# A First Tutorial on the MLX schema

Written by Lucas Scharenbroich
JPL Machine Learning Systems Group
Copyright 2002, California Institute of Technology.
ALL RIGHTS RESERVED. U.S. Government Sponsorship acknowledged

August 27, 2003

## Contents

Welcome to PyMLX! This is the first in a series of tutorials which will allow you to become familiar with the MLX (Machine Learning Infrastructure) schema developed jointly by Caltech/JPL. The MLX schema combined with the python interpreter provides a powerful and simple environment for loading, clustering and analyzing datasets.

# 1 Datasets, Views, and Labelings. . . Oh My!

This first section of this tutorial is intended to provide familiarity with the core PyMLX classes and how they are used to load, label, and view data. Further sections will address the topics of clustering and analyzing data. A separate tutorial is available which deals specifically with analyzing microarray data.

## 1.1 Datasets

The Dataset class is the fundamental class upon which all others are built. Many classes inherit from Dataset and extend it's abilities, while other, peer classes hook into the functionality of the Dataset. However, while a Dataset instance may exist independently, these other classes may not. If you have correctly set up your python environment as described in the preliminaries tutorial, type `python` to enter the interpreter. Once you are at the python prompt (`>>>` ), type the following command to import the Dataset class.

```
>>> from compClust.mlx.datasets import *
```

If you see the error `ImportError:`
`No module named compClust.mlx.datasets`, be sure that you have correctly set the `PYTHONPATH` environment variable. If warnings about shadowed variables apprear, make sure that your are running pyhthon2.2 or higher. The PyMLX codebase depends of the `nested_scopes` extension introduced in the 2.2 series. If there is no error you should be back at the prompt.

### 1.1.1 The first step

Now let's create our first dataset object. Simple type the following at the command prompt

```
>>> ds = Dataset([[1,1,1],[2,2,2],[3,3,3]])
```

You now have an instance of a Dataset in the variable `ds`. To check this, just type `ds` at the command prompt. You should see the following printed out

```
>>> ds
Dataset: None, 3 by 3
```

This tells you that the variable `ds` is an instance of class Dataset, it has no name (name is equal to `None`) and the size of the dataset is 3 rows by 3 columns. To view the data in the dataset, we use the getData() method. Try it now

```
>>> ds.getData()
[[ 1., 1., 1.,]
 [ 2., 2., 2.,]
 [ 3., 3., 3.,]]
```

As you can see, this is the same data which we entered when constructing the object.

### 1.1.2 Naming a Dataset

A plain, old dataset object is good, but it might be nice to give it a name that is a little more descriptive and just the variable name `ds`. To manipulate the name of a dataset we use a pair of methods – getName() and setName(). Try the following sequence of commands.

```
>>> ds.getName()
>>> print ds.getName()
None
>>> ds.setName('My First Dataset')
>>> ds.getName()
'My First Dataset'
>>> print ds.getName()
My First Dataset
```

Notice that unless we specified `print`, no name was printed after executing getName(). This is because the python `print` command prints a *string representation* of a python variable. By itself, `None` has nothing to show, but its string representation is the literal string 'None'. Compare this to the output of getName() after setting the object's name to `'My First Dataset'`. Can you see the parallel?

### 1.1.3 Retrieving data

Now, as you have seen, one is able to retrieve the full dataset via the getData() method, but what if your dataset contains many thousands of rows and you are only interested in one of them? It seems to be a waste to have to sift through a full dataset just to get a single entry. In fact, there are two methods which provide row and column access to the dataset. They are getRowData() and getColData(). These methods take a integer argument and returns the corresponding row or column as a 1-D vector. Note that the indices start at zero. If a row or column is too large, a ValueError() is raised. Let's try an example:

```
>>> ds.getRowData(0)
[ 1., 1., 1.,]
>>> ds.getRowData(1)
[ 2., 2., 2.,]
>>> ds.getRowData(2)
[ 3., 3., 3.,]
>>> ds.getRowData(3)
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
  File "/home/lscharen/checkout/code/python/compClust/mlx/
        Dataset.py", line 219,
  in getRowData
    raise ValueError()
ValueError
```

An interesting consequence of getRowData() raising an excepting, is that you can write a function which prints out each row of data without ever having to know how many rows of data there are! Consider the following function and its output:

```
>>> def foo(dataset):
       i = 0
       while (1):
          try:
            print str(i+1) + " " + str(dataset.getRowData(i))
            i = i + 1
          except:
             break
```

```
>>> foo(ds)
1 [ 1., 1., 1.,]
2 [ 2., 2., 2.,]
3 [ 3., 3., 3.,]
```

However, there is an easy way to determine how many rows and columns are in a Dataset object, and that is by using the getNumRows() and getNumCols() methods. For our example object, it should have three rows and three columns.

```
>>> ds.getNumRows()
3
>>> ds.getNumCols()
3
```

### 1.1.4 Outputting the data

At this point, there is really only one more method worth mentioning – the writeDataset() method which pretty-prints the dataset to a given output stream. By default, writeDataset() writes to `stdout` with a tab delimiter, but other combinations are possible. The example code below shows several variations, including dumping a dataset to a file which is readable by any program which can read tab-delimited text files (which includes Matlab and Excel).

```
>>> # need the os and sys package
... import os
... import sys
>>> ds.writeDataset()
1.0     1.0     1.0
2.0     2.0     2.0
3.0     3.0     3.0
>>> ds.writeDataset(sys.stderr)
1.0     1.0     1.0
2.0     2.0     2.0
3.0     3.0     3.0
>>> ds.writeDataset(delimiter='*')
1.0*1.0*1.0
2.0*2.0*2.0
3.0*3.0*3.0
>>> ds.writeDataset(sys.stderr, '*')
```

```
1.0*1.0*1.0
2.0*2.0*2.0
3.0*3.0*3.0
>>> stream = open('ds-dump.txt','w')
>>> ds.writeDataset(stream)
>>> stream.close()
>>> result = os.system('cat ds-dump.txt')
1.0     1.0     1.0
2.0     2.0     2.0
3.0     3.0     3.0
```

The last topic to touch on before moving on to the View section is to discuss how the Dataset constructor handles multiple data types as initializers.

The Dataset constructor can take in the following object types:

**StringType** Assumes the the string represents a tab-delimited filename and attempts to read the data from said file.

**FileType** A FileType represents an open file. The Dataset constructor will attempt to read from this file.

**ListType** The list is converted to a Numeric array.

**TupleType** The tuples is converted to a Numeric array.

**ArrayType** The array is used as is

**Dataset instance** A new object is created which references the existing dataset. This does **not** make a copy of the data.

Also, it is usefule to know that is the object type is a StringType, it may be a valid URL as well. This allows PyMLX to load dataset directly from the web. The file loader will also transparently handle Gzip compressed files is the file name ends in .gz.

The constructor attempts to be smart about loading files. A text input file may have optional string labels in the first column of the dataset. If the first column is not a valid number, then the loader assumes that the first column contains textual labels and will build the dataset from column two onward.

The implication of this is that if your dataset contains numerical labels in the first column, these must be dealt with in your own code as a special case.

And now, on to View ...

## 1.2 Views

View are a very powerful class in the PyMLX hierarchy and are probably the most used class when interactively working with the schema or doing data analysis. A View itself does just as its name implied – it produces a view of existing data. This view is abstract in the sense that the contents of a view may not look the same as its parent by inspection, by they are related in some functional way.

Since View inherit from the Dataset class, they themselves are dataset, and, thus, one may have View of View. All of the methods available to Dataset objects are also available to View. For simplicity, you may substitute a View for a Dataset object anywhere you see Dataset mentioned in this tutorial, unless specifically told otherwise.

Let's take a look at the many views which are available. and try some examples with each of them. As a preliminary step, at the python prompt execute the following command to get the View classes in your name space.

```
>>> from compClust.mlx.views import *
```

Now we're ready to proceed.

### 1.2.1 FunctionView

The FunctionView allows one to filter a dataset by unary function. These unary functions take in a single real number and produce a single real number. One can use many of the functions in the python math module or define your own via `lambda` functions.

For our first example, let's take our dataset and create a FunctionView which log-transforms the data. To do this we will need the `log` function which lives in the python math module. So, we will need to import that module first.

```
>>> import math
```

Now let's test to see that the log function is there and working correctly.

```
>>> math.log(1)
0.0
```

Looks good! Now we can use this function to build our FunctionView.

```
>>> func_view = FunctionView(ds, math.log)
```

And that's all that needs to be done. In general, creating any view is a one-line operation. Let's inspect this function view and use some of the methods from the Dataset class on it as well to make sure it behaves as expected.

```
>>> func_view.setName('Log-Transformed Data')
>>> func_view.getName()
'Log-Transformed Data'
>>> func_view
FunctionView: Log-Transformed Data, 3 by 3
>>> func_view.writeDataset()
0.0     0.0     0.0
0.69314718056   0.69314718056   0.69314718056
1.09861228867   1.09861228867   1.09861228867
>>> func_view.getNumRows()
3
>>> func_view.getNumCols()
3
>>> func_view.getRowData(1)
[ 0.69314718, 0.69314718, 0.69314718,]
>>> func_view.getColData(1)
[ 0.        , 0.69314718, 1.09861229,]
```

Everything looks to be working as it should, so let's try something a bit different. The log-transformed view we create takes the natural log of the values, but what if we wanted the base-2 log of the data. This is simple to do with a lambda function.

Remember the basic identity that $\log_a b = \frac{\ln b}{\ln a}$. If we create a lambda function which does that and pass that in as the argument to the FunctionView constructor, we will have our log-base-2 view. Let's try.

```
>>> log2_view = FunctionView(ds, lambda x : math.log(x) /
... math.log(2))
>>> log2_view.getData()
[[ 0.       , 0.       , 0.       ,]
 [ 1.       , 1.       , 1.       ,]
 [ 1.5849625, 1.5849625, 1.5849625,]]
```

Perfect. Now, for one last example, let's attach another FunctionView to the first one we created. This time, we'll use the exponential unary function and see if we can get our original dataset back.

```
>>> exp_view = FunctionView(func_view, math.exp)
>>> exp_view.getData()
[[ 1., 1., 1.,]
 [ 2., 2., 2.,]
 [ 3., 3., 3.,]]
>>> exp_view.getData() == ds.getData()
[[1,1,1,]
 [1,1,1,]
 [0,0,0,]]
```

As you can see, even though it appears that we reconstructed the original dataset, when comparing by equality, the third row is not exactly equal. This is not a problem specific to python or the MLX schema, but a generic problem inherent with floating point numbers used on any computer system. It is good to be reminded of the limitations of floating point from time to time.

The last thing we need to do before moving on is remove the views we have created. Python uses a reference counting scheme for its garbage collection and the internal structure of the schema is complicated enough that circular references are introduced. This means that the links between objects must be explicitly broken for them to be reclaimed by the garbage collection. Fortunately, the Dataset class, and by extension, the View classes, offer a removeView() method which takes in an instance of a View object and removes it. The View passed in needs to be a child of the view upon which the method is invoked. With this knowledge, let's clean up our views.

```
>>> func_view.getViews()
[FunctionView: None, 3 by 3]
>>> func_view.removeView(exp_view)
>>> func_view.getViews()
[]
>>> ds.getViews()
[FunctionView: Log-Transformed Data, 3 by 3,
 FunctionView: None, 3 by 3]
>>> ds.removeView(func_view)
>>> ds.removeView(log2_view)
>>> ds.getViews()
[]
```

Notice, we've also used the getViews() method. This returns a list of views currently associated with a dataset. Related to this is the getView()

method which can retrieve a view by name. We'll look at this method in the next section.

### 1.2.2 RowFunctionView

The RowFunctionView is analogous to the FunctionView, except that it operates on a full row of data at a time. The function passed into the construction must be capable of taking in a dataset object along with a row number and returning a vector equal in length to the number of columns in the view.

For our first example, we create a function which normalizes the rows. For this we'll need the Numeric module. This can be imported via the following command.

```
>>> import Numeric
```

Now let's construct our view and give it a name. Notice how similar this is to the creating of the FunctionView described in the previous example.

```
>>> def row_norm(ds, row):
        x = ds.getRowData(row)
        return x / Numeric.sqrt(Numeric.sum(x*x))
>>> rfv = RowFunctionView(ds, row_norm)
>>> rfv.getData()
[[ 0.57735027, 0.57735027, 0.57735027,]
 [ 0.57735027, 0.57735027, 0.57735027,]
 [ 0.57735027, 0.57735027, 0.57735027,]]
>>> rfv.setName('Row Normalized')
>>> rfv
RowFunctionView: Row Normalized, 3 by 3
```

All the row vectors were normalized to the same vector, which is correct since they were scalar multiples of each other. Now, let's create another, more specialized RowFunctionView. This view will only operate on vectors in $\mathbb{R}^3$ and convert from Cartesian to Spherical coordinates. We'll also give it a name.

```
>>> def cart2sph(ds, row):
        x = ds.getRowData(row)
        rho = Numeric.sqrt(Numeric.sum(x*x))
        theta = math.atan2(x[1], x[0])
```

```
        phi = math.acos(x[2] / rho)
        return [rho, theta, phi]
>>> c2s_view = RowFunctionView(ds, cart2sph)
>>> c2s_view.getData()
[[ 1.73205081, 0.78539816, 0.95531662,]
 [ 3.46410162, 0.78539816, 0.95531662,]
 [ 5.19615242, 0.78539816, 0.95531662,]]
>>> c2s_view.setName('Cartesian to Spherical')
>>> c2s_view
RowFunctionView: Cartesian to Spherical, 3 by 3
```

Just by inspecting this view we can see that the original dataset differ only in their magnitudes. Of course, this is the same information gotten from the row normalization, but it does help to illustrate that there are multiple ways to extract the same information from a dataset.

