PyEM, a python package for Gaussian mixture models

Tables of contents

PyEM is a package which enables to create Gaussian Mixture Models (diagonal and full covariance matrices supported), to sample them, and to estimate them from data using Expectation Maximization algorithm. It can also draw confidence ellipsoides for multivariate models, and compute the Bayesian Information Criterion to assess the number of clusters in the data. In a near future, I hope to add so-called online EM (ie recursive EM) and variational Bayes implementation.

PyEM is implemented in python, and uses the excellent numpy and scipy packages. Numpy is a python packages which gives python a fast multi-dimensional array capabilities (ala matlab and the likes); scipy leverages numpy to build common scientific features for signal processing, linear algebra, statistics, etc...

Installation

Pyem depends on several packages to work:

- *numpy*
- *matplotlib (if you wish to use the plotting facilities of pyem)*

Those packages are likely to be already installed in a typical numpy/scipy environment.

Since september 2006, pyem is included in the sandbox of scipy. The sandbox contains packages which are pending for approval in main scipy; that means it is not installed by default, and that you need to install scipy from sources. For the most up-to-date version of pyem, you need to download scipy from subversion, which contains the development branch of scipy.

To install pyem, you just need to edit (or create if it does not exist) the file Lib/sandbox/enabled_packages.t in scipy sources, and add one line with the name of the package (eg pyem). After, you just need to install scipy normally as explained here.

You can (and should) also test pyem installation using the following:

basic usage

Once you are inside a python interpreter, you can import pyem using the follwing command:

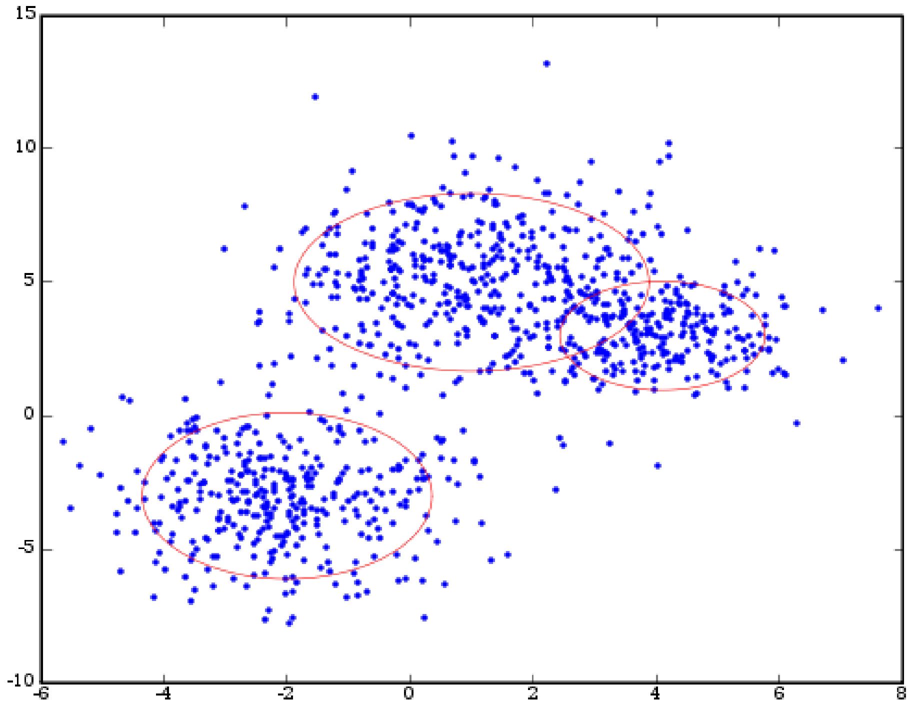
Creating, sampling and plotting a mixture

Importing pyem gives access to 3 classes: GM (for Gausssian Mixture), GMM (Gaussian Mixture Model) and EM (for Expectation Maximization). The first class GM can be used to create an artificial Gaussian Model, samples it, or plot it. The following example show how to create a 2 dimension Gaussian Model with 3 components, sample it and plot its confidence ellipsoids with matplotlib:

```
import numpy as N
import pylab as P
from scipy.sandbox.pyem import GM

#------------------------------
# Hyper parameters:
#   - K:    number of clusters
#   - d:    dimension
k   = 3
d   = 2

#-------------------------------------------------------
# Values for weights, mean and (diagonal) variances
#   - the weights are an array of rank 1
#   - mean is expected to be rank 2 with one row for one component
#   - variances are also expteced to be rank 2. For diagonal, one row
#   is one diagonal, for full, the first d rows are the first variance,
```

```
#   etc... In this case, the variance matrix should be k*d rows and d
#   colums
w   = N.array([0.2, 0.45, 0.35])
mu  = N.array([[4.1, 3], [1, 5], [-2, -3]])
va  = N.array([[1, 1.5], [3, 4], [2, 3.5]])

#-----------------------------------------
# First method: directly from parameters:
# Both methods are equivalents.
gm      = GM.fromvalues(w, mu, va)

#-------------------------------------
# Second method to build a GM instance:
gm      = GM(d, k, mode = 'diag')
# The set_params checks that w, mu, and va corresponds to k, d and m
gm.set_param(w, mu, va)

# Once set_params is called, both methods are equivalent. The 2d
# method is useful when using a GM object for learning (where
# the learner class will set the params), whereas the first one
# is useful when there is a need to quickly sample a model
# from existing values, without a need to give the hyper parameters

# Create a Gaussian Mixture from the parameters, and sample
# 1000 items from it (one row = one 2 dimension sample)
data    = gm.sample(1000)

# Plot the samples
P.plot(data[:, 0], data[:, 1], '.')
# Plot the ellipsoids of confidence with a level a 75 %
gm.plot(level = 0.75)
```

