A FACE RECOGNITION PLUG-IN FOR THE PHOTOCUBE BROWSER

Kristján Rúnarsson
Master of Science
Computer Science
December 2011
School of Computer Science
Reykjavík University

M.Sc. PROJECT REPORT

ISSN 1670-8539

A Face Recognition Plug-in for the PhotoCube Browser

by

Kristján Rúnarsson

Project report submitted to the School of Computer Science
at Reykjavík University in partial fulfillment of
the requirements for the degree of
Master of Science in **Computer Science**

December 2011

Project Report Committee:

Björn Þór Jónsson, Supervisor
Associate Professor, Reykjavik University

Hannes Högni Vilhjálmsson
Associate Professor, Reykjavik University

Jón Guðnason
Assistant Professor, Reykjavik University

A Face Recognition Plug-in for the PhotoCube Browser

Kristján Rúnarsson

December 2011

Abstract

PhotoCube is a media browser that enables users to browse digital media multi-dimensionally using an interactive three dimensional user interface similar to the OLAP applications used in business intelligence. Gathering metadata about the content of media is critical to the usability of the browser. This metadata is gathered by the browser using a collection of specialized plug-ins that analyze images for attributes such as colors, Exif extensions, location data, etc. This metadata can then be used as dimensions in multi-dimensional browsing. We describe a prototype of a plug-in that uses face-detecton and face-recognition algorithms to assist the user in classifying human faces that appear in photographic images. We compared effectiveness of the plug-in with that of two popular image browsing applications, Picasa and iPhoto. We found that the face detection rate of Picasa and iPhoto is roughly the same with our plug-in following closely behind. Picasa achieved the best face recognition rate followed by iPhoto while our plug-in came close to achieveing the same recognition rate as iPhoto.

Andlitskennslaviðbót fyrir PhotoCube myndavafrann

Kristján Rúnarsson

Desember 2011

Útdráttur

PhotoCube er myndvafri sem gerir notendum mögulegt að vafra stafrænt efni með fjölvíðu þrívíddar viðmóti svipuðu OLAP forritum sem notuð eru í viðskiptaheiminum. Söfnun lýsigagna um slíkt efni er afar mikilvægur þáttur í virkni vafrans. Þessum lýsigögnum er safnað af sérhæfðum viðbótarforritum sem rýna efnið eftir vissum eigindum eins og litum, Exif gögnum, staðsetningargögnum, o.s.frv. Þessi lýsigögn er svo hægt að nota sem víddir í fjölvíðum vafra. Við lýsum frumgerð af viðbótarforriti sem notar andlitsleitar- og andlitskennlareiknirit til þess að aðstoða notanda við tögun andlita í ljósmyndum. Við bárum saman skilvirkni þessarar viðbótar við skilvirkni tveggja vinsælla myndvafra, Picasa og iPhoto. Við komumst að því að Picasa og iPhoto stóðu sig álíka vel við leit að andlitum en viðbótin okkar kom ekki langt á eftir. Við kennsl á andlitum stóð Picasa sig best, iPhoto varð í öðru sæti en viðbótin stóð sig litlu verr en iPhoto.

Acknowledgements

I would like to thank Björn Þór for his guidance while writing this report and Jón for his help with the LDA class. I would like to thank Grímur Tómasson for his assistance with understanding the ObjectCube model, developing the plug-in API and getting started with the face recognition work. I would also like to thank Hlynur Sigurþórsson for his assistance with developing the preliminary user-interface for analyzing face recognition results in PhotoCube. All these individuals helped to define, frame and conclude the project, and deserve gratitude for all their work.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Image Browsers

We live in a world where the ability to create digital media is at our fingertips. It is, for example, difficult to find a mobile phone these days that does not double as a camera or a camcorder. This has resulted in most households having large collections of digital media, especially images and videos. Unfortunately, however, media collections have a tendency to rapidly become so large that their owners lose oversight and searching becomes a significant problem. Most users nevertheless try to organize their media collections as best they can. This often results in media being sorted into folder hierarchies where individual folders are named by events, subject matter or dates.

A more sophisticated solution is to use a media browser. There are a number of commercially available media browsers. One example of a modern media browser is Picasa from Google which allows the user to organize videos as well as images. Another modern and popular image browser is iPhoto from Apple which restricts itself to enabling the user to organize digital images. Neither Picasa nor iPhoto deviate much from the folder based approach to organizing a media collection. Picasa restricts itself to indexing existing image folders while iPhoto imports image collections but organizes them into folder-like structures of organizational units. Both of them gather certain types of metadata automatically such as Exif or color data. Other than this, the burden of organizing and categorizing image collections is largely placed on the user.

Both iPhoto and Picasa, however, offer the user the ability to automatically detect and recognize faces within images. The user must tag faces manually with the names of the individuals that appear in the images. While tagging, the user trains the browser's internal

recognition subsystem. As this training progresses, the user's workload becomes increasingly smaller and limited to simply confirming the automatically established identities suggested by the browser as well as occasionally correcting identifications or adding ones that the browser missed.

1.2 The PhotoCube Image Browser

Recently a team from Reykjavík University has been developing a research prototype of a new kind of data model for media browsing (Tómasson, 2011; Sigurþórsson, 2011). This data model has been implemented as the ObjectCube database engine which applies the basic concepts of OLAP (Online Analytical Processing) to media management and searching. As part of this ongoing project, an image browser named PhotoCube has also been developed as a front end to the ObjectCube database engine.

In ObjectCube, units of metadata are known as tags. Tags relating to some concept are called tag-sets. A simple example of this is a set of tags where each tag is the name of an individual that appears in a photograph or their role (child, parent, etc.); these tags can all be said to belong to the tag-set *people*. The tags in a set can be organized into any number of user defined hierarchies. These hierarchies can then be used as dimensions. Once the user has specified three browsing dimensions, PhotoCube will present the user with a cube shaped structure where each dimension is a hierarchy or a tag-set. The user can now perform a number of common OLAP operations on the results (readers unfamiliar with OLAP are referred to Coi (2011) for an introduction). For browsing purposes the most important operations are *drill down* and the opposite *roll up*. Drilling down amounts to going from a generic level in a hierarchy into a more specific one with correspondingly more detailed results. The act of rolling up is simply the reverse of this process. The PhotoCube browser exposes this functionality to the user through an intuitive graphical user interface.

To facilitate automatic harvesting of metadata, the ObjectCube database engine offers a plug-in architecture specifically designed for the easy integration of metadata analysis methods, where each method specializes in gathering a particular kind of metadata. Currently only two plug-ins are available, a plug-in that gathers Exif data and a color analysis plug-in. To bring PhotoCube closer to feature parity with current commercially available image browsers, a face recognition plug-in must be added to the ObjectCube database engine. This is the topic of this project.

1.3 Contributions of the Project

Since the goal of the ObjectCube project is an open source browser, the plug-in should be implemented using off-the-shelf FOSS (Free Open Source Software) libraries and pre-trained classifiers wherever possible. The FOSS community, has over the last couple of decades, produced a large flora of image processing and linear algebra toolkits. Chief among the image processing toolkits is OpenCV. Examples of open source linear algebra libraries include the GNU Scientific Library, Armadillo and Eigen.

This project makes four contributions:

1. We implemented a plug-in for the ObjectCube database engine that implements face detection and face recognition services using holistic face recognition.

2. We implemented two face recognizers that use the Eigenfaces and Fisherfaces face recognition algorithms, making as much use of off the shelf FOSS libraries as possible. Both of our recognizers return the n best matches found by the face recognizer. This is done to enable PhotoCube users to perform a quick correction in case the recognizer's first choice of identity is false.

3. We validated our recognizer's performance by repeating an experiment conducted by a third party, Marcialis and Roli (2002), using the well known AT&T and Yale face databases under 'laboratory' conditions. These experiments showed that the recognition rate of our Eigenfaces recognizer is comparable to that achieved by Marcialis and Roli, while our Fisherfaces recognizer performed better than the one tested by Marcialis and Roli. The results also indicated that the separability of our recognition results, the ability to divide distance values into probably correct and probably incorrect values, is acceptable when using these highly normalized scientific face databases.

4. We developed an experimental protocol for a browsing scenario using a typical real-world image collection. We then used this protocol to compare our plug-in to the Picasa and iPhoto commercial browsers. These experiments showed that the face detection performance of our plug-in is surprisingly close to that of Picasa and iPhoto. Face recognition performance is just shy of the lower end of what these two browsers achieved. Picasa had the highest recognition rate with iPhoto coming second. Our plug-in came off worst but was able to come quite close to the recognition rate achieved by iPhoto when it was using our Fisherfaces recognizer. Our suggestive feature performed better than we expected and made quick correction of falsely identified faces in a significant number of cases.

1.4 Overview of the Report

This report is organized as follows. In Chapter 2 we discuss PhotoCube and ObjectCube in greater detail. Chapter 3 contains an introduction to the problem of face recognition in near real-time software applications. In Chapter 4 we describe the implementation of our plug-in. In Chapter 5 we describe an experiment conducted to verify the functionality of our face recognizers. In Chapter 6 we describe the experimental protocol we developed to evaluate the face recognition performance of our plug-in and the results of those experiments. In Chapter 7 we outline future work and in Chapter 8 we present our conclusions.

Chapter 2

Photocube

In this chapter we give an overview of the ObjectCube data model, its implementation, and the PhotoCube image browser. We focus on the aspects that are relevant for this project. For more details of PhotoCube and ObjectCube respectively, the reader is referred to (Sigurþórsson, 2011) and (Tómasson, 2011). The chapter also assumes that the reader has basic knowledge of OLAP; readers that do not are referred to (Coi, 2011).

2.1 The ObjectCube Data Model

The basic building block of the ObjectCube database engine is metadata which in ObjectCube is known as a *tag*. These tags can be attributes of the file itself, or relating to the content, such as who or what is in the image, what are their roles within an organization or a family group, where was the image taken, etc.

There is no limit on how many tags are associated with an image or vice versa. Related items of metadata are organized into *tag-sets*. One example of a tag-set could be the *people* tag-set which could, for example, contain the tags: my family, child, mother, father, John, Jane, Janet, Josef, extended family, cousin, Jack. Another example might be the tag-set *Time* which could contain months, weeks and days of the week.

Tag-sets can be structured by the user using tag-hierarchies. Hierarchies in ObjectCube are tree shaped and are either level-based structures or value-based. In a level-based hierarchy all nodes in a level of the tree belong to the same category. In a value-based hierarchy nodes in a tree level can belong to different categories. Multiple hierarchies can be defined to structure the same tag-set and hierarchies can be created that do not contain all tags of the tag-set.

ObjectCube applies OLAP principles to media management. The user can specify up to three dimensions for browsing. When choosing dimensions, the user picks a tag-set or a tag hierarchy for each dimension. If the user only specifies one dimension the result will be a line, if two dimensions are specified the result will be a rectangle and if three dimensions are specified the result is a cube. For an introduction to multidimensional analysis as well as equivalences between ObjectCube and OLAP the reader is referred to Chapters 2 and 3 of (Tómasson, 2011). For detailed information on the inner workings of the ObjectCube database engine the user is referred to Chapters 4 and 5 of the same thesis.

2.2 The ObjectCube Database Layer

The ObjectCube database engine is written in C++ to maximize efficiency and cross platform portability. It does not rely upon any proprietary libraries. Figure 2.1 shows a schematic drawing of its architecture. An overview of the most important design principles of Object cube is given in Chapter 4 of (Tómasson, 2011)

As can be seen in Figure 2.1, the core of the ObjectCube database engine exposes four main interfaces; The C++ API; the Python Wrapper API; the Data Storage API; and the plug-in API. The main programming interface is the C++ API which is also made available as a Python wrapper for RAD support.

Next is the Data Storage API which abstracts tha data storage requirements. The final interface is the plug-in API, which written in C++ only and whose sole purpose is to enable third party developers to develop metadata harvesters. The plug-in API is described further in Section 2.3.

2.3 The ObjectCube Plug-in Framework

ObjectCube is designed to allow users to organize, browse and search media collections based on metadata. The metadata can describe, attributes of the image file itself, such as Exif data, GPS location data etc., or the subject matter of the image. For this to work well, PhotoCube requires copious amounts of metadata, far more than a user can be expected to input manually and it is critical that as much of this metadata as possible be harvested automatically.

