Missing Data and Expectation-Maximization (EM) Algorithm

Alyanur Al Azani, Alexander Chernikov, Bhavini Kadiwala, Jose Lopez, Darya Petrov

June 2022

Abstract: The aim of this project is to research the Expectation-Maximization algorithm and learn how it can be used to estimate the maximum likelihood estimate (MLE) of unknown parameters in a statistical model when the data contains missing values. The approach iterates through the estimation (E) step, which computes a function for a lower bound of the likelihood of the observed data, and is followed by the maximization (M) step, which maximizes that lower bound. model parameters. We implement the EM algorithm from scratch. A simulation study shows that the EM algorithm is able to the estimate the unknown parameters, but doesn't perform well when clusters of data are overlapping. Then, we show that the EM algorithm had an accuracy of 99.6% on the "Mouse" dataset, which is a two-dimensional with 490 observations and three gaussian clusters. Lastly, we show that the results and run-time of our EM algorithm from scratch was similar to that of the built-in function *GaussianMixture* from *sklearn.mixture.*

1 Introduction

The Expectation-Maximization Algorithm is an unsupervised, iterative algorithm that finds the maximum likelihood estimate of unknown parameters when there are latent (i.e. unobserved) variables in data [1]. The EM algorithm allows us to solve such non-convex problems that commonly arise from non supervised learning since the latent variables inhibit the computation of the log likelihood. The algorithm iterates between two steps: the Expectation (E-step) and Maximization (M-step). The E-Step estimates the missing variables by creating a heuristic of the distribution of the data with a lower bound on the log likelihood using the current estimates of the parameters. The M-Step optimizes the parameters of the model by computing the parameters that maximizing the E-step's expected log likelihood. These algorithm requires some initial values, and the two steps are iterated alternatively as the M-step is used for the next step E until the values converge [2]. The EM algorithm has many real life applications in machine learning that include Natural Language Processing, computer vision and quantitative analysis of genetics, and image reconstruction for structural engineering [3]. The focus of our project is to explore how the EM algorithm solves the problem of measuring the maximum likelihood estimates of a statistical model for the conditions when the latent variables are involved and the data is missing or incomplete by preforming a simulation study, implementing the algorithm by scratch (in Python), and comparing our results and the run-time to the built-in function *GaussianMixture* from *sklearn.mixture* which estimates the parameters of a Gaussian mixture distribution using the EM algorithm. The data used to preform these tasks is the "Mouse" data set. It is a 2 dimensional data set that contains 3 Gaussian clusters and 490 samples of coordinates that are labeled as "Head", "Ear left" or "Ear Right".

2 Proposed Methods

Suppose you have data $\tilde{x_1}, ..., \tilde{x_n} \stackrel{iid}{\sim} p(\theta)$ where $\tilde{\mathbf{x}}_i = (\mathbf{x}_i, Z_i)$ with $\mathbf{x}_i$ being the observed data and Z_i being the missing data. Suppose $Z_i \in \{1, 2, ..., K\}$, such as a cluster label.

$$
\begin{aligned}
l(\theta) &= \log \prod_{i=1}^{n} p(\tilde{\mathbf{x}}_i) \\
&= \sum_{i=1}^{n} \log p(\tilde{\mathbf{x}}_i) \qquad (1)
\end{aligned}
$$

Since Z_i and the distribution of Z_i is unknown, computing the maximum likelihood estimate of θ using equation 1 becomes a non-convex problem, and thus intractable.

2.1 Expectation Maximization Algorithm

The following section is based on a Non-Convex Optimization lecture by Dr. Balasubramanian at UC Davis [4].

The expectation maximization (EM) algorithm is an iterative algorithm that solves such non-convex problems that occur as a result of missing data. In each round t, the EM algorithm maximizes the lower bound on the likelihood $l(\theta)$, based on the current guess $\boldsymbol{\theta}^{(t)}$. Repeatedly constructing these bounds and maximizing them eventually leads to convergence to a local maximum. The EM algorithm is based on maximizing the following bound on the likelihood of the observed data:

$$
\begin{aligned}
l(\theta) &= \sum_{i=1}^{n} \log p_{\boldsymbol{\theta}}(\mathbf{x}_i) \qquad (2) \\
&= \sum_{i=1}^{n} \sum_{Z_i} (q_i(Z_i) \log p_{\boldsymbol{\theta}}(\mathbf{x}_i, Z_i) - q_i(Z_i) \log q_i(Z_i)) \qquad (3)
\end{aligned}
$$

where q_i are nonzero distributions. Note that $q_i(Z_i)$ and $\log q_i(Z_i)$ do not depend on $\boldsymbol{\theta}$, so the term $q_i(Z_i)\log q_i(Z_i)$ can be ignored for purposes of maximizing over $\boldsymbol{\theta}$. So we are maximizing the following with respect to $\boldsymbol{\theta}$:

$$
\sum_{i=1}^{n} \sum_{Z_i} q_i(Z_i) \log p_{\boldsymbol{\theta}}(\mathbf{x}_i, Z_i) \qquad (4)
$$

Let

$$
\begin{aligned}
q_i(Z_i) &= \frac{p_{\boldsymbol{\theta}'}(\mathbf{x}_i, Z_i)}{\sum_{Z_i} p_{\boldsymbol{\theta}'}(\mathbf{x}_i, Z_i)} \qquad (5) \\
&= p_{\boldsymbol{\theta}'}(Z_i|\mathbf{x}_i) \qquad (6)
\end{aligned}
$$

The EM Algorithm repeats the following steps:

Step 1 (E Step): Compute $p_{\boldsymbol{\theta}^{(t)}}(Z_i|\mathbf{x}_i)$ and the lower bound on the observed likelihood

$$
Q(\boldsymbol{\theta}, \boldsymbol{\theta}^{(t)}) = \sum_{i=1}^{n} \sum_{Z_i} p_{\boldsymbol{\theta}^{(t)}}(Z_i|\mathbf{x}_i) \log p_{\boldsymbol{\theta}}(\mathbf{x}_i, Z_i) \qquad (7)
$$

Step 2 (M Step): Maximize the lower bound to update new value $\boldsymbol{\theta}^{(t+1)}$

$$
\boldsymbol{\theta}^{(t+1)} = \underset{\boldsymbol{\theta}}{\operatorname{argmax}} Q(\boldsymbol{\theta}, \boldsymbol{\theta}^{(t)}) \qquad (8)
$$

Figure 1 below illustrates the convergence of the EM algorithm. The aim of the EM algorithm is to maximize the log-likelihood of the observed data. The E step constructs the function Q_t and the M step find the $\theta^{(t+1)}$ that maximizes Q_t [5]. The algorithm is guaranteed local convergence because with every iteration, the likelihood is monotone increasing.

