Assignment 1

Name：Shiqu Wu

Student ID：518021910665

1.

(1.1)

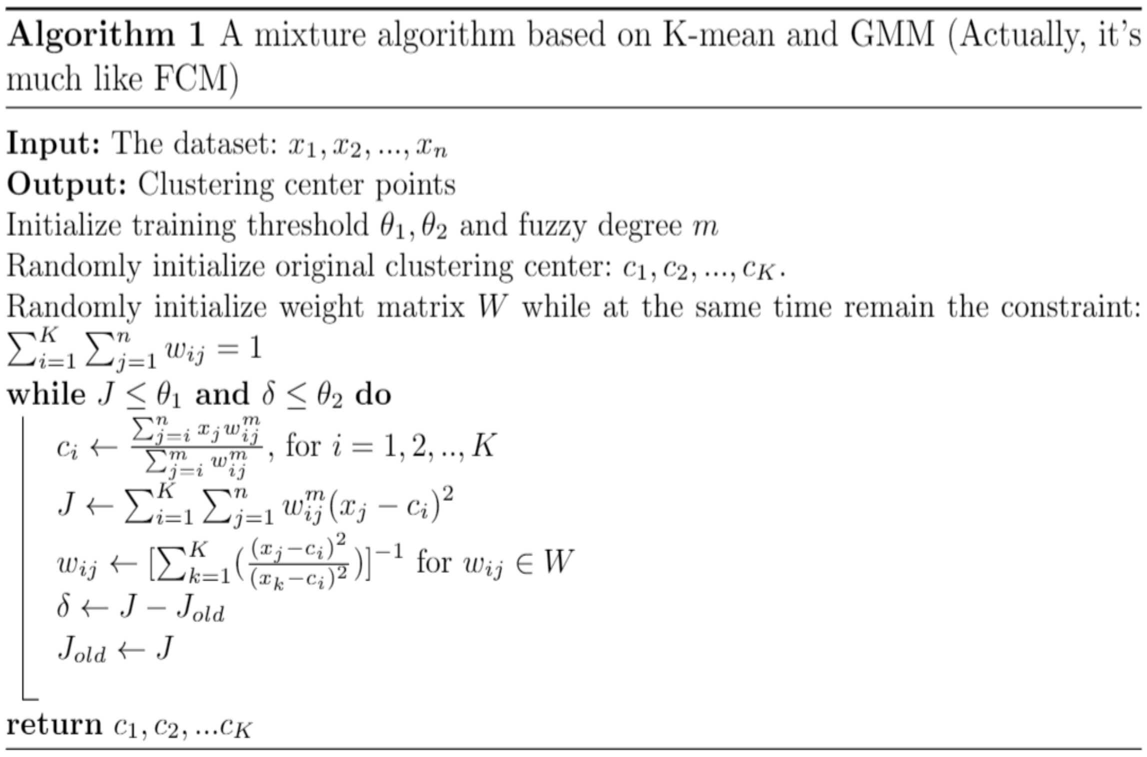
The detailed proof has been raised during the lecture. And here I briefly state the main idea of the proof. We know that the K-mean algorithm consist of the following two steps: 1. Assignment step 2. update step. In order to prove that the above two steps will never increase the k-mean objective function, we separate it to two sub-proof.:

- Firstly, we prove that step (1) satisfies the condition. For the fact that $(x_n - u_i)^2 <= (x_n - u_j)^2, for\ j = 1, 2, \ldots K$ if u_i is the nearest cluster center. It's obvious that step (1) will never increase $J(u_1, \ldots, u_K)$.
- Secondly, we prove that step (2) satisfies the condition. Here is an well known theory for arithmetic sum of squares: $\sum_{x_i \in N_i} ||x_i - u_x||^2 <= \sum_{x_i \in N_i} ||x_i - u_i||^2$, where u_x is the average value of x_i: $u_x = \frac{\sum_{i=1}^{n} x_i}{n}$. Thus, step (2) will increase $J(u_1, \ldots, u_K)$.

In summary, both two steps will never increase the k-mean objective function. Therefore, the proof is done.

2.

(2.1)



Algorithm 1 A mixture algorithm based on K-mean and GMM (Actually, it's much like FCM)

```
Input: The dataset: $x_1, x_2, ..., x_n$
Output: Clustering center points
Initialize training threshold $\theta_1, \theta_2$ and fuzzy degree $m$
Randomly initialize original clustering center: $c_1, c_2, ..., c_K$.
Randomly initialize weight matrix $W$ while at the same time remain the constraint: $\sum_{i=1}^{K} \sum_{j=1}^{n} w_{ij} = 1$
while $J \leq \theta_1$ and $\delta \leq \theta_2$ do
    $c_i \leftarrow \frac{\sum_{j=i}^{n} x_j w_{ij}^m}{\sum_{j=i}^{m} w_{ij}^m}$, for $i = 1, 2, .., K$
    $J \leftarrow \sum_{i=1}^{K} \sum_{j=1}^{n} w_{ij}^m (x_j - c_i)^2$
    $w_{ij} \leftarrow [\sum_{k=1}^{K} (\frac{(x_j - c_i)^2}{(x_k - c_i)^2})]^{-1}$ for $w_{ij} \in W$
    $\delta \leftarrow J - J_{old}$
    $J_{old} \leftarrow J$
return $c_1, c_2, ...c_K$
```

(2.2)

Advantages:

1. Better performance.
2. No need to make an assumption (Gaussian Distribution) for the input dataset. Theoretically, it can fit any kind of distribution.
3. More robust. Due to the use of soft assignment instead of hard assignment.

Disadvantages:

1. User need to determine the number of cluster centers K.
2. High computational cost compared to K-mean.
3. Based on EM algorithm, so it's likely to stuck in local optimal. And the performance highly depends on the initialization.

3.

(3.1)

Comparison between K-mean and Competitive learning:

Main components of CL and K-mean:

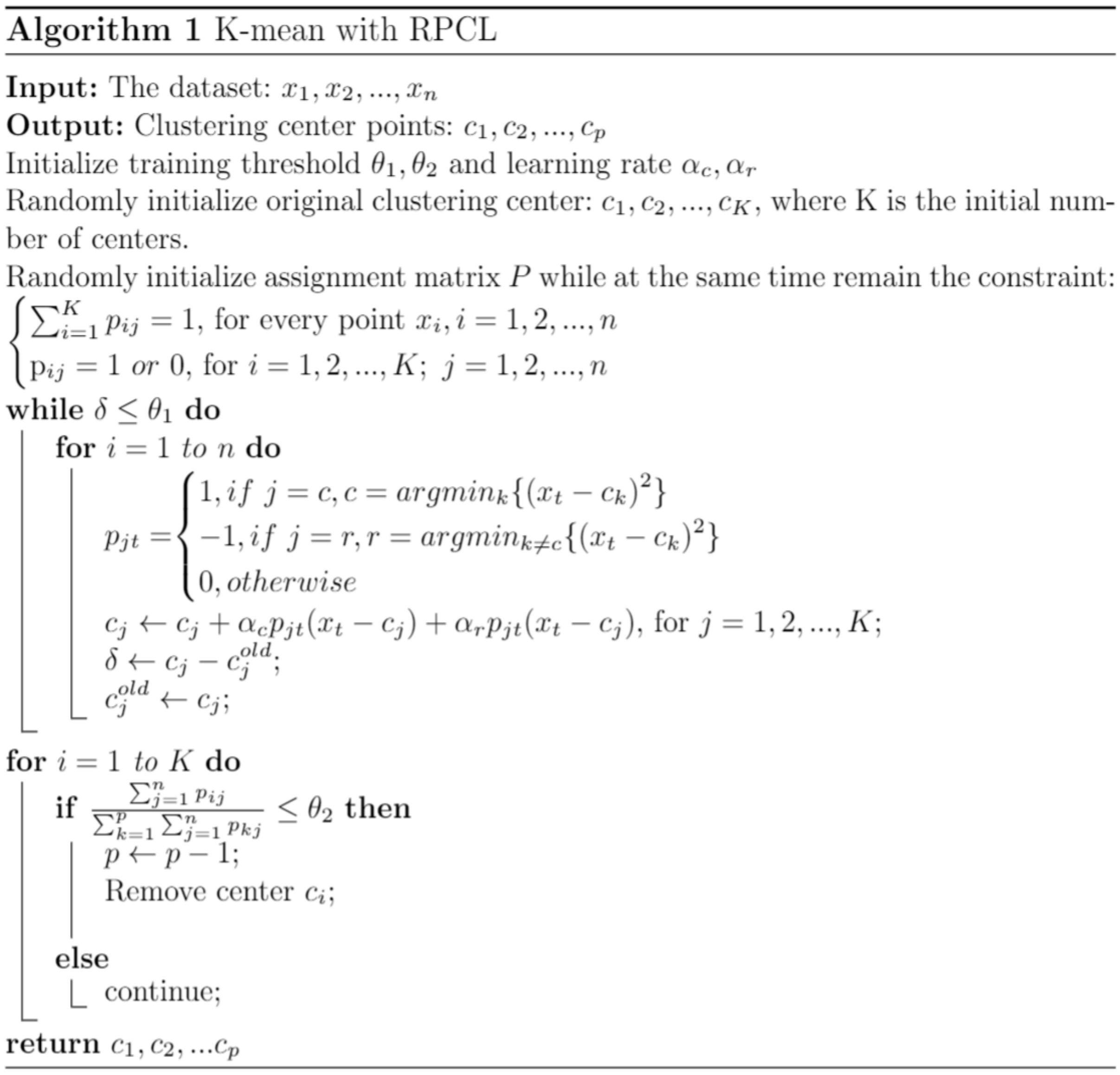
1. **Competitors** --> K-mean cluster centers
2. **Competing rules** --> E step in K-mean.
    a. winner takes all/most
    b. punish frequent winner
    c. punish second winner