Now, if we look at the original dataset, it has two views attached to it. In the previous section we removed the views by passing in a reference to the target view. However, if we have lost or overwritten the variable that contains the view instance, how can we properly remove it? As alluded to, we can find a view by name using the getView() method. So, removing our two views, even if we don't have a reference to them, is easy.

```
>>> ds.getViews()
[RowFunctionView: Row Normalized, 3 by 3,
 RowFunctionView: Cartesian to Spherical, 3 by 3]
>>> ds.removeView(ds.getView('Row Normalized'))
>>> ds.removeView(ds.getView('Cartesian to Spherical'))
>>> ds.getViews()
[]
```

### 1.2.3 ColumnFunctionView

The ColumnFunctionView operates identically to the RowFunctionView except on the columns of the dataset. Since our example dataset is square, try the same examples described in RowFunctionView, but substitute ColumnFunctionView where appropriate.

### 1.2.4 RowSubsetView

A SubsetView is an *extremely* useful view type which allows one to pull interesting data from a dataset and examine it independently. Although

this section only considers the RowSubsetView, the ColumnSubsetView is just as useful and operates identically.

For our example of using the RowSubsetView, we will construct a random matrix and create a subset which contains all the rows which have a magnitude above a certain threshold. An important aspect of using a SubsetView is the *key system*. Every row and column of a dataset has a unique key associated with it. When constructing a RowSubsetView, a list of row keys is passed in to indicate which rows are to be taken from the parent dataset. It should be noted that there is no limit on the length of the key list. It is perfectly acceptable to have a subset which is larger than the parent dataset. Obviously, in this case, some of the rows will be duplicates.

For this example we will need yet another module – the MLab module, which provides Matlab®-style functions. Execute the following to load these modules.

```
>>> import MLab
```

Now we'll build our random dataset and look for rows with a magnitude larger than 1.0.

```
>>> ds = Dataset(MLab.rand(10,2))
>>> data = ds.getData()
>>> sum_of_squares = Numeric.sum(data*data, 1)
>>> mag = Numeric.sqrt(sum_of_squares)
>>> large_mag = filter(lambda x : x > 1.0, mag)
>>> large_rows = map(lambda x : mag.tolist().index(x),
... large_mag)
```

Now that we have the list of rows which have magnitudes greater than 1.0, we can pass this list into the constructor of the RowSubsetView.

```
>>> large_subset = RowSubsetView(ds, large_keylist)
>>> large_subset
RowSubsetView: None, 2 by 2
>>> ds.getData()
[[ 0.04412353, 0.97403276,]
 [ 0.55236453, 0.0979613 ,]
 [ 0.19320844, 0.57923186,]
 [ 0.77776194, 0.4972173 ,]
 [ 0.76949269, 0.32971311,]
 [ 0.44192556, 0.12342816,]
```

```
 [ 0.35265034, 0.60821456,]
 [ 0.85184562, 0.27371284,]
 [ 0.54120302, 0.94661057,]
 [ 0.87526697, 0.54308629,]]
>>> large_subset.getData()
[[ 0.54120302, 0.94661057,]
 [ 0.87526697, 0.54308629,]]
```

So, for this particular random dataset, there are only two vectors with a magnitude larger than 1.0, which happen to be the last two vectors. For an example of replicating data in a SubsetView, let's create a view which triples the data above.

```
>>> larger_subset = RowSubsetView(ds, large_rows * 3)
>>> larger_subset.getData()
[[ 0.54120302, 0.94661057,]
 [ 0.87526697, 0.54308629,]
 [ 0.54120302, 0.94661057,]
 [ 0.87526697, 0.54308629,]
 [ 0.54120302, 0.94661057,]
 [ 0.87526697, 0.54308629,]]
```

Quick and easy. Now, before we move on, be sure to remove these two views from the original dataset. Refer to the previous sections if you are still unsure how to proceed.

### 1.2.5 ColumnSubsetView

The ColumnSubsetView behaves identically to the RowSubsetView except that it operates on columns, rather than rows. There is no difference in functionality, simply pass in a list of columns that you wish to place in the subset.

### 1.2.6 RowSupersetView

A SupersetView is the converse to a SubsetView. While the SubsetView reduces the size of a single dataset, the SupersetView combines datasets together to make one large, aggregate dataset. The RowSupersetView combines two datasets along their columns. That is, the rows of data in one dataset are appended to the rows of data in the other. A consequence of

this, is that the number of columns in both datasets must be equal. A dataset may be appended to itself.

For an example, we will concatenate two, square datasets.

```
>>> ds1 = Dataset(MLab.rand(2,2))
>>> ds2 = Dataset(MLab.rand(2,2))
>>> ds1.getData()
[[ 0.12192474, 0.89563274,]
 [ 0.3412022 , 0.25135848,]]
>>> ds2.getData()
[[ 0.12337618, 0.9021067 ,]
 [ 0.4221943 , 0.98156619,]]
>>> ss1 = RowSupersetView(ds1, ds2)
>>> ss1.getData()
[[ 0.12192474, 0.89563274,]
 [ 0.3412022 , 0.25135848,]
 [ 0.12337618, 0.9021067 ,]
 [ 0.4221943 , 0.98156619,]]
>>> ss1.getColData(1)
[ 0.89563274, 0.25135848, 0.9021067 , 0.98156619,]
```

### 1.2.7 ColumnSupersetView

Where the RowSupersetView concatenates along the columns, the ColumnSupersetView concatenates along the rows. There exists the analogous restriction that the datasets being merged must have the same number of rows.

Continuing with the previous example, let's combine the RowSupersetView with another dataset in a ColumnSupersetView.

```
>>> ds3 = Dataset(MLab.rand(4,1))
>>> ds3.getData()
[[ 0.4922913 ,]
 [ 0.44203174,]
 [ 0.75371045,]
 [ 0.12824887,]]
>>> ss2 = ColumnSupersetView(ss1, ds3)
>>> ss2.getData()
[[ 0.12192474, 0.89563274, 0.4922913 ,]
 [ 0.3412022 , 0.25135848, 0.44203174,]
```

```
 [ 0.12337618, 0.9021067 , 0.75371045,]
 [ 0.4221943 , 0.98156619, 0.12824887,]]
```

As you can see, the SupersetView provides a very convenient way to stitch together multiple, partial datasets and treat them as a contiguous whole. This is especially useful when a dataset has been split into multiple subsets and only 'interesting' subsets, which satisfy some criteria, are of further interest. The relevant subsets may be superset together to synthesize the desired dataset.

### 1.2.8 TransposeView

The TransposeView simply provides a way to view your data transposed without having to change the original dataset. This is most helpful for interfacing to other python modules or external programs that require row or column major ordering for their inputs. As this class is conceptually simple, only a brief example is offered.

```
>>> ds = Dataset(MLab.rand(4,2))
>>> ds.writeDataset()
0.94551807642   0.31146222353
0.323919504881  0.931670665741
0.728558063507  0.743269145489
0.614804148674  0.751430094242
>>> tv = TransposeView(ds)
>>> tv.writeDataset()
0.94551807642   0.323919504881   0.728558063507   0.614804148674
0.31146222353   0.931670665741   0.743269145489   0.751430094242
```

### 1.2.9 TransformView

The TransformView allows one to transform a dataset via an arbitrary transformation matrix. The usual rules of matrix algebra apply along with one other restriction – the matrix must not produce more columns than the parent dataset. This effectively restricts the matrix which can be used to an $NxN$ matrix where $N$ is the number of columns of the parent dataset.

For an example, let's create a view which transforms a 2D dataset from a standard orthogonal basis to the orthogonal basis $v_1 = \begin{pmatrix} 1 \\ 1 \end{pmatrix}$ $v_2 = \begin{pmatrix} 1 \\ -1 \end{pmatrix}$. The matrix we'll use is $m = \begin{pmatrix} \frac{1}{2} & \frac{1}{2} \\ \frac{1}{2} & -\frac{1}{2} \end{pmatrix}$.

```
>>> ds = Dataset(MLab.rand(4,2))
>>> tv = TransformView(ds, Numeric.array([[0.5, 0.5],
... [0.5, -0.5]]))
>>> ds.getData()
[[ 0.78208232, 0.64115632,]
 [ 0.6568898 , 0.91970539,]
 [ 0.19737017, 0.42274952,]
 [ 0.32603681, 0.25608519,]]
>>> tv.getData()
[[ 0.71161932, 0.070463  ,]
 [ 0.78829759,-0.1314078 ,]
 [ 0.31005985,-0.11268967,]
 [ 0.291061  , 0.03497581,]]
```

### 1.2.10 ColumnPCAView

The ColumnPCAView operates on the column space (rows) of a dataset and transforms the columns via a Principal Component Analysis (PCA) matrix. In PCA space each axis corresponds to one of the eigenvector in the covariance matrix of the dataset. These eigenvectors are sorted by eigenvalue (largest first), so the first dimension captures the most variance, the second dimension the second-most, etc. The ColumnPCAView is most useful for visualization and pre-processing.

The best way to illustrate the usefulness of the ColumnPCAView is to create a dataset with high linear dependence. In this case, the PCA projection will project the majority of the data into a single dimension. To try this, let's create a very simple 2D dataset.

```
>>> ds = Dataset([[1,1],[2,1.9],[2.9,3],[4,4],[4.95,5.02]])
>>> ds.getData()
[[ 1.  , 1.  ,]
 [ 2.  , 1.9 ,]
 [ 2.9 , 3.  ,]
 [ 4.  , 4.  ,]
 [ 4.95, 5.02,]]
>>> pv = ColumnPCAView(ds)
>>> pv.getData()
[[-1.41411205,-0.01694448,]
 [-2.75667127,-0.10374734,]
 [-4.17247777, 0.02071938,]
```

```
[-5.65644819,-0.06777792,]
[-7.04994162,-0.03497432,]]
```

Notice that the magnitude of the values in the second column are much smaller than the values in the first column. Also, the values in the first column are almost equal to the magnitude of each point, which is its distance along the line $y = x$.

### 1.2.11 RowPCAView

The RowPCAView in analogous to the ColumnPCAView except that it operates on the row-space (columns) of the dataset.

### 1.2.12 SortedView

A SortedView provides a mechanism to dynamically alter the order in which data appears. To this end, the SortedView provides the following additional methods:

**permuteCols(plist)** Sets the ordering of the columns to an arbitrary permutation relative to the order of the parent dataset.

**permuteRows(plist)** Sets the ordering of the rows to an arbitrary permutation relative to the order of the parent dataset.

**sortColsByFunction(func)** Sorts the columns in order according to the results of the function. This is not a comparator, but a function which maps a column vector of data to a single scalar value suitable for sorting. The vector may be mapped to a string, integer, real, tuple, or list.

**sortRowsByFunction(func)** Sorts the rows of the view in a manner identical to sortColsByFunction().

**sort(func)** Left in for backwards-compatibility. Calls sortRowsByFunction().

**reset()** Resets the view to its original ordering.

Let's try a few simple examples of sorting a 3x3 random dataset. The first example sorts the rows in 'natural order'. That is, comparing on the first column, then the second, and finally the third. The second example sorts the rows only on the second column. The final example sorts the columns

based on their magnitude, and finally the view is reset to its original state and the rows permuted once.

```
>>> ds = Dataset(MLab.rand(3,3))
>>> sv = SortedView(ds)
>>> sv.getData()
[[ 0.70310301, 0.61336708, 0.29631215,]
 [ 0.942316  , 0.32991922, 0.58734894,]
 [ 0.01950742, 0.57061231, 0.17892206,]]
>>> sv.sortRowsByFunction(lambda x : x.tolist())
>>> sv.getData()
[[ 0.01950742, 0.57061231, 0.17892206,]
 [ 0.70310301, 0.61336708, 0.29631215,]
 [ 0.942316  , 0.32991922, 0.58734894,]]
>>> sv.sortColsByFunction(lambda x :
... Numeric.sqrt(Numeric.sum(x*x)))
>>> sv.getData()
[[ 0.58734894, 0.32991922, 0.942316  ,]
 [ 0.17892206, 0.57061231, 0.01950742,]
 [ 0.29631215, 0.61336708, 0.70310301,]]
>>> sv.reset()
>>> sv.getData()
[[ 0.70310301, 0.61336708, 0.29631215,]
 [ 0.942316  , 0.32991922, 0.58734894,]
 [ 0.01950742, 0.57061231, 0.17892206,]]
>>> sv.permuteRows([2,1,0])
>>> sv.getData()
[[ 0.01950742, 0.57061231, 0.17892206,]
 [ 0.942316  , 0.32991922, 0.58734894,]
 [ 0.70310301, 0.61336708, 0.29631215,]]
```

The ability to sort views by arbitrary functions is a powerful feature whose functionality is limited only by the complexity of the sorting functions. The ability to sort based on lists and tuples, allows the functions to provide primary, secondary and tertiary keys for prioritization.

