PyML - a Python Machine Learning package

Release 0.7.0

Asa Ben-Hur

June 2, 2008

myfirstname@cs.colostate.edu

Abstract

PyML is an interactive object oriented framework for machine learning written in Python. PyML is focused on kernel-methods for classification and regression, including Support Vector Machines (SVM). It provides tools for feature selection, model selection, syntax for combining classifiers and methods for assessing classifier performance.

Contents

1 Installation

Requirements:

- Python version 2.3 and higher (2.2 has a peculiriarity in the way the list object is sub-classed that makes it not work).
- The `Numpy` package http://sourceforge.net/projects/numpy. There were some changes in the interface in version 1.0 of numpy, so 1.0 or higher is required.
- The `matplotlib` package (http://matplotlib.sourceforge.net) is required in order to use the graphical functionality of `PyML`, but can be used without it.

Currently Unix/Linux and Mac OS-X are supported (there is some C++ code so the library is not automatically portable). A setup.py script is provided so installation follows the standard python idiom:

```
python setup.py build
python setup.py install
```

To install `PyML` in a directory other than the standard site-packages directory uses the `prefix` option:

```
python setup.py install --prefix=a_directory
```

Note that python installs the package in a subdirectory of the specified directory. This requires you to add that subdirectory to your python path. An alternative is to issue the build command, and then add the appropriate build directory to your python path. When issuing the build command python creates the directory 'build' inside the PyML directory. The `PyML` code is found in a subdirectory with a name that reflects your architecture; on a macintosh running python 2.4 it can be something like: build/build/lib.macosx-10.5-i386-2.5/.

To check that `PyML` is installed correctly, go to the `data` directory of `PyML`, run the python interpreter and type

```
>>> from PyML import *
```

To test the installation:

```
>>> from PyML.demo import pyml_test
>>> pyml_test.test('svm')
```

which will test the SVM functionality of `PyML`. The check the version of `PyML` you're using do:

```
>>> import PyML
>>> PyML.__version__
```

NOTE: You may get an import error if you perform this import operation from the directory in which you performed the installation.

2 Getting help

To get help on any `PyML` class, method or function, type: `help('PyMLname')` at the python prompt or look at the API documentation which is at `doc/autodoc/index.html` of the `PyML` distribution. A copy of the tutorial is provided with the PyML distribution at `doc/tutorial.pdf`.

3 PyML overview

`PyML` has classes that contain data (dataset containers), and support computation of various kernels as well as various kernel-based classifiers. Additional functionality includes feature selection, model selection and preprocessing.

4 Importing modules and classes

NOTE: In version 0.7.0 the code has been restructured, so if you have used earlier versions you will need to change the way you import modules and classes.

There are several way of importing a `PyML` class. Suppose you want to import the `SparseDataSet` class which is in `PyML/containers/vectorDatasets`. The standard import statement

```
from PyML.containers.vectorDatasets import SparseDataSet
```

will obviously work. As a shortcut both

```
>>> from PyML.containers import SparseDataSet
```

and

```
>>> from PyML import *
```

will do the job. The `import *` statement imports some of the most commonly used `PyML` modules and classes. In what follows we will assume you have invoked this command, and we will point out when additional imports are required.

5 Data Containers

A dataset is a collection of patterns and their class labels. Class labels and pattern IDs are stored in a a `Labels` object accessible as the `labels` attribute of a dataset; it holds for each pattern its class label and pattern id.

PyML has several dataset containers:

- A container for vector data: VectorDataSet.
- A container for sparse vector data: SparseDataSet.
- A container for holding a precomputed kernel matrix: KernelData.
- A container for patterns that are composed of pairs of objects: PairDataSet.
- A container that collects an assortment of datasets into a single container: Aggregate.

PyML offers several methods for constructing a dataset instance:

1. Reading data from a file (the sparse format is compatible with the format used by libsvm and svmlight); delimited files are also supported.
2. Construction from a 2-dimensional array provided as a numpy array or a list of python lists.
3. Various flavors of copy construction.

5.1 Reading data from a file

Data is read from a file by calling the constructor of a dataset container class with a file name argument. PyML supports two file formats for vector data:

- Delimited format (comma, tab, or space delimited).
- Sparse format.

