Intro To Machine Learning – Assignment

UB PERSON NUMBER: 50425014
UB IT NAME: PARAVAMU

TASK: The task of the project is to perform unsupervised learning on Cifar 10 dataset. There are two tasks, the first is to perform K-means clustering on the raw data from scratch. The second is to perform K- means clustering on a representation generated by the Auto-Encoder method using library functions. The code should be written in Python using Keras.

STEPS INVOLVED:

1.) IMPORTING NECESSARY LIBRARIES

- ⇨ Importing numpy for data processing
- ⇨ Importing tensorflow for downloading data purposes
- ⇨ Importing matplotlib to show images
- ⇨ Using sklearn metrics to calculate silhouette score, pairwise distance and find
- ⇨ Using scipy.spatial distance function we can find the Euclidean distance.
- ⇨ Using validclust we can find the dunn's index.

```
import tensorflow as tf
from sklearn.metrics import silhouette_score
from sklearn.metrics import pairwise_distances
from validclust import dunn
import numpy as np
import matplotlib.pyplot as plt
from scipy.spatial import distance

```

2.) IMPORTING THE DATASET

```
[ ]  (x_train, y_train), (x_test, y_test)  =tf.keras.datasets.cifar10.load_data()

```

- ⇨ The Cifar 10 dataset has a training set of 50,000 examples, and a test set of 10,000 examples.
- ⇨ Each example is a 32x32 image, associated with a label from 10 classes.
- ⇨ Each image is 32 pixels in height and 32 pixels in width, for a total of 1024 pixels in total.
- ⇨ This pixel-value is an integer between 0 and 255 (RGB). The training and test data sets have 1025 columns including the labels.
- ⇨ We use tensorflow keras cifar10 dataset.
- ⇨ This comes already split, we destructure the data to x_train, y_train x_test and y_test

```
[ ]  1  print(x_train.shape)
     2  print(y_train.shape)
     3  print(x_test.shape)
     4  print(y_test.shape)
```

```
(50000, 32, 32, 3)
(50000, 1)
(10000, 32, 32, 3)
(10000, 1)
```

- ⇨ x_train is nothing but 50000 images with the shape of 32, 32, 3
- ⇨ x_test has 10000 images
- ⇨ y_train and y_test is the labels

3.) PROCESSING THE DATASET

- ⇨ The x_train data which we got is a 4 dim np array, we reshape it into a 2d array by flattening the image to a single row of 3072 (32 x 32 x 3)
- ⇨ We also divide the dataset by 255 for easier and faster processing.
- ⇨ 3072 is one data point in the dataset, we have 50000 data points in the train set and 10000 data points in the test set.

```
[7]  1  x_train = x_train.reshape(50000,3072)
     2  x_train = x_train/255
     3  x_test = x_test.reshape(10000,3072)
     4  x_test = x_test/255
     5
```

4.) CENTROIDS AND CLUSTERS

- ⇨ We know that the ciphar10 dataset has 10 classes of images, so we are initializing the cluster size to 10.
- ⇨ We also need ten data points centroids which will help us to find the clusters.

```
[8]  1  Clustersize = 10
     2  Centroids = np.random.rand(Clustersize,3072)
     3
```

- ⇨ We randomly get 10 data point clusters using np.random.rand

```
[10]  1  Centroids.shape
      2
```

```
(10, 3072)
```

5.) DIFFERENTIATION OF DATAPOINTS

- ⇨ We'll need clustersize (10) lists to store various clusters in its corresponding list based on the distance between the centroid data point and x_test data points
- ⇨ We create a list of lists, so this can be used to store the data points

```
[ ]  1  temp = []
     2  newCentroids = []
     3  lists = [[] for _ in range(Clustersize)]
```

6.) INITIALIZING MAX ITERATIONS AND LABELS

- ⇨ There are many ways to stop our Kmeans algorithm and using max iterations is one of its ways
- ⇨ Say the datapoint 0 belongs to class 8, these are labels, and we must store the label for each datapoint
- ⇨ Since our test set has 10000 datapoints we need 10 class labels, and each data point will point to one of the 10 labels.

```
[16]  1  max_iterations = 100
      2  labels = np.zeros((x_test.shape[0]))
      3  labels.shape
      4  # labels = np.zeros((x_train.shape[0]))

    (10000,)
```

