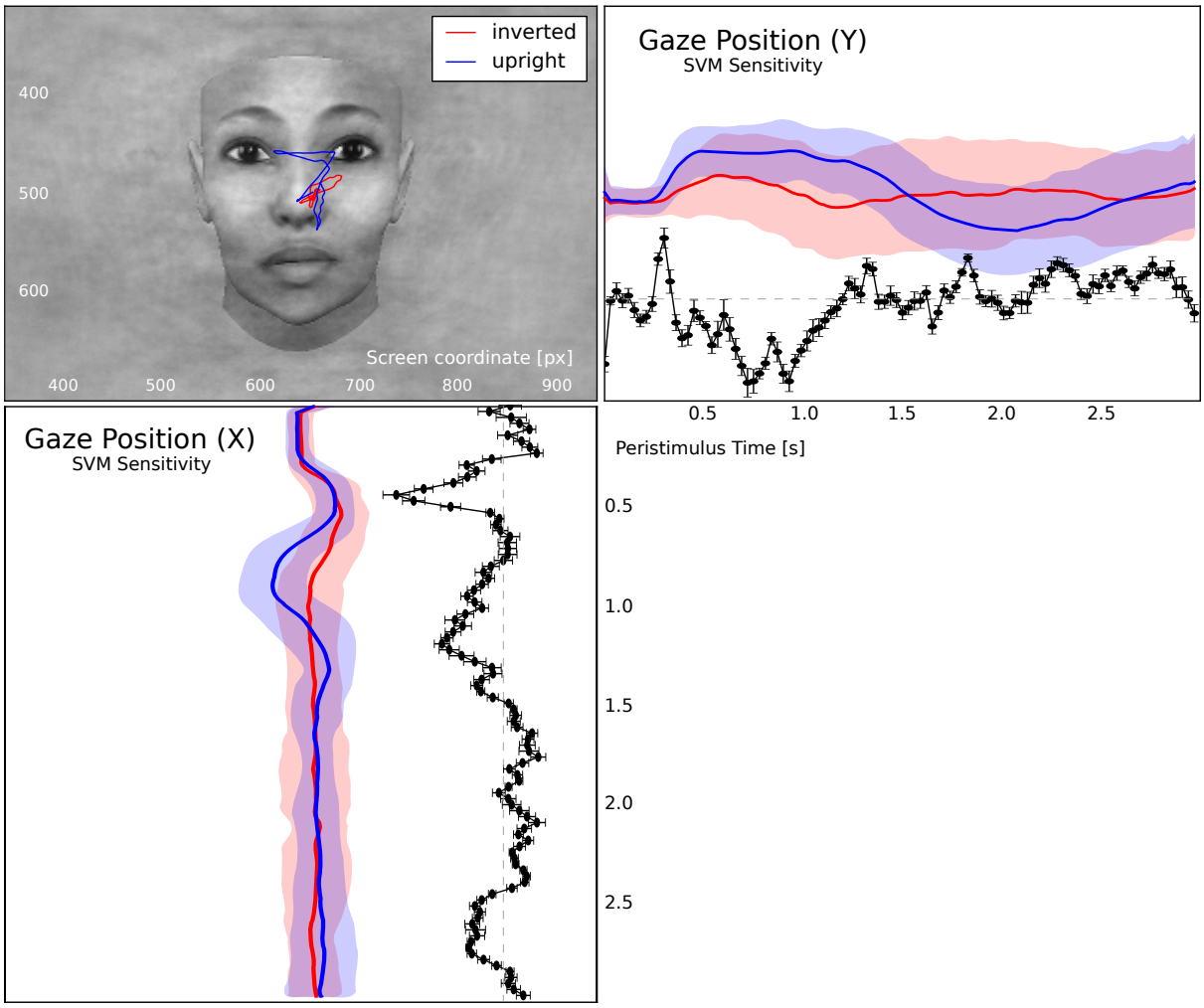


Fig. 6.7: Gaze track for viewing upright vs. inverted faces. The figure shows the mean gaze path for both conditions overlayed on an example face. The panels to the left and below show the X and Y coordinates over the trial timecourse (shaded aread corresponds to one standard deviation across all trials above and below the mean). The black curve depicts the associated temporal SVM weight profile for the classification of both conditions.

6.3 Visualization

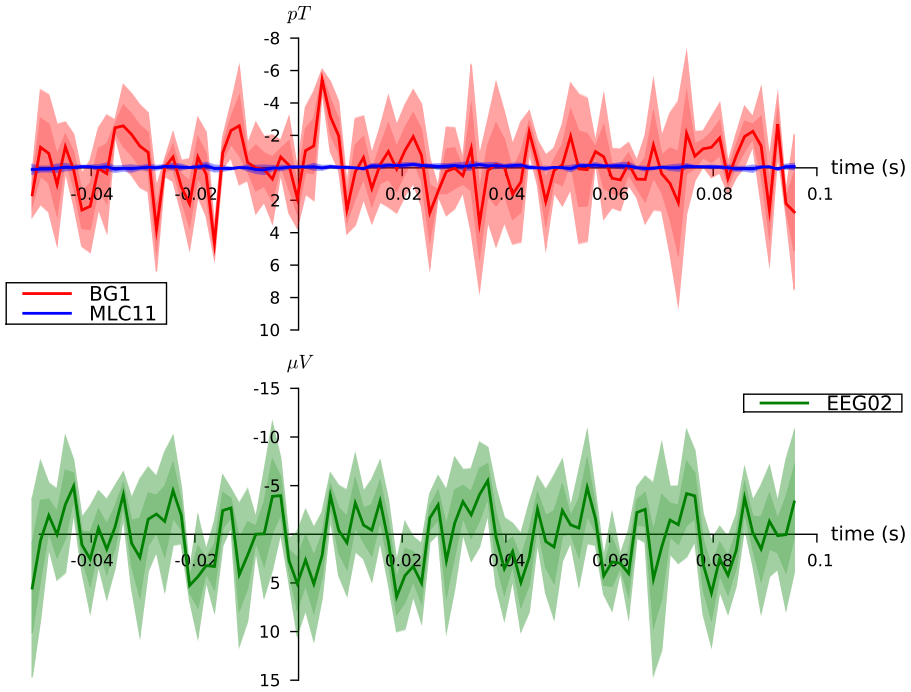
6.3.1 ERP/ERF-Plots

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 mvpa2.suite import *

# load data
meg = TuebingenMEG(os.path.join(pymvpa_dataroot, 'tueb_meg.dat.gz'))


# Define plots for easy feeding into plot_erp
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 = pl.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. plot_erp recreates its own axes centered at (0,0)
ax = fig.add_subplot(2, 1, 1, frame_on=False)
plot_erps(plots[:2], ylabel='$pT$', ymult=-1e12, ax=ax, **cargs)

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

# Additional example: plotting a single ERP on an existing plot
# without drawing axis:
#
# plot_erp(data=meg.data[:, 0, :], SR=meg.samplingrate, pre=pre,
```

```
#          pre_mean=pre, errtype=errtype, ymult=-1.0)
```

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`).

6.3.2 kNN – Model Flexibility in Pictures

TODO

```
import numpy as np
```

```
import mvpa2
from mvpa2.base import cfg
from mvpa2.misc.data_generators import *
from mvpa2.clfs.knn import kNN
from mvpa2.misc.plot import *

mvpa2.seed(0)                           # to reproduce the plot

dataset_kwargs = dict(nfeatures=2, nchunks=10,
    snr=2, nlabels=4, means=[ [0,1], [1,0], [1,1], [0,0] ])

dataset_train = normal_feature_dataset(**dataset_kwargs)
dataset_plot = normal_feature_dataset(**dataset_kwargs)


# make a new figure
pl.figure(figsize=(9, 9))
```

```
for i,k in enumerate((1, 3, 9, 20)):
    knn = kNN(k)

    print "Processing kNN(%i) problem..." % k
    pl.subplot(2, 2, i+1)
```

```
»    knn.train(dataset_train)

     plot_decision_boundary_2d(
         dataset_plot, clf=knn, maps='targets')
```

See also:

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

6.3.3 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 np

# set up the labeled data
# two skewed 2-D distributions
num_dat = 200
dist = 4
# Absolute max value allowed. Just to assure proper plots
xyamax = 10
feat_pos=np.random.randn(2, num_dat)
feat_pos[0, :] *= 2.
feat_pos[1, :] *= .5
feat_pos[0, :] += dist
feat_pos = feat_pos.clip(-xyamax, xyamax)
feat_neg=np.random.randn(2, num_dat)
feat_neg[0, :] *= .5
feat_neg[1, :] *= 2.
feat_neg[0, :] -= dist
feat_neg = feat_neg.clip(-xyamax, xyamax)

# set up the testing features
npoints = 101
x1 = np.linspace(-xyamax, xyamax, npoints)
x2 = np.linspace(-xyamax, xyamax, npoints)
x,y = np.meshgrid(x1, x2);
feat_test = np.array((np.ravel(x), np.ravel(y)))
```

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

```
from mvpa2.suite import *

# create the pymvpa dataset from the labeled features
patternsPos = dataset_wizard(samples=feat_pos.T, targets=1)
patternsNeg = dataset_wizard(samples=feat_neg.T, targets=0)
ds_lin = vstack((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 2
ds_nl = pure_multivariate_signal(30, 2)
l1 = ds_nl.sa['targets'].unique[1]
```

```
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_ca=['probabilities']),
        'RBF SVM': RbfNuSVMC(probability=1,
                       enable_ca=['probabilities']),
        'SMLR': SMLR(lm=0.01),
        'Logistic Regression': PLR(criterion=0.00001),
        '3-Nearest-Neighbour': kNN(k=3),
        '10-Nearest-Neighbour': kNN(k=10),
        'GNB': GNB(common_variance=True),
        'GNB(common_variance=False)': GNB(common_variance=False),
        'LDA': LDA(),
        'QDA': QDA(),
        }

# How many rows/columns we need
nx = int(ceil(np.sqrt(len(clfs))))
ny = int(ceil(len(clfs)/float(nx)))
```

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
    pl.figure(figsize=(nx*4, ny*4))

    print "Processing %s problem..." % id

    for c in sorted(clfs):
        # tell which one we are doing
        print "Running %s classifier..." % (c)

        # make a new subplot for each classifier
        fig += 1
        pl.subplot(ny, nx, fig)

        # select the clasifier
        clf = clfs[c]

        # enable saving of the estimates used for the prediction
        clf.ca.enable('estimates')

        # 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':
            # use the prediction
            res = np.asarray(pre)
        elif 'Nearest-Ne' in c:
```

```
            # Use the dictionaries with votes
            res = np.array([e[l1] for e in clf.ca.estimates]) \
                  / np.sum([e.values() for e in clf.ca.estimates], axis=1)
        elif c == 'Logistic Regression':
            # get out the values used for the prediction
            res = np.asarray(clf.ca.estimates)
        elif c in ['SMLR']:
            res = np.asarray(clf.ca.estimates[:, 1])
        elif c in ['LDA', 'QDA'] or c.startswith('GNB'):
            # Since probabilities are logprobs -- just for
            # visualization of trade-off just plot relative
            # "trade-off" which determines decision boundaries if an
            # alternative log-odd value was chosen for a cutoff
            res = np.asarray(clf.ca.estimates[:, 1]
                             - clf.ca.estimates[:, 0])
            # Scale and position around 0.5
            res = 0.5 + res/max(np.abs(res))
        else:
            # get the probabilities from the svm
            res = np.asarray([(q[1][1] - q[1][0] + 1) / 2
                    for q in clf.ca.probabilities])

        # reshape the results
        z = np.asarray(res).reshape((npoints, npoints))

        # plot the predictions
        pl.pcolor(x, y, z, shading='interp')
        pl.clim(0, 1)
        pl.colorbar()
        # plot decision surfaces at few levels to emphasize the
        # topology
        pl.contour(x, y, z, [0.1, 0.4, 0.5, 0.6, 0.9],
                   linestyles=['dotted', 'dashed', 'solid', 'dashed', 'dotted'],
                   linewidths=1, colors='black', hold=True)

        # plot the training points
        pl.plot(ds.samples[ds.targets == 1, 0],
                ds.samples[ds.targets == 1, 1],
                "r.")
        pl.plot(ds.samples[ds.targets == 0, 0],
                ds.samples[ds.targets == 0, 1],
                "b.")

        pl.axis('tight')
        # add the title
        pl.title(c)
```

See also:

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

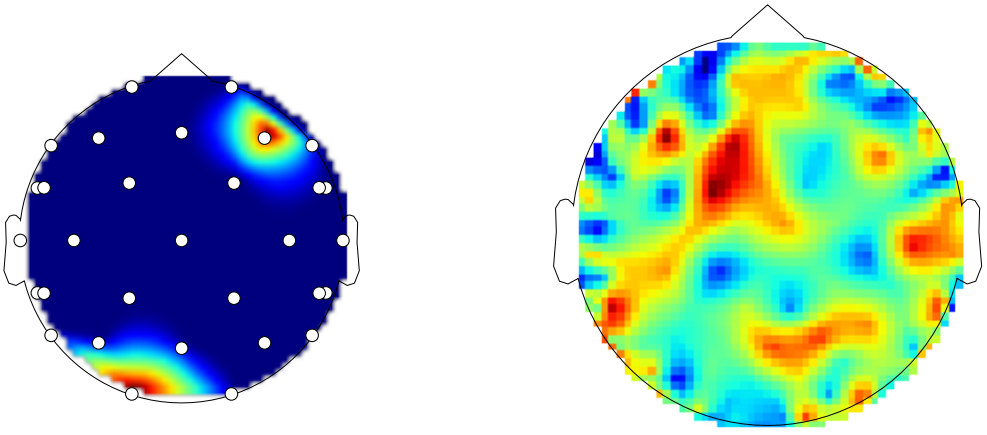
6.3.4 Generating Topography plots

Example demonstrating a topography plot.

```
from mvpa2.suite import *

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

# EEG example splot
```

```
pl.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 = np.zeros(len(sensors.names))
topo[sensors.names.index('O1')] = 1
topo[sensors.names.index('F4')] = 1

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


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

# load MEG sensor locations
sensors=TuebingenMEGSensorLocations(
            os.path.join(pymvpa_dataroot, 'tueb_meg_coord.xyz'))

# random values this time
topo = np.random.randn(len(sensors.names))

# plot without additional interpolation
plot_head_topography(topo, sensors.locations(),
                     interpolation='nearest')
```

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`).

6.3.5 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 mvpa2.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 = np.array(
         [[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']
```

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(colors)
```

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:

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

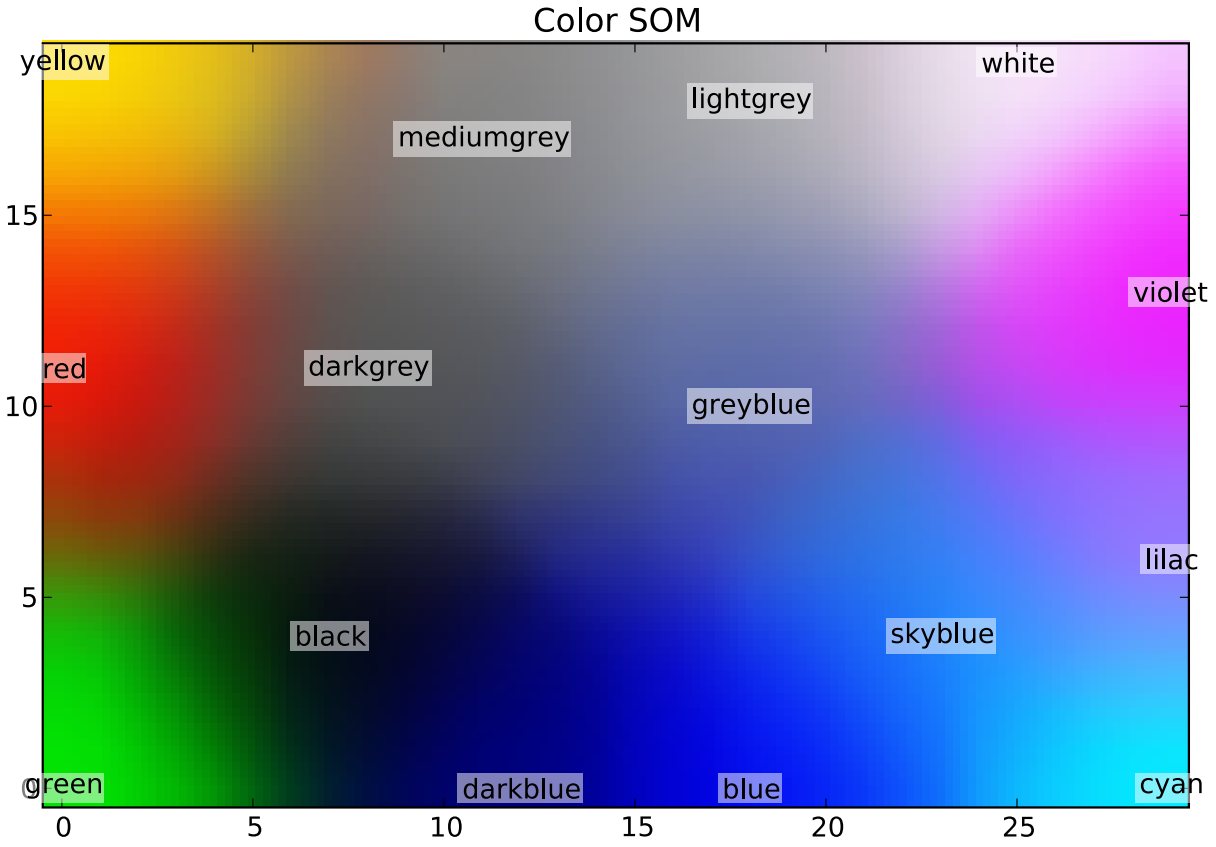
And now, let's take a look onto which coordinates the initial training prototypes were mapped to. The get those coordinates we can simply feed the training data to the mapper and plot the output.

