A Practical Application of Face Recognition on iOS

Shayan Yousefizadeh
B.A. (Mod.) Computer Science
Final Year Project - April 2015
Supervisor: Dr. Fergal Shevlin

School of Computer Science and Statistics
O’Reilly Institute, Trinity College, Dublin 2, Ireland
DECLARATION

I hereby declare that this project is entirely my own work and that it has not been submitted as an exercise for a degree at this or any other university.

______________________________________   ________________________
Name                                           Date
ABSTRACT

This is an investigation into the face recognition algorithms available on iOS. I built a checkin system that uses face recognition to identify the person checking in and I evaluated how each algorithm performed for this particular application, and finally concluded on the best available algorithm and the overall state of face recognition.

In addition, I built a drop-in library for face recognition on iOS that uses native Objective-C API to allow for easy adoption for new developers. I wrote an Objective-C wrapper around any C++ / OpenCV functions used so users don’t need any prior C++ or OpenCV experience to use it.
ACRONYMS

UI  User Interface
UX  User Experience
IDE  Integrated Development Environment
API  Application Programming Interface
PCA  Principal Component Analysis
LDA  Linear Discriminant Analysis
LBPH  Local Binary Patterns Histogram
SDK  Standard Development Kit
ARC  Automatic Reference Counting
DEFINITIONS

Objective-C
A language developed by Apple that is a superset of C. It is used for native development of iOS and OS X applications.

Face Detection
Detecting a region of pixels in an image that represents a possible occurrence of a face.

Face Recognition
Recognising a detected face in an image by distinguishes it from a dataset of face it is trained to recognise.

Xcode
A very powerful IDE provided by Apple for iOS development.

iOS
A mobile operating system used exclusively by Apple on their mobile devices (iPhones & iPads). The latest version at the time of writing this is iOS 8.3.

Wrapper
A wrapper around a library is a thin layer of code that allows you to interface with the library using a different language.

Dictionary
A key-value data structure, where the value can be retrieved by looking up the key.
TABLE OF CONTENTS

1. Introduction 3
   1.1 Motivation 3
   1.2 Practical Application 3

2. Background 4
   2.1 Objective-C & Objective-C++ 4
   2.2 OpenCV 4
   2.2 Cascade of Haar Classifiers 5
   2.3 AV Foundation 5
   2.4 Core Image 6
   2.5 Core Graphics 6

3. Design 7
   3.1 Drop-in Face Recognition Library 7
      3.1.1 Region of Interest 7
      3.1.2 Face Detector 8
      3.1.3 Eye Detector 8
      3.1.4 Face Recogniser 9
      3.1.5 Image Processing & Conversions 9

4. Implementation 11
   4.1 Face Detection 11
   4.3 Face Pre-processing 11
      4.3.1 Convert to Greyscale 11
      4.3.2 Equalise Histogram 11
      4.3.3 Geometric Transformations 12

5. The App 13
   5.1 Overview 13
   5.2 List of Teams 14
   5.3 Team Profile 16
1. INTRODUCTION

Face recognition is the process of putting a name to a known face. Similar to how humans can learn to recognise people by their face, machine learning algorithms exist that allow computers to learn to recognise a known face. This project aims to investigate face recognition and evaluate the performance and accuracy of any available algorithms on iOS. This goal will be achieved through developing an iOS application that makes use of face recognition practically. Finally I will develop a library to make performing face recognition on iOS easier.

1.1 Motivation

Mobile processors have only just become powerful and power efficient enough to perform vision problems well. iOS lacks any native framework for computer vision algorithms, which forces users to use OpenCV. This requires prior knowledge of C++ and OpenCV from the user as well as additional code to convert between OpenCV and iOS formats.

This was my motivation to try to find out what (if any) iOS currently offers for face detection and recognition. I wanted to see how much of the process could be done natively without needing to use OpenCV, and where OpenCV was needed I wanted to write a wrapper around it so new developers could have access to a drop-in solution to face recognition with an Objective-C API.

