Implementation of Two Gibbs Samplers

yu-shen

January 11, 2018

Contents

1 Introduction

This is an exercise to implement collapsed Gibbs samplers and chromatic Gibbs samplers according to specifications defined inGuassian Mixture Model

I present first the module decomposition, software construction (source code), and unit testing. Then I present the experiments with the implementation showing the performance of the samplers.

I use Literate Programming to present the software construction. The source code collapsed_gibbs_sampler.py andchromatic_gibbs_sampler.py are tangled from the documentgibbs-samplers.org. From which, this PDF document is also generated.

2 Top Level Program Layouts

This serves as the architecture description of the programs. I present the top level structure in terms of code blocks of the two programs:

- collapsed_gibbs_sampler.py
- chromatic_gibbs_sampler.py

  The referred blocks will be presented in subsequent sections.

2.1 collapsed_gibbs_sampler: main program

Note, collapsed_gibbs_sampler is instantiated as a generator. It will be used to produces samples.
main-collapsed:

```
########################################
# File: collapsed_gibbs_sampler.py       #
# Copyright Primer Technologies Inc 2017 #
########################################

"""Usage: python collapsed_gibbs_sampler.py [input_filename]"""

<<preamble>>

<<initial_z>>

<<sample_mu>>

<<sample_z_collapsed>>

<<collapsed_gibbs_sampler>>

def main():
    <<setup_data>>

    # Initialize ‘sampler‘ and take a single sample
    # (required to initialize the visualization).
    sampler = collapsed_gibbs_sampler(X, alpha, zeta, sigma, random_state=0)

    <<sample-and-visualize>>

if __name__ == '__main__':
    main()
```

2.2 chromatic_gibbs_sampler: main program

Note, chromatic_gibbs_sampler is instantiated as a generator. It will be used to produces samples.
main-chromatic:

```
########################################
# File: chromatic_gibbs_sampler.py       #
# Copyright Primer Technologies Inc 2017 #
########################################

"""Usage: python chromatic_gibbs_sampler.py [input_filename]"""

<<preamble>>

<<initial_z>>

<<sample_mu>>

<<sample_theta>>

<<sample_z_chromatic>>

<<chromatic_gibbs_sampler>>

def main():
    <<setup_data>>

    # Initialize ‘sampler‘ and take a single sample
    # (required to initialize the visualization).
    sampler = chromatic_gibbs_sampler(X, alpha, zeta, sigma, random_state=0)

    <<sample-and-visualize>>

if __name__ == '__main__':
    main()
```

3 Common Building Blocks

This section presents the common building blocks between the two programs. The blocks are used in the construction of the programs as outlined in the sections of "Top Level Program Layouts".

3.1 initial_z

initial_z:

```
def initial_z(alpha, N, K, random_state):
    """
    Initialize z (component assignments)
    required to boostrap the gibbs sampling.
    alpha: the seed vector for Dirchlet sampling of the component proportions.
    N: the number of data points to which
    the sampling of component means and component assignments are performed.
    K: the assumed number of components in the data to be sampled.
    random_state: the random state used as consistent context
    for random sampling.

    It returns the computed z.
    """
    theta = random_state.dirichlet(alpha)
    m = random_state.multinomial(N, theta)
    z = np.repeat(np.arange(K), m)
    random_state.shuffle(z)
    return z
```

3.2 sample_mu

sample_mu:

```
def sample_mu(z, X, K, D, N, sigma, zeta, random_state):
    """
    Perform the sampling of mu (the component means) in gibbs sample procedure.
    z: the component assignments prior or computed
    in the previous sampling iteration.
    X: the dataset of the data points to be sampled for
    component means and component assignments.
    K: the assumed number of components in the data to be sampled.
    D: the number of attributes of a data points in X.
    N: the number of data points in X to which
    the sampling of component means and component assignments are performed.
    sigma: the standard deviation of the noise to the data points
    zeta: the standard deviation for the Gaussian prior
    over the component means
    random_state: the random state used as consistent context
    for random sampling.

    It returns both the updated mu as numpy array and count_by_component,
    a list of counts of components in X according to z.

    """
    count_by_component = []
    mu = []
    for k in range(K):
        X_by_component_k = X[z == k]
        count_by_component_k = len(X_by_component_k)
        count_by_component.append(count_by_component_k)
        sum_by_component_k = (sum(X_by_component_k)
                              if (0 < len(X_by_component_k))
                              else np.zeros(D))
        denominator = count_by_component_k + sigma*sigma/(zeta*zeta)
        mean_for_mu_k = sum_by_component_k/denominator
        var_for_mu_k = (sigma*sigma/denominator)*np.eye(D)
        mu_k = random_state.multivariate_normal(mean_for_mu_k,
                                                var_for_mu_k, size=1)[0]
        mu.append(mu_k)
    return np.array(mu), count_by_component
```

3.3 component distribution probabilities

For sampling of component assignment, z_n, conditional probabilities $p(z_n = k|\dots)$ need to be calculated according to the following equations:

$$\tilde{p}(z_n = k|X, \{\mu\}, m_k^{-n}; \alpha, \sigma_0^2) = p(x_n|\mu_k; \sigma_0^2) \times (\alpha_k + m_k^{-n}) \quad (1)$$

$$p(z_n = k|\dots) = \frac{\tilde{p}(z_n = k|\dots)}{\sum_l \tilde{p}(z_n = l|\dots)} \quad (2)$$

$$where \quad (3)$$

$$p(x_n|\mu_k; \sigma_0^2) = \frac{1}{(2\pi\sigma_0^2)^{\frac{D}{2}}} \exp\left(-\frac{1}{2\sigma_0^2} \sum_{d=1}^{D} (x_{n,d} - \mu_{k,d})^2 \right) \quad (4)$$

$$and \quad (5)$$

$$m_k^{-n} = |z_{n'} \in z : z_{n'} = k, n' \neq n| \quad (6)$$

i.e. m_k^{-n} is the number of data points assigned to component k excluding data point n.

