Sample Code for an Autoencoder for Dr. Goswami

Shakil Rafi

Introduction and Dataset

In this notebook we will do a test-case with synthetic data comparing classic PCA with an autoencoder to see that the autoencoder is able to take high-dimensional data and extract out the relevant features.

Dataset

Our dataset will be a synthetic dataset in which we will use tha sklearn datasets method to create eight "blobs" of data and 30,000 observations. Each blob has a centroid around which the data is clustered in a Gaussian fashion with a standard deviation of 2, i.e. each element of the cluster c, x_c is such that $x_c \sim \mathcal{N}(\text{centroid}_c, c)$. We will have nine features to the data, to which we will add an extra feature, consisting of noise coming from a uniform distribution.

Note on dataset: The author regrets to inform that due to the ongoing shutdowns of internet and communications in Bangladesh that they have had difficulty and delay obtaining actual datasets. This Jupyter notebook therefore serves as a proof of concept that autoencoders can be beneficial and viable in analyzing genomic data via proxy with synthetic data. The author apologizes for this inconvenience.

Introduction and Methodology

Our methodology will be as follows:

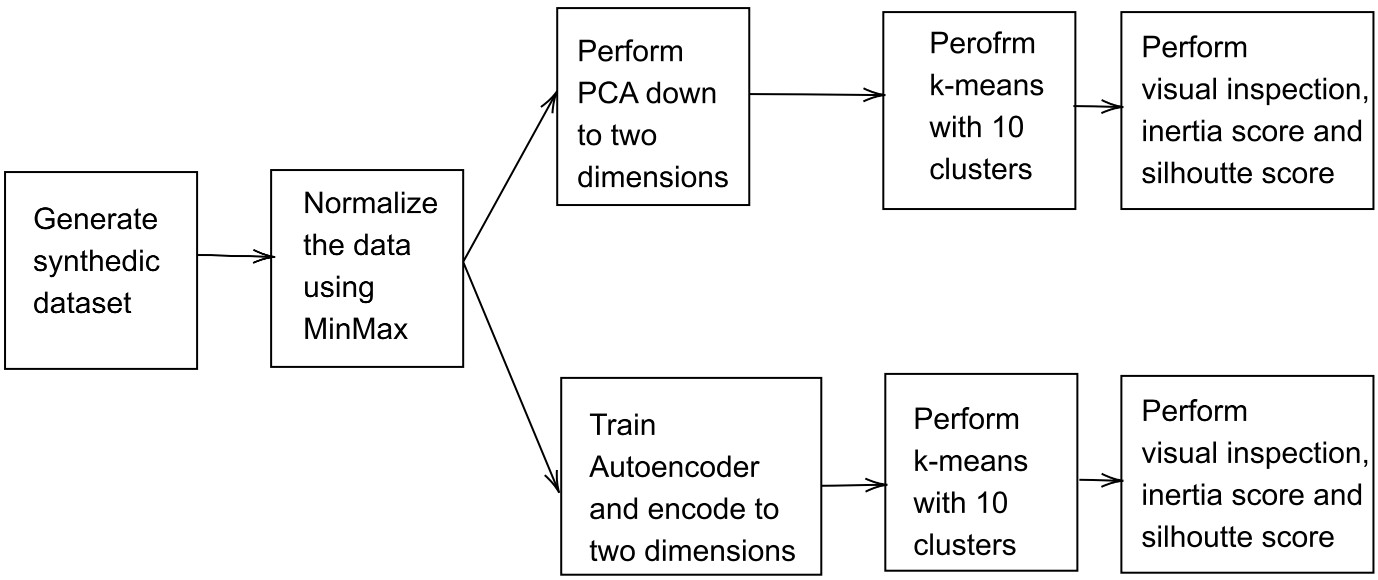


Figure 1: Methodology

In summary, we will take our synthetic dataset and perform normalization using the MinMax Scaler. Upon completion of the PCA transform we will perform a scatterplot and perform a basic visual inspection. Ideally, we would expect to see eight clusters, but even here we will see that PCA will fail dramatically. We will then perform a classic k-means clustering on this reduced space. We will then calculate the relevant scores inertia defined as:

$$\text{Inertia} = \sum_{i=1}^{n} \sum_{j=1}^{k} \parallel x_i - \mu_i \parallel^2 \cdot \mathbf{1}_{x_i \in C_i}$$

where n is the number of data points, k is the number of clusters μ_i is the centroid of each cluster.

Essentially, the inertial score tells us how "compact" each cluster is. Scores range from $[0, \infty)$. A lower score means a better cluster, although this is not the only metric for measuring clustering effectiveness. One other measure is silhoutte score, defined for a single datapoint x_i as:

$$\text{Silhoutte } (x_i) = \frac{b(x_i) - a(x_i)}{\max\{a(x_i), b(x_i)\}}$$

Where $a(x_i)$ is the average distance between x_i and all other points in the cluster, mathematicall speaking, this is:

$$a(x_i) = \frac{1}{|C_i| - 1} \left(\sum_{x_j \in C_k, x_j \neq x_i} \parallel x_i - x_j \parallel \right)$$

