Nonpar Bayes - HW1

Rongzhao Yan

02/27/2022

1 Overview

In this project, the dirichlet process univariate gaussian mixture model (DPGMM) is implemented, to be specific, the model has the following prior structure:

$$
\begin{aligned}
Y_i|\theta_i &\overset{\text{i.i.d}}{\sim} N(\theta_i, \sigma^2) \\
\theta_i|G &\overset{\text{iid}}{\sim} G \\
G \sim DP(M, G_0), G_0 &= N(m, B) \\
m &\sim N(m_0, D) \\
B &\sim Inv - Gamma(a, b) \\
M &\sim Gamma(c, d).
\end{aligned}
$$

As the prior structure suggested, there are 7 hyper parameters:

$$\{\sigma^2, m_0, D, a, b, c, d\}$$

need to be specified. There are are 7 variables need to be sampled:

$$\{\mathbf{s},\ ^*, m, B, M, \eta\},$$

where $\mathbf{s}$ is the component membership for each datapoint, * is the center for each component, η is an auxiliary variable.

`Python` is used for the algorithm implementation. Only `numpy` is used as the framework of scientific computing.

The implementation is encapsulated into a python class: `DPGMM`. To speed up the performance, `numba.jitclass` is used to compile the class `DPGMM` into machine code. We note that some `numpy` functions are not supported to compile into machine code by `numba`, which are then reimplemented by hand.

The python code will attached in the appendix, and the corresponding source code will also be attached as a seperate file.

2 Algorithm Analysis

2.1 A starting example

We will firstly dive deep into a synthetic data example and see how our model performs.

The datapoints are simulated from three components: $N(5,1), N(-4,1), N(13,1)$, each component has 50 data points. Figure 1 shows the data points with the color their class membership.

The hyperparameter setting in this problem is:

$$\sigma^2 = 1, m_0 = 1, D = 1, a = 1, b = 1, c = 1, d = 1,$$

the MCMC will run 10000 iterations with burn-in period of 2000.

Figure 2 shows the mean of the posterior sample of θ_j for each data point. For the left plot, both x axis and y axis is the observed value, resulting in a straight line. For the right plot, the x axis is the mean of the posterior sample of θ_j for each data point, the y axis is the observer value, the color is the actuall membership for both plot.

We see that the DPGMM sucessfully finds the "center" of each datapoint in the sense that mean of θ_j for datapoints in the same component is close to each other.

