The Capstone Project of MLND - 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 (https://medium.com/@ageitgey/machine-learning-is-fun-part-4-modern-face-recognition-with-deep-learning-c3cffc121d78)](https://medium.com/@ageitgey/machine-learning-is-fun-part-4-modern-face-recognition-with-deep-learning-c3cffc121d78). 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 use Google Inception V3 to extract features from the face photos. Google Inception V3 is a very popular deep learning model (convolutional neural network) for image recognition. By extracting its pooling layer, the feature space of an image will be drastically reduced from vector size of 270,000 to 2,048.

Given the context it's face recognition for my family members and that makes this project to solve a multi-class classification problem.

C. Metrics

In this project both accuracy and speed will be looked into when evaluating the performance of the models. 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.

The metric of time is quite intuitive, both the training and testing time will be considered. The accuracy will be slightly more complicated. In the classification problem, we can look at precision, recall and f-beta score. The `precision_recall_fscore_support` from `sklearn` can do all this in one go. I used to have great difficulties to distinguish precision to recall until I found a way to describe them in two sentences under the context that God is the ground truth. Precision measures how much God agrees with me when I say it's positive. Recall measures how good I can find it's positive for whatever God says it's positive. Below are the formulas for precision and recall rates.

1. *precision = Number of True Positive / (Number of True Positive + Number of False Positive)*
2. *recall = Number of True Positive / (Number of True Positive + Number of False Negative)*

In this project, I need to know when the model predicts label A how much of the predicted labels are truly label A, this is the precision. On the other hand, I also want to know for all instances with label A how much of them can be picked up by the model in such a way that the model predicts label A, this is recall. And f-beta is a metric to evaluate both precision and recall rates. I think both precision and recall are equally important hence the beta here will be set to 1.0 and f-beta eventually becomes f1 score with below formula.

*f1 = 2 * (precision * recall) / (precision + recall)*

Therefore, accuracy of the model will be based on the f1 score. For a better intuition, confusion matrix will also be used to have a better visual feeling on the accuracy.

D. Workflow of the Approach

1. Analysis and data preparation. A brief experiment to display sample image, explanation on the techniques and algorithms will be used.
2. Implementation.
   A. Scan all photos, detect faces and save them as new images with dimension of 299 * 299 pixels.
   B. Hand pick eligible photos and save them into respective folders. There will be seven folders named `me`, `wife`, `daughter`, `son`, `dad`, `mum`, `brother`.
   C. Apply Google Inception V3 model to extract the feature vectors of each face image.
   D. Initial try on applying the different machine models.
3. Refinement.
   A. Use image generator to generate more images.
   B. Apply K-fold cross validation.
4. Performance metrics and benchmark. Linear classifier as the baseline. Evaluate the accuracy (F1 score) and time spent for both training and testing.
5. 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]`

In [1]:
```
import os
import time
cur_dir = os.getcwd()
#print(cur_dir)
target_image_dir = os.path.join(cur_dir, 'images')
photo_dir = 'D:\Pictures'
os.listdir(photo_dir)
```

Out[1]:
```
['.SynologyWorkingDirectory',
 '2004',
 '2005',
 '2006',
 '2007',
 '2008',
 '2009',
 '2010',
 '2011',
 '2012',
 '2013',
 '2014',
 '2015',
 '2016',
 '2017',
 '2018',
 'Adobe Lightroom',
 'Camera Roll',
 'desktop.ini',
 'iCloud Photos',
 'Phone Photos',
 'Photography Works',
 'Readme.txt',
 'Saved Pictures',
 'SyncToy_6288703e-b478-4b80-9f48-ece12cdcb521.dat',
 'zbingjie',
 'zothers',
 '法蝶',
 '熊思宇和黄乐论辩论']
```

A glance of a recent photo directory. 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.

In [2]:
```
os.listdir(os.path.join(photo_dir, '2018'))
```

Out[2]:
```
['2018.01.01 - 冰洁爸妈证件照',
 '2018.01.01 - 彤彤和奕明',
 '2018.01.01 - 彤彤和祺祺',
 '2018.01.24 - Changi Business Park Airbus A380',
 '2018.01.26 - Changi Business Park, A350, A320, B777, A380',
 '2018.01.26 - 冰洁，祺祺',
 '2018.01.27 - Marina Bay',
 '2018.01.27 - 冰洁，彤彤祺祺',
 '2018.01.27 - 祺祺在游泳，哭',
 '2018.01.28 - 彤彤，祺祺和姥爷']
```

In [3]:
```
os.listdir(os.path.join(photo_dir, '2018', '2018.01.01 - 彤彤和祺祺'))
```

Out[3]:
```
['20180101-0933-2.JPG',
 '20180101-0933-2.NEF',
 '20180101-0933-2.xmp',
 '20180101-0933.JPG',
 '20180101-0933.NEF',
 '20180101-0933.xmp',
 '20180101-1039.jpg',
 '20180101-1039.NEF',
 '20180101-1039.xmp',
 '20180101-1042-2.jpg',
 '20180101-1042-2.NEF',
 '20180101-1042-2.xmp',
 '20180101-1042.jpg',
 '20180101-1042.NEF',
 '20180101-1042.xmp',
 '20180101-1043-10.jpg',
 '20180101-1043-10.NEF',
 '20180101-1043-10.xmp',
 '20180101-1043-11.jpg',
 '20180101-1043-11.NEF',
 '20180101-1043-11.xmp',
 '20180101-1043-12.jpg',
 '20180101-1043-12.NEF',
 '20180101-1043-12.xmp',
 '20180101-1043-7.jpg',
 '20180101-1043-7.NEF',
 '20180101-1043-7.xmp',
 '20180101-1043-8.jpg',
 '20180101-1043-8.NEF',
 '20180101-1043-8.xmp',
 '20180101-1043-9.jpg',
 '20180101-1043-9.NEF',
 '20180101-1043-9.xmp',
 '20180101-1043.jpg',
 '20180101-1043.NEF',
 '20180101-1043.xmp',
 '20180101-1044-2.jpg',
 '20180101-1044-2.NEF',
 '20180101-1044-2.xmp',
 '20180101-1044.jpg',
 '20180101-1044.NEF',
 '20180101-1044.xmp',
 '20180101-1045-2.jpg',
 '20180101-1045-2.NEF',
 '20180101-1045-2.xmp',
 '20180101-1045.jpg',
 '20180101-1045.NEF',
 '20180101-1045.xmp']
```

All the photo files are in folders whose names are in year format. There are slightly more than 40k photos, below function will dump the full paths of them. 5 photos were randomly picked to show the full address.

In [4]:
```
def get_all_file_path(folder_addr):
    """Return all jpg files' full path as a list
    Args:
        folder_add (str): The folder address.
    Returns:
        all_jpg (list): A list of strings which are the full path of jpg files.
    """
    all_jpg = []
    for root, dirs, files in os.walk(folder_addr):
        # All the target photos are in D:\Pictures\20xx. Get the jpgs from them only.
        path = root.split(os.sep)
        if len(path) < 3:
            continue
        else:
            year = path[2]
            if year[:2] != '20':
                continue
        #print((len(path) - 1) * '---', os.path.basename(root))
        for file in files:
            if file[-3:].lower() == 'jpg':
                #print(len(path) * '---', file)
                all_jpg.append(os.path.join(root, file))
    return all_jpg

all_jpg = get_all_file_path(photo_dir)
```

In [5]:
```
import random
print('Number of jpgs:', len(all_jpg))
for i in random.sample(range(len(all_jpg)), 5):
    print(all_jpg[i])
```

```
Number of jpgs: 40783
D:\Pictures\2008\2008.05.30~2008.06.01 - 表妹六一表演\20080530-1058-54.JPG
D:\Pictures\2007\2007.03.03 - 9th SM2合影\20070303-1348-7.JPG
D:\Pictures\2006\2006.05.28 - 和家人在木兰天池\20060528-1110.JPG
D:\Pictures\2010\2010.07.15 - OCBC TC1 4th floor\20100715-1305-26.jpg
D:\Pictures\2007\2007.01.22 - PRCSU main com成员\20070122-2210-18.JPG
```

B. Sample Photo Visualization

This section demonstrates how to read and display an image file.

In [5]:
```
# Import required libraries for this section
%matplotlib inline
import numpy as np
import matplotlib.pyplot as plt
import math
import cv2
DISPLAY = True
SAVE = True
NODISPLAY = False
NOSAVE = False
```

In [6]:
```
def get_numpy_from_file(file_path):
    """Return the image file as a numpy array
    Args:
        file_path (str): The full address of the image file.
    Returns:
        image (numpy.ndarray): The numpy array of the image file. Shape of (width, height, number of channels).

    The file_path may contain unicode characters, cannot use cv2.imread() directly. Below way works for both
    unicode and non-unicode paths.
    """
    file_stream = open(file_path, 'rb')
    bytes_arr = bytearray(file_stream.read())
    numpy_ar = np.asarray(bytes_arr, dtype=np.uint8)
    image = cv2.imdecode(numpy_ar, cv2.IMREAD_UNCHANGED)
    print('Image numpy array shape:', image.shape, type(image))
    return image

def display_from_numpy(image, fig_dim_x=10, fig_dim_y=10, plot_nrows=1, plot_ncols=1):
    """Display the image from a given numpy array (the returned result from get_numpy_from_file() function).
    Args:
        image (numpy.ndarray): The numpy array representation of an image.
        image (list[numpy.ndarr]): The list of numpy arrays of images.
    Returns:
        inline display of the image.
    """
    fig = plt.figure(figsize=(fig_dim_x, fig_dim_y))
    if isinstance(image, list):
        image_list = image
    else:
        image_list = []
        image_list.append(image)
    for i in range(len(image_list)):
        ax = fig.add_subplot(plot_nrows, plot_ncols, i+1, xticks=[], yticks=[])
        ax.set_title('Sample Image')
        ax.imshow(image_list[i])

def display_from_file(file_path):
    """Display one image file inline.
    Args:
        file_path (str): The full address of the file.
    Returns:
        None: Display image inline.
    """
    image = get_numpy_from_file(file_path)
    # Need to convert to RGB
    image = cv2.cvtColor(image, cv2.COLOR_BGR2RGB)
    display_from_numpy(image)
```

In [7]:
```
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'>
```

C. Algorithms and Techniques

a). Face Detector

In this project, I used Haar Cascades Classifier (https://docs.opencv.org/trunk/d7/d8b/tutorial_py_face_detection.html) from OpenCV for face detection.

This classifier is pre-trained to determine whether a region of interest (say size 20 *20) from the image is a face or not. So in order to find all the faces from the image, this detection needs to be repeated for all possible regions in the image. That's going to be a huge amount of computations. This classification for each region will be gone through by stages. If at an earlier stage 10 features are used to determine whether the region is a face or not, then classification will only continue if the result at this stage is positive. In this way, time won't be wasted to check the remaining features.

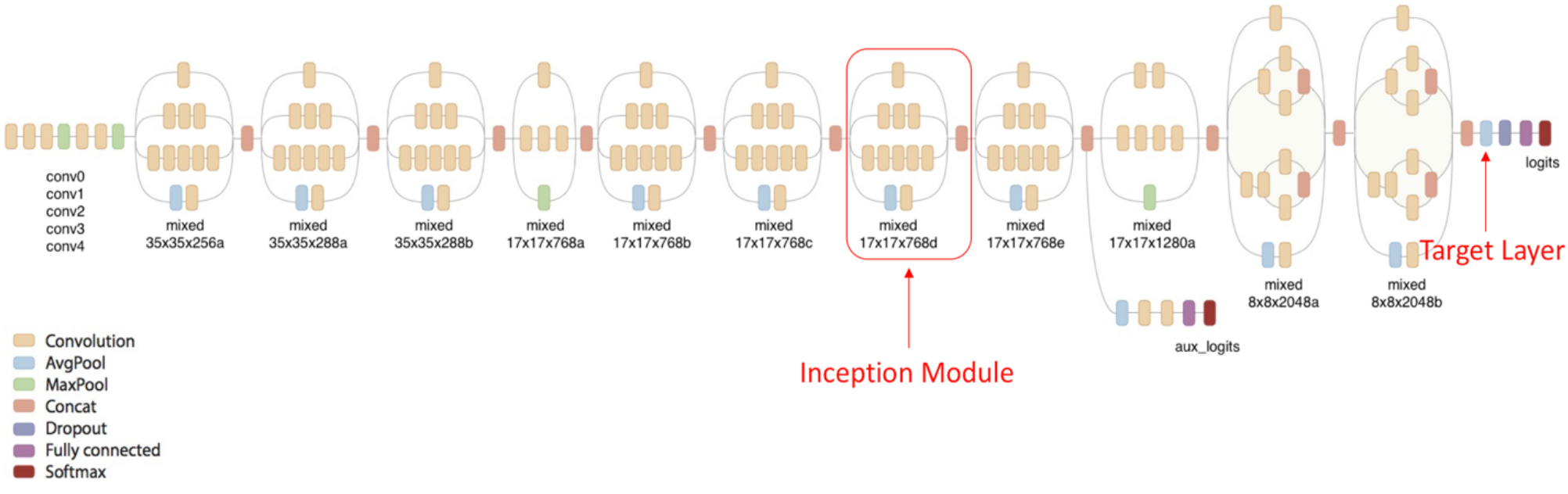
Then the image will be resized by a predefined scale to perform the detection one more time and this is repeated until the image is too small. This scaling factor is controlled by parameter `scaleFactor` . Its value should be slightly more than 1.

There is another parameter `minNeighbors` controlling how 'sensitive' the classifier is. The face detection is performed against different scales of the same image. A true face has a larger possibility of been detected as a 'face' at different scales for example 1.0, 0.9, 0.8, maybe not for scale 0.7 while a non-face object may be detected as 'face' at scale 1.0 only. The value of `minNeighbors` should be a value bigger than 3 or so.

It's difficult to have best values for the mentioned parameters. Haar Cascades Classifier is fast, but both the precision and recall rates are not perfect. Luckily I have a large raw dataset and I don't think it's a concern in this project. After few tries, I set the `scaleFactor` to 1.3 and `minNeighbors` to 5.

b). Feature Extractor

Below is the entire structure of Google Inception V3 model. The small yellow squares are the most common blocks in this model. They are convolution layers that extract the visual features from the images. The block surrounded by read line in below figure is a typical inception module. One such block is ending with a concat layer which is basically concatenating the output from different parallel layers to a huge matrix. In the highlighted inception module, 17 * 17 * 768 is the matrix dimension. Extracting features may not always needed to be from the final inception module. However, this is a very deep network, and the reason it goes so deep is because it produces better results. Hence, I will extract the features from the last inception module. The output shape is 8 * 8 * 2048. By flatting it, it’ll still be a big vector. So the target layer I want to extract features from is the average pooling layer as indicated, which produced a vector with size of 2048.



c). Machine Learning Models

Four models will be evaluated. SGD Classifier, K-nearest Neighbour, Logistic Regression and Deep Learning models. The winning one will be gone through the process of grid search cross validation to find the best hyper-parameters and hopefully it can push the accuracy even further.

1. SGD Classifier.

   Both SGD Classifier and Deep Learning using Stochastic Gradient Descent optimizer work quite similar as the 'SGD' way. When one or a batch of samples feeding into the model, a loss (the distance between output and expected output) will be computed. Then this loss value will be used to update the parameters/weights of the model to let it lean towards the expected output.

   The SGD classifier prefers data been preprocessed to be normalized with zero mean and unit variance. In this report, the preprocessed data will be normalized from [0, 255] to [0, 1]. Not exactly as what's favored by SGD Classifier, but I standardize it to [0, 1] for all the models here. Default hyper parameters will be used.
2. K-nearest Neighbour.

   Instead of constructing a generalized model, KNN is storing all the training data. When performing the prediction, distance (eg, euclidean distance) from the testing data point to all the training data point will computed. Then based on the defined K value, using majority vote to decide the class of the testing data point. After a few tries, I set K=7.

   KNN is simple but it can be very time consuming during testing as distance to all training data are needed. In addition, due to the curse of high dimensionality, it may not be effective always.
3. Logistic Regression.