Note that the non-sparse dataset container VectorDataSet, only supports the delimited, non-sparse format; the sparse containers handle both formats. If you need to convert a dataset from one format to another, read it into a sparse dataset container, and use its save method, specifying the file format.

Sparse file format

The sparse file format supported by PyML is similar to that used by LIBSVM (LIBSVM-format is recognized by PyML). Each pattern is represented by a line of the form:

```
[id,]label fid1:fval1 fid2:fval2 ....
```

or for unlabeled data:

```
[id,]fid1:fval1 fid2:fval2 ....
```

where:

- id - a pattern ID [optional].
- label - the class label associated with the pattern. Class labels can be arbitrary strings, not just -1/1; PyML converts the labels into an internal representation which is a number between 0 and the number of class - 1.

- data is provided as pairs fid:fval where fid is the feature ID and fval is its value; feature IDs can be integers or strings. A caveat for string based feature IDs – these are converted to integers using python's hash function. If the hash values of the feature IDs are not unique, an error will occur notifying you of a "non-unique hash". If that occurs convert your feature IDs into integers and the problem will go away. This usually happens only when the data contains a few hundred thousand features.

Using one of the example files in sparse format provided with the `PyML` distribution:

```
>>> data = SparseDataSet('heartSparse.data')
```

Typing the name of the variable gives some useful information about the data:

```
>>> data
<SparseDataSet instance>
number of patterns: 270
number of features: 13
class Label  /  Size
 +1 : 120
 -1 : 150
```

Other useful things: `len(data)` is the number of examples in a dataset. (Note that other `PyML` objects define a length function.) `data.numFeatures` is the number of features in the data. Containers of vector data also have a variety of other member functions such as `mean`, `std`, `scale`, `translate`, `eliminateFeatures` etc. Check the documentation for details.

Delimited files

Constructing a dataset from a delimited file is the same as for the sparse format, only now we have to give the parser some hints on which column contains the labels and which column contains the pattern IDs (if any). As in the sparse format, class labels can be arbitrary strings. For example:

```
>>> data = VectorDataSet('iris.data', labelsColumn = -1)
```

The labelsColumn keyword argument provides a hint to the parser which column of the file contains the labels. Column numbering follows the python array indexing convention: `labelsColumn = -1` denotes the last column, and counting starts at zero, i.e. `labelsColumn = 0` denotes the first column. If the file contains also pattern IDs, the `idColumn` keyword argument provides the column in which these are located. If the `labelsColumn` is equal to 1, then the parser assumes that the first column contains the ids. Labels and ids can only appear in certain columns. The following are the allowed values for them: (idColumn,labelsColumn) = (None,None), (None,0), (None,-1), (0,1), (0,-1). `None` means that a value is not provided.

A few notes:

- If the first non-comment line of a delimited file is non-numeric, its tokens are taken as feature IDs.
- `PyML` recognizes the type of file (sparse or delimited); if your file is not recognized correctly you can give a hint to the constructor which parser to invoke using the 'hint' keyword argument whose values can be 'sparse' or 'csv'.

Gzipped files

All dataset containers can read from gzipped files. You don't need to do anything – the parser automatically detects gzipped files and substitutes the default file handler with the python `gzip.GzipFile` handler. A file is detected as being a gzipped file if python's `gzip.GzipFile` handler can read from it.

5.2 Copy Construction

To make a copy of a dataset `data` of class `VectorDataSet`:

```
>>> data2 = VectorDataSet(data)
```

An even better way of doing this is:

```
>>> data2 = data.__class__(data)
```

This says "make me another copy of yourself", so that you don't even need to keep track of the class of the data object. This method of copy construction works for ANY `PyML` object, not just data containers.