However, there are situations where $\sum_l \tilde{p}(z_n = l|\dots)$ may become zero (or very close to). It will result in not able to produce $p(z_n = k|\dots)$ required for component assignment sampling.

$\sum_l \tilde{p}(z_n = l|\dots)$ being zero may be due to the following reasons:

- Some of the $(x_{n,d} - \mu_{k,d})^2$ terms are too large, thus too small approaching zero after application of the function $g(y) = \exp(-\frac{1}{2\sigma_0^2} y)$ having $y = \sum_{d=1}^{D} (x_{n,d} - \mu_{k,d})^2$
- With the D increasing, it would compound the above effect of approaching to zero.

Substituting $p(x_n|\mu_k; \sigma_0^2)$ for $\tilde{p}(z_n = k|X, \{\mu\}, m_k^{-n}; \alpha, \sigma_0^2)$ then we can rewrite $p(z_n = k|\dots)$:

$$p(z_n = k|\dots) = \frac{\tilde{p}(z_n = k|\dots)}{\sum_l \tilde{p}(z_n = l|\dots)} \quad (7)$$

$$= \frac{\frac{1}{(2\pi\sigma_0^2)^{\frac{D}{2}}} \exp\left(-\frac{1}{2\sigma_0^2} \sum_{d=1}^{D} (x_{n,d} - \mu_{k,d})^2 \right)}{\sum_l \frac{1}{(2\pi\sigma_0^2)^{\frac{D}{2}}} \exp\left(-\frac{1}{2\sigma_0^2} \sum_{d=1}^{D} (x_{l,d} - \mu_{k,d})^2 \right)} \quad (8)$$

Removing the common factor $\frac{1}{(2\sigma_0^2)^{\frac{D}{2}}}$ in both numerator and denominator:

$$p(z_n = k|\dots) = \frac{\exp\left(-\frac{1}{2\sigma_0^2} \sum_{d=1}^{D} (x_{n,d} - \mu_{k,d})^2 \right)}{\sum_l \exp\left(-\frac{1}{2\sigma_0^2} \sum_{d=1}^{D} (x_{n,d} - \mu_{l,d})^2 \right)} \quad (9)$$

We can perform the following pre-processing to reduce the likelihood of it approaching zero:
Let

- $y_{k,n,d} = (x_{n,d} - \mu_{k,d})^2$
- $y_{max} = max\{y_{k,n,d},\ k \in \{1, \dots, K\}, n \in \{1, \dots, N\}, d \in \{1, \dots, D\}\}$

$$p(z_n = k|\dots) = \frac{\exp\left(-\frac{1}{2\sigma_0^2}\sum_{d=1}^{D} y_{k,n,d}\right)}{\sum_l \exp\left(-\frac{1}{2\sigma_0^2}\sum_{d=1}^{D} y_{l,n,d}\right)} \tag{10}$$

$$= \frac{\exp\left(-\frac{1}{2\sigma_0^2}\left(D \cdot y_{max} + \sum_{d=1}^{D}(y_{k,n,d} - y_{max})\right)\right)}{\sum_l \exp\left(-\frac{1}{2\sigma_0^2}\left(D \cdot y_{max} + \sum_{d=1}^{D}(y_{l,n,d} - y_{max})\right)\right)} \tag{11}$$

$$= \frac{\exp\left(-\frac{1}{2\sigma_0^2} \cdot D \cdot y_{max}\right) \cdot \exp\left(-\frac{1}{2\sigma_0^2}\sum_{d=1}^{D}(y_{k,n,d} - y_{max})\right)}{\sum_l \exp\left(-\frac{1}{2\sigma_0^2} \cdot D \cdot y_{max}\right) \cdot \exp\left(-\frac{1}{2\sigma_0^2}\sum_{d=1}^{D}(y_{l,n,d} - y_{max})\right)} \tag{12}$$

Again, removing the common factor $\exp\left(-\frac{1}{2\sigma_0^2} \cdot D \cdot y_{max}\right)$ on both the numerator and denominator:

$$p(z_n = k|\dots) = \frac{\exp\left(-\frac{1}{2\sigma_0^2}\sum_{d=1}^{D}(y_{k,n,d} - y_{max})\right)}{\sum_l \exp\left(-\frac{1}{2\sigma_0^2}\sum_{d=1}^{D}(y_{l,n,d} - y_{max})\right)} \tag{13}$$

$$= \frac{\exp\left(\frac{1}{2\sigma_0^2}\sum_{d=1}^{D}(y_{max} - y_{k,n,d})\right)}{\sum_l \exp\left(\frac{1}{2\sigma_0^2}\sum_{d=1}^{D}(y_{max} - y_{l,n,d})\right)} \tag{14}$$

Now, we have $0 \leq (y_{max} - y_{l,n,d}),\ l \in \{1,\dots,K\}, n \in \{1,\dots,N\}, d \in \{1,\dots,D\}$ It would not be possible to have the risk of the denominator being zero. It's also less likely to have overflow with terms after applying exponent furtion, as each $(y_{max} - y_{l,n,d})$ is only the delta of the squared distance between the data points to the mean m_l.

But, it still does not guarantee that there will not be overflow with $\sum_l \exp\left(\frac{1}{2\sigma_0^2}\sum_{d=1}^{D}(y_{max} - y_{l,n,d})\right)$. It would be just much less likely so.