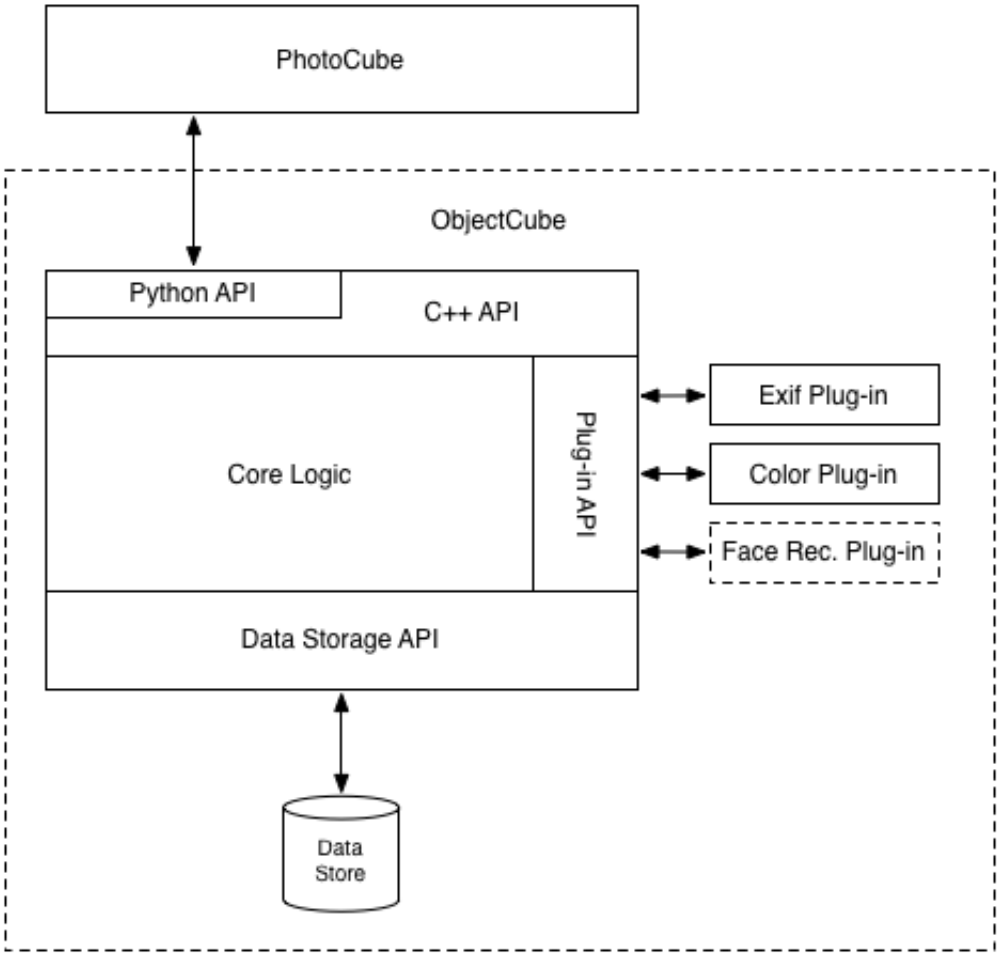


Figure 2.1: The whole ObjectCube engine is written in C++. The Python API is a wrapper for the C++ API.

The plug-in subsystem in ObjectCube can be thought of as a chain of processing units. When an object is added to ObjectCube it is passed through each of the registered plug-ins in the chain. These then perform their specialist analysis on the object.

Any ObjectCube plug-in must be written in C++ and it must conform to a standard interface by implementing a virtual C++ template class. A plug-in can have a data storage layer of its own. Such a data store is isolated from ObjectCube which cannot access this data other than via the plug-in interface. The plug-in itself is similarly isolated from ObjectCube.

Metadata harvesting in ObjectCube is thus done at import time rather than by a background process after importing. Thus some practical considerations limit the number of plug-ins that can be added. Exif data extraction is, for example, relatively speedy but face recognition is time-consuming and generic object detection can also be expected to be timeconsuming. This could be alleviated to some degree with parallelization and ag-

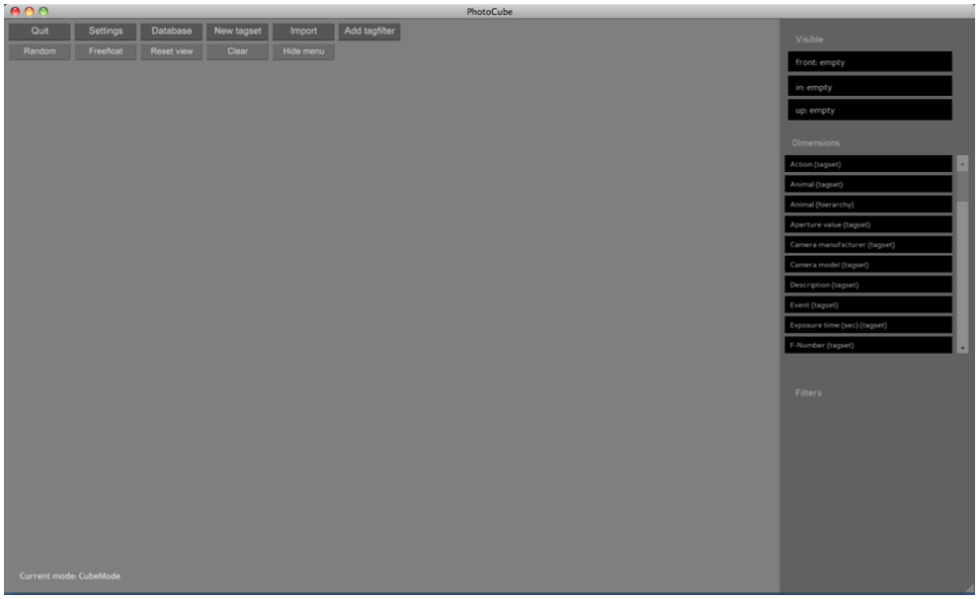


Figure 2.2: In the top left corner are various utility and settings options. At left are the visible, dimensions and filters menus. Image cubes and card views appear in the display are which is blank in this image.

gressive optimization efforts. Even so, metadata about media content is easily the most valuable of all the data harvested by ObjectCube. Thus the computational intensity of object detection, which grows rapidly with image size, represents the most severe limitation on ObjectCube's usefulness given our current model where we perform all analysis at import time. There are, of course, ways to alleviate this by using off-line processing where the images are simply scanned at import time and then processed sequentially in the background. One could even introduce a scheduling mechanism by which the photos the user shows the most interest in are moved to the top of the queue. This will be further discussed in Chapter 7.

2.4 The PhotoCube User Interface

The PhotoCube user interface is simple and functional by design. On start-up the browser presents the user with a single window that is divided into three areas; the utility menu pane at the top; the input pane on the left hand side; and the main display area (see Figure 2.2). The browser has two display modes, cube-mode and card-mode (see Figure 2.3) which it can alternate between seamlessly; when the interface is started the browser defaults to cube mode. The underlying PhotoCube APIs are designed for high extensibility making the addition of new display modes easy. PhotoCube is implemented completely in Python.

Figure 2.3: Photocube's two display modes. On the left is cube mode, on the right card mode.

When browsing an image collection the user must assign a tag-set or a tag hierarchy to at least one of the three dimensions. The results are displayed on screen as an intuitive three dimensional representation of the cube returned by ObjectCube. To give an example the user can look for and display all images that contain at least one grandparent (dimension one), at least one grandchild (dimension two) and were take at a some location (dimension tree). Figure 2.3 shows such a browsing scenario.

The multi-dimensional results that the ObjectCube database engine returns are represented in cube view (which, as the name indicates, is literally a cube of images) rendered using the Panda 3D graphics engine. The user can now, by means of pointing and clicking, perform a number of multi dimensional navigation and data manipulation operations including drill down, roll up, slicing, dicing and pivoting.

These operations are implemented by applying different filters to the selected tag-sets and hierarchies; the user can also apply various filters to the result set to make it more manageable. All of this can be done through a point-and-click interface and the effects of such operations are visible in the three dimensional interface. To quickly browse the content of some cell in the cube the user can switch to card mode and scroll through the images in that cell in a manner similar to the side scrolling document preview in the OS X finder, or the navigation feature in Windows 7 that allows one to scroll through open application windows.

2.5 Summary

The ObjectCube data model is based on the OLAP model. Instead of working with numerical data, however, ObjectCube is meant to be used to organize and browse large media collections based on metadata relating to the attributes of media items and their content. This model has been implemented as the ObjectCube database engine. This implementation written in C++, it is highly efficient, cross-platform and it implements a range of common OLAP operations. PhotoCube is a user interface that provides a user friendly presentation layer for the ObjectCube database engine using the Panda 3D graphics engine. It is written in Python to facilitate rapid development and it is designed to be highly extensible.

Chapter 3

Face Recognition

Traditionally face recognition algorithms have been divided into two types (Zhao, Chellappa, Phillips, & Rosenfeld, 2003); holistic algorithms and feature-based algorithms. Holistic algorithms use the entire data of a facial image for comparison purposes while feature-based algorithms rely on the geometric relationships between facial features for identification. Finally, hybrids of the two methods have also been proposed. In this chapter we begin by describing the overall face recognition process. We then proceed to discuss the individual phases of the face recognition process in more detail and the algorithms that are best suited to the processing done in each phase.

3.1 The Face Recognition Process

Both feature-based and holistic face detection methods divide the face recognition process into three distinct phases. These are:

Face detection: This can be done by marking the face region manually in the original image. Face detection can also be done computationally using an object detection algorithm that performs a face search and returns a set of regions of interest within the original image that mark the faces it found.

Feature location: This phase is mandatory for feature-based algorithms since they rely completely upon facial features to identify a face. Locating facial features in not a prerequisite for holistic algorithms but, facial features can nevertheless be useful for advanced automatic normalization of facial images which can significantly improve the recognition performance of a holistic face recognition algorithm (Martinez, 2002). Advanced normalization can include scaling, rotating and warping of the

face to ensure that facial features such as the eyes and the mouth are always located in the same pixel position within a face recognition image.

Recognition: An unknown face is put into a face recognizer which then compares it to a set of known training faces whose identity has been confirmed by a human. An identification corresponding to the closest matching training face is returned. Similarity is judged using some form of distance metric. One can define a 'cutoff' distance beyond which the likelihood of a correct identification is negligible which allows one to write off any identification with a greater distance than this cutoff as likely to be be incorrect.

In the remainder of this chapter we discuss each of these phases in more detail.

3.2 Phase 1: Face Detection

Since the 1990s several different approaches have been developed to solve the problem of face detection. An overview of early attempts was provided by Hjelmås and Low (2001) and Kriegman and Kriegman (2002). The former divided face detection algorithms into image-based and feature-based approaches. Image-based algorithms were further subdivided into linear subspace models, neural nets and statistical approaches. Feature-based algorithms were subdivided into the low-level analysis, feature analysis and active shape model approaches. These papers effectively summarize the state of face detection up until the years 2000-2001 with Hjelmås and Low concluding that feature-based methods had by then reached a sufficient level of maturity to make them suitable for real time applications. They also emphasized the importance of face detection for preprocessing/normalization in face recognition.

3.2.1 The Viola-Jones Object Detector Framework

The last decade has seen continued advances in face detection. The most important work was probably the work of Viola and Jones (2001) who proposed an object detection framework supporting Haar-like features, a method proposed earlier by Papageorgiou, Oren, and Poggio (1998). The Viola-Jones object detector (also called a Haar-cascade) uses a summed area table, AdaBoost learning and an attentional cascade structure to achieve high real-time performance. Viola-Jones detectors can be trained to recognize any kind of rigid object, including human faces and facial features such as eyes, mouths, etc. A survey of recent advances in face detection (Zhang & Zhang, 2010), hailed Viola-Jones as

the face detection algorithm that has had the most impact since 2000. Indeed much of the current research into face detection seems to concentrate on variations of the Viola-Jones detector which has become quite popular.

Viola-Jones has the advantage of speed and it has a quite high detection rate. The main disadvantage of Viola-Jones is that it requires a training set that can easily count 8-10.000 or more positive and negative samples. Creating this training set is time-consuming as is the training process (Seo, 2011). The Viola-Jones algorithm has two further disadvantages. Firstly, detection time increases in proportion to the size of the image. Secondly, the basic Viola-Jones algorithm is sensitive to the rotation of a face in an image. If the subject of a photograph tilts the head too far forward, backward or to the sides, Viola-Jones may fail to find it. The original Viola-Jones framework has been improved in recent years which has, among other things, provided better performance when detecting rotated faces (Lienhart & Maydt, 2002). By using generic rotated Haar-like features complete rotation invariance can theoretically be achieved (Messom & Barczak, 2006) but this idea has various practical problems that prevent it from being used (*Haar-like features*, 2011).

3.2.2 Choice of Face Detection Method

Turning to the choice of a face detection algorithm for our ObjectCube plug-in there have not been many comparative studies since Hjelmås and Low (2001) and Kriegman and Kriegman (2002). A recent effort was made by a team from the State University of Tula Degtyarev and Seredin (2010) which compared a number of freely and commercially available face detection toolkits. The toolkits tested included ones from VeriLook and Luxland, both of which use secret proprietary algorithms. The remaining toolkits tested were: a face detection toolkit developed by the IDIAP research institute, UniS developed by the university of Surrey, FDLib developed by a team from the Max-Planck-Institute, the SIF toolkit developed at the university of Tula, and finally OpenCV which is an open source computer vision library developed by the Intel Corporation and which implements Viola-Jones. The results of this comparative survey showed OpenCV to be the top performing open source face detection toolkit, outperformed only by the proprietary VeriLook toolkit. Given the FOSS restriction on software choice we concluded that OpenCV's Haar-cascade object detector is a perfectly acceptable choice for our ObjectCube plug-in.

3.3 Phase 2: Feature Location and Face Normalization

Many studies dealing with face recognition were performed on face databases intended for scientific research into face recognition and bypass the issue of face normalization since these face databases are pre-normalized. Examples of such databases are the FERET face database developed by the US Department of Defense (DoD) Counter-drug Technology Development Program, the Cambridge University Computer Laboratory (aka. AT&T) face database and the well known Yale face databases. These databases contain collections of faces that exhibit some translation and rotation variation (collectively know as 'pose variation') and variations in facial expression. Some of them focus on lighting, others on variations in occlusion or facial expression and, sometimes the images include a degree of in-plane rotation. Nevertheless, these scientific face databases are practically speaking collections of facial images whose format resembles that of passport-photos.

In a real world application, where faces must be extracted from larger images, variations, particularly in expression and rotation, are often greater than in scientific face databases. In such applications, compensating for pose variations can have a very high impact on the eventual recognition rate. This is important since in many real-world applications, like image browsers, it is desirable to achieve a high rate of recognition using relatively few training images. Traditional methods to alleviate problems caused by pose variation include pose correction and shape normalization.

