Final Project - STA 250, Fall 2013

Dirichlet Process Mixture Model Classifier with Gibbs Sampling Method

Yuanzhe(Roger) Li, 997083127

1 Introduction

In the context of machine learning, a Gaussian Mixture Model(GMM) is a probabilistic model for classification tasks that assumes all the data points are generated from a mixture of a finite number of Gaussian distributions with unknown parameters. Mixture models magage clustering by incorporating information about the covariance structure of the data as well as the centers of the latent Gaussians.

The problem with GMMs is that they assume a fixed number of clusters, which they need to be told to find, while most real-world data simply doesnt have a fixed number of clusters. Instead, a type of classifier caleed Dirichlet Process Mixture Model(DPMM) is often used for data classification with unknow (possibly infinite) number of clusters.

In this project, a particular type of Gibbs sampling algorithm is implemented in python for DPMM with conjugate priors. A simple example is given to compare Gibbs Sampling DPMM and GMM (The latter method is implemented by directly using the scikit-learn package in python[3]).

2 Methodology

The fundamental generative story for finding clusters in any set of data is to assume an infinite set of latent groups, where each group is described by some set of parameters. For example, each group could be a Gaussian with a specified mean μ_i and standard deviation σ_i, and these group parameters themselves are assumed to come from some base distribution G_0. Data is then generated in the following manner:

- Select a cluster.
- Sample from that cluster to generate a new point.

In different settings, this clustering process has different stories. Let's approach the final DPMM model from its relevant processes.

2.1 Dirichlet Process and relative representations

Formally, given a base distribution G_0 and a dispersion parameter α, a sample from the Dirichlet Process $DP(G_0, \alpha)$ is a distribution $G \sim \mathrm{DP}(G_0, \alpha)$. A DP can be viewed from different perspectives as discussed in the following subsections.

2.1.1 Chinese restaurant process (CRP)

This process works as follows: Imagine a restaurant with unlimited tables.

• Initially the restaurant is empty.
• The first person to enter sits down at a table (selects a group). He or she then orders food for the table (i.e., she selects parameters for the group); everyone else who joins the table will then be limited to eating from the food she ordered.
• The second person to enter sits down at a table. With probability $\frac{\alpha}{1+\alpha}$ he sits down at a new table (i.e., selects a new group) and orders food for the table; with probability $\frac{1}{1+\alpha}$ he sits with the first person and eats from the food he or shes already ordered (i.e., hes in the same group as the first person).
...
• The (n+1)-st person sits down at a new table with probability $\frac{\alpha}{n+\alpha}$, and at table k with probability $\frac{n_k}{n+\alpha}$, where n_k is the number of people currently sitting at table k.

Note that the probability of a new group depends on α, which could be treated as a dispersion parameter that affects the dispersion of the datapoints. The lower α is, the more tightly clustered the data points; the higher

it is, the more clusters we have in any finite set of points.
To summarize, given n data points, CRP specifies a distribution over partitions (table assignments) of the points.

2.1.2 Polya urn model

The Polya Urn model is very similar to the Chinese Restaurant Process:

• Start with an urn containing $\alpha G_0(x)$ balls of color x, for each possible value of x. (G_0 is the base distribution, and $G_0(x)$ is the probability of sampling x from G_0).
• At each time step, draw a ball from the urn, note its color, and then drop both the original ball plus a new ball of the same color back into the urn.

The connection between this process and the CRP is that balls correspond to people as data points, colors correspond to table assignments as clusters, α is also a dispersion but a prior parameter for the Polya urn model.
The difference between the CRP and the Polya Urn Model is that the CRP specifies only a distribution over partitions (i.e., table assignments), but doesnt assign parameters to each group, whereas the Polya Urn Model does both.

2.1.3 Stick-breaking process

By running either the Chinese Restaurant Process or the Polya Urn Model without stop. For each group i, it gives a proportion π_i of points that fall into group i.
Instead of running the CRP or Polya Urn model to figure out these proportions, it could be generated from the Stick-Breaking Process:

• Start with a stick of length one.
• Generate a random variable $\beta_1 \sim \text{Beta}(1, \alpha)$. β_1 will be a real number

between 0 and 1 with expected value $\frac{1}{1+\alpha}$. Break off the stick at β_1; π_1 is then the length of the stick on the left.
• Now take the stick to the right, and generate $\beta_2 \sim \text{Beta}(1, \alpha)$. Break off the stick β_2 into the stick. Now π_2 is the length of the stick to the left, i.e., $\pi_2 = (1 - \beta_1)\beta_2$.
...
• So on and so forth.