### 1.2.13 CachedView

A CachedView is a performance enhancing view generally used with deep view hierarchies. The view system generally processes data 'on-the-fly', which means that when data is requested via getData(), that request is

propagated up to the root dataset and then the data is filtered on the way down. When there are many layers, this process may become slow, especially for large datasets. A CachedView sits in between views and keeps a copy of its parent's getData() results. When data is requested from the CachedView, it is returned from this local copy, avoiding the need to propagate through the view tree.

There is no example of the CachedView as it returns identical data to its parent.

### 1.2.14 MinLDView

The MinLDView stands for Minimal Linear Dependence View and it does as its name suggests – minimize the linear dependence of a dataset. In this context, the linear dependence of a dataset is measured by fitting a power function to the eigenvalues of a dataset's covariance matrix. The function $x^{-\gamma}$ is fit and the parameter $\gamma$ is interpreted as the *degree of linear dependence* in the dataset. The larger $\gamma$, the higher the linear dependence.

Since high linear dependence greatly affects the ability to segregate the data, we would like a way to reduce this value. The solution, is to scale and rotate the dataset in such a way as to compensate for the intrinsic linear dependence. The formula used is

$$D' = DV^{\top}S^{-1}V \tag{1}$$

where $D$ is the original dataset, $V$ is the matrix from the $SVD$, and $S^{-1}$ is a diagonal matrix of eigenvalue reciprocals. This operation amounts to first rotating the data into PCA space, normalizing the variance in each dimension, and then rotating the data back to the original feature space.

Using the same dataset we used in the ColumnPCAView example to compare and contrast with:

```
>>> ds = Dataset([[1,1],[2,1.9],[2.9,3],[4,4],[4.95,5.02]])
>>> ds.getData()
[[ 1.  , 1.  ,]
 [ 2.  , 1.9 ,]
 [ 2.9 , 3.  ,]
 [ 4.  , 4.  ,]
 [ 4.95, 5.02,]]
>>> ldv = MinLDView(ds)
>>> ldv.getData()
[[ 1.42990478, 0.59329554,]
```

```
 [ 7.92372264,-3.93665154,]
 [-0.91718921, 6.8437997 ,]
 [ 5.71961912, 2.37318217,]
 [ 3.53328951, 6.52308277,]]
```

It is interesting to compare these values with those in the Column-PCAView. Notice that the values in the first columns are highly correlated, but the values in the second column have be scaled by a factor of 5 to 300.

## 1.3 Labelings

Labeling pick up the task of assigning meaningful (to the user) identifiers to raw datasets and making it easy to work with the data using these more abstract (but human-readable) identifiers. The Labeling class has many methods and one should look to the Labeling class documentation for the specifics.

Most methods contained in the Labeling class fall into one of a few categories, so we will examine the functionality one category at a time. The first functional category is simple housekeeping functions.

### 1.3.1 Creating and Destroying a Labeling

For our purposes, we'll create our familiar random 3 by 3 matrix and create a Labeling attached to it.

```
>>> from compClust.mlx.labelings import *
>>> ds = Dataset(MLab.rand(3,3))
>>> lab = Labeling(ds)
>>> lab
Labeling: None,  0 unique labels
```

The Labeling is instantiated bound to the particular dataset instance. Inspecting the Labeling shows that it has no name and zero unique labels. This shouldn't be surprising since we haven't added any labels yet.

As a first step, let's give the labeling a proper name.

```
>>> lab.setName('My Labeling')
>>> lab.getName()
'My Labeling'
>>> lab
Labeling: My Labeling,  0 unique labels
```

If we look at the dataset's list of current Labeling we can see the name of our new Labeling. To remove the Labeling from the Dataset we use the removeLabeling() method. This is analogous to the removeView() method for removing View, and the reasoning behind it is the same – the interaction between Dataset and Labeling is complicated in that it introduces circular references which must be explicitly broken. Let's remove the Labeling now.

```
>>> ds.getLabelings()
[Labeling: My Labeling,  0 unique labels]
>>> ds.removeLabeling(lab)
>>> ds.getLabelings()
[]
```

As with View we can retrieve Labeling by name for referencing and deleting. To try this, let's quickly create, name and remove a Labeling from the dataset.

```
>>> lab = Labeling(ds)
>>> lab.setName('foo')
>>> ds.getLabelings()
[Labeling: foo,  0 unique labels]
>>> ds.removeLabeling(ds.getLabeling('foo'))
>>> ds.getLabelings()
[]
```

This completes the range of use for instantiating and removing Labeling. Now let's do some real work with them!

### 1.3.2 Attaching Labels

Once a Labeling is instantiated, we usually want it to actually label something. The Labeling class has a rich set of methods which are enumerated below. For a more complete description of the functionality of each method, please refer to the API documentation.

- labelRows(obj)
- labelCols(obj)
- addLabelToRow(label, row)
- addLabelToCol(label, col)

- addLabel(label, key)
- addLabelToRows(label, rowList)
- addLabelToCols(label, colList)
- addLabels(label, keyList)

Hopefully most of these methods are self-explanatory. The only two which require explanation are `labelRows()` and `labelCols()`. These methods act similarly to the constructor of the Dataset object. That is, they attempt to intelligently deal with a multitude of data types and do the 'right thing' with each of them. A restriction on their use is that they must contain as many labels as there are rows or columns in the dataset. If the number of labels do not match, the behavior of these functions is undefined.

### 1.3.3 Removing Labels

There is an analogous set of methods for removing labels to those for attaching them. In brief, the methods are:

- removeAll()
- removeLabel(label)
- removeLabelFromRow(label, row)
- removeLabelFromCol(label, col)
- removeLabelFromKey(label, key)
- removeLabelsFromRow(row)
- removeLabelsFromCol(col)
- removeLabelsFromKey(key)

`removeAll()` will clear all the labels from a Labeling, effectively resetting the Labeling. This can be used to 'recycle' a Labeling for a different use. The `removeLabel()` method will remove every occurrence of a label from the Labeling regardless of where is appears. The rest of the methods allow for selective removal of labels.

### 1.3.4 Retrieving information

A Labeling isn't much good if you can't get any information out of it, so here are the methods available to query the Labeling about what label labels what.

- getLabels()
- getLabelsByRow(row)
- getLabelsByCol(col)
- getLabelsByKey(key)
- getLabelByRows(label)
- getLabelByCols(label)
- getLabelByKeys(label)
- getAllRowLabels()
- getAllColLabels()
- getAllKeyLabels()

Again, these methods parallel the functionality of the methods for attaching and removing labels. So, now that we have a full arsenal of methods to work with, let's try some examples. The first example will be to apply some simple labelings and retrieve data based on those labels.

```
>>> #
    # Create a dataset and labeling
    #

>>> ds = Dataset(MLab.rand(5,3))
>>> ds.setName('Test Dataset')
>>> lab = Labeling(ds)
>>> lab.setName('First Labeling')
>>> #
    # label all the rows and test querying
    #

>>> lab.labelRows(['row 0',1,'two','three',5.0])
```

```
>>> lab.getLabelsByRow(0)
['row 0']
>>> lab.getRowsByLabel('two')
[2]
>>> #
    # Add a single label to a column and test
    #

>>> lab.addLabelToCol('two',2)
>>> lab.getColsByLabel('two')
[2]
>>> lab.getKeysByLabel('two')
[2, 7]
>>> #
    # Add an existing row label to all rows, notice
    # that one row will be listed as being labeled
    # twice (which it is)

>>> lab.addLabelToRows(5,[0,2,4])
>>> lab.getLabels()
['three', 'two', 5.0, 'row 0', 1]
>>> lab.getRowsByLabel(5)
[0, 2, 4]
>>> lab.getRowsByLabel(5.0)
[0, 2, 4]
>>> #
    # Now remove some labels
    #

>>> lab.removeLabel(5)
>>> lab.getRowsByLabel(5.0)
[]
>>> lab.removeLabelsFromCol(2)
>>> lab.getLabels()
['row 0', 'two', 'three', 1]
>>> lab.removeAll()
>>> lab.getLabels()
[]
>>> ds.removeLabeling(lab)
>>> ds
```

```
Dataset: Test Dataset, 5 by 3
>>> ds.getLabelings()
[]
```

### 1.3.5 Global vs Local Labelings

In so far we've only dealt with local Labeling classes. That is the Labeling is only available to the Dataset or View that it was attached to upon instantiation. Often it is desirable to be able to work with labels that are persistent across all View of a Dataset. MLX provides a Labeling for this purpose and it's behavior is identical to that of a Labeling with the exception that it requires a view context as the first argument to all method calls that either retrieve or attach labels.

```
>>> #
... # Create a dataset and labeling
... #
...
>>> from compClust.mlx.datasets import Dataset
>>> from compClust.mlx.labelings import GlobalLabeling
>>> from compClust.mlx.views import *
>>> ds = Dataset(MLab.rand(5,3))
>>> ds.setName('Test Dataset')
>>> glab = GlobalLabeling(ds)
>>> glab.setName('First Global Labeling')
>>> # now we'll label all the rows 1,2,3 - remember it requires the context as first parameter
>>> glab.labelRows(ds, [0,1,2,3,4])
>>> glab
GlobalLabeling: First Global Labeling,  5 unique labels
>>> glab.getLabelByRows(ds)
[0, 1, 2, 3, 4]
>>> # Now we'll look at this labeling in a different view
>>> v1 = RowSubsetView(ds, [1,3])
>>> # Now if we look at all the row labels we should only see the label for rows 1 and 3
>>> glab.getLabelByRows(v1)
[1, 3]
```

Although often you will want to use a Labeling to attach labels to your data, often, once you've created a Labeling, you'll want it to "look and act" identically to a local Labeling, that is expose the same interface. This is

accomplished useing a Labelings which can be thought of as a local shell around a Labeling. A Labelings is a Labeling and behaves exactly the same except method calls to the Labelings act through a Labeling. A Labelings can be obtained in two ways: 1) through its constructer requireing a Labeling and a View. 2) From a View or Dataset's getLabelings() or getLabeling() methods. This is best explained through demonstration.

```
>>> from compClust.mlx.labelings import GlobalWrapper
>>> # This contiues from the previous example
>>> # We'll start by directly instantiating a GlobalWrapper
>>> gwrap = GlobalWrapper(v1, glabeling=glab)
>>> gwrap
GlobalWrapper: First Global Labeling,  5 unique labels
>>> gwrap.getLabelByRows()
[1, 3]
>>> # Now we'll get a GlobalWrapper using the datasets's getLabeling method
>>> gwrap = ds.getLabeling('First Global Labeling')
>>> gwrap.getLabelByRows()
[1, 2, 3, 4, 5]
```

And this concludes our quick example on basic Labeling usage. Remember also that View are conceptually identical to Dataset, so all of the Labeling techniques apply to View as well. Next we'll look at some of the sophisticated methods which leverage Dataset, View, and Labeling simultaneously.

## 1.4 Integrated Methods

The Integrate Methods heading is a bit of a misnomer. These method do nothing to integrate functionality, they merely take as arguments and/or return a mixture of Dataset, View, and Labeling objects.

### 1.4.1 subset

The Labelings method is a static method of the Labeling class. By passing in a Dataset, Labeling and a label or list of labels, it will return a SubsetView which contains the subset or rows and/or columns labeled by the particular label(s) chosen. If the labels only labels rows or columns, then all of the columns or rows, respectively, will remain intact. If the label labels both

rows *and* columns, then the view returned will be the intersection of the rows and columns.

More appropriately, this method may be thought of as creating a subset from a Labeling and labels(s). While it may seem simple, subsetting Dataset in this manner can be quite convenient. Let's try.

```
>>> from compClust.mlx.datasets import Dataset
>>> from compClust.mlx.labelings import *
>>> import MLab
>>> ds = Dataset(MLab.rand(15,5))
>>> labels = Labeling(ds)
>>> labels.addLabelToRows('x',[1,3,4,6,8])
>>> labels.addLabelToCols('y',[1,2,3])
>>> labels.addLabelToRows('z',[1,2,5,7,9,10])
>>> labels.addLabelToCols('z',[0,1,4])
>>> subset1 = subsetByLabeling(ds, labels, 'x')
>>> subset2 = subsetByLabeling(ds, labels, 'y')
>>> subset3 = subsetByLabeling(ds, labels, 'z')
>>> subset4 = subsetByLabeling(ds, labels, '')
>>> subset1
RowSubsetView: None, 5 by 5
>>> subset2
ColumnSubsetView: None, 15 by 3
>>> subset3
ColumnSubsetView: None, 6 by 3
>>> subset4
Dataset: None, 0 by 0
```

As you see, the subset returned matches the rows and columns specified in the Labeling. Also, when no rows were specified, all of the rows were returned, and the same for columns. Notice also, that creating a subset from non-existent label results in an empty Dataset.

