Project Report

Please refer to the source code within the Jupyter notebooks for more information. "PCA_LDA_GMM_SVM.ipynb" contains codes for Sections 1-4 while "NN.ipynb" contains codes for Section 5. Float figures, where applicable, are reported to 3 decimal places.

A. Introduction

This report is for the EE5907 CA2 task on Face Recognition.

B. Dataset

Data Selection

The project utilised the CMU PIE dataset plus personal selfies. Out of the 68 different subjects, 25 were chosen to be included in this study.

```
Selected subjects: [2, 8, 13, 14, 15, 17, 19, 20, 23, 24, 27, 31, 32, 33, 34, 40, 42, 44, 46, 48, 50, 52, 59, 63, 65]
```

Since each subject has 170 images, plus 10 selfies were included, a final total of 4260 images are utilised in this project. The 10 selfies were processed (centered, cropped, resized and grayscaled) via the script available under `format_own_images.ipynb`.

```
Number of selected PIE images: 4250
Number of selected selfies: 10
Number of selected images: 4260
```

Train Test Split

The CMU PIE images and selfies were each split into 70% for training and 30% for testing. The training and testing samples remain fixed throughout the report.

```
Number||Proportion of train PIE images: 2975 || 0.7
Number||Proportion of test PIE images: 1275 || 0.3
Number||Proportion of train selfies: 7 || 0.7
Number||Proportion of test selfies: 3 || 0.3
```

01. Principal Component Analysis (PCA)

Pre-processing

Each raw face image was converted from 32 x 32 pixels to a 1024-dimensional vector.

Sub-train dataset for visualisation

I further randomly sampled 500 images from the CMU PIE training set and added the 7 training selfies. This sub-train dataset is for the purpose of visualisations as requested by the task description. This resulted in a working matrix of the following shape:

```
Vectorised and loaded data: (507, 1024)
```

PCA 2D & 3D

PCA package from `sklearn.decomposition` was utilised to reduce the dimensionality of vectorized images to 2 and 3. The projected data vectors in 2D and 3D plots are shown below respectively in Figures 1 and 2, where the blue points reflect 'PIE' images while orange reflects the 'selfies'. Figures 3 and 4 correspond exactly to 1 and 2, but all the individual subjects belonging to PIE are now highlighted by a different colour each. There is a clear clustering effect of selfies in all four figures, but it is a little harder to discern the clustering patterns for PIE subjects.

```
2-PCs explained variance ratio: [0.416 0.263]
2-PCs singular values: [24469.939 19480.850]

3-PCs explained variance ratio: [0.416 0.263 0.073]
3-PCs singular values: [24469.939 19480.850 10266.821]
```

