## GMM implementation

The Gaussian Mixture Model is a statistical model that assumes that the observations follow a normal distribution, ie $f(x|z)$ is a Gaussian distribution with expectation $\mu_Z$ and covariance matrix $\Sigma_Z$. The random variable $Z$ is a hidden variable.

### *our data generator:*

after implementing the MLE algorithm, we will use it to cluster data that we will generate using the gaussian data generator (from exercise 0).

Let us define the datapiont generator:

## MLE Overview

this time we will use the MLE algorithm to cluster the data.

pseudo code: The MLE algorithm goes as follows

here is MLE algorithm in pseudo code

1.until convergence, repeat:

#### E-step:

```
1. For each data point $x_i$:
    a. Compute the probability that $x_i$ belongs to each of the clusters, $p_{i,j}$
    b. Use these probabilities to compute the expected value of the cluster
       assignment, $w(i,j) = \frac{P(x_i \in Z_j) * \phi_j}{\sum_{j=0}^{k} P(x_i \in Z_j) \cdot \phi_j}$
```

### *M-step:*

```
1. For each cluster $j$:
    a. compute the new mean, $\mu_j$
    b. compute the new variance, $\sigma_j^2$
    c. compute the new prior, $\varphi_j$
```

this is what the implementation looks like:

### Initial Model:

the MLE algorithm we needs an initial model to start with. as described above algorithm demands an initial value $\theta^0$ ie. initial values for: $\mu_j, \sigma_j^2, \phi_j$ for each gaussian $j$ in the model, in order to calculate the first E-step, ie. the probability of each data point to belong to each gaussian. $p(z_i = j|x_i, \theta^0)$ which is the first estimation of the latent variables $w_{ij}$.

one idea would be to use K-means algorithm to initialize the means of the model. we will experiment with tat later, but first we will try initiating with an arbitrary choise, as such:

## convergence conditon:

we will also like to decide when to stop the algorithm. for one thing, we will limit the maximal # of iterations to a fixed number. for another, the algorithm increases the log likelihood, i.e., $ log P(x^n |/theta^t$ so we will calculate the log likelihood at each step, as follows:

Log Likelihood:

$$logP(x^n|\theta^t) = log\left(\sum_j P(x^n, z^n; \theta^t)\right)$$

$$= log\left(\prod_i \sum_j N\left(x_i, \mu_j, \sigma_j^2; \theta^t\right) * \phi_j\right)$$

$$= \sum_i l\,og\left(\sum_j w\,(j, i)\right)$$

and compare to previous value, to check for convergence.

## first experiment: full & diagonal covariance matrix

## Case I:

*$\phi$ and $\sigma$ are known,*

here there is no need to update the variables responsible to the "shape" of the distribution, i.e. the covariance matrix $\sigma$ and the amplitude $\phi$ we are looking for the centroids, $\mu$

### generate the data

in case 1,we are given the variance matrecies, say:

$$\Sigma_1 = \begin{bmatrix} 0.6 & 0 \\ 0 & 0.6 \end{bmatrix}, \Sigma_2 = \begin{bmatrix} 0.8 & -0.3 \\ -0.3 & 0.6 \end{bmatrix}$$

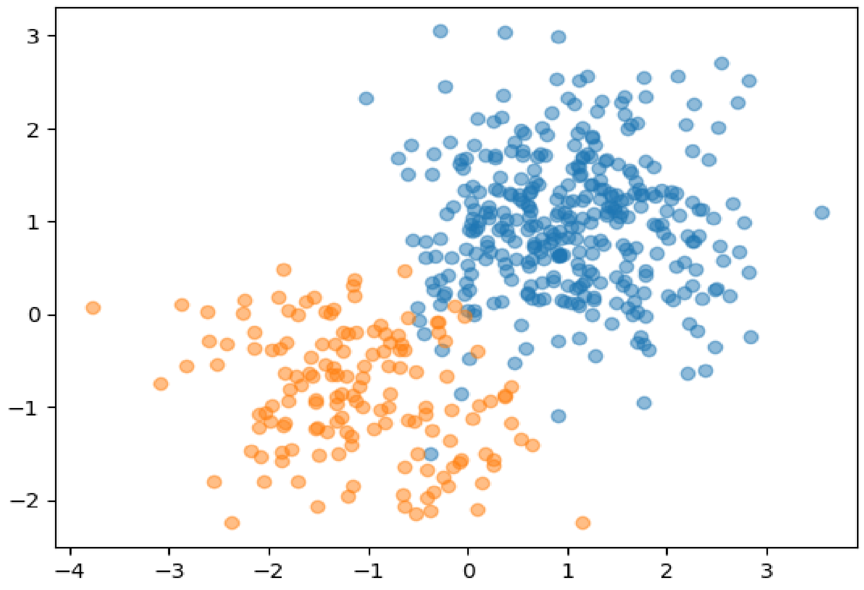
we are also given the probability for the first label:

$$\phi_1 = P_Z(1) = 0.7$$

and we only use the MLE algorithm to find the means for each of the clusters.

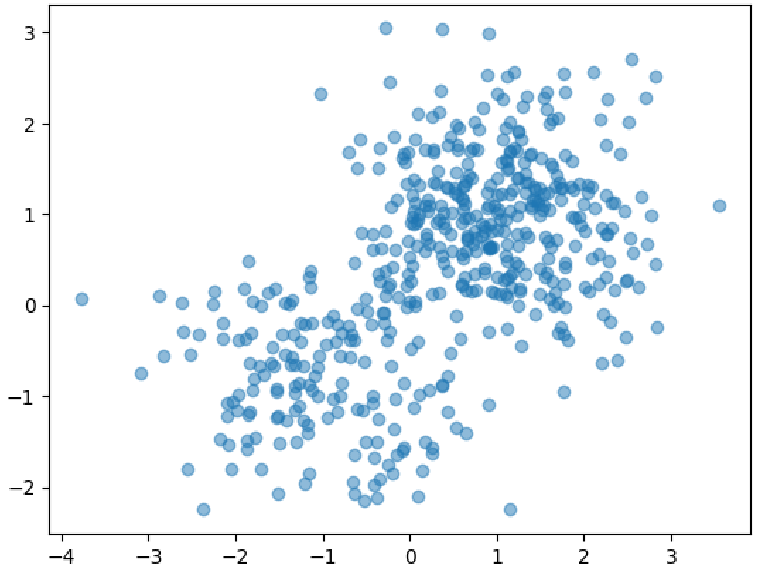
$$\mu_1 = \begin{bmatrix} a \\ b \end{bmatrix}, \mu_2 = \begin{bmatrix} c \\ d \end{bmatrix}$$

## plot the data

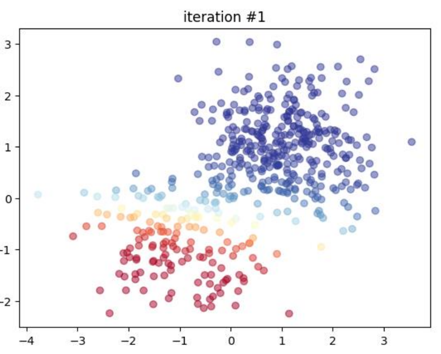


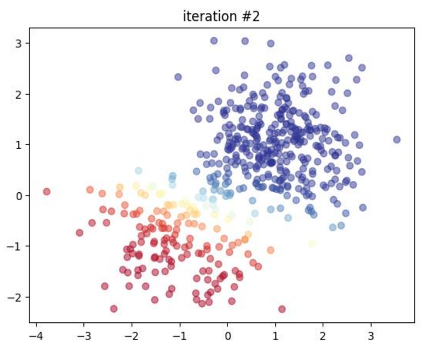
## plot the data without assignment to clusters

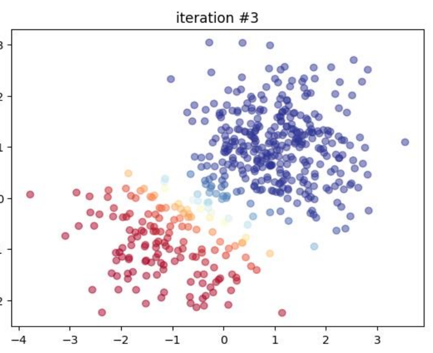
this is what the data looks like without our knowledge, the algorithm gets the data as such:

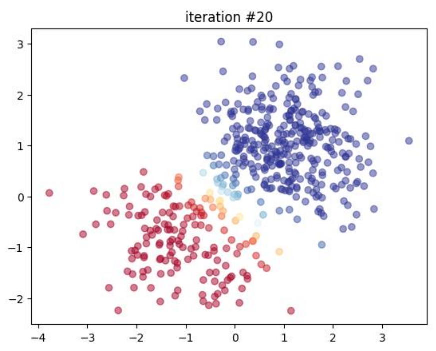


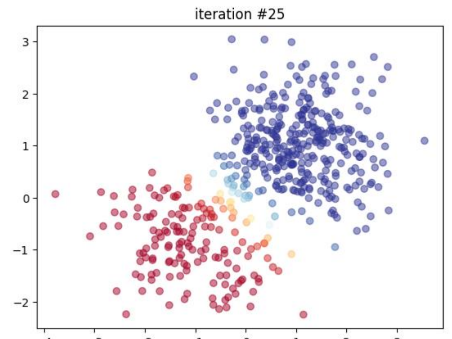
## run the algorithm

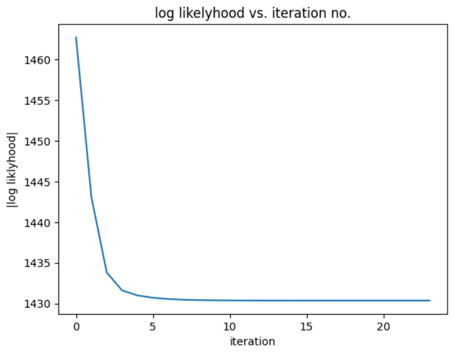












## results

even though we are not given the centroids, and the initial guessed centroids are in incorrect positions, the algorithm converged to the correct centroids in a pretty steep convergence curve. in 20 iterations, the algorithm returned, and we can see that the log likelihood is near its maximum after ~5 iterations, a remarkable rate indeed. this is because given the shape of the gaussians the algorithm can easily find where the shapes fit the data best.

