ML Challenge

Problem 1:

Convolutional neural networks have been really popular in different domains like Images, Time series etc. Use the 'ChlorineConcentration' dataset from the attachment and design a classifier using CNN. The training and test samples ratio should be 70% and 30% respectively. The minimum expected test accuracy is 90%. Feel free to try with different hyperparameters and include all the insights in your documentation.

Solution:

The python script can be found in Jupyter Notebook – 'Problem 1.ipynb'

Step 1: The Chlorine Concentration Dataset – Contains totally 166 features per sample and 3840 samples with 3 classes.

```
In [4]: # Retrieving the Dataset onto a pandas dataframe
        filename = 'ChlorineConcentration'
        data = pd.read_csv(filename, header = None, delimiter = ",", index_col = False)

In [5]: data.head()
```

Out[5]:

	0	1	2	3	4	5	6	7	8	9	...	157	158	159	160	161	162	163	
0	2	1.4682	4.1208	3.6959	3.4915	2.9480	2.7986	2.2477	1.7425	-0.49291	...	0.19750	2.07350	1.360300	-0.25118	0.885280	-0.490750	-0.64965	-0.87
1	2	4.3120	4.1491	3.7562	3.6419	3.1497	3.0162	2.6206	2.2513	1.72710	...	-0.15069	0.98590	0.464110	0.25694	0.282000	-0.097757	-0.32864	-0.72
2	1	-3.7883	6.4407	5.1232	2.5096	3.3153	3.8406	2.1552	3.8766	0.55248	...	-0.22846	-0.32982	-0.224710	-0.22838	-0.234030	-0.228380	-0.22839	-0.22
3	3	2.9490	3.3273	2.9524	2.8522	2.4667	2.4092	2.1532	1.9940	1.65710	...	-0.46454	1.41400	-0.024253	-0.17260	-0.205170	-0.429010	-0.56181	-0.90
4	2	1.3853	3.6172	3.2082	3.0444	2.6086	2.5178	2.2109	1.9498	1.16690	...	-0.26982	1.78760	0.238090	-0.11554	0.016351	-0.394990	-0.54357	-0.85

5 rows × 167 columns

```
In [6]: scalar = MinMaxScaler()
        features = scalar.fit_transform(data.iloc[:,1:167].values)
```

Figure 1: Chlorine Concentration Dataset

The data set is further normalized using standard MinMaxScaler() in the range (0,1).

Step 2: Fundamental Principal Component Analysis for Dimensionality Reduction and Visually Perceiving the Classes (166 features to 3 PC components)

```
In [8]: PC_Df.head()
```

Out[8]:

	PC1	PC2	PC3	Label
0	0.695876	-2.335562	0.634903	2
1	-0.655295	2.246137	0.102123	2
2	1.343061	2.329942	0.628292	1
3	-2.562263	0.158885	-0.000040	3
4	-2.247420	-0.413365	0.927342	2

Figure 2: PC components

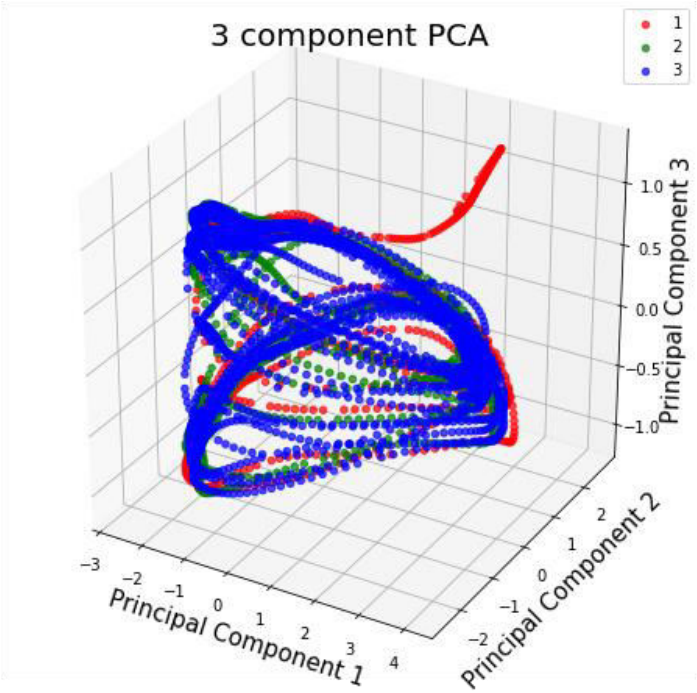

Figure 3: PC components plot colored class wise

In the above plot, the PC components of the 3 classes are represented. This indicates that the classes are not separable by simpler ML algorithms.

