**Name: V92 Rank:6 (digit data) Score : 0.71**

**Iris data Score:0.89**

**Approach:**

1. Loading data into numpy array

2.Preprocessing data using t-SNE and PCA

3. Implementing Kmeans:

1. Initiate random clusters

randomly pick some k centers from the data as starting value for centroids.

1. Form clusters by selecting data points which are nearest to the respective clusters

given some data and centroids for the data, allocate each datapoint to its closest centroid. This forms clusters.

1. Move centroids until there is no chance of further convergence.

form clusters around centroids, then keep moving the centroids until the moves are no longer significant.

1. No. of iteration played vital role in the formation of clusters. As the iterations increased from 30 to 170 , accuracy increased by atleast 20 and better clusters were formed eventually

4.Implementing bisecting kmeans:

* First considered whole dataset as cluster and find best cluster by evaluating respective SSE .
* Consider the cluster with maximum SSE to split
* Choose the cluster with minimum total SSE to store into the centroid list
* return the clusters until k clusters are formed

**Methodology:**

* Loading data:

Load data using numpy in the form of numpy arrays.

* Preprocessing:

In order to distinguish the zeros from the digits and to cope up with high dimensions what we can do is  reduce the dimensions while trying to retain the variation in the dataset. We try using PCA and t-SNE. Using PCA reduces the key information  and decreased the accuracy while t-SNE using n\_components as two and init as random gives maximum accuracy. Without t-SNE, algorithm was giving accuracy of 51 and it increased to almost 71. That is a drastic change. t-SNE is a technique used for dimensionality reduction which works best with the complex dataset like MNIST dataset. Unlike PCA it is a probabilistic technique. t-SNE looks at the original data that is given to the algorithm and gives us best approach to represent the data with less dimensions.

tsne = TSNE(n\_components=2, init='random', random\_state=0)

dataset = tsne.fit\_transform(data)

* Using kmeans and bisecting kmeans

1. Bisecting Kmeans used same kmeans algorithm to work on data but gave much lesser accuracy on these data. SSE was used to evaluate best cluster to split in the bisecting kmeans. Even after using PCA in bisecting lmeans, it consistently gave lower accuracy. One of the reason it did not work was because the data had lot of variations and as the data had lot of zeroes it was hard to use bisecting kmeans. Bisecting Kmeans gave me 0.37 on iris data and it could not implement on digit data due to its high dimension. Many issues were faced while implementing bisecting kmeans rather than kmeans.
2. In Kmeans , initially used without dimensionality reduction gave me 0.52 at 60 iterations. I kept the stopping criteria as when the clusters converged and the distance between the old and new centroids is 0 the algorithm would stop. As it ran for 30 iterations it gave me accuracy around 0.71 with iris but still 0.52 with digit dataset. After using t-sne score was increased to 0.71 with digit data for 170 iterations.

### Kmeans algorithm is based on Lloyd's principle which states that :

### 1.After set of centroids are found, clusters are updated to find points closest to the centroid

### C:\Users\Owner\Desktop\ml\latex.png

### 2.In the set of clusters, the centroids are recalculated until the distance between old and new centroid decreases to zero. While clusters are updated to find points closest to the updated centroids

\displaystyle \mu_k = \frac{1}{C_k}\sum_{\mathrm{x}_n \in C_k}\mathrm{x}_n\qquad(2)

**def kMeans(k, max\_dist=0):**

#use t-SNE

tsne = TSNE(n\_components=2, init='random', random\_state=0)

#initiate random centroids (randomly pick some k centers from the data as starting values for centroids)

np.random.randint(0, num\_instances - 1, size=k)

#eucliidean distance

def euclidian(a,b)

return np.linalg.norm(a-b)

#use euclidean distance to find distance between centroids and closest datapoints to form clusters (given some data and centroids for the data, allocate each datapoint to its closest centroid. This forms clusters.)

  clusters[index\_instance, 0] = np.argmin(dist\_vec)

#compute the mean (i.e. centroid at the middle) of a list of vectors (a cluster): take the sum and then divide by the size of the cluster.

centroid = np.mean(dataset[instances\_close], axis=0)

#iterations were carried out until distance between the old and new centroids is greater than zero

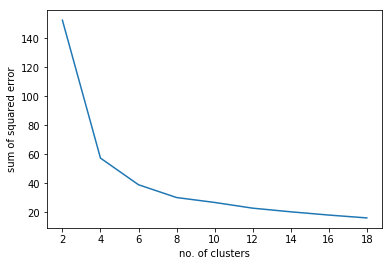
while norm > max\_dist:

iteration += 1

* Elbow method:

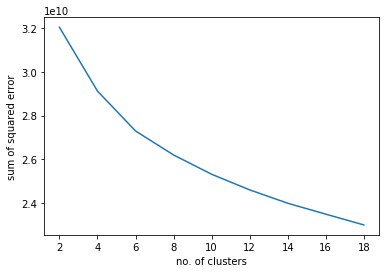
1. Iris data:

For iris data evaluating the sum of squared error versus number of clusters (k) shows me that best cluster formation is around 3 to 4 with lesser error



1. Digit Data:

For digit data , I see that ideal number of clusters is around the elbow i.e 6 to 10 in order to get lesser error.



**Further Improvements:**

1. In order to further improve the efficiency of the algorithm I tried using Naive sharding which if implemented properly could increase the accuracy of the algorithm considerably.

2.Scaling also can increase the accuracy in the cases like MNIST where the dimension of the data has lot of variance in the pixels.

3. Due to high dimension data , bisecting algorithm was not properly implemented on digits data which ran almost throughout night and still gave no results.

4. If visual graphic displays are used it can explain the better iteration number and cluster centroids.