

PyEM, a python package for Gaussian mixture models

Tables of contents

Installation	1
basic usage	2
Creating, sampling and plotting a mixture	2
Basic estimation of mixture parameters from data	4
Advanced topics	6
Bayesian Information Criterion and automatic clustering	6
Examples	8
Using EM for pdf estimation	8
Using EM for clustering	10
Using PyEM for supervised learning	10
Note on performances	14
TODO	15
Bibliography	15

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 multi-variate 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.` 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 following command:

Creating, sampling and plotting a mixture

Importing *pyem* gives access to 3 classes: GM (for Gaussssian 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
# columns
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 equivalent.
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)

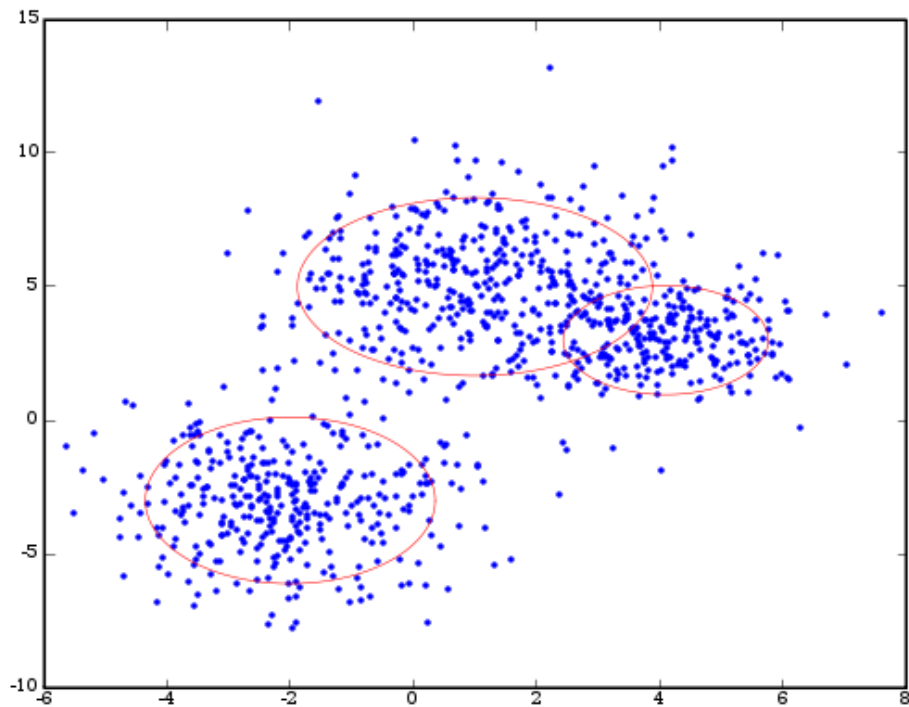
# Once set_params is called, both methods are equivalent. The 2d
# method is useful when using a GM object for learning (where
# the learner class will set the params), whereas the first one
# is useful when there is a need to quickly sample a model
# from existing values, without a need to give the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

```

# 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 approximated) 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_')

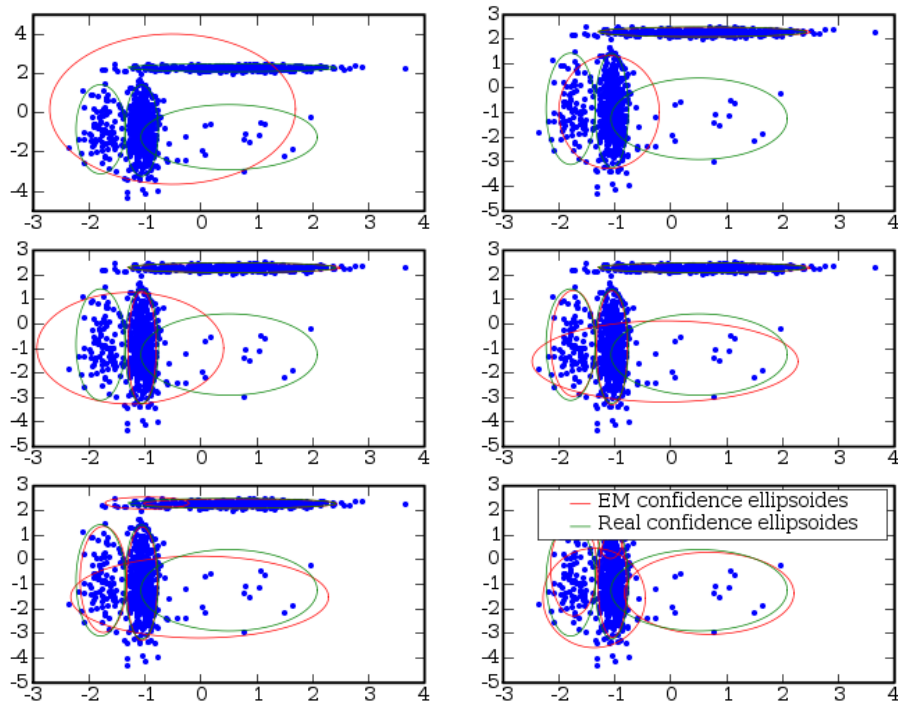
    # h keeps the handles of the plot, so that you can modify
    # its parameters like label or color
    h = lgm[k].plot(level = level)
    [i.set_color('r') for i in h]
    h[0].set_label('EM confidence ellipsoides')

    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

Using EM for pdf estimation

The following example uses the old faithful dataset and is available in the example directory. It models the joint distribution $(d(t), w(t+1))$, where $d(t)$ is the duration time, and $w(t+1)$ the waiting time for the next eruption. It selects the best model using the BIC.

