Computer Vision in Facebook Applications

Bachelor’s Thesis

Elena Teunissen
Department of Computer Science

Advisor: Dr. Mukta Prasad, Matthias Dantone
Supervisor: Prof. Dr. Luc van Gool

August 19, 2011
Abstract

The ongoing research in the field of object recognition, becomes an integral element of many applications surrounding us in the everyday life. Our project aims to explore the role of computer vision in the social platforms. The goal of this thesis was to implement a Facebook application that enables both automatic and manual tagging based on successful face recognition results. Furthermore, the application should perform a simple statistic analysis of friendship ties by means of calculating the amount of tags on the user’s photos and on the photos the user has been tagged on.
Acknowledgements

I’d like to thank my supervisor Mukta Prasad for providing the fascinating thesis idea that honed my hitherto nearly inexisting programming skills and allowed me to grow as a computer scientist. Many thanks for her guidance, patience and words of advice.

Many thanks to Matthias Dantone for his input and efforts to make this thesis a solid work. I hope his mail server didn’t explode from the sheer amount of e-mails exchanged throughout the project.

I can’t begin to thank Reto Lindegger for his great technical help, abilities to eliminate bugs and unyielding support during the entire endeavour. Many apologies for having to deal with my scatterbrained phases and occasional outbursts of frustration.

Finally, I want to thank my father, Bram Scheidegger and Armin Schopfer for proof-reading the report, pointing out some logical inconsistencies and trying to smooth my bad English.
# Contents

1 Introduction .......................... 1
   1.1 Goal .................................. 1
   1.2 Motivation ............................ 1
   1.3 Project Specification ................. 1
   1.4 Application Workflow ................. 2
   1.5 Road Map ............................ 3

2 Facebook API .......................... 5
   2.1 CGI Scripting ......................... 5
      2.1.1 Introduction to CGI ............... 5
      2.1.2 Choice of Platform ............... 6
   2.2 Core Concepts of Facebook API ....... 6
      2.2.1 Legacy REST API .................. 6
      2.2.2 Graph API ........................ 6
      2.2.3 OAuth2.0 Authentication Protocol .... 7
      2.2.4 Permissions ...................... 8
      2.2.5 Tagging with Open Graph API ....... 9
   2.3 Facebook Application ................. 9
      2.3.1 Facebook Application Architecture .... 9
      2.3.2 Creating Application ............. 10

3 Object Detection and Pattern Recognition ................. 11
   3.1 Face Detection ........................ 11
   3.2 Face Recognition ..................... 12
      3.2.1 Simple Euclidean Distance .......... 13
      3.2.2 PCA .............................. 14
      3.2.3 Recognition Using Face Alignment by Robust Nonrigid Mapping ...... 15
CONTENTS

4 Computer Vision applied to Facebook 17
  4.1 Application Technicalities ................................. 17
    4.1.1 Structure of the Application ....................... 17
    4.1.2 Pipeline ........................................... 18
    4.1.3 Evaluation .......................................... 21
  4.2 Porting the App to the ETH Server ...................... 22

5 Set Up 25
  5.1 Running CGI Scripts with Apache ...................... 25
  5.2 OpenCV 2.2 ............................................. 26

6 Future Work and Conclusions 27
### List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>Application Index Page</td>
<td>2</td>
</tr>
<tr>
<td>1.2</td>
<td>Application Workflow (simplified version)</td>
<td>3</td>
</tr>
<tr>
<td>2.1</td>
<td>CGI Pipeline</td>
<td>6</td>
</tr>
<tr>
<td>2.2</td>
<td>Server Side Flow</td>
<td>8</td>
</tr>
<tr>
<td>3.1</td>
<td>Haar features used by Viola-Jones</td>
<td>12</td>
</tr>
<tr>
<td>3.2</td>
<td>Simple Euclidean Distance Results</td>
<td>14</td>
</tr>
<tr>
<td>3.3</td>
<td>PCA results</td>
<td>15</td>
</tr>
<tr>
<td>3.4</td>
<td>Recognition with Rectified Images Results</td>
<td>16</td>
</tr>
<tr>
<td>4.1</td>
<td>Hierarchy</td>
<td>18</td>
</tr>
<tr>
<td>4.2</td>
<td>Friendship Ranking Results</td>
<td>20</td>
</tr>
<tr>
<td>4.3</td>
<td>Application Control Flow</td>
<td>21</td>
</tr>
</tbody>
</table>
List of Tables

4.1 Performance Evaluation Results ........................................ 22
Chapter 1

Introduction

1.1 Goal

The focus of this thesis lies in applying face recognition algorithms to images downloaded from the Facebook graph and uploading tags automatically. Additionally, we evaluated closeness between the user and their friends based on the visual results.