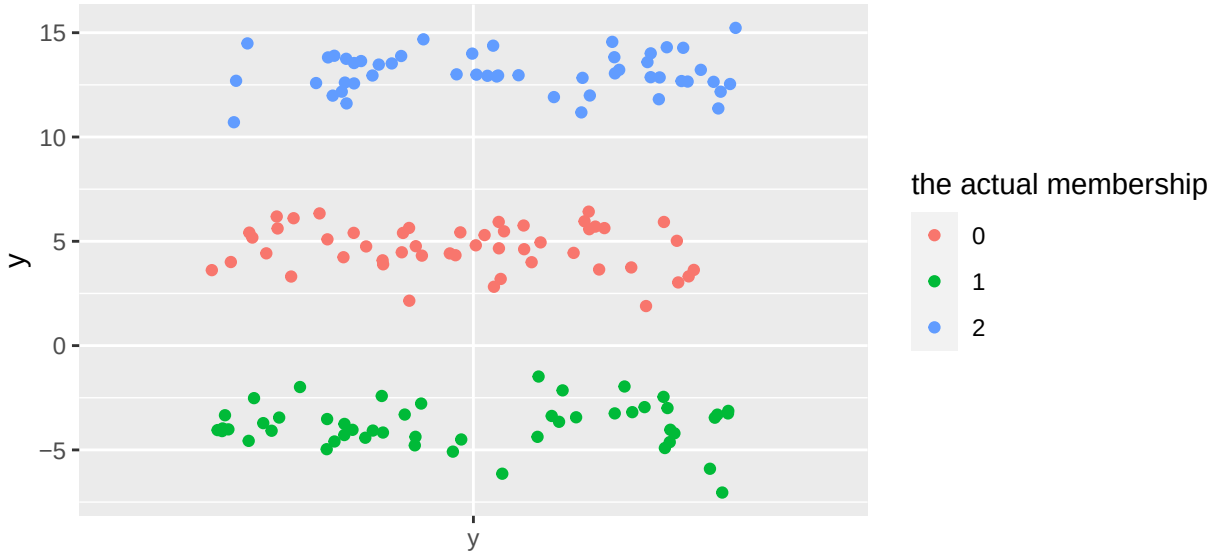

Figure 1: The jittered plot of the synthetic data with color its corresponding membership

Next, we will dive deep into one round of iteration to see how the model cluster the data in a specific round.

Figure 3 shows that in the round 8000, the sampled membership and θ_j of the model. The x axis is the index of each data point, the y axis is the sampled θ_j for each data point, the color of each point corresponding to the **sampled** membership, the shape of each point corresponding to the **actuall** membership.

We can see that the membership of data points that are actually of component 0 and component 2 (corresponding to shape $\circ$ and $\square$) are correctly sampled.

But the membership of data points that are actually of component 1 (corresponding to shape $\triangle$) are not perfectly sampled. We see that there are two very small estimated components consisting of only very few datapoints, whose center (i.e θ_j^*) are really close to the center of (actual) component 1.

This example shows us that the potential of DP to generate some very small components. Figure 4 shows us the barplot of the number of components for each round of the MCMC. We can see that the mode is 4, and most of the time the sampled number of components will be in $[3, 6]$.