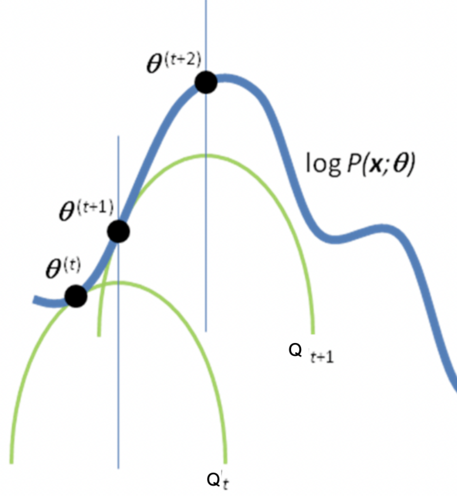

Figure 1: Convergence of the EM algorithm [5]

2.2 EM Algorithm for Mixture of Gaussians

The Gaussian mixture model assumes the following:

$$\mathbf{x} \sim p_{\boldsymbol{\theta}}(\mathbf{x}) = \sum_{k=1}^{K} p_{\boldsymbol{\theta}}(Z=k) p_{\boldsymbol{\theta}}(\mathbf{x}|Z=k) = \sum_{j=1}^{K} \pi_k N(\mathbf{x}; \boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k) \tag{9}$$

where $\pi_k = p_{\boldsymbol{\theta}}(Z=k)$, Z is the latent (hidden) variable, X is normally distributed with mean $\boldsymbol{\mu}$ and covariance $\boldsymbol{\Sigma}$, and the unknown parameter $\boldsymbol{\theta} = \{\boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k, \pi_k\}_{k=1}^{K}$. The goal is to estimate $\boldsymbol{\theta}$ given the N samples $\mathbf{x}_1, ..., \mathbf{x}_N$ with the above model 9. We can assign each sample to one of the K Gaussian clusters, after obtaining an estimate for $\boldsymbol{\theta}$.

The log-likelihood is:

$$\begin{aligned} l(\theta) &= \sum_{n=1}^{N} \log\, p_{\boldsymbol{\theta}}(\mathbf{x}_n) && (10) \\ &= \sum_{n=1}^{N} \log\, \sum_{k=1}^{K} \pi_k N(\mathbf{x}; \boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k) && (11) \end{aligned}$$

Let

$$\mathbf{F}_{nk}^{(t)} = p_{\boldsymbol{\theta}^{(t)}}(Z_n = k|\mathbf{x}_n) = \frac{p_{\boldsymbol{\theta}^{(t)}}(\mathbf{x}_n, Z_n = k)}{\sum_{Z_n} p_{\boldsymbol{\theta}^{(t)}}(\mathbf{x}_n, Z_n = k')} = \frac{N(\mathbf{x}; \boldsymbol{\mu}_k^{(t)}, \boldsymbol{\Sigma}_k^{(t)}) \pi_k^{(t)}}{\sum_{k'=1}^{K} N(\mathbf{x}; \boldsymbol{\mu}_{k'}^{(t)}, \boldsymbol{\Sigma}_{k'}^{(t)}) \pi_{k'}^{(t)}} \tag{12}$$

The above equation for $\mathbf{F}_{nk}^{(t)}$ follows from equation 5.

The steps of the EM algorithm are:

E Step:

$$\begin{aligned} Q(\boldsymbol{\theta}, \boldsymbol{\theta}^{(t)}) &= Q((\pi, \boldsymbol{\mu}, \boldsymbol{\Sigma}), (\pi^{(t)}, \boldsymbol{\mu}^{(t)}, \boldsymbol{\Sigma}^{(t)})) \quad (13) \\ &= \sum_{n=1}^{N} \sum_{k=1}^{K} \mathbf{F}_{nk}^{(t)} log \pi_k N(\mathbf{x}; \boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k) \quad (14) \\ & \quad (15) \end{aligned}$$

which follows from equation 7.

M Step:

$$\boldsymbol{\theta}^{(t+1)} = (\pi^{(t+1)}, \boldsymbol{\mu}^{(t+1)}, \boldsymbol{\Sigma}^{(t+1)}) = \text{argmax}_{\pi, \boldsymbol{\mu}, \boldsymbol{\Sigma}} Q((\pi, \boldsymbol{\mu}, \boldsymbol{\Sigma}), (\pi^{(t)}, \boldsymbol{\mu}^{(t)}, \boldsymbol{\Sigma}^{(t)})) \quad (16)$$

which follows from equation 8.

The following is the closed form solution to the above maximization problem:

$$\pi^{(t+1)} = \sum_{n=1}^{N} \mathbf{F}_{nk}^{(t)} / \sum_{n=1}^{N} \sum_{k'=1}^{K} \mathbf{F}_{nk'} = \sum_{n=1}^{N} \mathbf{F}_{nk}^{(t)} / N \quad (17)$$

$$\boldsymbol{\mu}_k^{(t+1)} = \sum_{n=1}^{N} \mathbf{F}_{nk}^{(t)} \mathbf{x}_n / \sum_{n=1}^{N} \mathbf{F}_{nk}^{(t)} \quad (18)$$

$$\boldsymbol{\Sigma}_k^{(t+1)} = \sum_{n=1}^{N} \mathbf{F}_{nk}^{(t)} (\mathbf{x}_n - \boldsymbol{\mu}_k^{(t+1)})(\mathbf{x}_n - \boldsymbol{\mu}_k^{(t+1)})^T / \sum_{n=1}^{N} \mathbf{F}_{nk}^{(t)} \quad (19)$$

3 Simulation Study

The purpose of conducting a simulation study is to evaluate the performance of our EM algorithm which was created from the ground up on Python. We are interested in the results of the algorithm when presented with randomly generated unlabeled data clusters. Our simulation study consists of three randomly generated normal distributions, each with unique means and covariances. We utilized numpy's *random.normal* function which returns drawn samples from the parameterized normal distribution, to produce the means and covariances of each cluster. These were then inputted into numpy's *multivariate_normal* function to generate data from the Gaussian distribution for each cluster. The EM algorithm was executed on the randomly generated data, returning the likelihoods, scores, and assigned clusters.

The clustering of the EM algorithm is visualized below in Figure 2 along with the original clusters of the generated data. For the most part, it appears that the EM clustering algorithm performed well, with a few points were misclassified in the area where the clusters are overlapping. Note the cluster numbers are different. This is a consequence of clustering algorithms, but we ensured the clusters had the same color for better visualization purposes. We achieved the color coordination by finding the norm of the means for each original and generated distribution of each cluster. After finding the norms we rank them by their size in order to (hopefully) have their ranks synced up. This is assuming the generated data and the EM parameters are close enough to have close to matching mean norm ranks. An improvement upon implementation of the color syncronization might come in the form of either comparing the ranks of individual coordinates norms, or some other way to sort similar distributions.