And where $b(x_i)$ is the minimum average distance between x_i and all other points in other clusters, mathematically speaking, this is:

$$b(x_i) = \min_{C_m \neq C_k} \left(\frac{1}{|C_m|} \sum_{x_j \in C_m} \parallel x_i - x_j \parallel \right)$$

```
####################################################################################
#
# The current cell installs packages if not already installed in the hosts
systems.
# This is supplemented by an extensive requirements.txt file in the folder
#
####################################################################################

# import subprocess
# import sys

# def install(package):
#     subprocess.check_call([sys.executable, "-m", "pip", "install", package])

# # List your packages here
# packages = ["numpy", "pandas", "seaborn","keras","tensorflow","matplotlib",
"scikit-learn"]

# for package in packages:
#     install(package)
```

```
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
import seaborn as sns
```

```
from sklearn.datasets import make_blobs
```

```



data = make_blobs(
    n_samples=30000,
    n_features=9,
    centers=8,
    cluster_std=2
```

```
)

X,y = data
```

```
# Create a random uniform column of noise to be added as a feature

np.random.seed(seed=101)
noise = np.random.uniform(size=len(X))
noise = pd.Series(noise)
```

```
# Add noise to our dataset. Note that since no one dimension is any
# special than another dimension it does not matter where we add
# the noise to

feat = pd.DataFrame(X)
feat = pd.concat([feat,noise],axis=1)
```

```
# Rename the columns to be human readable

feat.columns = [f"X{i+1}" for i in range(len(feat.columns))]
```

```
# Visually inspect our X

feat
```

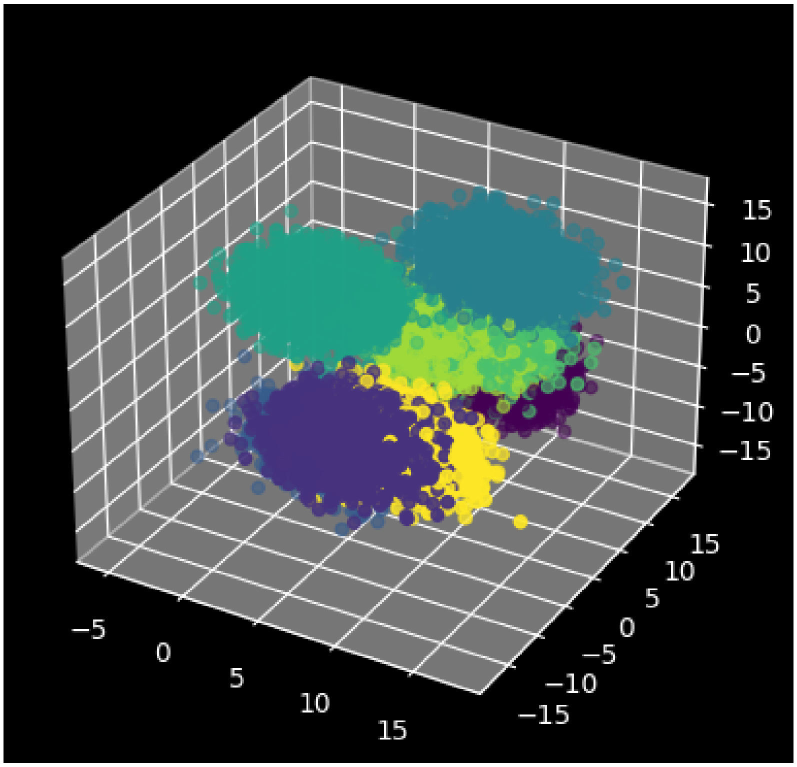
	X1	X2	X3	X4	X5	X6	X7	X8	X9	X10
0	3.832150	-0.885521	1.396813	6.518364	-7.181659	7.624980	9.863516	-3.958961	10.142876	0.516399
1	5.121997	-9.531866	-6.580018	3.576645	-3.133651	-3.158619	-7.563490	6.267551	-0.000102	0.570668
2	0.170303	-3.953752	8.746255	4.652394	1.368011	-0.426606	2.571776	-7.456172	-9.752707	0.028474
3	2.087428	-2.992573	5.104202	6.203770	0.087370	-4.681826	5.966582	-0.207810	-7.504352	0.171522
4	9.513307	6.389279	-2.908372	-0.090795	-6.912397	2.130114	-9.692870	5.897349	1.681667	0.685277
...	...	...	...	...	...	...	...	...	...	...
29995	13.124232	8.117214	8.370973	9.811386	0.173668	4.734505	2.027921	-5.618596	7.445326	0.655720
29996	2.451051	-4.309338	3.712033	5.494899	0.206910	-1.965390	4.748489	-4.381654	-7.359769	0.298044
29997	-0.500273	0.921920	2.131400	3.418651	-6.197750	11.418829	5.531861	2.188421	5.244825	0.537972
29998	3.285187	0.026805	6.310630	5.544782	-0.692804	-0.721316	4.952729	-3.549326	-3.656935	0.513805
29999	10.510771	-6.328579	12.581760	9.529466	-3.849963	-4.845036	3.824897	-1.863592	-5.485848	0.509921

```
# Summary statistics of our dataset for EDA

feat.describe()
```

