A Quick Start Guide to Microarray Analysis using compClust

Christopher E. Hart
California Institute of Technology
Pasadena, CA 91125

October 8, 2004

Contents

This tutorial is designed to bootstrap a user into using compClust and python for microarray data analysis. I will try and provide an explanation of what each piece of code is doing and why it works, but further reading my be required in you're favorite python reference. Code segments shown as below:

```
# this is a code segment
```

Should be "copy-and-pastable" directly into the python interpretor assuming all the modules are preloaded as specified below or using IPython the provided configuration files are being used.

1 Setting up the Environment - Python, IPython, compClust

I'll assume that Python, IPython and compClust have been installed by you or your system administrator. Follow the installation guide's instructions.

One of the greatest thing about python is the exposed interpretor. On a unix system, simply type python or on a windows system double click on the python.exe file and you'll be faced with a command prompt "»>" where you can enter any valid python statement and it will be evaluated when you hit enter. You can exit this environment by pressing ctrl-D. Through the rest of this tutorial I'll expect that the reader has at least read through some of the documentation or tutorials at http://python.org/doc/ and has a working understanding of python's basic syntax (which is quite straight forward and easy to learn).

```
\$ python
Python 2.2.3+ (#1, Jul  5 2003, 11:04:18)
[GCC 3.3.1 20030626 (Debian prerelease)] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> 1+1
2
```

1.0.1 IPython

Although the standard python interpretor is quite powerful, IPyhon written Fernando Perez and available at http://ipython.scipy.org/ adds many useful features to the standard python interface and I highly recommand that you use IPython. It creates a very rich and powerful command line interface (CLI) complete with tab-completion, object introspection, source code exploration, debuging tools, online help, variable management and many other very cool features.

Using IPython (or if you resist a custom $PYTHONSTARTUP file) you can set things up to autoload all the compClust modules that you find useful along with any other modules such as Numeric, MLab and stats perhaps. I've posted my ipython-compClust startup files at http://woldlab.caltech.edu/compClust/ and you can either use my setup or customize them, but be forewarned the rest of this tutorial will assume you've left the import semantics alone.

My IPython configuration preloads most compClust modules and many other modules which I find useful on a regular basis. The import semantics loads the modules into the name space of the top most module and avoids loading anything into the global name space to avoid conflicts. If you were to do these manually the import semantics would be as follows.

```
## compClust modules
from compClust.mlx import datasets
from compClust.mlx import labelings
from compClust.mlx import views
from compClust.mlx import models
from compClust import IPlot
from compClust.score import roc
from compClust.score import ConfusionMatrix
from compClust.util import listOps
from compClust.util import Histogram
from compClust.util import DistanceMetrics

## non-standard but nice modules
import MLab
import Numeric

## standard modules
import sys
import os
import string
import re

# now classes and methods are available as follows:

ds = datasets.Dataset(MLab.rand(10,10))
l  = labelings.Labeling(ds, 'labeling 1')
v  = views.RowSubsetView(ds, [1,2,3,9])
```

As a a usage note, I recommend using a text editor for drafting out code statements and then simply copy-and-pasting the segments into the IPython shell. This accomplishes several things: 1) Saves you're analysis path for further investigation. 2) provides a place to debug and edit your code. If you are using a X11 based system and can install xsel then the IPython magic function @pasteNrun may be quite useful to you. I find in practice as I dig deeper into unraveling a dataset I end up constructing a very specialized python module for interacting for a specific dataset. This dataset specific analysis module provides you with an instant notebook of your work which is critical to reevaluate your analysis and can be given to collaborators. Also remember that anything right of a # sign on a line is a comment and is not evaluated by python.

2 A First Step

I think the best way to understand the utility of compClust in terms of microarray analysis comes by example. The first step will be working with an uninteresting small synthetic dataset.

2.1 data creation

Lets create the dataset, throughout this tutorial I'll only write the python code and I'll leave out the command prompt, this allows you to easily copy and paste examples here into the IPython shell and explore the results.

