INDRAPRASTHA INSTITUTE OF INFORMATION TECHNOLOGY
NEW DELHI

Department of Computer Science & Engineering

CSE 343/ECE 363 : Machine Learning

Dr. Jainendra Shukla

Assignment - 4: CNN , K-Means

Anant Kumar Kaushal (2022067)

SECTION - A

A As per the question, input image is 'M×N' with P channels (dimensions)

(a) Dimensions of resulting feature map:-

Output height $= \left[\frac{M - K + 2 \times P}{\text{stride}}\right] + 1$

Output width $= \left[\frac{N - K + 2 \times P}{\text{stride}}\right] + 1$

Convolutional layer $\Rightarrow K \times K$

As stride $= 1$ & $P = 0$:-

~~Feature map dimension = [M - K+1] [N - K+1]~~

$\therefore$ Feature map dimension = (Output height)×(Output width)

$= (M-K+1)(N-K+1)$

(b) For each output pixel:
for P inputs, Total multiplicxns $= P \times K \times K = PK^2$
~~For~~ Total Addxns $= P \times K \times K - 1 = PK^2 - 1$
Total $= 2PK^2 - 1$

(c) As per '(a)', total number of output pixels per kernel $= (M-K+1)(N-K+1)$

For 'Q' kernels $= Q \times (M-K+1)(N-K+1)$

from '(b)', ~~total nos~~ order of operations per pixel $= P \times K^2$

Order of total ops. = $(P \times K^2)\ [Q \times (M-K+1) \times (N-K+1)]$

$= QPK^2\ (M-K+1)\ (N-K+1)$

$\because$ $\min(M,N) >> K$:-

$M-K+1 \simeq M$

$N-K+1 \simeq N$

$\therefore$ $= QPK^2 MN$

Final complexity = $O(Q \times P \times K^2 \times (M \times N))$

B I Assignment step:

1. Distance calculation: For each point x_i, calculate distance b/w x_i and centroid.

$$D(x_i, \mu_k) = |x_i - \mu_k|^2$$

2. Cluster assignment: To the cluster, we assign new data point where μ_k is closest:

$$C_k = \{ x_i : |x_i - \mu_k|^2 \le |x - \mu_j|^2\ \forall j,\ 1 \le j \le k \}$$

II Update step:

For each cluster C_K, the new centroid μ_K as the mean of all data points assigned to that cluster.

$$\mu_K = \frac{1}{|C_K|} \sum_{x_i \in C_K} x_i$$

The 'Elbow method' is a commonly used technique in clustering analysis to determine the optimal of clusters (k) for a dataset when using algorithms like K-means.

Within-Cluster Sum of Squares (WCSS):

The WCSS measures the sum of squared distances between each data point and the centroid of its cluster. It quantifies how compact the clusters are.

$$WCSS = \sum_{k=1}^{K} \sum_{x \in C_k} ||x - \mu_k||^2$$

K: No's of clusters
C_K: k^{th} Cluster
μ_K: Centroid of k^{th} cluster
$||x - \mu_k||^2$: Squared Euclidean distance

After the above computation, the plot would show a decreasing graph with increasing k. (k on the x-axis and WCSS on the y-axis).

The new point can be identified as one where the rate of decrease in WCSS sharply changes.

Yes, random ~~initi~~ intialization can somtimes arrive at the global minimum, especially if the inital centroids happen to be well-placed. Randomly assigning cluster leads to convergence but it is not guaranteed to find the global minimum. Different initial centroids can lead to different clustering leads to different results hence converging to local minimum (not the global one).