7.) KMEANS ALGORITHM

- ⇨ The intuition behind Kmeans: After we initialize the centroids, we must find the distance between the centroid and the datapoints. If it belongs to the ith label, then we store it in the ith list.
- ⇨ The distance is nothing but the Euclidean distance between two data points.
- ⇨ **Euclidean distance d = $\sqrt{[(x_2 - x_1)^2 + (y_2 - y_1)^2]}$**
- ⇨ This way for each iteration we'll have each data point in its corresponding label list.
- ⇨ After one iteration we find the mean of all the lists(indexed), this will be our new centroids
- ⇨ This will go on for max iterations, till we get valid clusters
- ⇨ After the max iterations are done, we will have 10 centroid data points, 10 lists with clustered data points, which is nothing but the images itself.

```
[48]  1  for l in range(0,max_iterations):
      2    lists = [[] for _ in range(Clustersize)]
      3    for x in range(0, x_test.shape[0]-1): # replace with x_train if you want to find the clusters of the x_train dataset
      4      for y in range(0, Centroids.shape[0]): #For every datapoint in test, we find the distance from all the centroids
      5        dist = distance.euclidean(x_test[x], Centroids[y]) # finds the euclidean distance the x_test datapoint and the centroids
      6        temp.append(dist)
      7      min_value = min(temp) #finds the minimum value of the datapoints present in temp
      8      min_index = temp.index(min_value) # finds the index  where the minimum value lies in
      9      temp = [] # Emptying temp array for next iteration
     10      lists[min_index].append(x_test[x]) # min_index in this case will be the label
     11      #We append the datapoint to list of label
     12      labels[x] = min_index
     13    for index, cluster in enumerate(lists):
     14      t = np.asarray(cluster) #we find the mean of each cluster list
     15      Centroids[index] = np.mean(t, axis=0) #The mean of each list will be the new centroids
     16    if(l%10==0):
     17      print("Iteration ",l," done") # Denoting that every 10th iteration is done
     18
     19
```

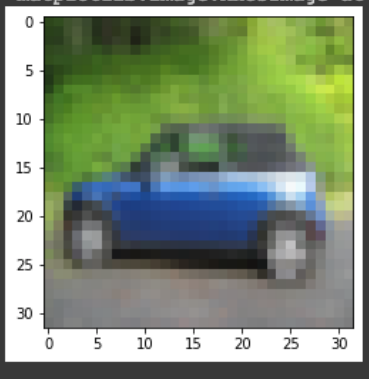
```
Iteration  0  done
Iteration  10  done
Iteration  20  done
Iteration  30  done
Iteration  40  done
Iteration  50  done
Iteration  60  done
Iteration  70  done
Iteration  80  done
Iteration  90  done
```

8.) RESULTS

- ⇨ We will have the Centroids which are the centers of the clusters present in the list
- ⇨ We will have various clusters stored in a list.
- ⇨ To verify this, we can reshape the datapoint to an image and we can view it with the help of matplotlib imshow

```
1  fig, ax = plt.subplots()
2  ax.imshow(lists[3][11].reshape(32,32,3))
```

```
<matplotlib.image.AxesImage at 0x7f37db2bc190>
```



9.) CLUSTER VALIDATION TECHNIQUES

- ⇨ When we want to validate our cluster, we evaluate the goodness of the clustering algorithm.
- ⇨ There is internal, external and relative cluster validation technique

- ⇨ Silhouette coefficient and Dunn's index are internal measure cluster validation techniques

10.) DUNN'S INDEX

- ⇨ Dunn's index is equal to the minimum inter cluster distance divided by the maximum cluster size.
- ⇨ Inter cluster distance must be large and the intra cluster distance must be small
- ⇨ We perform dunn's index to validate our cluster here using the sklearn metrics library
- ⇨ Also, to calculate the dunn's index we must calcalute the pairwise distance of the x_test datapoints

```
[105]  1  dist = pairwise_distances(x_test)

[106]  1  dist.shape

       (10000, 10000)
```

- ⇨ After we find the distance we pass it into our sklearn.metrics.dunn function

```
       1  dunn(dist, labels)

       0.11922183806289238
```

11.) AVERAGE SILHOUETTE COEFFICIENT

- ⇨ Cohesion of one cluster with each cluster is compared. The measure of comparison is called as the silhouette value.
- ⇨ This value ranges from -1 to +1
- ⇨ A +1 denotes the object is well matched with all the data points of its own cluster and not matched with the other clusters.
- ⇨ A -1 denotes the opposite.
- ⇨ We use sklearn.metrics library to calculate the ASC.

```
[147]  1  silhouette_score(x_test, labels)

       0.057042343854832055
```

Part 2: USING AUTO-ENCODER ON TRAINING DATA

1.) IMPORTING NECESSARY LIBRARIES

- ⇨ Importing Input, Dense and Model to make our Neural Network
- ⇨ Sklearn.cluster.kmeans does Kmeans clustering.

```
[41]  1  from tensorflow.keras import Input, Model
      2  from tensorflow.keras.layers import Dense
      3  from sklearn.cluster import KMeans
```

2.) DEFINING THE LAYERS OF THE AUTO ENCODER

- ⇨ We'll have to initialize the dimension to which the input image must be encoded to.

```
  1  # Our original dim is 3072, we encode that to 128 and the compression factor is 24
  2  encoding_dimension = 128
  3
```

- ⇨ In our case its 128. So, the image will be encoded (compressed) from 3072 to 128.
- ⇨ Compression factor (24) = Input Image dim (3072)/ Encoding dim (128)
- ⇨ We create three models, Autoencoder model, Encoder Model and the Decoder model.

