PyMVPA Manual

Release 0.2.0

Michael Hanke, Yaroslav Halchenko, Per B. Sederberg

June 01, 2008

CONTENTS

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 or pattern recognition techniques. There is a wealth of high-quality text books about this field available. A very good example is: Pattern Recognition and Machine Learning by Christopher M. Bishop.

A good starting point to learn about the application of machine learning algorithms to (f)MRI data are two recent reviews by Norman et al. [1] and Haynes and Rees [2].

This manual also does not describe every bit and piece of the PyMVPA package. For more information, please have a look at the API documentation, which is a comprehensive and up-to-date description of the whole package.

More examples and usage patterns extending the ones described here can be taken from the examples shipped with the PyMVPA source distribution (*doc/examples/*) 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 to 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

[1] 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.

[2] Haynes, J.D. & Rees, G. (2007). Decoding mental states from brain activity in humans. Nature Reviews Neuroscience, 7, 523–534.

fully object-oriented (in contrast to the MVPA toolbox), but should be at least as easy to use and extensible as the MVPA toolbox.

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

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

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

1.3 Prerequisites

Like every other Python module PyMVPA requires at least a basic knowledge of the Python language. However, if one has no prior experience with Python one can benefit from the simplicity of the Python language and acquire this knowledge within a few days by studying some of the many tutorials available on the web.

As PyMVPA is about pattern recognition a basic understanding about machine learning principles is necessary to correctly apply methods with PyMVPA to ensure interpretability of the results.

1.3.1 Dependencies

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

Python 2.4 (or later) With some modifications PyMVPA could probably work with Python 2.3, but as it is quite old already and Python 2.4 is widely available there should be no need to do this.

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

1.3.2 Strong Recommendations

While most parts of PyMVPA will work without any additional software, some functionality makes use of additional software packages. It is strongly recommended to install these packages as well.

SciPy: linear algebra, standard distributions SciPy is mainly used by the statistical testing and the logistic regression classifier code. However, in the long run SciPy might be used a lot more and could become a required dependency of PyMVPA.

PyNIfTI: access to NIfTI files PyMVPA provides a convenient wrapper for datasets stored in the NIfTI format. If you don't need that, PyNIfTI is not necessary, but otherwise it makes it really easy to read from and write to NIfTI images.

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.

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

1.3.3 Suggestions

The following list of software is not required by PyMVPA, but it might make life a lot easier and leads 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.

FSL: preprocessing and analysis of (f)MRI data PyMVPA provides some simple bindings to FSL output and filetypes (e.g. EV files 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.

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.

1.4 Obtaining PyMVPA

1.4.1 Binary packages

The easiest way to obtain PyMVPA is to use pre-built binary packages. Currently the Debian/Ubuntu family is the only environment for which binary packages are available (see below). If you manage to build PyMVPA on Windows or OS X, we would be glad to hear from you.

Debian

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

Debian backports and inofficial Ubuntu packages

Backports for the current Debian stable release and binary packages for recent Ubuntu releases are available from a repository at the University of Magdeburg. Please read the package repository instructions to learn about how to obtain them.

1.4.2 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 you can easily build PyMVPA on your own. Any recent linux distribution should be capable of doing it. Additionally, we are aware of successful builds on Mac OSX. The first step is obtaining the sources. The source code tarballs of all PyMVPA releases are available from the PyMVPA project website. Alternatively, one can also download a tarball of the latest development snapshot (i.e. the current state of the *master* branch of the PyMVPA

source code repository). If you want to have access to both, the full PyMVPA history and the latest development code, you can use the PyMVPA Git repository, which is publicly available. To view the repository, please point your web browser to gitweb:

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

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

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

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

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.

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 and the development files of libsvm (headers and library) installed on your system. Depending on where you installed them, it might be necessary to specify the full path to them with the *–include-dirs*, *–library-dirs* and *–swig* options.

PyMVPA needs a patched libsvm version, as the original distribution generates a huge amount of debugging messages and therefore makes the console and PyMVPA output almost unusable. Debian (since lenny: 2.84.0-1) and Ubuntu (since gutsy) already include the patched version. For all other systems it is easy to build patched libsvm (see Building patched libsvm from Source).

The command to build all extentions including the libsvm wrapper is:

```
PYMVPA_LIBSVM=1 python setup.py build_ext --swig-opts="-c++ -noproxy"
```

The installation procedure is equivalent to the a build setup without libsvm.