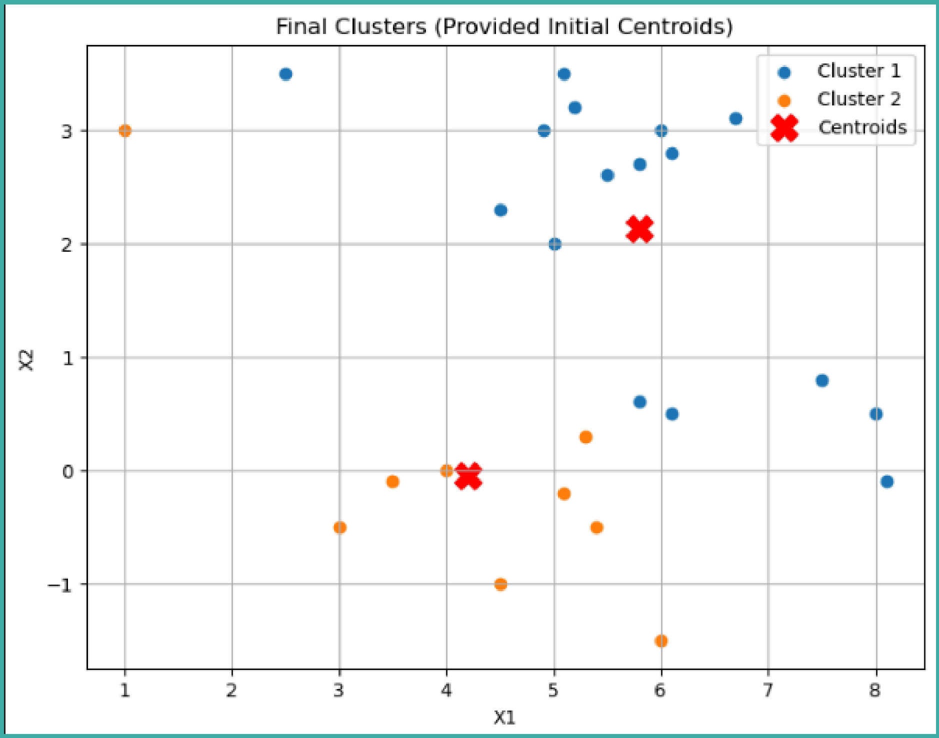
SECTION - B

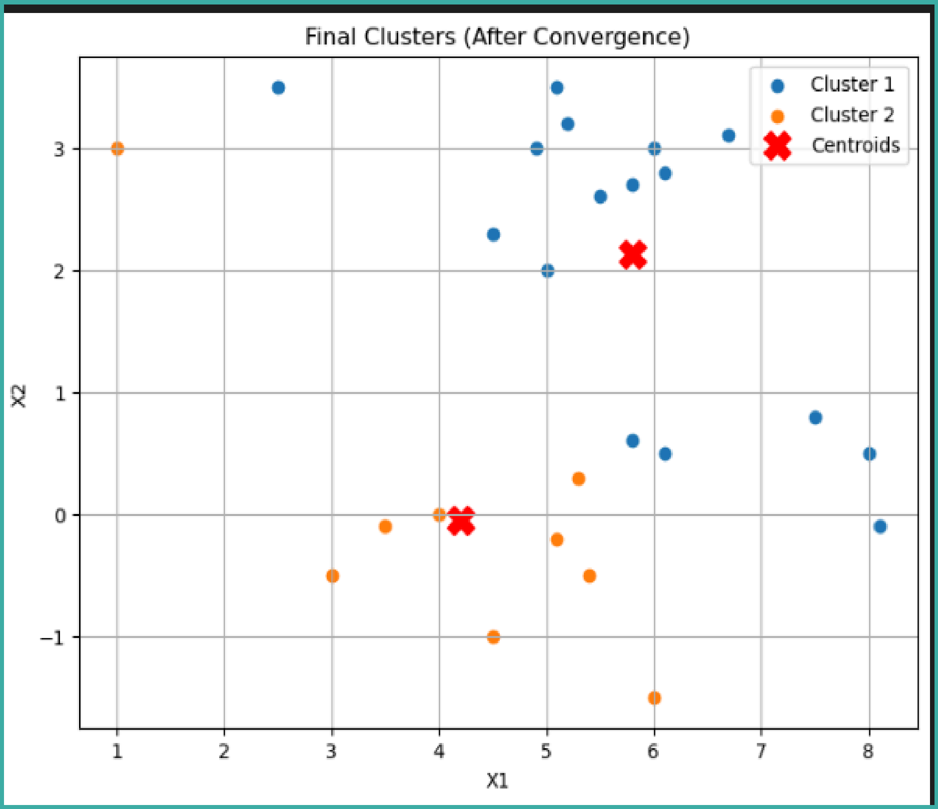
1. Implementation of K-Means Clustering from scratch including all important parameters is as follows:-

```
def kmeans_clustering(X, centroids, max_iter=100, threshold=1e-4):
    for iteration in range(max_iter):
        distances = np.linalg.norm(X[:, np.newaxis] - centroids, axis=2)
        labels = np.argmin(distances, axis=1)
        new_centroids = np.array([X[labels == k].mean(axis=0) if len(X[labels
== k]) > 0 else centroids[k] for k in range(len(centroids))])

        if np.linalg.norm(new_centroids - centroids) < threshold:
            print(f"Converged after {iteration + 1} iterations.")
            break
        centroids = new_centroids
    return centroids, labels
```

2. Final Values after algorithm convergence and the useful cluster plots are as follows:-

```
Final Centroids:
[[ 5.8         2.125     ]
 [ 4.2        -0.05555556]]
```

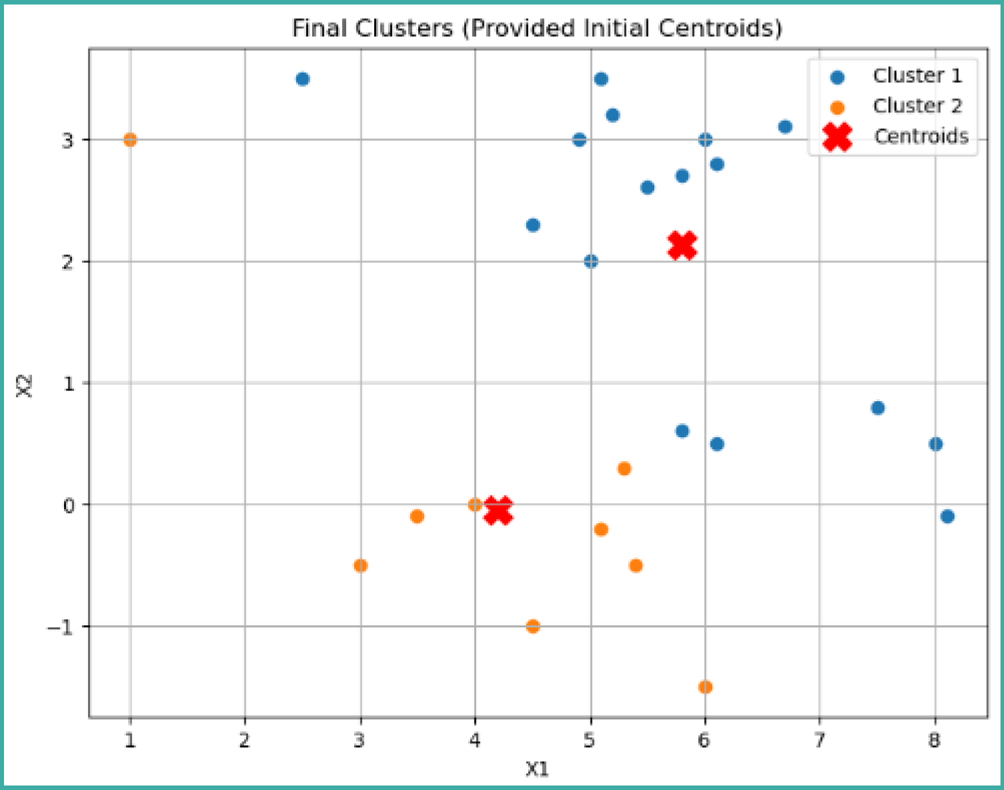

Final Clusters (Provided Initial Centroids)
Cluster 1
Cluster 2
Centroids
X1
X2

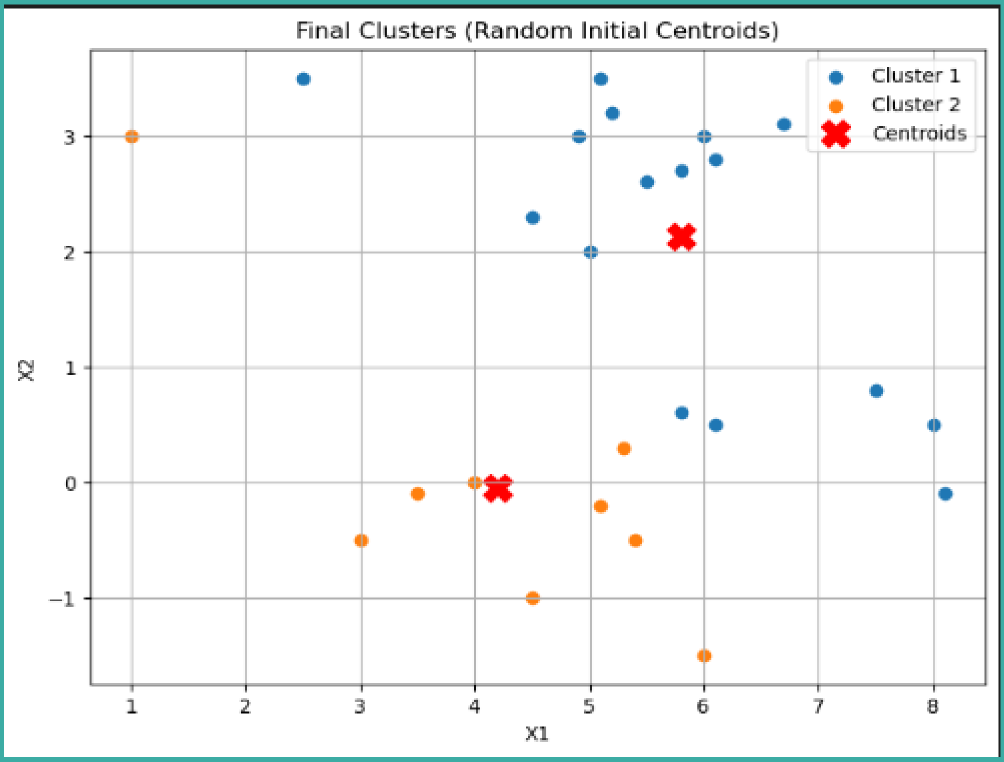

Final Clusters (After Convergence)
Cluster 1
Cluster 2
Centroids
X1
X2

3. Comparison of the results using the provided initial centroids versus using random initialization of centroids is as follows :-

Both approaches produced the same number of final clusters, and both converged at the same number of iterations. This suggests that the two initial centroid selection methods are not very different in the impact on the final clusters obtained in this particular data set. But again, it has to be noted that such might not always be the case. For other data sets, another initialization method could mean that different clustering results can be obtained, possibly necessitating more iterations to converge.