2.1) Auto-Encoder Model

- We'll have an input layer, one hidden layer and one output layer.
- The input layer will have a shape of 3072.
- The hidden layer is dense and is fully connected.
- The neuron in the dense layer receives our input from the previous layer.
- In background the dense layer performs matrix multiplication.
- Also, with the help of Relu activation (+ve - > same number, -ve → replace with zero) we'll get the encoded representation of the input image. (From 3072 to 128).
- Then our final layer will also be dense which will take the encoded layer output as input.
- Also, with the help of the sigmoid activation function we get a lossy representation of the image back (From 128 to 3072).
- We pass the input layer and output layer to model.
- Model (): groups layers into an object with training and inference features.

```
3
4  # This is our input image which is of size 3072(flattened 32,32,3)
5  input_img = Input(shape=(3072,))
6
7  # With the help of relu activation we get the encoded representation of the input image
8  encoded_img = Dense(encoding_dimension, activation='relu')(input_img)
9
10 # We get the lossy reconstruction of the image through decoding the encoded image
11 decoded_img = Dense(3072, activation='sigmoid')(encoded_img)
12
13 # Passing the input layer and the output layer to the model
14 autoencoderModel = Model(input_img, decoded_img)
15
```

2.2) Encoder Model

- We'll have an input and output layer.
- The input neurons will be the same size as the input image (3072).
- The output layer will have a shape of 128 (encoded image).
- This will help us encode the training images with the help of encoder.predict (x_train).

```
[21] 1
     2  encoderModel = Model(input_img, encoded_img)
     3
```

2.3) Decoder Model

- We'll have an input and output layer.
- The input layer will have a shape of 128 (encoded image size).
- The output layer will have a shape of 3072 (actual input image size).
- The output layer is as same as the last layer of the autoencoder model.
- This will help us decode the encoded images with the help of decoder.predict(encoded_images).

```
[24] 1  # Since we'll pass an input encoded image of size encoding dim
     2  encoded_input = Input(shape=(encoding_dimension,))
     3  # Retrieve the last layer of the autoencoder model which has a size of 3072
     4  decoder_layer = autoencoderModel.layers[-1]
     5  # Create the decoder model with an input of shape 128
     6  decoderModel = Model(encoded_input, decoder_layer(encoded_input))
     7
```

3.) COMPILING AND FITTING THE MODEL.

- ⇨ We then compile the model with the binary crossentropy loss function.
- ⇨ We use the Adam optimizer. The Adam optimizer is good with sparse data, it uses an adaptive learning rate.
- ⇨ We don't have to worry about the learning rate unlike Stochastic gradient descent.
- ⇨ It's the best among the adaptive optimizers.

```
[25]  1  autoencoderModel.compile(optimizer='adam', loss='binary_crossentropy')
```

```
[26]  1  autoencoderModel.fit(x_train, x_train,
      2                 epochs=10,
      3                 batch_size=256,
      4                 shuffle=True,
      5                 validation_data=(x_test, x_test))
```

- ⇨ After compiling the model, we fit the model using our xtrain data and the validation for this can be done using the x_test data

4.) GETTING ENCODED IMAGES

- ⇨ We can get the encoded images with the help of the encoder model.
- ⇨ This model converts 3072 features to 128 and has a compression factor of 24.
- ⇨ Since our xTrain data has 50000 images with 3072 features. It changes that to 50000 images with 128 feature

```
[27]  1  encodedImages = encoderModel.predict(x_train)
```

```
[39]  1  encodedImages.shape

      (50000, 128)
```

5.) APPLYING KMEANS ON OUR ENCODED IMAGES

- ⇨ We'll create list of lists. So, we can put the images in its corresponding list based on the label.

```
[29]  1  newlists = [[] for _ in range(Clustersize)]
```

- ⇨ We'll create Kmeans model with the help of Sklearn.cluster
- ⇨ We pass the encoded Images to the kmeans model to cluster it.
- ⇨ For each encoded image we get a corresponding label.

```
      1  KMModel = KMeans(
      2      n_clusters=10, init='random',
      3      n_init=10, max_iter=300,
      4      tol=1e-04, random_state=0
      5  )
      6  dataLabels = KMModel.fit_predict(encodedImages)
```

```
[32]  1  dataLabels.shape

      (50000,)
```

6.) DECODING IMAGES AND APPENDING IMAGES TO LISTS BASED ON THE LABEL

- ⇨ For Decoding the images, we call the decoder model's predict function.

⇨ We pass the encoded images as input.

```
[33]  1  decodedImages = decoderModel.predict(encodedImages)
      2
```

⇨ Based on the label we put the corresponding decoded image in the list [label].

```
1  for ls in range(0, encodedImages.shape[0]-1):
2    newlists[dataLabels[ls]].append(decodedImages[ls])
```

⇨ We can have a look at the sparse image present in the lists with the help of matplotlib.

```
1  fig, ax = plt.subplots()
2  ax.imshow(newlists[3][1].reshape(32,32,3))
```

```
<matplotlib.image.AxesImage at 0x7f86b85338d0>
```

7.) <u>COMPARING SILHOUETTE SCORE</u>

⇨ Before the silhouette score was 0.057.
⇨ Now we pass the encoded images and its labels to silhouette function and the ASC turns out to be 0.90.
⇨ We clearly see an improvement.

```
[36]  1  silhouette_score(encodedImages,dataLabels)
```

```
0.09092586
```