   Despite the 'regression' word in the name, it's indeed a classification model. Logistic regression takes the features into a logit function with output value in the range of [0, 1], which quantifies the probability of correctly predicting the class. When all the training points fed into the model, we get the average of the probability and that's the likelihood the training data is correctly predicted. So the objective in logistic regression training is to maximize this likelihood. There are many different algorithms, e.g. Newton-Raphson, Iteratively re-weighted least squares and etcs.

   Logistic regression is widely used industrial widely. But it doesn't perform well when the feature space is large and I have no idea whether it'll do a great job in this project. Default hyper parameters will be used.
4. Deep Learning.

   Without a doubt, deep learning has gained great exposure and evolvement over the past few years, especially in the domains of computer vision, natural language processing. In this project, the feature extractor is taking almost 95% of the Google Inception V3 layers. Instead of taking the remaining 5% layers, which will do a job to classify objects into 1000 classes, I will add another few dense layers to do the customization to suit my purpose of classifying objects into 7 classes. Essentially, this is just a transfer learning approach.

   Deep learning takes a lot of computation resources and time to do the gradient descent to find the best weights of the neurons. And in this project, I don't need to worry about the convolutional layers except the last few dense layers added by me.

d). Creating More Data

Despite I have a large photo library, there is still a concern on the quantity of the images with good quality. If each class only has an average of 5 images, probably I won't have any result as expected. So if it's needed, techniques like using [image generator (https://blog.keras.io/building-powerful-image-classification-models-using-very-little-data.html)](https://blog.keras.io/building-powerful-image-classification-models-using-very-little-data.html) will be explored in this project.

e). Stratified K-fold CV

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

f). GridSearchCV

When the best model is chosen, there is still some space to push the boundary of accuracy or speed. GridSearchCV will be used to fine tune the hyper-parameters and hopefully better result can be achieved for either accuracy or speed.

D. Benchmark

When I realized I was trying to solve a multiclass classification problem, the first intuition came to my mind is whether the data is linear separable. Therefore, the performance of a linear classifier will be set as the base of the benchmark model. Usually a linear classifier is fast and easy to train. If there is another more complicated model takes more time to train or test, it ought to perform better on accuracy (F1 score) to compensate the larger cost on time. If not, I'd rather choose the simple linear model.

III. Implementation

A1. Detect Faces, Save as New Files - Single Image

There are many supporting functions and `process_faces()` is the one with many flags to represent whether need to display or save the given image file.

In [8]:
```
# pathlib available from python 3.5
from pathlib import Path
def get_faces(image, scaleFactor, minNeighb):
    """Perform face detection and return the detected faces as a list of (x,y,w,h).
    Args:
        image (numpy.ndarray): The numpay array of an image.
        scaleFactor (float): The scaling factor to be used by the detectMultiScale() function.
        minNeighb (int): The number of minimum neighbors to be used by the detectMultiScale() function.
    Returns:
        faces (list of tuples): The list of the face locations.
    """
    # Convert to RGB then to grayscale
    image = np.copy(image)
    image = cv2.cvtColor(image, cv2.COLOR_BGR2RGB)
    gray = cv2.cvtColor(image, cv2.COLOR_RGB2GRAY)
    # Extract the pre-trained face detector from an xml file
    face_cascade = cv2.CascadeClassifier('detector_architectures/haarcascade_frontalface_default.xml')
    # Detect the faces in image
    faces = face_cascade.detectMultiScale(gray, scaleFactor, minNeighb)
    return faces

def draw_bounding_box(image, faces):
    """Draw the bounding box of faces on the image.
    Args:
        image (np.ndarray): Numpy array of the image.
        faces (list of tuples): The list of the face locations.
    Return:
        image_with_detections (np.ndarray): A image with bounding box on faces, in numpy array format,
            after converting to RGB.
        image_faces (list[np.ndarray]): List of face images.
    """
    # Use np.copy() to create duplicate images to avoid aleration of the original image.
    image_copy = np.copy(image)
    image_with_detections = np.copy(image)
    image_copy = cv2.cvtColor(image_copy, cv2.COLOR_BGR2RGB)
    image_with_detections = cv2.cvtColor(image_with_detections, cv2.COLOR_BGR2RGB)
    # The list of detected faces
    image_faces = []
    # Get the bounding box for each detected face
    for (x,y,w,h) in faces:
        # Add a red bounding box to the detections image
        if w > 200:
            line_width = w//20
        else:
            line_width = 3
        image_faces.append(image_copy[y:(y+h), x:(x+w)])
        cv2.rectangle(image_with_detections, (x,y), (x+w,y+h), (255,0,0), line_width)
    return image_with_detections, image_faces

def create_get_target_dir_file(file_path):
    """Create the target directory if it's not existent, return the target directory as string
    Args:
        None: Global variables.
    Returns:
        target_dir (str): The address of the target directory.
    """
    # Create the full path of the target images by replacing the photo_dir string into target_image_dir string.
    target_file = file_path.replace(photo_dir, target_image_dir)
    target_dir = os.path.dirname(target_file)
    target_path = Path(target_dir)

    # Create parents of directory, don't raise exception if the directory exists
    target_path.mkdir(parents=True, exist_ok=True)
    return target_dir, target_file

def save_faces(file_path, image_faces):
    """Save each face image into new files in target_iamge_dir.
    Args:
        file_path (str): The full path of the original photo.
        image_faces (list[np.ndarray]): The list of face images, in numpy array format.
    Returns:
        None
    """
    if len(image_faces) == 0:
        return

    target_dir, target_file = create_get_target_dir_file(file_path)

    # Resize and save each face image.
    for i, face in enumerate(image_faces):
        face = cv2.resize(face, (299, 299))
        os.chdir(target_dir)
        file_name = os.path.basename(target_file)
        cv2.imwrite(file_name + '-face-' + str(i) + '.jpg', cv2.cvtColor(face, cv2.COLOR_BGR2RGB))
        print(os.path.join(target_dir, file_name + '-face-' + str(i) + '.jpg'), 'saved.')

def process_faces(file_path, display=NODISPLAY, save=NOSAVE, scaleFactor=1.3, minNeighb=5):
    """Process the input image file by extracting face(s). Display and save based on the flags.
```

```
    Args:
        file_path (str): The full path of the input image file.
        display (bool): Default is NODISPLAY/False.
        save (bool): Default is NOSAVE/False.
        scaleFactor (float): The scaling factor used by face detection function.
        minNeighb (int): The number of minimum neighbors.
    Returns:
        None: Perform display or save actions based on the flags.
    """
    print('Image path', file_path)
    image = get_numpy_from_file(file_path)
    faces = get_faces(image, scaleFactor, minNeighb)
    print('Number of faces detected:', len(faces))

    image_with_detections, image_faces = draw_bounding_box(image, faces)

    if save:
        save_faces(file_path, image_faces)

    if display:
        # Display the image with the detections
        display_from_numpy(image_with_detections)

    # Return to this project's current working directory
    os.chdir(cur_dir)
```

In [10]:
```
# Load in color image for face detection
file_path = os.path.join(photo_dir, '2017\\2017.11.16 - Singapore Fintech Festival', '20171116-1923.jpg')
process_faces(file_path, DISPLAY, SAVE)
```

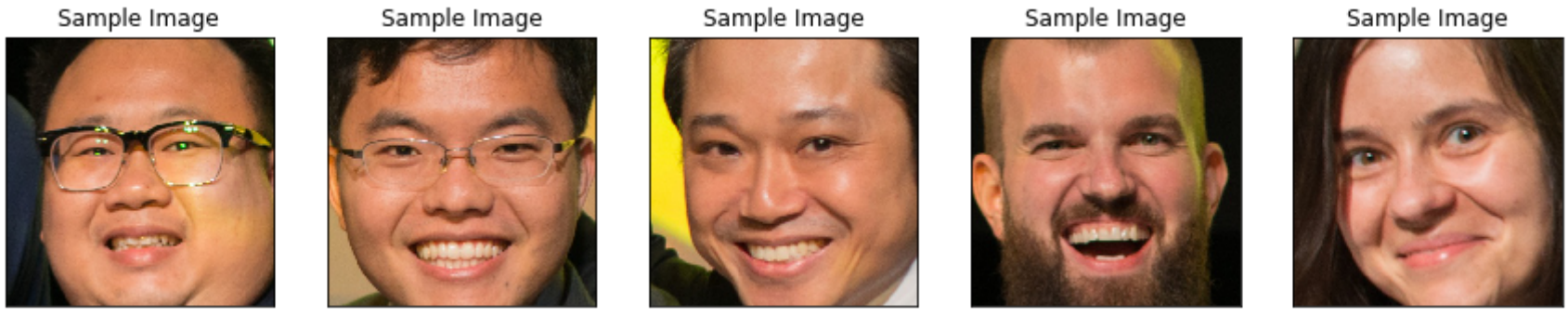
```
Image path D:\Pictures\2017\2017.11.16 - Singapore Fintech Festival\20171116-1923.jpg
Image numpy array shape: (4760, 7132, 3) <class 'numpy.ndarray'>
Number of faces detected: 5
D:\Google Drive\Study\Machine Learning Engineer Nanodegree - Udacity\Projects\MLND-Projects\projects\facial_recogniti
on_family_members\images\2017\2017.11.16 - Singapore Fintech Festival\20171116-1923.jpg-face-0.jpg saved.
D:\Google Drive\Study\Machine Learning Engineer Nanodegree - Udacity\Projects\MLND-Projects\projects\facial_recogniti
on_family_members\images\2017\2017.11.16 - Singapore Fintech Festival\20171116-1923.jpg-face-1.jpg saved.
D:\Google Drive\Study\Machine Learning Engineer Nanodegree - Udacity\Projects\MLND-Projects\projects\facial_recogniti
on_family_members\images\2017\2017.11.16 - Singapore Fintech Festival\20171116-1923.jpg-face-2.jpg saved.
D:\Google Drive\Study\Machine Learning Engineer Nanodegree - Udacity\Projects\MLND-Projects\projects\facial_recogniti
on_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_recogniti
on_family_members\images\2017\2017.11.16 - Singapore Fintech Festival\20171116-1923.jpg-face-4.jpg saved.
```

Read and display the extracted 5 faces from the above image.

In [9]:
```
def get_file_path_from_folder(folder_addr):
    """Return all jpg files' full paths as a list.
    Args:
        folder_add (str): The folder address.
    Returns:
        all_jpg (list): A list of strings which are the full path of jpg files.
    """
    all_jpg = []
    for root, dirs, files in os.walk(folder_addr):
        for file in files:
            if file[-3:].lower() == 'jpg':
                #print(len(path) * '---', file)
                all_jpg.append(os.path.join(root, file))
    return all_jpg

def get_numpy_from_folder(folder_addr):
    """Return all jpg files in a folder as numpy arrays.
    Args:
        folder_addr (str): The folder address.
    Returns:
        image_numpys (list[numpy.ndarrays]): The list of numpy arrays of jpg images in a folder.
    """
    all_jpg_addr = get_file_path_from_folder(folder_addr)
    image_numpys = []
    for jpg_addr in all_jpg_addr:
        image = get_numpy_from_file(jpg_addr)
        image = cv2.cvtColor(image, cv2.COLOR_BGR2RGB)
        image_numpys.append(image)
    return image_numpys
```

In [12]:
```
image_numpys = get_numpy_from_folder(os.path.join(cur_dir, 'images\\2017\\2017.11.16 - Singapore Fintech Festival'))
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'>
```



A2. Detect Faces, Save as New Files - All Images

In the earlier section, all the 40k image full pathes were store in `all_jpg` variable. So just need to be patient to wait for the results, it'll take quite several hours to finish.

In [ ]:
```
############################
### RUN WITH CAUSION#########
############################

# Scan through all 40k photos and extract faces
start = time.time()
for i in range(len(all_jpg)):
    process_faces(all_jpg[i], NODISPLAY, SAVE)
end = time.time()
```

In [13]:
```
def get_count_files(folder_addr):
    n = 0
    for root, dirs, files in os.walk(folder_addr):
        for file in files:
            n = n + 1
    return n
```

In [98]:
```
print('Extracting faces took %s hours' % round((end-start)/3600, 1))
print(get_count_files('images'), 'face images extracted')
```

```
Extracting faces took 13.2 hours
98059 face images extracted
```

About 13 hours has been taken and around 100k face images were generated. Below sample images are from one folder. It's quite clear that the openCV is not 100% reliable, some of the photos are not face images.

In [97]:
```
image_numpys = get_numpy_from_folder(os.path.join('images', '2017', '2017.10.25 - 祺祺吃饭'))
display_from_numpy(image_numpys, fig_dim_x=15, fig_dim_y=10, plot_nrows=3, plot_ncols=6)
```

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

B1. Preprocess - Manually Label the Face Images by Putting Them Into Different Folders

There are almost 100k faces extracted. Most of them are ir-relevant, and many of them are objects other than faces. I hand-picked about 500 of the face images and saved them into 7 categories/folders.

In [10]:
```
categories = os.listdir('./images')
face_jpgs = get_file_path_from_folder('./images/')
print('Categories:', categories)
print('Total number of images:', len(face_jpgs))
```

```
Categories: ['Brother', 'Dad', 'Daughter', 'Me', 'Mum', 'Son', 'Wife']
Total number of images: 419
```

Below are some high level statistics about how many images in each category.

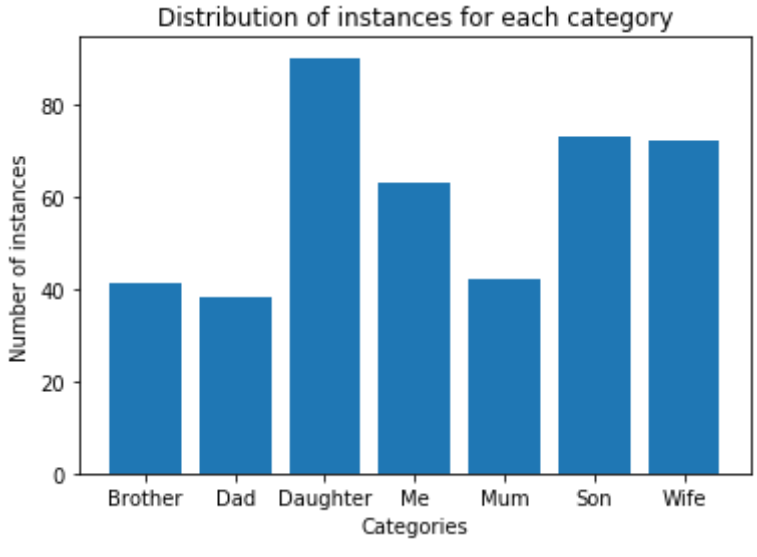
In [11]:
```
def count_each_category(categories, files):
    """Count the number of file for each category.
    Args:
        categories (list): A list of categories.
        files (list[str]): A list of strings which are the relative address of face images.
    Returns:
        stats_dict (dict): A dictionary of (category: number).
    """
    stats_dict = {}
    # Initilize the dictionary.
    for category in categories:
        stats_dict[category] = 0
    # Increment the value of the matching item.
    for file in files:
        # Convert the path string into Path object.
        file_path = Path(file)
        # str(file_path.parent) will return 'images\Brother', need to us os.sep as the delimiter
        # for cross platform use.
        file_category = str(file_path.parent).split(os.sep)[1]
        stats_dict[file_category] += 1
    return stats_dict
```

In [12]:
```
stats_dict = count_each_category(categories, face_jpgs)
print(stats_dict)
```

```
{'Me': 63, 'Mum': 42, 'Daughter': 90, 'Son': 73, 'Wife': 72, 'Brother': 41, 'Dad': 38}
```

Below bar plot shows the distribution of samples for each category. 'Daughter' category has the most images, 92. 'Son' has the second most, 73, so on and so forth. The distribution of samples are not well balanced. This observation will lead to few special 'treatment' in the later sections.

