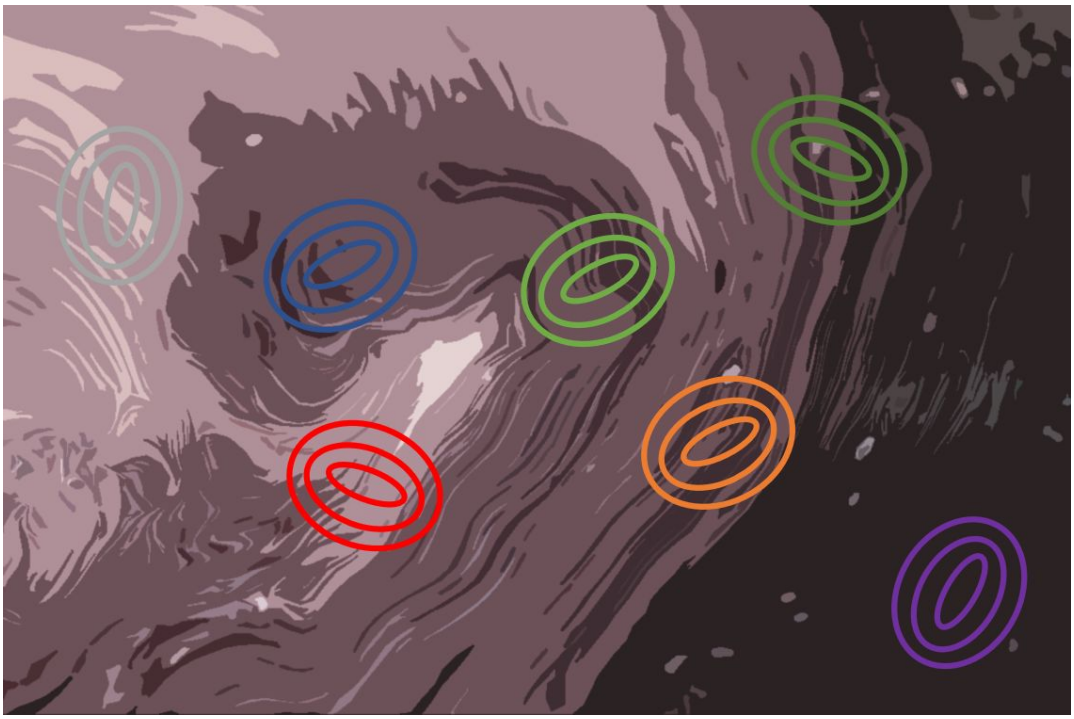
Pattern Recognition

Bayesian Module

Clustering using Gaussian Mixture Model



Kartik Aneja (SBU ID:112817127)

Nitin Asthana (SBU ID:112995559)

Introduction

The goal of the Bayesian Module is to understand and implement clustering using Gaussian Mixture Model. We first analysed and understood the difference between GMM and K-means and how GMM performs better and in which scenarios. We implemented it on stretched Gaussian isotropic generated dataset having 1000 samples for our Lab 3. Next we applied our algorithm on the Iris dataset for our creative component.