### 1.4.2 labelUsing

labelUsing() is a *very* powerful method of the Dataset class. It allows one to label *any* Dataset or View using the labeling from and other Dataset or View to which it is directly or indirectly attached. An example may help.

Let's assume we have a dataset which has two labelings attached to it. One defined the results of partitioning the data into three classes. It

contains the labels '0', '1', and '2' and has labeled every row. The second labeling contains various annotations about the data. Now, if we created a subset from the first labeling (say we want to look at partition '2') via the Labelings method, how can we find out the annotation information for the data in this subset? We could look at the values in each datapoint, find them in the original dataset and then look at the label, but that is tedious and error prone is there is duplicate data. Instead we use labelUsing() which returns a new labeling for the subset which contains the labels for the corresponding datapoints.

```
>>> ds = Dataset(MLab.rand(9,2))
>>> classes = Labeling(ds)
>>> classes.addLabelToRows('0',[0,1,2])
>>> classes.addLabelToRows('1',[3,4,5])
>>> classes.addLabelToRows('2',[6,7,8])
>>> annot = Labeling(ds)
>>> annot.labelRows(['a1','b2','c4','d','e','f',
... 'a2','g','e3'])
>>> sub = ds.subset(classes, '2')
>>> annot.getRowLabels()
['a1', 'b2', 'c4', 'd', 'e', 'f', 'a2', 'g', 'e3']
>>> ds.getData()
[[ 0.93310058, 0.59187013,]
 [ 0.62153167, 0.28014728,]
 [ 0.66348588, 0.06767605,]
 [ 0.19404136, 0.3603994 ,]
 [ 0.39297128, 0.4130477 ,]
 [ 0.3661778 , 0.89579421,]
 [ 0.32203168, 0.02883008,]
 [ 0.53443021, 0.62014562,]
 [ 0.79724813, 0.12202285,]]
>>> sub.getData()
[[ 0.32203168, 0.02883008,]
 [ 0.53443021, 0.62014562,]
 [ 0.79724813, 0.12202285,]]
>>> class2lab = sub.labelUsing(annot)
>>> class2lab.getRowLabels()
['a2', 'g', 'e3']
```

As you can see, the Labeling `class2lab` contains the row labels of the `ds` object for the rows from which it was derived. This functionality is not

limited to parent-child relationship. Let's create another subset of odd rows and label it using the `class2lab` labeling.

```
>>> odds = ds.subsetRows([1,3,5,7])
>>> odds.getData()
[[ 0.62153167, 0.28014728,]
 [ 0.19404136, 0.3603994 ,]
 [ 0.3661778 , 0.89579421,]
 [ 0.53443021, 0.62014562,]]
>>> odd_and_class2 = odds.labelUsing(class2lab)
>>> odd_and_class2.getLabels()
['g']
```

Thus, only the row annotated with 'g', is both a odd row and in the class '2' subset. By the clever use of labels, the labelUsing() method can effectively provide the *intersection* of two datasets. It will be noted throughout the tutorial whenever a method has an analogous set operation such as union, difference, etc.

### 1.4.3 sortDatasetByLabel

While a SortedView by itself may be sufficient for some purposes, there is often the situation where it is of interest to order the data by some external characteristic. It may be for readability (order by classification), information (order by probability), or some other, user-conceived purpose.

To facilitate this, the Labeling class contains a `sortDatasetByLabel()` method. This is special in the sense that it will *only* work on a SortedView and will return an error if applied to any other view type. For an example, let's sort our previous dataset to even rows first, odd rows next.

```
>>> classes.sortDatasetByLabel()
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
  File "~/checkout/code/python/compClust/mlx/Labeling.py",
  line 99, in sortDatasetByLabel
    raise TypeError
TypeError
>>> sort_view = SortedView(ds)
>>> even_odd = Labeling(ds)
>>> even_odd.addLabelToRows(1,[0,2,4,6,8])
>>> even_odd.addLabelToRows(2,[1,3,5,7])
```

```
>>> sort_lab = sort_view.labelUsing(even_odd)
>>> sort_view.getData()
[[ 0.93310058, 0.59187013,]
 [ 0.62153167, 0.28014728,]
 [ 0.66348588, 0.06767605,]
 [ 0.19404136, 0.3603994 ,]
 [ 0.39297128, 0.4130477 ,]
 [ 0.3661778 , 0.89579421,]
 [ 0.32203168, 0.02883008,]
 [ 0.53443021, 0.62014562,]
 [ 0.79724813, 0.12202285,]]
>>> sort_lab.getRowLabels()
[1, 2, 1, 2, 1, 2, 1, 2, 1]
>>> sort_lab.sortDatasetByLabel()
>>> sort_view.getData()
[[ 0.93310058, 0.59187013,]
 [ 0.66348588, 0.06767605,]
 [ 0.39297128, 0.4130477 ,]
 [ 0.32203168, 0.02883008,]
 [ 0.79724813, 0.12202285,]
 [ 0.62153167, 0.28014728,]
 [ 0.19404136, 0.3603994 ,]
 [ 0.3661778 , 0.89579421,]
 [ 0.53443021, 0.62014562,]]
>>> sort_lab.getRowLabels()
[1, 1, 1, 1, 1, 2, 2, 2, 2]
```

It's not just integer labels which can be sorted. Any python object which is capable of being sorted (which is also any object which can be used as a label) can be used to sort a Dataset. You can even mix different python objects and sort on them, though the results may be strange. i.e. Is 0.2 ¡ (1,2,3)?

## 1.5 Section Summary

At this point you have been introduced to the core classes which compose PyMLX. You should be able to load, save and manipulate datasets and also attach labels to elements of the dataset. You have been introduced to some of the advanced functionality of the schema with regard to View and Labeling. In the next section, we will take what we have learned ad

apply it in making use of the various clustering algorithms in our repository. You will also be introduced to the Model class which represents an abstract mathematical model of a clustering.

## 2 The Wonderful World of Clustering

"By utilizing our powerful cyclo-heuristic Bayes-o-matic clustering algorithms, you will expose the secrets of the universe by a single mouse-click. Uncover the origins of life, predict the stock market, amaze friends and family!!"

OK, it would be nice if clustering (unsupervised machine learning) algorithms actually could do all this, but, in reality, they can't. What they *can* do, is help focus your time and energy as a scientist. On the surface, clustering algorithms can give contradictory results, but remember, different algorithms are searching on different characteristics of your data. They will dutifully find what they are programmed to, but it is up to you and your human brain to interpret the results in way that is meaningful to what *you* are looking for.

### 2.1 The ML_Algorithm framework

All clustering algorithms are derived from the `ML_Algorithm` base class. The name of this class simply stands for "Machine-Learning Algorithm", and an instance of the class represents some algorithm which can be run on a dataset and produce a classification result.

There are only two methods which are essential to run an `ML_Algorithm`. They are `run()` and `getLabeling()`. Along with the construction, these two methods allow you to execute the algorithm and get at interesting results.

There are two major classes of machine learning algorithms – supervised and unsupervised. Unsupervised algorithms are given only a dataset, and from that, they must determine, based on their internal criterion, what the proper classification of the data are.

Supervised algorithms, on the other hand, go through two distinct phases, a *training* and *testing* phase. During the training phase, the algorithm is provided with a dataset and also a Labeling of the data. The Labeling identifies which data points belong to which class. After the supervised algorithm contructs a model which fits the training set, it can enter the testing phase in which datapoint are input to the model and their estimated class is returned as output.

## 2.2 Unsupervised Wrappers

Since running a machine-learning algorithm is a CPU-intensive task for all but trivial datasets, the algorithms themselves are implemented in C. In order to execute the algorithms from the python environment, wrapper are provided. A wrapper is a subclass of the `ML_Algorithm` class designed to run an particular binary application and provide a consistent interface to the python side.

To create a wrapper, one need only construct a new instance of the appropriate wrapper type. All wrapper have the same interface for their constructors: `<name>(Dataset, parameters)` where Dataset is any Dataset or View object and parameters is a python hash which contains the parameters as key/value pairs.

Each wrapper has unique requirements in terms of required parameters and environment variable. You should refer to the API documentation for full details, but the essentials will be summarized at the top of each wrapper section. Please, note that the summary reflected the absolute *minimum* requirements, there may be many optional parameters, or certain values of the required parameters may require other parameters to be set.

### 2.2.1 DiagEM

Requirements:

- environment variable DIAGEM_COMMAND set to the `diagem` executable.
- parameter 'k' set to the number of clusters to find
- parameter 'num_iterations' set to the maximum number of iterations to try. The algorithm may converge sooner as well.
- parameter 'distance_metric' set to the type of distance metric to use.
- parameter 'init_method' set to the type of mean initialization to use.

DiagEM stands for Diagonal EM (Expectation-Maximization) and is named such to distinguish it from the full EM algorithm described below. The diagonal refers to the covariance matrix used in the optimization. Typically, this will be a full $N$-by-$N$ matrix where $N$ is the number of dimensions of the dataset. This means that there are $\frac{1}{2}kN(N+3)$ parameters to optimize. Where $k$ is the number of clusters, $\frac{1}{2}N(N+1)$ if the number of free parameters in the covariance matrix of each cluster and $N$ is the number

of free parameters to specify the mean for each cluster. Thus, if $F$ is the number of free parameters,

$$F = \frac{1}{2}kN(N+1) + N$$

$$2F = k(N^2 + N + 2N)$$

$$2F = k(N^2 + 3N)$$

$$F = \frac{1}{2}kN(N+3)$$

$$F = \frac{1}{2}kN(N+3)$$

Since this value grows quadratically with $N$, we may not have enough datapoints in our dataset to statistically justify the resulting classification. So, instead of using a full covariance matrix, we only consider the *diagonal* of the matrix.

By limiting ourselves, the Gaussians which the algorithm finds are restricted to being axis-aligned with the dataset's coordinate system. However, this restriction reduces the number of free parameters to $2kN$, a vast improvement.

This aside is indented to ensure that you are aware of the limitation of DiagEM and are able to recognize when its results should be taken with a string dose of NaCl.

So, which that out of the way, one to our first example of clustering!. For all of the clustering examples, we will be using a dataset which can be found on the `woldlab` server. The file itself is located at

```
/proj/cluster_gazing2/data/synthetic/datasets/ ...
 synth_t_15c3_p_0750_d_03_v_0d3.txt.gz
```

Note that it is a gzip-ed file, so you will need to copy it to your working directory and uncompress it there. Also, you will need to set an environment variable to indicate what binary program to use for running DiagEM. The variable is named DIAGEM_COMMAND. To set it, type the following:

```
export DIAGEM_COMMAND=/proj/code/bin/diagem
```

Once you have done so, we are ready to work. Start up the python interpreter and try out the following.

```
>>> from compClust.mlx.datasets import Dataset
>>> from compClust.mlx.wrapper.DiagEM import DiagEM
>>> from compClust.util.DistanceMetrics import *
>>> import MLab
>>> import Numeric
>>> ds = Dataset('synth_t_15c3_p_0750_d_03_v_0d3.txt')
>>> ds
Dataset: None, 750 by 3
>>> parameters = {}
>>> parameters['k'] = 15
>>> parameters['num_iterations'] = 100
>>> parameters['distance_metric'] = 'euclidean'
>>> parameters['init_method'] = 'random_point'
>>> diagem = DiagEM(ds, parameters)
>>> diagem.validate()
1
>>> diagem.run()
1
>>> results = diagem.getLabeling()
>>> results.getLabels()
['8', '9', '6', '7', '4', '5', '2', '3', '1', '14', '15',
 '12', '13', '10', '11']
>>> len(results.getLabels())
15
>>> map(lambda x : len(results.getRowsByLabel(x)),
... results.getLabels())
[40, 57, 54, 12, 2, 66, 137, 69, 6, 17, 99, 10, 32, 78, 71]
```

So, from these results, we can see that the DiagEM algorithm produced 15 clusters (exactly what we asked for), and there are between 2 and 137 datapoints to a cluster. Now let's make a group of subsets, one for each class, and compare on a class basis. Specifically, we'll look at how closely correlated the classes are to each other.

```
>>> classes = map(ds.subsetRows,
... map(results.getRowsByLabel, results.getLabels()))
>>> class_means = map(lambda x : MLab.mean(x.getData()),
... classes)
>>> def corr_matrix(means):
...     matrix = []
```

```
...    for mean in means:
...       corrs = []
...       for m in means:
...          corrs.append(CorrelationDistance(mean, m))
...       matrix.append(corrs)
...    return matrix
...
>>> corrs = corr_matrix(class_means)
>>> corrs_rank = map(ranks, Numeric.array(corrs))
>>> coors_most_similar =
... map(lambda x : x.tolist().index(14), corrs_rank)
>>> coors_most_similar
[11, 4, 3, 8, 9, 2, 12, 8, 3, 4, 14, 0, 14, 2, 10]
```

What this analysis shows is which class each class is most strongly correlated to other than itself. Some interesting things to notice are the mutual neighbors (classes 0/11, 10/14, 3/8, and 4/9). This might show that even though these clusters may be far apart, they express similar data.