Building patched libsvm from source

First get the patched sources from:

```
http://packages.debian.org/source/sid/libsvm
```

Download the *diff.gz* and the *orig.tar.gz* files offered at the bottom of the page. Once downloaded extract the *tar.gz* file and patch it. The following example refers to *libsvm* version 2.85.0, please adjust the filenames and versions if you use a later version:

```
tar xvzf libsvm_2.85.0.orig.tar.gz
cd libsvm-2.85
zcat ../libsvm_2.85.0-1.diff.gz | patch -p1
```

If *zcat* does not work for you (which might happen on Mac OSX), simply decompress the diff manually and do:

```
patch -p1 < ../libsvm_2.85.0-1.diff
```

instead to patch the sources. If this is done build the library and install it:

```
make libsvm.so.2.85.0
DESTDIR=/usr/local make install
```

Set *DESTDIR* to your prefered installation path. For those running Mac OSX, there is also a *Makefile.osx*.

Alternative build procedure

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

```
export PYTHONPATH=$PWD
```

However, please note that this procedure also always builds the libsvm extension and therefore also required the patched libsvm version to be available.

1.5 Installation

If there are no binary packages for your operating system or platform yet, you need to build from source. Please refer to Building from Source for more information.

Otherwise just install the binary packages as you would do with any other package. For example on Debian or Ubuntu simply do:

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

1.6 How to cite PyMVPA

The PyMVPA toolbox was first presented with a poster at annual meeting of the *German Society for Psychophysiology and its Application* in Magdeburg, 2008. This is currently the prefered way to cite PyMVPA. However, we submitted a paper introducing the toolbox, which should become replace the poster soon.

1.7 Credits

(needs some more words, for now just a list)

- NumPy, SciPy
- libsvm
- Shogun

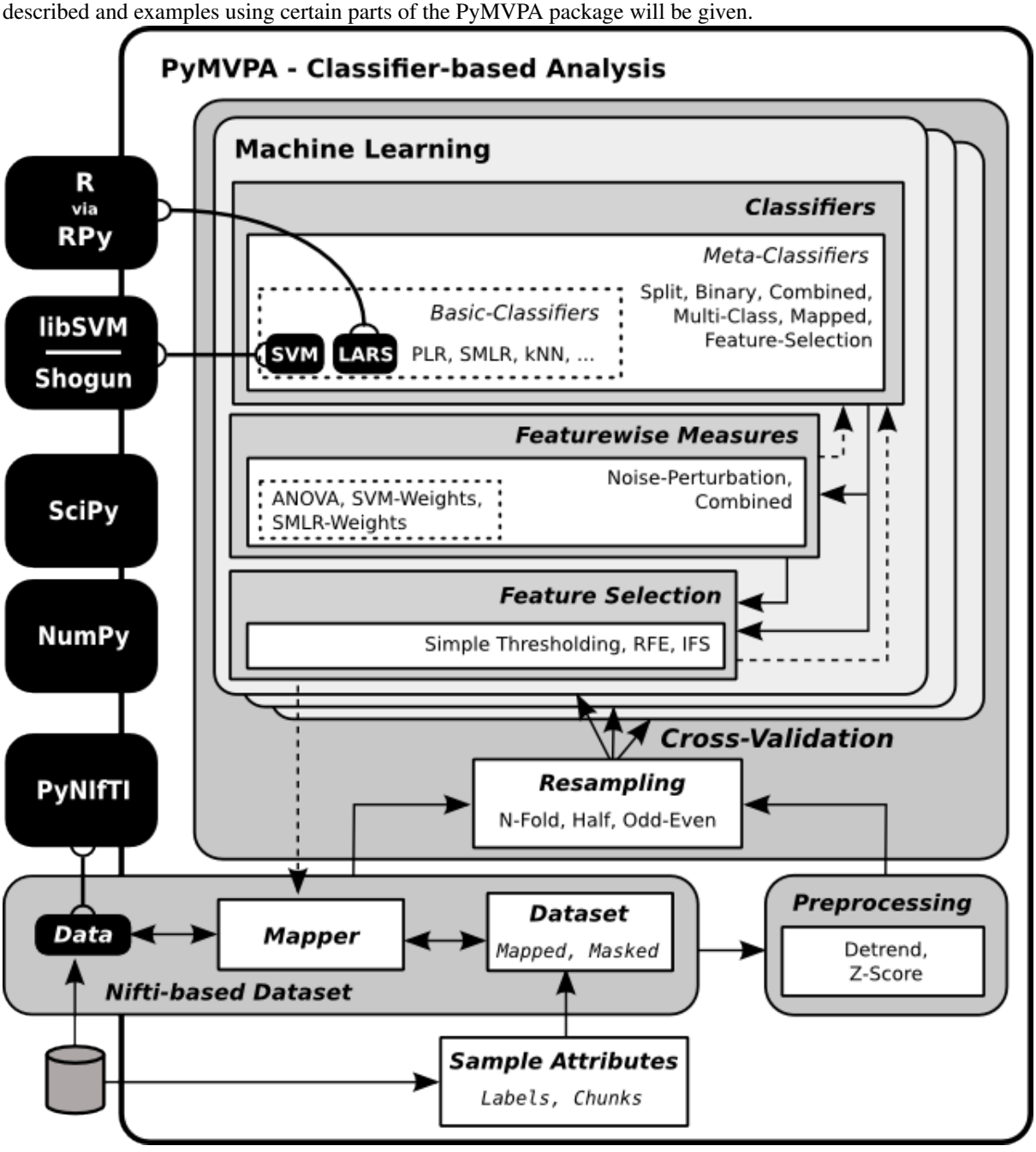
- IPython
- Debian (for hosting, environment, ...)
- FOSS community
- Credits to individual labs if they officially donate time ;-)

