

Expectation Maximization: Responsibility Method and MCMC Method

Noah Jackson

July 24th, 2025

Expectation Maximization (EM) is an algorithm designed for finding the mixes of a certain dataset. A mixed dataset means that data points can come from different distributions. For EM to work, we need to know two things about the data:

1. How many distributions are associated with the data.
2. What those distributions are.

For example, say we have a dataset call H that is mixed with a Normal, Exponential, and Uniform distribution. We would need to know that there are 3 distributions and that those distributions are a Normal, Exponential, and a Uniform.

There are multiple steps to this algorithm for it to work, in which I will explain each one:

1. Initialization, we set each mixture component weight, called π_j for distribution j to a value $\in [0, 1]$ such that all the mixture components added together = 1 (so basically a probability distribution of distributions). We also set initial parameters for all the distributions.
2. We repeat the following steps until we reach convergence or we have reached the number of iterations desired.
3. The E-step: We use what we know about the data points, the weights of the distributions, and the pdf's associated with each distributions to give us probabilities that a data point came from a particular distribution, or the number of samples from a particular distribution run.
4. The M-step: We update the mixing weights by averaging the probabilities that a data point come from a particular distribution over all data points for each component.
5. We check for convergence or if we reached number of iterations.

There are two strategies in doing Expectation Maximization, and my goal is to analyze each strategy and to look at the pros and cons of each.

I will start by explaining the first method, which involves computing responsibilities (Normal Method of EM). Responsibilities are the probabilities that a data point came from a particular distribution. This was basically stated in the start, but I didn't want to say this at the start because the other strategy doesn't use responsibilities, so I thought it would confuse some.

The difference between the strategies is how they change in the E-step and M-step. For the normal method, once we have initialized everything, our E-step goes as follows: Compute:

$$\gamma_{ij} = P(z_i = j \mid x_i, \theta) = \frac{\pi_j \cdot f_j(x_i)}{\sum_k \pi_k \cdot f_k(x_i)}$$

for all combinations of data point and distribution (or mix). Where γ_{ij} is the responsibility or probability that data point x_i came from the distribution j .

We then do the M-step for Normal EM, which is:

Update the mixture weights by taking the average over all responsibilities associated with mixture j , or:

$$\pi_j = \frac{1}{n} \sum_{i=1}^n \gamma_{ij}$$

For an intuitive sense, imagine that we acquire a responsibility super high associated with a data point and distribution (which likely means that the pdf value at the data point was high with respect to the others). This means that the probability the x_i is associated with that distribution is high. Since this is true, it would increase our mixture amount because we have a data point that is extremely "connected" to that certain distribution.

The next method is the point in which this project came to fruition in the first place.

The Expectation Maximization Monte Carlo Method

This is supposed to approximate the responsibility, or normal EM. The reason why someone may use a method is that the calculations for the responsibilities may be hard to compute.

You approximate the responsibilities by sampling in Monte Carlo style: for all the data points, make a Monte Carlo Markov Chain over for however many iterations, and basing our decision to go to a different node, or more specifically, making an educated guess on what distribution this data point comes from by using the acceptance ratio:

$$\alpha = \min \left(1, \frac{\pi_{z_{\text{new}}} \cdot f_{z_{\text{new}}}(x)}{\pi_z \cdot f_z(x)} \right)$$

What this becomes are samples of each data point, and for our M-step we look at all the different samples, and compute:

$$\pi_j = \frac{\text{number of samples where we get distribution } j}{\text{total number of samples}}$$

1 Methods

For my methods in analyzing these two methods, I will first start by implementing them into python and give a couple miniature examples of a mixed dataset and see how these methods favor. I will produce my own dataset to which I know all the true information of: The mixing weights, parameters, distributions, etc. and compare that as well to the results that both methods got.

Later on, we will create a random mixing distribution sampler where it creates mixes of distribution sizing from 2 to 8 mixes. The parameters will all be random in a certain predetermined interval for each.

To start, we must implement each Method, to which I have done so here:

2 Implementation

The first I did was the MCMC method...

```
1  import matplotlib.pyplot as plt
2  import numpy as np
3  import random
4  import math
5  import functools
6
7  #EM MC1#####
8  #####
9  #####
10 #####
11 #####
12
13 def pick_distribution(answer):
14
15     if answer == "Uniform":
16         while True:
17             try:
18                 a0 = float(input("Initial lower bound guess: "))
19                 break
20             except ValueError:
21                 print("Invalid input. Please enter a number.")
22         while True:
23             try:
24                 b0 = float(input("Initial upper bound guess: "))
25                 break
26             except ValueError:
27                 print("Invalid input. Please enter a number.")
28         def pdf(x):
29             return 1 / (b0 - a0) if a0 <= x <= b0 else 0
30
31         return pdf, (a0, b0)
32
33     if answer == "Exponential":
34         while True:
35             try:
36                 theta0 = float(input("Enter the mean guess: "))
37                 break
38             except ValueError:
39                 print("Invalid input. Please enter a number.")
40
41         def pdf(x):
42             return (1 / theta0) * math.exp(-x / theta0) if x >= 0 else 0
43
44         return pdf, (theta0,) # Corrected parameter return
45
46     if answer == "Normal":
47         while True:
48             try:
49                 mu0 = float(input("Enter the mean guess: "))
50                 break
51             except ValueError:
52                 print("Invalid input. Please enter a number.")
53         while True:
54             try:
55                 sigma0 = float(input("Enter the standard deviation guess: "))
56                 break
57             except ValueError:
58                 print("Invalid input. Please enter a number.")
59         def pdf(x_val):
```

```

60     return (1 / (sigma0 * math.sqrt(2 * math.pi))) * math.exp(-(x_val - mu0)**2 / (2 *
61         sigma0**2))
62     return pdf, (mu0, sigma0)
63
64 if answer == "Bernoulli":
65     while True:
66         try:
67             p = float(input("Enter probability of success (0 < p < 1): "))
68             if 0 < p < 1:
69                 break
70             else:
71                 print("p must be between 0 and 1.")
72         except ValueError:
73             print("Invalid input. Please enter a number.")
74
75 def pdf(x):
76     return p if x == 1 else (1 - p) if x == 0 else 0
77
78 return pdf, (p,)
79
80 if answer == "Poisson":
81     integer_data = [x for x in data if float(x).is_integer()]
82     estimated_lambda = np.mean(integer_data) if integer_data else 3.0
83
84     print(f"Suggested lambda (from integer data): {estimated_lambda:.2f}")
85     while True:
86         try:
87             lambda_val = float(input("Enter the Poisson mean (lambda): "))
88             if lambda_val > 0:
89                 break
90             else:
91                 print("lambda must be greater than 0.")
92         except ValueError:
93             print("Invalid input. Please enter a number.")
94
95 def pdf(x):
96     if x >= 0 and float(x).is_integer():
97         return (math.exp(-lambda_val) * (lambda_val ** x)) / math.factorial(int(x)) #
98         Corrected condition and factorial input
99     else:
100         return 0
101
102 return pdf, (lambda_val,)
103
104 # More distributions TBC
105
106 def e_step_MCMC(data, components, steps = 100):
107     n = len(data)
108     k = len(components)
109
110     z_samples = []
111
112     for i in range(n):
113         x = data[i]
114
115         z = np.random.choice(k)
116
117         chain = []
118
119         for t in range(steps):
120             if k > 1: # Add check for k > 1
121                 z_new = np.random.choice([j for j in range(k) if j != z])
122
123                 if components[z_new]["distr name"] == "Poisson" and not float(x).is_integer():
124                     continue
125                 if components[z_new]["distr name"] == "Bernoulli" and not float(x).is_integer():
126                     continue

