ECE421 - Winter 2022
University of Toronto

Assignment 3: Unsupervised Learning and Probabilistic Models

Due date: Monday, April 4, 2022

Student name: Mingyu Zheng

Student number: 1003797661

1. K-means

1.1 Learning K-means

1. Implement distance_func():

Derivation:

$$\mathcal{L}(\boldsymbol{\mu}) = \sum_{n=1}^{N} \min_{k=1}^{K} \|\mathbf{x}_n - \boldsymbol{\mu}_k\|_2^2,$$

= (x_n)^2 - 2 * x_n * miu_k + (miu_k)^2

Implementation:

```
# Distance function for K-means
def distance_func(X, mu):
    """ Inputs:
          X: is an NxD matrix (N observations and D dimensions)
          mu: is an KxD matrix (K means and D dimensions)

        Output:
          pair_dist: is the squared pairwise distance matrix (NxK)
    """
    # TODO
    X_sqr = tf.reduce_sum(tf.square(X), axis=1, keepdims=False, name=None)
    X_sqr = tf.reshape(X_sqr, [-1, 1])

    mu_T = tf.transpose(mu)
    X_mu = tf.matmul(X, mu_T)

    mu_sqr = tf.reduce_sum(tf.square(mu), axis=1, keepdims=False, name=None)
    mu_sqr = tf.reshape(mu_sqr, [1, -1])

    pair_dist = X_sqr - 2*(X_mu) + mu_sqr

    return pair_dist
```

Numerical result:

```
Final mu is:
 [[ 1.251753    0.24656856]
 [-1.0559269  -3.2431977 ]
 [ 0.12183347 -1.5230418 ]]
Final training loss is:  5110.94580078125
```

Plot:

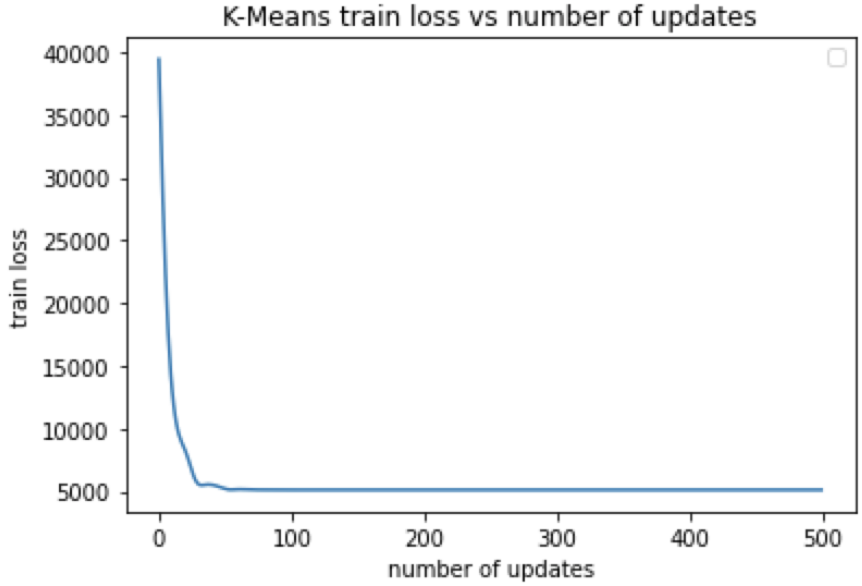
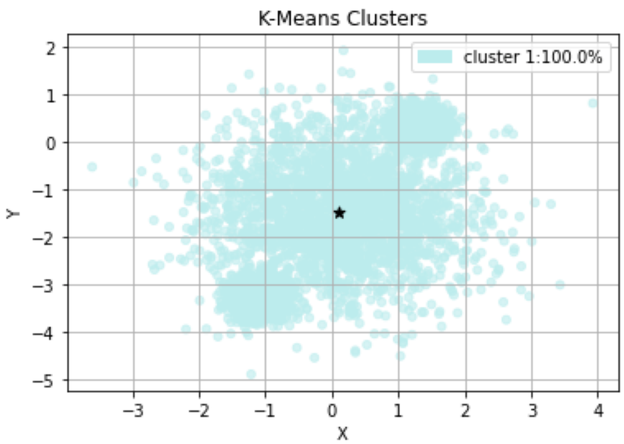


