MLND Nanodegree – Face Recognition for My Family Members

I. Definition

A. Project Overview

The iCloud photos from iOS or Mac OS provided a comprehensive way to tag faces. All photos in the library will be scanned, thereafter faces will be identified and tagged accordingly based on the earlier defined tags/names.

I am a photography enthusiast and I have started to take photos from year 2004. As of today, it's over 200k shots and I've kept 40k of them. There are many desktop applications can do the job of face recognition, but it's going to be super fun if I can build the solution end to end.

Many other interesting use cases can be further built/extended by this, for example, auto greeting to the person sitting in front of the computer by calling his/her name. However, this won't be covered in this project.

B. Problem Statement

There are many great online blogs/projects discussed about the detailed mathematics and implementation of face recognition. For example, this one. I don't have plan to approach my problem in that way which may be too difficult and time consuming for me.

The first two questions came to my mind about this project were how to minimize the distraction factors in the photos and how to extract features understood by computers. All my photos are about something or somebody and I bet none of them is about a single face only. This project is about face recognition and all the other factors other than faces will be 'noise' and should be avoided before any machine learning kicking in. A 300 * 300 pixel color photo is quite good for human eye to distinguish faces. But it's a vector of size 300 * 300 * 3 = 270,000 which sounds too big to be machine learnt by today's computer. After some research and experiments I decided to use OpenCV to detect/extract faces and then used Google Inception V3 to extract features from the face photos.

C. Metrics

Without any doubt, accuracy is the most important metric for any machine learning problem and it still holds true for this project. A model is useless if it performs only slightly better than a random guess. Speed is another important metric especially when we deal with large scale of data. I don't have the intention to scale the model to that level but I will evaluate the performance based on speed as well. Meaning, both the training and testing time will be considered.

The expected accuracy for this project is at least 80%. Formulas of the mentioned metrics are as below.

Accuracy = Number of correctly predicted label / Total number of testing cases

Training Time = End time of training – start time of training

Testing Time = End time of testing – start time of testing

D. Workflow of the Approach

I. Pre-process and manual labelling.
   i. Scan all photos, detect faces and save them as new images with dimension of 299 * 299 pixels.
   ii. Hand pick eligible photos and save them into respective folders. There will be seven folders named me, wife, daughter, son, dad, mum, brother.

II. Apply Google Inception V3 model to extract the feature vectors of each face image.
III. Apply different models.
    i. Apply linear classifier.
    ii. Apply KNN.
    iii. Apply logistic regression.
    iv. Deep learning model.
IV. Performance metrics and benchmark. Linear classifier as the baseline.
V. Conclusion.

II. Analysis and Data Preparation

A. Data Exploration

All my photos are in D:\Pictures, majority of them are in both .jpg and .nef format. The .nef is a raw image format for Nikon cameras and .jpg is the copy after image post-processing of raw file.

The output of below code chunk shows the root directory of my photo library. For each photo its full address always follows D:\Pictures\[yyyy]\[yyyy.mm.dd] - [event name]\[yyyymmdd-hhmm][-index].jpg