```

```

126     num = components[z_new]["pi"] * components[z_new]["pdf"](x) # Score function which
127     will idealize the data point with a certain distribution if the associated pi and pdf
    value
128     den = components[z]["pi"] * components[z]["pdf"](x)           # with the data point is
    more than that of the previous
129
130     alpha = min(1, num / den if den > 0 else 1)
131
132     if np.random.rand() < alpha:
133         z = z_new
134
135     chain.append(z)
136
137     z_samples.append(chain)
138
139     return z_samples
140
141 def m_step_MCMC(z_samples, components):
142
143     n = len(data)
144     k = len(components)
145
146     flatz = [z for chain in z_samples for z in chain]
147
148     total = len(flatz)
149
150     eps = 1e-6
151     min_weight = 0.05
152
153     for i in range(k):
154         count_i = flatz.count(i)
155         pi_i = (count_i + eps) / (total)
156         components[i]["pi"] = max(min_weight, pi_i) # Updates pi / how much each distribution
    is mixed with how many exposures we got of the the specific distribution for all the
    data samples / total
157
    # we give (count_i + e-6) for the case when
    our samples got us no cases when the data point was associated with pi_i distribution.
    This would make it so
158
    # that theres no chance for the
    distribution to be revived no matter the pdf we get exposed to in the e step.
159
160     new_total = sum(comp["pi"] for comp in components)
161
162     for i in range(k):
163         components[i]["pi"] /= new_total # normalizing because of min_weight
164
165     return components
166
167 def update_params_pdfs(data, components, z_samples):
168     k = len(components)
169     n = len(data)
170
171     flat_x = [x for i, x in enumerate(data) for j in range(len(z_samples[i]))]
172     flat_z = [z for chain in z_samples for z in chain]
173
174     for i in range(len(components)):
175         print(f"Component {i}: {components[i]['distr name']}, count = {flat_z.count(i)}")
176
177     print()
178
179     for i in range(k):
180
181         spec_data = [x for x, z in zip(flat_x, flat_z) if z == i] # filters data points into
    what distribution we think the data points are associated with
182
183         if components[i]["distr name"] == "Uniform":
184             if spec_data: # Add check for empty list

```

```

185         components[i]["params"] = (min(spec_data), max(spec_data))
186     else:
187         pass
188
189     if components[i]["distr name"] == "Exponential":
190         if spec_data: # Add check for empty list
191             components[i]["params"] = (np.mean(spec_data),)
192         else:
193             pass
194
195     if components[i]["distr name"] == "Normal":
196         if spec_data: # Add check for empty list
197             components[i]["params"] = (np.mean(spec_data), np.std(spec_data))
198         else:
199             pass
200
201     if components[i]["distr name"] == "Bernoulli":
202         if spec_data:
203             components[i]["params"] = (np.mean(spec_data),)
204         else:
205             pass
206
207     if components[i]["distr name"] == "Poisson":
208         if spec_data:
209             components[i]["params"] = (np.mean(spec_data),)
210         else:
211             pass
212
213     for comp in components:
214         name = comp["distr name"]
215         params = comp["params"]
216
217         if name == "Normal":
218             if len(spec_data) > 5:
219                 mu, sigma = params
220                 def f(x, mu=mu, sigma=sigma):
221                     return (1 / (sigma * math.sqrt(2 * math.pi))) * math.exp(-(x - mu)**2 / (2 *
sigma**2))
222                 comp["pdf"] = f
223             elif name == "Poisson":
224                 lamb = params[0]
225                 def f(x, lamb=lamb):
226                     if x >= 0 and float(x).is_integer():
227                         return (math.exp(-lamb) * (lamb ** x)) / math.factorial(int(x))
228                     else:
229                         return 0
230                 comp["pdf"] = f
231             elif name == "Exponential":
232                 theta = params[0]
233                 def f(x, theta=theta):
234                     return (1 / theta) * math.exp(-x / theta) if x >= 0 else 0
235                 comp["pdf"] = f
236             elif name == "Uniform":
237                 a, b = params
238                 def f(x, a=a, b=b):
239                     return 1 / (b - a) if a <= x <= b else 0
240                 comp["pdf"] = f
241             elif name == "Bernoulli":
242                 p = params[0]
243                 def f(x, p=p):
244                     return p if x == 1 else (1 - p) if x == 0 else 0
245                 comp["pdf"] = f
246             else:
247                 raise ValueError(f"Unknown distribution name: {name}")
248
249     return components
250
251 def compute_log_likelihood(data, components):

```

```

252     log_likelihood = 0
253     for x in data:
254         mixture_prob = sum(comp['pi'] * comp['pdf'](x) for comp in components)
255         if mixture_prob > 0:
256             log_likelihood += np.log(mixture_prob)
257     return log_likelihood
258
259
260 def EM_MCMC(data, num_iters):
261
262     components = []
263
264     # data = [0, 1, 8, 6, 2, 4] # Removed hardcoded data
265
266     num_components = int(input("How many distributions would you like to mix? "))
267
268     components = []
269     for i in range(num_components):
270         distr = ''
271         while distr not in ["Uniform", "Exponential", "Normal", "Poisson", "Bernoulli"]:
272             distr = input(f"Choose distribution {i+1}: ")
273         pdf_f, param = pick_distribution(distr)
274         components.append({"distr name": distr, "pdf": pdf_f, "params": param, "pi": 1 /
275                             # chance of being associated with one of the distributions.
276                             num_components})
277
278     ll_list = []
279     for i in range(num_iters):
280         z_samples = e_step_MCMC(data, components) #
281         components = m_step_MCMC(z_samples, components)
282         components = update_params_pdfs(data, components, z_samples)
283
284         ll = compute_log_likelihood(data, components)
285         ll_list.append(ll)
286         if i > 0:
287             print(f"Iteration {i+1}: Log-likelihood = {ll}")
288
289     # plot log likelihood changes
290     plt.figure(figsize=(10, 6))
291     plt.scatter(range(len(ll_list)), ll_list, label='Data')
292
293     coeffs = np.polyfit(range(len(ll_list)), ll_list, deg=3)
294     trendline = np.poly1d(coeffs)
295
296     plt.plot(range(len(ll_list)), trendline(range(len(ll_list))), color='blue', label='
297             Trendline')
298
299     plt.xlabel("iteration")
300     plt.ylabel("log likelihood")
301     plt.legend()
302     plt.show()
303
304     print("Final parameters:")
305
306     for i in range(len(components)):
307         print(f"Component {i+1}: {components[i]['distr name']}")
308         print("Parameters:", *(float(x) for x in components[i]['params']))
309         print(f"Weight: {float(components[i]['pi'])}")
310         print()
311     return
312
313 np.random.seed(924)
314
315 # Generate data
316 n = 300
317 data = []
318
319 # 40% Normal(2, 0.5)

```

```

317 data += list(np.random.normal(loc=2, scale=0.5, size=int(0.4 * n)))
318
319 # 30% Uniform(0,5)
320 data += list(np.random.uniform(0,5, size=int(0.3 * n)))
321
322 # 30% Exponential(1.5)
323 data += list(np.random.exponential(scale=1.5, size=int(0.3 * n)))
324
325 # Optional: Shuffle the data
326 random.shuffle(data)
327
328
329 EM_MCMC(data, 100)

```

This is a user input based code, where the user may pick the amount of mixes, distribution types, and parameters.