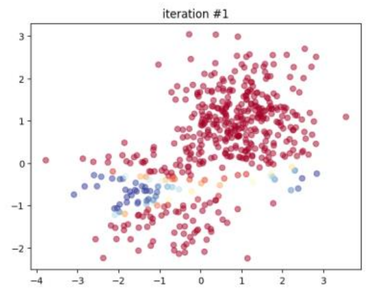
## Case II:

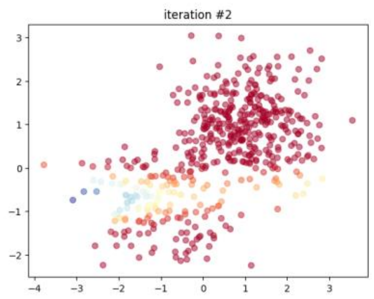
### *$\phi$ $\sigma$ and $\mu$ are unknown*

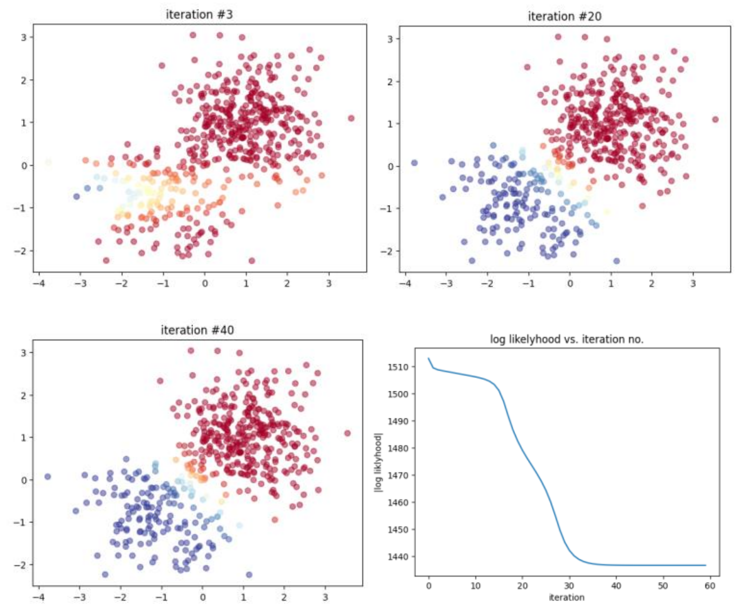
in case 2, we are not given the variance matrecies, nor the probability for the first label, so we are looking for the centroids, $\mu$ and the variance matrecies $\sigma$ and the amplitude $\phi$ could it be that the algorithm will converge to the same values as in case 1?

sience we are not given the variance matrecies, we will have to calculate them at each iteration, using the centroids we found so far, and the labels we assigned to each datapoint. we will initialize the model to a random gaussian distribution, and run the algorithm.

### *run the algorithm on the same data*

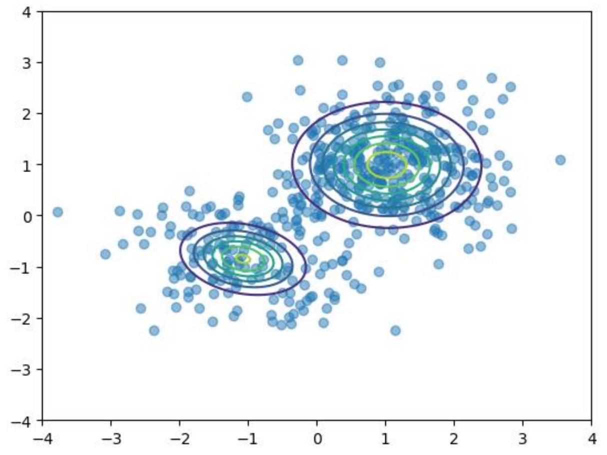






## results

as we can clearly see, the initial guess was way off, but the algorithm converged to the actual values of the centroids, and the log likelyhood converged, although it took much more iterations to run (~40), this comes in contrast to a very sharp convergence graph in case 1. the algorithm is able to figure out the shapes of the gaussians, but it takes more time, trial, and error to do so. we can see the final model returned by the algorithm, by plotting the contours of the gaussians, and the data, the outcome is exactly what we would expect.



the model's parameters settle on the actual gaussians that are behind the data generation, and the log likelyhood converges, indicating that thealgorithm is likely to have stayed around said values, was it given even more iterations to run.

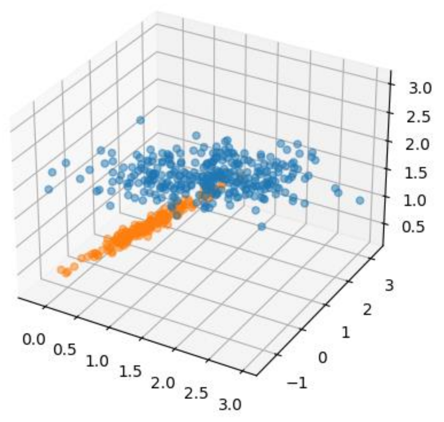
## second experiment: 3D data, full covariance matrix

### generate the data

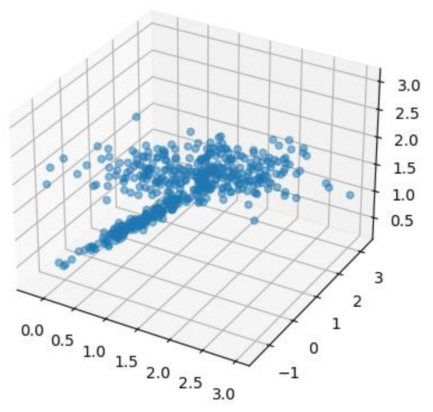
```
k=2          # number of clusters
d=3          # number of dimentions
N = 500    # number of datapoints
```

### plot the data on 3d plane

in this plot we can see the data on a 3d plane, plus, which datapoints belong to each cluster.



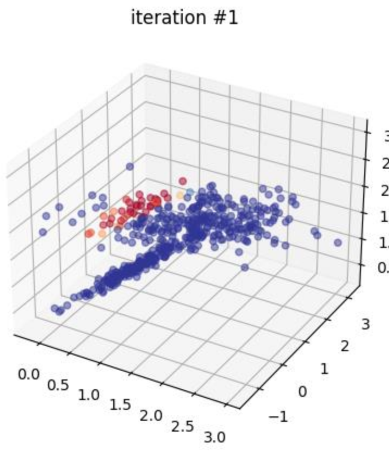
### plot the data without assignment to clusters

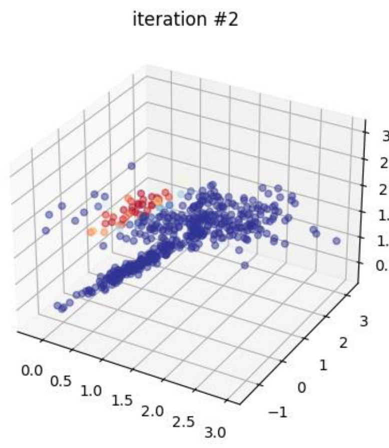


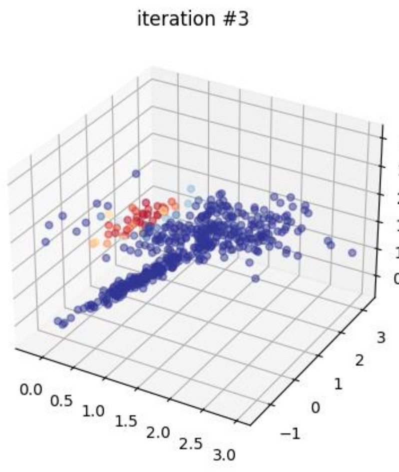
## case I:

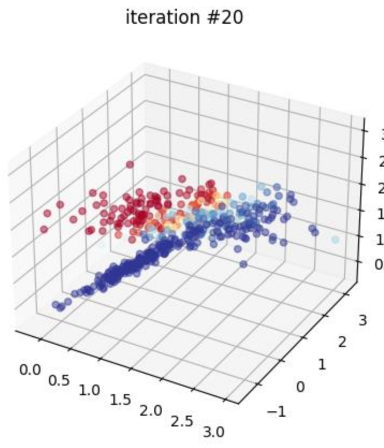
### run the algorithm

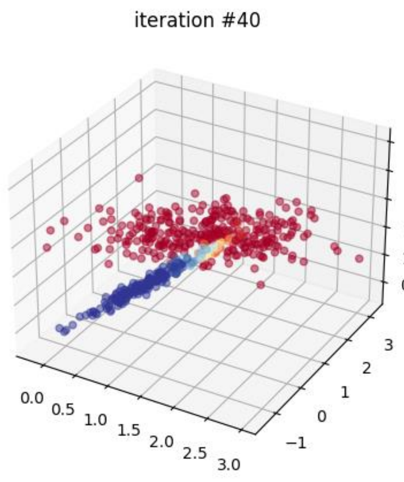
we will run the algorithm, and provide it with the underline $\phi$ and $\sigma$ matrecies, and see if it converges to the actual $\mu$ values.