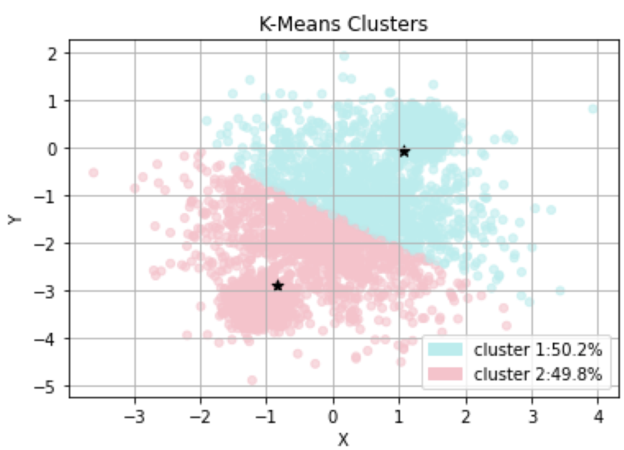
Figure 1. K-means train loss vs number of updates when K=3, using Adam optimizer: learning_rate = 0.1, beta1 = 0.9, beta2 = 0.99, epsilon = 1e-5

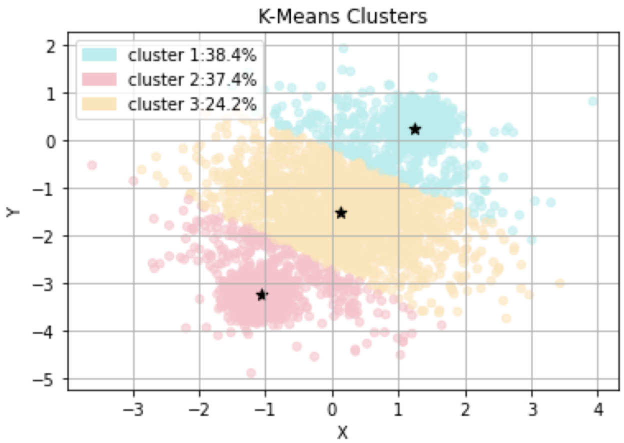
2. ⅓ validation data, for K = 1, 2, 3, 4, 5:
   Train a K-means model, including 2D scatter plot.

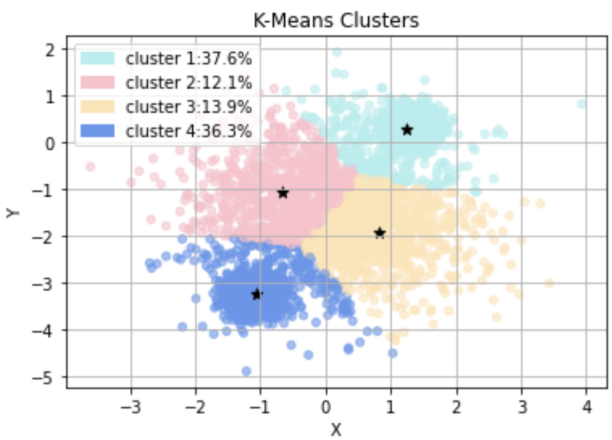
(a) + (b)

Table 1: Train for K-means model for K = [1, 5], tran loss vs number of updates ploat and 2D scatter plot

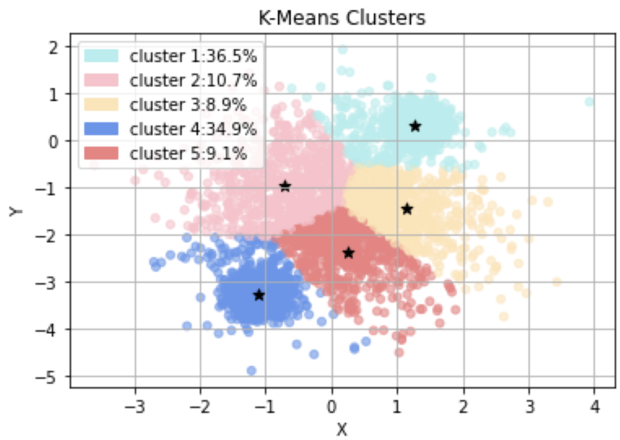
K	2D scatter plot
1	 `Class 1 percentage is:  100.0 %`

2	
	`Class 1 percentage is:  50.24748762561872 %`
	`Class 2 percentage is:  49.75251237438128 %`

<table>
<tr>
<td>3</td>
<td>


Class 1 percentage is: 38.3980800959952 %
Class 2 percentage is: 37.36313184340783 %
Class 3 percentage is: 24.23878806059697 %
</td>
</tr>
<tr>
<td>4</td>
<td>