It would be another topic to study on how to avoid the overflow problem for the future. Similar approach may be used to reduce the common factor in both the numerator and denominator.

The above pre-processing treatment also simplifies and reduces the float point calculations.

3.4 setup data

This segment of code read in the data points in X to be sampled. It also sets up the parameters, alpha, zeta, and sigma. For the programs to properly sample, the parameters should be consistent to how X was generated.

`setup_data`:

```
# Load sample 'X' and proceed with example parameters.
# (For the purpose of these project, these parameters should be identical
# to those used to generate the sample.)

X = np.loadtxt(sys.argv[1] if len(sys.argv) > 1 else 'X.tsv')

alpha = [1.0, 1.0, 1.0]
zeta = 2.0
sigma = 1.0
print 'alpha:', alpha
print 'zeta:', zeta
print 'sigma:', sigma
```

3.5 sample-and-visualize

This segment of the code gets next samples (mu, z) from the sampler and plots mu, and the data points in X according to the component assignments in z with different colors.

The markers of the means of the components show the labels of the means of the components as polygons. The number of sides of the polygons equals to $k + 3$ where k is the label to a component.

It performs the above procedure forever, until aborted by external signal.

`sample-and-visualize`:

```
<<index_alignment>>

<<errors-f>>

mu_original = np.loadtxt('./mu.tsv')
z_original = np.loadtxt('./z.tsv')
mu_dist_acc = np.empty(shape=[3, 0])
mu_dist_total_acc = []
z_err_acc = np.empty(shape=[3, 0])
z_err_total_acc = []

mu_, z_ = next(sampler)

# Setup visualization ...
plt.ion()

fig, ax = plt.subplots()

ax.grid(True)
ax.set_aspect('equal')

prop_cycle = cycle(plt.rcParams['axes.prop_cycle'])

mu_lines, X_lines = [], []
for k, (x, y) in enumerate(mu_):
    p = next(prop_cycle)
    mu_lines.extend(ax.plot(x, y, 'o', ms=8, mew=2.0, zorder=2.1,
                            color=p['color'], marker=(3+k, 0, 0)))

    x, y = X[z_ == k].T
    X_lines.extend(ax.plot(x, y, '.', color=p['color']))

try:
    # ... and sample forever.
    for n in count(1):
        sys.stdout.write('\rSamples: %d ... ' % n)
        sys.stdout.flush()

        sleep(0.02)

        mu_, z_ = next(sampler)

        <<collect-errors>>

        for k, ((x, y), mu_line, X_line) in enumerate(
                zip(mu_, mu_lines, X_lines)):
            mu_line.set_xdata(x)
            mu_line.set_ydata(y)
```

```
            x, y = X[z_ == k].T
            X_line.set_xdata(x)
            X_line.set_ydata(y)

        fig.canvas.draw()

        # This call helps the plot update continuously on some systems. See:
        # http://stackoverflow.com/a/19119738/3561
        plt.pause(0.1)
except KeyboardInterrupt as ex:
    <<plot-errors>>
    raise(ex)
finally:
    pass
```

3.5.1 collect-errors

Executed in the context of sampling and visualization, to accumulate the error data: `collect-errors`:

```
mu_distances, sum_mu_distances, z_errors, z_error_total \
    = errors_f(mu_, mu_original, z_, z_original)

mu_dist_acc = np.append(mu_dist_acc,
                        np.reshape(mu_distances, [3, 1]), axis=1)
mu_dist_total_acc.append(sum_mu_distances)
z_err_acc = np.append(z_err_acc,
                      np.reshape(z_errors, [3, 1]), axis=1)
z_err_total_acc.append(z_error_total)
```

3.5.2 plot-errors

Executed in the context of exception catch clause of sampling and visualization, to plot the error data into curves: `plot-errors`:

```
plt.subplot(2, 1, 1)
for i in range(mu_dist_acc.shape[0]):
    plt.plot(mu_dist_acc[i], marker=(3+i, 0, 0))
plt.plot(mu_dist_total_acc)
plt.ylabel('Error Distances of Means')

plt.subplot(2, 1, 2)
for i in range(z_err_acc.shape[0]):
    plt.plot(z_err_acc[i], marker=(3+i, 0, 0))
plt.plot(z_err_total_acc)
plt.ylabel('Error Component Assignments')
plt.show()
plt.pause(100)
```

3.6 preamble

The imports.
   `preamble`:

```
import sys
from itertools import count, cycle
from numbers import Integral
from time import sleep
```

```
import numpy as np
from matplotlib import pyplot as plt
```

4 Collapsed Gibbs Sampler

Here are the functions special for collapsed gibbs sampler.