The stick-breaking process could be summarized as:

$$\beta_k \sim \text{Beta}(1, \alpha)$$
$$\pi_k = \beta_k \prod_{l=1}^{k-1} (1 - \beta_l)$$

We denote $\pi = (\pi_k)_{k=1}^{\infty}$ as $\pi \sim \text{GEM}(\alpha)$

2.2 DPMM: the model

Assume we have N observations and K clusters, $i \in [1, N]$ is the indice for the observations, while $k \in [1, K]$ is the indice for the clusters. With z_i the cluster assignment of observation x_i, and θ_k the parameter of mixture k:

$$P(x_{1:N}) = \prod_{i=1}^{N} \sum_{k=1}^{K} P(x_i|\theta_{z_i}) P(z_i = k)$$

The conditional distributions of a DPGMM is as follows:

$\pi|\alpha \sim \text{GEM}(1, \alpha)$ (Mixing rate)
$z_i|\pi \sim \pi$ (cluster assignments)
$\theta_k|\lambda \sim G_0(\lambda)$ (cluster parameters)
$x_i|z_i, \{\theta_k\}_{k=1}^{\infty} \sim F(\theta_{z_i})$ (data values)

In this model, each datum x_i is generated by first selecting a cluster k, according to the multinomial distribution that is parameterized by π, and then sampling from the distribution of this cluster $F(\theta_{z_i})$ that is parameterized by θ_k. The mixture weight π is given a symmetric Dirichlet prior with a hyperparameter α, and the cluster parameters θ_k are given a common prior distribution $G_0(\lambda)$ with parameter λ. In practice, $F(\theta)$ is typically some exponential family of densities, and $G_0(\lambda)$ a corresponding conjugate prior.
Now we decompose the probability that the observation i belongs to cluster k into its two independent factors:

$$P(z_i = k|z_{i-1}, x, \alpha, \lambda) \propto P(z_i = k|z_{-i}, \alpha)P(x_i|x_{-i}, z_i = k, z_{-i}, \lambda)$$

The connection betweeen the DPMM and the Chinese restaurant process can now be illustrated from the conditional distributions of the indicator variables as:

$$P(z_i = k|z_{-i}, \alpha) = \begin{cases} \dfrac{N_{k,-i}}{\alpha + N - 1} & \text{if } k \text{ has been seen before} \\ \dfrac{\alpha}{\alpha + N - 1} & \text{if } k \text{ is a new cluster} \end{cases}$$

This above equation is derived by Yu in [3] by equation (4.1 - 4.11).
And

$$\begin{aligned} P(x_i|x_{-i}, z_i = k, z_{-i}, \lambda) &= P(x_i|x_{-i,k}, \lambda) \\ &= \frac{P(x_i, x_{-i,k}, \lambda)}{P(x_{-i,k}|\lambda)} \end{aligned}$$

Further,

$$P(x_i|x_{-i,k}, \lambda) = \int P(x_i|\theta_k)[\prod_{j \neq i, z_j = k} P(x_j|\theta_k)]G_0(\theta_k|\lambda)d\theta_k$$

which is the marginal probability of all the data being assigned to cluster k, including i.
For new clusters where we denote $z_i = k^*$, we have

$$P(x_i|x_{-i}, z_i = k^*, z_{-i}, \lambda) = \int P(x_i|\theta)G_0(\theta|\lambda)d\theta$$

In our assumption, F and G_0 are conjugate priors, also we want F to be multivariate normal, i.e., $F \sim \mathcal{N}(\mu, \Sigma)$ with μ and Σ unkown. In this case, G_0 is chosen to be normal-inverse-Wishart with prior parameters as follows:

μ_0: initial mean guess
κ_0: mean fraction smoothing parameter
ν_0: degrees of freedom
Ψ_0: pairwise deviation product matrix.