### 2.2.2 FullEM

Requirements:

- environment variable FULLEM_COMMAND set to the `fullem` executable.
- parameter 'k' set to the number of clusters to find
- parameter 'seed' set to a floating point number with which to seed the random number generator. The seed is specified to that runs may be repeated.

The FullEM algorithm is very similar to the DiagEM algorithm described above, except that it uses the full covariance matrix, thus having to solve for $\frac{1}{2}kN(N+3)$ unknowns. However, by using a full covariance matrix, the Gaussians may take on an arbitrary rotation which could fit the data more tightly. As an example, let's re-cluster the same dataset as we did in the DiagEM section and compare the results.

```
>>> from compClust.mlx.dataset import Dataset
>>> from compClust.mlx.wrapper.FullEM import FullEM
>>> from compClust.util.DistanceMetrics import *
```

```
>>> import MLab
>>> import Numeric
>>> ds = Dataset('synth_t_15c3_p_0750_d_03_v_0d3.txt')
>>> ds
Dataset: None, 750 by 3
>>> parameters = {}
>>> parameters['k'] = 15
>>> parameters['seed'] = 1234.0
>>> fullem = FullEM(ds, parameters)
1
>>> fullem.run()
>>> #
... # A bunch of fullEM messages print out
... #
...
1
>>> results = fullem.getLabeling()
>>> map(lambda x : len(results.getRowsByLabel(x)),
... results.getLabels())
[71, 49, 12, 73, 50, 30, 22, 35, 87, 29, 64, 66, 76,
 26, 60]
```

So, it looks like the results from FullEM are more consistent than those from DiagEM. The DiagEM class counts had a standard deviation of 38.4, while the FullEM has a standard deviation of only 23.0, an excellent improvement. Remember, this synthetic dataset, is composed of a total of 45 clusters – 15 clusters made of 3 sub-clusters each.

Let's see what happens if we try to find all 45 clusters at once instead of the 15 top-level clusters.

```
>>> parameters['k'] = 45
>>> fullem2 = FullEM(ds, parameters)
>>> fullem2.validate()
1
>>> fullem2.run()
...
1results2 = fullem2.getLabeling()
>>> map(lambda x : len(results2.getRowsByLabel(x)),
... results2.getLabels())
[25, 8, 18, 14, 22, 16, 13, 50, 16, 31, 19, 20, 13,
```

```
 4, 71, 18, 19, 20, 6, 16, 4, 14, 19, 14, 13, 15, 21,
 22, 14, 15, 5, 8, 17, 10, 16, 20, 4, 21, 4, 18, 10,
 22, 11, 10, 4]
>>>
```

We would expect each cluster to have a size of 16 (750/45), and many clusters are in the 15-20 datapoint range, though there are several outliers. The standard deviation of this clustering is 11.6, which is much larger relative to the number of cluster than our previous result with k = 15. Although we can't be sure without further analysis, this results may indicate that we are getting a less stable result by over-specifying k. We will continue this analysis later in the tutorial.

### 2.2.3 HutSOM

Requirements:

- environment variable MATLAB_CODE_HOME set to the directory where `matlab` is installed.
- environment variable SOM_TOOLBOX_HOME set to the `somtoolbox` directory. This is a self-organizing map package from the University of Helsinki.
- parameter 'transform_method' set to the type of pre-process transformation to apply to the data. Usually set to 'none' since all transforms can be applied by View.
- parameter 'init_method' set to the type of initialization
- parameter 'som_x_dimension' set to the dimension to use for the x-coordinate. Usually set to 1.
- parameter 'som_y_dimension' set to the dimension to use for the y-coordinate. Usually set to the number of target clusters. Equivalent the 'k' for most other algorithms.
- parameter 'num_iterations' set to the number of step to update each node position.

HutSOM is a Self-Organizing Map implementation from the University of Helsinki[1]. Self-organizing maps attempt to interpret and organize high-

[1] http://www.cis.hut.fi/research/som-research/

dimensional datasets by modeling the available observations by a restricted, low-dimensional set.

The HutSOM wrapper is instantiated identically to the previous examples. Let's try running the HutSOM algorithms and compare the results to the EM algorithms above.

```
>>> from compClust.mlx.dataset import Dataset
>>> from compClust.mlx.wrapper.HutSOM import HutSOM
>>> ds = Dataset('synth_t_15c3_p_0750_d_03_v_0d3.txt')
>>> parameters = {}
>>> parameters['transform_method'] = 'none'
>>> parameters['init_method'] = 'random'
>>> parameters['som_x_dimension'] = 1
>>> parameters['som_y_dimension'] = 15
>>> parameters['num_iterations'] = 100
>>> parameters['seed'] = 1234
>>> hutsom = HutSOM(ds, parameters)
>>> hutsom.validate()
1
>>> hutsom.run()
1
>>>results = hutsom.getLabeling()
>>> map(lambda x : len(results.getRowsByLabel(x)),
... results.getLabels())
[76, 21, 45, 48, 54, 76, 68, 36, 61, 20, 67, 23, 38, 43, 74]
```

The SOM appears to create a fairly solid partitioning with a standard deviation of counts of only 19.9. If we increase the som_y_dimension to 45 to reflect the true number of primitive clusters in the dataset, the standard deviation drops to 7.85. However, this may not be a true improvement. If we multiply the standard deviations by the average cluster size (15 and 45), we get aggregate values of 298.5 and 353.25 respectively. Thus we might consider the 15 class clustering superior.

### 2.2.4 KMeans

Requirements:

- environment variable KMEANS_COMMAND set to the `kmeans` executable.

- parameter 'k' set to the number of clusters to find
- parameter 'max_iterations' set to the maximum number of iterations to try. The algorithm may converge sooner as well.
- parameter 'distance_metric' set to the type of distance measure to use when computing the distance between points.
- parameter 'init_means' set to the type of mean initialization

The KMeans algorithm is a very straightforward clustering algorithm which simply assigned points to the nearest cluster centroid. The KMeans algorithms can be interpreted as a special case of the EM algorithms described previously where the objective function is binary. Because of the similarities to EM, it is appropriate to contrast KMeans results with the EM results.

```
>>> from compClust.mlx.dataset import Dataset
>>> from compClust.mlx.wrapper.KMeans import KMeans
>>> ds = Dataset('synth_t_15c3_p_0750_d_03_v_0d3.txt')
>>> parameters = {}
>>> parameters['k'] = 15
>>> parameters['distance_metric'] = 'euclidean'
>>> parameters['init_means'] = 'random'
>>> parameters['max_iterations'] = 100
>>> kmeans = KMeans(ds, parameters)
>>> kmeans.validate()
1
>>> kmeans.run()
1
>>> results = kmeans.getLabeling()
>>> map(lambda x : len(results.getRowsByLabel(x)),
... results.getLabels())
[169, 135, 109, 134, 203]
```

Notice that even though we asked for 15 cluster, KMeans only returned 5. Whenever KMeans find that a cluster has collapsed to a single point, it will back off and try again with k = k-1. To get around this, we can either try setting a different random seed, or turning on k-strict mode. Let's try setting the seed.

```
>>> parameters['seed'] = 1234
>>> kmeans = KMeans(ds, parameters)
>>> kmeans.validate()
1
>>> kmeans.run()
1
>>> results = kmeans.getLabeling()
>>> len(results.getLabels())
6
```

So this time, we only got 6 clusters. Still not what we asked for, so let's try turning on k-strict.

```
>>> parameters['k_strict'] = 'true'
>>> parameters['max_restarts'] = 10
>>> kmeans = KMeans(ds, parameters)
>>> kmeans.validate()
1
>>> kmeans.run()
-1
```

A return value of -1 indicates that KMeans was unable to find exactly 15 clusters. The last alternative is to try a different initialization method. The Church initialization places all the means as far apart from each other as possible which sounds like a good idea, so let's try it.

```
>>> parameters['init_means'] = 'church'
>>> kmeans = KMeans(ds, parameters)
>>> kmeans.validate()
1
>>> kmeans.run()
1
```

Good, KMeans ran successfully and we now have 15 clusters to examine.

```
>>> results = kmeans.getLabeling()
>>> map(lambda x : len(results.getRowsByLabel(x)),
... results.getLabels())
[38, 19, 32, 50, 65, 29, 71, 57, 81, 21, 50, 34, 62, 119, 22]
```

This looks like a fairly good partitioning that compares well with DiagEM. If fact, the standard deviation of the points per cluster from KMeans is 27.2, which is almost as good as FullEM. Just to be fair, DiagEM using Church initialization turns in a 33.7 for its results' standard deviation, which is an improvement, but still not up to the level of KMeans.

### 2.2.5 TSplit

Requirements:

- environment variable TSPLIT_COMMAND set to the `tsplit` executable.
- parameter 'k' set to the number of clusters to find
- parameter 'distance_metric' set to the type of distance measure to use when computing the distance between points.
- parameter 'splitting_method' set to the split either via PCA projection or energy minimization.
- parameter 'agglomerate_method' set as how to prune the resulting tree into 'k' clusters.

The TSplit algorithm is a clustering algorithm developed by Prof. Ruye Wang over the summer of 2001. It is a top-down clustering algorithm, and as such, runs much faster than the bottom-up XClust algorithm. However, since it globally assesses the data, it is very sensitive to overlap in high-variance datasets. This sensitivity come from the fact the when clusters overlap, they tend to create a region of artificially high density which can be mistaken for an independent cluster.

This algorithm provides a useful counterpart to others in the suite, but one its own is not robust enough to use without prior knowledge of the dataset. To illustrate these shortcomings, we will compare how well it performs on a low- and high-variance dataset. For this example, we will need another dataset. Grab the dataset located at

```
/proj/cluster_gazing2/data/synthetic/datasets/ ...
 synth_t_15c3_p_0750_d_03_v_2d0.txt.gz
```

and uncompress it in the same manner as the first. Now, on to clustering.

```
>>> from compClust.mlx.datasets import Dataset
>>> from compClust.mlx.wrapper.TSplit import TSplit
>>> low_variance =
... Dataset('synth_t_15c3_p_0750_d_03_v_0d3.txt')
>>> high_variance =
... Dataset('synth_t_15c3_p_0750_d_03_v_2d0.txt')
>>> parameters = {}
>>> parameters['k'] = 15
>>> parameters['distance_metric'] = 'euclidean'
>>> parameters['splitting_method'] = 'PCA'
>>> parameters['energy'] = 0.9
>>> parameters['agglomerate_method'] = 'clusterNumber'
>>> tsplit_low = TSplit(low_variance, parameters)
>>> tsplit_high = TSplit(high_variance, parameters)
>>> tsplit_low.validate()
1
>>> tsplit_high.validate()
1
>>> tsplit_low.run()
1
>>> tsplit_high.run()
1
>>> results_low = tsplit_low.getLabeling()
>>> results_high = tsplit_high.getLabeling()
>>> map(lambda x : len(results_low.getRowsByLabel(x)),
... results_low.getLabels())
[1, 1, 1, 1, 13, 15, 1, 1, 1, 1, 694, 1, 12, 6, 1]
>>> map(lambda x : len(results_high.getRowsByLabel(x)),
... results_high.getLabels())
[2, 1, 1, 735, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]
```

Interestingly, even with a low variance ratio of 0.3, TSplit still has difficulty partitioning the dataset. When the variance ratio is increased to 2.0, TSplit totally fails, placing nearly all the datapoints in a single, large cluster.

The clustering results here are so bad, that it is not even worth calculation the standard deviation, since by inspection it is seen to be much larger than any of the previous clustering algorithms (FYI, the standard deviations are 178.2 and 189.5 respectively). If we re-run the process with `parameters['k'] = 45`, we get the following results for the number of

points per class.

```
>>> map(lambda x : len(results_low.getRowsByLabel(x)),
... results_low.getLabels())
[3, 1, 3, 1, 1, 1, 1, 1, 10, 1, 1, 1, 3, 1, 6, 1, 32, 1,
 1, 12, 1, 1, 1, 1, 1, 1, 13, 15, 1, 1, 1, 1, 495, 1, 1,
 26, 1, 87, 2, 1, 1, 1, 2, 11, 1]
>>> map(lambda x : len(results_high.getRowsByLabel(x)),
... results_high.getLabels())
[1, 1, 1, 1, 2, 1, 1, 1, 1, 2, 1, 1, 1, 2, 1, 1, 2, 1, 1,
 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
 1, 1, 1, 1, 1, 702, 1]
```

Again, marginal behavior for the variance ratio of 0.3, degenerate behavior for a variance ratio of 2.0. The interested reader should try running TSplit on a dataset of many more point with a variance ratio of 0.1. It is instructive to see how the performance of TSplit falls off as a function of datapoints and variance ratio.