4.1 sample_z_collapsed

sample_z_collapsed:

```
def sample_z_collapsed(z, X, K, D, N, alpha, sigma, zeta, mu, random_state):
    """
    Perform the sampling of z (the component assignments)
    in collapsed gibbs sample procedure.
    z: the component assignments from the previous sampling iteration.
    X: the dataset of the data points to be sampled for
    component means and component assignments.
    K: the assumed number of components in the data to be sampled.
    D: the number of attributes of a data points in X.
    N: the number of data points in X to which
    the sampling of component means and component assignments are performed.
    alpha: the seed vector for Dirchlet sampling of the component proportions.
    sigma: the standard deviation of the noise to the data points
    zeta: the standard deviation for the Gaussian prior
    over the component means.
    mu: the component means.
    random_state: the random state used as consistent context
    for random sampling.

    It returns the updated z as numpy array.
    """

    diff_n_k_squared = np.array([y*y for y in [X-mu[k] for k in range(K)]])
    diff_n_k_sum = np.sum(diff_n_k_squared, axis=2)
    p_xn_k = np.exp(-diff_n_k_sum/(2.0*sigma))/np.power(2.0*np.pi*sigma, D/2.0)
    m_k_exclude_n = np.array([[len(X[[i for i in range(N) if (i != n and i == z[k])]])
                               for n in range(N)]
                              for k in range(K)])
    w = alpha[:, None] + m_k_exclude_n
    p_tild_xn_k = p_xn_k*w
    p_tild_xn_k_sum_over_k = np.sum(p_tild_xn_k, axis=0)
    p_tild_xn_k_sum_over_k[p_tild_xn_k_sum_over_k == 0] = 1.0
    # no normalization when the denominator is 0.0 to avoid div by 0.0 error
    # If the modeling is correct, the case should not happen
    p_z_n_k = p_tild_xn_k/p_tild_xn_k_sum_over_k
    z_next = []
    for n in range(N):
        try:
            z_next.append(random_state.choice(K, p=p_z_n_k[:,n]))
        except ValueError as ex:
            z_next.append(random_state.choice(K, p=None))
            print("Exception: {}; error probabilities: {}; \
            fix by random uniform".format(ex, p_z_n_k[:,n]))
        else:  # successful case
            pass
```

```
        # print("Indeed correct probabilities: {}".format(p_z_n_k[:,n]))

return np.array(z_next)
```

4.2 collapsed_gibbs_sampler

collapsed_gibbs_sampler:

```
def collapsed_gibbs_sampler(X, alpha, zeta, sigma, random_state=None):
    """A collapsed (serial) Gibbs sampler for a Gaussian Mixture Model with known
      ‘alpha‘,
      ‘zeta‘
      and
      ‘sigma‘."""
    if random_state is None:
        random_state = np.random.mtrand._rand
    elif isinstance(random_state, Integral):
        random_state = np.random.RandomState(random_state)

    X = np.atleast_2d(X)
    N, D = X.shape

    alpha = np.atleast_1d(alpha)
    K, = alpha.shape
    assert (alpha > 0.0).all()

    assert zeta > 0.0
    assert sigma > 0.0

    # Initialize ‘z‘.
    z = initial_z(alpha, N, K, random_state)
    # z = [np.random.choice(K) for n in range(N)]

    # Allocate ‘mu‘. (This does *not* initialize it.)
    # mu = np.empty((K, D)) # no longer needed

    while True:
        # Sample each ‘mu‘.
        mu, _ = sample_mu(z, X, K, D, N, sigma, zeta, random_state)

        # Sample each ‘z‘.
        z = sample_z_collapsed(z, X, K, D, N, alpha, sigma, zeta,
                               mu, random_state)

        yield mu, z
```

5 Chromatic Gibbs Sampler

Here are the functions special for chromatic gibbs sampler.

5.1 sample_theta

sample_theta:

```
def sample_theta(count_by_component, alpha, random_state):
    """
    Perform the sampling of theta (the component proportions)
    in chromatic gibbs sample procedure.
    count_by_component: a list of counts of components in data points
    according to z.
    alpha: the seed vector for Dirchlet sampling of the component proportions.
    random_state: the random state used as consistent context
    for random sampling.

    It returns the updated theta.
    """

    gamma = alpha + count_by_component
    theta = random_state.dirichlet(gamma)
    return theta
```

5.2 sample_z_chromatic

sample_z_chromatic:

```
def sample_z_chromatic(z, X, K, D, N, alpha, sigma, zeta, mu, theta,
                       random_state):
    """
    Perform the sampling of z (the component assignments)
    in chromatic gibbs sample procedure.
    z: the component assignments from the previous sampling iteration.
    X: the dataset of the data points to be sampled for
    component means and component assignments.
    K: the assumed number of components in the data to be sampled.
    D: the number of attributes of a data points in X.
    N: the number of data points in X to which
    the sampling of component means and component assignments are performed.
    alpha: the seed vector for Dirchlet sampling of the component proportions.
    sigma: the standard deviation of the noise to the data points
    zeta: the standard deviation for the Gaussian prior
    over the component means.
    mu: the component means.
    theta: the sampled theta (component proportions)
    from the previous iteration.
    random_state: the random state used as consistent context
    for random sampling.

    It returns the updated z as numpy array.
    """

    diff_n_k_squared = np.array([y*y for y in [X-mu[k] for k in range(K)]])
    diff_n_k_sum = np.sum(diff_n_k_squared, axis=2)
    p_xn_k = np.exp(-diff_n_k_sum/(2.0*sigma))/np.power(2.0*np.pi*sigma, D/2.0)

    p_tild_xn_k = p_xn_k*theta[:, None]

    p_tild_xn_k_sum_over_k = np.sum(p_tild_xn_k, axis=0)
    p_tild_xn_k_sum_over_k[p_tild_xn_k_sum_over_k == 0.0] = 1.0
    # no normalization when the denomenator is 0.0
    # to avoid div by 0.0, though it should not happen
    p_z_n_k = p_tild_xn_k/p_tild_xn_k_sum_over_k
    z_next = []
    for n in range(N):
        try:
            z_next.append(random_state.choice(K, p=p_z_n_k[:, n]))
        except ValueError as ex:
            z_next.append(random_state.choice(K, p=None))
            print("Exception: {}; error probabilities: {}; \
                fix by random uniform".format(ex, p_z_n_k[:,n]))
        else:  # successful case
            pass
            # print("Indeed correct probabilities: {}".format(p_z_n_k[:,n]))
    return np.array(z_next)
```