Note that the above initial parameters are something that we could play with depending on the dataset. In the simple example I setup in the next section. I set μ_0 to the mean of the whole datset, ν_0 to the number of dimensions, Ψ_0 to the identity matrix times a factor of 5. These initializations could indeed change the performance of the DPMM classifier.
According to Yu [2], the MAP estimates onparameters give us the following update:
$\mu_n = \dfrac{\kappa_0\mu_0 + nx}{\kappa_0 + n} = \mu$
$\kappa_n = \kappa_0 + n$
$\nu_n = \nu_0 + n$
$\Psi_n = \Psi_0 + \sum_{i=1}^{n}(x_i - \bar{x})(x_i - \bar{x})^T + \dfrac{\kappa_0 n}{\kappa_0 + n}(\bar{x} - \mu_0)(\bar{x} - \mu_0)^T$
$\Sigma = \dfrac{\kappa_n + 1}{\kappa_n(\nu_n - d + 1)}\Psi_n$

Then we have the posterior predictive for cluster k evaluated at x_i:

$$P(x_i|x_{-i}, z_i = k, z_{-i}, \lambda) \propto \mathcal{N}(\mu_{k,-i}, \Sigma_{k.-i})$$

2.3 Collapsed Gibbs sampling for DPMM

Basically, the Gibbs sampling approach works as follows:

• Take the set of data points, and randomly initialize group assignments.
• Pick a point. Fix the group assignments of all the other points, and assign the chosen point a new group (which can be either an existing cluster or a new cluster) with a probability as described in the models above that depends on the group assignments and values of all the other points.
• Repeat the previous step until convergence.

Neal(2000) [1] provides a description of the algorithm, Yu(2009) [2] provids more detailed schematics of the Collapsed Gibbs samling algorithm in **Algorithm 6**.
The Collapsed Gibbs Sampling algorithm adapted from Neal(2000) and Yu(2009) is as follows:

For each iteration before convergence:

1. for $i = 1 : N$ in random order:
   (a) remove x_i's sufficient statistics from cluster z_i
   (b) if it empties the cluster, remove the cluster and decrease K
   (c) for $k = 1 : K$:
      i. compute the predictive likelihood $P_k(x_i) = P(x_i|x_{-i} = k)$

ii. set $N_{k,-i} = dim(x_{-i} = k)$ and compute $P(z_i = k|z_{-i}, \alpha) = \frac{N_{k,-i}}{\alpha + N - 1}$

(d) compute $P^*(x_i) = P(x_i|\lambda)$

(e) compute predictive likelihood of potential new cluster $P(z_i = k^*|z_{-i}, \alpha) = \frac{\alpha}{\alpha + N - 1}$

(f) normalize $P(z_i|...)$

(g) sample $z_i$from $P(z_i|...)$

(h) add x_i's sufficient statistics to new cluster z_i

(i) if it's a new cluster, increase K

3 Results

A simple example with 2-dimensional Gaussian input data is used to compare the DPMM with Collapsed Gibbs sampling and GMM. Specific data setup and plotting codes are referencing to Scikit-learn: Gaussian Mixture Model Ellipsoids examole [4].
The data are actually generated as two clusters, each with five components to fit, the DPMM with Collapsed Gibbs Sampling successfully recognize the true clustering of the dataset, while GMM's clustering depends on the prior believe on the number of components of the data, in this case, it seems to randomly split the data points in some area and fit more components than necessary.

4 Conclusion

In this project, I used DPMM with Gibbs Sampling to solve data clustering task with unknown number of patterns, implemented the algorithm on a small dataset and compared it with original GMM method provided by

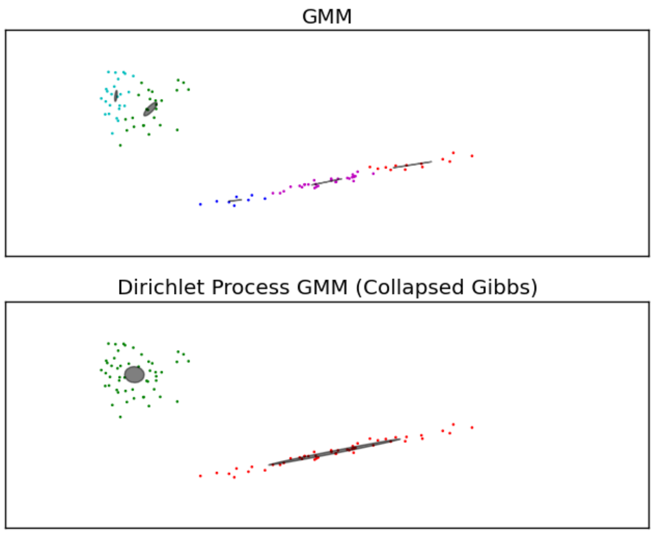

Figure 1: Comparison of GMM and DPMM

scikit-learn. Thanks to Yu's personal technical review ([2] and [3]) and all sorts of online resources such as wikipedia so I could have a better understanding at the details of variaous representations of Dirichlet Process and it's conceptional generative story in the context of maching learning.