Step 3: The final CNN model decided on is –

```
In [16]: model.summary()
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv1d_1 (Conv1D)            (None, 146, 64)           1408
_________________________________________________________________
conv1d_2 (Conv1D)            (None, 126, 64)           86080
_________________________________________________________________
max_pooling1d_1 (MaxPooling1 (None, 63, 64)            0
_________________________________________________________________
flatten_1 (Flatten)          (None, 4032)              0
_________________________________________________________________
dense_1 (Dense)              (None, 64)                258112
_________________________________________________________________
dense_2 (Dense)              (None, 64)                4160
_________________________________________________________________
dense_3 (Dense)              (None, 3)                 195
=================================================================
Total params: 349,955
Trainable params: 349,955
Non-trainable params: 0
_________________________________________________________________
```

Figure 4: Final CNN Model architecture

And the training result plots

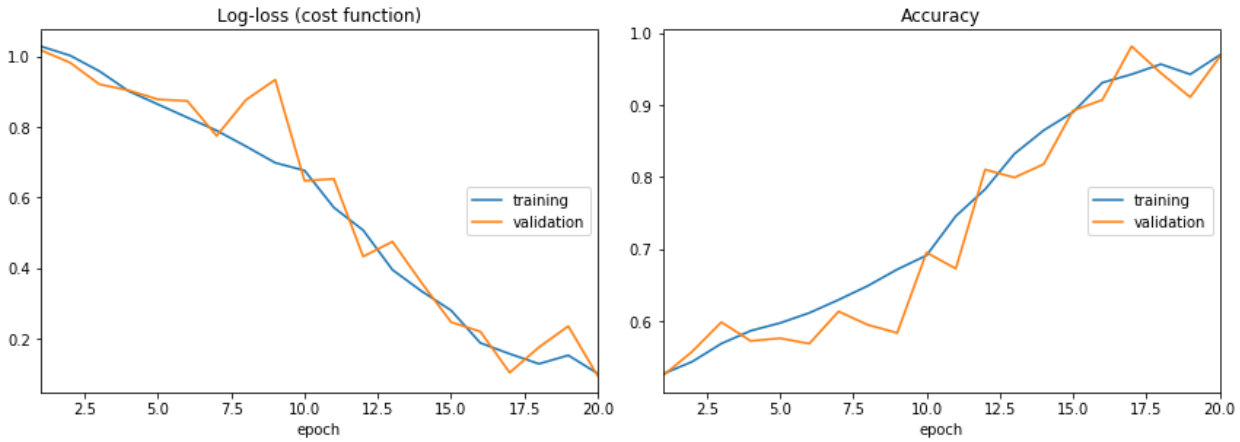

Figure 5: Final CNN Model results

The model reaches a test accuracy of 97.22% in 20 epochs.

Model Insights –

The Finalized model has two convolution layers (kernel size = 21) at the start to make use of the spatial information in the dataset. Further it has two dense layers at end. Optimizer is 'Adam', the loss function is 'categorical cross-entropy 'and the activation function used is 'Relu'.

Other Insights –

- Large Kernel size in the convolution layers is advantageous as the training time is reduced drastically E.g.:- using Kernel size = 9, the training time was almost 3 times as compared to Kernel size = 21. But showed similar results.

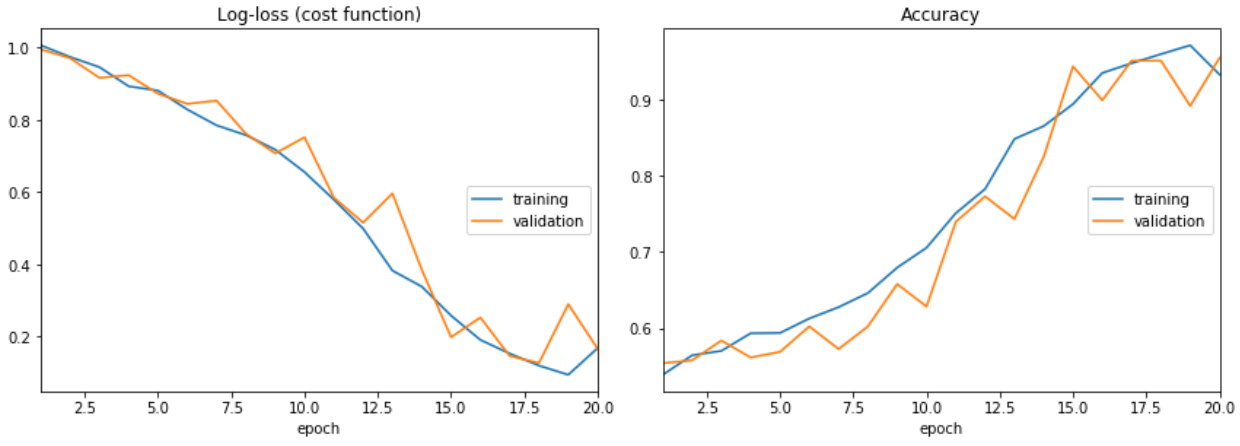

Figure 6: Results for Kernel size = 9

- Using Dropout layers (Dropout of 20%) in the model actually degraded the performance of the model. The convergence was slightly slower (see Fig 7).