Now going back to the iris data: suppose we only want to look at two out of the three classes then the following form of copy construction is used:

```
>>> data2 = data.__class__(data, classes = ['Iris-versicolor', 'Iris-virginica'])
```

The `classes` keyword gives a list of classes to be extracted. One can also give a list of patterns to copy in the form:

```
>>> data3 = data.__class__(data, patterns = listOfPatterns)
```

If you have a list of patterns you want to eliminate:

```
>>> data4 = data.__class__(data, patterns = misc.setminus(range(len(data)),
                                                            patternsToEliminate))
```

5.3 Constructing a dataset from an array

You can use `PyML` to create datasets on the fly from python lists or Numpy arrays. In this method of construction the input is a two dimensional array. Suppose this array is called `X`, then it is assumed that `X[i]` is pattern `i` in the dataset. Given `X` you can now create a dataset using:

```
>>> data = VectorDataSet(X)
```

Note that you can use `SparseDataSet` as well. If your dataset is labeled (a list of strings the length of your data matrix) you can add the labels by passing them using the 'L' keyword. You can also pass pattern IDs and feature IDs as well (by default the pattern and feature IDs are just running numbers).

```
>>> data = VectorDataSet(X, L = L, patternID = patternID, featureID = featureID)
```

5.4 Adding/modifying labels

If you want to add labels to a dataset or want to change a dataset's labels you can do so by using the dataset's `attachLabels` method that takes as an argument a `Labels` object. Suppose your labels are stored in a list `L` and the IDs are stored in a list called `ids`, proceed as follows:

```
>>> data.attachLabels(Labels(L, patternID = ids))
```

If you are dealing with a classification problem the elements of `L` should be strings. To convert a multi-class dataset into a two class dataset use the `oneAgainstRest` function of the datafunc module.

5.5 Using kernels

Many of the classifiers implemented in `PyML` are *kernel* classifiers in which classification is performed using a kernel function which measures the similarity of a pairs of patterns. A data container comes equipped with a kernel object (making it in effect a *feature space*). By default a linear kernel is attached to a container object. To change the kernel use the `attachKernel` method of the container. You can either construct a kernel object from the `ker` module, e.g. `k = ker.Polynomial(degree = 2)`, followed by `data.attachKernel(k)`, or alternatively do `data.attachKernel('poly', degree = 2)`. For the Gaussian kernel: `data.attachKernel('gaussian', gamma = 1)`. The keyword argument passed to `attachKernel` are passed to the kernel constructor. To view individual entries of the kernel matrix you can use its `eval` function as follows: `data.kernel.eval(data, 0, 0)` computes the 0,0 entry of the kernel matrix for a dataset `data`. In this command a kernel object is passed a data object since it is not aware of the dataset it was attached to.

The kernel matrix associated with a particular dataset (or rather with the feature space associated with the dataset by virtue of the kernel function attached to it) can be exracted by using the `getKernel` method of a dataset. This returns the kernel matrix as a two dimensional `Numpy` array. The kernel matrix can then be displayed using the `ker.showKernel` function (requires the matplotlib library).

5.6 Non-vector data

`PyML` supports several containers for non-vector data:

- A class for storing pre-computed kernels (`KernelData`). This class support kernels stored in tab/space/comma delimited format. Each row of the kernel is stored in a line in the file; pattern IDs appear in the first column, follwed by the corresponding kernel matrix row.
- A container for storing pairs of data objects (`PairDataSet`).

Usage example:

```
# construct the dataset:
>>> kdata = KernelData(kernelFile)
# construct a Labels object out of a file that contains the labels
# a labels file is a delimited file with two columns -- the first
# contains the pattern IDs, and the second the labels
>>> labels = Labels(labelsFile)
# attach the labels to the dataset:
>>> kdata.attachLabels(labels)
```

5.7 The aggregate container

In many applications you will be faced with heterogeneous data that is composed of several different types of features, where each type of feature would benefit from a different kernel. The `Aggregate` container is what you need in this case. To use this container, construct a dataset object for each set of features as appropriate. Suppose these are stored in a python list called `datas`. The aggregate is then constructed as:

```
>>> dataAggregate = Aggregate(datas)
```

It is assumed that each dataset in the list of datasets refers to the same set of examples in the same order. The kernel of the Aggregate object is the sum of the kernels of the individual datasets. In constructing the aggregate you can also set a weight for each dataset using the `weights` keyword argument of the `Aggregate` constructor. Also note that the `Aggregate` container works only with the C++ data containers.

