A Software Framework for PCA-based Face Recognition

by

Peng Peng

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Mathematics
in
Computer Science

Waterloo, Ontario, Canada, 2016

AUTHOR'S DECLARATION

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Abstract

Face recognition, as one of the major biometrics identification methods, has been applied in different fields involving economics, military, e-commerce, and security. Its touchless identification process and non-compulsory rule to users are irreplacable by other approaches, such as iris recognition or fingerprint recognition. Among all face recognition techniques, principal component anaylsis (PCA) was proposed in the earliest stage; however, it is still attracting researchers in this field because of its property of reducing data dimensionality without losing important information.

PCA-based face recognition has been studied for decades. There exist some image processing toolkits like OpenCV, which have implemented the PCA algorithm and associated methods. Nevertheless, establishing a PCA-based face recognition system is still time-consuming, since there are different problems that need to be considered in practical applications, such as illumination, facial expression, or shooting angle, which can hardly be solved by the toolkits. Furthermore, it still costs a lot of effort for software developers to integrate the implementations of the toolkits with their own applications.

Therefore, the thesis provides a software framework for PCA-based face recognition aimed at assisting software developers to customize their applications efficiently. The framework describes the complete process of PCA-based face recognition, and in each step, multiple variations are offered for different requirements. Through various combination of these variations, at least 108 variations can be produced by the framework. Moreover, some of the variations in the same step can work collaboratively and some steps can be omitted in specific situations; thus, the total number of variations exceeds 150. The implementation of all approaches presented in the framework is provided.

Acknowledgements

I would like to thank Professor Paulo Alencar, for his kindness, understanding and guidance. It has been my honor to be his student and I learned a lot while working with him. I'm very grateful to Professor Daniel Berry for serving as my co-supervisor. Thanks to Professor Donald Cowan for helping me revise my paper and thesis. I also thank Professor Donald Cowan and Professor Ladan Tahvildari for agreeing to read my thesis and providing me valuable feedback.

Immense gratitude towards my parents Zhenyun and Hongwei for their unconditional love.

Special thanks to my girlfriend Amy for her constant care and love.

Finally, thanks to my friends Akshat Kumar and Vishnu Srivastava for sharing their wisdom.

Dedication

Dedicated to my uncle Xiaohua Peng

Table of Contents

List of Figures

Chapter 1

Introduction

Face recognition has been studied for decades and has been applied in various areas. In 2012, Samsung released their new smart TV which incorporates face recognition by using built-in camera. This new feature allows users to log in to social network applications, such as Facebook, Twitter, or Skype without resorting to a userid and password. In addition, face recognition has also attracted the attention of governments because of its high security level and accessibility. DARPA (Defense Advanced Research Projects Agency) intends to supplant traditional digital passwords by scanning human faces [1]. Face recognition techniques even can support law enforcement. Karl Ricanek Jr. introduced an application of face recognition in detecting potential child pornography in computer storage. The results show significant progress in both detection speed and accuracy [2].

Recognizing human faces with computers originates from research in 1964, by Helen Chan and Charles Bisson [3]. Initially, most research focused on detecting individual features, such as eyes, nose, and mouth. As mathematical approaches developed, researchers focused their attention on describing the entire face with statistical methods, which led to further advances in face recognition. Currently, face recognition methods can be classified into categories such as feature-based recognition, appearance-based recognition, template-based recognition, etc. However, principal component analysis (PCA), as proposed by Alex P. Pentland in 1991 [4] is still one of the most popular analysis techniques. A number of variations based on the standard PCA approach have been developed for different situations. The PCA property of reducing data dimensionality without losing principal components is the key feature that makes it an object of continuing study.

Although PCA is a mature technique, it is still time-consuming to implement the algorithm, especially when adapting it to different types of data, or combining it with pre-processing and result generation steps. Some popular image processing toolkits, such as OpenCV, have standard PCA algorithms, but these libraries are not designed to help users customize applications. Furthermore, the multiple variations of PCA also need to be considered, since they produce better results when being used under extreme situations, such as non-uniform illumination, or exaggerated facial expressions. Additionally, associated steps such as face detection and pre-processing also play an important role in terms of the entire face recognition process. Selecting appropriate approaches in each step according to specific situations positively affects the final recognition accuracy.