A simple recognition image normalizer employs object detection to locate facial features, commonly the eyes and the mouth. The location of these facial features is then used to rotate the image to ensure that the line between the eyes is parallel with the horizontal edge of the image. The face can now be warped using simple scaling and translation algorithms such that the mouth and eyes are always a fixed distance apart and located in the same pixel locations within the image before it is cropped to a fixed size and used in the recognition process. Research done as early as 1997 (Fischer & Duc, 1997) found that correcting translation variance alone almost quadrupled the number of correctly recognized faces when using holistic recognition.

Another approach to the problem of pose variation, that is also effective in dealing with expression variation, is to 'harden' the face recognition algorithms themselves against pose variation. This has been show to yield significant improvements (Wang, Yan, Huang, Liu, & Tang, 2008; Martinez, 2002). Algorithm hardening improves performance with image sets that have a lot of expression variation, pose variation and occlusion.

3.4 Phase 3: Face Recognition

Modern image-based face recognition algorithms can be broadly divided into two categories, feature-based algorithms and holistic algorithms. Feature-based algorithms make use of the fact that all human faces share a (broadly similar) topological structure. These algorithms represent the human face as a graph where each node represents a facial feature like an eye, the nose, corners of the mouth. Holistic methods, on the other hand, compare one entire face to the another using statistical methods and linear algebra.

3.4.1 Feature-Based Methods

The most commonly used of the feature-based algorithms is EBGM (Elastic Bunch Graph Matching). This algorithm was first proposed by a team from the Universities of Bochum in Germany and Southern California in the United States (Wiskott, Fellous, Krüger, & Malsburg, 1997). Elastic Bunch Graph Matching was used in the US DoD sponsored Face Recognition Technology (FERET) program. This algorithm requires that the user either manually place the nodes of the face graph to mark facial features or that these be found automatically using object detection. Graph based methods have the advantage of being insensitive to the location of a face within an image as well as its rotation and lighting condition and EBGM is also insensitive to foreign objects in the facial area such as dirt, caps, hats, facial decorations and such. The disadvantages of EBGM is that mapping of facial features is highly reliant upon either object detection or manual input by the user and it is computationally expensive.

3.4.2 Holistic Face Recognition

There are a number of holistic face recognition methods but the most popular one is face recognition by Principal Component Analysis (PCA). Also known as Eigenfaces, this algorithm was first proposed by a team from the Massachusetts Institute of Technology Media Lab (Turk & Pentland, 1991). The idea is to extract meaningful information from a facial image by calculating an average image from a set of training faces and then subtracting this average from each individual training face. The result is an image that only consists of components that differ from the average. The variance in this data is then maximized and the dimensionality is reduced using principal component analysis. This process is known as projecting the face image into ‘face-space’. To recognize a face, the face-space projection is repeated for an unknown facial image and the distance between

the resulting projection and each of the projected training faces is calculated using any one of a variety of methods, a simple one being euclidean distance. The advantages of the Eigenfaces algorithm is that it is simple, not very computationally expensive, robust and it is completely unsupervised. Disadvantages of Eigenfaces are that it is sensitive to the presence of foreign objects in the facial area (occlusion), variations in backgrounds, lighting conditions and variations in pose and facial expression.

3.4.3 Improved Holistic Methods

One of the more significant improvements in PCA based face recognition is the so called *Fisherfaces* method. With Fisherfaces, Linear Discriminant Analysis (LDA) is used to complement PCA (Belhumeur, Hespanha, & Kriegman, 1997). While PCA maximizes the scatter of the training data and reduces its dimensionality, LDA maximizes the scatter between individual classes in the set of training data yet at the same time it also minimizes the scatter within each class. Using LDA reduces sensitivity to lighting variations, variations in expression and it increases tolerance for certain types of occlusions such as eyeglasses (Zhao, Chellappa, & Nandhakumar, 1998; Belhumeur et al., 1997). One small disadvantage is that in order to calculate within-class scatter Fisherfaces requires that at least two training images per person be available. This means that when only one training image is available per person performance will not differ much from Eigenfaces. Furthermore Martinez and Kak (2001) showed that for small training sets and for training sets that are not representative, LDA based algorithms can perform worse than Eigenfaces. They also showed that Eigenfaces is less sensitive to variations in the training data than LDA based algorithms.

3.4.4 Choice of Face Recognition Method

When it comes to face recognition, the selection of available open source software libraries is a bit more diverse than for face re detection. The OpenCV 1, framework has for some time now provided the components needed to implement Eigenfaces and with the OpenCV 2.X framework which has been rewrittein in C++ this has become even simpler to use. An effort to produce an open source face recognition library is libface. The libface toolkit builds on OpenCV but this effort is still in its early stages and provides little functionality beyond what can be easily implemented with OpenCV, i.e., it does not extend PCA with LDA. Finally there is the Colorado State University ‘CSU Face Identification Evaluation System’. The team at CSU have implemented four face recog-

nition algorithms: Eigenfaces, Eigenfaces with LDA, Elastic Bunch Graph Mapping and a Bayesian intrapersonal/extrapersonal image difference classifier. As the name indicates, this software is really an effort to compare algorithm performance but the source code is available for use without restrictions and the algorithms it provides could thus conceivably be integrated into our plugin.

The choice thus really boiled down to choosing between EBGM and the holistic methods. The EBGM algorithm provided by the CSU benchmark framework expects the landmark facial features to be already mapped out which means we would have to either expect our users to do so manually or try to automate this process using object detection. This requires finding a large number of facial features with object detection algorithms. Holistic algorithms also require some feature location for the normalization phase but the number of features that need to be found is smaller. Furthermore EBGM is more resource intensive and computationally expensive than Eigenfaces and its variants. The choice thus fell on the holistic approach.

We elected to start by implementing our own recognizer using pure Eigenfaces but keeping the option open to improve upon this with our own Fisherfaces implementation. This had the advantage of enabling us to use the proven principal component analyzer that ships with OpenCV. Eigenfaces is a quite adequate choice for our purposes. It is sensitive to background variations, variations in lighting and facial expression. Background variation can be overcome during the face normalization phase by applying an oval mask to all recognition images that leaves only the face itself visible. Lighting variance is certainly a factor but given that modern cameras do a quite acceptable job of ensuring proper lighting in photographic images we felt this would probably not be our most severe problem. Variance in facial expressions, however, is certainly a concern. It is by no means certain that Fisherfaces and LDA based algorithms in general will always perform better than the Eigenfaces algorithm as we already discussed in Section 3.4.3. Nevertheless Fisherfaces should, on the whole, outperform Eigenfaces. Thus implementing Fisherfaces in addition to Eigenfaces gives us a very useful additional capability.

Chapter 4

Implementation

In this chapter we describe the implementation of our plug-in. The logic of the plug-in is divided into three modules. These are: a data store component, a face-detector and a face-recognizer. In Section 4.1 we describe the overall implementation of our plug-in's interface, focusing in particular on the data store modifications. In Section 4.2 we then describe the implementation of the two face recognizers we implemented.

4.1 Implementing the ObjectCube Plug-in API

The cornerstone of every ObjectCube plug-in is a virtual C++ template class that defines a simple interface which the ObjectCube database engine uses to communicate with the plug-in. This interface provides several basic actions which are summarized in Table 4.1. These include processing an image [1] for tags/metadata, processing user submitted boxes for tags, confirming a tag, deleting a tag, merging tags and renaming tags. In Subsection 4.1.1 we describe the plug-in's data storage layer. In Subsection 4.1.2 we provide a list support classes common to many of the plug-in's actions. From Subsection 4.1.3 onward, for each action, we provide an overview of the data store modifications it makes and a short description. Note that the generic ObjectCube concept of a *tag* is a *person* as far as our plug-in is concerned and the ObjectCube concept of a *bounding box* equates to a *face*. In the following text these two terms are used interchangeably.

[1] The interface can handle any media object, in this description, however, we focus on images only.

Action Name	Description
Process	Performed when an object/image is added to ObjectCube. The object is sent to the plug-in which processes it and sends the analysis results, a set of tags, back to ObjectCube.
Process Bounding Boxes	In cases when automatic object detection fails ObjectCube can submit user drawn bounding boxes for analysis. The action is basically the same as *Process* except automatic face searching is not done.
Confirm Tagging	A bounding box is associated with a tag.
Delete Tagging	Deletes a given bounding box. If the associated tag has no more bounding boxes associated with it that tag is deleted.
Rename a Tag	Enables ObjectCube to notify the plug-in that a tag has been renamed. The plug-in will make the necessary adjustments to its internal data store.
Merge Tags	This enables ObjectCube to notify the plug-in that one tag is the same as another. The plug-in will then merge the tags in its private data store.

Table 4.1: Overview of the plug-in API.

4.1.1 Data Storage

Processed face recognition images are written to disk. The data associated with recognition images such as: image path, the bounding box framing a face within the original image, the name of the original image and the identity of the person the face belongs to are stored using a SQLite embedded database. Data from the SQLite database tables is de-serialized and re-serialized via C++ objects using object-relational mapping. An overview of the data storage architecture can be seen in Figure 4.1.

4.1.2 Support Classes

Some of the API methods accept as their argument an object of type *PluginObject* and return a *PluginReply* which in turn contains a set of *PluginTagging* objects. All of these classes are simple data storage objects.

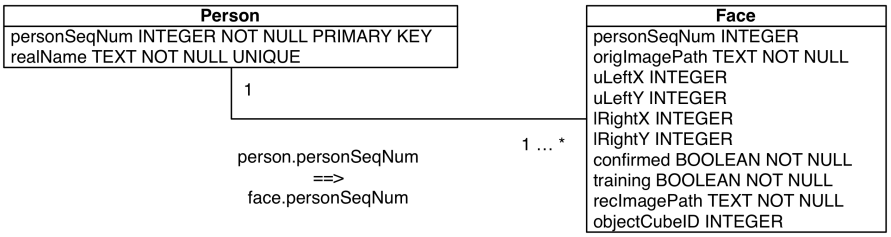


Figure 4.1: The internal database of the plug-in is very simple. It consists of two tables, person and face that are related by a foreign-key.

The PluginObject Class

Definition:

```
class PluginObject {
    protected:
        int id_;
        string name_;       // The name of an original image.
        char* binaryData_;  // An un-decoded image.
        long dataSize_;     // Size in bytes
};
```

Description: This class encapsulates the data needed by the plug-in to do its analysis. It contains a globally unique ObjectCube ID and a raw un-decoded image.

The PluginReply Class

Definition:

```
class PluginReply {
    private:
        int objectId_;
        vector<PluginTagging> taggings_;
};
```

Description: This class encapsulates a set of analysis results found by the plug-in within a single image. It contains the ObjectCube ID of the image that this reply pertains to and a list of PluginTagging objects that contain information about objects of interest that have been found.

The PluginTagging Class

Definition:

```
class PluginTagging {
    protected:
        string tagSetName_;
        vector<string> suggestedTags_;
        BoundingBox boundingBox_;
        bool confirmed_;
};
```

Description: The PluginTagging object encapsulates various data concerning items of interest found in an image by the plug-in. Specifically it contains a tag-set name, a vector

of suggested tags where the first element (suggestedTags_[0]) is the first choice, the associated bounding box and a flag that indicates the confirmation status of this bounding box.

4.1.3 Process

Definition:

```
virtual PluginReply process( const PluginObject& object );
```

Description: All faces in the incoming image are detected. They are then shape normalized and run through the face recognizer. The resulting recognition images and the n best provisional identities for each face are found. The best identity is stored in the internal data store of the plug-in as is the location of each face in the original object. The location data and set of provisional identities for each face are returned to ObjectCube.

Data Store Modifications: This method creates a detected face in the data store and links that face to what ever person the face recognizer determined it belongs to. If that person does not exist it is created. The face is not promoted to training status and its confirmation status is set to *false*.

4.1.4 Process Bounding Boxes

Definition:

```
virtual PluginReply process(
                const PluginObject& object,
                const vector<BoundingBox>& boundingBoxes );
```

Description: This action is usually triggered when face detection fails. The procedure is mostly the same as detailed above except that no face search is performed.

Data Store Modifications: Same as detailed above.

4.1.5 Confirm Tagging

Definition:

```
virtual void confirmTagging(
                const PluginObject &object,
                const string& tagSetName,
                const BoundingBox&
                boundingBox, const string& tag );
```

Description: A bounding box is associated with a tag. This action is triggered by the user and is regarded as confirmation of the validity of the incoming tag, i.e. confirmation of a person's identity. There are four cases:

The person does not exists. A new person corresponding to *tag* is created in the data store.

The person exists in the data store. The data for the person corresponding to *tag* is retrieved from the data store.

The face does not exist. A new face corresponding to the bounding box is created in the data store, it is normalized and associated with the person corresponding to *tag*. The recognition image generated during the normalization is saved.

The face already exists. The face corresponding to the bounding box is disassociated with its current owner and re-associated with the newly created person. If the old owner has no more associated faces, that person is deleted from the data store.

On confirmation, a face is promoted to a training face by marking it as such in the plug-in's internal data store and it is also marked as a confirmed face. The face recognizer is subsequently retrained.

Data Store Modifications: This action can modify and delete entries in both the *person* and *face* tables as described below. Note that this is the only action that will set the *confirmed* and *training* fields in the *face* table to *true*.

4.1.6 Delete Tagging

Definition:

```
virtual void deleteTagging(
                int objectId,
                const string& tagSetName,
                const BoundingBox& boundingBox,
                const string& tag );
```

Description: Deletes a given bounding box as well as the corresponding recognition image. The face is looked up and removed from the internal data store of the plug-in. If the person with which this bounding box was previously associated has no more faces, that person is also purged from the data store.

Data Store Modifications: The entry corresponding to *boundingBox* is removed from the *face* table, along with the corresponding entry in the *person* table if appropriate.