CHAPTER

TWO

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.



CHAPTER

THREE

Datasets

The foundation of PyMVPA's data handling is the Dataset class. Basically, this class stores data samples, sample attributes and dataset attributes. Sample attributes assign a value to each data sample and dataset attributes are additional information or functionality that applies to the whole dataset.

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

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

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

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

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

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

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

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

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

3.1 Data Mapping

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

where each data sample is typically a three-dimensional volume.

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

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

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

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

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

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

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

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

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

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

```
       [0, 0, 0, 2],
       [0, 3, 4, 0]])
```

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

3.2 Data Splitting

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

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

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

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

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

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

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

CHAPTER

FOUR

Classifiers

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

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

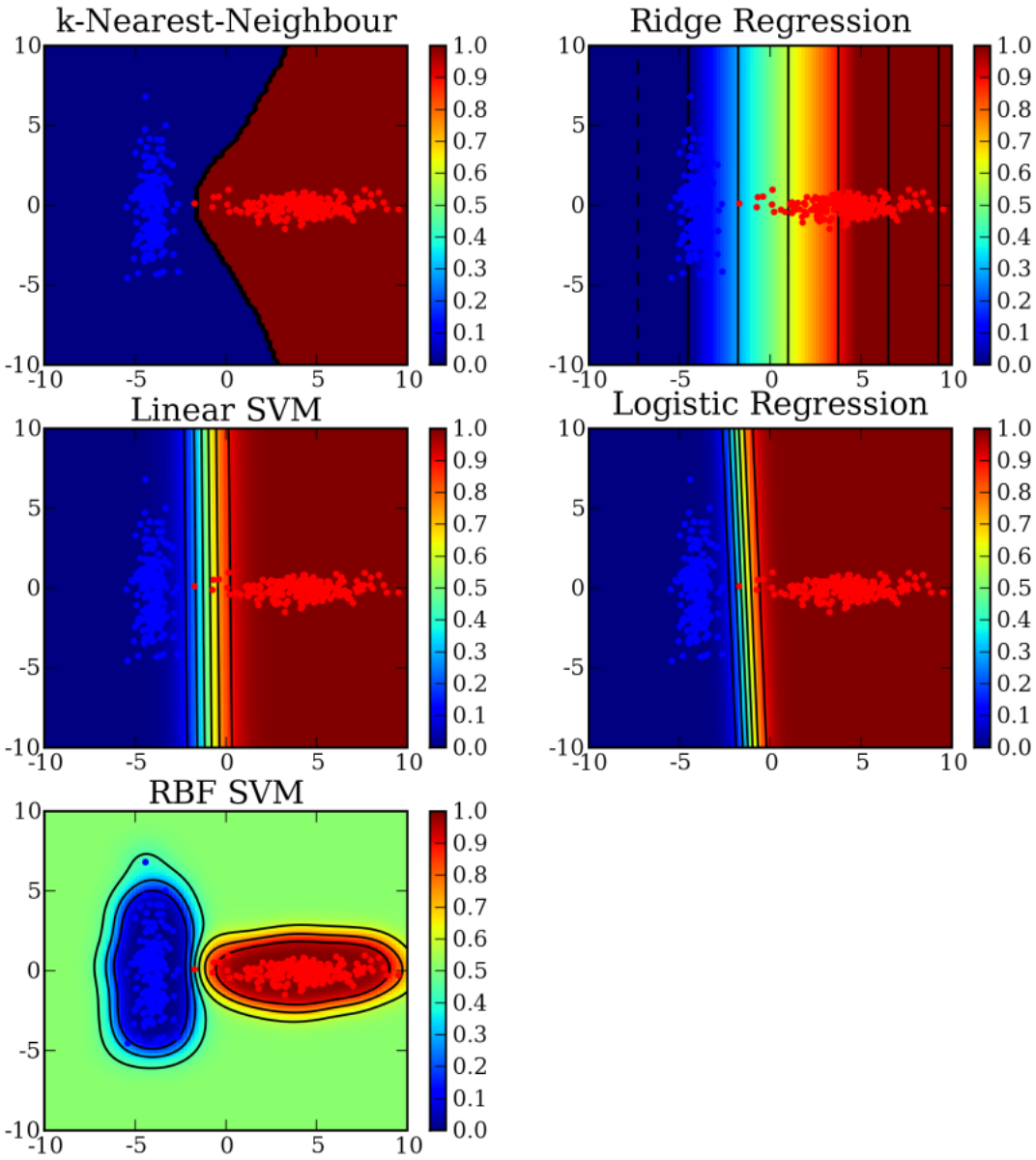
This examples demonstrates the typical daily life of a classifier.

