# AI_hw5

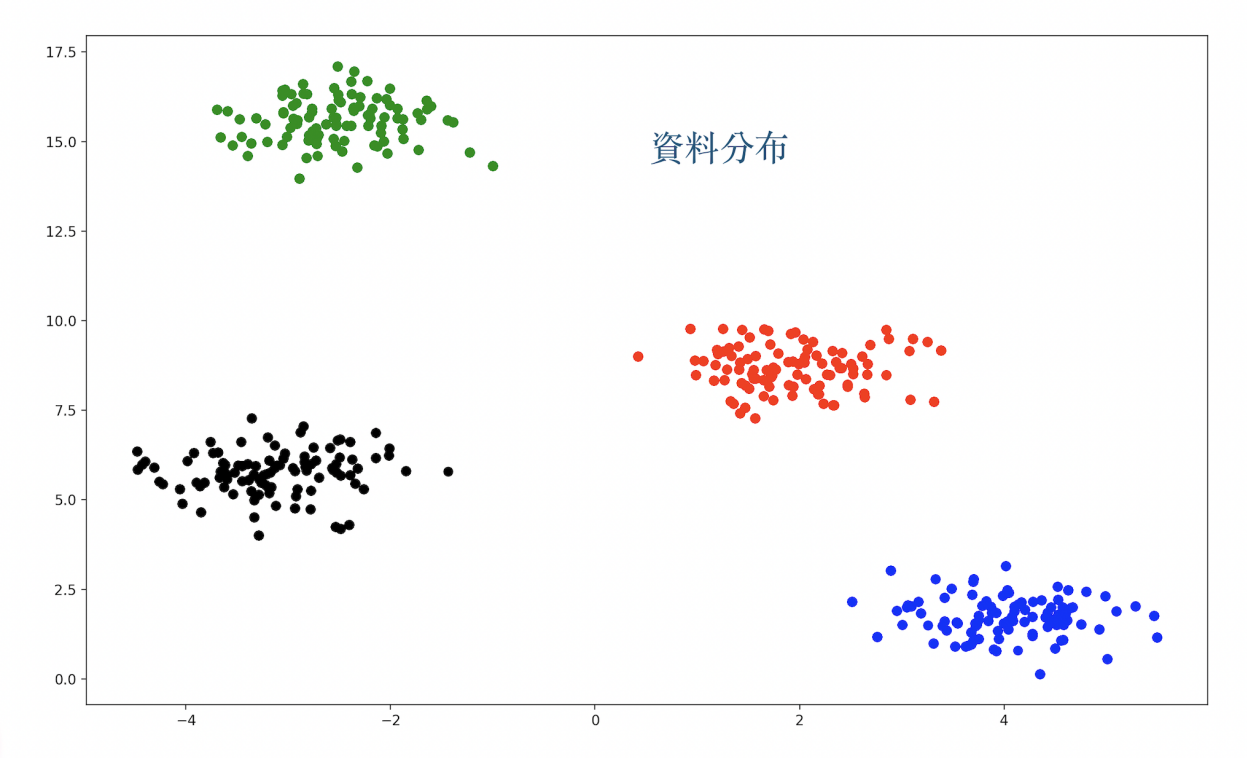
NE6101034 AI 碩二 柳譯筑

## 題目

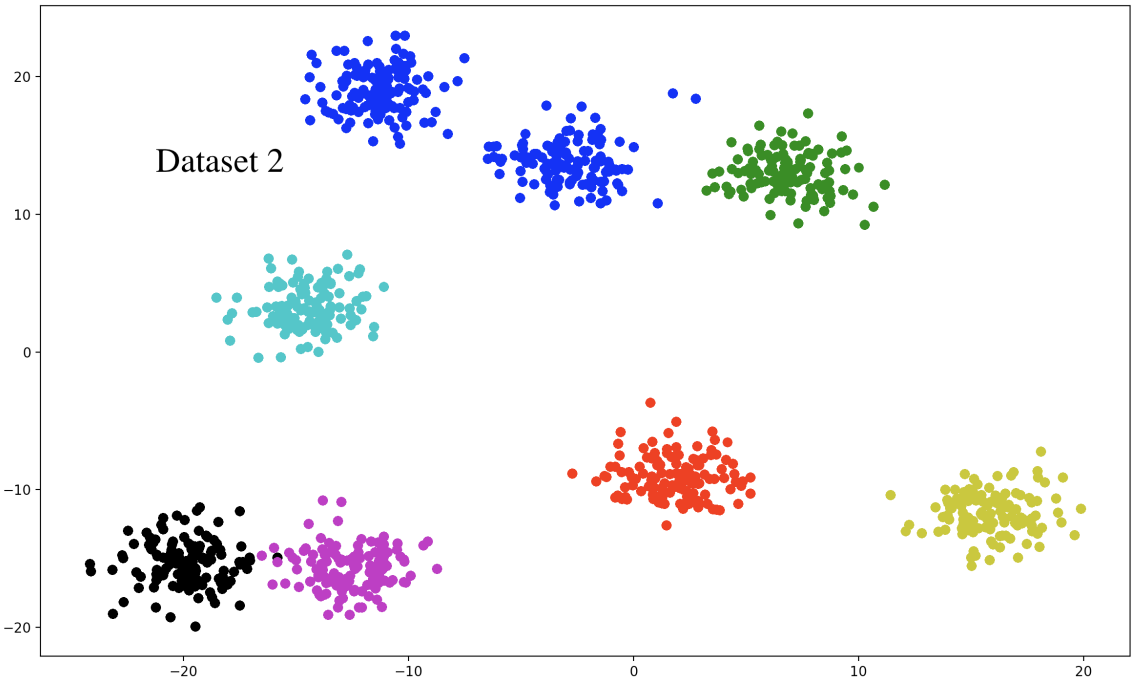
利用高斯混合模型（Gaussian Mixture Model， GMM）來做資料的分群。

## 資料分布

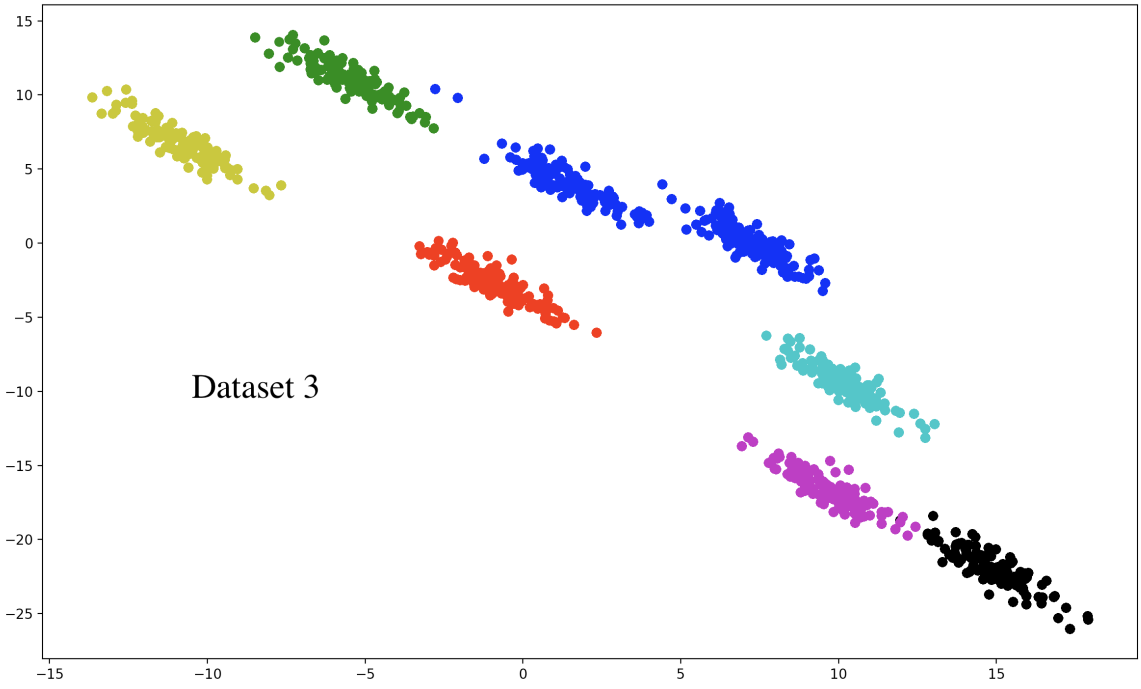
- 第一個 dataset 主要是由四群資料組成




- dataset 2 主要是由八群資料組成




- dataset 3 主要是由八群資料組成



# GMM模型概念

## Gaussian Model

- GMM 的概念講義有做了以下解釋

  - Let us begin with a very simple case: learning the parameters of a Gaussian density function on a single variable. That is, the data are generated as follows:

$$P(x) = \frac{1}{\sqrt{2\pi}\sigma} e^{-\frac{(x-\mu)^2}{2\sigma^2}}$$

    The parameters of this model are the mean $\mu$ and the standard deviation $\sigma$.