### 2.2.6 XClust

Requirements:

- environment variable XCLUST_COMMAND set to the `xclust` executable.
- parameter 'cluster_on' set to cluster on the rows or columns of the data.
- parameter 'transform_method' set to the type of pre-process transformation to apply to the data. Usually set to 'none' since all transforms can be applied by View.
- parameter 'distance_metric' set to the type of distance measure to use when computing the distance between points.
- parameter 'agglomerate_method' set as how to prune the resulting tree into 'k' clusters.

XClust is the wrapper around the XCluster[2] algorithm created by Prof. Gavin Sherlock of Stanford University. This is a bottom-up clustering algorithm, and as such, runs in $O(N^2 \log N)$ time. However, in our experience,

[2] http://genome-www.stanford.edu/~sherlock/cluster.html

XClust can produce consistently good results over a wide variety of datasets. In light of this experience, it would seem that XClust might be a good first choice for clustering a dataset if there is no obvious favorite.

To see just how well XClust can perform, let's cluster our favorite dataset again.

```
>>> from compClust.mlx.datasets import Dataset
>>> from compClust.mlx.wrapper.XClust import XClust
>>> ds = Dataset('synth_t_15c3_p_0750_d_03_v_0d3.txt')
>>> parameters = {}
>>> parameters['k'] = 15
>>> parameters['transform_method'] = 'none'
>>> parameters['cluster_on'] = 'rows'
>>> parameters['distance_metric'] = 'euclidean'
>>> parameters['agglomerate_method'] = 'clusterNumber'
>>> xclust = XClust(ds, parameters)
>>> xclust.validate()
1
>>> xclust.run()
1
>>> results = xclust.getLabeling()
>>> map(lambda x : len(results.getRowsByLabel(x)),
... results.getLabels())
[57, 65, 51, 82, 13, 86, 36, 12, 71, 22, 87, 29, 76, 13, 50]
```

These results look *much* better than what we've been able to produce before. The standard deviation of the number of points per cluster is 27.6, better than DiagEM, but FullEM is still better.

### 2.3 Supervised Wrappers

All supervised algorithms inherit from the Supervised class which, in turn, is inherited from the ML_Algorithm class. The Supervised class extends the ML_Algorithm class by optionally taking in a Labeling or model parameter. If a Labeling is passed in, the Supervised algorithm operates in a 'learn' mode, in which it will attempt to create a model which fits the labeling when its run() method is invoked.

If a model is passed in, the algorithm operates in a 'predict' mode, in which it will classify data points into their appropriate classes during the execution of the run() method.

### 2.3.1 Support Vector Machines

Requirements:

- environment variable 'SVM_TOOLBOX_HOME' set to the directory where the SVM Toolbox is located.
- parameter 'C' set to the maximum magnitude a slack variable may take. The slack variables provide a way for the SVM to be tolerant of misclassifications. By default this is 1.
- parameter 'kernel' set the type of kernel to use. By befault this is set to 'linear'.

[To Be Added]

### 2.3.2 Artificial Neural Network

Requirements:

- environment variable 'ANN_COMMAND' set to the executable.
- parameter 'seed' initializes a random number generator used to assign random values to network weights before training begins. Defaults to 42.
- parameter 'lrate' Learning Rate sets the rate at which the network converges to (learns) a model. The value of learning rate tends to produce a tradeoff in the number of iterations required to converge to a solution versus the overall quality of the solution. Defaults to 0.002 and must be in the range [0, 1].
- parameter 'numIterations' Each iteration is also referred to as an epoch. An epoch occurs after each pattern (datum) has been presented to the network and prediction errors have been backpropagated through the network to update network weights. The order in which patterns are presented is randomly rearranged after each epoch. Defaults to 10000.
- parameter 'hiddenUnits' Space separated list of integers, each ¿ 0, may be empty. The network may contain zero or more layers of hidden units and each hidden layer may have one or more hidden units. Defaults to " (no hidden units).

The ANN is wrapped around the back-propogation neural network code from the University of Wisconsin - Madison.

[To Be Added]

## 2.4 Models

A Model is only required to have a constructor for the initial creation of the model and a `fitness()` method which takes a set of points and evaluates the fitness of the set relative to the internal model. The better the model fits the data, the higher the score.

### 2.4.1 Mixture of Gaussians

The Mixture of Gaussians model represents a dataset by a set of mean and covariance matrices. Each class is centered at a mean and has a Gaussian which extends as described by it's matrix. Each class also has a weight associated with it which is simply its total fraction of points divided by the total number of points in the dataset.

The formula for computing the fitness of a dataset given a model is defined by

$$L = \prod_{x \in X} \prod_{k} \frac{1}{\sqrt{(2\pi)^d |\Sigma_k|}} e^{-(x_k - \mu_k)\Sigma_k(x_k - \mu_k)} \tag{2}$$

where $\mu_k$ is the mean of cluster $k$, $\Sigma_k$ is the covariance matrix of cluster $k$, $d$ is the dimensionality of the data, and $X$ is the set of test datapoints.

Typically, we compute the *log-likelihood* instead since the exponentiation in equation 2 will often exceed machine precision and the cumulative products will drive the answer to zero rather quickly. The formula for the likelihood then becomes

$$L = \sum_{x \in X} \sum_{k} -\frac{1}{2}(d \log 2\pi + \log |\Sigma_k| + (x_k - \mu_k)\Sigma_k(x_k - \mu_k)) \tag{3}$$

Typical values for fitness will be on the order of $-1 \times 10^3$. Now let's try an example where we divide the dataset in half putting even rows in one subset, odd rows in another. We will then cluster one of the subsets via the DiagEM wrapper (since it produces a Mixture of Gaussians model) and score the model based of the other subset.

```
>>>
>>> ds = Dataset('synth_t_15c3_p_0750_d_03_v_0d3.txt')
>>> evens = ds.subsetRows(range(0,750,2))
>>> odds = ds.subsetRows(range(1,750,2))
>>> parameters = {}
>>> parameters['k'] = 15
>>> parameters['num_iterations'] = 1000
>>> parameters['distance_metric'] = 'euclidean'
>>> parameters['init_method'] = 'church_means'
>>> diagem = DiagEM(evens, parameters)
>>> diagem.validate()
1
>>> diagem.run()
1
>>> model = diagem.getModel()
>>> model
MoG_Model(k=15, ...)
>>> model.evaluateFitness(odds.getData())
-963.03029940902013
```

The actual fitness number returned is meaningless on its own as it is a dimensionless number. Fitness scores are useful when used as relative comparisons. It is important to remember that since there is no intrinsic scale associated with a fitness score (it is model-dependent), you cannot say that a fitness of -500 is twice as good as a fitness of -1000. All you know is that it is a better score.

### 2.4.2 Distance from Means

The Distance from Means model keeps a list of means, one mean per class and evaluated fitness by computing distances to the nearest mean. The exact formula is

$$F(\mathbf{x}) = \frac{N}{\sum_{i=1}^{N}(\mu_a - x_i)^2} \tag{4}$$

where $\mu_a$ is the mean nearest to $x_i$. One known shortcoming of this fitness function is that as the number of clusters for a given dataset goes up, the fitness increases as well. This is because as more datapoints are added to a finite volume, the average distance between them decreases. This reduces the summation and increases the score.

Let's try an example which illustrated this problem. The KMeans algorithm used a Distance from Means model by default, so we'll use it.

```
>>> ds = Dataset('synth_t_15c3_p_0750_d_03_v_0d3.txt')
>>> evens = ds.subsetRows(range(0,750,2))
>>> odds = ds.subsetRows(range(1,750,2))
>>> parameters = {}
>>> parameters['k'] = 15
>>> parameters['distance_metric'] = 'euclidean'
>>> parameters['init_means'] = 'church'
>>> parameters['max_iterations'] = 1000
>>> kmeans = KMeans(ds, parameters)
>>> kmeans.validate()
1
>>> kmeans.run()
1
>>> model = kmeans.getModel()
>>> model.evaluateFitness(odds.getData())
5.6023786988275539
>>> #
... # Try for many different k's
...
>>> parameters['k'] = 30
>>> kmeans.run()
1
>>> kmeans.getModel().evaluateFitness(odds.getData())
12.404585259842882
>>> parameters['k'] = 50
>>> kmeans.run()
1
>>> kmeans.getModel().evaluateFitness(odds.getData())
12.404585259842882
>>> parameters['k'] = 100
>>> kmeans.run()
1
>>> kmeans.getModel().evaluateFitness(odds.getData())
20.875393776420339
```

## 2.5 Meta-Wrappers

There are some algorithms which use other algorithms in their 'inner-loop'. If possible, we try to implement these types of algorithms on top on the existing python wrappers. If we can fully implement the algorithm in this way, without having to call an external application, we term this a Meta-Wrapper, since it implements a meta-algorithm (an algorithm over an algorithm).

The advantage to implementing in this way, is that a meta-wrapper can be used in conjunction with *any* other instance of a `ML_Algorithm`, including itself or other meta-wrapper. This lends enormous power to the family of `ML_Algorithms` when taken as a whole.

### 2.5.1 MCCV

Requirements:

- parameter mccv_parameter_name set to the parameter name of which of loop over.
- parameter mccv_parameter_values set to a list of values. On each iteration, the parameters specified by 'mccv_parameter_name' will be set to one of the values in the list.
- parameter mccv_test_fraction set to a number between (0.0, 1.0) which is the faction of point to hold out from the original dataset for testing.
- parameter mccv_num_trials set to the number of trials to run per parameter value. The larger the number of trial, the more confident the results.

MCCV stands for Monte-Carlo Cross Validation and is a technique designed to increase the confidence in a result based on probabilistic laws. The core ideas is simple; do multiple runs using a different random seed. Your confidence in the results can be quantified using the mean and variance of the set of runs.

### 2.5.2 Fitness Tables

As MCCV runs, it incrementally builds a fitness table. The fitness table has one entry for every internal run. The total number of runs it the number of trials times the number of parameters values. Each entry of the fitness table has three elements. The first is the value of the chosen parameters

| Value | Clusters | Fitness |
|---|---|---|
| 0.01 | 13 | -1907.89 |
| 0.02 | 12 | -1918.08 |
| 0.04 | 13 | -1929.90 |
| 0.08 | 13 | -1925.97 |
| 0.16 | 12 | -1949.32 |
| 0.32 | 13 | -1960.61 |
| 0.64 | 12 | -1973.45 |
| 1.28 | 12 | -2004.83 |

Table 1: Sample Fitness Table

name for that trial, the second value if the number of clusters returned by the clustering algorithm, the third parameters is the fitness as computed by the `fitness()` method of the Model object internal to the chosen clustering algorithm. An example fictitious fitness table is show in Table 1.

### 2.5.3 Running MCCV

Typically within the schema, MCCV is used to find an optimal number of clusters in a dataset by generating fitness curves. Let's try an example of this style of use.

```
>>> from compClust.mlx.datasets import Dataset
>>> from compClust.mlx.wrapper.MCCV import MCCV
>>> from compClust.mlx.wrapper.DiagEM import DiagEM
>>> parameters = {}
>>> parameters['mccv_parameter_name'] = 'k'
>>> parameters['mccv_parameter_values'] = range(10,30)
>>> parameters['mccv_test_fraction'] = 0.5
>>> parameters['mccv_num_trials'] = 6
>>> parameters['num_iterations'] = 250
>>> parameters['distance_metric'] = 'euclidean'
>>> parameters['init_method'] = 'church_means'
>>> ds = Dataset('synth_t_15c3_p_0750_d_03_v_0d3.txt')
>>> diagem = DiagEM()
>>> mccv = MCCV(ds, parameters, diagem)
>>> mccv.validate()
1
```

```
>>> #
... # .run() takes a while....
...
>>> mccv.run()
1
>>> fitness_table = mccv.getFitnessTable()
>>> fitness_scores = map(lambda x : x[2], fitness_table)
>>> fitness_scores.sort()
>>> best = fitness_scores[-1]
>>> best
-889.39357467205207
>>> #
... # our best score was -889, what index was this?
...
>>> map(lambda x : x[2], fitness_table).index(best)
15
>>> fitness_table[15]
[25, 25, -889.39357467205207]
```

So, MCCV decided that 25 clusters is the optimal choice of k for this dataset, even though we know that k = 15 or k = 45 are the better choices. Let's see what MCCV got for k = 15.

```
>>> for i in range(0,20):
...    print fitness_table[i]
...
[10, 10, -1033.5521324482584]
[11, 11, -1065.8959512850583]
[12, 12, -1007.1365541756992]
[13, 13, -994.33707715086541]
[14, 14, -997.98790638981552]
[15, 15, -1034.5531481223265]
[16, 16, -1021.524089374353]
[17, 17, -959.20473715505]
[18, 18, -959.0622038406899]
[19, 19, -941.34529509893059]
[20, 20, -938.26333395371717]
[21, 21, -912.91811083260927]
[22, 22, -913.51046227335962]
[23, 23, -918.66792167430026]
```

```
[24, 24, -925.541551492542]
[25, 25, -889.39357467205207]
[26, 26, -894.10952528835833]
[27, 27, -900.9496687048254]
[28, 28, -1008.228402419493]
[29, 29, -1007.0726811982155]
```

Strangely, k = 15 is one of the worst scores in the entire fitness table! Perhaps this is an artifact of the sub-clustering. Let's see what MCCV returns looping from k = 40 to 50.

```
>>> mccv.getParameters()['mccv_parameter_values'] =
... range(40,50)
>>> mccv.run()
1
>>> fitness_table = mccv.getFitnessTable()
>>> best = max(map(lambda x : x[2], fitness_table))
>>> map(lambda x : x[2], fitness_table).index(best)
0
>>> fitness_table[0]
[40, 39, -907.89855160200625]
>>> for i in range(0,10):
...    print fitness_table[i]
...
[40, 39, -907.89855160200625]
[41, 40, -918.08015090848448]
[42, 41, -929.90250751382689]
[43, 41, -925.97350376475447]
[44, 42, -949.32511844651754]
[45, 43, -960.61586289174261]
[46, 44, -949.83066376493514]
[47, 44, -973.45056832036857]
[48, 46, -1004.8372568468484]
[49, 47, -1006.718583971027]
```

Again, MCCV chose something we didn't expect. It chose a score where we asked for 40 clusters, but only got back 39, so it effectively chose k = 39 for this problem. The actual score at k = 45 was -960, but none of the trials ended up with k = 45.