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

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

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

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



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

4.1 Stateful objects

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

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

the classifier has not seen the data yet and, thus, does not know the labels. In other words, 'clf' is not yet in the state to know anything about the labels, hence the name *Stateful*. We will refer to instances of classes derived from *Stateful* as 'statefull'. Any state variable can be enabled or disabled on per instance basis at any time of the execution.

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

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

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

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

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

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

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

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

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

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

4.2 Error Calculation

The TransferError class provides a convenient way to determine the transfer error of a trained classifier on some validation dataset. A TransferError object is instanciated by passing a classifier object to the constructor. Optionally a custom error function can be specified (see *errorfx* argument).

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

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

```
>>> from mvpa.clfs.transerror import TransferError
>>> clf = kNN(k=10)
```

```
>>> terr = TransferError(clf)
>>> terr(validation, training )
0.0
```

4.2.1 Cross-validated Transfer Error

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

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

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

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

4.3 Boosted and Multi-class Classifiers

(to be written)

4.4 Gaussian Process Regression

(Wikipedia entry about gaussian process regression).

4.5 k-Nearest-Neighbour

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

4.6 Least Angle Regression

[1]

[1] Bradley Efron, Trevor Hastie, Iain Johnstone and Robert Tibshirani (2004). A new method for variable subset selection, with the lasso and "epsilon" forward stagewise methods as special cases. *Annals of Statistics, 32*, 407-499.

4.7 Penalized Logistic Regression

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

4.8 Ridge Regression

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

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

4.9 Sparse Multinomial Logistic Regression

Sparse Multinomial Logistic Regression [2] is a fast multi-class classifier that can easily with high-dimensional problems (research paper about SMLR). PyMVPA include two implementations: one in pure Python and a faster one that makes use of a C extension for the performance critical pieces of the code.

4.10 Support Vector Machines

Support vector machines [3] classifiers (and regressions) are popular since they can deal with very high dimensional problems (Wikipedia entry about SVM), while maintaining reasonable generalization performance.

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

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

4.11 Classifiers "Warehouse"

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

```
>>> from mvpa.clfs.warehouse import clfs
>>> print len(clfs['multiclass', '!svm'])
8
```

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

[2] Krishnapuram, B., Figueiredo, M., Carin, L., & Hartemink, A. (2005). Sparse Multinomial Logistic Regression: Fast Algorithms and Generalization Bounds. *IEEE Transactions on Pattern Analysis and Machine Intelligence (PAMI)*, 957–968.

[3] Vapnik, V. (1995). *The Nature of Statistical Learning Theory*. Springer, New York.

CHAPTER

FIVE

Measures

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

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

5.1 Sensitivity Measures

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

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

By convention higher sensitivity values indicate more interesting features.

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

5.1.1 Basic Sensitivity (and related Measures)

ANOVA

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

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

Linear SVM Weights

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

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

Please note, that this class *cannot* extract reasonable weights from non-linear SVMs (e.g. with RBF kernels).

Noise Perturbation

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

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

5.1.2 Meta Sensitivity Measures

Meta Sensitivity Measures are FeaturewiseDatasetMeasures that internally use one of the *Basic Sensitivity Measures* to compute their sensitivity scores.

