ECE 421 Assignment 3

David Larsson, Robert Agnoletto

March 2019

note: 50/50 contribution between partners.

1 K-means

1.1 Learning K-means

1.1.1 K = 3

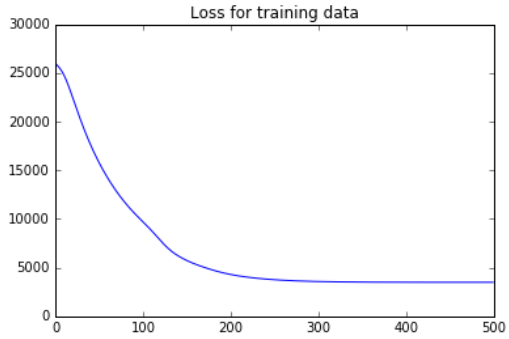

Figure 1: The loss of the training data plotted over the number of epochs of the data using K=3 different means.

1.1.2 K = 1, 2, 3, 4, 5

K	final loss of training data	percentage in each class
1	25589.0	(1)
2	6332.3	(0.49078, 0.50922)
3	3489.2	(0.37543, 0.38413, 0.24044)
4	2355.7	(0.13589, 0.37363, 0.37018, 0.12029)
5	1970.3	(0.35158, 0.36283, 0.10829, 0.08445, 0.09285)

Table 1: The final loss of Training data and the percentage of validation data in each class using the K-means algorithm for different number of means, K

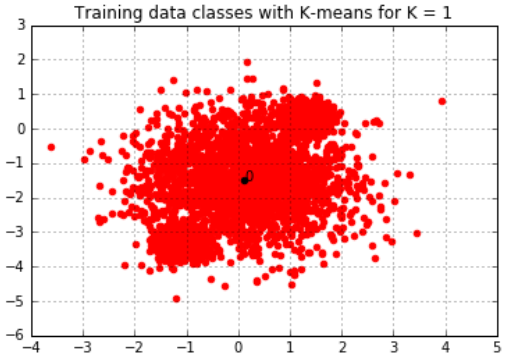

Figure 2: The classification of the training data points using K=1 means and the black dot represent the mean.

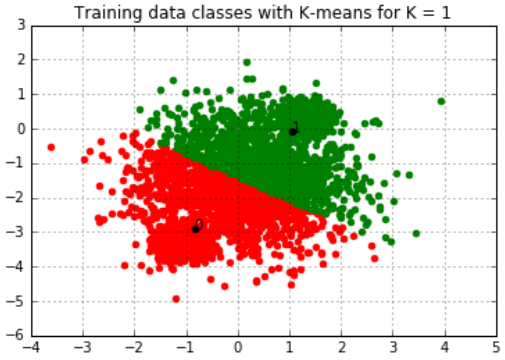

Figure 3: The classification represented by different colors of the training data points using K=2 means and the black dots represent the means.

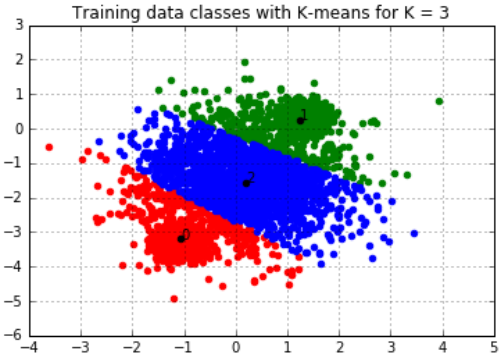

Figure 4: The classification represented by different colors of the training data points using K=3 means and the black dots represent the means.

Figure 5: The classification represented by different colors of the training data points using K=4 means and the black dots represent the means.

Figure 6: The classification represented by different colors of the training data points using K=5 means and the black dots represent the means,

Looking at the table 1 we can observe that the final loss keeps decreasing when increasing the number of means K which is expected and you can also observe that for all values of K the percentage of data in each classes seems to be fairly evenly divided or at least significant. However looking at the figures 2, 3, 4, 5 and 6 we can see that actually the case with five classes shown in 6 perhaps can be one class to much since it just looks like 5 which has four classes but with the 4:th class divided into two. Therefore based on this information a choice of K=4 which means four classes seems reasonable.