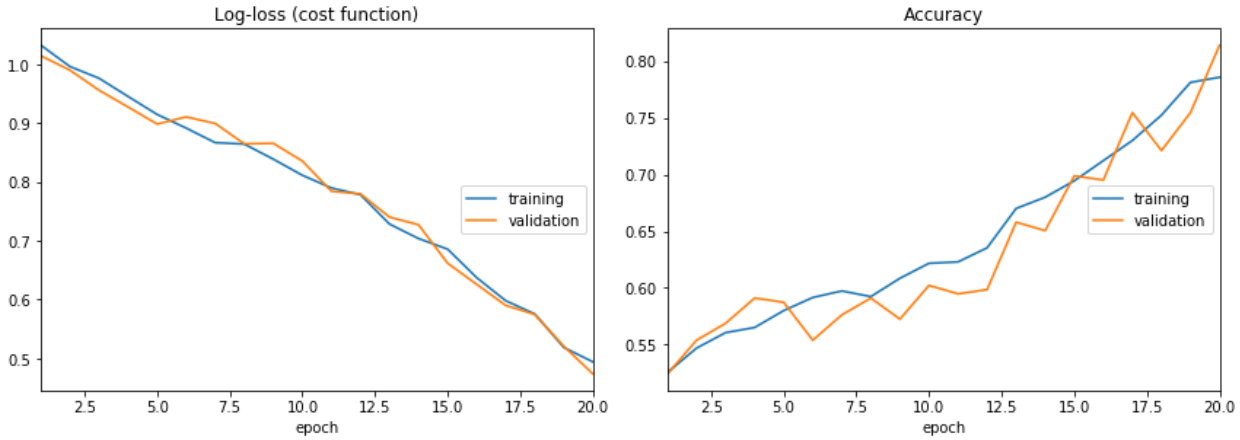

Figure 7: Results for model with dropout (20%)

- Using sigmoid as the activation function and stochastic gradient descent as the optimizer also yielded poor results.

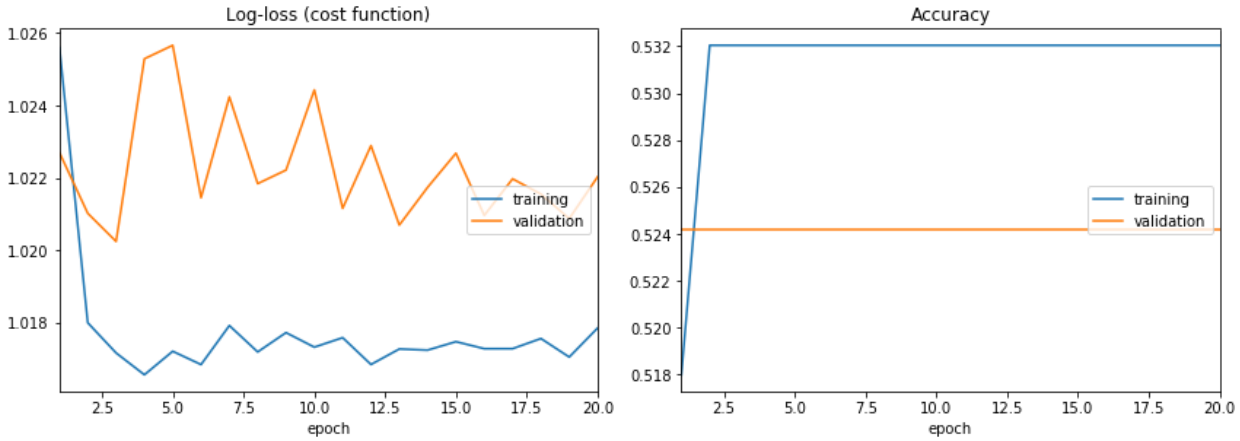

Figure 8: Results for model with (Activation='sigmoid' & Optimizer=' stochastic gradient descent')

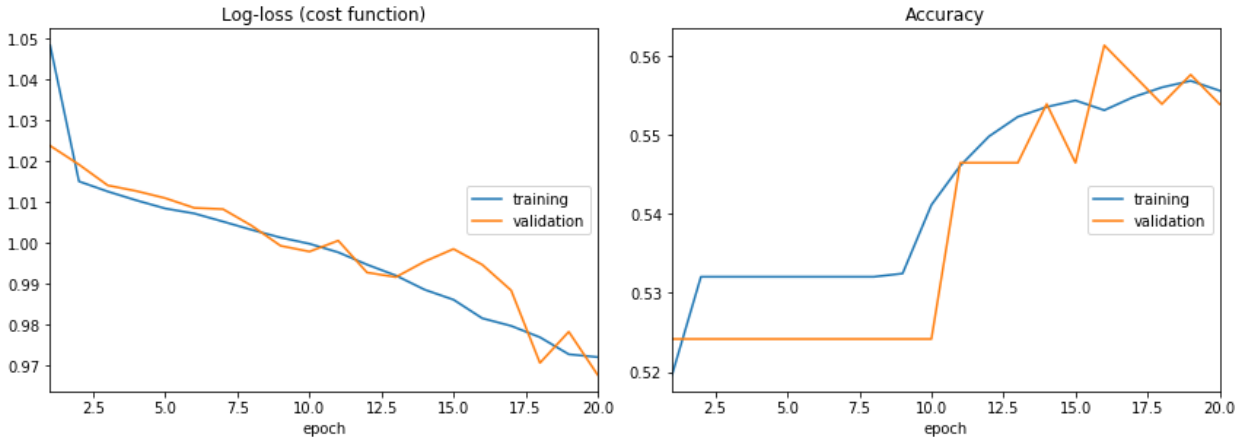

Figure 9: Results for model with (Activation='Relu' & Optimizer=' stochastic gradient descent')