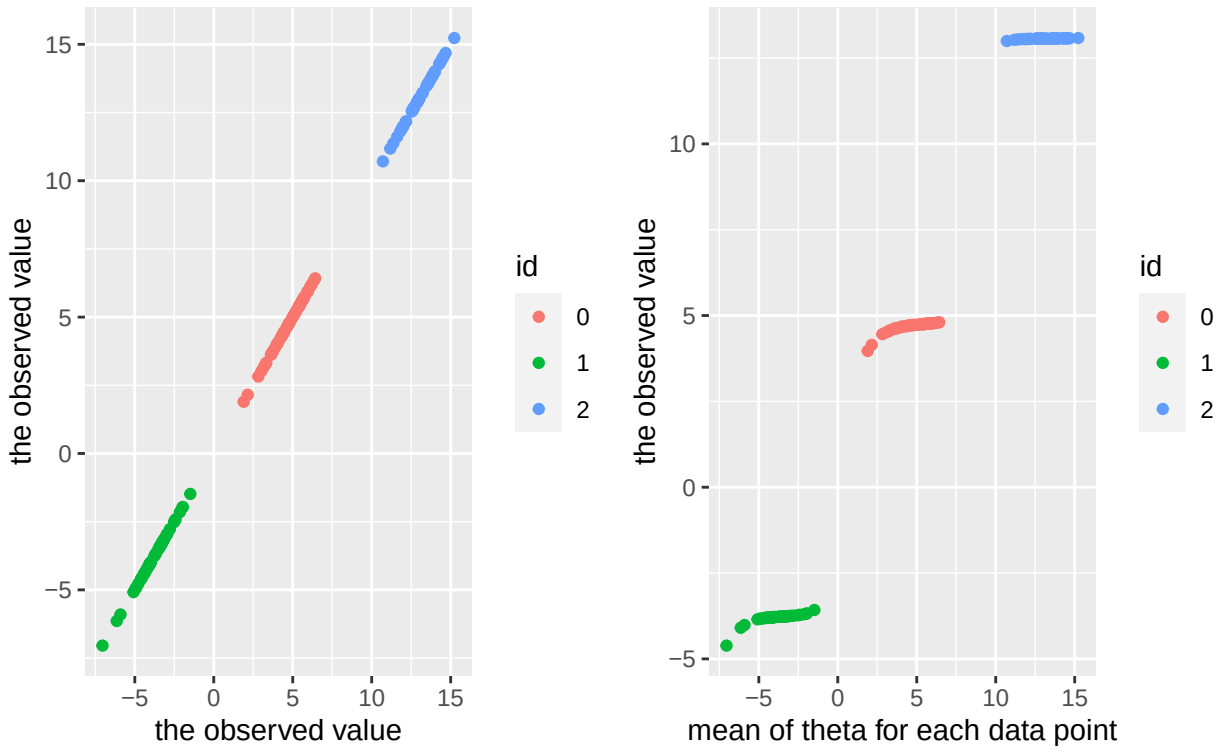

Figure 2: Left: observed value vs observed value, Right: mean of theta vs observed value