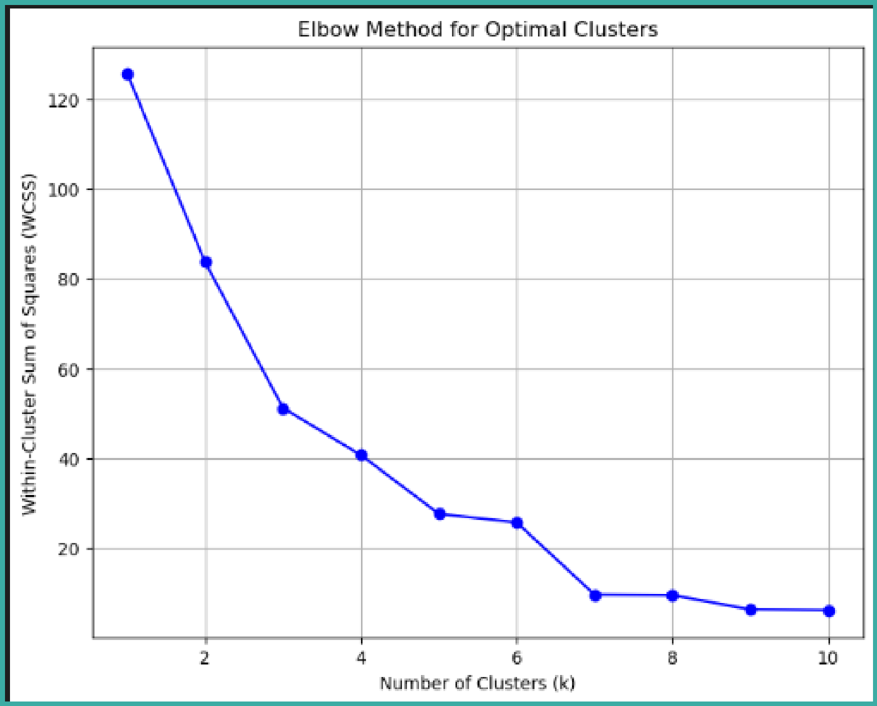
```
Final Centroids (Provided Initial Centroids):
[[ 5.8         2.125     ]
 [ 4.2        -0.05555556]]

Final Centroids (Random Initial Centroids):
[[ 5.8         2.125     ]
 [ 4.2        -0.05555556]]
```

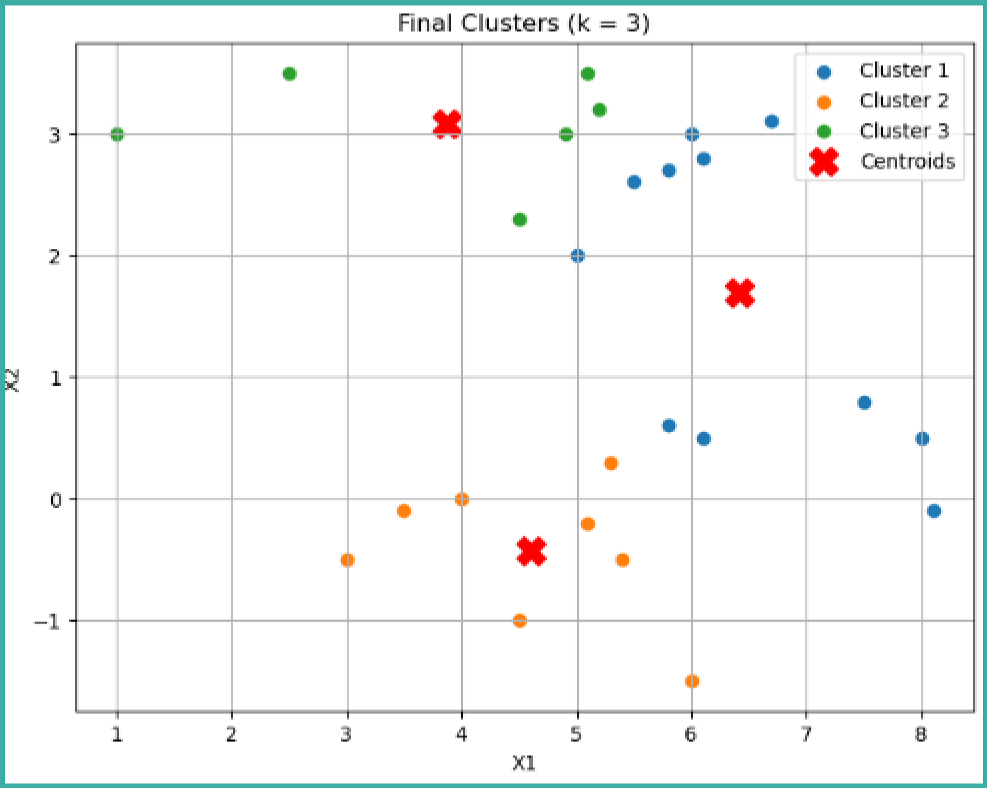

Final Clusters (Provided Initial Centroids)
Cluster 1
Cluster 2
Centroids
X1
X2


Final Clusters (Random Initial Centroids)
Cluster 1
Cluster 2
Centroids
X1
X2

4. Elbow Method , WCSS & Final Cluster Plot :-

As we know, the elbow method is a method to figure out the ideal number of clusters in a data set when doing k-means clustering. It involves plotting the within-cluster sum of squares, or WCSS, versus the number of clusters. The elbow point is the point where the curve bends, giving the best number of clusters to minimize variance without overfitting. So if we see the WCSS curve for this question we see the elbow is made at k = 3 therefore it is the optimal value of k . After this optimal value of k, there is no significant decrease in the y-axis values compared to before.