1.2 Practical Application

Jailbreak HQ is a student run charity event where teams from different universities in Ireland race to a mystery location somewhere in the world. Each team consists of two students and they must update their location every three hours by performing a checkin. Before my proposed solution checkins were done through phone calls, a manual task with no way of verifying the person’s location or identity. My proposed solution allows participants to checkin by simple looking into their phone’s camera. It uses face recognition to identify the participant and their phone’s GPS to verify their location, for a reliable and automatic checkin.
2. BACKGROUND

This chapter will explain the technologies used in this project to familiarise the reader with the relevant information needed for a sound understanding of the project.

2.1 Objective-C & Objective-C++

Objective-C is a superset of C, this means it can do everything C can. We can easily mix C code with Objective-C code. Similarly we can mix C++ in with Objective-C code, this is called Objective-C++. The Apple LLVM compiler can compile any C/C++/Objective-C files, and it will treat any file with the .mm extension as an Objective-C++ class (as opposed to the traditional .m extension).

When writing Objective-C++ we need to be careful about a few things. One such thing is memory management. Objective-C uses something called Automatic Reference Counting (ARC) to manage memory, this works by keeping a count of how many things are referencing an object. As long as an object has at least one strong reference to it, it will stay in memory, when the strong reference count drops to zero, the object will be deleted and the memory freed up. This however doesn’t work for Objective-C++ objects, therefore we have to be careful to ensure dealloc deletes the C++ object properly or else we’re leaking memory.

2.2 OpenCV

OpenCV is an open-source computer vision and machine learning library, originally developed by Intel in 1999. It contains thousands of optimised algorithms, there are so many that they are split into modules [9]. OpenCV offers face detection and face recognition, however they involve a fair bit of setup before working. In particular, OpenCV’s face recognition algorithms don’t perform any pre-processing on images to minimise error and optimise accuracy, this is our job to do. This project uses the C++ interface, there are also Python and Java interfaces.

Mobile support for OpenCV is fairly new. Support for iOS was only introduced in version 2.4.2 on 04/07/2012. Unlike Android (which uses Java), there is no way to interface with OpenCV using Objective-C (the language used by iOS). This means iOS projects need to use the C++ interface, this is somewhat problematic.

One problem is the different types and data structures between OpenCV and iOS. How iOS represents images (UIImage) is different to OpenCV (Mat). UIImage is a very high-level representation on an image and unlike Mat it doesn’t supply use with the low-
level pixels and information of the image. This means we have to manually convert between the two formats (using Core Graphics, see section 2.6).

Another problem is that some versions of OpenCV exhibit bugs. For example the latest version 2.4.11 doesn’t build properly in an Xcode project, you must revert to version 2.4.9 for it to work. Similarly some versions before 2.4.9 do not support the ARM 64-bit architecture (which was introduced with the iPhone 5).

2.2 Cascade of Haar Classifiers

This is a technique for object detection as presented by Viola and Lienhart. [5, 11] It learns to identify objects based on a number of positive and negative samples. Once trained the classifier is organised into cascades (a sequence of steps) and if the sub-image is rejected at any stage in the cascade it will not continue. This makes it very computationally efficient as in most cases negative sub-images are rejected after the first couple of stages in the cascade.

OpenCV uses this technique for feature detection (faces, eyes etc) and comes with pre-trained classifiers. It is able to detect features of different scales, this is possible because the classifiers are designed to be easily resized. To perform multi-scale detection we first need to specify a minimum size for the feature and a scale factor. The minimum size is in pixels and tells us what the smallest possible size for a feature can be and the scale factor tells us how many different scales of the feature to try. During detection, for example in a 400x400 image with a 20x20 feature size, the classifier will loop through every 20x20 sub-image in the 400x400 image, perform the necessary classification and see if the sub-image contains the feature. This loop terminates early when a sub-image returns positively for the object. The smaller the feature size is relative to the image size the slower the detection is, because there are more combinations of positions it has to try.