1.1.3 Validation

K	final loss of validation data	percentage in each class
1	12870.1	(1)
2	2960.7	(0.51815, 0.48185)
3	1629.3	(0.3735, 0.3978, 0.2286)
4	1054.5	(0.1266, 0.1197, 0.3657, 0.3879)
5	900.9676	(0.0738, 0.3639, 0.1077, 0.3843, 0.0702)

Table 2: The final loss of the validation data and the percentage of validation data in each class using the K-means algorithm for different number of means, K

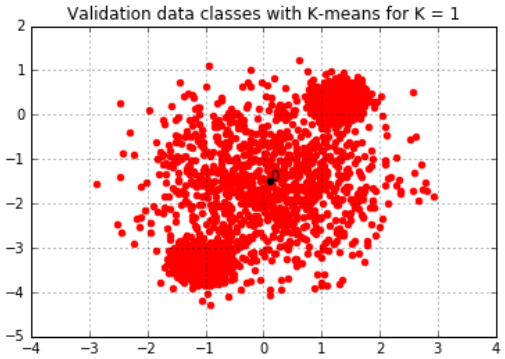

Figure 7: The classification represented by different colors of the validation data points using K=1 mean and the black dot represent the mean

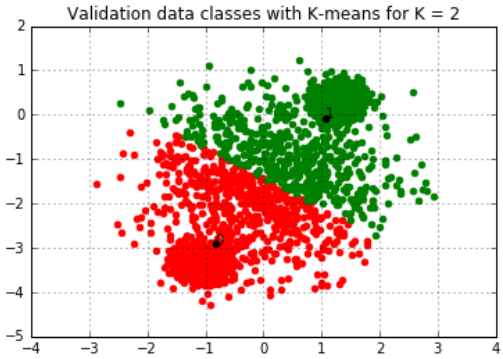

Figure 8: The classification represented by different colors of the validation data points using K=2 means and the black dots represent the means

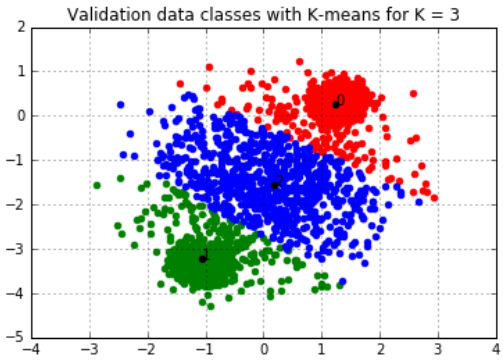

Figure 9: The classification represented by different colors of the validation data points using K=3 means and the black dots represent the means

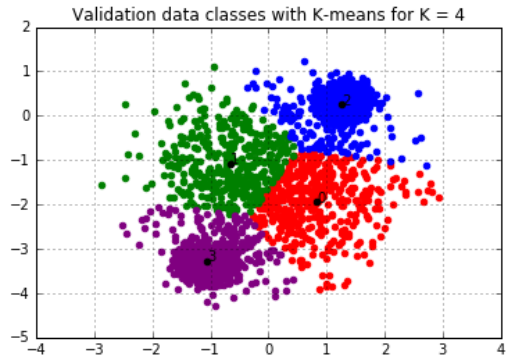

Figure 10: The classification represented by different colors of the validation data points using K=4 means and the black dots represent the means

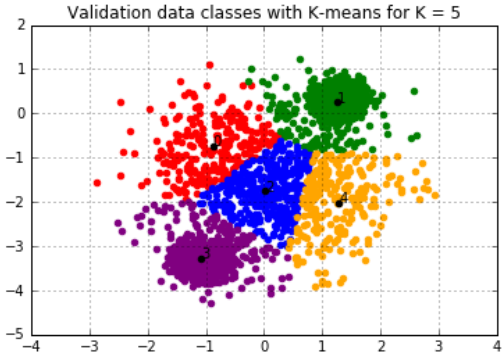

Figure 11: The classification represented by different colors of the validation data points using K=5 means and the black dots represent the means

Using the validation data and calculating the loss shown in figure 2 for different amount of classes K, we can see that just as for the training data the final loss decreases with increasing K. From this one would then assume that actually five classes might actually be better then using four classes. However the loss does not decrease as much increasing the number of classes from four to five as for increasing three to four. But looking at the classification plots in figures 7, 8, 9, 10 and 11 we can see that the means have converged so that this split the data into five classes that seems to make more sense. Therefore five classes seems better. Notice that the means in the validation is still trained using the training data, just from a different run then the one showed in 6 and therefore the initialization of the mean was different.

