Clustering analysis of skin lesion images

1. Introduction

The 2019 ISIC Skin Lesion Images Classification dataset was chosen for this study. It is essentially an image classification problem, with the data consisting of 25331 labeled images from 8 different diagnostic categories, namely Melanoma, Melanocytic nevus, Basal cell carcinoma, Actinic keratosis, Benign keratosis, Dermatofibroma, Vascular lesion, and Squamous cell carcinoma. The total data size is approximately 9.2 GB. The images depict skin lesions, which are any areas of skin that differ from the surrounding skin in color, shape, size, and texture.

Figure 1: *Sample Image of labeled skin lesion data*

The ground truth histogram shown below demonstrates the data imbalance in which a few categories dominate over others. We observe that class "NV" makes up half of the images while classes DF, VASC, and SCC do not even constitute 1000 images.

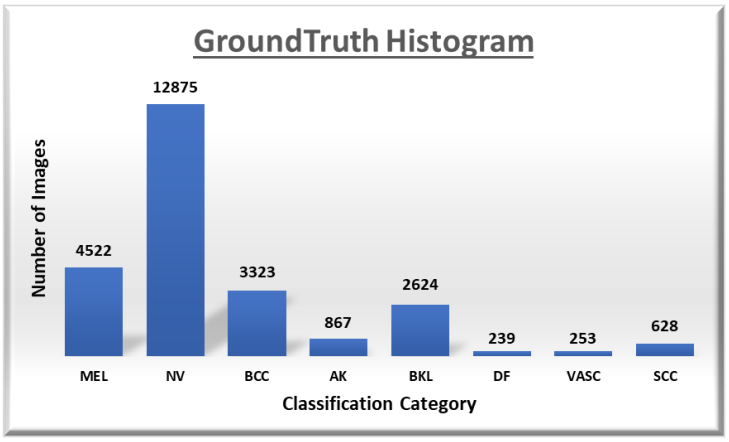


Figure 2: *Ground truth histogram of classification categories*

Although we considered a labeled data set, the goal was to perform unsupervised techniques to find clusters in the data. The label information is then used to evaluate the clustering performance. It is worth noting that the huge data imbalance observed in the images can affect clustering algorithm performances. Before implementing any clustering algorithms, we performed dimension reduction using a convolutional autoencoder.

2. Image Transformation

We had to transform our image data into a tensor form with features before the features could be fed into the convolutional autoencoder for dimension reduction. The process we used to extract features from an image is summarized below.

Figure 3: *Image transformation procedure*

The first step was to import the "skimage" module into python, which allows for easy import of image data into the Python environment. Once the module was imported, we fixed the image dimensions to 36x48x3. The pixel size represents its length, and width, and 3 represent the R, G, and B color channels. The image is then read from the dataset, resized to our specified dimensions, and saved as a third-order tensor (i.e. 3D array).

We worked with the R library EBImage to read image data into the R environment for project 2. We observed that using the python module "skimage" to import images into python is not only faster but also much easier than that of EBImage package.

3. Exploratory data analysis and dimension reduction

The original raw data downloaded from Kaggle consisted of 25,331 colored images of varying sizes. However, we observed that an aspect ratio of 4:3 was maintained in all images. Hence, all images were resized to a size that respects the aspect ratio. Of the 25,331 images, 75% of the data i.e. 18998 images were randomly selected to be used as the training set while the rest of the 6333 images were used as validation set images. Each image is represented as a 36x48x3 vector of pixel intensities. From Table 1, it is clear that there is a heavy **imbalance in the data** and that it is bound to affect clustering accuracy.

Table 1: Number of images from each class label in the training set.

Class	DF	VASC	SCC	AK	BKL	BCC	MEL	NV
Proportion	0.0095	0.0097	0.0246	0.0350	0.1023	0.1310	0.1798	0.5080

A histogram of average red (R), blue (B), and green (G) pixel intensities corresponding to images of each class are shown in Figure 4. We observe that the intensity histograms corresponding to blue and green are all left-skewed with maximum intensities close to 0.6. On the other hand, the red intensity histogram appears to be bimodal with the maximum intensity being close to 1. If we view image data to be sampled from the corresponding histograms in Figure 4, then the distributions from which different class images are being sampled look very similar.

Since each image corresponds to a feature vector of size 36x48x3 = 5184, the next logical step is to perform dimension reduction. Our goal is to perform non-linear dimension reduction using an autoencoder. However, to get an idea of how "large" our encoded images should be, we first applied Principal Component Analysis to vectorized image data. Scree plots in Figure 5, show that 15 principal components explain 90% of the variation in the data

while 90 principal components are required to capture 96% of the variation in the data. Keeping the computational costs in mind, we believe that retaining 15 principal components is enough. Furthermore, retaining 15 principal components is backed by Kaiser's rule as well i.e. there are exactly 15 eigenvalues greater than or equal to 1.

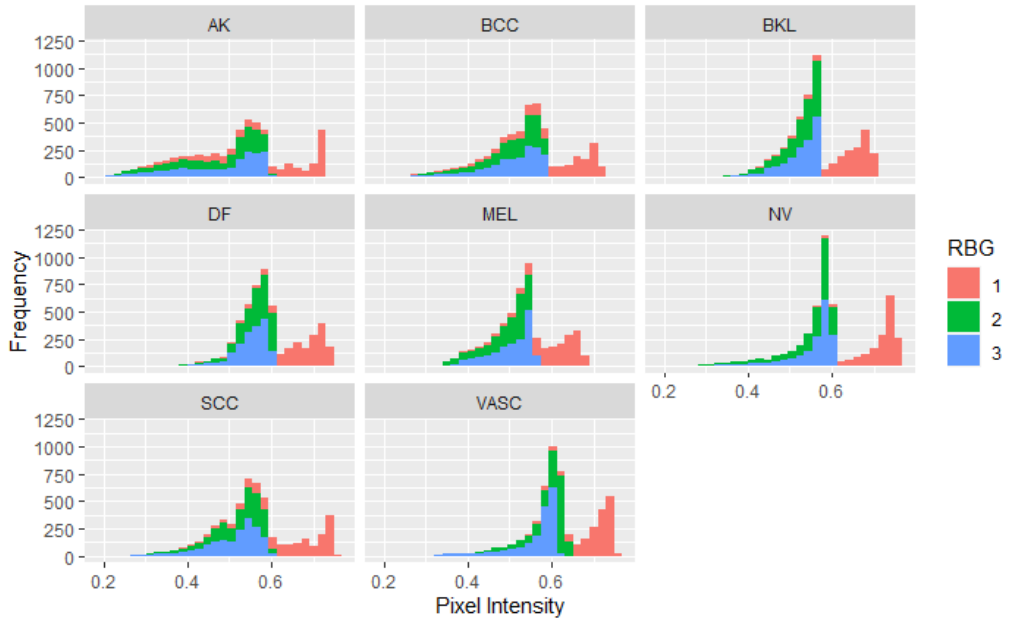
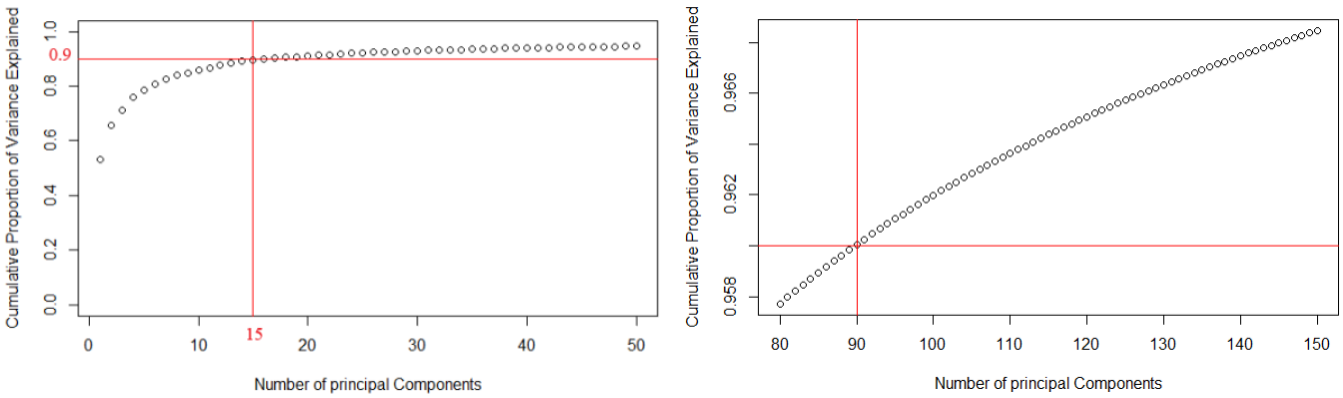


Figure 4: Histogram of RBG pixel intensities categorized by type of skin lesion.