6 Training and testing a classifier

All the classifiers in `PyML` offer the same interface:

- A constructor that offers both copy construction, and construction "from scratch".
- train(data) - train the classifier on the given dataset
- test(data)
- trainTest(data, trainingPatterns, testingPatters)
- cv(data)
- stratifiedCV(data)
- loo(data)

Most classifiers also implement a `classify(data, i)` and `decisionFunc(data, i)` that classify individual data points; these are not typically invoked by the user who is encouraged to use the "test" method instead, since it does some additional bookkeeping.

6.1 SVMs

Let's go back to the `'heart'` dataset and construct an SVM classifier for that problem:

```
>>> s=SVM()
>>> s
<SVM instance>
C : 10.000000
Cmode: classProb
trained: 0
```

Notes: If you would like to change the value of the parameter C, simply type:

```
>>> s.C = some_other_value
```

Or set C in the constructor:

```
>>> s = svm.SVM(C = someValue)
```

The `Cmode` attribute indicates how the C parameter is used; there are two modes: 'equal' - all classes get the same value of C, and 'classProb' [default] - the parameter C for each class is divided by the number of points of the other class, to handle datasets with unbalanced class distributions.

To train the svm use its train method:

```
>>> s.train(data)
```

By default the libsvm solver is used in training. To use the `PyML` SMO optimizer either set the `optimizer` attribute to `'mysmo'` or instantiate an svm instance as `svm.SVM(optimizer = 'mysmo')`. Note that for a non-vector dataset, the default libsvm optimizer cannot be used and the `PyML` native SMO implementation is automatically used instead (it is slower than libsvm so is not the default). To assess the performance of a classifier, use its `cv` (cross validation) method:

```
>>> r = s.cv(data, 5)
```

This performs 5 fold cross validation and stores the results in a `Results` object. An alternative way of specifying the number of folds is:

```
>>> r = s.cv(data, numFolds=5)
```

Stratified cross-validation `stratifiedCV` is a better choice when the data is unbalanced, since it samples according to the class size. There is also a leave-one-out method (`loo`). The `Results` object obtained by performing cross-validation stores information on classification accuracy in each of the folds, and averaged over the folds. Try printing the object to get an idea of what it provides. A detailed description of the `Results` object is found in Appendix B.

NOTE: As of version 0.6.9, SVM training is supported only for the C++ containers.

Saving results and models

To save a results object use its `save` method. This saves the object using python's pickle module. To load results import the `loadResults` function as `from PyML.evaluators.assess import loadResults`. Note that the save method of a `Results` first converts the object into an object whose representation will remain constant between versions, so that result object will continue to be readable across versions.

The model obtained after training an SVM can be saved for future use (saving of trained classifier is only available for SVM classifiers):

```
>>> s.train(data, saveSpace = False)
>>> s.save(fileName)
>>> from PyML.classifiers.svm import loadSVM
>>> loadedSVM = loadSVM(fileName)
>>> r = loadedSVM.test(testData)
```

Note that you may need to set the 'datasetClass' keyword of the loaded SVM instance since the data format used by the SVM to store its data need to agree with the format of your test data. See the API documentation for details. The 'saveSpace' keyword need to be set so that information needed for saving the object is kept.

Using kernels

As mentioned above, a dataset comes equipped with a kernel, so the `SVM` object knows what type of kernel to use in training the classifier. To override the kernel attached to a dataset define a kernel object and instantiate an SVM object that uses that kernel:

```
>>> k = ker.Polynomial()
>>> s = SVM(k)
```

Alternatively, attach a different kernel to the dataset:

```
>>> data.attachKernel('polynomial')
```

This attaches a polynomial kernel (default degree = 2) to the dataset (attachKernel also accepts a kernel object), and

```
>>> r = s.cv(data)
```

performs CV on an SVM with a polynomial kernel.

Linear SVMs have functionality not found in the nonlinear SVM, namely explicit computation of the weight vector; this results in more efficient classification, and is also used for feature selection (see the RFE class in the feature selection module).