Figure 1

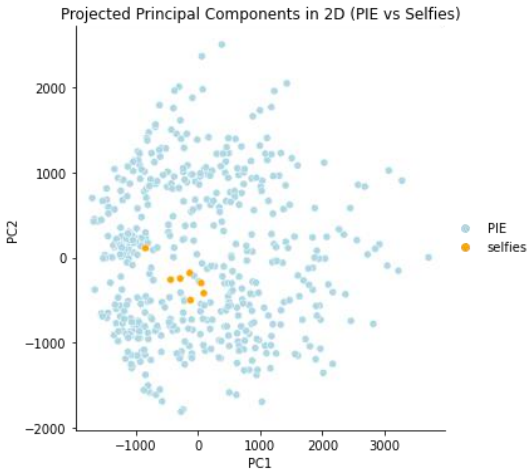


Figure 2

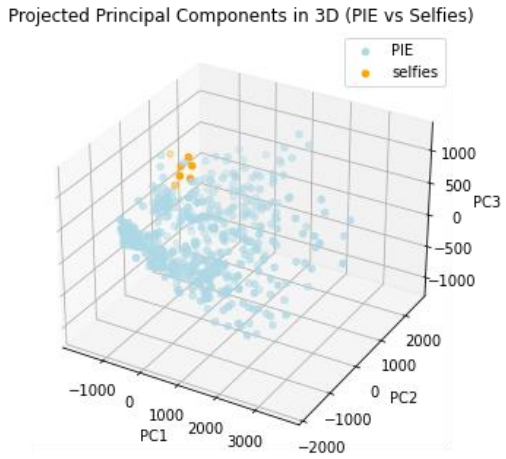


Figure 3

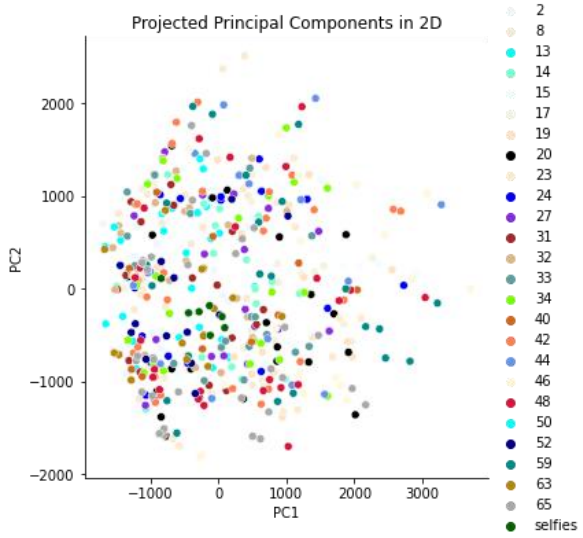
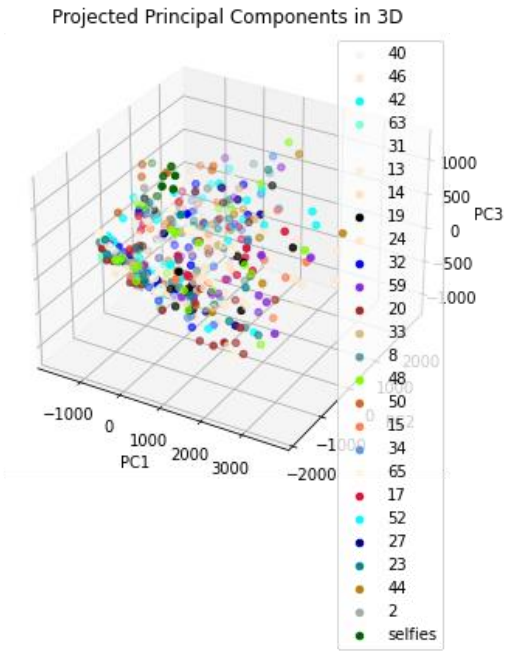
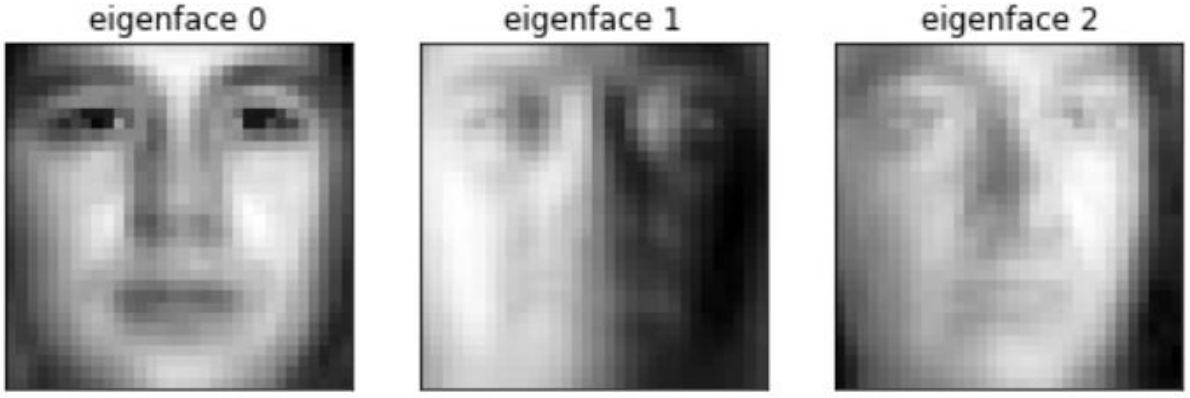


Figure 4



Eigenfaces

The following is the visualisation of the 3 eigenfaces used for dimensionality reduction:



The eigenfaces are derived from eigenvectors and covariance matrices of the high-dimensional vector space of the randomly sampled 507 training faces. As some rough observations, eigenface 0 looks like the representations of some front-facing faces while eigenface 2 looks like a right-facing face.

PCA Classification

I repeated PCA to reduce the dimensionality of face images to number of principal components n_pc=40, 80, 200. Here, I revert to use the full train and test dataset for modelling. For each n_pc value, 1-NN (nearest neighbour) classification using KNeighborsClassifier package from `sklearn.neighbors` with k=1 was performed. The classification accuracies are as follows:

```
Accuracy of 1-NN with 40-components PCA for ALL test set: 93.114%
Accuracy of 1-NN with 40-components PCA for PIE test set: 93.171%
Accuracy of 1-NN with 40-components PCA for selfies test set: 66.667%
Accuracy of 1-NN with 80-components PCA for ALL test set: 95.149%
Accuracy of 1-NN with 80-components PCA for PIE test set: 95.212%
Accuracy of 1-NN with 80-components PCA for selfies test set: 66.667%
Accuracy of 1-NN with 200-components PCA for ALL test set: 96.088%
Accuracy of 1-NN with 200-components PCA for PIE test set: 96.154%
Accuracy of 1-NN with 200-components PCA for selfies test set: 66.667%
```

Based on the above results, the greater the number of PCA components, the better the classification accuracy on PIE images but the changes are minimal. For selfies, the accuracy is the same throughout at (2/3)%.