5.3 chromatic_gibbs_sampler

chromatic_gibbs_sampler:

```
def chromatic_gibbs_sampler(X, alpha, zeta, sigma, random_state=None):
    """A chromatic Gibbs sampler for a Gaussian Mixture Model with known
    ‘alpha‘,
    ‘zeta‘
    and
    ‘sigma‘."""
    if random_state is None:
        random_state = np.random.mtrand._rand
    elif isinstance(random_state, Integral):
        random_state = np.random.RandomState(random_state)

    X = np.atleast_2d(X)
    N, D = X.shape

    alpha = np.atleast_1d(alpha)
    K, = alpha.shape
    assert (alpha > 0.0).all()

    assert zeta > 0.0
    assert sigma > 0.0

    # Initialize ‘z‘.
    z = initial_z(alpha, N, K, random_state)
    # z = [np.random.choice(K) for n in range(N)]

    # Allocate ‘mu‘. (This does *not* initialize it.)
    # mu = np.empty((K, D)) # no longer needed

    while True:
        # Sample each ‘mu‘.
        mu, count_by_component = sample_mu(z, X, K, D, N, sigma, zeta,
                                           random_state)
        # Sample theta
        theta = sample_theta(count_by_component, alpha, random_state)

        # Sample each ‘z‘.
        z = sample_z_chromatic(z, X, K, D, N, alpha, sigma, zeta, mu, theta,
                     random_state)
        yield mu, z
```

6 Experiments

Experiments are performed by executing the following commands:
==:

```
python collapsed_gibbs_sampler.py
```

and
==:

```
python chromatic_gibbs_sampler.py
```

The figures are manually saved from those displayed by the above programs.

6.1 Visual Examination

The diagrams below show the sampling of collapsed gibbs sampler and chromatic gibbs sampler. The diagram on the top is the original data points with the means and component partitions.

By examining the updated means of the components, and the component assignments illustrated by the different colors, and comparing that of the original, it seems that both samplers succeeded in sampling the means of the components and their assignments.

One key observation is that the labels of the components may not be the same as those originally generated in the data as the shape of the markers do not align from the original and the sampled.

We also observed that it only takes less than 100 iterations for both samplers to converge on the sampling.