This code will generate a 5-dimensional dataset from 3 distinct Guassian clusters. Imagine this begin a time-course of microarray results.

```
# this first cluster is mostly flat-liners (non-responders)

import RandomArray  # this comes with Numeric

# we'll be using the multivarate_normal function, learn more about it
# by typing multivarate_normal? in IPython.  Do the same for MLab.diag
# also know that you can get a list of all attributes (functions/classes)
# by typing RandomArray. and hitting the tab-key

nonResponders = RandomArray.multivariate_normal([0,0,0,0,0],
                                                MLab.diag(MLab.rand(5)*.5),
                                                [30])
posResponders = RandomArray.multivariate_normal([0,1,2,3,4],
                                                MLab.diag(MLab.rand(5)*.5),
                                                [20])
negResponders = RandomArray.multivariate_normal([0,-1,-2,-3,-4],
                                                MLab.diag(MLab.rand(5)*.5),
                                                [20])

# now we'll assemble a dataset, first by concatenating these
# three arrays together

a = Numeric.concatenate((nonResponders, posResponders, negResponders))

# now we can pass this array to the dataset constructor which is "smart" and
# can create a dataset from a list or lists , Numeric array, or a filename
# containing tab-delimanated data. NOTE: that module names are lowercase and
# class names are uppercase.

ds = datasets.Dataset(a)

# now we'll set the name of the dataset
```

```
ds.setName('Simple Example')

# lets also keep track of which group each of these vectors came from using
#labelings.  I'll describe in more detail the difference between
#GlobalLabelings GlobalWrappers and Labelings.  In microarray analysis  its
#almost always desirable to have GlobalLabelings of GlobalWrappers.

## I can't stress enough that you should us IPython's help system to
## investigate the parameters each of these function calls take.

originLab = labelings.GlobalLabeling(ds, 'origins')
originLab.addLabelToRows(ds, 'nonResponders', range(00,30))
originLab.addLabelToRows(ds, 'posResponders', range(30,50))
originLab.addLabelToRows(ds, 'negResponders', range(50,70))

# lets also keep track of the original index of each of the ''genes'' in this
# dataset

indexLab = labelings.GlobalLabeling(ds, 'index')
indexLab.labelRows(ds, range(ds.getNumRows()))
```

I highly recommend at this point you spend some time exploring the objects that you've created. IPython has pervasive tab-completion which is both great to save typing times, but also exploration. To use the tab-completion, simple start to type a variable, function, module or class name and hit the tab key. If when you hit tab you have unambiguously identified a variable it will automatically be completed to that variable. However, if it is ambiguous regarding which variable might be completed a list of all possible completions will be displayed. Continue type the name that you intend and hit tab again to repeat the process on a narrowed set of possibilities. This also works for member attributes for both classes and modules - for instance type "indexLab." then hit <tab>. You'll see a list of all the class functions available. Now you can learn more about any of these functions by completing the name and following it with a "?". At this point you should be able to learn about all the basic functionality of both labelings and datasets.

2.2 simple visualization

Visualization is essential to the analysis of most data and microarray data is no exception. Most of the visualization tools I'll be discussing reside in the IPlot module. Effectively IPlot strives to provide interactive plotting tools such that users can identify data by "clicking" on points on the plot.

The DatasetPlot and DatasetPlotView classes form the core of the IPlot system. Currently the DatasetPlot is built on top of the very nice Pmw.BLT.Graph widget - therefore any operations that can be done to a BLT.Graph can also be done to a DatasetPlot. The DatasetPlot is considered the rendering engine and the DatasetPlotView pro-

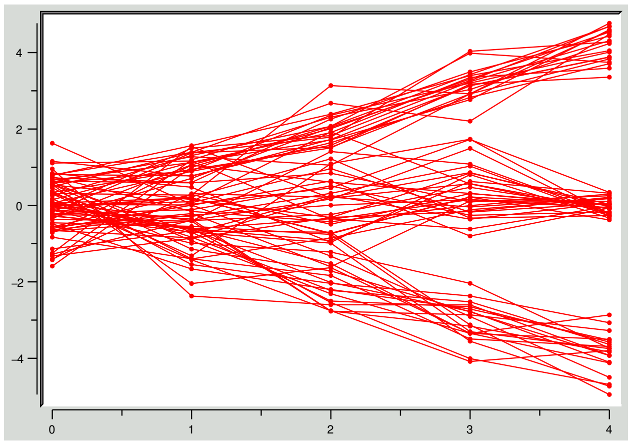


Figure 1: Generated Data Visualized in using a DatasetRowPlotView

vides mappings of arbitrary data attributes to plot characteristics such as marker and line coordinates, size and color.

