Unsupervised Learning

ECE 283 Hw3

May 14, 2020

Student: Ivan Arevalo
Email: ifa@ucsb.com

Department of Electrical and Computer Engineering, UCSB

0 Introduction

This lab focuses in introducing unsupervised learning approaches such as K-means and Expectation Maximization. K-means is an iterative algorithm that aims to group input data into K clusters based on their Euclidean distance to the cluster centers. Given that we can model data as Gaussian mixtures, we'll implement Expectation maximization to estimate each component's parameters. We will first test these algorithms with 2-dimensional data in order to gain some intuition and later extend our analysis to d-dimensional data.

1 K-Means

Given N unlabeled data points, K-means separates them into K clusters where members of a cluster are more similar to one another than members from other clusters. The Euclidean distance squared to a given cluster, measures how similar that data point is to those of a given group. In order to implement this algorithm, we will generate data from a Gaussian mixture as described in figure 1.

Component 1: $\pi_1 = \frac{1}{2}$, $\mathbf{m}_1 = (1, -1)^T$ and covariance matrix $\mathbf{C}_1$ with eigenvalue, eigenvector pairs:
$\lambda_1 = 1$, $\mathbf{u}_1 = (\cos\theta, \sin\theta)^T$, $\lambda_2 = 4$, $\mathbf{u}_2 = (-\sin\theta, \cos\theta)^T$, with $\theta = 0$.
Component 2: Gaussian mixture with two components:
Component 2: $\pi_2 = \frac{1}{6}$, $\mathbf{m}_2 = (-1, 0)^T$, $\mathbf{C}_2$ with eigenvalue, eigenvector pairs: $\lambda_1 = 4$, $\mathbf{u}_1 = (\cos\theta, \sin\theta)^T$, $\lambda_2 = 1/2$, $\mathbf{u}_2 = (-\sin\theta, \cos\theta)^T$, with $\theta = -\frac{3\pi}{4}$.
Component 3: $\pi_3 = \frac{1}{3}$, $\mathbf{m}_3 = (4, 1)^T$, $\mathbf{C}_3$ with eigenvalue, eigenvector pairs: $\lambda_1 = 1$, $\mathbf{u}_1 = (\cos\theta, \sin\theta)^T$, $\lambda_2 = 4$, $\mathbf{u}_2 = (-\sin\theta, \cos\theta)^T$, with $\theta = \frac{\pi}{4}$.

Figure 1: Gaussian mixture components

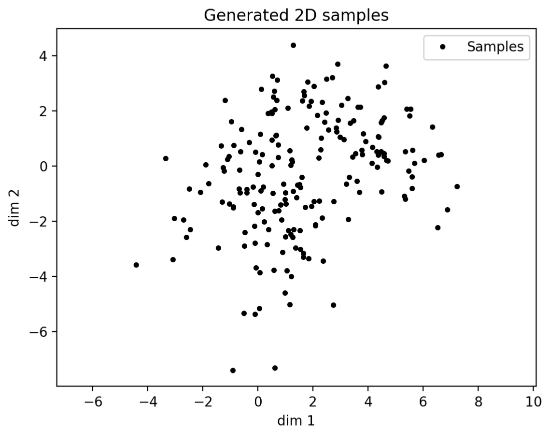


Figure 2: Unlabeled generated samples

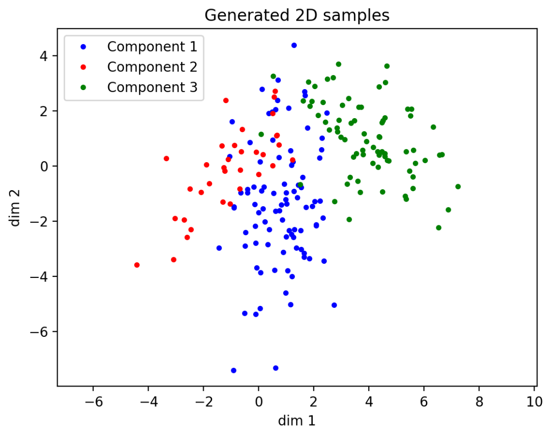


Figure 3: Labeled generated samples

In order to partition the samples into their corresponding clusters, we must minimize the squared error distortion defined as

$$\Delta^2 = \sum_{i=1}^{N} \sum_{k=1}^{K} p(k|x_i) ||x_i - m_k||^2 \tag{1}$$

where $p(k|x_i)$ is the posterior probability of sample x_i belonging to cluster k and must add up to 1. Given that the cost function in equation 1 is a not convex, optimization is difficult. Nevertheless, we can alternate between fixing assignments–minimizing cluster centers–and fixing cluster centers–minimizing over assignments–in order to converge to a local minimum. We can do so by implementing hard assignments to the closest cluster for each data sample and use these assignments to update cluster centers. Specifically,

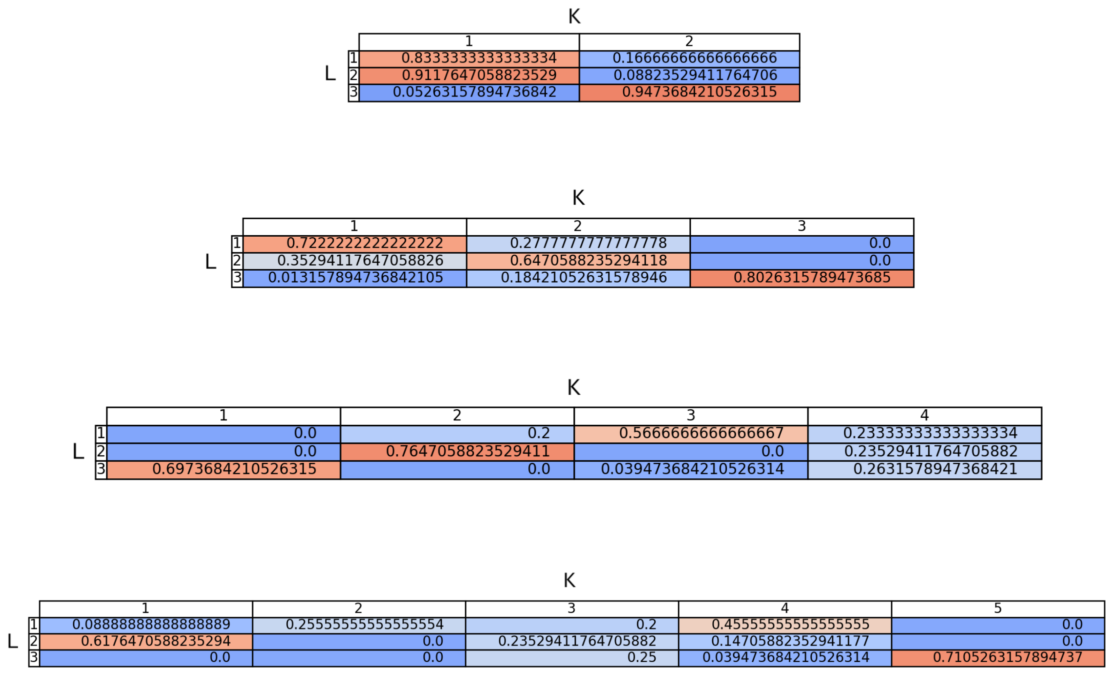
$$p(k|x_i) = \begin{cases} 1 & \text{for } k = arg\ min_{i \leq j \leq K} ||x_i - m_j|| \\ 0 & \text{otherwise} \end{cases} \tag{2}$$

$$\Delta^2 = \sum_{i=1}^{N} \sum_{i \in C_k} ||x_i - m_k||^2 \tag{3}$$

where C_k denotes the set of all samples assigned to that cluster. Minimizing over m_k,

$$\nabla_{m_k} \Delta^2 = \sum_{i \in C_k} -2||x_i - m_k|| = 0 \rightarrow m_k = \frac{1}{|C_k|} \sum_{i \in C_k} x_i \tag{4}$$

Given that the K-means algorithm consists on randomly initializing the cluster centers and alternating between each minimization, it converges to a local minimum and it is necessary run K-means several times and choose the local minimum with least distortion.



