



# PyMVPA Developer Guidelines

*Release 0.5.0.dev*

**PyMVPA Authors**

April 05, 2010



# CONTENTS

<b>1</b>	<b>Git Repository</b>	<b>1</b>
1.1	Layout	1
1.2	Commits	1
1.3	Merges	2
<b>2</b>	<b>Code</b>	<b>3</b>
2.1	Formatting	3
2.2	Docstrings	4
2.3	Classes	4
2.4	Function Arguments	4
<b>3</b>	<b>Documentation</b>	<b>5</b>
3.1	Examples	5
<b>4</b>	<b>Tests</b>	<b>7</b>
<b>5</b>	<b>Changelog</b>	<b>9</b>
<b>6</b>	<b>Extending PyMVPA</b>	<b>11</b>
6.1	Adding an External Dependency	11
6.2	Adding a new Dataset type	11
6.3	Adding a new Classifier	11
6.4	Adding a new DatasetMeasure	12
6.4.1	Classifier-independent Sensitivity Analyzers	13
6.4.2	Classifier-based Sensitivity Analyzers	13
6.5	Adding a new Algorithm	13
<b>7</b>	<b>Developer-TODO</b>	<b>15</b>
7.1	Things to implement for the next release (Release goals)	15
7.2	Long and medium term TODOs (aka stuff that has been here forever)	17
<b>8</b>	<b>Building a binary installer on MacOS X 10.5</b>	<b>19</b>
	<b>Index</b>	<b>21</b>



# GIT REPOSITORY

## 1.1 Layout

The git repository is structured by a number of branches. Originally we followed the scheme, where each developer prefixed his/her branches with a unique string plus `'/` (maybe initials or similar). That is why currently there are:

```
eo
  Emanuele Olivetti
mh
  Michael Hanke
per
  Per B. Sederberg
yoh
  Yaroslav O. Halchenko
```

At the moment, with availability of PyMVPA repository at [github](#), such convention slowly gets superseded with adopting a true distributed workflow: Anyone is welcome to fork the repository on [github](#) (just click on “Fork” button), and file a “Pull request” whenever he/she thinks that his changes are ready to be included (merged) into the main repository. Alternatively, if you just want quickly submit a patch – just email it to the mailing list. You might take advantage of `git format-patch` command.

Any developer can have an infinite number of branches. If the number of branches causes `gitk` output to exceed a usual 19” screen, the respective developer has to spend some bucks (or euros) on new screens for all others ;-)

The main release branch is called *master*. This is a merge-only branch. Features finished or updated by some developer are merged from the corresponding branch into *master*. At a certain point the current state of *master* is tagged – a release is done.

Only usable feature should end-up in *master*. Ideally *master* should be releasable at all times. Something must not be merged into *master* if *any* unit test fails. Maintenance branches should be gone into *maint/epoch.major* branches (e.g. *maint/0.4*).

Additionally, there are packaging branches (prefixed with *dist/*). They are labeled after the package target (e.g. *debian* for a Debian package). Releases are merged into the packaging branches, packaging get updated if necessary and the branch gets tagged when a package version is released. Packaging maintenance (as well as backport) releases should be gone under *dist/target/variant/codename* (e.g. *dist/debian/maint/lenny*, *dist/debian/security/lenny*). Branch *dist/debian/dev* is dedicated for Debian packages of development snapshots.

## 1.2 Commits

Please prefix all commit summaries with one (or more) of the following labels. This should help others to easily classify the commits into meaningful categories:

- *BF* : bug fix

- *RF* : refactoring
- *NF* : new feature
- *ENH* : enhancement of an existing feature/facility
- *BW* : addresses backward-compatibility
- *OPT* : optimization
- *BK* : breaks something and/or tests fail
- *PL* : making pylint happier
- *DOC*: for all kinds of documentation related commits

### 1.3 Merges

For easy tracking of what changes were absorbed during merge, we advice to enable merge summary within git:

```
git-config merge.summary true
```