This thesis intends to propose a software framework for PCA-based face recognition aiming at assisting software developers to customize their own applications efficiently. The framework describes the complete process of PCA-based face recognition, and in each step, multiple variations are offered for different requirements. Through different combinations of these variations, at least 108 variations can be produced by the framework. Moreover, some of the variations in the same step can work collaboratively and some steps can be omitted in specific situations; thus, the total of variations exceeds 150. The implementation of all approaches in the framework is provided.

With the framework, software developers working on face recognition applications are able to build their applications quickly through software reuse, as the task becomes a design process at a higher level. After clarifying the requirements of the applications, the framework helps developers to select appropriate variations for each step in the face recognition system. As the framework describes the entire PCA-based face recognition process and demonstrates what type of situations are dealt by the variations, developers simply choose a variation for each step according to the guide of the framework and then build their application.

As an example, if the developer intends to build a face recognition application used for security which works on a high performance computer, the framework will prioritise the recognition accuracy, whereas the responding speed becomes to a minor factor, since the high performance computer is able to provide enough computation resources. However, when the face recognition is used for smart phones, providing real-time feedback to users is more important, and some extreme environmental conditions such as non-uniform illumination need to be considered. Thus, the framework provides variations which generate results fast and can deal with different working environments.

The thesis presents four case studies based on the variations produced by the framework. The first case study is a face recognition system for smart phones. The other three case studies aim to cover all variations to give a comprehensive impression of the framework to readers. For instance, the Case Study 2 describes a face recognition application working on high performance computers. However, the possible applications which can be produced by the framework are not limited to the case studies.

1.1 Related Areas

1.1.1 Face Recognition System

Currently, personal identification still heavily relies on traditional password encryption. This method do help people protect their privacy; however, with the development of other high-tech fields, the security level provided by a password is not able to meet our requirements, as it is based on “what the person possesses” and “what the person remembers”, instead of “who the person is”. Fortunately, a new research

area, biometric recognition, offers a number of technical methods, which may make truly reliable personal identification come true.

In the field of biometrics recognition, face recognition is the friendliest, most direct and natural method. Compared with other recognition approaches, such as fingerprint recognition or iris recognition, face recognition does not invade personal privacy or disturb people. Additionally, a face image is easier to capture, even without making the person aware that an image is being made.

According to a report from The 3rd China Guangzhou International Biometric Identification Technology Expo in 2016, the total revenue of global biometric identification market reached 9.368 billon USD in 2014. Face recognition had a market share of 11.4%, the largest percentage of revenue among all the identification approaches [19].

Generally, an automatic face recognition system is divided into phases, face detection and face recognition. In the face detection stage, the face area is extracted from the background image, and the size of the area is also defined at the same time. In the face recognition stage, the face image will be represented with mathematical approaches to express as much information about the face as possible. Eventually, the new face image will be compared with known face images, which results in a similarity score for final verification.

Thanks to a human being's eyes, the aforementioned two phases can be easily completed. However, building an automatic face recognition system with high accuracy is challenging, as every phase in the recognition process is susceptible to internal physiological and external environmental factors. Therefore, face recognition is still attracting researchers.

1.1.2 Principal Component Analysis

The earliest principal component analysis dates back to 1901 when Karl Pearson proposed the concept and applied it to non-random variables [6]. In 1930, Harold Hotelling extended it to random variables [17] [18]. The technique is now being applied in a number of fields, such as mechanics, economics, medicine, and neuroscience. In computer science, PCA is utilized as a data dimensionality reduction tool. Especially in the age of Big Data, the data we process is often complex and huge. So reducing the computational complexity and saving computing resources are important issues.