2.3 AV Foundation

AV Foundation is one of iOS’s largest and most powerful frameworks. [6] It allows for creating, editing, encoding, decoding, and playback of media files, as well as capturing content form the camera and microphone. AV Foundation gives us full access to capture devices such as the camera, by creating a capture session we can get the full uncompressed stream of data coming from the camera and use this however we want, for example we can capture a particular frame from the stream and output it into a compressed JPEG still image. One of the interesting things we can do is tell AV
Foundation to detect faces from this capture session, which will notify us whenever it
detects a face object.

2.4 Core Image

Core Image is an Objective-C framework that is included as part of the iOS SDK [8].
It is an image processing and analysis technology designed to provide near real-time
processing for still and video image, with the option of rendering on the CPU or GPU. As
of iOS 5 it offers an API for face detection, with the option to identify facial features
such as eyes and mouth as well as detecting smiles and face angle. It’s a lot friendlier to
use in contrast to OpenCV as it’s not using C++ and much of the work is abstracted
away.

2.5 Core Graphics

Core Graphics is an Objective-C framework that is part of the iOS SDK [7]. It
provides a low-level API for drawing graphics and creating or displaying bitmap images.
Core Graphics doesn’t provide any image processing functions, for example there is no
function to simply resize, crop or rotate an image. What Core Graphics does is provide a
bitmap context that we can draw pixels into, as well as giving us direct access to the
image data. We can use this information to for example resize an image ourselves. This
requires a full understanding of how images are encoded otherwise we are prone to
unexpected results. This includes understanding things such as colour space, bits per
component (how many colours each channel can represent, usually 8-bit meaning 255
values per channel) and byte encoding (whether the array of data for the image is RGB,
BGR, etc.) and more.
3. **Design**

This chapter will examine the design of my library for face recognition on iOS as well as the practical application of face recognition in the app.

3.1 **Drop-in Face Recognition Library**

This is the library I wrote to handle face recognition on iOS. The main focus is usability, the interface should be familiar to the user, this is achieved with the use of meaningful function names that adhere to coding and naming conventions in Objective-C. The library is structured as follows:

3.1.1 Region of Interest

*JBRegion* is a class that holds the relevant information of a region of interest (ROI). ROI is a rectangular subsection of an image as demonstrated in the diagram below. For example if our ROI is a face in an image, we have direct access to the cropped face image through `regionImage`, the original full image through `parentImage` and the bounds of the face in the original image through `rectInParentCoordinateSpace`.

The class also offers a convenient function for drawing boxes inside an image and returning it as a new image, for example we can pass it our original image of two people, and an array containing two *JBRegion* objects representing the two faces in the image, and `drawRegions:inImage:` will return an image with a box around the detected faces.

```cpp
@interface JBRegion : NSObject
@property (nonatomic, strong) UIImage *regionImage;
@property (nonatomic, strong) UIImage *parentImage;
@property (nonatomic, assign) CGRect rectInParentCoordinateSpace;
@property (nonatomic, strong) JBRegion *leftEye;
@property (nonatomic, strong) JBRegion *rightEye;
+ (UIImage *)drawRects:(NSArray *)rects inImage:(UIImage *)image;
+ (UIImage *)drawRegions:(NSArray *)regions inImage:(UIImage *)image;
```

![Diagram](image-url)
3.1.2 Face Detector

This class offers three simple class initialisers for front-facing and side-profile faces, as well as one that can detect both. The first two use OpenCV behind the scene while the second uses Core Image. With just two function calls the user can setup a face detector and start using it. The function \texttt{detectFacesInImage} is used to find multiple faces in an image, and returns an array of \texttt{JBRegion} objects containing the faces. Alternatively the function \texttt{detectFaceInImage} will only detect the one (largest visible) face.