	X1	X2	X3	X4	X5	X6	X7	X8	X9	X10
count	30000.000000	30000.000000	30000.000000	30000.000000	30000.000000	30000.000000	30000.000000	30000.000000	30000.000000	30000.000000
mean	6.880943	1.773305	−0.392525	5.431513	−2.360405	1.193830	1.046358	−0.117281	−0.424195	0.500133
std	3.450987	7.130533	7.150479	3.050602	4.372199	5.664284	6.621440	4.890498	5.874961	0.289540
min	−5.490365	−17.244616	−17.053376	−4.412489	−17.099587	−13.346374	−16.297391	−12.580435	−16.440136	0.000002
25%	4.730572	−4.391468	−7.417902	3.151284	−5.552574	−3.265249	−5.086935	−3.982851	−4.665434	0.247814
50%	7.459597	3.072980	0.064210	5.355945	−1.532697	0.390769	2.553161	−1.276105	−1.152058	0.499429
75%	9.374422	8.119963	6.353483	7.721868	1.037876	6.500394	6.235492	3.957791	3.830021	0.750924
max	17.518416	16.887564	14.705751	15.562946	10.309994	15.120596	16.109644	14.901781	15.729132	0.999985

```
# Save our dataset

feat.to_csv("feat.csv")
```

```
# 3D scatterplot of three arbitrary dimensions

fig = plt.figure()
ax = fig.add_subplot(111, projection = '3d')
ax.scatter(feat['X1'],feat['X2'],feat['X9'], c = y)
```



Preprocessing the data

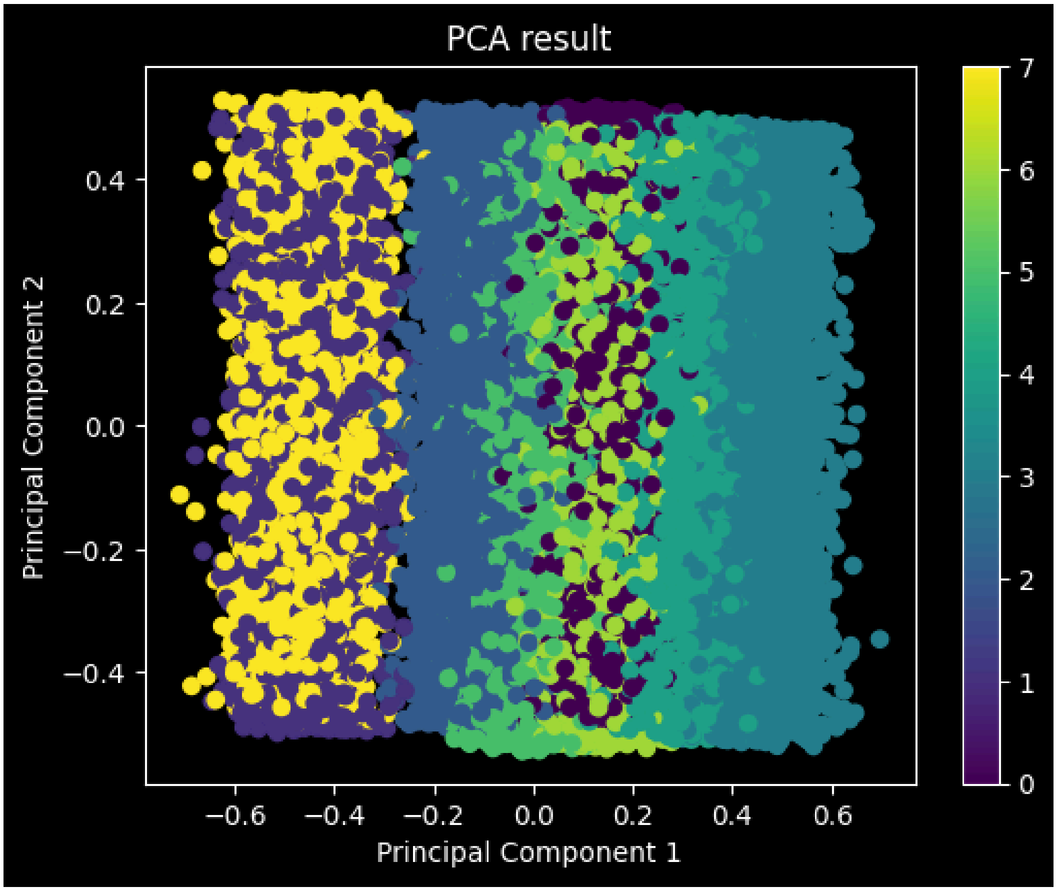
We will use the standard MinMaxScaler from sklearn to scale and preprocess the data

```
# Preprocess our data using MinMaxScaler

from sklearn.preprocessing import MinMaxScaler

scaler = MinMaxScaler()
scaled_data = scaler.fit_transform(feat)
```

The PCA decomposition

```
from sklearn.decomposition import PCA

pca = PCA(n_components=2)
pca_result = pca.fit_transform(scaled_data)
```

```
plt.scatter(pca_result[:, 0], pca_result[:, 1], c=y)
plt.title('PCA result')
plt.colorbar()
plt.xlabel("Principal Component 1")
plt.ylabel("Principal Component 2")
plt.show()
```