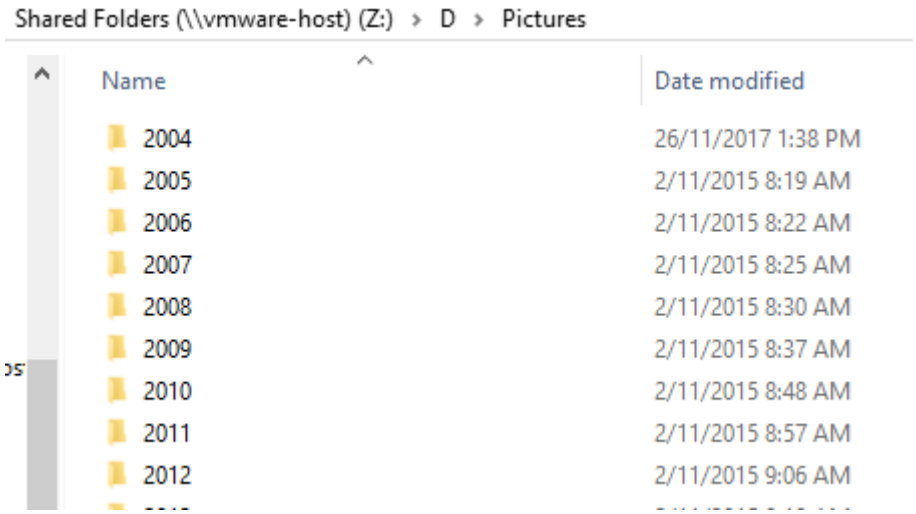


Figure 1. A glance of the folder structure

Each and every file use the time stamp as its file name. .nef is the raw image file, .xmp is generated by Adobe Lightroom, both are not applicable to this project. Only .jpg files are applicable to this project.

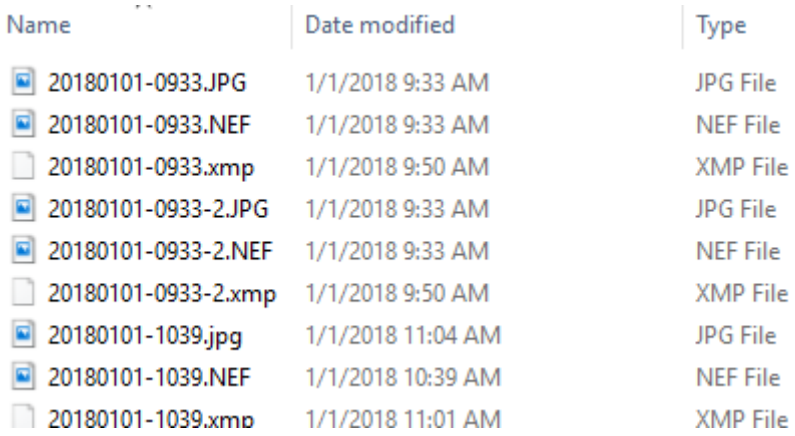


Figure 2. A glance of photo files

B. Sample Photo Visualization and Pre-processing

The photo file will be read from the disk as numpy array and then displayed. In this project and my particular case, some special cases were met during the workflow as I have a lot of phots whose full paths contain Chinese characters. It's not a big challenge, just slightly different when dealing with all English characters. Below figure illustrates the process.

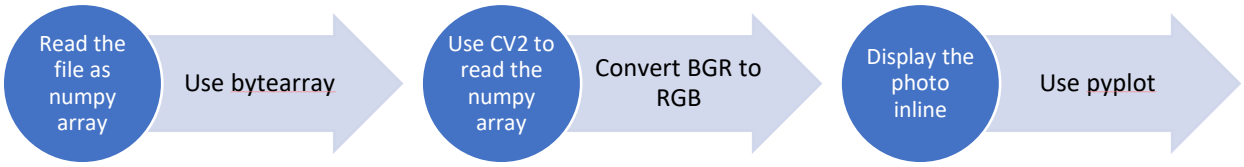


Figure 3. Process of reading and displaying a photo

And below is the result of a sample photo.

```
file_path = os.path.join(photo_dir, '2017', '2017.11.16 - Singapore Fintech Festival', '20171116-1923.jpg')
display_from_file(file_path)
```

```
Image numpy array shape: (4760, 7132, 3) <class 'numpy.ndarray'>
```

Sample Image

C. Detect Faces, Save as New Files

The objective of this project is for face recognition, it'll be time-consuming, or even non-sense if feeding the above entire photo to the machine learning models. OpenCV will be used in this project for face detection. And the detected faces will be resized to 299 * 299 pixels and saved as new files in a different directory for later machine learning pipeline.