Problem 2:

Autoencoder is a very popular deep learning model for unsupervised learning. The most amazing fact about Autoencoder is, it has capability of representing data in low dimensional space and extracting interesting patterns. Use Autoencoder to perform clustering over 'ElectricDevices' dataset and the minimum expectation is to bring good results than any traditional clustering algorithm (preferrably KMeans). Take a look at the paper link below and you can use the architecture proposed in the paper (You can use other architecture if you are not convinced by this paper, don't forget to cite the paper in your document).

Solution:

The python script can be found in Jupyter Notebook – 'Problem 2.ipynb'

Step 1: The Chlorine Concentration Dataset – Contains totally 96 features per sample and 7711 samples with 7 classes.

```
In [3]: # Retrieving the Dataset onto a pandas dataframe
        filename = 'ElectricDevices'
        data = pd.read_csv(filename, header = None, delimiter = ",", index_col = False)

In [4]: data.head()
```

Out[4]:

	0	1	2	3	4	5	6	7	8	9	...	87	88	89	90	91	92	
0	6	-0.19621	-0.19621	-0.19621	-0.19621	-0.19621	-0.19621	-0.19621	-0.19621	-0.19621	...	-0.19621	-0.19621	-0.19621	-0.19621	-0.19621	-0.19621	-0.1
1	6	-0.24118	-0.24118	-0.24118	-0.24118	-0.24118	-0.24118	-0.24118	-0.24118	-0.24118	...	-0.24118	-0.24118	-0.24118	-0.24118	-0.24118	-0.24118	-0.2
2	6	-0.23606	-0.23606	-0.23606	-0.23606	-0.23606	-0.23606	-0.23606	-0.23606	-0.23606	...	-0.23606	-0.23606	-0.23606	-0.23606	-0.23606	-0.23606	-0.2
3	6	-0.21032	-0.21032	-0.21032	-0.21032	-0.21032	-0.21032	-0.21032	-0.21032	-0.21032	...	-0.21032	-0.21032	-0.21032	-0.21032	-0.21032	-0.21032	-0.2
4	6	-0.17117	-0.17117	-0.17117	-0.17117	-0.17117	-0.17117	-0.17117	-0.17117	-0.17117	...	-0.17117	-0.17117	-0.17117	-0.17117	-0.17117	-0.17117	-0.1

5 rows × 97 columns

Figure 10: Electric Devices Dataset

The data set is further normalized using standard MinMaxScaler() in the range (0,1).

Step 2: Fundamental Principal Component Analysis for Dimensionality Reduction and Visually Perceiving the Classes (96 features to 3 PC components)

```
In [8]: PC_Df.head()
```

Out[8]:

	PC1	PC2	PC3	Label
0	0.155385	-0.082710	0.216273	6
1	0.198281	-0.085367	0.252602	6
2	0.282354	-0.069872	0.020575	6
3	0.156165	-0.095153	0.199286	6
4	0.106070	0.057237	-0.134120	6

Figure 11: PC components

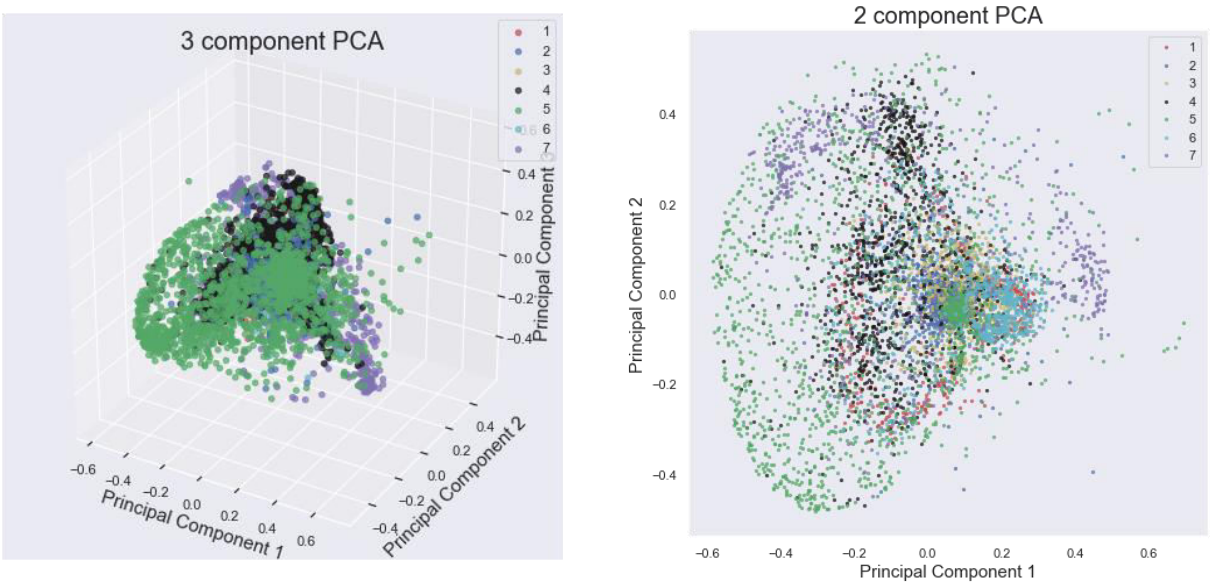

Figure 12: 3d and 2d PC components plot colored class wise

In the above plot, the PC components of the 7 classes are represented. Each class seems to naturally make a cluster groups to some degree, but there also is a presence of merging of these clusters at the cluster boundaries.