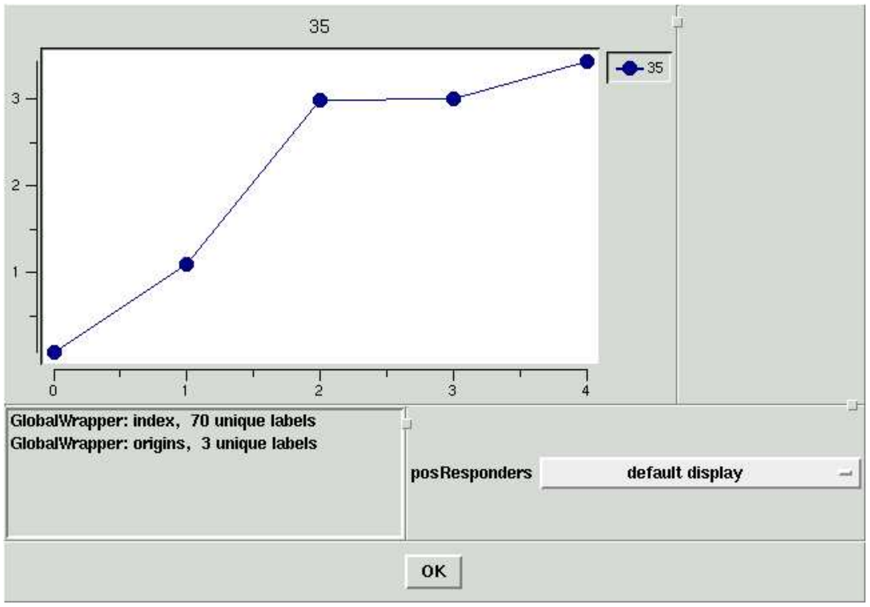
As usual this is best demonstrated. This is a continuation from the previous snippet of code, so make sure you are working from within the same python session.

```
# first we'll create a DatasetPlot and a DatasetRowPlotView

dp = IPlot.DatasetPlot()
v  = IPlot.DatasetRowPlotView(ds)

# now we render the plotview by tell the DatasetPlot to plot it
dp.plot(v)
```

You should see a plot something like figure 1

You should also find that clicking on a line results in a small information box being displaying in the upper left hand corner. Holding down Control and Left clicking on this box will open another dialog box with more information about that gene vector as shown in figure 2. From this box you can explore which labels are attached to this vector. By clicking on a labeling in the left box the right box shows what label(s) are attached to it from that labeling. Further you can highlight all gene-vectors which share that label using the pull-down action list on the right-hand box. You can zoom the plot by holding shift and the left mouse button, zoom-out by holding control and pressing the left mouse button.

Often you might want to have the plot display more information when you click on a gene-vector. This can be accomadated for using the primaryLabeling and secondary-Labeling parameters which work on almost all IPlot functions. The primaryLabeling



GlobalWrapper: index, 70 unique labels
GlobalWrapper: origins, 3 unique labels

Figure 2: Sample Information Box From 1

always defaults to the index or key of the gene-vector in the particular view or dataset but can be over ridden, but you *must* be sure that there is a unique label for each gene vector. It is usually best to leave the primaryLabeling alone. The seondaryLabeling however can be anything - we'll set it to be the origin in the example below.

```
v  = IPlot.DatasetRowPlotView(ds, secondaryLabeling=ds.getLabeling('origins'))
dp.clear()
dp.plot(v)
```

Now when you click on a line (or gene vector) you'll see the origin label in the information box.

Now it is important to understand that the DatasetRowPlotView is composed of 5 Mappers which take attributes of the data and map them onto attributes of the plot. The five mappers are listed below:

DataMapper Positions sets of x,y points for each row in the dataset.

ColorMapper Sets the color for each row in the dataset.

MarkersMapper Determines the marker size and color for each row in the dataset.

AnnotationMapper Determines the annotation that should be rendered for each row in the dataset.

BindingsMapper Determines what events should be triggered when data points are clicked on.

Next I'll demonstrate how to use the built in general purpose mappers to do some interesting analysis of the data. As you become more comfortable using compClust and python you may find it useful to subclass out these mappers for special purposes.