The moral of this section is that MCCV is a great way to perform a near=exhaustive search of your dataspace, but it is not a universal cure-all.

The answer depends greatly on the choice of Model for a clustering as that determines exactly what the "peak of the fitness curve" actually means.

### 2.5.4 Hierarchical

Requirements:

- None

One such meta-wrapper, is the Hierarchical clustering algorithm. This algorithm recursively applies a clustering algorithm over a dataset until some stopping criteria is met. The core algorithm in pseudo-code is:

```
function hierarchical(dataset)

if ok_to_cluster(dataset)
   cluster_results = cluster(dataset)
   for each subset in cluster_results
      hierarchical(subset)
```

### 2.5.5 Terminators

The Hierarchical clustering algorithm provides a very fine level of control over the clustering process via a set of functions called `Terminators`. Terminators are second-order functions which return a function that, when evaluated on a Hierarchical Node, determine if that node should progress or not. `Terminator` functions can also be chained so that multiple, independent criteria can be examined before taking action.

To complicate matter further, there are three classes of `Terminator` functions which are summarized below.

**Prologues** These functions are evaluated just before a node is sent to a clustering algorithm. If *any* on the `Terminators` in the prologue chain vote false, the run is terminated.

**Resets** These are evaluated immediately after returning from clustering. Their job is to examine the clustering results and determine if it was a 'good' clustering or not. A reset `Terminator` can change the parameters of a clustering algorithm and then vote false, to force the dataset to be re-run with the new parameters. This terminator can cause infinite loops, so you must be very careful in its implementation.

**Epilogue** Epilogues are evaluated after the clustering has passed through the Reset `Terminators`, but before the subsets are recursed upon. This `Terminator` is useful for creating behaviors such as, "If only one class was found, there is no use re-clustering it."

Examples of `Terminators` which are a standard part of the `Terminator` modules are:

**trueTerminator** Always returns true.

**falseTerminator** Always returns false.

**clusterNumTerminator** Returns false if the number of datapoints in a cluster is equal to one.

**clusterSize(size)** Returns false when the size of a dataset fall below its argument *size*.

**PDRatio(ratio)** Returns false when the ratio of number of datapoints to number of dimensions ($\frac{rows}{columns}$) falls below a threshold *ratio*.

You can create your own `Terminator` functions by writing a function which return lambda forms. As an example, the `clusterSize()` terminator is written as:

```
def clusterSize(size):
  return lambda node :
      node.algorithm.getDataset().getNumRows() >= size
```

As you see, it returns a function which takes a single argument `node` and checks the size of the dataset of the algorithm within the node.

### 2.5.6 Running Hierarchical

The Hierarchical clustering algorithm is most useful in conjunction with MCCV. One can let MCCV determine the optimal number of clusters per node (and thus the number of child nodes), while the Hierarchical wrapper drives the calculations. In the case of our test dataset, we would hope that a Hierarchical/MCCV loop would produce a tree of roughly 15 clusters at the top level with three sub-clusters each. Let's try this approach using the FullEM algorithm as our inner loop as it seemed to perform best in our previous examples.

## 2.6 Running wrappers from the UNIX command line

If you think that typing in all of these parameters list by hand, instantiating classes and creating files is a bit tedious, you're right! For times when you have many jobs to run off as a batch process, or just want to play around a bit you really need something more suited to that task. To cater to this need, the wrapper are able to be run from the command line.

Every wrapper support the same command line arguements; there are only three.

```
<wrapper_name> parameter_file input_filename output_filename
```

### 2.6.1 Parameter File Format

The parameters file is evaluated in the python interpreter, so it may cantain any legal python code. In practive it is best to limit the use of this power since it could make the interoperability of parameter files very difficult.

Any assignment in the parameters file is assigned to a variable of the same name in a parameter hash. So, a typical parameter file may look like the following:

```
transform_method   = "none"
distance_metric    = "euclidean"
cluster_on         = "rows"
agglomerate_method = "clusterNumber"

k = 15
save_intermediate_files = 'yes'
```

One may insert comments into the parameter file by beginning the line with a '#' character. Also, a convenient shortcut is to use the `range()` function when specifying an MCCV parameter range.

To activate a meta-wrapper, use the following options,

```
MCCV = {'on'|'off'}             default to 'off'
Hierarchical = {'on'|'off'}     default to 'off'
```

If both wrappers are activated, MCCV becomes the inner loop and Hierarchical the outer loop. Be warned, a Hierarchical, MCCV clustering algorithm will take a *lot* of time and could possibly exhaust your system memory.

### 2.6.2 Input File Format

The input file format is the same as that loaded in our previous examples. Any file which Dataset can load in its constructor is a valid input file.

### 2.6.3 Output File Format

The output file is a simple dump of the labels in the Labeling returned by `getLabeling()`. There will be $N$ lines where $N$ is the number of rows in the input file and each line will contain a single number representing the class of the datapoint after the clustering has completed.

### 2.6.4 Command-Line Options

As a shortcut, parameters to the clustering algorithms may be specified on the command-line via a `--parameter=value` syntax. *This is not officially supported behavior and may be removed at any time.* A use of this may be if one just want to try a different `k` value and distance metric. i.e.

```
ClusterAlgorithmFoo.py foo.params foo.in foo.out --k=10
--distance_metric=correlation
```

# 3 Analysis....and stuff

While the ability to load, manipulate and cluster datasets may be seen as useful in itself, they are not adequate for the needs of most researchers. To be truly effective, one needs tool at their disposal which can quickly and easily present the data in many different forms. If the researcher is bound by their tools, they cannot be effective.

## 3.1 Leveraging Labelings & Views

Although not fancy, the Labeling/View classes do provide the ability to analyze a dataset in several way. One of the most common operations, is to see which datapoints correspond to different classes based of different clusterings. That is, how consistent are the clusterings? Do they agree on which points belong together. To illustrate this technique, let's compare the results from the KMeans and FullEM clustering algorithms.

```
>>> from compClust.mlx.datasets import Dataset
>>> from compClust.mlx.wrapper.FullEM import FullEM
```

```
>>> from compClust.mlx.wrapper.KMeans import KMeans
>>> from compClust.util.unique import unique
>>> #
... # Create some duplicate subsets
...
>>> ds = Dataset('synth_t_15c3_p_0750_d_03_v_0d3.txt')
>>> kmeans_data = ds.subsetRows(range(750))
>>> fullem_data = ds.subsetRows(range(750))
>>> #
... # Build the parameters table, notice that is it
... # perfectly acceptable to use the same hash for
... # different algorithms as long as it contains
... # all the required parameters for both.
...
>>> p = {}
>>> p['k'] = 5
>>> p['max_iterations'] = 1000
>>> p['init_means'] = 'church'
>>> p['distance_metric'] = 'euclidean'
>>> p['seed'] = 1234
>>> kmeans = KMeans(kmeans_data, p)
>>> kmeans.validate()
1
>>> fullem = FullEM(fullem_data, p)
>>> fullem.validate()
1
>>> kmeans.run()
1
>>> fullem.run()
1
>>> kmeans_labeling = kmeans.getLabeling()
>>> fullem_labeling = fullem.getLabeling()
>>> classes = map(lambda x :
... fullem_data.subset(fullem_labeling, x),
... fullem_labeling.getLabels())
>>> #
... # Build a set of labelings
...
>>> class1labels = classes[0].labelUsing(kmeans_labeling)
>>> class2labels = classes[1].labelUsing(kmeans_labeling)
```

```
>>> class3labels = classes[2].labelUsing(kmeans_labeling)
>>> class4labels = classes[3].labelUsing(kmeans_labeling)
>>> class5labels = classes[4].labelUsing(kmeans_labeling)
>>> all_class_labels = [class1labels, class2labels,
... class3labels, class4labels, class5labels]
>>> for i in range(5):
...    print i,
...    print unique(all_class_labels[i].getRowLabels())
...
0 ['2', '4']
1 ['0', '4']
2 ['0', '1']
3 ['3']
4 ['2', '4']
```

By inspection the two algorithms appear to have a non-trivial amount of agreement. Notice that cluster 3 produced by FullEM contains *only* elements from cluster 3 of KMeans and that cluster 3 appear no where else. This indicate 100% agreement on the points which compose this cluster. The other cluster don't agree quite as well, but there is never more than two KMeans classes per FullEM class.

### 3.2 Confusion Matrices

A confusion matrix shown the level of agreement (or disagreement) between two classifications.

Building on the previous example. Let's see exactly how much the two clustering agree.

```
>>> from compClust.score.ConfusionMatrix import *
>>> cm.createConfusionMatrixFromLabeling(kmeans_labeling,
... fullem_labeling)
>>> cm.printCounts()
140 78  0   0   3
12  0   0   0   140
0   0   0   105 0
0   85  100 0   0
0   0   87  0   0
```

Our assumption appears to be well founded. Most of the entries in the confusion matrix are zero, indicating that there is a large amount of

| Clusters | NMI | NMI Transposed | NMI Averaged |
|---|---|---|---|
| KMeans vs. FullEM | 0.7257 | 0.7438 | 0.7348 |
| KMeans vs. Ground Truth | 0.9886 | 0.4057 | 0.6971 |
| FullEM vs. Ground Truth | 0.9846 | 0.4141 | 0.6993 |

Table 2: NMI Comparison Scores

agreement. To quantify the exact amount of agreement, we have various scoring metrics available. Let's look at them now.

### 3.2.1 NMI

NMI stands for Normalized Mutual Information and is a performance metric for confusion matrices published by A. D. Forbes in 1995 in the Journal of Clinical Monitoring. NMI attempts to determine how well one classification (assumed to be the columns of the confusion matrix) is able to predict the second classification.

One should also look at the transposed NMI score as well. This can help one determine if one set of classes over- or under-specifies the second set of classes. NMI allows us to establish this one-way relationship. If the NMI and transposed NMI scores are high, then either classification is good at predicting the other.

The NMI scores for the KMeans vs. FullEM classifications are shown in Table 2. The transposed NMI scores are misleading since the ground truth for this dataset has 45 clusters. The hope is that the 5 cluster results is able to find an clusters such that each one is composed of one or more complete clusters from the 45 cluster ground truth. A high NMI score indicates this is true. The ground truth labeling can be found in the file

```
/proj/cluster_gazing2/data/synthetic/results/reference/
clust_t_15c3_p_0750_d_03_v_0d3_a_reference.txt.gz
```

Let's look at two examples. One where there is a two-way relation, and another where one classification over-specifies the other. Pay close attention to how the transposed scores differ in the two examples.

```
>>> from compClust.mlx.datasets import *
>>> from compClust.mlx.labelings import *
>>> from compClust.score.ConfusionMatrix import ConfusionMatrix
>>> from random import random
```

```
>>> ds = PhantomDataset(1000,1)
>>> labeling1 = Labeling(ds)
>>> labeling2 = Labeling(ds)
>>> labeling3 = Labeling(ds)
>>> labeling1.labelRows([1,2,3,4,5]*200)
>>> labeling2.labelRows([1,2,3,4,5,6,7,8,9,10]*100)
>>> labeling3.labelRows(map(lambda x : x + int(random()*1.3),
... [1,2,3,4,5]*200))
>>> cm1.createConfusionMatrixFromLabeling(labeling1,
... labeling3)
>>> cm2.createConfusionMatrixFromLabeling(labeling2,
... labeling3)
>>> cm1.printCounts()
55  145 0   0   0   0
0   44  156 0   0   0
0   0   40  160 0   0
0   0   0   55  145 0
0   0   0   0   48  152
>>> cm2.printCounts()
32  68  0   0   0   0
0   20  80  0   0   0
0   0   17  83  0   0
0   0   0   24  76  0
0   0   0   0   28  72
23  77  0   0   0   0
0   24  76  0   0   0
0   0   23  77  0   0
0   0   0   31  69  0
0   0   0   0   20  80
>>> cm1.NMI()
0.68119067506097197
>>> cm1.transposeNMI()
0.73142895910285066
>>> cm2.NMI()
0.68310769380320679
>>> cm2.transposeNMI()
0.51268566272042138
```

Notice that the NMI scores are almost identical between the two confusion matrices, but the transposed scores are much different. Experiment

with difference confusion matrices and get a feel for how the NMI score changes with respect to the the matrix geometry.