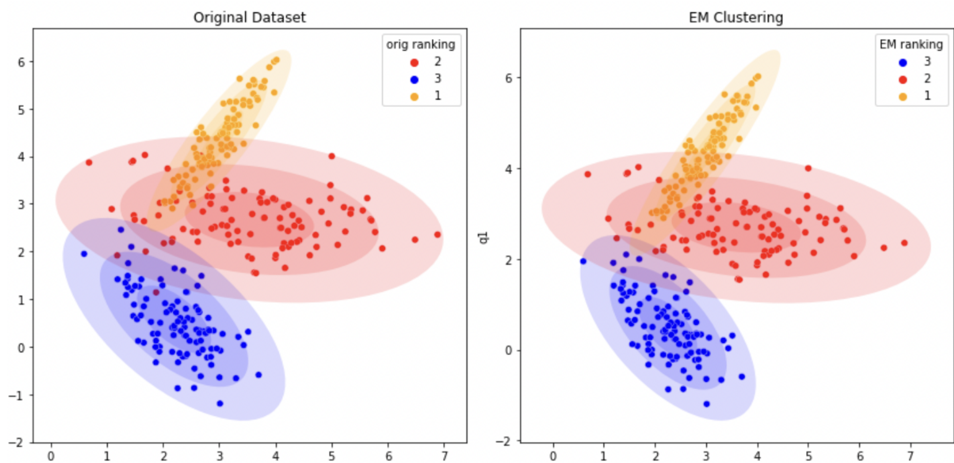

Figure 2: Original clusters (left) and clustering result from EM (right) in the simulation study

A confusion matrix (shown below in Figure 3) is used to further evaluate the performance of EM clustering. The calculated accuracy score of the algorithm is 96.6%, indicating that the model's accuracy drops when distributions overlap. There was a total of 10 misclassifed data points. The most difficult cluster for the EM algorithm was the red cluster, since it overlapped with both the yellow and blue cluster.

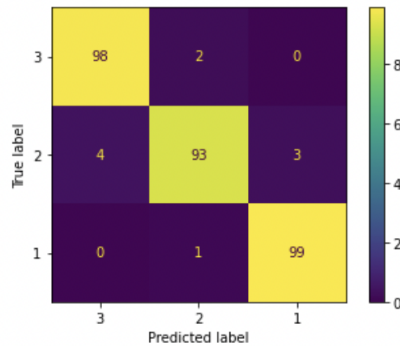

Figure 3: Confusion matrix for EM clustering results in the simulation study

A plot of the log-likelihood over time/iteration was produced to observe the convergence of the algorithm. The first log-likelihood was excluded to better visualize the convergence of the algorithm (the first log-likelihood is based on EM's initial guess, which throws off the scaling of the log-likelihood plot). As seen below, the algorithm appeared to converge after approximately 17 iterations.

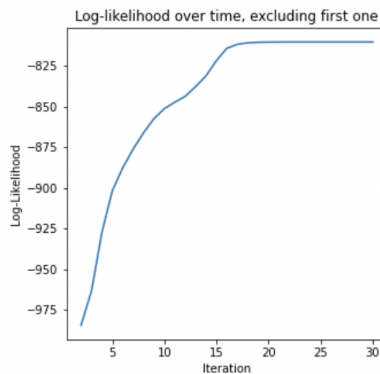

Figure 4: Log-Likelihood for every EM iteration in the simulation study

4 Real Data Analysis

4.1 Applying EM Algorithm to the "Mouse" Dataset

We perform the EM algorithm both by scratch and using built-in functions such as GaussianMixture from sklearn.mixture. The purpose of building the EM algorithm from scratch and using a built-in function allows us to gather insight into how the algorithm works in practice, and compare the run-time of our from scratch algorithm to the built-in to verify we optimized computation time in our code well. We used Carrasco's implementation of the EM algorithm from scratch as a starting point [6]. The visualization below depicts the three clusters within the "Mouse" dataset: "Head", "Left ear" and "right ear". To perform the EM algorithm, we remove the labels of the clusters before training the model on the dataset. The assumption made in our model is that there are 3 gaussian clusters.

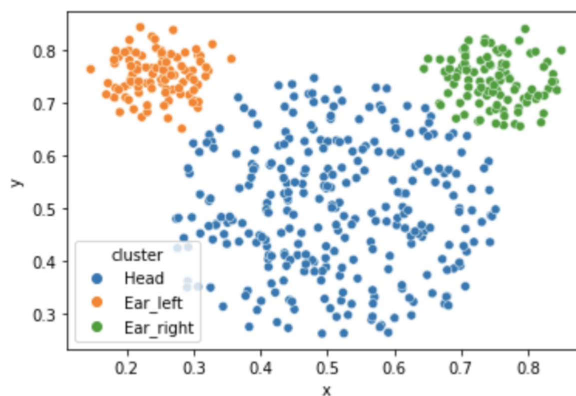

Figure 5: "Mouse" dataset

The trained EM algorithm was executed on the data, returning the likelihoods, scores, and assigned clusters. A plot of the log-likelihood over time/iteration was produced to observe the convergence of the algorithm. It can be seen below in Figure 6 that the algorithm converges after about 23 iterations. The execution time for the training of our model took 0.03 seconds [7].

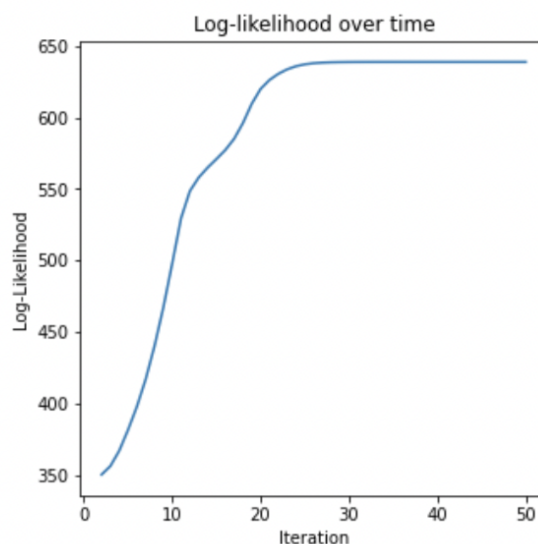

Figure 6: Log-Likelihood for every EM iteration on the "Mouse" dataset

The scatterplots of the clustering from the EM algorithm are visualized below in Figure 7 along with the original data clusters from the "mouse" dataset.