1. **Update rules** --> M step in K-mean
    - gradient update

What is CL:

Competitive learning is a learning theory that **Competitors** use **Competing rules** to scramble for resources, and use the resources to update themselves according to **Update rules**, which makes them more competitive in the next turn. It's intuition comes from rule of "Winner takes all/most" and law of expanded reproduction in real world.

(3.2)



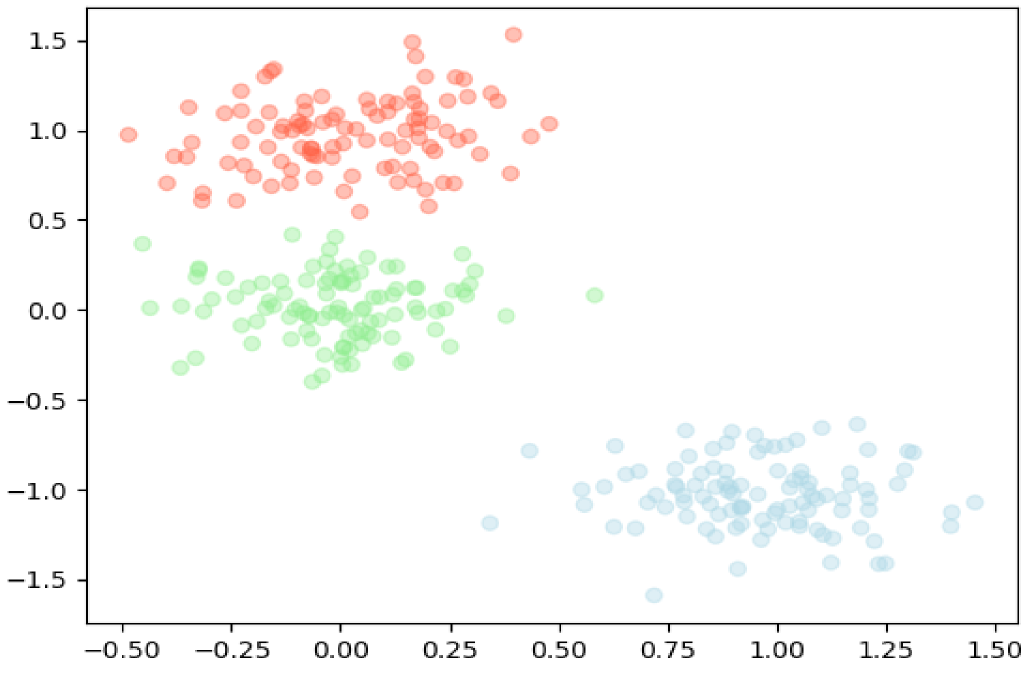
```
Algorithm 1 K-mean with RPCL

Input: The dataset: $x_1, x_2, ..., x_n$
Output: Clustering center points: $c_1, c_2, ..., c_p$
Initialize training threshold $\theta_1, \theta_2$ and learning rate $\alpha_c, \alpha_r$
Randomly initialize original clustering center: $c_1, c_2, ..., c_K$, where K is the initial number of centers.
Randomly initialize assignment matrix $P$ while at the same time remain the constraint:
$\begin{cases} \sum_{i=1}^{K} p_{ij} = 1, \text{ for every point } x_i, i = 1, 2, ..., n \\ \text{p}_{ij} = 1 \ or \ 0, \text{ for } i = 1, 2, ..., K; \ j = 1, 2, ..., n \end{cases}$
while $\delta \le \theta_1$ do
    for $i = 1$ to $n$ do
        $p_{jt} = \begin{cases} 1, if \ j = c, c = argmin_k\{(x_t - c_k)^2\} \\ -1, if \ j = r, r = argmin_{k \neq c}\{(x_t - c_k)^2\} \\ 0, otherwise \end{cases}$
        $c_j \leftarrow c_j + \alpha_c p_{jt}(x_t - c_j) + \alpha_r p_{jt}(x_t - c_j)$, for $j = 1, 2, ..., K$;
        $\delta \leftarrow c_j - c_j^{old}$;
        $c_j^{old} \leftarrow c_j$;
for $i = 1$ to $K$ do
    if $\frac{\sum_{j=1}^{n} p_{ij}}{\sum_{k=1}^{p} \sum_{j=1}^{n} p_{kj}} \le \theta_2$ then
        $p \leftarrow p - 1$;
        Remove center $c_i$;
    else
        continue;
return $c_1, c_2, ...c_p$
```

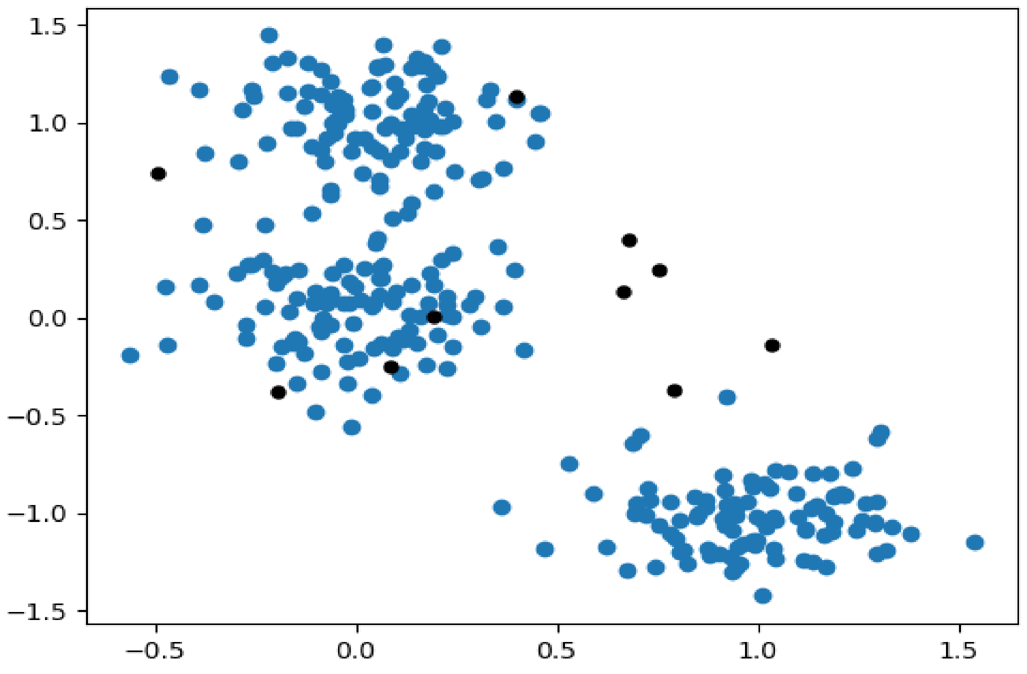
(3.3)