Step 3: K-means Clustering

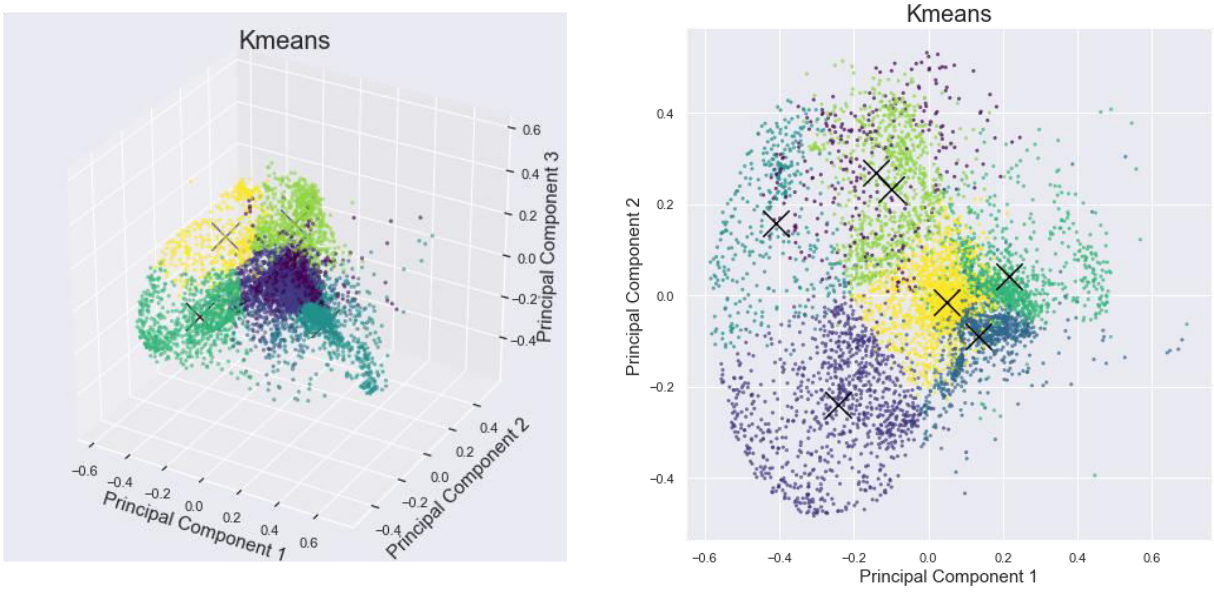

Figure 13: K-means cluster groups (7 clusters) with centroids marked with X

The K-means algorithm clusters data by trying to separate samples in n groups of equal variance, minimizing a criterion known as the inertia or within-cluster sum-of-squares. Cluster inertia is a

measure of how internally coherent clusters are. Inertia of the above clustering is 166.76. Equation (1) [1] is used to calculate inertia -

$$\sum_{i=0}^{n} \min_{\mu_j \in C} \left(\|x_i - \mu_j\| \right)^2 \quad (1)$$

x_i – Sample points

μ_j – Centroid of the cluster with contains the respective sample point x_i

Drawbacks of K-means -

- Inertia makes the assumption that clusters are convex and isotropic, which is not always the case.
- It responds poorly to elongated clusters, or manifolds with irregular shapes.
- Inertia is not a normalized metric: we just know that lower values are better and zero is optimal.
- But in very high-dimensional spaces, Euclidean distances tend to become inflated (this is an instance of the so-called "curse of dimensionality"). Running a dimensionality reduction algorithm such as PCA (see Step 2 and Step 3) prior to k-means clustering can alleviate this problem and speed up the computations [1].

Step 4: Autoencoder based clustering – The autoencoder architecture is similar to artificial neural network, and is used to efficiently encode a large set of features in an unsupervised manner. This aspect of the autoencoder can be used for dimensionality reduction [2].