@interface JBFaceDetector : NSObject
- (JBRegion *)detectFaceInImage:(UIImage *)image;
- (NSArray *)detectFacesInImage:(UIImage *)image; // of type JBRegion
+ (instancetype)frontFaceDetector;
+ (instancetype)profileFaceDetector;
+ (instancetype)frontOrProfileFaceDetector;
@end

3.1.3 Eye Detector

This class offers eye detection with just two function calls. The function \texttt{detectEyesInImage} returns a dictionary containing the left and right eye and the estimated regions. This dictionary can be easy searched using the constant keys declared below, e.g. \texttt{JBEyeDetectorLeftEye} to get the left eye. Note when we refer to the left eye it is the eye located on the right side of the image, because it is mirrored!

extern NSString *const JBEyeDetectorLeftEye;
extern NSString *const JBEyeDetectorRightEye;
extern NSString *const JBEyeDetectorLeftEyeEstimate;
extern NSString *const JBEyeDetectorRightEyeEstimate;

@interface JBEyeDetector : NSObject
- (NSDictionary *)detectEyesInImage:(UIImage *)image;
+ (instancetype)eyeDetector;
@end
3.1.4 Face Recogniser

This class offers three simple class initialiser for creating a face recogniser using PCA, LDA or LBPH. After choosing our algorithm we can check to see if our model can be loaded using `canBeLoaded`, if successful we can load it easily, otherwise we can train the model and save it afterwards using the built in functions `train` and `save`. The function `predictForImage` is used to recognise an image, it returns a dictionary containing the label (who it is), confidence and in the case of an error the error. These can be extracted easily using the constant keys declared below.

Finally the function `preprocessedFaceForRegion` is used to return pre-processed version of the face where the eyes are horizontally level, a standard distance apart and in some fixed position in the image, as well as being in greyscale with an equalised histogram.

```objective-c
extern NSString *const JBFaceRecognizerConfidence;
extern NSString *const JBFaceRecognizerLabel;
extern NSString *const JBFaceRecognizerError;

@interface JBFaceRecognizer : NSObject
- (void)save;
- (void)load;
- (void)train;
- (BOOL)canBeLoaded;
- (NSDictionary *)predictForImage:(UIImage *)image;
@end
```

3.1.5 Image Processing & Conversions

This class is a category on UIImage, it means it extends UIImage’s existing collection of functions with the one’s below. All the code in this library was written by me and this class is no exception.

The function `cvMat` can be used to convert a UIImage into OpenCV’s image format `Mat`, `cvMatGrayscale` goes a step further and also converts the colour space to greyscale. This is useful and indeed necessary for anyone wanting to do any image operations using OpenCV. And vice-a-versa `imageWithMat` can be used to convert from OpenCV’s `Mat`
format to iOS’ image format. This is useful and indeed necessary for displaying the result
of any image operation on screen (see 4.5 for implementation).

The function *equalizedHistogram* is self explanatory and distributes the intensity of
the grey-scales in an image, improving it’s overall luminance. Similarly *grayscaleImage* is
self explanatory.

@interface UIImage (JBAdditions)
-
(cv::Mat)cvMat;
-(cv::Mat)cvMatGrayscale;
-(UIImage *)equalizedHistogram;
-(UIImage *)grayscaleImage;
+(UIImage *)imageWithMat:(cv::Mat)mat;
@end
4. IMPLEMENTATION

This chapter will describe the face recognition part of the app, step by step, from capturing video from the device’s camera to recognising any detected faces from it.

4.1 Face Detection

Using AV Foundation we create a capture session from the front-facing camera. This displays the live video stream coming from the camera to the user. Using a built in method we are notified when a face object appears in the video, we wait until the camera has focused and adjusted it’s white balance and exposure and then we output this frame into a still image. The built in method is only useful for notifying us of a face, but it isn’t accurate enough for face detection.

Next we perform face detection on this still image we have just captured. From the face sub-image we perform eye detection. We’re now ready for pre-processing.

4.3 Face Pre-processing

Once we have a sub-image that represents a face we must perform some pre-processing on it. Pre-processing is a series of operations applied to the image that will be useful in improving the face recognition accuracy later on. These are the steps I devise for my pre-processing:

4.3.1 Convert to Greyscale

We must convert our images into the greyscale colour space before using it for training or recognition. There is no built in method on UIImage to do this so we have to do this manually. This is done by drawing the pixels from the original image into a graphics context with a greyscale colour space and finally removing the alpha channel from the bytes, so the image data is encoded as RGB instead of RGBA.