Final Clusters (k = 3)
Cluster 1
Cluster 2
Cluster 3
Centroids
X1
X2

SECTION - C

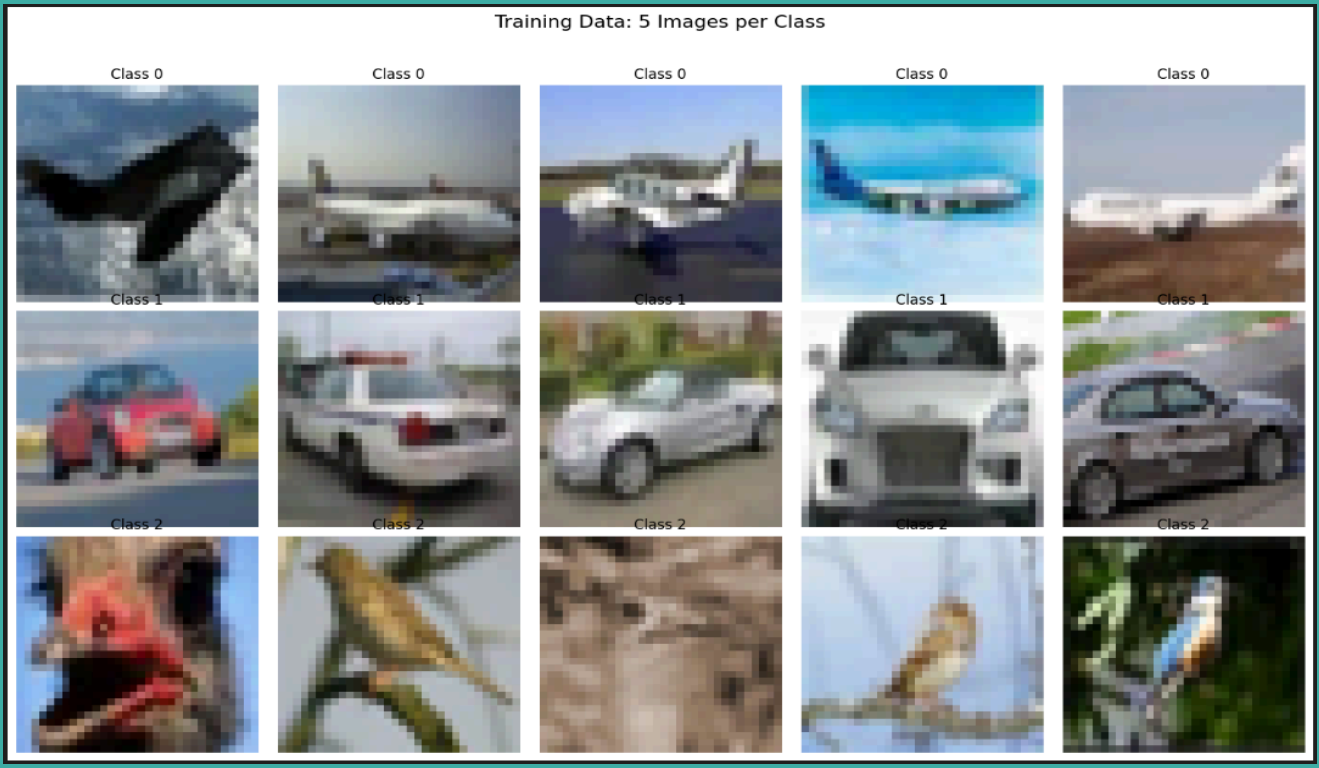
1. <u>Data Preparation:-</u>

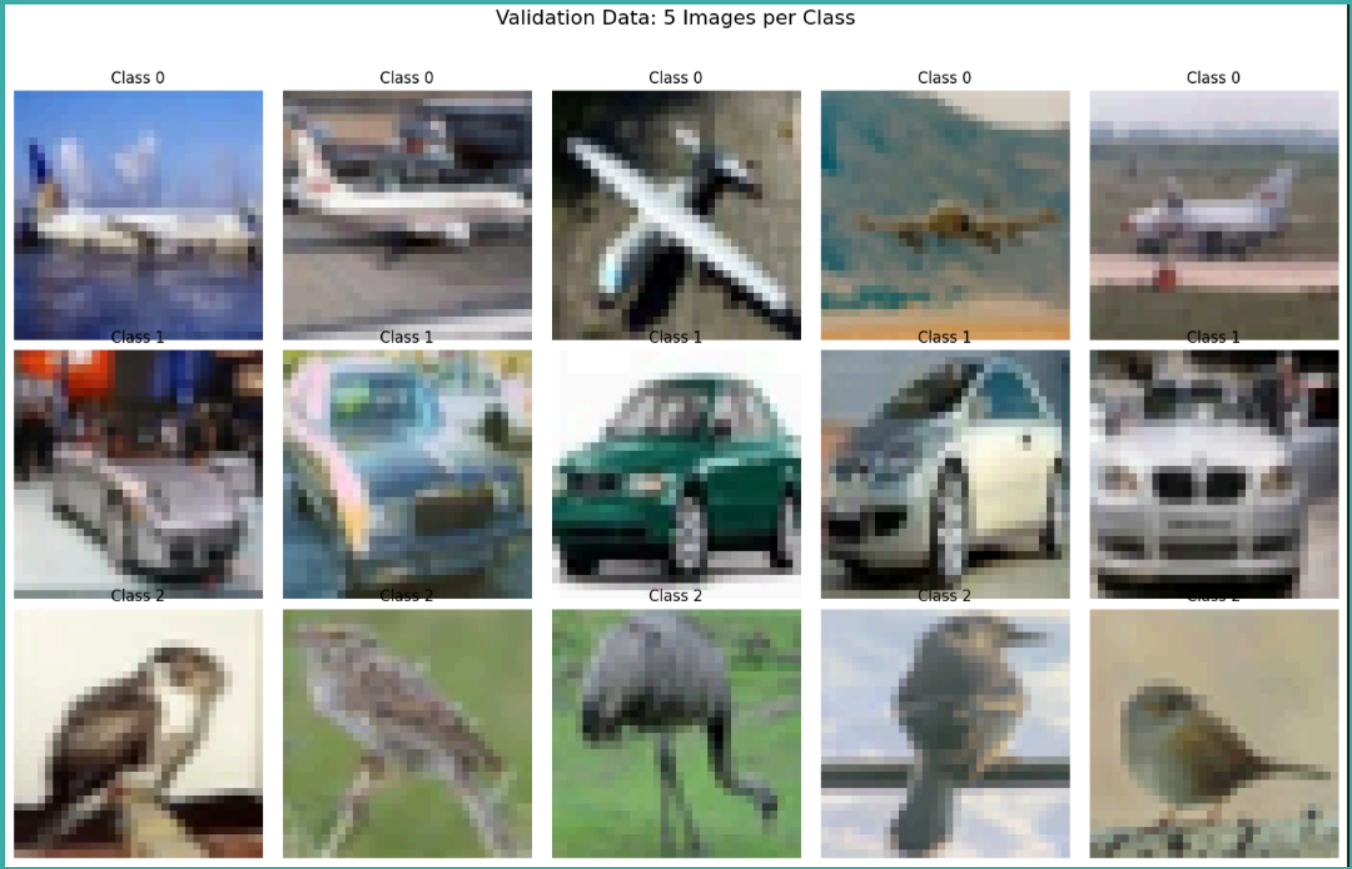
I created a custom Dataset class for the data and created data loaders for all the dataset splits - train, val, and test. Here, the 15,000 images from the training dataset are split into train-val via 80:20 split, and 3,000 images are retained as the testing data from the original test dataset . The output for the same is as follows:-

```
Train size: 12000
Validation size: 3000
Test size: 3000
```