2.2.1 Color Mapping

First we'll investigate color. Again we'll build on the code we've been workign with so don't close any windows or exit IPython. If you did this should all in a blink of an eye, so worry not.

```
# get the colorMapper that is currenlty being used by the DatasetRowPlotView
cm = v.getColorMapper()

# lets set the color based on the value a vector has at the t=0
cm.setColorByColValue(0)
# now tell the plot to update itself.
dp.plot()

# it will probably be more interesting to color based on the last column of
# the dataset
cm.setColorByColValue(4)
dp.plot()

# if you didn't want to scale between red and blue you can adjust the
# colorRange parameter.  Here we'll go between red and yellow.  If you
# think of a color wheel that starts at 0 and ends at 1 that is the
# how color is assigned.

cm.setColorByColValue(4, colorRange=(0,0.3))
dp.plot()

# You can also threshold the color scale.  Here we'll cap at +/- 0.5
cm.setColorByColValue(4, minValue=-0.5, maxValue=0.5)
dp.plot()

# all colors are represented as HSV colors and so far we've only
# adjusted the H - or hue parameter.  I'll leave it as an excersize to
# the reader to adjust the saturation and value parameters.

# perhaps we would like to color each line by there index in the file.  That is
# gene-vector 1 gets to be red and gene-vector 70 gets to be blue and
# everything else is in between
```

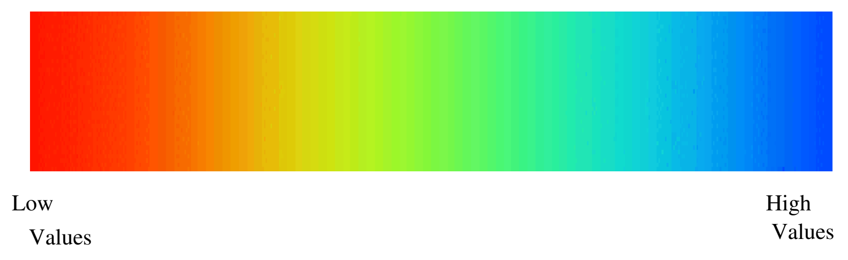


Figure 3: the standard color scale used in IPlot

```
cm.setColorByIndex()
dp.plot()

# we can also color based on a labeling (say the group of origin)
cm.setColorByLabeling(ds.getLabeling('origins'))
dp.plot()

# lastly we might want to color based on a function - say the average
# value of each vector.

def rowAvg(ds, row):
  return(MLab.mean(ds.getRowData(row)))

cm.setColorByFunction(rowAvg)
dp.plot()

# notice I had to define a function above which expects a dataset object
# and a row and returns a value.  The ColorMapper takes care of scalling
# this value and turning it into a color.  However, sometimes though you might
# want to do this in one line using a lambda function (look it up in the python
# docs, its quite powerful).  The below does exactly the same thing
# as above.

cm.setColorByFunction(lambda ds,row: MLab.mean(ds.getRowData(row)))
dp.plot()
```

The default color map used throughout IPlot is shown in 3 and the renderings of the above example code is shown in figure 4 we can see that we can quickly choose different coloring schemes.

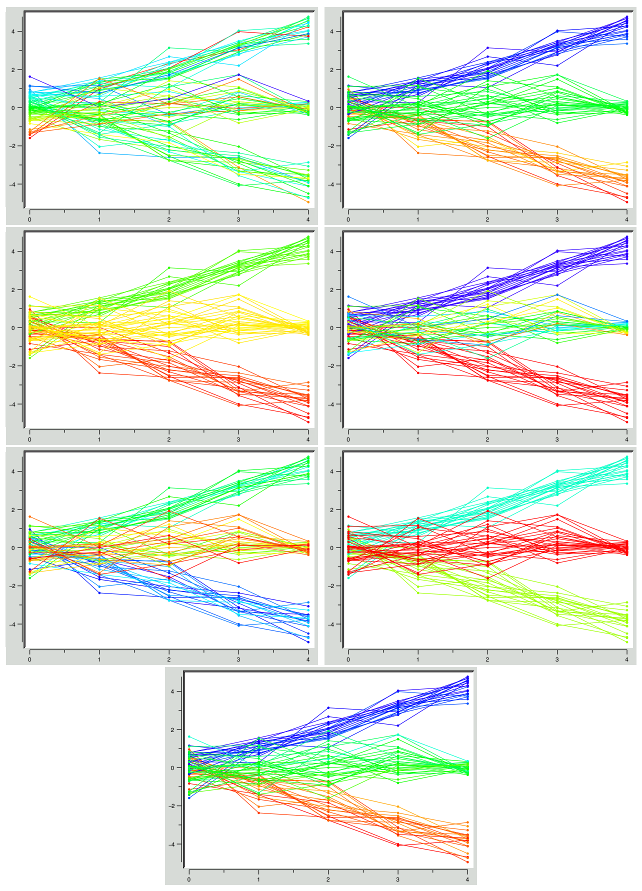


Figure 4: Various different ways to adjust the ColorMapper. Colored by: A) value at column 0 B) value at column 4 C) value at column 4 (only using Red-to-Green) D) value at column 4, truncating color scale from +/- .5 E) index in the dataset F) the origins labeling G) the mean of each vector