I will focus on two functions implemented here:

- **e_step_MCMC**: This step iterates over each data point, makes a list of our current weight mix times the PDF at that data point, sums it up to compute the probability weight for our initial z value. It then iterates over the number of MCMC steps we chose by looking at a random z , checking it against α , and making a random decision to transition to that z , which is then added to the list.
- **m_step_MCMC**: This step flattens our list of lists (or merges separate samples into one big sample). For each component, it counts the number of sample points assigned and divides by the total. An important detail is the inclusion of the `min_weight`, which ensures that no mixture component ever gets a weight below 0.05. Without this, some components could end up with a weight of 0, making them inaccessible in future iterations. This normalization step stabilizes the updates and prevents collapse.

It also printed:

```

1 Iteration 100: Log-likelihood = -444.778650236946497
2 Component 0: Normal, count = 10454
3 Component 1: Uniform, count = 6056
4 Component 2: Exponential, count = 9890
5
6 Final parameters:
7 Component 1: Normal
8 Parameters: 1.8400664463447932 0.5583405532495043
9 Weight: 0.46346666653115324
10
11 Component 2: Uniform
12 Parameters: 2.00142967992631 4.939653309234937
13 Weight: 0.2081666667981334
14
15 Component 3: Exponential
16 Parameters: 1.2523636181134291
17 Weight: 0.3290666666708323

```

Listing 1: EM Algorithm Final Output

and also the graph:

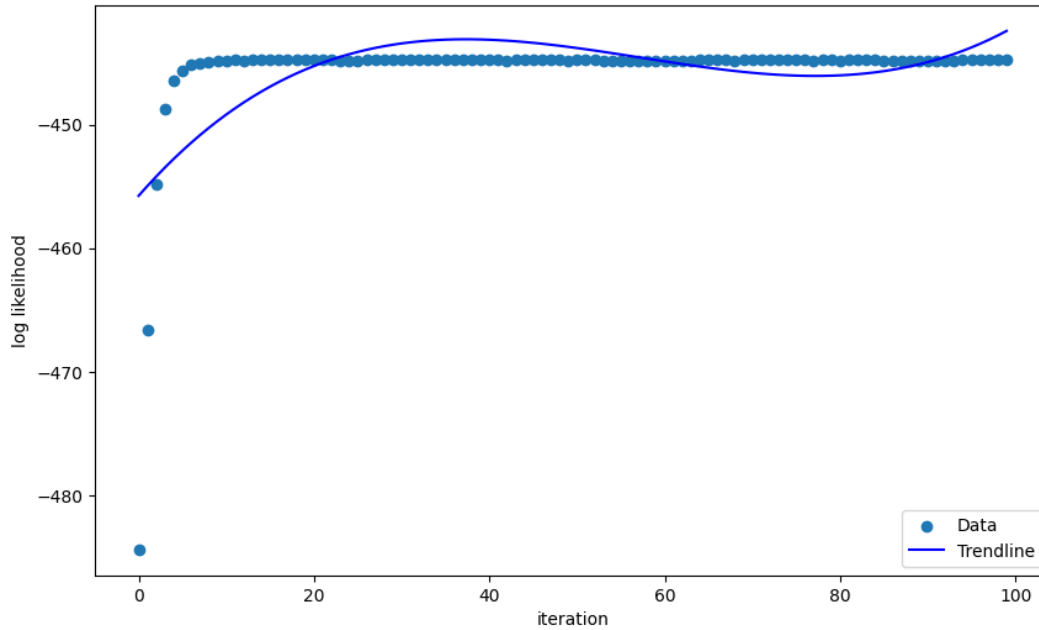


Figure 1: Log-likelihood progression over 100 EM iterations. The mixture includes Normal, Uniform, and Exponential components.

The second I did was the normal...

```

1  import matplotlib.pyplot as plt
2  import numpy as np
3  import random
4  import math
5  import functools
6
7  #Normal#####
8  #####
9  #####
10 #####
11 #####
12 def pick_distribution(answer):
13
14     if answer == "Uniform":
15         while True:
16             try:
17                 a0 = float(input("Initial lower bound guess: "))
18                 break
19             except ValueError:
20                 print("Invalid input. Please enter a number.")
21         while True:
22             try:
23                 b0 = float(input("Initial upper bound guess: "))
24                 break
25             except ValueError:
26                 print("Invalid input. Please enter a number.")
27         def pdf(x):
28             return 1 / (b0 - a0) if a0 <= x <= b0 else 0
29
30         return pdf, (a0, b0)
31
32     if answer == "Exponential":
33         while True:
34             try:
35                 theta0 = float(input("Enter the mean guess: "))

```

```

36         break
37     except ValueError:
38         print("Invalid input. Please enter a number.")
39
40     def pdf(x):
41         return (1 / theta0) * math.exp(-x / theta0) if x >= 0 else 0
42
43     return pdf, (theta0,) # Corrected parameter return
44
45 if answer == "Normal":
46     while True:
47         try:
48             mu0 = float(input("Enter the mean guess: "))
49             break
50         except ValueError:
51             print("Invalid input. Please enter a number.")
52     while True:
53         try:
54             sigma0 = float(input("Enter the standard deviation guess: "))
55             break
56         except ValueError:
57             print("Invalid input. Please enter a number.")
58     def pdf(x_val):
59         return (1 / (sigma0 * math.sqrt(2 * math.pi))) * math.exp(-(x_val - mu0)**2 / (2 *
sigma0**2))
60     return pdf, (mu0, sigma0)
61
62 if answer == "Bernoulli":
63     while True:
64         try:
65             p = float(input("Enter probability of success (0 < p < 1): "))
66             if 0 < p < 1:
67                 break
68             else:
69                 print("p must be between 0 and 1.")
70         except ValueError:
71             print("Invalid input. Please enter a number.")
72
73     def pdf(x):
74         return p if x == 1 else (1 - p) if x == 0 else 0
75
76     return pdf, (p,)
77
78 if answer == "Poisson":
79
80     integer_data = [x for x in data if float(x).is_integer()]
81     estimated_lambda = np.mean(integer_data) if integer_data else 3.0
82
83     print(f"Suggested lambda (from integer data): {estimated_lambda:.2f}")
84     while True:
85         try:
86             lambda_val = float(input("Enter the Poisson mean (lambda): "))
87             if lambda_val > 0:
88                 break
89             else:
90                 print("lambda must be greater than 0.")
91         except ValueError:
92             print("Invalid input. Please enter a number.")
93     def pdf(x):
94         if x >= 0 and float(x).is_integer():
95             return (math.exp(-lambda_val) * (lambda_val ** x)) / math.factorial(int(x)) #
Corrected condition and factorial input
96         else:
97             return 0
98
99     return pdf, (lambda_val,)
100
101