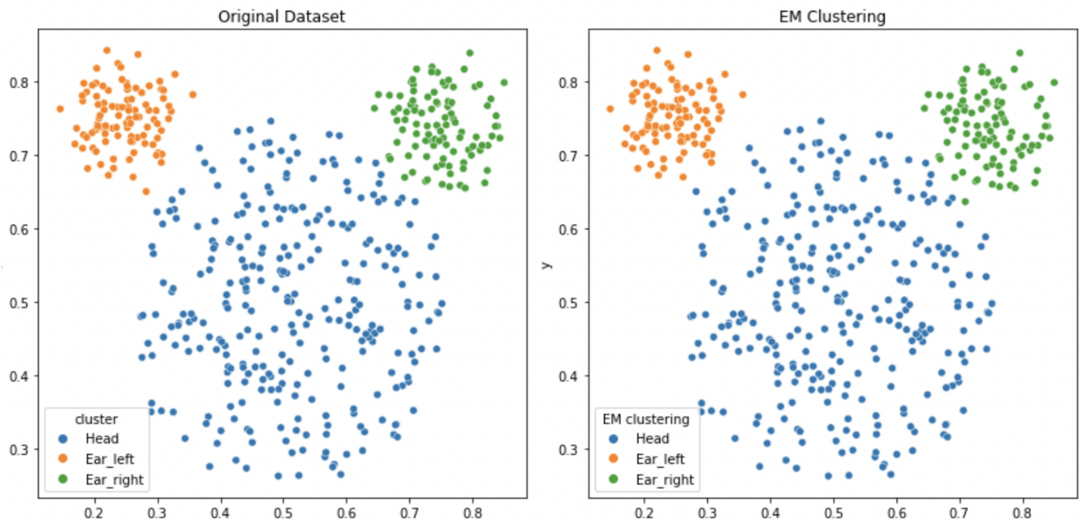

Figure 7: Original clusters (left) and clustering result from EM (right) on the "Mouse" dataset

Figure 8 below shows the gaussian distributions with the estimated maximum likelihood estimates of the unknown parameters. We can see the variance of the yellow and green clusters are much smaller than the variance of the blue cluster.

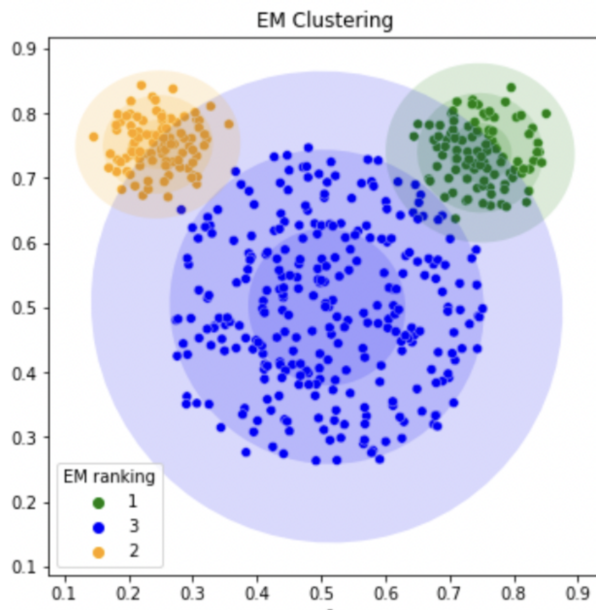

Figure 8: EM result on the "Mouse" dataset for the gaussian distributions for each cluster

A contingency table (shown below in Figure 9) is produced to further evaluate the performance of the EM clustering. The labeling over the clusters were highly accurate as the only mislabeled values was a value from “Head” was mislabeled as “Ear right” and one value from “Ear left” was mislabeled as “Head.” The calculated accuracy score of the algorithm on the "mouse" dataset using our method from scratch is 99.6 percent, which is highly accurate.

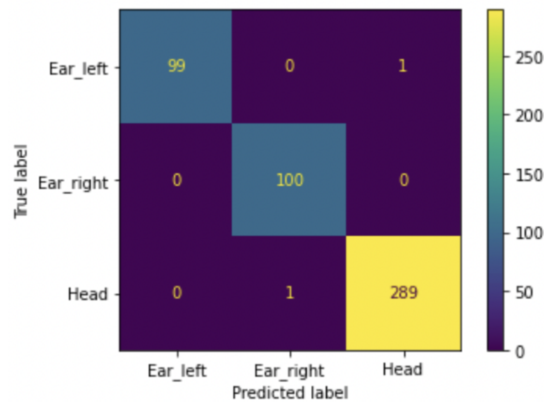

Figure 9: Confusion matrix for EM clustering results on the "Mouse" dataset

4.2 Gaussian Mixture Modelling (GMM) using sklearn

To compare the results found from building the EM algorithm from scratch, we utilize the built-in function GaussianMixture from sklearn.mixture. The GMM model assumes the underlying data is generated from Mixture of Gaussians, which was done by using the GaussianMixture function in Python. We predict the labels of the "mouse" dataset and plotted the scatterplots of the original dataset in comparison to the Sklearn EM Clustering algorithm. The clustering in the original appears similar to the clustering by sklearn. The clustering results from sklearn and our method from scratch is identical. Both methods show that the model assumptions of a Gaussian Mixture Model hold up in our method by scratch and using the built-in function. The run-time of the built-in function was about .002 seconds.

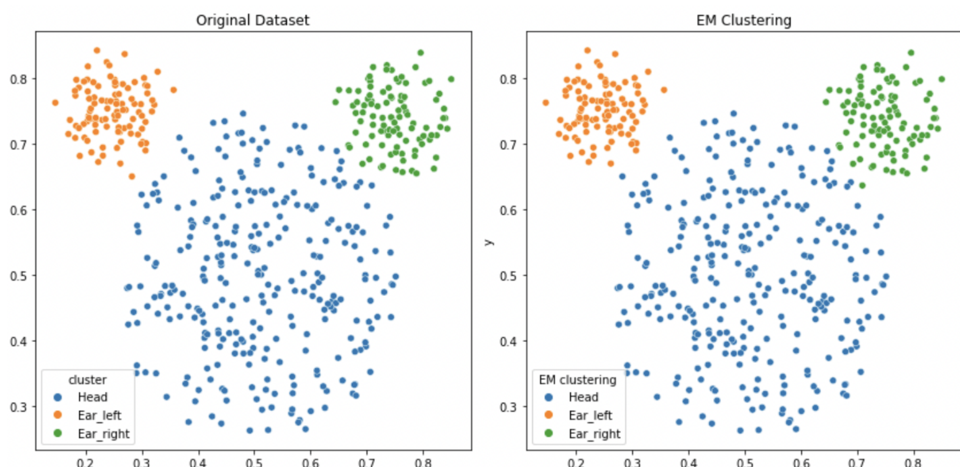

Figure 10: Original clusters (left) and clustering result from sklearn's GaussianMixture function (right) on the "Mouse" dataset

A contingency table (shown below in Figure 11) is produced to further evaluate the performance of EM clustering. The labeling over the clusters was highly accurate as the only mislabeled values was a value from "Head" was mislabeled as "Ear right" and one value from "Ear left" was mislabeled as "Head." The calculated accuracy score of the algorithm on the "mouse" dataset using a method from scratch is 99.6 percent, which is highly accurate. When comparing these results to the algorithm by scratch it can be noted that the values are exactly the same, thus revealing that our algorithm produces the same results as the built-in functions.