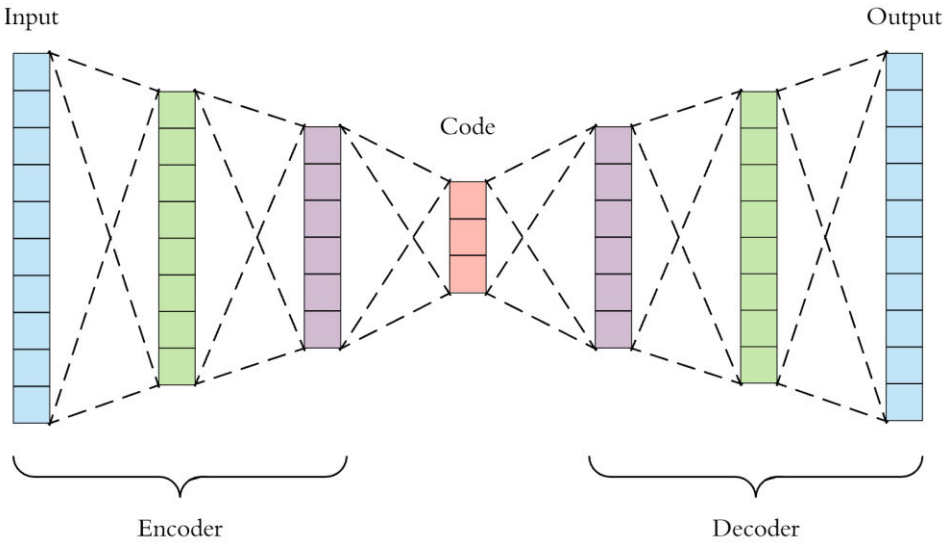

Figure 14: Autoencoder Architecture

The autoencoder architecture is similar to artificial neural network, and is used to efficiently encode a large set of features in an unsupervised manner. This aspect of the autoencoder can be used for dimensionality reduction [2].

```
Layer (type)                 Output Shape              Param #
=================================================================
input_1 (InputLayer)         (None, 96)                0
_________________________________________________________________
dense_1 (Dense)              (None, 100)               9700
_________________________________________________________________
leaky_re_lu_1 (LeakyReLU)    (None, 100)               0
_________________________________________________________________
dense_2 (Dense)              (None, 50)                5050
_________________________________________________________________
leaky_re_lu_2 (LeakyReLU)    (None, 50)                0
_________________________________________________________________
dense_3 (Dense)              (None, 3)                 153
_________________________________________________________________
leaky_re_lu_3 (LeakyReLU)    (None, 3)                 0
_________________________________________________________________
dense_4 (Dense)              (None, 50)                200
_________________________________________________________________
leaky_re_lu_4 (LeakyReLU)    (None, 50)                0
_________________________________________________________________
dense_5 (Dense)              (None, 100)               5100
_________________________________________________________________
leaky_re_lu_5 (LeakyReLU)    (None, 100)               0
_________________________________________________________________
dense_6 (Dense)              (None, 96)                9696
=================================================================
Total params: 29,899
Trainable params: 29,899
Non-trainable params: 0
```

Figure 15: Final Autoencoder Model

```
encoder.summary()
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_1 (InputLayer)         (None, 96)                0
_________________________________________________________________
dense_1 (Dense)              (None, 100)               9700
_________________________________________________________________
leaky_re_lu_1 (LeakyReLU)    (None, 100)               0
_________________________________________________________________
dense_2 (Dense)              (None, 50)                5050
_________________________________________________________________
leaky_re_lu_2 (LeakyReLU)    (None, 50)                0
_________________________________________________________________
dense_3 (Dense)              (None, 3)                 153
_________________________________________________________________
leaky_re_lu_3 (LeakyReLU)    (None, 3)                 0
=================================================================
Total params: 14,903
Trainable params: 14,903
Non-trainable params: 0
_________________________________________________________________
```

Figure 16: Encoder Component of the model (96 features reduced to 3 numbers)

```
decoder.summary()
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_2 (InputLayer)         (None, 3)                 0
_________________________________________________________________
dense_4 (Dense)              (None, 50)                200
_________________________________________________________________
leaky_re_lu_4 (LeakyReLU)    (None, 50)                0
_________________________________________________________________
dense_5 (Dense)              (None, 100)               5100
_________________________________________________________________
leaky_re_lu_5 (LeakyReLU)    (None, 100)               0
_________________________________________________________________
dense_6 (Dense)              (None, 96)                9696
=================================================================
Total params: 14,996
Trainable params: 14,996
Non-trainable params: 0
_________________________________________________________________
```

Figure 17: Decoder Component of the model (3 numbers back to 96 features)

```
encoded_df.head()
```

Out[127]:

	0	1	2
0	1.543353	0.498389	1.438301
1	-0.201593	0.628729	-0.294568
2	0.511625	-0.044511	1.307779
3	0.769402	-0.203470	0.646634
4	0.667685	1.024911	-0.164925

Figure 18: Encoder output for the Electric Devices data

The Autoencoder network is trained with 'stochastic gradient descent' optimizer with a custom loss function (check Problem 2.ipynb). Multiple training cases indicate that it is ideal to have – learning rate low (< 0.3) for better results. While the custom loss function is designed based Algorithm-1 provided in paper [3].