02. Linear Discriminant Analysis (LDA)

LDA 2D & 3D plots

LinearDiscriminantAnalysis package from `sklearn. discriminant_analysis` was utilised to reduce the dimensionality of vectorized images to 2, 3 and 9. The projected data vectors in 2D and 3D plots are shown below respectively in Figures 5 and 6, where the blue points reflect 'PIE' images while orange reflects the 'selfies'. Figures 7 and 8 correspond exactly to 5 and 6, but all the individual subjects belonging to PIE are now highlighted by a different colour each. There is a very clear clustering effect of selfies in all four figures, as shown by how disjoint the selfies' points are from the main group across axis X1. Figure 7 also show some clustering patterns for each PIE subject,

especially for subjects 48, 65, 20 and 17 that are spread out across axis X2 and 24 that is slightly further to the left on axis X1.

```
2-components LDA explained variance ratio: [0.144 0.094]
3-components LDA explained variance ratio: [0.144 0.094 0.074]
9-components LDA explained variance ratio: [0.144 0.094 0.074 0.068 0.063 0.0
55 0.053 0.046 0.041]
```

Figure 5

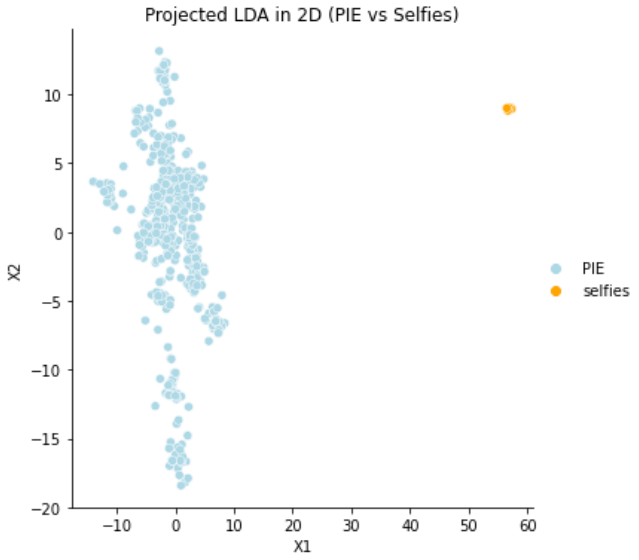


Figure 6

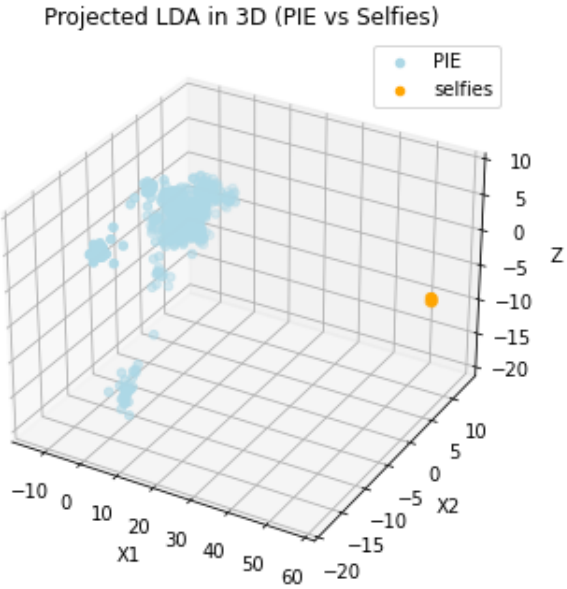


Figure 7

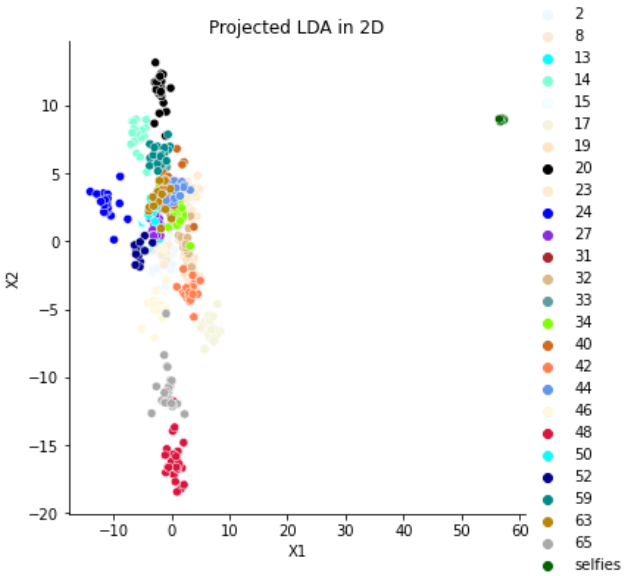
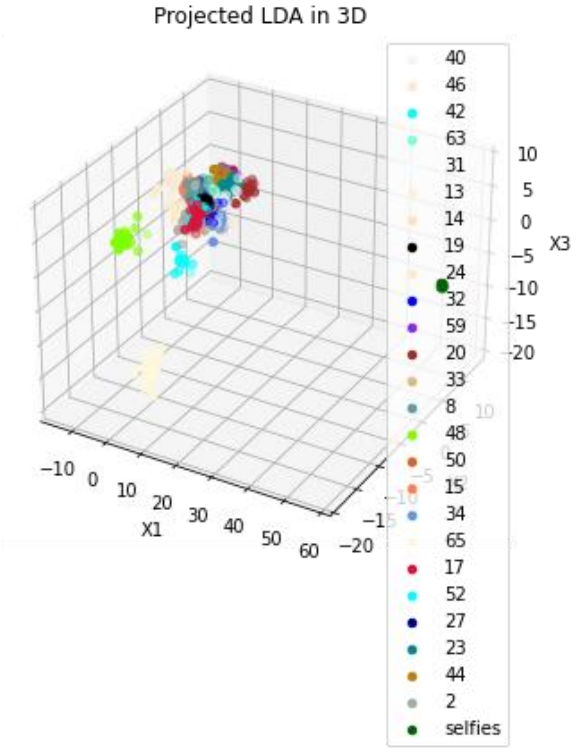


Figure 8



LDA Classification

I repeated PCA to reduce the dimensionality of face images to number of principal components n_pc=2,3,9. Again, I revert to use the full train and test dataset for modelling. For each n_pc value, 1-NN (nearest neighbour) classification using KNeighborsClassifier package from `sklearn.neighbors` with k=1 was performed. The classification accuracies are as follows:

```
Accuracy of 1-NN with 2-components LDA for ALL test set: 23.944%
Accuracy of 1-NN with 2-components LDA for PIE test set: 23.94%
Accuracy of 1-NN with 2-components LDA for selfies test set: 0.0%
Accuracy of 1-NN with 3-components LDA for ALL test set: 41.706%
Accuracy of 1-NN with 3-components LDA for PIE test set: 41.758%
Accuracy of 1-NN with 3-components LDA for selfies test set: 0.0%
Accuracy of 1-NN with 9-components LDA for ALL test set: 90.219%
Accuracy of 1-NN with 9-components LDA for PIE test set: 90.424%
Accuracy of 1-NN with 9-components LDA for selfies test set: 0.0%
```

Based on the above results, the greater the number of LDA components, the better the classification accuracy on PIE images, with rather drastic increases in accuracy (2-components to 3 warrants a doubling in accuracy). For selfies, the accuracy is the same throughout at 0%, which is rather surprising given how well clustered selfies seems in Figures 5-8. I think perhaps the few number of train selfies compared to PIE subjects' images (7 vs 2968) led to under-sampling bias, and the classifier is unable to properly discern selfies. As an exploratory exercise, I retrained the LDA x 1NN classifier with the 507-subset training data (where the ratio of selfies to PIE is 7 vs 500), and indeed, accuracy for selfies test set is high:

```
Accuracy of 1-NN with 2-components LDA for selfies test set: 100.0%
Accuracy of 1-NN with 3-components LDA for selfies test set: 66.667%
Accuracy of 1-NN with 9-components LDA for selfies test set: 100.0%
```

This makes sense since 507-subset training data holds (7/507)% of selfies compared to full training data that only holds (7/2982)%, making it hard for any classifier to probabilistically predict selfie as a class label.

03. Gaussian Mixture Model (GMM)

Using raw face images

Using vectorised raw face images (i.e. data of shape (2982, 1024)) as inputs to train a 3-Gaussian components GMM model, we perform clustering by assigning each data point by one of the three possible labels. Figures 9 and 10 plots the GMM clustering results on 2D and 3D spaces (based on PCA components from the earlier section). Due to the large number of PCs, I am only visualising scatterplots based on the first 2 and 3 components. The predicted labels are highlighted by 0=red, 1=green and 2=blue.

Figure 9

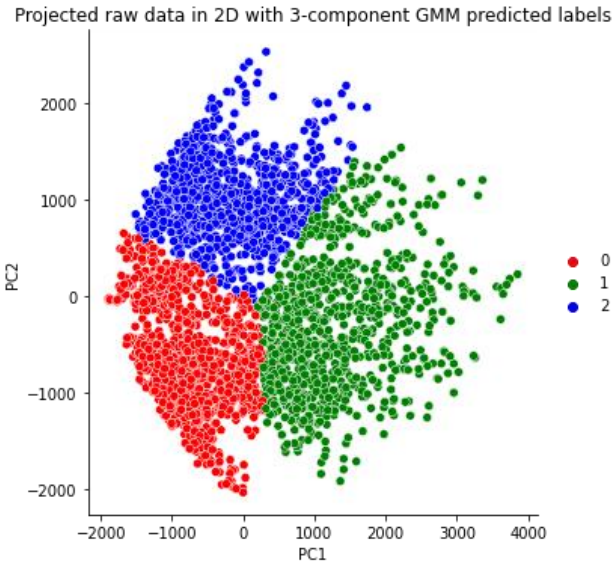
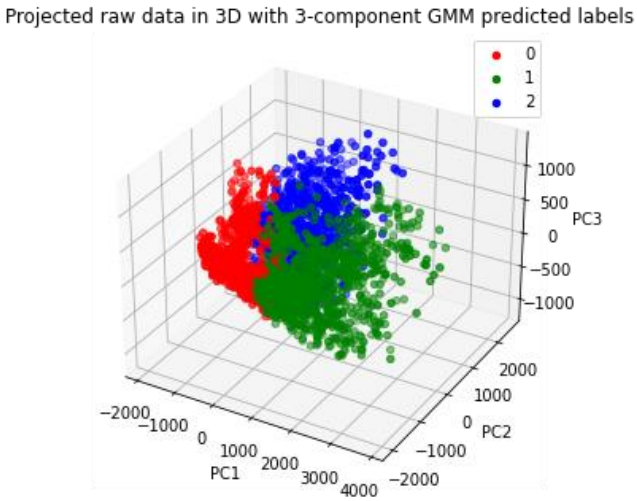


Figure 10



Using PCA pre-processed images

I used face vectors after PCA pre-processing at n_pc=80 and 200. These serve as training data which was fed into a 3-component GMM model. To observe the clustering by GMM, Figures 11-14 plots PCA transformed data in 2D and 3D spaces, and predicted labels are again highlighted by 0=red, 1=green and 2=blue.

Figure 11

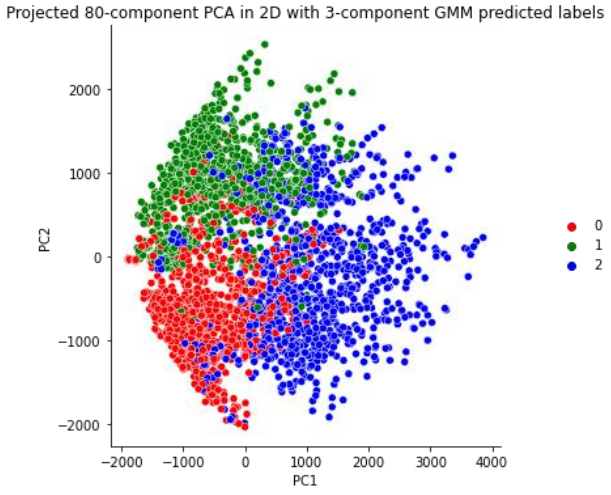


Figure 12

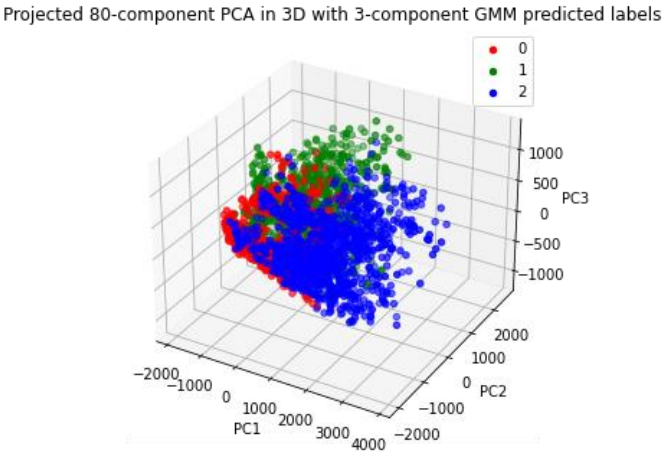


Figure 13

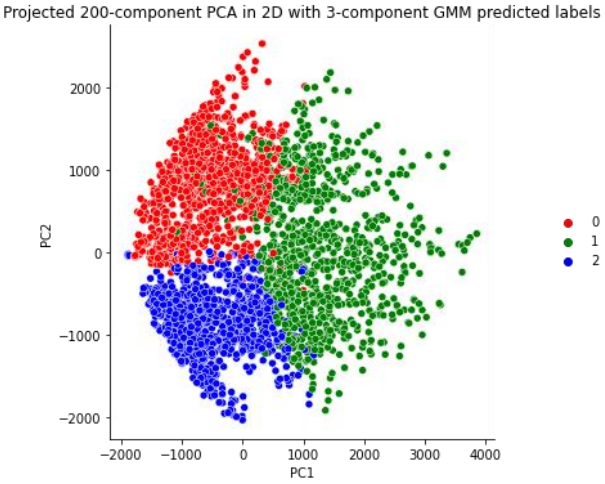
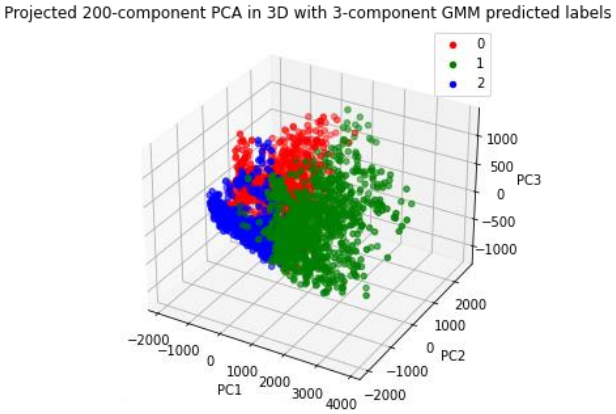


Figure 14



For all scatterplots in Figures 9-14, we observe three clusters per plot with differing mean, but some overlap of points across cluster boundaries. Usage of raw data inputs seems to generate the cleanest clusters in Figures 9 and 10 compared to the 80 and 200 component set-ups. This might be because GMM learnt latent characteristics that reflect PC1 and PC2. Intuitively, the input data is also largest with raw data (utilises all 1024 features), thus allowing for better expectation maximisation. On a similar note, we observe slightly better clustering (fewer overlapping points across clusters) for n_pc=200 compared to 80 plots. The PCA+GMM set-up that utilises dimensionality reduction via PCA does not help uncover clustering characteristics (Scrucca, 2015).

To dive deeper, the images assigned to the same GMM label based on the raw data set-up are plotted against PC1 and PC2. One notable point is selfies only appear for GMM label=2. For the PIE subjects, the plots are rather inconclusive and hard to observe any clear clustering of the actual subjects' labels. For this reason, I do not include the 80-component and 200-component plots, but they are available in the Jupyter notebook if interested.

Figure 15

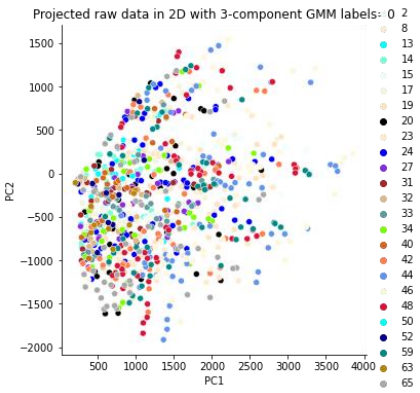


Figure 16

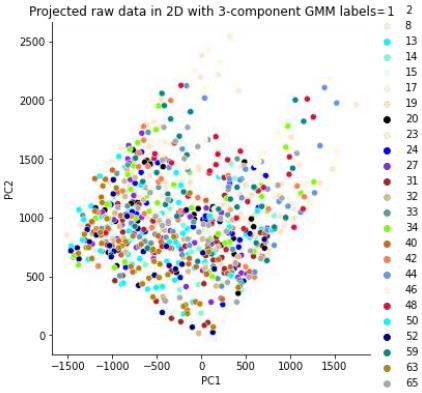
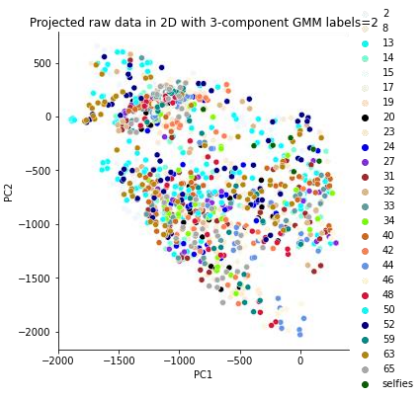


Figure 17



04. Support Vector Machine (SVM)

Using both raw face images and face vectors after PCA pre-processing at n_pc=80 and 200, I applied a linear SVM classification model on various penalty parameter C values. The classification accuracy on test set is as shown in the first four rows of Table 1.

For the linear model, as the data dimension becomes smaller (sorted from largest on right to smallest on left in Table 1), the accuracy does not seem to change much. For C=1e-2, it remains constant then decreases slightly. For C=1e-1, it decreases slightly but then increases back up. As the lowest pc=80 is still quite large, it is probably already able to capture the important features in an image to recognize the face apart. As an extension, I reduced n_pc further to 50, 30, and 10 with C=1e-2, which returned test accuracies of 97.731%, 97.027% and 74.804% respectively. This shows that smaller data dimension reduces accuracy, but only when we go beyond too small a data dimension in the first place.