4.1.7 Rename a Tag

Definition:

```
virtual void renameTag(
                const string& tagSetName,
                const string& oldTagName,
                const string& newTagName );
```

Description: Retrieves the entry for the person with the given old tag and replaces it with the new tag. The plug-in will make the necessary adjustments to its internal data store. Throws an exception if a person with the new tag already exists in the database. An example of this is when a user wants to re-tag a face in an image because the automatic identification made by the plug-in was wrong.

Data Store Modifications: Changes the *realName* field of an entry in the *person* table from *oldTagName* to *newTagName*.

4.1.8 Merge Tags

Definition:

```
virtual void mergeTags(
            const string& tagSetName,
            const string& oldTagName,
            const string& newTagName );
```

Description: This enables ObjectCube to notify the plug-in that one tag is the same as another. The plug-in will then merge the tags in its private data store. An example might be that due to a radical change in appearance the user did not previously notice that what he thought were separate individuals are actually one and the same person.

Data Store Modifications: For every face associated with *oldTagName* the *personSeqNum* field is updated to that associated with *newTagName* and the entry *oldTagName* in the *person* table is deleted.

4.2 Algorithms

4.2.1 Face Detection and Normalization

Due to time constraints we only searched for frontal faces and feature detection was restricted to searching for eyes. The face detector uses the standard OpenCV *CascadeClassifier* class configured to function as a Haar-Cascade Classifier (i.e. Viola-Jones, see Section 3.2). Given an original image the face detector searches it for faces. It then searches the faces that it found for facial features.

The results of face detection and feature detection are returned as a tree shaped hierarchy of ROIs (Regions Of Interest). We found that using two separate detectors where each detector specialized in searching for left or right hand eyes yielded significantly better results than using a single detector trained to search for both left and right eyes. Thus we decided to use two eye detectors. The object detector training files used were:

- *haarcascade_frontalface_alt.xml*
- *haarcascade_righteye_2splits.xml*
- *haarcascade_lefteye_2splits.xml*

The face-detector also contains static functions for face normalization which is an important preparatory step for the recognition phase. The normalization functions accept

Figure 4.2: Example of an image from our test set. At left the original, top right the region of interest identified by the face searching algorithm and below right the same face after normalization. This works well for in-plane rotated faces but out of plane rotated faces are harder to salvage.

a copy of an already analyzed original image and a ROI (Region Of Interest) structure. Normalization functions return a list of face recognition images that have been processed as described below. Figure 4.2 gives an overview of the of normalization process. The object detector returns an axis aligned region of interest (bounding box) that frames the face. The face inside this bounding box is then automatically normalized.

Faces for which both eyes are found are centered to ensure that the point between the eyes is always located in the same pixel position within the image. The face is then rotated around this point to ensure that the eyes are as perfectly horizontal as possible. Each face recognition image is cropped to a standard matrix of 120 rows and 100 columns, converted to gray-scale and histogram normalized. The final part of the normalization process is the application of a gray edge mask that leaves only the face exposed. This is done to prevent variations in the background from interfering with the recognition process. Faces for which only one eye or no eyes are found cannot be adjusted so they are simply subjected to cropping, histogram normalization and masking. The processed recognition images are returned as an array of OpenCV matrices.

4.2.2 Face Recognition

We wrote two recognizers each of which implements two different holistic face recognition algorithms. The first one implements the Eigenfaces algorithm as described by Turk and Pentland (1991). The second recognizer implements the Fisherfaces algorithm as described by Belhumeur et al. (1997). In the following text we give a short description of the architecture of these recognizers.

Figure 4.3 shows how both the *Eigenfaces* and *Fisherfaces* recognizer classes extend a (virtual) base class, *Recognizer*. This class provides a simple interface that all recog-

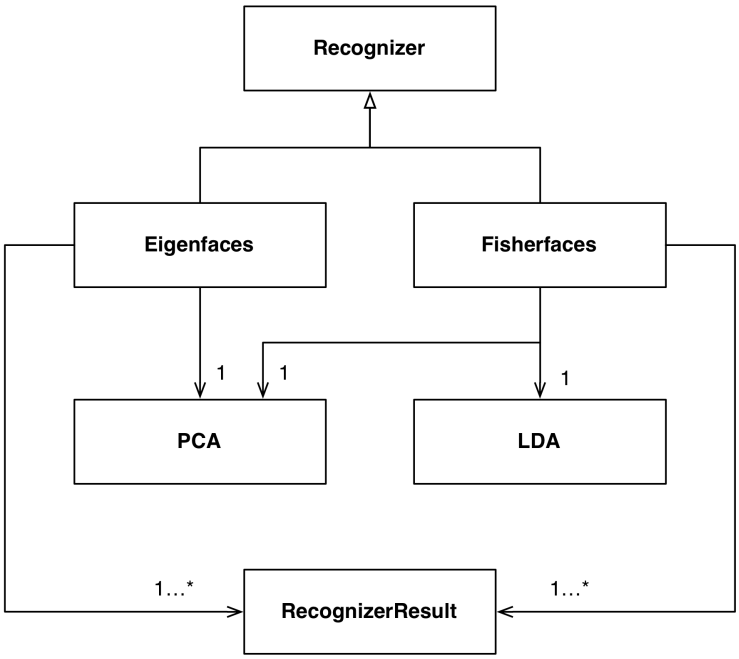


Figure 4.3: A high level overview of the recognizer class architecture.

nizers must implement. It also contains the functions used to evaluate which of a recognizer's training is the best match for an unknown face. At present we have only implemented a simple sequential scan nearest neighbors algorithm with Euclidean distance, but future work will add more evaluation algorithms. The *Recognizer* class is defined as follows:

```
class Recognizer {
    protected:
        ImageList imageList;
    public:
        virtual vector<RecognizerResult> recognize(const Mat& unknown)=0;
            vector<RecognizerResult> nearestNeighbor(
            const vector<Mat> &projTrFaces,
            const Mat &unkProjected,
            unsigned int retain=5);
};
```

Here *ImageList* is a simple data storage object that contains the name of a possible match for an unknown face and the corresponding distance value. The *nearestNeighbor* method will return the n best identity matches for *unknown*; the default value is five but this can be configured.

Both the Eigenfaces and Fisherfaces recognizer classes are fairly similarly structured. Both have a single constructor that accepts an OpenCV matrix of training images. The

constructor of the Eigenfaces recognizer also offers the option to drop an arbitrary number of principal components. As Figure 4.3 illustrates, both recognizers use the OpenCV *PCA* class for principal component analysis. The Fisherfaces recognizer also makes use of an LDA (linear discriminant analysis) class of our own design. This class implements LDA as described by Belhumeur et al. (1997) and its interface is patterned on the OpenCV *PCA* class. The LDA class uses the Armadillo linear algebra library for eigenvalue calculations. Projection of the training images into PCA or LDA space is done on instantiation.

Recognition is performed by passing a pre-processed recognition image depicting an unknown face to the *recognize* method. This method returns the five best identity matches for the unknown face. The number of identity matches returned can be changed by setting the *retain* parameter. For this prototype of our plug-in we followed the simple policy of designating a face as a training face when its identity is confirmed by the user. For real world applications this is problematic since the set of training faces would grow very quickly and it might be advantageous to hard-code an upper bound on the size to which the training set for each class can grow. For the purposes of our experiments, however, this is not a major issue.

Chapter 5

Recognizer Validation

In this chapter we describe an experiment we conducted to verify the effectiveness of the face recognizers we implemented. To this end we chose to repeat a standard face recognition experiment under 'laboratory conditions' that had already performed by a third party (Marcialis & Roli, 2002) on a well known face database, hoping to replicate their results. In Section 5.1 we describe our experimental setup while Section 5.2 is used to describe our results and Section 5.3 contains a short summary.

5.1 Experimental Setup

In Section 5.1.1 we detail the hardware and software we used. In Section 5.1.2 we describe the datasets we used. In Section 5.1.3 we outline the experimental protocol.

5.1.1 Hardware and Software

The recognizers were tested on Ubuntu Linux 11.04 running in a Parallels VM (Desktop edition) version 6.0.12106 which itself ran on an Apple MacBook, (model identifier 5,1) with 4GB of RAM. The plug-in was linked against OpenCV version 2.2, Armadillo version 2.2.4 and SQLite 3.

5.1.2 Data Set

In order to test our recognizer we used the same face databases used by used by Marcialis and Roli. The first of these is the AT&T face database which consists of 400 images of

40 subjects with 10 images per class. The images are all fully frontal, well lit and contain moderated variations in expression and rotation. The second database is the original Yale database. This face database consists of 165 images of 15 subjects with 11 images per class and it contains more pronounced variations in lighting and expression than the AT&T database does. The copy of the Yale database we obtained turned out to contain some duplicate images which, according to the README file, is because some of the original image files became corrupted. We carefully weeded these duplicates out by dropping the files named *subject<n>.normal.gif*, which eliminated the problem.

5.1.3 Experimental Protocol

Marcialis and Roli performed a two phased experiment. In the first phase they conducted recognition measurements using single Eigenfaces and Fisherfaces recognizers in isolation. In their second phase they ran both Eigenfaces and Fisherfaces recognizers in parallel, combined their results using a variety of algorithms and compared those results with the results from phase one. We restricted ourselves to repeating only phase one where Marcialis and Roli tested their Eigenfaces and Fisherfaces recognizers in isolation since all we wished to do was to verify the efficiency of our recognizers.

We divided our test image sets like Marcialis and Roli did. The face databases contained ten images per individual. The set of images for each individual was divided such that five images were reserved for training and the remainder for testing. Like Marcialis and Roli we repeated the test for 10 random divisions of the data set into five training and five test images and then calculated the average recognition percentage for the ten runs.

5.2 Results

The results of our series of experiments can be seen in Table 5.1. The table shows the results of running the 10 random divisions for both the AT&T face database and the Yale face database using both the Eigenfaces and the Fisherfaces recognizer.

We set out to verify that our recognizers are working properly and we achieved this by confirming the results obtained by Marcialis and Roli in phase one of their experiments.

	Eigen		Fisher		Fisher (OpenCV)
	AT&T	Yale	AT&T	Yale	Yale
R1	94.5%	81.3%	94.5%	93.3%	68.9%
R2	94.5%	74.7%	93.0%	98.7%	66.6%
R3	94.5%	77.3%	90.0%	93.3%	57.8%
R4	93.5%	74.7%	91.5%	96.0%	75.6%
R5	94.5%	81.3%	92.0%	96.0%	72.2%
R6	96.5%	76.0%	93.0%	96.0%	72.2%
R7	93.5%	80.0%	94.0%	100.0%	53.3%
R8	93.0%	86.7%	91.5%	97.3%	75.6%
R9	95.0%	85.3%	93.5%	96.0%	77.8%
R10	92.5%	76.0%	92.5%	90.7%	74.5%
Avg	94.2%	79.3%	92.5%	95.7%	69.5%

Table 5.1: Face recognizer performance. All numbers are recognition rates in percentages. Fisher (OpenCV) refers to a version where the OpenCV eigen solver was used.

5.2.1 Eigenfaces

When running our Eigenfaces recognizer on the AT&T face database we obtained results that closely mirror those obtained by Marcialis and Roli. Their experiment yielded an average recognition rate of 94.7% with an Eigenfaces recognizer for this database which is very close to our 94.2%. The results obtained when running our Eigenfaces recognizer on the Yale face database are slightly worse than the recognition rate obtained by Marcialis and Roli. They reported an average recognition rate of 83.0% for the Yale database whereas our experiment only yielded a recognition rate of 79.3%.

This difference (3.7%) can be explained by the fact that the Yale database exhibits more lighting variance than the AT&T database. It could be that Marcialis and Roli dropped the first three principal components in their Eigenfaces recognizer which decreases sensitivity to lighting variations (Belhumeur et al., 1997; Zhao et al., 2003). Since our Eigenfaces recognizer is based on OpenCV whose principal component analyzer does not offer this facility we could not test this hypothesis. Another possible explanation is that the images in our copy of the Yale face database contain a greater amount of background features than is usual for these scientific face databases. It is possible that Marcialis and Roli may have cropped this away before performing their experiment which would go some way toward explaining the difference in our results. Without knowing the exact set divisions and PCA analyzer configuration used by Marcialis and Roli and without having access to the exact same copy of the Yale face database they used it is impossible to know.

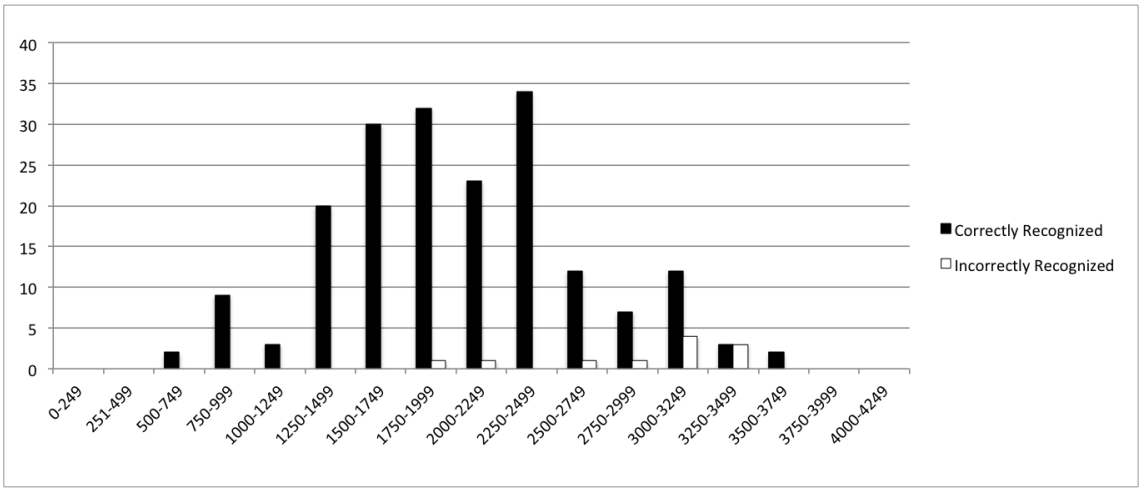
5.2.2 Fisherfaces

When testing our Fisherfaces recognizer with the AT&T face database we obtained an average recognition rate of 92.5% that was almost on par with the recognition rate achieved by our Eigenfaces recognizer but somewhat lower than the 96.0% obtained by Marcialis and Roli. When we tested our Fisherfaces recognizer with the Yale face database, however, we obtained a significantly better average recognition rate of 95.7% than we did with our own Eigenfaces recognizer. Remarkably, we also did considerably better than Marcialis and Roli who only obtained an average recognition rate of 82.8% when testing their Fisherfaces recognizer with the Yale database.