6.2 SVM regression

Reading data for a regression problem is different than for classification: you need to tell the parser to interpret the labels as numbers rather than class labels. The file formats are the same, simply replace class label by the numerical value you want to predict. To read data use e.g.:

```
>>> data = SparseDataSet(fileName, numericLabels = True)
```

Now construct a Support Vector Regression (SVR) object:

```
>>> from PyML.classifiers.svm import SVR
>>> s = SVR()
```

This object supports the standard interface; the classify function return the predicted value, performing the same function as the decisionFunc method. The result of any of the testing methods (cv, test etc.) contains attributes similar to the `Results` object used for classification problems, and contains the attributes `Y` - the predicted values, `givenY` - the given values, and `patternID` - the IDs of the patterns that were tested.

Note that SVR has an additional parameter – eps (epsilon). See any standard SVM reference for an explanation of the epsilon insensitive loss-function.

7 Other classifiers in PyML (the classifiers module)

Additional classifiers include:

- k-nearest neighbor classifier (class `KNN`).
- Ridge Regression classifier (class `RidgeRegression`).

Both the KNN classifier and the Ridge Regression are implemented as kernel methods.

8 Multi-class classification (the multi module)

Multi-class classifiers are found in the multi module. `PyML` supports one-against-one and one-against-the-rest classification.

To construct a one-against-the-rest classifier that uses a linear SVM as a base classifier:

```
>>> from PyML.classfiers import multi
>>> mc = multi.OneAgainstRest (SVM())
```

To assess the performance of the classifier proceed as usual:

```
>>> r = mc.cv(data)
```

A one-against-one classifier is provided by the class `OneAgainstOne`.

9 Model selection (the modelSelection module)

Selecting classifier parameters is performed using the `ModelSelector` class in the modelSelection module. In order for the `ModelSelector` object to know which sets of parameters it needs to consider it needs to be supplied with a `Param` class object. The `Param` object specifies both a classifier, the parameter that needs to be selected and a list of values to consider for that parameter:

```
>>> param = modelSelection.Param(svm.SVM(), 'C', [0.1, 1, 10, 100, 1000])
```

The `Param` object is now supplied to a `ModelSelector`:

```
>>> m = modelSelection.ModelSelector(param)
```

The ModelSelector class implements the standard classifier interface, and

```
>>> m.train(data)
```

performs cross-validation for each value of the parameter defined in the `Param` instance, and chooses the value of the parameter that gives the highest success-rate. It then trains a classifier using the best parameter choice. You can also create a `ModelSelector` that optimizes a different measure of accuracy, for example the ROC score:

```
>>> m = modelSelection.ModelSelector(param, measure = 'roc')
```

To perform a grid search for a two-parameter classifier (e.g. SVM with a Gaussian kernel), use the `ParamGrid` object. This generates a grid of parameter values using the values supplied by the user:

```
>>> param = modelSelection.ParamGrid(svm.SVM(), 'C', [0.1, 1, 10, 100, 1000],
                                     'kernel.gamma', [0.01, 0.1, 1, 10])
```

The `ParamGrid` object is then supplied to a `ModelSelector` as before. A more efficient method specifically for SVMs is the `SVMselect` class that performs a more efficient search by first searching for an optimal value of the width of the Gaussian kernel, using a relatively low value of the soft-margin constant, and then optimizing the soft-margin constant once the width parameter is chosen.

10 Feature selection

All feature selection methods offer a `select(data)` method that applies the feature selection criterion to the dataset, and selects a subset of features according to the setting of the feature selection object.

The feature selection methods offered by `PyML` are:

- Recursive Feature Elimination (RFE) (class `RFE`)
- Filter methods (class `Filter`)
- Random feature selection (class `Random`)

For example, to use RFE:

```
>>> rfe = featsel.RFE()
>>> rfe
<RFE instance>
mode: byFraction
Fraction to eliminate each iteration : 0.050000
target number of features : 20
automatic selection of the number of features : 1
```

At each iteration RFE trains an SVM and removes features with the smallest components of the vector w. Either a given fraction of the features are removed `(rfe.fractionToEliminate)` in the `'byFraction'` mode, or a given number (`'byNumber'` mode). `rfe.targetNumFeatures` is the number of features at which to stop the elimination process. If automatic selection of the number of features is chosen (default behavior) then the number of features is chosen as the smallest number of features that minimizes the number of support vectors (the number of features tested is limited from below by the variable targetNumFeatures).