1.2 Motivation

Face recognition algorithms play an increasingly major role in our society nowadays. Their usage range from security systems to entertainment and socialising. The techniques continually improve over the years, yielding quite impressive results (e.g. Google’s Picasa) as researchers come up with new ideas.

Social platforms, in particular Facebook, provide a face recognition interface as well. Even though one can argue over intrusion into the user’s privacy, it remains an indisputable fact that a recognition software is a very useful tool that simplifies staying in touch with one’s friends. The possibilities of integrating computer vision aspects into Facebook interface are still vastly unexplored.

1.3 Project Specification

The user should be able to select one of their albums for tagging. Our application provides a graphical interface for this purpose, where for each detected facial
image in the photo a suggestion would be offered to depict that person’s identity. Furthermore we wanted to offer an option where tags could be uploaded for multiple photos at once, which is referred to as “auto-tagging”. The users may only tag their own albums. The system should run on a server that is available 24/7, e.g. an ETH server.

1.4 Application Workflow

When the user loads the app, they send a request to the Facebook server, which contacts the local server for authorisation. After the authorisation has been completed, the application’s index page is displayed on the user’s screen (see screenshot below) with an option to select an album to tag in.

![Application Index Page](image)

Figure 1.1: Application Index Page

All the communication between the user, the local server and the Facebook server occurs through the Canvas, a special container where the Facebook application “lives”. The Canvas itself as well as the authorisation process is omitted from the Figure 1.2 for simplicity. At first, the user selects an album (Step 1 in Figure 1.2). The local server sends a query to Facebook with parameters such as album ID, user ID and the list of user’s friends. Using these parameters, we download the photos from Facebook as depicted by step 2 in the Figure 1.2 unless they are already downloaded. We also download the photos containing the tags of our friends. In the next step, we undertake a visual analysis of the downloaded photos, namely face detection and performing recognition algorithms. Step 4 sends the results from step 3 to the user. Lastly, the user can upload the tags onto Facebook.
Chapter 1. Introduction

The first chapter describes the structure of a Facebook application, demonstrates the process of authorisation and explains how our application came into being (Steps 2,6 in Figure 1.2). The basic behaviour of all web applications is elaborated by CGI (Common Gateway Interface), a protocol specifying the communication between the user’s browser and the server. Any web application consists of one or more CGI scripts—programs written in a language that supports the CGI protocol. The first chapter also considers various CGI scripting languages and assesses their elegance in regards to their integration into Facebook API. The second chapter is devoted to computer vision matters such as working environment, face detection and recognition (Step 3 in Figure 1.2). The third chapter puts all pieces together and describes how exactly computer vision was merged with CGI scripting. The achieved results are summarised and evaluated in the conclusion. Lastly this document provides a glimpse into future work.

1.5 Road Map

This section is a brief summary of the milestones on the road to deploying a Facebook application with computer vision. The first step in the road map was getting acquainted with CGI scripting, Facebook API and the structure of a Facebook application. Once that was covered, we switched to the computer vision part, first acquiring basic theoretic knowledge of several selected algorithms and then implementing them. Finally, we integrated the scripts handling the vision tasks into the application.

The list of questions we posed in order to fulfil the goal can be summarised as follows (in chronological order):
CHAPTER 1. INTRODUCTION

1. How does CGI scripting work, and which scripting language suits best to build the app?

2. How does one proceed at creating a Facebook app?

3. How does one develop an app aimed at various image operations such as downloading and tagging?

4. What are the restrictions of Facebook API in regards to our aim?

5. How does object detection and pattern recognition work, and which algorithm is best suited for our purposes?

6. How exactly can the computer vision procedures be integrated into the Facebook app in real-time?
Chapter 2

Facebook API

2.1 CGI Scripting

2.1.1 Introduction to CGI

CGI is a specification for Web Servers on how to interact with users. A CGI script—the program called by the web server through the CGI protocol—can be written in any language supporting stdin/stdout, however languages like Perl, PHP, Ruby and Python are most commonly encountered in practice. A typical script, stored and executed on a remote web server, usually follows a sequence of steps on the user’s request [7]:

1. Client sends CGI request to the server through a form or other means of information.

2. Server processes the request by running a suitable backend application.

3. The application sends the results back to the server

4. The server generates new HTML code and forwards it to the client
CHAPTER 2. FACEBOOK API

Nowadays nearly all web servers support CGI. The browser usually interprets URL as a path to the directory where CGI scripts are stored. See chapter 5 for details on how to run CGI scripts.