Datasets

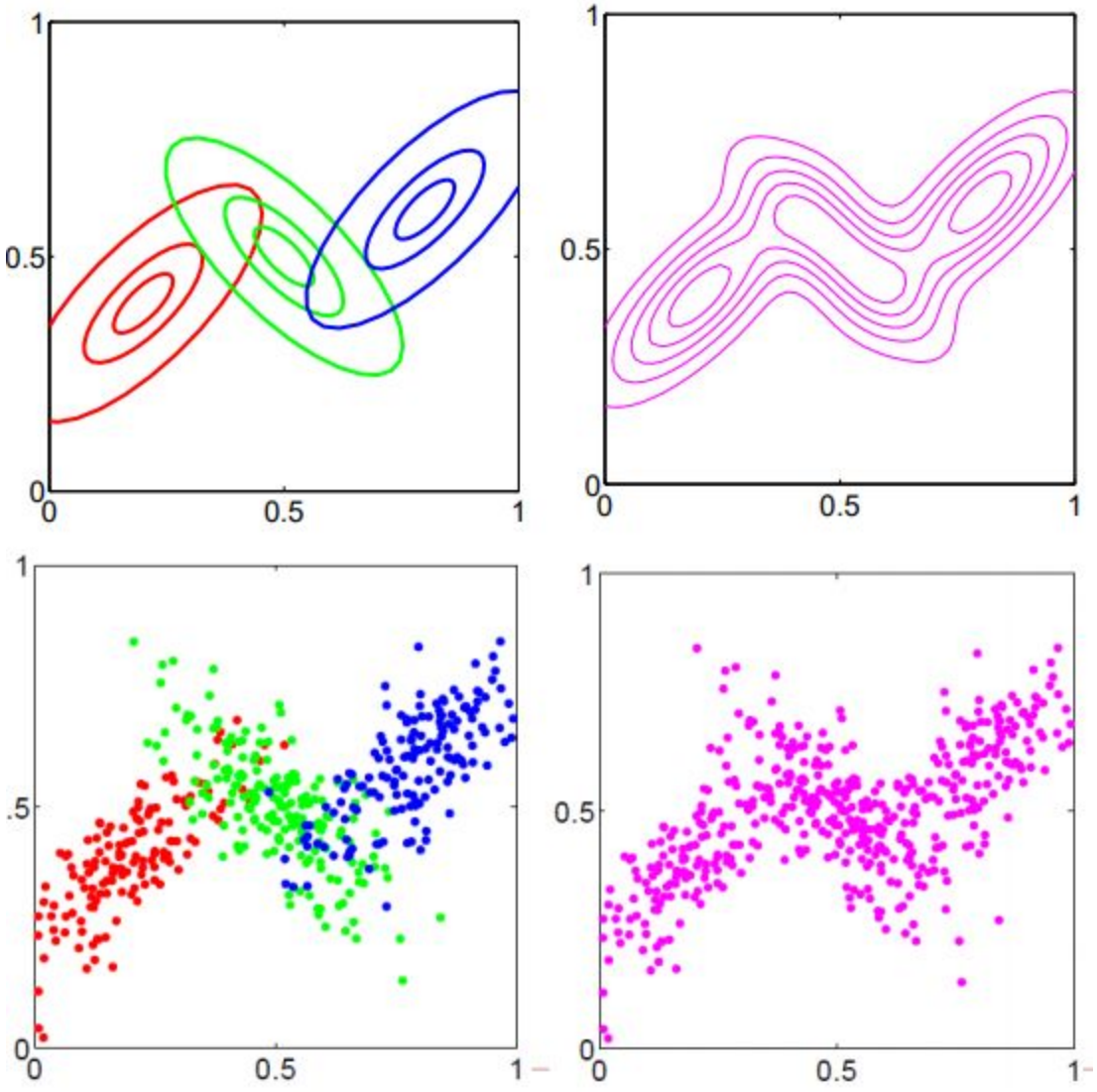
- Iris Species

  Link: https://www.kaggle.com/uciml/iris

  6 Features

GMM Clustering with EM

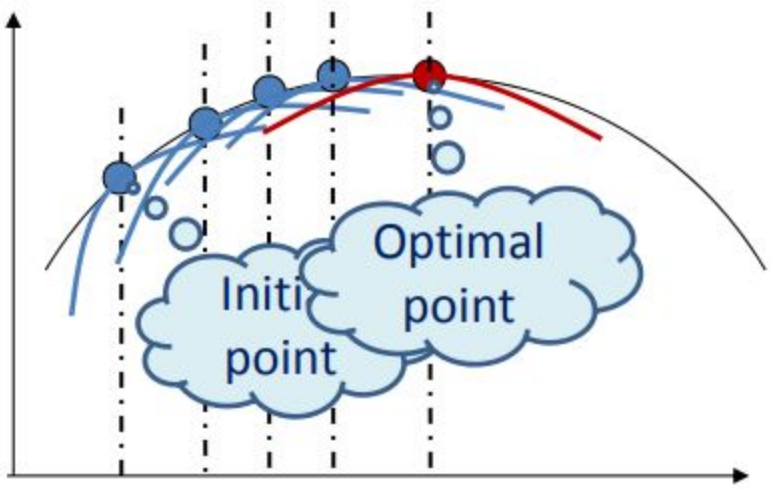
GMM is a type of a clustering algorithm which uses a flexible and probabilistic approach to model data means into soft assignments rather than hard assignments like K-means clustering algorithm. It basically means data points can be generated with any distributions with a corresponding probability. Therefore, each distribution will have some responsibility for data points.



So now the question arises how can we estimate this type of model?

The solution for it is the EM algorithm.

EM algorithm is an iterative optimisation technique which is operated locally.



EM algorithm consists of two steps - Expectation step and Maximization step.

E-Step: It creates a function for the expectation of the log-likelihood which is evaluated using the current estimate for the parameters .

M-Step: It computes parameters maximizing the expected log-likelihood found on the *E* step.

Given a Gaussian mixture model, the goal is to maximize the likelihood function with respect to the parameters comprising the means and covariances of the components and the mixing coefficients).

1. Initialize the means, covariances and mixing coefficients, and evaluate the initial value of the log likelihood.

2. E step. Evaluate the responsibilities using the current parameter values

$$\gamma_j(\mathbf{x}) = \frac{\pi_k \mathcal{N}(\mathbf{x} \mid \mu_k, \Sigma_k)}{\sum_{j=1}^{K} \pi_j \mathcal{N}(\mathbf{x} \mid \mu_j, \Sigma_j)}$$

3. M step. Re-estimate the parameters using the current responsibilities

$$\mu_j = \frac{\sum_{n=1}^{N} \gamma_j(\mathbf{x}_n)\mathbf{x}_n}{\sum_{n=1}^{N} \gamma_j(\mathbf{x}_n)} \qquad \Sigma_j = \frac{\sum_{n=1}^{N} \gamma_j(\mathbf{x}_n)(\mathbf{x}_n - \mu_j)(\mathbf{x}_n - \mu_j)^T}{\sum_{n=1}^{N} \gamma_j(\mathbf{x}_n)} \qquad \pi_j = \frac{1}{N}\sum_{n=1}^{N} \gamma_j(\mathbf{x}_n)$$

4. Evaluate log likelihood

$$\ln p(X \mid \mu, \Sigma, \pi) = \sum_{n=1}^{N} \ln \left\{ \sum_{k=1}^{K} \pi_k N(x_n \mid \mu_k, \Sigma_k) \right\}$$

   If there is no convergence, return to step 2.

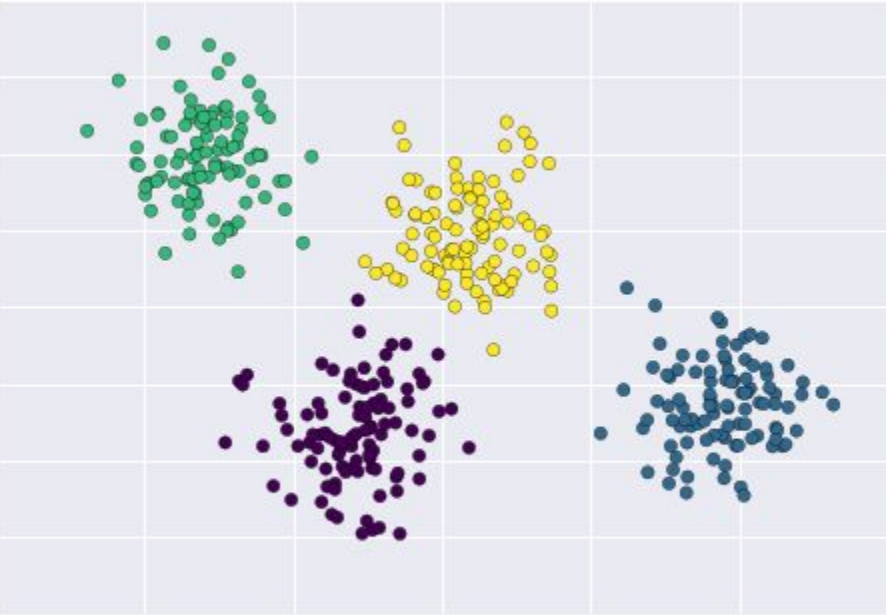
Implementation and Results

Generated Dataset (Lab 3)