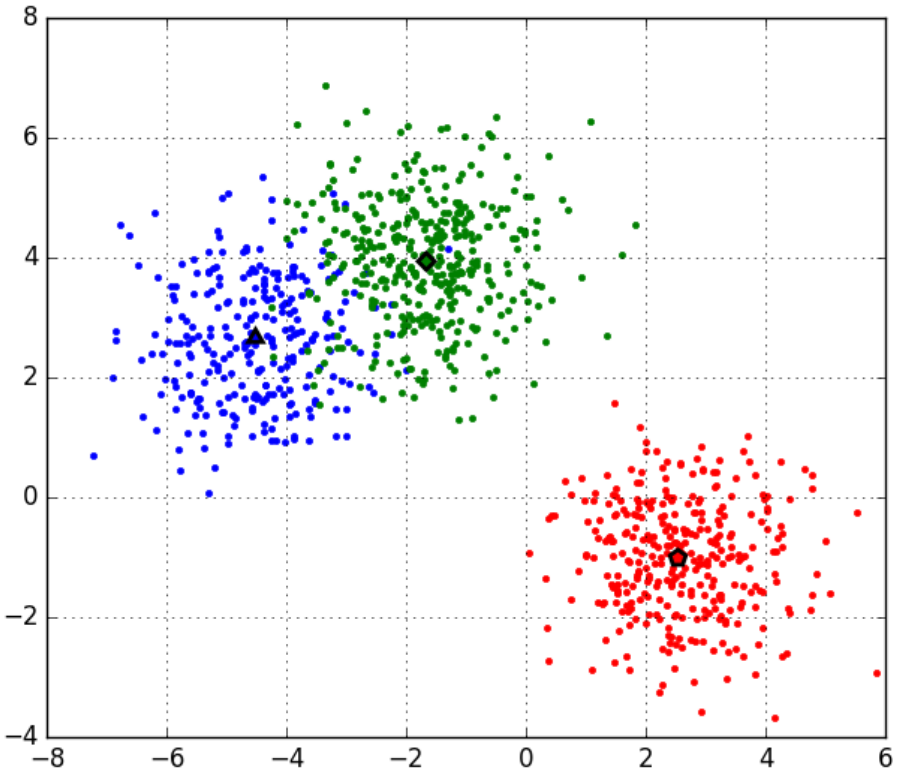

Figure 1: Original Mixture

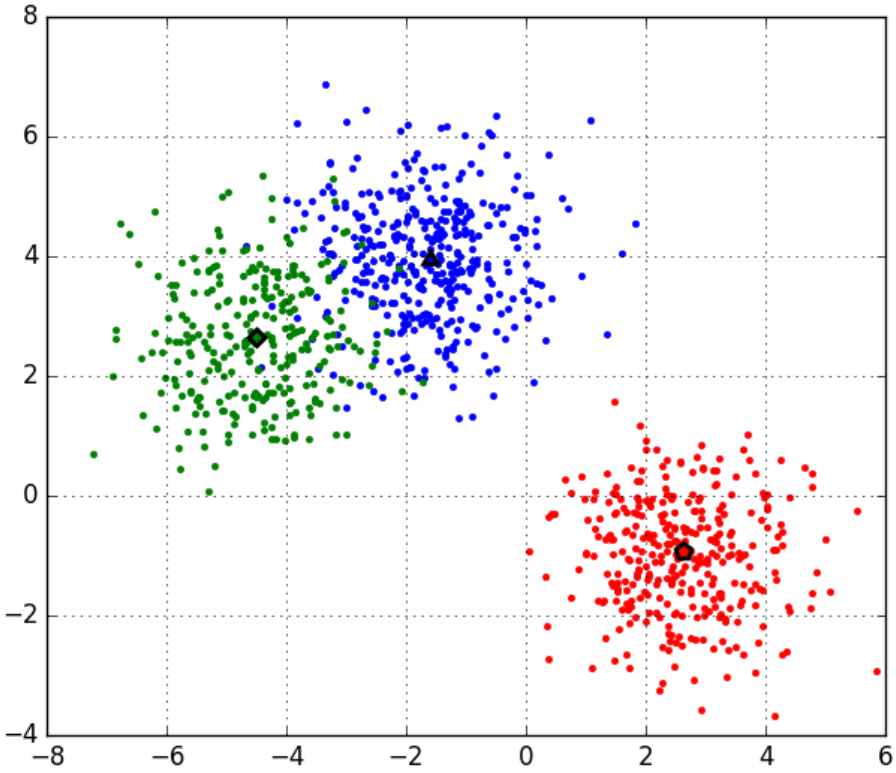

Figure 2: Samples from Collapsed Gibbs Sampler

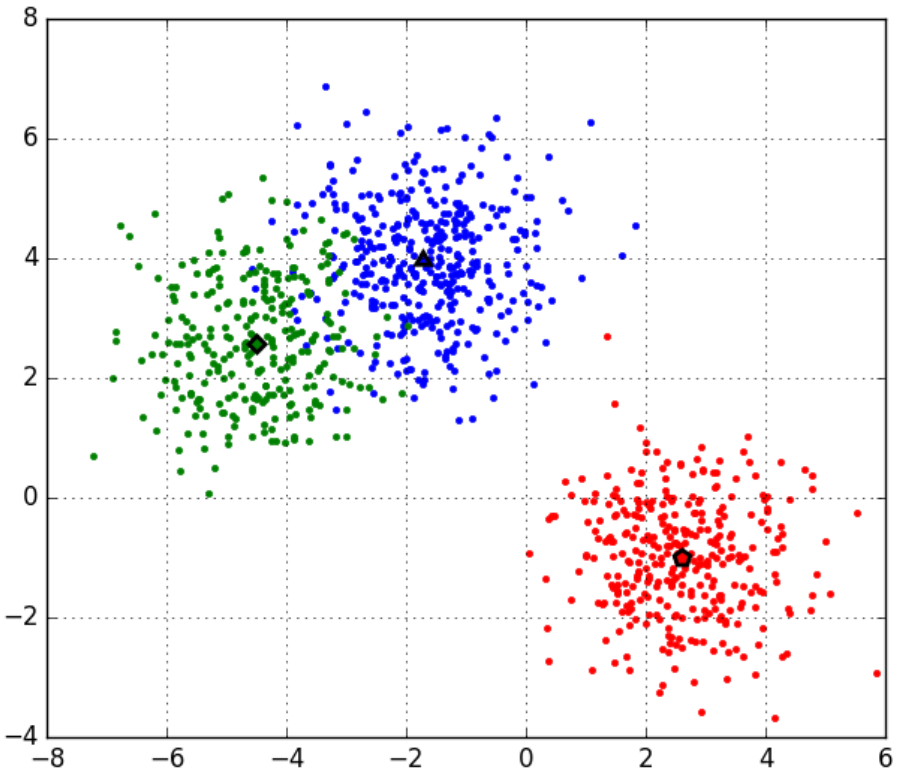

Figure 3: Samples from Chromatic Gibbs Sampler

6.2 Quantified Error Analysis

Here we measure the extent how the samplers correctly sampled the component means and component assignments.

