# 第五次作品：探討混合常態分佈的參數估計與評估：MLE 與 GMM 方法之比較

學號：411078064

姓名：謝意盛

---

## 作品目標：

本作品針對兩組來自不同參數的混合常態分配樣本進行實驗，旨在探討利用最大概似估計法（MLE）準確估計出兩組資料各自的參數值，並通過蒙地卡羅模擬實驗，計算參數估計值的平均值（Mean）、偏差（Bias）以及均方根誤差（RMSE），以評估 MLE 的參數估計表現。此外，本作品亦採用 `sklearn.mixture.GaussianMixture` 方法對資料進行參數估計，並重複相同的模擬實驗步驟，比較該方法與 MLE 在參數估計結果上的表現差異。

---

## 目標一：

針對兩組來自不同參數的混合常態分配樣本進行實驗，探討利用最大概似估計法（MLE）準確估計出兩組資料各自的參數值。

- 機率密度函數（PDF）：

  $$f(x|\Omega) = \pi_1 f(x|\mu_1, \sigma_1^2) + \pi_2 f(x|\mu_2, \sigma_2^2)$$

  其中 $\Omega = \{\pi_1, \pi_2, \mu_1, \sigma_1^2, \mu_2, \sigma_2^2 | \pi_1 + \pi_2 = 1\}$

- 對數聯合概似函數的最大值：

  $$max_{\Omega=\{\pi_1,\pi_2,\mu_1,\sigma_1^2,\mu_2,\sigma_2^2|\pi_1+\pi_2=1,\pi_1,\pi_2,\sigma_1^2,\sigma_2^2>0\}} L(\Omega)$$

  其中目標函數為：

  $$\begin{aligned} L(\Omega) &= \ln \prod_{i=1}^{N} (\pi_1 f(x_i|\mu_1, \sigma_1^2) + \pi_2 f(x_i|\mu_2, \sigma_2^2)) \\ &= \sum_{i=1}^{N} \ln(\pi_1 f(x_i|\mu_1, \sigma_1^2) + \pi_2 f(x_i|\mu_2, \sigma_2^2)) \end{aligned}$$

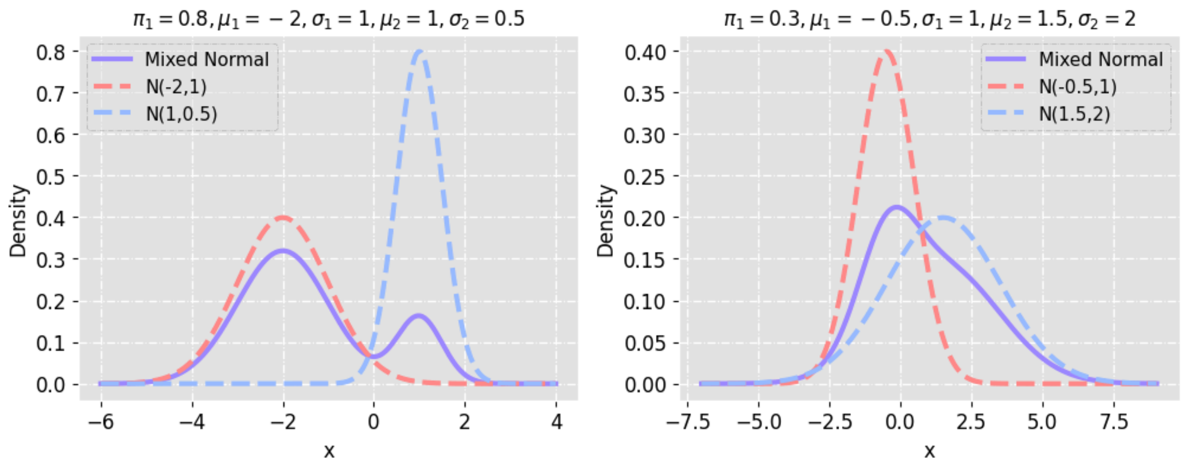
**Step 1**：分別決定出本次實驗的兩組樣本所來自的母體參數值，即混合常態分配之參數值，並繪圖呈現其分佈情況，以及加上組成該分配的兩組常態分配的分佈圖

- 第 1 組母體參數值：$\pi_1 = 0.8, \mu_1 = -2, \sigma_1 = 1, \mu_2 = 1, \sigma_2 = 0.5$
- 第 2 組母體參數值：$\pi_1 = 0.3, \mu_1 = -0.5, \sigma_1 = 1, \mu_2 = 1.5, \sigma_2 = 2$

In [45]:

```
from scipy.stats import norm, binom
from scipy.optimize import minimize
import numpy as np
import matplotlib.pyplot as plt

n = 100
```

```
pi1s = [0.8, 0.3]
mu1s, s1s = [-2, -0.5], [1, 1]
mu2s, s2s = [1, 1.5], [0.5, 2]
x_ranges = [-6, -7]
y_ranges = [4, 9]

plt.style.use('ggplot')
fig, axs = plt.subplots(1, 2, figsize = (12, 4))

for i, (pi1, mu1, s1, mu2, s2, x_range, y_range) in enumerate(zip(pi1s, mu1s, s1s,
    f = lambda x: pi1 * norm.pdf(x, mu1, s1) + (1 - pi1) * norm.pdf(x, mu2, s2)
    x = np.linspace(x_range, y_range, 1000)
    y = f(x)

    ax = axs.ravel()
    ax[i].plot(x, y, color = '#9F88FF', label = 'Mixed Normal', linewidth = 3)
    ax[i].plot(x, norm.pdf(x, mu1, s1), color = '#FF8888', linestyle = '--', linewi
               , label = 'N({},{})'.format(mu1, s1))
    ax[i].plot(x, norm.pdf(x, mu2, s2), color = '#99BBFF', linestyle = '--', linewi
               , label = 'N({},{})'.format(mu2, s2))
    ax[i].set_title('$\pi_1 = {}, \mu_1 = {}, \sigma_1 = {}, \mu_2 = {}, \sigma_2 =
                    .format(pi1, mu1, s1, mu2, s2), fontsize = 12, color = 'black')
    ax[i].set_xlabel('x', fontsize = 12, color = 'black')
    ax[i].set_ylabel('Density', fontsize = 12, color = 'black')
    ax[i].grid(True, linestyle = '-.')
    ax[i].tick_params(axis = 'both', labelsize = 12, colors = 'black')
    lgd = ax[i].legend(edgecolor = '#666666', prop = {'size': 11})
    lgd.get_frame().set_linestyle('-.')
    lgd.get_frame().set_alpha(0.4)

plt.show()
```

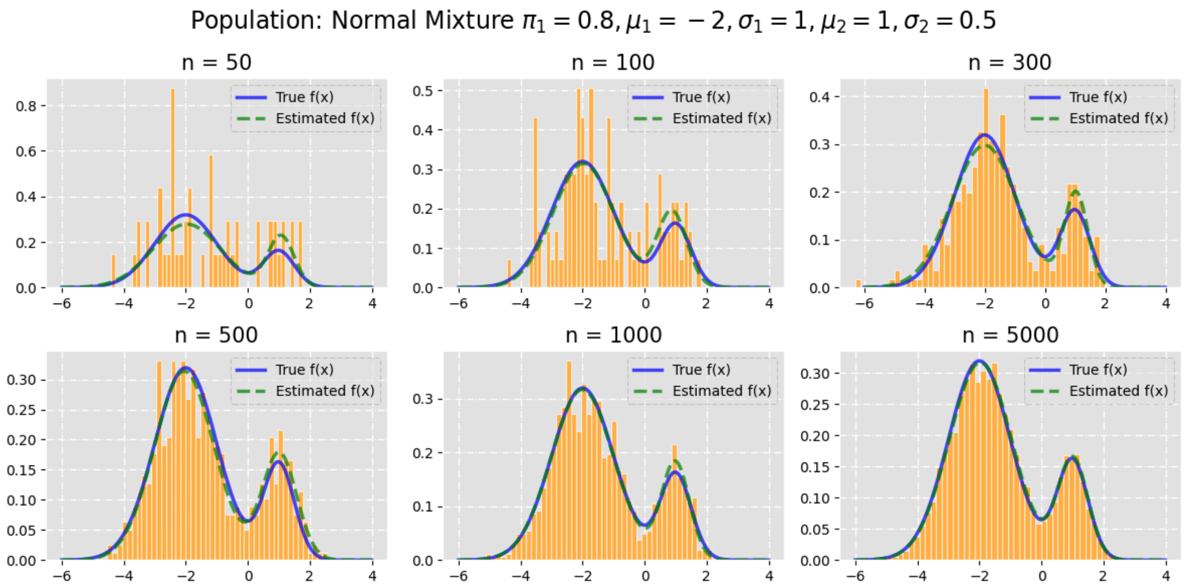


**Step 2**：生成 n 個來自混合常態的隨機樣本，其中分佈參數為 $\pi_1 = 0.8, \mu_1 = -2, \sigma_1 = 1, \mu_2 = 1, \sigma_2 = 0.5$，樣本數為 n = 50，100，300，500，1000，5000。針對每個樣本數 n 畫出其分佈直方圖，並利用 MLE 估計其參數值，畫出其樣本分佈圖，以及真實母體分佈圖，觀察它們之間隨著樣本數增加而產生的變化以及差異。