2.1.2 Choice of Platform

We tried Ruby 1.9.2 and Python 2.7 as a possible development platform. In the end we chose PHP 5.3.6, which proved to be a good choice for developing apps with Facebook. There is a vast amount of documentation available on the Internet, and Facebook provides an official PHP SDK which is relatively friendly to use.

Facebook provides an official SDK for other programming languages as well, for instance for Python [6].

2.2 Core Concepts of Facebook API

Facebook launched Facebook Platform in May 2007, a set of API allowing third-parties to integrate their apps or websites into Facebook. Two versions of Facebook API currently exist—Legacy REST API and Graph API.

2.2.1 Legacy REST API

The Facebook REST API is in the process of being deprecated and Facebook advises that developers use the new Graph API [8]. However, some methods used for interaction with canvas have not yet been upgraded and are therefore unavailable in the new Graph API. We didn’t use REST API in our project.

2.2.2 Graph API

The core of Facebook is the social graph of users, their connections - friends, events, pages - and their artefacts such as albums, photos and tags. Every ob-
ject in the graph has a unique ID [5]. Generally, each item can be accessed via https://graph.facebook.com/object_ID. All the basic information about user including their ID, name and profile picture are publicly available. It can be retrieved using https://graph.facebook.com/555555

If the user is logged in, the keyword me can replace the Facebook user ID in the URL. The query result is formatted as a JSON string. An example result for the fictive user 555555 is illustrated below.

```json
{
  "id": "555555",
  "name": "Jane Doe",
  "first_name": "Jane",
  "last_name": "Doe",
  "username": "janedoe",
  "locale": "en_GB"
}
```

More private information, e.g. photos or wall posts require permission, which is obtained through a so-called access token. Details on authorisation follow in the next session. It is possible to select specific fields and set a limit to the number of items returned. For example, the following query access the first two albums of the user with the (fictive) ID 555555:

```
https://graph.facebook.com/555555/albums?limit=2&access_token=...
```

### 2.2.3 OAuth2.0 Authentication Protocol

The idea of Open Authorization protocol goes back to sharing sensitive information with third parties located on another site without revealing one’s own credentials [9]. A special access token is granted instead, which allows getting hold of the specific data for a limited amount of time. For example, Entity A can give Entity B an access token to view their albums, but Entity B can’t use this token to access a list of A’s events.

OAuth authorisation normally requires three steps [4].

1. **User Authentication**: The system prompts for user’s credentials to ensure their identity.

2. **App Authorisation**: a dialog prompts a user to grant permissions to share their private information necessary for the app. Examples of such capabilities include access to photos and videos, permission to send e-mail or posting to user’s wall. Should the user decline sharing the information, the app is not authorised.
3. **App Authentication**: Had the user granted access in the previous step, the server generates the authorisation code needed to gain the access token so as to make API calls.

The diagram below demonstrates the sequence of HTTP requests exchanged when the local server makes a call to Graph API. This process is referred to as server-side flow.

![Server Side Flow Diagram](image)

**Figure 2.2: Server Side Flow**

Facebook also supports client-side flow, which is used when call to the Graph API is made from the client, e.g. from JavaScript running on user’s browser.

### 2.2.4 Permissions

In many cases the app needs a specific permission to perform certain operations even with a valid access token. The complete list of all permissions can be found on [12]. When the user loads the app for the first time, they will be asked to grant extended permissions to the application. Our code to get permissions is displayed in the listing below:

```javascript
function getPermissions($site){
    // Prompt for permissions for a logged-in user
    // Executes only on the first login
    $url=urlencode(
        'http://apps.facebook.com/telenatest/'.basename($site));
    $app_id='206455756037684';

    return $url;
}
```
As a side note, the user’s account privacy settings have no influence on the app’s functionality and on the authorisation process. However, if the user’s friends have very strict privacy settings, e.g. allowing only specific people accessing their photos, the app cannot access their information despite being in possession of an access token and permissions.

