K-means clustering Project Report

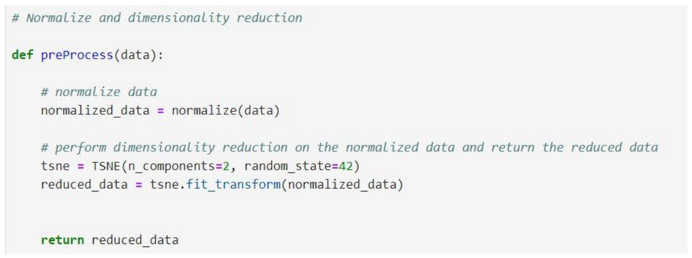
Harsith Vemuri

Iris dataset: 0.94

Image data: 0.75

Implementation

For pre-processing, I normalized the data using ‘normalize’ from “sklearn.preprocessing” module. Where the ‘normalize’ function is used to scale the data to have unit magnitude. Following this, I perform dimensionality reduction using t-SNE (t-distributed Stochastic Neighbor Embedding). t-SNE is a dimensionality reduction technique that is well-suited to visualize high-dimensional data in lower-dimensional space, such as two or three dimensions, while still being representative. Additionally, t-SNE is also preferred for nonlinear data. In my case, I reduce my data to be two dimensions, The implementation is as follows:



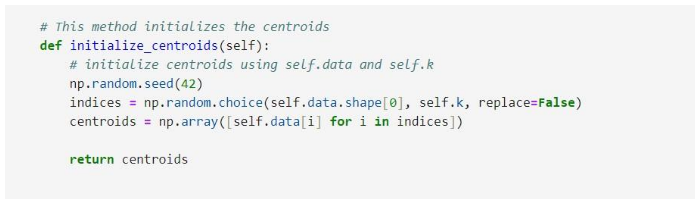
```
# Normalize and dimensionality reduction

def preProcess(data):

    # normalize data
    normalized_data = normalize(data)

    # perform dimensionality reduction on the normalized data and return the reduced data
    tsne = TSNE(n_components=2, random_state=42)
    reduced_data = tsne.fit_transform(normalized_data)


    return reduced_data
```

As for the implementation of k-means algorithm: centroids were selected randomly, and the calculation of distances between data points and respective centroids was done using the Euclidean function.

Specific details:

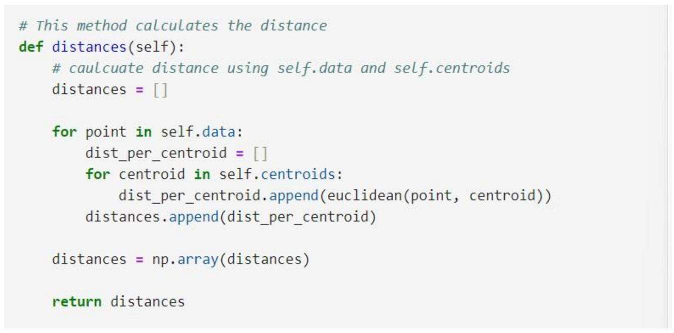
- Centroids were selected randomly **(Random seed was set to 42)**
- The max numbers of iterations for updating centroids was set to **2500** as default value. But when running for the image data the number of iterations was set to **1000**.

The implementation for initializing centroids is as follows:



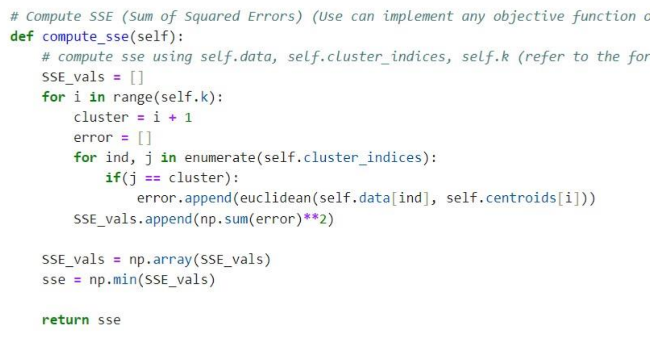
```
    # This method initializes the centroids
    def initialize_centroids(self):
        # initialize centroids using self.data and self.k
        np.random.seed(42)
        indices = np.random.choice(self.data.shape[0], self.k, replace=False)
        centroids = np.array([self.data[i] for i in indices])

        return centroids
```