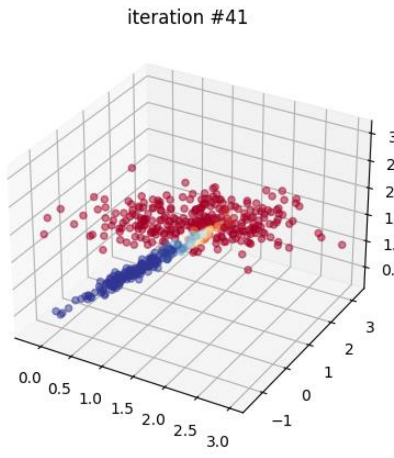

iteration #1

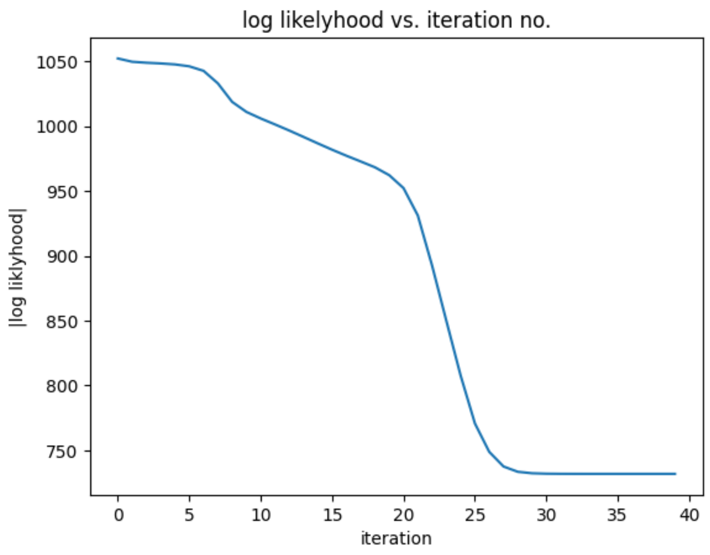

iteration #2


iteration #3


iteration #20


iteration #40


iteration #41

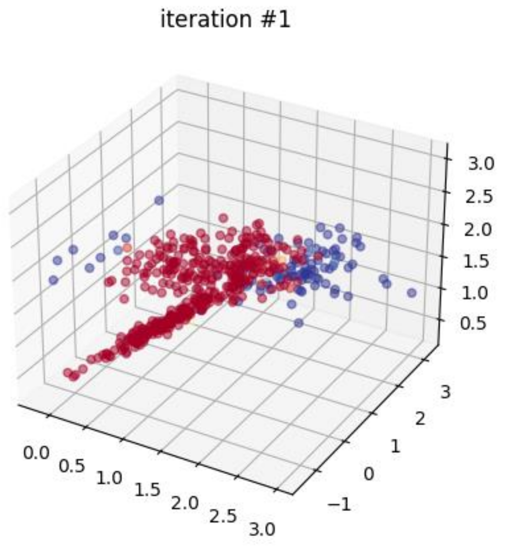


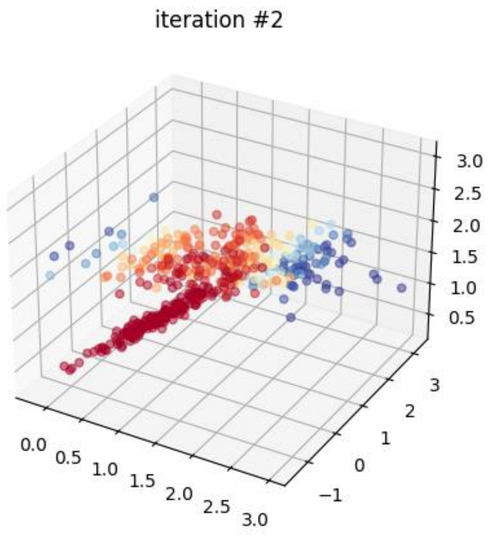
## results

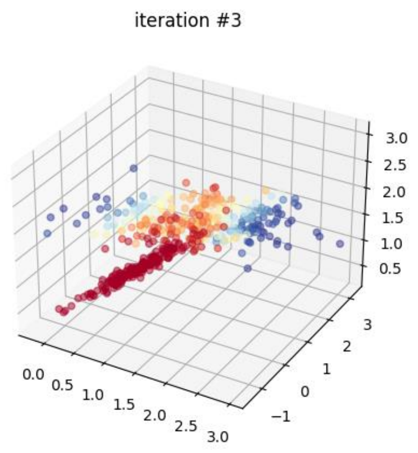
it took the algorithm ~20 iterations to converge.

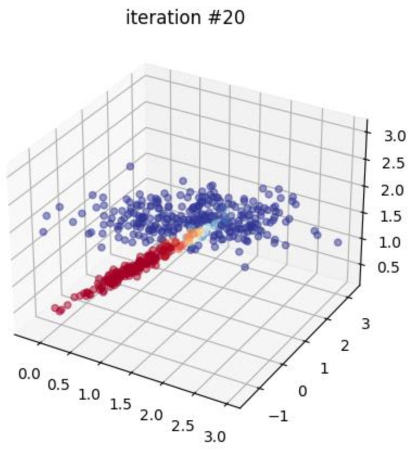
## case II:

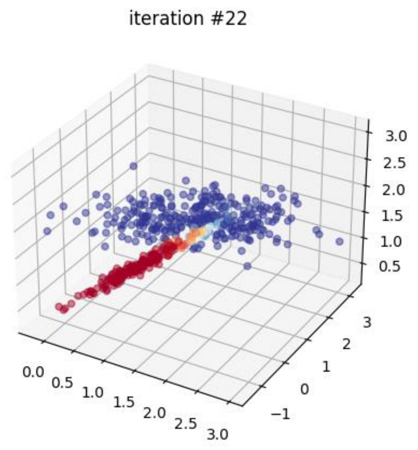
```
initiating a random model...
```

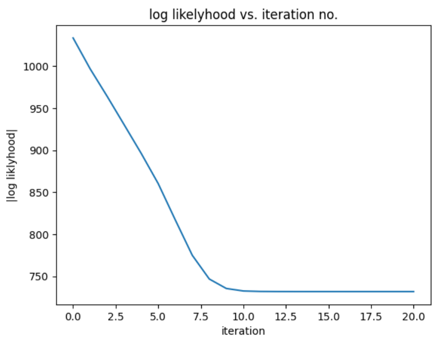












## results

the results here are quite surprising, the algorithm converged to the actual centroids, and the log likelyhood converged, again, showing the effectiveness of the algorithm. but it took much more iterations to run (~40) in case I, ie. *with* sigma and phi given,then *without* the knowledge of the variance matrecies and the intensity of each cluster. why? because having the variance matrecies and the intensity of each cluster given makes the algorithm more volnurable to the initial guess, a 'bad' asignment of the centers can lead to a very slow convergence, as long as the estimated gaussian is far from the actual cluster, the estimated $\sigma^2$ metrix cannot be changed in the process, which slows down the process even more.

we can see on the convergence graph of CASE I that indeed the log likelyhood is not as steep as in case I, until the 'breakthrough' point around $iter = 22$, where the algorithm finally assign most of the datapoints of the cluster with the bigger volume the correct cluster, then the log likelyhood converges rapidly.

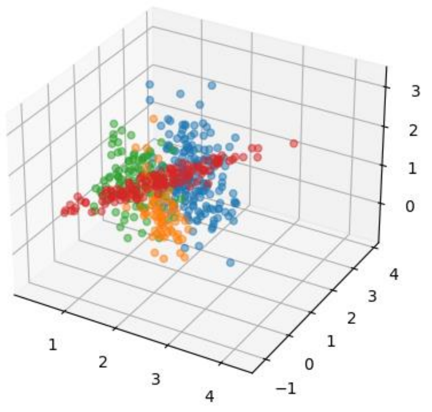
in CASE II the convergence is faster, because the algorithm is not 'stuck' with the initial guess of the variance matrecies, and can change them as it sees fit, which makes the algorithm more robust to the initial guess of the centroids. it gets very close around ~10 iterations

## third experiment: 3D data, full covariance matrix, 4 clusters

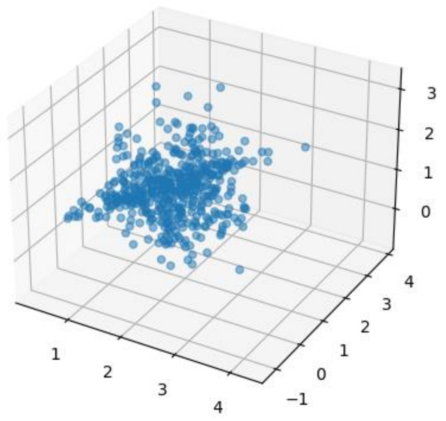
### generate the data

```
k=4          # number of clusters
d=3          # number of dimentions
N = 500    # number of datapoints
```

### plot the data on 3d plane

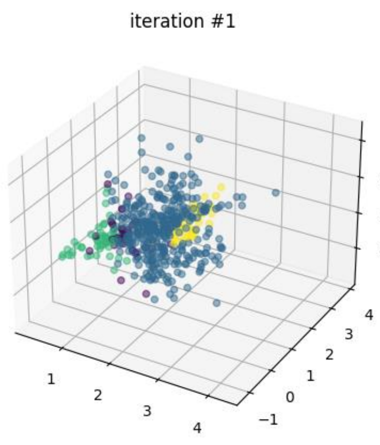


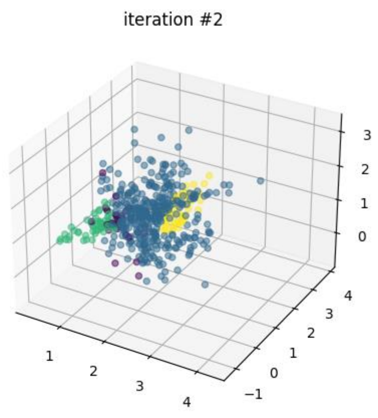
### plot the data without knowladge of the latent

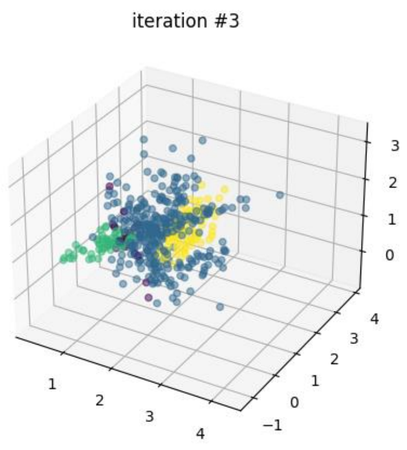


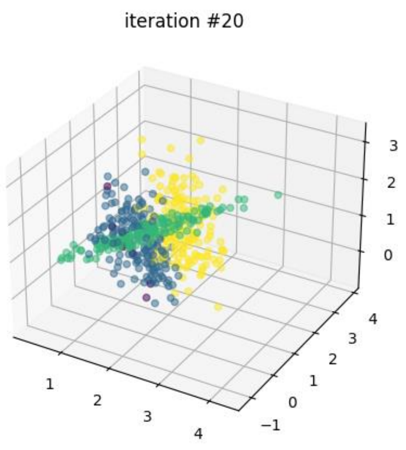
## case I:

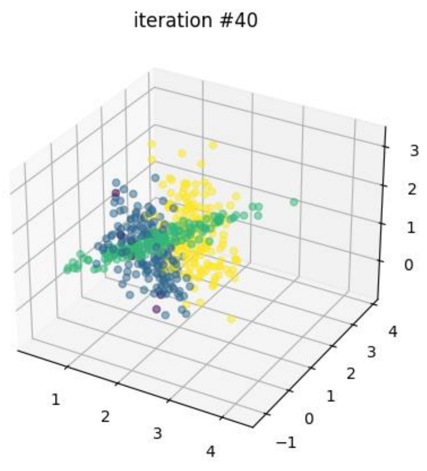
### run the algorithm

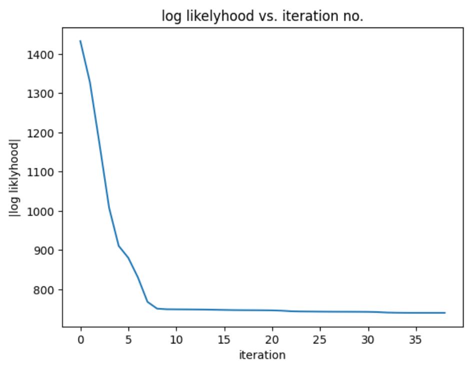








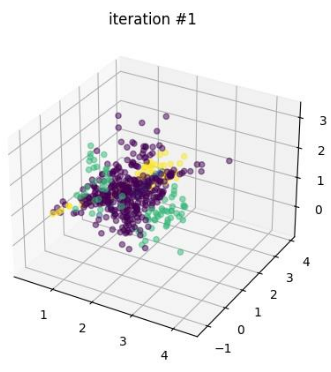


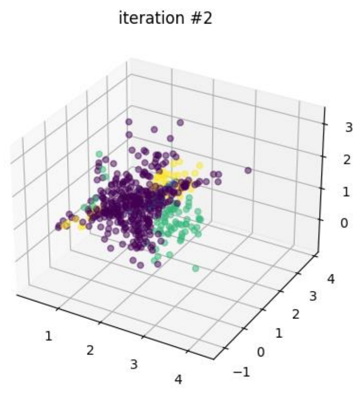


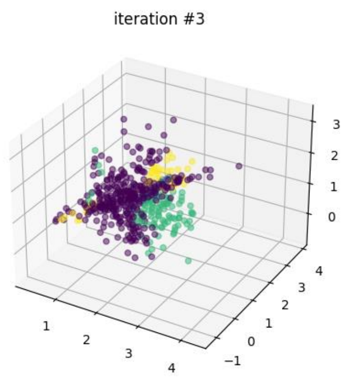
## results

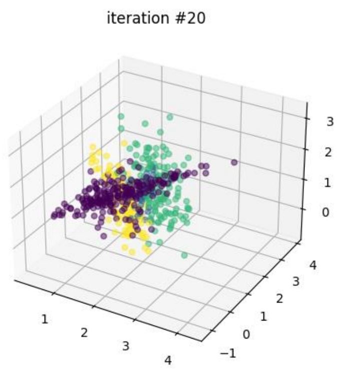
## case II:

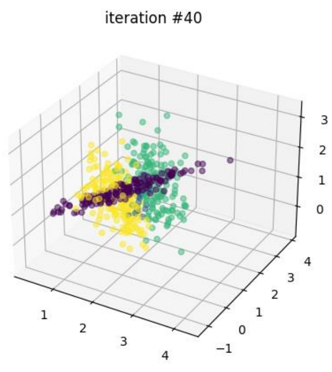
```
initiating a random model...
```

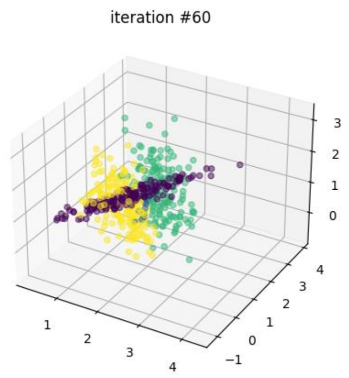


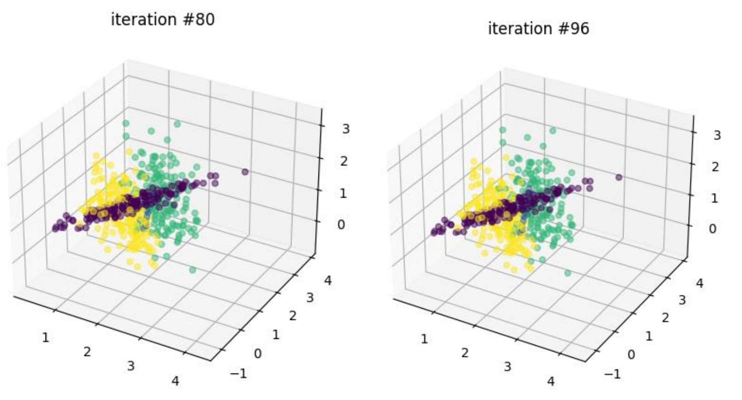


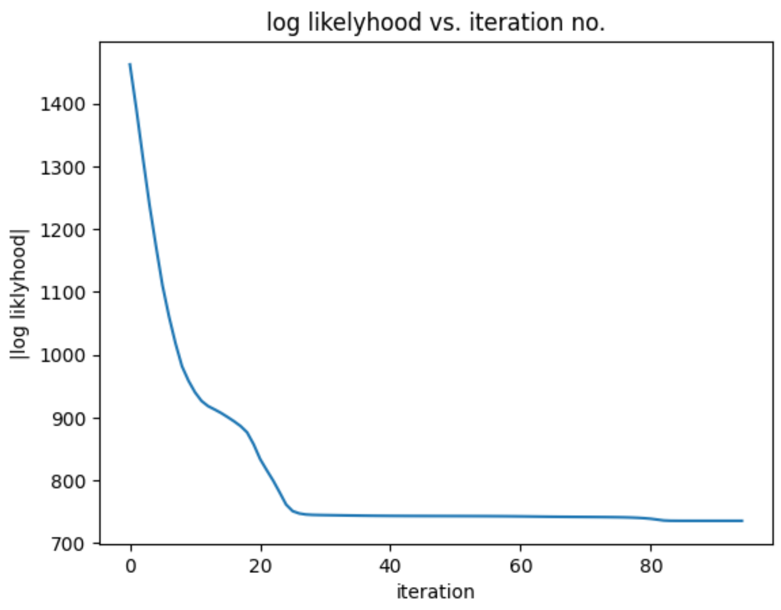









iteration #80
iteration #96
0
1
2
3
4
-1


log likelyhood vs. iteration no.
|log liklyhood|
iteration
700
800
900
1000
1100
1200
1300
1400
0
20
40
60
80

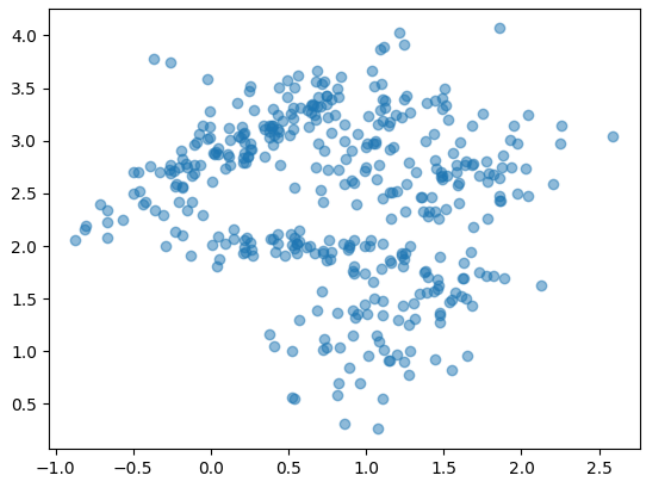
## results

## K - Means

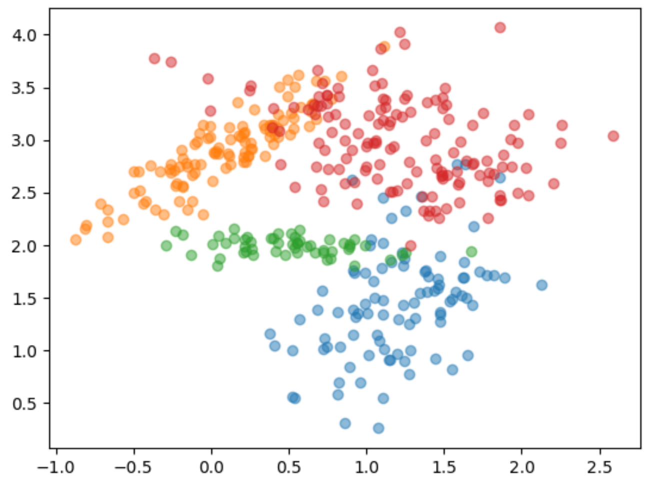
### fourth experiment: 3D data, full covariance matrix, 4 clusters, K-Means as initial model

let us try the MLE on a dataset which is a bit challenging: we have gaussians of different shapes sizes and intensities, even when given $\sigma$ and $\phi$ matrecies, the algorithm is not likely to find an initial guess that will allow it to converge. then we will try to see if the K-Means algorithm can help us with that.

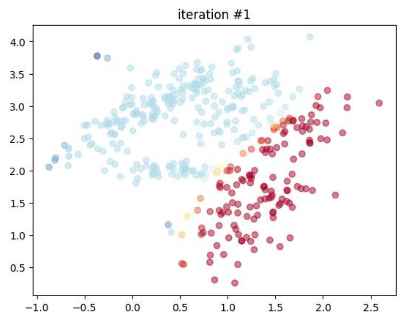
#### plot the data:

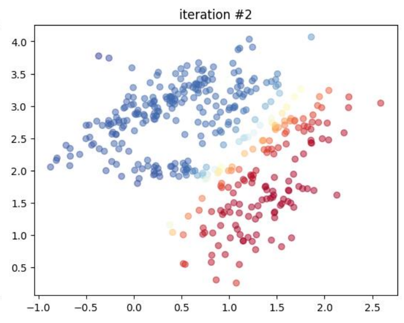


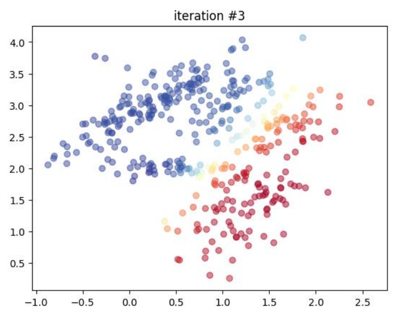
#### plot the data with cluster assignments:



## let’s run the MLE algorithm on the data:







the algorithm diverged, because of a 'bad' initial guess. in fact, most 'guesses' are likely to fail the algorithm in cases of a dataset such as right here. so, is there a smarter way to initialize the agorithm? we will try to use the K-Means algorithm to initialize the algorithm, and see if it helps.