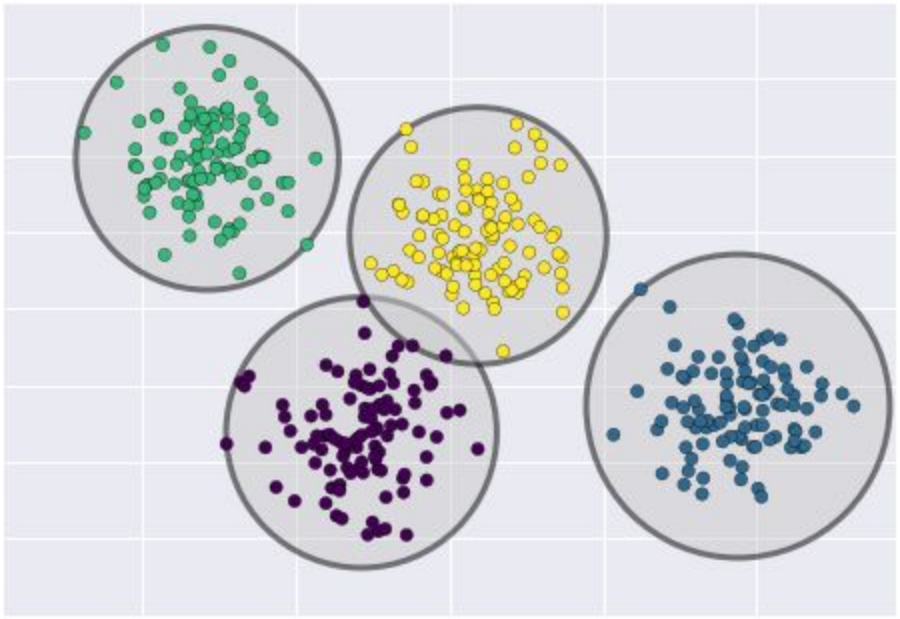
We generated a simple dataset with gaussian blobs using a sklearn sample generator. Next we applied K-mean clustering on the dataset.

```
:  # Generate some data
   from sklearn.datasets.samples_generator import make_blobs
   X, y_true = make_blobs(n_samples=1000, centers=4,
                          cluster_std=0.60, random_state=0)
   X = X[:, ::-1] # flip axes for better plotting
```

Dataset Generated



The k-means can easily and quickly label this simple dataset. After applying K-means we get the clusters as follows:

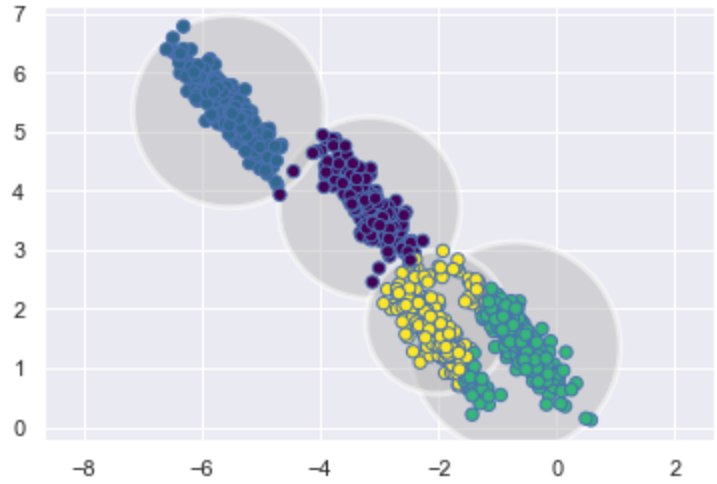


We can clearly see K-means does a good job in clustering. There appears to be a very slight overlap between the two middle clusters, such that we might not have complete confidence in the cluster assignment of points between them.

One way to think about the *k*-means model is that it places a circle (or, in higher dimensions, a hyper-sphere) at the center of each cluster, with a radius defined by the most distant point in the cluster. This radius acts as a hard cutoff for cluster assignment within the training set: any point outside this circle is not considered a member of the cluster.

An important observation for *k*-means is that these cluster models *must be circular*. *K*-means has no built-in way of accounting for oblong or elliptical clusters. So, for example, if we take the same data and transform it, the cluster assignments end up becoming muddled.

K-means on Stretched Data



The transformed clusters are non-circular, and thus circular clusters are a poor fit. *K*-means is not flexible enough to account for this, and tries to force-fit the data into four circular clusters. This results in a mixing of cluster assignments where the resulting circles overlap, in the bottom right corner.

These two disadvantages of *k*-means—its lack of flexibility in cluster shape and lack of probabilistic cluster assignment—mean that for many datasets (especially low-dimensional datasets) it may not perform as well.

With addressing these weaknesses by generalizing the *k*-means model: for example, we could measure uncertainty in cluster assignment by comparing the distances of each point to *all* cluster centers, rather than focusing on just the closest. We might also imagine allowing the cluster boundaries to be ellipses rather than circles, so as to account for non-circular clusters.

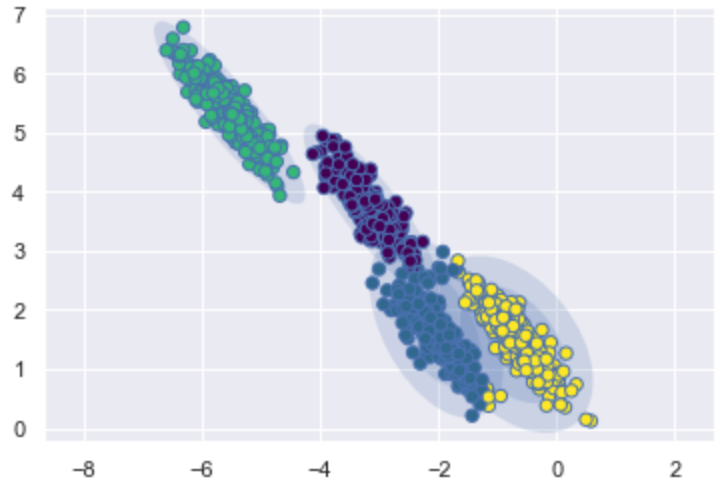
It turns out these are two essential components of a different type of clustering model, Gaussian mixture models.

If we implement GMM on the above dataset, we hope to find a better fit to clusters and we eventually did. Below are the plots for iteration of GMM on the dataset.