In [4]:

```
from scipy.stats import norm, binom
from scipy.optimize import minimize
import numpy as np
import matplotlib.pyplot as plt

# 設定參數
n = [50, 100, 300, 500, 1000, 5000]
pi1 = 0.8
mu1, s1 = -2, 1
mu2, s2 = 1, 0.5
```

```
# 理論混合常態分配 pdf
f = lambda x: pi1 * norm.pdf(x, mu1, s1) + (1 - pi1) * norm.pdf(x, mu2, s2)
x = np.linspace(-6, 4, 1000)
y = f(x)

# 設定演算法初始設定
params0 = [0.8, -2, 1, 1, 0.5]
bnd = [(0, 1), (None, None), (None, None), (None, None), (None, None)]
opts = {'disp': False, 'maxiter': 8000}

plt.style.use('ggplot')
fig, axs = plt.subplots(2, 3, figsize = (12, 6))

for j in range(len(n)):
    # 生成樣本
    n1 = binom.rvs(n[j], pi1)
    X = np.r_[norm.rvs(mu1, s1, size = n1), norm.rvs(mu2, s2, size = n[j] - n1)]

    # 定義 mixed normal 對數最大概似函數
    neg_log_likelihood = lambda x: -np.sum(np.log(x[0] * norm.pdf(X, x[1], x[2]) +
                                        (1 - x[0]) * norm.pdf(X, x[3], x[4])))

    # 計算最大概似估計
    result = minimize(neg_log_likelihood, params0, method = 'Nelder-Mead'
                    , bounds = bnd, options = opts)

    ax = axs.ravel()
    ax[j].plot(x, y, color = 'blue', label = 'True f(x)', linewidth = 2.5, alpha =
    ax[j].plot(x, result.x[0] * norm.pdf(x, result.x[1], result.x[2]) + \
            (1 - result.x[0]) * norm.pdf(x, result.x[3], result.x[4]), label = 'Est
            , color = 'green', linewidth = 2.5, linestyle = '--', alpha = 0.7)
    ax[j].hist(X, bins = 45, density = True, color = '#FFB347', edgecolor = 'white'
    ax[j].set_title('n = {}'.format(n[j]), fontsize = 15)
    ax[j].grid(True, linestyle = '-.')
    ax[j].tick_params(axis = 'both', labelsize = 10, colors = 'black')
    lgd = ax[j].legend(edgecolor = '#666666', prop = {'size': 10})
    lgd.get_frame().set_linestyle('-.')
    lgd.get_frame().set_alpha(0.4)

fig.suptitle('Population: Normal Mixture $\pi_1 = {}, \mu_1 = {}, \sigma_1 = {}, \m
                .format(pi1, mu1, s1, mu2, s2), fontsize = 17)
plt.tight_layout()
plt.show()
```



## 注意事項與討論：

### 觀察：

- 從直方圖和樣本分佈的圖形可以觀察到，當樣本數 n 增加時，樣本分佈與真實分佈之間的差異逐漸縮小。
- 當 n = 50 時，可見直方圖中樣本分佈的雙峰特徵並不明顯，且估計曲線受隨機波動的影響較大，與真實分佈曲線之間的吻合程度不高，但其已經開始呈現出混合常態分佈的雙峰特徵。
- 當 n 增加至 100、300、500 時，直方圖逐漸顯示出雙峰形狀，且估計曲線與真實分佈曲線也在逐漸吻合。
- 當 n 達到 1000 和 5000 時，估計曲線幾乎完全貼合於真實分佈曲線，顯示出參數估計的穩定性和準確性隨著樣本數的增加而顯著提高。

### 結論：

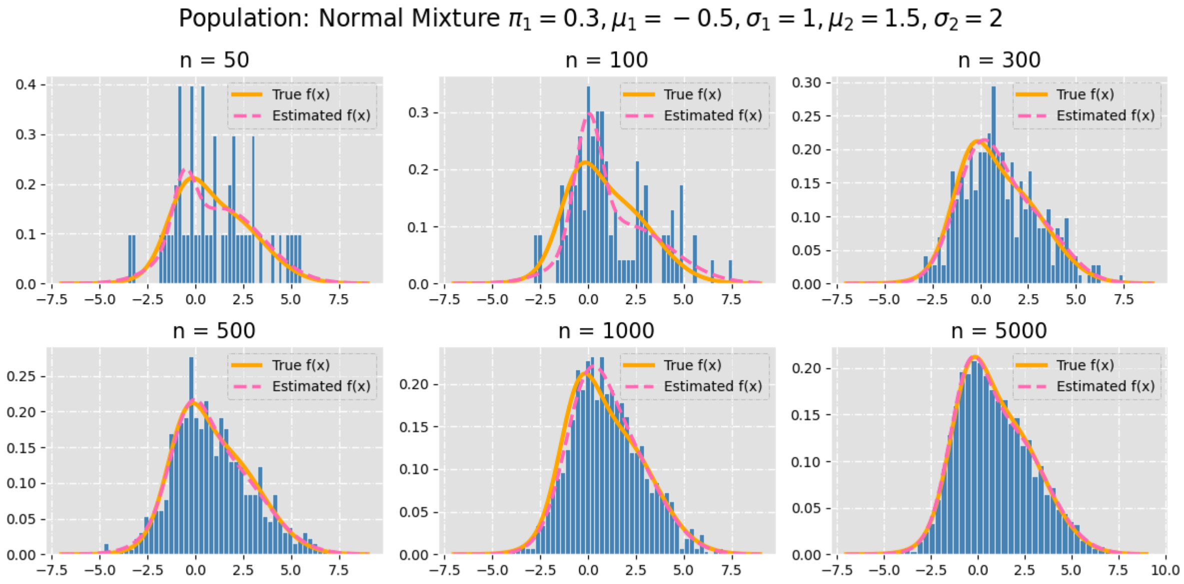
- 最大概似估計 MLE 能夠有效估計混合常態分佈的參數，且樣本數的增加能顯著提升估計結果的準確性。
- 當樣本數較少時，受隨機性影響，估計的參數值可能偏離真實值，但隨著樣本數增加，參數估計會逐漸趨於穩定，並能更準確地反映真實的母體分佈。
- 實驗結果說明了樣本數對於參數估計的重要性，尤其在分析具有複雜結構的分佈時，大樣本的支持能更清晰地揭示分佈特徵。

---

**Step 3**：與前面的實驗步驟相同，但其中分佈參數改為 $\pi_1 = 0.3, \mu_1 = -0.5, \sigma_1 = 1, \mu_2 = 1.5, \sigma_2 = 2$。

In [24]:

```
from scipy.stats import norm, binom
from scipy.optimize import minimize
import numpy as np
import matplotlib.pyplot as plt

# 設定參數
n = [50, 100, 300, 500, 1000, 5000]
pi1 = 0.3
mu1, s1 = -0.5, 1
mu2, s2 = 1.5, 2

# 理論混合常態分配 pdf
f = lambda x: pi1 * norm.pdf(x, mu1, s1) + (1 - pi1) * norm.pdf(x, mu2, s2)
x = np.linspace(-7, 9, 1000)
y = f(x)

# 設定演算法初始設定
params0 = [0.3, -0.5, 1, 1.5, 2]
bnd = [(0, 1), (None, None), (None, None), (None, None), (None, None)]
opts = {'disp': False, 'maxiter': 8000}

plt.style.use('ggplot')
fig, axs = plt.subplots(2, 3, figsize = (12, 6))

for j in range(len(n)):
    # 生成樣本
    n1 = binom.rvs(n[j], pi1)
    X = np.r_[norm.rvs(mu1, s1, size = n1), norm.rvs(mu2, s2, size = n[j] - n1)]

    # 定義 mixed normal 對數最大概似函數
    neg_log_likelihood = lambda x: -np.sum(np.log(x[0] * norm.pdf(X, x[1], x[2]) +
                                            (1 - x[0]) * norm.pdf(X, x[3], x[4])))

    # 計算最大概似估計
    result = minimize(neg_log_likelihood, params0, method = 'Nelder-Mead'
                    , bounds = bnd, options = opts)

    ax = axs.ravel()
    ax[j].plot(x, y, color = '#FFA500', label = 'True f(x)', linewidth = 3)
    ax[j].plot(x, result.x[0] * norm.pdf(x, result.x[1], result.x[2]) + \
            (1 - result.x[0]) * norm.pdf(x, result.x[3], result.x[4]), label = 'Est
            , color = '#FF69B4', linestyle = '--', linewidth = 2.5)
    ax[j].hist(X, bins = 45, density = True, color = '#4682B4', edgecolor = 'white'
    ax[j].set_title('n = {}'.format(n[j]), fontsize = 15)
    ax[j].grid(True, linestyle = '-.')
    ax[j].tick_params(axis = 'both', labelsize = 10, colors = 'black')
    lgd = ax[j].legend(edgecolor = '#666666', prop = {'size': 10})
    lgd.get_frame().set_linestyle('-.')
    lgd.get_frame().set_alpha(0.4)

fig.suptitle('Population: Normal Mixture $\pi_1 = {}, \mu_1 = {}, \sigma_1 = {}, \m
                .format(pi1, mu1, s1, mu2, s2), fontsize = 17)
plt.tight_layout()
plt.show()
```



## 注意事項與討論：

**觀察**：

- 與 Step 2 中的分佈相比，此次分佈的參數設置使得兩個常態分佈的重疊程度較高，分佈的雙峰特徵更加模糊。
- 當 n = 50 和 n = 100 時，直方圖幾乎沒有呈現出明顯的雙峰結構，難以區分兩個常態分佈的特徵。此外，估計曲線雖然已經可以看出些許的雙峰特徵，但其與真實分佈曲線的吻合程度明顯很低，容易受到隨機波動的影響。

- 隨著樣本數增加至 n = 300 和 n = 500，直方圖開始呈現常態分佈的基本形狀，但混合常態分佈的雙峰特徵仍然不夠明顯，且由於兩個常態分佈的重疊區域較廣，估計曲線仍略有偏差，但也開始逐漸向真實分佈曲線接近。
- 當樣本數進一步增加至 n = 1000 和 n = 5000 時，直方圖的形狀與真實分佈曲線逐漸吻合，估計曲線也逐漸貼合真實母體分佈，表示最大概似估計的結果幾乎完全貼合於真實分佈，但由於兩個分佈的重疊部分較大，雙峰的分界仍然不如 Step 2 那麼清晰。

**結論**：

- 與 Step 2 的實驗相比，當組成混合常態分佈的分佈參數之間差異較小時會使得在區分上更加困難，這也顯示了估計參數的難度會隨著分佈特性而改變。
- 當兩個常態分佈高度重疊時，使得小樣本下參數估計的難度更大，幾乎無法觀察到混合常態分佈的雙峰特徵，且估計參數可能會偏離真實值，進一步顯示了小樣本在高重疊分佈中的限制。
- 當樣本數足夠大，即 n = 5000 時，即使分佈重疊較大，MLE 仍然可以逼近真實分佈參數，但整體估計過程的穩定性和準確性明顯不如重疊較少的分佈，如 Step 2。
- 與 Step 2 的分佈相比，本次實驗更加強調了樣本數對於較為複雜的分佈結構的參數估計的重要性，尤其當混合常態分佈的組成分佈高度重疊時，大量的樣本數將會是準確描述分佈特徵的關鍵。