```

```

102 # More distributions TBC
103
104 #-----
105
106 def e_step(data, components):
107     responsibilities = []
108     for x in data:
109         num = [comp["pi"] * comp["pdf"](x) for comp in components]
110         tot = sum(num)
111         if tot == 0:
112             probs = [1 / len(components)] * len(components)
113         else:
114             probs = [n / tot for n in num]
115         responsibilities.append(probs)
116     return responsibilities
117
118 def m_step(data, responsibilities, components):
119     n = len(data)
120     k = len(components)
121
122     for i in range(k):
123         r_i = [resp[i] for resp in responsibilities]
124         total_r = sum(r_i)
125         components[i]["pi"] = total_r / n if total_r > 0 else 1e-6
126
127         spec_data = [x * r for x, r in zip(data, r_i)]
128
129         if components[i]["distr name"] == "Uniform":
130             # Weighted min/max
131             weights = np.array(r_i)
132             if np.sum(weights) > 0:
133                 x_array = np.array(data)
134                 components[i]["params"] = (np.min(x_array[weights > 0]), np.max(x_array[
135 weights > 0]))
136             else:
137                 pass # Keep previous parameters if no data points assigned to this component
138
139         elif components[i]["distr name"] == "Exponential":
140             if total_r > 0:
141                 theta = sum(spec_data) / total_r
142                 components[i]["params"] = (theta,)
143             else:
144                 pass # Keep previous parameters if no data points assigned to this component
145
146         elif components[i]["distr name"] == "Normal":
147             if total_r > 0:
148                 mu = sum(spec_data) / total_r
149                 var = sum(r * ((x - mu)**2) for x, r in zip(data, r_i)) / total_r
150                 sigma = math.sqrt(var)
151                 components[i]["params"] = (mu, sigma)
152             else:
153                 pass # Keep previous parameters if no data points assigned to this component
154
155         elif components[i]["distr name"] == "Bernoulli":
156             if total_r > 0:
157                 p = sum(spec_data) / total_r
158                 components[i]["params"] = (p,)
159             else:
160                 pass # Keep previous parameters if no data points assigned to this component
161
162         elif components[i]["distr name"] == "Poisson":
163             if total_r > 0:
164                 lambda_ = sum(spec_data) / total_r
165                 components[i]["params"] = (lambda_,)
166             else:
167                 pass # Keep previous parameters if no data points assigned to this component
168

```

```

169     update_pdfs(components)
170     return components
171
172 def update_pdfs(components):
173     for comp in components:
174         name = comp["distr name"]
175         params = comp["params"]
176
177         if name == "Normal":
178             mu, sigma = params
179             def f(x, mu=mu, sigma=sigma):
180                 return (1 / (sigma * math.sqrt(2 * math.pi))) * math.exp(-(x - mu)**2 / (2 *
sigma**2))
181             comp["pdf"] = f
182
183         elif name == "Poisson":
184             lamb = params[0]
185             def f(x, lamb=lamb):
186                 if x >= 0 and float(x).is_integer():
187                     return (math.exp(-lamb) * (lamb ** x)) / math.factorial(int(x))
188                 else:
189                     return 0
190             comp["pdf"] = f
191
192         elif name == "Exponential":
193             theta = params[0]
194             def f(x, theta=theta):
195                 return (1 / theta) * math.exp(-x / theta) if x >= 0 else 0
196             comp["pdf"] = f
197
198         elif name == "Uniform":
199             a, b = params
200             def f(x, a=a, b=b):
201                 return 1 / (b - a) if a <= x <= b else 0
202             comp["pdf"] = f
203
204         elif name == "Bernoulli":
205             p = params[0]
206             def f(x, p=p):
207                 return p if x == 1 else (1 - p) if x == 0 else 0
208             comp["pdf"] = f
209
210         else:
211             raise ValueError(f"Unknown distribution name: {name}")
212
213 # -----
214 def update_params_pdfs(data, components, z_samples):
215     k = len(components)
216     n = len(data)
217
218     flat_x = [x for i, x in enumerate(data) for j in range(len(z_samples[i]))]
219     flat_z = [z for chain in z_samples for z in chain]
220
221     for i in range(len(components)):
222         print(f"Component {i}: {components[i]['distr name']}, count = {flat_z.count(i)}")
223
224     print()
225
226     for i in range(k):
227
228         spec_data = [x for x, z in zip(flat_x, flat_z) if z == i] # filters data points into
what distribution we think the data points are associated with
229
230         if components[i]["distr name"] == "Uniform":
231             if spec_data: # Add check for empty list
232                 components[i]["params"] = (min(spec_data), max(spec_data))
233             else:
234                 pass

```

```

235
236     if components[i]["distr name"] == "Exponential":
237         if spec_data: # Add check for empty list
238             components[i]["params"] = (np.mean(spec_data),)
239         else:
240             pass
241
242     if components[i]["distr name"] == "Normal":
243         if spec_data: # Add check for empty list
244             components[i]["params"] = (np.mean(spec_data), np.std(spec_data))
245         else:
246             pass
247
248     if components[i]["distr name"] == "Bernoulli":
249         if spec_data:
250             components[i]["params"] = (np.mean(spec_data),)
251         else:
252             pass
253
254     if components[i]["distr name"] == "Poisson":
255         if spec_data:
256             components[i]["params"] = (np.mean(spec_data),)
257         else:
258             pass
259
260     for comp in components:
261         name = comp["distr name"]
262         params = comp["params"]
263
264         if name == "Normal":
265             if len(spec_data) > 5:
266                 mu, sigma = params
267                 def f(x, mu=mu, sigma=sigma):
268                     return (1 / (sigma * math.sqrt(2 * math.pi))) * math.exp(-(x - mu)**2 / (2 *
sigma**2))
269                 comp["pdf"] = f
270             elif name == "Poisson":
271                 lamb = params[0]
272                 def f(x, lamb=lamb):
273                     if x >= 0 and float(x).is_integer():
274                         return (math.exp(-lamb) * (lamb ** x)) / math.factorial(int(x))
275                     else:
276                         return 0
277                 comp["pdf"] = f
278             elif name == "Exponential":
279                 theta = params[0]
280                 def f(x, theta=theta):
281                     return (1 / theta) * math.exp(-x / theta) if x >= 0 else 0
282                 comp["pdf"] = f
283             elif name == "Uniform":
284                 a, b = params
285                 def f(x, a=a, b=b):
286                     return 1 / (b - a) if a <= x <= b else 0
287                 comp["pdf"] = f
288             elif name == "Bernoulli":
289                 p = params[0]
290                 def f(x, p=p):
291                     return p if x == 1 else (1 - p) if x == 0 else 0
292                 comp["pdf"] = f
293             else:
294                 raise ValueError(f"Unknown distribution name: {name}")
295
296     return components
297
298 def compute_log_likelihood(data, components):
299     log_likelihood = 0
300     for x in data:
301         mixture_prob = sum(comp['pi'] * comp['pdf'](x) for comp in components)