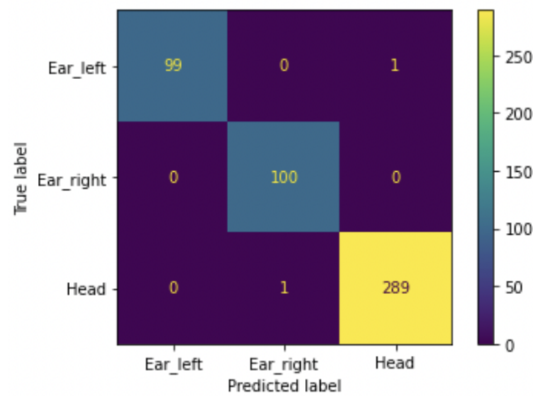

Figure 11: Confusion matrix for sklearn's GaussianMixture function clustering results on the "Mouse" dataset

5 Conclusion

Based on the results, we evaluated that our EM algorithm from scratch was computationally efficient (with a run-time of about .03 seconds), but not quite as efficient as the built-in function GaussianMixture (with a run-time of about .002 seconds). Both the from scratch and built-in EM algorithm had an accuracy of 99.6 percent. The clustering results from our from scratch algorithm to the built-in function were identical. Its important to note these results are specific to the "Mouse" dataset, and the performance of the EM algorithm may be worse with a different dataset. For example, as we noticed in the simulation study, the EM algorithm struggles to differentiate overlapping clusters. Overall, we successfully accomplished all aspects of our project, including learning the theory behind the EM algorithm, determining the capability performance of the Expectation-Maximization algorithm with a simulation study, implementing the algorithm in Python from scratch and using a built-in function, and applying it to the "Mouse" dataset.

6 References

[1] Wikimedia Foundation. (2022, May 24). Expectation–maximization algorithm. Wikipedia. Retrieved from https://en.wikipedia.org/wiki/Expectation%E2%80%93maximization_algorithm

[2] Brownlee, J. (2019, November 1). A Gentle Introduction to Expectation-Maximization (EM algorithm). Machine Learning Mastery. Retrieved from https://machinelearningmastery.com/expectation-maximization-em-algorithm/

[3] Tyagi, N. (2021, March 28). Expectation-Maximization (EM) Algorithm in Machine Learning. Retrieved from https://www.analyticssteps.com/blogs/expectation-maximization-em-algorithm-machine-learning

[4] Balasubramanian, K. (2020, May). Non-Convex Optimization. Lecture, University of California, Davis.

[5] Duke University. (n.d.). Convergence of the Em algorithm. Expectation Maximization (EM) Algorithm. Retrieved from https://people.duke.edu/ ccc14/sta-663/EMAlgorithm.html.

[6] Carrasco, O. C. (2019, June 2). Gaussian Mixture Models Explained. Medium. Retrieved from https://towardsdatascience.com/gaussian-mixture-models-explained-6986aaf5a95

[7] Biarnes, A. (2020, September 11). Gaussian Mixture Models and Expectation-Maximization (A full explanation). Medium. Retrieved from https://towardsdatascience.com/gaussian-mixture-models-and-expectation-maximization-a-full-explanation-50fa94111ddd

7 Appendix

7.1 Proof of local convergence of the EM algorithm

$$\begin{aligned} l(\theta^{(t+1)}) &\geq \sum_{i=1}^{n} \sum_{Z_i} p_{\boldsymbol{\theta}^{(t)}}(Z_i|\mathbf{x}_i) \log\, p_{\boldsymbol{\theta}^{(t+1)}}(\mathbf{x}_i, Z_i) && (20) \\ &\geq \sum_{i=1}^{n} \sum_{Z_i} p_{\boldsymbol{\theta}^{(t)}}(Z_i|\mathbf{x}_i) \log\, p_{\boldsymbol{\theta}^{(t)}}(\mathbf{x}_i, Z_i) && (21) \\ &= l(\theta^{(t)}) && (22) \end{aligned}$$

7.2 Derivation of equation 1

$$\begin{aligned} l(\theta) &= \sum_{i=1}^{n} \log\, p_{\boldsymbol{\theta}}(\mathbf{x}_i) && (23) \\ &= \sum_{i=1}^{n} \log \sum_{Z_i} p_{\boldsymbol{\theta}}(\mathbf{x}_i, Z_i) && (24) \\ &= \sum_{i=1}^{n} \log \sum_{Z_i} q_i(Z_i) \frac{p_{\boldsymbol{\theta}}(\mathbf{x}_i, Z_i)}{q_i(Z_i)} && (25) \\ &= \sum_{i=1}^{n} \log\, E_{q_i} \left(\frac{p_{\boldsymbol{\theta}}(\mathbf{x}_i, Z_i)}{q_i(Z_i)} \right) && (26) \\ &\geq \sum_{i=1}^{n} E_{q_i} \left(\log\, \frac{p_{\boldsymbol{\theta}}(\mathbf{x}_i, Z_i)}{q_i(Z_i)} \right) && (27) \\ &= \sum_{i=1}^{n} \sum_{Z_i} q_i(Z_i) \log\, \frac{p_{\boldsymbol{\theta}}(\mathbf{x}_i, Z_i)}{q_i(Z_i)} && (28) \\ &= \sum_{i=1}^{n} \sum_{Z_i} \left(q_i(Z_i) \log\, p_{\boldsymbol{\theta}}(\mathbf{x}_i, Z_i) - q_i(Z_i) \log\, q_i(Z_i)\right) && (29) \end{aligned}$$

where q_i are nonzero distributions. Step (23)-(24) applies the law of total probability. Step (25)-(26) applies the definition of expectation. Step (26)-(27) applies Jensen's inequality to the concave function f(x)=log(x). Step (27)-(28) applies the definition of expectation again. Lastly, step (28)-(29) simplifies the logarithm.

7.3 Derivation of equation 6

$$
\begin{aligned}
q_i(Z_i) &= \frac{p_{\boldsymbol{\theta}'}(\mathbf{x}_i, Z_i)}{\sum_{Z_i} p_{\boldsymbol{\theta}'}(\mathbf{x}_i, Z_i)} \\
&= \frac{p_{\boldsymbol{\theta}'}(Z_i|\mathbf{x}_i)p_{\boldsymbol{\theta}'}(\mathbf{x}_i)}{\sum_{Z_i} p_{\boldsymbol{\theta}'}(Z_i|\mathbf{x}_i)p_{\boldsymbol{\theta}'}(\mathbf{x}_i)} \\
&= \frac{p_{\boldsymbol{\theta}'}(Z_i|\mathbf{x}_i)p_{\boldsymbol{\theta}'}(\mathbf{x}_i)}{p_{\boldsymbol{\theta}'}(\mathbf{x}_i)\sum_{Z_i} p_{\boldsymbol{\theta}'}(Z_i|\mathbf{x}_i)} \\
&= \frac{p_{\boldsymbol{\theta}'}(Z_i|\mathbf{x}_i)}{\sum_{Z_i} p_{\boldsymbol{\theta}'}(Z_i|\mathbf{x}_i)} \\
&= p_{\boldsymbol{\theta}'}(Z_i|\mathbf{x}_i)
\end{aligned}
$$

since $\sum_{Z_i} p_{\boldsymbol{\theta}'}(Z_i|\mathbf{x}_i)=1$.