  - Let the observed values be $x_1, ..., x_N$. Then the log likelihood is

$$L = \sum_{j=1}^{N} \log \frac{1}{\sqrt{2\pi}\sigma} e^{-\frac{(x_j-\mu)^2}{2\sigma^2}} = N(-\log\sqrt{2\pi} - \log\sigma) - \sum_{j=1}^{N} \frac{(x_j-\mu)^2}{2\sigma^2}$$

  - Setting the derivatives as zero, we obtain

$$\begin{aligned} \frac{\partial L}{\partial \mu} &= -\frac{1}{\sigma^2}\sum_{j=1}^{N}(x_j-\mu) = 0 \quad &\Rightarrow \quad \mu &= \frac{\sum_j x_j}{N} \\ \frac{\partial L}{\partial \sigma} &= -\frac{N}{\sigma} + \frac{1}{\sigma^3}\sum_{j=1}^{N}(x_j-\mu)^2 = 0 \quad &\Rightarrow \quad \sigma &= \sqrt{\frac{\sum_j (x_j-\mu)^2}{N}}. \end{aligned} \tag{20.4}$$

## Gaussian Mixture Model

- Let the random variable $C$ denote the component, with values $1, \ldots, k$; then the mixture distribution is given by $P(\mathbf{x}) = \sum_{i=1}^{k} P(C=i)\, P(\mathbf{x} \mid C=i)$

  where $\mathbf{x}$ refers to the values of the attributes for a data point. For continuous data, a natural choice for the component distributions is the *multivariate Gaussian*, which gives the so-called **mixture of Gaussians**.

## EM algorithm

EM 概念：

- The basic idea of EM is to *pretend* that we know the parameters of the model and then to infer the probability that each data point belongs to each component. After that, we refit (整修) the components to the data, where each component is fitted to the entire data set with each point weighted by the probability that it belongs to that component. The process iterates until convergence.

上述所說的兩個步驟：

1. Expectation step:

   - **E-step**: Compute the probabilities $p_{ij} = P(C = i \mid \mathbf{x}_j)$, the probability that datum $\mathbf{x}_j$ was generated by component $i$. By Bayes' rule, we have $p_{ij} = \alpha\, P(\mathbf{x}_j | C = i)P(C = i)$. The term $P(\mathbf{x}_j | C = i)$ is just the probability at $\mathbf{x}_j$ of the $i$th Gaussian, and the term $P(C = i)$ is just the *weight parameter* for the $i$th Gaussian.
   - Define $n_i = \sum_j p_{ij}$, the effective number of data points currently assigned to component $i$.

2. Maximization step

   - **M-step:** Compute the new *mean*, *covariance*, and component *weights* using the following steps in sequence:

$$\boldsymbol{\mu}_i \leftarrow \sum_j p_{ij}\mathbf{x}_j / n_i$$

$$\boldsymbol{\Sigma}_i \leftarrow \sum_j p_{ij}(\mathbf{x}_j - \boldsymbol{\mu}_i)(\mathbf{x}_j - \boldsymbol{\mu}_i)^\top / n_i$$

$$w_i \leftarrow n_i / N$$

- 總之，就是用 EM algorithm，讓資料 fit 一個模型，這個模型是由 k 個 components (Gaussian models) 組成，我們可以決定總共幾個 components。原本不知道每座 components 的 means 跟

variances 。初始化隨機的 means, variances，然後 iterate 去估算 means, variances。

## 實作與架構

- 程式架構

```
hw5_NE6101034.zip
|
|-GMM.py
|-report.pdf
```

- requirements

```
matplotlib==3.4.2
numpy==1.20.3
scikit_learn==1.1.3
```

- 執行程式

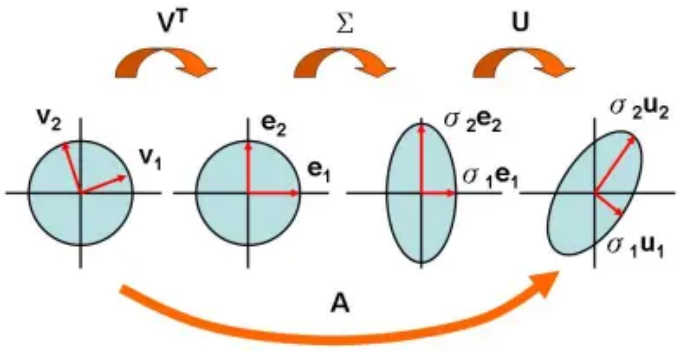
```
python GMM.py <n_components> <max_iteration>
```

- 實作
  - 我套用了 sklearn.mixture 的 GaussianMixture

```
from sklearn.mixture import GaussianMixture

gmm = GaussianMixture(n_components = components, max_iter = max_iteration, init_params = 'kmeans')
gmm = gmm.fit(x1)
pred = gmm.predict(x1)
```

  - 參考網路上資源，用 matplotlib.patches 的 Ellipse 畫出符合每個 component 分布的橢圓
    - 要畫出這個橢圓會牽扯到一系列矩陣運算，大致上的概念就是去計算兩個變異數的共變異數(Covariance)，然後用numpy.linalg的奇異值分解svd求出他的U矩陣跟s(sigma)，藉由arctan得到旋轉角度，橢圓的寬跟高就是2*sqrt(sigma)




```
from matplotlib.patches import Ellipse

def draw_ellipse(position, covariance, ax=None, **kwargs):
    #Draw an ellipse with a given position and covariance
    ax = ax or plt.gca()
    # Convert covariance to principal axes
    #covariance.shape == (2, 2):
    U, s, Vt = np.linalg.svd(covariance)
    angle = np.degrees(np.arctan2(U[1, 0], U[0, 0]))
    width, height = 2 * np.sqrt(s)
    # Draw the Ellipse
    for nsig in range(1, 4):
        ax.add_patch(Ellipse(position, nsig * width, nsig * height,
                             angle, **kwargs))


ax = plt.gca()
ax.scatter(x1_1, x1_2, c=pred, s=40, cmap='viridis', zorder=2)
ax.axis('equal')

w_factor = 0.2 / gmm.weights_.max()
for pos, covar, w in zip(gmm.means_, gmm.covariances_, gmm.weights_):
    draw_ellipse(pos, covar, alpha=w * w_factor)
plt.show()
```

# 結果與分析

這次實作我挑了幾個參數做探討：

**提供實驗數據 Dataset**

- Dataset 1
- Dataset 2
- Dataset3

**independent variable**

- n_components :

  user define
- init_params :

  ‘kmeans’ : responsibilities are initialized using kmeans

  ‘ random’ : responsibilities are initialized randomly
- max_iter :

  user define

**dependent variable**

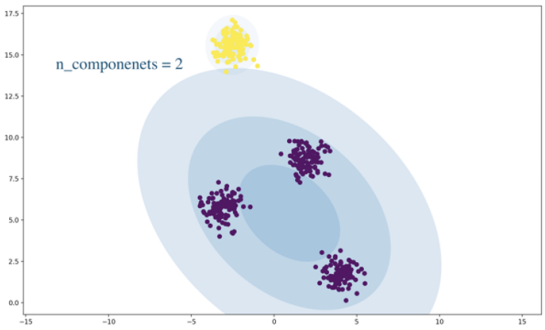
- means
- weights
- covariances

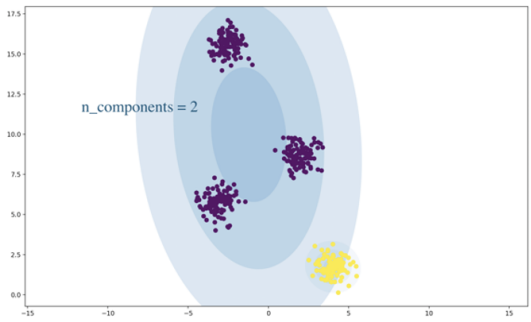
# 探討 n_components（independent variable） 跟 mean, Covariance, weight (dependent variable) 關係

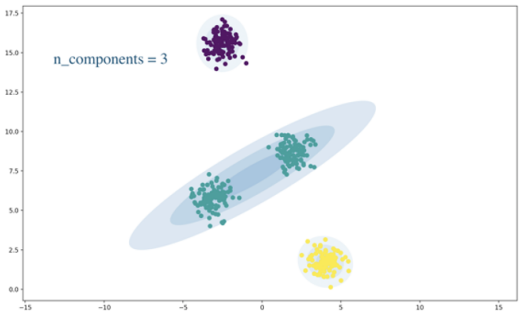
## 這個實驗我用 Dataset 1 做測試， visualize 不同群的標準差