In [13]:
```
import matplotlib.pyplot as plt
%matplotlib inline
plt.bar(stats_dict.keys(), stats_dict.values())
plt.xlabel('Categories')
plt.ylabel('Number of instances')
plt.title('Distribution of instances for each category')
plt.show()
```



B2. Preprocess - Loading the Images and Split Them into Train, Validate and Test Datasets

In [14]:
```
# Function to load the images as a list of file names and one hot code categories
from sklearn.datasets import load_files
from sklearn.model_selection import train_test_split
from keras.utils import np_utils
from glob import glob

# Read all the files and return 2 numpy arrays.
# One is the address of the files and the other one is the one hot encode of the category.
def load_dataset(folder_addr):
    """Load all the files in the given directory. The name of each subdirectory will be the category name.
    Args:
        folder_addr (str): The folder address in which there are subfolders.
    Returns:
        face_files (numpy.ndarray): Numpy array of face file address strings.
        face_targets (numpy.ndarray): Numpy array of categories, value from 0 to 6, without one-hot encoding.
    """
    data = load_files(folder_addr)
    face_files = np.array(data['filenames'])
#     face_targets = np_utils.to_categorical(np.array(data['target']), 7)
    face_targets = np.array(data['target'])

    # Many of the times Windows and Mac are producing 'desktop.ini' or '.DS_Store' file automatically.
    # Need to omit them.
    invalid_files_idx = []
    for i in range(len(face_files)):
        if 'desktop.ini' in face_files[i] or 'DS_Store' in face_files[i]:
            invalid_files_idx.append(i)
    face_files = np.delete(face_files, invalid_files_idx)
    face_targets = np.delete(face_targets, invalid_files_idx)
    return face_files, face_targets
```

```
Using TensorFlow backend.
```

In [15]:
```
# Load the list of images and categories
faces, targets = load_dataset('./images')
face_names = [item[9:-1] for item in glob('./images/*/')]
print('There are %d face categories.' % len(face_names))
print(face_names)
print('There are %d total faces.' % len(faces))
print(count_each_category(categories, faces))
```

```
There are 7 face categories.
['Brother', 'Dad', 'Daughter', 'Me', 'Mum', 'Son', 'Wife']
There are 419 total faces.
{'Me': 63, 'Mum': 42, 'Daughter': 90, 'Son': 73, 'Wife': 72, 'Brother': 41, 'Dad': 38}
```

A random check on the file name and category name.

In [16]:
```
def parse_image_category(file_addr, category_code, is_one_hot=False):
    """Given the category in index or one-hot code, return the index and name of the category of an image.
    Args:
        file_addr (str): The address of the image whose name contains the category name.
        category_code (int or numpy.nparray): Integer index of the category or numpy array after one-hot encoding.
        is_one_hot (bool): Default it's False, set to True if the passed in category_code is in one-hot format.
    Returns:
        category_index (int): The index of the category, from 0 to 6.
        face_names[category_index] (str): The name of the category.
    """
    # Print file path and category in one hot code
    print(file_addr, category_code)
    if is_one_hot:
        category_index = np.argmax(category_code)
    else:
        category_index = category_code
    return category_index, face_names[category_index]

parse_image_category(faces[100], targets[100])
```

```
./images\Dad\20160701-1737.jpg-face-0.jpg 1
```

Out[16]:
```
(1, 'Dad')
```

From this point, the data set is finally ready for the coming machine learning piplines.

C. Extract Feature Vectors

In [17]:
```
from keras.applications.inception_v3 import InceptionV3
from keras.models import Model
from keras.layers import Dense, GlobalAveragePooling2D, Input
from keras import backend as K
from keras.applications.imagenet_utils import preprocess_input, decode_predictions
from keras.callbacks import ModelCheckpoint, EarlyStopping, LambdaCallback, ReduceLROnPlateau
from keras.models import load_model
from keras.preprocessing import image
```

In [18]:
```
def path_to_tensor(img_path):
    """Given one image path, return it as a numpy array.
    Args:
        img_path (str): The full path string of a image.
    Returns:
        np.expand_dims(x, axis=0) (numpy.ndarray): A 4D numpy array of a image after expanding from 3D to 4D.
    """
    img = image.load_img(img_path, target_size=(299,299))
    x = image.img_to_array(img)
    return np.expand_dims(x, axis=0)

def paths_to_tensor(img_paths):
    """Given the image paths, return them as a vertically stacked numpy array.
    Args:
        img_paths (list[str]): The full paths of the images in a list.
    Returns:
        np.vstack(list_of_tensors) (numpy.ndarray): 4D numpy array of all the images, after normalization.
    """
    list_of_tensors = [path_to_tensor(img_path) for img_path in img_paths if 'desktop.ini' not in img_path]
    return np.vstack(list_of_tensors).astype('float32')/255
```

In [19]:
```
# Load the inception v3 model, include the dense layers
base_model = InceptionV3(weights='imagenet', include_top=True)
vector_out = base_model.get_layer('avg_pool')
feature_model = Model(inputs=base_model.input, outputs=vector_out.output)
```

In [20]:
```
def get_features(feature_model, tensors):
    """Using the given model to convert a tensor/image to a feature vector
    Args:
        feature_model (Keras Model): In this project, it's the Google Inception V3 model without the last dense layer.
        tensors (numpy.ndarray): Group of images in a 4D numpy array.
    Returns:
        feature_outputs (numpy.ndarray): Feature vectors of the group of images, dimension of (x, 2018)
    """
    tensor_inputs = np.expand_dims(tensors, axis=0)
    feature_outputs = feature_model.predict(tensors)
    return feature_outputs
```

Split the dataset into train, validate and test datasets. The data are not well balanced based on the earlier bar plot diagram. Hence, stratified split function will be used here.

In [21]:
```
train_faces, test_faces, train_targets, test_targets = train_test_split(faces, targets, test_size=0.15, random_state=1
, stratify=targets)
train_faces, validate_faces, train_targets, validate_targets = train_test_split(train_faces, train_targets, test_size=
0.2, random_state=1, stratify=train_targets)
print('There are %d training faces.' % len(train_faces))
print(count_each_category(categories, train_faces))
print('There are %d validate faces.' % len(validate_faces))
print(count_each_category(categories, validate_faces))
print('There are %d test faces.' % len(test_faces))
print(count_each_category(categories, test_faces))
```

```
There are 284 training faces.
{'Me': 43, 'Mum': 29, 'Daughter': 61, 'Son': 49, 'Wife': 49, 'Brother': 28, 'Dad': 25}
There are 72 validate faces.
{'Me': 11, 'Mum': 7, 'Daughter': 15, 'Son': 13, 'Wife': 12, 'Brother': 7, 'Dad': 7}
There are 63 test faces.
{'Me': 9, 'Mum': 6, 'Daughter': 14, 'Son': 11, 'Wife': 11, 'Brother': 6, 'Dad': 6}
```

In [22]:
```
# Read the images as numpy arrays
train_tensors = paths_to_tensor(train_faces)
test_tensors = paths_to_tensor(test_faces)
validate_tensors = paths_to_tensor(validate_faces)
print("Train tensor shape.", train_tensors.shape)
print('Test tensor shape.', test_tensors.shape)
print('Validate tensor shape.', validate_tensors.shape)
```

```
Train tensor shape. (284, 299, 299, 3)
Test tensor shape. (63, 299, 299, 3)
Validate tensor shape. (72, 299, 299, 3)
```

In [23]:
```
train_features = get_features(feature_model, train_tensors)
validate_features = get_features(feature_model, validate_tensors)
test_features = get_features(feature_model, test_tensors)
print('Train features shape:', train_features.shape, '\nValidate feature_modelres shape: ', validate_features.shape,
      '\nTest features shape:', test_features.shape)
```

```
Train features shape: (284, 2048)
Validate feature_modelres shape:  (72, 2048)
Test features shape: (63, 2048)
```

D. Different Models - Attempt to Train and Test on Various Models

'Traditional' machine learning models of SGDClassifier, LogistricRegression, KNeighborsClassifier will be explored first. Then followed by deep neural network.

a). 'Traditional' Models

In [24]:
```
from sklearn.linear_model import SGDClassifier, LogisticRegression
from sklearn.neighbors import KNeighborsClassifier
clf_sgd = SGDClassifier(random_state=0)
clf_knn = KNeighborsClassifier(n_neighbors=7)
clf_log = LogisticRegression(random_state=0)
```

In [25]:
```
%time model_sgd = clf_sgd.fit(train_features, train_targets)
%time model_knn = clf_knn.fit(train_features, train_targets)
%time model_log = clf_log.fit(train_features, train_targets)
```

```
C:\Users\Xiaowei\Anaconda3\envs\tfkeras\lib\site-packages\sklearn\linear_model\stochastic_gradient.py:128: FutureWarn
ing: max_iter and tol parameters have been added in <class 'sklearn.linear_model.stochastic_gradient.SGDClassifier'>
in 0.19. If both are left unset, they default to max_iter=5 and tol=None. If tol is not None, max_iter defaults to ma
x_iter=1000. From 0.21, default max_iter will be 1000, and default tol will be 1e-3.
  "and default tol will be 1e-3." % type(self), FutureWarning)

Wall time: 41 ms
Wall time: 43 ms
Wall time: 799 ms
```

The default parameters of the 3 models are as following. Most of the hyper-parameters are the default ones except the `random_state` for SGD and Logistic regression, `n_neighbors` from KNN.

In [26]:
```
model_sgd.get_params
```

Out[26]:
```
<bound method BaseEstimator.get_params of SGDClassifier(alpha=0.0001, average=False, class_weight=None, epsilon=0.1,
       eta0=0.0, fit_intercept=True, l1_ratio=0.15,
       learning_rate='optimal', loss='hinge', max_iter=None, n_iter=None,
       n_jobs=1, penalty='l2', power_t=0.5, random_state=0, shuffle=True,
       tol=None, verbose=0, warm_start=False)>
```

In [27]:
```
model_knn.get_params
```

Out[27]:
```
<bound method BaseEstimator.get_params of KNeighborsClassifier(algorithm='auto', leaf_size=30, metric='minkowski',
           metric_params=None, n_jobs=1, n_neighbors=7, p=2,
           weights='uniform')>
```

In [28]:
```
model_log.get_params
```

Out[28]:
```
<bound method BaseEstimator.get_params of LogisticRegression(C=1.0, class_weight=None, dual=False, fit_intercept=Tru
e,
          intercept_scaling=1, max_iter=100, multi_class='ovr', n_jobs=1,
          penalty='l2', random_state=0, solver='liblinear', tol=0.0001,
          verbose=0, warm_start=False)>
```

In [29]:
```
# start = time.time()
%time predict_test_sgd = model_sgd.predict(test_features)
%time predict_test_knn = model_knn.predict(test_features)
%time predict_test_log = model_log.predict(test_features)
# end = time.time()
# print('%.2gs' %(end - start))
```

```
Wall time: 7.01 ms
Wall time: 56.1 ms
Wall time: 1e+03 µs
```

Due to the imbalanced data as mentioned earlier, the average of precision, recall and fbeta score will be set as `weighted` .

In [30]:
```
from sklearn.metrics import fbeta_score, precision_recall_fscore_support, confusion_matrix
import pandas as pd
def get_score_numpy(test_targets, predict_targets, average = 'weighted'):
    score = precision_recall_fscore_support(test_targets, predict_targets, average=average)
    score = np.array(score)
    score = score[score != None]
    return score
```

In [31]:
```
score_sgd = get_score_numpy(test_targets, predict_test_sgd)
score_knn = get_score_numpy(test_targets, predict_test_knn)
score_log = get_score_numpy(test_targets, predict_test_log)
```

In [32]:
```
data = pd.DataFrame(np.stack((score_sgd, score_knn, score_log)),
                    columns=['Precision', 'Recall', 'F1'], index=['SGD', 'KNN', 'Log'])
data
```

Out[32]:

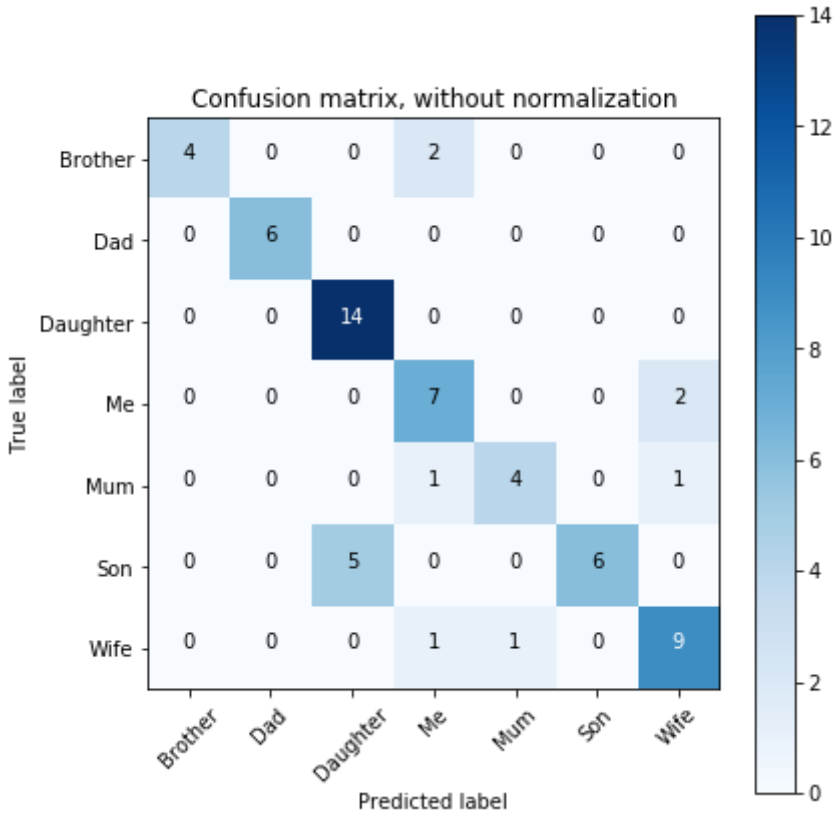
	Precision	Recall	F1
SGD	0.792347	0.777778	0.775751
KNN	0.717082	0.68254	0.683151
Log	0.826874	0.793651	0.78914

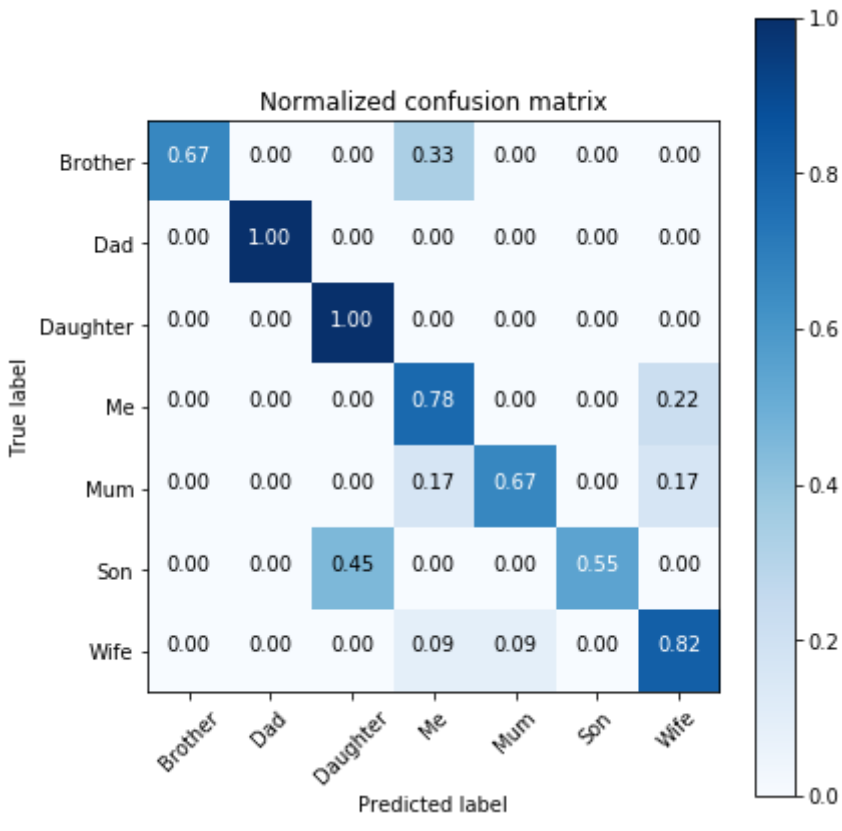
Based on above table, logisitic regression performed the best based on every angel of precision, recall and f1 scores. It is fast on testing but not training.