```
mapped = som(colors)

pl.title('Color SOM')
# SOM's kshape is (rows x columns), while matplotlib wants (X x Y)
for i, m in enumerate(mapped):
    pl.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
```

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`).

6.3.6 Basic (f)MRI plotting

When running an fMRI data analysis it is often necessary to visualize results in their original dataspace, typically as an overlay on some anatomical brain image. PyMVPA has the ability to export results into the NIfTI format, and via this data format it is compatible with virtually any MRI visualization software.

However, sometimes having a scriptable plotting facility within Python is desired. There are a number of candidate tools for this purpose (e.g. Mayavi), but also PyMVPA itself offers some basic MRI plotting.

In this example, we are showing a quick-and-dirty plot of a voxel-wise ANOVA measure, overlaid on the respective brain anatomy. Note that the plotting is not specific to ANOVAs. Any feature-wise measure can be plotted this way.

We start with basic steps: loading PyMVPA and the example fMRI dataset, only select voxels that correspond to some pre-computed gray matter mask, do basic preprocessing, and estimate ANOVA scores. This has already been described elsewhere, hence we only provide the code here for the sake of completeness.

```
from mvpa2.suite import *

# load PyMVPA example dataset
datapath = os.path.join(mvpa2.cfg.get('location', 'tutorial data'), 'haxby2001')
dataset = load_tutorial_data(roi='gray')

# do chunkswise linear detrending on dataset
poly_detrend(dataset, chunks_attr='chunks')

# exclude the rest conditions from the dataset, since that should be
# quite different from the 'active' conditions, and make the computation
# below pointless
dataset = dataset[dataset.sa.targets != 'rest']

# define sensitivity analyzer to compute ANOVA F-scores on the remaining
# samples
sensana = OneWayAnova()
sens = sensana(dataset)
```

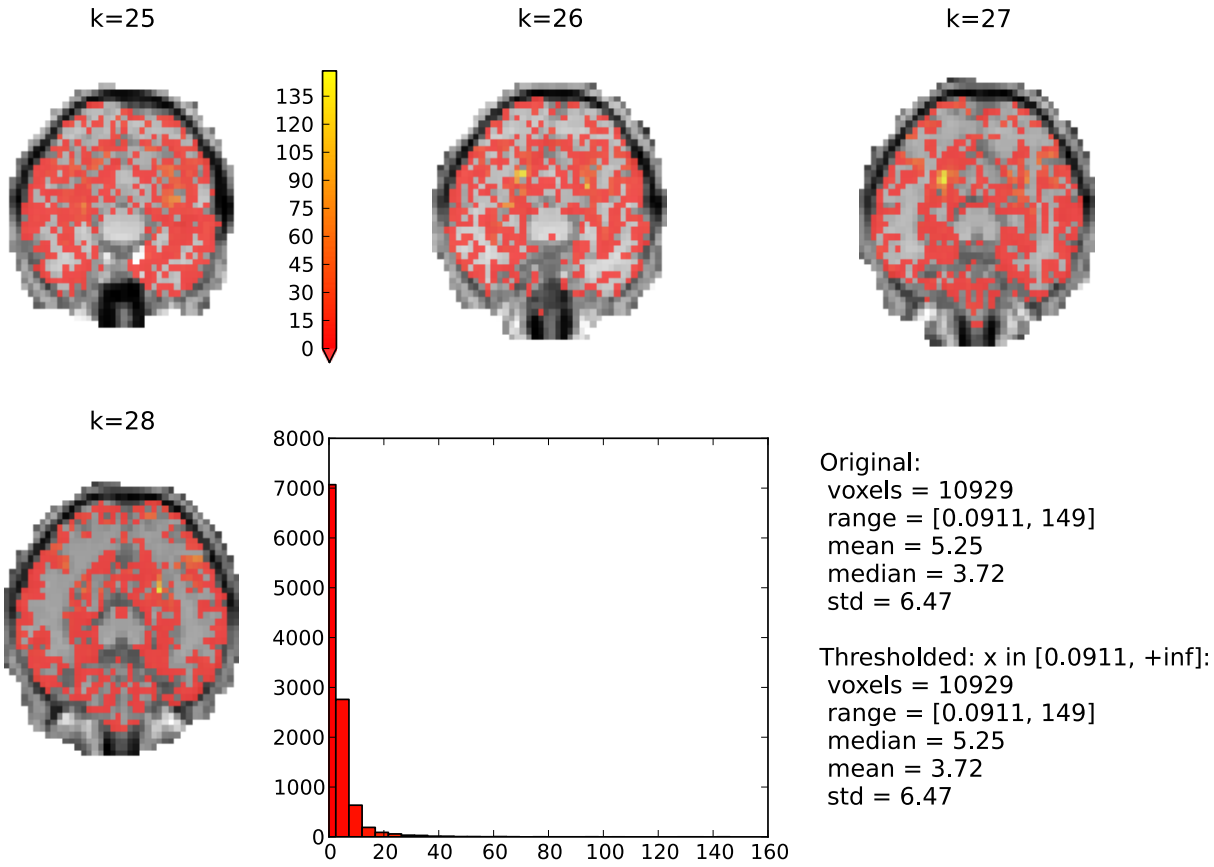
The measure is computed, and we can look at the actual plotting. Typically, it is useful to pre-define some common plotting arguments, for example to ensure consistency throughout multiple figures. This following sets up which backround image to use (`background`), which portions of the image to plot (`background_mask`), and which portions of the overlay images to plot (`overlay_mask`). All these arguments are actually NIfTI images of the same dimensions and orientation as the to be plotted F-scores image. the remaining settings configure the colormaps to be used for plotting and trigger interactive plotting.

```
mri_args = {
    'background' : os.path.join(datapath, 'sub001', 'anatomy', 'highres001.nii.gz'),
    'background_mask' : os.path.join(datapath, 'sub001', 'masks', 'orig', 'brain.nii.gz'),
    'overlay_mask' : os.path.join(datapath, 'sub001', 'masks', 'orig', 'gray.nii.gz'),
    'cmap_bg' : 'gray',
    'cmap_overlay' : 'autumn', # YlOrRd_r # pl.cm.autumn
    'interactive' : cfg.getboolean('examples', 'interactive', True),
    }
```

All that remains to do is a single call to `plot_lightbox()`. We pass it the F-score vector. `map2nifti` uses the mapper in our original dataset to project it back into the functional MRI volume space. We treshold the data with the interval [0, +inf] (i.e. all possible values and F-Score can have), and select a subset of slices to be plotted. That's it.

```
fig = plot_lightbox(overlay=map2nifti(dataset, sens),
                    vlim=(0, None), slices=range(25,29), **mri_args)
```

The resulting figure would look like this:



In interactive mode it is possible to click on the histogram to adjust the thresholding of the overlay volumes. Left-click sets the value corresponding to the lowest value in the colormap, and right-click set the value for the upper end of the colormap. Try right-clicking somewhere at the beginning of the x-axis and left on the end of the x-axis.

See also:

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

6.4 Integrate with 3rd-party software

6.4.1 Using scikit-learn transformers with PyMVPA

Scikit-learn is a rich library of algorithms, many of them implementing the transformer API. PyMVPA provides a wrapper class, `SKLTransformer` that enables the use of all of these algorithms within the PyMVPA framework. With this adaptor the transformer API is presented as a PyMVPA mapper interface that is fully compatible with all other building blocks of PyMVPA.

In this example we demonstrate this interface by mimicking the "Comparison of Manifold Learning methods" example from the scikit-learn documentation – applying the minimal modifications necessary to run a variety of scikit-learn algorithm implementation on PyMVPA datasets.

This script also prints the same timing information as the original.

```
print(__doc__)

from time import time

import pylab as pl
from mpl_toolkits.mplot3d import Axes3D
from matplotlib.ticker import NullFormatter
```

```
from sklearn import manifold
# Next line to silence pyflakes. This import is needed.
Axes3D

n_points = 1000
n_neighbors = 10
n_components = 2
```

So far the code has been identical. The first difference is the import of the adaptor class. We also load the scikit-learn demo dataset, but also with the help of a wrapper function that yields a PyMVPA dataset.

```
# this first import is only required to run the example a part of the test suite
from mvpa2 import cfg
from mvpa2.mappers.skl_adaptor import SKLTransformer

# load the S-curve dataset
from mvpa2.datasets.sources.skl_data import skl_s_curve
ds = skl_s_curve(n_points)
```

And we continue with practically identical code.

```
fig = pl.figure(figsize=(15, 8))
pl.suptitle("Manifold Learning with %i points, %i neighbors"
            % (1000, n_neighbors), fontsize=14)

try:
    # compatibility matplotlib < 1.0
    X = ds.samples
    ax = fig.add_subplot(241, projection='3d')
    ax.scatter(X[:, 0], X[:, 1], X[:, 2], c=ds.targets, cmap=pl.cm.Spectral)
    ax.view_init(4, -72)
except:
    X = ds.samples
    ax = fig.add_subplot(241, projection='3d')
    pl.scatter(X[:, 0], X[:, 2], c=ds.targets, cmap=pl.cm.Spectral)

methods = ['standard', 'ltsa', 'hessian', 'modified']
labels = ['LLE', 'LTSA', 'Hessian LLE', 'Modified LLE']

for i, method in enumerate(methods):
    t0 = time()
    # create an instance of the algorithm from scikit-learn
    # and wrap it by SKLTransformer
```

The following lines are an example of the only significant modification with respect to a pure scikit-learn implementation: the transformer is wrapped into the adaptor. The result is a mapper, hence can be called with a dataset that contains both samples and targets – without explcitly calling `fit()` and `transform()`.

```
»    lle = SKLTransformer(manifold.LocallyLinearEmbedding(n_neighbors,
                                                          n_components,
                                                          eigen_solver='auto',
                                                          method=method))
     # call the SKLTransformer instance on the input dataset
     Y = lle(ds)
```

The rest of the example is unmodified except for the wrapping of the respective transformer into the Mapper adaptor.

```
»    t1 = time()
     print("%s: %.2g sec" % (methods[i], t1 - t0))

     ax = fig.add_subplot(242 + i)
     pl.scatter(Y[:, 0], Y[:, 1], c=ds.targets, cmap=pl.cm.Spectral)
     pl.title("%s (%.2g sec)" % (labels[i], t1 - t0))
```

```
    ax.xaxis.set_major_formatter(NullFormatter())
    ax.yaxis.set_major_formatter(NullFormatter())
    pl.axis('tight')

t0 = time()
# create an instance of the algorithm from scikit-learn
# and wrap it by SKLTransformer
iso = SKLTransformer(manifold.Isomap(n_neighbors=10, n_components=2))
# call the SKLTransformer instance on the input dataset
Y = iso(ds)
t1 = time()
print("Isomap: %.2g sec" % (t1 - t0))
ax = fig.add_subplot(246)
pl.scatter(Y[:, 0], Y[:, 1], c=ds.targets, cmap=pl.cm.Spectral)
pl.title("Isomap (%.2g sec)" % (t1 - t0))
ax.xaxis.set_major_formatter(NullFormatter())
ax.yaxis.set_major_formatter(NullFormatter())
pl.axis('tight')


t0 = time()
# create an instance of the algorithm from scikit-learn
# and wrap it by SKLTransformer
mds = SKLTransformer(manifold.MDS(n_components=2, max_iter=100,
                                  n_init=1, dissimilarity='euclidean'))
# call the SKLTransformer instance on the input dataset
Y = mds(ds)
t1 = time()
print("MDS: %.2g sec" % (t1 - t0))
ax = fig.add_subplot(247)
pl.scatter(Y[:, 0], Y[:, 1], c=ds.targets, cmap=pl.cm.Spectral)
pl.title("MDS (%.2g sec)" % (t1 - t0))
ax.xaxis.set_major_formatter(NullFormatter())
ax.yaxis.set_major_formatter(NullFormatter())
pl.axis('tight')


t0 = time()
# create an instance of the algorithm from scikit-learn
# and wrap it by SKLTransformer
se = SKLTransformer(manifold.SpectralEmbedding(n_components=n_components,
                                               n_neighbors=n_neighbors))
# call the SKLTransformer instance on the input dataset
Y = se(ds)
t1 = time()
print("SpectralEmbedding: %.2g sec" % (t1 - t0))
ax = fig.add_subplot(248)
pl.scatter(Y[:, 0], Y[:, 1], c=ds.targets, cmap=pl.cm.Spectral)
pl.title("SpectralEmbedding (%.2g sec)" % (t1 - t0))
ax.xaxis.set_major_formatter(NullFormatter())
ax.yaxis.set_major_formatter(NullFormatter())
pl.axis('tight')
```

See also:

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

6.4.2 Using scikit-learn classifiers with PyMVPA

Scikit-learn is a rich library of algorithms, many of them implementing the estimator and predictor API. PyMVPA provides the wrapper class, `SKLLearnerAdapter` that enables the use of all of these algorithms within the

PyMVPA framework. With this adaptor these aspects of the scikit-learn API are presented through a PyMVPA learner interface that is fully compatible with all other building blocks of PyMVPA.

In this example we demonstrate this interface by mimicking the "Nearest Neighbors Classification" example from the scikit-learn documentation – applying the minimal modifications necessary to run two variants of the scikit-learn k-nearest neighbors algorithm implementation on PyMVPA datasets.

```
print(__doc__)

import numpy as np
import pylab as pl
from matplotlib.colors import ListedColormap
from sklearn import neighbors

n_neighbors = 15
```

So far the code has been identical. The first difference is the import of the adaptor class. We also load the scikit-learn demo dataset, but also with the help of a wrapper function that yields a PyMVPA dataset.

```
# this first import is only required to run the example a part of the test suite
from mvpa2 import cfg
from mvpa2.clfs.skl.base import SKLLearnerAdapter

# load the iris dataset
from mvpa2.datasets.sources.skl_data import skl_iris
iris = skl_iris()
# compact dataset summary
print iris
```

The original example uses only the first two features of the dataset, since it intends to visualize learned classification boundaries in 2-D. We can do the same slicing directly on our PyMVPA dataset.

```
iris=iris[:,[0,1]]

d = {'setosa':0, 'versicolor':1, 'virginica':2}
```

For visualization we will later map the literal class labels onto numerival values. Besides that, we continue with practically identical code.

```
h = .02  # step size in the mesh

cmap_light = ListedColormap(['#FFAAAA', '#AAFFAA', '#AAAAFF'])
cmap_bold = ListedColormap(['#FF0000', '#00FF00', '#0000FF'])

for weights in ['uniform', 'distance']:
    # create an instance of the algorithm from scikit-learn,
    # wrap it by SKLLearnerAdapter and finally train it
    clf = neighbors.KNeighborsClassifier(n_neighbors, weights=weights)
    wrapped_clf=SKLLearnerAdapter(clf)
    wrapped_clf.train(iris)
```

The following lines are an example of the only significant modification with respect to a pure scikit-learn implementation: the classifier is wrapped into the adaptor. The result is a PyMVPA classifier, hence can be called with a dataset that contains both samples and targets.

```
»    # shortcut
     X = iris.samples
     # Plot the decision boundary. For that, we will assign a color to each
     # point in the mesh [x_min, m_max]x[y_min, y_max].
     x_min, x_max = X[:, 0].min() - 1, X[:, 0].max() + 1
     y_min, y_max = X[:, 1].min() - 1, X[:, 1].max() + 1
     xx, yy = np.meshgrid(np.arange(x_min, x_max, h),
                          np.arange(y_min, y_max, h))
     Z = wrapped_clf.predict(np.c_[xx.ravel(), yy.ravel()])
```

```
    # to put the result into a color plot we now need numerical values
    # this can be done nicely in PyMVPA
    from mvpa2.datasets.formats import AttributeMap
    Z = AttributeMap().to_numeric(Z)
    Z = Z.reshape(xx.shape)

    pl.figure()
    pl.pcolormesh(xx, yy, Z, cmap=cmap_light)

    # For plotting the training points we convert to numerical values again
    pl.scatter(X[:, 0], X[:, 1], c=AttributeMap().to_numeric(iris.targets),
                                 cmap=cmap_bold)
    pl.xlim(xx.min(), xx.max())
    pl.ylim(yy.min(), yy.max())
    pl.title("3-Class classification (k = %i, weights = '%s')"
             % (n_neighbors, weights))
```

This example shows that a PyMVPA classifier can be used in pretty much the same way as the corresponding scikit-learn API. What this example does not show is that with the `SKLLearnerAdapter` class any scikit-learn classifier can be employed in arbitrarily complex PyMVPA processing pipelines and is enhanced with automatic training and all other functionality of PyMVPA classifier implementations.

See also:

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

6.4.3 Using scikit-learn regressions with PyMVPA

This is practically part two of the example on `using scikit-learn classifiers with PyMVPA`. Just like classifiers, implementations of regression algorithms in scikit-learn use the estimator and predictor API. Consequently, the same wrapper class (`SKLLearnerAdapter`) as before is applicable when using scikit-learn regressions in PyMVPA.

The example demonstrates this by mimicking the "Decision Tree Regression" example from the scikit-learn documentation – applying the minimal modifications necessary to the scikit-learn decision tree regression algorithm (with two different parameter settings) implementation on a PyMVPA dataset.