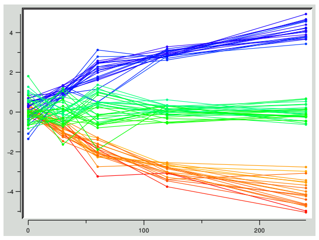


Figure 5: example modifing the RowDataMapper to scale the X-axis

2.2.2 Data Mapping

Next we'll explore different DataMappers. DataMappers project a high-dimensional vector into some representation inside a 2d graph. In all of the proceding figures we've been using the default DataMapper - the RowDataMapper. This data mapper for each vector in the dataset plots its points vs the index value. However, sometimes you would like the X-axis to be scaled according to time or some other non-evenly spaced interval 5

```
# first we'll create a labeling on ds that describes a time axis
# imagine we've taken RNA samples at the below time interval.

time = labelings.GlobalLabeling(ds, 'time')
time.labelCols(ds, [0,30,60,120,240])

# now we'll grab the DataMapper
rdm = v.getDataMapper()
rdm.setXAxisLabeling(ds.getLabeling('time'))
dp.plot()

%% FIXME: how do you get this plot to rescale?
```

Sometimes you may want to generate a scatter matrix of your data where the X-axis represents the values of expression during 1 measurement and the Y-axis another. We can use the ColumnScatterDataMapper for this 6

```
# we construct a pcaDM
csDM = IPlot.ColumnScatterDataMapper(v)

# now we can set v's dataMapper to be this new dataMapper
v.setDataMapper(csDM)
dp.plot()
```

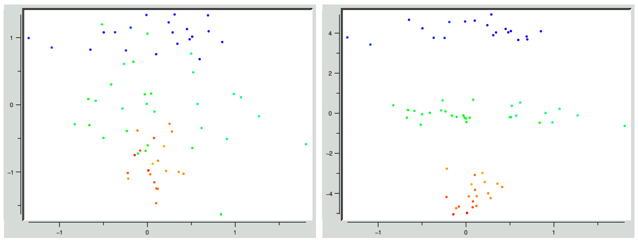


Figure 6: example using the ColumnScatterDataMapper. A) column 0 vs 1 B) column 0 vs 4

```
# Now we can modify which two columns are plotted against each other
csDM.setXColumn(0)
csDM.setYColumn(4)
dp.plot()
```

Principle component analysis (PCA) has become quite a useful tool in mapping out overall data structure of a dataset. We can simply swap out the RowDataMapper with the PCADataMapper 7. Note that even though the plot has made quite a pronounced change, all the other Mappers are still the same, so color, annotation and bindings are all maintained. To better understand the PCA projection I've written IPlot.PCAExplorer which will be discussed below.

```
# we construct a pcaDM
pcaDM = IPlot.PCADataMapper(v)

# now we can set v's dataMapper to be this new dataMapper
v.setDataMapper(pcaDM)
dp.plot()
```

2.2.3 Marker Mapper

The final Mapper we'll discuss is the Marker Mapper. Using the MarkerMapper you can adjust the size of each marker much like you can adjust the color using the color mapper 8.

```
# first we get a handle to the MarkersMapper
mm = v.getMarkersMapper()
```

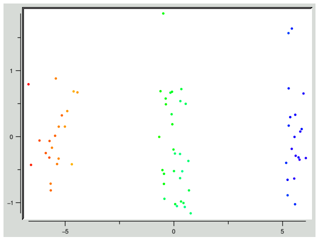


Figure 7: PCA view of the same data that was shown in 5

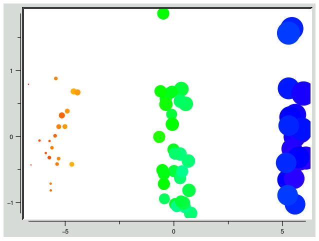


Figure 8: PCA view of the dataset, but now the marker size is proportional to value in the 4th column of each vector.

```
# now we'll adjust the size proportionally to the value in col 4
mm.setMarkerSizeByColValue(4)

dp.plot()
```

2.2.4 Binding Mapping and Annotation Mapping

These mapper are an implementation detail that can be ignored by most users.