Below confusion matrix plots provide more intuitive ways of how good or bad the logistic regression model is doing for each label. The first plot is based on the absolute number of samples. The second plot is after normalization, which shows a more accurate estimation.

In [33]:
```
import itertools
def plot_confusion_matrix(cm, classes,
                          normalize=False,
                          title='Confusion matrix',
                          cmap=plt.cm.Blues):
    """
    This function prints and plots the confusion matrix.
    Normalization can be applied by setting `normalize=True`.
    """
    if normalize:
        cm = cm.astype('float') / cm.sum(axis=1)[:, np.newaxis]
#         print("Normalized confusion matrix")
    else:
        pass
#         print('Confusion matrix, without normalization')

#     print(cm)

    plt.imshow(cm, interpolation='nearest', cmap=cmap)
    plt.title(title)
    plt.colorbar()
    tick_marks = np.arange(len(classes))
    plt.xticks(tick_marks, classes, rotation=45)
    plt.yticks(tick_marks, classes)

    fmt = '.2f' if normalize else 'd'
    thresh = cm.max() / 2.
    for i, j in itertools.product(range(cm.shape[0]), range(cm.shape[1])):
        plt.text(j, i, format(cm[i, j], fmt), horizontalalignment="center", color="white" if cm[i, j] > thresh else "b
lack")

    plt.tight_layout()
    plt.ylabel('True label')
    plt.xlabel('Predicted label')
```

In [34]:
```
cnf_matrix = confusion_matrix(test_targets, predict_test_log)
plt.figure(figsize=(6,6))
plot_confusion_matrix(cnf_matrix, classes=face_names, title='Confusion matrix, without normalization')
plt.figure(figsize=(6,6))
plot_confusion_matrix(cnf_matrix, classes=face_names, normalize=True, title='Normalized confusion matrix')
plt.show()
```





b). Deep Learning Model

So far all the chosen models performed training and prediction in fractions of a second. The best accuracy is about 80% from logistic regression while KNN only got the lowest 68%, even though it's much better than random guess of 1/7 = 16.7%. Below is an exploration on using DNN model. After a few tries I decided to have 3 dense layers as it provides a balance between accuracy and time.

In [35]:
```
Inp = Input(shape=(2048,))
x = Dense(300, activation='relu')(Inp)
x = Dense(50, activation='relu')(x)
output = Dense(7, activation='softmax')(x)
model_dnn = Model(inputs=Inp, outputs=output)
model_dnn.summary()
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_2 (InputLayer)         (None, 2048)              0
_________________________________________________________________
dense_1 (Dense)              (None, 300)               614700
_________________________________________________________________
dense_2 (Dense)              (None, 50)                15050
_________________________________________________________________
dense_3 (Dense)              (None, 7)                 357
=================================================================
Total params: 630,107
Trainable params: 630,107
Non-trainable params: 0
_________________________________________________________________
```

As a classification problem, deep learning models will require the output in one-hot format. If not, the class 1 and 2 will be misunderstood as ordinal instead of nominal.

In [36]:
```
train_targets_one_hot = np_utils.to_categorical(train_targets)
validate_targets_one_hot = np_utils.to_categorical(validate_targets)
test_targets_one_hot = np_utils.to_categorical(test_targets)
```

In [37]:
```
model_dnn.compile(optimizer='rmsprop', loss='categorical_crossentropy', metrics=['accuracy'])
model_dnn.save_weights('models/init_weights.h5')
checkpointer = ModelCheckpoint(filepath='models/model_dnn.h5', verbose=1, save_best_only=True)
```

In [33]:
```
%%time
model_dnn.load_weights('models/init_weights.h5')
hist_1 = model_dnn.fit(train_features, train_targets_one_hot,
            validation_data=(validate_features, validate_targets_one_hot),
            epochs=50, verbose=1, batch_size=20,
            callbacks=[checkpointer])
```

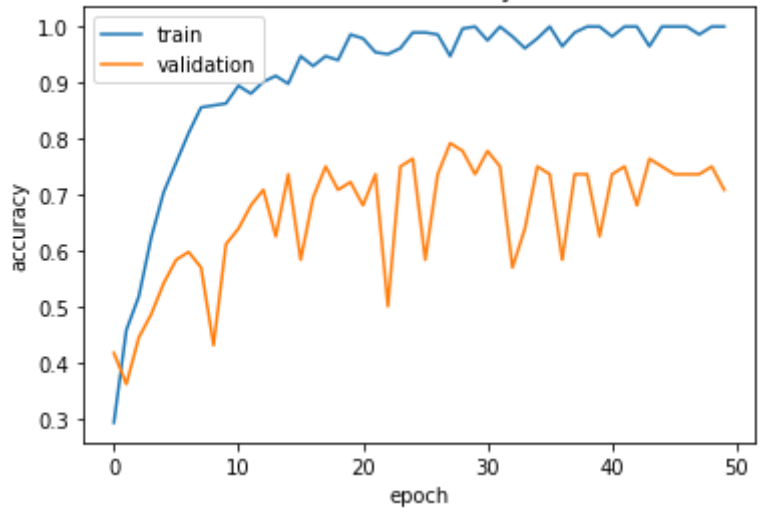
```
Train on 284 samples, validate on 72 samples
Epoch 1/50
200/284 [====================>.........] - ETA: 0s - loss: 2.2020 - acc: 0.2650Epoch 00000: val_loss improved from in
f to 1.64527, saving model to models/model_dnn.h5
284/284 [==============================] - 0s - loss: 1.9837 - acc: 0.2923 - val_loss: 1.6453 - val_acc: 0.4167
Epoch 2/50
220/284 [======================>.......] - ETA: 0s - loss: 1.4043 - acc: 0.4682Epoch 00001: val_loss improved from 1.
64527 to 1.32060, saving model to models/model_dnn.h5
284/284 [==============================] - 0s - loss: 1.3760 - acc: 0.4577 - val_loss: 1.3206 - val_acc: 0.3611
Epoch 3/50
200/284 [====================>.........] - ETA: 0s - loss: 1.2114 - acc: 0.5150Epoch 00002: val_loss did not improve
284/284 [==============================] - 0s - loss: 1.1602 - acc: 0.5176 - val_loss: 1.4567 - val_acc: 0.4444
Epoch 4/50
200/284 [====================>.........] - ETA: 0s - loss: 0.9996 - acc: 0.6200Epoch 00003: val_loss improved from 1.
32060 to 1.15075, saving model to models/model_dnn.h5
284/284 [==============================] - 0s - loss: 0.9435 - acc: 0.6232 - val_loss: 1.1507 - val_acc: 0.4861
Epoch 5/50
200/284 [====================>.........] - ETA: 0s - loss: 0.8163 - acc: 0.6700Epoch 00004: val_loss did not improve
284/284 [==============================] - 0s - loss: 0.7795 - acc: 0.7042 - val_loss: 1.1881 - val_acc: 0.5417
Epoch 6/50
220/284 [======================>.......] - ETA: 0s - loss: 0.7250 - acc: 0.7364Epoch 00005: val_loss improved from 1.
15075 to 0.89543, saving model to models/model_dnn.h5
284/284 [==============================] - 0s - loss: 0.6895 - acc: 0.7570 - val_loss: 0.8954 - val_acc: 0.5833
Epoch 7/50
220/284 [======================>.......] - ETA: 0s - loss: 0.6183 - acc: 0.7955Epoch 00006: val_loss did not improve
284/284 [==============================] - 0s - loss: 0.5816 - acc: 0.8099 - val_loss: 0.9426 - val_acc: 0.5972
Epoch 8/50
180/284 [==================>...........] - ETA: 0s - loss: 0.4804 - acc: 0.8500Epoch 00007: val_loss did not improve
284/284 [==============================] - 0s - loss: 0.4561 - acc: 0.8556 - val_loss: 1.5625 - val_acc: 0.5694
Epoch 9/50
200/284 [====================>.........] - ETA: 0s - loss: 0.4331 - acc: 0.8400Epoch 00008: val_loss did not improve
284/284 [==============================] - 0s - loss: 0.4140 - acc: 0.8592 - val_loss: 1.5881 - val_acc: 0.4306
Epoch 10/50
180/284 [==================>...........] - ETA: 0s - loss: 0.3935 - acc: 0.8889Epoch 00009: val_loss did not improve
284/284 [==============================] - 0s - loss: 0.4125 - acc: 0.8627 - val_loss: 1.0297 - val_acc: 0.6111
Epoch 11/50
200/284 [====================>.........] - ETA: 0s - loss: 0.3153 - acc: 0.8850Epoch 00010: val_loss improved from 0.
89543 to 0.72555, saving model to models/model_dnn.h5
284/284 [==============================] - 0s - loss: 0.2918 - acc: 0.8944 - val_loss: 0.7256 - val_acc: 0.6389
Epoch 12/50
200/284 [====================>.........] - ETA: 0s - loss: 0.2787 - acc: 0.8950Epoch 00011: val_loss did not improve
284/284 [==============================] - 0s - loss: 0.3066 - acc: 0.8803 - val_loss: 0.8356 - val_acc: 0.6806
Epoch 13/50
180/284 [==================>...........] - ETA: 0s - loss: 0.2786 - acc: 0.9111Epoch 00012: val_loss did not improve
284/284 [==============================] - 0s - loss: 0.2695 - acc: 0.9014 - val_loss: 0.8242 - val_acc: 0.7083
Epoch 14/50
160/284 [===============>..............] - ETA: 0s - loss: 0.2178 - acc: 0.9125Epoch 00013: val_loss did not improve
284/284 [==============================] - 0s - loss: 0.2566 - acc: 0.9120 - val_loss: 1.0343 - val_acc: 0.6250
Epoch 15/50
200/284 [====================>.........] - ETA: 0s - loss: 0.1871 - acc: 0.9200Epoch 00014: val_loss did not improve
284/284 [==============================] - 0s - loss: 0.2138 - acc: 0.8979 - val_loss: 0.8379 - val_acc: 0.7361
Epoch 16/50
200/284 [====================>.........] - ETA: 0s - loss: 0.1408 - acc: 0.9600Epoch 00015: val_loss did not improve
284/284 [==============================] - 0s - loss: 0.1598 - acc: 0.9472 - val_loss: 1.0514 - val_acc: 0.5833
Epoch 17/50
220/284 [======================>.......] - ETA: 0s - loss: 0.1832 - acc: 0.9227Epoch 00016: val_loss did not improve
284/284 [==============================] - 0s - loss: 0.1784 - acc: 0.9296 - val_loss: 1.0808 - val_acc: 0.6944
Epoch 18/50
200/284 [====================>.........] - ETA: 0s - loss: 0.1459 - acc: 0.9450Epoch 00017: val_loss did not improve
284/284 [==============================] - 0s - loss: 0.1369 - acc: 0.9472 - val_loss: 0.8053 - val_acc: 0.7500
Epoch 19/50
220/284 [======================>.......] - ETA: 0s - loss: 0.1815 - acc: 0.9273Epoch 00018: val_loss did not improve
284/284 [==============================] - 0s - loss: 0.1622 - acc: 0.9401 - val_loss: 0.8255 - val_acc: 0.7083
Epoch 20/50
240/284 [========================>.....] - ETA: 0s - loss: 0.0648 - acc: 0.9875Epoch 00019: val_loss did not improve
284/284 [==============================] - 0s - loss: 0.0743 - acc: 0.9859 - val_loss: 0.8420 - val_acc: 0.7222
Epoch 21/50
220/284 [======================>.......] - ETA: 0s - loss: 0.0994 - acc: 0.9773Epoch 00020: val_loss did not improve
284/284 [==============================] - 0s - loss: 0.0954 - acc: 0.9789 - val_loss: 0.8178 - val_acc: 0.6806
Epoch 22/50
200/284 [====================>.........] - ETA: 0s - loss: 0.1632 - acc: 0.9350Epoch 00021: val_loss did not improve
284/284 [==============================] - 0s - loss: 0.1244 - acc: 0.9542 - val_loss: 0.8032 - val_acc: 0.7361
Epoch 23/50
220/284 [======================>.......] - ETA: 0s - loss: 0.0985 - acc: 0.9591Epoch 00022: val_loss did not improve
284/284 [==============================] - 0s - loss: 0.1190 - acc: 0.9507 - val_loss: 1.7890 - val_acc: 0.5000
Epoch 24/50
240/284 [========================>.....] - ETA: 0s - loss: 0.1148 - acc: 0.9625Epoch 00023: val_loss improved from 0.
72555 to 0.72137, saving model to models/model_dnn.h5
284/284 [==============================] - 0s - loss: 0.1110 - acc: 0.9613 - val_loss: 0.7214 - val_acc: 0.7500
Epoch 25/50
220/284 [======================>.......] - ETA: 0s - loss: 0.0562 - acc: 0.9864Epoch 00024: val_loss did not improve
284/284 [==============================] - 0s - loss: 0.0464 - acc: 0.9894 - val_loss: 0.7821 - val_acc: 0.7639
Epoch 26/50
240/284 [========================>.....] - ETA: 0s - loss: 0.0189 - acc: 1.0000Epoch 00025: val_loss did not improve
284/284 [==============================] - 0s - loss: 0.0438 - acc: 0.9894 - val_loss: 1.2698 - val_acc: 0.5833
Epoch 27/50
220/284 [======================>.......] - ETA: 0s - loss: 0.0648 - acc: 0.9818Epoch 00026: val_loss did not improve
284/284 [==============================] - 0s - loss: 0.0541 - acc: 0.9859 - val_loss: 0.7611 - val_acc: 0.7361
```