Below figures illustrate that 5 faces detected and saved as new files.

```
D:\Google Drive\Study\Machine Learning Engineer Nanodegree - Udacity\Projects\MLND-Projects\projects\facial_recognition_family_
members\images\2017\2017.11.16 - Singapore Fintech Festival\20171116-1923.jpg-face-3.jpg saved.
D:\Google Drive\Study\Machine Learning Engineer Nanodegree - Udacity\Projects\MLND-Projects\projects\facial_recognition_family_
members\images\2017\2017.11.16 - Singapore Fintech Festival\20171116-1923.jpg-face-4.jpg saved.
```

Figure 4. Faces detected and saved as new files

```
Image numpy array shape: (299, 299, 3) <class 'numpy.ndarray'>
Image numpy array shape: (299, 299, 3) <class 'numpy.ndarray'>
Image numpy array shape: (299, 299, 3) <class 'numpy.ndarray'>
Image numpy array shape: (299, 299, 3) <class 'numpy.ndarray'>
Image numpy array shape: (299, 299, 3) <class 'numpy.ndarray'>
```

Figure 5. Files can be read and with expected size

D. Batch Process All Photos and Manual Labelling

Once one photo can be processed like what's been done above, it's just a matter of time to batch process all the 40k photos, indeed it took 10 hours.

After that, I got 100k face photos. Due to the low accuracy of OpenCV face detection. Many of them are not faces. I spent about 2 hours to hand pick about 500 photos and below are the distributions for the 7 categories.

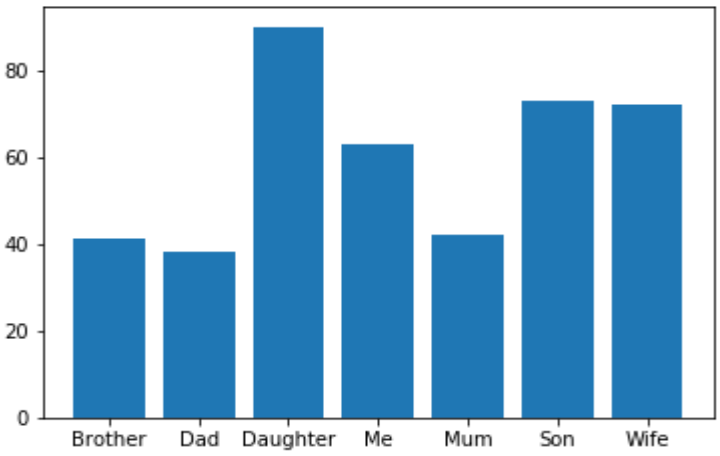


Figure 6. Distribution of the hand-picked photos

From this point of time, the dataset for machine learning is finally ready.

III. Methodology and Implementation

A. Algorithms and Techniques

The input image is with size of 299 * 299 pixels, 3 channels for a color pixel, that's a space with almost 270k dimensions. It's too big to be processed directly by the machine learning algorithms. Hence, Google Inception V3 will be applied first to extract the feature vectors with dimensionality of 2048. Maybe it's still too big in this project but we'll see.

A linear classifier will be applied first. Its performance will be used as the baseline of the benchmark model. Subsequently, KNN, Logistic Regression and Deep Learning models will be applied and measured against the linear classifier.

As shown in the earlier section, around 500 images were chosen and number of samples for each category is not so balanced. Despite Google Inception V3 helped eliminate the necessity of having many samples for image recognition, it's still possible that my dataset is too small. So, if it's needed, techniques like using image generator will be explored in this project.

B. Extract Feature Vectors

Below is the entire structure of Google Inception V3 model. The target layer I want to extract is the average pooling layer as indicated.