4.3.2 Equalise Histogram

In real-world conditions we will encounter images where there is shadow or direct light on the face, causing it to be very dark or very bright respectively. Since face recognition algorithms are very sensitive to changes in illumination [2] we need to alleviate this. One way to do this is by equalising the histogram of the image.

A histogram (in our case a 1 dimensional histogram since our image is greyscale) is the frequencies of each greyscale intensity (0-255 representing pure black to white).
Histogram equalisation is a technique for improving the distribution of grey-scales in an image. \[1\] After applying equalisation we minimise the difference in illumination across images, so the dark and bright faces in an image will more closely resemble each other, which in turn improves accuracy of the recognition.

To perform histogram equalisation we use the Accelerate framework \[4\]. It uses a C API but provides a fast vectorised implementation. We first convert our UIImage into a lower level format called vImageBuffer. Then we apply the histogram equalisation using the vImageEqualization_Planar8 function. It is important to use Planar8 since we are in the greyscale colour space and this means our buffer contains a single colour channel encoded as an array of packed unsigned chars.

4.3.3 Geometric Transformations

The face sub-image returned by the face detector is prone to variance with regards to the face position. This means that when using two images of the same person, the detected face sub-images will not align perfectly if we were to stack them on top of each other. Furthermore the head in an image could be tilted slightly and in general there is too much variance that can cause different parts and features of faces to not line up. We can end up comparing an eye from image1 to an eyebrow in image2 if this misalignment happens, obviously this will negatively affect our recognition.

The solution to this is performing some geometric transformations on the image to minimise variance between all images. The way I implemented this was heavily reliant on accurate eye positions. First we can alleviate head tilt by getting the angle between the two eyes and rotating the image so they are perfectly horizontally level. Next we can measure the distance between the eyes to make sure they are always the same distance apart (and hence always in the same x-axis position in the image). This is done by scaling the image by the desiredDistance / actualDistance. Finally we choose to crop the face to remove the background and forehead.

![Fig. 1. Face pre-processing correcting head tilt and aligning eyes.](image)
5. THE APP

This chapter will demonstrate the iPhone & iPad app that I built for Jailbreak HQ. We will not go into any detail about the code as there is a lot of it and it would take focus away from the face recognition part of this report, however every aspect of the app was designed and developed by me.

5.1 Overview

The app gets its data by interacting with an API, the API allows us to communicate with the backend, this is a server that stores the database of information. We can call the API and request data from, for example we can make an API call to request the list of all teams participating in Jailbreak HQ. These requests are forwarded by the API to the backend, which then figures out what to do and compiles a list of the teams and all their information for us, it then sends this response through the API call back to us. We then take this data and manipulate and display it on our app.

The Jailbreak API was built by Kevin Baker and not me. The API plays no part whatsoever in the face recognition, that part is completely separate and all done locally on the device. The API is just the datasource for the app, it provides the content for the app to show, everything else is all my own work.
5.2 List of Teams

Fig. 2. List of Teams Prototype 1

Fig. 3. List of Teams Prototype 2
Fig. 4. List of Teams Prototype 3

Fig. 5. List of Teams Final Design
5.3 Team Profile

This part of the app displays information about the team, everything from their current location to amount raised, donations, bio, YouTube video and finally by clicking on the map an annotated map with each checkin and a graph of their route.

Patrick & Enda
48th Place
NUIG

First year engineering students in NUIG. We hope to break the stereotype of boring unadventurous engineers by doing Jailbreak barefoot!

About Patrick & Enda

Donations

<table>
<thead>
<tr>
<th>Name</th>
<th>Amount</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alma hynes</td>
<td>€30</td>
</tr>
<tr>
<td>Bucket</td>
<td>€55</td>
</tr>
<tr>
<td>Mr. Costello</td>
<td>€20</td>
</tr>
<tr>
<td>Dermot Connolly</td>
<td>€30</td>
</tr>
<tr>
<td>Ciarán Kelly</td>
<td>€10</td>
</tr>
<tr>
<td>Aideen Kelly</td>
<td>€10</td>
</tr>
</tbody>
</table>