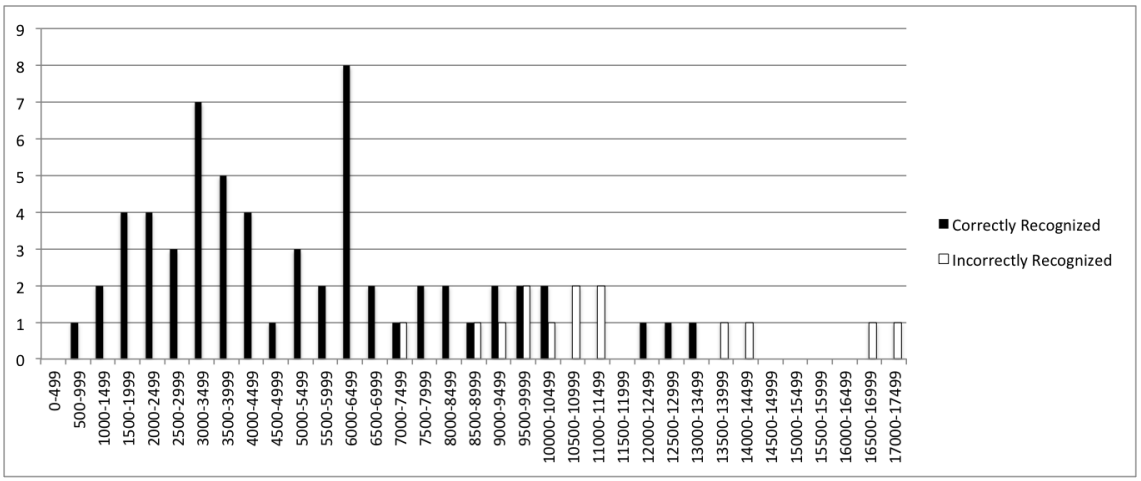
It is clear that both our recognizers are competitive with those used by Marcialis and Roli and our Fisherfaces recognizer in particular performs better on the more difficult and lighting variant Yale face database than Eigenfaces does. We are at a loss to explain why our Fisherfaces recognizer performed better than the one used by Marcialis and Roli. One explanation we can think of is that during the development of our Fisherfaces recognizer we encountered a problem with the OpenCV *eigen* function, which is due to the fact that it is unable to find the eigenvalues of non-symmetric matrices. After substituting OpenCV's built in linear algebra routines for Armadillo, a more powerful linear algebra library we began to see behavior more consistent with that reported in (Belhumeur et al., 1997) and (Zhao et al., 2003). Marcialis and Roli do not offer any implementation details for their recognizers so it is not possible for us to do more than speculate that this might be the cause of the discrepancy. For the sake of completeness we repeated our experiment with the OpenCV *eigen* function being used in place of Armadillo. The results can be seen in column six of Table 5.1. The recognition rate obtained when testing our Fisherfaces recognizer thus configured with the Yale database, actually dropped well below that of our own Eigenfaces recognizer when tested with the same database (column tree).

5.2.3 Separability

Separability is the ability to divide distance values returned by the face recognizer into correct and probably incorrect values. We tried to establish the separability of the recognition data obtained from these face databases. Anyone who has used iPhoto or Picasa will notice that even after receiving extensive training both browsers will still label some faces as unknown and both of these browsers have quite small false positive rates in face recognition. This means that both Picasa and iPhoto have a mechanism to evaluate whether the identification of a face is correct or whether it is likely to be incorrect.



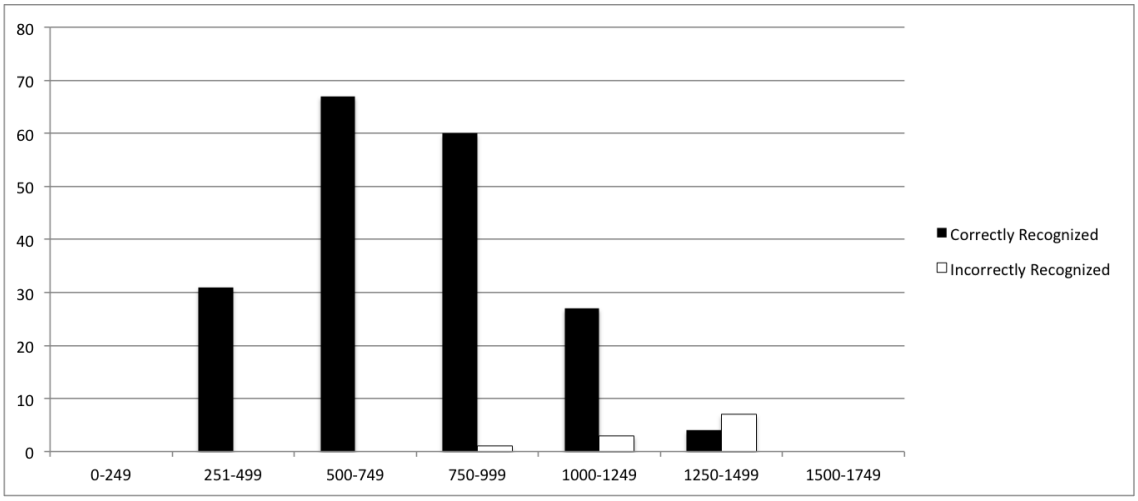
(a) Eigenfaces with the AT&T database.



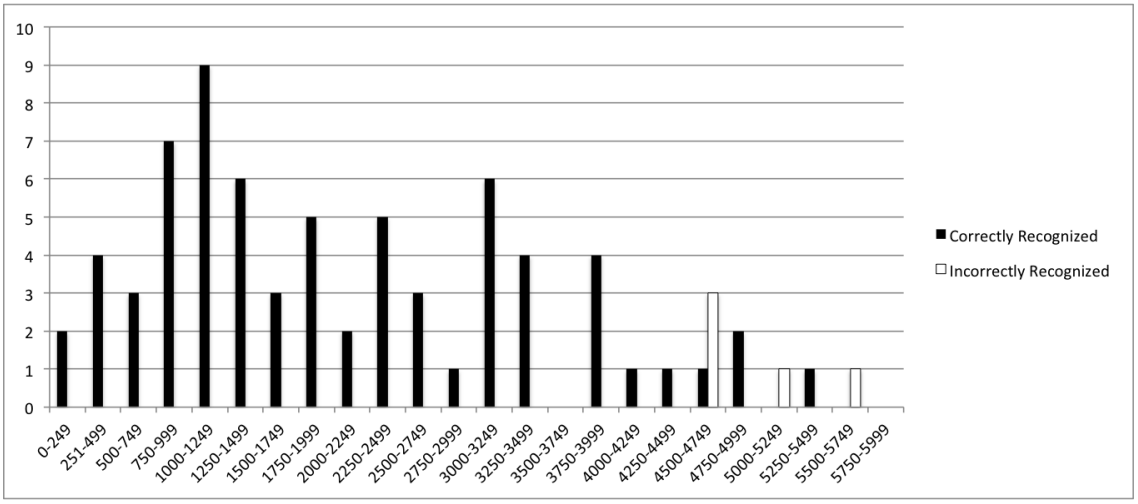
(b) Eigenfaces with the Yale database.

Figure 5.1: Separability analysis, Eigenfaces. The x-axis contains distance values, the y-axis contains number of matches.

In order to do this, one needs to be somehow able to divide the distance values provided by the face recognizers into these two sets by establishing a cut-off value. This is best illustrated by an example. Consider Figure 5.1. The x-axis contains distance values, the y-axis contains number of matches. These two graphs illustrate the results of running Eigenfaces on the AT&T database and the Yale database. Both these databases are neatly pre-normalized which mostly eliminates distortion due to bad normalization. As one would expect the falsely identified distance values bunch up on the right hand side of the x-axis while the correct ones bunch up to on the left hand side. As we had hoped this allows us to set a distance cutoff value such that any distance that exceeds this cutoff value is considered highly likely to be a falsely recognized face and could be labeled as



(a) Fisherfaces with the AT&T database.

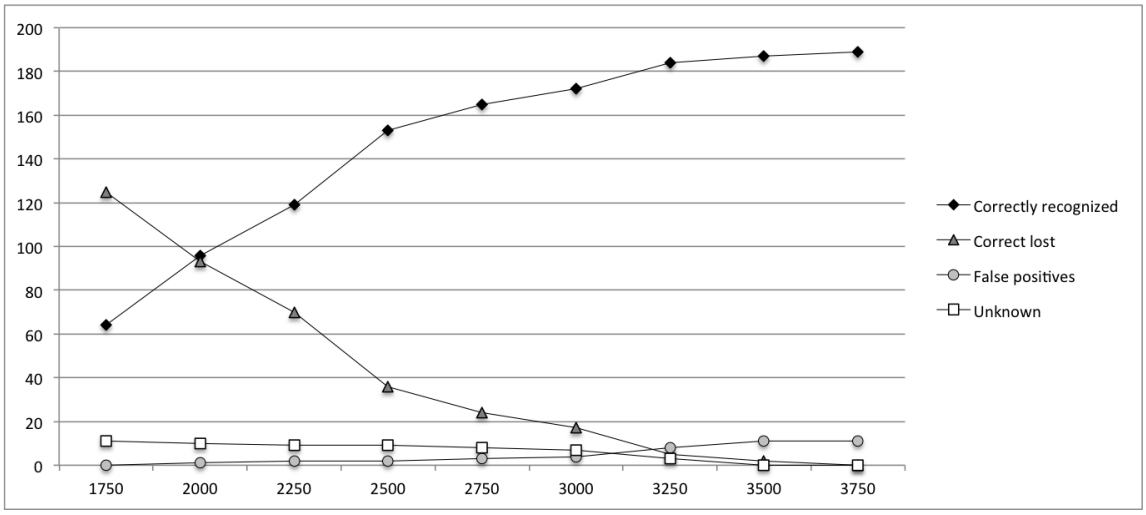


(b) Fisherfaces with the Yale database.

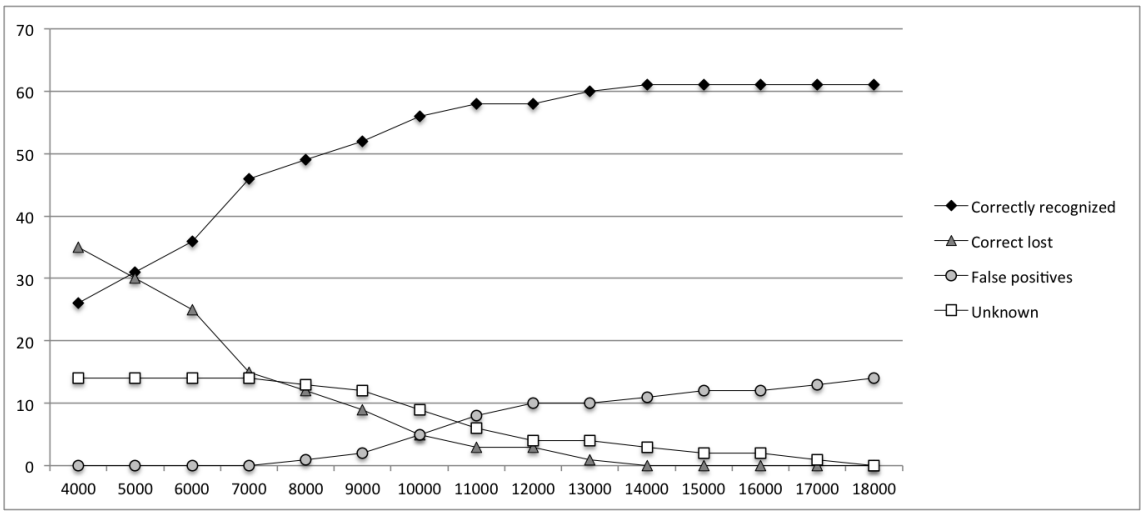
Figure 5.2: Separability analysis, Fisherfaces. The x-axis contains distance values, the y-axis contains number of matches.

an *unknown face* by our plug-in just as Picasa and iPhoto do. Figure 5.2 shows the same statistics for the Fisherfaces recognizer.

Both our Eigenfaces and Fisherfaces recognizers exhibited a fairly high degree of separation between the distance values for wrongly identified faces and correctly identified ones. It is possible to establish a cutoff point that allows one to write off distance values that are too high as being highly probably false positives. Consider Figure 5.3, the graphs depicts correctly recognized faces, correctly recognized faces lost because they were beyond the cutoff point, false positives and unknown faces for different cutoff distances. Data from the Eigenfaces recognizer was used to produce both graphs, Figure 5.3(a) is for the AT&T database Figure 5.3(b) is for the Yale database. Note that as the cutoff value



(a) Cutoff with Eigenfaces and the AT&T database.