which plots this figure:



There are basically two ways to create a GM instance: either an empty one (eg without mean, weights and covariances) by giving hyper parameters (dimension, number of clusters, type of covariance matrices) during instanciation, or giving all parameters using the class method GM.fromvalues. The first method is mostly useful as a container for learning. There are also methods to create random (but valid !) parameters for a Gaussian Mixture: this can be done by the function method GM.generate_params (a class method).

Basic estimation of mixture parameters from data

If you want to learn a Gaussian mixture from data with EM, you need to use two classes from pyem: GMM and EM. You first create a GMM object from a GM instance; then you can give the GMM object as a parameter to the EM class to compute iterations of EM; once the EM has finished the computation, the GM instance of GMM contains the computed parameters.

```
from numpy.random import seed

from scipy.sandbox.pyem import GM, GMM, EM
import copy
```

```python
# To reproduce results, fix the random seed
seed(1)

#+++++++++++++++++++++++++++++
# Meta parameters of the model
#   - k: Number of components
#   - d: dimension of each Gaussian
#   - mode: Mode of covariance matrix: full or diag (string)
#   - nframes: number of frames (frame = one data point = one
#   row of d elements)
k       = 2
d       = 2
mode    = 'diag'
nframes = 1e3

#+++++++++++++++++++++++++++++++++++++++++++
# Create an artificial GM model, samples it
#+++++++++++++++++++++++++++++++++++++++++++
w, mu, va   = GM.gen_param(d, k, mode, spread = 1.5)
gm          = GM.fromvalues(w, mu, va)

# Sample nframes frames  from the model
data    = gm.sample(nframes)

#++++++++++++++++++++++++
# Learn the model with EM
#++++++++++++++++++++++++

# Create a Model from a Gaussian mixture with kmean initialization
lgm = GM(d, k, mode)
gmm = GMM(lgm, 'kmean')

# The actual EM, with likelihood computation. The threshold
# is compared to the (linearly appromixated) derivative of the likelihood
em      = EM()
like    = em.train(data, gmm, maxiter = 30, thresh = 1e-8)

# The computed parameters are in gmm.gm, which is the same than lgm
# (remember, python does not copy most objects by default). You can for example
# plot lgm against gm to compare
```

GMM class do all the hard work for learning: it can compute the sufficient statistics given the current state of the model, and update its parameters from the sufficient statistics; the EM class uses a GMM instance to compute several iterations. The idea is that you can implements a different mixture model and uses the same EM class if you want (there are several optimized models for GMM, which are subclasses of GMM).

Advanced topics

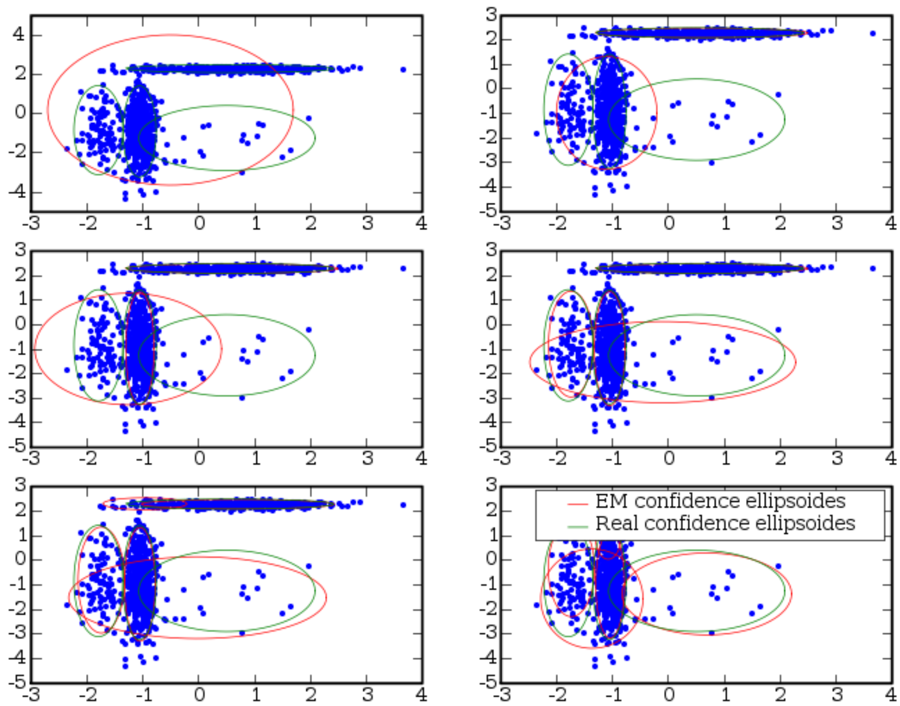
Bayesian Information Criterion and automatic clustering