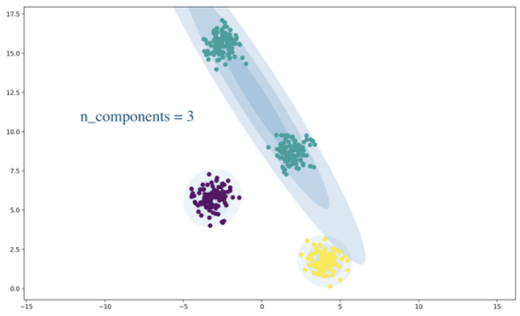
Parameters: Only change n_components, others: default

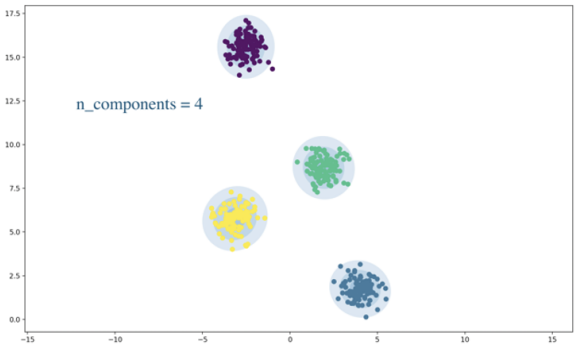
- n_components < Ground n_components
  - model 會將靠近的群合成一群，每次混合的群不一定一樣



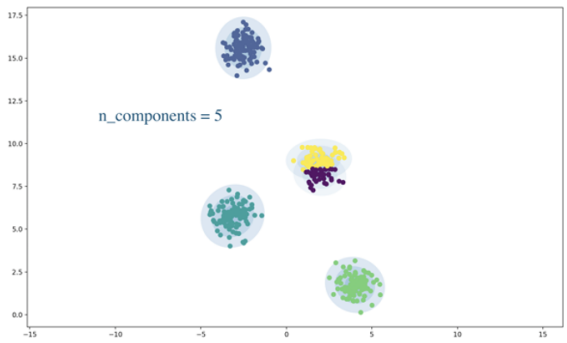



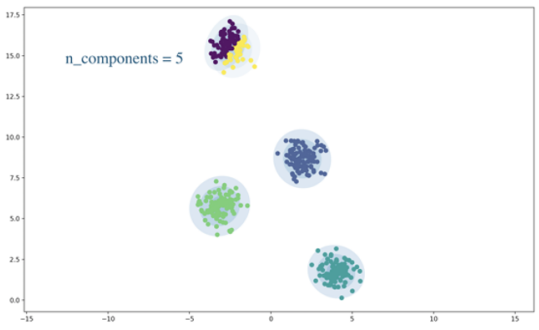



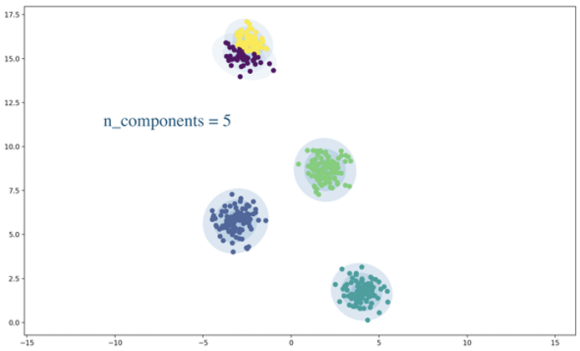



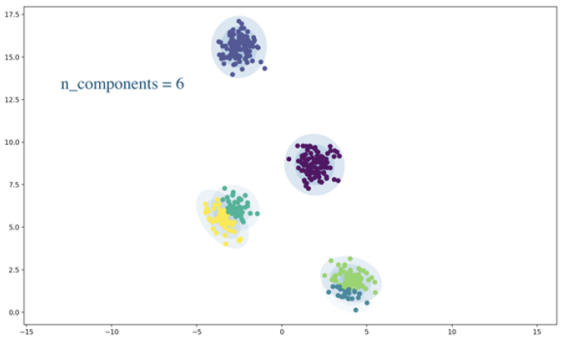




- n_components > Ground n_components
  - model 會將某一群切割成兩群，每次被分割的群不一定一樣




- output 一下參數
  - 針對不同的維度，每次可能會收斂到不同的數值，在 n_components < ground n_components 的時候光從數據看，X1 mean, X2 mean 可能會差很多，因為是好幾個群中心的中心
  - 不過當 n_components > ground n_components 並且的時候會發現有幾群的 X1 mean, X2 mean 是一模一樣或者是很近的數值
  - weights 則是看有多少的 data 被歸類為那一群，n_components = 4 的時候就完全符合 dataset 分布 (0.25, 0.25, 0.25, 0.25)

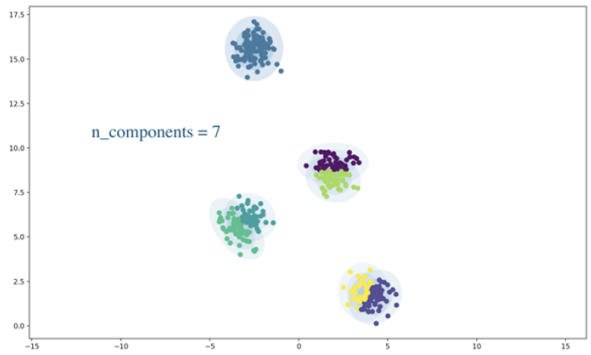
| | X1 mean | X2 mean | Covariance | weights |
|---|---|---|---|---|
| n_components=2 | 0.9278405 | 5.40956244 | [[ 9.37629608 -3.21282416] [-3.21282416 8.57872217]] | 0.75145538 |
| | 2.52283395 | 15.59820179 | [[ 0.29102574 0.01057715] [ 0.01057715 0.36473272]]] | 0.24854462 |
| n_components=2 | 1.22216296 | 9.97365202 | [[ 5.51191027 -1.25470993] [-1.25470993 17.56092298]] | 0.75353683 |
| | 4.02144766 | 1.72999394 | [[ 0.33370137 -0.01870183] [-0.01870183 0.29054381]]] | 0.24646317 |
| | | | | |
| n_components=3 | -2.51722518 | 15.59154145 | [[ 2.99022466e-01 3.55196325e- | 0.25 |

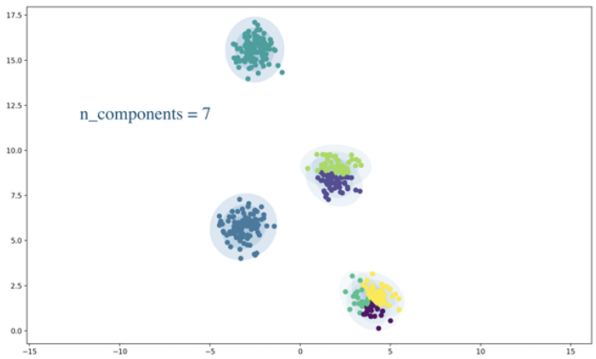
| | | | | |
|---|---|---|---|---|
| | | | 03] [ 3.55196325e-03 3.71275116e-01]] | |
| | 0.60694668 | 7.21945692 | [[ 6.78896180e+00 3.69104539e+00] [ 3.69104539e+00 2.48545241e+00]] | 0.5 |
| | 4.01189429 | 1.73712037 | [[ 3.42685724e-01 -2.61082856e-02] [-2.61082856e-02 2.99622353e-01]]] | 0.25 |
| n_components=3 | -3.14057398 | 5.76638109 | [[ 0.38820872 0.03042453] [ 0.03042453 0.38249831]] | 0.25 |
| | -0.28906504 | 12.1201286 | [[ 5.28831389 -7.74341125] [-7.74341125 12.44014185]] | 0.50060128 |
| | 4.00981912 | 1.73596232 | [[ 0.34088155 -0.02640491] [-0.02640491 0.29888045]]] | 0.24939872 |
| | | | | |
| n_components=4 | -3.14057398 | 5.76638109 | [[ 0.38820872 0.03042453] [ 0.03042453 0.38249831]] | 0.25 |
| | -2.51722518 | 15.59154145 | [[ 0.29902247 0.00355196] [ 0.00355196 0.37127512]] | 0.25 |
| | 4.01189429 | 1.73712037 | [[ 0.34268572 -0.02610829] [-0.02610829 0.29962235]] | 0.25 |
| | 1.92668063 | 8.67253275 | [[ 0.35118024 -0.01143895] [-0.01143895 0.36554777]] | 0.25 |
| | | | | |
| n_components=5 | 1.95836991 | 8.16776704 | [[ 0.26762507 -0.01432587] [-0.01432587 0.16662137]] | 0.11015542 |
| | -2.51722518 | 15.59154145 | [[ 0.29902247 0.00355196] [ 0.00355196 0.37127512]] | 0.25 |
| | -3.14057398 | 5.76638109 | [[ 0.38820872 0.03042453] [ 0.03042453 0.38249831]] | 0.25 |
| | 4.01189429 | 1.73712037 | [[ 0.34268572 -0.02610829] [-0.02610829 0.29962235]] | 0.25 |
| | 1.90171901 | 9.07013611 | [[ 0.41558245 0.01335967] [ 0.01335967 0.16345689]]] | 0.13984458 |
| n_components=5 | -2.66173057 | 15.11272277 | [[ 0.37144768 -0.07249315] [-0.07249315 0.19753491]] | 0.109773 |
| | -3.14057398 | 5.76638109 | [[ 0.38820872 0.03042453] [ 0.03042453 0.38249831]] | 0.25 |
| | 4.01189429 | 1.73712037 | [[ 0.34268572 -0.02610829] [-0.02610829 0.29962235]] | 0.25 |