2.3 Complex Visualization

Many of the concepts behind these visualization and analysis techniques are discussed in more detail in [**?**] and will be minimally discussed in this tutorial.

2.3.1 Trajectory Summaries

Often given a partitioning or classification of your data you'll want to inspect each of these clusters. The TracjectorySummary is a quick visualization to draw a mean/std dev. plot for each cluster. Inside of compClust you'll usually have your classification described in terms of a labeling. If this is a globalLabeling than all is good, if it

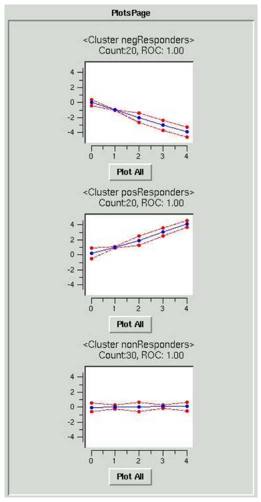


Figure 9: A trajectory summary of the synthetic dataset divided by the origins labeling.

is not, then you'll have to cast the local labeling to a globalLabeling using the labelings.castToGlobalLabeling function. Here we'll generate a trajectory summary based each gene vector's cluster of origin, the origin's labeling contains this information. Figure 9 illistrates the results of the below function call. Note you can get a detailed view of any of the clusters by clicking on "plot all" button.

```
ts =IPlot.TrajectorySummary(ds, ds.getLabeling('origins'))
```

2.3.2 Confusion Matrices

Comparitive analysis of clustering results can be quite a powerful tool [**?**]. Briefly a confusion matrix describes the cardinalities of all pairwise intersections between two clustering results or partitions. We've defined a confusionArray to be an array of sets containing all pairwise intersections between two clustering results or partitions. The software blurs this distinction and refers to a confusion array as a confusion matrix. There are two basic ways of working with confusion matrices: 1) graphically, 2) computationally.

Before we can create a confusion matrix we need two labelings to compare, and insofar we only have one. We'll generate a quick clustering of the data. The details on clustering is described in more detail below and in the pyMLX user tutorial. On purpose I'm using a k=4 to forse a slight description between the two partitions for demonstrative purposes.

```
# first we create a parameters dictionary.
# In this case, we're using DiagEM, so here are the parameters
p = {'distance_metric': 'euclidean',
     'init_method': 'random_sample',
     'samples': 1,
     'k': 4,
     'k_strict': 'false',
     'num_iterations': 50,
     'seed': 664361265
    }
# first we create an algorithm wrapper
aw = wrapper.DiagEM(ds, p)
aw.run()

# now I like to make the resulting labeling global
l = labelings.castToGlobalLabeling(aw.getLabeling(), 'Diagem')
```

First I'll describe the visualization of the confusion matrix (fig 10). followed below by some of the backend API calls that can be useful when setting up analysis chains or scripts. Middle clicking on any of these graphs will bring up a detailed version of the plot and all the previous interactions descriped above in the datasetPlot section apply to these graphs as well.

```
cm = IPlot.ConfusionMatrixSummary(ds,
                                  ds.getLabeling('origins'),
                                  ds.getLabeling('Diagem'))
```

Now lets compare the 'diagem' labeling with the 'origins' labeling using the API calls and no visualization.

```
# first we create a confusion matrix, then we populate it.
cm = score.ConfusionMatrix()
cm.createConfusionMatrixFromLabeling(ds.getLabeling('origins'),
                                     ds.getLabeling('Diagem'))
```

Now we can start to integate the matrix using the confusion matrix's methods.

```
# this returns the confusion matrix (cardinalities)
cm.getCounts()

## Now we can quantify the simularity
# by linear assignment
cm.linearAssignment()
# by normalized mutual information
cm.NMI()
# or the by transposed NMI
cm.transposedNMI()

# We can also determine the adjancy mapping
cm.getAdjacencyList()
```