Splitting Measures

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

CHAPTER

SIX

Feature Selection

6.1 Recursive Feature Elimination

RFE

(to be written)

6.2 Incremental Feature Search

IFS

(to be written)

CHAPTER

SEVEN

Analysis Scenarios

7.1 Searchlight

The term Searchlight refers to an algorithm that runs a scalar *DatasetMeasure* on all possible spheres of a certain size within a dataset. The measure typically computed is a cross-validated transfer error (see *CrossValidatedTransferError*). The idea to use a searchlight as a sensitivity analyzer stems from a paper by Kriegeskorte and colleagues [1].

A searchlight analysis is can be easily performed. The following code snippet shows a draft of a complete analysis.

```
>>> from mvpa.datasets.maskeddataset import MaskedDataset
>>> from mvpa.datasets.splitter import OddEvenSplitter
>>> from mvpa.clfs.svm import LinearCSVMC
>>> from mvpa.clfs.transerror import TransferError
>>> from mvpa.algorithms.cvtranserror import CrossValidatedTransferError
>>> from mvpa.measures.searchlight import Searchlight
>>> from mvpa.misc.data_generators import normalFeatureDataset
>>>
>>> # overcomplicated way to generate an example dataset
>>> ds = normalFeatureDataset(perlabel=10, nlabels=2, nchunks=2,
...                           nfeatures=10, nonbogus_features=[3, 7],
...                           snr=5.0)
>>> dataset = MaskedDataset(samples=ds.samples, labels=ds.labels,
...                         chunks=ds.chunks)
>>>
>>> # setup measure to be computed in each sphere (cross-validated
>>> # generalization error on odd/even splits)
>>> cv = CrossValidatedTransferError(
...          TransferError(LinearCSVMC()),
...          OddEvenSplitter())
>>>
>>> # setup searchlight with 5 mm radius and measure configured above
>>> sl = Searchlight(cv, radius=5)
>>>
>>> # run searchlight on dataset
>>> sl_map = sl(dataset)
```

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. The *example section* contains a typical application of this algorithm.

[1] Kriegeskorte, N., Goebel, R. & Bandettini, P. (2006). 'Information-based functional brain mapping.' Proceedings of the National Academy of Sciences of the United States of America 103, 3863-3868.

7.2 Statistical Testing of classifier-based Analyses

It is often desirable to be able to make statements like *"Performance is significantly above chance-level"*. However, as with other applications of statistics in classifier-based analyses there is the problem that we do not know the distribution of a variable like error or performance under the *H0* hypothesis to assign the adored p-values, i.e. the probability of a result given that there is no signal. Even worse, the chance-level or guess probability of a classifier depends on the content of a validation dataset, e.g. balanced or unbalanced number of samples per label and total number of labels).

One approach to deal with this situation is to estimate the *NULL* distribution. A generic way to do this are permutation tests (aka *Monte Carlo*). The *NULL* distribution is estimated by computing some measure multiple times using datasets with no relevant signal in them. These datasets are generated by permuting the labels of all samples in the training dataset each time the measure is computed, and therefore randomizing/removing any possible relevant information.

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

PyMVPA supports such permutations test for *transfer errors* and all *dataset measures*. In both cases the object computing the measure or transfer error takes an optional contructor argument *null_dist*. The value of this argument is an instance of some Distribution estimator. If this is provided the respective TransferError or DatasetMeasure instance will automatically use it to estimate the *NULL* distribution and store the associated *p*-values in a state variable named *null_prob*.

```
>>> # lazy import
>>> from mvpa.suite import *
>>>
>>> # some example data with signal
>>> train = normalFeatureDataset(perlabel=50, nlabels=2, nfeatures=3,
...                              nonbogus_features=[0,1], snr=3, nchunks=1)
>>>
>>> # define class to estimate NULL distribution of errors
>>> # use left tail of the distribution since we use MeanMatchFx as error
>>> # function and lower is better
>>> # in a real analysis the number of permutations should be MUCH larger
>>> terr = TransferError(clf=SMLR(),
...                      null_dist=MCNullDist(permutations=10,
...                                           tail='left'))
>>>
>>> # compute classifier error on training dataset (should be low :)
>>> err = terr(train, train)
>>> err < 0.4
True
>>> # check that the result is highly significant since we know that the
>>> # data has signal
>>> terr.null_prob < 0.01
True
```

CHAPTER

EIGHT

Miscellaneous

8.1 Progress Tracking

There are 3 types of messages PyMVPA can produce:

verbose regular informative messages about generic actions being performed

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

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

8.1.1 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

8.1.2 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 optVerbose for precrafted command line option for to give facility to change it from your script (see examples)
- environment variable `MVPA_VERBOSE`
- code: verbose.level property