```
#!/usr/bin/env python
# Last Change: Mon Jul 02 03:00 PM 2007 J

# Example of doing pdf estimation with EM algorithm. Requires matplotlib.
import numpy as N
import pylab as P

from scipy.sandbox import pyem
```



```

import utils

oldfaithful = utils.get_faithful()

# We want the relationship between  $d(t)$  and  $w(t+1)$ , but get_faithful gives
#  $d(t)$ ,  $w(t)$ , so we have to shift to get the "usual" faithful data
waiting = oldfaithful[1:, 1:]
duration = oldfaithful[:len(waiting), :1]
dt = N.concatenate((duration, waiting), 1)

# Scale the data so that each component is in [0..1]
dt = utils.scale(dt)

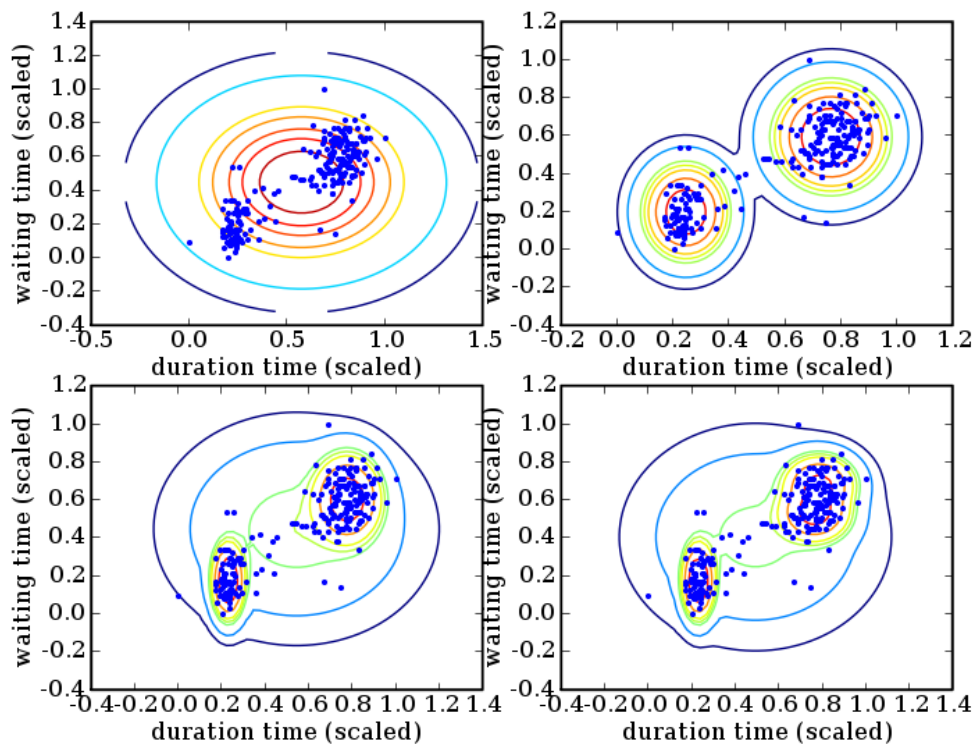
# This function train a mixture model with k components, returns the trained
# model and the BIC
def cluster(data, k, mode = 'full'):
    d = data.shape[1]
    gm = pyem.GM(d, k, mode)
    gmm = pyem.GMM(gm)
    em = pyem.EM()
    em.train(data, gmm, maxiter = 20)
    return gm, gmm.bic(data)