Basically, the PCA process projects the original data with high dimensionality to a lower dimensionality subspace through a linear transformation. Nevertheless, the projection is not arbitrary. It has to obey a rule that the most representative data needs to be retained, i.e. the data after transforming cannot be distorted. Hence, those dimensionalities which are reduced by PCA are actually redundant or even noisy. Therefore, the ultimate goal of conducting PCA is to refine data so that the noisy and

redundant part can be removed and only the useful part is retained. It is because of this feature that PCA is widely used in face recognition. Images are represented as a high dimensional matrix in computers, and removing noise from images is a necessary pre-processing step.

1.1.3 Object-oriented Framework

An object-oriented framework is a group of correlated classes for a specific domain of software. It defines the architecture of a class of user applications, the separation of object and class, the functionality of each part, how the object and class collaborate, and the controlling process. Therefore, one focus of an object-oriented framework is software reusability [58].

Software reuse uses existing knowledge of a software to build a new software, so that to reduce the cost of development and maintainence. In 1992, Charles W. Krueger suggested five dimensions for a good software reuse, which are abstraction and classification in terms of building for software reuse process, and selection, specialization and integration in terms of building with software reuse process. Abstraction and classification means that in software reuse, the reusable knowledge should be represented concisely and classified. Selection, specialization and integration indicate that reusable knowledge should be parameterized for query, specialized for new situations, and integrated for customer projects [59].

A framework can be viewed as the combination of abstract class and concrete class. The abstract class is defined in the framework, whereas the concrete class is implemented in the application. Simply, a framework is the outline of an application, which contains the common objects for a specific domain. In addition, a framework includes some design parameters, which can be used as interfaces, to be applied to different applications.

1.1.4 Machine Learning Approaches

Machine learning is an interdisciplinary subject consisting of many different areas, such as probability, statistics, approximation theory, and algorithm theory. Arthur Samuel first defined machine learning as a "Field of study that gives computers the ability to learn without being explicitly programmed" [3].

Machine learning focuses on simulating human beings' behaviors to gain new knowledge and skills with a computer. Furthermore, it is able to recombine the learned knowledge and keep improving its performance.

Typically, machine learning is classified into three categories, which are supervised learning, unsupervised learning, and reinforcement learning [20]. The difference mainly depends on whether the computer is taught or not. In supervised learning, the computer is given input along with its corresponding output. However, in unsupervised learning, no labels are provided, so the computer needs

to learn on its own. Unsupervised learning does not always have an explicit goal, which means that it is allowed to find a goal by itself. Reinforcement learning can be treated as a compromise between the two aforementioned approaches. It has an explicit goal, but it needs to interact in a dynamic environment in which no teaching is provided.

1.2 Problem

Although there exists a number of image processing toolkits like OpenCV, which have PCA algorithm as well as associated approaches for face recognition, it is still time-consuming for software developers who intend to integrate face recognition implementations with their own applications. Furthermore, selecting appropriate approaches for each step in the process of face recognition is non-trivial, since it directly impacts the final recognition result. For face recognition systems which run under extreme situations, such as non-uniform illumination, exaggerated facial expression, or facial region occlusion, approach selection becomes even more significant. In fact, building a PCA-based face recognition system should not cost a lot of effort for developers, as the technique has been studied for years and is mature. The time spent on implementing the algorithms and integrating with their applications should not be necessary.

1.3 Proposed Approach

This thesis provides a software framework for PCA-based face recognition aiming at assisting software developers to customize their own applications efficiently. The framework describes the complete process of PCA-based face recognition including image representation, face detection, feature detection, pre-processing, PCA, and verification, and in each step, multiple variations are offered to fit different requirements. Through various combinations of these variations, at least 108 variations can be generated by the framework. Moreover, some of the variations in the same step can work collaboratively and some steps can be omitted in specific situations; thus, the total number of variations exceeds 150. The implementation of all approaches presented in the framework is provided. As the framework strictly follows the normal process of PCA-based face recognition, it can be easily extended, which means more approaches are able to be attached to any of the steps.

1.4 Evaluations

In the thesis we present a framework followed by four case studies. The first case study is for face recognition using on smart phones. The other three case studies cover almost every variation supported by the framework.