L \ K	1	2
1	0.8333333333333334	0.16666666666666666
2	0.9117647058823529	0.08823529411764706
3	0.05263157894736842	0.9473684210526315

L \ K	1	2	3
1	0.7222222222222222	0.2777777777777778	0.0
2	0.35294117647058826	0.6470588235294118	0.0
3	0.013157894736842105	0.18421052631578946	0.8026315789473685

L \ K	1	2	3	4
1	0.0	0.2	0.5666666666666667	0.23333333333333334
2	0.0	0.7647058823529411	0.0	0.23529411764705882
3	0.6973684210526315	0.0	0.039473684210526314	0.2631578947368421

L \ K	1	2	3	4	5
1	0.08888888888888889	0.25555555555555554	0.2	0.45555555555555555	0.0
2	0.6176470588235294	0.0	0.23529411764705882	0.14705882352941177	0.0
3	0.0	0.0	0.25	0.039473684210526314	0.7105263157894737

Figure 4: Empirical probabilities (low: blue, high: red) of how samples from each components (L = 1, 2, 3) map to the learned clusters (k = 1, .., K) for iterations with K = 2, 3, 4, 5 (top to bottom)

As observed in figure 4, for values of K close to the number of components in the mixture, K-means has some success in separating the majority of each component's samples into a learned cluster. Given that we are able to determine the correct number of components components from which the samples were drawn (K = 3), our results shows that 72.2% samples from component one were assigned to cluster one, 64.7% of samples from component two were assigned to cluster two, and 80.2% of samples from component three were assigned to cluster three. As expected, the number of samples successfully separated into the number of components they came from decreases as the number of K increases. In order for K-means to successfully segment data, it is necessary to have a

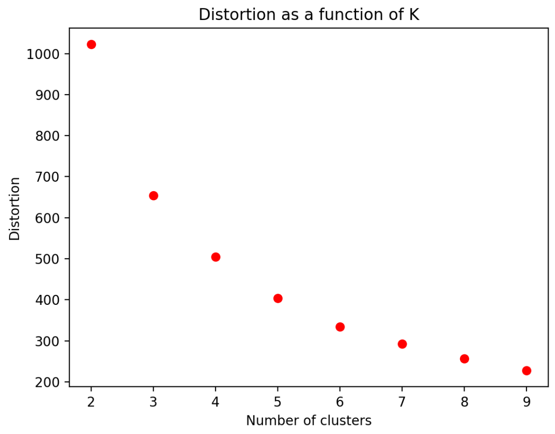


Figure 5: Distortion for each value of K

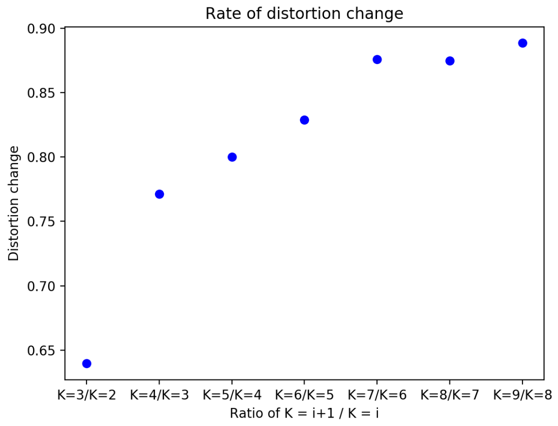


Figure 6: Rate of distortion change

good estimate of the number of components (L) from which the data originated. We can choose the number of clusters to fit the data (K) by implementing the "elbow" method in which we monitor the distortion as a function of K and pick the value for which the distortion changes most drastically and subsequently slows down. As shown in figure 5, the drop in distortion at K = 3 is significant while the drop to K =4 is not drastic. Furthermore, figure 6 shows a steep rate of distortion change at K = 3 and becomes flatter for k = 4, 5, and 6.

2 K-means++

Standard K-means initialization relies on running K-means algorithm several times in order to pick the best local minimum. On the other hand, K-means++ is a K-means initialization that only needs to be run once and it relies on biasing the random cluster center initialization in order to pick them far away from each other. This is achieved by choosing the very first cluster center at random and iteratively choosing the next cluster center with a probability proportional to the distance of each sample to their closest cluster over the summation of all the distances of all the samples to their corresponding closest cluster center. Specifically,

$$\delta_i^2 = min_{i \leq j \leq J} ||x_i - m_j||^2 \tag{5}$$

$$p_i = \frac{\delta^2}{\sum_{i'} \delta_{i'}^2} \tag{6}$$

where δ_i is the distance of the i^{th} sample to the closest of the current J cluster centers, and p_i is the probability that the i^{th} sample will be chosen to be the next cluster center. It is worth noting that the probability of a sample that is already a cluster center of being any following cluster centers is 0 given that the distance between the sample and the closest cluster center is 0. Running the K-means++ algorithm on the same generated samples as before (figure 2), we arrive at the exact same empirical probability tables as in figure 4 and distortion plots as in figures 5 and 6. This is expected since K-means was previously run several times in order to find the best local minima. On the other hand, K-means++ was able to find the same "good" local minima significantly faster than standard K-means algorithm.

3 Expectation Maximization

Gaussian mixtures are a widely use data modeling approach that requires the estimation of the mixture parameters in order to fit the data. The Maximum Likelihood (ML) estimation can be used to fit a Gaussian mixture model to the data as so,

$$\hat{\Theta}_{ML} = arg\ max_{\theta} \sum_{i=1}^{N} logp(x_i|\Theta) \tag{7}$$

where Θ are the estimated parameters. Given that we direct ML estimation is difficult in this case, we opt apply the Expectation Maximization (EM) rule in order to obtain an estimation that converges to a local minimum. The parameters characterizing the the mixture are means (m_k), covariance (C_k), and prior probabilities (π_k) where $1 \leq k \leq K$. The mixture density is given by,

$$p(x|\Theta) = \sum_{k=1}^{K} \pi_k N(x|m_k, C_k) \tag{8}$$

and the cost function,

$$J(\Theta) = \sum_{i=1}^{N} log(p(x_i|\Theta) \tag{9}$$

Once again, optimization is difficult given that we don't know which components each sample came from. However, we can first run K-means or K-means++ followed by estimating the parameters for each component in the mixture given the K-means assignments. Parameter estimation for a single Gaussian distribution is not difficult and can be performed iteratively by assigning the posterior probabilities that each sample belongs to each component k using Baye's rule. Specifically,