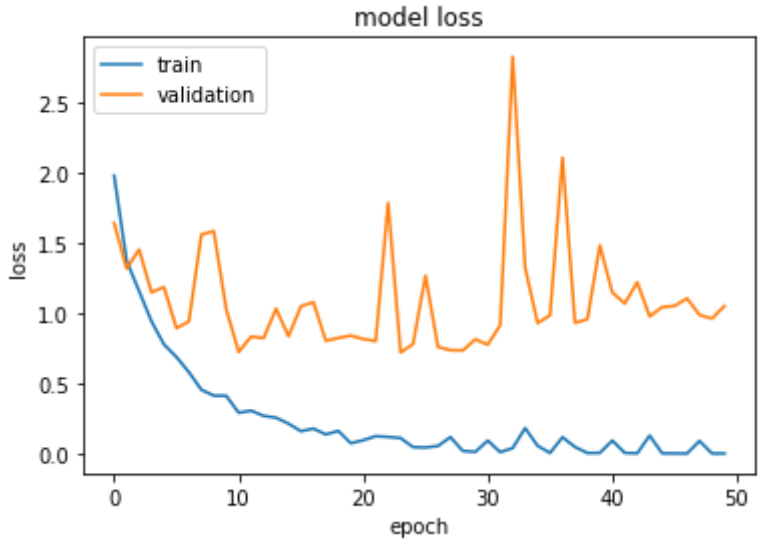
```
Epoch 28/50
200/284 [====================>.........] - ETA: 0s - loss: 0.0947 - acc: 0.9500Epoch 00027: val_loss did not improve
284/284 [==============================] - 0s - loss: 0.1176 - acc: 0.9472 - val_loss: 0.7381 - val_acc: 0.7917
Epoch 29/50
220/284 [======================>.......] - ETA: 0s - loss: 0.0148 - acc: 1.0000Epoch 00028: val_loss did not improve
284/284 [==============================] - 0s - loss: 0.0191 - acc: 0.9965 - val_loss: 0.7364 - val_acc: 0.7778
Epoch 30/50
220/284 [======================>.......] - ETA: 0s - loss: 0.0141 - acc: 1.0000Epoch 00029: val_loss did not improve
284/284 [==============================] - 0s - loss: 0.0120 - acc: 1.0000 - val_loss: 0.8157 - val_acc: 0.7361
Epoch 31/50
200/284 [====================>.........] - ETA: 0s - loss: 0.1200 - acc: 0.9700Epoch 00030: val_loss did not improve
284/284 [==============================] - 0s - loss: 0.0929 - acc: 0.9754 - val_loss: 0.7778 - val_acc: 0.7778
Epoch 32/50
180/284 [==================>...........] - ETA: 0s - loss: 0.0131 - acc: 1.0000Epoch 00031: val_loss did not improve
284/284 [==============================] - 0s - loss: 0.0104 - acc: 1.0000 - val_loss: 0.9176 - val_acc: 0.7500
Epoch 33/50
220/284 [======================>.......] - ETA: 0s - loss: 0.0143 - acc: 0.9909Epoch 00032: val_loss did not improve
284/284 [==============================] - 0s - loss: 0.0396 - acc: 0.9824 - val_loss: 2.8326 - val_acc: 0.5694
Epoch 34/50
220/284 [======================>.......] - ETA: 0s - loss: 0.1699 - acc: 0.9773Epoch 00033: val_loss did not improve
284/284 [==============================] - 0s - loss: 0.1820 - acc: 0.9613 - val_loss: 1.3268 - val_acc: 0.6389
Epoch 35/50
220/284 [======================>.......] - ETA: 0s - loss: 0.0692 - acc: 0.9727Epoch 00034: val_loss did not improve
284/284 [==============================] - 0s - loss: 0.0561 - acc: 0.9789 - val_loss: 0.9311 - val_acc: 0.7500
Epoch 36/50
220/284 [======================>.......] - ETA: 0s - loss: 0.0052 - acc: 1.0000Epoch 00035: val_loss did not improve
284/284 [==============================] - 0s - loss: 0.0047 - acc: 1.0000 - val_loss: 0.9880 - val_acc: 0.7361
Epoch 37/50
220/284 [======================>.......] - ETA: 0s - loss: 0.0166 - acc: 1.0000Epoch 00036: val_loss did not improve
284/284 [==============================] - 0s - loss: 0.1168 - acc: 0.9648 - val_loss: 2.1115 - val_acc: 0.5833
Epoch 38/50
220/284 [======================>.......] - ETA: 0s - loss: 0.0602 - acc: 0.9864Epoch 00037: val_loss did not improve
284/284 [==============================] - 0s - loss: 0.0475 - acc: 0.9894 - val_loss: 0.9336 - val_acc: 0.7361
Epoch 39/50
220/284 [======================>.......] - ETA: 0s - loss: 0.0050 - acc: 1.0000Epoch 00038: val_loss did not improve
284/284 [==============================] - 0s - loss: 0.0043 - acc: 1.0000 - val_loss: 0.9593 - val_acc: 0.7361
Epoch 40/50
200/284 [====================>.........] - ETA: 0s - loss: 0.0036 - acc: 1.0000Epoch 00039: val_loss did not improve
284/284 [==============================] - 0s - loss: 0.0049 - acc: 1.0000 - val_loss: 1.4864 - val_acc: 0.6250
Epoch 41/50
220/284 [======================>.......] - ETA: 0s - loss: 0.1170 - acc: 0.9773Epoch 00040: val_loss did not improve
284/284 [==============================] - 0s - loss: 0.0932 - acc: 0.9824 - val_loss: 1.1501 - val_acc: 0.7361
Epoch 42/50
220/284 [======================>.......] - ETA: 0s - loss: 0.0062 - acc: 1.0000Epoch 00041: val_loss did not improve
284/284 [==============================] - 0s - loss: 0.0052 - acc: 1.0000 - val_loss: 1.0714 - val_acc: 0.7500
Epoch 43/50
200/284 [====================>.........] - ETA: 0s - loss: 0.0021 - acc: 1.0000Epoch 00042: val_loss did not improve
284/284 [==============================] - 0s - loss: 0.0021 - acc: 1.0000 - val_loss: 1.2223 - val_acc: 0.6806
Epoch 44/50
220/284 [======================>.......] - ETA: 0s - loss: 0.1624 - acc: 0.9545Epoch 00043: val_loss did not improve
284/284 [==============================] - 0s - loss: 0.1290 - acc: 0.9648 - val_loss: 0.9794 - val_acc: 0.7639
Epoch 45/50
220/284 [======================>.......] - ETA: 0s - loss: 0.0024 - acc: 1.0000    Epoch 00044: val_loss did not impr
ove
284/284 [==============================] - 0s - loss: 0.0021 - acc: 1.0000 - val_loss: 1.0456 - val_acc: 0.7500
Epoch 46/50
240/284 [========================>.....] - ETA: 0s - loss: 0.0014 - acc: 1.0000Epoch 00045: val_loss did not improve
284/284 [==============================] - 0s - loss: 0.0013 - acc: 1.0000 - val_loss: 1.0542 - val_acc: 0.7361
Epoch 47/50
220/284 [======================>.......] - ETA: 0s - loss: 8.8702e-04 - acc: 1.0000Epoch 00046: val_loss did not impr
ove
284/284 [==============================] - 0s - loss: 9.3300e-04 - acc: 1.0000 - val_loss: 1.1077 - val_acc: 0.7361
Epoch 48/50
220/284 [======================>.......] - ETA: 0s - loss: 0.1158 - acc: 0.9818Epoch 00047: val_loss did not improve
284/284 [==============================] - 0s - loss: 0.0904 - acc: 0.9859 - val_loss: 0.9899 - val_acc: 0.7361
Epoch 49/50
220/284 [======================>.......] - ETA: 0s - loss: 0.0012 - acc: 1.0000    Epoch 00048: val_loss did not impr
ove
284/284 [==============================] - 0s - loss: 0.0013 - acc: 1.0000 - val_loss: 0.9638 - val_acc: 0.7500
Epoch 50/50
220/284 [======================>.......] - ETA: 0s - loss: 9.2503e-04 - acc: 1.0000Epoch 00049: val_loss did not impr
ove
284/284 [==============================] - 0s - loss: 0.0010 - acc: 1.0000 - val_loss: 1.0532 - val_acc: 0.7083
Wall time: 6.16 s
```

In [34]:
```
## TODO: Visualize the training and validation loss of your neural network
import matplotlib.pyplot as plt
def plt_hist(hist):
    print(hist.history.keys())
    # summarize history for accuracy
    plt.plot(hist.history['acc'])
    plt.plot(hist.history['val_acc'])
    plt.title('model accuracy')
    plt.ylabel('accuracy')
    plt.xlabel('epoch')
    plt.legend(['train', 'validation'], loc='upper left')
    plt.show()
    # summarize history for loss
    plt.plot(hist.history['loss'])
    plt.plot(hist.history['val_loss'])
    plt.title('model loss')
    plt.ylabel('loss')
    plt.xlabel('epoch')
    plt.legend(['train', 'validation'], loc='upper left')
    plt.show()
```

In [35]:
```
plt_hist(hist_1)
```

```
dict_keys(['val_loss', 'acc', 'loss', 'val_acc'])
```





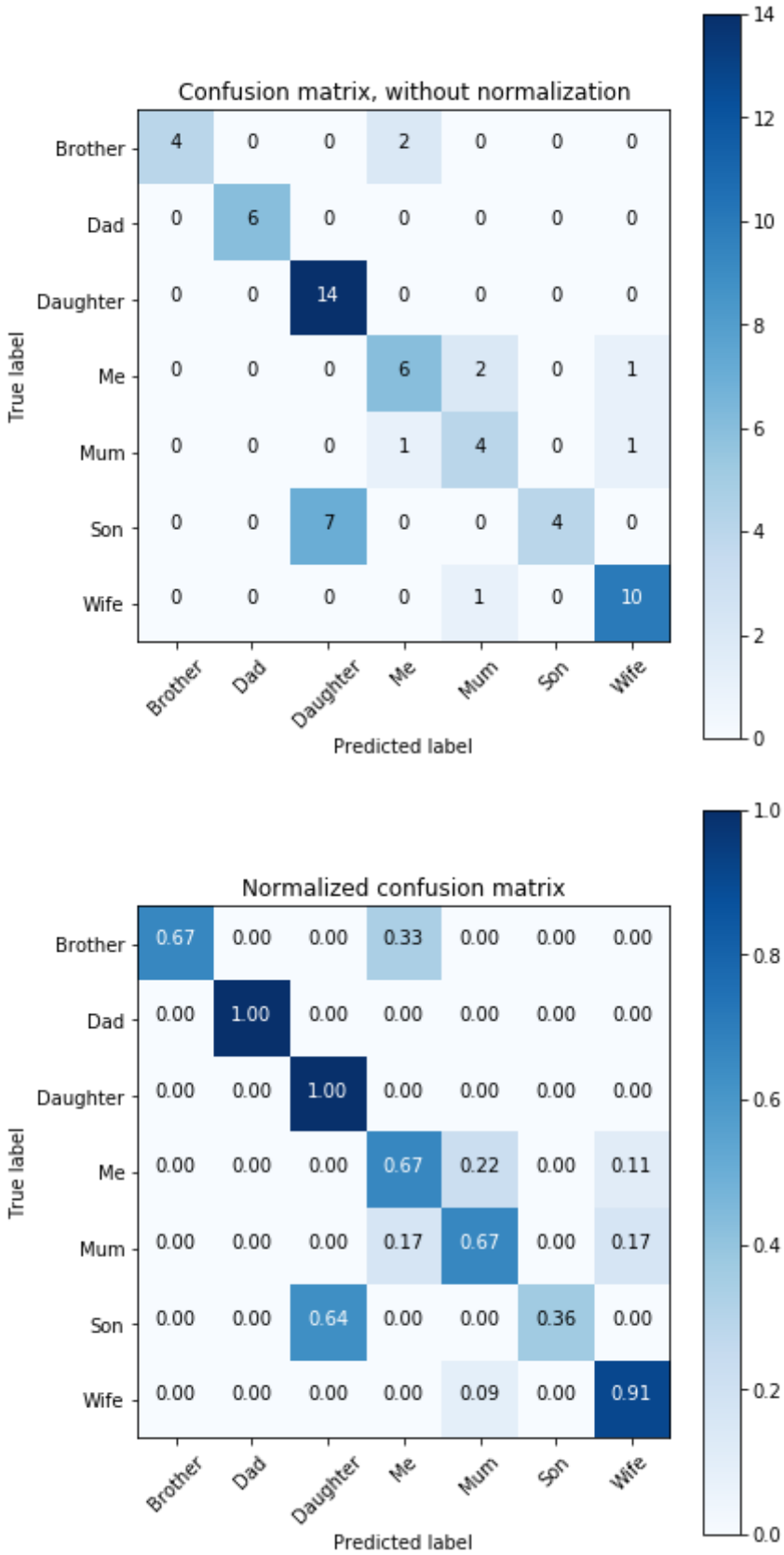
Epoch of 50 looks like a reasonable number of iterations. The validation accuracy is stable and the loss is starting to grow, that's a sign of overfitting. Techniques of reducing learning rate, early stopping could be further explored, but not covered in this project.

Below is the metrics of the DNN model.

In [38]:
```
model_dnn = load_model('./models/model_dnn.h5')
%time predict_test_dnn = model_dnn.predict(test_features)
predict_test_dnn = predict_test_dnn.argmax(axis=-1)
print('Precision, Recall, F1')
print(get_score_numpy(test_targets, predict_test_dnn))
```

```
Wall time: 204 ms
Precision, Recall, F1
[0.80839002267573701 0.76190476190476186 0.74800304365521764]
```

In [39]:
```
cnf_matrix = confusion_matrix(test_targets, predict_test_dnn)
plt.figure(figsize=(6,6))
plot_confusion_matrix(cnf_matrix, classes=face_names, title='Confusion matrix, without normalization')
plt.figure(figsize=(6,6))
plot_confusion_matrix(cnf_matrix, classes=face_names, normalize=True, title='Normalized confusion matrix')
plt.show()
```



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, as well as prediction.

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 I - Choose Model

A. Create More Images

Use image generator (https://blog.keras.io/building-powerful-image-classification-models-using-very-little-data.html) to create more images. This is a very useful technique when having little data.

In this project, I put the new images in folder `images2` and 10 new images will be generated for each original image. Generating new images is basically to do some distortion on the images. I think it's reasonable to narrow, widen, rotate, zoom and flip the images a little, in a reasonable range. Hence, values of the most parameters I set to 20%.

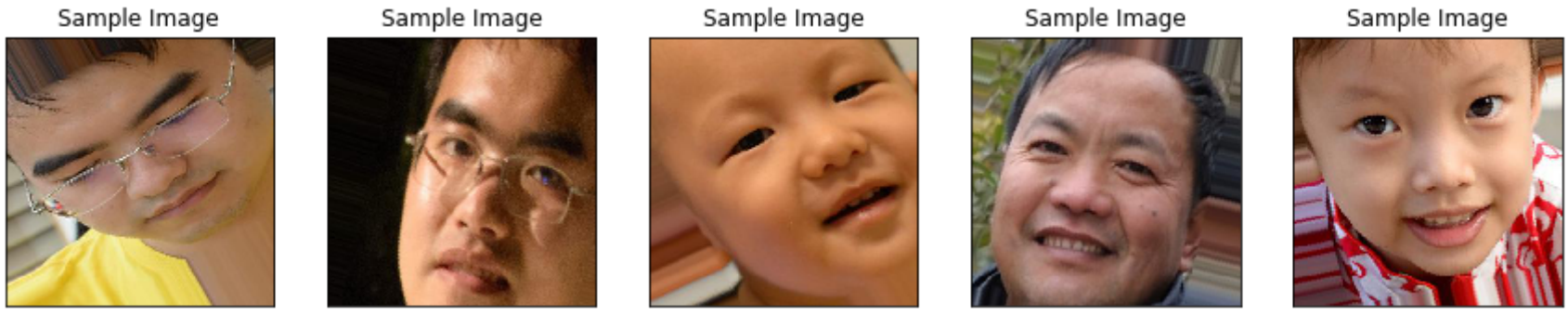
Before the image generator I had only 419 image for both training and testing. And now I have 10 times more images. For each round of training I'll have more than 4,000 images.

In [40]:
```
from keras.preprocessing.image import ImageDataGenerator, array_to_img, img_to_array, load_img
import shutil

datagen = ImageDataGenerator(
        rotation_range=40,
        width_shift_range=0.2,
        height_shift_range=0.2,
        shear_range=0.2,
        zoom_range=0.2,
        horizontal_flip=True,
        fill_mode='nearest')

def get_target_dir(face_file):
    """ Given the address of the original image, create directory for the new generated images if needed.
        Also return the new directory address.
    Args:
        face_file (str): The address of the original face file, may or may not be in full address.
    Returns:
        target_dir (str): The address of the directory for the generated images.
    """
    target_file = face_file.replace('images', 'images2')
    target_dir = os.path.dirname(target_file)
    target_path = Path(target_dir)

    # Create parents of directory, don't raise exception if the directory exists
    target_path.mkdir(parents=True, exist_ok=True)
    return target_dir
```

```
def generate_images(faces):
    """ Given the address of face images, use image generator to generate new images.
    Args:
        faces ([str]): The list of face images addresses.
    Returns:
        None: Generate new images and return None.
    """
    if os.path.exists('images2'):
        shutil.rmtree('images2', ignore_errors=True)
        # sleep for 2 seconds to allow OS finish the previous deletion action.
        time.sleep(2)
    for n, face in enumerate(faces):
        if n % 100 == 0:
            print('Generating images, ' + str(n) + ' of ' + str(len(faces)) + ' faces')
        target_dir = get_target_dir(face)

        img = load_img(face)
        x = img_to_array(img)  # this is a Numpy array with shape (3, 299, 299)
        x = x.reshape((1,) + x.shape)  # this is a Numpy array with shape (1, 3, 299, 299)

        # Copy the current face image (before augumentation) to target directory.
        shutil.copy2(face, target_dir)
        # the .flow() command below generates batches of randomly transformed images
        # and saves the results to the target directory
        i = 0
        for batch in datagen.flow(x, batch_size=1, save_to_dir=target_dir, save_prefix='gen', save_format='jpg'):
            i += 1
            if i > 10:
                break  # otherwise the generator would loop indefinitely
```

In [41]:
```
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'>
```



B. With Stratified K-fold CV

For a more 'stable' performance of each model, K-fold CV will be used in this project. So that each and every image will have the chance to be used for both training and testing. In this way, we will have a more 'averaged' and thus more 'stable' performance evaluation to show the robustness of the model.