2 Mixtures of Gaussians

2.1 The Gaussian cluster mode

2.1.1 Log Probability Density Function

$$log(\mathcal{N}(x;\mu_k,\sigma_k^2)) = log(\frac{1}{\sqrt{2\pi\sigma_k^2}}e^{-\frac{(x-\mu_k)^2}{2\sigma_k^2}})$$
$$log(\mathcal{N}(x;\mu_k,\sigma_k^2)) = -\frac{(x-\mu_k)^2}{2\sigma_k^2} - \frac{log(2\pi\sigma_k^2)}{2} \quad (1)$$

Now that we have the equation for the log probability density function for cluster k, we now need to vectorize it so that we can calculate the pdf for all K clusters, and plugging in all the values of x in each cluster. This will result in a NxK matrix. See the python implementation of this vectorized log pdf below.

```
def log_GaussPDF(X, mu, sigma):
    dist = distanceFunc(X,mu)
    exp = -tf.divide(tf.divide(dist,tf.reshape(sigma,[1,-1])),2)
    fact = tf.sqrt(tf.multiply(2*m.pi, tf.reshape(sigma,[1,-1])))
    log_pdf = exp - tf.log(fact)
    return log_pdf
```

Here the same distance function from the K-means part of the assignment was used. This gives us back an NxK array of $(x_n - \mu_k)^2$ for all n and k. Next, we calculate the first term in (1) by dividing the distance by sigma and 2. We reshape the Kx1 sigma matrix to be 1xK so that we can make use of broadcasting and have it divide all n points. We do the same thing for the sigma in the second term so that they can once again be combined with broadcasting so the return value remains NxK.

2.1.2 Log Probability of cluster variable z given x

$$log(P(z|x)) = log\left(\frac{\pi_z\mathcal{N}(x;\mu_z,\sigma_z^2)}{\sum_{j=1}^{K}\pi_j\mathcal{N}(x;\mu_j,\sigma_j^2)}\right)$$
$$log(P(z|x)) = log(\pi_z) + log(\mathcal{N}(x;\mu_z,\sigma_z^2)) - log\left(\sum_{j=1}^{K}\pi_j\mathcal{N}(x;\mu_j,\sigma_j^2)\right)$$
$$log(P(z|x)) = log(\pi_z) + log(\mathcal{N}(x;\mu_z,\sigma_z^2)) - logsumexp(log(\underline{\pi}) + log(\underline{\mathcal{N}})$$

Since we are working in the log domain, adding/subtracting is equivalent to multiplying/dividing the original numbers and then taking the log.

Therefore, if we want to sum the original numbers, we cannot use the standard tf.reduce_sum as it will do the equivalent of multiplying all the original values and then taking the log. The provided helper function logsumexp takes care of this and returns the log of the sum of the original values.

```
def log_posterior(log_PDF, log_pi):
    num = log_PDF + tf.reshape(log_pi, [1,-1])
    denom = hlp.reduce_logsumexp(num,1,True)
    return num - denom
```

In our code, we reshape log_pi so that we can broadcast when adding a 1xK matrix an NxK one. In the end we get the numerator to be an NxK matrix, and the denominator to be a Nx1 matrix. This is because the sum in the denominator is the same for all clusters. Once again we are able to make use of broadcasting to do the division through the subtraction of the log values.

2.2 Learning the MoG

$$P(X) = \prod_{n=1}^{N} \sum_{k=1}^{K} \pi_k \mathcal{N}(x_n; \mu_k, \sigma_k^2) \tag{2}$$

$$\mathcal{L}(\mu, \sigma, \pi) = -log(PX))$$

$$\mathcal{L}(\mu, \sigma, \pi) = -\sum_{n=1}^{N} log\left(\sum_{k=1}^{K} \pi_k \mathcal{N}(x_n; \mu_k, \sigma_k^2)\right)$$

$$\mathcal{L}(\mu, \sigma, \pi) = -\sum_{n=1}^{N} logsumexp(log(\underline{\pi}) + log(\underline{\mathcal{N}}(x_n; \mu, \sigma^2))) \tag{3}$$

2.2.1 K = 3

First, we implemented the loss function (3) with the following code.

```
def neg_log_likelihood(log_PDF, log_pi):
    loss = -tf.reduce_sum(hlp.reduce_logsumexp(log_PDF + tf.reshape(log_pi,
        ↪ [1,-1])))
    return loss
```

Then we initialized the variables and set up the ADAM optimizer to minimize the loss function above.