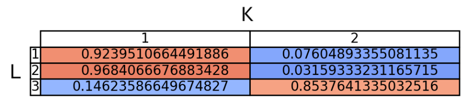
$$p(k|x_i) = \frac{\pi_k p(x_i|k)}{p(x_i)} = \frac{\pi_k N(x|m_k, C_k)}{\sum_{j=1}^{K} \pi_j N(x_i|m_j, C_j)} \tag{10}$$

Given that we use the posterior probabilities as weights to update the parameters of each distribution, it is convenient to think of these weights as the the fraction of each sample that contributes towards the update and the $\sum_{i=1}^{N} p(k|x_i) = N_k^{eff}$, the effective number of data points used to update each component k. The Expectation Maximization algorithm boils down to iteratively assigning the posterior probabilities of each sample belonging to each component k and update each component's parameters m_k, C_k, and π_k as so,

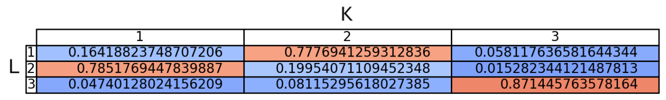
$$m_k = \frac{\sum_{i=1}^{N} p(k|x_i) x_i}{N_k^e ff} \tag{11}$$

$$C_k = \frac{\sum_{i=1}^{N} p(k|x_i)(x_i - m_k)(x_i - m_k)^T}{N_k^{eff}} \tag{12}$$

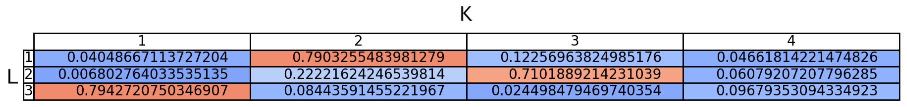
$$\pi_k = \frac{N_k^{eff}}{N} \tag{13}$$



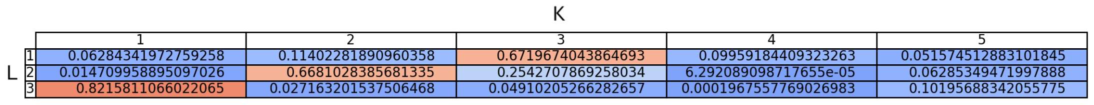
L \ K	1	2
1	0.9239510664491886	0.07604893355081135
2	0.9684066676883428	0.03159333231165715
3	0.14623586649674827	0.8537641335032516



L \ K	1	2	3
1	0.16418823748707206	0.7776941259312836	0.05811763658164434
2	0.7851769447839887	0.19954071109452348	0.015282344121487813
3	0.04740128024156209	0.08115295618027385	0.871445763578164



L \ K	1	2	3	4
1	0.04048667113727204	0.7903255483981279	0.12256963824985176	0.04661814221474826
2	0.006802764033535135	0.22221624246539814	0.7101889214231039	0.06079207207796285
3	0.7942720750346907	0.08443591455221967	0.024498479469740354	0.09679353094334923



L \ K	1	2	3	4	5
1	0.06284341972759258	0.11402281890960358	0.6719674043864693	0.09959184409323263	0.051574512883101845
2	0.014709958895097026	0.6681028385681335	0.2542707869258034	6.292089098717655e-05	0.06285349471997888
3	0.8215811066022065	0.027163201537506468	0.04910205266282657	0.0001967557769026983	0.10195688342055775

Figure 7: Empirical probabilities (low: blue, high: red) of how samples from each components (L = 1, 2, 3) map to the learned clusters (k = 1, .., K) for iterations with K = 2, 3, 4, 5 (top to bottom)

By using the K-means++ assignments from the same generated samples as in figure 2, the EM algorithm achieved the empirical probabilities displayed in Figure 7. Comparing figure 4 and 7 we can see that the EM algorithm was able to improve the probability of grouping samples from the same component into a learned cluster. Focusing on K = 3, (the number of components from which samples were generated), the probability of grouping samples generated component one, two, and three to its corresponding learned cluster increased from 72.2% to 77.7%, 64.7% to 78.5%, and 80.2% to 87.1% respectively. Furthermore, all other iterations with different values of K resulted in a higher empirical probability of mapping samples from the same generated components to the same learned clusters.

4 Higher dimensional data

Let us now experiment with higher dimensional data (d = 30) while keeping the effective dimension smaller than d by generating Gaussian mixture samples from six i.i.d quasi-orthogonal vectors $\boldsymbol{u}$ with the following probability,

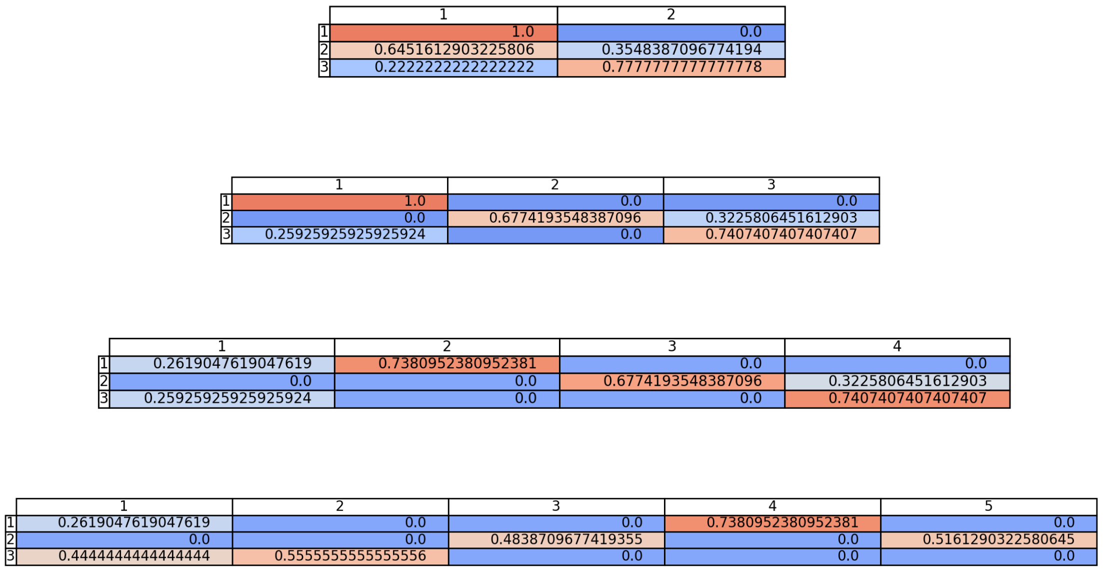
$$P[u[i] = 0] = 2/3,\ P[u[i] = +1] = 1/6,\ P[u[i] = -1] = 1/6 \tag{14}$$

N d-dimensional generated samples will be drawn equiprobably from 3 components described as follows,

Component 1: Generate $\mathbf{X} = \mathbf{u}_1 + V_1\mathbf{u}_2 + V_2\mathbf{u}_3 + \mathbf{N}$.
Component 2: Generate $\mathbf{X} = 2\mathbf{u}_4 + \sqrt{2}V_1\mathbf{u}_5 + V_2\mathbf{u}_6 + \mathbf{N}$.
Component 3: Generate $\mathbf{X} = \sqrt{2}\mathbf{u}_6 + V_1(\mathbf{u}_1 + \mathbf{u}_2) + \frac{1}{\sqrt{2}}V_2\mathbf{u}_5 + \mathbf{N}$.

Figure 8: D-dim Gaussian mixture components

where $\boldsymbol{V1}$ and $\boldsymbol{V_2}$ are coefficients drawn from a standard Gaussian distribution and $\boldsymbol{N}$ is a noise vector $\boldsymbol{N} \sim N(0, \sigma^2\boldsymbol{I}_d)$ with $\sigma^2 = 0.01$.



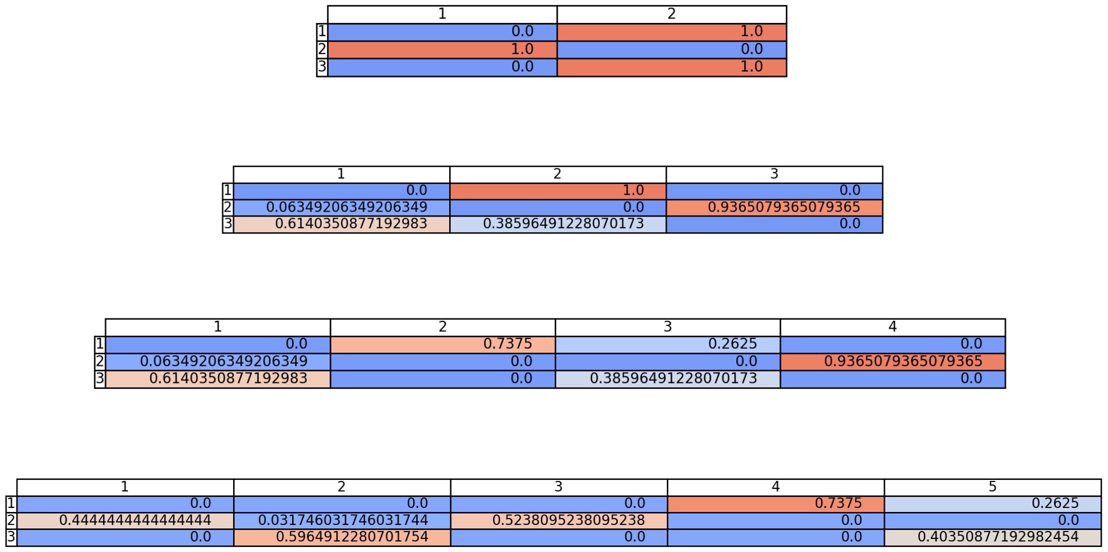
	1	2
1	1.0	0.0
2	0.6451612903225806	0.3548387096774194
3	0.2222222222222222	0.7777777777777778

	1	2	3
1	1.0	0.0	0.0
2	0.0	0.6774193548387096	0.3225806451612903
3	0.25925925925925924	0.0	0.7407407407407407

	1	2	3	4
1	0.2619047619047619	0.7380952380952381	0.0	0.0
2	0.0	0.0	0.6774193548387096	0.3225806451612903
3	0.25925925925925924	0.0	0.0	0.7407407407407407

	1	2	3	4	5
1	0.2619047619047619	0.0	0.0	0.7380952380952381	0.0
2	0.0	0.0	0.4838709677419355	0.0	0.5161290322580645
3	0.4444444444444444	0.5555555555555556	0.0	0.0	0.0

Figure 9: Empirical probabilities (low: blue, high: red) of how 100 samples from each components (L = 1,2,3) map to the learned clusters (k = 1, .., K) for iterations with K = 2,3,4,5 (top to bottom)



	1	2
1	0.0	1.0
2	1.0	0.0
3	0.0	1.0

	1	2	3
1	0.0	1.0	0.0
2	0.06349206349206349	0.0	0.9365079365079365
3	0.6140350877192983	0.38596491228070173	0.0

	1	2	3	4
1	0.0	0.7375	0.2625	0.0
2	0.06349206349206349	0.0	0.0	0.9365079365079365
3	0.6140350877192983	0.0	0.38596491228070173	0.0

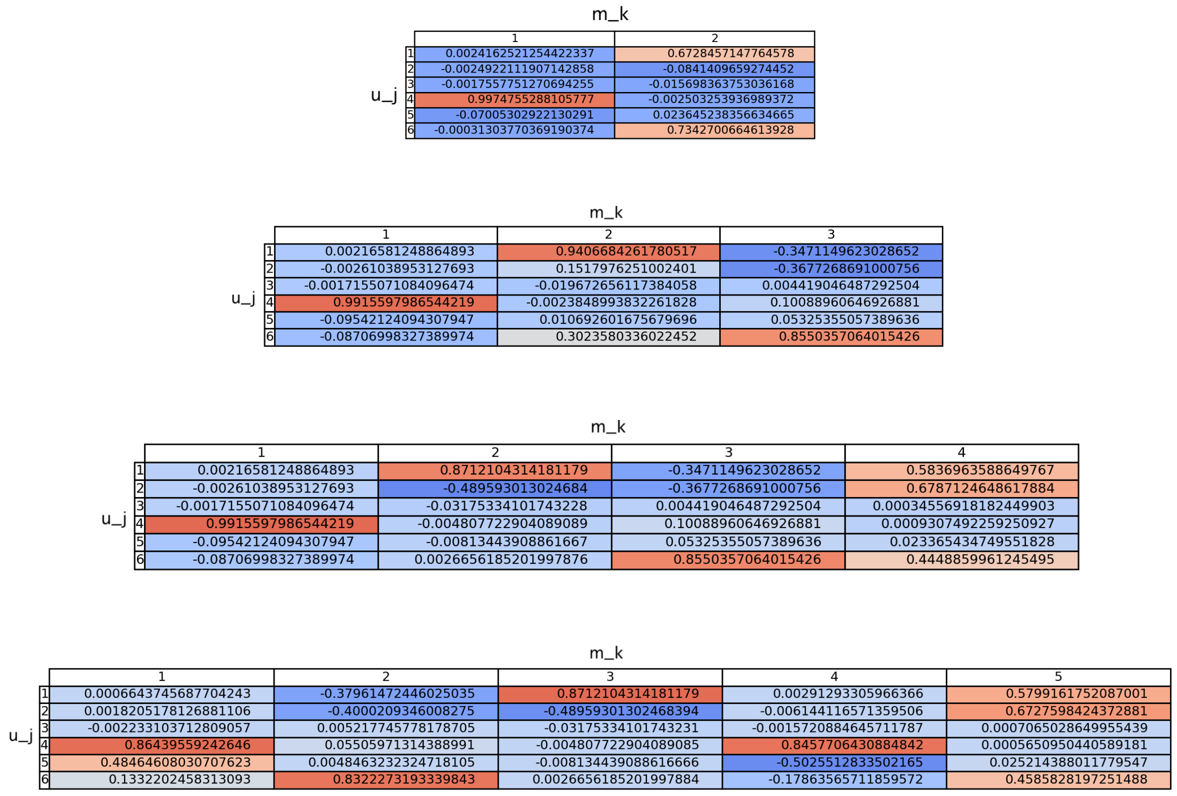
	1	2	3	4	5
1	0.0	0.0	0.0	0.7375	0.2625
2	0.4444444444444444	0.031746031746031744	0.5238095238095238	0.0	0.0
3	0.0	0.5964912280701754	0.0	0.0	0.40350877192982454

Figure 10: Empirical probabilities (low: blue, high: red) of how 200 samples from each components (L = 1,2,3) map to the learned clusters (k = 1, .., K) for iterations with K = 2,3,4,5 (top to bottom)

In order to determine a value for N (the number of samples), we run K-means algorithm with different values of N and plot their empirical probabilities of successfully clustering samples generated from the same components. Figure 9 and 10 show these probabilities as a result of running K-means from 100 and 200 generated samples respectively. Comparing the tables for K = 3 (second table down), we can see that the probability of clustering samples from component 2 increased from 67.7% to 93.6% while from component 3 decreased slightly from 74% to 61.4%. I found that running K-means on 200 generated samples yielded approximately the same clustering probabilities than on higher number of samples, while 100 samples or less decreased the performance. We therefore focus on figure 10 and move forward in our analysis with these number of samples.

5 Geometric insight

Given that the data samples were generated from six quasi-orthogonal vectors, we now attempt to get some geometric insight between these and the cluster centers found by K-means. In order to so, we compute the normalized cross-correlation between vectors and compare how aligned they are.



m_k

u_j	1	2
1	0.0024162521254422337	0.6728457147764578
2	-0.002492211907142858	-0.0841409659274452
3	-0.001755775127069425	-0.015698363753036168
4	0.9974755288105777	-0.002503253936989372
5	-0.07005302922130291	0.023645238356634665
6	-0.00031303770369190374	0.7342700664613928

m_k

u_j	1	2	3
1	0.00216581248864893	0.9406684261780517	-0.3471149623028652
2	-0.00261038953127693	0.1517976251002401	-0.3677268691000756
3	-0.0017155071084096474	-0.019672656117384058	0.004419046487292504
4	0.9915597986544219	-0.0023848993832261828	0.10088960646926881
5	-0.09542124094307947	0.010692601675679696	0.05325355057389636
6	-0.08706998327389974	0.3023580336022452	0.8550357064015426

m_k

u_j	1	2	3	4
1	0.00216581248864893	0.8712104314181179	-0.3471149623028652	0.5836963588649767
2	-0.00261038953127693	-0.489593013024684	-0.3677268691000756	0.6787124648617884
3	-0.0017155071084096474	-0.03175334101743228	0.004419046487292504	0.00034556918182449903
4	0.9915597986544219	-0.004807722904089089	0.10088960646926881	0.0009307492259250927
5	-0.09542124094307947	-0.00813443908861667	0.05325355057389636	0.023365434749551828
6	-0.08706998327389974	0.0026656185201997876	0.8550357064015426	0.4448859961245495

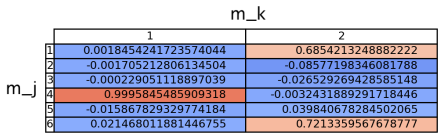
m_k

u_j	1	2	3	4	5
1	0.0006643745687704243	-0.37961472446025035	0.8712104314181179	0.00291293305966366	0.5799161752087001
2	0.0018205178126881106	-0.4000209346008275	-0.48959301302468394	-0.006144116571359506	0.6727598424372881
3	-0.002233103712809057	0.005217745778178705	-0.03175334101743231	-0.0015720884645711787	0.0007065028649955439
4	0.86439559242646	0.05505971314388991	-0.004807722904089085	0.8457706430884842	0.0005650950440589181
5	0.48464608030707623	0.004846323232324718105	-0.008134439088616666	-0.5025512833502165	0.025214388011779547
6	0.1332202458313093	0.8322273193339843	0.0026656185201997884	-0.17863565711859572	0.4585828197251488

Figure 11: Cross correlation (low: blue, high: red) between the sample basis ($\boldsymbol{u_j}$ = 1,...,6) and the K-means-learned cluster means m_k (k = 1, .., K) for each iteration of K = 2, 3, 4, 5 (top to bottom)

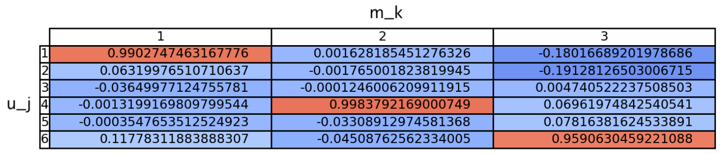
Figure 11 shows that most learned cluster means are highly correlated with exactly one of the basis $\boldsymbol{u_j}$ from which the samples were generated. It is less so the case when running K-means with K (number of clusters to be learned) higher than L (number of components from which data originated). For K = 3 (second top down) in figure 11, each cluster mean is highly aligned with one of the six basis vectors, giving us some insight into how K-means clusters data. For high dimensional data, it finds a vector that best represents or aligns with the samples in that "direction" and groups them together. We have empirically shown that for high dimensional data, the learned means are in the "directions" of the quasi-orthogonal basis which has the highest correlation with the samples from that cluster.

We can expand our analysis to the EM algorithm and note that although the components estimated from the EM algorithm won't strongly correlate with those from which we generated the data, the means and eigenvectors of the covariance matrices learned by the algorithm are strongly correlated to the the $\boldsymbol{u_j}$ basis vectors.



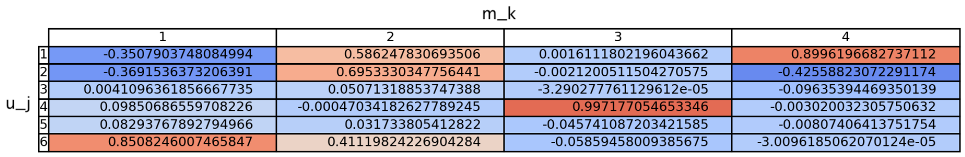
m_k

m_j	1	2
1	0.0018454241723574044	0.6854213248882222
2	-0.0017052128061345504	-0.08577198346081788
3	-0.000229051118897039	-0.026529269428585148
4	0.9995845485909318	-0.0032431889291718446
5	-0.015867829329774184	0.039840678284502065
6	0.021468011881446755	0.7213359567678777



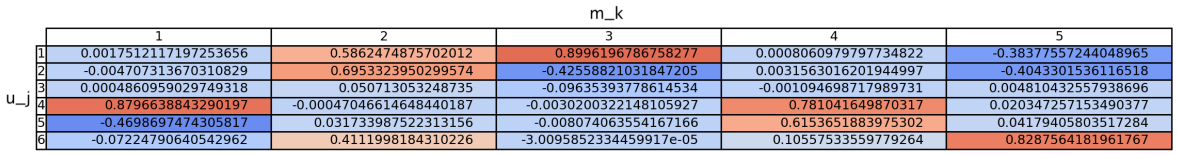
m_k

u_j	1	2	3
1	0.9902747463167776	0.001628185451276326	-0.18016689201978686
2	0.06319976510710637	-0.001765001823819945	-0.19128126503006715
3	-0.03649977124755781	-0.0001246006209911915	0.004740522237508503
4	-0.0013199169809799544	0.9983792169000749	0.06961974842540541
5	-0.0003547653512524923	-0.03308912974581368	0.07816381624533891
6	0.11778311883888307	-0.04508762562334005	0.9590630459221088



m_k

u_j	1	2	3	4
1	-0.3507903748084994	0.586247830693506	0.0016111802196043662	0.8996196682737112
2	-0.3691536373206391	0.6953330347756441	-0.0021200511504270575	-0.42558823072291174
3	0.0041096361856667735	0.05071318853747388	-3.290277761129612e-05	-0.09635394469350139
4	0.09850686559708226	-0.00047034182627789245	0.997177054653346	-0.003020032305750632
5	0.08293767892794966	0.031733805412822	-0.045741087203421585	-0.00807406413751754
6	0.8508246007465847	0.41119824226904284	-0.05859458009385675	-3.0096185062070124e-05



m_k

u_j	1	2	3	4	5
1	0.0017512117197253656	0.5862474875702012	0.8996196786758277	0.0008060979797734822	-0.38377557244048965
2	-0.004707313670310829	0.6953323950299574	-0.42558821031847205	0.0031563016201944997	-0.4043301536116518
3	0.0004860959029749318	0.050713053248735	-0.09635393778614534	-0.001094698717989731	0.004810432557938696
4	0.8796638843290197	-0.00047046614648440187	-0.0030200322148105927	0.781041649870317	0.020347257153490377
5	-0.4698697474305817	0.031733987522313156	-0.008074063554167166	0.6153651883975302	0.04179405803517284
6	-0.07224790640542962	0.4111998184310226	-3.0095852334459917e-05	0.10557533559779264	0.8287564181961767

Figure 12: Cross correlation (low: blue, high: red) between the sample basis ($\boldsymbol{u_j}$ = 1,...,6) and the EM-learned cluster means m_k (k = 1, .., K) for each iteration of K = 2, 3, 4, 5 (top to bottom)

As shown on on figure 12, most learned cluster means are highly correlated with exactly one of the basis $\boldsymbol{u_j}$ from which the samples were generated with an even greater correlation than K-means. For brevity, we skip plotting further graphs showing the correlation of covariance eigenvectors with basis vectors but the idea has been established that they strongly align.

6 Conclusion

We have explored unsupervised learning approaches such as K-means, K-means++ and Expectation Maximization algorithms. They were implemented by iteratively assigning conditional probabilities (hard or soft) and updating parameters (mean, covariance, and priors). We showed that K-means++ is an efficient algorithm to initialize the cluster centers that relies on biasing our next cluster center choice depending on how close each sample is to the closest cluster. It was shown that both K-means and K-means++ algorithm arrived at the same local minima with the latter running at a fraction of the time. Expectation Maximization (EM) was then introduced to estimate the parameters of a Gaussian mixture from which the samples were generated given the K-means cluster center estimation. We empirically showed that the EM algorithm was able to improve the probability achieved by K-means of grouping samples from the same component into a learned cluster. Finally, we provided geometric insight into how the cluster centers, component means, and covariance found by K-means and EM relate to the basis vectors from which the data was generated. It's worth noting that performance for each algorithm varies depending on the random data generated but should stay within the vicinity of the presented performance.

References

[1] Upamanyu Madhow, "K Means, Gaussian Mixtures and the EM Algorithm", *Notes from Machine Learning: A Signal Processing Perspective*, 2018-2020.

7 Code

```
import sample_generation
import numpy as np
import matplotlib.pyplot as plt
from scipy.stats import multivariate_normal
import pprint
import math


class K_means():

    def __init__(self, num_clusters, samples):
        self.num_clusters = num_clusters
        self.samples = samples
        self.one_hot_assignments = []
        self.cluster_means = []
        self.num_iterations = 0
        self.distortion = None
        self.jth_center_probabilities =
            np.ones(self.samples.shape[0])/self.samples.shape[0]

    def update_cluster_centers(self, initial_assignment=False, k_meansplusplus=False):

        if initial_assignment: # No seed provided, each run has a random cluster
            initialization
            np.random.seed()
            while len(self.cluster_means) < self.num_clusters:
                index = np.random.choice(np.arange(0, np.shape(self.samples)[0]),
                    p=self.jth_center_probabilities)
                # index = np.random.randint(0, np.shape(self.samples)[0]) # Generate
                    random index for cluster center
                if not any((self.samples[index] == cluster_mean).all() for cluster_mean in
                    self.cluster_means):
                    self.cluster_means.append(self.samples[index])
                if k_meansplusplus and len(self.cluster_means) < self.num_clusters:
                    self.assign_data_points(update_jth_center_probability=True)
        else:
            for i in range(self.num_clusters):
                self.cluster_means[i] =
                    np.mean(self.samples[np.array(self.one_hot_assignments[:, i],
                    dtype=bool)], axis=0)

    def assign_data_points(self, update_jth_center_probability=False):

        distances = np.array([]) # Euclidean distance list of each sample to each cluster
            center

        for cluster_mean in self.cluster_means:
            distance_to_cluster = np.square(np.linalg.norm(self.samples - cluster_mean,
                axis=1))
            # distance_to_cluster = np.square(np.linalg.norm(self.samples - cluster_mean,
                axis=1))
            if np.any(distances): # If not empty
                distances = np.c_[distances, distance_to_cluster] # Append norm of kth
                    cluster to array
            else:
                distances = distance_to_cluster # If empty, create matrix with norm of
                    first cluster

        self.one_hot_assignments =
            np.eye(len(self.cluster_means))[np.argmin(distances.reshape(-1,
            len(self.cluster_means)), axis=1).reshape(-1)] # One hot assignment
        self.distortion =
            sum([np.sum(np.square(np.linalg.norm(self.samples[np.array(self.one_hot_assignments[:,i],
            dtype=bool)]- self.cluster_means[i], axis=1))) for i in
            range(len(self.cluster_means))])
        if update_jth_center_probability:
            self.jth_center_probabilities =
                np.divide(distances[np.array(self.one_hot_assignments,
                dtype=bool).squeeze()], self.distortion)
        else:
            self.num_iterations += 1

    def print_info(self):
        print("K = {} clusters\nTotal number of iterations: {}\nDistortion: {}\n\n".
             format(self.num_clusters, self.num_iterations, self.distortion))


class Expectation_Maximization():

    def __init__(self, kmean_estimation):
        self.num_clusters = kmean_estimation.num_clusters
        self.samples = kmean_estimation.samples
        self.mixture_means = kmean_estimation.cluster_means
        self.num_iterations = 0
        self.posterior_probabilities = kmean_estimation.one_hot_assignments
        self.kth_effective_points = np.sum(self.posterior_probabilities, axis=0)
        self.priors = self.kth_effective_points / len(self.samples)
        self.mixture_covariences = list(np.zeros([self.num_clusters,
            self.samples.shape[1], self.samples.shape[1]]))
        self.update_mixture_covariences()
        self.update_cost()

    def update_conditional_probabilities(self):
        self.cond_probabilites = np.zeros(self.posterior_probabilities.shape)
        self.log_cond_probabilites = np.zeros(self.posterior_probabilities.shape)
        for k in range(self.num_clusters):
            gaussian_distribution = multivariate_normal(self.mixture_means[k],
                self.mixture_covariences[k])
            self.cond_probabilites[:, k] = gaussian_distribution.pdf(self.samples)
            self.log_cond_probabilites[:, k] = gaussian_distribution.logpdf(self.samples)

    def update_posterior_probabilities(self):
        # self.update_conditional_probabilities()
        self.posterior_probabilities = (1/(np.matmul(self.cond_probabilites,
            self.priors)).reshape([-1, 1]))*(self.cond_probabilites*self.priors)
        self.kth_effective_points = np.sum(self.posterior_probabilities, axis=0)


    def update_priors(self):
        self.priors = self.kth_effective_points / len(self.samples)

    def update_mixture_means(self):
        for k in range(self.num_clusters):
            self.mixture_means[k] =
                np.sum(self.posterior_probabilities[:,k].reshape([-1,1])*self.samples,
                axis=0) / self.kth_effective_points[k]

    def update_mixture_covariences(self):
        for k in range(self.num_clusters):
            self.mixture_covariences[k] = np.sum(
                [self.posterior_probabilities[i,
                    k]*np.outer(self.samples[i]-self.mixture_means[k],
                    self.samples[i]-self.mixture_means[k])
                 for i in range(len(self.samples))], axis=0) / self.kth_effective_points[k]

    def update_cost(self):
        self.cost = 0
        self.update_conditional_probabilities()
        for k in range(self.num_clusters):
             for i in range(len(self.samples)):
                 self.cost += self.posterior_probabilities[i, k] *
                     (self.log_cond_probabilites[i, k] + math.log(self.priors[k]))
        self.num_iterations += 1


def find_quasi_orthogonal_basis(dim, num_basis, from_memory=False):
    if from_memory:
        return sample_generation.generate_d_dim_samples(dim, num_basis, best_seeds=[3770,
            350, 86, 861, 5616, 2738])
    else:
        cost_list = []
        best_seeds = [3770]
        for j in range(2, num_basis + 1):
            for i in range(10000):
                trial_seed = list(best_seeds)
                trial_seed.append(i)
                q_orthoganal_vectors = sample_generation.generate_d_dim_samples(dim, j,
                    best_seeds=trial_seed)
                normalized_vectors = q_orthoganal_vectors /
                    np.linalg.norm(q_orthoganal_vectors, axis=1).reshape([-1, 1])
                check_orthogonality = normalized_vectors.dot(normalized_vectors.T)
                cost = np.sum(abs(check_orthogonality -
                    np.diag(check_orthogonality.diagonal())))
                if np.count_nonzero(q_orthoganal_vectors[j-1]) in range(8, 13):
                    cost_list.append(cost)
                else:
                    cost_list.append(100)
            print("Best cost: {}\nBest seed: {}\n".format(min(cost_list),
                cost_list.index(min(cost_list))))
            best_seeds.append(cost_list.index(min(cost_list)))
            cost_list.clear()
        return sample_generation.generate_d_dim_samples(dim, num_basis,
            best_seeds=best_seeds)


def generate_d_dim_data(basis, num_samples=50):
    #  Create component coefficients V1, V2, component choice, and Noise vectors
    np.random.seed(785) #785
    v1 = np.random.normal(size=num_samples)
    np.random.seed(3589) #3589
    v2 = np.random.normal(size=num_samples)
    random_state = np.random.RandomState(13489) # 13489
    noise = random_state.multivariate_normal(np.zeros(len(basis[0])),
        0.01*np.eye(len(basis[0])), num_samples)
    np.random.seed(89) #89
    choice = np.random.choice([1,2,3], p=[1/3, 1/3, 1/3], size=num_samples)
    samples = np.zeros([num_samples, len(basis[0])])
    one_hot_labels = np.zeros([num_samples, 3])
    for i in range(num_samples):
        if choice[i] == 1:
            samples[i, :] = basis[0] + v1[i]*basis[1] + v2[i]*basis[2] + noise[i]
            one_hot_labels[i, :] = np.array([1, 0, 0])
        elif choice[i] == 2:
            samples[i, :] = 2*basis[3] + math.sqrt(2)*v1[i]*basis[4] + v2[i]*basis[5] +
                noise[i]
            one_hot_labels[i, :] = np.array([0, 1, 0])
        else:
            samples[i, :] = math.sqrt(2)*basis[5] + v1[i]*(basis[0] + basis[1]) +
                1/math.sqrt(2)*v2[i]*basis[4] + noise[i]
            one_hot_labels[i, :] = np.array([0, 0, 1])
    return samples, one_hot_labels


def run_Kmeans_algorithm(samples, one_hot_labels, rangeK=np.arange(1, 10),
    num_trials=100, k_means_plusplus=False, plot_tables=False):
```

```
def plot_distortion():
    # Plot the distortion as a function of K.
    fig = plt.figure()
    distortions = [k_means.distortion for k_means in k_means_list]
    plt.plot(rangeK, distortions, 'ro')
    plt.title("Distortion as a function of K")
    plt.xlabel("Number of clusters")
    plt.ylabel("Distortion")

    # Plot the ratio as a function of K.
    fig2 = plt.figure()
    distortion_ratios = [k_means_list[i + 1].distortion / k_means_list[i].distortion
        for i in range(len(k_means_list) - 1)]
    x_labels = ["K={}/K={}".format(i+1, i) for i in rangeK[:len(rangeK)-1]]
    plt.plot(x_labels, distortion_ratios, 'bo')
    plt.title("Rate of distortion change")
    plt.xlabel("Ratio of K = i+1 / K = i")
    plt.ylabel("Distortion change")
    plt.show()

# K-means Algorithm
jth_k_mean = 0
epsilon = 0.000001
k_means_list = []
probability_matrix = []
for num_clusters in rangeK: # For K = 2, 3, 4, 5

    for trial in range(num_trials): # Get best estimate from num_trials

        k_means_trial = K_means(num_clusters, samples)
        k_means_trial.update_cluster_centers(initial_assignment=True,
            k_meansplusplus=k_means_plusplus) # Change for k-means++
        k_means_trial.assign_data_points()
        while k_means_trial.num_iterations < 20:
            previous_means = k_means_trial.cluster_means.copy()
            k_means_trial.update_cluster_centers()
            k_means_trial.assign_data_points()
            if np.all(abs(np.subtract(k_means_trial.cluster_means, previous_means)) <
                epsilon):
                break
        # k_means_trial.print_info()
        try:
            if k_means_trial.distortion < k_means_list[jth_k_mean].distortion: #
                Append trial with least distortion
                k_means_list[jth_k_mean] = k_means_trial
        except IndexError:
            k_means_list.append(k_means_trial)

    print("Best Iteration:")
    k_means_list[jth_k_mean].print_info()

    L = np.count_nonzero(one_hot_labels, axis=0) # Number of samples from each class L
    kth_probability_matrix = []
```

```
        for l in range(one_hot_labels.shape[1]): # For each mixture component
            ith_label_indices = np.nonzero(one_hot_labels[:, l])[0] # Get sample indices
                generate from lth component
            K = np.array(
                [np.count_nonzero(k_means_list[jth_k_mean].one_hot_assignments[ith_label_indices,
                    k], axis=0) for k in
                 range(num_clusters)]).squeeze() # Number of samples assigned to K that
                     came from L
            kth_probability_matrix.append(np.divide(K, L[l]))

        probability_matrix.append(np.array(kth_probability_matrix))
        jth_k_mean += 1

    # Plot empirical probability tables
    if plot_tables:
        plot_probability_tables(probability_matrix) # Generates plot figures
        plot_distortion() # Plot the distortion as a function of K.
    for i in range(len(probability_matrix)): print("{}K = {}\n{}\n".format((2 * i + 2) *
        " ", i + 2, probability_matrix[i]))

    return k_means_list


def run_EM_algorithm(kmeans_tests, one_hot_labels, plot_tables=False, epsilon=0.0001):
    jth_estimation = 0
    gaussian_mixture_list = []
    probability_matrix = []
    for kmeans_test in kmeans_tests:
        mixture_estimation = Expectation_Maximization(kmeans_test)
        while mixture_estimation.num_iterations < 300:
            mixture_estimation.update_posterior_probabilities()
            mixture_estimation.update_priors()
            mixture_estimation.update_mixture_means()
            mixture_estimation.update_mixture_coveriences()
            previous_cost = mixture_estimation.cost
            mixture_estimation.update_cost()
            print("Iteration : {}\nCost: {}\n".format(mixture_estimation.num_iterations,
                mixture_estimation.cost))
            if abs(mixture_estimation.cost-previous_cost) < epsilon:
                break
        gaussian_mixture_list.append(mixture_estimation)
        # pprint.pprint(mixture_estimation.__dict__)
        pprint.pprint(mixture_estimation.num_iterations)
        pprint.pprint(mixture_estimation.cost)

        L = np.count_nonzero(one_hot_labels, axis=0) # Number of samples from each class L
        kth_probability_matrix = []
        for l in range(one_hot_labels.shape[1]):
            ith_label_indices = np.nonzero(one_hot_labels[:, l])[0] # Get sample indices
                generate from lth component
            K = np.array(
                [np.sum(gaussian_mixture_list[jth_estimation].posterior_probabilities[ith_label_indices,
                    k], axis=0) for k in
                range(gaussian_mixture_list[jth_estimation].num_clusters)]).squeeze() #
                    Number of samples assigned to K that came from L

            kth_probability_matrix.append(np.divide(K, L[l]))

        probability_matrix.append(np.array(kth_probability_matrix))
        jth_estimation += 1

    # Plot empirical probability tables
    if plot_tables: plot_probability_tables(probability_matrix) # Generates plot figures
    for i in range(len(probability_matrix)): print("{}K = {}\n{}\n".format((2 * i + 2) *
        " ", i + 2, probability_matrix[i]))
    return gaussian_mixture_list


def generate_2D_test_data(num_samples=200, plot_data=False):
    seed1, seed2, seed3 = 14, 8, 1228 # Seed for each class 14, 8, 1228
    class_1, _, class_2, class_3 = sample_generation.generate_samples(num_samples,
        seed0=seed1, seed1A=seed2, seed1B=seed3)
    class_1.prior, class_2.prior, class_3.prior = 1/2, 1/6, 1/3

    np.random.seed(seed1 + seed2 + seed3)
    prior = np.random.rand(num_samples)
    samples = np.zeros([num_samples, 2])
    one_hot_labels = np.zeros([num_samples, 3])
    for i in range(num_samples):
        if prior[i] < class_1.prior:
            samples[i, :] = class_1.data[i, :] # Matrix with samples
            one_hot_labels[i, :] = np.array([1, 0, 0]) # One hot encoding matrix
        elif prior[i] < class_1.prior + class_2.prior:
            samples[i, :] = class_2.data[i, :] # Matrix with samples
            one_hot_labels[i, :] = np.array([0, 1, 0]) # One hot encoding matrix
        else:
            samples[i, :] = class_3.data[i, :] # Matrix with samples
            one_hot_labels[i, :] = np.array([0, 0, 1]) # One hot encoding matrix

    if plot_data:
        plt.figure(1)
        plt.plot(class_1.data[np.equal(class_1.data, samples)[:, 0], 0],
                 class_1.data[np.equal(class_1.data, samples)[:, 1], 1], 'b.')
        plt.plot(class_2.data[np.equal(class_2.data, samples)[:, 0], 0],
                 class_2.data[np.equal(class_2.data, samples)[:, 1], 1], 'r.')
        plt.plot(class_3.data[np.equal(class_3.data, samples)[:, 0], 0],
                 class_3.data[np.equal(class_3.data, samples)[:, 1], 1], 'g.')
        plt.axis('equal')
        plt.legend(['Component 1', 'Component 2', 'Component 3'])
        plt.title('Generated 2D samples')
        plt.xlabel('dim 1')
        plt.ylabel('dim 2')

        plt.figure(2)
        plt.plot(samples[:, 0], samples[:, 1], 'k.')
        plt.axis('equal')
        plt.legend(['Samples'])
        plt.title('Generated 2D samples')
        plt.xlabel('dim 1')
        plt.ylabel('dim 2')
        plt.show()
    return samples, one_hot_labels


def plot_probability_tables(probability_matrix):
    for i in range(len(probability_matrix)):
        # Prepare table
        columns = np.arange(1, probability_matrix[i].shape[1] + 1)
        rows = np.arange(1, probability_matrix[i].shape[0] + 1)
        cell_data = probability_matrix[i]
        normal = plt.Normalize(cell_data.min() - 0.3, cell_data.max() + 0.3) # 0.3 works
            well
        colors = plt.cm.coolwarm(normal(cell_data))

        F = plt.figure(figsize=(15, 8))
        ax = F.add_subplot(111, frameon=True, xticks=[], yticks=[])
        ax.axis('tight')
        ax.axis('off')
        plt.xlabel("K")
        plt.ylabel("l")
        the_table = ax.table(cellText=cell_data, colWidths=[0.2] * columns.size,
                             colLabels=columns, rowLabels=rows, loc='center',
                                 cellColours=colors)
    plt.show()


def get_geometric_insight(test, basis):
    normalized_basis = basis / np.linalg.norm(basis, axis=1).reshape([-1, 1])
    mean_correlation = []
    for i in range(len(test)):
        normalized_means = test[i].mixture_means / np.linalg.norm(test[i].mixture_means,
            axis=1).reshape([-1, 1])
        mean_cross_correlation = normalized_basis.dot(normalized_means.T)
        mean_correlation.append(mean_cross_correlation)
    plot_probability_tables(mean_correlation)


if __name__ == '__main__':

    # Generate 2D test samples
    samples, one_hot_labels = generate_2D_test_data(num_samples=250, plot_data=False)
    # Run Kmeans algorithm
    run_Kmeans_algorithm(samples, one_hot_labels, rangeK=np.arange(2, 10),
        plot_tables=True) # Can only plot for K > 1
    # Run Kmeans++ algorithm
    run_Kmeans_algorithm(samples, one_hot_labels, rangeK=np.arange(2, 6),
        k_means_plusplus=False, plot_tables=True)

    # Run Expectation Maximization algorithm on 2D test samples
    kmeans_tests = run_Kmeans_algorithm(samples, one_hot_labels, rangeK=np.arange(2, 6),
        k_means_plusplus=True, plot_tables=False)
```

```
run_EM_algorithm(kmeans_tests, one_hot_labels, plot_tables=True)

# Generate quasi-orthogonal basis
q_orthoganal_basis = find_quasi_orthogonal_basis(dim=30, num_basis=6,
    from_memory=True)
# Generate D-dim samples from basis
d_dim_samples, one_hot_labels = generate_d_dim_data(q_orthoganal_basis,
    num_samples=250)

# Run K-Means algorithm with K_means++ initialization
d_dim_kmeans_tests = run_Kmeans_algorithm(d_dim_samples, one_hot_labels,
    rangeK=np.arange(2, 6), k_means_plusplus=True, plot_tables=False)
get_geometric_insight(d_dim_kmeans_tests, q_orthoganal_basis)

# Run EM algorithm to estimate gaussian mixture parameters
d_dim_EM_tests = run_EM_algorithm(d_dim_kmeans_tests, one_hot_labels,
    plot_tables=False, epsilon=0.001)
get_geometric_insight(d_dim_EM_tests, q_orthoganal_basis)
```