To assess the performance of a feature selection method one should do as follows:

- divide the dataset set into a training set and a test set
- perform feature selection on the training set, and train a classifier on that dataset
- project the features of the test set on the chosen features, and test the classifier

This process is automated by using the `FeatureSelect` classifier template:

```
>>> from PyML.classifiers.composite import FeatureSelect
>>> featureSelector = FeatureSelect(svm.SVM(), featsel.RFE())
```

FeatureSelect(classifier, featureSelector) is a classifier that is trained using the classifier's `train` method after applying the featureSelector's `select` method to the data. Note that training a `FeatureSelect` object affects the data on which it was trained: It will now contain only the selected set of features, and looking at its `featureID` attribute, you can see which features were selected. Note however, that cross-validation does not affect the data: for each fold of cross-validation a fresh copy of the data is used. The `log` property of a `Results` object provides among other things the number of features selected.

There is also a very simple simple feature selection class based on a feature scoring function (a filter method) – featsel.Filter. This class's `select` method applies a feature scoring function to the data to obtain a ranking of the features and keeps the numFeatures highest scoring features.

Let's see how this works:

```
>>> score = featsel.FeatureScore('golub')
>>> filter = featsel.Filter(score)
>>> filter
<Filter instance>
mode: byNum
number of features to keep : 100
<FeatureScore instance>
score name : golub
mode : oneAgainstOne
```

The filter has several modes of operation:

- 'byNum' - the user specifies the number of features to keep. To change the number of features to keep, modify the attribute numFeatures.
- 'byThreshold' - the user specifies the threshold below which a feature is discarded.
- 'bySignificance' - the score for a feature is compared to the score obtained using random labels. Only features that score a number of standard deviations above the average value are retained (set the variable 'sigma' to control the number of standard deviations).

The filter constructed above is composed of a `FeatureScore` of type 'golub' which scores a feature by the difference in the means of that feature between classes, weighted by the standard deviation (see code for how this works for a multi-class problem).

To use with a classifier define a featureSelector:

```
>>> featureSelector = FeatureSelect(classifier, filter)
```

Note that since `featsel.Filter` and `featsel.RFE` both have the same interface, both can be used in the same way in conjunction with the FeatureSelect classifier.

11 The Chain Classifier Object

When designing a classifier one often needs to do a series of operations before training a classifier – preprocessing, feature selection etc. When performing CV, such operations need to be part of the training procedure. Instead of coding a class whose train method does a series of operations, you can use the `Chain` object. For example, instead of using the FeatureSelect classifier:

```
>>> from PyML.classifiers.composite import Chain
>>> chain = Chain([featureSelector, classifier])
```

The constructor takes a list of classes, the last of them being a classifier. Each member of the chain needs to implement a `train` and `test` method. The command

```
>>> chain.train(data)
```

performs `featureSelector.train(data)` (which invokes the feature selector's `select` method), followed by `classifier.train(data)`, sequentially performing the actions in the chain. Note that by default a `Chain` classifier uses deepcopy copying of data, as does the `FeatureSelect` object.

12 Preprocessing and Normalization

There are many ways to normalize a kernel/dataset. We differentiate between two major types of normalization methods:

- Normalize the features or the kernel such that $k(x, x') = 1$, i.e. feature vectors have unit length.
- Normalize the features such that each feature has magnitude $O(1)$.

First we begin by considering normalizing data such that inputs are unit vectors. If your data is explicitly represented as vectors you can directly normalize your data vectors to be unit vectors by using the `normalize` method of a dataset container: `data.normalize(norm)`, where norm is either 1 or 2 for the L1 or L2 norms (this operation divides each vector by its L1 or L2 norm). If possible, use this method of normalizing your data. When your data is not in explicit vector form, normalization can be performed at the level of the kernel function, i.e. in feature space. Given a kernel $k(x, x')$, `cosine` normalization computes the kernel

$$k_{cosine}(x, x') = \frac{k(x, x')}{\sqrt{k(x, x)k(x', x')}}. \tag{1}$$

In `PyML` this is achieved by attaching to your data a kernel with the cosine form of normalization associated with it. For example:

```
>>> data.attachKernel('polynomial', degree = 3, normalization = 'cosine')
```