Listing 1: Building MoG model

```
def gMM():
    X = tf.placeholder(tf.float32,[None, dim], name="X")
    mu = tf.get_variable('mean',dtype = tf.float32,shape = [K,dim], initializer =
        ↪ tf.truncated_normal_initializer(stddev=2))

    phi = tf.get_variable('stdDev',dtype = tf.float32,shape = [K,1],initializer =
        ↪ tf.truncated_normal_initializer(mean=1,stddev=0.25))
    sigma = tf.pow(phi,2)

    psi = tf.get_variable('logPiProb',dtype = tf.float32,shape = [K,1],
        ↪ initializer = tf.truncated_normal_initializer(mean=1,stddev=0.25))
    log_pi = hlp.logsoftmax(psi)

    log_PDF = log_GaussPDF(X, mu, sigma)
    log_rnj = log_posterior(log_PDF, log_pi)
    lossfunc = neg_log_likelihood(log_PDF, log_pi)
    belong = tf.arg_max(log_rnj,dimension = 1)
    optimizer = tf.train.AdamOptimizer(learning_rate = 0.05, beta1=0.9, beta2
        ↪ =0.99, epsilon=1e-5)
    train = optimizer.minimize(loss=lossfunc)
    return X,mu,sigma,lossfunc,log_pi,log_PDF,log_rnj,train,belong
```

For mu, we found that since there are only 6 values, it was possible to get a bad starting point with the random values. We wanted to increase the standard deviation because if all three starting points were too close to (0,0) it would sometimes ignore one or two of the points and simply lower its π value to nearly zero. This would essentially treat it like K=1 or K=2. We found raising the standard deviation helped and generally only one or two starting points (and picking the best result) yielded good results.

For sigma, we found that setting it equal to e^{ϕ} made it to unstable as small changes in ϕ could result in huge changes in sigma. This often resulted in one of the clusters' standard deviation shooting up and encapsulating all of the points. By replacing it with ϕ^2 we were able to stabilize it while still ensuring that sigma stays positive. Considering the problems we had with sigma, we wanted to make sure each cluster had a fair chance so we initialized phi with a mean of 1 and made use of the truncated normal initializer with a low standard deviation.

For pi, we found that the suggested log-softmax function worked well to ensure the constraint without causing stability issues. For the initial values, again we wanted to ensure each cluster had a fair chance so we used the same initializer as for sigma with truncated normal and small standard deviation.

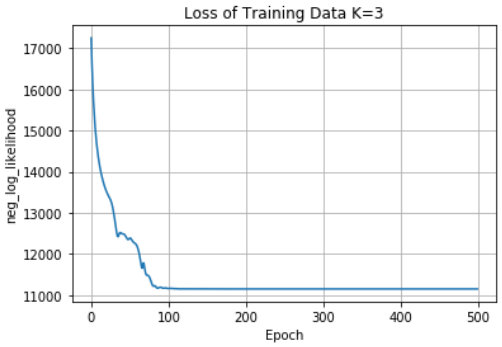

Figure 12: The negative log likelihood of the training data during 500 epochs with K=3

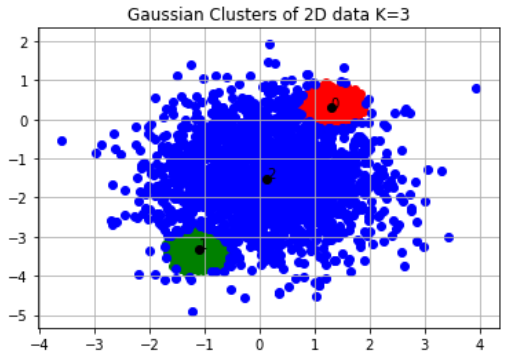

Figure 13: The 2D data plotted and coloured according to its most likely cluster as well as the cluster centres for K=3

Cluster	μ	σ^2	$log(\pi)$
1	(1.295,0.308)	0.0735	-1.2461
2	(0.123,-1.513)	2.5865	-0.8346
3	(-1.105,-3.310)	0.0720	-1.2788

Table 3: The trained parameters after 500 epochs for K=3.