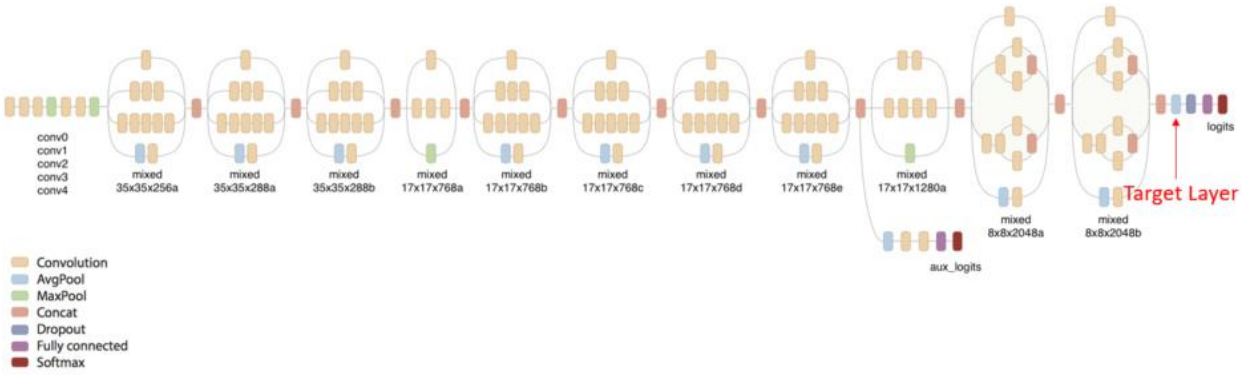


Figure 7. Google Inception V3

Based on the below model summary screenshot, the output of this layer is a vector with size 2048. And that vector is going to be the feature input of the machine learning algorithms in this project.

```
__________________________________________________________________________________________________
mixed10 (Concatenate)           (None, None, None, 20 0           activation_86[0][0]
                                                                  mixed9_1[0][0]
                                                                  concatenate_2[0][0]
                                                                  activation_94[0][0]
__________________________________________________________________________________________________
avg_pool (GlobalAveragePooling2D (None, 2048)         0           mixed10[0][0]
__________________________________________________________________________________________________
predictions (Dense)             (None, 1000)          2049000     avg_pool[0][0]
==================================================================================================
Total params: 23,851,784
Trainable params: 23,817,352
Non-trainable params: 34,432
```

Figure 8. Google Inception V3 summary

The Inception model has some requirements of input data feed into it. The data has to be in 4D tensor shape, value need to be normalized from [0, 255] to [0, 1.0] and below shows the workflow. Eventually a 4D numpy array with shape (x, 299, 299, 3) is obtained.

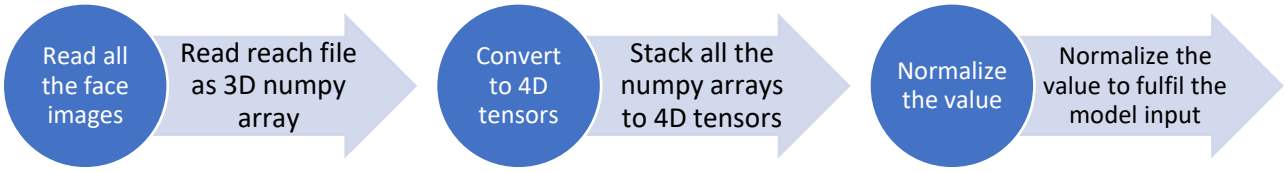


Figure 9. From images to 4D tensors

C. Split the Dataset

Due to the slightly imbalance of the data for each category, I used stratified split in this project. 70% for testing, 15% for validation and 15% for testing.

D. First Attempt for Each Model

Four models were explored, they are SGD Classifier, KNN, Logistic Regression and DNN. The first 3 models I used default parameters and I used 3 dense layers for the DNN model.

The number of epochs was set to 50, based on the loss and accuracy of the validation dataset in the DNN model.