```
print(__doc__)

import numpy as np
from sklearn.tree import DecisionTreeRegressor

rng = np.random.RandomState(1)
X = np.sort(5 * rng.rand(80, 1), axis=0)
y = np.sin(X).ravel()
y[::5] += 3 * (0.5 - rng.rand(16))
```

So far the code has been identical. The first difference is the import of the adaptor class. We also use a convenient way to convert the data into a proper `Dataset`.

```
# this first import is only required to run the example a part of the test suite
from mvpa2 import cfg
from mvpa2.clfs.skl.base import SKLLearnerAdapter
from mvpa2.datasets import dataset_wizard
ds_train=dataset_wizard(samples=X, targets=y)
```

The following lines are an example of the only significant modification with respect to a pure scikit-learn implementation: the regression is wrapped into the adaptor. The result is a PyMVPA learner, hence can be called with a dataset that contains both samples and targets.

```
clf_1 = SKLLearnerAdapter(DecisionTreeRegressor(max_depth=2))
clf_2 = SKLLearnerAdapter(DecisionTreeRegressor(max_depth=5))

clf_1.train(ds_train)
clf_2.train(ds_train)

X_test = np.arange(0.0, 5.0, 0.01)[:, np.newaxis]
y_1 = clf_1.predict(X_test)
y_2 = clf_2.predict(X_test)


# plot the results
# which clearly show the overfitting for the second depth setting
import pylab as pl

pl.figure()
pl.scatter(X, y, c="k", label="data")
pl.plot(X_test, y_1, c="g", label="max_depth=2", linewidth=2)
pl.plot(X_test, y_2, c="r", label="max_depth=5", linewidth=2)
pl.xlabel("data")
pl.ylabel("target")
pl.title("Decision Tree Regression")
pl.legend()
```

See also:

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

6.4.4 Classifying the MNIST handwritten digits with MDP

This example will demonstrate how to embed MDP's flows into a PyMVPA-based analysis. We will perform a classification of a large number of images of handwritten digits from the MNIST database. To get a better sense of how MDP blends into PyMVPA, we will do the same analysis with MDP only first, and then redo it in PyMVPA – only using particular bits from MDP.

But first we need some helper to load the MNIST data. The following function will load four NumPy arrays from an HDF5 file in the PyMVPA Data DB. These arrays are the digit images and the numerical labels for training and testing dataset respectively. All 28x28 pixel images are stored as flattened vectors.

```
import os
from mvpa2.base.hdf5 import h5load
from mvpa2 import pymvpa_datadbroot

def load_data():
    data = h5load(os.path.join(pymvpa_datadbroot, 'mnist', "mnist.hdf5"))
    traindata = data['train'].samples
    trainlabels = data['train'].sa.labels
    testdata = data['test'].samples
    testlabels = data['test'].sa.labels
    return traindata, trainlabels, testdata, testlabels
```

MDP-style classification

Here is how to get the classification of the digit images done in MDP. The data is preprocessed by whitening, followed by polynomial expansion, and a subsequent projection on a nine-dimensional discriminant analysis solution. There is absolutely no need to do this particular pre-processing, it is just done to show off some MDP features. The actual classification is performed by a Gaussian classifier. The training data needs to be fed in different ways to the individual nodes of the flow. The whitening needs only the images, polynomial expansion

needs no training at all, and FDA as well as the classifier also need the labels. Moreover, a custom iterator is used to feed data in chunks to the last two nodes of the flow.

```
import numpy as np
import mdp

class DigitsIterator:
    def __init__(self, digits, labels):
        self.digits = digits
        self.targets = labels
    def __iter__(self):
        frac = 10
        ll = len(self.targets)
        for i in xrange(frac):
            yield self.digits[i*ll/frac:(i+1)*ll/frac], \
                  self.targets[i*ll/frac:(i+1)*ll/frac]

traindata, trainlabels, testdata, testlabels = load_data()

fdaclf = (mdp.nodes.WhiteningNode(output_dim=10, dtype='d') +
          mdp.nodes.PolynomialExpansionNode(2) +
          mdp.nodes.FDANode(output_dim=9) +
          mdp.nodes.GaussianClassifier())

fdaclf.verbose = True

fdaclf.train([[traindata],
              None,
              DigitsIterator(traindata,
                             trainlabels),
              DigitsIterator(traindata,
                             trainlabels)
              ])
```

After training, we feed the test data through the flow to obtain the predictions. First through the pre-processing nodes and then through the classifier, extracting the predicted labels only. Finally, the prediction error is computed.

```
feature_space = fdaclf[:-1](testdata)
guess = fdaclf[-1].label(feature_space)
err = 1 - np.mean(guess == testlabels)
print 'Test error:', err
```

Doing it the PyMVPA way

Analog to the previous approach we load the data first. This time, however, we convert it into a PyMVPA dataset. Training and testing data are initially created as two separate datasets, get tagged as 'train' and 'test' respectively, and are finally stacked into a single Dataset of 70000 images and their numerical labels.

```
import pylab as pl
from mvpa2.suite import *

traindata, trainlabels, testdata, testlabels = load_data()
train = dataset_wizard(
        traindata,
        targets=trainlabels,
        chunks='train')
test = dataset_wizard(
        testdata,
        targets=testlabels,
        chunks='test')
# merge the datasets into on
ds = vstack((train, test))
```

```
ds.init_origids('samples')
```

For this analysis we will use the exact same pre-processing as in the MDP code above, by using the same MDP nodes, in an MDP flow that is shortened only by the Gaussian classifier. The glue between these MDP nodes and PyMVPA is the `MDPFlowMapper`. This mapper is able to supply nodes with optional arguments for their training. In this example a `DatasetAttributeExtractor` is used to feed the labels of the training dataset to the FDA node (in addition to the training data itself).

```
fdaflow = (mdp.nodes.WhiteningNode(output_dim=10, dtype='d') +
           mdp.nodes.PolynomialExpansionNode(2) +
           mdp.nodes.FDANode(output_dim=9))
fdaflow.verbose = True

mapper = MDPFlowMapper(fdaflow,
                       ([], [], [DatasetAttributeExtractor('sa', 'targets')]))
```

The `MDPFlowMapper` can represent any MDP flow as a PyMVPA mapper. In this example, we attach the MDP-based pre-processing flow, wrapped in the mapper, to a classifier (arbitrarily chosen to be SMLR) via a `MappedClassifier`. In doing so we achieve that the training data is automatically pre-processed before it is used to train the classifier, and later on the same pre-processing it applied to the testing data, before the classifier is asked to make its predictions.

At last we wrap the MappedClassifier into a `TransferMeasure` that splits the dataset into a training and testing part. In this particular case this is not really necessary, as we could have left training and testing data separate in the first place, and could have called the classifier's `train()` and `predict()` manually. However, when doing repeated train/test cycles as, for example, in a cross-validation this is not very useful. In this particular case the TransferMeasure computes a number of performance measures for us that we only need to extract.

```
tm = TransferMeasure(MappedClassifier(SMLR(), mapper),
                     Splitter('chunks', attr_values=['train', 'test']),
                     enable_ca=['stats', 'samples_error'])
tm(ds)
print 'Test error:', 1 - tm.ca.stats.stats['ACC']
```

Visualizing data and results

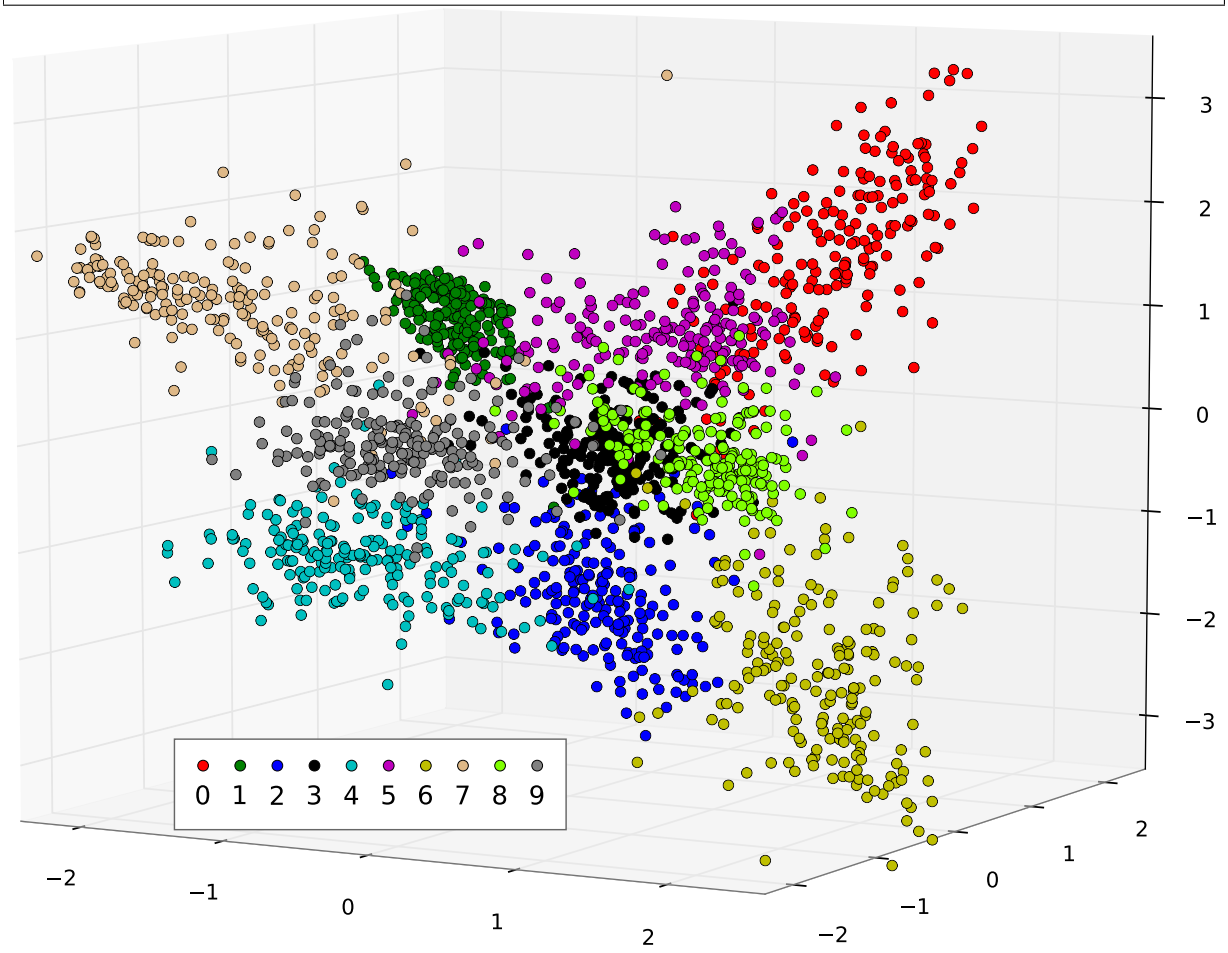
The analyses are already done. But for the sake of completeness we take a final look at both data and results. First and few examples of the training data.

```
examples = [3001 + 5940 * i for i in range(10)]

pl.figure(figsize=(2, 5))

for i, id_ in enumerate(examples):
    ax = pl.subplot(2, 5, i+1)
    ax.axison = False
    pl.imshow(traindata[id_].reshape(28, 28).T, cmap=pl.cm.gist_yarg,
              interpolation='nearest', aspect='equal')

pl.subplots_adjust(left=0, right=1, bottom=0, top=1,
                   wspace=0.05, hspace=0.05)
pl.draw()
```

And finally we take a peak at the result of pre-processing for a number of example images for each digit. The following plot shows the training data on hand-picked three-dimensional subset of the original nine FDA dimension the data was projected on.

```
if externals.exists('matplotlib') \
   and externals.versions['matplotlib'] >= '0.99':
    from mpl_toolkits.mplot3d import Axes3D
    pts = []
```

```
    for i, ex in enumerate(examples):
        pts.append(mapper.forward(ds.samples[ex:ex+200])[:, :3])

    fig = pl.figure()

    ax = Axes3D(fig)
    colors = ('r','g','b','k','c','m','y','burlywood','chartreuse','gray')
    clouds = []
    for i, p in enumerate(pts):
        print i
        clouds.append(ax.plot(p[:, 0], p[:, 1], p[:, 2], 'o', c=colors[i],
                              label=str(i), alpha=0.6))

    ax.legend([str(i) for i in range(10)])
    pl.draw()
```



Note: The data visualization depends on the 3D matplotlib features, which are available only from version 0.99.

See also:

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

6.5 Special interest and Miscellaneous

6.5.1 Kernel-Demo

This is an example demonstrating various kernel implementation in PyMVPA.

```
import numpy as np
from mvpa2.support.pylab import pl

#from mvpa2.suite import *
from mvpa2.base import cfg
from mvpa2.kernels.np import *


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

for kernel_class, kernel_args in (
    (ConstantKernel, {'sigma_0':1.0}),
    (ConstantKernel, {'sigma_0':1.0}),
    (GeneralizedLinearKernel, {'Sigma_p':np.eye(data.shape[1])}),
    (GeneralizedLinearKernel, {'Sigma_p':np.ones(data.shape[1])}),
    (GeneralizedLinearKernel, {'Sigma_p':2.0}),
    (GeneralizedLinearKernel, {}),
    (ExponentialKernel, {}),
    (SquaredExponentialKernel, {}),
    (Matern_3_2Kernel, {}),
    (Matern_5_2Kernel, {}),
    (RationalQuadraticKernel, {}),
    ):
    kernel = kernel_class(**kernel_args)
    print kernel
    result = kernel.compute(data)

# In the following we draw some 2D functions at random from the
# distribution N(O,kernel) defined by each available kernel and
# plot them. These plots shows the flexibility of a given kernel
# (with default parameters) when doing interpolation. The choice
# of a kernel defines a prior probability over the function space
# used for regression/classfication with GPR/GPC.
count = 1
for k in kernel_dictionary.keys():
    pl.subplot(3, 4, count)
    # X = np.random.rand(size)*12.0-6.0
    # X.sort()
    X = np.arange(-1, 1, .02)
    X = X[:, np.newaxis]
    ker = kernel_dictionary[k]()
    ker.compute(X, X)
    print k
    K = np.asarray(ker)
    for i in range(10):
        f = np.random.multivariate_normal(np.zeros(X.shape[0]), K)
        pl.plot(X[:, 0], f, "b-")

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

See also:

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

6.5.2 Efficient cross-validation using a cached kernel

This is a simple example showing how to use cached kernel with a SVM classifier from the Shogun library. Pre-caching of the kernel for all samples in dataset eliminates necessity of possibly lengthy recomputation of the same

kernel values on different splits of the data. Depending on the data it might provide considerable speed-ups.

```
from mvpa2.suite import *
from time import time
```

The next few calls load an fMRI dataset and do basic preprocessing.

```
# load PyMVPA example dataset
attr = SampleAttributes(os.path.join(pymvpa_dataroot,
                         'attributes_literal.txt'))
dataset = fmri_dataset(samples=os.path.join(pymvpa_dataroot, 'bold.nii.gz'),
                       targets=attr.targets, chunks=attr.chunks,
                       mask=os.path.join(pymvpa_dataroot, 'mask.nii.gz'))

# do chunkswise linear detrending on dataset
poly_detrend(dataset, polyord=1, chunks_attr='chunks')

# zscore dataset relative to baseline ('rest') mean
zscore(dataset, chunks_attr='chunks', param_est=('targets', ['rest']))
```

Cached kernel is just a proxy around an existing kernel.

```
# setup a cached kernel
kernel_plain = LinearSGKernel(normalizer_cls=False)
kernel = CachedKernel(kernel_plain)
```

Lets setup two cross-validation, where first would use cached kernel, whenever the later one plain kernel to demonstrate the speed-up and achievement of exactly the same results

```
# setup a classifier and cross-validation procedure
clf = sg.SVM(svm_impl='libsvm', C=-1.0, kernel=kernel)
cv = CrossValidation(clf, NFoldPartitioner())

# setup exactly the same using a plain kernel for demonstration of
# speedup and equivalence of the results
clf_plain = sg.SVM(svm_impl='libsvm', C=-1.0, kernel=kernel_plain)
cv_plain = CrossValidation(clf_plain, NFoldPartitioner())
```

Although it would be done internally by cached kernel during initial computation, it is advisable to make initialization of origids for samples explicit. It would prepare dataset by cleaning up attributes used by cached kernel possibly on another version of the same dataset prior to this analysis in real use cases.

```
dataset.init_origids(which='samples')
```

Cached kernel needs to be computed given the full dataset which would later on be used during cross-validation.

```
# compute kernel for the dataset
t0 = time()
kernel.compute(dataset)
t_caching = time() - t0
```

Lets run both cross-validation procedures using plain and cached kernels and report the results.

```
# run cached cross-validation
t0 = time()
error = np.mean(cv(dataset))
t_cached = time() - t0

# run plain SVM cross-validation for validation and benchmarking
t0 = time()
error_plain = np.mean(cv_plain(dataset))
t_plain = time() - t0

# UC: unique chunks, UT: unique targets
print "Results for %i-fold cross-validation on %i-class problem:" \
```

```
      % (len(dataset.UC), len(dataset.UT))
print " plain kernel:  error=%.3f computed in %.2f sec" \
      % (error_plain, t_plain)
print " cached kernel: error=%.3f computed in %.2f sec (cached in %.2f sec)" \
      % (error, t_cached, t_caching)
```

The following is output from running this example:

```
Results for 12-fold cross-validation on 9-class problem:
 plain kernel:  error=0.273 computed in 35.82 sec
 cached kernel: error=0.273 computed in 6.50 sec (cached in 3.68 sec)
```

See also:

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

6.5.3 Curve-Fitting

Here we are going to take a look at a few examples of fitting a function to data. The first example shows how to fit an HRF model to noisy peristimulus time-series data.