In [42]:
```
from sklearn.model_selection import StratifiedKFold
n_splits = 10
skf = StratifiedKFold(n_splits=n_splits, random_state=0)
skf.get_n_splits(faces, targets)
```

Out[42]: 10

In [43]:
```
def perform_training(clf, features, targets):
    ''' Perform training, return the trained model and time taken in seconds.
    Args:
        clf (sklearn-model): The sklearn model.
        features (numpy.array): Training input, 4D numpy array.
        targets (numpy.array): Training output, 1D numpy array.
    Returns:
        model (sklean-model): The trained model.
        end - start: Time taken, in seconds.
    '''
    start = time.time()
    model = clf.fit(features, targets)
    end = time.time()
    return model, round(end - start, 3)

def perform_testing(model, features):
    ''' Perform testing, return the accuracy and time taken in seconds.
    Args:
        model (sklearn-model): The sklearn model.
        features (numpy.array): Testing input, 4D numpy array.
    Returns:
        prediction ([int]): The list of predicted labels.
        end - start: Time taken, in seconds.
    '''
    start = time.time()
    predictions = model.predict(features)
    end = time.time()
    return predictions, round(end - start, 3)

def perform_dnn_training(model, features, targets):
    ''' Perform training, return the trained model and time taken in seconds.
    Args:
        clf (sklearn-model): The sklearn model.
        features (numpy.array): Training input, 4D numpy array.
        targets (numpy.array): Training output, 1D numpy array.
    Returns:
        model (sklean-model): The trained model.
        end - start: Time taken, in seconds.
    '''
    targets = np_utils.to_categorical(targets)
    model.compile(optimizer='rmsprop', loss='categorical_crossentropy', metrics=['accuracy'])
    start = time.time()
    model.fit(features, targets, epochs=50, verbose=0, batch_size=20)
    end = time.time()
    return model, round(end - start, 3)

def perform_dnn_testing(model, features):
    ''' Perform testing, return the accuracy and time taken in seconds.
    Args:
        model (keras.model): The keras model
        features (numpy.array): Testing input, 4D numpy array.
    Returns:
        predictions ([int]): The list of predicted labels.
        end - start: Time taken, in seconds.
    '''
    start = time.time()
    probas = model.predict(features)
    predictions = probas.argmax(axis=-1)
    end = time.time()
    return predictions, round(end - start, 3)
```

In [88]:
```
all_test_targets = []
all_predict_targets = []
all_train_times = []
all_test_times = []
n = 0
for train_index, test_index in skf.split(faces, targets):
    n = n + 1
    print('Round ' + str(n) + ' of ' + str(n_splits))
    x_train, x_test = faces[train_index], faces[test_index]
    y_train, y_test = targets[train_index], targets[test_index]

    generate_images(x_train)
    # After generating the images, reaload the training dataset, testing dataset remains the same
    x_train, y_train = load_dataset('./images2')
    face_names = [item[10:-1] for item in glob('./images2/*/')]

    x_train_tensors = paths_to_tensor(x_train)
    x_test_tensors = paths_to_tensor(x_test)
    x_train_features = get_features(feature_model, x_train_tensors)
    x_test_features = get_features(feature_model, x_test_tensors)

    # Define and initialize models.
    clf_sgd = SGDClassifier(random_state=0)
    clf_knn = KNeighborsClassifier(n_neighbors=7)
    clf_log = LogisticRegression(random_state=0)
    predict_targets = []
    train_times = []
    test_times = []
    for clf in [clf_sgd, clf_knn, clf_log]:
        model, train_time = perform_training(clf, x_train_features, y_train)
        cur_predict_targets, test_time = perform_testing(model, x_test_features)
        all_predict_targets.append(cur_predict_targets)
        all_test_targets.append(y_test)
        train_times.append(train_time)
        test_times.append(test_time)

    # DNN model must use this way to 're-initialize' the weights. The 'models/init_weights.h5' was
    # created in the earlier section for the first try of DNN.
    model_dnn.load_weights('models/init_weights.h5')
    model_dnn, train_time = perform_dnn_training(model_dnn, x_train_features, y_train)
    cur_predict_targets, test_time = perform_dnn_testing(model_dnn, x_test_features)
    all_predict_targets.append(cur_predict_targets)
    all_test_targets.append(y_test)
    train_times.append(train_time)
    test_times.append(test_time)

    all_train_times.append(train_times)
    all_test_times.append(test_times)
```

```
Round 1 of 10
Generating images, 0 of 373 faces
Generating images, 100 of 373 faces
Generating images, 200 of 373 faces
Generating images, 300 of 373 faces

C:\Users\Xiaowei\Anaconda3\envs\tfkeras\lib\site-packages\sklearn\linear_model\stochastic_gradient.py:128: FutureWarn
ing: max_iter and tol parameters have been added in <class 'sklearn.linear_model.stochastic_gradient.SGDClassifier'>
in 0.19. If both are left unset, they default to max_iter=5 and tol=None. If tol is not None, max_iter defaults to ma
x_iter=1000. From 0.21, default max_iter will be 1000, and default tol will be 1e-3.
  "and default tol will be 1e-3." % type(self), FutureWarning)

Round 2 of 10
Generating images, 0 of 374 faces
Generating images, 100 of 374 faces
Generating images, 200 of 374 faces
Generating images, 300 of 374 faces
Round 3 of 10
Generating images, 0 of 376 faces
Generating images, 100 of 376 faces
Generating images, 200 of 376 faces
Generating images, 300 of 376 faces
Round 4 of 10
Generating images, 0 of 378 faces
Generating images, 100 of 378 faces
Generating images, 200 of 378 faces
Generating images, 300 of 378 faces
Round 5 of 10
Generating images, 0 of 378 faces
Generating images, 100 of 378 faces
Generating images, 200 of 378 faces
Generating images, 300 of 378 faces
Round 6 of 10
Generating images, 0 of 378 faces
Generating images, 100 of 378 faces
Generating images, 200 of 378 faces
Generating images, 300 of 378 faces
Round 7 of 10
Generating images, 0 of 378 faces
Generating images, 100 of 378 faces
Generating images, 200 of 378 faces
Generating images, 300 of 378 faces
Round 8 of 10
Generating images, 0 of 378 faces
Generating images, 100 of 378 faces
Generating images, 200 of 378 faces
Generating images, 300 of 378 faces
Round 9 of 10
Generating images, 0 of 379 faces
Generating images, 100 of 379 faces
Generating images, 200 of 379 faces
Generating images, 300 of 379 faces
Round 10 of 10
Generating images, 0 of 379 faces
Generating images, 100 of 379 faces
Generating images, 200 of 379 faces
Generating images, 300 of 379 faces
```

V. Results and Benchmark

In this section, results from the previous refinement section will be analysed, and the best model will be determined.

A. Pre-processing

Convert the predicted targets and test targets into a list with size 4.

In [89]:
```
print('The length of all_predict_targets:', len(all_predict_targets))
print('Shape of the element in all_predict_targets:', all_predict_targets[0].shape)
```

```
The length of all_predict_targets: 40
Shape of the element in all_predict_targets: (46,)
```

In [107]:
```
import pickle
pickle.dump(all_predict_targets, open('accuracy_time_metrics/all_predict_targets', 'wb'))
pickle.dump(all_test_targets, open('accuracy_time_metrics/all_test_targets', 'wb'))
pickle.dump(all_train_times, open('accuracy_time_metrics/all_train_times', 'wb'))
pickle.dump(all_test_times, open('accuracy_time_metrics/all_test_times', 'wb'))
```

In [90]:
```
flat_predict_targets = all_predict_targets.copy()
flat_test_targets = all_test_targets.copy()
for i in range(4, len(flat_predict_targets)):
    r = i % 4
    flat_predict_targets[r] = np.concatenate((flat_predict_targets[r], flat_predict_targets[i]))
    flat_test_targets[r] = np.concatenate((flat_test_targets[r], flat_test_targets[i]))

flat_predict_targets = flat_predict_targets[:4]
flat_test_targets = flat_test_targets[:4]
```

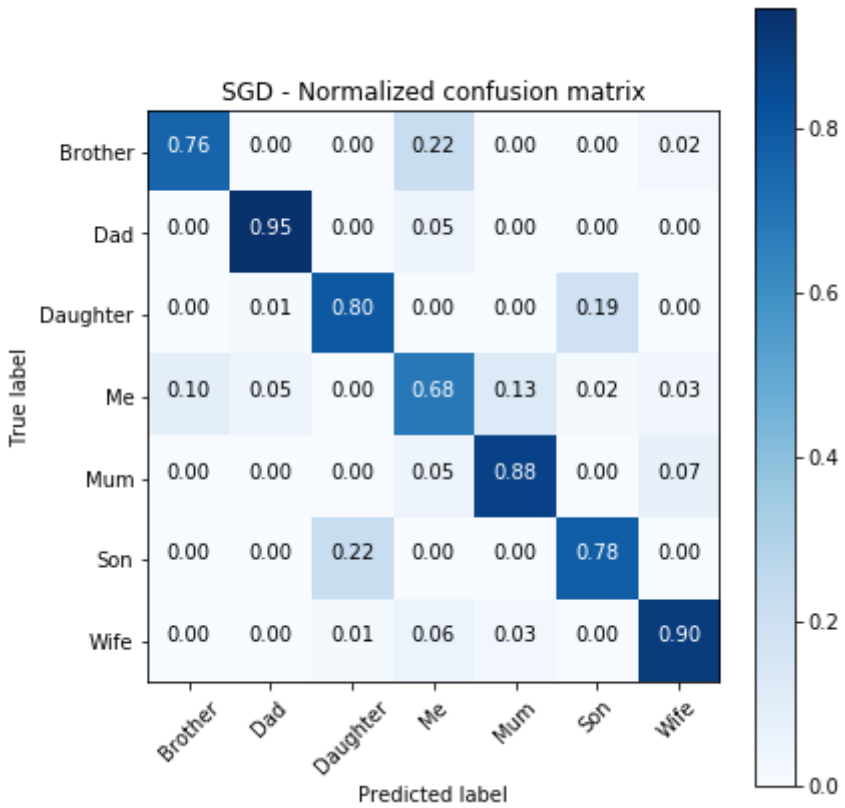
In [91]:
```
score_sgd = get_score_numpy(flat_test_targets[0], flat_predict_targets[0])
score_knn = get_score_numpy(flat_test_targets[1], flat_predict_targets[1])
score_log = get_score_numpy(flat_test_targets[2], flat_predict_targets[2])
score_dnn = get_score_numpy(flat_test_targets[3], flat_predict_targets[3])
```

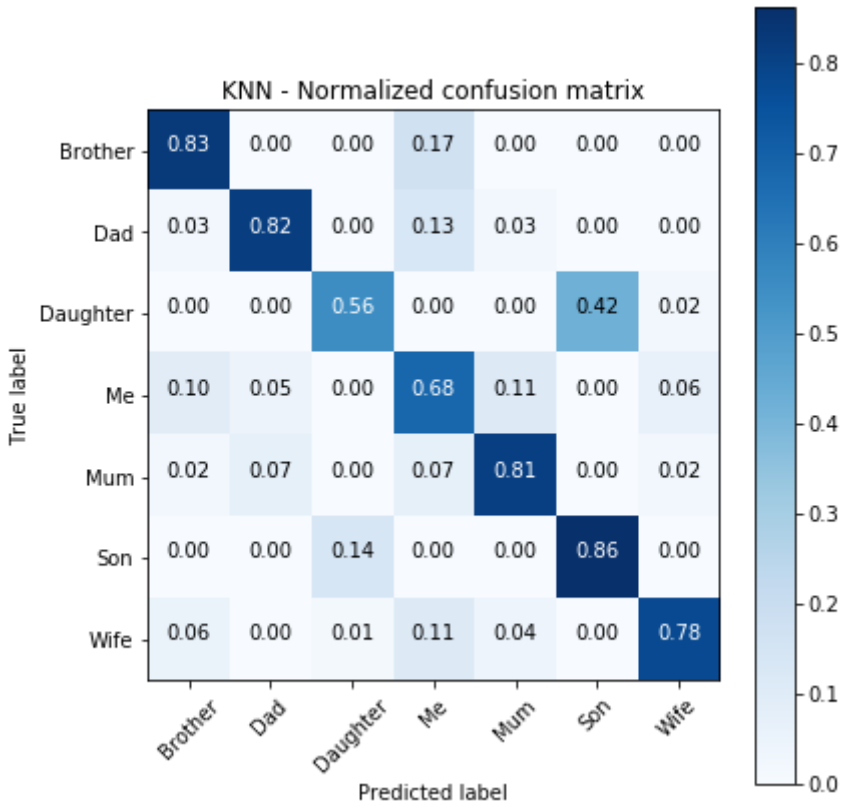
In [92]:
```
cnf_matrix_sgd = confusion_matrix(flat_test_targets[0], flat_predict_targets[0])
cnf_matrix_knn = confusion_matrix(flat_test_targets[1], flat_predict_targets[1])
cnf_matrix_log = confusion_matrix(flat_test_targets[2], flat_predict_targets[2])
cnf_matrix_dnn = confusion_matrix(flat_test_targets[3], flat_predict_targets[3])
```

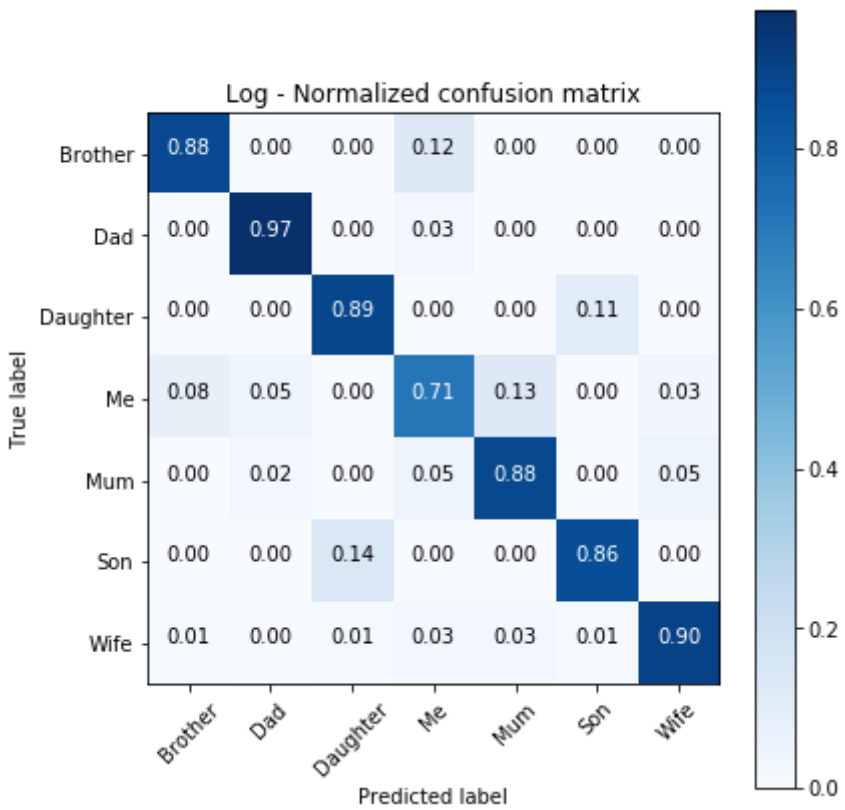
B. Detailed Look into Score

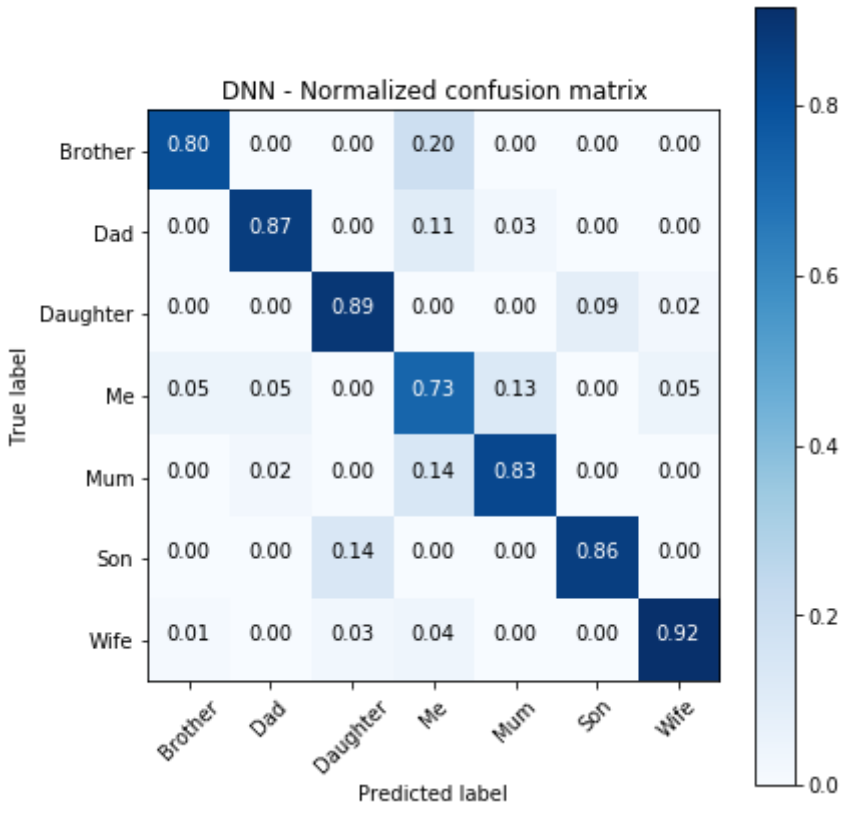
Logistic Regression returned the highest 0.83 of F1 score. And below is the confusion matrix plots (after normalization) of the 4 models. From them we can clearly see the precision and recall rates.