Donate
5.4 Unified Feed

This was the most challenging part of the app. I built a news feed that display a variety of posts from Twitter, Facebook, Instagram, Vine, and Jailbreak’s own custom checkin and news posts. Each cell looks different and has different functionality and managing this is quite some work with `UITableView`. Additionally iOS doesn’t natively support table cells with dynamic height, that means the cells don’t grow based on the amount of text. I had to manually write UI logic to intelligently size the cell’s height based on the text. This was challenging as the cells in iOS’s `UITableView` are re-usable, that means for a table of 100 cells it does not create 100 cells, instead it creates a couple and reused them. This makes smooth scrolling difficult as the heights can’t be pre-calculated, instead they are calculated as you scroll, because we are simply re-using the same object we have to update the height each time instead of pre-allocated all cells with the correct height. The height calculation needs to be fast, thankfully I managed to pull this off.

Apart from the UI the feed was able to persist it’s content for offline reading, would remember the reading position of the user, intelligently refresh it’s content and more. There was also a lot of interacting with the Twitter and Facebook APIs, this allowed users to favourite or like posts within the app. I also wrote a nifty Vine player using AV Foundation. The Vine videos would auto-play and auto-loop.
Benedict & Thomas
Had a bunch of black guys in a twerk circle is probably the two releases that got me properly into techno

Benedict & Thomas
Arca was definitely a hater.

Benedict & Thomas
Jack & Jill have just left Europe.

Donate Now

Benedict & Thomas
I've watched so many Liam Neeson movies. Another two long days of progress.

Benedict & Thomas
"Going east!"

- Dijon, France

Ibrahim Halawa, an Irish citizen, is a prisoner of conscience, held in a prison in Egypt detained solely for peaceably exercising his right to freedom of expression and assembly. A new trial has been scheduled to start on 8 February. If convicted he could face the death penalty.

Sign Petition Now
5.5 Checkin

The checkin system uses the front-facing camera of a device to verify a person’s identity. There is no need to take a picture, it simply performs everything in real-time on the live video stream. As you can see I’ve added some additional checking to improve results. For example as show by Fig. 6 it will ensure front-facing images by telling the user to look directly into the camera if they aren’t. Fig 7. shows the app checking for tilted heads, while the face pre-processing alleviates this it is better to prevent extreme angles of head title. Finally Fig 8. demonstrates when something is obstructing the users eyes, this sort of feedback to the user is very important.
6. EVALUATION

All tests were performed on an iPhone 6 which features an Apple A8, a 1.4GHz dual-core 64-bit ARM processor.

6.1 Detection Speed

We want the whole process of recognition to happen in as close to real-time as possible. We have two options for face and eye detection, let’s see which is faster. These tests were performed on 122 images, varying from 960x1280 all the way up to 2620x3600 pixels. These are relatively high resolution images to use for detection, detection works about 67% faster on 220x220 images (and usually just as well). So why not scale down the image before performing detection? We keep them at full resolution to see how well each algorithm scales.

OpenCV uses Haar-Cascade classifiers for face and eye detection. We use the pre-trained `haarcascade_frontalface_default` XML for detecting frontal faces, `haarcascade_mcs_lefteye` and `haarcascade_mcs_righteye` for detecting eyes (with no glasses). We choose to detect each eye separately as this is 40% more reliable than using `haarcascade_eye` to simply detect both eyes at once. Instead of performing eye detection on the whole face, we perform it on estimated eye regions (an area in the image where we think the eye exists), this helps reduce false positives (detecting an eye where an eye
doesn’t exist). This measures are necessary as eye position accuracy is essential to our face pre-processing.

In contrast, iOS offers face and eye detection using their Core Image framework, available from iOS 5. As with all things Apple we don’t know anything about the implementation behind the detector (“using image processing” is all they mention). Face detection also returns the eye positions automatically, and we have the option of running it on the CPU or GPU.

From the chart above we can see that Core Image performs face and eye detection faster than OpenCV, however the difference is only a few milliseconds. Despite being faster, Core Image is not the better approach for detection. The accuracy of it’s eye detection is abysmal compared to OpenCV, leading to inaccurate pre-processing (see below). There were also some issues while using Core Image such as memory leaks and unexpected restarts of the device. It seems that there are definitely some bugs that Apple still needs to fix (on iOS 8.3) and overall the tiny ~50ms speed up from using Core Image is not worth.