(b) Cutoff with Eigenfaces and the Yale database.

Figure 5.3: Separability analysis, Fisherfaces. The x-axis contains the cuttoff value, the y-axis contains number of matches.

goes up the number of correctly recognized faces and false positives increases while the number of lost correct faces and unknown faces goes down. For each face database there is an optimal cutoff value. For the AT&T database this optimal cutoff value is about 2750-3000 while for the Yale database it is about 9000-11000. Thus our problem is not just to achieve separation between the set of distance value for correctly recognized and incorrectly recognized faces, we must also find a way to calculate the correct optimum cutoff value which can vary between face databases. Both of these problems require further research.

5.3 Summary

For our Eigenfaces recognizer we were able to repeat the results obtained by Marcialis and Roli when they tested their Eigenfaces implementation in isolation. Our Fisherfaces recognizer obtained significantly better results than Marcialis and Roli did when they tested their Fisherfaces implementation in isolation. Results indicate that both our recognizers seem to achieve a degree of separability between probably correctly identified faces and ones that are probably wrongly identified when the recognizers are tested with highly normalized scientific face databases.

Chapter 6

Effectiveness in a Browsing Scenario

In this chapter we describe an experiment to measure the effectiveness of our plug-in in a browsing scenario. These experiments are aimed at comparing the performance of the ObjectCube plugin with two leading commercial browsers, Google Picasa and Apple iPhoto. We measure two performance metrics. The first is face detection performance and the second metric we measure is face recognition performance. We also attempt to quantify how effective our strategy of returning multiple matches for face identity is, as opposed to the strategy used by Picasa and iPhoto which only return the best identity match.

In Section 6.1 we describe the experimental setup and protocol. In Section 6.2 we discuss the peculiarities of face tagging in Google Picasa, Apple iPhoto and our preliminary PhotoCube tagging interface. Section 6.3 is used to describe our results and Section 6.4 contains a short summary.

6.1 Experimetal Setup

In Subsection 6.1.1 we detail the hardware and software we used. In Subsection 6.1.2 we describe the experimental data set used. In Subsection 6.1.3 we describe our experimental protocols and metrics.

6.1.1 Hardware and Software

We used Picasa version 3.8.9.390 and iPhoto 9 for our experiments which were performed on Mac OS X 10.7.2. The ObjectCube plug-in was tested using a preliminary PhotoCube

Pose	Set S1	Set S2	Total S1+S2
Profile	6.6%	3.0%	5.0%
Frontal	88.6%	76.0%	83.0%

Table 6.1: Proportions of profile and frontal faces.

GUI written in QT 4.7.4 on Ubuntu Linux 11.04 running in a Parallels VM (Desktop Edition) version 6.0.12106. All experiments were conducted on an Apple MacBook, (model identifier 5,1) with 4GB of RAM. The plug-in was linked against OpenCV version 2.2, Armadillo version 2.2.4 and SQLite 3.

6.1.2 Data Set

In our experiments we used two sets of images, one for training and one for testing. Both these test sets were taken from a sequence of 1,140 chronologically continuous holiday snaps taken during a hiking trip in Iceland.

The two image sets used for performance testing each consisted of one hundred images. Care was taken to ensure, that set S1 predated set S2. This enabled us to simulate a user processing two days worth of photographic material in two face tagging sessions. The images contained overwhelmingly individuals whose identity was known to us and extremely few other faces. The individuals we deemed to be our *persons of interest* for the purpose of face recognition was a group of hikers. This group was in turn subdivided fairly cleanly into smaller nuclear family groups. Table 6.1 shows how the two sets break down into profile and frontal faces where a frontal face being defined as a face where only one eye is visible. The composition of the group by age and sex is detailed in table 6.2, a more detailed breakdown can be found in Table 6.3. There were seventeen persons of interest and each person in the group was identified by a single letter from the alphabet. Each person of interest appeared at least five times in each image set and each image set contained at least five landscape images that did not depict a human. No animal visages featured prominently in any of the images.

Before proceeding with any experiments we established a ground truth. This was achieved by analyzing the two image sets and counting all visible members of our persons of interest group that appeared in these photographs. People were counted regardless of whether their faces were partially visible or even whether the person's face was visible at all in the image. Any member of the persons of interest group that could be visually identified in any way whatsoever was counted. We found that members of our persons of interest group appeared, 333 times in image set S1, and 267 times in image set S2. This gave us a

Figure 6.1: Images A and B are easily detected and normalized, image D demonstrates occlusion while image C completely defeated the object detector. Image E is a group shot that shows the entire persons of interest group.

total ground truth of 600 individuals from our persons of interest group that were visually identifiable in some way by a human in both image sets.

6.1.3 Experimental Protocol

For each image browser being tested we repeated the same protocol. We began by importing image set S1 into the browser. The number of faces automatically detected by the browser was determined and the faces were then fully tagged. We used a *tagging-key* to ensure that the face recognition subsystems of all browsers received the exact same training data. This tagging-key consisted of a list that detailed which individuals should be tagged in each image in set S1, see Table 6.3 for an overview of the group of persons tagged. All faces that had been automatically detected were tagged with the name of the relevant person and faces that the face detection system had missed were added by hand. Details of the tagging procedure for each browser can be found in Section 6.2. Once the tagging phase was complete the second image set was imported. We now established the detection rate once again and then proceeded to measure face recognition performance.

The first step in measuring face recognition performance was to establish a ground truth and selecting our set of persons of interest as described in Section 6.1.2. We then imported image set S1 into the browser and fully tagged it as described in Section 6.2. For each image in the set we counted the number of automatically detected faces. After fully tagging image set S1 we proceeded to import image set S2. Once again we counted suc-

Age group	Male	Female	Total
Children	4	2	6
Adolescents	1	2	3
Adults	4	4	8
Total	10	8	17

Table 6.2: The people in the browsing scenario.

cessfully auto detected faces for each image in the set and then quantified how many faces were correctly identified and misidentified. In the case of our plug-in we also kept track of which false positives could be quickly corrected using the drop down box described in Subsection 6.2.3. Finally we calculated what percentage of the ground truth the browser had correctly detected and identified automatically.

We performed one experimental run for Picasa and iPhoto. For our plug-in we performed two runs, one run for each of the recognizers that we had developed. We proceeded on the assumption that most users will only tag faces that are reasonably visible and constitute the main subjects of the image and possibly also other reasonably clearly visible individuals in the background. Thus we tagged all visible faces from our persons of interest group including ones with mild occlusions such as images where a hand, a finger or a bottle obscured the face. We also tagged heavily occluded faces such as an individual wearing a scarf or dust goggles. The guiding criterion was that the face should be completely within the image.

6.2 The Face Tagging Procedure

In this section we briefly explain how facial tagging is performed in each of the image browsers we tested. Subsection 6.2.1 describes tagging in Picasa, 6.2.2 describes tagging in iPhoto, and 6.2.3 describes tagging in our temporary test GUI.

6.2.1 Picasa

Picasa does not import images into a library. It indexes existing image collections in the computer's file system. The user adds an image set to the Picasa database with a folder manager that can be accessed via the *Tools* menu. Once the desired image set has been indexed it is displayed in a central image list pane in the Picasa user interface. It can take some time after indexing is finished before Picasa's face recognition subsystem has processed all the images.

	Gender		Age			Appearances	
ID	Male	Female	Child	Teen	Adult	Set S1	Set S2
A	x		x			21	9
B	x		x			28	10
C	x				x	9	9
D	x				x	17	9
E		x		x		22	14
F		x	x			26	17
G		x			x	14	10
H	x		x			17	9
I		x	x			17	16
J	x		x			19	9
K		x			x	15	19
L	x				x	13	12
M		x		x		16	10
N	x				x	11	15
O	x			x		24	15
P		x			x	10	8
Q		x			x	16	12
Total:	9	8	6	3	8	295	203

Table 6.3: Overview of the photo collection.

The user can then select an arbitrary image in the set which is displayed as a thumbnail list in the main display pane. This causes the image itself to be displayed in the main display pane and the faces Picasa detected in that particular image are displayed in a side pane known as the *People* pane (see Figure 6.2). If the identity offered by Picasa is correct all the user has to do is confirm it by pressing a button next to the entry for that face in the people pane. Unconfirmed automatic identifications are always followed by a question mark.

If Picasa was unable to identify a face or the identity it offered is wrong the face must be manually tagged. There is more than one way to tag a face. The fastest one is to hover the mouse pointer over the face which causes a floating box containing an attached text field to appear. The user enters the proper name for this face in the text field which brings up suggested names from Picasa's internal people list to speed up identification. If this person has already been added to Picasa's people list all the user has to do is press the *Return* button which confirms the identity and navigate to the next image with an arrow key.

If the person the face belongs to is not in Picasa's internal people list a pop-up window is displayed on pressing the return button. The user must fill in a form identifying the subject the face belongs to before navigating to the next image as described above.

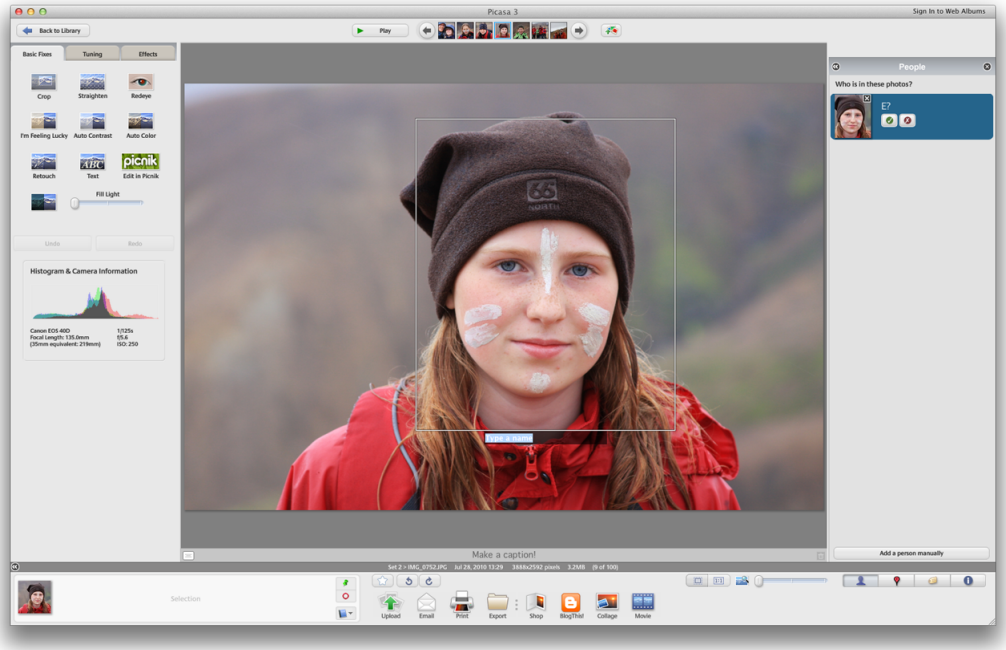


Figure 6.2: The Picasa main window in image-view. On the right is the *People* pane. The main display pane in the middle shows a selected image, the face has been framed and a name has been selected from the people list. The button in the lower right corner is used to add faces manually, the one in the upper right corner takes one back to list-view.

If a face in the image has not been detected the user must add it manually. This is in done in image-view by pressing a button in the lower left corner of the screen which causes a face-frame to appear which the user must position. Once that is done the user can proceed as described above.

6.2.2 iPhoto

With iPhoto, face tagging is somewhat simpler than it is with Picasa. Once iPhoto has finished importing the image set, the user must wait until an indicator next to the icon in the navigation pane that represents the newly imported image set indicates that processing of the set has finished. Presumably this processing involves, among other things, face detection and face recognition.

Once this processing is finished the user can simply select the imported image set, which takes the user to list view, and then double click the first image which takes the user to image view. The user must now put the browser into face tagging mode, by pressing the *Name* button in the lower left corner of the window. This causes a small floating

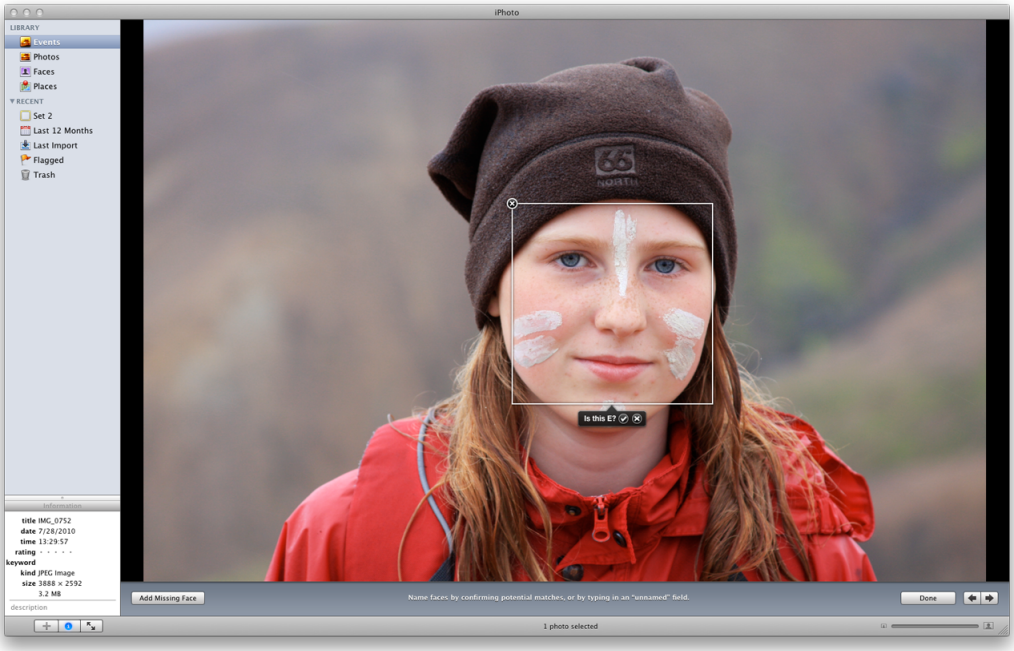


Figure 6.3: The iPhoto main window in image-view with face tagging activated. The main display pane is in the middle of the window. When the system pointer hovers over a face the face-frame is displayed along with the tagging menu. The button in the lower left corner is used to add faces manually. The navigation pane on the right takes one back to list view.

tagging box to be displayed below each face that iPhoto detected in the image (see Figure 6.3).