## the k-means algorithm as a model initializer

GMM implementation with K-Means as an initial Model

we talked about the algorithm's need of an initial model, for the algorithm to start, and experimented with initiating at random (K - Gaussians picked at random, defined by:

- $\mu_j$ - the mean of each gaussian
- $\sigma_j^2$ - the covariance (matrix) of each gaussian
- $\phi_j$ - the "amplitude" of each gaussian

this time we will try a more interesting method - use the K means algorithm's output to initiate $\mu_j$ . K- means is thoroughly explained in another paper in this repository, and is "hard" algorithm, in contrast to EM, which is considered "soft". in a few iterations,we get very close to the conclusion of the actual $\mu_j$ pseduo code:

## Input

- Data points: $X = x_1, x_2, \dots, x_n$ (a set of n data points)
- Number of clusters: $K$

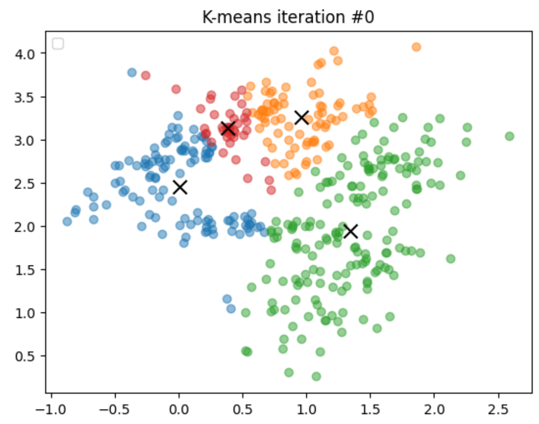
## Output:

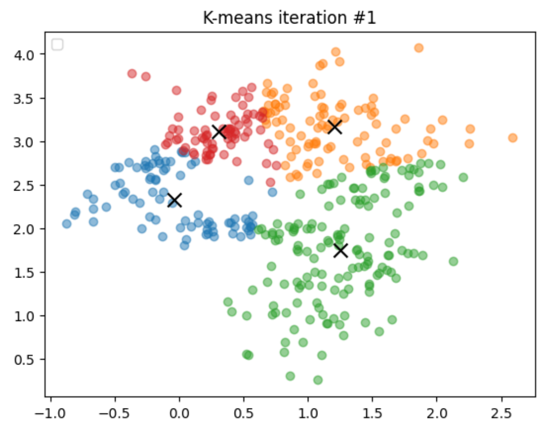
- Cluster assignments: $c_i$ (each data point is assigned to a cluster)
- Centroids: $\{C_j\}_{j=0}^{K}$ (a set of $K$ cluster centroids)

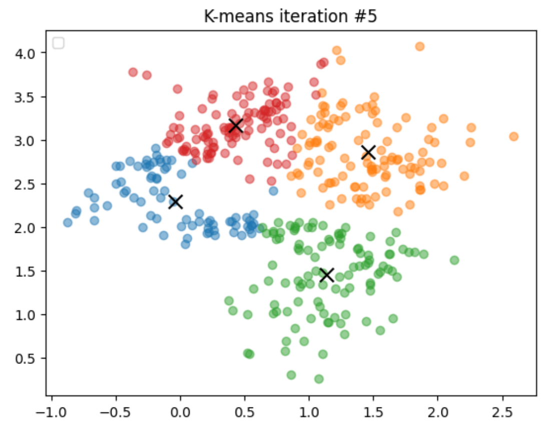
## Pseudo-Code:

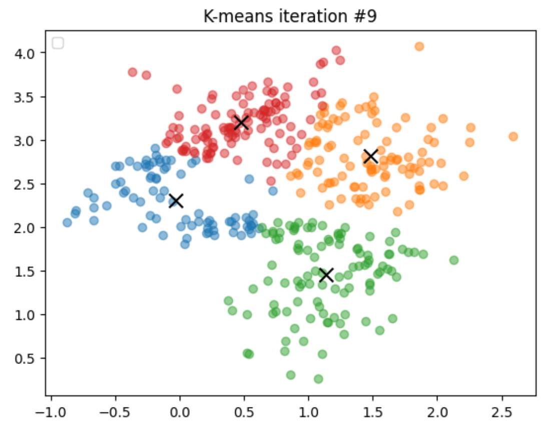
```
1. Initialization: Start by randomly selecting $K$ initial cluster centroids, denoted as $C = c_1, c_2, \dots, c_K$.
2. Iteration: -Assigning Data Points to Clusters:
    – For each data point $x_i$, find the nearest centroid and assign the point to that cluster. The assignment is based on minimizing the squared Euclidean distance:
          $c_i = arg\min_j ||x_i - c_j|\quad|^2$
• Updating Cluster Centroids: For each cluster $C_j$, update the cluster centroid by calculating the new mean of all the data points assigned to that cluster.
          $c_j = \frac{1}{|C_j|} \sum_{x_i \in C_j} x_i$
1. repeat until the centroids converge:
    – if the centroids don't change, stop the algorithm
    – else, go back to step 2
```

## initialize the model with K-means









now we can see k-mean algorithm's output, and the clusters it found. we can see that the clusters are not exactly the same as the ones we used to generate the data, but eans did an impressing job finding the centroids, plus it returned after mere 9 iterations. it is important to also remember, the computational intensity of each of those iteration is negligible compared to the MLE's iterations.

## K-means as an Initial Model:

K- means finds centroids. But, as said, EM algorithm demands initial value $\theta_0$ ie. initial values for: $\mu_j, \sigma_j^2, \phi_j$

We want to use K - means algorithm to help us generate the **entire** initial guess, not just $\mu$. k-means outputs, as its name indicates, k - vctors, representing the means, $\mu_j$ of the

predicted gaussians.
this is an asset for us, as $\mu_j$ is an important part of the model.

but we need not stop there, as K-means inner operation deals with another vital piece of information: the size of each predicted cluster. we can leverage this information to estimate $\phi_j$ for our initial guess:

$$\phi_j = P(Z = j)$$

so K-means predicts:

$$\phi_j = \frac{|cluster_j|}{|X|}$$

where $|X|$ is the number of datapoints

furthermore, after estimating $mu$, and fetching the clusters predicted by k-means nothing stops us from getting a pretty good initial guess of $\sigma_j^2$:

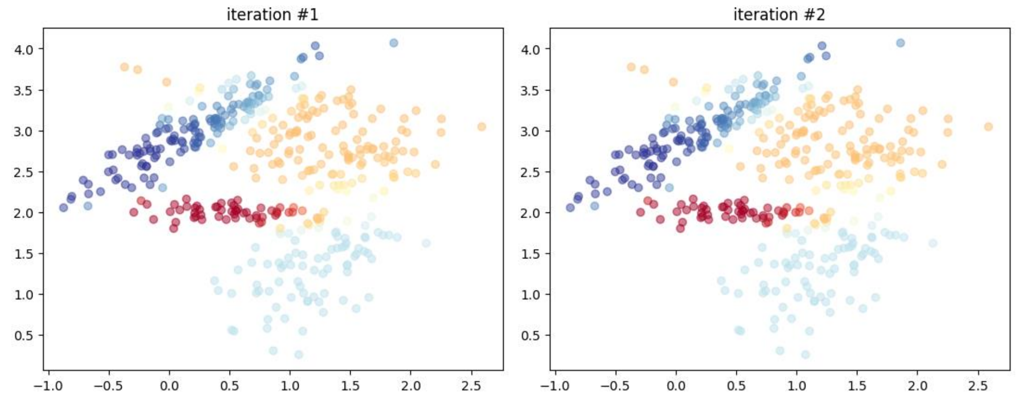
$$\left(\sigma_j^2\right)^{(0)} = \frac{1}{n}\sum_{x\in c_j}\left(x - \mu_j^{(0)}\right)^2$$

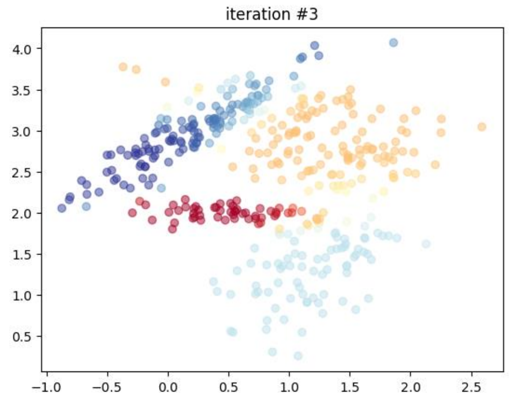
as this is a fair assumption for the value of $\sigma^2$ given the cluster.

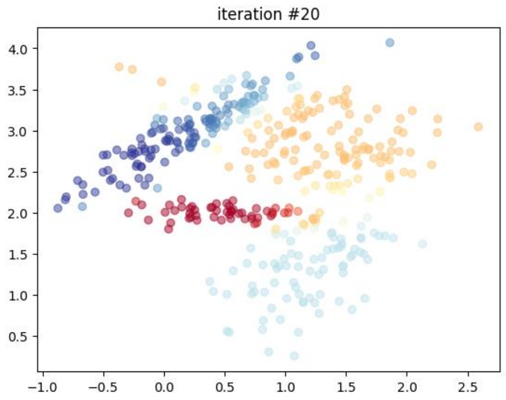
so, we get valuable information at a low computational price.