Class 1 percentage is: 37.6181190940453 %
Class 2 percentage is: 12.14939253037348 %
Class 3 percentage is: 13.949302534873256 %
Class 4 percentage is: 36.283185840707965 %
</td>
</tr>
</table>

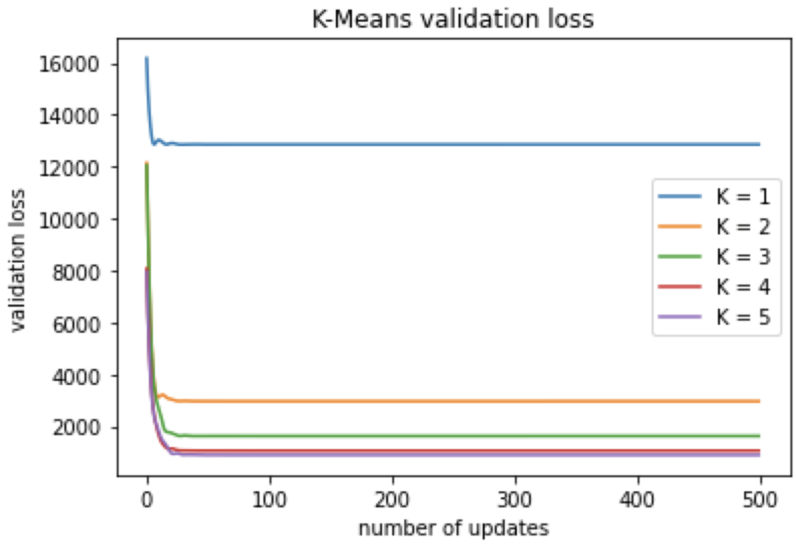
5	 Class 1 percentage is: 36.52317384130794 % Class 2 percentage is: 10.679466026698664 % Class 3 percentage is: 8.87955602219889 % Class 4 percentage is: 34.858257087145645 % Class 5 percentage is: 9.059547022648868 %

(c)

Table 2: Summary of percentage of training data points belonging to each of the K clusters

K/ Class	Class 1	Class 2	Class 3	Class 4	Class 5
K = 1	100%	N/A	N/A	N/A	N/A
K = 2	50.247%	49.752	N/A	N/A	N/A
K = 3	38.398%	37.363%	24.239%	N/A	N/A
K = 4	37.618%	12.149%	13.949%	36.283%	N/A
K = 5	36.523%	10.679%	8.8796%	34.858%	9.0595%

(d) loss function over validation data



Comment:
Based on the scatter plots, the best number of clusters to use is K = 3. Although when K is bigger, the training loss and validation loss are smaller. However, K can go to as much as the number of data points, and the loss can be zero at that time. From the scatter plots, we can notice that when K equals 3, the percentages of the three classes are evenly distributed. In addition, there isn’t much difference between the validation loss of K=3 and K=4 or 5, compared with K = 1 and 2. Therefore, in conclusion, among the 5 numbers provided, 3 is the best number of clusters to use.

2. Mixtures of Gaussians

2.1 The Gaussian cluster mode

1. Implement log_gauss_pdf, use distance_func

   Derivation:

   Since N(x | mu_k^2, sigma_k^2) = P(x | mu_k^2, sigma_k^2),
   log(N(x | mu_k^2, sigma_k^2)) = log(P(x | mu_k^2, sigma_k^2))

$$= \log\left(\frac{1}{\sqrt{2\pi\sigma^2 k}} e^{\frac{-(x-\mu k)^2}{2\sigma k^2}}\right)$$

$$= \log\left(\Pi dim \frac{1}{\sqrt{2\pi\sigma^2 k}}\right) + \frac{-(x-\mu k)^2}{2\sigma k^2}$$

$$= \frac{dim}{2}\, log\left(\frac{1}{2\pi\sigma k^2}\right) - \frac{(x-\mu k)^2}{2\sigma k^2}$$

   Implementation:

```python
def log_gauss_pdf(X, mu, sigma):
    """ Inputs:
            X: N X D
            mu: K X D
            sigma: K X 1

        Outputs:
            log Gaussian PDF (N X K)
    """
    # TODO
    # dimension conversion
    sigma = tf.squeeze(sigma)
    coef =  tf.log(2 * np.pi * sigma)
    exp = distance_func(X, mu) / (2 * sigma)
    dim = tf.to_float(tf.rank(X))
    gauss_pfd = (- dim / 2) * coef - exp
    return gauss_pfd
```

Figure 1: screenshot of log_gauss_pdf implementation.