2. <u>Visualization:-</u>

The image of each class for both training and validation data can be found below:-


Training Data: 5 Images per Class
Class 0
Class 0
Class 0
Class 0
Class 0
Class 1
Class 1
Class 1
Class 1
Class 1
Class 2
Class 2
Class 2
Class 2
Class 2


Validation Data: 5 Images per Class
Class 0
Class 0
Class 0
Class 0
Class 0
Class 1
Class 1
Class 1
Class 1
Class 1
Class 2
Class 2
Class 2
Class 2
Class 2

3. CNN Implementation:-

I have implemented the following CNN class satisfying all requirements with proper flattening that was asked in the question:-

```python
class CustomCNN(nn.Module):
    def __init__(self, num_classes=3):
        super(CustomCNN, self).__init__()
        self.conv1 = nn.Conv2d(in_channels=3, out_channels=16, kernel_size=5, stride=1, padding=1)
        self.pool1 = nn.MaxPool2d(kernel_size=3, stride=2)
        self.conv2 = nn.Conv2d(in_channels=16, out_channels=32, kernel_size=3, stride=1, padding=0)
        self.pool2 = nn.MaxPool2d(kernel_size=3, stride=3)
        self.fc1 = nn.Linear(32 * 4 * 4, 16)
        self.fc2 = nn.Linear(16, num_classes)

    def forward(self, x):
        x = F.relu(self.conv1(x))
        x = self.pool1(x)
        x = F.relu(self.conv2(x))
        x = self.pool2(x)
        x = torch.flatten(x, 1)
        x = F.relu(self.fc1(x))
        x = self.fc2(x)
        return x

num_classes = len(chosen_classes)
model = CustomCNN(num_classes=num_classes)
print(model)
```

✓ 0.0s

```
CustomCNN(
  (conv1): Conv2d(3, 16, kernel_size=(5, 5), stride=(1, 1), padding=(1, 1))
  (pool1): MaxPool2d(kernel_size=3, stride=2, padding=0, dilation=1, ceil_mode=False)
  (conv2): Conv2d(16, 32, kernel_size=(3, 3), stride=(1, 1))
  (pool2): MaxPool2d(kernel_size=3, stride=3, padding=0, dilation=1, ceil_mode=False)
  (fc1): Linear(in_features=512, out_features=16, bias=True)
  (fc2): Linear(in_features=16, out_features=3, bias=True)
)
```

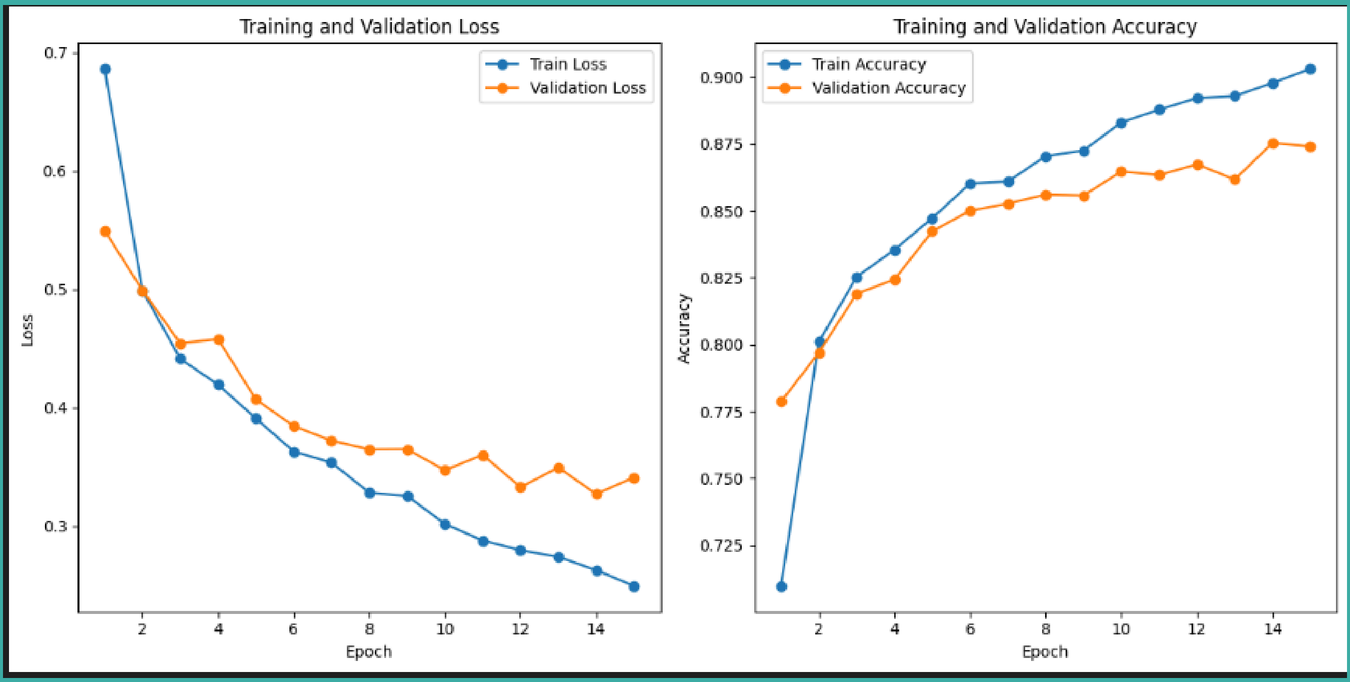
4. Model Training(CNN) :-

Implementation of the above CNN model has been done for all 15 epochs and all performance metrics has been recorded which can be found below :-