The following is the code used to run the optimizer at each step.

```
epochs = 500
Loss = np.zeros(epochs)
for step in range(0, epochs):
    _,current_loss = sess.run([train,lossfunc],feed_dict = {X:train_data})
    Loss[step] = current_loss
    if step % 10 == 0:
        print(step,current_loss)
```

2.2.2 K = 1, 2, 3, 4, 5

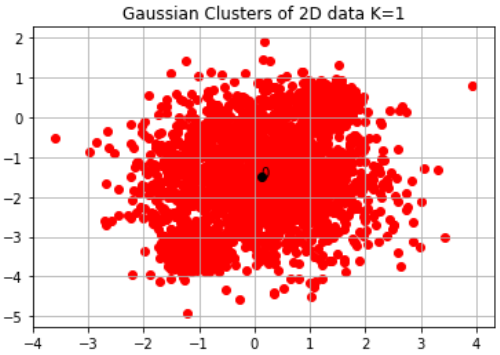

Figure 14: Cluster plot for K=2

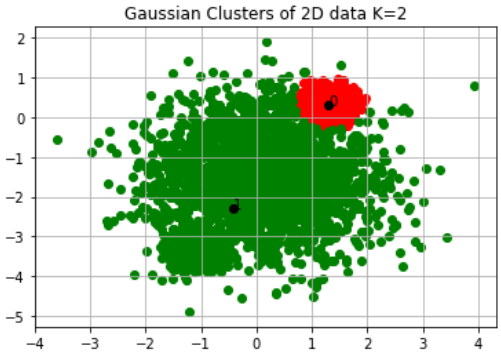

Figure 15: Cluster plot for K=2

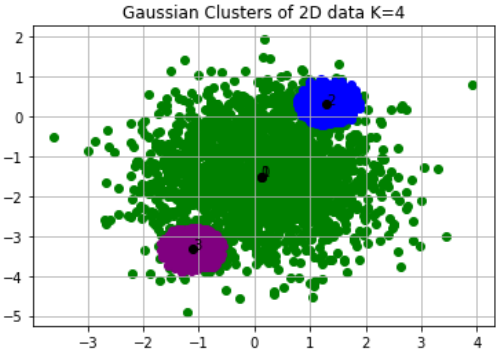

Figure 16: Cluster plot for K=4

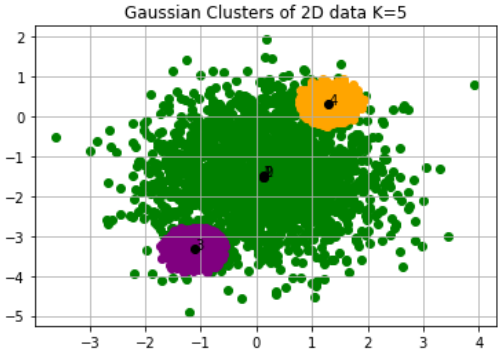

Figure 17: Cluster plot for K=5

Number of Clusters	1	2	3	4	5
Validation Loss	6980.85	6146.3	5522.063	5521.882	5521.937

Table 4: The final loss of the validation data as a function of number of clusters after 500 epochs.

Looking at the values in table 4 we can see that increasing the number of clusters from K=1 decreases the validation loss until K=3, at which point it levels out at around 5522.

Looking at the cluster plots (the K=3 plot is in the previous section and figure 13) we can see that when we increase K past 3 there is not much change in the clusters. The additional clusters end up mirroring the values of the centre cluster but with small π values and no members. Taking both these observations into account it is clear that 3 is the ideal value for K as smaller values have a higher loss and larger values still converge to 3 clusters.

2.2.3 K-means With 100-D Data

K	Validation data significant classes	Training error	Validation error
5	4	246954	123296
10	5	143581	71857
15	5	143530	71801
20	5	143590	71833
30	6	141050	70765

Table 5: The number of classes with percentage of data points different from zero using the data from data.100D and increasing the number of classes using the K-means Algorithm. Also the final training error and validation error for those cases.

2.2.4 MoG With 100-D Data

In order to use the previous model we tailored to the 2 dimension data for this new set of data with many more dimensions, we had to make some small changes. Again, we were faced with a similar problem of our model converging to all data points belonging to a single class (or two in some case) regardless of the K chosen. To investigate this, for K=5, we graphed the pi and sigmas of each cluster as a function of epochs during training. We found again like before there was an issue with the sigma of a certain cluster growing huge too fast and overtaking the other clusters which were left with no members.

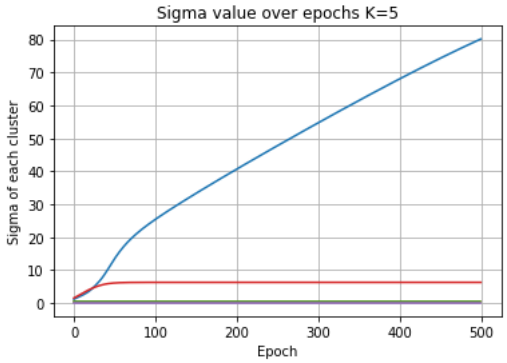

Figure 18: Sigma Plot for K=5, D=100

To try and further stabilize sigma we changed our code from Listing 1 in 12 from:

sigma = tf.pow(phi,2) to sigma = tf.abs(phi)

This change was an improvement, as it would sometimes result in a single large sigma taking over (but not getting as large) to sometimes allowing other clusters to exist.