First, importing the necessary pieces:

```
import numpy as np
from scipy.stats import norm

from mvpa2.support.pylab import pl
from mvpa2.misc.plot import plot_err_line, plot_bars
from mvpa2.misc.fx import *
from mvpa2 import cfg
```

BOLD-Response parameters

Let's generate some noisy "trial time courses" from a simple gamma function (40 samples, 6s time-to-peak, 7s FWHM and no additional scaling:

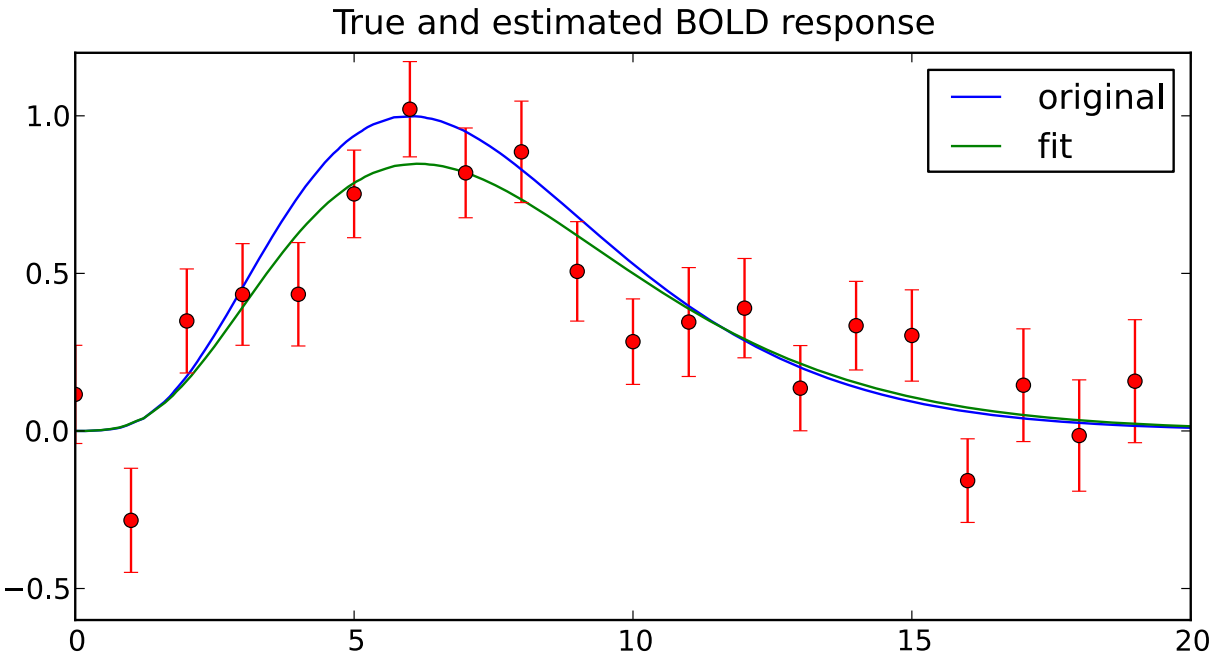
```
a = np.asarray([single_gamma_hrf(np.arange(20), A=6, W=7, K=1)] * 40)
# get closer to reality with noise
a += np.random.normal(size=a.shape)
```

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 = least_sq_fit(single_gamma_hrf, [5,5,1], a)
```

With these parameters we can compute high-resultion curves for the estimated time course, and plot it together with the "true" time course, and the data:

```
x = np.linspace(0,20)
curves = [(x, single_gamma_hrf(x, 6, 7, 1)),
          (x, single_gamma_hrf(x, *fpar))]

# plot data (with error bars) and both curves
plot_err_line(a, curves=curves, linestyle='-')

# add legend to plot
pl.legend(('original', 'fit'))
pl.title('True and estimated BOLD response')
```



Searchlight accuracy distributions

When doing a searchlight analysis one might have the idea that the resulting accuracies are actually sampled from two distributions: one causes by an actual signal source and the chance distribution. Let's assume the these two distributions can be approximated by a Gaussian, and take a look at a toy example, how we could explore the data.

First, we generate us a few searchlight accuracy maps that might have been computed in the folds of a cross-validation procedure. We generate the data from two normal distributions. The majority of datapoints comes from the chance distribution that is centered at 0.5. A fraction of the data is samples from the "signal" distribution located around 0.75.

```
nfolds = 10
raw_data = np.vstack([np.concatenate((np.random.normal(0.5, 0.08, 10000),
                                      np.random.normal(0.75, 0.05, 500)))
                      for i in range(nfolds)])
```

Now we bin the data into one histogram per fold and fit a dual Gaussian (the sum of two Gaussians) to the total of 10 histograms.

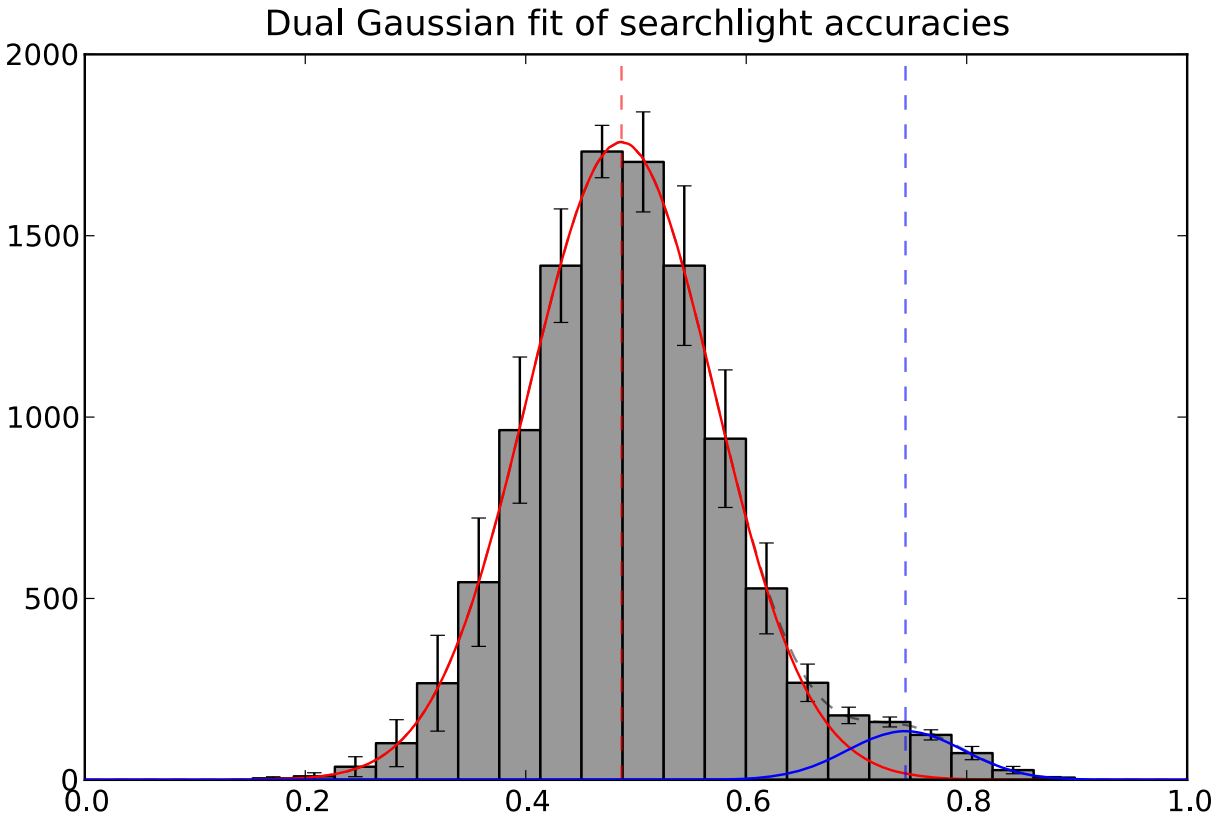
```
histfit = fit2histogram(raw_data,
                        dual_gaussian, (1000, 0.5, 0.1, 1000, 0.8, 0.05),
                        nbins=20)
H, bin_left, bin_width, fit = histfit
```

All that is left to do is composing a figure – showing the accuracy histogram and its variation across folds, as well as the two estimated Gaussians.

```
# new figure
pl.figure()

# Gaussian parameters
params = fit[0]

# plot the histogram
plot_bars(H.T, xloc=bin_left, width=bin_width, yerr='std')

# show the Gaussians
x = np.linspace(0, 1, 100)
# first gaussian
pl.plot(x, params[0] * norm.pdf(x, params[1], params[2]), "r-", zorder=2)
```

```
pl.axvline(params[1], color='r', linestyle='--', alpha=0.6)
# second gaussian
pl.plot(x, params[3] * norm.pdf(x, params[4], params[5]), "b-", zorder=3)
pl.axvline(params[4], color='b', linestyle='--', alpha=0.6)
# dual gaussian
pl.plot(x, dual_gaussian(x, *params), "k--", alpha=0.5, zorder=1)
pl.xlim(0, 1)
pl.ylim(ymin=0)

pl.title('Dual Gaussian fit of searchlight accuracies')
```

And this is how it looks like.



See also:

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

6.5.4 Classifier Sweep

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

```
from mvpa2.suite import *

# no MVPA warnings during this example
warning.handlers = []

def main():

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


    # Load Haxby dataset example
```

```
    haxby8 = load_example_fmri_dataset(literal=True)
    haxby8.samples = haxby8.samples.astype(np.float32)

    # preprocess slightly
    poly_detrend(haxby8, chunks_attr='chunks', polyord=1)
    zscore(haxby8, chunks_attr='chunks', param_est=('targets', 'rest'))

    haxby8_no0 = haxby8[haxby8.targets != 'rest']

    dummy2 = normal_feature_dataset(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[:]),
        ((pure_multivariate_signal(8, 3),
          "Dummy XOR-pattern"),
          clfswh['non-linear']),
        ((haxby8_no0,
          "Haxby 8-cat subject 1"),
          clfswh['multiclass']),
        ]:
        # XXX put back whenever there is summary() again
        #print "%s\n %s" % (datasetdescr, dataset.summary(idhash=False))
        print " Classifier on %s\n" \
              "                                                 :   %%corr    " \
              "#features\t train  predict full" % datasetdescr
        for clf in clfs_:
            print "  %-40s: "  % clf.descr,
            # Let's prevent failures of the entire script if some particular
            # classifier is not appropriate for the data
            try:
                # Change to False if you want to use CrossValidation
                # helper, instead of going through splits manually to
                # track training/prediction time of the classifiers
                do_explicit_splitting = True
                if not do_explicit_splitting:
                    cv = CrossValidation(
                        clf, NFoldPartitioner(), enable_ca=['stats', 'calling_time'])
                    error = cv(dataset)
                    # print cv.ca.stats
                    print "%5.1f%%      -    \t   -       -    %.2fs" \
                          % (cv.ca.stats.percent_correct, cv.ca.calling_time)
                    continue

                # To report transfer error (and possibly some other metrics)
                confusion = ConfusionMatrix()
                times = []
                nf = []
                t0 = time.time()
                #TODO clf.ca.enable('nfeatures')
                partitioner = NFoldPartitioner()
                for nfold, ds in enumerate(partitioner.generate(dataset)):
                    (training_ds, validation_ds) = tuple(
                        Splitter(attr=partitioner.space).generate(ds))
                    clf.train(training_ds)
                    #TODO nf.append(clf.ca.nfeatures)
                    predictions = clf.predict(validation_ds.samples)
                    confusion.add(validation_ds.targets, predictions)
```

```
                    times.append([clf.ca.training_time, clf.ca.predicting_time])

                tfull = time.time() - t0
                times = np.mean(times, axis=0)
                #TODO nf = np.mean(nf)
                # print confusion
                #TODO print "%5.1f%%   %-4d\t %.2fs  %.2fs   %.2fs" % \
                print "%5.1f%%      -   \t %.2fs  %.2fs   %.2fs" % \
                      (confusion.percent_correct, times[0], times[1], tfull)
                #TODO      (confusion.percent_correct, nf, times[0], times[1], tfull)
            except LearnerError, e:
                print " skipped due to '%s'" % e

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`).

6.5.5 Analysis of the margin width in a soft-margin SVM

Width of the margin of soft-margin SVM (`mvpa2.clfs.svm.LinearCSVMC`) is not monotonic in its relation with SNR of the data. In case of not perfectly separable classes margin would first shrink with the increase of SNR, and then start to expand again after learning error becomes sufficiently small.

This brief examples provides a demonstration.

```
import mvpa2
import pylab as pl
import numpy as np
from mvpa2.misc.data_generators import normal_feature_dataset
from mvpa2.clfs.svm import LinearCSVMC
from mvpa2.generators.partition import NFoldPartitioner
from mvpa2.measures.base import CrossValidation
from mvpa2.mappers.zscore import zscore
```

Generate a binary dataset without any signal (snr=0).

```
mvpa2.seed(1);
ds_noise = normal_feature_dataset(perlabel=100, nlabels=2, nfeatures=2, snr=0,
                                  nonbogus_features=[0,1])

# signal levels
sigs = [0, 0.5, 1.0, 1.5, 2.0, 2.5, 3.0]
```

To mimic behavior of hard-margin SVM whenever classes become separable, which is easier to comprehend, we are intentionally setting very high C value.