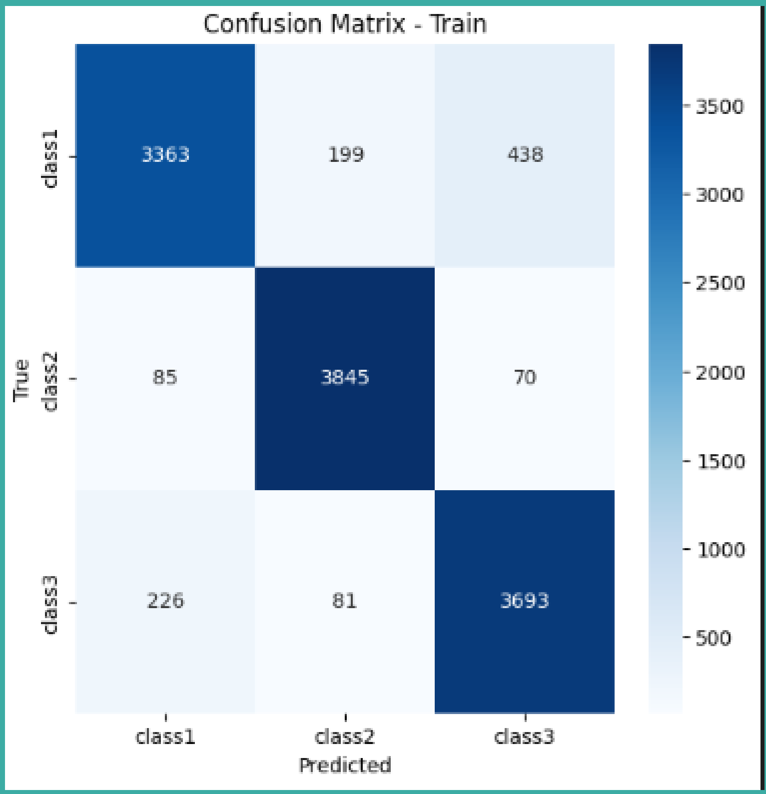
```
Model already trained and loaded.
Epoch [1/15] - Train Loss: 0.6863, Train Accuracy: 0.7098 - Val Loss: 0.5498, Val Accuracy: 0.7787
Epoch [2/15] - Train Loss: 0.4991, Train Accuracy: 0.8010 - Val Loss: 0.4993, Val Accuracy: 0.7970
Epoch [3/15] - Train Loss: 0.4414, Train Accuracy: 0.8253 - Val Loss: 0.4545, Val Accuracy: 0.8190
Epoch [4/15] - Train Loss: 0.4196, Train Accuracy: 0.8354 - Val Loss: 0.4581, Val Accuracy: 0.8243
Epoch [5/15] - Train Loss: 0.3910, Train Accuracy: 0.8472 - Val Loss: 0.4071, Val Accuracy: 0.8423
Epoch [6/15] - Train Loss: 0.3632, Train Accuracy: 0.8602 - Val Loss: 0.3846, Val Accuracy: 0.8500
Epoch [7/15] - Train Loss: 0.3537, Train Accuracy: 0.8609 - Val Loss: 0.3723, Val Accuracy: 0.8527
Epoch [8/15] - Train Loss: 0.3283, Train Accuracy: 0.8704 - Val Loss: 0.3649, Val Accuracy: 0.8560
Epoch [9/15] - Train Loss: 0.3256, Train Accuracy: 0.8725 - Val Loss: 0.3651, Val Accuracy: 0.8557
Epoch [10/15] - Train Loss: 0.3019, Train Accuracy: 0.8830 - Val Loss: 0.3473, Val Accuracy: 0.8647
Epoch [11/15] - Train Loss: 0.2878, Train Accuracy: 0.8878 - Val Loss: 0.3603, Val Accuracy: 0.8633
Epoch [12/15] - Train Loss: 0.2797, Train Accuracy: 0.8921 - Val Loss: 0.3329, Val Accuracy: 0.8673
Epoch [13/15] - Train Loss: 0.2741, Train Accuracy: 0.8928 - Val Loss: 0.3495, Val Accuracy: 0.8617
Epoch [14/15] - Train Loss: 0.2629, Train Accuracy: 0.8978 - Val Loss: 0.3272, Val Accuracy: 0.8753
Epoch [15/15] - Train Loss: 0.2496, Train Accuracy: 0.9030 - Val Loss: 0.3412, Val Accuracy: 0.8740
```

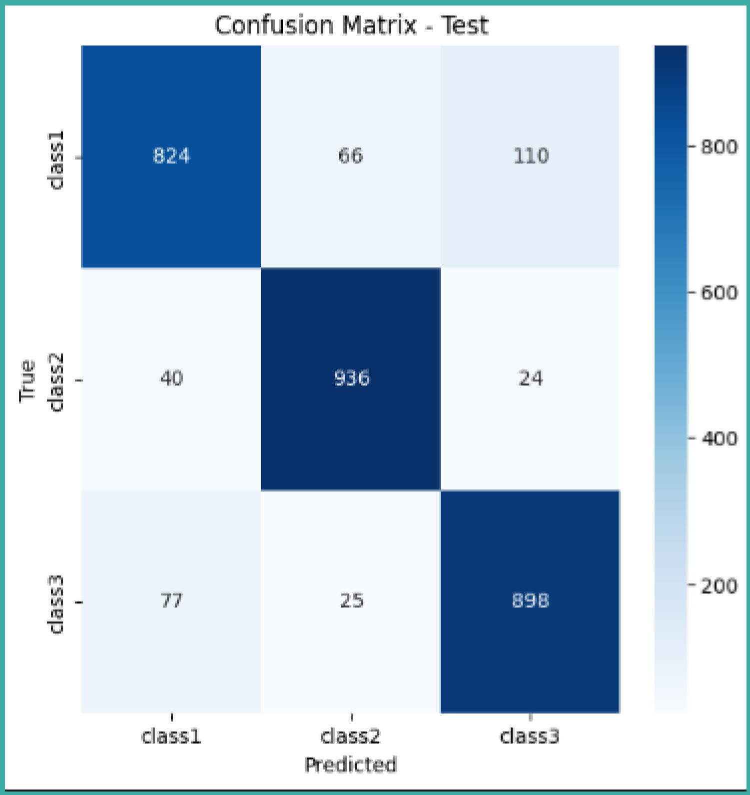
5. Observation (CNN) :-

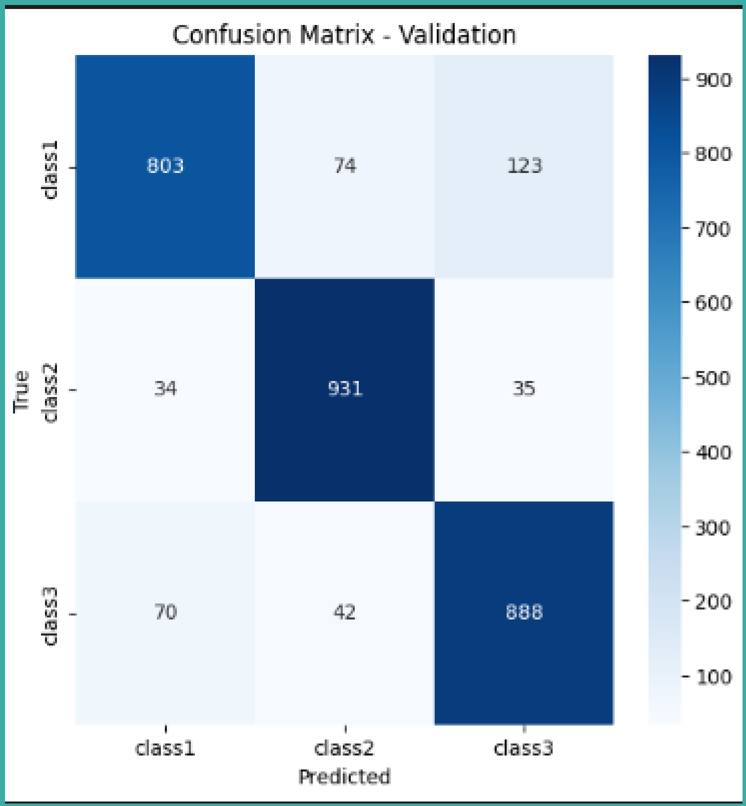
Confusion matrix , plots , F1-score and accuracy can be found below :-



```
Test Accuracy: 0.8860
Test F1-Score: 0.8855
```