5 Possible Extention and Self-Criticism

Hugely constrained by the time put in this project, I haven't had a chance to implement and compare more alogrithms presented in Neal(2000)[1] and Yu(2009)[2], which definitely would be worth doing. Another possible extension is to examine these algorithms to different set of real-world data, and delve into more of the tunning process of model parameters, and try to understand how different variations of DPMM classifier's behavior corresponding to different inherent structures of the dataset.

References

[1] R. M. Neal. Markov Chain Sampling Methods for Dirichlet Process Mixture Models. *Journal of Computational and Graphical Statistics*, 9:249-265,2000. (`http://www.stat.purdue.edu/~rdutta/24.PDF`)

[2] Yu Xiaodong, Gibbs Sampling Methods for Dirichlet Process Mixture Model: Technical Details. 2009. (`http://yuxiaodong.files.wordpress.com/2009/09/technical-details-in-gibbs-sampling-for-dp-mixture-model.pdf`)

[3] Yu Xiaodong, Derivation of Gibbs Sampling for Finite Gaussian Mixture Model. 2009. (`http://yuxiaodong.files.wordpress.com/2009/09/derivation_of_gibbs_sampling_gmm1.pdf`)

[4] Scikit-learn: Machine Learning in Python - Gaussian Mixture Model Ellipsoids. (`http://scikit-learn.org/stable/auto_examples/mixture/plot_gmm.html#example-mixture-plot-gmm-py`)