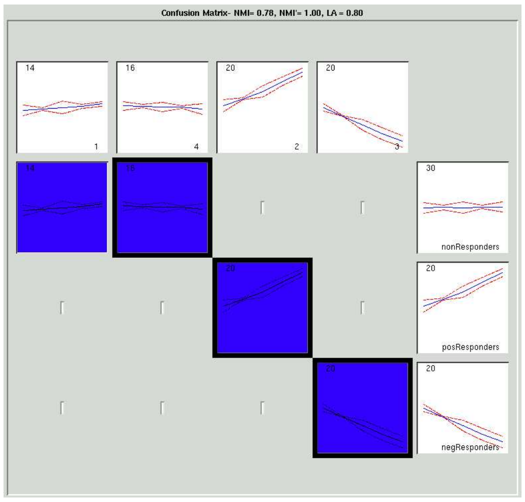


Figure 10: Confusion Matrix of origns vs EM/MoDG clustering results. White boxes across the columns show a mean (blue) and standard deviation (red) summary plots for each cluster in the EM/MoDG. White boxes aligned down the rows are the mean and standard deviation summary plots for each class in the origins Classification. The upper left corner of each plot displays the feature (gene vector) count. Each cell (Ci, j) within the confusion array shows a mean and standard deviation summary plot for the intersection set of genes that are in common between origins class i and the EM/MoDG result cluster j. The background of each plot is colored according to a heat-map that registers the proportionate number of genes in the cell compared with the corresponding class in the EM/MoDG result (coloring can also be specified relative to origins input or other functions). Intersection cells with dark outlines indicate are optimal pairings between the two data partitions, as determined during the LA calculation

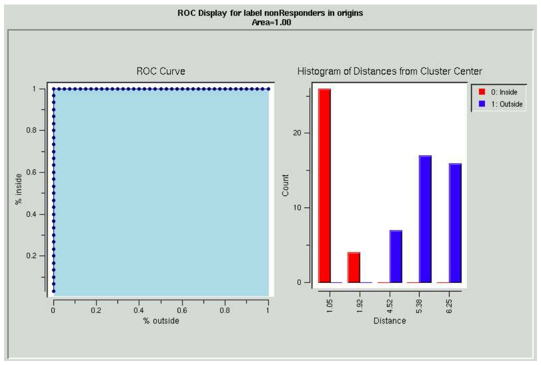


Figure 11: A curve and distance histograms

2.3.3 ROC Analysis

Receiver Operator Characteristic (ROC) analysis provides a way to assess the degree a cluster overlaps with other clusters. Again IPlot impliments a nice visualization and the score module impliments the underlying algorithm and functionality. I'll demonstrate the IPlot visualization 11 Note that in this example dataset things are a little too perfect.

```
IPlot.rocPlot(ds, ds.getLabeling('0origins'), 'nonResponders')
```

2.3.4 Histograms

Often it array analysis understanding the distribution of data is critical - further being able to explore either outliers or the center of the distribution is an often desired task. The compClust.Histogram module coupled with the IPlot.HistogramPlotter provides a very flexiable system for looking at data distributions.

Effectively the IPlot.HistogramPlotter simple plots a bar graph of counts based on a labeling. For example to get a feel for the cluster sizes of the origins labeling we can simple do this 12:

```
g = IPlot.HistogramPlotter(ds.getLabeling('origins'))
# we can reorder the elements by size
g.plot(sortBy='size')
# or we sort by label name
g.plot(sortBy='label')
```

As you can see any labeling can be passed to the IPlot.HistogramPlotter and it will simple create a bar graph of sizes. You can also middle click on any bar to open a detailed trajectory summmary of genes contained within that group. If you need to change the attributes of that opened plot, there is a handle for it as a member variable selectedPlot and selectedView.

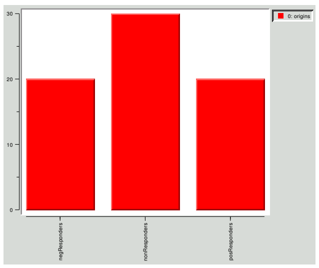


Figure 12: Bar graph of cluster sizes

To actually create a histogram then we need to create a labeling which maps bin names to data elements. By creating labeling to represent the histogram all the functionality of labelings are available for interogation of the histogram including confusion matrixes. In the compClust.util.Histogram module several methods for creating histogram labelings are provided. The labelings can be attached across rows or columns. Below I'll demonstrate building histograms across the rows.

```
# to create a histogram bining of RowData
h =  Histogram.binOnColData(ds, 4, 20)
# now we can plot this:
IPlot.HistogramPlotter(h)
```

Figure **??** shows the result above. We can see that there is effectively 3 peaks which correspond with the origins of the cluster. Using the IPlot.ConfsionMatrixSummary we can inspect the relationships of these bins with the origin labelings.

```
h = labelings.castToGlobalLabeling(h)
cm = IPlot.ConfusionMatrixSummary(ds,
                                  ds.getLabeling('origins'),
                                  h,
                                  l2Order=listOps.sort(h.getLabels()))
```

Note above we use the l1Order optional argument to insist that the histogram labels are arranged according to their value as opposed to the default ordering which is based on cluster size.