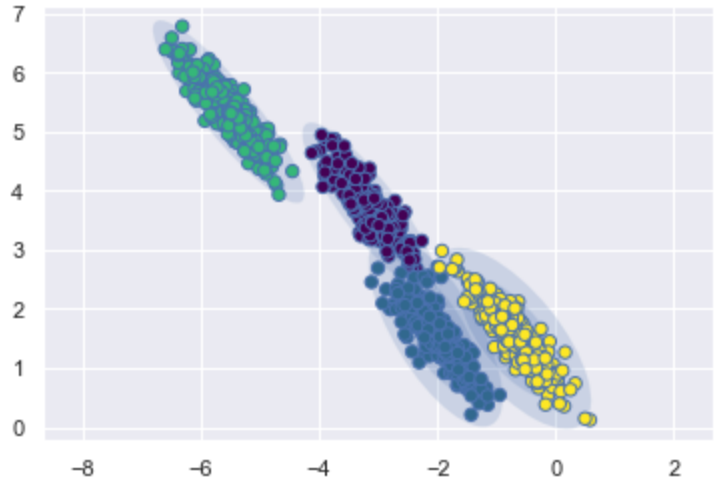
```
def plot_gmm(gmm, X, label=True, ax=None):
    ax = ax or plt.gca()
    labels = gmm.fit(X).predict(X)
    if label:
        ax.scatter(X[:, 0], X[:, 1], c=labels, s=40, cmap='viridis',edgecolor ='b', zorder=2)
    else:
        ax.scatter(X[:, 0], X[:, 1], s=40, zorder=2)
    ax.axis('equal')

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

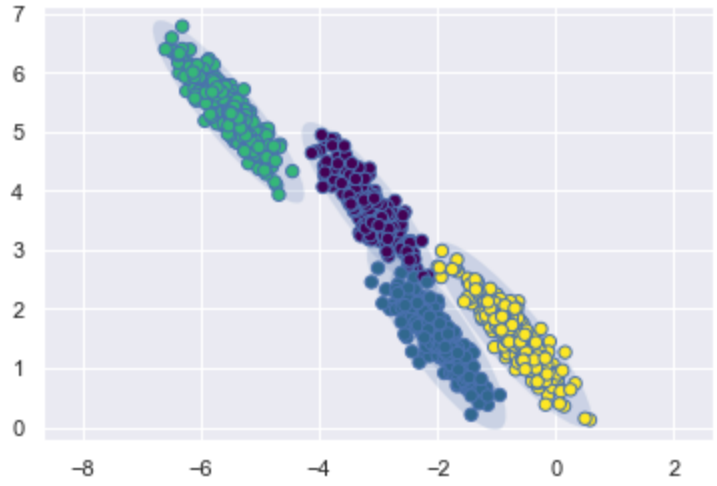
Plotting GMM ellipse and plots



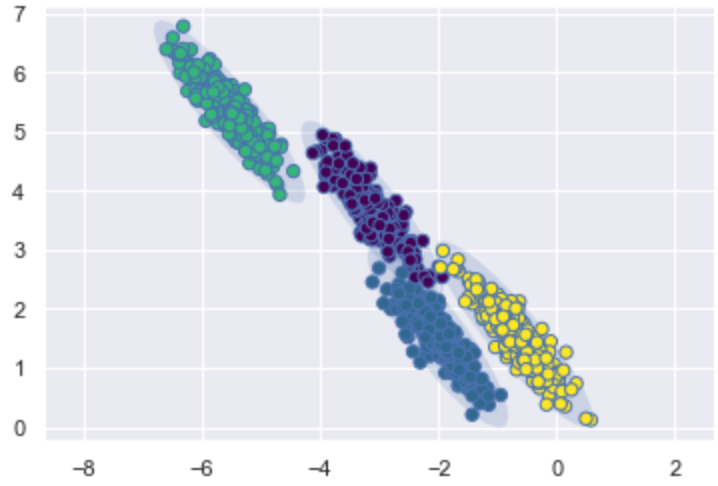
Iteration 1