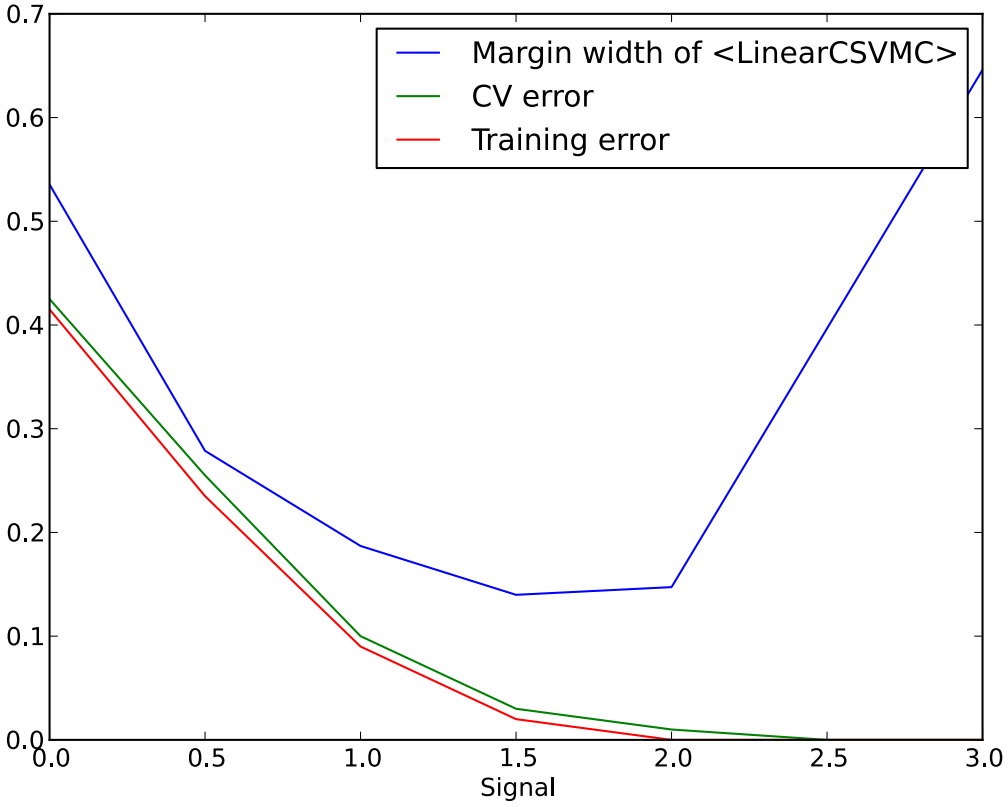
```
clf = LinearCSVMC(C=1000, enable_ca=['training_stats'])
cve = CrossValidation(clf, NFoldPartitioner(), enable_ca='stats')
sana = clf.get_sensitivity_analyzer(postproc=None)

rs = []
errors, training_errors = [], []

for sig in sigs:
    ds = ds_noise.copy()
    # introduce signal into the first feature
    ds.samples[ds.T == 'L1', 0] += sig

    error = np.mean(cve(ds))
```

```
    sa = sana(ds)
    training_error = 1-clf.ca.training_stats.stats['ACC']

    errors.append(error)
    training_errors.append(training_error)

    w = sa.samples[0]
    b = np.asscalar(sa.sa.biases)
    # width each way
    r = 1./np.linalg.norm(w)

    msg = "SIGNAL: %.2f training_error: %.2f error: %.2f |w|: %.2f r=%.2f" \
      %(sig, training_error, error, np.linalg.norm(w), r)
    print msg

    # Drawing current data and SVM hyperplane+margin
    xmin = np.min(ds[:,0], axis=0)
    xmax = np.max(ds[:,0], axis=0)
    x = np.linspace(xmin, xmax, 20)
    y  =    -(w[0] * x - b) /w[1]
    y1 = ( 1-(w[0] * x - b))/w[1]
    y2 = (-1-(w[0] * x - b))/w[1]

    pl.figure(figsize=(10,4))

    for t,c in zip(ds.UT, ['r', 'b']):
        ds_ = ds[ds.T == t]
        pl.scatter(ds_[:, 0], ds_[:, 1], c=c)
    # draw the hyperplane
    pl.plot(x, y)
    pl.plot(x, y1, '--')
    pl.plot(x, y2, '--')
    pl.title(msg)
    ca = pl.gca()
    ca.set_xlim((-2, 4))
    ca.set_ylim((-1.2, 1.2))
    pl.show()
    rs.append(r)
```

So what would be our dependence between signal level and errors/width of the margin?

```
pl.figure()
pl.plot(sigs, rs, label="Margin width of %s" % clf)
pl.plot(sigs, errors, label="CV error")
pl.plot(sigs, training_errors, label="Training error")
pl.xlabel("Signal")
pl.legend()
pl.show()
```

And this is how it looks like.



See also:

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

6.5.6 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 mvpa2.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 = np.random.randn(nsamp,nfeat)
samp1[:,:goodfeat] += offset

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

```
# create the pymvpa training dataset from the labeled features
patternsPos = dataset_wizard(samples=samp1[:ntrain,:], targets=1)
patternsNeg = dataset_wizard(samples=samp2[:ntrain,:], targets=0)
trainpat = vstack((patternsPos, patternsNeg))

# create patters for the testing dataset
patternsPos = dataset_wizard(samples=samp1[ntrain:,:], targets=1)
patternsNeg = dataset_wizard(samples=samp2[ntrain:,:], targets=0)
testpat = vstack((patternsPos, patternsNeg))

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

# enable saving of the estimates used for the prediction
smlr.ca.enable('estimates')

# 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.UT, targets=testpat.targets,
    predictions=pre)

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

# enable saving of the estimates used for the prediction
lsvm.ca.enable('estimates')

# 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.UT, targets=testpat.targets,
    predictions=pre)

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

keepInd = (np.abs(smlr.weights).mean(axis=1)!=0)
newtrainpat = trainpat[:, keepInd]
newtestpat = testpat[:, keepInd]

# 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.UT, targets=newtestpat.targets,
    predictions=pre)
```

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

See also:

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

6.5.7 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 mvpa2.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.sin_modulated(train_size, F)

# Generate dataset for testing:
test_size = 100
dataset_test = data_generators.sin_modulated(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 = np.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
pl.figure(figsize=(12, 12))
for i in range(rows*columns):
    pl.subplot(rows, columns, i+1)
    regression = True
    logml = True

    data_train = dataset.samples
    label_train = dataset.sa.targets
    data_test = dataset_test.samples
    label_test = dataset_test.sa.targets
```

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 = SquaredExponentialKernel(length_scale=length_scale,
                                    sigma_f=sigma_f)
     g = GPR(kse, sigma_noise=sigma_noise)
     if not regression:
         g = RegressionAsClassifier(g)
     print g

     if regression:
         g.ca.enable("predicted_variances")

     if logml:
         g.ca.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 = np.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:
         pl.title(r"$\sigma_f=%0.2f$, $length_s=%0.2f$, $\sigma_n=%0.2f$" \
                 % (sigma_f,length_scale,sigma_noise))
         pl.plot(data_train, label_train, "ro", label="train")
         pl.plot(data_test, prediction, "b-", label="prediction")
         pl.plot(data_test, label_test, "g+", label="test")
         if regression:
             pl.plot(data_test, prediction - np.sqrt(g.ca.predicted_variances),
                     "b--", label=None)
             pl.plot(data_test, prediction+np.sqrt(g.ca.predicted_variances),
                     "b--", label=None)
             pl.text(0.5, -0.8, "$RMSE=%.3f$" %(accuracy))
             pl.text(0.5, -0.95, "$LML=%.3f$" %(g.ca.log_marginal_likelihood))
         else:
             pl.text(0.5, -0.8, "$accuracy=%s" % accuracy)

         pl.legend(loc='lower right')

     print "LML:", g.ca.log_marginal_likelihood
```

See also:

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

6.5.8 Simple model selection: grid search for GPR

Run simple model selection (grid search over hyperparameters' space) of Gaussian Process Regression (GPR) on a simple 1D example.

```
__docformat__ = 'restructuredtext'

import numpy as np
from mvpa2.suite import *
import pylab as pl

# Generate train and test dataset:
train_size = 40
test_size = 100
F = 1
dataset = data_generators.sin_modulated(train_size, F)
dataset_test = data_generators.sin_modulated(test_size, F, flat=True)

print "Looking for better hyperparameters: grid search"

# definition of the search grid:
sigma_noise_steps = np.linspace(0.1, 0.5, num=20)
length_scale_steps = np.linspace(0.05, 0.6, num=20)

# Evaluation of log maringal likelohood spanning the hyperparameters' grid:
lml = np.zeros((len(sigma_noise_steps), len(length_scale_steps)))
lml_best = -np.inf
length_scale_best = 0.0
sigma_noise_best = 0.0
i = 0
for x in sigma_noise_steps:
    j = 0
    for y in length_scale_steps:
        kse = SquaredExponentialKernel(length_scale=y)
        g = GPR(kse, sigma_noise=x)
        g.ca.enable("log_marginal_likelihood")
        g.train(dataset)
        lml[i, j] = g.ca.log_marginal_likelihood
        if lml[i, j] > lml_best:
            lml_best = lml[i, j]
            length_scale_best = y
            sigma_noise_best = x
            # print x,y,lml_best
            pass
        j += 1
        pass
    i += 1
    pass

# Log marginal likelihood contour plot:
pl.figure()
X = np.repeat(sigma_noise_steps[:, np.newaxis], sigma_noise_steps.size,
              axis=1)
Y = np.repeat(length_scale_steps[np.newaxis, :], length_scale_steps.size,
              axis=0)
step = (lml.max()-lml.min())/30
pl.contour(X, Y, lml, np.arange(lml.min(), lml.max()+step, step),
               colors='k')
pl.plot([sigma_noise_best], [length_scale_best], "k+",
           markeredgewidth=2, markersize=8)
pl.xlabel("noise standard deviation")
pl.ylabel("characteristic length_scale")
pl.title("log marginal likelihood")
```

```
pl.axis("tight")
print "lml_best", lml_best
print "sigma_noise_best", sigma_noise_best
print "length_scale_best", length_scale_best
print "number of expected upcrossing on the unitary intervale:", \
      1.0/(2*np.pi*length_scale_best)


# TODO: reincarnate by providing a function within gpr.py
#
# Plot predicted values using best hyperparameters:
# pl.figure()
# compute_prediction(1.0, length_scale_best, sigma_noise_best, True, dataset,
#                    dataset_test.samples, dataset_test.targets, F, True)
```

See also:

The full source code of this example is included in the PyMVPA source distribution (`doc/examples/gpr_model_selection0.py`).

CHAPTER

SEVEN

FREQUENTLY ASKED QUESTIONS

7.1 General

7.1.1 I'm a Matlab user. How hard is learning Python and PyMVPA for me?

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"*.

7.1.2 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...

7.1.3 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 mvpa2.suite as mvpa2
```

makes everything directly accessible through the mvpa namespace, e.g. `mvpa2.datasets.base.Dataset` becomes `mvpa2.Dataset`. Really lazy people can even do:

```
>>> from mvpa2.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.

7.1.4 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.

7.1.5 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://github.com/PyMVPA/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`. Any recent version of `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.

7.1.6 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!

7.2 Data import, export and storage

7.2.1 What file formats are understood by PyMVPA?

Please see the data_formats section.

7.2.2 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.

7.3 Data preprocessing

7.3.1 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 `mvpa2.datasets.miscfx` module provides a convenience function `remove_invariant_features()` that strips such features from a dataset.

7.3.2 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 = normal_feature_dataset()
>>> print dataset
<Dataset: 100x4@float64, <sa: chunks,targets>>
```

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

```
>>> from mvpa2.mappers.fx import mean_group_sample
>>>
>>> m = mean_group_sample(['targets', 'chunks'])
>>> mapped_dataset = dataset.get_mapped(m)
>>> print mapped_dataset
<Dataset: 10x4@float64, <sa: chunks,targets>, <a: mapper>>
```

`mean_group_sample` creates an `FxMapper` that applies a function to every group of samples in each chunk individually and therefore yields one sample of each label per chunk.

7.4 Data analysis

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

All feature selection classifier use a built-in mapper to slice datasets. This mapper can be queried for selected features, or simply used to apply the same feature selection to other datasets.

```
>>> clf = FeatureSelectionClassifier(
...           kNN(k=5),
...           SensitivityBasedFeatureSelection(
...               SMLRWeights(SMLR(lm=1.0), postproc=maxofabs_sample()),
...               FixedNElementTailSelector(1, tail='upper', mode='select')))
>>> clf.train(dataset)
>>> len(clf.mapper.slicearg)
1
>>> final_dataset = clf.mapper.forward(dataset)
>>> print final_dataset
<Dataset: 100x1@float64, <sa: chunks,targets>>
```

In the above code snippet a kNN classifier is defined, that performs a feature selection step prior training. Features are selected according to the maximum 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.

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

In various parts of PyMVPA it is possible to extract information from inside loops via callbacks. To extract sensitivities from inside a cross-validation analysis, without unnecessary retraining of the classifier one only needs to write a corresponding callback function. here is a sketch:

```
>>> sensitivities = []
>>> def store_me(data, node, result):
...     sens = node.measure.get_sensitivity_analyzer(force_train=False)(data)
...     sensitivities.append(sens)
>>>
>>> cv = CrossValidation(SMLR(), OddEvenPartitioner(), callback=store_me)
>>> merror = cv(dataset)
>>> len(sensitivities)
2
>>> sensitivities[0].shape == (len(dataset.uniquetargets), dataset.nfeatures)
True
```

First we set up a container (a list) to store the sensitivies for a cross-validation folds. next is the callback: It takes three arguments, as described in the documentation of `RepeatedMeasure`. The second argument is the node that is evaluated inside the loop. For a cross-validation this is a `TransferMeasure` that exposes its internal classifier via the `measure` property. The rest is straightforward. We contruct a sensitivity analyzer and pass the input dataset. Finally, we store the returned sensitivities.

7.4.3 Can PyMVPA deal with literal class labels?

Yes. For all external machine learning libraries that do not support literal labels, PyMVPA will transparently convert them to numerical ones, and also revert this transformation for all output values.

CHAPTER

EIGHT

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 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.

Classifier
: A model that maps an arbitrary feature space into a discrete set of labels.

Meta-classifier
: An internal to PyMVPA term to describe a classifier which is usually a proxy to the main classifier which it wraps to provide additional data preprocessing (e.g. feature selection) before actually training and/or testing of the wrapped classifier.

Cross-validation
: A technique to assess the *generalization* of the constructed model by the analysis of accuracy of the model predictions on presumably independent dataset.

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.

Conditional Attribute
: An attribute of a *learner* which might be enabled or disabled, grouped within `.ca` attributes collection. If enabled, it might cause additional computation and memory consumption, so the "heaviest" conditional attributes are disabled by default.

Confusion Matrix
: Visualization of the *generalization* performance of a *classifier*. Each row of the matrix represents the instances in a predicted class, while each column represents the *sample*s in an actual (target) class. Each cell provides a count of how many *sample*s of the target class were (mis)classifier into the corresponding class. In PyMVPA instances of `ConfusionMatrix` class provide not only confusion matrix itself but a bulk of additional statistics.

Dataset
: In PyMVPA a dataset is the combination of samples, and their *Dataset attribute*s.

Dataset attribute
: An arbitrary auxiliary information that is stored in a dataset.

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*.

Exemplar
: Another term for *sample*.

Feature
: A variable that represents a dimension in a *dataset*. This might be the output of a single sensor, such as a voxel, or a refined measure reflecting specific aspect of data, such as a specific spectral component.

Feature attribute
: Analogous to a *sample attribute*, this is a per-feature vector of auxiliary information that is stored in a dataset.

Feature Selection
: A technique that targets detection of features relevant to a given problem, so that their selection improves generalization of the constructed model.

fMRI
: This acronym stands for *functional magnetic resonance imaging*.

Generalization
: An ability of a model to perform reliably well on any novel data in the given domain.

Label
: A label is a special case of a *target* for specifying discrete categories of *sample*s in a classification analyses.

Learner
: A model that upon training given some data (*sample*s and may be *target*s) develops an ability to map an arbitrary *feature* space of *sample*s into another space. If *target*s were provided, such learner is called *supervised* and tries to achieve mapping into the space of *target*s. If the target space defined by a set of discrete set of labels, such learner is called a *classifier*.

Machine Learning
: A field of Computer Science that aims at constructing methods, such as classifiers, to integrate available knowledge extracted from existing data.

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*.

Neural Data Modality
: A reflection of neural activity collected using some available instrumental method (e.g., EEG, *fMRI*).

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 times by simply calling them with new data.

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

Sample attribute
: A per-sample vector of auxiliary information that is stored in a dataset. This could, for example, be a vector identifying specific *chunk*s of samples.

Sensitivity
: A sensitivity is a score assigned to each *feature* with respect to its impact on the performance of the learner. So, for a classifier, sensitivity of a feature might describe its influence on *generalization* performance of the classifier. In case of linear classifiers, it could simply be coefficients of separating hyperplane given by *weight vector*. There exist additional scores which are similar to sensitivities 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 the result of significance testing is assigned.

Statistical Learning
: A field of science related to *machine learning* which aims at exploiting statistical properties of data to construct robust models, and to assess their convergence and *generalization* performances.

Supervised
: Is a *learner* which obtains both *sample*s data and *target*s within a *training dataset*.

Target
: A target associates each *sample* in the *dataset* with a certain category, experimental condition or, in case of a regression problem, with some metric variable. In case of supervised learning algorithm targets define the model to be trained, and provide the "ground truth" for assessing the model's *generalization* performance.

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

Training Dataset
: *Dataset* which is used for training of the *learner*.

Testing Dataset
: *Dataset* which is used to assess the *generalization* of the *learner*.

Weight Vector
: See *Sensitivity*.

CHAPTER

NINE

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.

Adluru, N., Hanlon, B. M., Lutz, A., Lainhart, J. E., Alexander, A. L. & Davidson, R. J. (2013). Penalized Likelihood Phenotyping: Unifying Voxelwise Analyses and Multi-Voxel Pattern Analyses in Neuroimaging. *Neuroinformatics*, 1-21.
DOI: http://dx.doi.org/10.1007/s12021-012-9175-9

Albanese, D., Visintainer, R., Merler, S., Riccadonna, S., Jurman, G. & Furlanello, C. (2012). mlpy: machine learning Python. *arXiv preprint arXiv:1202.6548*.

Andersson, P., Ramsey, N. F., Viergever, M. A. & Pluim, J. P. (2013). 7T fMRI reveals feasibility of covert visual attention-based brain–computer interfacing with signals obtained solely from cortical grey matter accessible by subdural surface electrodes. *Clinical neurophysiology*, *124*, 2191-2197.
DOI: http://dx.doi.org/10.1016/j.clinph.2013.05.009

Bandettini, P. A. (2009). Seven topics in functional magnetic resonance imaging. *Journal of Integrative Neuroscience*, *8*, 371–403.
URL: http://www.ncbi.nlm.nih.gov/pubmed/19938211

Baumgartner, F., Hanke, M., Geringswald, F., Zinke, W., Speck, O. & Pollmann, S. (2013). Evidence for feature binding in the superior parietal lobule. *NeuroImage*, *68*, 173-180.
DOI: http://dx.doi.org/10.1016/j.neuroimage.2012.12.002

Carlin, J. D., Calder, A. J., Kriegeskorte, N., Nili, H. & Rowe, J. B. (2011). A head view-invariant representation of gaze direction in anterior superior temporal sulcus. *Curr Biol*, *21*, 1817–21.
DOI: http://dx.doi.org/10.1016/j.cub.2011.09.025

Carlin, J. D., Rowe, J. B., Kriegeskorte, N., Thompson, R. & Calder, A. J. (2011). Direction-Sensitive Codes for Observed Head Turns in Human Superior Temporal Sulcus. *Cerebral Cortex*, **, .
Keywords: `pymvpa`, `fMRI`, `searchlight`

DOI: http://dx.doi.org/10.1093/cercor/bhr061

URL: http://cercor.oxfordjournals.org/content/early/2011/06/27/cercor.bhr061.short

Carter, R. M., Bowling, D. L., Reeck, C. & Huettel, S. A. (2012). A distinct role of the temporal-parietal junction in predicting socially guided decisions. *Science*, *337*, 109-111.
DOI: http://dx.doi.org/10.1126/science.1219681

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`, `feature selection stability`

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

URL: http://eutils.ncbi.nlm.nih.gov/entrez/eutils/elink.fcgi?cmd=prlinks&dbfrom=pubmed&retmode=ref&id=16565951

Clithero, J. A., Smith, D. V., Carter, R. M. & Huettel, S. A. (2010). Within- and cross-participant classifiers reveal different neural coding of information. *NeuroImage*.
DOI: http://dx.doi.org/10.1016/j.neuroimage.2010.03.057

URL: http://www.ncbi.nlm.nih.gov/pubmed/20347995

Cohen, J. (1994). The earth is round (p< 0.05). *American Psychologist*, *49*, 997–1003.
Classical critic of null hypothesis significance testing

Keywords: `hypothesis testing`

URL: http://www.citeulike.org/user/mdreid/article/2643653

Cohen, J. R., Asarnow, R. F., Sabb, F. W., Bilder, R. M., Bookheimer, S. Y., Knowlton, B. J. & Poldrack, R. A. (2010). Decoding developmental differences and individual variability in response inhibition through predictive analyses across individuals. *Frontiers in Human Neuroscience*, *4:47*.
DOI: http://dx.doi.org/10.3389/fnhum.2010.00047

URL: http://www.ncbi.nlm.nih.gov/pubmed/20661296

Cole, M. W., Etzel, J. A., Zacks, J. M., Schneider, W. & Braver, T. S. (2011). Rapid transfer of abstract rules to novel contexts in human lateral prefrontal cortex. *Frontiers in Human Neuroscience*, *5*.
DOI: http://dx.doi.org/10.3389/fnhum.2011.00142

Connolly, A. C., Guntupalli, J. S., Gors, J., Hanke, M., Halchenko, Y. O., Wu, Y., Abdi, H. & Haxby, J. V. (2012). The Representation of Biological Classes in the Human Brain. *Journal of Neuroscience*, *32*, 2608-2618.
DOI: http://dx.doi.org/10.1523/JNEUROSCI.5547-11.2012

URL: http://www.jneurosci.org/content/32/8/2608#aff-4

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

Duff, E. P., Trachtenberg, A. J., CE, C. E. M., Howard, M. A., Wilson, F., Smith, S. M. & Woolrich, M. W. (2011). Task-driven ICA feature generation for accurate and interpretable prediction using fMRI. *NeuroImage*, *60*, 189-203.
URL: http://www.ncbi.nlm.nih.gov/pubmed/22227050

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

Ekman, M., Derrfuss, J., Tittgemeyer, M. & Fiebach, C. J. (2012). Predicting errors from reconfiguration patterns in human brain networks. *Proceedings of the National Academy of Sciences*, *109*, 16714-16719.
DOI: http://dx.doi.org/10.1073/pnas.1207523109

Farrell, D., Webb, H., Johnston, M. A., Poulsen, T. A., O'Meara, F., Christensen, L. L., Beier, L., Borchert, T. V. & Nielsen, J. E. (2012). Toward Fast Determination of Protein Stability Maps: Experimental and Theoretical Analysis of Mutants of a Nocardiopsis prasina Serine Protease. *Biochemistry*, *51*, 5339-5347.
DOI: http://dx.doi.org/10.1021/bi201926f

Fisher, R. A. (1925). Statistical methods for research workers. Oliver and Boyd: Edinburgh.
One of the 20th century's most influential books on statistical methods, which coined the term 'Test of significance'.

Keywords: `statistics`, `hypothesis testing`

URL: http://psychclassics.yorku.ca/Fisher/Methods/

Garcia, S. & Fourcaud-Trocmé, N. (2009). OpenElectrophy: An Electrophysiological Data- and Analysis-Sharing Framework. *Front Neuroinformatics*, *3*, 14.
DOI: http://dx.doi.org/10.3389/neuro.11.014.2009

URL: http://www.ncbi.nlm.nih.gov/pubmed/19521545

Gilliam, T., Wilson, R. C. & Clark, J. A. (2010). Scribe Identification in Medieval English Manuscripts. Proceedings of the International Conference on Pattern Recognition.
URL: ftp://ftp.computer.org/press/outgoing/proceedings/juan/icpr10b/data/4109b880.pdf

Gorlin, S., Meng, M., Sharma, J., Sugihara, H., Sur, M. & Sinha, P. (2012). Imaging prior information in the brain. *Proceedings of the National Academy of Sciences*, *109*, 7935-7940.
DOI: http://dx.doi.org/10.1073/pnas.1111224109

URL: http://www.pnas.org/content/109/20/7935.abstract

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., Baumgartner, F. J., Ibe, P., Kaule, F. R., Pollmann, S., Speck, O., Zinke, W. & Stadler, J. (in press). A high-resolution 7-Tesla fMRI dataset from complex natural stimulation with an audio movie. *Scientific Data*.
URL: http://www.studyforrest.org

Hanke, M., Halchenko, Y. O., Haxby, J. V. & Pollmann, S. (2010). Statistical learning analysis in neuroscience: aiming for transparency. *Frontiers in Neuroscience*, *4*, 38–43.
Focused review article emphasizing the role of transparency to facilitate adoption and evaluation of statistical learning techniques in neuroimaging research.

DOI: http://dx.doi.org/10.3389/neuro.01.007.2010

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`, `feature selection`, `recursive feature elimination`, `RFE`

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

Hanson, S. J. & Schmidt, A. (2011). High-resolution imaging of the fusiform face area (FFA) using multivariate non-linear classifiers shows diagnosticity for non-face categories. *NeuroImage*, *54*, 1715-1734.
DOI: http://dx.doi.org/10.1016/j.neuroimage.2010.08.02

Hanson, S. J., Matsuka, T. & Haxby, J. V. (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

Hassabis, D., Spreng, R. N., Rusu, A. A., Robbins, C. A., Mar, R. A. & Schacter, D. L. (2013). Imagine all the people: How the brain creates and uses personality models to predict behavior. *Cerebral Cortex*.
DOI: http://dx.doi.org/10.1093/cercor/bht042

Hastie, T., Tibshirani, R. & Friedman, J. H. (2009). The Elements of Statistical Learning: Data Mining, Inference, and Prediction. Springer: New York.
Excellent summary of virtually all techniques relevant to the field. A free PDF version of this book is available from the authors' website at http://www-stat.stanford.edu/%7Etibs/ElemStatLearn/

DOI: http://dx.doi.org/10.1007/b94608

URL: http://www-stat.stanford.edu/%7Etibs/ElemStatLearn/

Haxby, J. V., Gobbini, M. I., Furey, M. L., Ishai, A., Schouten, J. L. & 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

Haxby, J. V., Guntupalli, J. S., Connolly, A. C., Halchenko, Y. O., Conroy, B. R., Gobbini, M. I., Hanke, M. & Ramadge, P. J. (2011). A Common, High-Dimensional Model of the Representational Space in Human Ventral Temporal Cortex. *Neuron*, *72*, 404–416.
DOI: http://dx.doi.org/10.1016/j.neuron.2011.08.026

URL: http://www.cell.com/neuron/abstract/S0896-6273%2811%2900781-1

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

Helfinstein, S. M., Schonberg, T., Congdon, E., Karlsgodt, K. H., Mumford, J. A., Sabb, F. W., Cannon, T. D., London, E. D., Bilder, R. M. & Poldrack, R. A. (2014). Predicting risky choices from brain activity patterns. *Proceedings of the National Academy of Sciences*, *111*, 2470-2475.
DOI: http://dx.doi.org/10.1073/pnas.1321728111

URL: http://www.pnas.org/content/111/7/2470.abstract

Hiroyuki, A., Brian, M., Li, N., Yumiko, S. & Massimo, P. (2012). Decoding Semantics across fMRI sessions with Different Stimulus Modalities: A practical MVPA Study. *Frontiers in Neuroinformatics*, *6*.
Keywords: `pymvpa`, `fmri`

DOI: http://dx.doi.org/10.3389/fninf.2012.00024

URL: http://www.frontiersin.org/Neuroinformatics/10.3389/fninf.2012.00024/full

Hollmann, M., Rieger, J. W., Baecke, S., Lützkendorf, R., Müller, C., Adolf, D. & Bernarding, J. (2011). Predicting decisions in human social interactions using real-time fMRI and pattern classification. *PloS one*, *6*, e25304.
DOI: http://dx.doi.org/10.1371/journal.pone.0025304

Ioannidis, J. P. A. (2005). Why most published research findings are false. *PLoS Med*, *2*, e124.
Simulation study speculating that it is more likely for a research claim to be false than true. Along the way the paper highlights aspects to keep in mind while assessing the 'scientific significance' of any given study, such as, viability, reproducibility, and results.

Keywords: `hypothesis testing`

DOI: http://dx.doi.org/10.1371/journal.pmed.0020124

Jain, A. & Kemp, C. C. (2012). Improving robot manipulation with data-driven object-centric models of everyday forces. *Autonomous Robots*, 1-17.
DOI: http://dx.doi.org/10.1007/s10514-013-9344-1

URL: http://www.hrl.gatech.edu/pdf/improve_everyday_forces.pdf

Jimura, K. & Poldrack, R. (2011). Analyses of regional-average activation and multivoxel pattern information tell complementary stories. *Neuropsychologia*.
DOI: http://dx.doi.org/10.1016/j.neuropsychologia.2011.11.007

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

Jäkel, F., Schölkopf, B. & Wichmann, F. A. (2009). Does Cognitive Science Need Kernels?. *Trends in Cognitive Sciences*, *13*, 381–388.
A summary of the relationship of machine learning and cognitive science. Moreover it also points out the role of kernel-based methods in this context.

Keywords: `kernel methods`, `similarity`

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

URL: http://www.sciencedirect.com/science/article/B6VH9-4X4R9BC-1/2/e2e90008d0a8887878c72777462335fd

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

Kaplan, J. T. & Meyer, K. (2012). Multivariate pattern analysis reveals common neural patterns across individuals during touch observation. *Neuroimage*, *60*, 204-212.
DOI: http://dx.doi.org/10.1016/j.neuroimage.2011.12.059

Kaunitz, L. N., Kamienkowski, J. E., Olivetti, E., Murphy, B., Avesani, P. & Melcher, D. P. (2011). Intercepting the first pass: rapid categorization is suppressed for unseen stimuli. *Frontiers in Perception Science*, *2*, 198.
Keywords: `pymvpa`, `eeg`

DOI: http://dx.doi.org/10.3389/fpsyg.2011.00198

URL: http://www.frontiersin.org/perception_science/10.3389/fpsyg.2011.00198/full

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.

Kohler, P. J., Fogelson, S. V., Reavis, E. A., Meng, M., Guntupalli, J. S., Hanke, M., Halchenko, Y. O., Connolly, A. C., Haxby, J. V. & Tse, P. U. (2013). Pattern classification precedes region-average hemodynamic response in early visual cortex. *NeuroImage*, *78*, 249-260.
DOI: http://dx.doi.org/10.1016/j.neuroimage.2013.04.019

Kriegeskorte, N., Goebel, R. & Bandettini, P. A. (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. A. (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

URL: http://eutils.ncbi.nlm.nih.gov/entrez/eutils/elink.fcgi?cmd=prlinks&dbfrom=pubmed&retmode=ref&id=15943426

Kubilius, J., Wagemans, J. & Beeck, H. O. d. (2011). Emergence of perceptual gestalts in the human visual cortex: The case of the configural superiority effect. *Psychological Science*, *in press*.
Keywords: `pymvpa`, `fMRI`

DOI: http://dx.doi.org/10.1177/0956797611417000

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

Laconte, S. M. (2010). Decoding fMRI brain states in real-time. *NeuroImage*.
DOI: http://dx.doi.org/10.1016/j.neuroimage.2010.06.052

URL: http://www.ncbi.nlm.nih.gov/pubmed/20600972

Lecun, Y., Bottou, L., Bengio, Y. & Haffner, P. (1998). Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, *86*, 2278–2324.
Paper introducing Modified NIST (MNIST) dataset for performance comparisons of character recognition performance across a variety of classifiers.

Keywords: handwritten character recognition, multilayer neural networks, MNIST, statistical learning

DOI: http://dx.doi.org/10.1109/5.726791

Legge, D. & Badii, A. (2010). An Application of Pattern Matching for the Adjustment of Quality of Service Metrics. The International Conference on Emerging Network Intelligence.

Lescroart, M. D. & Biederman, I. (2013). Cortical representation of medial axis structure. *Cerebral Cortex*, *23*, 629-637.
DOI: http://dx.doi.org/10.1093/cercor/bhs046

Liang, M., Mouraux, A., Hu, L. & Iannetti, G. (2013). Primary sensory cortices contain distinguishable spatial patterns of activity for each sense. *Nature communications*, *4*.
DOI: http://dx.doi.org/10.1038/ncomms2979

Man, K., Kaplan, J. T., Damasio, A. & Meyer, K. (2012). Sight and sound converge to form modality-invariant representations in temporoparietal cortex. *The Journal of Neuroscience*, *32*, 16629-16636.
DOI: http://dx.doi.org/10.1523/JNEUROSCI.2342-12.2012

Manelis, A. & Reder, L. M. (2013). He Who Is Well Prepared Has Half Won The Battle: An fMRI Study of Task Preparation. *Cerebral Cortex*.
DOI: http://dx.doi.org/10.1093/cercor/bht262

URL: http://cercor.oxfordjournals.org/content/early/2013/10/02/cercor.bht262.abstract

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

DOI: http://dx.doi.org/10.1002/hbm.20992

Manelis, A., Reder, L. M. & Hanson, S. J. (2011). Dynamic Changes In The Medial Temporal Lobe During Incidental Learning Of Object–Location Associations. *Cerebral Cortex*.
Keywords: pymvpa, fMRI

DOI: http://dx.doi.org/10.1093/cercor/bhr151

Margulies, D. S., Böttger, J., Long, X., Lv, Y., Kelly, C., Schäfer, A., Goldhahn, D., Abbushi, A., Milham, M. P., Lohmann, G. & Villringer, A. (2010). Resting developments: a review of fMRI post-processing methodologies for spontaneous brain activity. *Magnetic Resonance Materials in Physics, Biology and Medicine*, *23*, 289–307.
DOI: http://dx.doi.org/10.1007/s10334-010-0228-5

URL: http://www.ncbi.nlm.nih.gov/pubmed/20972883

McNamee, D., Rangel, A. & O'Doherty, J. P. (2013). Category-dependent and category-independent goal-value codes in human ventromedial prefrontal cortex. *Nature neuroscience*, *16*, 479-485.
DOI: http://dx.doi.org/10.1038/nn.3337

Merrill, J., Sammler, D., Bangert, M., Goldhahn, D., Lohmann, G., Turner, R. & Friederici, A. D. (2012). Perception of words and pitch patterns in song and speech. *Frontiers in psychology*, *3*, 76.
DOI: http://dx.doi.org/10.3389/fpsyg.2012.000

Meyer, K. & Kaplan, J. T. (2011). Cross-Modal Multivariate Pattern Analysis. *Journal of visualized experiments: JoVE*.
DOI: http://dx.doi.org/10.3791/3307

Meyer, K., Kaplan, J. T., Essex, R., Damasio, H. & Damasio, A. (2011). Seeing Touch Is Correlated with Content-Specific Activity in Primary Somatosensory Cortex. *Cerebral Cortex*.
DOI: http://dx.doi.org/10.1093/cercor/bhq289

URL: http://www.ncbi.nlm.nih.gov/pubmed/21330469

Meyer, K., Kaplan, J. T., Essex, R., Webber, C., Damasio, H. & Damasio, A. (2010). Predicting visual stimuli based on activity in auditory cortices. *Nature Neuroscience*.
DOI: http://dx.doi.org/10.1038/nn.2533

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

Olivetti, E., Greiner, S. & Avesani, P. (2012). Induction in Neuroscience with Classification: Issues and Solutions. Machine Learning and Interpretation in Neuroimaging, 42-50.
DOI: http://dx.doi.org/10.1007/978-3-642-34713-9_6

Olivetti, E., Veeramachaneni, S., Greiner, S. & Avesani, P. (2010). Brain Connectivity Analysis by Reduction to Pair Classification. The 2nd IAPR International Workshop on Cognitive Information Processing. **Oosterhof, N. N., Wiestler, T., Downing, P. E. & Diedrichsen, J.** (2011). A comparison of volume-based and surface-based multi-voxel pattern analysis. *NeuroImage*, *56*, 593-600.

Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., Blondel, M., Prettenhofer, P., Weiss, R., Dubourg, V., Vanderplas, J., Passos, A., Cournapeau, D., Brucher, M., Perrot, M. & Duchesnay, E. (2011). Scikit-learn: Machine Learning in Python. *Journal of Machine Learning Research*, *12*, 2825-2830.
URL: http://dl.acm.org/citation.cfm?id=1953048.2078195

Pereira, F. & Botvinick, M. (2011). Information mapping with pattern classifiers: a comparative study. *Neuroimage*, *56*, 476-496.
DOI: http://dx.doi.org/10.1016/j.neuroimage.2010.05.026

Pereira, F., Mitchell, T. & Botvinick, M. (2009). Machine learning classifiers and fMRI: A tutorial overview. *NeuroImage*, *45*, 199–209.
DOI: http://dx.doi.org/10.1016/j.neuroimage.2008.11.007

URL: http://www.ncbi.nlm.nih.gov/pmc/articles/PMC2892746/

Pernet, C. R., Sajda, P. & Rousselet, G. A. (2011). Single-trial analyses: why bother?. *Front Psychol*, *2*, 322.
DOI: http://dx.doi.org/10.3389/fpsyg.2011.00322

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

Raizada, R. D. & Connolly, A. C. (2012). What makes different people's representations alike: neural similarity-space solves the problem of across-subject fMRI decoding. *Journal of Cognitive Neuroscience*, *24*, 868-877.
URL: http://raizadalab.org/publications.html

Rueschemeyer, S., Ekman, M., van Ackeren, M. & Kilner, J. (2014). Observing, Performing, and Understanding Actions: Revisiting the Role of Cortical Motor Areas in Processing of Action Words. *Journal of Cognitive Neuroscience*.
DOI: http://dx.doi.org/10.1162/jocn_a_00576

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

Scholkopf, B. & Smola, A. (2001). Learning with Kernels: Support Vector Machines, Regularization. MIT Press: Cambridge, MA.
Good coverage of kernel methods and associated statistical learning aspects (e.g. error bounds)

Keywords: `statistical learning`, `kernel methods`, `error estimation`

Schrouff, J., Rosa, M. J., Rondina, J., Marquand, A., Chu, C., Ashburner, J., Phillips, C., Richiardi, J. & Mourão-Miranda, J. (2013). PRoNTo: Pattern Recognition for Neuroimaging Toolbox. *Neuroinformatics*, 1-19.
DOI: http://dx.doi.org/10.1007/s12021-013-9178-1

Shackman, A. J., Salomons, T. V., Slagter, H. A., Fox, A. S., Winter, J. J. & Davidson, R. J. (2011). The integration of negative affect, pain and cognitive control in the cingulate cortex. *Nature Reviews Neuroscience*, *12*, 154–167.
DOI: http://dx.doi.org/10.1038/nrn2994

URL: http://www.ncbi.nlm.nih.gov/pubmed/21331082

Shiffrin, R. (2010). Perspectives on Modeling in Cognitive Science. *Topics in Cognitive Science*, *2*, 736–750.
DOI: http://dx.doi.org/10.1111/j.1756-8765.2010.01092.x

Smith, D. V., Clithero, J. A., Rorden, C. & Karnath, H. (2013). Decoding the anatomical network of spatial attention. *Proceedings of the National Academy of Sciences*, *110*, 1518-1523.
DOI: http://dx.doi.org/10.1073/pnas.1210126110

Sobhani, M., Fox, G. R., Kaplan, J. & Aziz-Zadeh, L. (2012). Interpersonal liking modulates motor-related neural regions. *PloS one*, *7*, e46809.
DOI: http://dx.doi.org/10.1371/journal.pone.0046809

Spacek, M. & Swindale, N. (2009). Python in Neuroscience. *The Neuromorphic Engineer*.
DOI: http://dx.doi.org/10.2417/1200907.1682

Stelzer, J., Chen, Y. & Turner, R. (2012). Statistical inference and multiple testing correction in classification-based multi-voxel pattern analysis (MVPA): Random permutations and cluster size control. *NeuroImage*, *65*, 69-82.
DOI: http://dx.doi.org/10.1016/j.neuroimage.2012.09.063

Strnad, L., Peelen, M. V., Bedny, M. & Caramazza, A. (2013). Multivoxel Pattern Analysis Reveals Auditory Motion Information in MT+ of Both Congenitally Blind and Sighted Individuals. *PloS one*, *8*, e63198.
DOI: http://dx.doi.org/10.1371/journal.pone.0063198

Sun, D., van Erp, T. G., Thompson, P. M., Bearden, C. E., Daley, M., Kushan, L., Hardt, M. E., Nuechterlein, K. H., Toga, A. W. & Cannon, T. D. (2009). Elucidating an MRI-Based Neuroanatomic Biomarker for Psychosis: Classification Analysis Using Probabilistic Brain Atlas and Machine Learning Algorithms. *Biological Psychiatry*, *66*, 1055–1060.
First published study employing PyMVPA for MRI-based analysis of Psychosis.

Keywords: `PyMVPA`, `psychosis`, `MRI`

DOI: http://dx.doi.org/10.1016/j.biopsych.2009.07.019

Trautmann, E., Ray, L. & Lever, J. (2009). Development of an autonomous robot for ground penetrating radar surveys of polar ice. The 2009 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS), 1685–1690.
Study using PyMVPA to perform immobilization detection to improve navigation reliability of an autonomous robot.

DOI: http://dx.doi.org/10.1109/IROS.2009.5354290

Van der Laan, L. N., De Ridder, D. T., Viergever, M. A. & Smeets, P. A. (2012). Appearance matters: neural correlates of food choice and packaging aesthetics. *PloS one*, *7*, e41738.
DOI: http://dx.doi.org/10.1371/journal.pone.0041738

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

Varma, S. & Simon, R. (2006). Bias in error estimation when using cross-validation for model selection. *BMC Bioinformatics*, *7*, 91.
Demonstration of overfitting and introducing the bias in the error estimation using cross-validation on entire dataset for performing model selection.

Keywords: `statistical learning`, `model selection`, `error estimation`, `hypothesis testing`

DOI: http://dx.doi.org/10.1186/1471-2105-7-91

URL: http://www.ncbi.nlm.nih.gov/pubmed/16504092

Vickery, T. J., Chun, M. M. & Lee, D. (2011). Ubiquity and Specificity of Reinforcement Signals throughout the Human Brain . *Neuron *, *72*, 166-177.
DOI: http://dx.doi.org/10.1016/j.neuron.2011.08.011

URL: http://www.sciencedirect.com/science/article/pii/S089662731100732X

Viswanathan, S., Cieslak, M. & Grafton, S. T. (2012). On the geometric structure of fMRI searchlight-based information maps. *arXiv preprint arXiv:1210.6317*.

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

Woolgar, A., Thompson, R., Bor, D. & Duncan, J. (2010). Multi-voxel coding of stimuli, rules, and responses in human frontoparietal cortex. *NeuroImage*.
DOI: http://dx.doi.org/10.1016/j.neuroimage.2010.04.035

URL: http://www.ncbi.nlm.nih.gov/pubmed/20406690

Wright, D. (2009). Ten Statisticians and Their Impacts for Psychologists. *Perspectives on Psychological Science*, *4*, 587–597.
Historical excurse into the life of 10 prominent statisticians of XXth century and their scientific contributions.

Keywords: `statistics`, `hypothesis testing`

DOI: http://dx.doi.org/10.1111/j.1745-6924.2009.01167.x

Xu, H., Lorbert, A., Ramadge, P. J., Guntupalli, J. S. & Haxby, J. V. (2012). Regularized hyperalignment of multi-set fMRI data. Proceedings of the 2012 IEEE Signal Processing Workshop.

Zou, H. & Hastie, T. (2005). Regularization and variable selection via the elastic net. *Journal of the Royal Statistical Society Series B*, *67*, 301–320.
Keywords: `feature selection`, `statistical learning`

URL: http://www-stat.stanford.edu/%7Ehastie/Papers/B67.2%20(2005)%20301-320%20Zou%20%26%20Hastie.pdf

CHAPTER

TEN

LICENSE

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