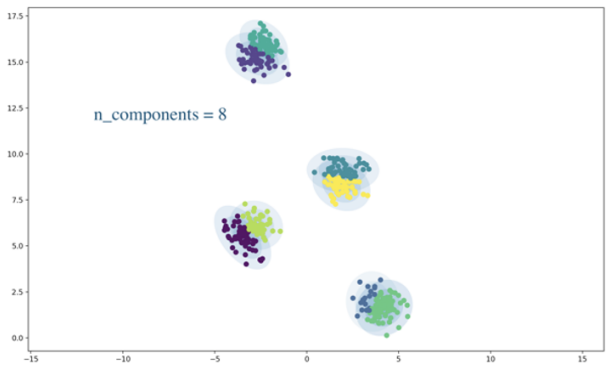
| | | | | |
|---|---|---|---|---|
| | 1.92668063 | 9.07013611 | [[ 0.35118024 -0.01143895]<br>[-0.01143895 0.36554777]] | 0.25 |
| | -2.40410296 | 15.96637197 | [[ 0.2131829 -0.03348483]<br>[-0.03348483 0.18730926]] | 0.140227 |

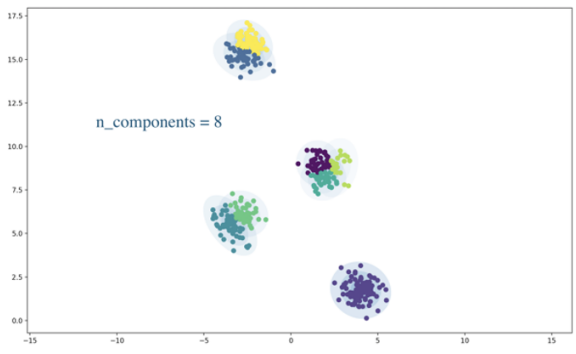
- n_components = 7 的時候，分析兩次跑的結果可以發現它可能會把一群切成三份，這樣會有兩群是完全正確的。也有可能把三群各切成兩份，這樣有一群會是對的。

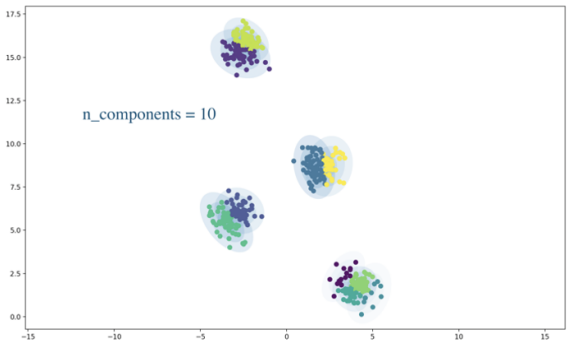
  關於 n_components 分析的數據就大概呈現到這邊。



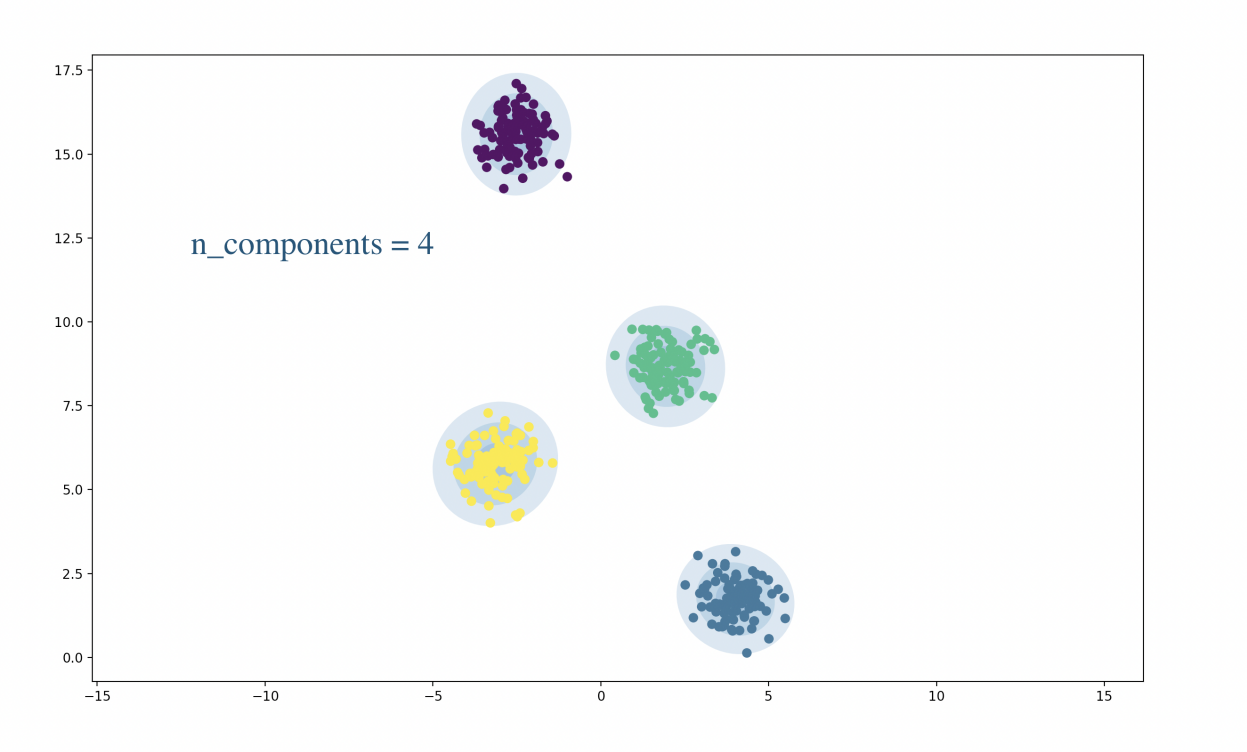



| | X1 mean | X2 mean | Covariance | weights |
|---|---|---|---|---|
| n_components=6 | -2.332678 | 16.02526492 | [[ 0.19319393 -0.0634859 ] [-0.0634859 0.17699123]] | 0.11999301 |
| | 1.92668063 | 8.67253275 | [[ 0.35118024 -0.01143895] [-0.01143895 0.36554777]] | 0.25 |
| | 4.25553155 | 2.01287528 | [[ 0.27704198 -0.10486294] [-0.10486294 0.1954363 ]] | 0.14090068 |
| | -3.14057398 | 5.76638109 | [[ 0.38820872 0.03042453] [ 0.03042453 0.38249831]] | 0.25 |
| | 3.69723921 | 1.38098567 | [[ 0.25179444 -0.12322471] [-0.12322471 0.20913978]] | 0.10909932 |
| | -2.68755732 | 15.1912262 | [[ 0.33625205 -0.07663747] [-0.07663747 0.21671556]]] | 0.13000699 |
| n_components=6 | 4.01189429 | 1.73712037 | [[ 0.34268572 -0.02610829] [-0.02610829 0.29962235]] | 0.25 |
| | -2.17413116 | 15.81665357 | [[ 0.19685307 -0.13115126] [-0.13115126 0.34172931]] | 0.12702486 |
| | -2.73074394 | 6.12956178 | [[ 0.22826263 -0.0255614 ] [-0.0255614 0.19850682]] | 0.11381606 |
| | 1.92668063 | 8.67253275 | [[ 0.35118024 -0.01143895] [-0.01143895 0.36554777]] | 0.25 |
| | -2.87161771 | 15.35901612 | [[ 0.1573724 -0.01949216] [-0.01949216 0.2953816 ]] | 0.12297514 |
| | -3.48309047 | 5.46285192 | [[ 0.26419297 -0.15114421] [-0.15114421 0.33390377]] | 0.13618394 |

| | X1 mean | X2 mean | Covariance | weights |
|---|---|---|---|---|
| n_components=7 | 1.91841256 | 9.1314746 | [[ 0.4434458 0.01060235] [ 0.01060235 0.1519269 ]] | 0.11832549 |
| | | | | |