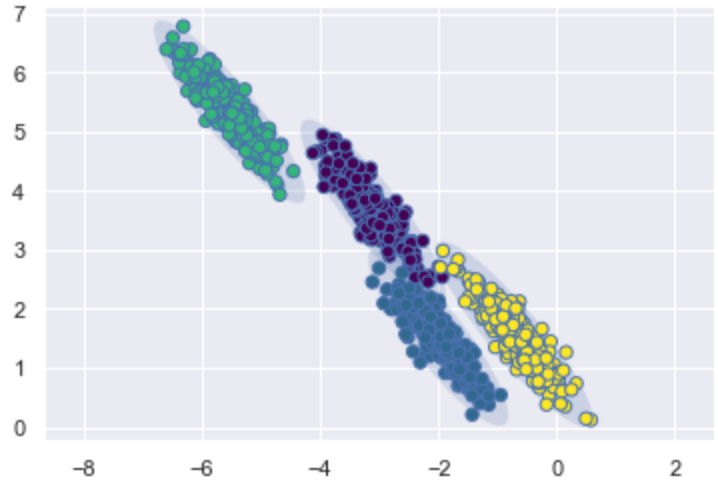
Iteration 2



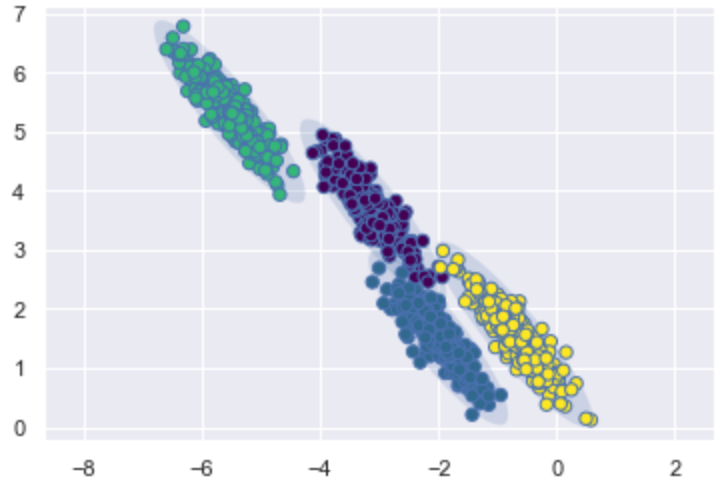
Iteration 3



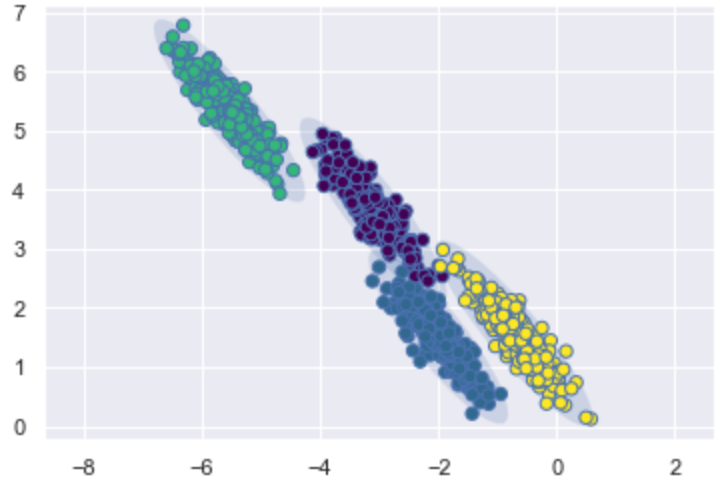
Iteration 4



Iteration 5



Iteration 6



Iteration 7

The GMM converges on the 7th iteration. This makes clear that GMM addresses the two main practical issues with *k*-means encountered before.

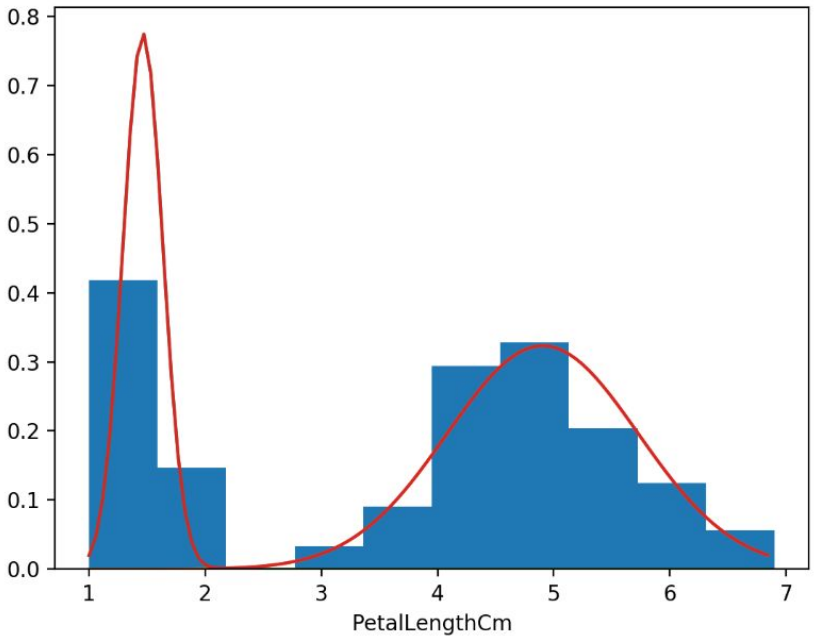
Creative Component

GMM Density Estimation of Iris Dataset:

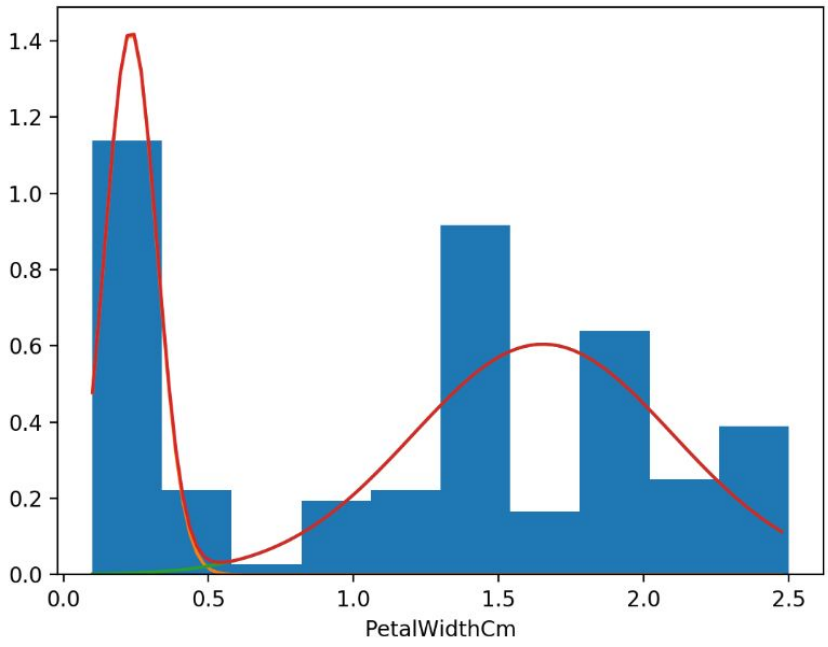
We took the Iris Dataset to perform the GMM using the EM algorithm.