Experiment Results:

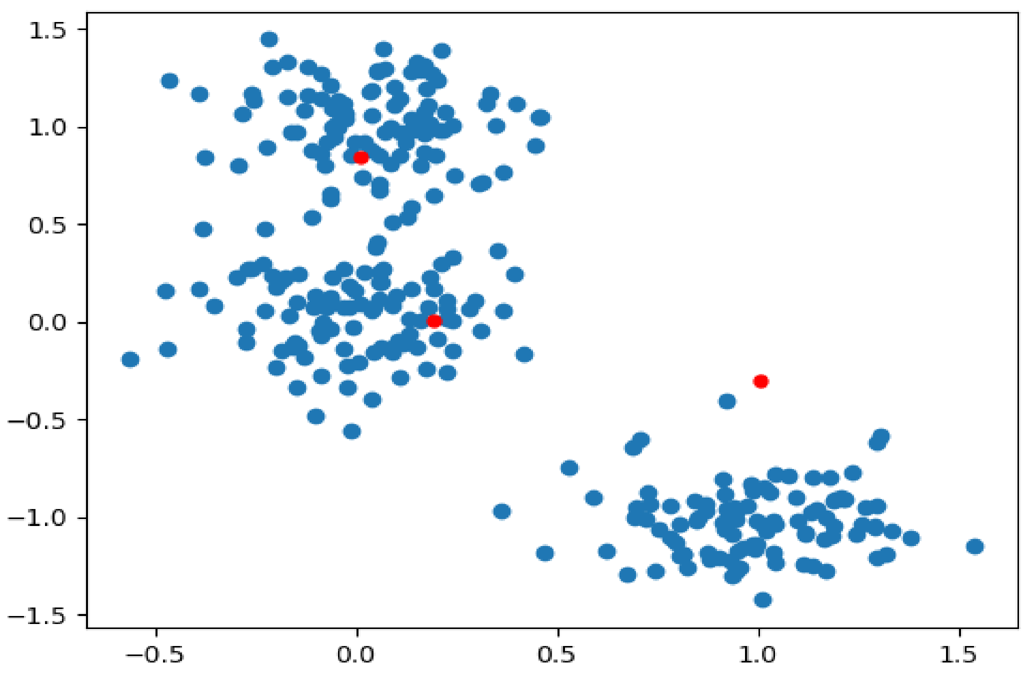
Dataset:



Initial state: (black points represents randomly initialized center)



State after running RPCL: (red points represents cluster center)



4.

(4.1)

AIC,BIC:

AIC (Akaike Information Criterion) and BIC (Bayesian Information Criterion) are metrics for choosing the best model. In order to solve model selection problem, scientists introduce these metrics to quantify the effectiveness and efficiency of models. They reflect the trade-off between the **complexity** of the model (measured by total number of parameters) and the **ability** of the model to describe the data set (measured by maximum likelihood function value). Here are the definition:

$$AIC = 2k - 2ln(L)$$
$$BIC = k * ln(n) - 2ln(L)$$

where k is total number of parameters, L is the maximum likelihood function and n is the number of samples. From the definition we can see that BIC has a larger penalty item $k * ln(n)$ than AIC , which is $2k$. Thus, BIC is more effective in the case that small dataset has high feature dimension (avoid the curse of high dimensionality).

(4.2)

Code Implementation:

AIC:

```
covariance_type_list = ['spherical', 'tied', 'diag', 'full']
for covariance_type in covariance_type_list:
    for n_components in n_components_range:
        gmm = mixture.GaussianMixture(n_components=n_components,
covariance_type=covariance_type)
        gmm.fit(X)
        AIC_list.append(gmm.aic(X))
        if AIC_list[-1] < lowest_AIC:
            lowest_AIC = AIC_list[-1]
            best_model = gmm
AIC_list = np.array(AIC_list)
```

BIC:

```
covariance_type_list = ['spherical', 'tied', 'diag', 'full']
for covariance_type in covariance_type_list:
    for n_components in n_components_range:
        gmm = mixture.GaussianMixture(n_components=n_components,
covariance_type=covariance_type)
        gmm.fit(X)
        BIC_list.append(gmm.bic(X))
        if BIC_list[-1] < lowest_BIC:
            lowest_BIC = BIC_list[-1]
            best_model = gmm
BIC_list = np.array(BIC_list)
```

VBEM:

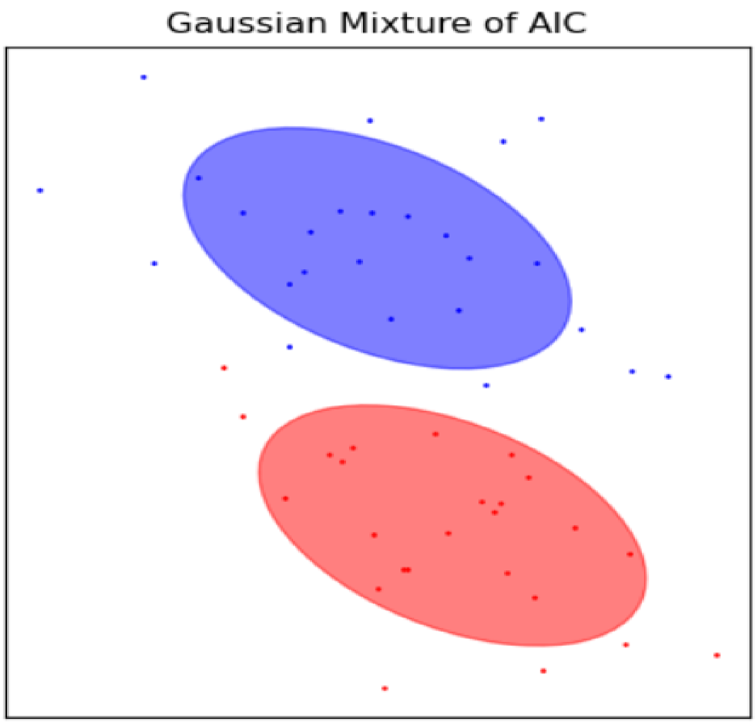
```
model =
mixture.BayesianGaussianMixture(n_components=8,covariance_type='ful
l').fit(X)
```

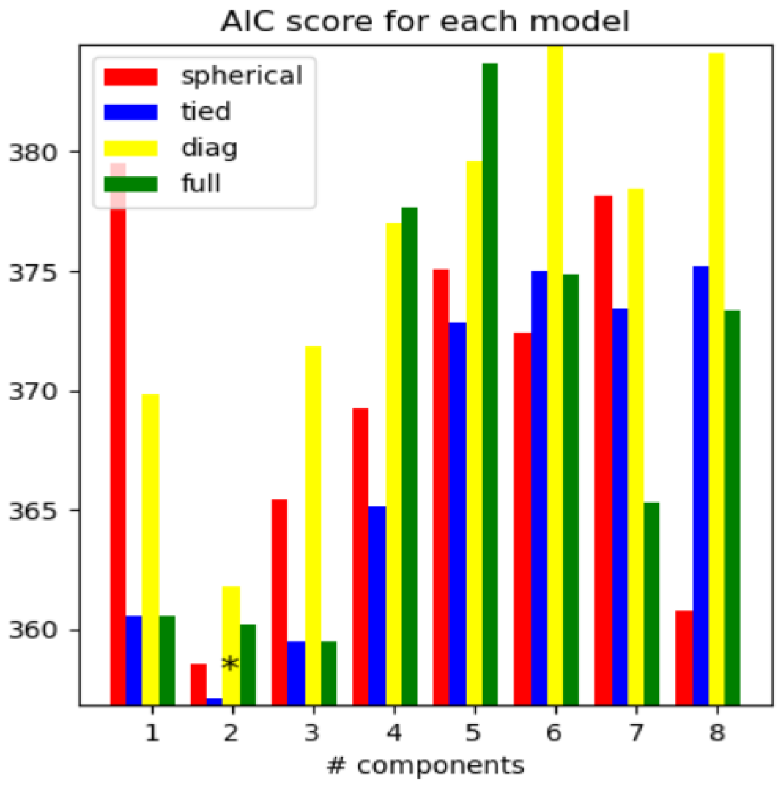
Experiments:

Note: the visualization code following the project in Github: https://gist.github.com/CarstenSchelp