### 2.2.5 Tagging with Open Graph API

A tag is uploaded by issuing a HTTP POST request with the ID of the photo we add the tag to, the ID of the tagged user, coordinates of the tag as percentage to the photo width and height and the app ID [13]. Below is a code snippet demonstrating the URL string used for uploading:

```php
// Tag friend_to_tag at position (x_coordinate, y_coordinate)
$post_url = "https://graph.facebook.com/"
    . $photo_id . "/tags/" . $friend_to_tag
    . "/access_token=" . $access_token
    . "/x=" . $x_coordinate . "/y=" . $y_coordinate . "/method=POST";
```

### 2.3 Facebook Application

#### 2.3.1 Facebook Application Architecture

A Facebook application consists of two parts: a container application and a web application [2]. The former is called the Canvas and is defined on facebook.com while the latter is responsible for the application logic and behaviour and is hosted on an external web server. A prerequisite for creating an app is a Facebook SDK library, for example Facebook PHP SDK 3.0.
2.3.2 Creating Application

Canvas Page is created through the official Facebook Developers App by selecting "Create New App" option. The first thing one is required to do is entering a suitable name for your application.

The "About" tab contains mainly basic information about the app such as name, description, icon etc.

Once the Canvas Page is set up, the external application needs to be integrated into Facebook, which is done in the 'On Facebook' tab. Each application has a unique App ID and App Secret [1], which are automatically generated by Facebook. The Canvas Page needs to be given a unique name so that the application can be called from any location. The Canvas Name has the format http://apps.facebook.com/your_app_name/. Canvas URL points to the directory on the server where all CGI scripts, HTML and JavaScripts are located. In our case, Canvas URL is http://people.ee.ethz.ch/~telena/testapp/ and all files are stored in the testapp directory.

The app directory on the local server should contain Facebook SDK library (Note: even though it can be placed anywhere, the most common practice is to store it in the app's directory) and an index script. The provided example file example.php (e.g. available for download under 'examples' at PHP-SDK) is an excellent way to get familiar with Facebook API. The only thing that needs to be done is editing it with our App ID and App Secret. If we enter the URL to the Canvas Name in our browser, we will be forwarded to the application index page.
Chapter 3

Object Detection and Pattern Recognition

This chapter introduces algorithms we used for visual processing of the images. Given a photo, all facial images needed to be detected first. These images were stored as separate files to be used later for recognition. Moreover, the facial images were described, i.e. converted into vector representation, the so-called features. Once the pre-processing was done, recognition could be applied to the “faces”. We evaluated three recognition techniques, which are explained further in the text. Finally, the identities of the recognised persons needed to be labeled. Details on labelling follow in chapter 4.

Visual analysis was performed using OpenCV library 2.2 with Python. Details on how to install OpenCV can be found in chapter 5.

3.1 Face Detection

Since detection is a built-in functionality in OpenCV, we could just use the appropriate functions without writing our own. In the background, OpenCV employs Haar Cascade Classifier for face detection, a method that is based on the approach developed by Paul Viola and Michael Jones in 2001 [15]. The detection is done in real-time and combines several key concepts [3]:

Haar-like features: a set of light and dark adjacent rectangles. The value of the feature is the sum of pixels of light rectangles subtracted from the sum of pixels of the black rectangle. If that value is above some threshold, the feature is considered to be present, otherwise it is absent. The intuitive behind
CHAPTER 3. OBJECT DETECTION AND PATTERN RECOGNITION

Haar features are differences in lighting intensities in a human face. For example, the eye region is mostly darker than the region of the cheeks, so the darker rectangle would be placed over the eyes.

![Figure 3.1: Haar features used by Viola-Jones](image)

**Integral Image**: an image representation with which Haar features can be calculated in constant time. In this context integrating means summing up all pixel values above and to the left of the given pixel. Thus, a two-rectangle Haar feature needs 6 look-ups, and a three-rectangle feature requires 8 calculations.

**AdaBoost classifier**: a combination of differently weighted ”weak” classifier that creates a ”strong” classifier. It is formed as a degenerate tree or a chain of filters. If a rectangle region passes through all stages, it is classified as face. Rejection from one of the intermediate filters is enough to classify image subregion as Non-Face.

In OpenCV brightness distribution within the image is usually equalised with OpenCV cv.EqualizeHist() command prior to the detection. The detection is performed with the cv.HaarDetectObjects() command using the 'haarcascade_frontalface_alt.xml' cascade. There are numerous kinds of cascades, each best suited different purposes, e.g. frontal or profile detection. This thesis used exclusively frontal face cascade.

### 3.2 Face Recognition

For all recognition algorithms we had a reference set and an incoming set of pictures. The incoming set of pictures compassed the photos in the album that the user
had previously selected, where the identity of the people is mostly unknown. The reference set are the pictures containing tags of the user’s friends and of the user himself/herself, where we know the identity of the people. At the moment, the size of both incoming and reference set are about 4-5 people, since the app is still in the testing stage and not available to the public yet.