| | | | | |
|---|---|---|---|---|
| | 4.28887681 | 1.61674537 | [[ 0.25332747 0.03573939] [ 0.03573939 0.26472219]] | 0.14006908 |
| | -2.51722518 | 15.59154145 | [[ 0.29902247 0.00355196] [ 0.00355196 0.37127512]] | 0.25 |
| | -2.73941116 | 6.12497084 | [[ 0.23242467 -0.0226766 ] [-0.0226766 0.19931972]] | 0.11519585 |
| | -3.48338462 | 5.45995095 | [[ 0.26629077 -0.1521742 ] [-0.1521742 0.33525005]] | 0.13480415 |
| | 1.93411049 | 8.26011794 | [[ 0.26815183 -0.02477166] [-0.02477166 0.19815159]] | 0.13167451 |
| | 3.65897552 | 1.89049683 | [[ 0.23423803 -0.0082998 ] [-0.0082998 0.30210355]]] | 0.10993092 |
| n_components=7 | 4.02828669 | 1.30993689 | [[ 0.24557179 -0.03949777] [-0.03949777 0.19749147]] | 0.07017479 |
| | 1.88917176 | 8.27996305 | [[ 0.27633671 -0.04934486] [-0.04934486 0.21369653]] | 0.13341914 |
| | -3.14057398 | 5.76638109 | [[ 0.38820872 0.03042453] [ 0.03042453 0.38249831]] | 0.25 |
| | -2.51722518 | 15.59154145 | [[ 0.29902247 0.00355196] [ 0.00355196 0.37127512]] | 0.25 |
| | 3.49199001 | 1.70588909 | [[ 0.19328066 -0.08897943] [-0.08897943 0.28988266]] | 0.06786454 |
| | 1.96960706 | 9.12180302 | [[ 0.43338095 -0.00419531] [-0.00419531 0.16111788]] | 0.11658086 |
| | 4.31675784 | 2.02380147 | [[ 0.23716438 -0.07245862] [-0.07245862 0.17238388]] | 0.11196067 |

## n_components = 4 （正解）



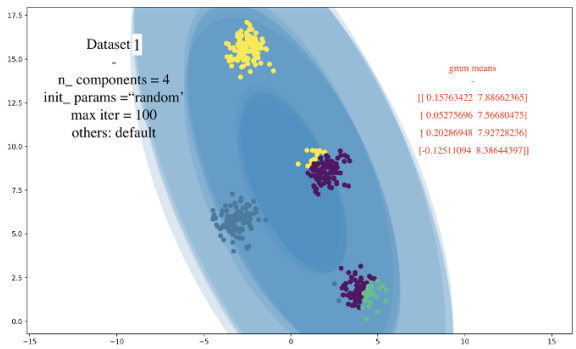

## 探討 init_params（independent variable）跟 means, (dependent variable) 關係

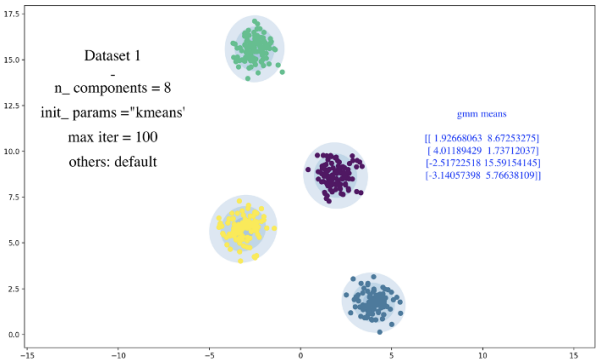
可以很明顯看出一開始初始化如果用**隨機**的結果通常很差，model 找不到各群資料分別的 means 是在哪裡，而使用 default 的 k-means, model 就可以幾乎完美找到各群資料分布

### Dataset 1

max_iter = 100

n_components = 4



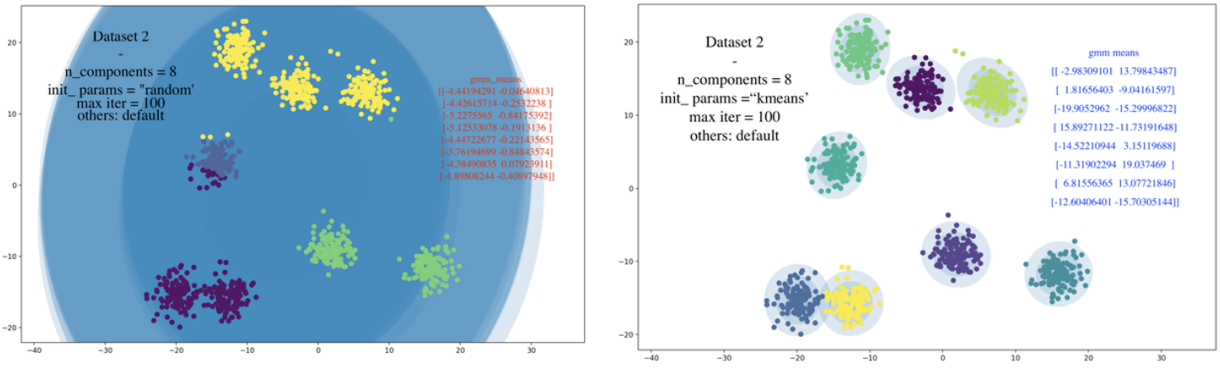



( 左邊用 random initialize, 右邊用 k-means initialize )

### Dataset 2

max_iter = 100

n_components = 8

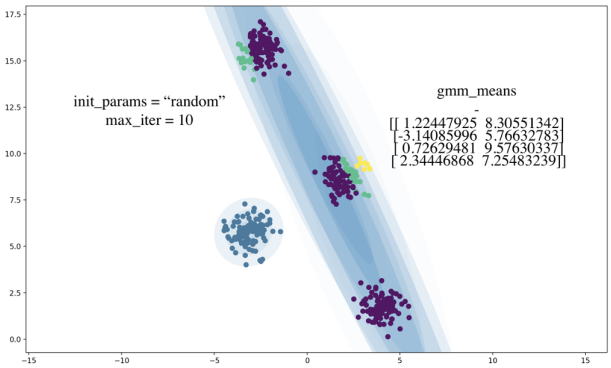


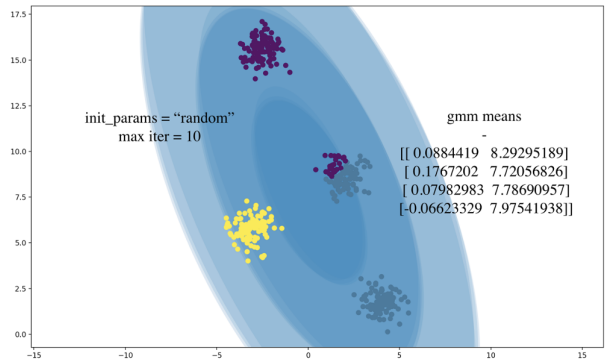

---

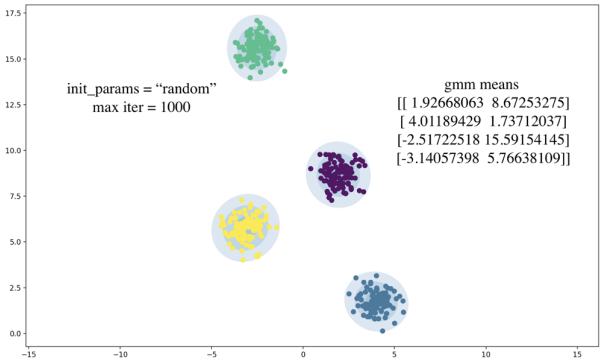
## 探討在不同 Dataset 以及不同的 max_iter 底下，經由 EM algorithm 後所得到的 means

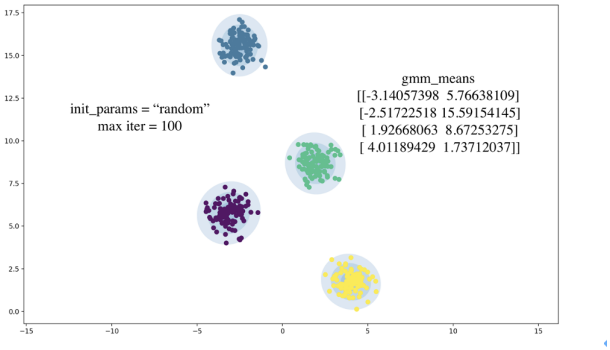
### Dataset 1

當 init_params = 'kmeans' ：從 max_iter = 1 就可以預測出準確的 means，所以我用差異比較明顯的 init_params = 'random' 來實驗：