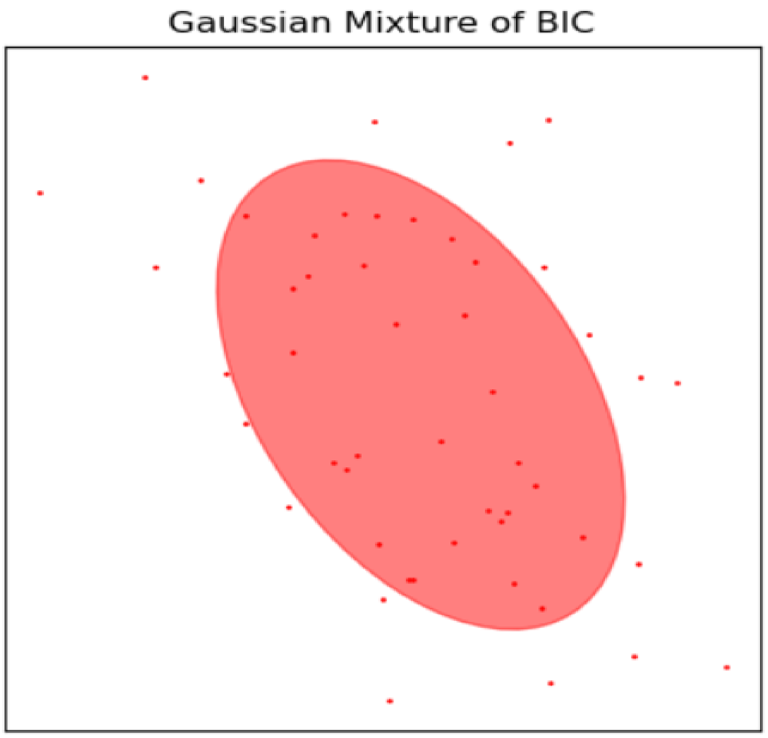
Setting: # cluster =2, # samples =100

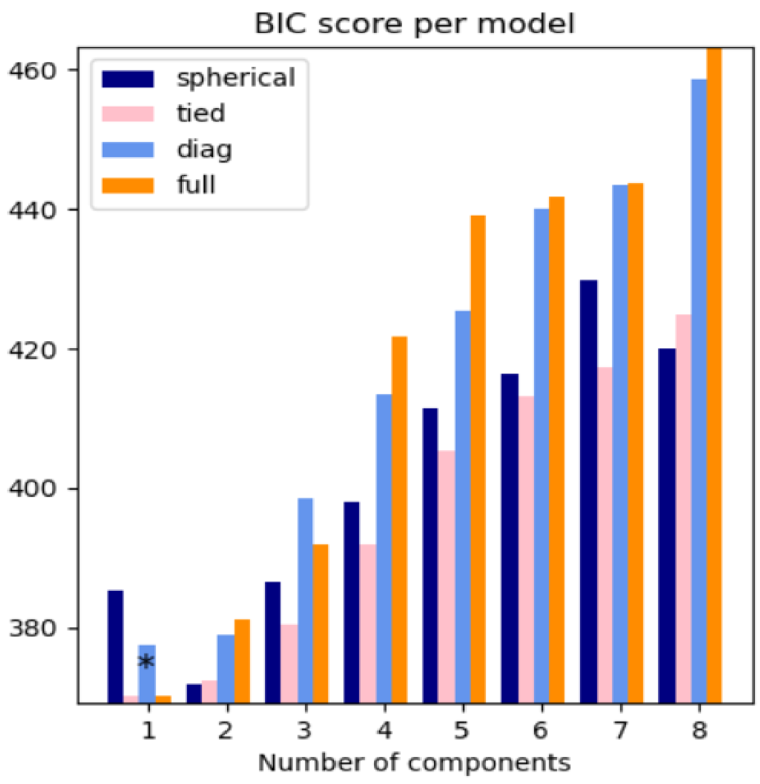
AIC:



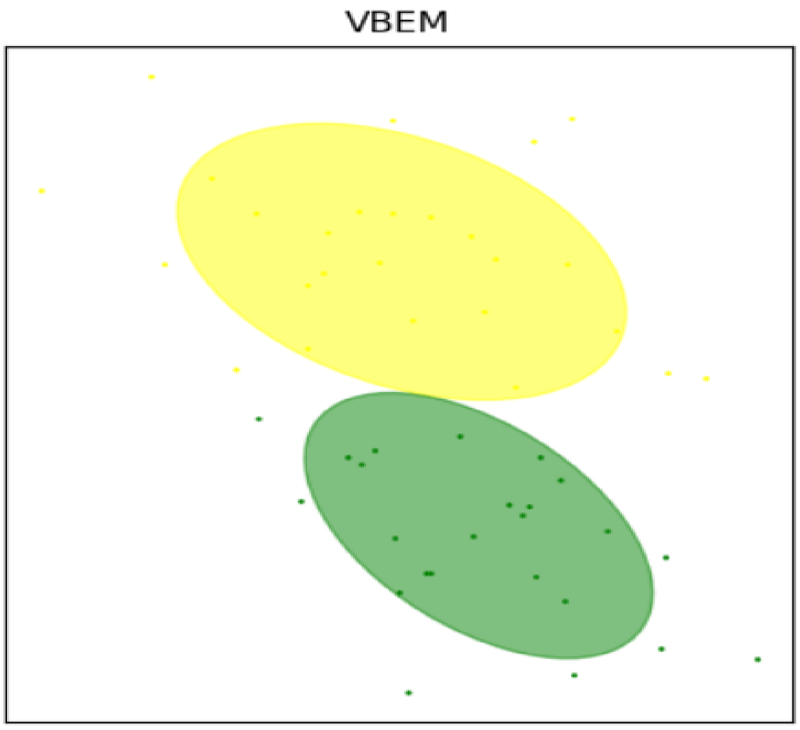

AIC score for each model
spherical
tied
diag
full
380
375
370
365
360
1
2
3
4
5
6
7
8
# components
*

BIC:


Gaussian Mixture of BIC

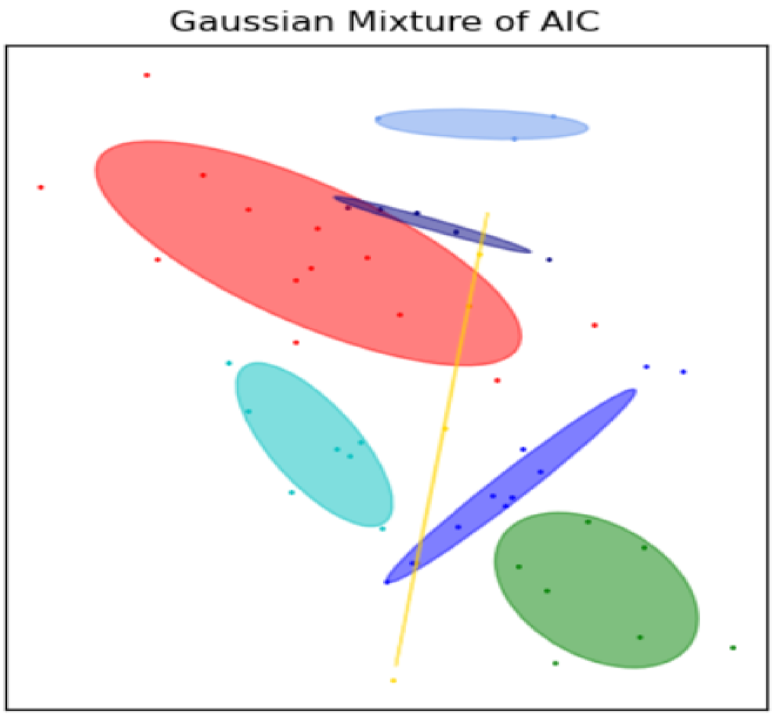

BIC score per model
spherical
tied
diag
full
460
440
420
400
380
1
2
3
4
5
6
7
8
Number of components