### 3.2.1 Simple Euclidean Distance

The first attempt at face recognition was made by computing a Euclidean distance of features obtained from Binary Robust Independent Elementary Features (BRIEF) face descriptor. BRIEF represents an image patch as binary strings calculated as simple intensity difference tests in real-time. BRIEF implementation code was kindly provided by the members of the Computer Vision lab, ETH Zurich.

All the programming was done in Python, among others with OpenCV library for detection and NumPy module for calculations. Recognition was performed according to the following steps:

1. Detecting faces in each image, storing them as separate .jpg files and applying BRIEF descriptor to them.

2. Coordinates of the face position and features were stored in a separate file.

3. For each input face the Euclidean distance was calculated between its feature and the feature of each face in the reference set. That distance was stored in a matrix, or in case of a single input face in a vector. The smallest distance was the best match.

The illustration below shows some of the results. The first column is the input image and the subsequent columns are matches in descending order. As one can observe, the success rate of this approach is about 20%. The calculation speed was unsatisfactory as well.
3.2.2 PCA

The Principal Component Analysis technique applied to face recognition realises the concept of reducing data dimension without significant loss of information. Face images are projected onto a feature subspace that most efficiently captures variances across the data. [14]

Each 2-D face can be represented as a 1-D vector by stacking its columns on top of each other. These vectors are normalised by subtracting the average face $\bar{I}$ defined by

$$\bar{I} = \frac{1}{N} \sum_{j=1}^{N} (I_i)$$

where $N$ is the number of images in the dataset.

In the next step we compute the covariance matrix

$$C = I \ast I^T$$

where $I = [I_1 I_2 I_3 \ldots I_N]$ is the matrix assembled from our face images.

Singular Value Decomposition (SVD) of the covariance matrix yields its eigenvalues and eigenvectors; the eigenvectors are stored in the $V_k$ matrix. According to
the SVD’s property of low rank approximation we consider only K largest eigenvectors associated with the K largest eigenvalues, which form the feature subspace. Each face in the data set can be represented as a linear combination of average face and weighted K eigenfaces: \( I_i = \bar{I} + \sum_{j=1}^{K} (V_i \ast w_i) \)

The weights form a vector \( w' = [w_1, w_2, ..., w_K] \) that captures the contribution of each eigenface to the given original image. This vector is used in pattern recognition by computing the Euclidian distance between the incoming face and compressed faces in the data set.

We implemented PCA using SVD in MATLAB. A sample output of results is illustrated in the figure below. All images from our dataset were resized to 50x50 format; the basis size of 40 eigenvectors was chosen out of 320 images with trial-and-error approach.

![PCA results](image)

Figure 3.3: PCA results

In our experiments PCA performed even worse than simple Euclidean distance. The success performance is about 10-20% in the first top five matches.

### 3.2.3 Recognition Using Face Alignment by Robust Nonrigid Mapping

The last approach we chose is based on face alignment: finding a representation between two images so that they can be optimally matched regardless to different
poses and expressions. [16] Face alignment positions the object into a canonical pose so that relative differences of significant features can be evaluated with respect to a fixed coordinate system. A mesh is placed onto the facial image which is then deformed in such a way that vertex coordinates of selected points, e.g. the corner of the right eye, are uniform across the number of images. Recognition is accomplished by calculating the Euclidean distance of the features that describe the aligned images.

The figures below illustrate this algorithm’s performance. Owing to alignment, the facial images are larger than in the previous approaches. The figure on the left also contains an example where face detection returned a false positive. As in previous figures in this document, the first column represents input images and the remaining columns are the matches in descending order.

The algorithm performs better than the previous approaches. The success performance is about 40% in the first top five matches. It is very probable that the performance shall improve even further when more data (i.e. reference facial images for more people) is available. As of now, there are only a few reference objects in the database since it is yet unavailable for a broader audience. A minor open issue remains, however: many Facebook users block photos they have been tagged in. The app can’t access them despite a valid access token.
Chapter 4

Computer Vision applied to Facebook

4.1 Application Technicalities

The welcoming screen displays thumbnails of the cover photos of the user’s albums. There are two tagging options: auto-tagging mode where all algorithm’s suggestions are uploaded straight to Facebook and manual mode where the user can click through the photos, correct identities of the friends to tag and upload only selected tags. A further functionality of the app is displaying the list of the user’s closest friends on the basis of tagging frequency.