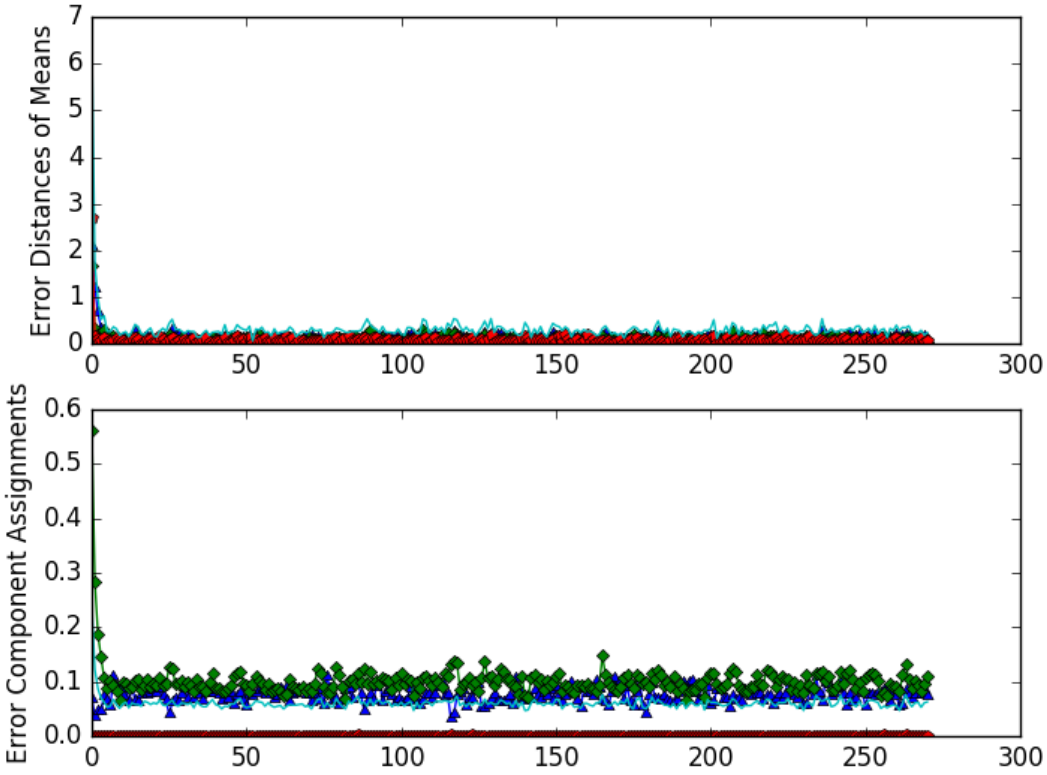

Figure 4: Errors of Collapsed Gibbs Sampler

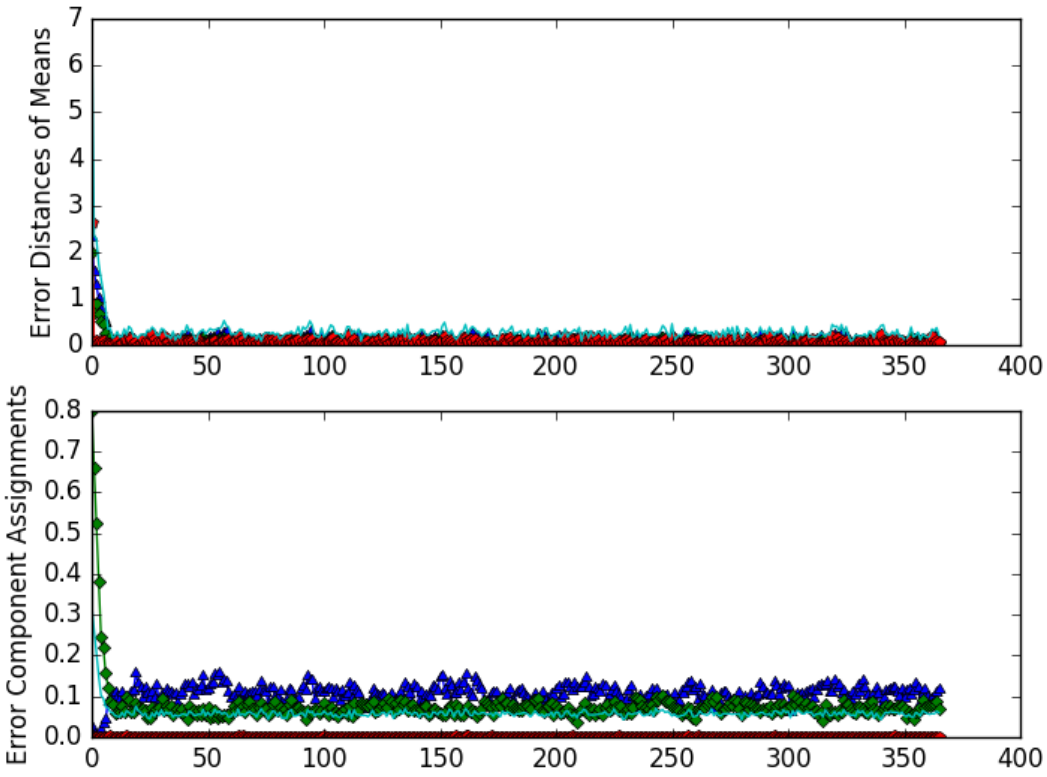

Figure 5: Errors of Chromatic Gibbs Sampler

It seems that the errors are rapidly reduced and remain relatively flat.

The following of the source code for error analysis.

6.2.1 Alignment of Component Indices

First of all, as the component index k has no functional bearing in the sampling outcome, as the prior statistics for the components are all identical to across the components. Therefore, even the samplers can discriminate the components, but the index of the components may not be necessarily the same as those used in the original data. It has been confirmed in the figures above.

Hence, we need to align the index of components between the original and the sampled so that we can perform correctness analysis.

The alignment problem can be characterized as follows:

Given two sets of mean points (the original, and the sampled), pair between the original and the sampled so that the mean points in each pair is the closest.

The following diagram illustrates the idea.

