



PyMVPA Developer Guidelines

Release 0.2.2

Michael Hanke, Yaroslav Halchenko, Per B. Sederberg

July 03, 2008

CONTENTS

1	Documentation	1
1.1	Code Documentation	1
2	Code Formatting	3
3	Coding Conventions	5
4	Naming Conventions	7
4.1	Function Arguments	7
5	Tests	9
6	Extending PyMVPA	11
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
6.5	Adding a new Algorithm	13
7	Git Repository	15
7.1	Layout	15
7.2	Commits	15
7.3	Merges	16
8	Changelog	17
9	Developer-TODO	19
9.1	Things to implement for the next release (Release goals)	19
9.2	Long and medium term TODOs (aka stuff that has been here forever)	21

Documentation

Documentation of the code and supplementary material (such as this file) should be done in [reST](#) (reStructured-Text) light markup language. See [Demo](#) or a [Cheatsheet](#) for a quick demo.

1.1 Code Documentation

Code must be documented in accordance to [epydoc + reST usage guidelines](#)

Parameter lists should be written as definition lists and not bulleted lists. For an example how to do it right, please see [mvpa/datasets/dataset.py](#)

Code Formatting

pylint Code should be conformant with [Pylint](#) driven by config located at `doc/misc/pylintrc`. It assumes camel-back notation (classes start with capitals, functions with lowercase) and indentation using 4 spaces (ie no tabs) Variables are low-case and can have up to 2 `_`s. 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 (ie later than sooner)

XXX some concern/question

YYY comment/answer to above mentioned XXX concern

Coding Conventions

`__repr__` most of the classes should provide meaningful and concise summary over their identity (name + parameters + some summary over results if any)

Naming Conventions

4.1 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(Stateful):
    ...
    def train(self, dataset):
    ...
    def predict(self, data):

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

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

Tests

- 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

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 state variables.

Supported states:

State Name	Description	Default
feature_ids	Feature IDS which were used for the actual training. Some classifiers might internally do feature selection (SMLR).	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_labels	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 *_prediction()* 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 states 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, where 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 *N.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 states:

State Name	Description	Default
null_prob	State variable.	Enabled
raw_result	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 states:

State Name	Description	Default
base_sensitivities	Stores basic sensitivities if the sensitivity relies on combining multiple ones.	Disabled
null_prob	State variable.	Enabled
raw_result	Computed results before applying any transformation algorithm.	Disabled

Outstanding Questions:

- What is a ProxyClassifierSensitivityAnalyzer useful for?
- Shouldn't there be a *sensitivities* state?

6.5 Adding a new Algorithm

go into *mvpa/algorithms/*

Git Repository

7.1 Layout

The repository is structured by a number of branches. Each developer should prefix his/her branches with a unique string plus *'* (maybe initials or similar). Currently there are:

mh Michael Hanke
per Per B. Sederberg
yoh Yaroslav Halchenko

Each 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.

Additionally, there are packaging branches. 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. Maintenance (as well as backport) releases should be gone under *maint/codename.flavor* (e.g. *maint/lenny*, *maint/lenny.security*, *maint/sarge.bpo*).

7.2 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
- *OPT* : optimization
- *BK* : breaks something and/or tests fail
- *PL* : making pylint happier
- *DOC*: for all kinds of documentation related commits

7.3 Merges

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

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

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 bugtrackers, 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.

Developer-TODO

9.1 Things to implement for the next release (Release goals)

- 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 StateVariable
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 (ie use from `pymvpa`)

- **Feature selection algorithms** * Simple thresholding
 - * RFE

* IFS

.mapper state variable

mvpa.featsel (NB no featsel.featsel.featsel more than 4 times!) mvpa.featsel.rfe
mvpa.featsel.ifs

- several base classes with framework infrastructure (Harvester, Stateful, virtual properties, ...)
 - Classifiers
 - Regressions
 - 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 states of the classifiers need to have a consistent format. Currently, SVM returns a list of dictionaries for values and SMLR returns a numpy ndarray.

9.2 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 `clfs['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 ?
- **Think about how to deal with Transformers to serve them with** basic_analyzers... May be transformer can be an argument for any analyzer! Ha! Indeed... may be later
- Renaming of the modules transerror.py -> errors.py
- **SVM: getSV and getSVCoef 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)
- Agree on how to describe parameters to functions. Describe in NOTES.coding.
- 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?
- **Basic documentation: Examples (more is better) describing various use cases** (everything in the cncre should be done in examples)
- 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)
- **provide for Dataset – Dataset.__featattr which has attributes for** features similar to `__dsattr` way.
 - in -> data -> dataShape out -> features ->