---

## 目標二：

通過蒙地卡羅模擬實驗，分別針對兩組樣本的參數估計值進行平均值（**Mean**）、偏差（**Bias**）以及均方根誤差（**RMSE**）的計算，評估 **MLE** 在參數估計上的表現。

1. Mean 計算：
   - 理論值：

$$Mean(\hat{\theta}) = E(\hat{\theta})$$

   - 實驗值：

$$\begin{aligned} Mean(\hat{\theta}) &= \frac{1}{N}\sum_{k=1}^{N}\hat{\theta}_k \\ &= \bar{\theta} \end{aligned}$$

2. Bias 計算：
   - 理論值：

$$Bias(\hat{\theta}) = E(\hat{\theta}) - \theta$$

   - 實驗值：

$$\begin{aligned} Bias(\hat{\theta}) &= \frac{1}{N}\sum_{k=1}^{N}\hat{\theta}_k - \theta \\ &= \bar{\theta} - \theta \end{aligned}$$

3. Variance 計算：

- 理論值：

$$Var(\hat{\theta}) = E[\hat{\theta} - E(\hat{\theta})]^2$$

- 實驗值：

$$Var(\hat{\theta}) = \frac{1}{N-1}\sum_{k=1}^{N}(\hat{\theta}_k - \bar{\theta})^2$$

4. RMSE 計算：

- 理論值：

$$\begin{aligned} RMSE(\hat{\theta}) &= \sqrt{MSE(\hat{\theta})} \\ &= \sqrt{E(\hat{\theta} - \theta)^2} \end{aligned}$$

- 實驗值：

$$\begin{aligned} RMSE(\hat{\theta}) &= \sqrt{MSE(\hat{\theta})} \\ &= \sqrt{\frac{1}{N}\sum_{k=1}^{N}(\hat{\theta}_k - \theta)^2} \end{aligned}$$

**Step 1**：生成 n 個來自混合常態的隨機樣本，其中分佈參數為 $\pi_1 = 0.8, \mu_1 = -2, \sigma_1 = 1, \mu_2 = 1, \sigma_2 = 0.5$，樣本數為 n = 50，100，300，500，1000，5000。針對每個樣本數 n 進行 10000 次的蒙地卡羅模擬實驗，計算參數估計值的平均值（Mean）、偏差（Bias）以及均方根誤差（RMSE），以此評估 MLE 在參數估計上的表現。

In [ ]:

```python
from scipy.stats import norm, binom
from scipy.optimize import minimize
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd

# 設定參數
N = 10000   # number of simulations
n = [50, 100, 300, 500, 1000, 5000]
pi1 = 0.8
mu1, s1 = -2, 1
mu2, s2 = 1, 0.5

# 設定演算法初始設定
params0 = [0.8, -2, 1, 1, 0.5]
bnd = [(0, 1), (None, None), (None, None), (None, None), (None, None)]
opts = {'disp': False, 'maxiter': 8000}

mean_RESULT = np.zeros((len(n), len(params0)))
bias_RESULT = np.zeros((len(n), len(params0)))
rmse_RESULT = np.zeros((len(n), len(params0)))

# 添加数值稳定性处理
def safe_norm_pdf(x, loc, scale):
    scale = np.clip(scale, 1e-10, None)  # 防止 scale 为零或接近于零
    return norm.pdf(x, loc, scale)

for j in range(len(n)):
    RESULT = np.zeros((N, 5))
    # 生成來自 normal mixture 的資料
    n1 = binom.rvs(n[j], pi1)
    SAMPLE = np.r_[norm.rvs(mu1, s1, size = (n1, N)), \
                norm.rvs(mu2, s2, size = (n[j] - n1, N))]

    for i in range(N):
        X = SAMPLE[:, i]

        neg_log_likelihood = lambda x: -np.sum(np.log(np.clip(x[0] * safe_norm_pdf(
                                    (1 - x[0]) * safe_norm_pdf(X, x[3], x[4]), 1e-1

        result = minimize(neg_log_likelihood, params0, method = 'Nelder-Mead', \
                        bounds = bnd, options = opts)
        RESULT[i, :] = result.x

    # Evaluation
    mean_RESULT[j, :] = np.mean(RESULT, axis = 0)
    bias_RESULT[j, :] = mean_RESULT[j, :] - params0
    rmse_RESULT[j, :] = np.sqrt(np.mean((RESULT - params0) ** 2, axis = 0))
```

In [ ]:

```
# 整理表格
columns = ['pi1={}'.format(pi1), 'mu1={}'.format(mu1), 's1={}'.format(s1)
           , 'mu2={}'.format(mu2), 's2={}'.format(s2)]

columns_mean = pd.MultiIndex.from_product([['Mean for the estimated parameters at N
                                            pd.Index(columns, name = 'n')])
columns_bias = pd.MultiIndex.from_product([['Bias for the estimated parameters at N
                                            pd.Index(columns, name = 'n')])
columns_rmse = pd.MultiIndex.from_product([['RMSE for the estimated parameters at N
                                            pd.Index(columns, name = 'n')])

df_mean = pd.DataFrame(mean_RESULT, index = n, columns = columns_mean)
df_bias = pd.DataFrame(bias_RESULT, index = n, columns = columns_bias)
df_rmse = pd.DataFrame(rmse_RESULT, index = n, columns = columns_rmse)

# 使用 Styler 对象来设置列名和单元格内容的对齐方式
df_mean1 = df_mean.style.set_table_styles({
    ('Mean for the estimated parameters at N = {}'.format(N), 'pi1={}'.format(pi1))
    ('Mean for the estimated parameters at N = {}'.format(N), 'mu1={}'.format(mu1))
    ('Mean for the estimated parameters at N = {}'.format(N), 's1={}'.format(s1)):
    ('Mean for the estimated parameters at N = {}'.format(N), 'mu2={}'.format(mu2))
    ('Mean for the estimated parameters at N = {}'.format(N), 's2={}'.format(s2)):
}, overwrite=False).set_properties(**{'text-align': 'center'})

df_bias1 = df_bias.style.set_table_styles({
    ('Bias for the estimated parameters at N = {}'.format(N), 'pi1={}'.format(pi1))
    ('Bias for the estimated parameters at N = {}'.format(N), 'mu1={}'.format(mu1))
    ('Bias for the estimated parameters at N = {}'.format(N), 's1={}'.format(s1)):
    ('Bias for the estimated parameters at N = {}'.format(N), 'mu2={}'.format(mu2))
    ('Bias for the estimated parameters at N = {}'.format(N), 's2={}'.format(s2)):
}, overwrite=False).set_properties(**{'text-align': 'center'})

df_rmse1 = df_rmse.style.set_table_styles({
    ('RMSE for the estimated parameters at N = {}'.format(N), 'pi1={}'.format(pi1))
    ('RMSE for the estimated parameters at N = {}'.format(N), 'mu1={}'.format(mu1))
    ('RMSE for the estimated parameters at N = {}'.format(N), 's1={}'.format(s1)):
    ('RMSE for the estimated parameters at N = {}'.format(N), 'mu2={}'.format(mu2))
    ('RMSE for the estimated parameters at N = {}'.format(N), 's2={}'.format(s2)):
}, overwrite=False).set_properties(**{'text-align': 'center'})
```

```
# 添加水平線在每個 row
df_mean1.set_table_styles([{'selector': 'tr', 'props': [('border-top', '1px solid #
df_bias1.set_table_styles([{'selector': 'tr', 'props': [('border-top', '1px solid #
df_rmse1.set_table_styles([{'selector': 'tr', 'props': [('border-top', '1px solid #
df_mean1.set_table_styles([{'selector': 'tr', 'props': [('border-bottom', '1px soli
df_bias1.set_table_styles([{'selector': 'tr', 'props': [('border-bottom', '1px soli
df_rmse1.set_table_styles([{'selector': 'tr', 'props': [('border-bottom', '1px soli

# 显示 DataFrame
display(df_mean1)
display(df_bias1)
display(df_rmse1)
```

**Mean for the estimated parameters at N = 10000**

| n | pi1=0.8 | mu1=-2 | s1=1 | mu2=1 | s2=0.5 |
|---|---|---|---|---|---|
| **50** | 0.882622 | -2.029409 | 0.958130 | 0.934326 | 0.421721 |
| **100** | 0.826651 | -2.005126 | 0.984224 | 0.983900 | 0.475902 |
| **300** | 0.779679 | -2.000381 | 0.995307 | 0.998152 | 0.494394 |
| **500** | 0.817710 | -2.000018 | 0.997165 | 0.998010 | 0.495933 |
| **1000** | 0.799799 | -2.000190 | 0.998564 | 0.999147 | 0.498290 |
| **5000** | 0.797402 | -1.999801 | 1.000031 | 0.999760 | 0.499570 |

**Bias for the estimated parameters at N = 10000**

| n | pi1=0.8 | mu1=-2 | s1=1 | mu2=1 | s2=0.5 |
|---|---|---|---|---|---|
| **50** | 0.082622 | -0.029409 | -0.041870 | -0.065674 | -0.078279 |
| **100** | 0.026651 | -0.005126 | -0.015776 | -0.016100 | -0.024098 |
| **300** | -0.020321 | -0.000381 | -0.004693 | -0.001848 | -0.005606 |
| **500** | 0.017710 | -0.000018 | -0.002835 | -0.001990 | -0.004067 |
| **1000** | -0.000201 | -0.000190 | -0.001436 | -0.000853 | -0.001710 |
| **5000** | -0.002598 | 0.000199 | 0.000031 | -0.000240 | -0.000430 |

**RMSE for the estimated parameters at N = 10000**

| n | pi1=0.8 | mu1=-2 | s1=1 | mu2=1 | s2=0.5 |
|---|---|---|---|---|---|
| **50** | 0.113471 | 0.201090 | 0.152067 | 0.449325 | 0.264423 |
| **100** | 0.044319 | 0.136662 | 0.107603 | 0.200066 | 0.138525 |
| **300** | 0.025142 | 0.077077 | 0.061709 | 0.082260 | 0.061472 |
| **500** | 0.020940 | 0.058719 | 0.045672 | 0.072023 | 0.052866 |
| **1000** | 0.007776 | 0.041494 | 0.032678 | 0.047391 | 0.034731 |
| **5000** | 0.004288 | 0.018658 | 0.014705 | 0.020816 | 0.015541 |