Confusion Matrix - Test
824
66
110
40
936
24
77
25
898
class1
class2
class3
True
class1
class2
class3
Predicted
800
600
400
200


Confusion Matrix - Validation
803
74
123
34
931
35
70
42
888
class1
class2
class3
True
class1
class2
class3
Predicted
900
800
700
600
500
400
300
200
100

6. <u>Model Training(MLP)</u> :-

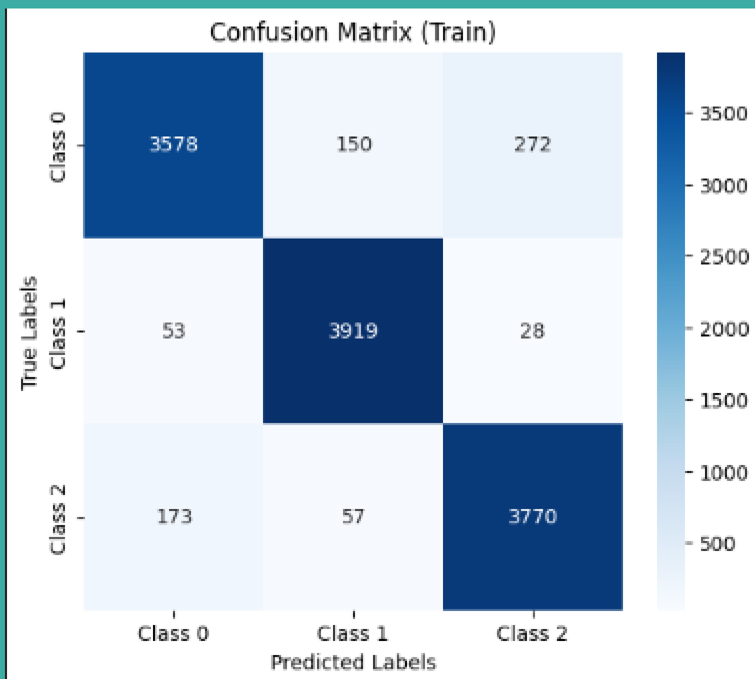
Implementation of the above MLP model has been done for all 15 epochs and all performance metrics has been recorded which can be found below :-

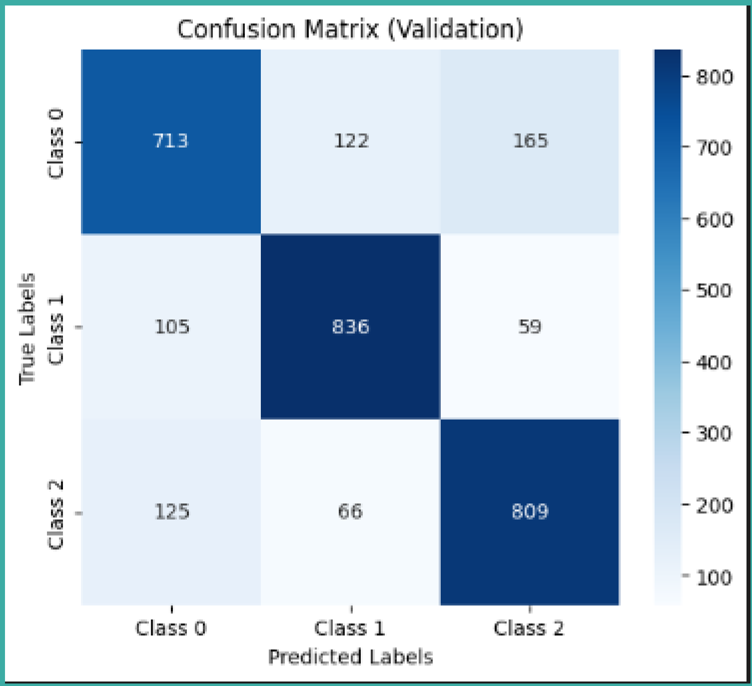
```
Model already trained and loaded.
Training and Validation Metrics:
Epoch [1/15] - Train Loss: 0.7128, Train Accuracy: 0.6973 - Val Loss: 0.5457, Val Accuracy: 0.7777
Epoch [2/15] - Train Loss: 0.5058, Train Accuracy: 0.7965 - Val Loss: 0.4800, Val Accuracy: 0.8087
Epoch [3/15] - Train Loss: 0.4494, Train Accuracy: 0.8213 - Val Loss: 0.4479, Val Accuracy: 0.8227
Epoch [4/15] - Train Loss: 0.4112, Train Accuracy: 0.8408 - Val Loss: 0.4315, Val Accuracy: 0.8273
Epoch [5/15] - Train Loss: 0.3857, Train Accuracy: 0.8456 - Val Loss: 0.4321, Val Accuracy: 0.8360
Epoch [6/15] - Train Loss: 0.3667, Train Accuracy: 0.8556 - Val Loss: 0.3843, Val Accuracy: 0.8473
Epoch [7/15] - Train Loss: 0.3475, Train Accuracy: 0.8623 - Val Loss: 0.3684, Val Accuracy: 0.8567
Epoch [8/15] - Train Loss: 0.3283, Train Accuracy: 0.8703 - Val Loss: 0.3847, Val Accuracy: 0.8503
Epoch [9/15] - Train Loss: 0.3227, Train Accuracy: 0.8715 - Val Loss: 0.3528, Val Accuracy: 0.8687
Epoch [10/15] - Train Loss: 0.3085, Train Accuracy: 0.8794 - Val Loss: 0.3365, Val Accuracy: 0.8693
Epoch [11/15] - Train Loss: 0.2971, Train Accuracy: 0.8842 - Val Loss: 0.3408, Val Accuracy: 0.8653
Epoch [12/15] - Train Loss: 0.2908, Train Accuracy: 0.8858 - Val Loss: 0.3363, Val Accuracy: 0.8733
Epoch [13/15] - Train Loss: 0.2802, Train Accuracy: 0.8918 - Val Loss: 0.3412, Val Accuracy: 0.8687
Epoch [14/15] - Train Loss: 0.2703, Train Accuracy: 0.8942 - Val Loss: 0.3188, Val Accuracy: 0.8783
Epoch [15/15] - Train Loss: 0.2593, Train Accuracy: 0.9000 - Val Loss: 0.3676, Val Accuracy: 0.8603
```