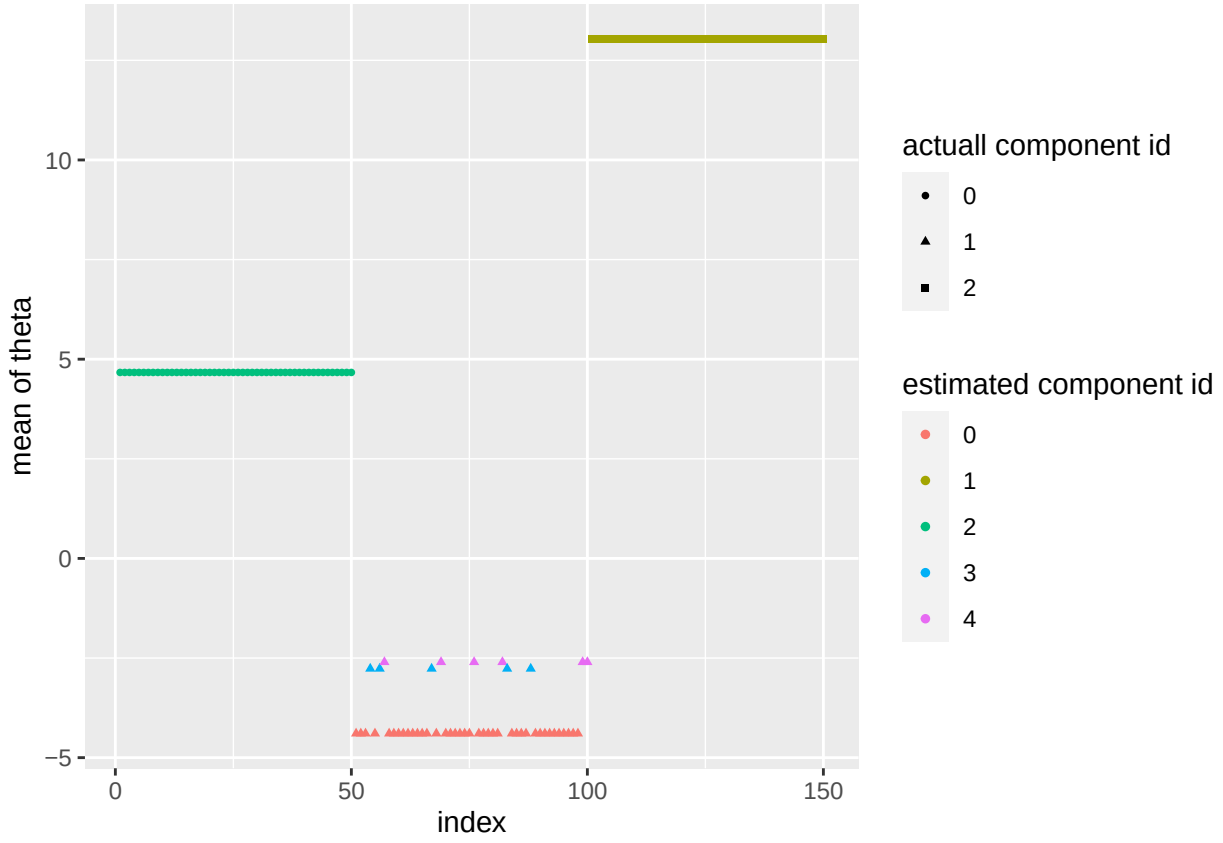

Figure 3: The sampled membership and actuall membership in round 8000

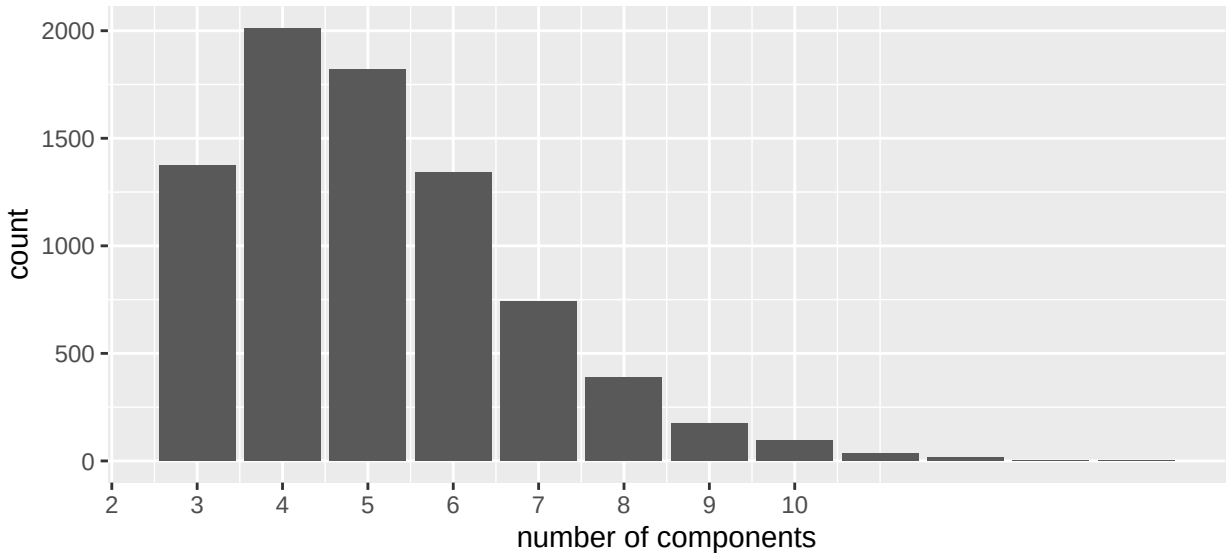

Figure 4: The barplot of the number of sampled membership

3 appendix

```
# %%

import numpy as np  # type: ignore
import pandas as pd  # type: ignore
from typing import Tuple
import numba  # type: ignore
from numba import float64, int64
import matplotlib.pyplot as plt  # type: ignore

# %%
# %%
spec = [("y", numba.typeof(np.array([1.0]))), ("sigma2", float64),
        ("m0", float64), ("D", float64), ("a", float64), ("b", float64),
        ("c", float64), ("d", float64), ("initial_components", int64),
        ("iters", int64)]


@numba.experimental.jitclass(spec)
class DPGMM():
    """
    Dirichlet process gaussian mixture model.
    Currently the 1-dimensional case
    with observations having fixed prior variance is implemented.
    Assume a `G ~ DP(M, G_0), theta_i ~ G` prior, and `G_0 ~ N(m, B)`

    Params:

    `y`: np array of length n.

    `sigma2`: float, the prior of variance of `y_i`,
        assuming `y_i | theta_i ~ N(theta_i, sigma^2)`.
```

```
    `m0`: float, the prior mean of the mean of the base measure `G_0`: m.

    `D`: float, the prior variance of the mean of the base measure `G_0`: m.

    `a`: float, the gamma prior parameter of
        the variance of the base measure `G_0`: B.

    `b`: float, the gamma prior parameter of
        the variance of the base measure `G_0`: B.

    `c`: float, the gamma prior parameter of
        the scale parameter of the DP: M.

    `d`: float, the gamma prior parameter of
        the scale parameter of the DP: M.

    `initial_components`: int, the number of the components at the inital.

    `iters`: int, the size of posterior sample.

    """

    def __init__(self, y: np.ndarray, sigma2: float, m0: float, D: float,
                 a: float, b: float, c: float, d: float,
                 initial_components: int, iters: int) -> None:
        self.y = y
        # self.label = label,
        self.sigma2 = sigma2
        self.m0 = m0
        self.D = D
        self.a = a
        self.b = b
        self.c = c
        self.d = d
        self.initial_components = initial_components
        self.iters = iters

    def initialize(self):
        """
        initialize the value of the parameters to be sampled.

        return:
        `hyperparms_value`: `[m, M, B, eta]`, the initial value of hyperparams
        `s`: np.array of length `n`, the initial value of membership
        `theta`: np.array of length `n`, the initial value of components center
        """
        m = np.random.normal(loc=self.m0, scale=np.sqrt(self.D))
        M = np.random.gamma(shape=self.c, scale=1 / self.d)
        B = 1 / np.random.gamma(shape=self.a, scale=1 / self.b)
        eta = np.random.beta(M + 1, len(self.y))
        hyperparams_value = np.array([m, M, B, eta])

        # initialize the components membership
```

```
        s = np.random.choice(self.initial_components, len(self.y))
        s = self.rearrange_s(s)
        theta_star = np.random.normal(loc=m,
                                      scale=np.sqrt(B),
                                      size=self.initial_components)

        # initialize the corresponding center for each data point
        theta = np.array([theta_star[i] for i in s])

        return hyperparams_value, s, theta

    def rearrange_s(self, s: np.ndarray) -> np.array:
        """