## 注意事項與討論：

**觀察**：

- 從表中可見，混合常態分佈參數的最大概似估計 MLE 在不同樣本數 n 下，其平均值皆相差不大，無論樣本數的大小，估計出來的參數值都很接近真實值。

- 但是，從偏差和 RMSE 的表中則可以明顯看出隨著樣本數的增加，其值皆會隨之下降，表示參數估計會受到樣本數的影響。

**結論**：

- 本次實驗表明，混合常態分佈的 MLE 參數估計確實會受樣本數影響。在小樣本下，參數估計結果較不穩定，偏差和 RMSE 較高；而在樣本數增加後，估計的穩定性和準確性均有顯著提升，進一步驗證了前一目標的結論。

- 此外，本次實驗顯示，無論樣本數大小，各參數的平均值皆接近母體真實值。這是因為組成混合常態的兩個分佈參數差異較大，即使在小樣本下，只要模擬次數足夠，其參數估計仍能準確區分這兩個分佈，因此平均值在小樣本下也得以接近真實值。

- 整體而言，MLE 在混合常態分佈的參數估計中表現穩定且準確，而樣本數的增加則是降低偏差與 RMSE 的關鍵。因此，進行參數估計時，建議採用較大的樣本數以提升準確性。

---

**Step 2**：與前面的實驗步驟相同，但其中分佈參數改為 $\pi_1 = 0.3, \mu_1 = -0.5, \sigma_1 = 1, \mu_2 = 1.5, \sigma_2 = 2$。

In [3]:

```
from scipy.stats import norm, binom
from scipy.optimize import minimize
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd

# 設定參數
N = 10000  # number of simulations
n = [50, 100, 300, 500, 1000, 5000]
pi1 = 0.3
mu1, s1 = -0.5, 1
mu2, s2 = 1.5, 2

# 設定演算法初始設定
params0 = [0.3, -0.5, 1, 1.5, 2]
bnd = [(0, 1), (None, None), (None, None), (None, None), (None, None)]
opts = {'disp': False, 'maxiter': 8000}

mean_RESULT = np.zeros((len(n), len(params0)))
bias_RESULT = np.zeros((len(n), len(params0)))
rmse_RESULT = np.zeros((len(n), len(params0)))

# 添加数值稳定性处理
def safe_norm_pdf(x, loc, scale):
    scale = np.clip(scale, 1e-10, None)  # 防止 scale 为零或接近于零
    return norm.pdf(x, loc, scale)

for j in range(len(n)):
    RESULT = np.zeros((N, 5))
    # 生成來自 normal mixture 的資料
    n1 = binom.rvs(n[j], pi1)
    SAMPLE = np.r_[norm.rvs(mu1, s1, size = (n1, N)), \
                   norm.rvs(mu2, s2, size = (n[j] - n1, N))]
    for i in range(N):
        X = SAMPLE[:, i]

        # neg_log_likelihood = lambda x: -np.sum(np.log(x[0] * norm.pdf(X, x[1], x[
                                    # (1 - x[0]) * norm.pdf(X, x[3], x[4])))
        neg_log_likelihood = lambda x: -np.sum(np.log(np.clip(x[0] * safe_norm_pdf(
                                    (1 - x[0]) * safe_norm_pdf(X, x[3], x[4]), 1e-1

        result = minimize(neg_log_likelihood, params0, method = 'Nelder-Mead', \
                        bounds = bnd, options = opts)
        RESULT[i, :] = result.x

    # Evaluation
    mean_RESULT[j, :] = np.mean(RESULT, axis = 0)
    bias_RESULT[j, :] = mean_RESULT[j, :] - params0
    rmse_RESULT[j, :] = np.sqrt(np.mean((RESULT - params0) ** 2, axis = 0))
```

In [4]:

```
# 整理表格
columns = ['pi1={}'.format(pi1), 'mu1={}'.format(mu1), 's1={}'.format(s1)
           , 'mu2={}'.format(mu2), 's2={}'.format(s2)]

columns_mean = pd.MultiIndex.from_product([['Mean for the estimated parameters at N
                                            pd.Index(columns, name = 'n')])
columns_bias = pd.MultiIndex.from_product([['Bias for the estimated parameters at N
                                            pd.Index(columns, name = 'n')])
columns_rmse = pd.MultiIndex.from_product([['RMSE for the estimated parameters at N
                                            pd.Index(columns, name = 'n')])

df_mean = pd.DataFrame(mean_RESULT, index = n, columns = columns_mean)
df_bias = pd.DataFrame(bias_RESULT, index = n, columns = columns_bias)
df_rmse = pd.DataFrame(rmse_RESULT, index = n, columns = columns_rmse)

# 使用 Styler 对象来设置列名和单元格内容的对齐方式
df_mean1 = df_mean.style.set_table_styles({
    ('Mean for the estimated parameters at N = {}'.format(N), 'pi1={}'.format(pi1))
    ('Mean for the estimated parameters at N = {}'.format(N), 'mu1={}'.format(mu1))
    ('Mean for the estimated parameters at N = {}'.format(N), 's1={}'.format(s1)):
    ('Mean for the estimated parameters at N = {}'.format(N), 'mu2={}'.format(mu2))
    ('Mean for the estimated parameters at N = {}'.format(N), 's2={}'.format(s2)):
}, overwrite=False).set_properties(**{'text-align': 'center'})

df_bias1 = df_bias.style.set_table_styles({
    ('Bias for the estimated parameters at N = {}'.format(N), 'pi1={}'.format(pi1))
    ('Bias for the estimated parameters at N = {}'.format(N), 'mu1={}'.format(mu1))
    ('Bias for the estimated parameters at N = {}'.format(N), 's1={}'.format(s1)):
    ('Bias for the estimated parameters at N = {}'.format(N), 'mu2={}'.format(mu2))
    ('Bias for the estimated parameters at N = {}'.format(N), 's2={}'.format(s2)):
}, overwrite=False).set_properties(**{'text-align': 'center'})

df_rmse1 = df_rmse.style.set_table_styles({
    ('RMSE for the estimated parameters at N = {}'.format(N), 'pi1={}'.format(pi1))
    ('RMSE for the estimated parameters at N = {}'.format(N), 'mu1={}'.format(mu1))
    ('RMSE for the estimated parameters at N = {}'.format(N), 's1={}'.format(s1)):
    ('RMSE for the estimated parameters at N = {}'.format(N), 'mu2={}'.format(mu2))
    ('RMSE for the estimated parameters at N = {}'.format(N), 's2={}'.format(s2)):
}, overwrite=False).set_properties(**{'text-align': 'center'})

# 添加水平線在每個 row
df_mean1.set_table_styles([{'selector': 'tr', 'props': [('border-top', '1px solid #
df_bias1.set_table_styles([{'selector': 'tr', 'props': [('border-top', '1px solid #
df_rmse1.set_table_styles([{'selector': 'tr', 'props': [('border-top', '1px solid #
df_mean1.set_table_styles([{'selector': 'tr', 'props': [('border-bottom', '1px soli
df_bias1.set_table_styles([{'selector': 'tr', 'props': [('border-bottom', '1px soli
df_rmse1.set_table_styles([{'selector': 'tr', 'props': [('border-bottom', '1px soli
```

```python
# 显示 DataFrame
display(df_mean1)
display(df_bias1)
display(df_rmse1)
```

| | Mean for the estimated parameters at N = 10000 | | | | |
|---|---|---|---|---|---|
| **n** | **pi1=0.3** | **mu1=-0.5** | **s1=1** | **mu2=1.5** | **s2=2** |
| **50** | 0.411844 | -0.440901 | 0.983009 | 2.544887 | 1.491840 |
| **100** | 0.460144 | -0.368204 | 0.983868 | 2.256263 | 1.681016 |
| **300** | 0.407524 | -0.414070 | 1.004490 | 1.880262 | 1.857332 |
| **500** | 0.372859 | -0.438794 | 1.008044 | 1.760601 | 1.903367 |
| **1000** | 0.351742 | -0.468482 | 1.010547 | 1.640061 | 1.950991 |
| **5000** | 0.316450 | -0.496656 | 1.001760 | 1.516145 | 1.994743 |

| | Bias for the estimated parameters at N = 10000 | | | | |
|---|---|---|---|---|---|
| **n** | **pi1=0.3** | **mu1=-0.5** | **s1=1** | **mu2=1.5** | **s2=2** |
| **50** | 0.111844 | 0.059099 | -0.016991 | 1.044887 | -0.508160 |
| **100** | 0.160144 | 0.131796 | -0.016132 | 0.756263 | -0.318984 |
| **300** | 0.107524 | 0.085930 | 0.004490 | 0.380262 | -0.142668 |
| **500** | 0.072859 | 0.061206 | 0.008044 | 0.260601 | -0.096633 |
| **1000** | 0.051742 | 0.031518 | 0.010547 | 0.140061 | -0.049009 |
| **5000** | 0.016450 | 0.003344 | 0.001760 | 0.016145 | -0.005257 |

| | RMSE for the estimated parameters at N = 10000 | | | | |
|---|---|---|---|---|---|
| **n** | **pi1=0.3** | **mu1=-0.5** | **s1=1** | **mu2=1.5** | **s2=2** |
| **50** | 0.318928 | 1.057956 | 0.575525 | 1.749577 | 0.791833 |
| **100** | 0.297480 | 0.592609 | 0.409914 | 1.414929 | 0.594496 |
| **300** | 0.217092 | 0.326768 | 0.275157 | 0.839512 | 0.322494 |
| **500** | 0.173577 | 0.253111 | 0.226887 | 0.635822 | 0.237443 |
| **1000** | 0.122093 | 0.161707 | 0.154560 | 0.407277 | 0.148472 |
| **5000** | 0.040625 | 0.062419 | 0.064067 | 0.110417 | 0.039375 |

## 注意事項與討論：

**觀察**：

- 從表中可見，與 Step 1 中的結果相比，本次實驗的參數估計整體表現較差，此次可見在小樣本情況下，偏差和 RMSE 明顯高於 Step 1。
- 當樣本數為 n = 50 和 n = 100 時，參數的偏差明顯較大，平均值明顯偏離母體真實值，反映出估計的穩定性較差。這是因為兩個常態分佈的重疊區域較廣，導致混合常態分佈的特徵難以被準確捕捉。