Next we'll build a histogram based on a row function. This becomes a little tricky and will involve the use of a lambda funtion, this is just like we did in the colorMapper.setColorByRowFunction. The function will require two parameters a dataset and a row and it will return a single numeric value.

```
h = Histogram.binOnRowFunction(ds, lambda ds,row: MLab.mean(ds.getRowData(row)),
IPlot.HistogramPlotter(h)
```

Again this looks identical to plot of cluster sizes for the origins labeling.

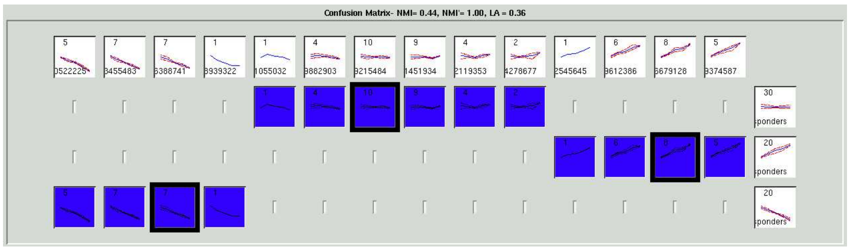


Figure 13: Confusion Matrix Summary of a histogram of values column 4 of the dataset vs origins. Notice here each bin in the histogram corresponds to one and only one origin.

2.3.5 PCA Explorer

This PCAExplorer tool provides tools to better understand the PCA space of the data. Upon initialization of the pcaExplorer you'll see 3 windows. A PCA scatter plot, a list of eigen vectors (or sometimes they're called eigen genes) and a native space projection. The PCA space represents a projection of the data into a space where each dimension or basis represents a linear combination of real dimensions in the dataset. The order of these PCA dimensions are ranked according to the amount of variance they explain or capture. The eigen vector window shows what each PCA dimension is representing. The red "X" in the PCA plot indicates the position in that particular PCA projection that the native-space projection vector is illustrating. You can either middle click on the PCA plot or use the sliders next to each eigen vector to move the indicator "X" around in the PCA space. Note, if you are moving the "X" along a PCA dimension that is not be plotted (by default the first two) the native-space projection will change but the "X" will not move. That is because you move orthoganolly to the two displayed dimensions. Use the pull down menus on the bottom of the PCA display to select the PCA dimensions to plot. Often the first PCA dimension describes uninteresting aspects of the data, such as magnitude of expression.

3 Running with your data

Hopefully in reading through the first section you've become comfortable with the pyMLX schema and IPlot functionality.

3.1 Loading Data

The easiest way to load data is to create a tab-deliminated text file with each line containing just the expression measurements.

```
1.2 .8  0.2  0.4  -1.2  -0.4
-4  0.2  1.2 -0.02  .05 -0.1
```

Now create labeling files with one label per line corresponding to the data file

```
geneA
geneB
```

You can do the same thing for column labels, one label per line correpsonding to the columns

```
time0
time1
time2
time3
```

If you had stored your data in a file called data.dat and the gene names in a file called genes.rlab and the time labels in a file called times.clab (the extension names are only for convention and aren't used by any of the software, but helps keep track of things. dat for dataset files, rlab for row labels and clab for column labels). We can now instantiate a pyMLX dataset with labelings with thing following code:

```
ds = datasets.Dataset('data.dat')

genes = labelings.GlobalWrapper(ds, 'gene names')
genes.labelRows('genes.rlab')

times = labelings.GlobalWrapper(ds, 'time')
times.labelCols('times.clab')
```

It is important to note that the Dataset constructor has is "smart" and can load a dataset from a file or from a numeric array or list of lists as we saw above. The labeling.labelRows or labeling.labelCols method is also smart and will work with a filename or a list of labels for each row. If it is working with a filename, then each label is assumed to be a string and sometimes you may want them to be either int or float labels. This can be easily accomplished using a map. The map function in python apples a function to a list, so its sytex is map(function, list). In the below example imagine we have a file named seconds which contains the number of seconds past some treatment for each column in a dataset. Below the list is created by opening the file and then immediately invoking the readlines command which returns a list of strings where each item in the list is the string representing a line in the file. The map function simply casts each line into a float creating a list of floats which is then passed to the labelCols constructor.

```
secs =labelings.GlobalWrapper(ds, 'seconds')
secs.labelCols(map(float,open('seconds.clab').readlines()))
```