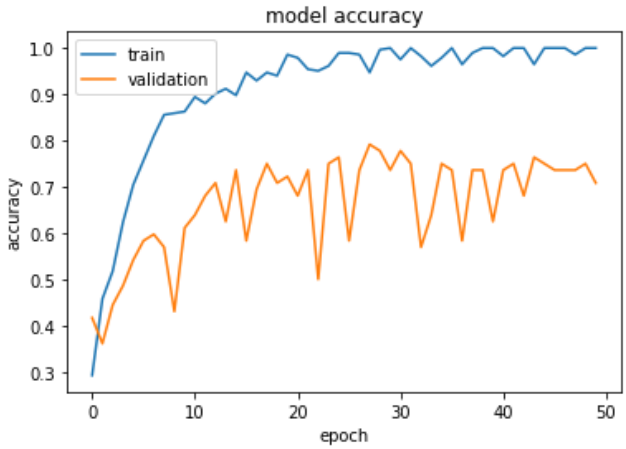


Figure 10. DNN model accuracy

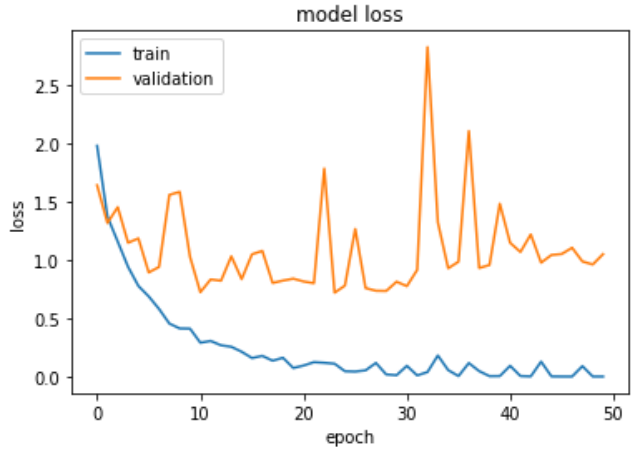


Figure 11. DNN model loss

Below table is the summary of accuracy and time taken.

Model	**Training time (s)**	**Testing time (s)**	**Accuracy**
SGD Classifier	0.036	0.035	78%
KNN	0.018	0.054	68%
Logistic Regression	0.518	0.0	79%
DNN	6.16	0.005	76%

Based on the current split of the dataset, logistic regression performed the best. KNN reached merely 70% while SGD classifier, logistic regression and DNN reached accuracy of 75% to 80%. DNN took significant longer time on training.

But what if it's just a coincidence, what if I had more images? The next section will look into the refinement of the models from two ways. The first one is to use image generator to have a bigger dataset. The other one will focus on using cross validation to determine the best model.

IV. Refinement

A. Image Generator

Use image generator to create more images. This is a very useful technique when having little data. In this project, I put the new images in a new folder and 10 new images will be generated for each original image.

Below is a peak on what's been generated. The images are distorted, rotated, filled with near pixels and etcs.

```
image_numpys = get_numpy_from_folder('sample_generated')
display_from_numpy(image_numpys, fig_dim_x=15, fig_dim_y=5, plot_nrows=1, plot_ncols=5)
```

```
Image numpy array shape: (299, 299, 3) <class 'numpy.ndarray'>
Image numpy array shape: (299, 299, 3) <class 'numpy.ndarray'>
Image numpy array shape: (299, 299, 3) <class 'numpy.ndarray'>
Image numpy array shape: (299, 299, 3) <class 'numpy.ndarray'>
Image numpy array shape: (299, 299, 3) <class 'numpy.ndarray'>
```

Figure 12. Generated images

B. K-fold Cross Validation

Even though splitting the dataset into training and testing is random, there is still a chance that this specific split favoured one model over the other. So, k-fold cross validation will be employed in this project.

I will choose k = 10. The original dataset of 419 image will be split into 10 folds. Each of the training and evaluation hold 1 fold as the testing dataset and use the other 9 folds to generate the new training images. The generated new image from the 9 testing folds will then be used to train the 4 models mentioned earlier. Performance on accuracy and testing time will be measure against the 1 original untouched testing fold.

After a long time of waiting, below is the boxplot of the 3 metrics for the 4 models.