2. Implement vectorized function that computes logP(z|x), use log_gauss_pdf and reduce_logsumexp()

   Derivation:

   Since $P(z = k|\mathbf{x}) = P(\mathbf{x}, z = k) / \sum_{j=1}^{K} P(\mathbf{x}, z = j).$

   And from Bayer's rule: P(x, z = k) = P(x | z = k) * P(z = k), P(x, z = j) = P(x | z = k) * P(z = j).

   Therefore, log(P(z = k | x))

   = log(P(x | z = k)) + log(P(z = k)) - logΣ P(x | z = k)* P(z = j)

   = log_gauss_pdf + log_pi - logΣ exp(log_gauss_pdf + log_pi)

   Implementation:

```
def log_posterior(log_PDF, log_pi):
    """ Inputs:
            log_PDF: log Gaussian PDF N X K
            log_pi: K X 1

        Outputs
            log_post: N X K
    """
    # TODO
    log_pi = tf.squeeze(log_pi)
    sum = log_PDF + tf.transpose(log_pi)
    log_sum = reduce_logsumexp(sum, reduction_indices=1, keep_dims=True)
    log_post = sum - log_sum
    return log_post
```

   Figure 2: screenshot of log_posterior implementation

   Comment on why use log-sum-exp instead of tf.reduce_sum:

   Because from the derivation we know that we want to add the log of sum to the returned value, not the sum of logs.

2.2 Learning the MoG

$$P(\mathbf{X}) = \prod_{n=1}^{N} P(\mathbf{x}_n) = \prod_{n=1}^{N}\sum_{k=1}^{K} P(z_n = k)P(\mathbf{x}_n \mid z_n = k)$$
$$= \prod_{n}\sum_{k} \pi^k \mathcal{N}(\mathbf{x}_n \,;\, \boldsymbol{\mu}^k, {\sigma^k}^2)$$

1. Implement loss function using logsumexp, logsoftmax
   Use data2D.npy, set K=3

   Best model parameters learned:

```
Final Model Parameters are:
 mu =  [[ 0.10601641 -1.5273218 ]
       [-1.0981965  -3.3092222 ]
       [ 1.2989453   0.31042182]]
 sigma =  [[0.98716843]
           [0.03908375]
           [0.03887114]]
 pi =  [[0.14273617]
        [0.13444257]
        [0.13930401]]
```

   Plot of the loss vs number of updates:

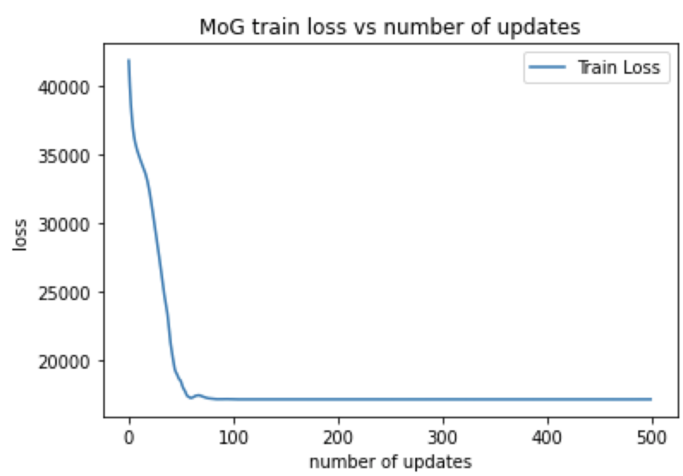
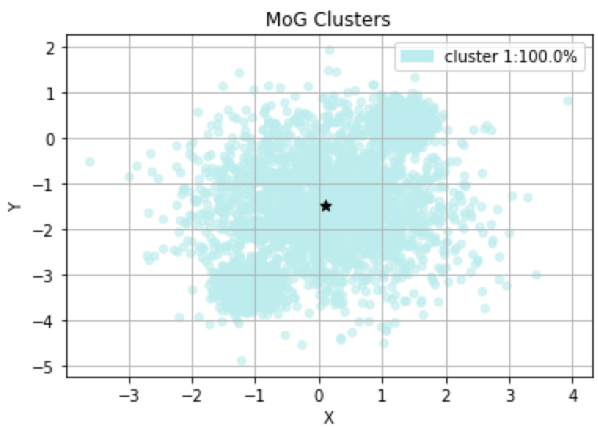
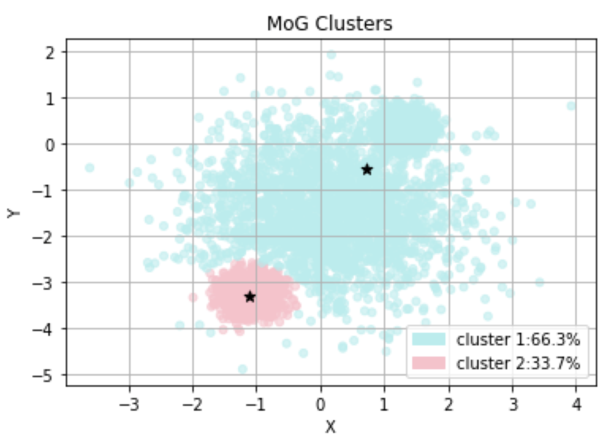
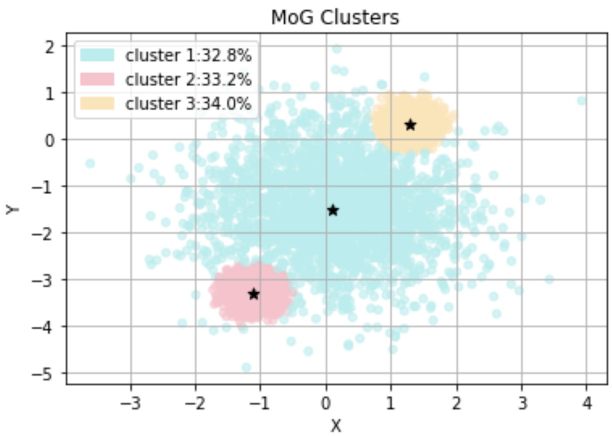