An important point to make is that the NMI score of a random confusion matrix is zero. If we create two mostly random labelings and compute their score, we will see that they are very near zero.

```
>>> labeling1.removeAll()
>>> labeling2.removeAll()
>>> labeling1.labelRows(map(lambda x : int(random()*10),
... [None]*1000))
>>> labeling2.labelRows(map(lambda x : int(random()*10),
... [None]*1000))
>>> cm1.createConfusionMatrixFromLabeling(labeling1,
... labeling3)
>>> cm1.printCounts()
8  17  19  19  20  10
2  25  16  22  12  17
8  31  23  20  19  19
4  17  21  25  20  15
5  16  25  26  17  23
5  18  24  18  28  20
5  13  15  12  22  12
7  18  11  18  24  10
2  14  26  30  16  13
9  20  16  25  15  13

>>> cm1.NMI()
0.015745466195284386
>>> cm1.transposeNMI()
0.011853264106865047
```

### 3.2.2 Linear Assignment

Linear assignment dies not depend of the rows/column ordering as does NMI. However, it does penalize for over or under-fitting. As shown in Table 3, the scores between the KMeans and FullEM results are very high relative to the ground truth, even through the NMI scores to ground truth were almost 1.0. This should that the clustering did not come close to finding the true number of clusters. Specifically in this case, they were under-specified.

Of course, in real data, one has no idea of the true number of clusters, so

| Clusters | Linear Assignment Score |
|---|---|
| KMeans vs. FullEM | 0.7427 |
| KMeans vs. Ground Truth | 0.1453 |
| FullEM vs. Ground Truth | 0.1453 |

Table 3: Linear Assignment Comparison Scores

a bootstrapping method is needed. By computing scores of many clustering against each other, one may be able to identify "consistent" clusters which may be part of an underlying structure in the data.

### 3.2.3 Adjacencies

What one typically wants to know when looking at a ConfusionMatrix is which clusters go together. That is, can we infer from the confusion matrix what the optimal mapping between different clustering classes is? The answer is yes. The problem itself is one of graph-matching. To solve this problem, we implement a graph matcher using an algorithm described by H. Gabow in his 1973 Ph.D. Thesis from Stanford University called N-cubed weighted matching.

While the details are complex, the usage is simple – pass in a confusion matrix and get a matrix with ones where classes match, zeros elsewhere. Let's try an example using our FullEM and KMeans results.

```
>>> cm.getAdjacencyMatrix()
[[1,0,0,0,0,]
 [0,0,0,0,1,]
 [0,0,0,1,0,]
 [0,1,0,0,0,]
 [0,0,1,0,0,]]
```

So, from this, we can see that the class mappings are $(0 \to 0, 1 \to 3, 2 \to 4, 3 \to 2, 4 \to 1)$. From this matrix we would be able to perform "best-guess" comparative analysis between the two clusterings.

There is also a related method called `getAdjacencyList()` which returns the mapping which we just derived by hand from the matrix.

```
>>> cm.getAdjacencyList()
[('4', '4'), ('2', '1'), ('3', '3'), ('0', '5'), ('1', '2')]
```

Another adjacency-type method is the `getAgreementList()` which will return a list equal to the number of elements in the ConfusionMatrix where each entry is a zero or one. A one indicates that both clustering agree that the given point belongs to the same class, a zero means the disagree. For a perfect clustering, all the entries would be one. This method provides a way to easily subset out the points which are 'stable'. So see how many points are agreed upon in this example, run the following command.

```
>>> len(filter(None, cm.getAgreementList()))
557
```

So 557 out of 750, or 74.3% of the datapoints were agreed upon. Notice how similar this percentage is to the NMI scores.

### 3.3 Confusion Hypercubes

The confusion matrix code is not limited to comparing just two labelings at a time. Any number of labelings may be packed together in a ConfusionHypercube. This related functionality completes the API for the ConfusionMatrix class. For reference, here is the list of all the methods implemented in the ConfusionMatrix class.

- getNumElements()
- getDimensionalLabeling()
- projectConfusionHypercube(labels)
- getAgreementList()
- getStarburst(index)
- getInverseStarburst(index)
- getConfusionHypercubeCell(cellCoordinates)
- removeCellFromHypercube(cellCoordinates)
- removeIndexFromHypercube(index)
- findCellCoordinates(index)
- createConfusionHypercubeFromLabeling(labelings, dimensionLabels)
- createConfusionMatrixFromLabeling(labeling1, labeling2)

- createConfusionMatrixFromFile(clusteringFile1, clusteringFile2)
- getHypercubeCounts()

Remember that the scoring method (NMI, Linear Assignment) are undefined for the Hypercubes. If you need to produce scores, the projectConfusionHypercube() method allows you to project the N-dimensional cube down to a matrix.

The most fundatments aspect of the ConfusionHypercube is the relation between indices and coordinates. When you create a Hypercube from multiple files or labelings, it is assumed that each row of the inputs relates to the same item. That is, the label for row 5 in two different labelings both refer to the same logical piece of data. These row numbers are the indices.

If one uses $N$ labelings to create a hypercube, then each cell is references by an $N$-dimensional coordinate. However, most of these cells will be empty. Therefore, the class maintains a internal mapping from indices (row numbers) to hypercube coordinates. By using the findCellCoordinates() method, you can convert from indices to coordinates.

Let's try an example using the FullEM, KMeans, and ground truth labelings from the previous example.

```
>>> cm.createConfusionHypercubeFromLabeling([truth,
... kmeans_labeling, fullem_labeling], ["Truth",
... "KMeans","FullEM"])
>>> cm.getNumElements()
750
>>> cm.getDimensionalLabeling()
['Truth', 'KMeans', 'FullEM']
>>> tmp = cm.projectConfusionHypercube(['KMeans','FullEM'])
>>> tmp.printCounts()
140 78  0   0   3
12  0   0   0   140
0   0   0   105 0
0   85  100 0   0
0   0   87  0   0

>>> cm.findCellCoordinates(1)
(23, 0, 0)
>>> cm.findCellCoordinates(7)
(13, 1, 4)
>>> #
```

```
>>> # Find all the points strongly related to index 7
...
>>> cm.getConfusionHypercubeCell((13, 1, 4))
[7, 85, 240, 292, 392, 445, 469, 472, 486, 495, 500, 502,
 504, 523, 555, 559, 582, 639, 684, 689, 706, 729]
>>> #
... # Find all the points weakly related to index 7
>>> cm.getStarburst(7)
[5, 36, 61, 66, 122, 123, 140, 163, 241, 243, 257, 277, 336,
 377, 535, 572, 609, 624, 636, 678, 120, 138, 193, 222, 343,
 371, 406, 410, 485, 594, 625, 633, 642, 710, 37, 73, 83, 101,
 144, 154, 303, 419, 482, 516, 579, 601, 608, 695, 702, 716,
 721, 35, 55, 169, 187, 234, 276, 283, 307, 354, 363, 450,
 454, 503, 529, 567, 580, 635, 650, 709, 724, 27, 49, 116,
 173, 198, 210, 304, 311, 396, 398, 428, 703, 4, 161, 228,
 350, 366, 444, 475, 593, 658, 671, 714, 23, 59, 63, 64, 79,
 93, 195, 221, 301, 367, 378, 386, 389, 403, 412, 426, 484,
 538, 565, 626, 637, 675, 686, 694]
>>> #
... # Show the cells these points occupy
...
>>> unique(map(cm.findCellCoordinates, a))
[(30, 1, 4), (32, 1, 4), (31, 1, 4), (34, 1, 4), (33, 1, 4),
 (14, 1, 4), (12, 1, 4)]
```

Notice that the confusion counts when projected onto the KMeans, FullEM plane are identical to the confusion matrix of just KMeans and FullEM. The starburst required a bit of explanation. It is composed of the indices of all the points who coordinates differ from the reference point by one index. This is the group of points "loosely-related" to the reference point.

### 3.4 ROC Curves

ROC stands for Receiver-Operator characteristics. The ROC curve identifies how many false positives one must tolerate to be guaranteed a certain percentage of true positives. An ideal ROC curve is the step-function.

The basic function in the ROC module is the computeRocForLabel() function. This function returns a dictionary with four items.

**area** The area under the ROC curve. The closer this value it to 1.0, the better the ROC curve.

**curve** The raw data for the ROC curve.

**insideHistogram** A Histogram object of points inside the class versus distance. This is an instance of the Scientific.Statistics.Histogram class.

**outsideHistogram** A Histogram object of points outside the class versus distance.

Using the results from the FullEM run,

```
>>> import bisect
>>> roc_stats = computeRocForLabel('1', fullem_labeling, ds)
>>> #
>>> # What percentage if we want a 0.0 percent false
>>> # positive rate? (minus 1 because bisect says where
>>> # to insert the point)
...
>>> (bisect.bisect(roc_stats['curve'][0], 0.0) - 1) / 750.0
0.16933333333333334
>>> #
>>> # 10 percent
...
>>> (bisect.bisect(roc_stats['curve'][0], 0.1) - 1) / 750.0
0.26933333333333331
>>> #
>>> # 50 percent
...
>>> (bisect.bisect(roc_stats['curve'][0], 0.5) - 1) / 750.0
0.59333333333333338
>>> #
>>> # 90 percent
...
>>> (bisect.bisect(roc_stats['curve'][0], 0.9) - 1) / 750.0
0.91733333333333333
```

So, it looks like we can only be confident of about 17 percent of the points classified into class '1'. Let's look at a summary of the stats for all the classes.

```
>>> from __future__ import nested_scopes
>>> roc_stats = computeRocFromDataset(fullem_labeling, ds)
>>> def tolerance(x, percent):
...     return (bisect.bisect(x['curve'][0],
...     percent) - 1) / 750.0
...
>>> def avg_roc(stats, percent):
...     return reduce(lambda x,y : x+y,
...     map(lambda x : tolerance(x, percent),
...     stats.values())) / len(stats.keys())
...
>>> #
>>> # On average, what's the zero false-positive percentage
...
>>> avg_roc(roc_stats, 0.0)
0.13066666666666665
>>> avg_roc(roc_stats, 0.5)
0.59813333333333329
>>> avg_roc(roc_stats, 0.9)
0.91813333333333325
```

In fact, class '1' was one of the better classes. On average, only 13 percent of the points are confidently correct. Of course, in real life we are willing to tolerate some error. For a 5 percent error rate (95 percent confidence), 21.2 percent of the datapoints are acceptable.

Only last feature of the roc module is the ability to graphically output the ROC curves. Let's look at an interesting one – class '2' vs. class '1'.

```
>>> plot1 = makeRocPlots(roc_stats['1'])
>>> plot2 = makeRocPlots(roc_stats['2'])
```

After running these commands, you should see two windows whose contents appear as Figure 1. The plot with the very sharp rise represents class 1 and the other, class 2. As you can see, the curve for class 1 is clearly superior to that of class 2.

### 3.5 Graphical Analysis

The ROC plots introduced you to one of the many tools in the MLX package for doing graphical analysis. In many cases, this is the fastest, most convenient way to analyze your data. Humans are invariably good at identifying

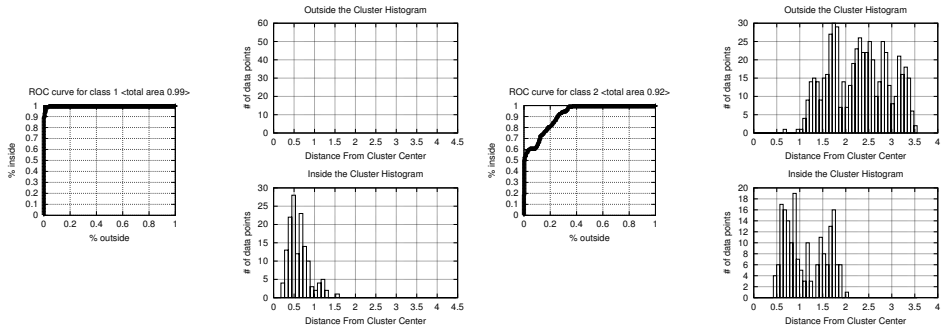


Figure 1: Example ROC Curves

patters in data as long as we have a succinct way of summarizing that data. The graphical tools provide us with this summarization.

### 3.5.1 IPlot

The IPlot package contants a wide variety of classes which allow you to interactively work with and visualize your data. This package was written by Christopher Hart, see a detailed description of its use in microarray data analysis [**?**].

The IPlot module provides several visualization tools, many of them are built atop the BLT graphing tool kit.

### 3.5.2 IPlot.plot

The most basic tool provided is a very simple 2d plotter. Given data to plot, it will attempt to figure out how to render it best.

# 4 Auxiliary Utilities

## 4.1 Interactive Tools