# bc will contain a list of BIC values for each model trained
bc = []
mode = 'full'
P.figure()
for k in range(1, 5):
    # Train a model of k component, and plots isodensity curve
    P.subplot(2, 2, k)
    gm, b = cluster(dt, k = k, mode = mode)
    bc.append(b)

    X, Y, Z, V = gm.density_on_grid()
    P.contour(X, Y, Z, V)
    P.plot(dt[:, 0], dt[:, 1], '.')
    P.xlabel('duration time (scaled)')
    P.ylabel('waiting time (scaled)')

print "According to the BIC, model with %d components is better" % (N.argmax(bc) + 1)

```



isodensity curves for the old faithful data modeled by a 1, 2, 3 and 4 components model (up to bottom, left to right).

Using EM for clustering

TODO (this is fundamentally the same than pdf estimation, though)

Using PyEM for supervised learning

The following example shows how to do classification using discriminative analysis on the famous iris dataset. The idea is to model each class distribution $P(\text{data}|\text{class})$ by a mixture of Gaussian, and to classify the data with Maximum A Posteriori (eg takes the class which maximizes $P(\text{class} | \text{data})$). Not surprisingly, the errors lie on the border between the two classes which are not linearly separable.

```
#!/usr/bin/env python
# Last Change: Mon Jul 02 09:00 PM 2007 J
```

```
__doc__ = """Example of doing classification with mixture of Gaussian. Note
that this is really a toy example: we do not use testing testset nor cross
validation.
```

We use the famous iris database used by Sir R.A. Fisher. You can try to change the attributes used for classification, number of components used for the mixtures, etc..."

```
import numpy as N
import pylab as P
import matplotlib as MPL

from scipy.sandbox import pyem
import utils

data = utils.iris.load()
# cnames are the class names
cnames = data.keys()

#-----
# Data pre processing
#-----
# we use 25 samples of each class (eg half of iris), for
# learning, and the other half for testing. We use sepal width and petal width
# only
ln = 25
tn = 25
xdata = {}
ydata = {}
# learning data
ldata = {}

# you can change here the used attributes (sepal vs petal, width vs height)
for i in cnames:
    xdata[i] = data[i]['sepal width']
    ydata[i] = data[i]['petal width']
    ldata[i] = N.concatenate((xdata[i][:ln, N.newaxis],
                              ydata[i][:ln, N.newaxis]),
                              axis = 1)

tx = N.concatenate([xdata[i][ln:] for i in cnames])
ty = N.concatenate([ydata[i][ln:] for i in cnames])
tdata = N.concatenate((tx[:, N.newaxis], ty[:, N.newaxis]), axis = 1)

# tclass makes the correspondance class <-> index in the testing data tdata
tclass = {}
for i in range(3):
```

```

tclass[cnames[i]] = N.arange(tn * i, tn * (i+1))

#-----
# Learning and classification
#-----
# This function train a mixture model with k components
def cluster(data, k, mode = 'full'):
    d = data.shape[1]
    gm = pyem.GM(d, k, mode)
    gmm = pyem.GMM(gm)
    em = pyem.EM()
    em.train(data, gmm, maxiter = 20)
    return gm

# Estimate each class with a mixture of nc components
nc = 2
mode = 'diag'
lmod = {}
for i in cnames:
    lmod[i] = cluster(ldata[i], nc, mode)