In [96]:
```
plt.figure(figsize=(6,6))
plot_confusion_matrix(cnf_matrix_sgd, classes=face_names, normalize=True, title='SGD - Normalized confusion matrix')

plt.figure(figsize=(6,6))
plot_confusion_matrix(cnf_matrix_knn, classes=face_names, normalize=True, title='KNN - Normalized confusion matrix')

plt.figure(figsize=(6,6))
plot_confusion_matrix(cnf_matrix_log, classes=face_names, normalize=True, title='Log - Normalized confusion matrix')

plt.figure(figsize=(6,6))
plot_confusion_matrix(cnf_matrix_dnn, classes=face_names, normalize=True, title='DNN - Normalized confusion matrix')

plt.show()
```


SGD - Normalized confusion matrix
Brother 0.76 0.00 0.00 0.22 0.00 0.00 0.02
Dad 0.00 0.95 0.00 0.05 0.00 0.00 0.00
Daughter 0.00 0.01 0.80 0.00 0.00 0.19 0.00
Me 0.10 0.05 0.00 0.68 0.13 0.02 0.03
Mum 0.00 0.00 0.00 0.05 0.88 0.00 0.07
Son 0.00 0.00 0.22 0.00 0.00 0.78 0.00
Wife 0.00 0.00 0.01 0.06 0.03 0.00 0.90
Brother Dad Daughter Me Mum Son Wife
True label
Predicted label
0.8 0.6 0.4 0.2 0.0


KNN - Normalized confusion matrix
Brother 0.83 0.00 0.00 0.17 0.00 0.00 0.00
Dad 0.03 0.82 0.00 0.13 0.03 0.00 0.00
Daughter 0.00 0.00 0.56 0.00 0.00 0.42 0.02
Me 0.10 0.05 0.00 0.68 0.11 0.00 0.06
Mum 0.02 0.07 0.00 0.07 0.81 0.00 0.02
Son 0.00 0.00 0.14 0.00 0.00 0.86 0.00
Wife 0.06 0.00 0.01 0.11 0.04 0.00 0.78
Brother Dad Daughter Me Mum Son Wife
True label
Predicted label
0.8 0.7 0.6 0.5 0.4 0.3 0.2 0.1 0.0


Log - Normalized confusion matrix
Brother 0.88 0.00 0.00 0.12 0.00 0.00 0.00
Dad 0.00 0.97 0.00 0.03 0.00 0.00 0.00
Daughter 0.00 0.00 0.89 0.00 0.00 0.11 0.00
Me 0.08 0.05 0.00 0.71 0.13 0.00 0.03
Mum 0.00 0.02 0.00 0.05 0.88 0.00 0.05
Son 0.00 0.00 0.14 0.00 0.00 0.86 0.00
Wife 0.01 0.00 0.01 0.03 0.03 0.01 0.90
Brother Dad Daughter Me Mum Son Wife
True label
Predicted label
0.8 0.6 0.4 0.2 0.0



All the models seem to have slight difficulty to distinguish between my son and daughter.

C. Accuracy and Time

In [93]:
```
score = pd.DataFrame(np.stack((score_sgd, score_knn, score_log, score_dnn)),
                     columns=['Precision', 'Recall', 'F1'], index=['SGD', 'KNN', 'Log', 'DNN'])
score
```

Out[93]:

	Precision	Recall	F1
SGD	0.81377	0.813842	0.813182
KNN	0.75949	0.742243	0.741014
Log	0.866684	0.866348	0.865503
DNN	0.852237	0.849642	0.850502

In [94]:
```
np_train_times = np.vstack(all_train_times)
np_test_times = np.vstack(all_test_times)
```

In [95]:
```
time = pd.DataFrame(np.stack((np.mean(np_train_times, axis=0), np.mean(np_test_times, axis=0))),
                     columns=['SGD', 'KNN', 'Log', 'DNN'], index=['Train (seconds)', 'Test (seconds)']).T
print('Average time for training and testing.')
time
```

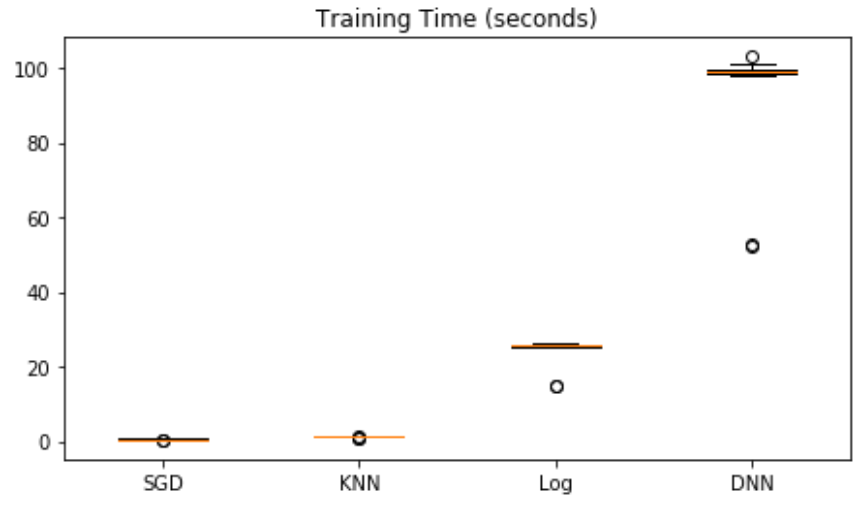
```
Average time for training and testing.
```

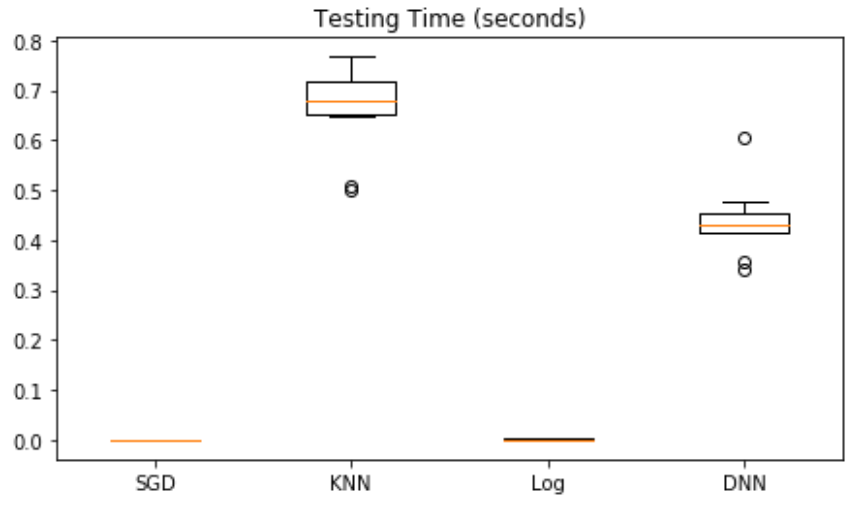
Out[95]:

	Train (seconds)	Test (seconds)
SGD	0.4943	0.0006
KNN	1.1900	0.6588
Log	23.5282	0.0011
DNN	90.3595	0.4382

More detailed look into time

In [97]:
```
fig = plt.figure(figsize=(16, 4))
ax = fig.add_subplot(1, 2, 1)
ax.boxplot(np_train_times)
ax.set_title('Training Time (seconds)')
ax.set_xticklabels(['SGD', 'KNN', 'Log', 'DNN'])

ax = fig.add_subplot(1, 2, 2)
ax.boxplot(np_test_times)
ax.set_title('Testing Time (seconds)')
ax.set_xticklabels(['SGD', 'KNN', 'Log', 'DNN'])
plt.show()
```





SGD Classifier is the fastest on both training and testing. KNN is fast on training but it took the longest time for testing because it needs to compute the distance to all the training points during testing. Logistic Regression takes a long time on training but the testing is super fast and indeed it's fasting on testing. DNN took the longest time for training and its testing time is the 2nd longest.

D. Summary on Performance

The linear model (SGDClassifier) is doing quite ok in the first try and refinement section. It is almost the fastest one on both training and testing. And its performance is the baseline of this project's benchmark.

KNN doesn't perform well in terms of accuracy (F1 score). It also takes a significant longer time on testing.

DNN produced the second highest F1 score. But it took significant longer time on both training and testing.

Logistic regression did the best job on accuracy (F1 score) but took a longer training time. And in my opinion, this is the model has the best balance between accuracy and speed.

VI. Refinement II - Choose Hyper-Parameters

Based on the earlier refinement and benchmark section, logistic regression was determined as the best model. But its parameters are based on the default ones. Maybe accuracy or speed can be further pushed by tuning the hyper-parameters.

In [195]:
```
# Load the list of images and categories
faces, targets = load_dataset('./images')
face_names = [item[9:-1] for item in glob('./images/*/')]
train_faces, test_faces, train_targets, test_targets = train_test_split(faces, targets, test_size=0.3, random_state=0,
stratify=targets)
```

In [196]:
```
# Load the list of images and categories
faces, targets = load_dataset('./images')
face_names = [item[9:-1] for item in glob('./images/*/')]
train_faces, test_faces, train_targets, test_targets = train_test_split(faces, targets, test_size=0.3, random_state=0,
stratify=targets)
```

In [ ]:
```
# Read the images as numpy arrays
train_tensors = paths_to_tensor(train_faces)
test_tensors = paths_to_tensor(test_faces)
```

In [ ]:
```
# Extract features
train_features = get_features(feature_model, train_tensors)
test_features = get_features(feature_model, test_tensors)
print('Train features shape:', train_features.shape, '\nTest features shape:', test_features.shape)
```

In [ ]:
```
from sklearn.grid_search import GridSearchCV
from sklearn.metrics import make_scorer

model_log_base = LogisticRegression(random_state = 0, penalty='l1')
model_log_default = LogisticRegression(random_state = 0)
parameters = {'penalty':['l1', 'l2'], 'max_iter':[50, 100, 200, 500]}
scorer = make_scorer(fbeta_score, beta = 1.0, average='weighted')
grid_obj = GridSearchCV(model_log_base, parameters, scoring = scorer)
grid_fit = grid_obj.fit(train_features, train_targets)

model_log_best = grid_fit.best_estimator_
print('Base model is with L1 penalty and max iteration of 100.\n', model_log_base)
print('\nDefault model is with L2 penalty and max iteration of 100.\n', model_log_best)
print('\nOptimized model is with L2 penalty and max iteration of only 50.\n', model_log_best)

print('\nTime take for training and testing. Base, default and optimized.')
%time base_predictions = (model_log_base.fit(train_features, train_targets)).predict(test_features)
%time default_predictions = (model_log_default.fit(train_features, train_targets)).predict(test_features)
%time best_predictions = (model_log_best.fit(train_features, train_targets)).predict(test_features)

# Report the before-and-afterscores
print("\nBase model\n------")
print("F-score on testing data:", fbeta_score(test_targets, base_predictions, beta = 1.0, average='weighted'))
print("\nDefault Model\n------")
print("F-score on the testing data:", fbeta_score(test_targets, default_predictions, beta = 1.0, average='weighted'))
print("\nOptimized Model\n------")
print("Final F-score on the testing data:", fbeta_score(test_targets, best_predictions, beta = 1.0, average='weighted'
))
```

The f1 score in this section cannot be used to compare with the scores from earlier sections because a different dataset split is used here. This section demonstrates the hyper-parameter tuning of the winning model - Logistic Regression. GridSearchCV was used to find the best combination of `panelty` and `max_iter` . It turns out that the default model already comes with the 'best' hyper-parameters except `max_iter` can be reduced from 100 to 50 to make it run faster without any sacrifice on accuracy.

VII. Conclusion

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

SGD classifier is the base of the benchmark and it reached 81% for f1 score. And it’s the fastest one on both training and testing. If time is a big concern, probably SGD classifier is the best in terms of both accuracy and time.

Logistic regression model has an average accuracy of almost 87% which pushed the boundary by 6%. The only cost is that logistic regression model takes longer time to train.

KNN doesn't perform so well in terms of accuracy. I guess it's due to the curse of dimensionality (https://en.wikipedia.org/wiki/Curse_of_dimensionality#Nearest_neighbor_search). And it’s slow on training, 40 times longer than SGD Classifier.

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. There is one thing very interesting that DNN is the only one who gained more than 10% improvement on the F1 score due to the technique of image generator. It is said that deep learning is taking over ‘traditional’ machine learning models when there is adequate data available and based on this project’s result it’s probably very true.

I will choose logistic regression as my final winning model in this project. It doesn’t show a perfect result (>95%) 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. In addition, the hyper-parameters were also explored, and it turned out the max_iter from logistic regression can be reduced from the default 100 to 50 without the sacrifice on accuracy. This will help improve the time spent for both training and testing.