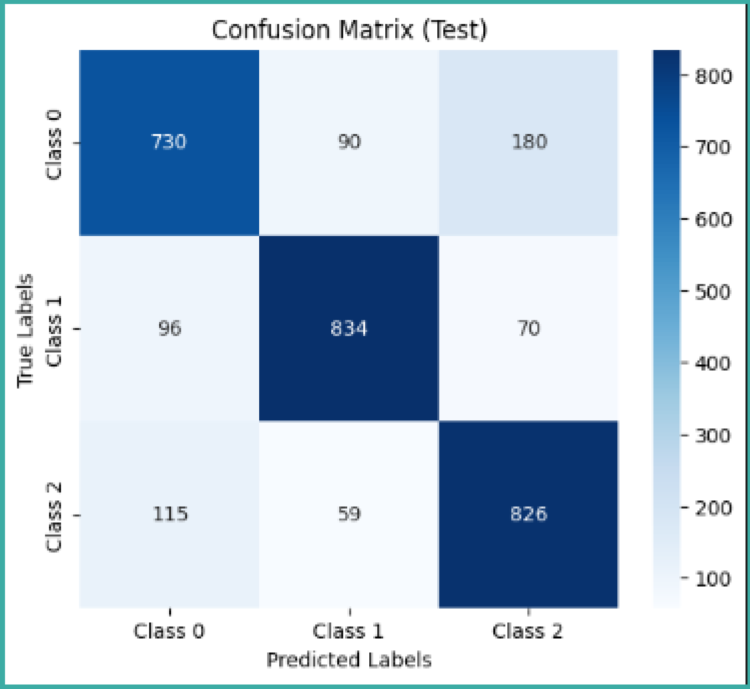
7. <u>Observation (MLP)</u> :-

Confusion matrix , plots , F1-score and accuracy can be found below :-

```
Test Accuracy: 0.7967
Test F1-Score: 0.7964
```




Confusion Matrix (Validation)
True Labels
Class 0
Class 1
Class 2
713
122
165
105
836
59
125
66
809
Predicted Labels
800
700
600
500
400
300
200
100

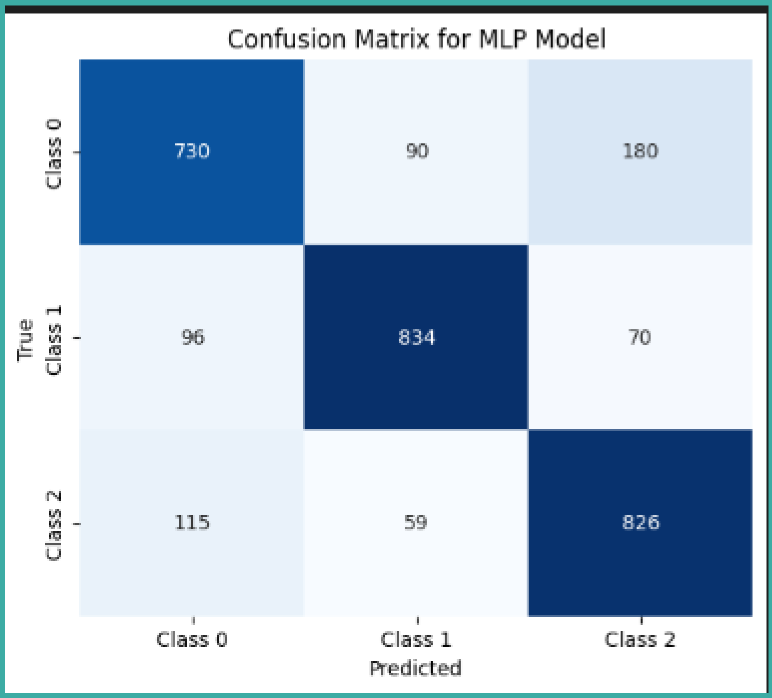

Confusion Matrix (Test)
True Labels
Class 0
Class 1
Class 2
730
90
180
96
834
70
115
59
826
Predicted Labels
800
700
600
500
400
300
200
100

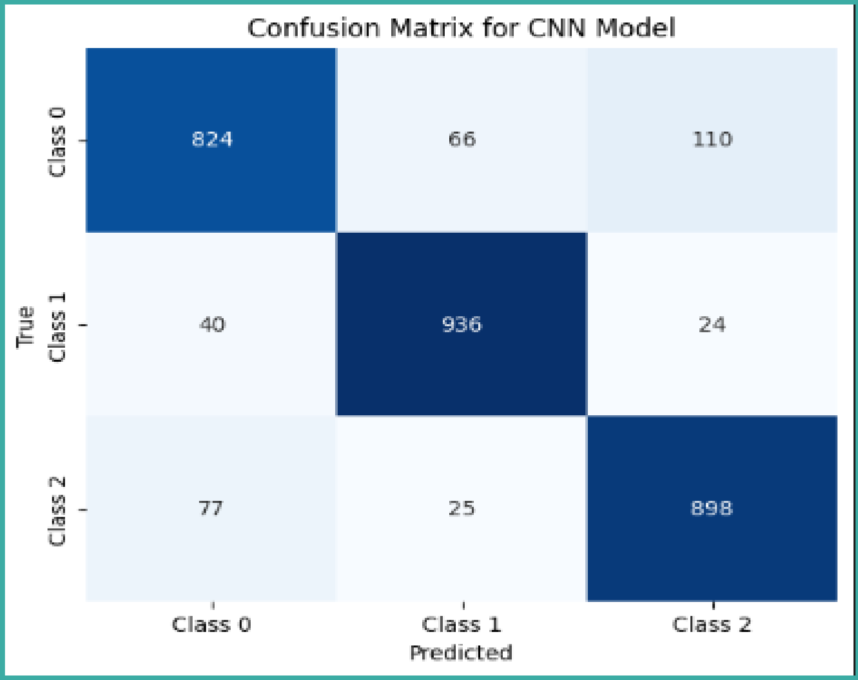
8. Comparison :-

Since the plots are already posted above so the key insights of them is written below :-

The MLP model has a steady improvement in accuracy and decreasing loss over epochs, which means it is learning effectively. However, the validation loss is still higher than the training loss, which indicates some overfitting. The CNN model has a higher accuracy than the MLP model, which means that it generalizes better. The training and validation loss curves for the CNN model are more stable and consistent, with less overfitting compared to the MLP model.

```
MLP Model - Test Accuracy: 0.7967, F1-score: 0.7964
CNN Model - Test Accuracy: 0.8860, F1-score: 0.8855
```


Confusion Matrix for MLP Model
730
90
180
96
834
70
115
59
826
Class 0
Class 1
Class 2
True
Class 0
Class 1
Class 2
Predicted


Confusion Matrix for CNN Model
824
66
110
40
936
24
77
25
898
Class 0
Class 1
Class 2
True
Class 0
Class 1
Class 2
Predicted