- 隨著樣本數增加至 n = 300、500 和 1000，估計結果的穩定性逐步提升，偏差和 RMSE 明顯逐漸下降，但整體平均值仍與真實值存在一定差異，表明高重疊分佈下的估計難度仍然存在。

- 當樣本數達到 n = 5000 時，參數估計的平均值接近母體真實值，偏差和 RMSE 明顯下降，但由於分佈的雙峰結構仍然不夠明顯，其偏差和 RMSE 仍不如 Step 1 來得低。

**結論**：

- 與 Step 1 的結果相比，本次實驗中組成混合常態分佈的兩個組成分佈參數差異較小，導致重疊程度較高，為 MLE 的參數估計增加了難度。在小樣本情況下，估計值的偏差和 RMSE 顯著增加，反映出高度重疊分佈下參數估計的穩定性較低。

- 隨著樣本數的增加，MLE 的估計性能逐步提升，但整體的估計準確性仍受分佈特徵影響較大。僅在樣本數達到 n = 5000 時，估計值的平均值才接近母體真實值，顯示出高度重疊分佈需要更大的樣本數以提升估計準確性。

- 本次實驗進一步證實，當混合常態分佈的組成分佈參數差異較小時，樣本數對於降低偏差和 RMSE 的重要性尤為重要。因此，在使用 MLE 進行參數估計時，需特別注意分佈參數的特徵，並確保樣本數充足，以提升模型的估計穩定性與準確性。

---

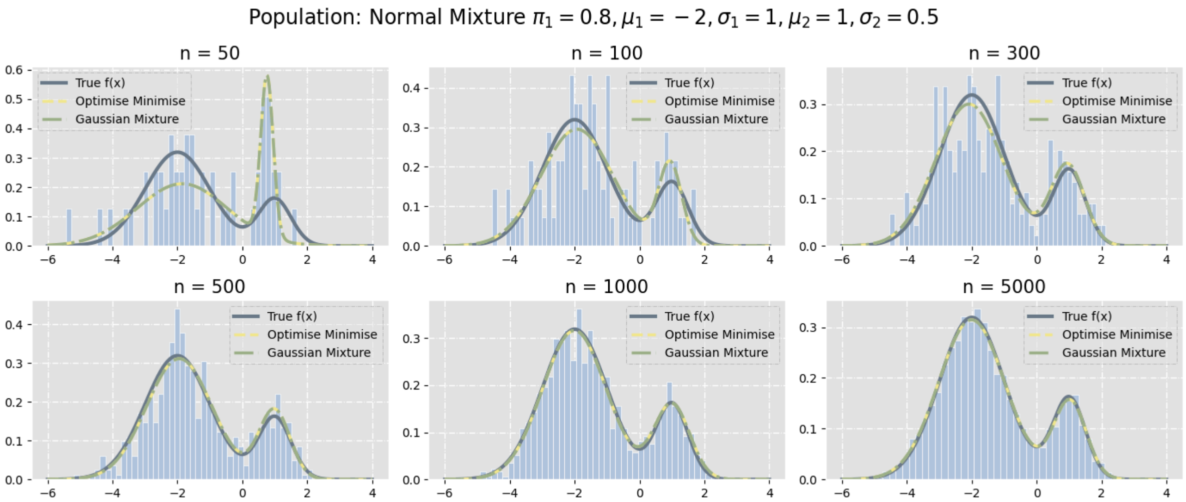
## 目標三：

採用 `sklearn.mixture.GaussianMixture` （後續簡稱 **GMM**）方法對資料進行參數估計，並重複前面兩個目標的模擬實驗步驟，比較該方法與 **MLE** 在參數估計結果上的表現差異。

**Step 1**：與目標一的 step 2 實驗步驟相同，生成 n 個來自混合常態的隨機樣本，其中分佈參數為 $\pi_1 = 0.8, \mu_1 = -2, \sigma_1 = 1, \mu_2 = 1, \sigma_2 = 0.5$，樣本數為 n = 50，100，300，500，1000，5000。針對每個樣本數 n 畫出其分佈直方圖，並利用 MLE 估計其參數值，畫出其樣本分佈圖，以及真實母體分佈圖。此外，利用 GMM 估計出參數值，並一樣畫出其分佈圖，觀察 MLE 與 GMM 在參數估計上的差異。

In [28]:
```python
from scipy.stats import norm, binom
from scipy.optimize import minimize
import numpy as np
import matplotlib.pyplot as plt
from sklearn.mixture import GaussianMixture

# 設定參數
n = [50, 100, 300, 500, 1000, 5000]
pi1 = 0.8
mu1, s1 = -2, 1
mu2, s2 = 1, 0.5

# 理論混合常態分配 pdf
f = lambda x: pi1 * norm.pdf(x, mu1, s1) + (1 - pi1) * norm.pdf(x, mu2, s2)
x = np.linspace(-6, 4, 1000)
y = f(x)

# 設定演算法初始設定
params0 = [0.8, -2, 1, 1, 0.5]
bnd = [(0, 1), (None, None), (None, None), (None, None), (None, None)]
opts = {'disp': False, 'maxiter': 8000}

plt.style.use('ggplot')
```

```python
fig, axs = plt.subplots(2, 3, figsize = (14, 6))

for j in range(len(n)):
    # 生成樣本
    n1 = binom.rvs(n[j], pi1)
    X = np.r_[norm.rvs(mu1, s1, size = n1), norm.rvs(mu2, s2, size = n[j] - n1)]

    # 定義 mixed normal 對數最大概似函數
    neg_log_likelihood = lambda x: -np.sum(np.log(x[0] * norm.pdf(X, x[1], x[2]) +
                                            (1 - x[0]) * norm.pdf(X, x[3], x[4])))

    # 計算最大概似估計
    result = minimize(neg_log_likelihood, params0, method = 'Nelder-Mead'
                    , bounds = bnd, options = opts)

    # Gaussian Mixture 演算法
    gmm = GaussianMixture(n_components = 2, covariance_type = 'full', \
            verbose = 0, max_iter = 1000, tol = 1e-6)
    gmm.fit(X.reshape(-1, 1)) # sample is the normal mixture data as a column vecto

    ax = axs.ravel()
    ax[j].plot(x, y, color = '#6c7b8b', label = 'True f(x)', linewidth = 3)
    ax[j].plot(x, result.x[0] * norm.pdf(x, result.x[1], result.x[2]) + \
            (1 - result.x[0]) * norm.pdf(x, result.x[3], result.x[4]), label = 'Opt
            , color = '#f0e68c', linestyle = '--', linewidth = 2.5)
    ax[j].plot(x, gmm.weights_[0] * norm.pdf(x, gmm.means_[0][0], np.sqrt(gmm.covar
             gmm.weights_[1] * norm.pdf(x, gmm.means_[1][0], np.sqrt(gmm.covariance
                , label = 'Gaussian Mixture', color = '#9caf88', linestyle = '-.',
    ax[j].hist(X, bins = 45, density = True, color = '#b0c4de', edgecolor = 'white'
    ax[j].set_title('n = {}'.format(n[j]), fontsize = 15)
    ax[j].grid(True, linestyle = '-.')
    ax[j].tick_params(axis = 'both', labelsize = 10, colors = 'black')
    lgd = ax[j].legend(edgecolor = '#666666', prop = {'size': 10})
    lgd.get_frame().set_linestyle('-.')
    lgd.get_frame().set_alpha(0.4)

fig.suptitle('Population: Normal Mixture $\pi_1 = {}, \mu_1 = {}, \sigma_1 = {}, \m
                .format(pi1, mu1, s1, mu2, s2), fontsize = 17)
plt.tight_layout()
plt.show()
```



## 注意事項與討論：

### 觀察：

- 從圖中可以看出，無論樣本數的大小，MLE 和 GMM 的估計曲線均顯示出高度重疊的情況。
- 此外，兩者在樣本數較少時，估計曲線與真實分佈曲線之間存在明顯差異，但隨著樣本數的增加，估計曲線逐漸趨近真實分佈曲線。

**結論**：

- 根據實驗結果，未能顯著區分 MLE 和 GMM 之間的差異，這可能是因為混合常態分佈的組成參數差異較大，從而使得兩者在區分分佈時呈現相似的結果。
- 由此可見，GMM 在參數估計方面同樣表現出色，尤其是在分佈參數差異較大且樣本數較多的情況下，兩種方法均能準確地還原母體分佈的特徵，包括雙峰結構和分佈形狀。

---

**Step 2**：生成 n 個來自混合常態的隨機樣本，其中分佈參數為 $\pi_1 = 0.8, \mu_1 = -2, \sigma_1 = 1, \mu_2 = 1, \sigma_2 = 0.5$，樣本數為 n = 50，100，300，500，1000，5000。針對每個樣本數 n 進行 10000 次的蒙地卡羅模擬實驗，並分別使用 MLE 和 GMM 去估計參數，計算各自參數估計值的平均值（Mean）、偏差（Bias）以及均方根誤差（RMSE），以此評估 MLE 和 GMM 在參數估計上的表現。