```
The MIT License

Copyright (c) 2006-2015 Michael Hanke
              2007-2015 Yaroslav Halchenko
              2012-2015 Nikolaas N. Oosterhof

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.
```

10.1 3rd Party Code

Some code distributed within PyMVPA was not developed by PyMVPA team, hence you should adhere to the copyright and license terms of respective authors if you are to use corresponding parts.

10.1.1 LIBSVM

```
Files: 3rd/libsvm/*

BSD-3 License

Copyright (c) 2000-2009 Chih-Chung Chang and Chih-Jen Lin.
```

10.1.2 NIPY

```
Files: mvpa2/support/nipy/_*py, mvpa2/tests/test_emp_null.py

BSD-3 License

Copyright (c) 2006-2010, NIPY Developers
```

10.1.3 PDFBook

```
Files: tools/pdfbook.c

GPL version 2 License

Copyright:
  Tigran Aivazian <tigran at aivazian.fsnet.co.uk>
  Jaap Eldering <eldering at a-eskwadraat.nl>
  Roman Buchert <roman.buchert at arcor.de>
  Pierre Francois <pf at romanliturgy.org>
```

CHAPTER

ELEVEN

DEVELOPMENT CHANGELOG

This changelog only lists rather macroscopic changes to PyMVPA. The full VCS changelog for 2.x series of PyMVPA is available here:

https://github.com/PyMVPA/PyMVPA/commits/master

Note:

You could find relevant information on 0.4.x series at http://v04.pymvpa.org .

In addition there is also a somewhat unconventional visual changelog.

'Closes' and 'Fixes' statement IDs refer to the Debian and Github bug tracking systems accordingly and can be queried by visiting the URLs:

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

https://github.com/PyMVPA/PyMVPA/issues/<fixed bug id>
```

11.1 Releases

- 2.4.0 (Mon, 11 May 2015)
  - New functionality
    - Support for CoSMoMVPA (http://cosmomvpa.org) in `cosmo` providing dataset input/output (`cosmo_dataset()` and `map2cosmo()`) and neighborhood input (`CosmoQueryEngine`). This allows for for running searchlights (`mvpa2.datasets.cosmo.CosmoSearchlight`) on data from CoSMoMVPA (fMRI and MEEG).
    - `remove_nonfinite_features()` removes features with non-finite values, i.e. NaNs or Infs, for any sample.
    - *`binomial_proportion_ci()`* for computing confidence intervals on proportions of Bernoulli trial outcomes.
    - New mapper for removing sample means from features.
    - New algorithm for statistical evaluation of clusters in accuracy maps of group-based searchlight classification analyses. This is essentially an improved implementation of Stelzer et al., NeuroImage, 2013.
    - New identity mapper. Does nothing, but goes were only mappers can go.
    - Simplified selection of samples/feature in a dataset. One can now specify sets of attribute values to define sample/feature subsets.
    - IO adaptor for OpenFMRI-formated datasets. Load arbitrary bits from such a dataset, or automatically build event-related dataset (optionally with NiPy-based HRF-modeling).

`tutorial_data_25mm` was converted to OpenFMRI layout and extended also with `1slice` flavor.

* New command line command to generate a motion plot for an OpenFMRI-formated dataset.
* New convenience functions for boxplots and outlier detection.
* Reincarnated (similar functionality was removed for 2.0 release) convenience methods (`match()` and `select()`) to ease selecting parts of a dataset

– Enhancements

* `ProductFlattenMapper` accepts explicit names of factors in the constructor.
* HollowSphere() can now, optionally, include the center feature.
* `fmri_dataset()` no longer stores original copy of the NIfTI file header – it converts it to `dict` representation to remain portable. Use `strip_nibabel()` to convert old datasets to new format if/when necessary.

– Fixes

* `Hyperalignment` with regularization (alpha != 1.0) was producing incorrect transformations because they were driven by offsets of the last subject. Fixed by not "auto_train"ing regularization projection.
* `plot_lightbox()` should take a slice index from the last dimension, not the leading one if no `slices` argument was provided.
* Improved Python3k compatibility in `state`, `tests`, and `stats` modules, and in libsvmlrc msvc building.
* Partial fix for compatibility with ancient scipy on SPARC using `cosmo`.

- 2.3.1 (Tue, 20 May 2014)

  Primarily a bugfix release pushed out to avoid mvpa2.suite meltdown if new scipy 1.4.0 is used.

  – API changes

  * Deprecation: `Parameter` now uses `constraints` argument of type `Constraint` instead of string `allowedtype`. `allowedtype` argument will be removed completely in the future 2.4 release.

  – New changes

  * `dummies` now provides utterly useful `RandomClassifier` and others for code testing which could also be used to verify absent double-dipping etc.

  – Enhancements

  * `FxMapper` now will provide consistent order of groups of items. It also got a new argument `order` with available value of 'occurrence' to that groups would get ordered by their occurance in the original dataset.

  – Fixes

  * `CorrStability` should be able to deal with other sample attributes (not only 'targets') and should divide by variance correctly to provide correlation coefficient as output.
  * robustify check scipy's rdist which should avoid crash upon import of mvpa2.suite because of stripped down scipy 1.4.0 API.
  * various typos in docstrings (we do welcome contributions ;)).

- 2.3.0 (Thu, 5 March 2014)

  – **Warning**: Due to a significant number of new features and some internal changes, loading of HDF5 files saved with previous versions might be impaired.

  – New functionality (>73 commits)

    * Multi-threaded, surface-based searchlight queryengine `disc_surface_queryengine()` supporting as output space either surfaces (`SurfaceVerticesQueryEngine`; recommended) or volumes (`SurfaceVoxelsQueryEngine`).
    * `FlatSurfacePlotter` supporting flattened cortical surfaces.
    * I/O support for AFNI NIML (dset, `afni_niml_dset`; annotation, `afni_niml_annot`), AFNI 1D datasets (`afni_suma_1d`), SUMA surface specification (`afni_suma_spec`), Freesurfer ASCII surfaces (`surf_fs_asc`), GIFTI surfaces (`surf_gifti` through nibabel). Input support for Caret binary surfaces (`surf_caret`), EEGlab ASCII (`mvpa2.datasets.eeglab.eeglab_dataset()`), AFNI NIML (ROI, `afni_niml_roi`).
    * Experimental support for saving AttrDataset files directly to NIML format using `niml`.
    * Freesurfer/AFNI/SUMA preprocessing wrapper script `pymvpa2-prep-afni-surf` for surface-based analyses.
    * Winner-take-all measures `winner`.
    * New command line interface that provides access to the most commonly used functionality (dataset creation, pre-processing, cross-vlaidation, searchlights and data export).
    * `StatsmodelsGLMMapper` and `NiPyGLMMapper`.
    * Spectral filtering mapper `IIRFilter`.
  - Enhancements (>130 commits)
    * `fmri_dataset()` load support for AFNI NIFTI volumes with time in the fifth dimension (data is automagically squeezed to 4D).
    * `map2nifti()` sets `cal_` fields to correspond to the range of the data.
    * Many parts of the tutorial were reworked, set of examples expanded, and also converted to IPython notebooks.
    * `eventrelated()` extended to output GLM regression fits as features from HRF models (relying on NiPy for GLM modeling).
    * Class parameters now support programatic validation of values, auto-generated documentation and improved error messages.
    * More informative progress bar for long running processes, such as searchlights.
    * Replace all (broken) implementations for similarity structure analyses with new measures for computing pairwise pattern distances, and their consistency, or similarity to a target structure (`rsa`).
    * New examples to show integration with scikit-learn implementations of classification, regression, and data transformation algorithm.
  - Fixes (>88 commits)
    * Makefile fetch-data retrieves the data correctly.
    * HDF5 backend fixes to deal with nested/recursive structures and higher tolerance in loading HDF5 from older version.
- 2.2.0 (Sun, Sep 16 2012)
  - New functionality (14 commits)
  - New HDF5-based storage backend for `Searchlight` the can significantly speed up serialization of large result dataset in parallelized computations.
  - New fast searchlight `M1NNSearchlight` (and helper `sphere_m1nnsearchlight()`) to run mean-1-nearest-neighbor searchlights.
  - New mappers for adding an axis to a dataset (`AddAxisMapper`), and for transposing a dataset (`TransposeMapper`).