```

```

302         if mixture_prob > 0:
303             log_likelihood += np.log(mixture_prob)
304         return log_likelihood
305
306
307 def EM_MCMC(data, num_iters):
308
309     components = []
310
311     # data = [0, 1, 8, 6, 2, 4] # Removed hardcoded data
312
313     num_components = int(input("How many distributions would you like to mix? "))
314
315     components = []
316     for i in range(num_components):
317         distr = ''
318         while distr not in ["Uniform", "Exponential", "Normal", "Poisson", "Bernoulli"]:
319             distr = input(f"Choose distribution {i+1}: ")
320         pdf_f, param = pick_distribution(distr)
321         components.append({"distr name": distr, "pdf": pdf_f, "params": param, "pi": 1 /
322                             num_components}) # --> Start by assuming that each data point has equal
323
324                                     # chance of being associated with one of the distributions.
325
326     ll_list = []
327     for i in range(num_iters):
328         z_samples = e_step_MCMC(data, components) #
329         components = m_step_MCMC(z_samples, components)
330         components = update_params_pdfs(data, components, z_samples)
331
332         ll = compute_log_likelihood(data, components)
333         ll_list.append(ll)
334         if i > 0:
335             print(f"Iteration {i+1}: Log-likelihood = {ll}")
336
337     # plot log likelihood changes
338     plt.figure(figsize=(10, 6))
339     plt.scatter(range(len(ll_list)), ll_list, label='Data')
340
341     coeffs = np.polyfit(range(len(ll_list)), ll_list, deg=3)
342     trendline = np.poly1d(coeffs)
343
344     plt.plot(range(len(ll_list)), trendline(range(len(ll_list))), color='blue', label='
345             Trendline')
346
347     plt.xlabel("iteration")
348     plt.ylabel("log likelihood")
349     plt.legend()
350     plt.show()
351
352     print("Final parameters:")
353
354     for i in range(len(components)):
355         print(f"Component {i+1}: {components[i]['distr name']}")
356         print("Parameters:", *(float(x) for x in components[i]['params']))
357         print(f"Weight: {float(components[i]['pi'])}")
358         print()
359     return
360
361 np.random.seed(924)
362
363 # Generate data
364 n = 300
365 data = []
366
367 # 40% Normal(2, 0.5)
368 data += list(np.random.normal(loc=2, scale=0.5, size=int(0.4 * n)))
369
370 # 30% Uniform(0,5)

```

```

367 data += list(np.random.uniform(0,5, size=int(0.3 * n)))
368
369 # 30% Exponential(1.5)
370 data += list(np.random.exponential(scale=1.5, size=int(0.3 * n)))
371
372 # Optional: Shuffle the data
373 random.shuffle(data)
374
375
376 EM_MCMC(data, 100)

```

This code goes off of how the Normal EM is a simple plug and chug method (which also makes it easier on the computer).

I will analyze two functions again:

1. **e_step**: This creates us a list of responsibilities, where each row corresponds to a data point and each column is a distribution.
2. **m_step**: Updates by using the formula, which is the average of responsibilities for all data points. Also updates the parameters makes use of the `update_pdfs` function, which the MCMC EM didn't do.

And it printed:

```

1 Iteration 98: Log-likelihood = -444.7794301620609
2 Component 0: Normal, count = 13965
3 Component 1: Uniform, count = 6015
4 Component 2: Exponential, count = 10020
5
6 Iteration 99: Log-likelihood = -444.7785021597151
7 Component 0: Normal, count = 13964
8 Component 1: Uniform, count = 6041
9 Component 2: Exponential, count = 9995
10
11 Iteration 100: Log-likelihood = -444.7976364106383
12
13 Final parameters:
14 Component 1: Normal
15 Parameters: 1.8423191947752293 0.5628494357388811
16 Weight: 0.46546666666534534
17
18 Component 2: Uniform
19 Parameters: 2.001249637992631 4.939563309234937
20 Weight: 0.201366666667986333
21
22 Component 3: Exponential
23 Parameters: 1.2616382356757725
24 Weight: 0.33316666666666833

```

Listing 2: EM Algorithm Final Output

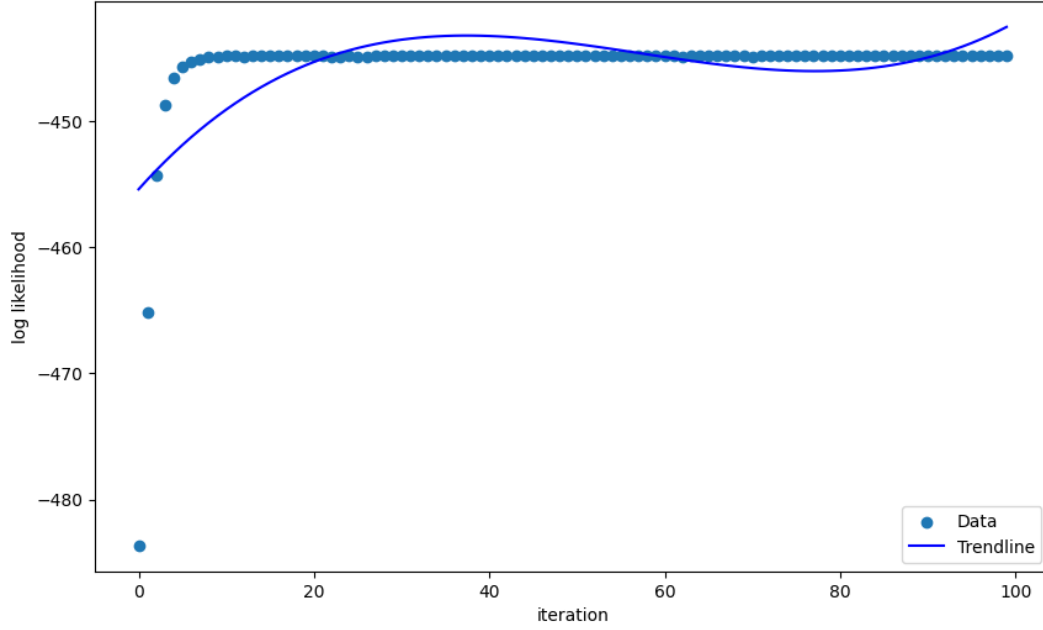


Figure 2: (Normal) Log-likelihood progression over 100 EM iterations. The mixture includes Normal, Uniform, and Exponential components.

3 Analysis

After implementing, we now analyze certain data set that we already know the mixes, distributions, and parameters to. This dataset is 40% normal, 30% uniform and 30% Exponential. This gave us:

Normal EM - Final parameters:

- Component 1: Normal
 - Parameters: 1.8423191947752293, 0.5628494357388811
 - Weight: 0.4654666666534534
- Component 2: Uniform
 - Parameters: 2.001249637992631, 4.939563309234937
 - Weight: 0.20136666667986333
- Component 3: Exponential
 - Parameters: 1.2616382356757725
 - Weight: 0.33316666666666833

EMMC: Final parameters

- Component 1: Normal
 - Parameters: 1.8460064463447392, 0.5583405533495043

- Weight: 0.46846666665315334
- Component 2: Uniform
 - Parameters: 2.001249637992631, 4.939563309234937
 - Weight: 0.20186666667981334
- Component 3: Exponential
 - Parameters: 1.2533636181134291
 - Weight: 0.3296666666670333

This is one example of one of the tests I did for a certain mix.