Using these 15 principal components, we obtain a reduced feature vector of length five corresponding to each image. Note that the data was not scaled prior to dimension reduction since all features are pixel intensities measured on the same scale. Based on this analysis, we concluded that the encoded image must be at least 15-dimensional.

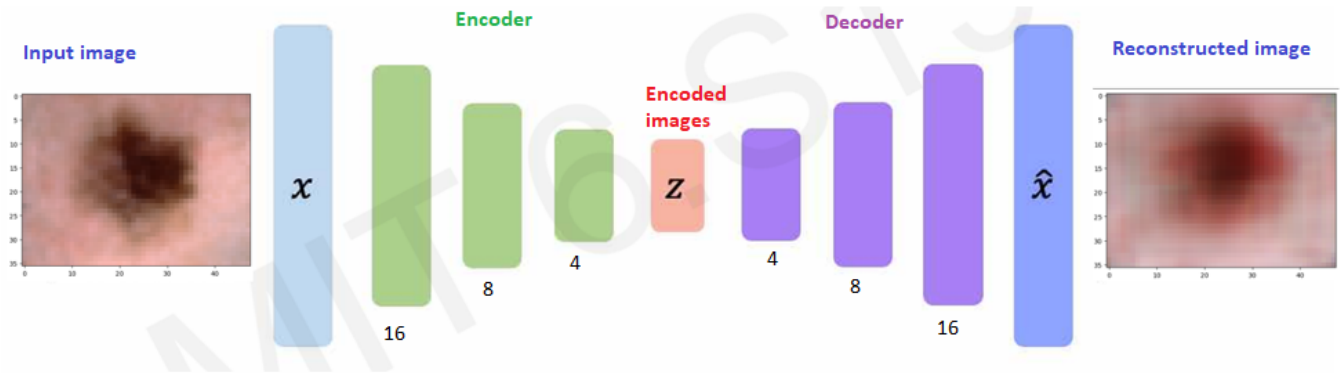
Figure 5: Scree plots to identify the required number of PCs.



4. Dimension reduction

Since we are dealing with image data, we chose to implement a convolutional autoencoder i.e. an autoencoder with convolutional layers like that of CNN. A pictorial representation of our autoencoder is given in Figure 5. The input image, denoted by "**x**", is of dimension 36x48x3. The reconstructed image "**x-hat**" is also an image of dimensions 36x48x3. The encoded image (reduced features), denoted by **z,** is of size 3x4x4. The choice of this dimension for the encoded image has two reasons. First, as mentioned in the previous section, the number of reduced features is more than 15. Additionally, the reduced features can be interpreted as four images of size 3x4 stacked one behind the other. Notice that the encoded image also respects the 4:3 aspect ratio.

Figure 5: Convolutional Autoencoder



4.1. Encoder Architecture

Pictorially, the three green layers in Figure 5 represent the three layers of our encoder. Each green layer represents one convolutional layer followed by a max-pooling layer. The three convolutional layers were padded such that the output of the convolutional layer has the same dimension as that of the input. The number of filters used in each of the three layers is 16, 8, and 4 respectively. The convolution kernels were matrices of size 3x3 while the max-pooling layers used filters of size 2x2 (i.e. downsampling by a factor of 2).

4.2 Decoder Architecture

The purple layers in Figure 5 denote the layers associated with the decoder. Each layer constitutes a convolutional layer followed by an upsampling procedure. Since we want to reconstruct the original image, the decoder architecture should be an "inverse" of the encoder architecture. As is evident from Figure 5, the number of filters used in each convolutional layer is inverted. Convolutional layers are padded like in the Encoder. A convolutional kernel of size 3x3 is again used followed by upsampling the rows and columns by a factor of 2. Upsampling is performed by repeating each row and each column twice.

Note that the choice of kernel dimension and up/downsampling factor is based on the parameter values that are used most often.

5. Autoencoder Training

Of the 25,331 images, we used 18,998 images to train the autoencoder for 10 epochs using mini-batch stochastic gradient descent with batch size 512. In particular, we used the Adam optimizer to minimize binary cross-entropy loss between input and reconstructed images. The loss function value was monitored on the validation set consisting of 6333 images. The loss converged to 0.58 approximately after 10 epochs. The total number of parameters involved in the autoencoder is 3947.

In the sections that follow, K-means and Gaussian mixture model techniques were implemented on the 3x4x4 dimensional encoded images instead of the entire images. A point to note is that these reduced features are **correlated.** Unlike PCA, autoencoders, do not output uncorrelated features.

6. K-Means Algorithm

Kmeans algorithm is an iterative algorithm that tries to partition the dataset into Kpre-defined distinct non-overlapping subgroups (clusters) where each data point belongs to only one group. It tries to make the intra-cluster data points as similar as possible while also keeping the clusters as different (far) as possible. It assigns data points to a cluster such that the sum of the squared distance between the data points and the cluster's centroid (arithmetic mean of all the data points that belong to that cluster) is at the minimum. The less variation we have within clusters, the more homogeneous (similar) the data points are within the same cluster.

The Adjusted Rand Index (ARI) results for different random initializations are shown in the table below.

Random State	ARI
0	0.0326
1	0.0325
2	0.0332
3	0.0331
4	0.0334

It is evident form the table that the ARI value does not change significantly with different random initializations. Further, the ARI values for all of the random initializations are close to 0 indicating that the predicted labels do not match well with the true labels..

The average ARI values for different number of clusters (K) are shown below

Number of Clusters	ARI
4	0.058
5	0.057
6	0.052
7	0.027
8	0.033
9	0.027

Again, the ARI value does not change significantly with the number of cluster. However, ARI is maximum for K=4, this is probably due to the imbalance in the dataset. With lesser number of clusters, the data belonging to classes having fewer number of images are combined with the dominant classes, improving the overall ARI.

7. Gaussian Mixture Model (GMM)

Gaussian mixture models are probabilistic models that are used to represent normally distributed subpopulations within a larger population. In general, mixture models do not require knowing which subpopulation a data point belongs to, allowing the model to learn the subpopulations on its own. Because subpopulation assignment is unknown, this is a form of unsupervised learning. GMM is an expectation maximisation algorithm that is similar to the K-means algorithm. The gaussian mixture has a more flexible decision boundary than the K-means algorithm. Furthermore, GMM employs a probabilistic algorithm, which means that we know the degree of certainty behind the classification of a specific data point to a specific cluster.

The encoded data is passed through GMM algorithm and following ARI values are obtained for various cluster values.