Two variants of clustering error were determined and the goal is to achieve lesser inertia when the encoded data is clustered (k-means). The two variants include –

1) Euclidean Distance (encompasses all three dimensions of the encoded data) calculated for all from the cluster centroids and the encoded data. This distance component is added to the loss equation (mean square error)

$$Clustering\ error = \sqrt{(x - cx)^2 + (y - cy)^2 + (z - cz)^2} \quad (2)$$

$$Loss = mean\ square\ error\ (X, X_{predicted}) + \lambda \cdot Clustering\ error \quad (3)$$

2) Euclidean Distances of each dimension is separated and propagated thorough the decoder layers before adding them to the existing loss function (mean square error)

$$Encoder\ error = Vector\left(\sqrt{(x-cx)^2}, \sqrt{(y-cy)^2}, \sqrt{(z-cz)^2}\right) \quad (4)$$

$$Clustering\ error = Decoder(Encoder\ error) \quad (5)$$

$$Loss = mean\ square\ error\ (X, X_{predicted}) + \lambda \cdot Clustering\ error \quad (6)$$

The second variant performed better in reducing the overall inertia of the clustered output.

Test Cases:

<u>Learning Rate = 0.1; Lambda = 1.0; Epochs = 10</u>

Inertia = 34.05

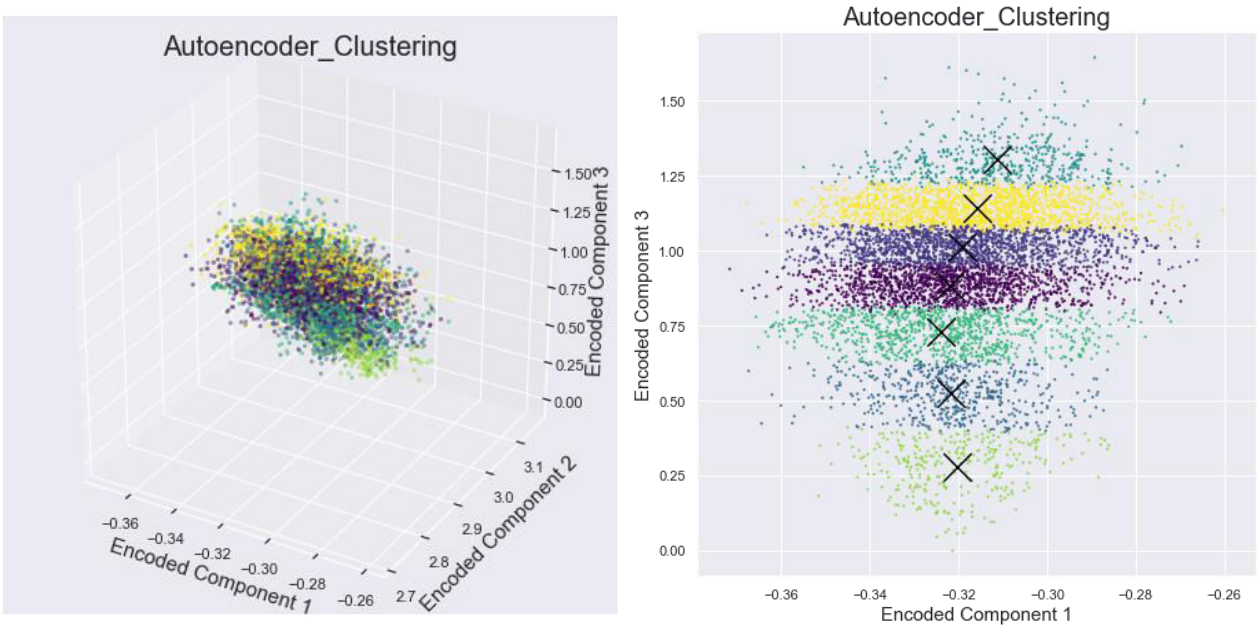

Figure 19: K-means cluster groups (7 clusters) of encoded data

Learning Rate = 0.1; Lambda = 10; Epochs = 10