These values seem to be very closely related, showing that these 2 strategies are almost equivalent. Of course, both methods can't exactly bring us back to the parameters we chose, as that depends on the sample itself, and the parameters we chose.

I did multiple tests with a mix of Continuous and Discrete distributions as well, and a common theme of it was that the continuous distributions dominated. I think this is possibly due to the fact that continuous distributions can take on values that discrete can, but the inverse is not true.

What was also interesting was that Discrete Mixes converged more rapidly than in Continuous mixes (and I think this is also related to how discrete only takes on a subset of continuous values).

To further my investigation, I created a mixing function which allowed to create random mixes of different distributions. This code allowed to no user input and purely off computer randomness.

```

1  import matplotlib.pyplot as plt
2  import numpy as np
3  import random
4  import math
5  #----- Distribution Picker -----#
6  def pick_distribution(answer):
7      """
8      Automatically assigns parameters and returns (pdf, param_tuple)
9      based on the chosen distribution name.
10     """
11
12     if answer == "Uniform":
13         a0 = random.randint(0, 5)
14         b0 = random.randint(a0 + 1, a0 + 8) # ensure b > a
15         def pdf(x): return 1 / (b0 - a0) if a0 <= x <= b0 else 0
16         return pdf, (a0, b0)
17
18     if answer == "Exponential":
19         theta0 = random.uniform(0.5, 5.0)
20         def pdf(x): return (1 / theta0) * math.exp(-x / theta0) if x >= 0 else 0
21         return pdf, (theta0,)
22
23     if answer == "Normal":
24         mu0 = random.uniform(-5, 5)
25         sigma0 = random.uniform(0.5, 3.0)
26         def pdf(x): return (1 / (sigma0 * math.sqrt(2 * math.pi))) * math.exp(-(x - mu0)**2
27         / (2 * sigma0**2))
28         return pdf, (mu0, sigma0)
29
30     if answer == "Poisson":
31         lambda_val = random.uniform(1, 8)
32         def pdf(x):
33             if x >= 0 and float(x).is_integer():
34                 x_int = int(x)

```

```

34         log_pdf = -lambda_val + x_int * math.log(lambda_val) - math.lgamma(x_int +
1)
35         return math.exp(log_pdf)
36         return 0
37         return pdf, (lambda_val,)
38
39     if answer == "Bernoulli":
40         p = random.uniform(0.1, 0.9)
41         def pdf(x): return p if x == 1 else (1 - p) if x == 0 else 0
42         return pdf, (p,)
43
44     raise ValueError(f"Unsupported distribution name: {answer}")
45
46 #####EM Normal
47 #####
48 #
49 #####
50 #
51 #####
52 #
53 #####
54 #
55 #####
56 #
57 #####
58
59 #----- PDF Updater -----#
60 def update_pdfs(components):
61     """
62     After parameter updates, refresh each component's PDF.
63     """
64     for comp in components:
65         name = comp["distr name"]
66         params = comp["params"]
67
68         if name == "Normal":
69             mu, sigma = params
70             comp["pdf"] = lambda x, mu=mu, sigma=sigma: (1 / (sigma * math.sqrt(2 * math.pi)
)) * math.exp(-(x - mu)**2 / (2 * sigma**2))
71
72         elif name == "Poisson":
73             lamb = params[0]
74             comp["pdf"] = lambda x, lamb=lamb: math.exp(-lamb + x * math.log(lamb) - math.
lgamma(x + 1)) if x >= 0 and float(x).is_integer() else 0
75

```

```

76         elif name == "Exponential":
77             theta = params[0]
78             comp["pdf"] = lambda x, theta=theta: (1 / theta) * math.exp(-x / theta) if x >=
0 else 0
79
80         elif name == "Uniform":
81             a, b = params
82             comp["pdf"] = lambda x, a=a, b=b: 1 / (b - a) if a <= x <= b else 0
83
84         elif name == "Bernoulli":
85             p = params[0]
86             comp["pdf"] = lambda x, p=p: p if x == 1 else (1 - p) if x == 0 else 0
87         else:
88             raise ValueError(f"Unknown distribution name: {name}")
89
90 #----- E-step -----#
91 def e_step(data, components):
92     """
93     Estimate responsibilities: P(z_i = j | x_i, )
94     """
95     responsibilities = []
96     for x in data:
97         weighted_probs = [comp["pi"] * comp["pdf"](x) for comp in components]
98         total = sum(weighted_probs)
99         probs = [wp / total for wp in weighted_probs] if total > 0 else [1 / len(components)
] * len(components)
100         responsibilities.append(probs)
101     return responsibilities
102
103 #----- M-step -----#
104 def m_step(data, responsibilities, components):
105     """
106     Update weights and distribution parameters based on responsibilities.
107     """
108     n = len(data)
109     k = len(components)
110
111     for i in range(k):
112         r_i = [resp[i] for resp in responsibilities]
113         total_r = sum(r_i)
114         components[i]["pi"] = total_r / n if total_r > 0 else 1e-6
115
116         weighted_data = [x * r for x, r in zip(data, r_i)]
117
118         name = components[i]["distr name"]
119
120         if name == "Uniform":
121             if np.sum(r_i) > 0:
122                 x_array = np.array(data)
123                 components[i]["params"] = (np.min(x_array), np.max(x_array))
124
125         elif name == "Exponential" and total_r > 0:
126             theta = sum(weighted_data) / total_r
127             components[i]["params"] = (theta,)
128
129         elif name == "Normal" and total_r > 0:
130             mu = sum(weighted_data) / total_r
131             var = sum(r * ((x - mu)**2) for x, r in zip(data, r_i)) / total_r
132             sigma = math.sqrt(var)
133             components[i]["params"] = (mu, sigma)
134
135         elif name == "Bernoulli" and total_r > 0:
136             p = sum(weighted_data) / total_r
137             components[i]["params"] = (p,)
138
139         elif name == "Poisson" and total_r > 0:
140             lambda_ = sum(weighted_data) / total_r
141             components[i]["params"] = (lambda_,)