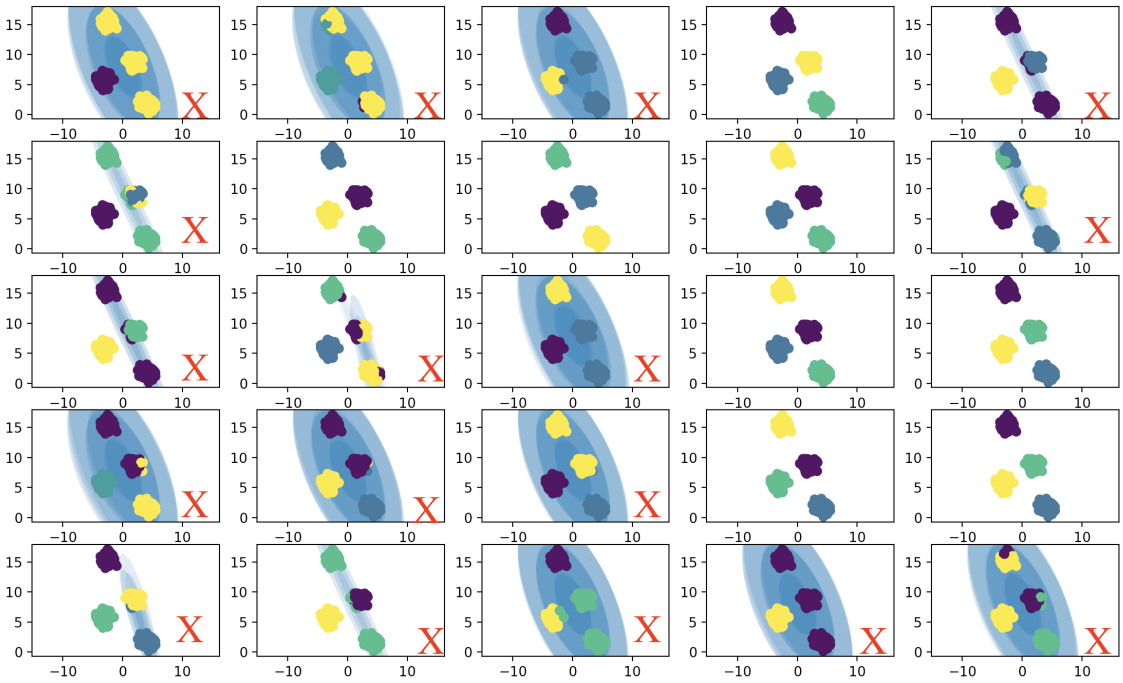


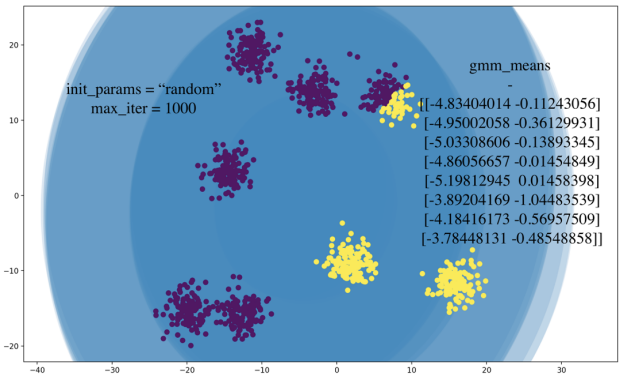



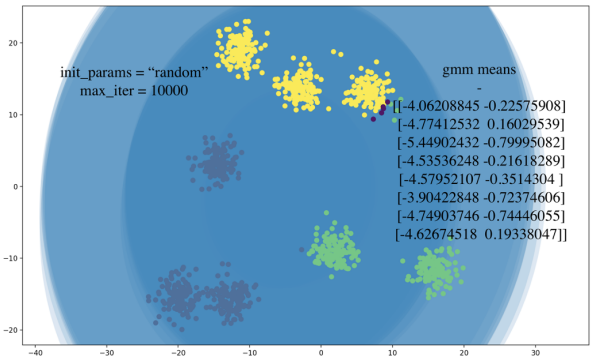

- 在 max_iter = 10 基本上得不到正確結果，當 max_iter 超過 100 之後在只有四個群的 dataset 1通常可以得到準確預測，但也有 max_iter = 1000 的時候仍然預測錯誤的狀況
  - max_iter = 1000 錯誤率仍高達 17/25



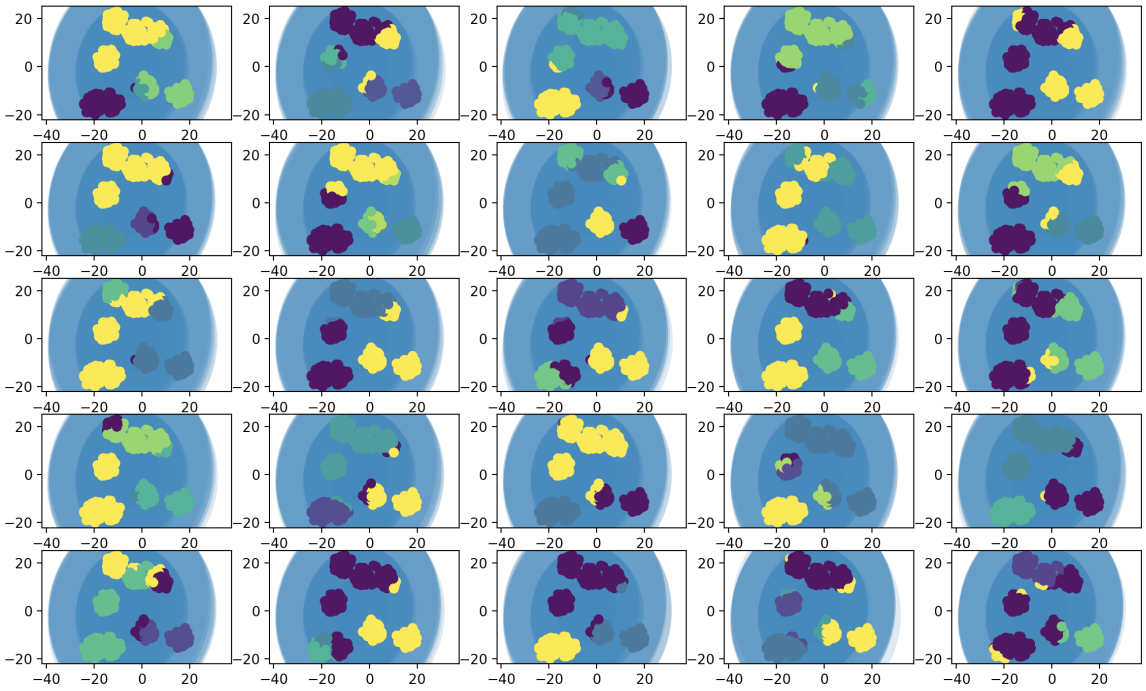
## Dataset 2

Dataset 2 跟 Dataset 由 8 個 components 組成，儘管 max_iter 給到很大，也無法改善一開始選用 random 初始化 params 的問題