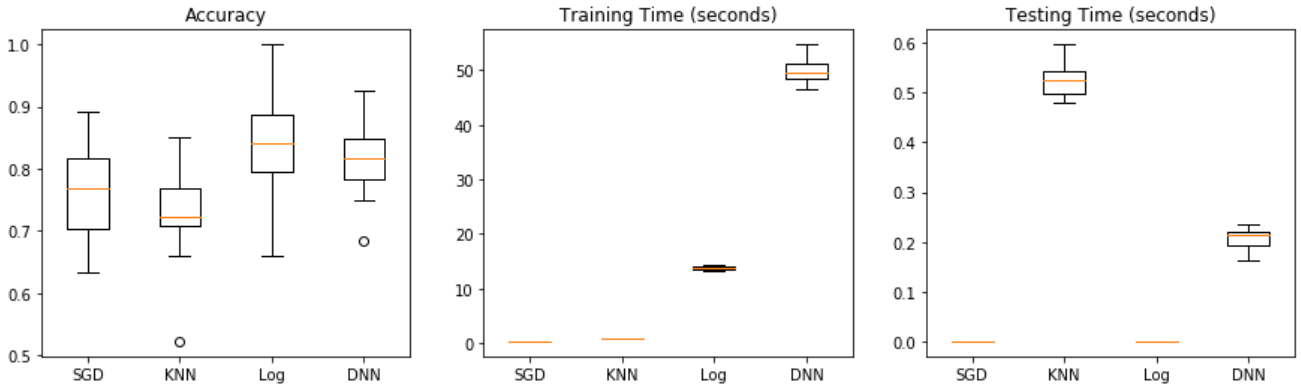


Figure 13. K-fold CV

Logistic regression produced the highest average accuracy of 85%. SGD classifier and KNN are fast one training but KNN took the longest time on testing. DNN's accuracy is the 2nd highest but it's training and testing time are significant longer.

V. Benchmark and Conclusion

Among all the 4 models, logistic regression is the one giving highest accuracy and also acceptable training and testing time.

The accuracy of logistic regression model is 5% to 10% higher than the other 3 models. The training time is longer than SGD classifier and KNN, but not as much longer as DNN. The testing time is very fast, and actually it's the fastest one.

KNN doesn't perform so well in terms of accuracy. I guess it's due to the curse of dimensionality.

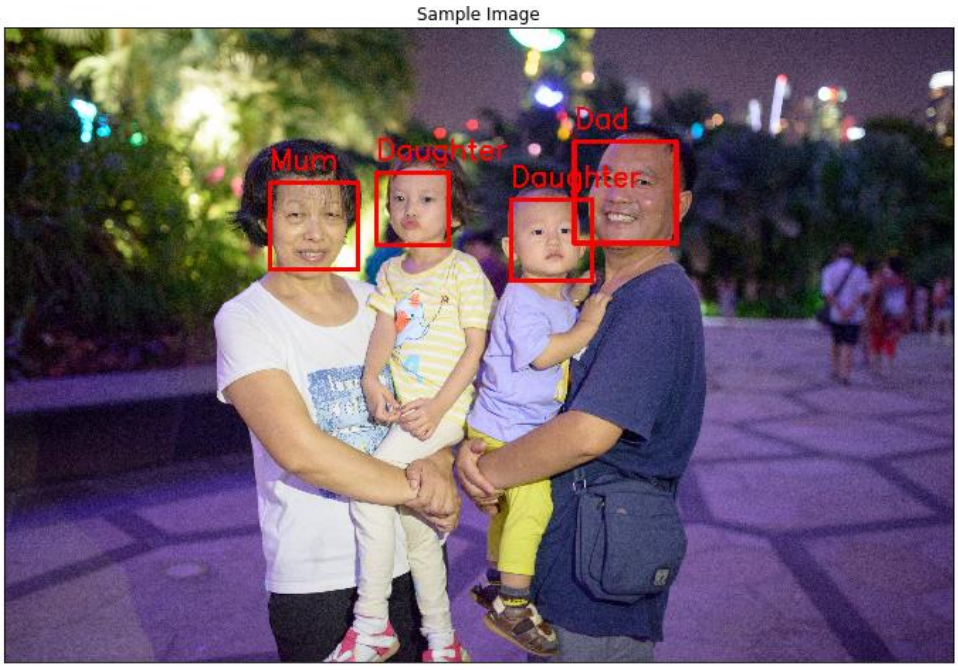
The DNN model in this project is exactly a transfer learning approach. Its accuracy is quite good, but it takes a long time to train and it's also costly. In this project I'm using GTX 1070 Ti, a GPU costs around USD 500. If the training is under the same CPU environment, I won't doubt it'll take at least 10 more times of current time.