For the linear model, as the parameter C value increases (sorted in increasing order in Table 1 per model type), the accuracy again does not change much. For n_pc=None (raw face images), accuracy maintains constant throughout. For n_pc=200, accuracy decreases slightly but increases back up again (C=1e-3 has same accuracy as C=1). Lastly, for n_pc=80, accuracy fluctuates around with no apparent trend. Since the parameter C reflects the inverse strength of regularization, it seems that regularisation is not too important in this setting.

Table 1

Model Description	SVM Settings	Features								
		None			n_pc=200			n_pc=80		
		ALL	PIE	selfies	ALL	PIE	selfies	ALL	PIE	selfies
Linear	'-t 0 -c 1e-3'	**98.513%**	98.587%	66.667%	**98.592%**	98.666%	66.667%	**98.357%**	98.430%	66.667%
	'-t 0 -c 1e-2'	**98.513%**	98.587%	66.667%	**98.513%**	98.587%	66.667%	**98.279%**	98.352%	66.667%
	'-t 0 -c 1e-1'	**98.513%**	98.587%	66.667%	**98.435%**	98.509%	66.667%	**98.513%**	98.587%	66.667%
	'-t 0 -c 1'	**98.513%**	98.587%	66.667%	**98.592%**	98.666%	66.667%	**98.435%**	98.509%	66.667%
Polynomial	'-t 1 -c 1e-2'	**96.322%**	96.389%	66.667%	**96.635%**	96.703%	66.667%	**96.948%**	97.017%	66.667%
	'-t 1 -c 1e-1'	**96.322%**	96.389%	66.667%	**96.557%**	96.625%	66.667%	**96.948%**	97.017%	66.667%
Radial Basis Function	'-t 2 -c 1e-2'	**3.130%**	3.140%	0.000%	**3.130%**	3.140%	0.000%	**3.130%**	3.140%	0.000%
Sigmoid	'-t 3 -c 1e-2'	**3.130%**	3.140%	0.000%	**5.556%**	5.573%	0.000%	**5.556%**	5.573%	0.000%

* Highest ALL accuracy score per row is underlined

I also experimented with 3 other kernel types (polynomial, radial basis function (RBF), sigmoid) and report their accuracy scores. Linear SVM appears to be the best model across all assessed feature dimensions. RBF and sigmoid performs extremely poorly.

As a side note, I used libsvm package to report the above scores. I checked for two instance that using the SVC package from `sklearn.svm` leads to same results (E.g. 98.513% for linear SVM with c=1e-2 for raw face images).

05. Neural Networks (NN)

NN for Classification

Using Pytorch framework, I created 2-layer CNN plus 1 feedforward NN model. The full codes are also within the Jupyter notebook but the main pipeline is shown below in Figure 18. I refer to this model as the base case for this section of the report.

Rectified linear unit (ReLU) activation function was included between convolutions and linear layers for better performance. The number of the nodes in the output layer is fixed as 26 (to predict 25 PIE + 1 selfie labels) before parsing into a log_softmax layer to obtain probabilities per class label. Argmax was taken to predict the most probable class label per sample. I allowed the model to train for up to 15 epochs, with no early stopping as the time taken was quite short. The trained model was then used to predict upon the test data, ultimately achieving an overall accuracy of 97.574%, with further breakdown as follows:

```
Accuracy of NN for ALL test set: 97.574%
Accuracy of NN for PIE test set: 97.725%
Accuracy of NN for selfies test set: 33.333%
```

Figure 18

```
class CNN(nn.Module):
    def __init__(self, embed_dim, hidden_dim1, hidden_dim2, hidden_dim3, output_dim, kernel_size, pool_k_size, pool_stride):
        super(CNN, self).__init__()
        self.conv1 = nn.Conv2d(embed_dim, hidden_dim1, kernel_size=kernel_size)
        self.conv2 = nn.Conv2d(hidden_dim1, hidden_dim2, kernel_size=kernel_size)
        self.pool = nn.MaxPool2d(kernel_size=pool_k_size, stride=pool_stride)
        self.fc1 = nn.Linear(hidden_dim2*5*5, hidden_dim3)
        self.fc2 = nn.Linear(hidden_dim3, output_dim)

    def forward(self, feats_in):
        batch_size, e_, e_, s_ = feats_in.shape
        x = self.pool(F.relu(self.conv1(feats_in.float())))
        x = self.pool(F.relu(self.conv2(x)))
        x = torch.flatten(x, start_dim=1)
        x = F.relu(self.fc1(x))
        x = F.log_softmax(self.fc2(x),dim=1)
        return x

model = CNN(embed_dim=1,
            hidden_dim1=20,
            hidden_dim2=50,
            hidden_dim3=500,
            output_dim=26,
            kernel_size=(5,5),
            pool_k_size=(2,2),
            pool_stride=2
           ).to(device)
loss_function = nn.CrossEntropyLoss().to(device)
optimizer = optim.SGD(model.parameters(), lr=LEARNING_RATE)
```

Experimenting with different network architectures

1. Without ReLU between CNN Layers

As an extension, I removed the ReLU layers that were not originally requested by the task description (See Figure 19 vs 18 for changes) and show that such a model does not train at all in Figure 20, where loss is high and accuracies do not improve over iterations and epochs. As mentioned in lecture, a composition of linear functions is still linear. Using ReLU applies piece-wise linear tiling to help include non-linearities between layers.