In [ ]:

```python
from scipy.stats import norm, binom
from scipy.optimize import minimize
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
from sklearn.mixture import GaussianMixture

# 設定參數
N = 10000  # number of simulations
n = [50, 100, 300, 500, 1000, 5000]
pi1 = 0.8
mu1, s1 = -2, 1
mu2, s2 = 1, 0.5

# 設定演算法初始設定
params0 = [0.8, -2, 1, 1, 0.5]
bnd = [(0, 1), (None, None), (None, None), (None, None), (None, None)]
opts = {'disp': False, 'maxiter': 8000}

mean_mini = np.zeros((len(n), len(params0))); mean_gmm = np.zeros((len(n), len(para
bias_mini = np.zeros((len(n), len(params0))); bias_gmm = np.zeros((len(n), len(para
rmse_mini = np.zeros((len(n), len(params0))); rmse_gmm = np.zeros((len(n), len(para

for j in range(len(n)):
    # 生成結果矩陣
    RESULT_mini = np.zeros((N, 5))
    RESULT_gmm = np.zeros((N, 5))
    # 生成來自 mix beta distribution 的資料
    n1 = binom.rvs(n[j], pi1)
    SAMPLE = np.r_[norm.rvs(mu1, s1, size = (n1, N)), \
                   norm.rvs(mu2, s2, size = (n[j] - n1, N))]

    for i in range(N):
        X = SAMPLE[:, i]

        neg_log_likelihood = lambda x: -np.sum(np.log(x[0] * norm.pdf(X, x[1], x[2]
                                 (1 - x[0]) * norm.pdf(X, x[3], x[4])))
        result = minimize(neg_log_likelihood, params0, method = 'Nelder-Mead', \
                        bounds = bnd, options = opts)
        RESULT_mini[i, :] = result.x

        # Gaussian Mixture 演算法
        gmm = GaussianMixture(n_components = 2, covariance_type = 'full', \
                              weights_init = [pi1, 1-pi1], \
                              means_init = np.array([[mu1], [mu2]]).reshape(2, 1),
                              verbose = 0, max_iter = 8000, tol = 1e-6)
        gmm.fit(X.reshape(-1, 1))
        a, b = np.argmax(gmm.weights_), np.argmin(gmm.weights_)
        RESULT_gmm[i, :] = [gmm.weights_[a], gmm.means_[0][0], np.sqrt(gmm.covarian
                             gmm.means_[1][0], np.sqrt(gmm.covariances_[1][0][0])]

    # minimize
    mean_mini[j, :] = np.mean(RESULT_mini, axis = 0)
    bias_mini[j, :] = mean_mini[j, :] - params0
    rmse_mini[j, :] = np.sqrt(np.mean((RESULT_mini - params0) ** 2, axis = 0))

    # gmm
    mean_gmm[j, :] = np.mean(RESULT_gmm, axis = 0)
    bias_gmm[j, :] = mean_gmm[j, :] - params0
    rmse_gmm[j, :] = np.sqrt(np.mean((RESULT_gmm - params0) ** 2, axis = 0))
```

In [6]:

```
# 整理表格
columns = ['pi1={}'.format(pi1), 'mu1={}'.format(mu1), 's1={}'.format(s1)
           , 'mu2={}'.format(mu2), 's2={}'.format(s2)]

# 創建 MultiIndex 列名
columns_mean = pd.MultiIndex.from_product([['Mean for the estimated parameters at N
columns_bias = pd.MultiIndex.from_product([['Bias for the estimated parameters at N
columns_rmse = pd.MultiIndex.from_product([['RMSE for the estimated parameters at N

# 创建新的 MultiIndex
methods = np.repeat(['optimize.minimize', 'GaussianMixture'], len(n))
ns = np.tile(n, 2)
multi_index = pd.MultiIndex.from_arrays([methods, ns])

# 创建 DataFrame 并设置 MultiIndex
df_mean = pd.DataFrame(np.vstack([mean_mini, mean_gmm]), index=multi_index, columns
df_bias = pd.DataFrame(np.vstack([bias_mini, bias_gmm]), index=multi_index, columns
df_rmse = pd.DataFrame(np.vstack([rmse_mini, rmse_gmm]), index=multi_index, columns

# 使用 Styler 对象来设置列名和单元格内容的对齐方式
df_mean1 = df_mean.style.set_table_styles({
    ('Mean for the estimated parameters at N = {}'.format(N), 'pi1={}'.format(pi1))
    ('Mean for the estimated parameters at N = {}'.format(N), 'mu1={}'.format(mu1))
    ('Mean for the estimated parameters at N = {}'.format(N), 's1={}'.format(s1)):
    ('Mean for the estimated parameters at N = {}'.format(N), 'mu2={}'.format(mu2))
    ('Mean for the estimated parameters at N = {}'.format(N), 's2={}'.format(s2)):
}, overwrite=False).set_properties(**{'text-align': 'center'})

df_bias1 = df_bias.style.set_table_styles({
    ('Bias for the estimated parameters at N = {}'.format(N), 'pi1={}'.format(pi1))
    ('Bias for the estimated parameters at N = {}'.format(N), 'mu1={}'.format(mu1))
    ('Bias for the estimated parameters at N = {}'.format(N), 's1={}'.format(s1)):
    ('Bias for the estimated parameters at N = {}'.format(N), 'mu2={}'.format(mu2))
    ('Bias for the estimated parameters at N = {}'.format(N), 's2={}'.format(s2)):
}, overwrite=False).set_properties(**{'text-align': 'center'})

df_rmse1 = df_rmse.style.set_table_styles({
    ('RMSE for the estimated parameters at N = {}'.format(N), 'pi1={}'.format(pi1))
    ('RMSE for the estimated parameters at N = {}'.format(N), 'mu1={}'.format(mu1))
    ('RMSE for the estimated parameters at N = {}'.format(N), 's1={}'.format(s1)):
```

```
    ('RMSE for the estimated parameters at N = {}'.format(N), 'mu2={}'.format(mu2))
    ('RMSE for the estimated parameters at N = {}'.format(N), 's2={}'.format(s2)):
}, overwrite=False).set_properties(**{'text-align': 'center'})

# 添加水平線在每個 row
df_mean1.set_table_styles([{'selector': 'tr', 'props': [('border-top', '1px solid #
df_bias1.set_table_styles([{'selector': 'tr', 'props': [('border-top', '1px solid #
df_rmse1.set_table_styles([{'selector': 'tr', 'props': [('border-top', '1px solid #
df_mean1.set_table_styles([{'selector': 'tr', 'props': [('border-bottom', '1px soli
df_bias1.set_table_styles([{'selector': 'tr', 'props': [('border-bottom', '1px soli
df_rmse1.set_table_styles([{'selector': 'tr', 'props': [('border-bottom', '1px soli

# 显示 DataFrame
display(df_mean1)
display(df_bias1)
display(df_rmse1)
```

| | | **Mean for the estimated parameters at N = 10000** | | | | |
|---|---|---|---|---|---|---|
| | **n** | **pi1=0.8** | **mu1=-2** | **s1=1** | **mu2=1** | **s2=0.5** |
| **optimize.minimize** | **50** | 0.640450 | -1.998102 | 0.974622 | 0.994598 | 0.475778 |
| | **100** | 0.778291 | -2.003073 | 0.987354 | 0.993320 | 0.483653 |
| | **300** | 0.816026 | -2.000869 | 0.996373 | 0.997135 | 0.492551 |
| | **500** | 0.789798 | -2.000498 | 0.997708 | 0.998412 | 0.496854 |
| | **1000** | 0.795955 | -2.000182 | 0.998900 | 0.999874 | 0.497616 |
| | **5000** | 0.799132 | -2.000271 | 0.999408 | 0.999694 | 0.499903 |
| **GaussianMixture** | **50** | 0.640690 | -2.000823 | 0.972168 | 0.991966 | 0.478246 |
| | **100** | 0.777815 | -2.004435 | 0.986237 | 0.991507 | 0.485169 |
| | **300** | 0.815715 | -2.001692 | 0.995645 | 0.995764 | 0.493547 |
| | **500** | 0.789512 | -2.001283 | 0.997007 | 0.997306 | 0.497671 |
| | **1000** | 0.795679 | -2.000938 | 0.998217 | 0.998777 | 0.498426 |
| | **5000** | 0.798866 | -2.000999 | 0.998746 | 0.998623 | 0.500694 |

| | | Bias for the estimated parameters at N = 10000 | | | | |
|---|---|---|---|---|---|---|
| | **n** | **pi1=0.8** | **mu1=-2** | **s1=1** | **mu2=1** | **s2=0.5** |
| **optimize.minimize** | **50** | -0.159550 | 0.001898 | -0.025378 | -0.005402 | -0.024222 |
| | **100** | -0.021709 | -0.003073 | -0.012646 | -0.006680 | -0.016347 |
| | **300** | 0.016026 | -0.000869 | -0.003627 | -0.002865 | -0.007449 |
| | **500** | -0.010202 | -0.000498 | -0.002292 | -0.001588 | -0.003146 |
| | **1000** | -0.004045 | -0.000182 | -0.001100 | -0.000126 | -0.002384 |
| | **5000** | -0.000868 | -0.000271 | -0.000592 | -0.000306 | -0.000097 |
| **GaussianMixture** | **50** | -0.159310 | -0.000823 | -0.027832 | -0.008034 | -0.021754 |
| | **100** | -0.022185 | -0.004435 | -0.013763 | -0.008493 | -0.014831 |
| | **300** | 0.015715 | -0.001692 | -0.004355 | -0.004236 | -0.006453 |
| | **500** | -0.010488 | -0.001283 | -0.002993 | -0.002694 | -0.002329 |
| | **1000** | -0.004321 | -0.000938 | -0.001783 | -0.001223 | -0.001574 |
| | **5000** | -0.001134 | -0.000999 | -0.001254 | -0.001377 | 0.000694 |

| | | RMSE for the estimated parameters at N = 10000 | | | | |
|---|---|---|---|---|---|---|
| | **n** | **pi1=0.8** | **mu1=-2** | **s1=1** | **mu2=1** | **s2=0.5** |
| **optimize.minimize** | **50** | 0.165830 | 0.231012 | 0.175975 | 0.161613 | 0.124185 |
| | **100** | 0.038599 | 0.140463 | 0.110936 | 0.160685 | 0.116602 |
| | **300** | 0.021922 | 0.075819 | 0.059802 | 0.093935 | 0.069560 |
| | **500** | 0.015067 | 0.058759 | 0.046746 | 0.065051 | 0.048221 |
| | **1000** | 0.008719 | 0.041211 | 0.032814 | 0.046618 | 0.034725 |
| | **5000** | 0.003503 | 0.018600 | 0.014689 | 0.020834 | 0.015593 |
| **GaussianMixture** | **50** | 0.164688 | 0.229943 | 0.175701 | 0.165571 | 0.127425 |
| | **100** | 0.038325 | 0.139359 | 0.110218 | 0.159958 | 0.116063 |
| | **300** | 0.021691 | 0.075696 | 0.059690 | 0.094200 | 0.069778 |
| | **500** | 0.015272 | 0.058683 | 0.046675 | 0.065282 | 0.048401 |
| | **1000** | 0.008863 | 0.041167 | 0.032776 | 0.046793 | 0.034854 |
| | **5000** | 0.003583 | 0.018600 | 0.014699 | 0.020939 | 0.015685 |

## 注意事項與討論：

**觀察**：

- 從結果中可以看到，與 Step 1 的情況相似，無論樣本數的大小，MLE 和 GMM 在參數估計上的表現幾乎沒有顯著差異。
- 此外，兩者在樣本數較少時，估計的平均值與真實值之間仍然存在差異，但隨著樣本數的增加，平均值會逐漸接近真實值，偏差和 RMSE 也隨之降低。