`PyML` recognizes two additional forms of feature space normalization similar to the cosine kernel—Tanimoto normalization (named after the Tanimoto coefficient), which is given by:

$$k_{tanimoto}(x, x') = \frac{k(x, x')}{k(x, x) + k(x', x') - k(x, x')} \tag{2}$$

and a form which is called after Dice's coefficient:

$$k_{dices}(x, x') = \frac{2k(x, x')}{k(x, x) + k(x', x')} \tag{3}$$

These can be used e.g. setting the `normalization` keyword to `'tanimoto'` or `'dices'`. These should give very similar results to cosine normalization.

`PyML` implements a method for normalizing each feature separately, namely standardization, i.e. for each feature, subtracting the mean and dividing by the standard deviation). Standardization is implemented by the class `preproc.Rescale`. For example:

```
p = Standardizer()
p.train(data)
```

Standardization destroys the sparsity of sparse data. Therefore this operation is not recommended for such data; moreoever, do not use it on `SparseDataSet` objects, since the result is not the one you would expect.

13 Visualizing classifier decision surface

`PyML.demo.demo2d` is a module for visualizing the decision surface of a classifier. SVMs and other classifiers base their classification on what's called a *decision function* or *discriminant function*. For the SVM and ridge regression classifier, the classification is determined by the sign of this function. To get an intuition of how the decision function behaves as a function of classifier and kernel parameters you can use the `demo2d` module. To use it proceed as follows:

```
>>> from PyML.demo import demo2d
# first create a dataset by following the instructions onscreen:
>>> demo2d.getData()
# decision surface of an SVM with Gaussian kernel:
>>> demo2d.decisionSurface(svm.SVM(ker.Gaussian(gamma = 0.5)))
```

A Practical notes on SVM usage

The SVM is not converging — what can I do?

The default kernel attached to a dataset object is the linear kernel. In some cases SMO-type SVM training algorithms do not converge when using a linear kernel, unless the data is first normalized, or a non-linear kernel is used. When the SVM does not converge even for nonlinear kernels / normalized data consider using a lower value for the SVM soft-margin constant (C): a lower value of C makes the problem easier because outliers can be more easily ignored. When all fails, try the 'gradientDescent' or 'gist' solvers — they are much slower, but may provide a result when SMO algorithms fail.

B Results objects

`PyML` stores the results of using a classifier's `test` or various flavors of cross-validation in a `Results` object. The type of object returned depends on the type of learner: it's a `RegressionResults` for regression and a `ClassificationResults` for a classifier. A `Results` object is generated either by cross-validation

```
>>> r = classifier.cv(data)
```

or by testing a trained classifier:

```
>>> classifier.train(trainingData)
>>> r = classifier.test(testingData)
```

In the following we focus on the `ClassificationResults` object, as it is more developed, but the ideas are applicable to `RegressionResults` as well. The `Results` objects are `lists`, where each element of the list groups the results on a chunk of data. When the object is the result of cross-validation, the number of elements in the list is equal to the number of cross-validation folds. When it is the result of using a classifier's `test` function, it contains a single element. The results in fold i of a results object `r` are accessed as `r[i]`. The `ClassificationResults` object has a set of accessor functions that allow the user to obtain detailed information about the results of testing the classifier. For example `r.getSuccessRate()` returns the average success rate over the cross-validation folds, while `r.getSuccessRate(0)` returns the success rate in the first fold (folds are indexed from 0 to numFolds - 1). All the accessor function follow the same interface: `getAttribute(fold = None)`, where `getAttribute` is one of the above. The 'fold' parameter is the cross-validation fold that you want to query. If it is not specified then the attribute is "pooled" over the different folds. In the case of statistics that measure the success of the classifier, "pooling" means an average of the results in the different folds. When the attribute is say the list of predicted classes for a given fold, then these lists are aggregated into a list whose length equals the number of fold, and each element contains the list of results pertinent to a particular fold.