The following verbosity levels are supported:

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

8.1.3 Warning Messages

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

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

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

8.1.4 Debug Messages

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

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

- command line: you can use optDebug for precrafted command line option to provide it from your script (see examples). If in command line if optDebug is used, '`-d list`' is given, PyMVPA will print out list of known ids.
- environment: variable `MVPA_DEBUG` can contain comma-separated list of ids or python regular expressions to match multiple ids. Thus specifying `MVPA_DEBUG=CLF.*` would enable all ids which start with `CLF`, and `MVPA_DEBUG=.*` would enable all known ids.
- code: debug.active property (e.g. '`debug.active = [ 'RFEC', 'CLF' ]`')

Besides printing debug messages, it is also possible to print some metric. You can define new metrics or select predefined ones (vmem, asctime, pid). To enable list of metrics you can use `MVPA_DEBUG_METRICS` environment variable to list desired metric names comma-separated.

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. E.g. `RETRAIN` id would cause additional checking of the data in retraining phase. Such additional testing might spot out some bugs in the internal logic.

8.2 Additional Little Helpers

8.2.1 Random Number Generation

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

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

8.2.2 Others

(to be written)

8.3 FSL Bindings

(to be written)

CHAPTER

NINE

Data vs. Dataset: A Glossary

(to be written)

CHAPTER

TEN

PyMVPA for Matlab Users

(to be written)

CHAPTER

ELEVEN

Frequently Asked Questions

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

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

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

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

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

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

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

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

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

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

CHAPTER

TWELVE

Examples

Each of the following examples 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 include some more special-interest examples which are not listed here.

Some examples need to access sample dataset available under *data/* directory within root of PyMVPA hierarchy, thus they have to be invoked directly from PyMVPA root (e.g. *doc/examples/searchlight_2d.py*).

12.1 Simple Plotting of Classifier Behavior

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

```
#!/usr/bin/env python
#emacs: -*- mode: python-mode; py-indent-offset: 4; indent-tabs-mode: nil -*-
#ex: set sts=4 ts=4 sw=4 et:
### ### ### ### ### ### ### ### ### ### ### ### ### ### ### ### ### ### ##
#
#   See COPYING file distributed along with the PyMVPA package for the
#   copyright and license terms.
#
### ### ### ### ### ### ### ### ### ### ### ### ### ### ### ### ### ### ##
"""Example demonstrating a simple classifiction of a 2-D dataset"""

from mvpa.suite import *
"""
# Command above substitutes the following list

import numpy as N
import pylab as P

# local imports
from mvpa.datasets import Dataset
from mvpa.clfs.plr import PLR
from mvpa.clfs.ridge import RidgeReg
from mvpa.clfs.svm import RbfNuSVMC,LinearNuSVMC
from mvpa.clfs.knn import kNN
"""


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

```
feat_neg=N.random.randn(2, num_dat)
feat_neg[0, :] *= .5
feat_neg[1, :] *= 2.
feat_neg[0, :] -= dist

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

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

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

# loop over classifiers and show how they do
fig = 0

# make a new figure
P.figure(figsize=(8,12))
for c in clfs:
    # tell which one we are doing
    print "Running %s classifier..." % (c)

    # make a new subplot for each classifier
    fig += 1
    P.subplot(3,2,fig)

    # plot the training points
    P.plot(feat_pos[0, :], feat_pos[1, :], "r.")
    P.plot(feat_neg[0, :], feat_neg[1, :], "b.")

    # select the clasifier
    clf = clfs[c]

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

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

    # run the predictions on the test values
    pre = clf.predict(feat_test.T)

    # if ridge, use the prediction, otherwise use the values
    if c == 'Ridge Regression' or c == 'k-Nearest-Neighbour':
        # use the prediction
        res = N.asarray(pre)
    elif c == 'Logistic Regression':
        # get out the values used for the prediction
        res = N.asarray(clf.values)
    else:
```

```
        # get the probabilities from the svm
        res = N.asarray([(q[1][1] - q[1][0] + 1) / 2
                for q in clf.probabilities])

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

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

    # add the title
    P.title(c)

# show all the cool figures
P.show()
```

12.2 Easy Searchlight

Run a searchlight analysis on the example fMRI dataset that is shipped with PyMVPA. This example is part of the PyMVPA source distribution: doc/examples/searchlight_2d.py‘.