# CODE

## 2.1 Formatting

### pylint

Code should be conformant with [Pylint](#) driven by config located at `doc/misc/pylintrc`. Previously we used camelBack notation, but since 0.5.0.dev we decided to mainly follow [PEP8](#). Provided `pylintrc` file assures consistent coding style and also warns about obvious errors in the code. To engage, use 1 of 3 methods:

- place it in `~/pylintrc` for user-wide installation
- use within a call to pylint:

```
pylint --rcfile=$PWD/doc/misc/pylintrc
```

- export environment variable from mvpa sources top directory:

```
export PYLINTRC=$PWD/doc/misc/pylintrc
```

### 2 empty lines

According to original python style guidelines: single empty line to separate methods within class, and 2 empty lines between classes **BUT** we do 2 empty between methods, 3 empty between classes

### module docstring

Each module should start with a docstring describing the module (which is not inside the hashed-comment of each file) look at `mapper` or `neighbor` for tentative organization if copyright/license has to be present in each file.

### header

Each file should contain a header from `doc/misc/header.py`.

### notes

Use following keywords will be caught by pylint to provide a summary of what yet to be done in the given file

#### FIXME

something which needs fixing (sooner than later)

#### TODO

future plan (i.e. later than sooner)

#### XXX

some concern/question

#### YYY

comment/answer to above mentioned XXX concern

#### WiP

Work in Progress: API and functionality might rapidly change

## 2.2 Docstrings

All documentation should be written using NumPy documentation conventions:

<http://projects.scipy.org/numpy/wiki/CodingStyleGuidelines#docstring-standard>

## 2.3 Classes

Classes should define:

`__repr__`  
whenever possible, should be the string representation of the object which could be digested with `func:eval` if necessary.

`__str__`  
most of the classes should provide meaningful and concise summary over their identity (name + parameters + some summary over results if any), not necessarily *evaluable*.

**Note:** *Classes derived from `ClassWithCollections` and using `params` and `ca` collections for their need of parametrization (e.g. `Classifier`) would obtain an acceptable definitions of `__repr__` and `__str__` automatically.*

## 2.4 Function Arguments

dataset vs data

Ones which are supposed to be derived from `Dataset` class should have suffix (or whole name) `dataset`. In contrast, if argument is expected to be simply a NumPy array, suffix should be `data`. For example:

```
class Classifier(ClassWithCollections):
    ...
    def train(self, dataset):
    ...
    def predict(self, data):

class FeatureSelection(ClassWithCollections):
    ...
    def __call__(self, dataset, testdataset):
```

Such convention should be enforced in all `*train`, `*predict` functions of classifiers.

# DOCUMENTATION

## 3.1 Examples

Examples should be complete and stand-alone scripts located in `doc/examples`. If an example involves any kind of interactive step, it should honor the `MVPA_EXAMPLES_INTERACTIVE` setting, to allow for automatic testing of all examples. In case of a matplotlib-based visualization such snippet should be sufficient:

```
from mvpa import cfg
if cfg.getboolean('examples', 'interactive', True):
    pl.show()
```

All examples are automatically converted into RsT documents for inclusion in the manual. Each of them is preprocessed in the following way:

- Any header till the first docstring is stripped.
- Each top-level (non-assigned) docstring is taken as a text block in the generated RsT source file. Such a docstring might appear anywhere in the example, not just at the beginning. In this case, the code snippet is properly split and the text block is inserted at the corresponding location.
- All remaining lines are treated as code and inserted in the RsT source with appropriate markup.

The first docstring in each example must have a proper section heading (with '=' markup).

Finally, each example should be added to the appropriate `toctree` in `doc/examples.rst` and included into a test battery by adding it to `testexamples` rule within `Makefile` in the `toplevel` directory of the source tree.