4.1.1 Structure of the Application

The directory where the all files that make up the app consists of the following subdirectories:

facebook-sdk : contains official Facebook SDK to make API calls

images : this directory contains a folder where images of each app user are stored. The name of the user’s photos directory is the same as their Facebook user ID. The user’s directory is subdivided into separate directories where photos from appropriate album are stored. The album’s directory name and photo file names also match their Facebook ID. There is also a subdirectory tags where all photos containing tags of the user and their friends are stored. Each Facebook user has their own directory. Tags holds tagged photos of all possible app users and friends in case different app users have an overlapping set of friends. This way common friends’ tags are downloaded once only.
style: this directory hosts various images used for CSS and HTML styling.

logs: this directory contains error logs for the Python scripts and the PHP script which downloads the tags.

deprecated: the contents of this folder are the Python scripts which implement PCA, a discarded approach for recognition.

users: this directory contains faces_AppUserID.xml for each app user. This XML file includes identity of recognised people on the photos, coordinates of the suggested tags, whether a given photo has already got a tag, and some system internal information needed for tagging.

The figure below shows the hierarchy.

![Figure 4.1: Hierarchy](image)

4.1.2 Pipeline

The system works as follows:

1. index.php displays thumbnails of the cover photos of the albums the user has on Facebook. When the user loads the application for the first time, the script will create a directory for each album stored online. On all subsequent calls the album directories are synchronised with Facebook: the albums the user had deleted online are pruned and a directory for any new albums currently not on the disk are created. The index script does no actual downloading yet, it merely creates directories for the albums. The photos are downloaded in the later scripts.

18
2. First course of action: the user clicks on the thumbnail of one of their albums to start tagging:

(a) The user will be forwarded to the script `intermediate.php`, which executes all the necessary scripts for the preparation in background. An intermediate script is necessary because of regular server timeouts when a large amount of data is encountered (e.g. a user which has 2000 friends all of which possess an impressive amount of tags). During the execution time of the background scripts `intermediate.php` runs a javascript function which polls for the result of the working scripts.

(b) `getfriendtags.php` is the first script to be executed by the intermediate script, which downloads all photos from the album the user has selected as well as the photos where the user and their friends have been tagged in and creates an XML file `database_AppUserID.xml` with information about the path to the photos, Facebook coordinates of the tag and identity of the user tagged. Should a photo already be deleted from Facebook, the script clears the appropriate images from the local disk. If the photo is already on the disk, it won’t be downloaded again.

(c) `getfriendtags.py` extracts a facial image of the tagged user and aligns it. The script also edits `database_AppUserID.xml` with path information on the features of the aligned image.

(d) `recognise.py` extracts and aligns facial images from the album the user has selected for tagging. Each face from the album is compared with each face in the database, a set of tags acquired in the previous step. After finding the best matching face, the scripts writes the path to the original image from the album, coordinates of the suggested tag, the identity of the recognised face which is read from `database_AppUserID.xml` and other information necessary for tagging into `faces_AppUserID.xml`.

(e) All background scripts have finished their work by this time, so the user is forwarded to `results.php`, which displays the photos from the album, provides the previously calculated tag suggestions and supplies the user to either accept and upload the tag, or reject it. In case the identity of a person in the photo was guessed erroneously, which is probable to occur if there are no entries for this person in the database, the user has the possibility to enter the correct name by clicking on the photo and typing the name into the textbox.

3. The second course of action: the user wants to auto-tag all the pictures in the selected album. In this case the steps 1–4 are the same as in the previous
CHAPTER 4. COMPUTER VISION APPLIED TO FACEBOOK

option, and step 5 is omitted.\(^1\) Tags have to be uploaded once at a time, since Facebook API doesn’t allow posting multiple tags in one request. Auto-tagging process is split into two scripts: auto_tag_script.php does the actual tags’ uploading whereas auto_tag.php has the similar function as intermediate.php, namely preventing timeout.

4. The third course of action: the user wants to analyse the closeness with his friends. In this case they are forwarded to friendranks.php upon following the hyperlink with the appropriate option. See Figure 4.2 for sample results.

![Post to your Wall](image)

Figure 4.2: Friendship Ranking Results

The pipeline is summarised in the illustration below.