  - Improved implementation of SciPy's *ttest_1samp()* with support for masked arrays and alternative hypotheses.
  - Individual tutorial chapters are now available for download as IPython notebooks. A `rst2ipynb` converter is available in `tools/`.
  - New `pymvpa2-tutorial` command line utility to start a PyMVPA tutorial session, either in a console IPython session, or using the IPython notebook server.
  - New wrapper functions for data generators/loaders in `sklearn.datasets`, available in `mvpa2.datasets.sources.sklearn_data`.
  - Enhancements (89 commits)
    * Initial Python 3 compatibility (spear-headed by Tiziano Zito).
    * Bayesian hypothesis testing with `BayesConfusionHypothesis` now supports literal hypotheses specification, custom hypotheses subsets, and computing of posterior probabilities.
    * Allow for accessing fitted distributions in `MCNullDist`.
    * Extensions and improvements to the tutorial chapter on statistical evaluation.
    * Expose distance function as a property `dfx` of `kNN`.
    * Extended `Sifter` with ability to discard unbalanced partitions.
    * `h5save()` now creates missing directories automatically by default.
    * Dedicated training for `Hyperalignment`, and new auto-train capability.
    * `BayesConfusionHypothesis` now computes optional posterior probabilities, and supports hypothesis definitions using literal labels.
  - API changes
    * All command line tools have been renamed to have a consistent 'pymvpa2-' prefix.
  - Fixes (77 commits)
    * HDF5 now properly stores object-type ndarray, where it the array shape was unintentionally modified on-load before (Fixes #84).
    * HDF5 can now reconstruct 'builtin' objects (Fixes #86).
    * Check value data type and convert to float when collecting performance statistics to avoid numerical problems.
    * Do not fail in `BayesConfusionHypothesis` when a dataset does not provide class labels.

- 2.1.0 (Fri, June 29 2012)
  - Fixes
    * *mask2slice()* failed to convert an array of `False` values into `slice(None, 0, None)` (Fixes #56).
    * A number of fixes to the HDF5 IO code that ignored parts of an object's state when custom `__reduce__()` implementations were used (Fixes #42), and had problems storing metaclass types (Fixes #78).
    * Proper single quotes in documentation code snippets within PDFs.
    * Memory leak (model pointer) in LIBSVM bindings.
  - Enhancements
    * All searchlight implementations can now optionally store the IDs of all features for each generated ROI (conditional attr. `roi_feature_ids`)
    * Add *scatter_neighborhoods()* to aid sparse sampling of spaces.

    * Add `ConfusionMatrixError` to compute confusion matrices with an error function interface (e.g. for `CrossValidation(errorfx=...)`). This class existed for a long time, but was hidden in the unit tests.
    * Add `Confusion` to compute confusion matrices with a Node interface (e.g. for `CrossValidation(postproc=...)`). This is useful if confusion matrices are necessary as an intermediate result and further processing with other nodes is desired.
  - New functionality
    * Add `BayesConfusionHypothesis` to perform Bayesian hypothesis testing of multi-class confusion statistics. This is useful to assess the likelihood of a particular (or all possible) grouping of classes being distinguishable.
    * Add `FxyMapper` to perform arbitrary computations involving two datasets.
    * Add `CombinedMapper` to run a dataset through a set of mappers and combine their outputs.
    * Add `UnivariateStatsModels` a wrapper for using models from the statsmodels package as a FeaturewiseMeasure.
    * Add `dCOV` and `dcorcoef()` to quantify independence of (multivariate) signals.
  - API changes
    * Deprecating `GLM` that is now implemented with UnivariateStatsModels. This deprecated GLM class no longer supports the `zstat` calculation, and none of its previous conditional attributes are available anymore.

- 2.0.1 (Tue, Mar 27 2012)

  Primarily a bugfix release

  - Fixes (21 BF commits)
    * HDF5 storage – handle loading of objects with bound `builtin_function_or_method`.
    * Use system-wide `autosummary/generate.py` for sphinx >= 1.1.2 (Closes: #658593).
    * `ConditionalAttribute` should not loose value in deepcopy when default off (Fixes #63).
    * Correct handling of scalar mean/std values for `ZScoreMapper`.
    * MRI data import via _img2data now works with unicode filenames (Fixes #60).
    * Should work with IPython >= 0.11 now (Fixes #59).
    * Various small fixes to improve tests and functionality.
    * Fix `SMLR` segfaults on Windows (Thanks cgohlke for the patch).
  - Enhancements (29 ENH, OPT, and NF commits)
    * `FxMapper` calls functions natively (instead of a slow row/column at a time) if they carry `axis` as the 2nd argument. Provides tremendous speed up for `mean_sample()` etc.
    * *`xrandom_unique_combinations()`* generator for random unique combinations.
    * *`dual_gaussian()`* made more robust by not handling negative values for the standard deviations.
    * *`dual_positive_gaussian()`*.
    * Expose "sensitivities" for `PLR`.
    * `NFoldPartitioner` float option for `cvtype` and intelligent behavior on 'random' limited by 'count' given a large number of folds.
    * Add few additional learners from `sklearn` to the warehouses: ExtraTrees, RandomForest, LassoLarsIC.
    * `__repr__` for :class:'Partitioner's.

        * Add new performance metric – F1 score – in the confusion matrix summary stats.
        * `CachedQueryEngine` does not rely now on id but on the exact value of the query parameters (converted to hashable types). Before it could behave incorrectly on rare occasions.
    - API changes
        * Deprecating `.splitattr` if favor of `.attr` in `mvpa2.generators.partition.Partitioner`
- 2.0.0 (Mon, Dec 19 2011)

  This release aggregates all the changes occurred between official releases in 0.4 series and various snapshot releases (in 0.5 and 0.6 series). To get better overview of high level changes see release notes for 0.5 and 0.6 as well as summaries of release candidates below
    - Fixes (23 BF commits)
        * Significance level in the right tail was fixed to include the value tested – otherwise resulted in optimistic bias (or absurdly high significance in improbable case if all estimates having the same value).
        * Compatible with the upcoming IPython 0.12 and renamed sklearn (Fixes #57).
        * Do not double-train `slave` classifiers while assessing sensitivities (Fixes #53).
    - Enhancements (30 ENH + 3 NF commits)
        * Resolving voting ties in kNN based on mean distance, and randomly in SMLR.
        * `kNN`'s `ca.estimates` now contains dictionaries with votes for each class.
        * Consistent zscoring in `Hyperalignment`.
- 2.0.0~rc5 (Wed, Oct 19 2011)
    - **Major**: to allow easy co-existence of stable PyMVPA 0.4.x, 0.6 development `mvpa` module was renamed into mod:`mvpa2`.
    - Fixes
        * Compatible with the new Shogun 1.x series.
        * Compatible with the new h5py 2.x series.
        * `mvpa-prep-fmri` – various compatibility fixes and smoke testing.
        * Deepcopying `SummaryStatistics` during __add__.
    - Enhancements
        * Tutorial uses `mvpa2.tutorial_suite` now.
        * Better suppression of R warnings when needed.
        * Internal attributes of many classes were exposed as properties.
        * More unification of `__repr__` for many classes.
- 0.6.0~rc4 (Wed, Jun 14 2011)
    - Fixes
        * Finished transition to `nibabel` conventions in `plot_lightbox`.
        * Addressed `matplotlib.hist` API change.
        * Various adjustments in the tests batteries (`nibabel` 1.1.0 compatibility, etc)
    - New functionality
        * Explicit new argument `flatten` to from_wizard – default behavior changed if mapper was provided as well
    - Enhancements

    * Elaborated __str__ and __repr__ for some Classifiers and Measures

- 0.6.0~rc3 (Thu, Apr 12 2011)
  - Fixes
    * Bugfixes regarding the interaction of FlattenMapper and BoxcarMapper that affected event-related analyses.
    * Splitter now handles attribute value None for splitting properly.
    * GNBSearchlight handling of roi_ids.
    * More robust detection of mod:scikits.learn and nipy externals.
  - New functionality
    * Added a Repeater node to yield a dataset multiple times and Sifter node to exclude some datasets. Consequently, the "nosplitting" mode of Splitter got removed at the same time.
    * tools/niils – little tool to list details (dimensionality, scaling, etc) of the files in nibabel-supported formats.
  - Enhancements
    * Numerous documentation fixes.
    * Various improvements and increased flexibility of null distribution estimation of Measures.
    * All attribute are now reported in sorted order when printing a dataset.
    * fmri_dataset now also stores the input image type.
    * Crossvalidation can now take a custom Splitter instance. Moreover, the default splitter of CrossValidation is more robust in terms of number and type of created splits for common usage patterns (i.e. together with partitioners).
    * CrossValidation takes any custom Node as errorfx argument.
    * ConfusionMatrix can now be used as an errorfx in Crossvalidation.
    * LOE(ACC): Linear Order Effect in ACC was added to ConfusionMatrix to detect trends in performances across splits.
    * A Nodes postproc is now accessible as a property.
    * RepeatedMeasure has a new 'concat_as' argument that allows results to be concatenated along the feature axis. The default behavior, stacking as multiple samples, is unchanged.
    * Searchlight now has the ability to mark the center/seed of an ROI in with a feature attribute in the generated datasets.
    * debug takes args parameter for delayed string comprehensions. It should reduce run-time impact of debug() calls in regular, non -O mode of Python operation.
    * String summaries and representations (provided by __str__ and __repr__) were made more exhaustive and more coherent. Additional properties to access initial constructor arguments were added to variety of classes.
  - Internal changes
    * New debug target STDOUT to allow attaching metrics (e.g. traceback, timestamps) to regular output printed to stdout
    * New set of decorators to help with unittests
      · @nodebug to disable specific debug targets for the duration of the test.
      · @reseed_rng to guarantee consistent random data given initial seeding.
      · @with_tempfile to provide a tempfile name which would get removed upon completion (test success or failure)

    * Dropping daily testing of `maint/0.5` branch – RIP.
    * `Collections`s were provided with adequate `(deep|)copy`. And `Dataset` was refactored to use `Collections` `copy` method.
    * `update-*` Makefile rules automatically should fast-forward corresponding `website-updates` branch
    * `MVPA_TESTS_VERBOSITY` controls also `numpy` warnings now.
    * `Dataset.__array__` provides original array instead of copy (unless dtype is provided)

  Also adapts changes from 0.4.6 and 0.4.7 (see corresponding changelogs).

- 0.6.0~rc2 (Thu, Mar 3 2011)
  - Various fixes in the mvpa.atlas module.
- 0.6.0~rc1 (Thu, Feb 24 2011)
  - Many, many, many
  - For an overview of the most drastic changes see constantly evolving release notes for 0.6
- 0.5.0 (sometime in March 2010)

  This is a special release, because it has never seen the general public. A summary of fundamental changes introduced in this development version can be seen in the release notes.

  Most notably, this version was to first to come with a comprehensive two-day workshop/tutorial.
- 0.4.7 (Tue, Mar 07 2011) (Total: 12 commits)

  A bugfix release
  - Fixed
    * Addressed the issue with input NIfTI files having `scl_` fields set: it could result in incorrect analyses and map2nifti-produced NIfTI files. Now input files account for scaling/offset if `scl_` fields direct to do so. Moreover upon map2nifti, those fields get reset.
    * `doc/examples/searchlight_minimal.py` - best error is the minimal one
  - Enhancements
    * `GNB` can now tolerate training datasets with a single label
    * `TreeClassifier` can have trailing nodes with no classifier assigned
- 0.4.6 (Tue, Feb 01 2011) (Total: 20 commits)

  A bugfix release
  - Fixed (few BF commits):
    * Compatibility with numpy 1.5.1 (histogram) and scipy 0.8.0 (workaround for a regression in legendre)
    * Compatibility with libsvm 3.0
    * `PLR` robustification
  - Enhancements
    * Enforce suppression of numpy warnings while running unittests. Also setting verbosity >= 3 enables all warnings (Python, NumPy, and PyMVPA)
    * `doc/examples/nested_cv.py` example (adopted from 0.5)
    * Introduced base class `LearnerError` for classifiers' exceptions (adopted from 0.5)
    * Adjusted example data to live upto nibabel's warranty of NIfTI standard-compliance
    * More robust operation of MC iterations – skip iterations where classifier experienced difficulties and raise an exception (e.g. due to degenerate data)

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

  A bugfix release

  - Fixed (13 BF commits):
    - Compatible with LIBSVM >= 2.91 (Closes: #583018)
    - No string exceptions raised (Python 2.6 compatibility)
    - Setting of shrinking parameter in `sg` interface
    - Deducing number of SVs for SVR (LIBSVM)
    - Correction of significance in the tails of non-parametric tests
  - Miscellaneous:
    - Development repository moved to http://github.com/PyMVPA/PyMVPA

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

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

  - New functionality (19 NF commits):
    - `GNB` implements Gaussian Naïve Bayes Classifier.
    - `read_fsl_design()` to read FSL FEAT design.fsf files (Contributed by Russell A. Poldrack).
    - `SequenceStats` to provide basic statistics on labels sequence (counter-balancing, autocorrelation).
    - New exceptions `DegenerateInputError` and `FailedToTrainError` to be thrown by classifiers primarily during training/testing.
    - Debug target `STATMC` to report on progress of Monte-Carlo sampling (during permutation testing).
  - Refactored (15 RF commits):
    - To get users prepared to 0.5 release, internally and in some examples/documentation, access to states and parameters is done via corresponding collections, not from the top level object (e.g. `clf.states.predictions` instead of soon-to-be-deprecated `clf.predictions`). That should lead also to improved performance.
    - Adopted copy.py from python2.6 (support Ellipsis as well).
  - Fixed (38 BF commits):
    - GLM output does not depend on the enabled states any more.
    - Variety of docstrings fixed and/or improved.
    - Do not derive NaN scaling for SVM's C whenever data is degenerate (lead to never finishing SVM training).
    - `sg`:
      - KRR is optional now – avoids crashing if KRR is not available.
      - tolerance to absent `set_precompute_matrix` in svmlight in recent shogun versions.
      - support for recent (present in 0.9.1) API change in exposing debug levels.
    - Python 2.4 compatibility issues: `kNN` and `IFS`

- 0.4.3 (Sat, 5 Sep 2009) (Total: 165 commits)
  - Online documentation editor is no longer available due to low demand – please submit changes via email.
  - Performance (Contributed by Valentin Haenel) (3 OPT commits):

    * Further optimized LIBSVM bindings.
    * Copy-if-sorted in `selectFeatures`.
  - New functionality (25 NF commits):
    * `ProcrusteanMapper` with orthogonal and oblique transformations.
    * Ability to generate simple reports using `reportlab`. See/run `examples/match_distribution.py` for example.
    * `TreeClassifier` – construct simple hierarchies of classifiers.
    * `wtf()` to report information about the system/PyMVPA to be included in the bug reports.
    * Parameter 'reverse' to swap training/testing splits in `Splitter` .
    * Example code for the analysis of event-related dataset using `ERNiftiDataset`.
    * `toEvents()` to create lists of `Event`.
    * `mvpa-prep-fmri` was extended with plotting of motion correction parameters.
    * `ColumnData` can be explicitly told either file contains a header.
    * In `XMLBasedAtlas` (e.g. `fsl` atlases) it is now possible to provide custom 'image_file' to get maps or indexes for the areas given an atlas's volume registered into subject space.
    * Updated included LIBSVM version to 2.89 and provided support for its "silencing".
  - Refactored (27 RF commits):
    * Dataset's `copy()` with deep=False allows for shallow copying the dataset.
    * `FeatureSelectionClassifier` s in `warehouse` not to reuse the same classifiers, but to use clones.
  - Fixed (70 BF commits):
    * `OneWayAnova`: previously degrees of freedom were not considered while computing F-scores.
    * Majority voting strategy in `kNN`: it was not working.
    * Various fixes to ensure cross-platform building (`numpy` header locations, etc).
    * Stability fixes in `ConfusionMatrix`.
    * `idsonboundaries()`: samples at the end of the sequence were not handled properly.
    * Proper "untraining" of `FeatureSelectionClassifier` s classifiers which use sensitivities: it could lead to various unpleasant side-effects if the same slave classifier was used simultaneously by multiple `MetaClassifiers` (like `TreeClassifier`).
  - Documentation (25 DOC commits): citations, spelling corrections, etc.
- 0.4.2 (Mon, 25 May 2009)
  - New correlation stability measure (`CorrStability`).
  - New elastic net classifier (`ENET`).
  - New GLM-Net regression/classifier (`GLMNET`).
  - New measure `CompoundOneWayAnova`.
  - New measure `DSMDatasetMeasure`.
  - New meta-measure `TScoredFeaturewiseMeasure`.
  - New basic `GLM` implementation.
  - New examples for Gaussian process regression.
  - New example showing a searchlight analysis employing a dissimilarity matrix based measure.
  - New `ZScoreMapper`.

  - New import helper for FSL design matrices (`FslGLMDesign`).
  - New implementation of a mapper using a self-organizing map (`SimpleSOMMapper`) and a corresponding example.
  - Matplotlib backend is now configurable via `MVPA_MATPLOTLIB_BACKEND`.
  - PyMVPA version is now 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 conjunction with `mvpa.test()`, this allow to easily run unittests from within Python.
  - `NiftiDataset` capable to handle files with less than 4 dimensions, which can, optionally, be provided as a list of filenames or `NiftiImage` objects. That makes it easy to load data from a sequence of files.
  - Changes (code refactorings) which *might impact* any user who imports from `suite`:
    * Pre-populated warehouses of classifiers and regressions are renamed from clfs and regrs into `clfswh` and `regrswh` respectively.
    * `Hamster` is not derived from `dict` any longer – just from a basic `object` class. API includes methods 'dump', '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 compatibility with MacOS.
  - Extended FAQ section of the manual.
  - Bugfix: double_gamma_hrf() 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.

m

m

L

M

N

O

P

R

S

T

U

V

W