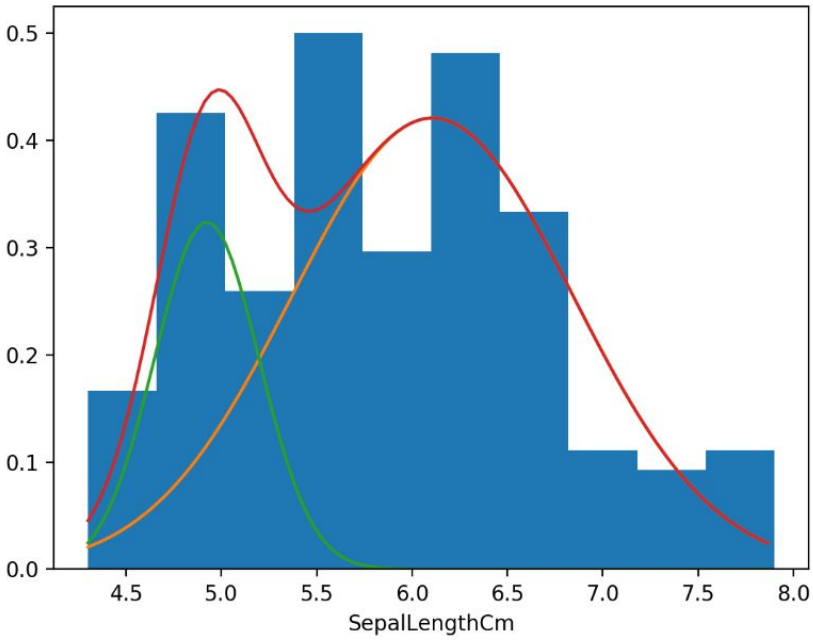
Firstly, for each feature in the Iris dataset, we estimated GMM parameters by using two or three GMM components.

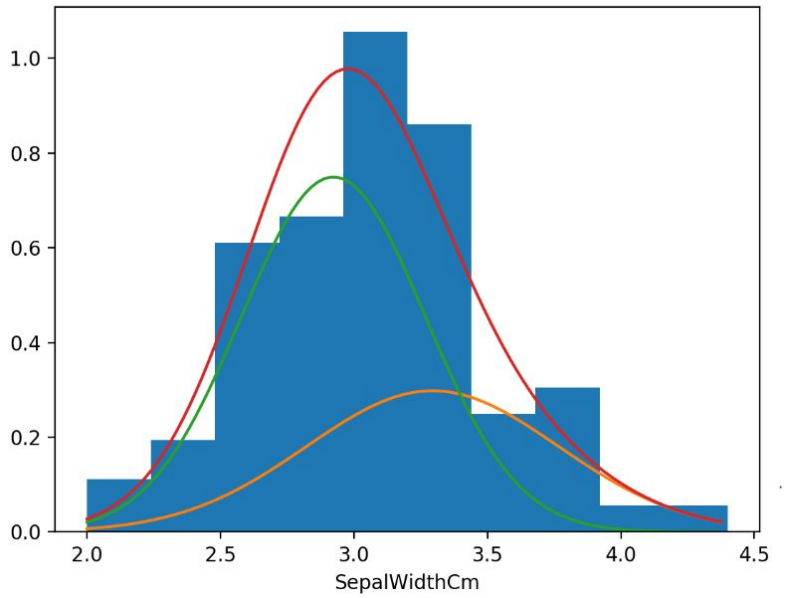
The Number of components in GMM are determined by visualizing the respective feature's histogram.



Two components are required for this feature

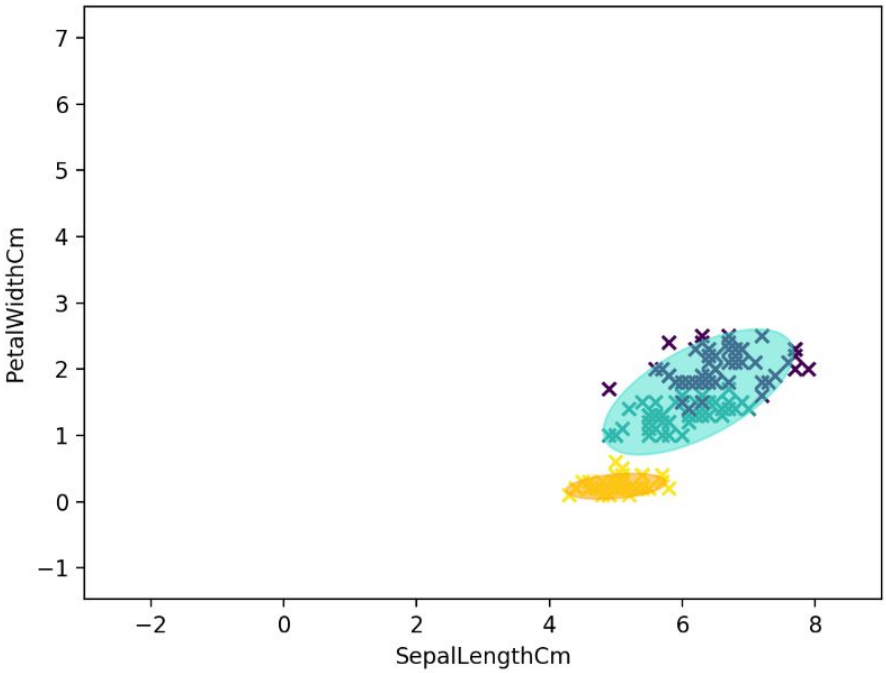


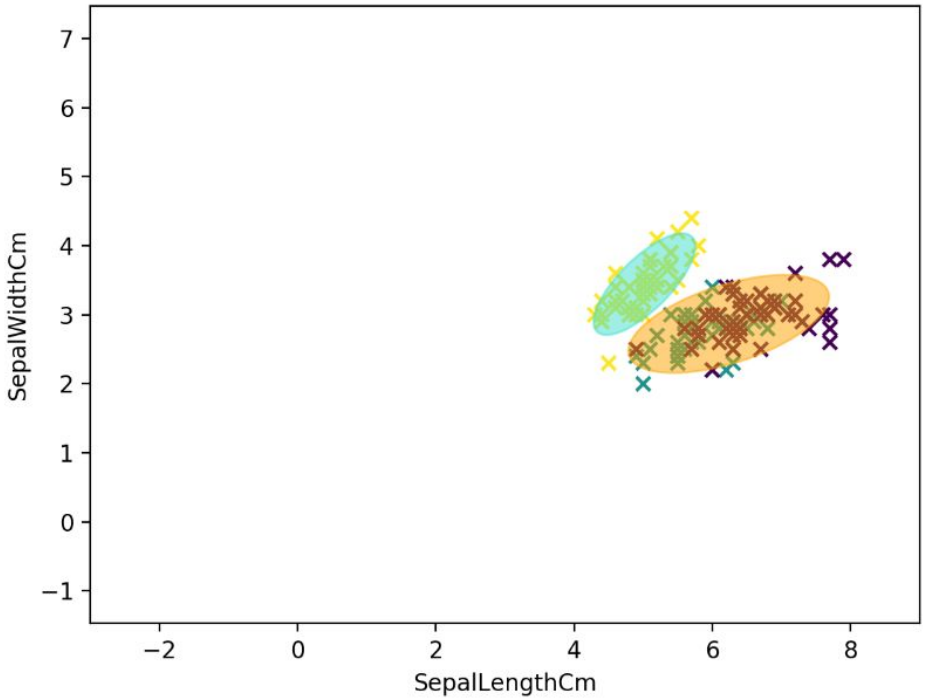


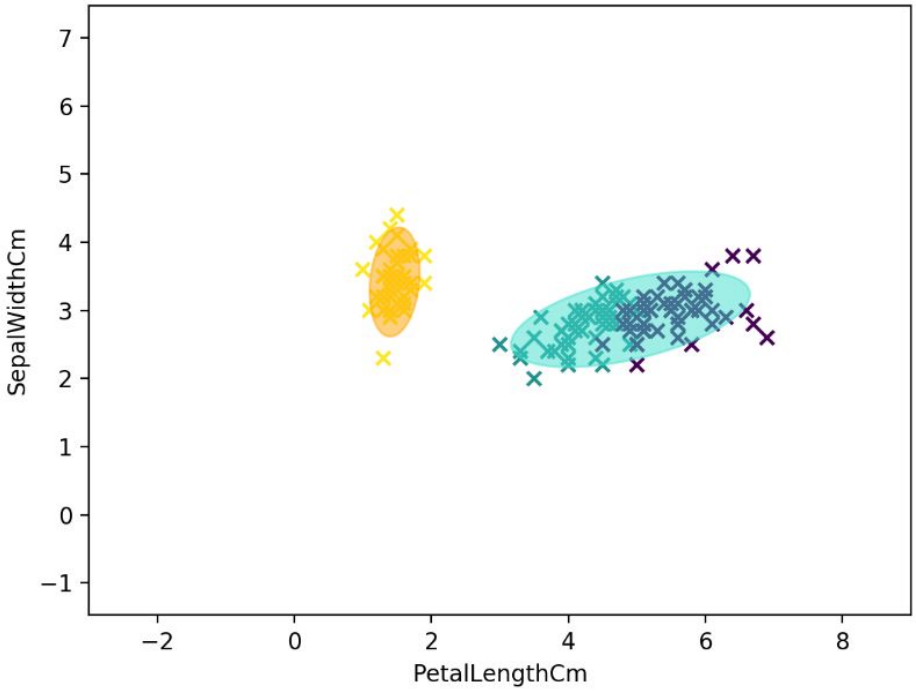


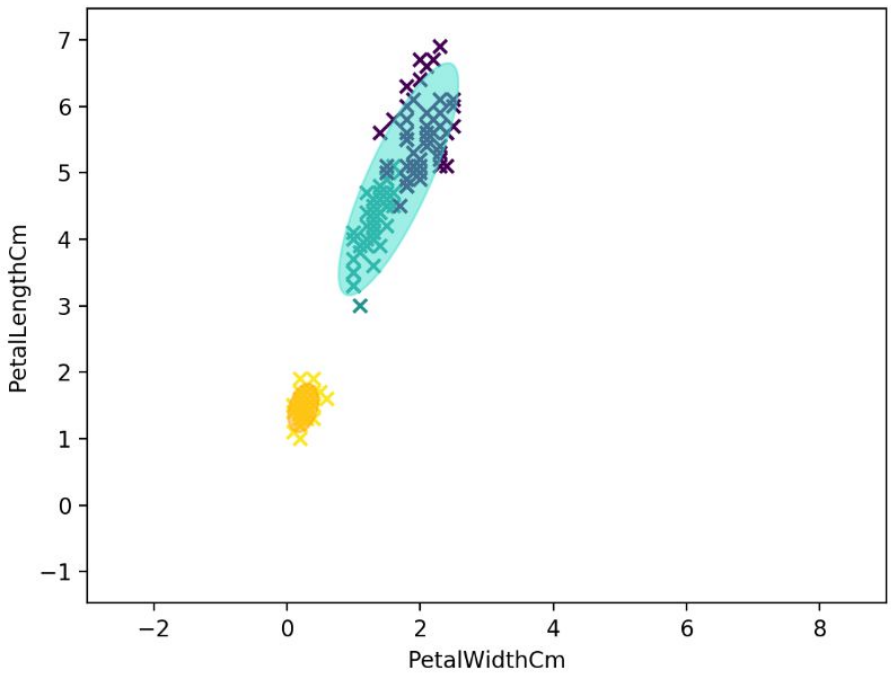
GMM Estimation for Two Features:

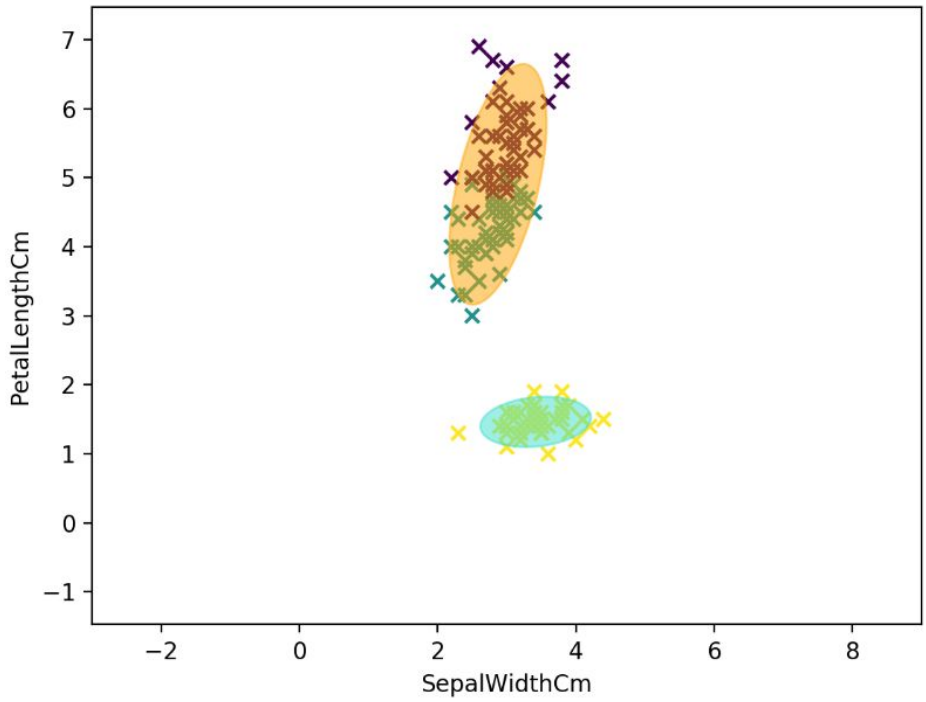
GMM can properly learn the distribution with only two components.




SepalWidthCm
SepalLengthCm
7
6
5
4
3
2
1
0
−1
−2
0
2
4
6
8


SepalWidthCm
PetalLengthCm
7
6
5
4
3
2
1
0
−1
−2
0
2
4
6
8


PetalLengthCm
PetalWidthCm
7
6
5
4
3
2
1
0
−1
−2
0
2
4
6
8


PetalLengthCm
SepalWidthCm
7
6
5
4
3
2
1
0
−1
−2
0
2
4
6
8

We used an external cluster validation index, the adjusted Rand Score which generates a score between -1 and 1 (where an exact match will be scored as 1).

The ADR for GMM comes around .90.