Note that PCA fails dramatically at this dimension reduction task. A priori we know that this dataset has clusters as it was constructed this way but PCA fails to capture the clustering. This is because PCA is a linear transformation and fails to capture any non-linear trends in the data. Note also that the uniform noise dimension results in a projection that is "smeared" yielding no obvious clusters on visual inspection. We will run two separate metrics later on to show how much more effective autoencoders are than regular PCA as a pre-step towards clustering.

```
from sklearn.metrics import mean_squared_error

X_reconstructed = pca.inverse_transform(pca_result)
print("Reconstruction loss of PCA:",mean_squared_error(X_reconstructed,
scaled_data))
```

```
Reconstruction loss of PCA: 0.021622192886071475
```

K-means done on the encoded data

We now seek to apply classic K-Means on this PCA-result. We will do the same on the

```
from sklearn.cluster import KMeans
```

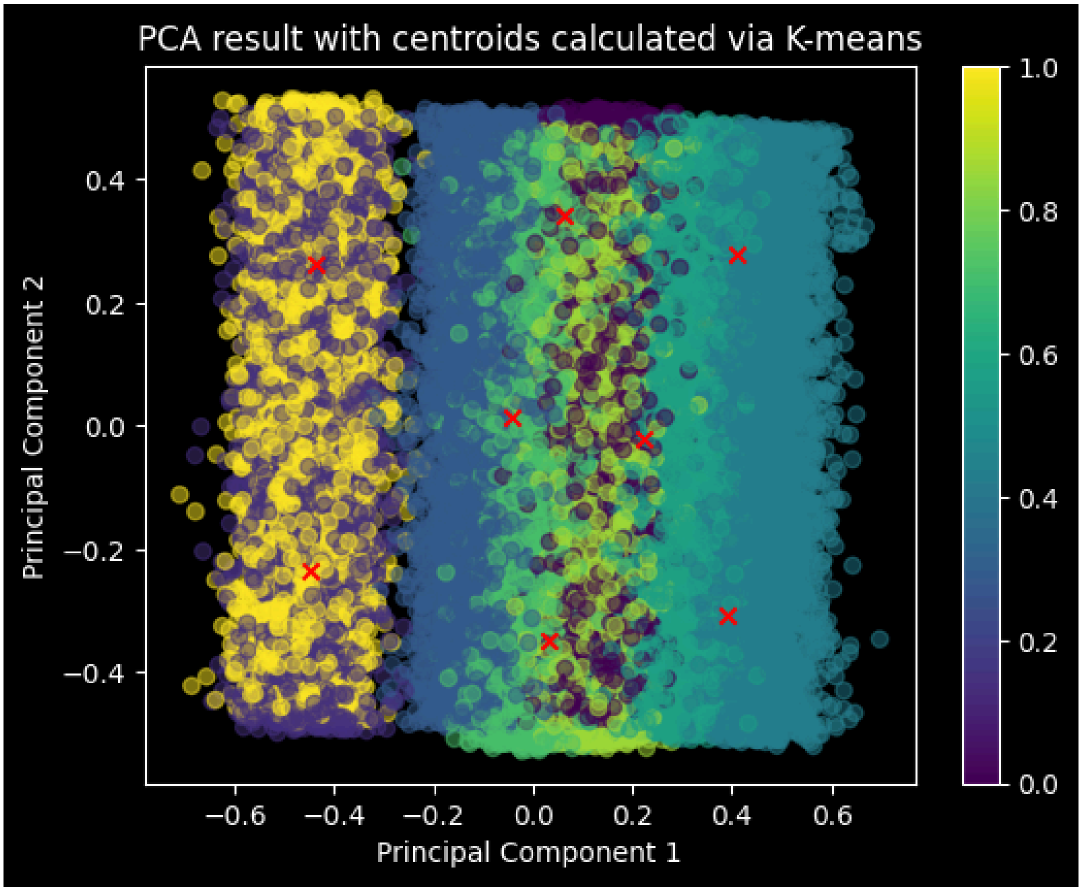
```
kmeans_for_pca = KMeans(n_clusters=8)
kmeans_for_pca.fit(pca_result)
```

```
KMeans()
```

```
kmeans_for_pca.cluster_centers_
```

```
array([[-0.44910219, -0.23529575],
       [-0.04385551,  0.0132341 ],
       [-0.43505372,  0.26351184],
       [ 0.03283007, -0.34681073],
       [ 0.39055912, -0.30722012],
       [ 0.40882084,  0.27793082],
       [ 0.22417627, -0.01915176],
       [ 0.06386291,  0.34105549]])
```

```
plt.scatter(pca_result[:, 0], pca_result[:, 1], c=y, alpha=0.5)
plt.scatter(kmeans_for_pca.cluster_centers_[:,
0],kmeans_for_pca.cluster_centers_[:,1],marker='x',c='red')
plt.title('PCA result with centroids calculated via K-means')
plt.colorbar()
plt.xlabel("Principal Component 1")
plt.ylabel("Principal Component 2")
plt.show()
```




```
kmeans_for_pca.inertia_
```

```
665.493846052085
```

The autoencoder decomposition

We will now create an autoencoder. We will choose a rather deep model, with 10 –> 5 –> 2 –> 5 –>10 layer widths. Our autoencoder will first encode the 10 dimensional data to a 2 dimensional latent space, and then decode it back to 10 dimensions. It will be trained against our dataset until it learns an optimal representation for this dataset into two latent dimensions.