```

```

142     update_pdfs(components)
143     return components
144 #####EM MC
145 #####
146 #
147 #####
148 #
149 #####
150 #
151 #####
152 #
153 #####
154 #
155 #####
156
157 #----- E-Step (MCMC) -----#
158 def e_step_MCMC(data, components, steps=100):
159     z_samples = []
160     k = len(components)
161     for x in data:
162         weights = [comp["pi"] * comp["pdf"](x) for comp in components]
163         total = sum(weights)
164         weights = [w / total for w in weights] if total > 0 else [1 / k] * k
165         z = np.random.choice(range(k), p=weights)
166         chain = []
167         for _ in range(steps):
168             if k > 1:
169                 z_new = np.random.choice([j for j in range(k) if j != z])
170                 if components[z_new]["distr name"] in {"Poisson", "Bernoulli"} and not float
(x).is_integer():
171                     chain.append(z)
172                     continue
173                 num = components[z_new]["pi"] * components[z_new]["pdf"](x)
174                 den = components[z]["pi"] * components[z]["pdf"](x)
175                 alpha = min(1, num / den if den > 0 else 1)
176                 if np.random.rand() < alpha:
177                     z = z_new
178             chain.append(z)
179         z_samples.append(chain)
180     return z_samples
181
182
183 #----- M-Step -----#
184 def m_step_MCMC(z_samples, components):
185     flatz = [z for chain in z_samples for z in chain]

```

```

186 total = len(flatz)
187 eps = 1e-6
188 min_weight = 0.05
189 for i in range(len(components)):
190     count_i = flatz.count(i)
191     pi_i = (count_i + eps) / total
192     components[i]["pi"] = max(min_weight, pi_i)
193 total_pi = sum(comp["pi"] for comp in components)
194 for comp in components:
195     comp["pi"] /= total_pi
196 return components
197
198
199 #----- Update Parameters -----#
200 def update_params_pdfs(data, components, z_samples):
201     flat_x = [x for i, x in enumerate(data) for _ in z_samples[i]]
202     flat_z = [z for chain in z_samples for z in chain]
203     for i, comp in enumerate(components):
204         spec_data = [x for x, z in zip(flat_x, flat_z) if z == i]
205         if not spec_data:
206             continue
207         name = comp["distr name"]
208         if name == "Uniform":
209             comp["params"] = (min(spec_data), max(spec_data))
210         elif name == "Exponential":
211             comp["params"] = (np.mean(spec_data),)
212         elif name == "Normal":
213             comp["params"] = (np.mean(spec_data), np.std(spec_data))
214         elif name == "Bernoulli":
215             comp["params"] = (np.mean(spec_data),)
216         elif name == "Poisson":
217             comp["params"] = (np.mean(spec_data),)
218
219         # Update pdfs
220         params = comp["params"]
221         if name == "Normal":
222             mu, sigma = params
223             comp["pdf"] = lambda x, mu=mu, sigma=sigma: (1 / (sigma * math.sqrt(2 * math.pi)
224 )) * math.exp(-(x - mu)**2 / (2 * sigma**2))
225         elif name == "Poisson":
226             lamb = params[0]
227             comp["pdf"] = lambda x, lamb=lamb: math.exp(-lamb + x * math.log(lamb) - math.
228 lgamma(x + 1)) if x >= 0 and float(x).is_integer() else 0
229         elif name == "Exponential":
230             theta = params[0]
231             comp["pdf"] = lambda x, theta=theta: (1 / theta) * math.exp(-x / theta) if x >=
232 0 else 0
233         elif name == "Uniform":
234             a, b = params
235             comp["pdf"] = lambda x, a=a, b=b: 1 / (b - a) if a <= x <= b else 0
236         elif name == "Bernoulli":
237             p = params[0]
238             comp["pdf"] = lambda x, p=p: p if x == 1 else (1 - p) if x == 0 else 0
239     return components
240
241 #----- EM-MCMC Main -----#
242 def EM_MCMC(data, true_comps, iter = 25):
243     complen = len(true_comps)
244     components = []
245     for i in range(complen):
246         distr = true_comps[i]["distr name"]
247         pdf_f, param = pick_distribution(distr)
248         components.append({
249             "distr name": distr,
250             "pdf": pdf_f,
251             "params": param,
252             "pi": 1/complen # we assume dont know the exact mix of the data

```

```

251     })
252
253     for i in range(iter):
254         z_samples = e_step_MCMC(data, components)
255         components = m_step_MCMC(z_samples, components)
256         components = update_params_pdfs(data, components, z_samples)
257
258     return [components[i]['pi'] for i in range(complen)]
259
260 #----- EM Main -----#
261 def EM(data, true_comps, iter = 25):
262
263     num_components = len(true_comps)
264     components = []
265     for i in range(num_components):
266         distr = true_comps[i]["distr name"]
267         pdf_f, param = pick_distribution(distr)
268         components.append({
269             "distr name": distr,
270             "pdf": pdf_f,
271             "params": param,
272             "pi": 1 / num_components
273         })
274
275     for i in range(iter):
276         responsibilities = e_step(data, components)
277         components = m_step(data, responsibilities, components)
278
279     return [components[i]['pi'] for i in range(num_components)]
280
281
282
283 import random as rd
284
285
286 def give_data_with_mixes_distrib_continuous():
287
288     num_mixes = rd.randint(2, 8)
289
290     distributions = ["Uniform", "Exponential", "Normal"]
291     components = []
292
293     true_components = []
294     sum = 0
295     for i in range(num_mixes):
296         distr = rd.choice(distributions)
297         pdf_f, param = pick_distribution(distr)
298         pi_i = rd.uniform(0, 1 - sum)
299         sum += pi_i
300         true_components.append({
301             "distr name": distr,
302             "pdf": pdf_f,
303             "params": param,
304             "pi": pi_i
305         })
306
307     # create data
308     data = []
309     n = rd.randint(5, 1000)
310     for comp in true_components:
311         if comp["distr name"] == "Uniform":
312             data += list(np.random.uniform(comp["params"][0], comp["params"][1], size=int(comp["pi"] * n)))
313         if comp["distr name"] == "Exponential":
314             data += list(np.random.exponential(comp["params"][0], size=int(comp["pi"] * n)))
315         if comp["distr name"] == "Normal":
316             data += list(np.random.normal(comp["params"][0], comp["params"][1], size=int(comp["pi"] * n)))
317     return data, true_components

```

```

317 #####
318
319 def give_data_with_mixes_distribs_discrete():
320
321     num_mixes = rd.randint(2, 8)
322
323     distributions = ["Poisson", "Bernoulli"]
324     components = []
325
326     true_components = []
327     sum = 0
328     for i in range(num_mixes):
329         distr = rd.choice(distributions)
330         pdf_f, param = pick_distribution(distr)
331         pi_i = rd.uniform(0, 1 - sum)
332         sum += pi_i
333         true_components.append({
334             "distr name": distr,
335             "pdf": pdf_f,
336             "params": param,
337             "pi": pi_i
338         })
339     # create data
340     data = []
341     n = rd.randint(5, 1000)
342     for comp in true_components:
343         if comp["distr name"] == "Poisson":
344             data += list(np.random.poisson(comp["params"][0], size=int(comp["pi"] * n)))
345         if comp["distr name"] == "Bernoulli":
346             data += list(np.random.binomial(1, comp["params"][0], size=int(comp["pi"] * n)))
347         if comp["distr name"] == "Binomial":
348             data += list(np.random.binomial(comp["params"][0], comp["params"][1], size=int(comp["pi"] * n)))
349     return data, true_components
350 #####
351
352 def do_test(iter = 10):
353
354     num_test = 20
355
356     difflistpi = []
357     for i in range(num_test):
358         decide = rd.randint(0, 1)
359         if decide == 0:
360             data, true_components = give_data_with_mixes_distribs_continuous()
361             MCMC_pi = EM_MCMC(data, true_components, iter)
362             EM_pi = EM(data, true_components, iter)
363         if decide == 1:
364             data, true_components = give_data_with_mixes_distribs_discrete()
365             MCMC_pi = EM_MCMC(data, true_components, iter)
366             EM_pi = EM(data, true_components, iter)
367         #differences in pi
368         diff_pi = [abs(MCMC_pi[i] - EM_pi[i]) for i in range(len(MCMC_pi))]
369         #list of differences
370         difflistpi.append(sum(diff_pi))
371         print([round(diff, 6) for diff in diff_pi])
372
373     plt.scatter(range(len(difflistpi)), difflistpi, label='')
374     plt.xlabel("Iteration")
375     plt.ylabel("Difference in pi values")
376     plt.legend()
377     plt.show()
378
379 do_test()