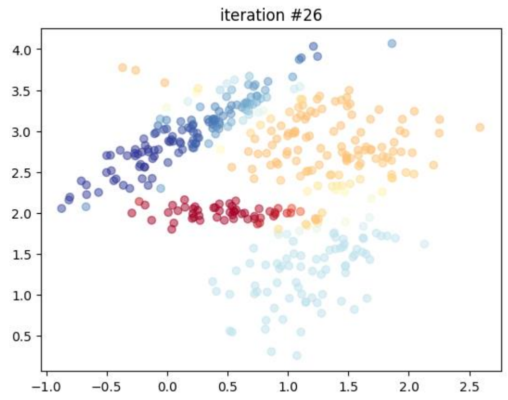
## run MLE

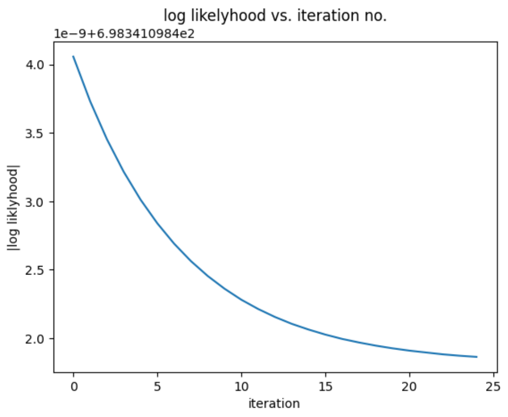
```
# run MLE
```





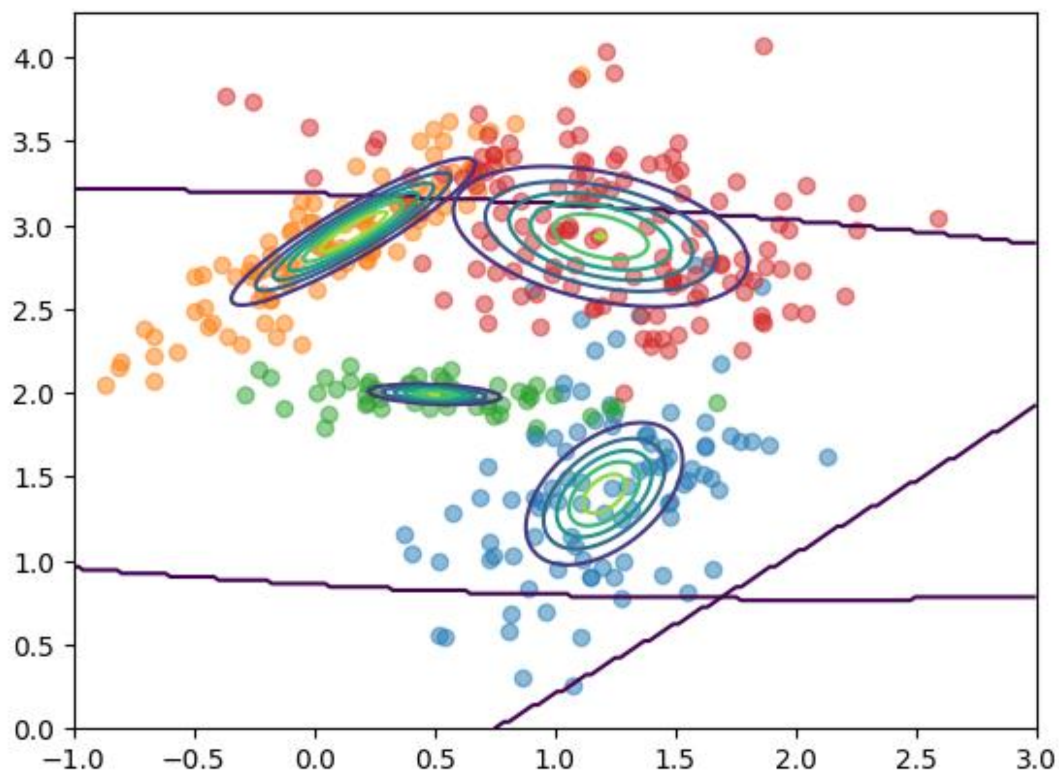






## results

the initialization proved itself to be very useful, and the algorithm converged to the right solution, after ~26 iterations. notice, in contrary to other clustering algorithms we will see in the future, it got the shaps even despite a major intersection between the 2 major clusters.


4.0
3.5
3.0
2.5
2.0
1.5
1.0
0.5
0.0
−1.0
−0.5
0.0
0.5
1.0
1.5
2.0
2.5
3.0

## Fifth experiment: False K Clustering:

In this experiment, we will deliberately provide our EM Clustering implementation with a false K value, inconsistent with how our data is formed. think of it like looking at a dataset and thinking that it can be divided $k$ normaly distributed sub groups when in fact our dataset follows $k' \neq k$ bell curves (normally distributed sub groups)

in this example, as in previous ones, we will generate our data according to mixed gaussian distribution consisting of two gaussians: $\mu_1 = \begin{bmatrix} -1 \\ -1 \end{bmatrix}$ $\mu_2 = \begin{bmatrix} 1 \\ 1 \end{bmatrix}$

$$\Sigma_1 = \begin{bmatrix} 0.8 & 0 \\ 0 & 0.8 \end{bmatrix}$$

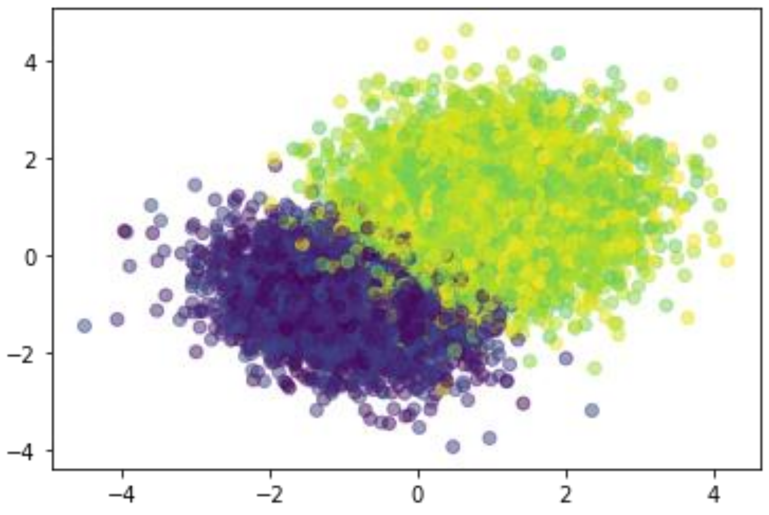
$$\Sigma_2 = \begin{bmatrix} 0.75 & -0.2 \\ -0.2 & 0.6 \end{bmatrix}$$

we are given the probability for the first label:

$$P_Z(1) = 0.7$$

so, K should equal 2, but we will set it to 3, to see the effect.

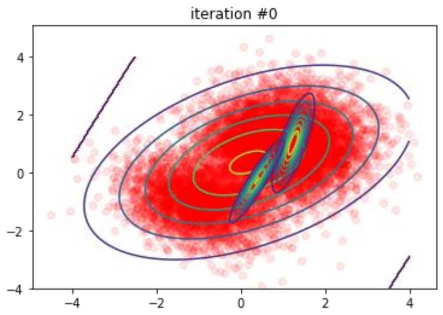
### generating a dataset

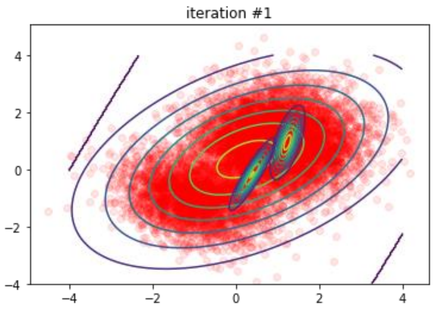


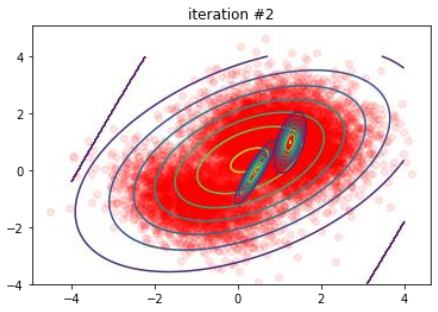
### False K MLE Clustering:

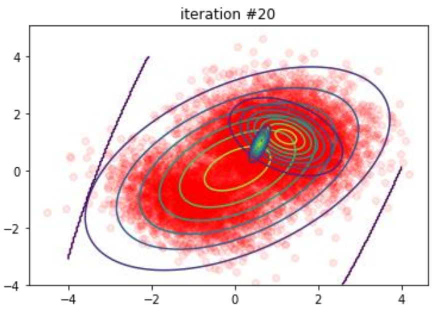
let us check how the algorithm behaves when provided with a false K value, 3 when in fact the data fits a 2 gaussians model. We will plot the contour lines of the gaussians throughout the iteration to show an interesting phenomena, difficult to spot solely by coloring the data:

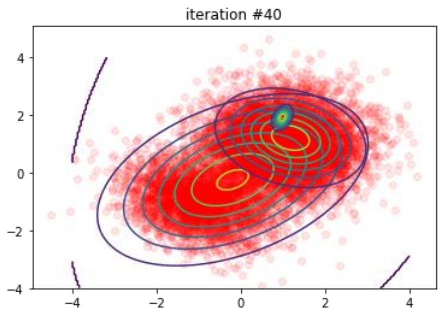
k=3
d=2

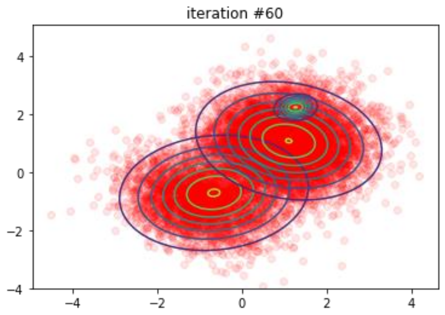

iteration #0

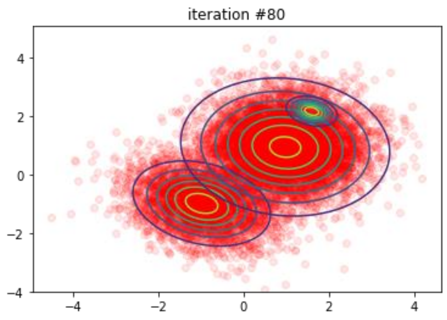

iteration #1

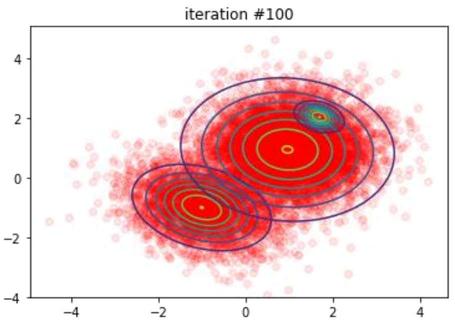

iteration #2

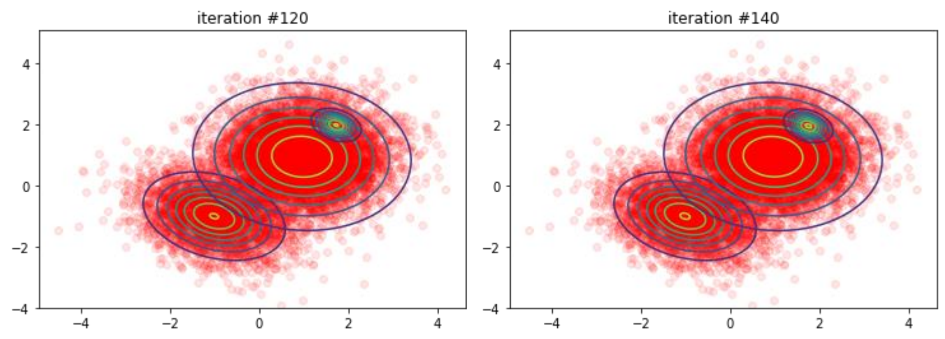

iteration #20

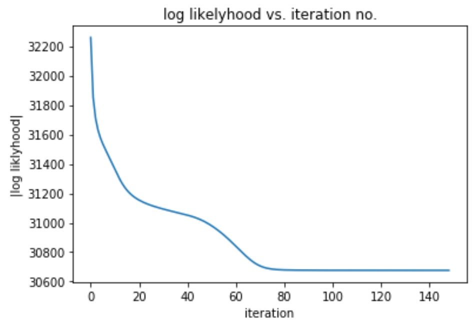

iteration #40


iteration #60


iteration #80


iteration #100





```
ϕ:  [0.29611105 0.67118295 0.03270599]

μ:  [[-1.01009774 -1.00751533]
 [ 0.96361567  0.94243613]
 [ 1.75512325  1.94522033]]

σ2:  [[[ 0.72993677 -0.19421725]
  [-0.19421725  0.6017172 ]]

 [[ 0.77015193 -0.03745781]
  [-0.03745781  0.75313727]]

 [[ 0.7969664  -0.24196099]
  [-0.24196099  0.77689287]]]

log likelyhood:  30676.511788466305
```

surprisingly, the algorithm delt pretty good with the false $K$ assignment, we can see two relatively accurate mean and variance values as well as two gaussians that fit the shape of the original data generating gaussians. we see a smaller one blending in one of the formers, the amplitude of both sums up to one of the true amplitudes, $\phi_2$ , so the extra ‘parasitic’ cluster "steals" from its host's amplitude, but merely effects it's $\mu$ and $\sigma^2$ computationally, a significant cost was added, so **mischoosing k wasn't a good idea, yet not such a big disaster.**

## sixth experiment: False K Clustering with K means as Initial model:

In this experiment, we will, again, deliberately provide our EM Clustering implementation with a false K value, inconsistent with the dataset nature, **using the same dataset as in the previous experiment**. This time we will use the K-means algorithm as an initializer, to see the difference in results.