Appendix

Python code:

```
# -*- coding: utf-8 -*-
import itertools, random

import numpy as np
from scipy import linalg
import pylab as pl
import matplotlib as mpl
import math

epsilon = 10e-8
max_iter = 1000

class Gaussian:
    def __init__(self, X=np.zeros((0,1)), \
    kappa_0=0, nu_0=1.0001, mu_0=None,
            Psi_0=None):
        self.n_points = X.shape[0]
        self.n_var = X.shape[1]

        self._hash_covar = None
        self._inv_covar = None

        if mu_0 == None: # initial mean for the cluster
            self._mu_0 = np.zeros((1, self.n_var))
        else:
            self._mu_0 = mu_0
        self._kappa_0 = kappa_0 # mean fraction

        self._nu_0 = nu_0 # degrees of freedom
        if self._nu_0 < self.n_var:
            self._nu_0 = self.n_var

        if Psi_0 == None:
            self._Psi_0 = 5*np.eye(self.n_var)
            #  this 5  factor should be a prior,
            #~ dependent on the dataset
        else:
            self._Psi_0 = Psi_0
        assert(self._Psi_0.shape == (self.n_var, self.n_var))

        if X.shape[0] > 0:
            self.fit(X)
        else:
            self.default()


    def default(self):
        self.mean = np.matrix(np.zeros((1, self.n_var)))
        self.covar =  np.matrix(np.eye(self.n_var))


    def recompute_ss(self):
        """ need to have actualized _X, _sum, and _square_sum """
        self.n_points = self._X.shape[0]
        self.n_var = self._X.shape[1]
        if self.n_points <= 0:
            self.default()
            return
        kappa_n = self._kappa_0 + self.n_points
        nu = self._nu_0 + self.n_points
        mu = np.matrix(self._sum) / self.n_points
        mu_mu_0 = mu - self._mu_0

        C = self._square_sum - self.n_points * \
        (mu.transpose() * mu)
        Psi = (self._Psi_0 + C + self._kappa_0 * self.n_points
             * mu_mu_0.transpose() * mu_mu_0 /\
              (self._kappa_0 + self.n_points))

        self.mean = ((self._kappa_0 * self._mu_0 + self.n_points * mu)
                    / (self._kappa_0 + self.n_points))
        self.covar = (Psi * (kappa_n + 1)) / \
        (kappa_n * (nu - self.n_var + 1))


    def inv_covar(self):
        """ memoize the inverse of the covariance matrix """
        if self._hash_covar != hash(self.covar):
            self._hash_covar = hash(self.covar)
            self._inv_covar = np.linalg.inv(self.covar)
        return self._inv_covar


    def fit(self, X):
        """ to add several points at once without recomputing """
        self._X = X
        self._sum = X.sum(0)
        self._square_sum = np.matrix(X).transpose() * np.matrix(X)
        self.recompute_ss()
```

```
def add_point(self, x):
    """ add a point to this Gaussian cluster """
    if self.n_points <= 0:
        self._X = np.array([x])
        self._sum = self._X.sum(0)
        self._square_sum = np.matrix(self._X).\
        transpose() * np.matrix(self._X)
    else:
        self._X = np.append(self._X, [x], axis=0)
        self._sum += x
        self._square_sum += np.matrix(x).\
        transpose() * np.matrix(x)
    self.recompute_ss()


def rm_point(self, x):
    """ remove a point from this Gaussian cluster """
    # Find the indice of the point x in self._X, be careful with
    indices = (abs(self._X - x)).argmin(axis=0)
    indices = np.matrix(indices)
    ind = indices[0,0]
    for ii in indices:
        if (ii-ii[0] == np.zeros(len(ii))).all():
        # ensure that all coordinates match \
        #(finding [1, 1] in [[1, 2], [1, 1]]\
        # would otherwise return indice 0)
            ind = ii[0,0]
            break
    tmp = np.matrix(self._X[ind])
    self._sum -= self._X[ind]
```

```
        self._X = np.delete(self._X, ind, axis=0)
        self._square_sum -= tmp.transpose() * tmp
        self.recompute_ss()


    def pdf(self, x):
        """ probability density function for a multivariate Gaussian """
        size = len(x)
        #assert(size == self.mean.shape[1])
        #assert((size, size) == self.covar.shape)
        det = np.linalg.det(self.covar)
        #assert(det != 0)
        norm_const = 1.0 / (math.pow((2*np.pi), float(size)/2)
                * math.pow(det, 1.0/2))
        x_mu = x - self.mean
        inv = self.covar.I
        result = math.pow(math.e, -0.5 * (x_mu * inv * x_mu.transpose()))
        return norm_const * result



class DPMM:
    def _get_means(self):
        return np.array([g.mean for g in self.params.itervalues()])


    def _get_covars(self):
        return np.array([g.covar for g in self.params.itervalues()])


    def __init__(self, n_components=-1, alpha=1.0):
        self.params = {0: Gaussian()}
        self.n_components = n_components
        self.means_ = self._get_means()
        self.alpha = alpha


    def fit_collapsed_Gibbs(self, X):
        """ according to Neal(2000) and Xu(2009) """
        mean_data = np.matrix(X.mean(axis=0))
        self.n_points = X.shape[0]
        self.n_var = X.shape[1]
        self._X = X
        if self.n_components == -1:
            # initialize with 1 cluster for each datapoint
            self.params = dict([(i, Gaussian(\
            X=np.matrix(X[i]), mu_0=mean_data))\
             for i in xrange(X.shape[0])])
            self.z = dict([(i,i) for i in range(X.shape[0])])
            self.n_components = X.shape[0]
            previous_means = 2 * self._get_means()
            previous_components = self.n_components