7.4 Derivation of equation 11

$$
\begin{aligned}
l(\theta) &= \sum_{n=1}^{N} \log\, p_{\boldsymbol{\theta}}(\mathbf{x}_n) \\
&= \sum_{n=1}^{N} \log \sum_{Z_n} p_{\boldsymbol{\theta}}(\mathbf{x}_n, Z_n) \\
&= \sum_{n=1}^{N} \log \sum_{k=1}^{K} p_{\boldsymbol{\theta}}(\mathbf{x}_n, Z_n = k) \\
&= \sum_{n=1}^{N} \log \sum_{k=1}^{K} \pi_k N(\mathbf{x}; \boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k)
\end{aligned}
$$

7.5 Python Code

There were not any inverses that needed to be calculated in the EM algorithm that we could try to optimize, however, we did try to optimize our code by:

1. Saving computations in variables which were reused frequently (especially in the M step)

2. Using matrix multiplication when calculating $\Sigma_k^{(t+1)}$ in the M step, instead of a for loop to calculate the sum.

Unfortunately because of the nature of the EM algorithm, a for loop was unavoidable to execute all iterations of the EM algorithm. We also could not utilize parallel computing because EM is an iterative algorithm, so each iteration relies on the previous iteration.

```
import os
import pandas as pd
import numpy as np
from scipy import stats
import matplotlib.pyplot as plt
from scipy.special import logsumexp
import imageio
import matplotlib.animation as ani
```

```
import matplotlib.cm as cmx
import matplotlib.colors as colors
import matplotlib.pyplot as plt
import numpy as np
import seaborn as sns
from matplotlib.patches import Ellipse
from PIL import Image
from sklearn import datasets
from sklearn.cluster import KMeans
from scipy.stats import multivariate_normal
import sklearn
from sklearn.metrics import confusion_matrix
import seaborn as sn
from sklearn.metrics import ConfusionMatrixDisplay
import time
import timeit
```

EM Algorithm

```
def initialize_clusters_d(X, num_clusters=3):

    #randomly

    np.random.seed(1)

    clusters = []
    for i in range(num_clusters):
        clusters.append({
            'pi_k': np.random.uniform(0, 1),
            'mu_k': np.random.uniform(0, 1, size=(2,)),
            #covariance matrix has to be positive semidefinite matrix
            'cov_k': sklearn.datasets.make_spd_matrix(2, random_state=1)
        })

    return clusters
```

Expectation Step

```
#calculcate F_nk
def expectation_step_d(X,N,K, clusters):

    F_nk = np.zeros((N, K), dtype=np.float64)


    #calculcate F_nk for each data point in each cluster
    for k, cluster in enumerate(clusters):
        pi_k = cluster['pi_k']
        mu_k = cluster['mu_k']
        cov_k = cluster['cov_k']

        #numerator
```

```
        F_nk[:, k] = pi_k * multivariate_normal.pdf(X, mean=mu_k, cov=cov_k)

    #denominator (sum values in each of the k rows)
    F_n = np.sum(F_nk, 1).reshape((N,1))

    F_nk = F_nk/F_n

    return F_nk,F_n


Maximization Step

def maximization_step_d(X, N, clusters,F_nk):

    N = float(N)
    clusters_updated = []
    clusters_updated_no_string = []

    for k, cluster in enumerate(clusters):

        #F_nk for cluster k
        F_k = F_nk[:,k].reshape((int(N),1))

        #sum F_k for each of the N samples
        N_k = np.sum(F_k)

        #closed form solution

        pi_k = N_k / N

        #sum is of all N entries in each column
        mu_k = np.sum(F_k * X, axis=0) / N_k

        diff = X - mu_k
        #(X^T)X will do the inner product
        #(hence the sum in the equation)
        cov_k = (F_k * diff).T @ diff / N_k

        #udpate clusters
        clusters_updated.append({
            'pi_k': pi_k,
            'mu_k': mu_k,
            'cov_k': cov_k
        })
#           clusters_updated_no_string.append({
#               pi_k,
#               mu_k,
#               cov_k
#           })

    return clusters_updated


Return likelihood of our parameters
```

```
def get_likelihood_d(X, clusters,F_nk,F_n):
    likelihood = np.sum(np.log(F_n))
    return likelihood


Combine Steps to form EM Algorithm

def train_gmm_d(X, n_clusters, n_epochs):

    #number of samples in the dataset
    N = X.shape[0]

    #randomly initialize values for the algorithm to have a starting point
    clusters = initialize_clusters_d(X,n_clusters)

    #likelihood value we are trying to maximize
    likelihoods = np.zeros((n_epochs, ))

    #score of how likely a data point is to be from a certain cluster
    scores = np.zeros((X.shape[0], n_clusters))

    #for each epoch
    for i in range(n_epochs):


        F_nk,F_n = expectation_step_d(X,N,n_clusters,clusters)
        clusters = maximization_step_d(X, N, clusters, F_nk)

        likelihood = get_likelihood_d(X, clusters,F_nk,F_n)
        likelihoods[i] = likelihood


    #scores = np.log(F_nk)
    scores = F_nk

    return clusters, likelihoods, scores

#-----------------------------------------------------------------------------

Simulation Study with Random generated data set with EM

This code generates k number of random bivariate gausiaan distributions.
We chose bivariate as they are the easiest multivariate distributions to visualize.


import random
## generate random positive semi definite matrix
def get_random_psd(n):
    x = np.random.normal(0, 1, size=(n, n))
    return np.dot(x, x.transpose())

## generate random positive semi definite matrix, but smaller so our distributions
## axes are not so differnt
```

```
def get_random_psd_smaller(n):
    x = np.random.normal(0, 0.5, size=(n, n))
    return np.dot(x, x.transpose())

def gen_data_2d_2(k,N,PrintVars=False):

    #k = number clusters
    #N = num of obs in each cluster
    #PrintVars = whether to print out vars, good for debugging,
        #defualt is not to print
    data=[]
    N_new=0
    N_new_stack=[]
    mean_list=[]
    cov_list=[]
    for i in range(k):
        N_new=N
        random_number = np.random.randint(1, 4, size=(2))
        mean=np.random.normal(0, 1, size=(2,))+random_number

        cov=get_random_psd_smaller(2)
        cov_list.append(cov)
        mean_list.append(mean)
        q0, q1 = np.random.multivariate_normal(mean, cov, N_new).T
        cluster=np.repeat(i, N_new)
        data.append(np.vstack((q0, q1,cluster)).T)
        if PrintVars==True:
            print("mean shift:",random_number, sep=" ")
            print("cov of:",i,"\n",cov, sep=" ")
            print("mean of:",i,"\n",mean, sep=" ")
            print("size:",N_new)
            print("\n")


    #returns data as an array with 2 qoords and a label for the cluster
    #also returns covariance and means as a list for visualization purposes
    data=np.vstack(np.asarray(data))

    return data,cov_list,mean_list
```