We will then extract out the encoder and visually inspect our 2d latent space. Because neural networks are efficient at learning the underlying structure of unstructured data it's latent space should be noise-free, moreso than the PCA representation. As such, even visually we should see much more clustering than with PCA. Schematically, our autoencoder looks as follows:

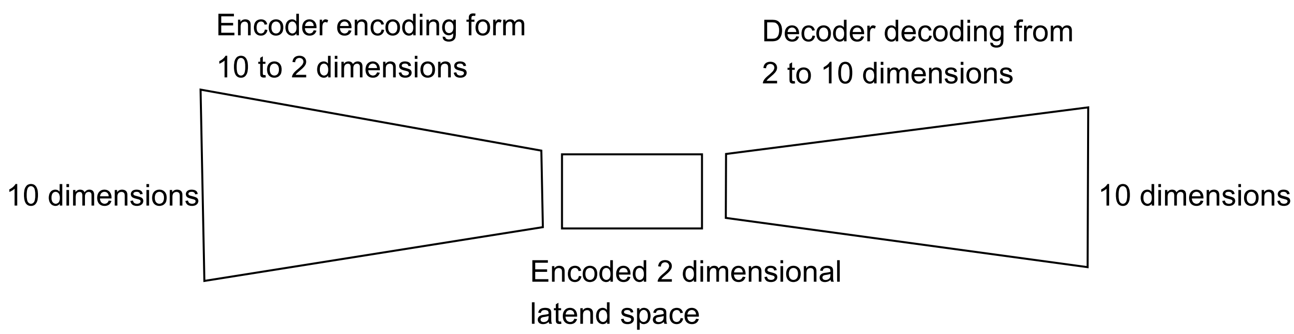


Figure 2: Autoencoder

```
from tensorflow.keras.models import Model
from tensorflow.keras.layers import Input, Dense
from tensorflow.keras.optimizers import SGD
```

```
# # This is the input layer
# input_layer = Input(shape=(scaled_data.shape[1],))

# # This is the encoding layers
# encoded = Dense(8, activation='relu')(input_layer)
# encoded = Dense(6, activation='relu')(encoded)
# encoded = Dense(2, activation='relu')(encoded)

# # This is the decoding layers
# decoded = Dense(2, activation='relu')(encoded)
# decoded = Dense(6, activation='relu')(decoded)
# decoded = Dense(8, activation='relu')(decoded)
# decoded = Dense(scaled_data.shape[1], activation='linear')(decoded)


# # This is the full autoencoder
# autoencoder = Model(inputs=input_layer, outputs=decoded)
```

```
input_layer = Input(shape=(scaled_data.shape[1],))

# Define the encoding layers
encoded = Dense(8, activation='sigmoid')(input_layer)
encoded = Dense(6, activation='sigmoid')(encoded)
encoded_output = Dense(2, activation='linear')(encoded)
```

```
# Define the encoder model
encoder = Model(inputs=input_layer, outputs=encoded_output)

# Define the decoding layers
encoded_input = Input(shape=(2,))
decoded = Dense(6, activation='sigmoid')(encoded_input)
decoded = Dense(8, activation='sigmoid')(decoded)
decoded_output = Dense(scaled_data.shape[1], activation='sigmoid')(decoded)

# Define the decoder model
decoder = Model(inputs=encoded_input, outputs=decoded_output)

# Define the autoencoder model
autoencoder_input = Input(shape=(scaled_data.shape[1],))
encoded_repr = encoder(autoencoder_input)
reconstructed = decoder(encoded_repr)

autoencoder = Model(inputs=autoencoder_input, outputs=reconstructed)

# Print model summaries
print("Encoder summary:")
encoder.summary()
print("\nDecoder summary:")
decoder.summary()
print("\nAutoencoder summary:")
autoencoder.summary()
```

```
Encoder summary:

Decoder summary:

Autoencoder summary:
```

```
Model: "functional"
```

Layer (type)	Output Shape	Param #
input_layer (InputLayer)	(None, 10)	0
dense (Dense)	(None, 8)	88
dense_1 (Dense)	(None, 6)	54

dense_2 (Dense)	(None, 2)	14

```
Total params: 156 (624.00 B)
```

```
Trainable params: 156 (624.00 B)
```

```
Non-trainable params: 0 (0.00 B)
```

```
Model: "functional_1"
```

Layer (type)	Output Shape	Param #
input_layer_1 (InputLayer)	(None, 2)	0
dense_3 (Dense)	(None, 6)	18
dense_4 (Dense)	(None, 8)	56
dense_5 (Dense)	(None, 10)	90

```
Total params: 164 (656.00 B)
```

```
Trainable params: 164 (656.00 B)
```

```
Non-trainable params: 0 (0.00 B)
```

```
Model: "functional_2"
```

Layer (type)	Output Shape	Param #
input_layer_2 (InputLayer)	(None, 10)	0
functional (Functional)	(None, 2)	156

```
│ functional_1 (Functional)          │ (None, 10)             │           164 │
└────────────────────────────────────┴────────────────────────┴───────────────┘
```

```
 Total params: 320 (1.25 KB)
```

```
 Trainable params: 320 (1.25 KB)
```

```
 Non-trainable params: 0 (0.00 B)
```

```
autoencoder.summary()
```

```
Model: "functional_2"
```

```
┏━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━┳━━━━━━━━━━━━━━━━━━━━━━━━┳━━━━━━━━━━━━━━━┓
┃ Layer (type)                       ┃ Output Shape           ┃       Param # ┃
┡━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━╇━━━━━━━━━━━━━━━━━━━━━━━━╇━━━━━━━━━━━━━━━┩
│ input_layer_2 (InputLayer)         │ (None, 10)             │             0 │
├────────────────────────────────────┼────────────────────────┼───────────────┤
│ functional (Functional)            │ (None, 2)              │           156 │
├────────────────────────────────────┼────────────────────────┼───────────────┤
│ functional_1 (Functional)          │ (None, 10)             │           164 │
└────────────────────────────────────┴────────────────────────┴───────────────┘
```

```
 Total params: 320 (1.25 KB)
```

```
 Trainable params: 320 (1.25 KB)
```

```
 Non-trainable params: 0 (0.00 B)
```

```
autoencoder.compile(loss='mse',
                    optimizer=SGD(learning_rate=0.01))
```

```
autoencoder.fit(scaled_data,
                scaled_data,
                batch_size = 64,
                shuffle = True,
                epochs = 50)
```

```
Epoch 1/50
469/469 ━━━━━━━━━━━━━━━━━━━━ 0s 391us/step - loss: 0.0658
Epoch 2/50
469/469 ━━━━━━━━━━━━━━━━━━━━ 0s 793us/step - loss: 0.0600
Epoch 3/50
469/469 ━━━━━━━━━━━━━━━━━━━━ 0s 381us/step - loss: 0.0549
Epoch 4/50
469/469 ━━━━━━━━━━━━━━━━━━━━ 0s 355us/step - loss: 0.0507
Epoch 5/50
469/469 ━━━━━━━━━━━━━━━━━━━━ 0s 374us/step - loss: 0.0478
Epoch 6/50
469/469 ━━━━━━━━━━━━━━━━━━━━ 0s 372us/step - loss: 0.0453
Epoch 7/50
469/469 ━━━━━━━━━━━━━━━━━━━━ 0s 336us/step - loss: 0.0436
Epoch 8/50
469/469 ━━━━━━━━━━━━━━━━━━━━ 0s 340us/step - loss: 0.0423
Epoch 9/50
469/469 ━━━━━━━━━━━━━━━━━━━━ 0s 356us/step - loss: 0.0414
Epoch 10/50
469/469 ━━━━━━━━━━━━━━━━━━━━ 0s 356us/step - loss: 0.0409
Epoch 11/50
469/469 ━━━━━━━━━━━━━━━━━━━━ 0s 354us/step - loss: 0.0404
Epoch 12/50
469/469 ━━━━━━━━━━━━━━━━━━━━ 0s 348us/step - loss: 0.0400
Epoch 13/50
469/469 ━━━━━━━━━━━━━━━━━━━━ 0s 352us/step - loss: 0.0399
Epoch 14/50
469/469 ━━━━━━━━━━━━━━━━━━━━ 0s 350us/step - loss: 0.0398
Epoch 15/50
469/469 ━━━━━━━━━━━━━━━━━━━━ 0s 348us/step - loss: 0.0398
Epoch 16/50
469/469 ━━━━━━━━━━━━━━━━━━━━ 0s 349us/step - loss: 0.0397
Epoch 17/50
469/469 ━━━━━━━━━━━━━━━━━━━━ 0s 429us/step - loss: 0.0397
Epoch 18/50
469/469 ━━━━━━━━━━━━━━━━━━━━ 0s 380us/step - loss: 0.0396
Epoch 19/50
469/469 ━━━━━━━━━━━━━━━━━━━━ 0s 501us/step - loss: 0.0397
Epoch 20/50
469/469 ━━━━━━━━━━━━━━━━━━━━ 0s 369us/step - loss: 0.0395
Epoch 21/50
469/469 ━━━━━━━━━━━━━━━━━━━━ 0s 388us/step - loss: 0.0396
Epoch 22/50
469/469 ━━━━━━━━━━━━━━━━━━━━ 0s 367us/step - loss: 0.0396
Epoch 23/50
469/469 ━━━━━━━━━━━━━━━━━━━━ 0s 361us/step - loss: 0.0396
Epoch 24/50
469/469 ━━━━━━━━━━━━━━━━━━━━ 0s 345us/step - loss: 0.0395
```

```
Epoch 25/50
469/469 ━━━━━━━━━━━━━━━━━━━━ 0s 344us/step - loss: 0.0396
Epoch 26/50
469/469 ━━━━━━━━━━━━━━━━━━━━ 0s 383us/step - loss: 0.0396
Epoch 27/50
469/469 ━━━━━━━━━━━━━━━━━━━━ 0s 334us/step - loss: 0.0397
Epoch 28/50
469/469 ━━━━━━━━━━━━━━━━━━━━ 0s 342us/step - loss: 0.0397
Epoch 29/50
469/469 ━━━━━━━━━━━━━━━━━━━━ 0s 340us/step - loss: 0.0396
Epoch 30/50
469/469 ━━━━━━━━━━━━━━━━━━━━ 0s 341us/step - loss: 0.0394
Epoch 31/50
469/469 ━━━━━━━━━━━━━━━━━━━━ 0s 339us/step - loss: 0.0396
Epoch 32/50
469/469 ━━━━━━━━━━━━━━━━━━━━ 0s 348us/step - loss: 0.0396
Epoch 33/50
469/469 ━━━━━━━━━━━━━━━━━━━━ 0s 348us/step - loss: 0.0396
Epoch 34/50
469/469 ━━━━━━━━━━━━━━━━━━━━ 0s 339us/step - loss: 0.0396
Epoch 35/50
469/469 ━━━━━━━━━━━━━━━━━━━━ 0s 340us/step - loss: 0.0396
Epoch 36/50
469/469 ━━━━━━━━━━━━━━━━━━━━ 0s 342us/step - loss: 0.0396
Epoch 37/50
469/469 ━━━━━━━━━━━━━━━━━━━━ 0s 641us/step - loss: 0.0395
Epoch 38/50
469/469 ━━━━━━━━━━━━━━━━━━━━ 0s 342us/step - loss: 0.0395
Epoch 39/50
469/469 ━━━━━━━━━━━━━━━━━━━━ 0s 341us/step - loss: 0.0395
Epoch 40/50
469/469 ━━━━━━━━━━━━━━━━━━━━ 0s 345us/step - loss: 0.0396
Epoch 41/50
469/469 ━━━━━━━━━━━━━━━━━━━━ 0s 344us/step - loss: 0.0395
Epoch 42/50
469/469 ━━━━━━━━━━━━━━━━━━━━ 0s 347us/step - loss: 0.0397
Epoch 43/50
469/469 ━━━━━━━━━━━━━━━━━━━━ 0s 374us/step - loss: 0.0396
Epoch 44/50
469/469 ━━━━━━━━━━━━━━━━━━━━ 0s 343us/step - loss: 0.0395
Epoch 45/50
469/469 ━━━━━━━━━━━━━━━━━━━━ 0s 340us/step - loss: 0.0396
Epoch 46/50
469/469 ━━━━━━━━━━━━━━━━━━━━ 0s 341us/step - loss: 0.0396
Epoch 47/50
469/469 ━━━━━━━━━━━━━━━━━━━━ 0s 337us/step - loss: 0.0395
Epoch 48/50
469/469 ━━━━━━━━━━━━━━━━━━━━ 0s 345us/step - loss: 0.0395
```

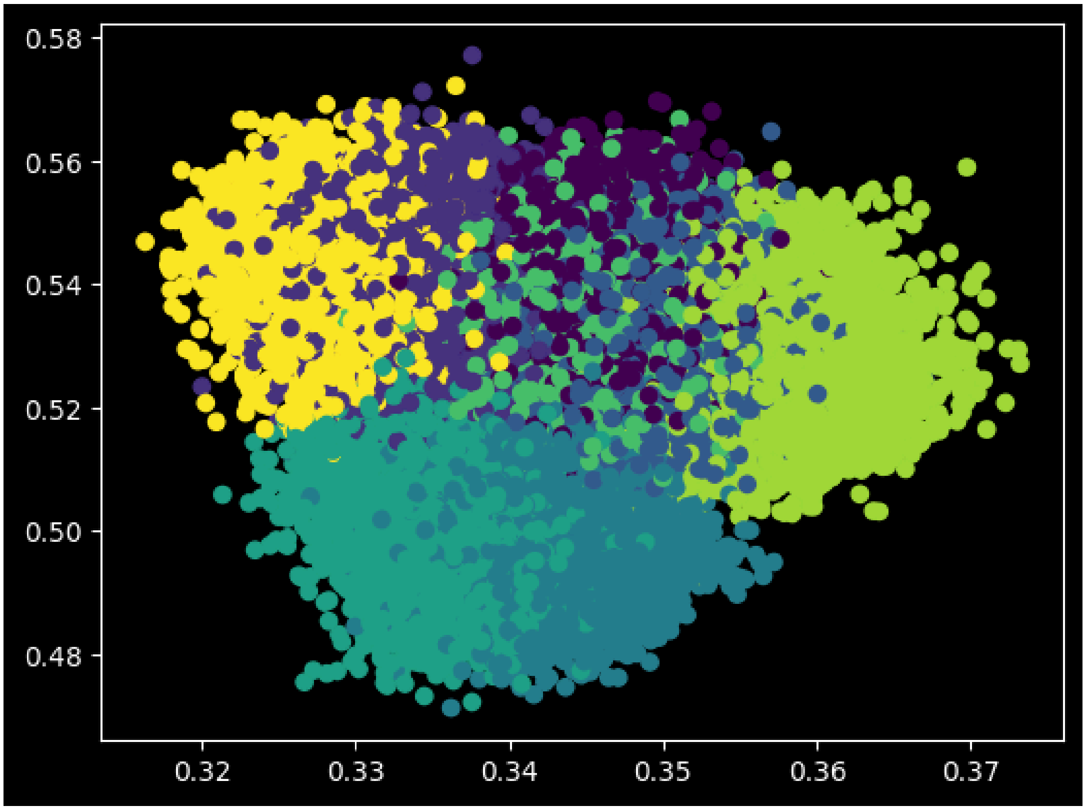
```
Epoch 49/50
469/469 ━━━━━━━━━━━━━━━━━━━━ 0s 342us/step - loss: 0.0396
Epoch 50/50
469/469 ━━━━━━━━━━━━━━━━━━━━ 0s 348us/step - loss: 0.0395
```

```
<keras.src.callbacks.history.History at 0x30913c2c0>
```

```
# Encode our data down to two dimensions using our
# trained autoencoder

encoder = Model(inputs=input_layer, outputs=encoded)
encoded_2dim = encoder.predict(scaled_data)
```

```
938/938 ━━━━━━━━━━━━━━━━━━━━ 1s 517us/step
```

```
plt.scatter(encoded_2dim[:,0],encoded_2dim[:,1], c = y)
```