Again, we will use the same two gaussians as before:

$$\mu_1 = \begin{bmatrix}-1\\-1\end{bmatrix} \mu_2 = \begin{bmatrix}1\\1\end{bmatrix}$$

$$\Sigma_1 = \begin{bmatrix}0.8 & 0\\0 & 0.8\end{bmatrix}$$

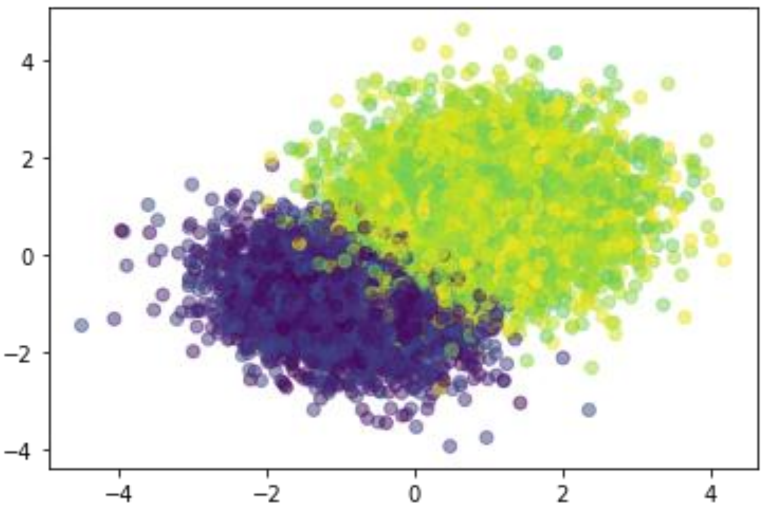
$$\Sigma_2 = \begin{bmatrix}0.75 & -0.2\\-0.2 & 0.6\end{bmatrix}$$

we are given the probability for the first label:

$$P_Z(1) = 0.7$$

so, K should equal 2, but we will set it to 3, to see the effect.

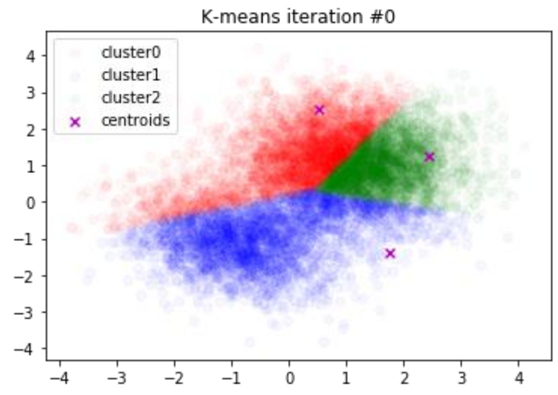
### generating a dataset

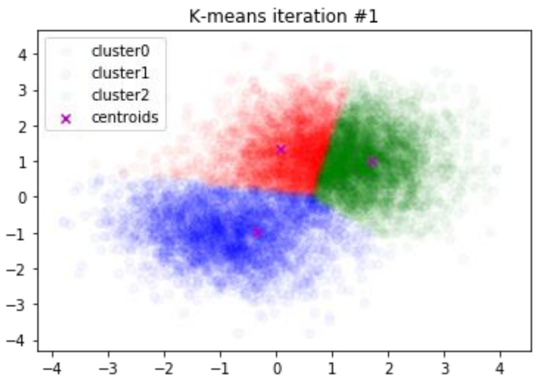


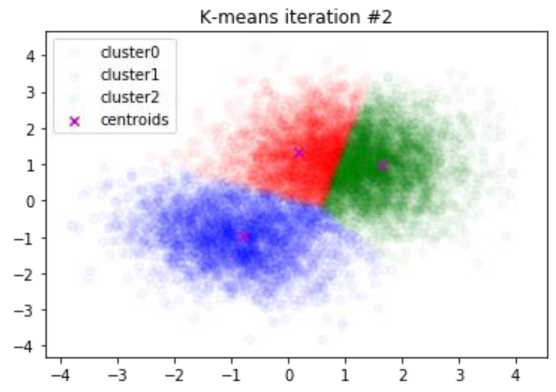
### False K means Clustering:

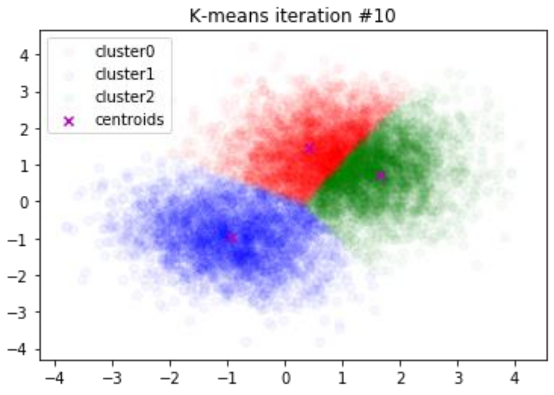
We will initialize the algorithm with the K-means initialization schene, starting from here we will provide it with a $k' \neq k$ value

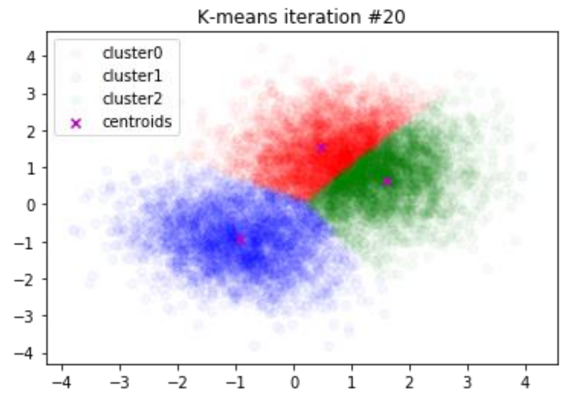
```
k=3
d=2
```

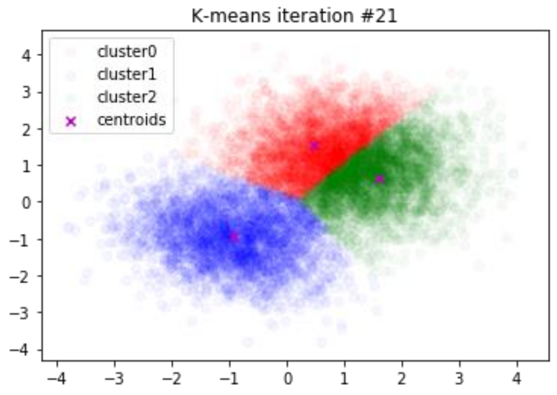

K-means iteration #0
cluster0
cluster1
cluster2
centroids

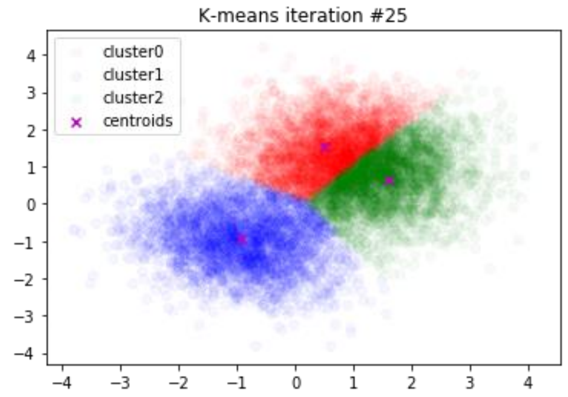

K-means iteration #1
cluster0
cluster1
cluster2
centroids


K-means iteration #2
cluster0
cluster1
cluster2
centroids


K-means iteration #10
cluster0
cluster1
cluster2
centroids


K-means iteration #20
cluster0
cluster1
cluster2
centroids

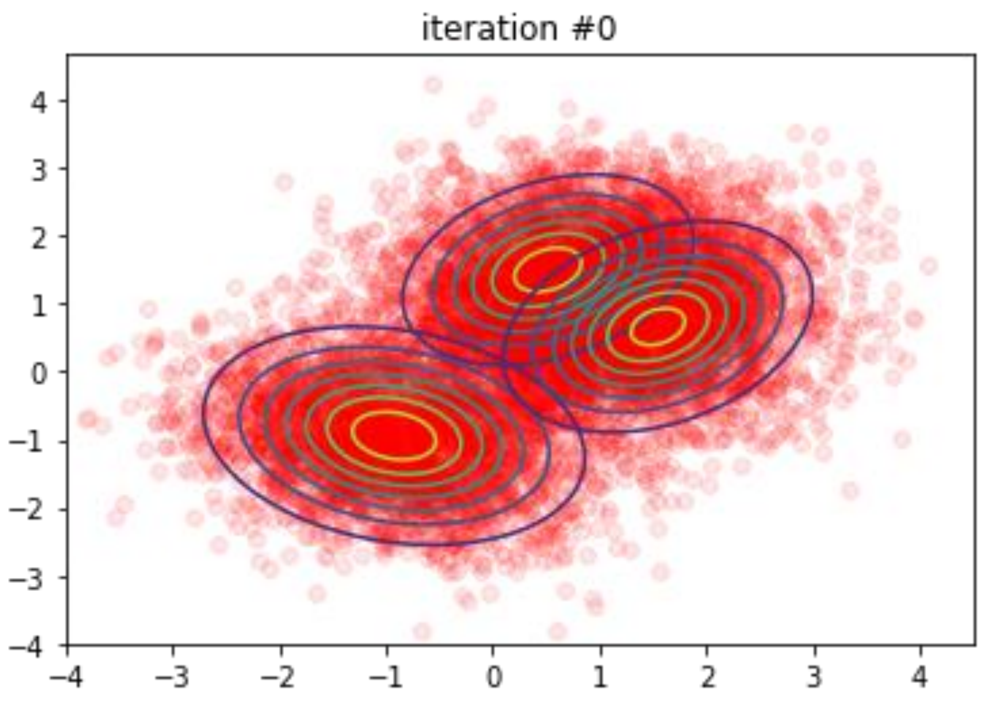

K-means iteration #21
cluster0
cluster1
cluster2
centroids


K-means iteration #25
cluster0
cluster1
cluster2
centroids

Seems that the algorithm converges to cover most of one of the clusters and assigns them together (marked in blue) and split the other cluster in half. Let's see if it keeps this partition or changes it in the MLE phase.