```
#!/usr/bin/python
#emacs: -*- mode: python-mode; py-indent-offset: 4; indent-tabs-mode: nil -*-
#ex: set sts=4 ts=4 sw=4 et:
### ### ### ### ### ### ### ### ### ### ### ### ### ### ### ### ### ### ##
#
#   See COPYING file distributed along with the PyMVPA package for the
#   copyright and license terms.
#
### ### ### ### ### ### ### ### ### ### ### ### ### ### ### ### ### ### ##
"""Example demonstrating a searchlight analysis on an fMRI dataset"""

from mvpa.suite import *
"""
# Command above substitutes commands below

import numpy as N
import pylab as P

# local imports
from mvpa.misc.iohelpers import SampleAttributes
from mvpa.datasets.niftidataset import NiftiDataset
from mvpa.datasets.misc import zscore
from mvpa.misc.signal import detrend
from mvpa.clfs.knn import kNN
from mvpa.clfs.svm import LinearNuSVMC
from mvpa.clfs.transerror import TransferError
from mvpa.datasets.splitter import NFoldSplitter, OddEvenSplitter
from mvpa.algorithms.cvtranserror import CrossValidatedTransferError
from mvpa.measures.searchlight import Searchlight
from mvpa.misc import debug
"""

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

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

#
# preprocessing
#

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

# only use 'rest', 'house' and 'scrambled' samples from dataset
dataset = dataset.selectSamples(
               N.array([ l in [0,2,6] for l in dataset.labels], dtype='bool'))

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

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

#
# Run Searchlight
#

# choose classifier
clf = LinearNuSVMC()

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

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


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

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

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

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

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

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

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


# show all the cool figures
P.show()
```

12.3 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.

```
#!/usr/bin/env python
#emacs: -*- mode: python-mode; py-indent-offset: 4; indent-tabs-mode: nil -*-
#ex: set sts=4 ts=4 sw=4 et:
### ### ### ### ### ### ### ### ### ### ### ### ### ### ### ### ### ### ### ##
#
#   See COPYING file distributed along with the PyMVPA package for the
#   copyright and license terms.
#
### ### ### ### ### ### ### ### ### ### ### ### ### ### ### ### ### ### ### ##
"""Example demonstrating some FeaturewiseDatasetMeasures performing on a fMRI
dataset with brain activity recorded while perceiving images of objects
(shoes vs. chairs).

Generated images show sensitivity maps computed by six sensitivity analyzers.

This example assumes that the PyMVPA example dataset is located in data/.
"""

from mvpa.suite import *
"""
# Command above substitutes commands below

import numpy as N
import pylab as P

# local imports
from mvpa.datasets.niftidataset import NiftiDataset
from mvpa.misc.iohelpers import SampleAttributes
from mvpa.measures.anova import OneWayAnova
from mvpa.clfs.svm import LinearNuSVMC
from mvpa.datasets.misc import zscore
from mvpa.misc.signal import detrend
from mvpa.measures.splitmeasure import SplitFeaturewiseMeasure
from mvpa.datasets.splitter import OddEvenSplitter, NFoldSplitter
"""

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

```
                        mask='data/mask.nii.gz')

# define sensitivity analyzer
sensanas = {'a) ANOVA': OneWayAnova(transformer=N.abs),
            'b) Linear SVM weights': LinearNuSVMC().getSensitivityAnalyzer(
                                                        transformer=N.abs),
            'c) Splitting ANOVA (odd-even)':
                SplitFeaturewiseMeasure(OneWayAnova(transformer=N.abs),
                                        OddEvenSplitter()),
            'd) Splitting SVM (odd-even)':
                SplitFeaturewiseMeasure(
                    LinearNuSVMC().getSensitivityAnalyzer(transformer=N.abs),
                                    OddEvenSplitter()),
            'e) Splitting ANOVA (nfold)':
                SplitFeaturewiseMeasure(OneWayAnova(transformer=N.abs),
                                        NFoldSplitter()),
            'f) Splitting SVM (nfold)':
                SplitFeaturewiseMeasure(
                    LinearNuSVMC().getSensitivityAnalyzer(transformer=N.abs),
                                    NFoldSplitter())
           }

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

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

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

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

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

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

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

    # compute sensitivies
    smap = sensanas[s](dataset)

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

    # make a new subplot for each classifier
    fig += 1
    P.subplot(3,2,fig)

    P.title(s)

    P.imshow(masked_orig_smap[0],
             interpolation='nearest',
             aspect=1.25,
```

```
            cmap=P.cm.autumn)

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

    P.colorbar(shrink=0.6)


# show all the cool figures
P.show()
```