![Fig. 9. Examples of inaccurate eye position degrading the pre-processed image.](image)

(For each pair: OpenCV on left, Core Image on right.)
Next we compare the time it takes to recognise a face. The results are the average of 18 recognitions on a model trained to recognise 18 individuals. To recognise a face we first need to find the face and eyes in the image so we can apply pre-processing. The pre-processing is part of the overall time it takes to recognise, but it’s speed is not in any way related to the recognition algorithm we use, the difference in the above chart is just normal variance between runs.

In terms of the actual recognition part (after pre-processing is done), both Fisherface and LBPH outperform Eigenface by ~200ms, which is a significant enough difference. Between Fisherface and LBPH however the difference is minuscule so with respects to recognition speed either of them is a good choice.
6.3 Size

The face recognisers can be saved and loaded from disk to avoid retraining the algorithm each time (a timely process). We need to consider the size of the saved data as both storage and memory are limited on a mobile environment. For this benchmark we trained each model to recognise 18 individuals with 7 images per individual (on average) so a total of 122 images. Our pre-processed images were greyscale and 200x200 pixel.

Due to how the Eigenface algorithm operates, it needs to store 122 Eigenfaces (eigenvector) and eigenvalues to differentiate the 122 faces in our training set, hence the large size. In comparison, Fisherface only calculates one Fisherface (eigenvector) and eigenvalue for each additional face, so in our training set it would only store 17, hence the much smaller size.

The size will only get larger as we add more individuals to the algorithm to recognise, for this reason only Fisherface and LBPH are viable options as memory is limited and we need to keep the whole model in there. Note however that the models are saved in XML or YAML format, and the data is encoded as floating point values in scientific notation, for this reason file size is slightly larger than the space occupied in memory.

Regardless, iOS has a very strict memory usage policy and will issue warnings (followed by app termination) even when using 35-60% of the available memory. If we take our Jailbreak use case for example, there would be roughly 170 individuals, which
needs about 1GB of memory for Eigenface but only about 150MB for LBPH. Apple’s top of the line iPhone 6 only has 1GB of RAM so LBPH is easily the best choice regarding size.

### 6.4 Load Time

<table>
<thead>
<tr>
<th>Method</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Eigenface (PCA)</td>
<td>8.5s</td>
</tr>
<tr>
<td>Fisherface (LDA)</td>
<td>1.07s</td>
</tr>
<tr>
<td>LBPH</td>
<td>0.45s</td>
</tr>
</tbody>
</table>

Directly related to the size as discussed above is the loading time. Due to the large size but also the number of lines in the saved file for Eigenface it takes a whopping 8.5 seconds to parse and load the recogniser, this is an unacceptable first time delay. Again Fisherface and LBPH are both the best choice regarding loading time.
So far the best algorithm to use has been LBPH, then LDA followed by PCA last. Before declaring an algorithm as the winner, we need to test the accuracy of each. The first test (Individuals Recognised) checks to see how good the algorithm is at distinguishing the people from each other, i.e. how many of the people it was trained to recognise could it correctly recognise. Again LBPH performs the best, recognising 15 of the 18 correctly, followed by LDA and PCA.
The second test (Individual Accuracy) checks to see how resilient or robust each algorithm is, i.e. how many different images of the same individuals does it correctly recognise, where the face angle in the different images varies. Here LBPH and PCA both performed the best with 7/10 accuracy.

Depending on the application of face recognition, you may not have access to 7 images of each individual for training purposes. The two tests above are also performed when the algorithm is trained with only 2 images to see what the best algorithm is for those type of applications. PCA gets the best individuals recognition but very poor individual accuracy, overall LBPH is the best performing here.