**結論**：

- 實驗結果與 Step 1 類似，未能明顯顯示 MLE 和 GMM 之間的差異。原因與 Step 1 相同，兩個常態分佈的參數差異較大，因此能夠較容易區分，無法顯現兩者在處理分佈細節上的不同。

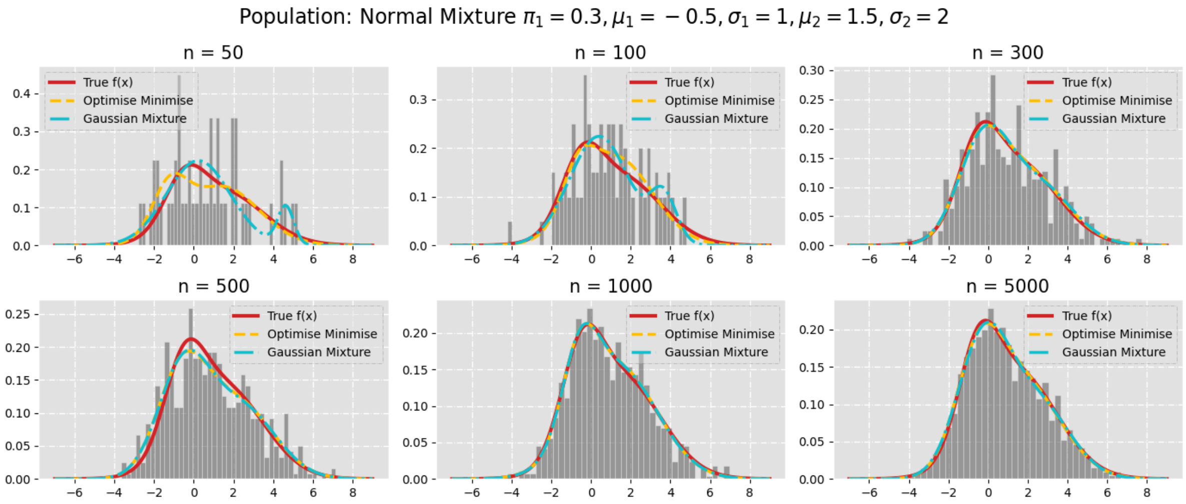
- 兩者在樣本數增大後，都展現出估計準確性的提高，並且在大樣本下，其估計結果與真實母體參數值相當接近。因此，GMM 同樣是一個可靠的參數估計方法。

---

**Step 3**：與 Step 1 的實驗步驟相同，但其中分佈參數改為 $\pi_1 = 0.3, \mu_1 = -0.5, \sigma_1 = 1, \mu_2 = 1.5, \sigma_2 = 2$，觀察 MLE 與 GMM 在參數估計上的差異。

In [ ]:

```python
from scipy.stats import norm, binom
from scipy.optimize import minimize
import numpy as np
import matplotlib.pyplot as plt
from sklearn.mixture import GaussianMixture
import os
os.environ["OMP_NUM_THREADS"] = "1"

# 設定參數
n = [50, 100, 300, 500, 1000, 5000]
pi1 = 0.3
mu1, s1 = -0.5, 1
mu2, s2 = 1.5, 2

# 理論混合常態分配 pdf
f = lambda x: pi1 * norm.pdf(x, mu1, s1) + (1 - pi1) * norm.pdf(x, mu2, s2)
x = np.linspace(-7, 9, 1000)
y = f(x)

# 設定演算法初始設定
params0 = [0.3, -0.5, 1, 1.5, 2]
bnd = [(0, 1), (None, None), (None, None), (None, None), (None, None)]
opts = {'disp': False, 'maxiter': 8000}

plt.style.use('ggplot')
fig, axs = plt.subplots(2, 3, figsize = (14, 6))

for j in range(len(n)):
    # 生成樣本
    n1 = binom.rvs(n[j], pi1)
    X = np.r_[norm.rvs(mu1, s1, size = n1), norm.rvs(mu2, s2, size = n[j] - n1)]

    # 定義 mixed normal 對數最大概似函數
    neg_log_likelihood = lambda x: -np.sum(np.log(x[0] * norm.pdf(X, x[1], x[2]) +
                                          (1 - x[0]) * norm.pdf(X, x[3], x[4])))

    # 計算最大概似估計
    result = minimize(neg_log_likelihood, params0, method = 'Nelder-Mead'
                    , bounds = bnd, options = opts)

    # Gaussian Mixture 演算法
    gmm = GaussianMixture(n_components = 2, covariance_type = 'full', \
            verbose = 0, max_iter = 1000, tol = 1e-6)
    gmm.fit(X.reshape(-1, 1)) # sample is the normal mixture data as a column vecto

    ax = axs.ravel()
    ax[j].plot(x, y, color = '#d62728', label = 'True f(x)', linewidth = 3)
    ax[j].plot(x, result.x[0] * norm.pdf(x, result.x[1], result.x[2]) + \
            (1 - result.x[0]) * norm.pdf(x, result.x[3], result.x[4]), label = 'Opt
            , color = '#ffbf00', linestyle = '--', linewidth = 2.5)
    ax[j].plot(x, gmm.weights_[0] * norm.pdf(x, gmm.means_[0][0], np.sqrt(gmm.covar
             gmm.weights_[1] * norm.pdf(x, gmm.means_[1][0], np.sqrt(gmm.covariance
                , label = 'Gaussian Mixture', color = '#17becf', linestyle = '-.',
    ax[j].hist(X, bins = 45, density = True, color = '#4d4d4d', edgecolor = 'white'
    ax[j].set_title('n = {}'.format(n[j]), fontsize = 15)
    ax[j].grid(True, linestyle = '-.')
    ax[j].tick_params(axis = 'both', labelsize = 10, colors = 'black')
    lgd = ax[j].legend(edgecolor = '#666666', prop = {'size': 10})
    lgd.get_frame().set_linestyle('-.')
    lgd.get_frame().set_alpha(0.4)

fig.suptitle('Population: Normal Mixture $\pi_1 = {}, \mu_1 = {}, \sigma_1 = {}, \m
                .format(pi1, mu1, s1, mu2, s2), fontsize = 17)
plt.tight_layout()
plt.show()
```



## 注意事項與討論：

### 觀察：

- 在樣本數 n = 50 和 n = 100 時，MLE 和 GMM 的估計結果均存在偏差，且兩者之間的趨勢也不一樣。相較之下，GMM 的估計曲線較明顯可看出雙峰形狀，但兩者對於分佈的高峰與尾部的擬合均不理想。

- 當樣本數增加至 n = 300 和 n = 500 時，MLE 和 GMM 的估計曲線逐步接近真實分佈。此時，MLE 和 GMM 的估計曲線已幾乎重疊在一起。

- 當樣本數增至 n = 1000 和 n = 5000 時，MLE 和 GMM 的估計曲線與真實分佈曲線十分接近，顯示在大樣本的情況下，兩種方法的估計性能差異趨近於零。

### 結論：

- 在小樣本條件下，GMM 相較於 MLE，展現出更穩定的估計性能，其分佈曲線能更精準地捕捉混合常態分佈的主要特徵，而 MLE 的估計結果則受樣本波動影響較大，容易導致較大的偏差。

- 隨著樣本數增加，MLE 的估計性能逐漸提升，與 GMM 的結果趨於一致，兩者在大樣本的情況下均能準確地反映混合常態分佈的結構特徵，顯示出相似的估計能力和趨勢。

- 本實驗顯示，在混合常態分佈中組成分佈參數的差異較小時，雖然 MLE 和 GMM 在大樣本下均能提供相對準確的估計，但在樣本數不足的情況下，GMM 更具穩定性和準確性，能更有效地捕捉混合常態分佈的主要特徵，因此特別適用於小樣本或資料特徵不明顯的情境。

---

**Step 4**：與 Step 2 的實驗步驟相同，但其中分佈參數改為 $\pi_1 = 0.3, \mu_1 = -0.5, \sigma_1 = 1, \mu_2 = 1.5, \sigma_2 = 2$，以此評估 MLE 和 GMM 在參數估計上的表現。

In [ ]:

```
from scipy.stats import norm, binom
from scipy.optimize import minimize
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
from sklearn.mixture import GaussianMixture

# 設定參數
N = 10000  # number of simulations
n = [50, 100, 300, 500, 1000, 5000]
pi1 = 0.3
mu1, s1 = -0.5, 1
mu2, s2 = 1.5, 2

# 設定演算法初始設定
params0 = [0.3, -0.5, 1, 1.5, 2]
bnd = [(0, 1), (None, None), (None, None), (None, None), (None, None)]
opts = {'disp': False, 'maxiter': 8000}

mean_mini = np.zeros((len(n), len(params0))); mean_gmm = np.zeros((len(n), len(para
bias_mini = np.zeros((len(n), len(params0))); bias_gmm = np.zeros((len(n), len(para
rmse_mini = np.zeros((len(n), len(params0))); rmse_gmm = np.zeros((len(n), len(para

for j in range(len(n)):
    # 生成結果矩陣
    RESULT_mini = np.zeros((N, 5))
    RESULT_gmm = np.zeros((N, 5))
    # 生成來自 mix beta distribution 的資料
    n1 = binom.rvs(n[j], pi1)
    SAMPLE = np.r_[norm.rvs(mu1, s1, size = (n1, N)), \
                norm.rvs(mu2, s2, size = (n[j] - n1, N))]

    for i in range(N):
        X = SAMPLE[:, i]

        neg_log_likelihood = lambda x: -np.sum(np.log(x[0] * norm.pdf(X, x[1], x[2]
                                    (1 - x[0]) * norm.pdf(X, x[3], x[4])))

        result = minimize(neg_log_likelihood, params0, method = 'Nelder-Mead', \
                        bounds = bnd, options = opts) # L-BFGS-B
        RESULT_mini[i, :] = result.x

        # Gaussian Mixture 演算法
        gmm = GaussianMixture(n_components = 2, covariance_type = 'full', \
                              weights_init = [pi1, 1-pi1], \
                              means_init = np.array([[mu1], [mu2]]).reshape(2, 1),
                              verbose = 0, max_iter = 8000, tol = 1e-6)
        gmm.fit(X.reshape(-1, 1))
        # a, b = np.argmax(gmm.weights_), np.argmin(gmm.weights_)
        RESULT_gmm[i, :] = [gmm.weights_[0], gmm.means_[0][0], np.sqrt(gmm.covarian
                             gmm.means_[1][0], np.sqrt(gmm.covariances_[1][0][0])]
    # minimize
    mean_mini[j, :] = np.mean(RESULT_mini, axis = 0)
    bias_mini[j, :] = mean_mini[j, :] - params0
    rmse_mini[j, :] = np.sqrt(np.mean((RESULT_mini - params0) ** 2, axis = 0))

    # gmm
    mean_gmm[j, :] = np.mean(RESULT_gmm, axis = 0)
    bias_gmm[j, :] = mean_gmm[j, :] - params0
    rmse_gmm[j, :] = np.sqrt(np.mean((RESULT_gmm - params0) ** 2, axis = 0))
```