If iPhoto offers a correct identification the user can simply confirm it by pressing a button. If iPhoto has offered the wrong identity the user must reject the identification and correct it by typing the correct name into the floating tagging box. When typing into the tagging box the browser searches its internal persons list and suggests names that match what the user has typed.

If a face has not been automatically detected by iPhoto it must be added manually. This is done using the *Add Missing Face* button in the lower left corner of the window. This pops up a small face frame with an attached tagging menu which the user must resize and position over the face in question. Once that is done the user can proceed as described above.

Once all faces in an image have been processed the user navigates to the next image. This can be done by either pressing an keyboard arrow key, clicking an arrow button at the bottom of the iPhoto user interface or by using a trackpad gesture.

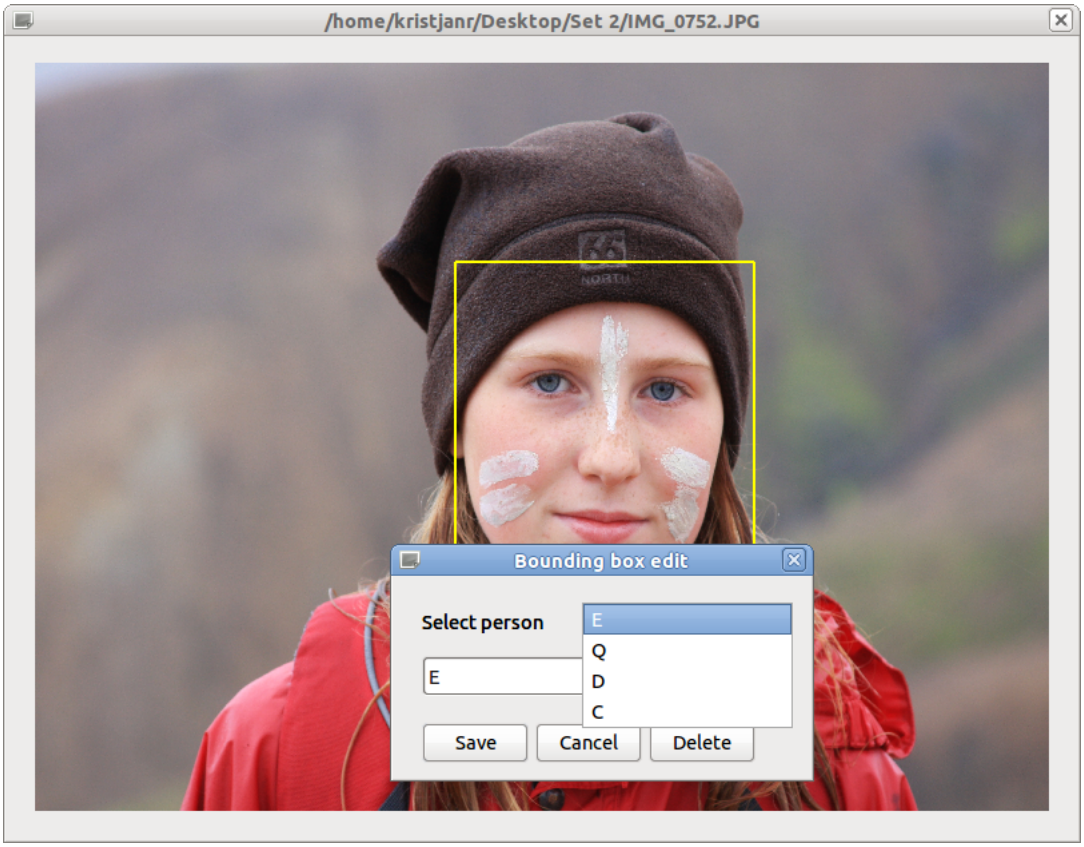


Figure 6.4: The ObjectCube plug-in test GUI. All faces are framed in a red square. Clicking inside the square brings up a dialog. The drop down box contains the n best matches for the identity of this face with the best match being repeated in the text box.

6.2.3 Our Preliminary PhotoCube Tagging GUI

We had originally intended to test our plug-in with the full PhotoCube user interface but due to one of our team members being involved in a serious accident towards the end of the project this did not prove to be possible. We were thus forced to resort to implementing a preliminary test interface. We decided to model this on the interfaces used by Picasa and iPhoto.

Our user interface consists of a simple import wizard that takes as its input a directory containing the image set to be recognized. Once this has been accomplished the import wizard uses our plug-in to process the image and displays the results in a pop-up window which can be seen in Figure 6.4. Every face found is framed by a red square which becomes yellow when the mouse pointer hovers over it. Clicking on the square brings up a pop-up window containing a drop down box. This drop down box contains the n best matches returned by the recognizer for the identity of the framed face. We propose to use the secondary matches to facilitate quick correction in case the recognizer's first choice of identity was false.

Browser	Set 1	Set 2	False Pos.
Picasa	76.6%	66.3%	1.5%
iPhoto	72.4%	65.2%	1.0%
ObjectCube	61.6%	58.8%	16.3%

Table 6.4: Face detection performance (browsing scenario).

If neither the first choice nor the $n-1$ other suggested matches presented in the dialog box are correct the user can type the subject's true identity into the text box. The tagging procedure is terminated by pressing the *Save* button. Once the identity of a face has been confirmed, the square framing it turns green. The user can delete a face at any time. Missing faces are added by simply clicking and dragging a framing box around a face; after this has been done the user can proceed as described above.

6.3 Results

We have designed an experimental protocol and we feel that it is representative of a typical browsing session as performed by an average computer user. We have also devised a way of quantifying the face recognition performance of a number of image browsers and a method for comparing their performance. The protocol we have designed allows us to answer our primary question which is how our plug-in performs when compared to state of the art face recognition software in terms of detection and recognition. In Subsection 6.3.1 we discuss results for face detection. Subsection 6.3.2 deals with face recognition results while Subsection 6.3.3 presents separability results.

6.3.1 Face Detection

The results of our measurements of face-detection performance can be found in Table 6.4. All figures in this table are a percentage of ground truth for the image set in question. Columns two and three are detection rates for each image set and the final column contains the total false positive rate.

To establish face detection rates we counted the number of faces automatically detected by each browser for both test sets. We found that Picasa did a marginally better job at face detection than iPhoto did, while ObjectCube did not perform as well. Both iPhoto and ObjectCube had more false positives than Picasa (see Table 6.4).

As one can see from Table 6.4 our plug-in suffers from a relatively high rate of false positives during the face detection phase. Fortunately, many of these false positives were typically very relatively small features compared to the faces we were tagging. The problem of false positives during the face detection phase can thus be solved relatively easily through a combination of object detector tuning and rejecting detected faces that are smaller than a certain percentage of the total image area. Our rate of correctly detected faces is surprisingly high and indeed higher than we had expected given that we were restricted to using the detector training files that shipped with the OpenCV distribution and the fact that we were not searching for profile faces which Picasa and iPhoto are doing. As one can see in Table 6.1 our Set S2, our test set, is rather light on profile faces but processing profile faces nevertheless has the potential to increase detection and recognition rates somewhat. It is thus clear that there is some room to improve the performance of our object detector's face and feature detection performance by training the object detector ourselves rather than using prepared third party training files. As we described in Section 4.2 we only search for full frontal faces. A large majority of the faces in our test sets are full frontal but searching for profile faces would nevertheless bring us somewhat closer to the detection rates of Picasa and iPhoto.

6.3.2 Face Recognition

The face recognition results (see Table 6.5) were somewhat more interesting than the detection results. All figures are a percentage of ground truth for the test set in question. Column two contains recognition rates. Column three contains the percentage of faces where the recognizer's first choice of identity was wrong but where a correct identification was found among the $n - 1$ secondary results returned by the browser's recognizer. Column five contains false positive rates.

We found that once again Picasa is the winner with a recognition rate of 37.5% of ground truth and a false positive rate of only 3.7%. The runner up was iPhoto with a recognition rate of 26.2% of ground truth and a false positive rate over twice that of Picasa at 8.2%. Neither of these browsers offers more than one possible match per face so their suggestive rates are zero. When using our Eigenfaces recognizer the ObjectCube plug-in achieved a recognition rate of 19.1% of ground truth this rose significantly when we tested our Fisherfaces recognizer who achieved a recognition rate of no less than 23.2% of ground truth. Suggestion rates for Eigenfaces and Fisherfaces were 11.6% and 12.4% of ground truth. This is an encouraging result. If the secondary results could be packaged into a cleverly designed corrective GUI feature, such a feature could significantly cut down on

Browser	Recognized	Suggested	Rec.+Sugg.	False Pos.
Picasa	37.5%	–	–	3.8%
iPhoto	26.2%	–	–	8.2%
ObjectCube (E)	19.1%	11.6%	30.7%	39.0%
ObjectCube (F)	23.2%	12.4%	35.6%	35.0%

Table 6.5: Face recognition performance (browsing scenario).

tagging time. The false positive rate for our plug-in is quite high since we did not have time to tackle the problem of separability in our implementation. This will be discussed at some length below.

When using our Eigenfaces recognizer the ObjectCube plug-in achieved a lower recognition rate than both Picasa and iPhoto. The results did not surprise us since we had not anticipated beating top of the line commercial products in a semester long project. We poured much of our effort into writing two recognizers and this decision appears to have paid off since our Fisherfaces recognizer performed noticeably better than our Eigenfaces recognizer, bringing us close to the recognition rate achieved by iPhoto. This is encouraging both because it reflects the results obtained in the verification experiments described in Chapter 5 and because the face recognition performance of any face recognition system is upper-bounded by the performance of its face-detector and normalizer. Both our normalizer and face detection modules have significant room for improvement and any improvement in these should yield a corresponding improvement in the recognition rate. These issues will be addressed in future versions of our plug-in.

6.3.3 Separability

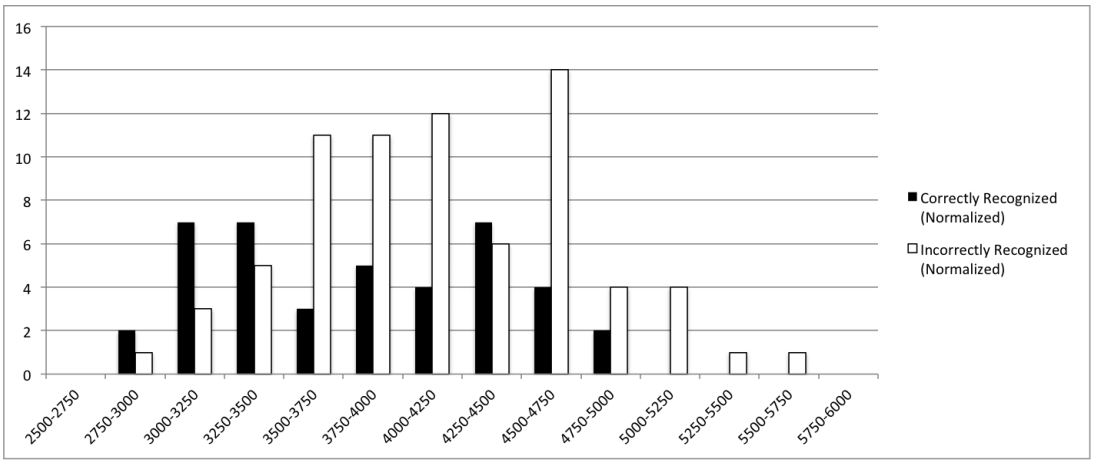
Neither the Eigenfaces recognizer nor the Fisherfaces recognizer returned a set of distance values for our real world image sets that allowed us to achieve the degree of separability that would enable us to label a majority of falsely recognized faces as *unknown*. In an attempt to discover why this is we broke the distance values for image set S2 down into normalized and non-normalized faces. This data was then used to create the histograms in Figures 6.5 and 6.6. At top are distance values for normalized faces, at bottom those for un-normalized faces. As the reader can see the datasets overlap each other closely even when the distance values have been broken down according to whether the unknown face being identified was based on an normalized or un-normalized face.