\(^{1}\)Caution! The current version of the app automatically uploads only the ID of Anakin Sparrow, the fake profile that was created to test and develop the app. The correct line with the correct user ID is commented out in the code.
4.1.3 Evaluation

As of this moment, the app doesn’t show the best speed performance. We measured elapsed wall clock time to determine performance bottleneck. All measurements were done per unit, meaning that we appraised e.g. how long it took to download or process a single photo instead of looking how long it took until all photos were downloaded. The reasoning behind it is that the app’s speed depends very much on the number of friends the user possesses, or on the amount of photos in one’s photo album. For instance, if a user A has 100 friends each of them having 100 photos, the program takes longer to calculate his results than for user B who has only 8 friends with few photos. The total time elapsed isn’t a measurement precise enough for our purposes.

The table below illustrates some of the key results. All measurements are given in seconds.
CHAPTER 4. COMPUTER VISION APPLIED TO FACEBOOK

<table>
<thead>
<tr>
<th>Operation</th>
<th>Average time in seconds</th>
</tr>
</thead>
<tbody>
<tr>
<td>Downloading a single photo</td>
<td>0.4</td>
</tr>
<tr>
<td>Disk access</td>
<td>10</td>
</tr>
<tr>
<td>Querying Facebook API</td>
<td>0.7</td>
</tr>
<tr>
<td>XML operations</td>
<td>0.0001</td>
</tr>
<tr>
<td>Detection (single face)</td>
<td>0.8</td>
</tr>
<tr>
<td>Aligning (single face)</td>
<td>2</td>
</tr>
<tr>
<td>Norms calculation</td>
<td>0.3</td>
</tr>
<tr>
<td>Sorting the distances vector, determining identity and writing it to XML</td>
<td>0.0001</td>
</tr>
</tbody>
</table>

Table 4.1: Performance Evaluation Results

As one can observe, the bottlenecks are local disk access, which is valuable to synchronise the contents of the local disk with Facebook, and OpenCV operations such as alignment and detection. Python scripts responsible for visual image processing are quite slow because they have to process a solid amount of photos where a single photo might contain several faces.

4.2 Porting the App to the ETH Server

Since the ultimate goal of this thesis was a running application on the ETH server, the last step of the project involved porting the system developed at the local computer to the server at the BIWI lab. In the time period immediate to the porting, the app used the PCA recognition algorithm. So, the code had to be slightly adapted to fit the new algorithm, recognition on results of the facial alignment. We used Python bindings for the library written in C++ that was kindly provided by the BIWI lab members.

A number of problems occurred during the porting phase. Firstly, sample libraries residing at the `/scratch.net/` root directory are invisible for the web server, so they had to be copied to the home directory. Python and OpenCV had to be re-installed at the home directory so that all the dependencies would remain preserved. The corresponding script `python-install.sh` is available. As disk quota was exceeded during the transmission progress, the available disk space had to be increased to 1GB by the system administrator. Secondly, the server has a
limited maximum execution time of 30 seconds, which is hardly enough for processing a large amount of images.

The chosen solution for the timeout problem comprises introducing an extra PHP script to execute all the background scripts. The new script uses a simple javascript that periodically checks the file where the background scripts write their execution status. Once javascript notes the completion of a background script, it notifies the PHP script and it triggers execution of the next background script in the pipeline.

Thirdly, once all the necessary libraries were re-installed and recompiled, the Python scripts still wouldn’t run on the server due to the fact environment variables could be set only temporarily. The problem was rectified by writing a shell script that sets the environment variables and executes the program. The listing below prints the scripts and demonstrates how it is being used.

```bash
#!/bin/bash
export LD_LIBRARY_PATH=/home/telena/opt/lib:
    /home/telena/opt/lib/python2.7
/home/telena/opt/bin/python $@

> ./exec_python.sh getfriendtags.py <AppUserID>
> ./exec_python.sh recognise.py <AppUserID> <AlbumID>
```

Lastly, we received an automatically generated warning from the ProcGuard about the extensive CPU last caused by one of the Python scripts. The script used up to 95% of the computing power. We were suggested to run the job using renice 19 or Condor. Nothing has been ventured yet in regards of using Condor. As a temporary solution we limited the number of friends to ten people. The limit shall be removed as soon as we undertake the measures to enable our jobs running without taking too much power.
Chapter 5

Set Up

5.1 Running CGI Scripts with Apache

If someone has never done any CGI scripting before, it makes sense to get accustomed to it from your local computer first. A web server like Apache is necessary in order to run the scripts on localhost. On Linux systems it is included in the repository. For example, to install Apache on Ubuntu one needs to type `sudo apt-get install apache` into the shell.