# Classifiy the testing data. Of course, the data are not really IID, because
# we did not shuffle the testing data, but in this case, this does not change
# anything.
p = N.empty((len(tdata), 3))
for i in range(3):
    # For each class, computes the likelihood for the testing data
    p[:, i] = lmod[cnames[i]].pdf(tdata)

# We then take the Maximum A Posteriori class (same than most likely model in
# this case, since each class is equiprobable)
cid = N.argmax(p, 1)
classification = {}
for i in range(3):
    classification[cnames[i]] = N.where(cid == i)[0]

correct = {}
incorrect = {}
for i in cnames:
    correct[i] = N.intersect1d(classification[i], tclass[i])
    incorrect[i] = N.setdiff1d(classification[i], tclass[i])

#-----
# Plot the results

```

```

#-----
csym = {'setosa' : 's', 'versicolor' : 'x', 'virginica' : 'o'}
r = 50.
P.figure(figsize = [600/r, 400/r])

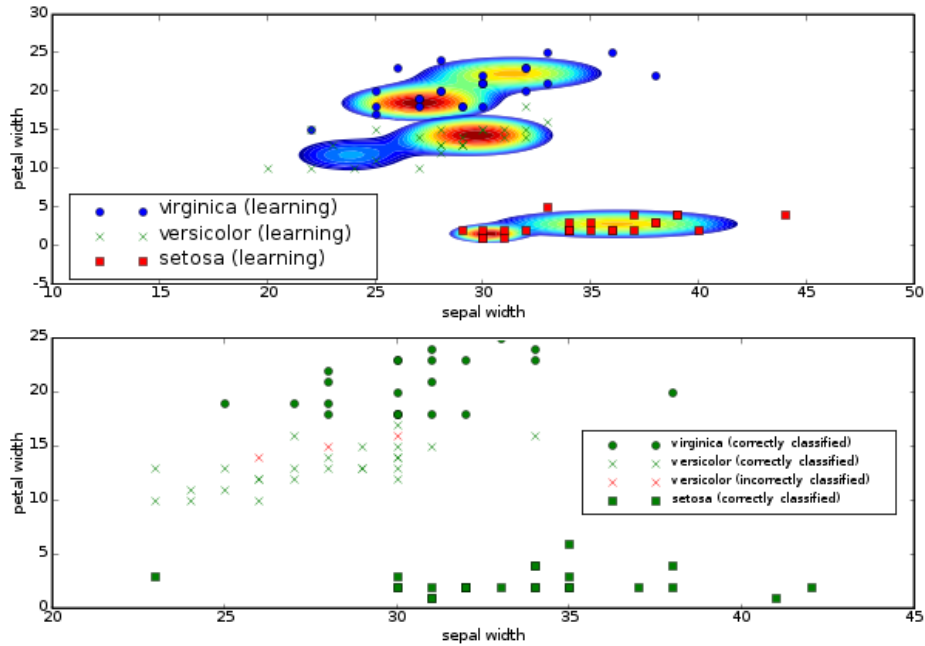
prop = MPL.font_manager.FontProperties(size='smaller')

# Plot the learning data with the mixtures
P.subplot(2, 1, 1)
for i in lmod.values():
    #i.plot()
    X, Y, Z, V = i.density_on_grid()
    P.contourf(X, Y, Z, V)

for i in cnames:
    P.plot(ldata[i][:, 0], ldata[i][:, 1], csym[i], label = i + ' (learning)')
P.xlabel('sepal width')
P.ylabel('petal width')
P.legend(loc = 'best')

# Plot the results on test dataset (green for correctly classified, red for
# incorrectly classified)
P.subplot(2, 1, 2)
for i in cnames:
    P.plot(tx[correct[i]], ty[correct[i]], 'g' + csym[i],
           label = '%s (correctly classified)' % i)
    if len(incorrect[i]) > 0:
        P.plot(tx[incorrect[i]], ty[incorrect[i]], 'r' + csym[i],
               label = '%s (incorrectly classified)' % i)
P.legend(loc = 'best', prop = prop)
P.xlabel('sepal width')
P.ylabel('petal width')
P.savefig('dclass.png', dpi = 60)

```



learning data points on the left with isodensity curves as estimated by each mixture. On the right, the misclassified points are in red, green are correctly classified.

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.