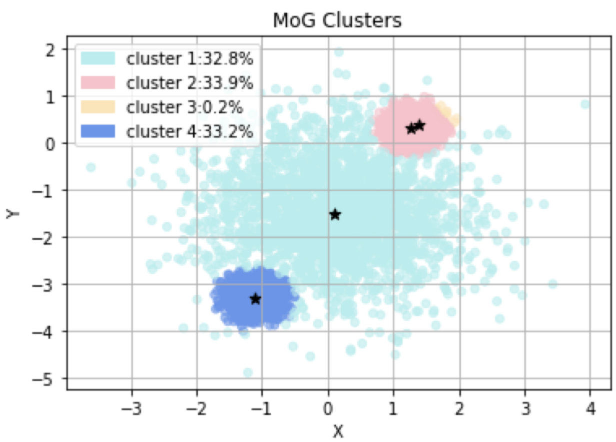
Figure 3: MoG train loss vs number of updates, when K = 3

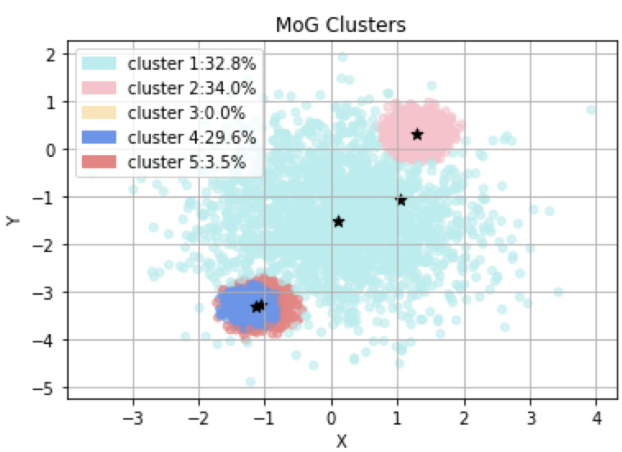
2. ⅓ validation data, for each K = 1, 2, 3, 4, 5

Table 1: Train for MoG model for K = [1, 5], validation loss vs number of updates plot and 2D scatter plot