# TESTS

We are slowly moving toward utilizing [nose testing framework](#). It allows to carry out not only unit testing, but also verify correctness of the code snippets provided in the docstrings and the manual. All unit tests are stored in `mvpa.tests`, and they make use of `mvpa.testing` which provides

`tools`  
basic tools (imported within `__init__`)

`clfs`  
some additional classifiers to be used in the unittests

`datasets`  
pre-crafted datasets *warehouse* to be used in the tests

`sweepargs`  
defines a custom decorator to allow running the same unittest on a range of input values and later on nicely summarize the detected failures

While working on the project we adhere to the following rules:

- Every more or less “interesting” bugfix should be accompanied by a unittest which might help to prevent it in the future refactoring
- Every new feature should have a unittest
- Unit tests that might be non-deterministic (e.g. depending on classifier performance, which is turn is randomly initialized) should be made conditional like this:

```
>>> from mvpa import cfg
>>> if cfg.getboolean('tests', 'labile', default='yes'):
...     pass
```

- Every additional unit test submodule (or a unittest method itself) requiring specific external being present should make use of `skip_if_no_external()`, e.g.

```
>>> from mvpa.testing import *
>>> skip_if_no_external('numpy')
```

Furthermore we encourage detailed docstrings for the classes, including *Examples* section with the demonstration of most typical use cases and aspects of the classes. Those snippets are also part of the tests battery



# CHANGELOG

The PyMVPA changelog is located in the toplevel directory of the source tree in the `Changelog` file. The content of this file should be formatted as restructured text to make it easy to put it into manual appendix and on the website.

This changelog should neither replicate the VCS commit log nor the distribution packaging changelogs (e.g. `debian/changelog`). It should be focused on the user perspective and is intended to list rather macroscopic and/or important changes to the module, like feature additions or bugfixes in the algorithms with implications to the performance or validity of results.

It may list references to 3rd party bug trackers, in case the reported bugs match the criteria listed above.

Changelog entries should be tagged with the name of the developer(s) (mainly) involved in the modification – initials are sufficient for people contributing regularly.

Changelog entries should be added whenever something is ready to be merged into the master branch, not necessarily with a release already approaching.



# EXTENDING PYMVPA

This section shall provide a developer with the necessary pieces of information for writing extensions to PyMVPA. The guidelines given here, must be obeyed to ensure a maximum of compatibility and inter-operability. As a consequence, all modifications that introduce changes to the basic interfaces outlined below have to be documented here and also should be announced in the changelog.

## 6.1 Adding an External Dependency

Introducing new external dependencies should be done in a completely optional fashion. This includes both build-dependencies and runtime dependencies. With `mvpa.base.externals` PyMVPA provides a simple framework to test the availability of certain external components and publish the results of the tests throughout PyMVPA.

## 6.2 Adding a new Dataset type

- Required interface for Mapper.
- only new subclasses of `MappedDataset` + new Mappers (all other as improvements into the `Dataset` base class)?

go into `mvpa/datasets/`

## 6.3 Adding a new Classifier

To add a new classifier implementation it is sufficient to create a new sub-class of `Classifier` and add implementations of the following methods:

`__init__(**kwargs)`  
Additional arguments and keyword arguments may be added, but the base-class constructor has to be called with `**kwargs`!

`_train(dataset)`  
Has to train the classifier when it is called with a `Dataset`. Successive calls to this methods always have to train the classifier on the respective datasets. An eventually existing prior training status has to be cleared automatically. Nothing is returned.

`_predict(data)`  
Unlike `_train()` the method is not called with a `Dataset` instance, but with any sequence of data samples (e.g. arrays). It has to return a sequence of predictions, one for each data sample.

With this minimal implementation the classifier provides some useful functionality, by automatically storing some relevant information upon request in conditional attributes.

Supported conditional attributes:

Name	Description	Default
feature_ids	Feature IDS which were used for the actual training.	Disabled
predicting_time	Time (in seconds) which took classifier to predict.	Enabled
predictions	Most recent set of predictions.	Enabled
trained_dataset	The dataset it has been trained on.	Disabled
trained_targets	Set of unique labels it has been trained on.	Enabled
training_confusion	Confusion matrix of learning performance.	Disabled
training_time	Time (in seconds) which took classifier to train.	Enabled
values	Internal classifier values the most recent predictions are based on.	Disabled

If any intended functionality cannot be realized by implementing above methods. The `Classifier` class offers some additional methods that might be overridden by sub-classes. For all methods described below it is strongly recommended to call the base-class methods at the end of the implementation in the sub-class to preserve the full functionality.