So finally considering all benchmarks LBPH is the best algorithm to us, with LDA as the next best algorithm, while PCA sometimes outperforms LDA [3] the size difference clearly marks LDA as the better option.
6.6 Datasets

Fig. 10. Dataset of images used for testing accuracy with various face angles. Those correctly recognised are marked with a checkmark.
Fig. 11. Dataset of test images for the 18 people the algorithm is trained to recognise. Those correctly recognised are marked with a checkmark.
7. FUTURE WORK

7.1 Face Verification

When the face recogniser algorithm tries to recognise an image of an unknown person (one that it isn’t trained to recognise), it should be unable to find a match. For example if a human looks at an image of someone they’ve never met before, they shouldn’t be able to recognise them. This process is called face verification (or authentication) and it differs from face recognition (i.e. identifying a person). The available algorithms are not capable of doing this automatically. Instead they will find a match for any given image even if they’re of an unknown individual.

This is the single biggest drawback of the project. Simply thresholding the confidence value of each match (returned by the algorithm) will not work, as the confidence value doesn’t lie between 0-100, it’s based on the distance in eigen-subspace and is not a reliable method. For example a correct match could have a confidence of 800 or 2000, we can’t simply determine any confidence above X to be accurate.

The way to solve this is by reconstructing the facial image using the eigenvectors and eigenvalues (which are the underlying data of the algorithms) and comparing it to the input image. If the person has many faces in the training set, the reconstructed face will resemble the input image, otherwise it is an unknown person. I was unable to get this to work.

7.2 Improve Histogram Equalisation

As mentioned in the report face recognition is very sensitive to changes in illumination. We can improve upon the current histogram equalisation in the pre-processing stage by applying it on each side of the face separately and blending the two for a smooth result. A limitation of the current approach is that equalising the whole face does not produce great results if there is shadow on one side of the face only. Equalising each side of the face separately means that both sides of the face will be evenly lit.

7.3 Improve Eye Detection

Face recognition does not work very well when individuals are wearing glasses in an image. This comes down to the eye detector not being able to find the eyes, possibly due to the glare coming for the lens of the glasses. As a result of not detecting the eyes we
cannot use the image for training or recognising since we require the eye positions to apply the face preprocessing.

We may use the `haarcascade_eye_tree_eyeglasses` HaarCascade file to detect eyes with glasses, but it is only 15% reliable versus 80% for the HaarCascades used in this project. This leaves much room for future work but it would be of a rather technical level and could be a research project of itself.
8. CONCLUSION

The primary aim of this project was to investigate face recognition and evaluate the performance and accuracy of any available algorithms on iOS. This goal was achieved. The only free and openly available algorithms that also work on iOS were the three that we have mentioned throughout the project, PCA, LDA and LBPH. As we saw in Chapter 6 through intensive testing LBPH was found to be the best approach to face recognition, offering the highest accuracy, lowest model size and fastest recognition speed. It performed recognition in under 1 second, was very memory efficient, recognised 83% of our trained dataset (15 out of 18 people) and finally was very robust to different face angles and lighting conditions.

The secondary aim of this project was to develop an iOS application that made use of face recognition practically. This goal was also achieved. I first developed a beautiful app for the Jailbreak HQ event, where users could follow the teams progress through checkins and social posts and view their profiles and a lot more (see Chapter 5). On top of that I integrated face recognition into the checkin system for Jailbreak HQ, where participants could simply checkin by looking into their front-facing camera.

The tertiary aim of this project was to create an easy-to-use library for face recognition on iOS. This goal was also achieved. I wrote a library with a familiar and easy to use Objective-C interface that doesn’t require any prior knowledge of C++ or OpenCV from the user. The library performs any necessary face pre-processing and abstracts away much of the work for setting up.

Overall I found face recognition as a whole to be at a relatively good state. I found it’s performance on mobile to be good, and it’s accuracy in a real-world setting to be acceptable (depending on the application). For example the results I got are not good enough for biometrics, but acceptable for tagging people in images similar to what iPhoto does. These findings are not about the state of the technology as a whole, but rather the state of the non-proprietary implementations of the technology available at the time of writing. There are certainly much better implementation out there, such as Facebook’s DeepFace [10], which boasts human like accuracy, but it is years of research and proprietary work that will most likely never be available to developer.