```
iris['gmm_pred'] = pred_gmm_iris

# TODO: calculate adjusted rand score passing in the original
# labels and the GMM predicted labels iris['species']
iris_gmm_score = metrics.adjusted_rand_score(iris['species'],pred_gmm_iris)

# Print the score
iris_gmm_score
```

```
0.90387423177481241
```

```
# TODO: Import adjusted rand score
from sklearn import metrics

# TODO: calculate adjusted rand score passing in the original labels and the kmeans predicted
labels
iris_kmeans_score = metrics.adjusted_rand_score(iris['species'],iris['kmeans_pred'])

# Print the score
iris_kmeans_score
```

```
0.73023827228346971
```

The ADR for K-means comes around .73, which shows GMM did much better at finding the actual species clusters. GMM tries to fit normally distributed clusters, which is probably the case with this data, so it fits it better. k-means is biased towards spherically distributed clusters.

Application for GMM

The aim of this part is to train an unsupervised learning model for identification of objects with different color distributions present in an image.

Why do we need a GMM and not a random Gaussian Distribution?

Mixture Model, is a probabilistic model used to represent the subsets within an overall dataset (It can be a mixture of Gaussian or a mixture of any other distribution). A Gaussian mixture model is simply a function which contains several Gaussian distributions within itself and each of these can be identified by $k \in \{1,\ldots, K\}$, where K is the number of clusters of our dataset.

We used GMM to perform color segmentation on an input image. The image's pixel values are extracted using python CV library and are stored in a numpy array. Then these values are taken as input in the GMM algorithm and GMM with the help of EM finds and appropriates colors to the picture using the number of components used. We performed this on various images and tried to train the model on the certain type of picture which can be used to detect the same type of images. Due to shortage of time we were unable to train the model and test it. But the GMM holds a good opportunity in color segmentation and detection.

+ Code + Text

```
[1] import numpy as np
    import cv2
```

```
[3] img = cv2.imread("test.jpg")
```

```
[4] img2 = img.reshape((-1,3))
```

```
[5] from sklearn.mixture import GaussianMixture as gmm
```

```
[6] gmm_model = gmm(n_components=2, covariance_type='tied').fit(img2)
```

```
[7] gmm_labels = gmm_model.predict(img2)
```

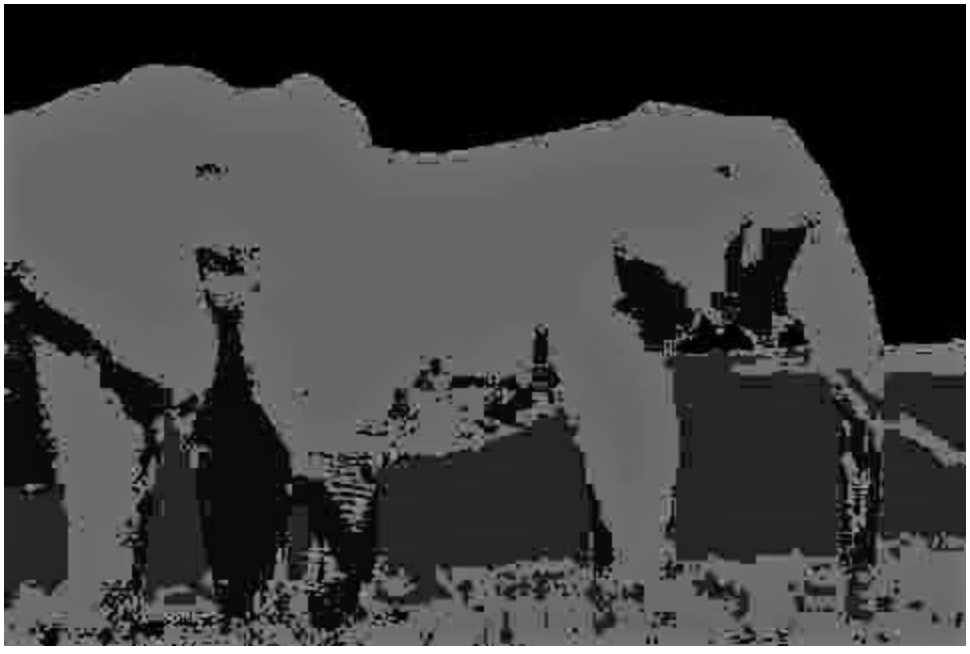
```
[8] original_shape = img.shape
    segmented = gmm_labels.reshape(original_shape[0], original_shape[1])
```

```
[9] cv2.imwrite("segmented_test.jpg", segmented)
```

```
True
```

Left image is a test image and right image is ouput segmented image.

The above image is a test image



The above image is a output segmented image.

Libraries Used

pandas - for handling the data frames and reading csv files

numpy - for transformation of datasets

Scipy - for statistics

matplotlib.pyplot

Sklearn.mixture.GMM

Statistics - for calculating mean

Cv2 - To read image

References

1. https://towardsdatascience.com/gaussian-mixture-modelling-gmm-833c88587c7f
2. https://wikimedia.org/api/rest_v1/media/math/render/svg/cbd5cf1ea52696745ea83c2e67493e9e41d8b67f
3. http://www.cse.iitm.ac.in/~vplab/courses/DVP/PDF/gmm.pdf
4. https://scikit-learn.org/0.16/modules/generated/sklearn.mixture.GMM.html
5. https://tinyheero.github.io/2016/01/03/gmm-em.html
6. https://stephens999.github.io/fiveMinuteStats/intro_to_em.html
7. http://statweb.stanford.edu/~tibs/stat315a/LECTURES/em.pdf
8. http://www.ee.bgu.ac.il/~haimp/ml/lectures/lec2/lec2.pdf
9. https://medium.com/@siddharthvadgama/gaussian-mixture-model-gmm-using-em-algorithm-from-scratch-6b7c764aac9f
10. https://www.analyticsvidhya.com/blog/2019/10/gaussian-mixture-models-clustering
11. https://levelup.gitconnected.com/gaussian-mixture-models-gmm-816f549940c5
12. http://iacs-courses.seas.harvard.edu/courses/am207/blog/lab-on-em.html
13. https://www.youtube.com/watch?v=qMTuMa86NzU
14. https://docs.opencv.org/2.4/modules/ml/doc/expectation_maximization.html
15. http://people.csail.mit.edu/dsontag/courses/ml12/slides/lecture21.pdf