Inertia = 19.08

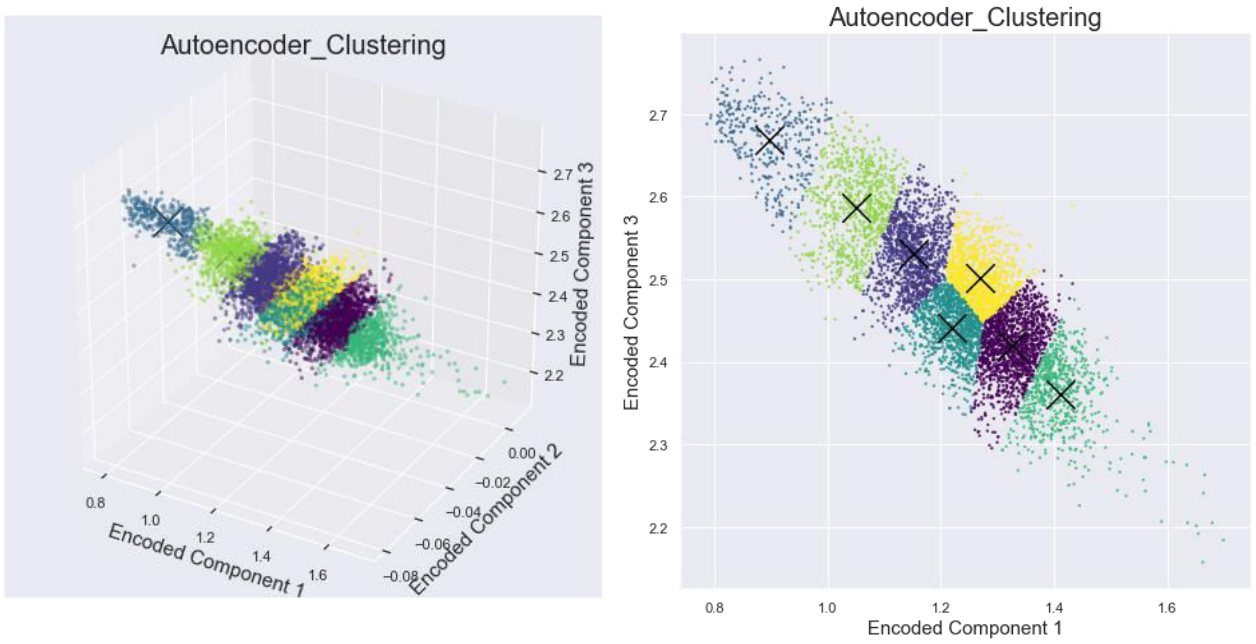

Figure 20: K-means cluster groups (7 clusters) of encoded data

Learning Rate = 0.3; Lambda = 10; Epochs = 10

Inertia = 31.28

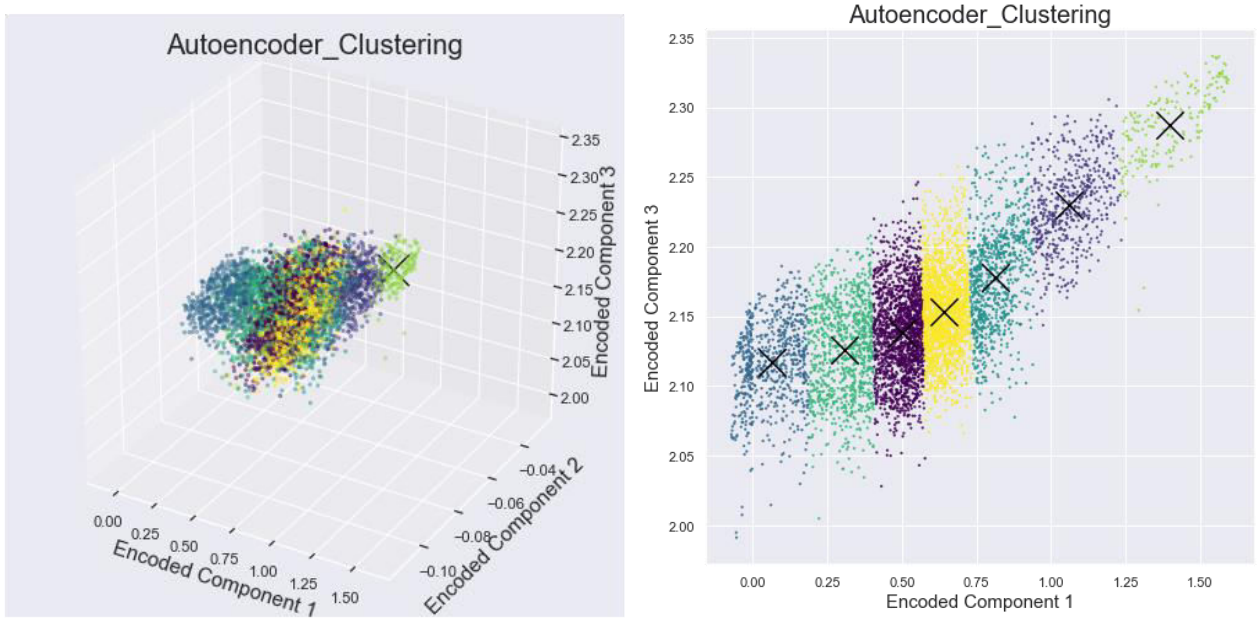

Figure 21: K-means cluster groups (7 clusters) of encoded data

Learning Rate = 0.6; Lambda = 10; Epochs = 10

Inertia = 506.48

Figure 21: K-means cluster groups (7 clusters) of encoded data

The criteria to be satisfied for the given task in accordance to K-means clustering are that, the inertia value has to be reduced. This has been achieved as the results achieved were <166.76 (K-means on PC Components). From the above test cases – the observations indicate that higher learning rate increases the inertia and higher lambda values reduce the inertia. However there are considerable limitations in the way k-means work as mentioned already.

References

[1] Scikit-learn.org. (2018). 2.3.Clustering — scikit-learn 0.20.0 documentation. [online] Available at: http://scikit-learn.org/stable/modules/clustering.html

[2] En.wikipedia.org (2018) Autoencoder - https://en.wikipedia.org/wiki/Autoencoder

[3] Song, C., Liu, F., Huang, Y., Wang, L. and Tan, T. (2013). Auto-encoder Based Data Clustering. Progress in Pattern Recognition, Image Analysis, Computer Vision, and Applications, pp.117-124.