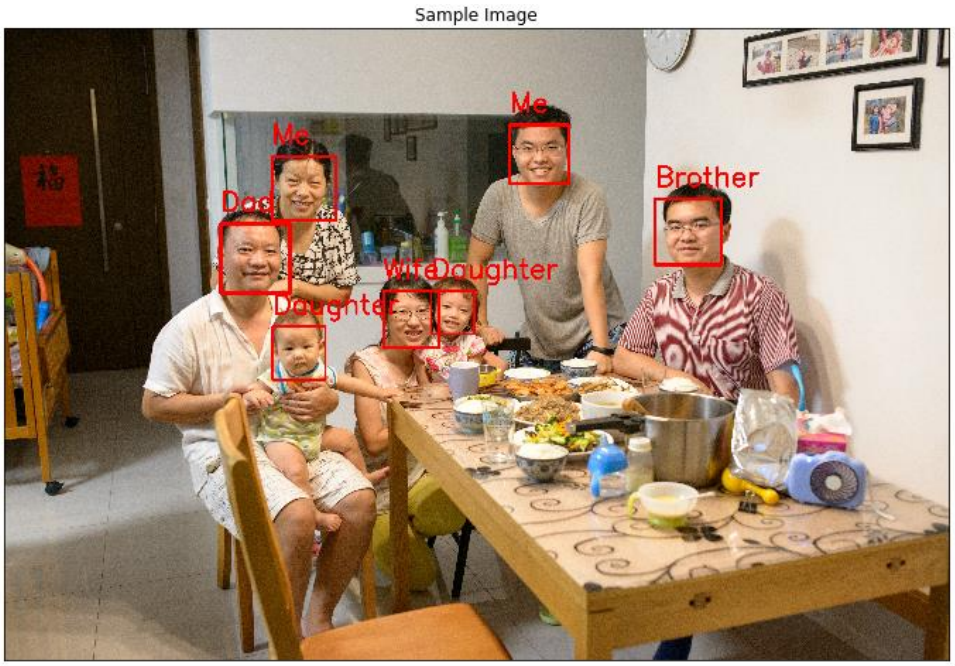
The accuracy of 76% from SGD classifier is the base of the benchmark. Logistic regression model has an average accuracy of almost 85% which pushed the boundary by almost 10%. The only cost is that logistic regression model takes longer time to train.

The final model doesn’t show a perfect result on the accuracy, especially there are only 7 categories in this project. I doubt it’ll work well in the real-world scenario that people need to perform face recognition among thousands to millions of faces. However, it still shows the effectiveness of feature engineering by leveraging transfer learning and applying ‘traditional’ machine learning models.

One possible way to improve the model may be using transfer learning to detect the facial key points (centre and corners of eyes, mouth, nose and etc.) first. Then build mathematic models to represent the relative positions of the key points. Normalization may be needed to make the face centred and ‘starring’ at the camera. Then the recognition problem will become to compute similarity of the key points map.

VI. Fun Part





The model showed 85% accuracy in the earlier section. I have chosen 2 photos that never used for training or testing to evaluate the performance of the model in the real-world scenario. The accuracy is not as good as expected, especially the 2nd image has 2 misclassified faces. Well, perfect accuracy is not the purpose for this project. The most important thing is, it's fun and it works!