Generate data

```
np.random.seed(2029)

#number of members of each cluster
N=100
#number of cluster
k=3
#x is our generated bivariate data, also returns covariance and mean for each
#cluster
x,cov_list,mean_list=gen_data_2d_2(k,N)
plt.figure(figsize=(6,6))
```

```
ax=sns.scatterplot(x[:,0],x[:,1],hue=x[:,2])
ax.set_title("Scatter Plot of Generated Dataset")
ax.legend(title='Clusters')
ax.set_xlabel("first coordinate")
ax.set_ylabel("second coordinate")


Set parameters for GMM EM training

x_unlabeled = x[:,0:2]
X=x_unlabeled
n_clusters = k
n_epochs = 30 #tuning hyperparameter, which will be discussed in below


Train EM GMM model on generated data

start = time.time()
clusters, likelihoods, scores = train_gmm_d(X, n_clusters, n_epochs)
end=time.time()
print("Execution Time:", end-start)


Function to extract cov and mu from generated clusters


#our EM algorithm returns parameters as list of dict, so we extract each element
#of dict and append to a list
#this is done to match the formatting of our data generation function
def get_mu_cov_clusters(clusters):
    mu_k=[]
    cov_k=[]
    length=len(clusters)
    for k in range(length):
        mu_k.append(clusters[k]['mu_k'])
        cov_k.append(clusters[k]['cov_k'])
    return mu_k, cov_k


Get parameters of EM output

mu_gen, cov_gen=get_mu_cov_clusters(clusters)


Tune iterations until convergence.

We found that for 300 points, usually the likelihood would converge
before 200 iterations. For this seed, it was around 20 iterations.


plt.figure(figsize=(5, 5))
plt.title('Log-likelihood over time, excluding first one')
plt.ylabel('Log-Likelihood')
plt.xlabel('Iteration')
```

```
plt.plot(np.arange(2, n_epochs + 1), likelihoods[1:])
plt.show()


Dataframe with EM and Original clusters for each point.


test_df=pd.DataFrame(X)
test_df.columns=['q0','q1']
test_df_scores=pd.DataFrame(scores)
test_df = pd.concat([test_df, test_df_scores], axis=1)
#find which score is closest to 0
test_df['EM clustering'] = test_df.iloc[:,2:].idxmax(axis=1)
test_df["original clusters"]=x[:,2]
test_df


Code below is added to sync colors of clusters:

Color code each distribution

The idea here is to find the norm of the means for each original and generated
distribution of each cluster. After finding the norms we rank them by their
size in order to (HOPEFULLY) have their ranks synced up. This is assuming
the generated data and the EM parameters are close enough to have close to
matching mean norm ranks. An improvement upon implementation of the color
syncronization might come in the form of either comparing the ranks of
individual coordinates norms, or some other way to sort similar distributions.


mean_df=[]
for k in range(len(test_df)):
    l2norm=np.linalg.norm(mu_gen[int(test_df['EM clustering'][k])])
    mean_df.append(l2norm)
test_df["EM mean norm"]=np.asarray(mean_df)


mean_df=[]
for k in range(len(test_df)):
    l2norm=np.linalg.norm(mean_list[int(test_df['original clusters'][k])])
    mean_df.append(l2norm)
test_df["original mean norm"]=np.asarray(mean_df)



#in order to sync colors we find the norms of the original and em means and rank
them by their norms
#we then sync the color to the norm
test_df['orig ranking'] = pd.factorize(-test_df["original mean norm"],
sort=True)[0] + 1
test_df['EM ranking'] = pd.factorize(-test_df["EM mean norm"], sort=True)[0] + 1
test_df
```

```
Function to draw covariance ellipses around on our scatterplots


def drawbands(mean_list,cov_list,colors,axes):
    i=0
    for m, cv, col in zip(mean_list,cov_list,colors):
        if cv.shape == (2, 2):
                U, s, Vt = np.linalg.svd(cv)
                angle = np.degrees(np.arctan2(U[1, 0], U[0, 0]))
                width, height = 2 * np.sqrt(s)
        else:
            angle = 0
            width, height = 2 * np.sqrt(cv)

        # Draw the Ellipse
        for nsig in range(1, 4):
            axes.add_patch(Ellipse(m, nsig * width, nsig * height,
                                   angle, fc = colors[i],alpha = 0.15 ))

        i=i+1



Draw original and generated distributions

#sync colors, will only go up to 5 clusters unless more colors added

colors = ['green','orange','red','blue','yellow',"purple"]

#colors for original
indexlist = np.linalg.norm(mean_list,axis=1)
indexlist2_o =pd.factorize(-indexlist, sort=True)[0]+1
sub_array = np.asarray(colors)[indexlist2_o]
colors2=sub_array.tolist()
colors2

#colors for em
indexlist = np.linalg.norm(mu_gen,axis=1)
indexlist2_e =pd.factorize(-indexlist, sort=True)[0]+1
sub_array = np.asarray(colors)[indexlist2_e]
colors3=sub_array.tolist()
colors3



fig, axes = plt.subplots(nrows=1, ncols=2, figsize=(12, 6))

sns.scatterplot('q0','q1', data=test_df, hue="orig␣ranking",hue_order=indexlist2_o,
palette= colors2,ax=axes[0])
drawbands(mean_list,cov_list,colors2, axes[0])
axes[0].set_title("Original␣Dataset")

sns.scatterplot('q0','q1', data=test_df, hue='EM␣ranking',hue_order=indexlist2_e,
palette= colors3,ax=axes[1])
```

```
drawbands(mu_gen, cov_gen,colors3, axes[1])
axes[1].set_title("EM Clustering")
fig.tight_layout()
#Em ranking corresponds to the cluster,but also help with syncing colors


#sometimes labels will differ, so contingency table may need the labels to be
#adjusted.
conf_table = confusion_matrix(test_df["orig_ranking"],test_df['EM_ranking'])
disp = ConfusionMatrixDisplay(confusion_matrix=conf_table,
display_labels=indexlist2_e)
disp.plot()


num_correct = np.sum(np.diag(conf_table))
total = np.sum(conf_table)
accuracy = (num_correct/total)*100
accuracy

#----------------------------------------------------------------------------

Apply EM to Mouse Data

Load Data

#set directory on your device
os.chdir("D:\\Documents\\STA141C\\Final")
#extract data from csv into dataframe,
mouse_df = pd.read_csv("mouse2.csv", header=None,comment='#', sep=" ",)
mouse_df.columns=['x','y','cluster']
mouse_df



def initialize_clusters_d(X,num_clusters=3):

    #randomly

    np.random.seed(1)

    clusters = []
    for i in range(num_clusters):
        clusters.append({
            'pi_k': np.random.uniform(0, 1),
            'mu_k': np.random.uniform(0, 1, size=(2,)),
            #covariance matrix has to be positive semidefinite matrix
            'cov_k': sklearn.datasets.make_spd_matrix(2, random_state=1)
        })

    return clusters


sns.scatterplot('x','y', data=mouse_df, hue='cluster')
```

```
#turn the supervised data into unsupervised (i.e. remove the cluster labels )
mouse_df_unsupervised=mouse_df.iloc[:,0:2]
x_unlabeled = mouse_df_unsupervised.values
X=x_unlabeled

n_clusters = 3
n_epochs = 50

start = time.time()
clusters, likelihoods, scores = train_gmm_d(X, n_clusters, n_epochs=50)
end=time.time()
print("Execution Time:", end-start)


likelihoods[1:].shape

plt.figure(figsize=(5, 5))
plt.title('Log-likelihood over time')
plt.ylabel('Log-Likelihood')
plt.xlabel('Iteration')
plt.plot(np.arange(2, n_epochs + 1), likelihoods[1:])
plt.show()


The EM algorithm converges after about 20 iterations.

mouse_df['1'],mouse_df['2'],mouse_df['3'] = scores.T
#find which score is closest to 0
mouse_df['EM clustering'] = mouse_df.iloc[:,3:6].idxmax(axis=1)
mouse_df

mouse_df["EM clustering"] = mouse_df["EM clustering"].replace(["2","1","3"],
['Head',"Ear_right","Ear_left"])

fig, axes = plt.subplots(nrows=1, ncols=2, figsize=(12, 6))
hue_order=['Head',"Ear_left","Ear_right"]
sns.scatterplot('x','y', data=mouse_df, hue='cluster',ax=axes[0],
hue_order=hue_order)
axes[0].set_title("Original Dataset")
sns.scatterplot('x','y', data=mouse_df, hue='EM clustering',ax=axes[1],
hue_order=hue_order)
axes[1].set_title("EM Clustering")
fig.tight_layout()

conf_table = confusion_matrix(mouse_df["cluster"],mouse_df["EM clustering"])
disp = ConfusionMatrixDisplay(confusion_matrix=conf_table,
display_labels=['Ear_left',"Ear_right","Head"])
disp.plot()

# verify the labels are correct in the confusion matrix
(mouse_df["EM clustering"]=="Ear_left").sum()

num_correct = np.sum(np.diag(conf_table))
total = np.sum(conf_table)
```

```
accuracy = (num_correct/total)*100
accuracy
```