Figure 19

```
def forward(self, feats_in):
    batch_size, e_, e_, s_ = feats_in.shape
    x = self.pool(self.conv1(feats_in.float()))
    x = self.pool(self.conv2(x))
    x = torch.flatten(x, start_dim=1)
    x = F.relu(self.fc1(x))
    x = F.log_softmax(self.fc2(x),dim=1)
    return x
```

Figure 20

```
Epoch 14 -----------------------------------------------------
Step 0 | Training Loss: 3.247, Accuracy: 0.0%
Step 50 | Training Loss: 3.255, Accuracy: 10.0%
Step 100 | Training Loss: 3.232, Accuracy: 0.0%
Step 150 | Training Loss: 3.32, Accuracy: 10.0%
Step 200 | Training Loss: 3.258, Accuracy: 0.0%
Step 250 | Training Loss: 3.208, Accuracy: 0.0%
Average Accuracy: 4.295%
-----------------------------------------------------------
Testing Accuracy: 3.13%
Accuracy of NN for ALL test set: 3.13%
Accuracy of NN for PIE test set: 3.137%
Accuracy of NN for selfies test set: 0.0%
```

2. Reducing & Increasing batch sizes

In the above NN architectures, train batch size (bs) was set at 10. I tried two other runs where bs=5 and bs=50, which returned classification accuracies of 97.731% and 96.792% respectively. In this set up, it seems like bs=5 to 50 with 15 epochs were still suitable for training. The slight drops fluctuations from base case might be due to randomness. I also tried to increase bs further to 128 and increased the number of epochs to 300 (with early stopping rule be if the last epoch iteration could hit 99% accuracy) so that there were sufficient update opportunities. The accuracy was fell slightly to 95.696%

and exited at epoch=60. In general, for this exploration, a larger bs seems to return slightly worse performance, although the differences in outcomes might be due to some random factor.

3. Comments on reproducibility of accuracies

```
# for reproducibility
def set_seed(seed):
    torch.manual_seed(seed)
    torch.cuda.manual_seed_all(seed)
    torch.backends.cudnn.deterministic = True
    torch.backends.cudnn.benchmark = False
    np.random.seed(seed)
    random.seed(seed)
    os.environ['PYTHONHASHSEED'] = str(seed)
set_seed(123)
```

Figure 21

While building the model, there were a few runs where given the same model, the accuracy might be extremely low (<20%) and loss did not reduce over iterations. It seems that there were tendencies for the model to get stuck at some local minima. Therefore, one way is to do gradient clipping using a pre-built Pytorch function `clip_grad_value_` with clipping value set at 1, which was done for all the above cases. On top of this, I also set the random seeds to a fixed number (seed=123) for better reproducibility as shown in Figure 21.

To show the importance of gradient clipping and seeds, I reran the base case three times without gradient clipping and with three different seed numbers (123 (same as base), 234, 345). The corresponding test accuracies were 87.559%, 97.418%, 93.192%, suggesting some instability in results if gradient clipping and seeds were not set properly when one uses NN models.

C. Conclusion

In summary, this report analysed face images by performing feature extraction, visualisation, classification and clustering using techniques like PCA, LDA, GMM, SVM, and NNs.

For feature extraction, both PCA and LDA are viable, but the second utilises class information. Yet, the classification accuracy on test data for PCA was higher than LDA on average. In both cases, the greater the number of components/features, the better the overall classification accuracy images. This finding is similar to how we observe slightly better clustering results with GMM when more features (1024>200>80) were incorporated into the models.

However, the number of features did not affect the accuracy of the Linear SVM classification much. One possible reason for this finding is that pc=80 is still quite large and thus, probably able to capture the needed features for face recognition. A short extension to much lower n_pcs = 50, 30, and 10 shows that data dimensions that are too small significantly reduces accuracy.

The best face recognition classifier in this study was the linear SVM which was able to achieve 98.513% accuracy at best. This was followed by NN model, which was able to achieve 97.574% at best. I believe NN might be able to outdo SVM if further complexities, layers and hyperparameter tuning were implemented, but this is out of scope for this report. Nevertheless, SVMs were much easier to implement with less intricacies, and thus might be preferred in some situations (e.g. prototyping).

D. References

Scrucca, Luca. “Dimension Reduction for Model-Based Clustering.” Statistics and Computing, vol. 20, no. 4, 1 July 2009, pp. 471–484, 10.1007/s11222-009-9138-7. Accessed 12 Nov. 2019.