Since Apache’s default settings prohibit executing CGI scripts, one has to edit one or more configuration files. In our case it was the file `default.conf` located at `/etc/apache2/sites-enabled`. The following block of code was appended inside the `<VirtualHost>` tags:

```
Alias /cgi-bin/ /home/elenawww/cgi-bin/
<Directory /home/elenawww/cgi-bin>
   AllowOverride None
   Options +ExecCGI -MultiViews +SymLinksIfOwnerMatch
   Order allow,deny
   Allow from all
   AddHandler cgi-script cgi python php
</Directory>
```

A widely seen convention is to place all executables into a directory called `cgi-bin` or `public_html`. A valid CGI script needs to be executable. The permissions can be set on Ubuntu via `chmod a+x myscript.cgi`. The listing below shows the simplest CGI script in Python.

```
#!/home/elenapp/python-2.7A/bin/python
import cgi
print "Content-type: text/html\n\n"
print "<html><body>This is a test </body></html>"
```
CHAPTER 5. SET UP

The first line is the path to the scripting language on your computer, in our case it is the path to Python. The second line tells Python that the program is supposed to run on the web. The next line is a content-type header, which needs to be the first line any CGI script outputs. The header tells the remote user what kind of file to expect, here we are dealing with an HTML output. The final line is an arbitrary output, so that we can see whether the script “is doing something”.

Assuming that the script is stored in the cgi-bin directory, we can open the browser and type http://localhost/cgi-bin/test.cgi into the address field. The output This is a test can now be seen.

5.2 OpenCV 2.2

OpenCV is an open-source library for computer vision available for download at http://opencv.willowgarage.com/wiki/. It was originally written in C++, but it also contains C and Python bindings. We decided to use Python as it is has a better ease of handling than the C/C++ alternative. The latest stable release of OpenCV is 2.2.

According to the official installation guide [11], common prerequisites for installing OpenCV with Python bindings on a Linux machine include a C/C++ compiler, CMake (version 2.6 or later, we used version 2.8.3), ‘build-essential’ package, pkg-config, libgtk2.0-dev and supplementary package ‘python-dev’ that contains headers. The crucial step is running cmake with option ‘BUILD_PYTHON_SUPPORT=ON’.

One of the main difficulties that occurred during the installation was a minor incompatibility of the library and kernel 2.6.38 that comes with Ubuntu 11.04. The problem emerges in the highgui module, because some functionalities from there are not provided by the kernel anymore. As referenced by [10], one has to adapt three files to compile OpenCV 2.2. on Ubuntu 11.04:
modules/highgui/src/cap.cpp and
modules/highgui/src/cap_libv4l.cpp.
Chapter 6

Future Work and Conclusions

Even though the thesis is completed and the app is fully functional, a number of improvements could be integrated into the system:

- Whereas multiple users can use the app at the same time, a single user cannot use multiple instances of the app, e.g. in two tabs or in different browsers. The XML files are named with the user’s Facebook ID, and can therefore not be simultaneously updated by two processes.

- Recognition results for tagging, e.g. coordinates, paths to diverse files are currently stored in XML files. This approach was chosen for the sake of simplicity and also due to the time constraints. However, an SQL database would be a more elegant solution.

- Python scripts that handle image processing are quite slow, mostly due to the fact that alignment is expensive. Scaling images would be one possibility for a speed-up.

- Visual UI could be improved in some places, e.g. better formatting in Facebook’s native Feed Dialogs.

- Closeness analysis could be taken a step further: the current version of the app evaluates relationship between objects by counting the amount of tags on their pictures. A more advanced possibility would be real-time face recognition. This approach wasn’t progressed due to timely constraints and to the still error-prone face recognition.

Two versions of the application are ready to hand-in: the first one, TestApp has been created using the Facebook PHP SDK 2.2. The second one, FaceTagger, is identical to TestApp except for the fact that it uses the new Facebook PHP SDK.
3.0. All files are stored in the `testapp` and `facetagger` directories respectively. For both applications, assessment of the achieved results with regards to the original aim can be summarised as follows:

- The app is able to perform a simple visual analysis of the user’s photos, i.e. detecting and recognising facial images in pictures.
- The app enables successful uploading of one tag at a time as well as multiple tags at user’s request.
- The app functions correctly when multiple users access it simultaneously.
- The app maintains an efficient usage of the storage space by deleting unnecessary items, i.e. data deleted by user on Facebook is pruned on the disk.
- The app is stable against timeouts.
- Basic knowledge into recognition and detection algorithms was acquired.
- Exposure to Facebook API
- A solid basis in CGI scripting was obtained.
Bibliography