To calculate the Euclidean distance between points I made use of the Euclidean function from the 'scipy.spatial.distance' module. Implementation of the calculating distances is as follows:



```python
# This method calculates the distance
def distances(self):
    # caulcuate distance using self.data and self.centroids
    distances = []

    for point in self.data:
        dist_per_centroid = []
        for centroid in self.centroids:
            dist_per_centroid.append(euclidean(point, centroid))
        distances.append(dist_per_centroid)

    distances = np.array(distances)

    return distances
```

Coming to the objective function, I implemented SSE (Sum of Squared Errors). The implementation is as follows:

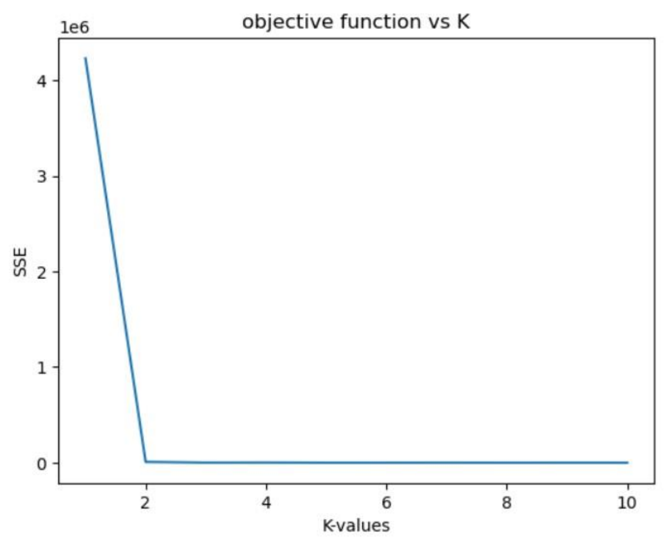


```python
# Compute SSE (Sum of Squared Errors) (Use can implement any objective function c
def compute_sse(self):
    # compute sse using self.data, self.cluster_indices, self.k (refer to the for
    SSE_vals = []
    for i in range(self.k):
        cluster = i + 1
        error = []
        for ind, j in enumerate(self.cluster_indices):
            if(j == cluster):
                error.append(euclidean(self.data[ind], self.centroids[i]))
        SSE_vals.append(np.sum(error)**2)

    SSE_vals = np.array(SSE_vals)
    sse = np.min(SSE_vals)

    return sse
```

For each cluster, taking the distance of each point to the respective centroid as error. I square each error and sum all the squared errors for all data points respective to each centroid. And consider the minimum SSE among the K number of SSEs as my final SSE.

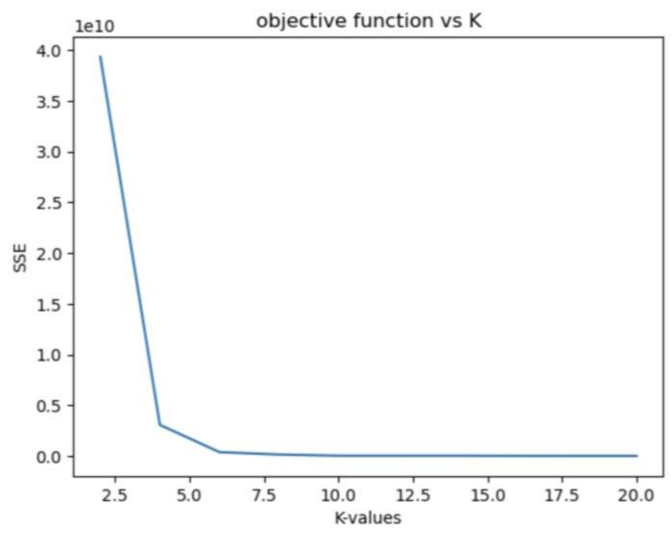
<u>Analysis of K-values w.r.t objective function:</u>

Iris Dataset:



Based on the above elbow graph, it is evident that the most desired SSE value is produced At K equals 2. Hence, the most optimal number of clusters for the iris dataset is 2.

Images Dataset:



Based on the above elbow graph, it is evident that the most desired SSE value is produced At K equals 4 (approximately). Hence, the most optimal number of clusters for the iris dataset is 4 (approximately).

Note: the max iterations for the SSE plots code for the image dataset was set to 100 because of time-complexity issues.