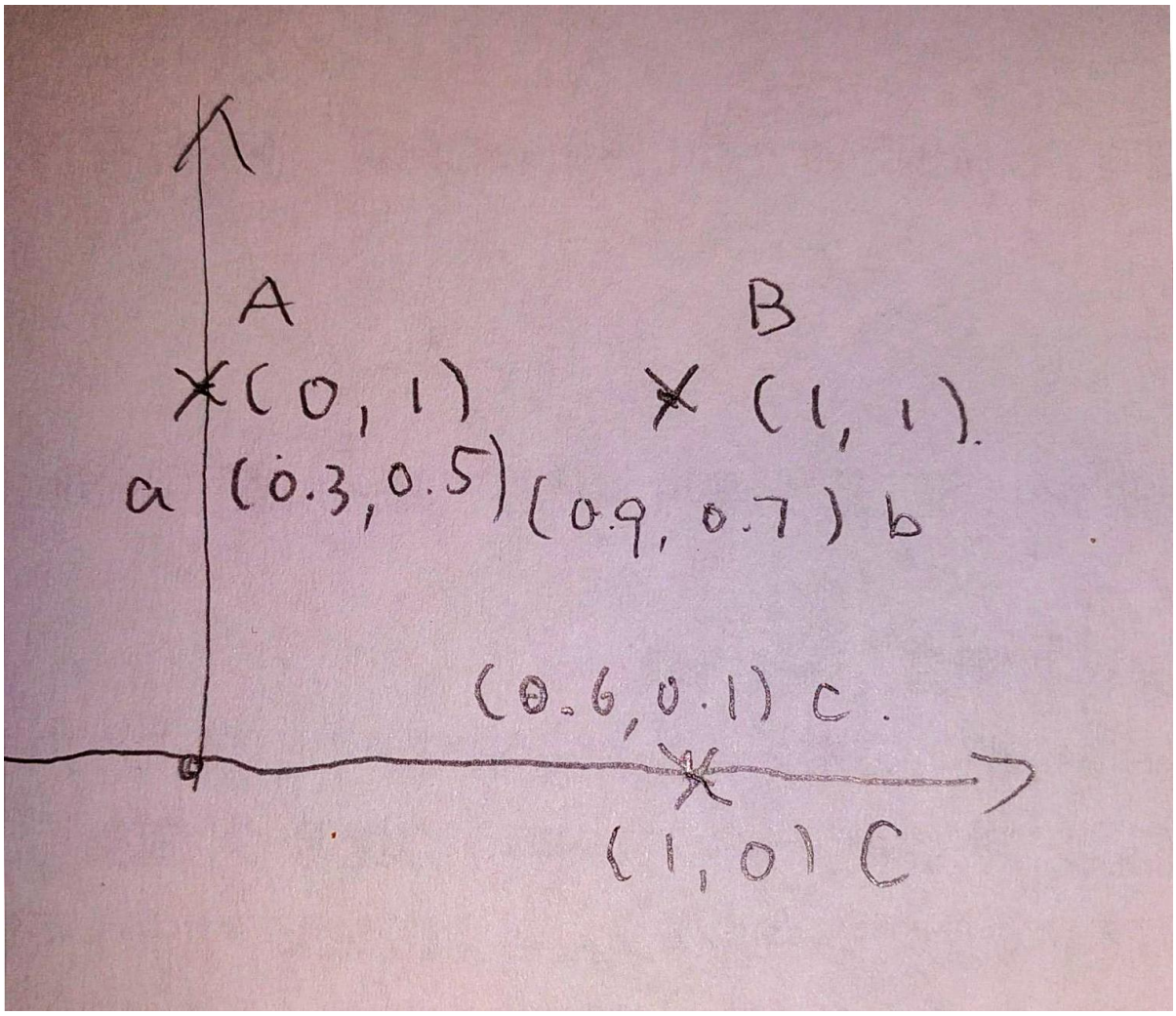

Figure 6: Alignment of means (the original and the sampled)

1. index_alignment

   Here is the code for index_alignment with test driven approach of unit testing.

   `index_alignment`:

```
original = [[0, 1],
            [1, 1],
            [1, 0]]

sampled = [[0.9, 0.7],
           [0.3, 0.5],
           [0.6, 0.1]]

# from the indices of sampled to those of the original
expected_map = {0:1,
                1:0,
                2:2}

def index_alignment(sampled, original):
    """
    map the indices of the sampled to those of the original
    so that the data points of the mapped pair are the closest in distance.

    sampled: a list of data points sampled.
    original: a list of data points original

    The length of sampled and original should be equal.

    return a map of index from that of the sampled to that of the original
    """
    assert(len(sampled) == len(original))
    len_sampled = len(sampled)
    len_original = len(original)
    sampled = np.array(sampled)
    original = np.array(original)
    available = np.full(len_original, True)
    # indices whether the data point has been mathched

    result = {}
    mu_distances = []
    for i in range(0, len_sampled):
        least = -1
        for j in range(0, len_original):
            if available[j]:     # not yet matched
                diff = sampled[i]-original[j]
                inner_diff = np.inner(diff, diff)
                if (least < 0) or (inner_diff < least):
                    least = inner_diff
                    idx_least = j
                # end of if (least < 0) or (inner_diff < least)
            # end of if available[j] == 0
        # end of for j in range(idx_original, len_original)
        result[i] = idx_least
        mu_distances.append(np.sqrt(least))
        available[idx_least] = False
    # end of for i in range(idx_sampled, len_sampled)
    # reorder the distance from the perspective of the original:
    mu_distances_reordered = [None] * len(original)
    for i in range(len(original)):
        mu_distances_reordered[result[i]] = mu_distances[i]
    return result, mu_distances_reordered
```

```
assert(index_alignment(sampled, original)[0] == expected_map)
```

6.2.2 Compute the Errors

For sampling iteration, consider the following error quantification:

- Errors of Means:: For each component, the distance between the original mean and the sampled. Also consider the sum of the error distances across all components.
- Errors of Component Assignments:: For each components, the number of wrong assignments. Also consider the sum of the wrong assignments for all components.

Given the sampled means in mu, the component assignments in z, and mu_original, and z_original for the corresponding original respectively.

1. errors-f

```
errors-f:

<<z_errors_f>>

def errors_f(mu, mu_original, z, z_original):
    index_map, mu_distances = index_alignment(mu, mu_original)

    z_aligned = map(lambda i: index_map[i], z)

    z_errors, z_error_total = z_errors_f(len(original), z_aligned, z_original)
    return mu_distances, sum(mu_distances), z_errors, z_error_total

z_errors_f:

def z_errors_f(K, z_aligned, z_original):
    z_errors = np.zeros(K)
    for i in range(len(z_aligned)):
        if z_original[i] != z_aligned[i]:
            z_errors[int(z_original[i])] += 1
    z_original_component_counts \
        = map(len, [z_original[z_original == k] for k in range(K)])
    return (z_errors/np.array([x if x != 0 else 1
                          for x in z_original_component_counts]),
       sum(z_errors)/len(z_aligned))
```

7 Further Considerations

There may need more unit testing of the implementation of the gibbs samplers. Very often, human errors in programming may cause errors in the computations. It's difficult to validate on the random sampling operations themselves. But it is possible to validate or sanitize on the implementation of the mathematical equations that have deterministic outcome.

This will be left for further improvement, when there is more time.