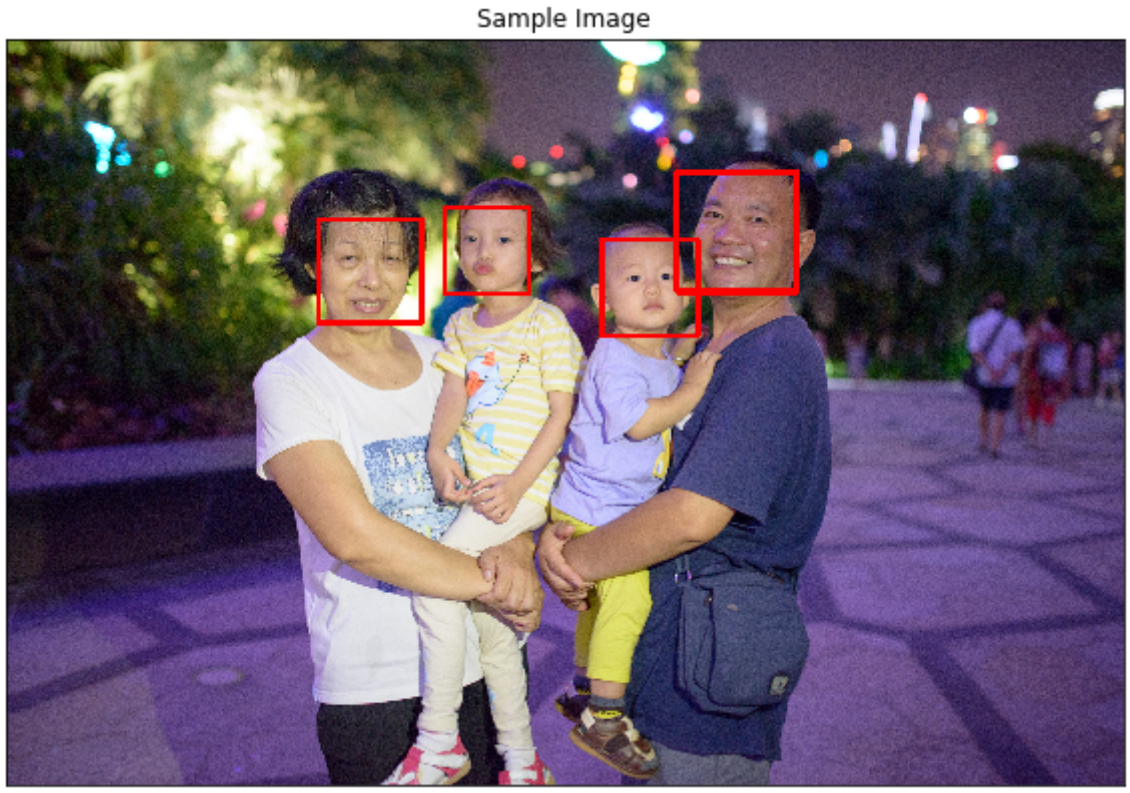
One possible way to improve the model may be using transfer learning to detect the facial key points (centre and corners of eyes, mouth, nose etc.) first. Then build mathematic models to represent the relative positions of the key points. Normalization may be needed to make the face centred and ‘starring’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

VIII. Fun Part - in Real World

There are two family photos `resources/test_1.jpg` and `resources/test_2.jpg` in the root of working directory. Both of them were not used for training or testing in the earlier sections. Let's first perform a face detection using the `process_faces()` .

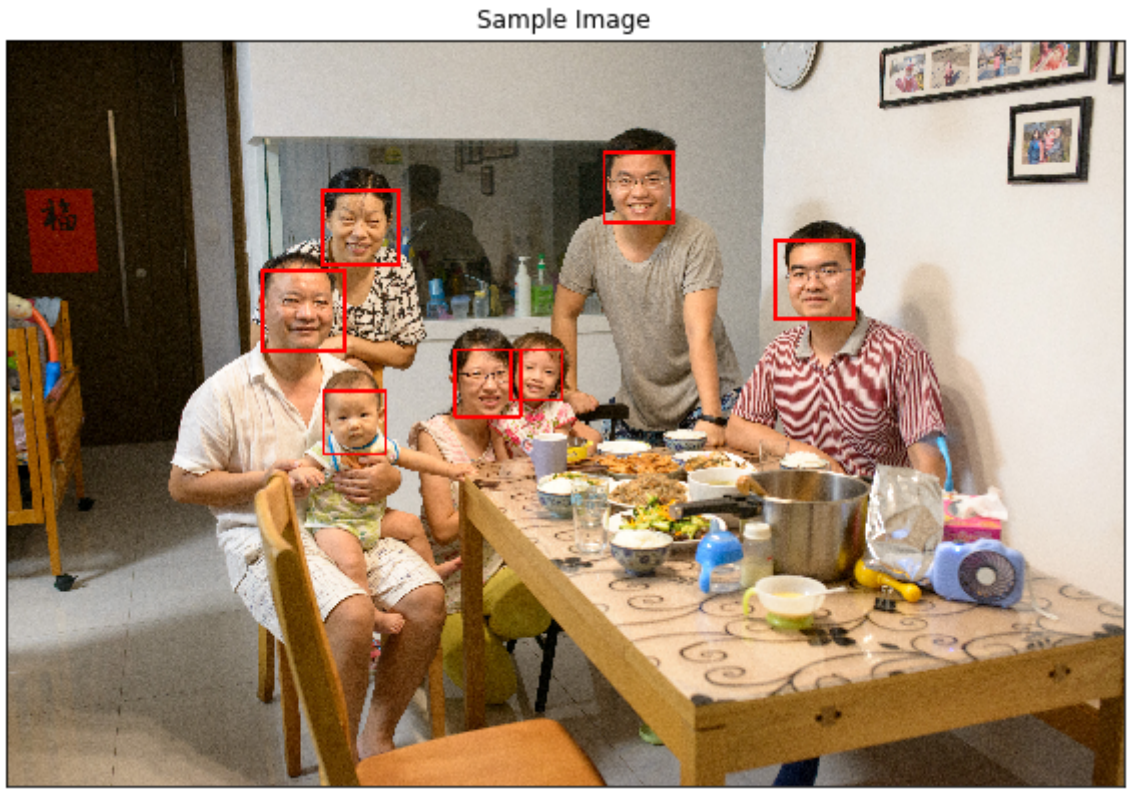
In [99]:
```
process_faces('./resources/test_1.jpg', DISPLAY, NOSAVE, 1.35, 5)
```

```
Image path ./resources/test_1.jpg
Image numpy array shape: (4766, 7141, 3) <class 'numpy.ndarray'>
Number of faces detected: 4
```



In [100]:
```
process_faces('./resources/test_2.jpg', DISPLAY, NOSAVE, 1.32, 7)
```

```
Image path ./resources/test_2.jpg
Image numpy array shape: (4912, 7360, 3) <class 'numpy.ndarray'>
Number of faces detected: 7
```



Now we can train a logistic regression model by using the `./images2` dataset.

In [115]:
```
faces, _ = load_dataset('./images')
generate_images(faces)
```

```
Generating images, 0 of 419 faces
Generating images, 100 of 419 faces
Generating images, 200 of 419 faces
Generating images, 300 of 419 faces
Generating images, 400 of 419 faces
```

In [121]:
```
faces, targets = load_dataset('./images2')
all_train_tensors = paths_to_tensor(faces)
all_train_targets = targets
all_train_features = get_features(feature_model, all_train_tensors)

clf_log = LogisticRegression(random_state=0)
%time model_log = clf_log.fit(all_train_features, all_train_targets)
```

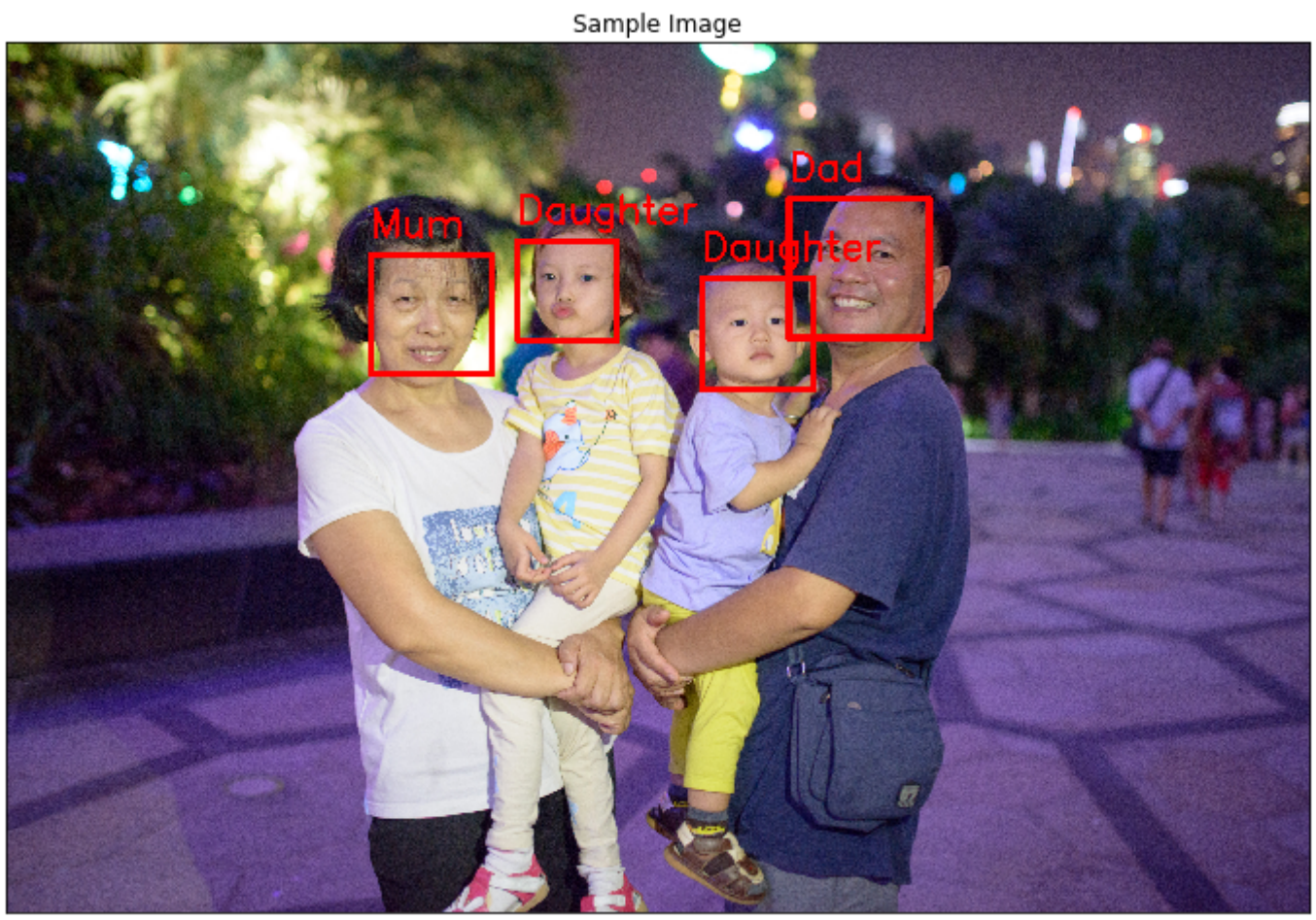
```
Wall time: 16 s
```

In [122]:
```
import pickle
pickle.dump(clf_log, open('models/clf_log', 'wb'))
clf_log = pickle.load(open('models/clf_log', 'rb'))
```

In [123]:
```
def get_name(model, image_face):
    ''' Use the model to make prediction and return the result as the name of the face.
    Args:
        model (sklearn.model): A trained model.
        image_face (numpy.array): A face image in 3D numpy array.
    Returns:
        face_names[predict_idx] (str): The name of the face.
    '''
    # Conver the 3 channel RGB to 4-d tensor and normalize it
    image_face_tensors = image_face.reshape(-1, 299, 299, 3)/255
    image_face_features = get_features(feature_model, image_face_tensors)
    predict_idx = model.predict(image_face_features)[0]
    return face_names[predict_idx]

def process_faces_names(file_path, scaleFactor=1.3, minNeighb=5):
    """Process the input image file by extracting face(s). Draw bounding box and detected name.
    Args:
        file_path (str): The full path of the input image file.
        scaleFactor (float): The scaling factor used by face detection function.
        minNeighb (int): The number of minimum neighbors.
    Returns:
        None: Display the image.t
    """
    print('Image path', file_path)
    image = get_numpy_from_file(file_path)
#     image = cv2.fastNlMeansDenoisingColored(image,None,5,5,7,15)
    faces = get_faces(image, scaleFactor, minNeighb)
    print('Number of faces detected:', len(faces))

    image_with_detections, image_faces = draw_bounding_box(image, faces)

    for i, (x,y,w,h) in enumerate(faces):
        cur_face = cv2.resize(image_faces[i], (299,299))
        name = get_name(model_log, cur_face)

        # Write the returned name on the image
        cv2.putText(image_with_detections, name,
                    (x,y-100),cv2.FONT_HERSHEY_SIMPLEX, 7, (255,0,0),20,cv2.LINE_AA)

    display_from_numpy(image_with_detections, 12, 12)
```

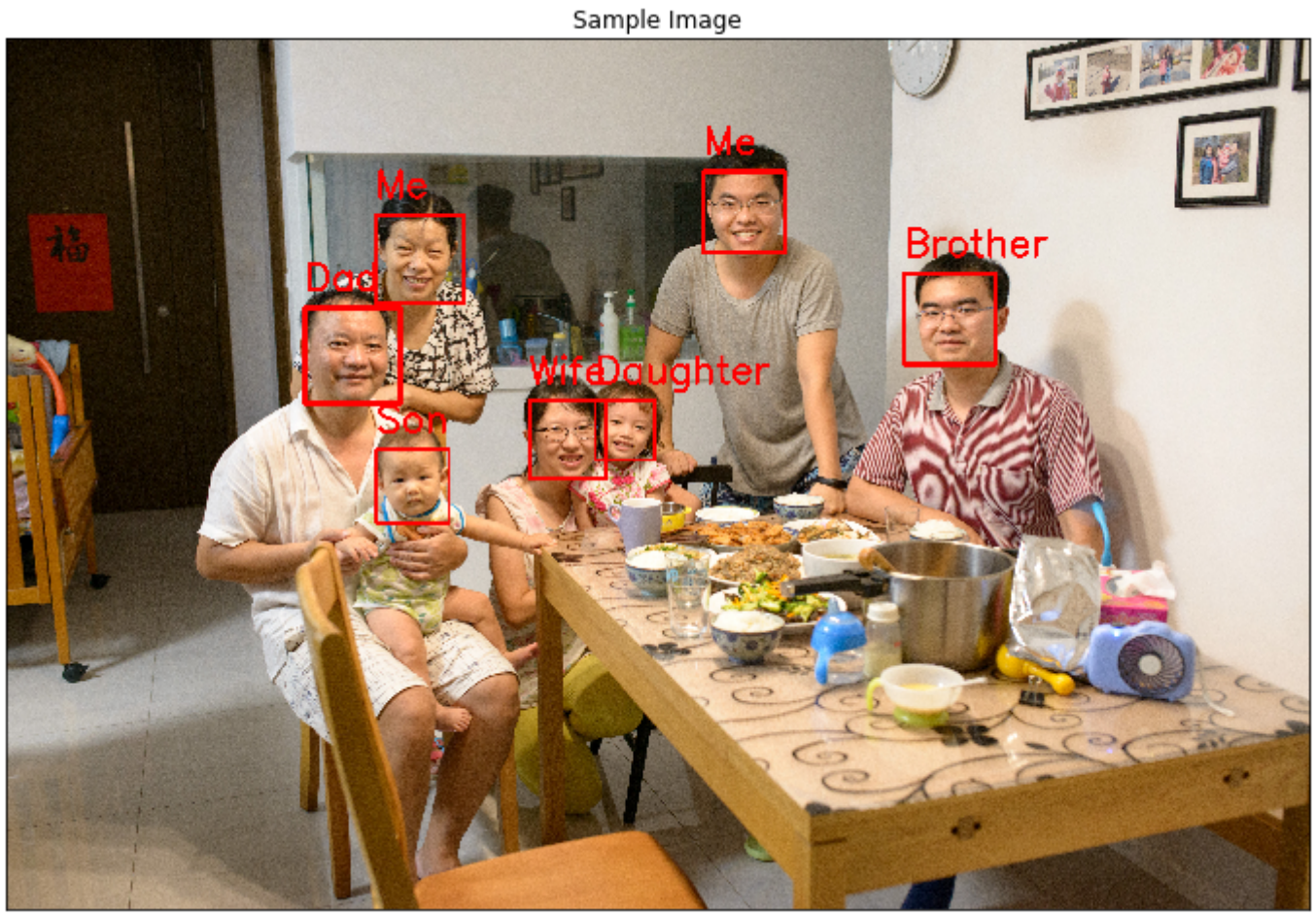
In [124]:
```
process_faces_names('./resources/test_1.jpg', 1.35, 5)
```

```
Image path ./resources/test_1.jpg
Image numpy array shape: (4766, 7141, 3) <class 'numpy.ndarray'>
Number of faces detected: 4
```



In [125]:
```
process_faces_names('./resources/test_2.jpg', 1.32, 7)
```

```
Image path ./resources/test_2.jpg
Image numpy array shape: (4912, 7360, 3) <class 'numpy.ndarray'>
Number of faces detected: 7
```



The result is not perfect, both photos have misclassified face(s). Not good, but not too bad, because we are family and we are born to look similar. Each photo has one face misclassified. Based on the earlier confusion matrix plot, there is no surprise on misclassifying my son and mum.

In [ ]: