Bruno Calogero
Daniel Gonzales

ECE 420 Final Report - EMOTIFY

Github repo: https://github.com/brunocalogero/ECE420FinalProject

Introduction

During the last half of this course, we have planned, designed, and implemented a working Android application: Emotify. Using the device camera, this personalized app learns to recognize the different facial features of a user and will then play music from a playlist corresponding to his or her mood and more concisely his/her facial emotion. Our app uses the Fisherface and Haar cascade classifiers for detecting the user's face and emotion with a relatively high level of accuracy and will be discussed in the bulk of this report. The main algorithm includes Fisher Linear Discriminant Analysis (FLDA) which also makes use of Principal component Analysis (PCA).

Literature/Research Review/semi-technical description

The first and simpler algorithm that we have considered in order to detect faces is the eigenface approach with principal component analysis. This algorithm takes a set of training images in order to find the principal components that correlate with the object (or facial expression) of interest. It then takes a query image to compare to each PCA subspace (for each expression) in order to find the best fit. This is explained in detail in relation to facial expressions in "Eigenface Based Recognition of Emotion Variant Faces".

The method in finding eigenfaces is separated into four steps. The first is to find the mean vector of the training images corresponding to a particular facial expression. This will be used in normalizing each image. The next step is to compute the covariance matrix from the subtracted image vectors. Using this matrix, we can then directly solve for every eigenvalue and its corresponding eigenvector(eigenface). Lastly, we put the eigenvalues in descending order. The importance of this technique is for dimensionality reduction. We want to take the most significant principal components (calculated using the largest eigenvalues) of the training data in order to reduce computation time and storage while still being able to identify the facial expression. Once we have the most significant principal components, we can compute a PCA subspace which will be used to find the euclidean distance between the test images and the query image.

This process is to be done for every set of training images, each set corresponding to a particular facial expression. Once the euclidean distance is found between the query image and every set of test images, the set with the minimum euclidean distance is to be the facial expression of the query image. This forms the basis of the more complex fisherface approach which is explored in the next paragraphs.

"The Principal Component Analysis (PCA), which is the core of the Eigenfaces method, finds a linear combination of features that maximizes the total variance in data. While this is clearly a powerful way to represent data, it doesn't consider any classes and so a lot of discriminative information *may* be lost when throwing components away." For instance in our case we encounter the issue of removing a component that includes two features that distinguish a certain emotion, thus resulting in increased error. Classification for different emotions at this point becomes quite problematic.

This is why our research has led us to consider a more appropriate algorithm method that still relates to our previous evaluation of PCA in terms of lower-dimensional representation. “The Linear Discriminant Analysis performs a class-specific dimensionality reduction”. A wonderful description from openCV allows us to clearly relate our previous approach with the more suitable LDA method, indeed we are trying to classify emotions here, this would be similar to classifying, say, flowers.
“In order to find the combination of features that separates best between classes the Linear Discriminant Analysis maximizes the ratio of between-classes to within-classes scatter, instead of maximizing the overall scatter. The idea is simple: same classes should cluster tightly together, while different classes are as far away as possible from each other in the lower-dimensional representation.” Indeed, we are not comparing classes like human and frogs but a human face with another human face.
We will follow the algorithm proposed by openCV and that proposed in the sourced papers.

1. X be a random vector with samples drawn from $\mathbf{c}$ classes, one vector of images is setup for each class that is labeled during training:

$$X = \{X_1, X_2, \ldots, X_c\}$$
$$X_i = \{x_1, x_2, \ldots, x_n\}$$

2. Scatter matrices S_B and S_W are calculated, these are pretty obvious, we want to maximize the ratio between the “between” class scatter and the “within” class scatter so as to not only recognize a face but also the different emotions of that same face:

$$S_B = \sum_{i=1}^{c} N_i(\mu_i - \mu)(\mu_i - \mu)^T$$
$$S_W = \sum_{i=1}^{c} \sum_{x_j \in X_i} (x_j - \mu_i)(x_j - \mu_i)^T$$

Where , μ is the total mean:

$$\mu = \frac{1}{N} \sum_{i=1}^{N} x_i$$

and, μ_i is the mean of class $i \in \{1, \ldots, c\}$:

$$\mu_i = \frac{1}{|X_i|} \sum_{x_j \in X_i} x_j$$

3. Projection of X onto the rotational matrix W_{opt} in (N-c) subspace where N is the number of samples of X and c the number of unique classes (maximizing the class separability criterion):

$$W_{opt} = \arg\max_W \frac{|W^T S_B W|}{|W^T S_W W|}$$

- Principal Component Analysis on the data and projecting the samples into the (N-c) dimensional subspace. A Linear Discriminant Analysis was then performed on the reduced data, because S_W isn't singular anymore.

$$W_{pca} = \arg\max_W |W^T S_T W|$$

$$W_{fld} = \arg\max_W \frac{|W^T W_{pca}^T S_B W_{pca} W|}{|W^T W_{pca}^T S_W W_{pca} W|}$$

- Final Transformation matrix W is easily computed, this projects a sample test image into the (c-1) dimensional space given by:

$$W = W_{fld}^T W_{pca}^T$$

To visualize this, the following figure further explains the different clusters formed and show the Scattering and use of means to form the space onto which we project our images, this example was given to us courtesy of Kristian Lauszus.

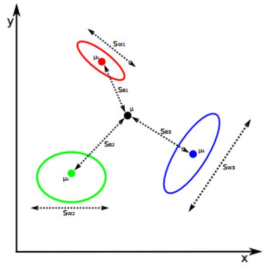


Figure 1: FLDA example

Technical Description

The technical flow of our application is as follows: Upon opening the app, the user is presented a front camera view with a button at the bottom to take a picture. Once the user presses the button, the image will be sent through four different Haar cascade classifiers for general face detection. The model for each classifier is trained on different sets of a face variation, effectively increasing the accuracy of detection. If no face is detected in the image, the user will be prompted to take the picture again. Once the face is detected, the image will continue to be processed.

Next, the image goes through preprocessing to improve classification accuracy. Using the first detected face from the cascade classifiers, the image will be cropped down to just the face. It is then equalized using contrast limited adaptive histogram equalization (CLAHE) in order emphasize facial features. This output is then resized to a 200x200 pixel image to reduce computation time while keeping the quality of features. This preprocessing is done to improve the quality of images going into our training set for facial expression detection and has effectively increased the accuracy of our classifier overall. A figure of the preprocessing effects is shown below.

Without preprocessing

With preprocessing

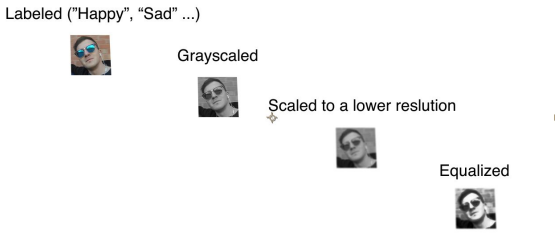
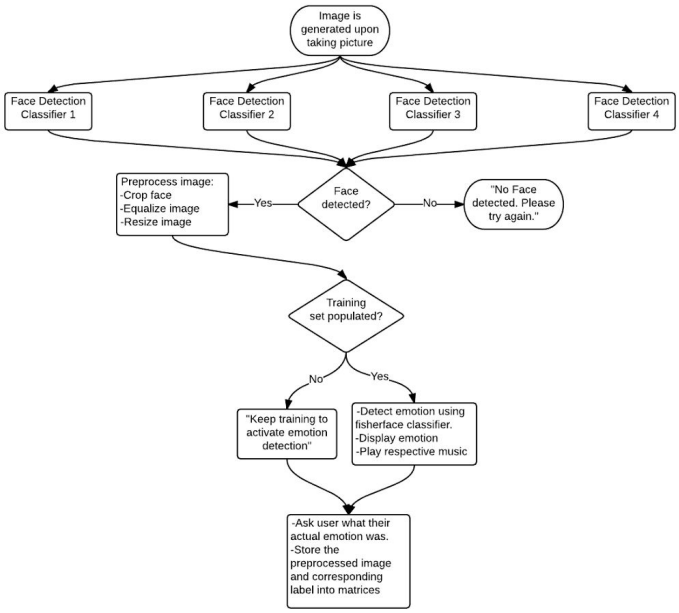


Figure 2: Preprocessing steps

Before our app can begin detecting facial expressions, the training set must adequately be trained: there must be at least two emotion labels (i.e. happy and sad) with a minimum of two images in each labeled class. Until then, the user will be prompted to take pictures and train the app by telling it what faces are made until the criteria is met. With each picture taken, the user will be prompted to select the actual emotion made from an existing class or they can select a new class emotion to be added to the training set. The images and labels are then stored in respective vectors where they will then be used as inputs for the fisherface classifier. After the training set has been adequately trained, the detected emotion will be outputted on the bottom of the screen upon each picture taken and the respective music will begin to play. A high level representation of this technical flow is shown in the figure below.



The music selected for each emotion is taken from the music in a playlist on the user's music app. Our app communicates with the music app by looking for the playlist with the same name as the emotion detected. The music app must be running in the background in order for this to work.
The user also has the ability to look at the statistics of the training set by selecting the info button at the top left corner. For each emotion label, the user can see how many images are used for that class. This allows the user to see if there is an emotion that is undertrained compared to others which can cause bias in the classifier. The user also has the option to clear the training set, allowing them to retrain the app or have a different user train the app to their face. Statistics of a trained classifier are shown in the figure below.

Figure 3: Emotify App Training Set Visualization

It is also worth mentioning the whole C++ workflow for the FLDA algorithm once again that has been implemented with great help from Kristian Lauszus after our realization that OpenCV wasn't so android friendly after all (this will be talked about in the results section).

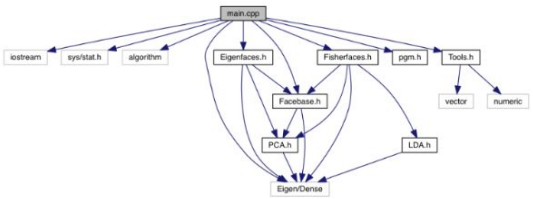


Figure 4: C++ code distribution

Results

We will be limiting ourselves to only the major issues and important results otherwise this section would be endless.
The results upon completion of our final project have developed through multiple iterations of the app design. One of the first challenges we faced was adequate accuracy in emotion detection. We initially decided to have our fisherface classifier use training data from a general emotion database: the Cohn-Kanade database. While this database was rich in the amount of images and emotions, the accuracy with the fisherface algorithm was poor in practice. It would detect certain emotions at less than 50% accuracy and other emotions would not be detected at all for certain users. Because of the personalized nature of our app, we were motivated to switch from using a general database to a training set that the user will train themselves. Not only did we see a dramatic increase in accuracy, but we also achieved a less complicated classifier and a thus a much better UX (user-experience).
Even with increased accuracy using a personalized training set, the classifier still had difficulty detecting emotions, especially after moving to environments with different lighting. To fix this, we employed facial detection cropping and histogram equalization. By cropping the image to only the face, most of the background of the image is removed so it will not bias the classifier as much. The use of equalization evenly distributes the contrast of the image, making the images more similar when there are changes in lighting or environment. Adding this preprocess to our design gave another significant boost in accuracy. For perspective, without preprocessing, our app struggled to identify between three emotions with accuracy higher than 60%. With preprocessing, the app does this quite well with less training and is more consistent in different environments (especially with different lighting). The current accuracy is of around 70 to 90% depending on the training by the user which is clearly a great result for our application. The number of false predictions is really the only type of errors we looked up, some false-negatives have also been an issue in bad lighting, the latter being the lack of detection of a face even though it is there.

It is also worth mentioning the enormous amount of problems we have had trying to use the infamous JavaCV wrapper (java wrapper for OpenCV in android). The latter was incredibly incomplete and resulted in us losing a week of research and development due to how hard it was to set up. Once we managed to setup the latter it was clear that it was deprecated and creating an object for say a "fisherface

class" would result in an instant crash. Please contact us if you have managed to implement JavaCV successfully on Android Studio, We'd be extremely curious. However, this resulted in us having to implement the whole LDA from scratch which we guess was a much better thing, but this also meant an extra 400-500 lines of code, indeed we plotted a nice code distribution as can be seen below, representative of how much android development was learnt in the process:

Figure 5: Code Distribution for EMOTIFY (lines are numbers in parentheses)

Extensions/Modifications

- Multiple users Implementation
- Facial landmarking for better emotion detection
- Buffer out bugs
    - Plays music once sometimes
    - Needs an extra image put into training set when restarting app (although classifier is already trained)
    - App crashes when there are many classes with little images in each class
    - Replay playlist and have the user go to the next song
    - Instead of pausing the song and playing another song when another emotion is detected have the new song play as the current version does but the previous song stop instead of pausing.
- Different modules
    - Emoji corresponding to emotion on detected face with object tracking
    - Web recommendations corresponding to mood.
    - Implementation within other applications
- App works in the background
- Implement app with Spotify API