Number of Clusters	ARI
4	0.0861
5	0.1199
6	0.0797
7	0.0634
8	0.0541
9	0.0497

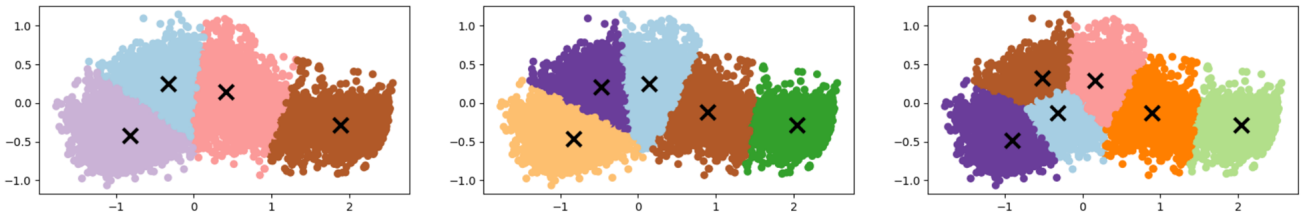
From the above ARI values it is evident that when compared to the k-means algorithm, the GMM performance is undeniably better. The ARI varies with cluster size, reaching a maximum at cluster size 5. Furthermore, for any cluster size, the ARI value still remains close to zero, indicating that the predicted labels do not match well with the True labels.

8. Principal Component Analysis (PCA)

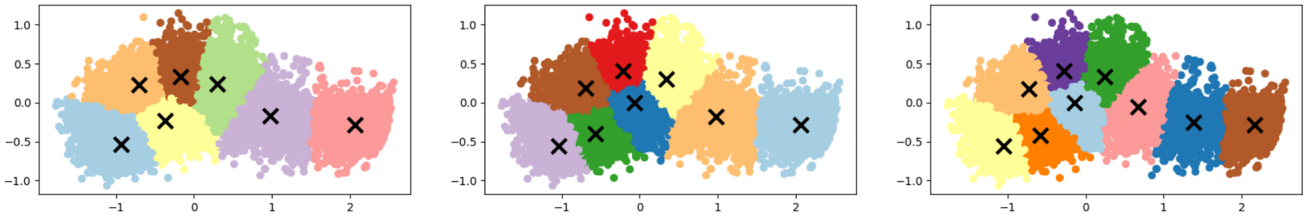
PCA was applied to 3*4*4 encoded data outputted by autoencoders to reduce the data to 2D.

K-means algorithm implemented for different number of clusters on the 2D data yielded the results as shown in the figure below.

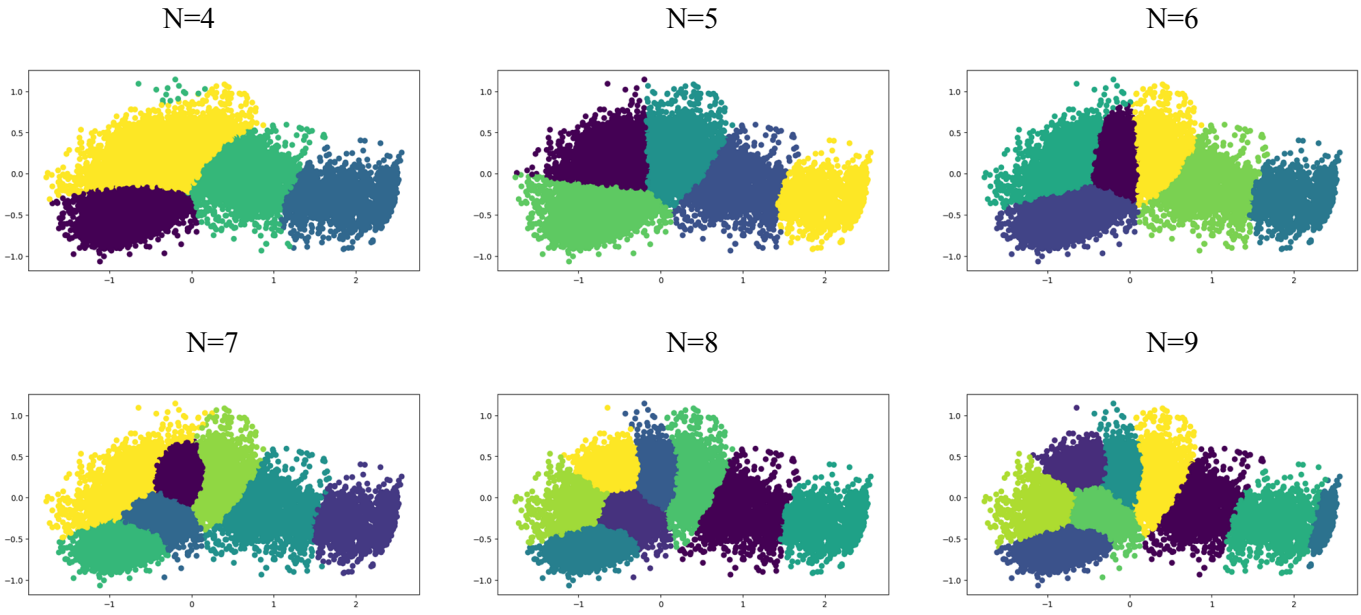
N=4 N=5 N=6



N=7 N=8 N=9



GMM algorithm implemented for different number of clusters on the 2D data yielded the results as shown in the figure below.



9. Scope for further improvement

First, the most obvious improvement in clustering performances can be obtained by using images of better resolution rather than the current resolution of 36x48x3. The architecture of the autoencoder used is very basic. Fine-tuning the autoencoder by increasing the number of convolutional layers, increasing the number of filters used per layer, changing the filter size, and up/downsampling factors can improve the autoencoder's performance further.

Another direction of improvement is to apply a log transform to the pixel data before applying the GMM algorithm. Since the pixel intensities are all right-skewed (Figure 4). This will further improve the performance of GMM as a log transform will make the distribution more symmetric, as assumed in the GMM model.

10 . References

[1] Tschandl P., Rosendahl C. & Kittler H. The HAM10000 dataset, a large collection of multi-source dermatoscopic images of common pigmented skin lesions. Sci. Data 5, 180161 doi.10.1038/sdata.2018.161 (2018)

[2] Noel C. F. Codella, David Gutman, M. Emre Celebi, Brian Helba, Michael A. Marchetti, Stephen W. Dusza, Aadi Kalloo, Konstantinos Liopyris, Nabin Mishra, Harald Kittler, Allan Halpern: "Skin Lesion Analysis Toward Melanoma Detection: A Challenge at the 2017 International Symposium on Biomedical Imaging (ISBI), Hosted by the International Skin Imaging Collaboration (ISIC)", 2017; arXiv:1710.05006.

[3] Marc Combalia, Noel C. F. Codella, Veronica Rotemberg, Brian Helba, Veronica Vilaplana, Ofer Reiter, Allan C. Halpern, Susana Puig, Josep Malvehy: "BCN20000: Dermoscopic Lesions in the Wild", 2019; arXiv:1908.02288.

[4] Blogpost given by the official keras website - click here

Contributions (First names mentioned alphabetically)

1. Aditya: Applied K-means algorithm to the encoded data from the autoencoder to identify the clusters and evaluated it's performance for different random initialization and cluster sizes. Applied Principal Companent Analysis(PCA), to the encoded data to convert it to 2D and applied K-means to visually evaluate the clusters.
2. Padma: The python/R implementation and report writing correspond to sections 1 (Section 1 done together with Sudhir), 3, 4, 5, and 9. (EDA, Implementation of autoencoder)
3. Sudhir: Unsupervised clustering was performed using the Gaussian Mixture Model algorithm on encoded data. The algorithm's performance was evaluated using the ARI index, and the variation in ARI with cluster size was investigated. PCA was used to reduce the data dimension to two, and the GMM was repeated and plotted for better visualisation and understanding.

In [ ]:
```
import numpy as np
import pandas as pd

from keras import models
from keras import layers
from keras import optimizers
from keras import Input, layers  # for 7.1.1
from keras.models import Sequential, Model  # for 7.1.1

import rpy2.robjects as robjects # used to import .Rdata into python

import cmath
import matplotlib.pyplot as plt
```

Reading images into Python environment

In [ ]:
```
robjects.r['load']('features.RData')

features = robjects.r['features']
features = np.array(features)

df = pd.DataFrame(data=features[0:,0:],
           index=[i for i in range(features.shape[0])],
           columns=['x'+str(i) for i in range(features.shape[1])])


from skimage.io import imread_collection
Data_dir = "Data/*/*.jpg"
imdata = imread_collection(Data_dir)

imdata[0] #  to access images 0,1,....25330
type(imdata)
type(imdata[0])
imdata[0].shape
```

```
KeyboardInterrupt
```

Store images as tensors

In [ ]:
```
im_array = np.empty([25331,36,48,3])
im_array[0,:,:,:].shape

from skimage.transform import resize

for i in range(25331):
    img_resized = resize(imdata[i], output_shape= (36,48,3))
    im_array[i,:,:,:] = img_resized
    if i%1000 == 0:
        print(i)
```

In [ ]:
```
np.save("im_array.npy", im_array)
```

Split into training and validation sets

In [ ]:
```
labels = ['AK'] * 867 + ['BCC'] * 3323 + ['BKL']*2624 + ['DF']*239 + ['MEL']*4522 + [
len(labels)

from sklearn.model_selection import train_test_split

train, test, label_train, label_test  = train_test_split(im_array, labels, test_size=0

np.save("label_train.npy", label_train)
np.save("label_test.npy", label_test)
```

Encoder Architecture

In [ ]:
```
input_img = Input(shape=(36, 48, 3))


x = layers.Conv2D(16, (3, 3), activation='relu', padding='same')(input_img) # 36x48x16
x = layers.MaxPooling2D((2, 2), padding='same')(x)
x = layers.Conv2D(8, (3, 3), activation='relu', padding='same')(x)
x = layers.MaxPooling2D((2, 2), padding='same')(x)
x = layers.Conv2D(4, (3, 3), activation='relu', padding='same')(x)
encoded = layers.MaxPooling2D((3, 3), padding='valid')(x)
```

In [ ]:
```
encoded
```

Decoder Architecture

In [ ]:
```
x = layers.Conv2D(4, (3, 3), activation='relu', padding='same')(encoded)
x = layers.UpSampling2D((3, 3))(x)
x = layers.Conv2D(8, (3, 3), activation='relu', padding='same')(x)
x = layers.UpSampling2D((2, 2))(x)
x = layers.Conv2D(16, (3, 3), activation='relu', padding='same')(x)
x = layers.UpSampling2D((2, 2))(x)

decoded = layers.Conv2D(3, (3, 3), activation='sigmoid', padding='same')(x)
```

In [ ]:
```
decoded
```

Define model, loss function, optimzation algorithm

In [ ]:
```
autoencoder = Model(input_img, decoded)
autoencoder.compile(optimizer='adam', loss='binary_crossentropy')
autoencoder.summary()
```

Fit the model to the training data

In [ ]:
```
autoencoder.fit(train, train, epochs=10, batch_size=512, shuffle=True, validation_data
```

Predict using the fitted model

In [ ]:
```
decoded_imgs = autoencoder.predict(test)
```

In [ ]:
```
plt.imshow(test[2]) # original image
```

In [ ]:
```
plt.imshow(decoded_imgs[2]) # reconstructed image
```

Encode the training images

In [ ]:
```
encoder = Model(input_img, encoded)
encoded_train = encoder(train)

encoded_train.shape # encoded features
```

In [ ]:
```
encoded_train_array = np.asarray(encoded_train)
encoded_train_array.shape
np.save("encoded_train_new.npy", encoded_train_array )
```

Encode validation set as well

In [ ]:
```
encoded_test = encoder(test)
encoded_test_array = np.asarray(encoded_test)
encoded_test_array.shape

np.save("encoded_test_new.npy", encoded_test_array )
```

P3

December 9, 2022

[75]:
```
import sys
import numpy as np
import random
import matplotlib.pyplot as plt
from sklearn.cluster import KMeans
from sklearn.decomposition import PCA
from sklearn import metrics
np.set_printoptions(threshold=sys.maxsize)
```

[79]:
```
Train_Data=np.load('encoded_train_new.npy')
Test_Data=np.load('encoded_test_new.npy')
Train_Labels=np.load('label_train.npy')
Test_Labels=np.load('label_test.npy')
C = ['AK', 'BCC', 'BKL', 'DF', 'MEL', 'NV', 'SCC', 'VASC']
def CLS(n):
    if n=='AK':
        return 1
    elif n=='BCC':
        return 2
    elif n=='BKL':
        return 2
    elif n=='DF':
        return 3
    elif n=='MEL':
        return 4
    elif n=='NV':
        return 5
    elif n=='SCC':
        return 6
    elif n=='VASC':
        return 7
Train_Labels_N = np.array(list(map(CLS,Train_Labels)))
```

[80]:
```
print(Train_Labels_N.shape)
print(Train_Data.shape[0])
print(Test_Data.shape)
print(Train_Labels.shape)
print(Test_Labels.shape)
```

```
(18998,)
18998
(6333, 3, 4, 4)
(18998,)
(6333,)
```

[8]:
```
#print(Train_Data[0])
#print(Train_Data[-1])
```

[9]:
```
Train_vector=np.reshape(Train_Data,(Train_Data.shape[0],Train_Data.
 ↪shape[1]*Train_Data.shape[2]*Train_Data.shape[3]))
Test_vector=np.reshape(Test_Data,(Test_Data.shape[0],Test_Data.
 ↪shape[1]*Test_Data.shape[2]*Test_Data.shape[3]))
```

[10]:
```
print(Train_vector.shape)
print(Test_vector.shape)
```

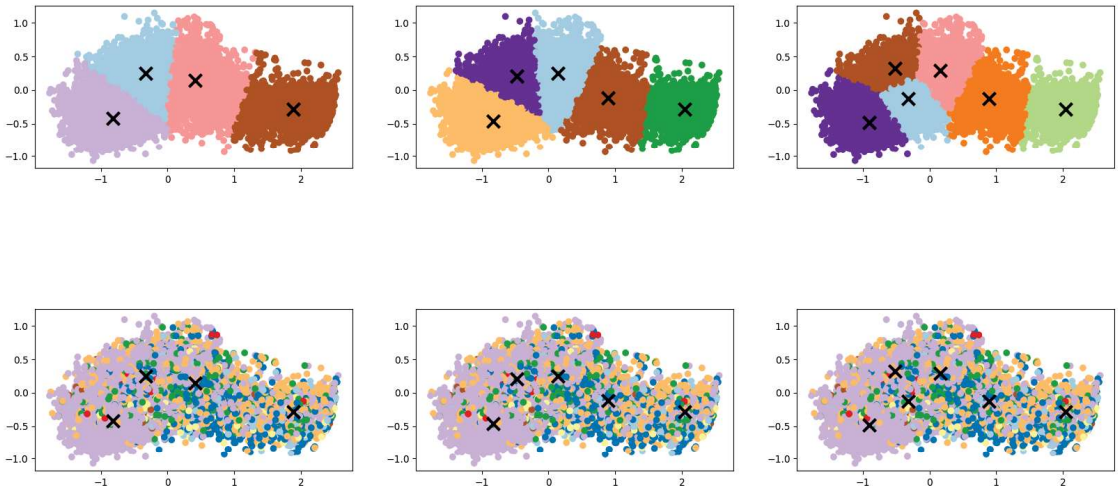
```
(18998, 48)
(6333, 48)
```

[11]:
```
#print(Train_vector[0])
#print(Train_vector[-1])
```

[24]:
```
pca = PCA(n_components=2)
principalComponents = pca.fit_transform(Train_vector)
np.shape(principalComponents)
N_C=[4,5,6,7,8,9]
```

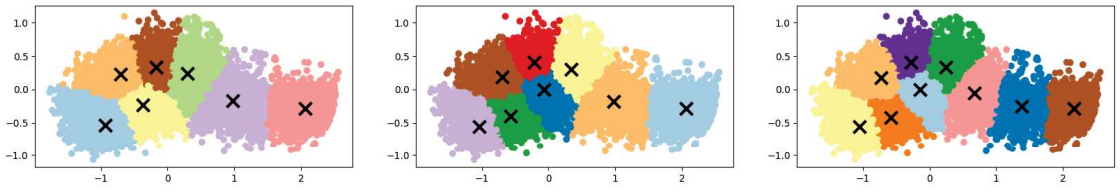
[67]:
```
fig, axs = plt.subplots(1, 3)
for i in range(3):
    KM_PCA = KMeans(n_clusters=N_C[i], random_state=0,max_iter=1000).
 ↪fit(principalComponents)
    Predicted_Train_PCA=KM_PCA.labels_.reshape(18998,1)
    centroids=KM_PCA.cluster_centers_
    axs[i].scatter(principalComponents[:,0],principalComponents[:
 ↪,1],c=Predicted_Train_PCA, cmap=plt.cm.Paired)

    axs[i].scatter(
    centroids[:, 0],
    centroids[:, 1],
    marker="x",
    s=169,
    linewidths=3,
    color="k",
    zorder=10)
    # square plot
    axs[i].set_aspect('equal', adjustable='box')
    fig.set_figwidth(20)
```

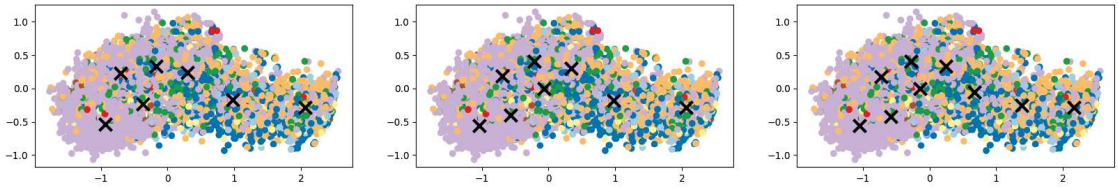
```
    fig.set_figheight(20)

fig, axs = plt.subplots(1, 3)
for i in range(3):
    KM_PCA = KMeans(n_clusters=N_C[i], random_state=0,max_iter=1000).
 ↪fit(principalComponents)
    Predicted_Train_PCA=KM_PCA.labels_.reshape(18998,1)
    centroids=KM_PCA.cluster_centers_
    axs[i].scatter(principalComponents[:,0],principalComponents[:
 ↪,1],c=Train_Labels_N, cmap=plt.cm.Paired)

    axs[i].scatter(
    centroids[:, 0],
    centroids[:, 1],
    marker="x",
    s=169,
    linewidths=3,
    color="k",
    zorder=10)
    # square plot
    axs[i].set_aspect('equal', adjustable='box')
    fig.set_figwidth(20)
    fig.set_figheight(20)
```



[72]:
```
fig, axs = plt.subplots(1, 3)
for i in range(3):
    KM_PCA = KMeans(n_clusters=N_C[i+3], random_state=0,max_iter=1000).
 ↪fit(principalComponents)
    Predicted_Train_PCA=KM_PCA.labels_.reshape(18998,1)
    centroids=KM_PCA.cluster_centers_
```

```
    axs[i].scatter(principalComponents[:,0],principalComponents[:
 ↪,1],c=Predicted_Train_PCA, cmap=plt.cm.Paired)

    axs[i].scatter(
    centroids[:, 0],
    centroids[:, 1],
    marker="x",
    s=169,
    linewidths=3,
    color="k",
    zorder=10)
    # square plot
    axs[i].set_aspect('equal', adjustable='box')
    fig.set_figwidth(20)
    fig.set_figheight(20)

fig, axs = plt.subplots(1, 3)
for i in range(3):
    KM_PCA = KMeans(n_clusters=N_C[i+3], random_state=0,max_iter=1000).
 ↪fit(principalComponents)
    Predicted_Train_PCA=KM_PCA.labels_.reshape(18998,1)
    centroids=KM_PCA.cluster_centers_
    axs[i].scatter(principalComponents[:,0],principalComponents[:
 ↪,1],c=Train_Labels_N, cmap=plt.cm.Paired)

    axs[i].scatter(
    centroids[:, 0],
    centroids[:, 1],
    marker="x",
    s=169,
    linewidths=3,
    color="k",
    zorder=10)
    # square plot
    axs[i].set_aspect('equal', adjustable='box')
    fig.set_figwidth(20)
    fig.set_figheight(20)
```





[84]:
```
#print(Train_Labels[(Predicted_Train_PCA==0).reshape(18998,)])
#print(Train_Labels[(Predicted_Train_PCA==1).reshape(18998,)])
#print(Train_Labels[(Predicted_Train_PCA==2).reshape(18998,)])
```

[ ]:
```
```

[89]:
```
for i in range(5):
    KM = KMeans(n_clusters=8, random_state=i,max_iter=1000).fit(Train_vector)
    Train_Labels_Predicted=KM.labels_.reshape(18998,)
    ARI_0=metrics.adjusted_rand_score(Train_Labels_N,Train_Labels_Predicted)
    print(ARI_0)
```

```
0.03257147723726654
0.03253302267653264
0.03322015897594239
0.03311362414789281
0.033408990647601305
```

[90]:
```
for i in range(5):
    KM = KMeans(n_clusters=7, random_state=i,max_iter=1000).fit(Train_vector)
    Train_Labels_Predicted=KM.labels_.reshape(18998,)
    ARI_0=metrics.adjusted_rand_score(Train_Labels_N,Train_Labels_Predicted)
    print(ARI_0)
```

```
0.027437009860177223
0.02743032370117866
0.027373612457792303
0.027629647796892493
0.027671894139486345
```

[91]:
```
for i in range(5):
    KM = KMeans(n_clusters=6, random_state=i,max_iter=1000).fit(Train_vector)
    Train_Labels_Predicted=KM.labels_.reshape(18998,)
    ARI_0=metrics.adjusted_rand_score(Train_Labels_N,Train_Labels_Predicted)
    print(ARI_0)
```

```
0.06228033258952991
0.06212128023785067
0.06228206651773841
0.06227490336845291
```

```
0.06228033258952991
```

[92]:
```
for i in range(5):
    KM = KMeans(n_clusters=5, random_state=i,max_iter=1000).fit(Train_vector)
    Train_Labels_Predicted=KM.labels_.reshape(18998,)
    ARI_0=metrics.adjusted_rand_score(Train_Labels_N,Train_Labels_Predicted)
    print(ARI_0)
```

```
0.05676287739628209
0.05677390543494453
0.05677390543494453
0.05679435094490519
0.05677390543494453
```

[93]:
```
for i in range(5):
    KM = KMeans(n_clusters=4, random_state=i,max_iter=1000).fit(Train_vector)
    Train_Labels_Predicted=KM.labels_.reshape(18998,)
    ARI_0=metrics.adjusted_rand_score(Train_Labels_N,Train_Labels_Predicted)
    print(ARI_0)
```

```
0.05832684330627737
0.058071042943392194
0.05832684330627737
0.058071042943392194
0.05826033187178842
```

[95]:
```
for i in range(5):
    KM = KMeans(n_clusters=9, random_state=i,max_iter=1000).fit(Train_vector)
    Train_Labels_Predicted=KM.labels_.reshape(18998,)
    ARI_0=metrics.adjusted_rand_score(Train_Labels_N,Train_Labels_Predicted)
    print(ARI_0)
```

```
0.02716416369390437
0.027540936533692308
0.02726511957242683
0.027412766939942237
0.02700257644098349
```

[ ]:
```

```

P3_GMM

December 9, 2022

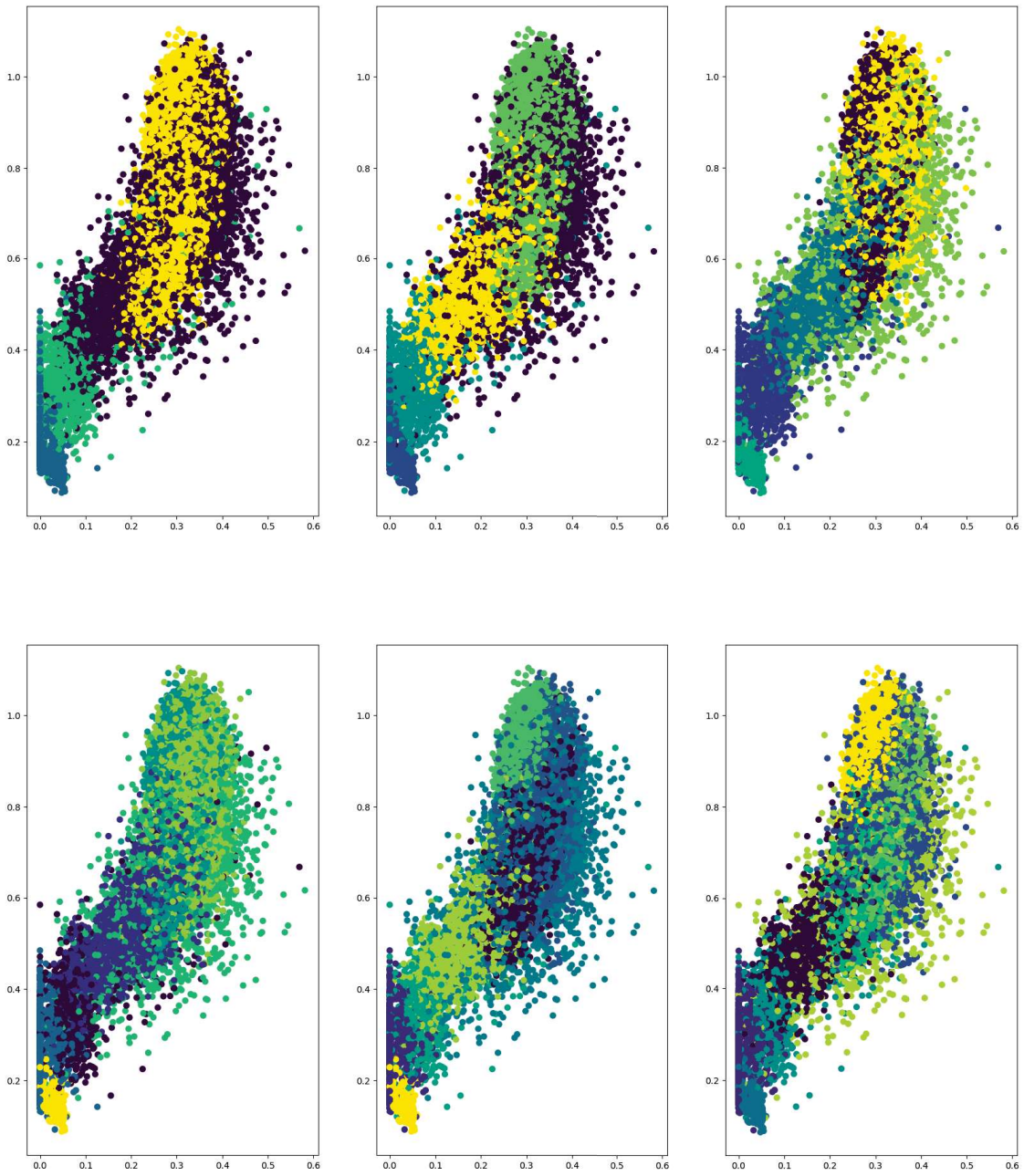
[84]:
```
import numpy as np
from keras import models
import matplotlib.pyplot as plt
from keras import layers
from sklearn import metrics
from sklearn.decomposition import PCA
from sklearn.metrics import adjusted_rand_score
from sklearn import mixture
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
Cell In [84], line 9
      7 from sklearn.metrics import adjusted_rand_score
      8 from sklearn import mixture
----> 9 np.set_printoptions(threshold=sys.maxsize)

NameError: name 'sys' is not defined
```

[34]:
```
Train_Data=np.load('encoded_train_new.npy')
Test_Data=np.load('encoded_test_new.npy')
Train_Labels=np.load('label_train.npy')
Test_Labels=np.load('label_test.npy')
C = ['AK', 'BCC', 'BKL', 'DF', 'MEL', 'NV', 'SCC', 'VASC']
def CLS(n):
    if n=='AK':
        return 1
    elif n=='BCC':
        return 2
    elif n=='BKL':
        return 2
    elif n=='DF':
        return 3
    elif n=='MEL':
        return 4
    elif n=='NV':
        return 5
```

```
    elif n=='SCC':
        return 6
    elif n=='VASC':
        return 7
Train_Labels_N = np.array(list(map(CLS,Train_Labels)))

Train_vector=np.reshape(Train_Data,(Train_Data.shape[0],Train_Data.
 ↪shape[1]*Train_Data.shape[2]*Train_Data.shape[3]))
Test_vector=np.reshape(Test_Data,(Test_Data.shape[0],Test_Data.
 ↪shape[1]*Test_Data.shape[2]*Test_Data.shape[3]))
N_C=[4,5,6,7,8,9]
```

[38]:
```
fig, axs = plt.subplots(1, 3)
for i in range(3):
    gmm = mixture.GaussianMixture(n_components=N_C[i], covariance_type='full')
    gmm.fit(Train_vector)
    pred_labels = gmm.predict(Train_vector)
    axs[i].scatter(Train_vector[:, 0], Train_vector[:, 1], c=pred_labels, s=40,␣
 ↪cmap='viridis');
    axs[i].set_aspect('equal', adjustable='box')
    fig.set_figwidth(20)
    fig.set_figheight(20)

fig, axs = plt.subplots(1, 3)
for i in range(3):
    gmm = mixture.GaussianMixture(n_components=N_C[i+3], covariance_type='full')
    gmm.fit(Train_vector)
    pred_labels = gmm.predict(Train_vector)
    axs[i].scatter(Train_vector[:, 0], Train_vector[:, 1], c=pred_labels, s=40,␣
 ↪cmap='viridis');
    axs[i].set_aspect('equal', adjustable='box')
    fig.set_figwidth(20)
    fig.set_figheight(20)
```



[55]:
```
for i in range(5):
    gmm = mixture.GaussianMixture(n_components=3, covariance_type='full').
 ↪fit(Train_vector)
    pred_labels=gmm.predict(Train_vector)
    iris_gmm_score = adjusted_rand_score(Train_Labels_N,pred_labels)
    print(iris_gmm_score)
```

```
0.10078573727001604
0.10078573727001604
```

```
0.10069711435607817
0.1006587190519865
0.10078573727001604
```

[54]:
```
for i in range(5):
    gmm = mixture.GaussianMixture(n_components=4, covariance_type='full').
 ↪fit(Train_vector)
    pred_labels=gmm.predict(Train_vector)
    iris_gmm_score = adjusted_rand_score(Train_Labels_N,pred_labels)
    print(iris_gmm_score)
```

```
0.08307219281428575
0.08610417816577928
0.08610417816577928
0.08302707877316572
0.08307396129933324
```

[56]:
```
for i in range(5):
    gmm = mixture.GaussianMixture(n_components=5, covariance_type='full').
 ↪fit(Train_vector)
    pred_labels=gmm.predict(Train_vector)
    iris_gmm_score = adjusted_rand_score(Train_Labels_N,pred_labels)
    print(iris_gmm_score)
```

```
0.11985876502936947
0.11990490402873248
0.11985548080639095
0.11990490402873248
0.11990490402873248
```

[57]:
```
for i in range(5):
    gmm = mixture.GaussianMixture(n_components=6, covariance_type='full').
 ↪fit(Train_vector)
    pred_labels=gmm.predict(Train_vector)
    iris_gmm_score = adjusted_rand_score(Train_Labels_N,pred_labels)
    print(iris_gmm_score)
```

```
0.056213179598270095
0.05552708760852427
0.10337784627070433
0.05645087586342813
0.10304120092997304
```

[58]:
```
for i in range(5):
    gmm = mixture.GaussianMixture(n_components=7, covariance_type='full').
 ↪fit(Train_vector)
    pred_labels=gmm.predict(Train_vector)
    iris_gmm_score = adjusted_rand_score(Train_Labels_N,pred_labels)
    print(iris_gmm_score)
```

```
0.04966131679215823
0.08605365844570809
0.0609475579190771
0.06141330482909476
0.061089039355665904
```

[59]:
```
for i in range(5):
    gmm = mixture.GaussianMixture(n_components=8, covariance_type='full').
 ↪fit(Train_vector)
    pred_labels=gmm.predict(Train_vector)
    iris_gmm_score = adjusted_rand_score(Train_Labels_N,pred_labels)
    print(iris_gmm_score)
```

```
0.0536513603881418
0.053256774303710445
0.05202740042029475
0.05400150316808485
0.05209107173762925
```

[60]:
```
for i in range(5):
    gmm = mixture.GaussianMixture(n_components=9, covariance_type='full').
 ↪fit(Train_vector)
    pred_labels=gmm.predict(Train_vector)
    iris_gmm_score = adjusted_rand_score(Train_Labels_N,pred_labels)
    print(iris_gmm_score)
```

```
0.04957817373127986
0.04962130624586098
0.04970024768878609
0.0495999130415391
0.04957855676479886
```

[29]:
```
probs = model.predict_proba(Train_vector)
print(probs[:5].round(3))
```

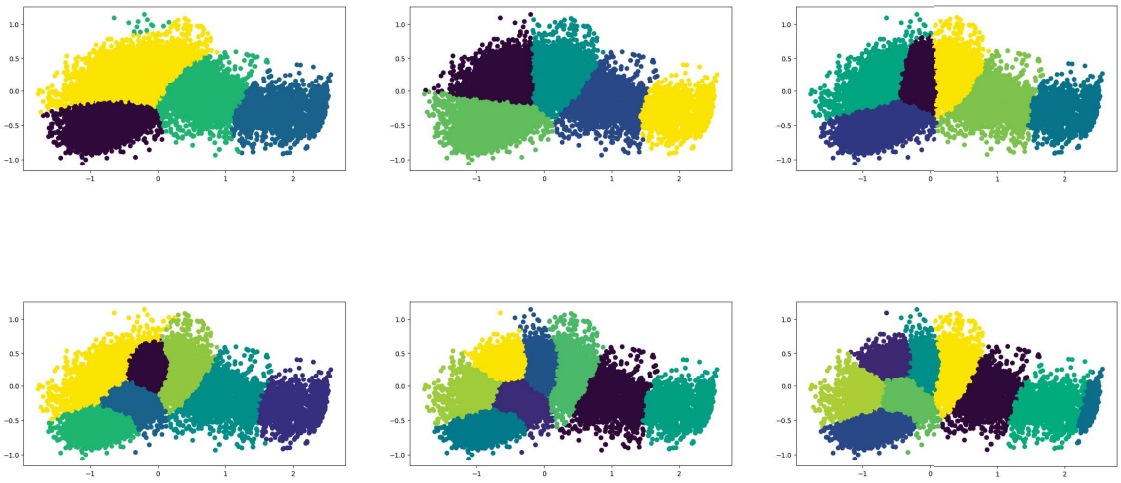
```
[[0.    0.    0.    0.    0.727 0.    0.    0.273]
 [0.999 0.    0.    0.    0.    0.    0.    0.   ]
 [0.    0.    1.    0.    0.    0.    0.    0.   ]
 [0.    0.    0.    0.    0.003 0.    0.    0.997]
 [0.    0.    0.    0.    0.    0.    0.034 0.966]]
```

[63]:
```
pca = PCA(n_components=2)
principalComponents = pca.fit_transform(Train_vector)
np.shape(principalComponents)
```

[63]:
```
(18998, 2)
```

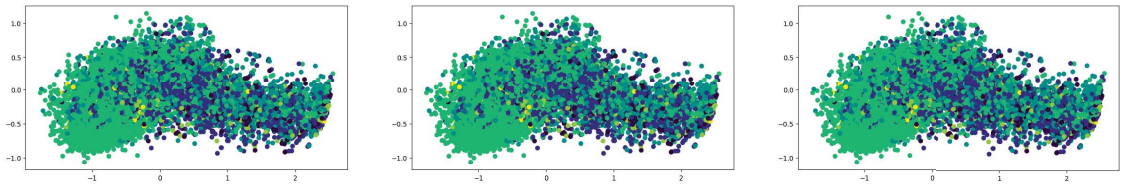
[85]:
```
fig, axs = plt.subplots(1, 3)
for i in range(3):
```

```
    gmm_pca = mixture.GaussianMixture(n_components=N_C[i],␣
 ↪covariance_type='full').fit(principalComponents)
    pred_labels = gmm_pca.predict(principalComponents)
    axs[i].scatter(principalComponents[:, 0], principalComponents[:, 1],␣
 ↪c=pred_labels, s=40, cmap='viridis');
    axs[i].set_aspect('equal', adjustable='box')
    fig.set_figwidth(30)
    fig.set_figheight(30)


fig, axs = plt.subplots(1, 3)
for i in range(3):
    gmm = mixture.GaussianMixture(n_components=N_C[i+3],␣
 ↪covariance_type='full').fit(principalComponents)
    pred_labels = gmm.predict(principalComponents)
    axs[i].scatter(principalComponents[:, 0], principalComponents[:, 1],␣
 ↪c=pred_labels, s=40, cmap='viridis');
    axs[i].set_aspect('equal', adjustable='box')
    fig.set_figwidth(30)
    fig.set_figheight(30)
```



[81]:
```
fig, axs = plt.subplots(1, 3)
for i in range(3):
    gmm = mixture.GaussianMixture(n_components=8, covariance_type='full').
 ↪fit(principalComponents)
    pred_labels = gmm.predict(principalComponents)
    axs[i].scatter(principalComponents[:, 0], principalComponents[:, 1],␣
 ↪c=Train_Labels_N, s=40, cmap='viridis');
    axs[i].set_aspect('equal', adjustable='box')
```

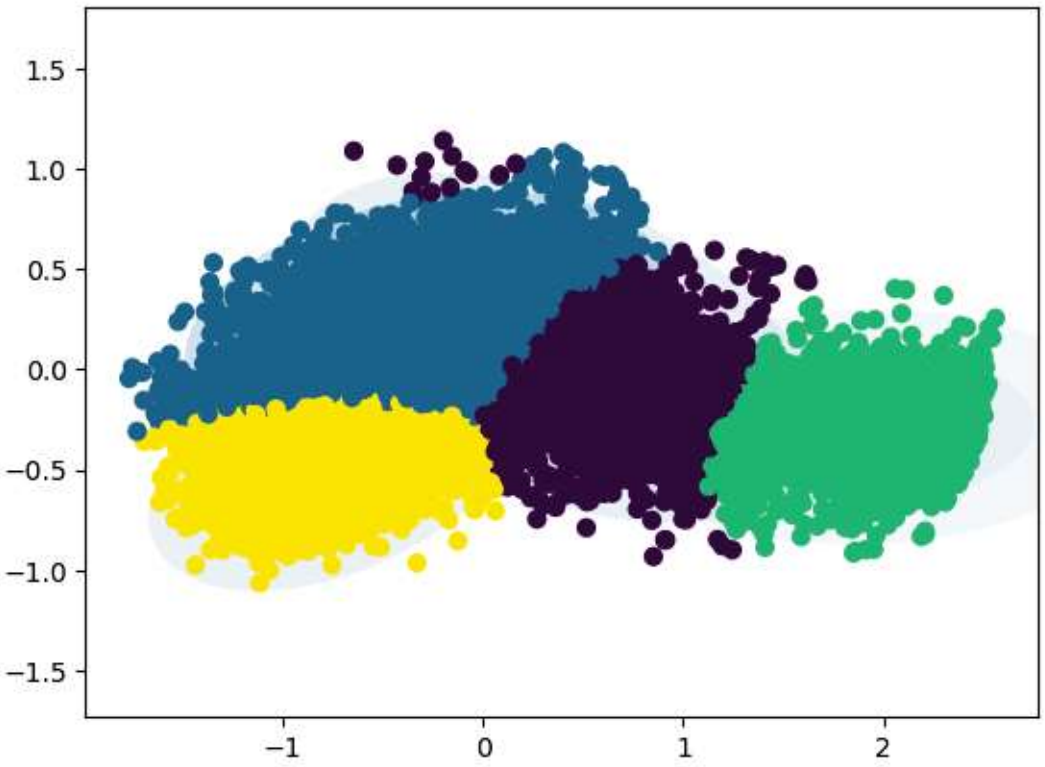
```
    fig.set_figwidth(30)
    fig.set_figheight(30)
```



[72]:
```
from matplotlib.patches import Ellipse

def draw_ellipse(position, covariance, ax=None, **kwargs):
    """Draw an ellipse with a given position and covariance"""
    ax = ax or plt.gca()

    # Convert covariance to principal axes
    if covariance.shape == (2, 2):
        U, s, Vt = np.linalg.svd(covariance)
        angle = np.degrees(np.arctan2(U[1, 0], U[0, 0]))
        width, height = 2 * np.sqrt(s)
    else:
        angle = 0
        width, height = 2 * np.sqrt(covariance)

    # Draw the Ellipse
    for nsig in range(1, 4):
        ax.add_patch(Ellipse(position, nsig * width, nsig * height,
                             angle, **kwargs))

def plot_gmm(gmm, X, label=True, ax=None):
    ax = ax or plt.gca()
    labels = gmm.fit(X).predict(X)
    if label:
        ax.scatter(X[:, 0], X[:, 1], c=labels, s=40, cmap='viridis', zorder=2)
    else:
        ax.scatter(X[:, 0], X[:, 1], s=40, zorder=2)
    ax.axis('equal')

    w_factor = 0.2 / gmm.weights_.max()
    for pos, covar, w in zip(gmm.means_, gmm.covariances_, gmm.weights_):
        draw_ellipse(pos, covar, alpha=w * w_factor)
```

[73]:
```
gmm_pca = mixture.GaussianMixture(n_components=4, covariance_type='full').
↪fit(principalComponents)
```

```
pred_labels = gmm_pca.predict(principalComponents)
#plt.scatter(principalComponents[:, 0], principalComponents[:, 1],␣
 ↪c=pred_labels, s=40, cmap='viridis');
plot_gmm(gmm_pca, principalComponents)
```



[74]:
```
for i in range(5):
    gmm = mixture.GaussianMixture(n_components=3, covariance_type='full').
 ↪fit(principalComponents)
    pred_labels=gmm.predict(principalComponents)
    iris_gmm_score = adjusted_rand_score(Train_Labels_N,pred_labels)
    print(iris_gmm_score)
```

```
0.046991949062423204
0.04728566609307114
0.045229038803277076
0.047363725864195044
0.047453123857841124
```

[75]:
```
for i in range(5):
    gmm = mixture.GaussianMixture(n_components=4, covariance_type='full').
 ↪fit(principalComponents)
    pred_labels=gmm.predict(principalComponents)
```

```
    iris_gmm_score = adjusted_rand_score(Train_Labels_N,pred_labels)
    print(iris_gmm_score)
```

```
0.05308083887629136
0.05211378794640307
0.052883710598860115
0.052379716702504665
0.05258859934169465
```

[76]:
```
for i in range(5):
    gmm = mixture.GaussianMixture(n_components=5, covariance_type='full').
 ↪fit(principalComponents)
    pred_labels=gmm.predict(principalComponents)
    iris_gmm_score = adjusted_rand_score(Train_Labels_N,pred_labels)
    print(iris_gmm_score)
```

```
0.05732661323750912
0.05688408719960093
0.05869146245346531
0.057047501161783774
0.0572330752760708
```

[77]:
```
for i in range(5):
    gmm = mixture.GaussianMixture(n_components=6, covariance_type='full').
 ↪fit(principalComponents)
    pred_labels=gmm.predict(principalComponents)
    iris_gmm_score = adjusted_rand_score(Train_Labels_N,pred_labels)
    print(iris_gmm_score)
```

```
0.04260723871490613
0.04625821440581058
0.047936608122825394
0.05474186626832874
0.04324911860772387
```

[78]:
```
for i in range(5):
    gmm = mixture.GaussianMixture(n_components=7, covariance_type='full').
 ↪fit(principalComponents)
    pred_labels=gmm.predict(principalComponents)
    iris_gmm_score = adjusted_rand_score(Train_Labels_N,pred_labels)
    print(iris_gmm_score)
```

```
0.042883048770105944
0.04456888447053922
0.04738085692899096
0.04445260218168275
0.04514117224580357
```

[79]:
```
for i in range(5):
    gmm = mixture.GaussianMixture(n_components=8, covariance_type='full').
 ↪fit(principalComponents)
    pred_labels=gmm.predict(principalComponents)
    iris_gmm_score = adjusted_rand_score(Train_Labels_N,pred_labels)
    print(iris_gmm_score)
```

```
0.034345497049986184
0.03591093267716896
0.031074930654965294
0.03555584359125267
0.04308728150082124
```

[86]:
```
for i in range(5):
    gmm = mixture.GaussianMixture(n_components=9, covariance_type='full').
 ↪fit(principalComponents)
    pred_labels=gmm.predict(principalComponents)
    iris_gmm_score = adjusted_rand_score(Train_Labels_N,pred_labels)
    print(iris_gmm_score)
```

```
0.030558708404088423
0.03981534467318
0.039906309212827264
0.029435064484042192
0.02835025464592849
```

[ ]:
```

```