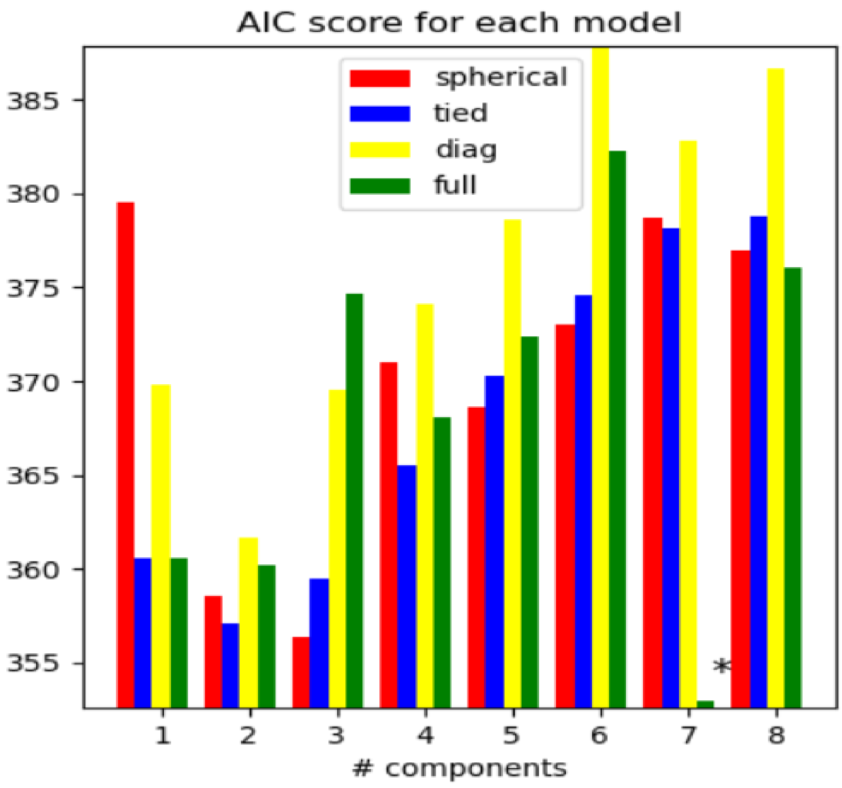
VBEM:


VBEM

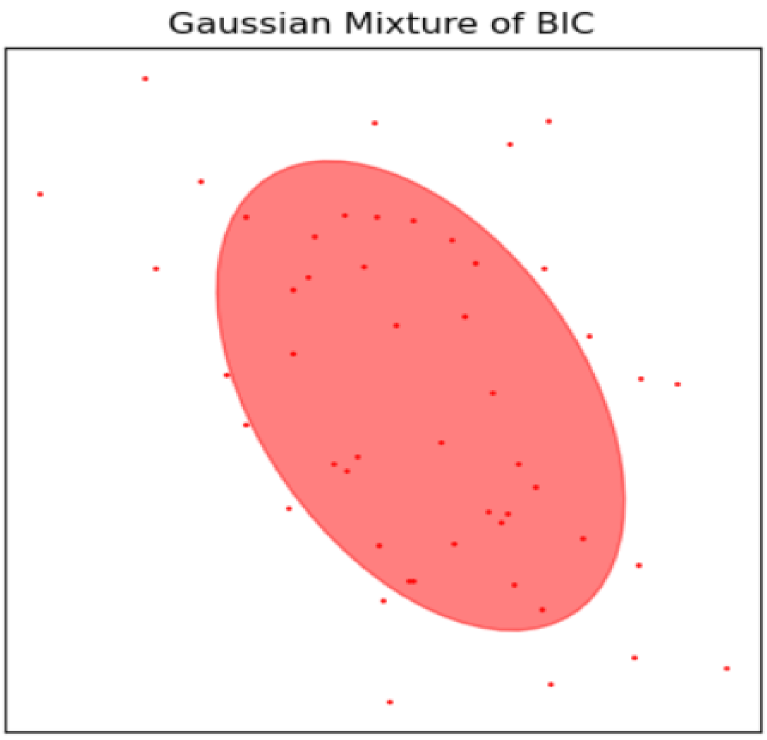
Setting: # cluster =2, # samples =1000

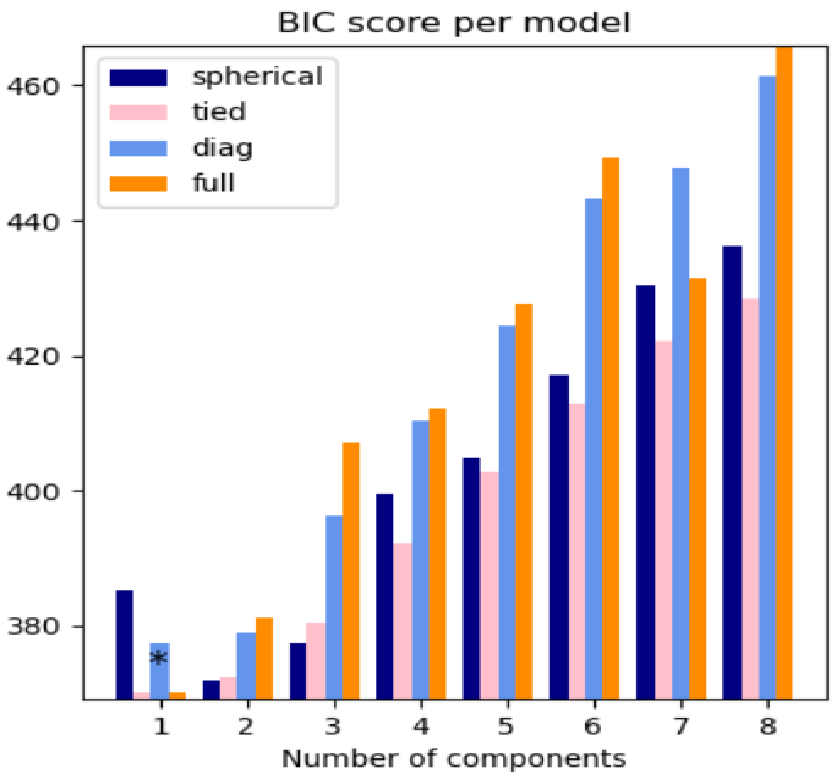
AIC:



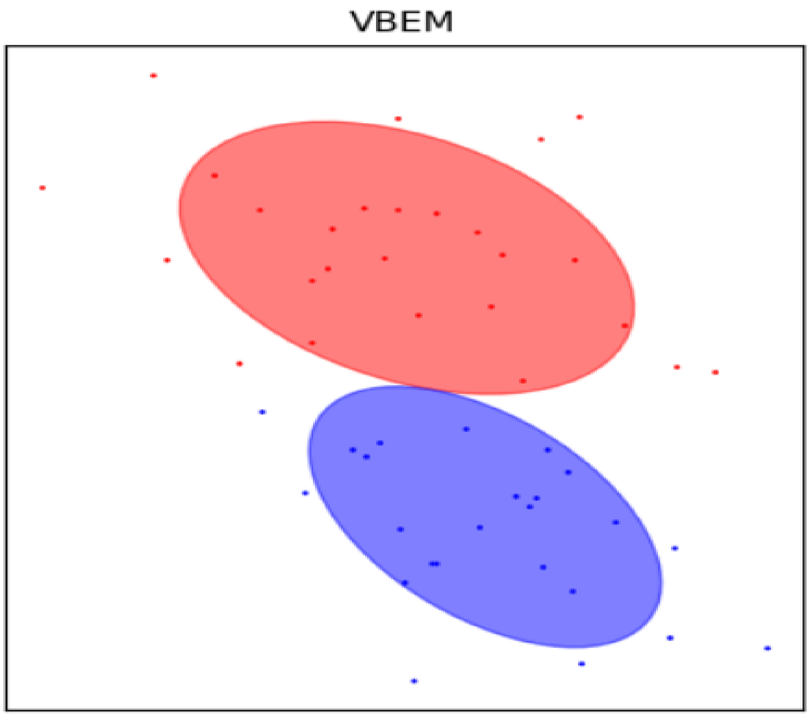


BIC:


Gaussian Mixture of BIC

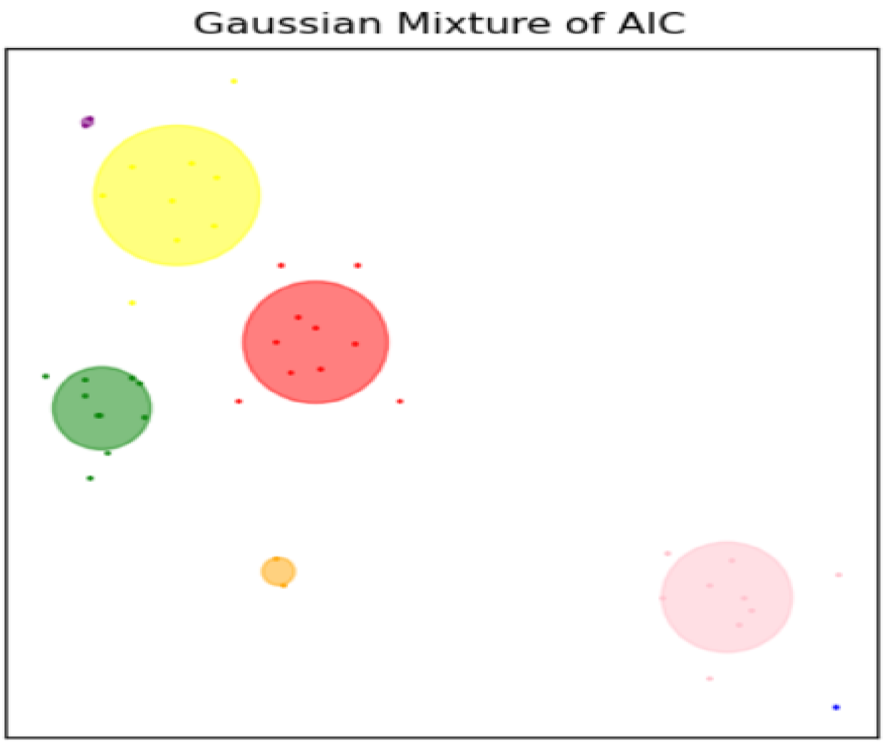

BIC score per model
spherical
tied
diag
full
460
440
420
400
380
*
1
2
3
4
5
6
7
8
Number of components

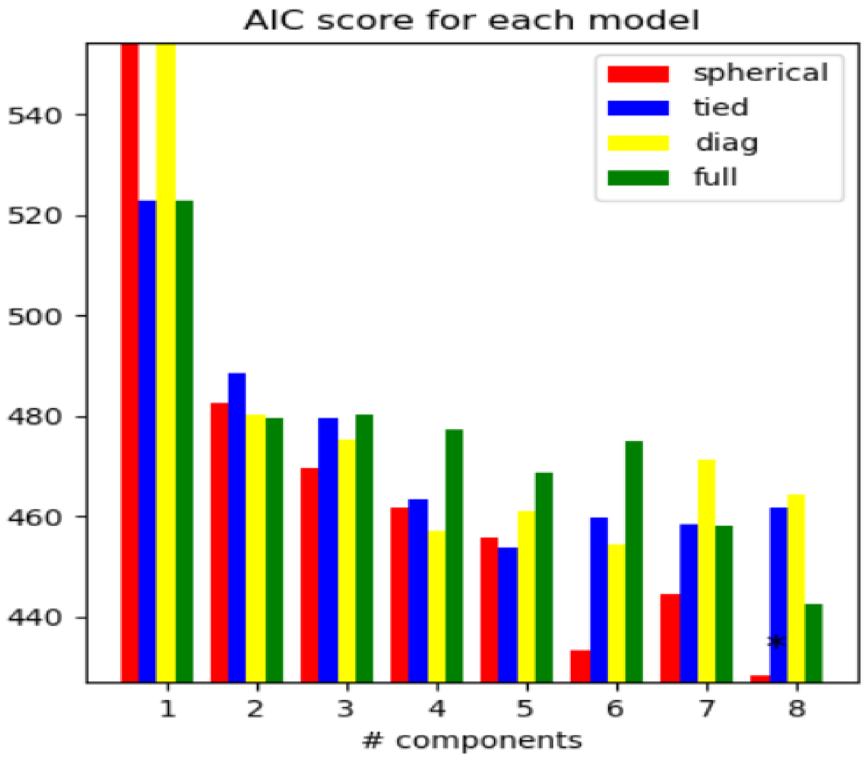
VBEM:



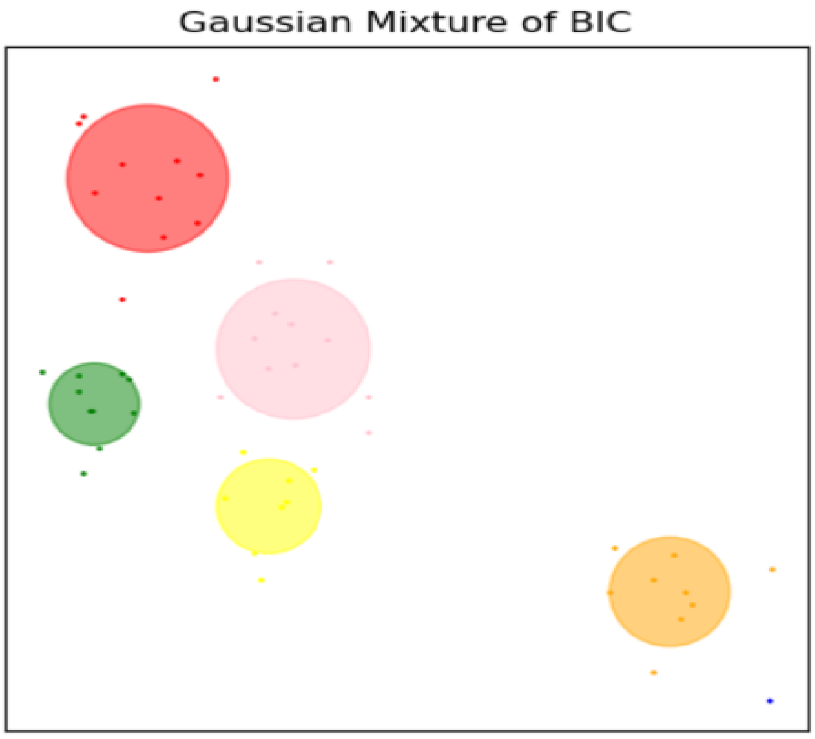
Setting: # cluster =5, # samples =100

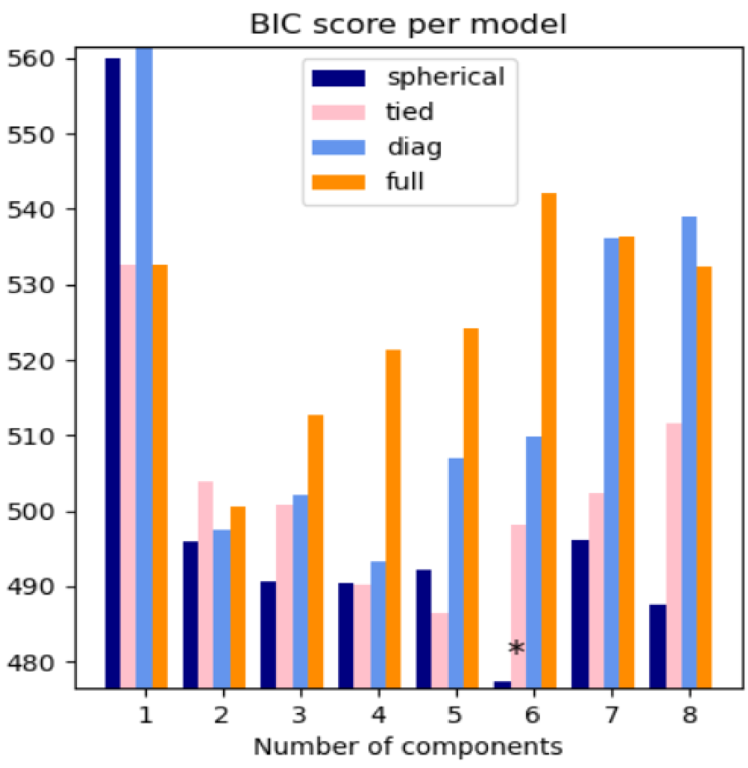
AIC:



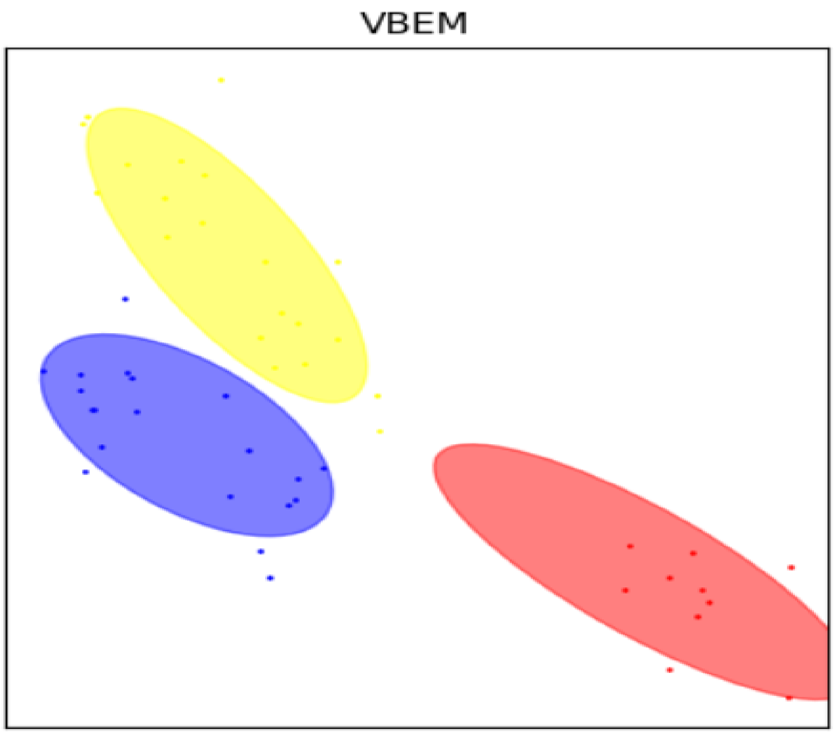

AIC score for each model
spherical
tied
diag
full
540
520
500
480
460
440
1
2
3
4
5
6
7
8
# components

BIC:


Gaussian Mixture of BIC


BIC score per model
560
550
540
530
520
510
500
490
480
spherical
tied
diag
full
*
1
2
3
4
5
6
7
8
Number of components

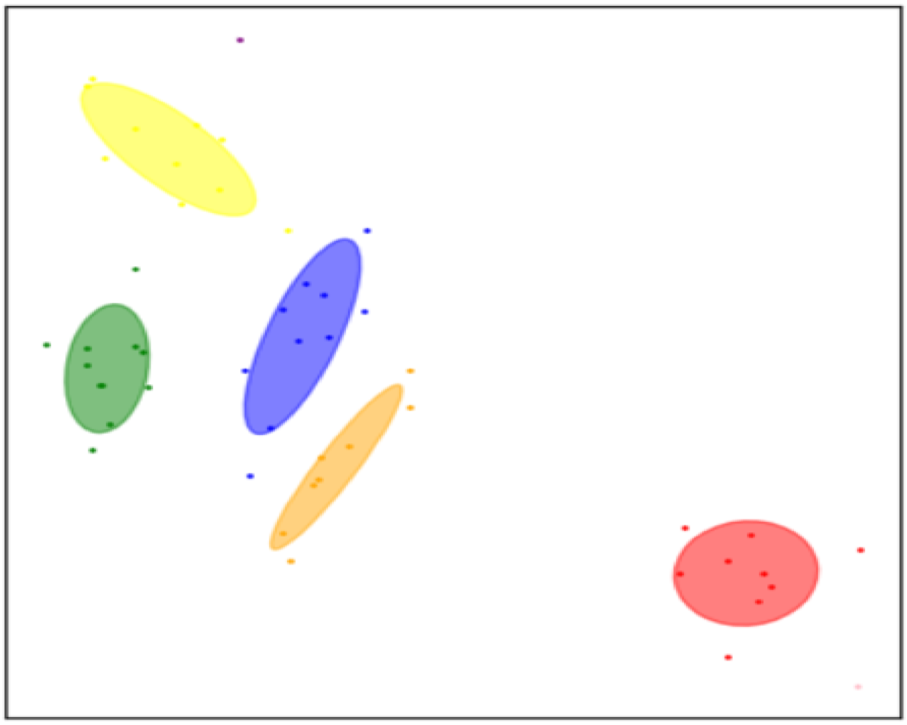
VBEM:

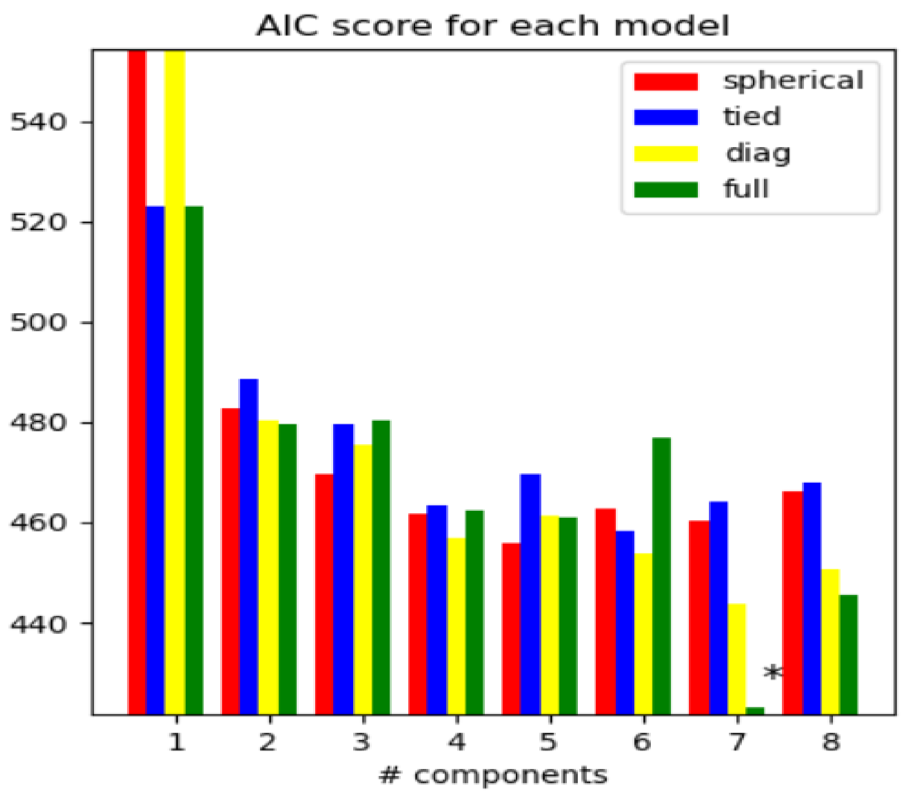

VBEM

Setting: # cluster =5, # samples =1000

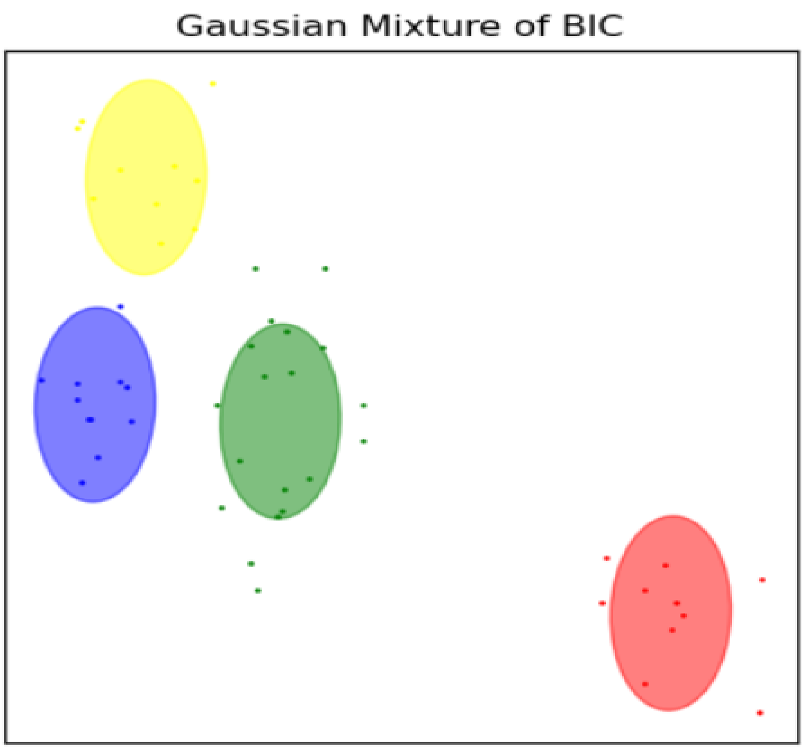
AIC:

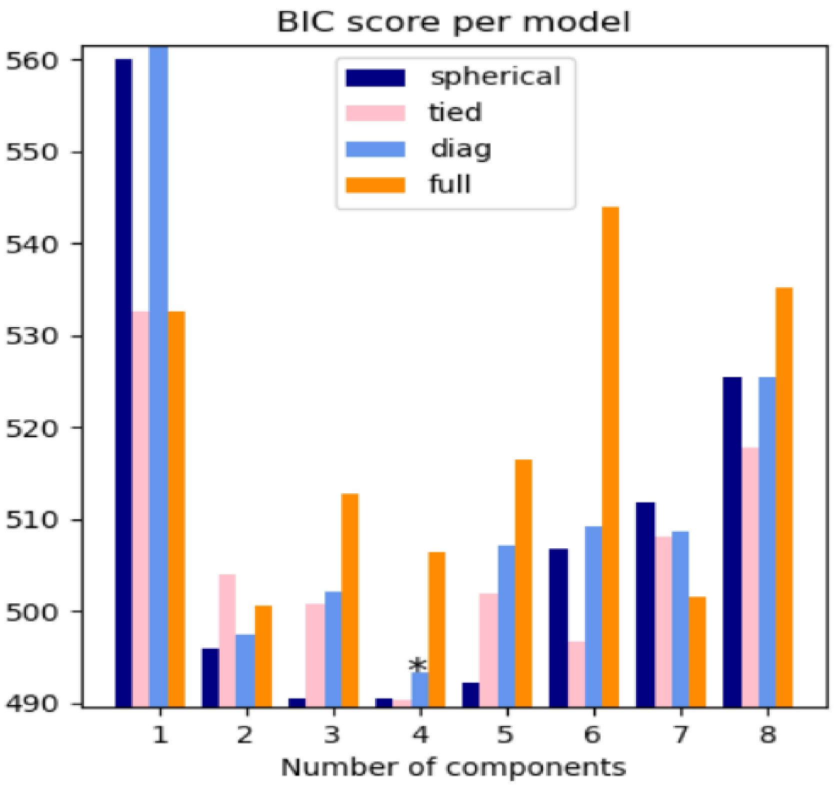
Gaussian Mixture of AIC



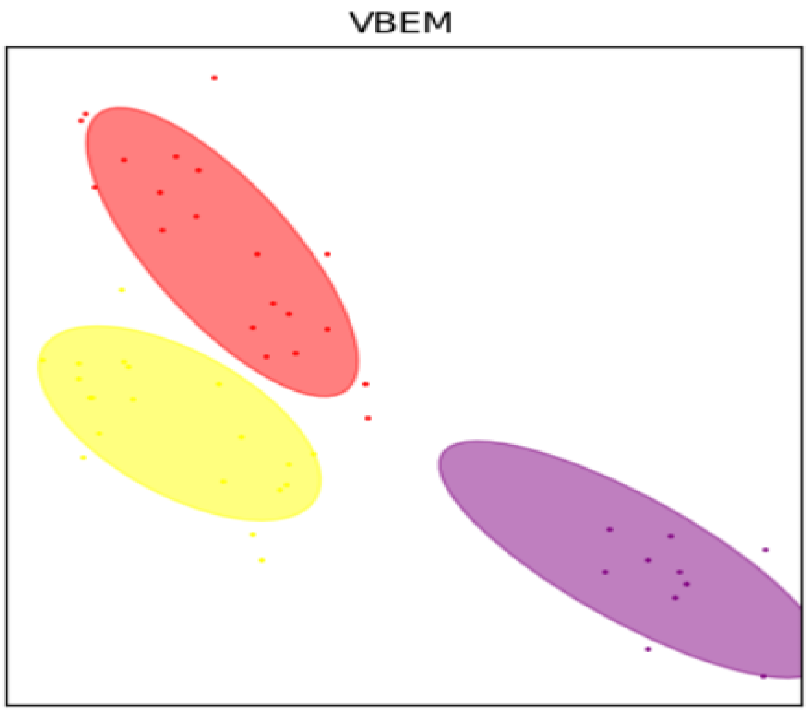


BIC:


Gaussian Mixture of BIC


BIC score per model
spherical
tied
diag
full
560
550
540
530
520
510
500
490
1
2
3
4
5
6
7
8
*
Number of components

VBEM:



Appendix:

K_means_RPCL.py

```
from math import exp
import numpy as np
import matplotlib.pyplot as plt
from sklearn.datasets import make_blobs


def distance(x1, x2):
    return (x1[0] - x2[0]) ** 2 + (x1[1] - x2[1]) ** 2


def dataset_construct():
    X_ps, Y_ps = make_blobs(n_samples=[100, 100, 100], centers=[[0,
0], [0, 1], [1, -1]], cluster_std=0.2)
```

```
    plt.scatter(X_ps[:, 0], X_ps[:, 1], c=[["lightgreen", "tomato",
"lightblue"][k] for k in Y_ps], alpha=0.4)
    plt.show()
    return X_ps, Y_ps


class DataCenter(object):
    def __init__(self, x=0, y=0):
        self.x = x
        self.y = y
        self.data = None
        self.count = 0


class MyKmeans(object):
    def __init__(self, init_centers=10):
        self.init_centers = init_centers
        self.data, self.Y = dataset_construct()
        self.centers = []
        self.rho = []
        self.alpha_c = 0.05
        self.alpha_r = 0.05
        self.theta_1 = 0.01
        self.theta_2 = 45

    def initialization(self):
        for i in range(self.init_centers):
            self.centers.append(
                DataCenter(x=np.random.normal(loc=0.5, scale=0.5),
y=np.random.normal(loc=0., scale=0.5)))

    def get_density(self):
        b = 0.
        for i in self.data:
            b += distance(self.data[0], i)
        for i in self.data:
            v = 0.0
            for j in self.data:
                v += distance(i, j) / b
            self.rho.append(exp(-v))

    def visualize(self, fp=None, color="red"):
        plt.figure()
        plt.scatter(self.data[:, 0], self.data[:, 1])
        for c in self.centers:
            if c != None:
                plt.scatter(c.x, c.y, c=color, s=25)
        if fp == None:
            fp = 'images/tmp.png'
        plt.savefig(fp)
        plt.show()

    def RPCL(self):
        self.get_density()
        self.initialization()
        self.visualize(fp="images/RPCL_init.png", color='black')
        dx = 0
        dy = 0
        while True:
            for idx, i in enumerate(self.data):
                dists = []
                for c in self.centers:
                    dists.append(distance(i, (c.x, c.y)))
                c_idx = np.argmax(dists)  # winner index
                self.centers[c_idx].x += self.alpha_c *
self.rho[idx] * (i[0] - self.centers[c_idx].x)
                self.centers[c_idx].y += self.alpha_c *
self.rho[idx] * (i[1] - self.centers[c_idx].y)

                dists[c_idx] = -1000
                r_idx = np.argmax(dists)  # second winner index
                dx = self.alpha_r * self.rho[idx] * (i[0] -
self.centers[r_idx].x)
                dy = self.alpha_r * self.rho[idx] * (i[1] -
self.centers[r_idx].y)
                self.centers[r_idx].x -= dx
                self.centers[r_idx].y -= dy
            if (dx + dy) < self.theta_1:
                break
        # assign each point to nearest center
        for i in self.data:
            d_max = 100000
            index = None
            for idx, c in enumerate(self.centers):
                d = distance(i, (c.x, c.y))
                if d < d_max:
                    d_max = d
                    index = idx
            self.centers[index].count += 1
        # remove extra centers
        remove_list = []
        for idx, c in enumerate(self.centers):
            if c.count < self.theta_2:
                remove_list.append(idx)
                self.centers[idx] = None
        print("Removed centers:\n", remove_list)
        self.visualize(fp="images/RPCL.png", color='red')


if __name__ == "__main__":
    # X_ps, Y_ps = dataset_construct()
    model = MyKmeans(init_centers=10)
    model.RPCL()
```