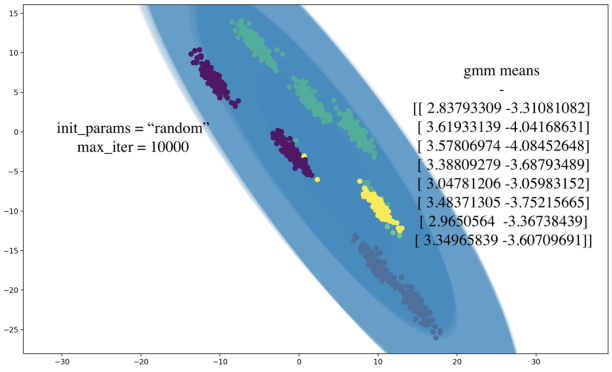



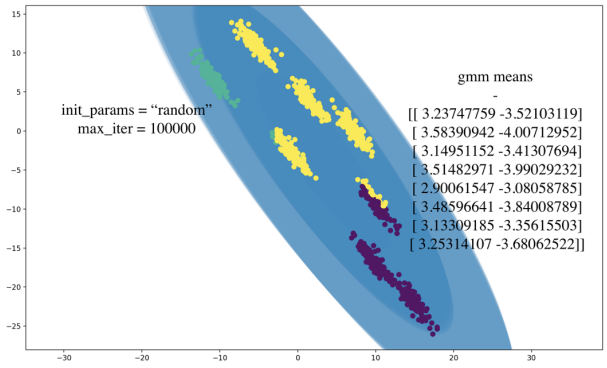

- 做 25 次，沒有正確結果
  - max_iter = 1000



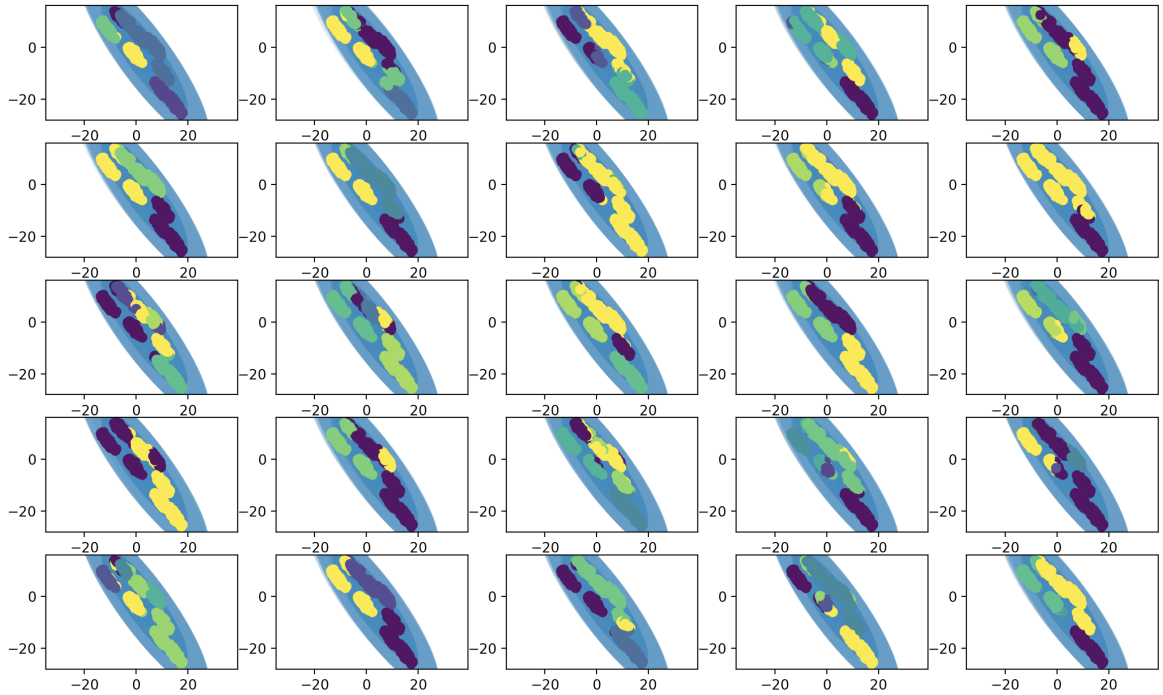
## Dataset 3

max_iter = 100000 仍然改善不了 init_params 用錯的狀況



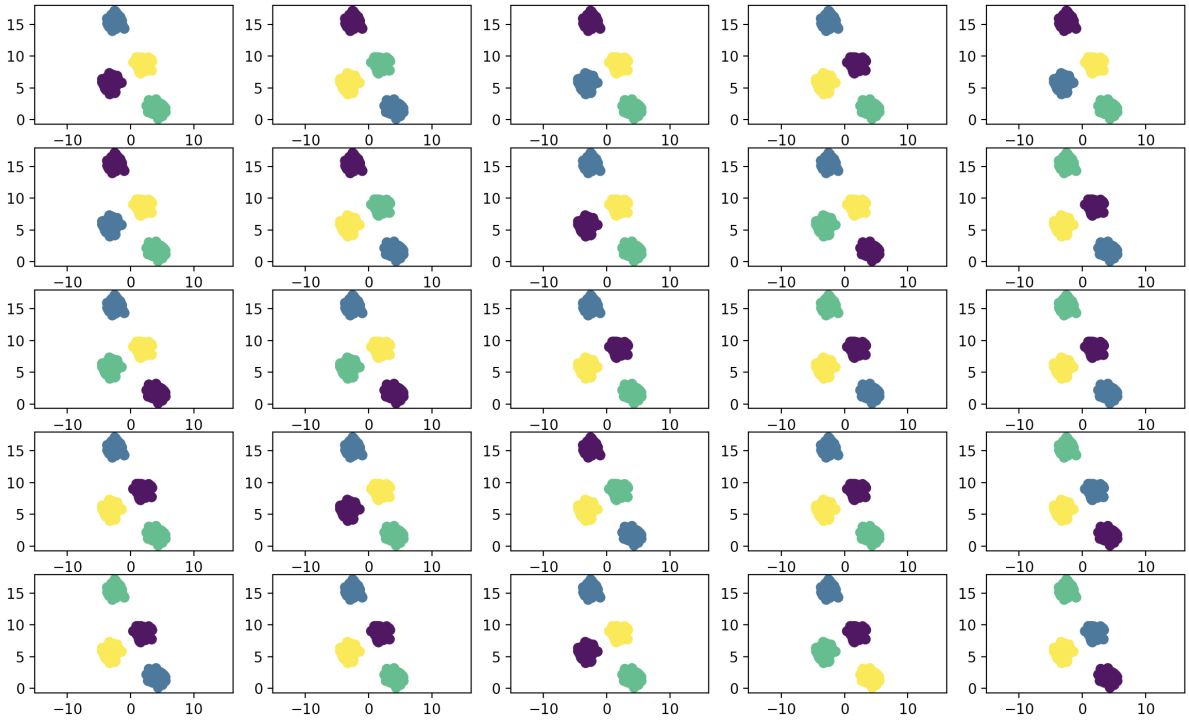

- 一樣做 25 次，沒有正確結果



## 探討不同 Dataset, K-means clustering 結果的差異

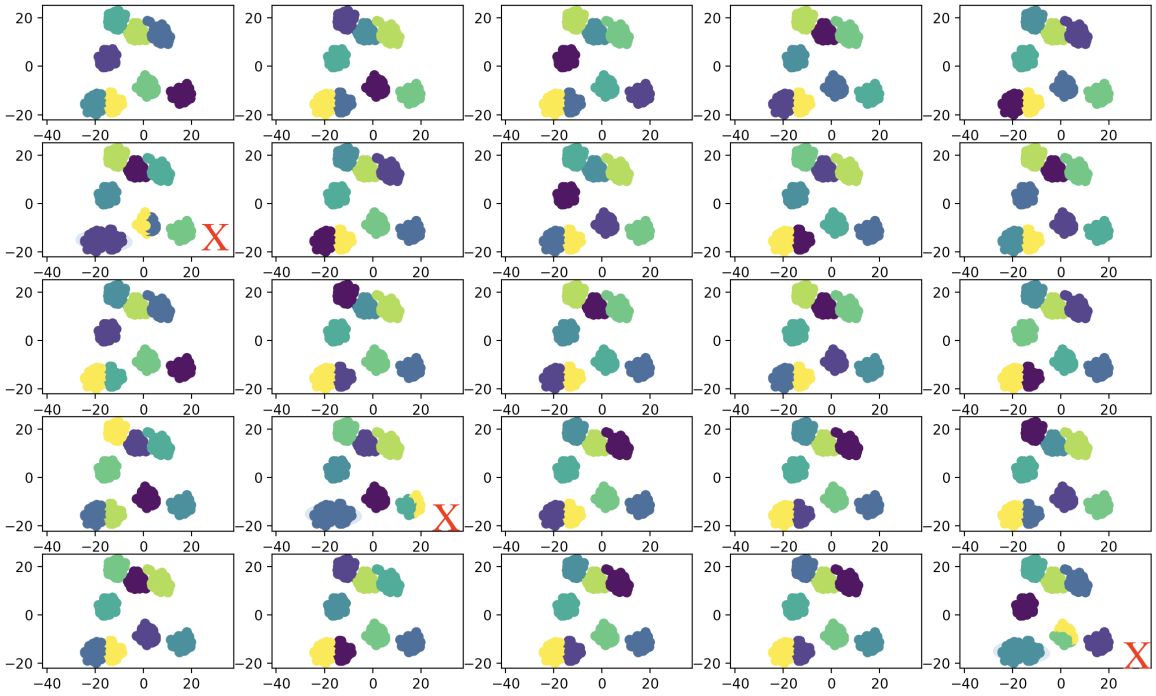
### Dataset 1

- 在 n_components 還只有四群、比較簡單的分群時，從 max_iter = 1 開始就幾乎都可以完美預測
  - n_components = 4
  - max_iter = 100

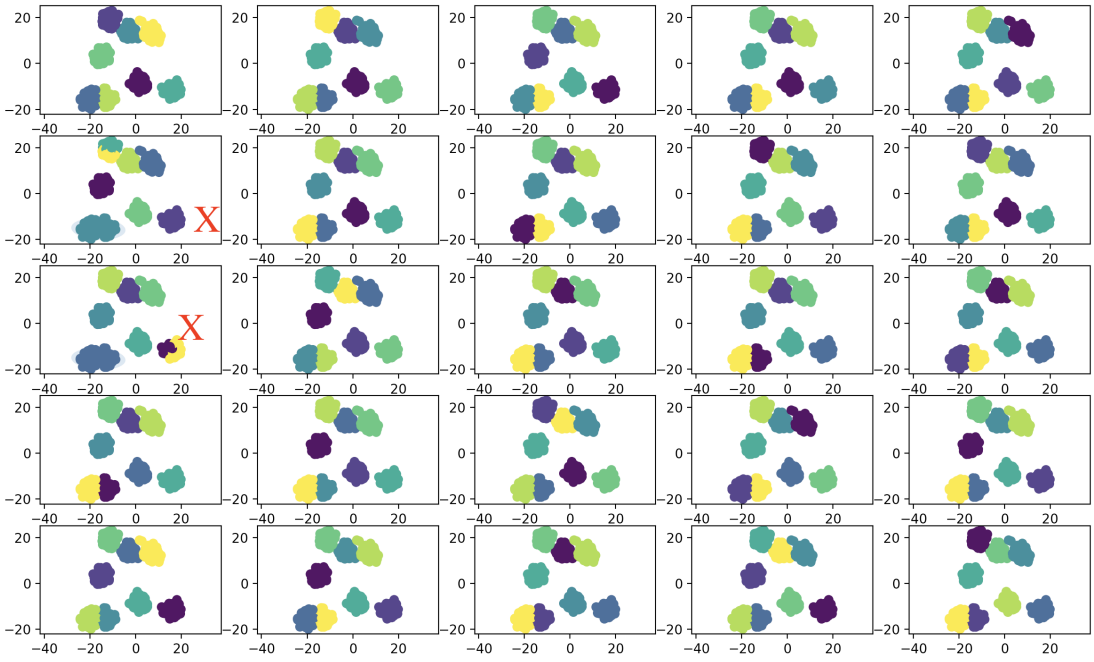


## Dataset 2

- 從 Dataset 2 跟 3 看出其實多一點群的 dataset，max_iter 設定大一點會比較準
  - n_components = 8
  - max_iter = 100

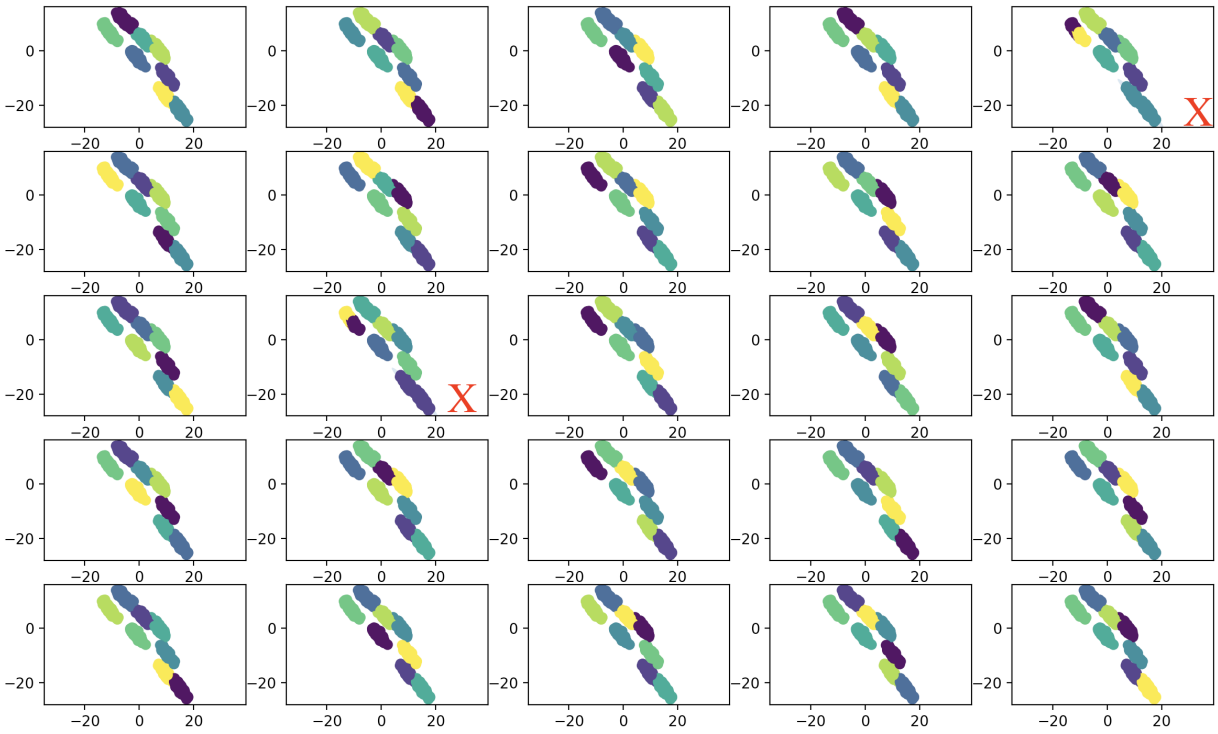


- Dataset 2 : max_iter = 1000 時仍然會有錯誤率 2/25
  - n_components = 8
  - max_iter = 1000

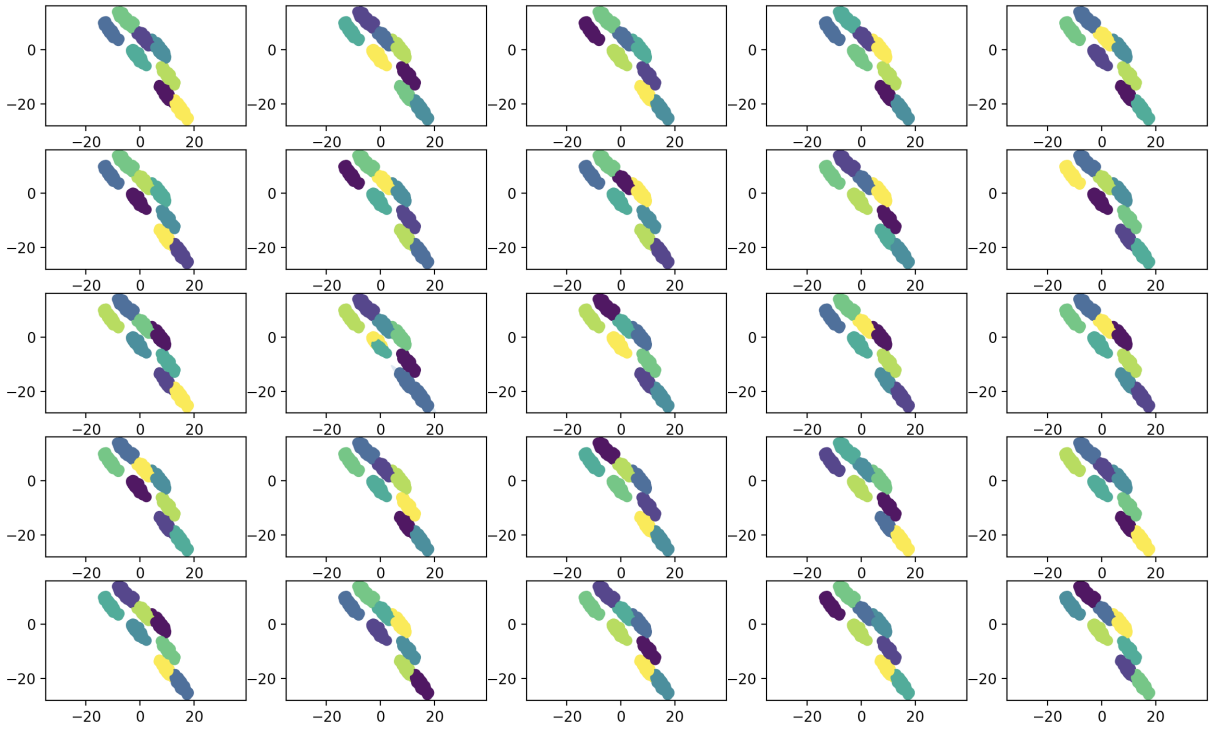


## Dataset 3

- max_iter = 100 的時候，錯誤率是 2/25
  - n_components = 8
  - max_iter = 100



- 在 max_iter = 1000 的時候分群大致上是正確的
  - n_components = 8
  - max_iter = 1000



## 結論

EM algorithm 是個偉大的演算法、結合了 least square 跟 GMM 去解決不知道一個 Dataset 是什麼樣的面貌的狀況。

這次的實作使用了 sklearn 的 GMM model，能夠使用這麼方便的套件讓我們可以花更多時間探討參數對分群結果的影響。透過這個實作可以明顯看到不同的參數對於結果的影響程度很不一樣，如果一開始就使用不對的 initialization 那 train 再多 iterator 也對 model 的收斂沒有幫助，而不同 components 數量的 dataset 在用錯參數時僥倖找到對的 means 的機率也不同，看到 plot 出來的結果真的覺得現成 model 真是方便好用又很神奇。

## Reference


https://jakevdp.github.io/PythonDataScienceHandbook/05.12-gaussian-mixtures.html

高斯混合模型（GMM）介紹以及學習筆記 - 程式人生 (796t.com)

sklearn.mixture.GaussianMixture — scikit-learn 1.1.3 documentation