K	MoG cluster 2D scatter plot
1	
2	

<table>
<tr><td>3</td><td>


</td></tr>
<tr><td>4</td><td>


</td></tr>
<tr><td>5</td><td>


</td></tr>
</table>

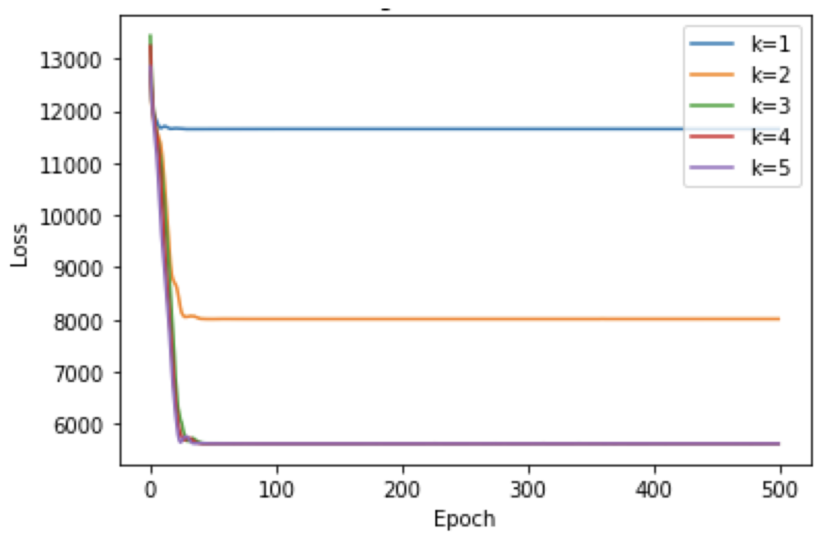


Figure 4: screenshot of validation loss of MoG when K=1,2,3,4,5

Comment: based on validation loss of different K, we can conclude that 3 is the best. Because after K reaches 3, there isn't much change in terms of validation loss. In addition, from the cluster plots, we noticed that when K is bigger than 3, there are many overlaps, which can be avoided by decreasing K value.

3. Use data100D.npy, run both K-means and MoG algorithms, for K = 5, 10, 15, 20, 30. ⅓ validation data.

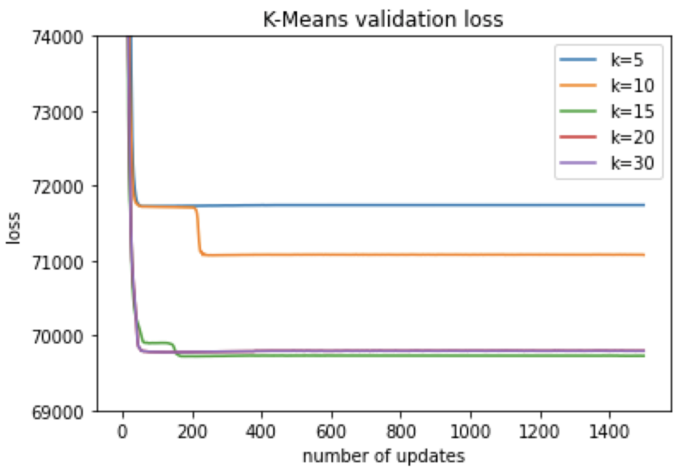


Figure 5: screenshot of K-means validation loss for K=5,10,15,20,30

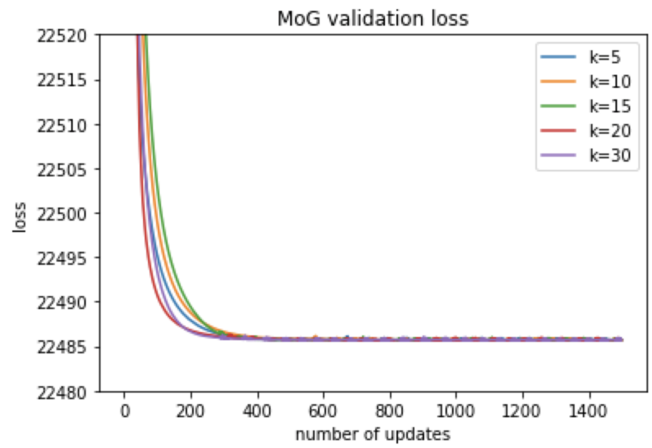


Figure 6: screenshot of MoG validation loss for K=5,10,15,20,30

	K-means final validation loss	MoG final validation loss
K=5	71741.75	**22485.6875**
K=10	71076.0234375	22485.7109375
K=15	**69727.46875**	22485.693359375
K=20	69793.5859375	22485.673828125
K=30	68923.8046875	22485.7421875

Comment: Based on the validation loss, K=15 is the best validation loss for K-means and K=5 is the best for MoG. This is because for K-means, after K=15 the validation loss changes are slight. And for MoG, the final validation losses are similar for these K values, so choosing the smallest number would be preferred. In addition, from the two validation loss figures above, these trends can also be demonstrated.