        else:
            # init randomly (or with k-means)
            self.params = dict([(j, Gaussian(X=np.zeros((0, \
            X.shape[1])), mu_0=mean_data)) for \
            j in xrange(self.n_components)])
            self.z = dict([(i, random.randint(0, self.n_components - 1))
                      for i in range(X.shape[0])])
            previous_means = 2 * self._get_means()
            previous_components = self.n_components
            for i in xrange(X.shape[0]):
                self.params[self.z[i]].add_point(X[i])
        print "Initialized␣collapsed␣Gibbs␣sampling␣\
␣␣␣␣␣␣␣␣with␣%i␣cluster" % (self.n_components)

        n_iter = 0
        while (n_iter < max_iter
                and (previous_components != self.n_components
                or abs((previous_means - \
                self._get_means()).sum()) > epsilon)):
            n_iter += 1
            previous_means = self._get_means()
            previous_components = self.n_components

            for i in xrange(X.shape[0]):
                # remove X[i]'s sufficient statistics from z[i]
                self.params[self.z[i]].rm_point(X[i])
                # if it empties the cluster, remove it and decrease K
                if self.params[self.z[i]].n_points <= 0:
                    self.params.pop(self.z[i])
                    self.n_components -= 1

                tmp = []
                for k, param in self.params.iteritems():
                    # compute P_k(X[i]) = P(X[i] | X[-i] = k)
                    marginal_likelihood_Xi = param.pdf(X[i])
                    # set N_{k,-i} = dim({X[-i] = k})
                    # compute P(z[i] = k | z[-i],\
                     Data) = N_{k,-i}/(   +N-1)
                    mixing_Xi = param.n_points / \
                    (self.alpha + self.n_points - 1)
                    tmp.append(marginal_likelihood_Xi * mixing_Xi)

                # compute P*(X[i]) = P(X[i]|   )
```

```
    base_distrib = Gaussian(X=np.zeros\
    ((0, X.shape[1])))
    prior_predictive = base_distrib.pdf(X[i])

    # compute P(z[i] = * | z[-i], Data) =  /( +N-1)
    prob_new_cluster = self.alpha / \
    (self.alpha + self.n_points - 1)
    tmp.append(prior_predictive * prob_new_cluster)

    # normalize P(z[i]) (tmp above)
    s = sum(tmp)
    tmp = map(lambda e: e/s, tmp)

    # sample z[i] ~ P(z[i])
    rdm = np.random.rand()
    total = tmp[0]
    k = 0
    while (rdm > total):
        k += 1
        total += tmp[k]
    # add X[i]'s sufficient statistics to cluster z[i]
    new_key = max(self.params.keys()) + 1
    if k == self.n_components: # create a new cluster
        self.z[i] = new_key
        self.n_components += 1
        self.params[new_key] = Gaussian(X=np.matrix(X[i]))
    else:
        self.z[i] = self.params.keys()[k]
        self.params[self.params.keys()[k]].add_point(X[i])


print "still sampling, %i clusters currently"\
                % (self.n_components)

            self.means_ = self._get_means()


        def predict(self, X):
            """ produces and returns the \
            clustering of the X data """
            if (X != self._X).any():
                self.fit_collapsed_Gibbs(X)
            mapper = list(set(self.z.values()))
             # to map our clusters id to
            # incremental natural numbers starting at 0
            Y = np.array([mapper.index(self.z[i]) \
            for i in range(X.shape[0])])
            return Y



# Number of samples per component
n_samples = 50

# Generate random sample, two components
np.random.seed(0)

# 2, 2-dimensional Gaussians
C = np.array([[0., -0.1], [2, .5]])
X = np.r_[np.dot(np.random.randn(n_samples, 2), C),
            .7 * np.random.randn(n_samples, 2) + np.array([-6, 3])]
```

```
from sklearn import mixture

# Fit a mixture of gaussians with EM using five components
gmm = mixture.GMM(n_components=5, covariance_type='full')
gmm.fit(X)



dpmm = DPMM(n_components=5) #
# n_components is the number of initial clusters
dpmm.fit_collapsed_Gibbs(X)

color_iter = itertools.cycle(['r', 'g', 'b', 'c', 'm'])

X_repr = X
if X.shape[1] > 2:
    from sklearn import manifold
    X_repr = manifold.Isomap(n_samples/10, \
    n_components=2).fit_transform(X)

for i, (clf, title) in enumerate([(gmm, 'GMM'),
          (dpmm, 'Dirichlet Process GMM (Collapsed Gibbs)')]):
    splot = pl.subplot(2, 1, 1 + i)
    Y_ = clf.predict(X)
    print Y_
    for j, (mean, covar, color) in enumerate(zip(
            clf.means_, clf._get_covars(), color_iter)):

        if not np.any(Y_ == j):
            continue

        pl.scatter(X_repr[Y_ == j, 0], X_repr[Y_ == j, 1], \
        .8, color=color)
```

```
            if clf.means_.shape[len(clf.means_.shape) - 1] == 2:
                # Plot an ellipse to show the Gaussian component
                v, w = linalg.eigh(covar)
                u = w[0] / linalg.norm(w[0])
                angle = np.arctan(u[1] / u[0])
                angle = 180 * angle / np.pi  # convert to degrees
                if i == 1:
                    mean = mean[0]
                ell = mpl.patches.Ellipse(mean, v[0], v[1], 180\
                 + angle, color='k')
                ell.set_clip_box(splot.bbox)
                ell.set_alpha(0.5)
                splot.add_artist(ell)

        pl.xlim(-10, 10)
        pl.ylim(-3, 6)
        pl.xticks(())
        pl.yticks(())
        pl.title(title)


pl.savefig('dpmm.png')
```