12.4 Classification of SVD-mapped Datasets

Demonstrate the usage of a dataset mapper performing singular value decomposition within a cross-validation.

```
#!/usr/bin/python
#emacs: -*- mode: python-mode; py-indent-offset: 4; indent-tabs-mode: nil -*-
#ex: set sts=4 ts=4 sw=4 et:
### ### ### ### ### ### ### ### ### ### ### ### ### ### ### ### ### ### ##
#
#   See COPYING file distributed along with the PyMVPA package for the
#   copyright and license terms.
#
### ### ### ### ### ### ### ### ### ### ### ### ### ### ### ### ### ### ##
"""Example demonstrating a how to use data projection onto SVD components
for *any* clasifier"""

from mvpa.suite import *
"""
# Command above substitutes commands below

import numpy as N
import pylab as P

# local imports
from mvpa.misc.iohelpers import SampleAttributes
from mvpa.datasets.niftidataset import NiftiDataset
from mvpa.datasets.misc import zscore
from mvpa.misc.signal import detrend
from mvpa.clfs.transerror import TransferError
from mvpa.datasets.splitter import NFoldSplitter
from mvpa.algorithms.cvtranserror import CrossValidatedTransferError
from mvpa.clfs.svm import LinearCSVMC
from mvpa.clfs.base import MappedClassifier
from mvpa.mappers import SVDMapper

from mvpa.misc import debug
"""

debug.active += ["CROSSC"]

# plotting helper function
def makeBarPlot(data, labels=None, title=None, ylim=None, ylabel=None):
    xlocations = N.array(range(len(data))) + 0.5
    width = 0.5

    # work with arrays
```

```
    data = N.array(data)

    # plot bars
    plot = P.bar(xlocations,
                 data.mean(axis=1),
                 yerr=data.std(axis=1) / N.sqrt(data.shape[1]),
                 width=width,
                 color='0.6',
                 ecolor='black')
    P.axhline(0.5, ls='--', color='0.4')

    if ylim:
        P.ylim(*(ylim))
    if title:
        P.title(title)

    if labels:
        P.xticks(xlocations+ width/2, labels)

    if ylabel:
        P.ylabel(ylabel)

    P.xlim(0, xlocations[-1]+width*2)


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

#
# preprocessing
#

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

# only use 'rest', 'face' and 'house' samples from dataset
dataset = dataset.selectSamples(
                N.array([ l in [0,4,5] for l in dataset.labels], dtype='bool'))

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

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

# define some classifiers: a simple one and several classifiers with built-in
# SVDs
clfs = [('All orig. features', LinearCSVMC()),
        ('All PCs', MappedClassifier(LinearCSVMC(), SVDMapper())),
        ('First 3 PCs', MappedClassifier(LinearCSVMC(),
                         SVDMapper(selector=range(5)))),
        ('First 50 PCs', MappedClassifier(LinearCSVMC(),
                         SVDMapper(selector=range(50)))),
        ('PCs 3-50', MappedClassifier(LinearCSVMC(),
                         SVDMapper(selector=range(3,50))))]
```

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

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

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

makeBarPlot(results,labels=labels, title='Linear C-SVM classification')
P.show()
```

12.5 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.

```
#!/usr/bin/env python
#emacs: -*- mode: python-mode; py-indent-offset: 4; indent-tabs-mode: nil -*-
#ex: set sts=4 ts=4 sw=4 et:
### ### ### ### ### ### ### ### ### ### ### ### ### ### ### ### ### ### ##
#
#   See COPYING file distributed along with the PyMVPA package for the
#   copyright and license terms.
#
### ### ### ### ### ### ### ### ### ### ### ### ### ### ### ### ### ### ##
"""Example demonstrating a SMLR classifier"""

from mvpa.suite import *
"""
# Command above substitutes commands below

import numpy as N

from mvpa.datasets import Dataset
from mvpa.clfs.smlr import SMLR
from mvpa.clfs.svm import LinearNuSVMC
from mvpa.clfs.transerror import ConfusionMatrix
"""

from mvpa.misc import debug
debug.active.append('SMLR_')

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

# create the sample datasets
```

```
samp1 = N.random.randn(nsamp,nfeat)
samp1[:,:goodfeat] += offset

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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


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

CHAPTER

THIRTEEN

License

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

CHAPTER

FOURTEEN

PyMVPA Development Changelog

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

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

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

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

Unreleased changes Changes described here are not yet released, but available from VCS repository.

(currently none)

14.1 Releases

- 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.

INDEX

R

S

T

U

V

W