The GMM class is also able to compute the bayesian information criterion (BIC), which can be used to assess the number of clusters into the data. It was first suggested by Schwarz (see bibliography), who gave a Bayesian argument for adopting the BIC. The BIC is derived from an approximation of the integrated likelihood of the model, based on regularity assumptions. The following code generates an artificial mixture of 4 clusters, runs EM with models of 1 to 6 clusters, and prints which number of clusters is the most likely from the BIC:

```
import numpy as N
from numpy.random import seed

from scipy.sandbox.pyem import GM, GMM, EM
import copy

seed(2)

k       = 4
d       = 2
mode    = 'diag'
nframes = 1e3

#+++++++++++++++++++++++++++++++++++++++++++
# Create an artificial GMM model, samples it
#+++++++++++++++++++++++++++++++++++++++++++
w, mu, va    = GM.gen_param(d, k, mode, spread = 1.0)
gm           = GM.fromvalues(w, mu, va)

# Sample nframes frames  from the model
data    = gm.sample(nframes)

#++++++++++++++++++++++++
# Learn the model with EM
#++++++++++++++++++++++++
```

```
# List of learned mixtures lgm[i] is a mixture with i+1 components
lgm     = []
kmax    = 6
bics    = N.zeros(kmax)
em      = EM()
for i in range(kmax):
    lgm.append(GM(d, i+1, mode))

    gmm = GMM(lgm[i], 'kmean')
    em.train(data, gmm, maxiter = 30, thresh = 1e-10)
    bics[i] = gmm.bic(data)

print "Original model has %d clusters, bics says %d" % (k, N.argmax(bics)+1)

#+++++++++++++++
# Draw the model
#+++++++++++++++
import pylab as P
P.subplot(3, 2, 1)

for k in range(kmax):
    P.subplot(3, 2, k+1)
    level   = 0.9
    P.plot(data[:, 0], data[:, 1], '.', label = '_nolegend_')

    # h keeps the handles of the plot, so that you can modify
    # its parameters like label or color
    h   = lgm[k].plot(level = level)
    [i.set_color('r') for i in h]
    h[0].set_label('EM confidence ellipsoides')

    h   = gm.plot(level = level)
    [i.set_color('g') for i in h]
    h[0].set_label('Real confidence ellipsoides')

P.legend(loc = 0)
# depending on your configuration, you may have to call P.show()
# to actually display the figure
```

which plots this figure:



The above example also shows that you can control the plotting parameters by using returned handles from plot methods. This can be useful for complex drawing.

Examples

TODO.

Using EM for clustering

Using PyEM for supervised learning

Note on performances

Pyem is implemented in python (100% of the code has a python implementation), but thanks to the Moore Law, it is reasonably fast so that it can be used for other problems than toys problem. On my computer (linux on bi xeon 3.2 Ghz, 2Gb RAM), running 10 iterations of EM algorithm on 100 000 samples of dimension 15, for a diagonal model with 30 components, takes around 1 minute and 15 seconds: this makes the implementation usable for moderately complex problems such as speaker recognition using MFCC. If this is too slow, there is a C implementation for Gaussian densities which can be enabled by commenting out one line in pyem/gmm_em.py, which should gives a speed up of a factor 2 at least; this has not been tested much, though, so beware.

Also, increasing the number of components and/or dimension is much more expensive than increasing the number of samples: a 5 dimension model of 100 components will be much slower to estimate with 500 samples than a 5 dimension 10 components with 5000 samples. This is because loops on dimension/components are in python, whereas loops on samples are in C (through numpy). I don't think there is an easy fix to this problem.

Full covariances will be slow, because you cannot avoid nested loop in python this case without insane amount of memory. A C implementation may be implemented, but this is not my top priority; most of the time, you should avoid full covariance models anyway.

TODO

I believe the current API simple and powerful enough, except maybe for plotting (if you think otherwise, I would be happy to hear your suggestions). Now, I am considering adding some more functionalities to the toolbox:

- *add simple methods for regularization of covariance matrix (easy)*
- *add bayes prior (using variational Bayes approximation) for overfitting and model selection problems (not trivial, but doable)*
- *improve online EM*

Other things which are doable but which I don't intend to implement are:

- *add other models (mixtures of multinomial: easy, simple HMM: easy, other ?)*
- *add bayes prior using MCMC (hard, use PyMCMC for sampling ?)*

Bibliography

TODO.