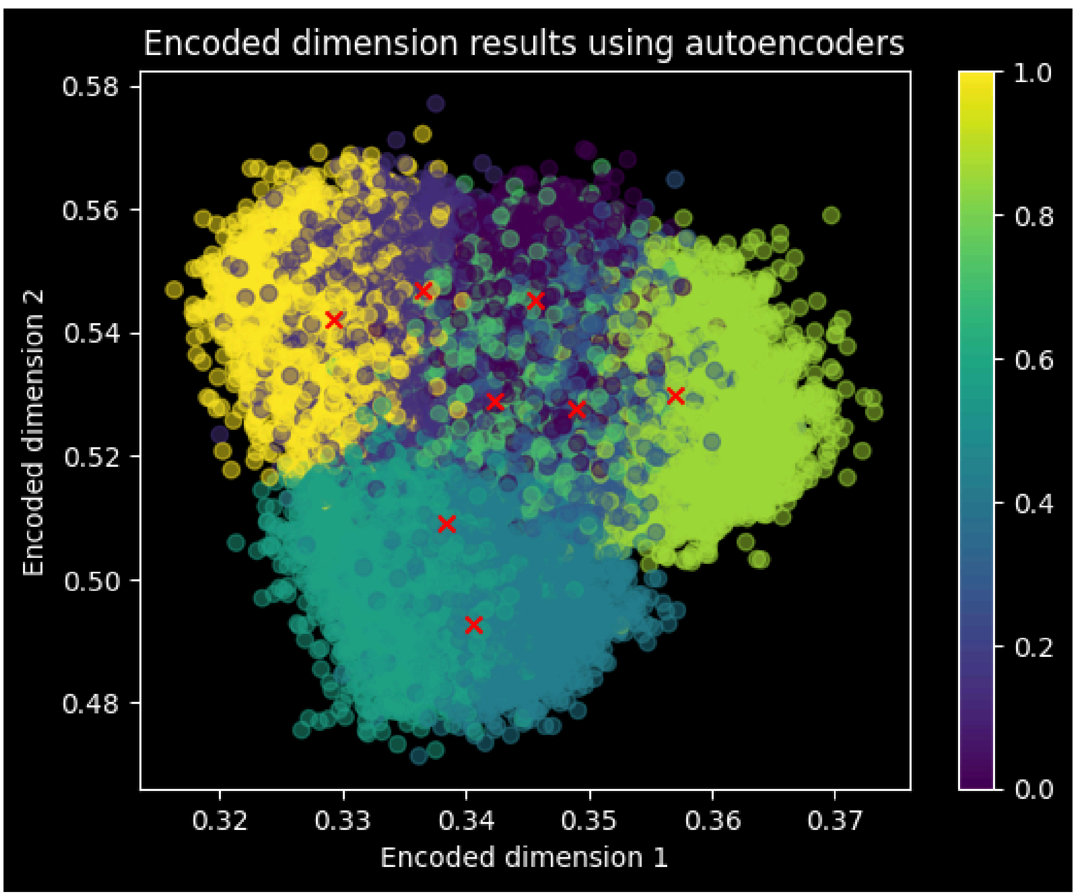


Visually we see that our autoencoder has performed much better than the PCA. We see four clusters very clearly. Not only that the clusters are much more compact than with PCA.

```
# Train our k-means on the encoded 2 dimensional data

kmeans_for_autoencoder = KMeans(n_clusters=8)
kmeans_for_autoencoder.fit(encoded_2dim)
```

```
KMeans()
```

```
plt.scatter(encoded_2dim[:, 0], encoded_2dim[:, 1], c=y, alpha=0.5)
plt.scatter(kmeans_for_autoencoder.cluster_centers_[:,
0],kmeans_for_autoencoder.cluster_centers_[:,1],marker='x',c='red')
plt.title('Encoded dimension results using autoencoders')
plt.colorbar()
plt.xlabel("Encoded dimension 1")
plt.ylabel("Encoded dimension 2")
plt.show()
```




```
kmeans_for_autoencoder.inertia_
```

```
6.157865047454834
```

```
kmeans_for_pca.inertia_
```

```
665.493846052085
```

```
kmeans_for_pca.labels_
```

```
array([6, 2, 3, ..., 6, 1, 2], dtype=int32)
```

```
from sklearn.metrics import silhouette_score
```

```
silhouette_score(scaled_data,kmeans_for_pca.labels_)
```

```
0.13132647285236687
```

```
silhouette_score(scaled_data, kmeans_for_autoencoder.labels_)
```

```
0.19285495491413215
```

Saving our models

Finally we will save our model for future use

```
autoencoder.save("autoencoder.keras")
```

```
import pickle

with open('pca_model.pkl', 'wb') as file:
    pickle.dump(pca, file)
```