Cov of EM applied to mouse data

```
# Here we can see how the EM algorithm assigns clusters to points
#based on what the probability is that they are within each cluster.


def drawbands2(mean_list,cov_list,colors,fig):
    i=0
    for m, cv, col in zip(mean_list,cov_list,colors):
        if cv.shape == (2, 2):
                U, s, Vt = np.linalg.svd(cv)
                angle = np.degrees(np.arctan2(U[1, 0], U[0, 0]))
                width, height = 2 * np.sqrt(s)
        else:
            angle = 0
            width, height = 2 * np.sqrt(cv)

        # Draw the Ellipse
        for nsig in range(1, 4):
            fig.add_patch(Ellipse(m, nsig * width, nsig * height,
                                  angle, fc = colors[i],alpha = 0.15 ))

        i=i+1


test_df=pd.DataFrame(X)
test_df.columns=['q0','q1']
test_df_scores=pd.DataFrame(scores)
test_df = pd.concat([test_df, test_df_scores], axis=1)
#find which score is closest to 0
test_df['EM clustering'] = test_df.iloc[:,2:].idxmax(axis=1)
test_df["original clusters"]=mouse_df['cluster']

mu_gen, cov_gen=get_mu_cov_clusters(clusters)

mean_df=[]
for k in range(len(test_df)):
    l2norm=np.linalg.norm(mu_gen[int(test_df['EM clustering'][k])])
    mean_df.append(l2norm)
test_df["EM mean norm"]=np.asarray(mean_df)

#in order to sync colors we find the norms of the original and
#em means and rank them by their norms
#we then sync the color to the norm
test_df['EM ranking'] = pd.factorize(-test_df["EM mean norm"],
sort=True)[0] + 1

#sync colors, will only go up to 5 clusters unless more colors added

colors = ['purple','green','orange','blue']
```

```
#colors for em
indexlist = np.linalg.norm(mu_gen,axis=1)
indexlist2_e =pd.factorize(-indexlist , sort=True)[0]+1
sub_array = np.asarray(colors)[indexlist2_e]
colors3=sub_array.tolist()
colors3


fig = plt.subplots(nrows=1, ncols=1, figsize=(6, 6))

ax=sns.scatterplot('q0','q1', data=test_df, hue='EM ranking',
hue_order=indexlist2_e, palette= colors3)
drawbands2(mu_gen, cov_gen,colors3, fig=ax)
ax.set_title("EM Clustering")

#Em ranking corresponds to the cluster,but also help with syncing colors


#------------------------------------------------------------------------

GMM using sklearn

from sklearn.mixture import GaussianMixture
gmm = GaussianMixture(n_components=n_clusters, random_state=0).fit(X)

start = time.time()
labels = gmm.predict(X)
end=time.time()
print("Execution Time:", end-start)

mouse_df["Sklearn"] = mouse_df["EM clustering"].replace(["2","1","3"],
['Head',"Ear_right","Ear_left"])
mouse_df["Sklearn"]= mouse_df["Sklearn"].replace(["0","2","1"],
['Head',"Ear_right","Ear_left"])
mouse_df

fig, axes = plt.subplots(nrows=1, ncols=2, figsize=(12, 6))
hue_order=['Head',"Ear_left","Ear_right"]
sns.scatterplot('x','y', data=mouse_df, hue='cluster',ax=axes[0],
hue_order=hue_order)
axes[0].set_title("Original Dataset")
sns.scatterplot('x','y', data=mouse_df, hue='EM clustering',ax=axes[1],
hue_order=hue_order)
axes[1].set_title("EM Clustering")
fig.tight_layout()

conf_table = confusion_matrix(mouse_df["cluster"],mouse_df["Sklearn"])
disp = ConfusionMatrixDisplay(confusion_matrix=conf_table,
display_labels=['Ear_left',"Ear_right","Head"])
disp.plot()

num_correct = np.sum(np.diag(conf_table))
```

```
total = np.sum(conf_table)
accuracy = (num_correct/total)*100
accuracy
```