The verification experiments described in Chapter 5 indicated that our recognizers have a fairly clean break between correctly and incorrectly labeled faces on a pair of highly normalized face databases (see Figure 5.1). We suspect that the reason for this lack of

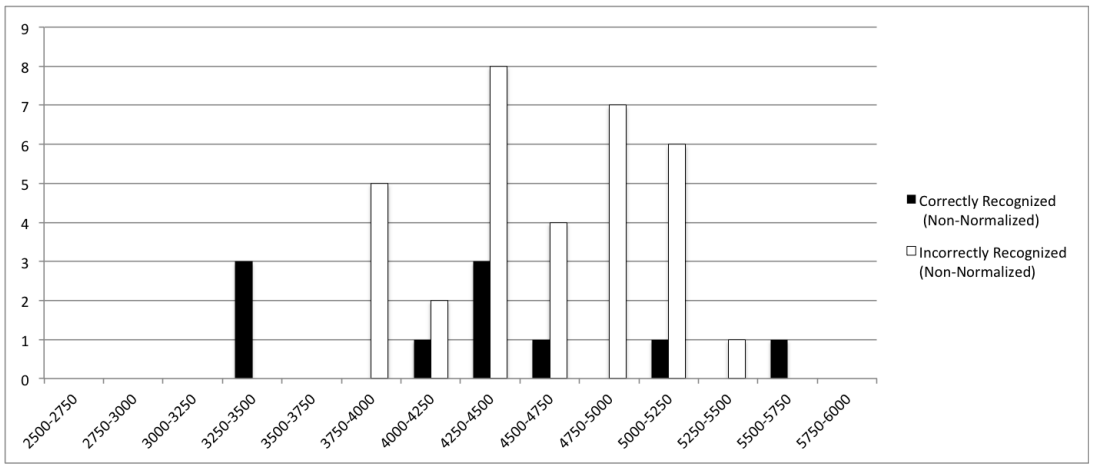
separability with our real world image sets is the relatively primitive automatic normalization we are performing. Another factor that might influence separability is the selection algorithms we are using to choose the best match for the identity of an unknown face. Currently we are using a very simple form of nearest neighbor search that uses Euclidean distance. Using a more sophisticated method for choosing the best matching identity might bring improvements in separability. Further research into this area is clearly needed.

6.4 Summary

We attempted to quantify how well our plug-in performs when compared with the leading commercial image browsers Picasa and iPhoto. Picasa had the highest detection rate closely followed by iPhoto with our plug-in coming last. It was also Picasa which scored the highest recognition rate, followed by iPhoto. Our plugin-achieved the lowest recognition rate although when tested with our Fisherfaces recognizer it came within 3% of matching iPhoto. In stead of returning only the best match for an unidentified face we returned a set of the n best identity matches with the intention of using the secondary matches to implement a quick correction feature. This idea worked better than we expected and returned a correct secondary identity match in a significant number of cases. False positives are an issue during the detection phase but these are relatively easy to fix. The problem of false positives during the recognition phase ties into the subject of separability which remains a problem for us that requires more work. We suspect that the separability issue is due to insufficient normalization of recognition images and possibly due to our primitive method of determining identity matches.

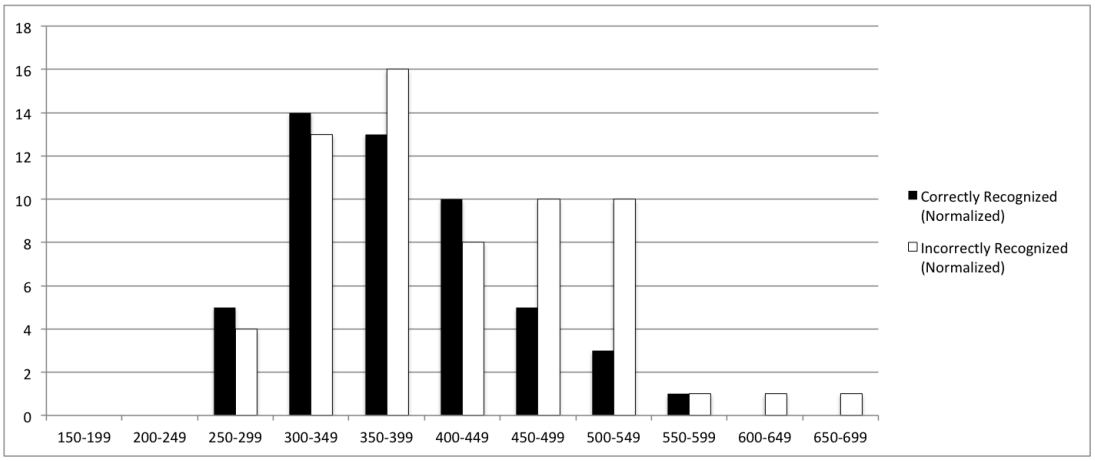


(a) Unknown face was normalized

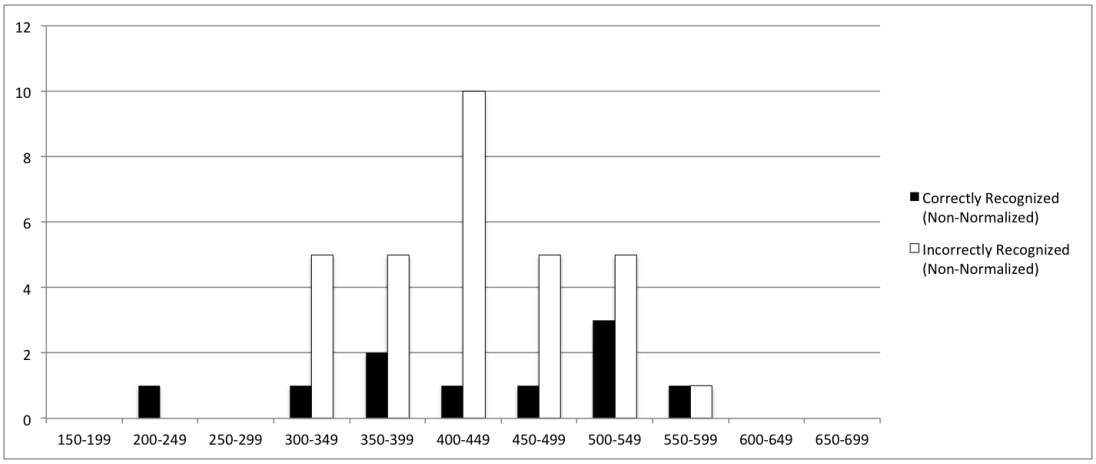


(b) Unknown face was not normalized

Figure 6.5: Separability histogram for the Eigenfaces recognizer. The x-axis contains distance values, the y-axis contains number of matches.



(a) Unknown face was normalized



(b) Unknown face was not normalized

Figure 6.6: Separability histogram for the Fisherfaces recognizer. The x-axis contains distance values, the y-axis contains number of matches.

Chapter 7

Future Work

We chose to dedicate much of our effort on implementing a Fisherfaces recognizer in addition to the Eigenfaces recognizer we implemented originally. There are, however, other areas that need improvement, some of whom have been mentioned in Chapter 6. What follows is a list of areas of future work.

There are several aspects of our software that need to be improved and where speed and resource usage optimizations can be made. We would also like to replace Armadillo, the linear algebra library we used, with one that has fewer dependencies. Armadillo is a wrapper for the ATLAS and LAPACK libraries but there are linear algebra libraries that have no dependencies other than the standard C++ template library. There are also many ways in which we can improve the speed of our plug-in. Both the recognizer and the normalization functions in particular can be optimized significantly for speed.

Due to various factors beyond our control we were not able to fully integrate the plug-in with the ObjectCube database engine nor were we able to integrate face recognition into the PhotoCube browser. We had originally intended to create a user interface in PhotoCube for face tagging. This user interface was to rely extensively on dragging and dropping rather than the linear face-tagging procedures used by Picasa and iPhoto. Once work on ObjectCube and PhotoCube can resume and this user interface has been implemented we will conduct a program of usability tests since we feel that user interface design can have just as much of an effect on tagging speed as recognizer performance.

We would like to improve our face normalization procedure to the point where we are able to fully normalize as many faces as possible using more facial features than just the eyes. In order to do this we will have to improve the efficiency of facial feature detection in particular. Improvements in normalization performance will hopefully also bring with it a significant improvement in separability. Another thing that could possibly be affecting

separability is the way we determine the identity of unknown faces. Currently we are only using a very simple form of nearest neighbor search. There are many other and much better methods to establish the best identity match for an unknown face. Increasing the separability of our face recognition results will allow us to distinguish between faces that are correctly identified and those that are highly probably incorrectly identified. This is an area of research that requires further work and that will demand a large amount of testing and experimentation.

We know that our object detector can be updated for profile face detection fairly easily. We tested the relavant detector training files that ship with OpenCV and we were not satisfied with their performance. We would like to create or own training files for profile faces. This would bring our detection rate into the same region as Picasa and iPhoto. We also feel that we can improve on the performance of our object detector during the feature detection phase by producing our own training files for eye and mouth detection.

We must modify the plug-in to do its processing in the background rather than at import time as it does now. Both iPhoto and Picasa do this and it is an important feature. A simple way to implement this feature is to simply thread the import process. A further refinement would be to link the import process to the user's browsing activity such that if the user browses unprocessed images these would be moved to the top of the importers list of pending objects. This would, however, necessitate the expansion of our plug-in protocol to allow it to track user interface events.

We would like to perform exhaustive testing to further validate the correctness of our face recognizers. This includes gathering further performance data both under 'laboratory conditions' and while using more real world image sets. We would particularly like to test the recognizers with a larger number of real world image sets that display variations in lighting as well as the facial expressions and changes in the physical appearance of the image subjects.

Chapter 8

Conclusions

We have designed a plug-in interface for the ObjectCube database engine and implemented a prototype plug-in. As part of this we also implemented two different face recognition algorithms, Eigenfaces and Fisherfaces, plus an automatic face normalizer. We developed an experimental protocol to compare the performance of our plug-in to that of two leading commercial image browsers. We found that our plug-in achieves recognition rates that are close to the lower end of what these browsers are capable of. The clear winner in our performance measurements is Google Picasa, followed by Apple's iPhoto with our plug-in coming as a close third in terms of face recognition rate. We do, however have a significant amount of further work that needs to be done. Most of the effort will be focused on improving automatic detection in arbitrary images and automatic face normalization.

Bibliography

Belhumeur, P. N., Hespanha, J. P., & Kriegman, D. J. (1997). Eigenfaces vs. Fisherfaces: Recognition Using Class Specific Linear Projection. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, *19*(7), 711–720.

Coi, Y. M. (2011). *Web-enabled OLAP tutorial.* Available from `http://www.ischool.drexel.edu/faculty/song/courses/info607/tutorial_OLAP/`

Degtyarev, N., & Seredin, O. (2010). Comparative Testing of Face Detection Algorithms. In *ICISP* (Vol. 6134, pp. 200–209). Springer-Verlag.

Fischer, S., & Duc, B. (1997). *Shape Normalisation for Face Recognition* (J. Bigün, G. Chollet, & G. Borgefors, Eds.). Springer Berlin / Heidelberg.

Haar-like features. (2011, 9). Retrieved 3.1.2012, from `http://en.wikipedia.org/wiki/Haar-like_features`

Hjelmås, E., & Low, B. K. (2001). Face Detection: A Survey. *Computer Vision and Image Understanding*, *83*(3), 236–274.

Kriegman, D. J., & Kriegman, D. J. (2002). Detecting Faces in Images: A Survey. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, *24*(1), 34–58.

Lienhart, R., & Maydt, J. (2002, 9). An Extended Set of Haar-like Features for Rapid Object Detection. In *IEEE ICIP* (Vol. 1, pp. I-900–I-903). IEEE.

Marcialis, G. L., & Roli, F. (2002). Fusion of LDA and PCA for face recognition. In *Proceedings of the Workshop on Machine Vision and Perception, 8 th Workshop of the Italian Association for Artificial Intelligence* (pp. 1–5). Siena (Italy).

Martinez, A. M. (2002). Recognizing imprecisely localized, partially occluded, and expression variant faces from a single sample per class. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, *24*(6), 748–763.

Martinez, A. M., & Kak, A. C. (2001). PCA versus LDA. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, *23*(2), 228–233.

Messom, C., & Barczak, A. (2006). Fast and Efficient Rotated Haar-like Features Using Rotated Integral Images. *Australian Conference on Robotic and Automation*, 1-6.

Papageorgiou, C. P., Oren, M., & Poggio, T. (1998). A General Framework for Object Detection. *Sixth International Conference on Computer Vision IEEE*, *6*(1), 555–562.

Seo, N. (2011). *OpenCV Haar-training (rapid object detection with a cascade of boosted classifiers).* http://note.sonots.com/SciSoftware/haartraining.html.

Sigurþórsson, H. (2011). *PhotoCube, A Multi-Dimensional Image Browser*. Master's thesis, Reykjavík University.

Tómasson, G. (2011). *ObjectCube - A Generic Multi-Dimensional Model For Media Browsing*. Master's thesis, Reykjavík University.

Turk, M., & Pentland, A. (1991). Eigenfaces for Recognition. *Journal of Cognitive Neuroscience*, *3*(1), 71–86.

Viola, P., & Jones, M. (2001). Robust Real-time Object Detection. *International Journal of Computer Vision*, *57*(2), 137–154.

Wang, H., Yan, S., Huang, T., Liu, J., & Tang, X. (2008). Misalignment-Robust Face Recognition. *IEEE Conference on Computer Vision and Pattern Recognition*, *19*(4), 1087–1096.

Wiskott, L., Fellous, J.-M., Krüger, N., & Malsburg, C. V. D. (1997). Face Recognition by Elastic Bunch Graph Matching. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, *19*(7), 775-779.

Zhang, C., & Zhang, Z. (2010). *A Survey of Recent Advances in Face detection* (Tech. Rep. No. MSR-TR-2010-66). Microsoft Research.

Zhao, W., Chellappa, R., & Nandhakumar, N. (1998). Empirical Performance Analysis of Linear Discriminant Classifiers. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition* (pp. 164–169). Santa Barbara, CA , USA.

Zhao, W., Chellappa, R., Phillips, P. J., & Rosenfeld, A. (2003). Face Recognition: A Literature Survey. *ACM Computing Surveys*, *35*(4), 399–458.

School of Computer Science
Reykjavík University
Menntavegi 1
101 Reykjavík, Iceland
Tel. +354 599 6200
Fax +354 599 6201
www.reykjavikuniversity.is
ISSN 1670-8539