A complete list of accessor functions follows. First is a list of accessor functions for statistics that summarize classifier performance:

- `getSuccessRate`, `balancedSuccessRate`, `getSensitivity`, `getPPV`. The success rate is the fraction of correctly classified examples; the balanced success rate takes into account the size of each class – useful for unbalanced datasets. sensitivity is the fraction of examples from the positive class that are correctly classified (number of correct positive predictions over the number of examples in the positive class); the ppv is the fraction of correct predictions made (number of correct positive predictions over number of positive predictions).
- The area under ROC curve is accessed through the `getROC` accessor function, or simply as `r.roc`. In the case of cross-validation results this is an average over the different folds. `res[0].roc` returns the ROC score for the first fold. The area under the ROC50 curve is accessed through `getROCn` that also accepts an optional parameter that specifies a value different than 50 (`getROCn(rocN, fold)`, and by default rocN is 50). You can also type something like `r.roc10`. If you want to specify the fraction of false positives rather than their number, use `getROCn('1%')` or `getROCn(0.01)`.
- The confusion matrix (`getConfusionMatrix`) – element i, j is the number of patterns in class j that were classified into class i (in other words, column j in the matrix describes the breakdown of how members of class j were classified). The string label of class j is accessed through `getClassLabels`. When no fold is specified, the confusion matrices of individual folds are summed.

Additional accessor functions are:

- `getPatternID` returns a list of the pattern IDs of the classified examples.
- `getPredictedLabels` returns a list of predicted (string) labels. The order matches that of the patternIDs, so that `getPredictedLabels(fold)[i]` is the label of `getPatternID(fold)[i]`.

- `getGivenLabels` returns the labels provided by the user.
- `getPredictedClass`, `getGivenClass` provide class IDs rather than their string names. These are lists of numbers between in the range [0:numClasses] (note that in python notation does not include the last index).
- `getDecisionFunction` — the decision function values produced by the classifier.
- `getInfo` — a description of the dataset and classifier used in each fold.
- `getLog` returns information about the training/testing process. Information includes training and testing time; each classifier may include different information—an SVM classifier for example provides the number of support vectors.

If you have `matplotlib` installed on your system, the ROC curve can be displayed by `r.plotROC()`. If you want to plot an ROC50 curve use `r.plotROC(rocN = 50)`, and `r.plotROC('roc.eps')` saves the ROC curve to an eps file. See the documentation for more details.

In addition to the accessor functions shared with `ClassificationResults`, the `RegressionResults` object has a `getRMSE` accessor function.

Python note: to determine the attributes of an object `o` type `dir(o)`, or if you have tab completion enabled, just type `o.` and use the tab to obtain the list of attributes.

C Registered Attributes

In some cases one may want to associate some additional information with a dataset. You can always do something like:

```
data.attr = someObject
```

However, under copy construction, e.g. `data2 = data.__class__(data, patterns = ...)`, that attribute does not get copied. In order for the attribute to be copied in copy construction you need to "register" it. Supposed you have an attribute called 'auxiliaryData' that you want to attach to a dataset, then proceed as follows:

```
# read a dataset:
>>> data = ...
# creat the auxiliary data
>>> auxiliaryData = ...
>>> data.registerAttribute('auxiliaryData', auxiliaryData)
```

An alternative form is:

```
>>> data.auxiliaryData = auxiliaryData
>>> data.registerAttribute('auxiliaryData', auxiliaryData)
```

The semantics of copying the auxiliary data under copy construction is as follows: if the attribute is a list or a dataset whose length matches the length of the dataset, it is assumed that pattern i matches pattern i in the auxiliary dataset or element i of the list. Under copy construction the appropriate elements of the dataset/list are copied. Otherwise, the copied dataset will simply contain a new reference to the auxiliary data.

An example where you may want to use registered attributes is in setting a value of the SVM C parameter on a pattern by pattern basis. Given a list ov values `Clist` the syntax for using them with an SVM classifier is:

```
>>> data.registerAttribute('C', Clist)
>>> s = svm.SVM(optimizer = 'mysmo', Cmode = 'fromData')
```

Note that you need to use the ’mysmo’ optimizer since libsvm does not support setting individual C values.