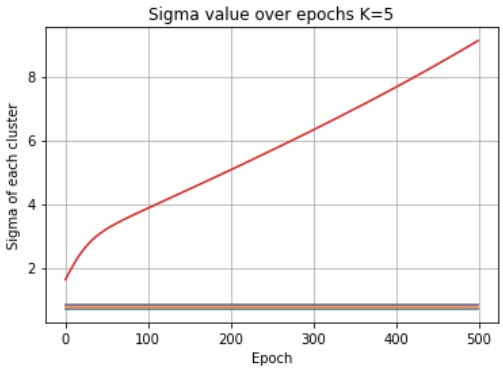

Figure 19: Sigma Plot using tf.abs. Example when one takes over

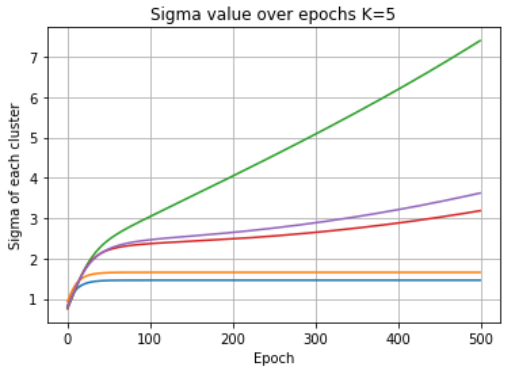

Figure 20: Sigma Plot using tf.abs. Example when multiple clusters have members

What we observed from this table was that all the clusters that ended up with members tended to a standard deviation of around 2-6 which was much large than what we initialized the values to. What was likely happening was that whichever cluster started increasing its sigma earlier and faster had the upper hand and would sometimes engulf the other clusters. To try and solve this problem we intialized our sigmas to higher values. So before where we had:

```
phi = tf.get_variable('stdDev',dtype = tf.float32,shape = [K,1],initializer = tf.
    ↪ truncated_normal_initializer(mean=1,stddev=0.25))
sigma = tf.pow(phi,2)
```

We now have:

```
phi = tf.get_variable('stdDev',dtype = tf.float32,shape = [K,1],initializer = tf.
    ↪ truncated_normal_initializer(mean=4,stddev=0.5))
sigma = tf.abs(phi)
```

This resulted in much more stable and predictable behaviour and was used for all the results in table 6. In addition to this, these changes to the model did not decrease the models' effectiveness at clustering the 2D data in section 1.1.2.

K	Validation data significant classes	Training error	Validation error
5	4	30156	15119
10	5	29394	14727
15	5	29240	14648
20	5	29278	14649
30	5	28717	14374

Table 6: The number of classes with percentage of data points different from zero using the data from data.100D and increasing the number of classes using the MoG Algorithm. Also the final training error and validation error for those cases.

2.2.5 K-means vs MoG D=100

From these results it appears that the data set has 5 clusters since almost all trials resulted in 5, and the increase to 6 that happened with k-means in table 5 did not result in an decrease in loss like it did with the increase from 4 to 5 seen in the same table.

For the Mixture of Gaussian models, the decrease in loss seen in table 6 between K=5 and K=10 was much smaller than it was for K-means. This is despite the fact that it caused the same change in number of classes with non-zero members. One reason for this is that the MoG model has more parameters than k-means. This means it is more flexible and can cope with the wrong number of clusters better. For example, having one less cluster can be offset by increasing the standard deviation of a nearby cluster. This will result in less loss than the same scenario in k-means which cannot do that.

Another thing to notice, is that when K was set to 30 (which is much higher than needed based on our observations) the k-means algorithm used 6 of the clusters, and had a small drop in loss compared to 10,15 and 20. The MoG model on the other hand did not increase the number of clusters it made use of, but still had a similar drop in loss (small, but noticeable). One possible reason for why it still experienced a reduction in loss despite making use of the same number of clusters is that it had more starting clusters to choose from and was able to converge to a slightly better minimum. However, this is at a cost of processing power and we could achieve the same thing with and K larger than 5 as long as we run it multiple times with different initial values and then choose the best result.