```

This gave us a graph:

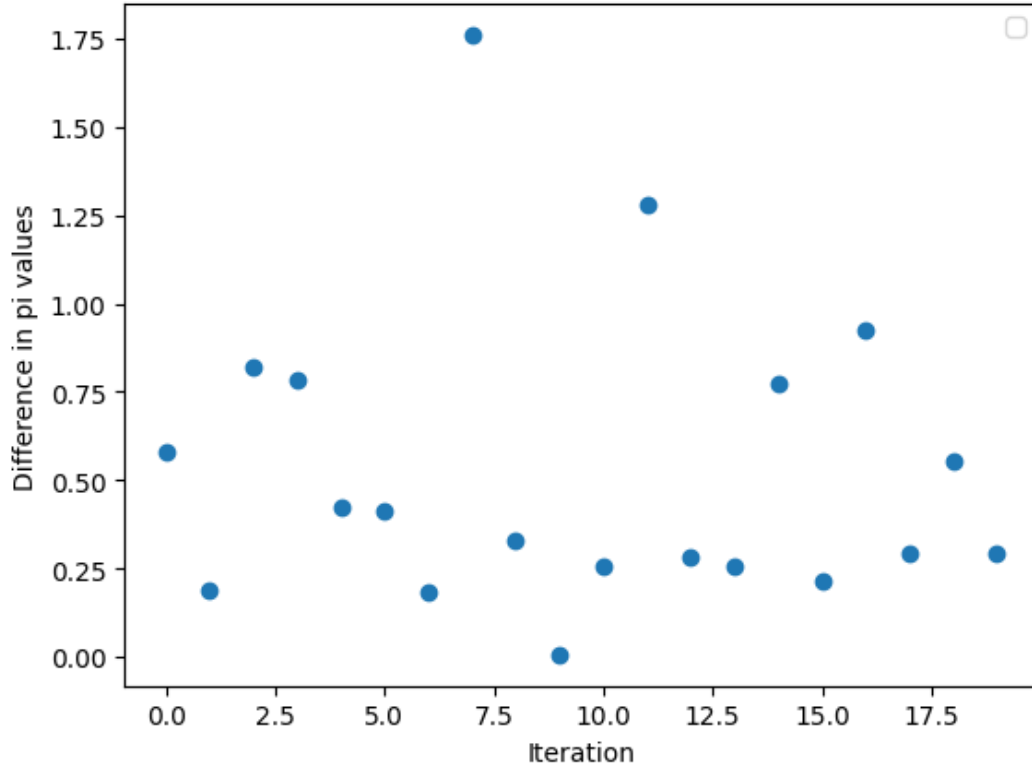


Figure 3: Total difference of mixing weights for each iteration.

This code changed each of the EM and EMMC implementations from user based input to computer based, in which the inputs are randomly selected for each mix. We also make a new function called `give_data_with_distributions` (both continuous and discrete cases). This randomly select number of mixes, what each distribution is (depending on if its continuous function or discrete), and create the data for our random selection.

This implementation uses the fact that we know what distributions are being generated. It also utilizes the random function for multiple decisions - if we are going discrete or if not (decided to separate because of the experiment that happened earlier when combining the 2 together), the amount of mixes we are doing for a sample, etc.

ChatGPT was used here for cleanup of the code in the previous segments, in which I had to argue for it to not do too much to the code.

The graph shows the absolute differences in our finished mixing weights for each mixing we randomly selected. We see that the differences are quite random, which is what we should expect - the graph doesn't serve much meaning, only for a look into seeing what is happening under the hood more.

4 Conclusion

From our brief analysis of these 2 method for Expectation Maximization, I've found that:

- The Monte Carlo method requires more computation and has higher complexity than that of the Normal EM. This was mainly seen in when we first implemented each method

- That being said, Monte Carlo gave surprisingly similar results, making it a viable alternative for EM. This was emphasized in our last implementation in which we made random mixes.

Overall, the analysis showed that the MC method was not as bad as expected, even though it is an approximation of Normal EM.

AI Use

Model(s): ChatGPT 4o
Date of Use: July 18th-22nd

Prompts Entered:

- “Computing MLE for a mixture of distributions from data”
- “What is this log likelihood telling us for this iteration”
- “Why would the sum be 0 if the data fit perfectly”
- “Give me a good dataset to use this on”
- “This doesn’t seem to give me accurate results”
- “`lambda x, mu=mu, sigma=sigma`: what does this line mean”
- “Are you talking about it being `(count_i + 1e-6) / total`? for 4.”
- “Is there possibly other problems in my code”
- “Is there a trendline function for `plt`”
- “`coeffs = np.polyfit(range(len(ll_list)), ll_list, deg=3)`
`trendline = np.poly(coeffs)`
Does this work?”
- “Component 1: Poisson, count = 236
Component 2: Exponential, count = 17088
...
Final parameters:
Component 1: Normal
Parameters: 2.0323434577749007, 0.5859832865135821
Weight: 0.4252
...
What can you say about this information and the problems with the mixtures”
- “How would I change my Poisson initialization”
- “It’s better, but still seeing problems with Poisson”
- “What is the formula for responsibilities”
- “If we have data that is from a binomial distribution that we don’t know what p is, how can we estimate p ”
- “How would we update parameters in binomial”
- “... Iteration 26: Log-likelihood = -1088.03...
Why does the output do this?”
- “Clean up this code and make it look nice, and provide comments and help to all the functions”
- “You removed the try and except blocks. Doesn’t that cause issues if I pick something bad?”
- “Clean up this code and make it look nice, and provide comments and help to all the functions. I told you not to remove anything and you removed the try and except — keep it in and still make everything look pretty and have help for functions”

- “Now comment on this code and clean it up”
- “Convert this such that all the user inputs are done by the computer for a given answer”
- “How do classes work”
- “Can you change the name of the functions such that none of them have the same name — you’ll also have to look through the code for old function names and change them”
- “Am I missing anything? I want to make random samples of random mixes of either discrete or continuous distributions and then test the differences in π with EM and EM MCMC methods”
- “What are all the steps associated with expectation maximization”
- “The M-step: We update the mixing weights by the average of the probability of data point x coming from distribution j for all x in the data set.
Is this worded right?”
- “Can you put this in LaTeX”