In [8]:

```
# 整理表格
columns = ['pi1={}'.format(pi1), 'mu1={}'.format(mu1), 's1={}'.format(s1)
           , 'mu2={}'.format(mu2), 's2={}'.format(s2)]

# 创建 MultiIndex 列名
columns_mean = pd.MultiIndex.from_product([['Mean for the estimated parameters at N
columns_bias = pd.MultiIndex.from_product([['Bias for the estimated parameters at N
columns_rmse = pd.MultiIndex.from_product([['RMSE for the estimated parameters at N

# 创建新的 MultiIndex
methods = np.repeat(['optimize.minimize', 'GaussianMixture'], len(n))
ns = np.tile(n, 2)
multi_index = pd.MultiIndex.from_arrays([methods, ns])

# 创建 DataFrame 并设置 MultiIndex
df_mean = pd.DataFrame(np.vstack([mean_mini, mean_gmm]), index=multi_index, columns
df_bias = pd.DataFrame(np.vstack([bias_mini, bias_gmm]), index=multi_index, columns
df_rmse = pd.DataFrame(np.vstack([rmse_mini, rmse_gmm]), index=multi_index, columns

# 使用 Styler 对象来设置列名和单元格内容的对齐方式
df_mean1 = df_mean.style.set_table_styles({
    ('Mean for the estimated parameters at N = {}'.format(N), 'pi1={}'.format(pi1))
    ('Mean for the estimated parameters at N = {}'.format(N), 'mu1={}'.format(mu1))
    ('Mean for the estimated parameters at N = {}'.format(N), 's1={}'.format(s1)):
    ('Mean for the estimated parameters at N = {}'.format(N), 'mu2={}'.format(mu2))
    ('Mean for the estimated parameters at N = {}'.format(N), 's2={}'.format(s2)):
}, overwrite=False).set_properties(**{'text-align': 'center'})

df_bias1 = df_bias.style.set_table_styles({
    ('Bias for the estimated parameters at N = {}'.format(N), 'pi1={}'.format(pi1))
    ('Bias for the estimated parameters at N = {}'.format(N), 'mu1={}'.format(mu1))
    ('Bias for the estimated parameters at N = {}'.format(N), 's1={}'.format(s1)):
    ('Bias for the estimated parameters at N = {}'.format(N), 'mu2={}'.format(mu2))
    ('Bias for the estimated parameters at N = {}'.format(N), 's2={}'.format(s2)):
}, overwrite=False).set_properties(**{'text-align': 'center'})

df_rmse1 = df_rmse.style.set_table_styles({
    ('RMSE for the estimated parameters at N = {}'.format(N), 'pi1={}'.format(pi1))
    ('RMSE for the estimated parameters at N = {}'.format(N), 'mu1={}'.format(mu1))
    ('RMSE for the estimated parameters at N = {}'.format(N), 's1={}'.format(s1)):
    ('RMSE for the estimated parameters at N = {}'.format(N), 'mu2={}'.format(mu2))
    ('RMSE for the estimated parameters at N = {}'.format(N), 's2={}'.format(s2)):
}, overwrite=False).set_properties(**{'text-align': 'center'})

# 添加水平線在每個 row
df_mean1.set_table_styles([{'selector': 'tr', 'props': [('border-top', '1px solid #
df_bias1.set_table_styles([{'selector': 'tr', 'props': [('border-top', '1px solid #
df_rmse1.set_table_styles([{'selector': 'tr', 'props': [('border-top', '1px solid #
df_mean1.set_table_styles([{'selector': 'tr', 'props': [('border-bottom', '1px soli
df_bias1.set_table_styles([{'selector': 'tr', 'props': [('border-bottom', '1px soli
df_rmse1.set_table_styles([{'selector': 'tr', 'props': [('border-bottom', '1px soli

# 显示 DataFrame
```

```
display(df_mean1)
display(df_bias1)
display(df_rmse1)
```

|  |  | Mean for the estimated parameters at N = 10000 |  |  |  |  |
|---|---|---|---|---|---|---|
|  | **n** | **pi1=0.3** | **mu1=-0.5** | **s1=1** | **mu2=1.5** | **s2=2** |
| **optimize.minimize** | **50** | 0.446100 | -0.412465 | 0.975336 | 2.536241 | 1.484449 |
|  | **100** | 0.433034 | -0.333698 | 1.007337 | 2.355011 | 1.645619 |
|  | **300** | 0.378302 | -0.394459 | 1.006634 | 1.912721 | 1.850601 |
|  | **500** | 0.351241 | -0.428686 | 1.008896 | 1.761181 | 1.907780 |
|  | **1000** | 0.333224 | -0.463616 | 1.014430 | 1.653855 | 1.946495 |
|  | **5000** | 0.309999 | -0.496257 | 1.002192 | 1.517366 | 1.995069 |
| **GaussianMixture** | **50** | 0.477665 | -0.282196 | 1.089355 | 2.556630 | 1.449344 |
|  | **100** | 0.462771 | -0.258445 | 1.115915 | 2.374892 | 1.631468 |
|  | **300** | 0.387986 | -0.391991 | 1.055320 | 1.902553 | 1.851630 |
|  | **500** | 0.353534 | -0.432535 | 1.036292 | 1.745613 | 1.913320 |
|  | **1000** | 0.330153 | -0.470500 | 1.026102 | 1.626841 | 1.959095 |
|  | **5000** | 0.322381 | -0.483872 | 1.023862 | 1.547471 | 1.991205 |

|  |  | Bias for the estimated parameters at N = 10000 |  |  |  |  |
|---|---|---|---|---|---|---|
|  | **n** | **pi1=0.3** | **mu1=-0.5** | **s1=1** | **mu2=1.5** | **s2=2** |
| **optimize.minimize** | **50** | 0.146100 | 0.087535 | -0.024664 | 1.036241 | -0.515551 |
|  | **100** | 0.133034 | 0.166302 | 0.007337 | 0.855011 | -0.354381 |
|  | **300** | 0.078302 | 0.105541 | 0.006634 | 0.412721 | -0.149399 |
|  | **500** | 0.051241 | 0.071314 | 0.008896 | 0.261181 | -0.092220 |
|  | **1000** | 0.033224 | 0.036384 | 0.014430 | 0.153855 | -0.053505 |
|  | **5000** | 0.009999 | 0.003743 | 0.002192 | 0.017366 | -0.004931 |
| **GaussianMixture** | **50** | 0.177665 | 0.217804 | 0.089355 | 1.056630 | -0.550656 |
|  | **100** | 0.162771 | 0.241555 | 0.115915 | 0.874892 | -0.368532 |
|  | **300** | 0.087986 | 0.108009 | 0.055320 | 0.402553 | -0.148370 |
|  | **500** | 0.053534 | 0.067465 | 0.036292 | 0.245613 | -0.086680 |
|  | **1000** | 0.030153 | 0.029500 | 0.026102 | 0.126841 | -0.040905 |
|  | **5000** | 0.022381 | 0.016128 | 0.023862 | 0.047471 | -0.008795 |

|  |  | RMSE for the estimated parameters at N = 10000 |  |  |  |  |
|---|---|---|---|---|---|---|
|  | **n** | **pi1=0.3** | **mu1=-0.5** | **s1=1** | **mu2=1.5** | **s2=2** |
| **optimize.minimize** | **50** | 0.323319 | 0.929130 | 0.534819 | 1.728364 | 0.803988 |
|  | **100** | 0.306647 | 0.762120 | 0.474082 | 1.541460 | 0.634753 |
|  | **300** | 0.217394 | 0.382926 | 0.315779 | 0.901248 | 0.337064 |
|  | **500** | 0.172837 | 0.288231 | 0.248217 | 0.643691 | 0.237353 |
|  | **1000** | 0.124111 | 0.183979 | 0.171528 | 0.434855 | 0.156128 |
|  | **5000** | 0.038678 | 0.062865 | 0.065578 | 0.109753 | 0.037791 |
| **GaussianMixture** | **50** | 0.313462 | 1.181555 | 0.516099 | 1.674204 | 0.803330 |
|  | **100** | 0.291316 | 0.838832 | 0.454340 | 1.445340 | 0.624232 |
|  | **300** | 0.193529 | 0.366849 | 0.286641 | 0.777671 | 0.324736 |
|  | **500** | 0.147221 | 0.252280 | 0.224414 | 0.550008 | 0.223927 |
|  | **1000** | 0.094619 | 0.155174 | 0.150993 | 0.325478 | 0.128375 |
|  | **5000** | 0.039842 | 0.061252 | 0.066352 | 0.105952 | 0.036736 |

## 注意事項與討論：

### 觀察：

- 從結果可以看出，本次實驗的表現與 Step 3 有些差異。無論是 MLE 還是 GMM，兩者在平均值、偏差和 RMSE 上的表現相近，難以判定哪種方法更為優越。

- 此外，兩種方法在樣本數較少時，其估計的平均值與真實值之間存在明顯差異。然而，隨著樣本數增加，平均值逐漸趨於真實值，偏差和 RMSE 也隨之下降。

### 結論：

- 本次實驗的結果與 Step 3 不同，更接近 Step 2 的情況。這可能是因為經過 10000 次蒙地卡羅模擬後，兩種方法的估計偏差被進一步縮小，因此難以顯現出 MLE 和 GMM 在參數估計上的顯著差異。

- 本實驗表明，儘管在實驗次數較少且樣本數不足的情況下，GMM 的表現會優於 MLE，但當樣本數充足或模擬次數足夠多時，兩種方法的參數估計性能幾乎無差異，證實它們都是參數估計中相對穩健的方法。