`__pretrain(dataset)`

Called with the `Dataset` instance that shall be trained with, but before the actual training is performed.

`__posttrain(dataset)`

Called with the `Dataset` instance the classifier was trained on, just after training was performed.

`__prepredict(data)`

Called with the data samples the classifier should do a prediction with, just before the actual `__predict()` call.

`__postpredict(data, result)`

Called with the data sample for which predictions were made and the resulting predictions themselves.

Source code files of all classifier implementations go into `mvpa/clfs/`.

Outstanding Questions:

- when `ca` and when properties?

## 6.4 Adding a new DatasetMeasure

There are few possible base-classes for new measures (former sensitivity analyzers). First, `DatasetMeasure` can directly be sub-classed. It is a base class for any measure to be computed on a `Dataset`. This is the more generic approach. In the most of the cases, measures are to be reported per each feature, thus `FeaturewiseDatasetMeasure` should serve as a base class in those cases. Furthermore, for measures that make use of some classifier and extract the sensitivities from it, `Sensitivity` (derived from `FeaturewiseDatasetMeasure`) is a more appropriate base-class, as it provides some additional useful functionality for this use case (e.g. training a classifier if needed).

All measures (actually all objects based on `DatasetMeasure`) support a `transformer` keyword argument to their constructor. The functor passed as its value is called with the to be returned results and its outcome is returned as the final results. By default no transformation is performed.

If a `DatasetMeasure` computes a characteristic, were both large positive and large negative values indicate high relevance, it should nevertheless *not* return absolute sensitivities, but set a default transformer instead that takes the absolute (e.g. plain `np.absolute` or a convenience wrapper `Absolute`).

To add a new measure implementation it is sufficient to create a new sub-class of `DatasetMeasure` (or `FeaturewiseDatasetMeasure`, or `Sensitivity`) and add an implementation of the `__call(dataset)` method. It will be called with an instance of `Dataset`. `FeaturewiseDatasetMeasure` (e.g. `Sensitivity` as well) has to return a vector of featurewise sensitivity scores.

Supported conditional attributes:

Name	Description	Default
null_prob	Conditional attribute.	Enabled
raw_results	Computed results before applying any transformation algorithm.	Disabled

Source code files of all sensitivity analyzer implementations go into `mvpa/measures/`.

### 6.4.1 Classifier-independent Sensitivity Analyzers

Nothing special.

### 6.4.2 Classifier-based Sensitivity Analyzers

A `Sensitivity` behaves exactly like its classifier-independent sibling, but additionally provides support for embedding the necessary classifier and handles its training upon request (boolean `force_training` keyword argument of the constructor). Access to the embedded classifier object is provided via the `clf` property.

Supported conditional attributes:

Name	Description	Default
<code>base_sensitivities</code>	Stores basic sensitivities if the sensitivity relies on combining multiple ones.	Disabled
<code>null_prob</code>	Conditional attribute.	Enabled
<code>raw_results</code>	Computed results before applying any transformation algorithm.	Disabled

Outstanding Questions:

- What is a `mvpa.measures.base.ProxyClassifierSensitivityAnalyzer` useful for?
- Shouldn't there be a `sensitivities` state?

## 6.5 Adding a new Algorithm

go into `mvpa/algorithms/`



# DEVELOPER-TODO

## 7.1 Things to implement for the next release (Release goals)

- A part of below restructuring TODO but is separate due to its importance: come up with cleaner hierarchy and tagging of classifiers and regressions – now they are all `Classifier`
- Unify parameter naming across all classifiers and come up with a labeling guideline for future classifier implementations and wrappers:

Numeric parameters can be part of `.params` `Collection` now, so they are joined together.

- Provide sufficient documentation about internal variable naming to make `Harvester/Harvesting` functionality usable. Currently the user is supposed to know, how a particular *local* variable is called to be able to harvest e.g. `feature_ids` of classifiers over cross-validation folds:

```
class.HARVESTABLE={'blah' : ' some description'}
```

Add information on `HARVESTABLE` and `ConditionalAttribute`  
`Collectable` -> `Attribute`

```
base.attributes
```

- Restructure code base (incl. renaming and moving pieces)

Let's use the following list to come up with a nice structure for all logical components we have:

- Datasets
- Sensitivity analyzers (maybe: featurewise measures) \* Classifier sensitivities (SVM, SMLR)  
-> respective classifiers \* ANOVA -> `mvpa.measures.anova` \* Noise perturbation -> ->  
`mvpa.measures.noisepertrubation` \* meta-algorithms (splitting) -> `mvpa.measures`

`DatasetMeasure` -> `Measure` (transformers)

`FeaturewiseDatasetMeasure?`

combiners to be absorbed withing transformers? and then gone? `{Classifier?}Sensitivity?`

Mappers::

`mvpa.mappers` (AKA `mvpa.projections` `mvpa.transformers`)

- \* Along with PCA/ICA mappers, we should add a PLS mapper:

```
PCA.train(learningdataset)  
    .forward,  
    .backward
```

Package `pychem` for Debian, see how to use from `PyMVPA!` ;-) Same for MDP  
(i.e. use from `pymvpa`)

Feature selection algorithms

- \* Simple thresholding
    - \* RFE
    - \* IFS
  - .mapper conditional attribute
    - mvpa.featsel (NB no featsel.featsel.featsel more than 4 times!) mvpa.featsel.rfe
    - mvpa.featsel.ifs
  - several base classes with framework infrastructure (Harvester, ClassWithCollections, virtual properties, ...)
  - Transfer error calculation
  - Cross-validation support
  - Monte-Carlo-based significance testing
  - Dataset splitter
  - Metrics and distance functions
  - Functions operating on dataset for preprocessing or transformations
  - Commandline interface support
  - Functions to generate artificial datasets
  - Error functions (i.e. for TransferError)
  - Custom exception types
  - Python 2.5 copy() aka external code shipped with PyMVPA
  - Several helpers for data IO
  - Left-over from the last attempt to establish a generic parameter interface
  - Detrending (operating on Datasets)
  - Result 'Transformers' to be used with 'transformer=' kwarg
  - Debugging and verbosity infrastructure
  - plus additional helpers, ranging from simple to complex scattered all over the place
- Resultant hierarchy:
    - mvpa
      - \* datasets
      - \* clfs
      - \* measures
      - \* featsel
  - Add ability to add/modify custom attributes to a dataset.
  - Possibly make NiftiDataset default to float32 when it sees that the data are ints.
  - Add kernel methods as option to all classifiers, not just SVMs. For example, you should be able to run a predefined or custom kernel on the samples going into SMLR.
  - TransferError needs to know what type of data to send to any specific ErrorFX. Right now there is only support for predictions and labels, but the area under the ROC and the correlation-based error functions expect to receive the "values" or "probabilities" from a classifier. Just to make this harder, every classifier is different. For example, a ridge regression's predictions are continuous values, whereas for a SVM you need to pass in the probabilities.

For binary: 1 value  
 multiclass: 1 value, or N values

- In a related issue, the predictions and values ca of the classifiers need to have a consistent format. Currently, SVM returns a list of dictionaries for values and SMLR returns a `NumPy` ndarray.

## 7.2 Long and medium term TODOs (aka stuff that has been here forever)

- `selected_ids` -> implement via `MaskMapper`?

yoh:

it might be preferable to manipulate/expose `MaskMapper` instead of plain list of `selected_ids` within `FeatureSelection` classes

- unify naming of working/testing
  - `transerror.py` for instance uses `testdata/trainingdata`
  - `rfe.py` dataset, `testdataset`
- implement proper cloning of classifiers. `untrain()` doesn't work in some cases, since we can create somewhat convolved object definitions so it is hard, if not impossible, to get to all used classifiers. See for instance `clfsw[ 'SVM/Multiclass+RFE' ]`. We can't get all the way into classifier-based sensitivity analyzer. Thus instead of tracking all the way down in hierarchy, we should finally create proper 'parametrization' handling of classifiers, so we could easily clone basic ones (which might have active SWIG bindings), and top-level ones should implement `.clone()` themselves. or may be some other way, but things should be done. Or may be via proper implementation of `__reduce__` etc
- `mvpa.misc.warning` may be should use stock python warnings module instead of custom one?
- `ConfusionBasedError` -> `InternalError` ?
- Renaming of the modules `transerror.py` -> `errors.py`
- SVM: `get_sv` and `get_sv_coef` return very 'packed' presentation whenever classifier is multiclass. Thus they have to be unpacked before proper use (unless it is simply a binary classifier).
- Regression tests: for instance using sample dataset which we have already, run `doc/examples/searchlight.py` and store output to validate against. Probably the best would be to create a regression test suite within unit tests which would load the dataset and run various algorithms on it a verify the results against previously obtained (and dumped to the disk)
- `feature_selector` – may be we should return a tuple (`selected_ids`, `discarded_ids`)?

Michael:

Is there any use case for that? `ElementSelector` can 'select' and 'discard' already. DO we need both simultaneously?

- Non-linear SVM RFE
- `ParameterOptimizer` (might be also `OptimizedClassifier` which uses `parameterOptimizer` internally but as the result there is a classifier which automatically optimizes its parameters. It is close in idea to classifier based on RFE)



# BUILDING A BINARY INSTALLER ON MACOS X 10.5

A simple way to build a binary installer for Mac OS is `bdist_mpkg`. This is a `setuptools` extension that uses the proper native parts of MacOS to build the installer. However, for PyMVPA there are two problems with `bdist_mpkg`: 1. PyMVPA uses `distutils` not `setuptools` and 2. current `bdist_mpkg` 0.4.3 does not work for MacOS X 10.5 (Leopard). But both can be solved.

Per 1) A simple wrapper script in `tools/mpkg_wrapper.py` will enable the use of `setuptools` on top of `distutils`, while keeping the `distutils` part in a usable state.

Per 2) The following patch (against 0.4.3.) makes `bdist_mpkg` compatible with MacOS 10.5. It basically changes the way `bdist_mpkg` determined the GID of the admin group. 10.5 removed the `nidump` command:

```
diff -rNu bdist_mpkg-0.4.3/bdist_mpkg/tools.py bdist_mpkg-0.4.3.leopard/bdist_mpkg/tools.py
--- bdist_mpkg-0.4.3/bdist_mpkg/tools.py      2006-07-09 00:39:00.000000000 -0400
+++ bdist_mpkg-0.4.3.leopard/bdist_mpkg/tools.py      2008-08-21 07:43:35.000000000 -0400
@@ -79,15 +79,12 @@
         yield os.path.join(root, fn)

     def get_gid(name, _cache={}):
-        if not _cache:
-            for line in os.popen('/usr/bin/nidump group .'):
-                fields = line.split(':')
-                if len(fields) >= 3:
-                    _cache[fields[0]] = int(fields[2])
-        try:
-            return _cache[name]
-        except KeyError:
-            raise ValueError('group %s not found' % (name,))
+        for line in os.popen("dscl . -read /Groups/" + name + " PrimaryGroupID"):
+            fields = [f.strip() for f in line.split(':')]
+            if fields[0] == "PrimaryGroupID":
+                return fields[1]
+        raise ValueError('group %s not found' % (name,))

     def find_root(path, base='/'):
         """
```



# INDEX

## E

environment variable

    MVPA\_EXAMPLES\_INTERACTIVE, 5

## M

MVPA\_EXAMPLES\_INTERACTIVE, 5