        given a membership array s,
        which may take form like [0, 0, 1, 1, 1, 4, 4, 8]

        rearrange the membership array to be
        [0, 0, 1, 1, 1, 2, 2, 3]

        params:
        `s`: np.array, the membership array

        returns:
        `s`: np.array, the rearranged array.
        """
        s_unique = np.unique(s)
        for i in range(len(s_unique)):
            s[s == s_unique[i]] = i
        return s

    def sample(self) -> Tuple[np.ndarray, np.ndarray, np.ndarray]:
        '''
        Implement the Gibbs sampler to obtain the posterior sample.

        return:
        `hyperparams_sample`: np.array of shape `(iters, 4)` related to
            `[m, M, B, eta]`.
        `s_sample`: np.array of shape `(iters, n)` related to the membership
            for each datapoints.
        `theta_sample`: np.array of shape `(iters, n)` related to components
            center for each datapoints.
        '''
        hyperparams_value, s, theta = self.initialize()
        # hyperparams_value = [12, 0.1, 0.5, 1]
        hyperparams_sample = np.empty((self.iters, 4), dtype=np.float64)
        s_sample = np.empty((self.iters, len(self.y)), dtype=np.int64)
        theta_sample = np.empty((self.iters, len(self.y)), dtype=np.float64)

        for i in range(self.iters):
            s = self.update_s(
                s,
                hyperparams_value[0],
                hyperparams_value[1],
```

```
                hyperparams_value[2])

            theta = self.update_theta(
                theta,
                s,
                hyperparams_value[0],
                hyperparams_value[2])

            hyperparams_value = self.update_hyperparams(
                theta,
                hyperparams_value[0], hyperparams_value[1],
                hyperparams_value[2], hyperparams_value[3]
            )

            s_sample[i, :] = s
            theta_sample[i, :] = theta
            hyperparams_sample[i, :] = hyperparams_value

        return hyperparams_sample, s_sample, theta_sample

    def update_s(self, s: np.ndarray, m: float, M: float,
                 B: float) -> np.ndarray:
        """
        The gibbs sampler for updating s the membership array.
        """

        for i in range(len(s)):
            s_minus = np.delete(s, i)
            y_minus = np.delete(self.y, i)

            # the unique components id excluding y_i
            components_minus = np.unique(s_minus)

            n_j_minus, v_j_minus, m_j_minus = \
                self.calculate_n_v_m_at_j(
                    components_minus, s_minus, m, B, y_minus)

            weights_at_j = (n_j_minus *
                            self.dnorm(self.y[i],
                                       mean=m_j_minus,
                                       std=np.sqrt(v_j_minus + self.sigma2)))

            weights_at_new = M * self.dnorm(
                self.y[i], mean=m, std=np.sqrt(B + self.sigma2))
            weights = (np.append(weights_at_j, weights_at_new) /
                       (weights_at_j.sum() + weights_at_new))

            s_i = self.sample_by_prob(np.append(components_minus,
                                                components_minus.max() + 1),
                                      prob=weights)
            s[i] = s_i

        # after update s, rearrange the id
```

```
        s = self.rearrange_s(s)
        return s

    def update_theta(self, theta: np.ndarray, s: np.ndarray, m: float,
                     B: float) -> np.ndarray:
        """
        The gibbs sampler for updating theta the components center array.
        """
        components = np.unique(s)
        theta_star = np.zeros_like(components, dtype=np.float64)

        for i in range(len(components)):
            sum_y_j = self.y[s == components[i]].sum()
            n_j = (s == components[i]).sum()
            theta_mean = (self.sigma2 * m + B * sum_y_j) / (self.sigma2 +
                                                            B * n_j)

            theta_var = (self.sigma2 * B) / (self.sigma2 + B * n_j)
            theta_star[i] = np.random.normal(loc=theta_mean,
                                             scale=np.sqrt(theta_var))
            theta[s == components[i]] = theta_star[i]

        return theta

    def update_hyperparams(self, theta: np.ndarray, m: float, M: float,
                           B: float, eta: float) -> np.ndarray:
        """
        The gibbs sampler for updating the hyperparams.
        """

        theta_star = np.unique(theta)
        k = len(theta_star)

        # update m
        D1 = 1 / (1 / self.D + k / B)
        m1 = D1 * (self.m0 / self.D + theta_star.sum() / B)
        m = np.random.normal(m1, np.sqrt(D1))

        # update B
        a1 = self.a + k / 2
        b1 = self.b + ((theta_star - m)**2).sum() / 2
        B = 1 / np.random.gamma(shape=a1, scale=1 / b1)

        # update eta
        eta = np.random.beta(M + 1, self.y.shape[0])

        # update M
        weight = ((self.c + k - 1) / (self.c + k - 1 + self.y.shape[0] *
                                      (self.d - np.log(eta))))
        unif = np.random.rand()
        if unif < weight:
            c1 = self.c + k
            d1 = self.d - np.log(eta)
```

```
            M = np.random.gamma(shape=c1, scale=1 / d1)

        else:
            c1 = self.c + k - 1
            d1 = self.d - np.log(eta)
            M = np.random.gamma(shape=c1, scale=1 / d1)

        return np.array([m, M, B, eta])

    def calculate_n_v_m_at_j(self, components: np.ndarray, s: np.ndarray,
                             m: np.ndarray, B: np.ndarray, y: np.ndarray):
        """
        calculate some important statistics for updating s.
        especially the mean(`m_j`) and variance(`v_j`)
            of center for each components
        (excluding the ith data point).
        """

        # the number of observations in each components
        n_j = np.array([(s == i).sum() for i in components])

        sum_of_y_in_group_j = np.array([y[s == i].sum() for i in components])

        v_j = 1 / (1 / B + n_j / self.sigma2)
        m_j = v_j * (m / B + sum_of_y_in_group_j / self.sigma2)

        return n_j, v_j, m_j

    def dnorm(self, x: float, mean: np.ndarray, std: np.ndarray) -> float:
        """
        since numba.jitclass does not support scipy.stat
        a simple function to calculate the normal density
        """
        return np.exp(-((x - mean) / std)**2 / 2) / (std * np.sqrt(2 * np.pi))

    def sample_by_prob(self, x: np.ndarray, prob: np.ndarray) -> int:
        """
        since numba.jitclass does not support np.random.choice
            with argument `p`.
        a simple function to implement sample with probability.
        """
        cumprob = np.cumsum(prob)
        unif = np.random.rand()
        y = x[-1]
        for i in range(len(cumprob)):
            if unif < cumprob[i]:
                y = x[i]
                break
        return y
```