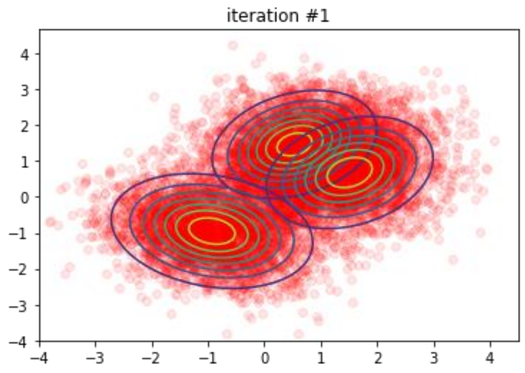
## False K MLE Clustering:

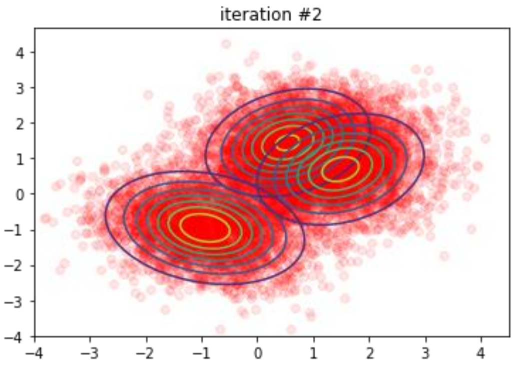
let us check how the algorithm behaves when provided with a false K value, 3 when in fact the data fits a 2 gaussians model, this time using the K-means initialization scheme. We will once again plot the contour lines of the gaussians throughout the iteration to show another very interesting phenomena, difficult to spot solely by coloring the data:

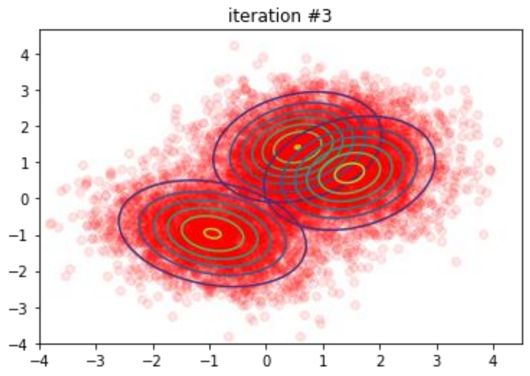


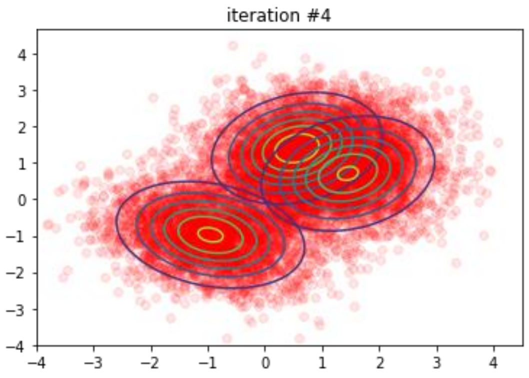
```
ϕ:  [0.3200364  0.33883855 0.34112505]

μ:  [[ 0.51715191  1.49717561]
     [-0.92425612 -0.94422572]
     [ 1.54712998  0.66066745]]

σ2:  [[[ 0.51052674  0.16245569]
       [ 0.16245569  0.54103765]]

     [[ 0.76801268 -0.13946314]
      [-0.13946314  0.62019181]]

     [[ 0.51063136  0.16381481]
      [ 0.16381481  0.58758656]]]
```

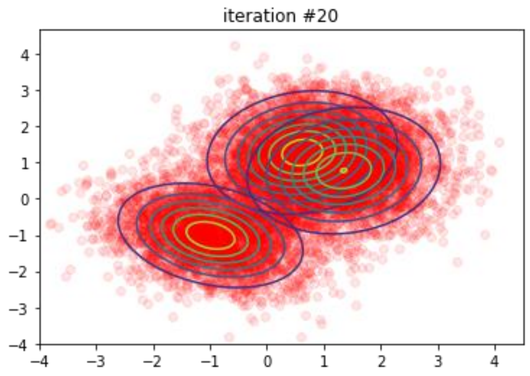
* these are already some pretty accurate results for the first cluster distribution provided by K-Means initializer!

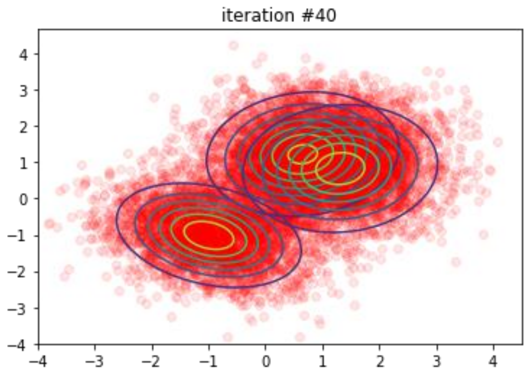

iteration #1

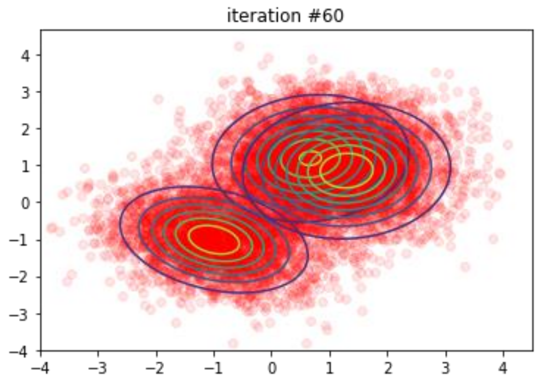

iteration #2

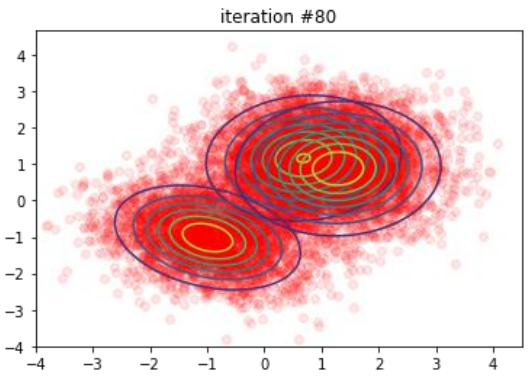

iteration #3

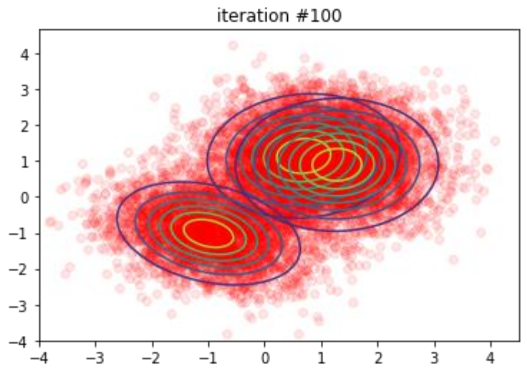

iteration #4

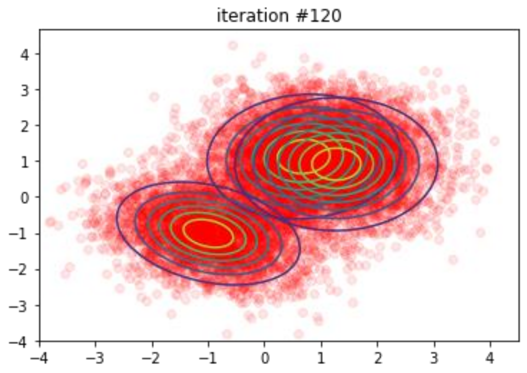

iteration #20

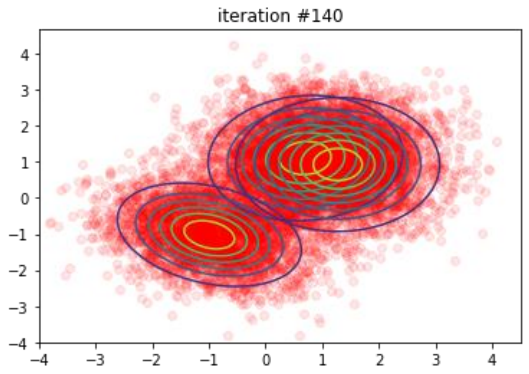

iteration #40

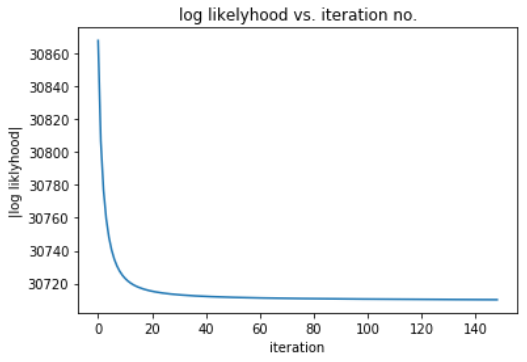

iteration #60


iteration #80









## Conclusion

so, once more, the algorithm did not converge at under 150 iterations, but an amazing result emerged, one of the clusters was provided to us by the K-Means initializer,  the other two converged to overlap and cover the second cluster!

The K-means algorithm provide us with a good initial model, right on the start one gaussian has relatively accurate parameters, the other is "cut in half". From there, EM "piles up" the oter two on each other, making them two copies of the same gaussian, with roughly the same parameters $\mu, \sigma^2$ as the corresponding original gaussian, and half the amplitude $\phi$. the two sum up to the original distribution. As surprising as it is, enven though we provided an unmaching parameter to the model, the algorithm performed well nonetheless