1.5 Contributions

The main contributions described in this thesis are:

1. A model which includes the entire facial recognition process using PCA with multiple variations in each phase for different facial conditions;
2. A high-level framework design;
3. Implementation of the framework; and
4. A support tool for facial recognition with PCA

1.6 Thesis Outline

Chapter 2 presents work related to the research which mainly includes four sections. The first section introduces general requirements of face recognition system. The second section focuses on principal component analysis which is the core algorithm of this research. The third section explains the concept of object-oriented frameworks. The last section talks about machine learning. Chapter 3 demonstrates the framework. Chapter 4 describes the case studies based on the proposed framework. Chapter 5 concludes the thesis and suggests future work.

Chapter 2

Related Work

As mentioned in Chapter 1, this chapter introduces work in four areas related to our research. First, a classical face recognition framework is demonstrated. Then, we present a brief introduction to principal component anaylsis (PCA) describing the history of the approach, the mathematical principal behind PCA, and its development in face recognition. Third, object-oriented frameworks are discussed. Last, we investigate machine learning approaches, since its outstanding classifying ability has been attracting researchers in face recognition.

2.1 Face Recognition Framework

Generally, a face recognition framework is divided into two sequential processes, which are face detection and face recognition. As introduced in the previous chapter, face detection focuses on capturing the face region from the image. Then the face region is delivered to a face recognition process for verification. The structure of this process is shown in Figure 1.

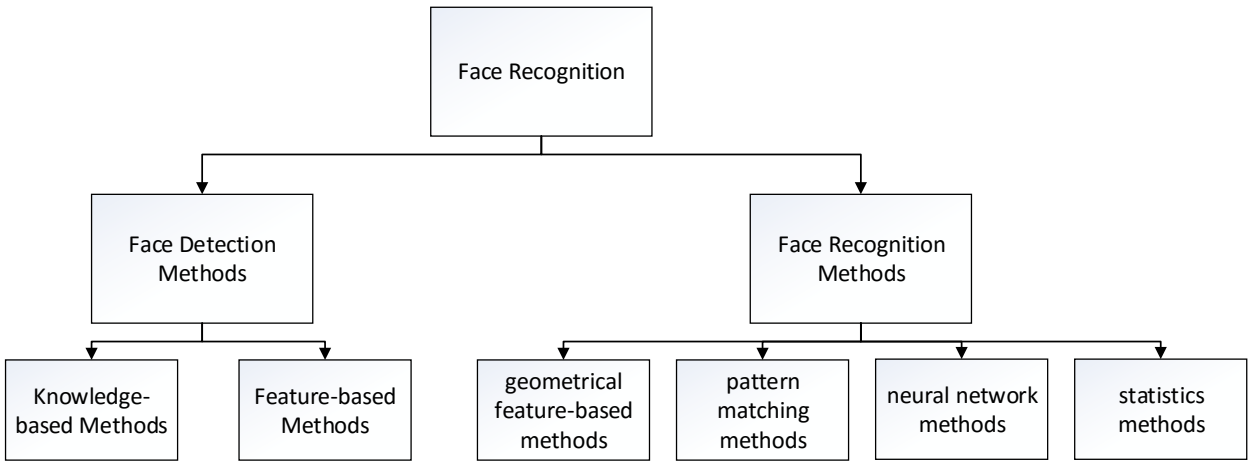

Figure 1 Face recognition system

2.1.1 Face detection

Face detection is a necessary step in face recognition systems, which localize and extract the face region from the background. Basically, face detection can be classified into two categories, which are knowledge-based methods, and feature-based methods [47].

Knowledge-based methods are actually based on a series of rules generated from researchers' prior knowledge of human faces, such as the face color distribution, distance or angular relationship between eyes, nose, and mouth. Most of these rules are straightforward and easy to find.

Yang and Huang [21] proposed a layered knowledge-based face detection method in 1994. Their system is consisted of three different levels. At the highest level, the face candidates are found by scanning the input image windows and applying the rule set for each component on face. At the second highest level, the rule set is used for describing what a human face looks like. More prior knowledge is added at this level. The lowest level depends on detailed facial features. The process actually refines the detection process step by step. The reason is that higher step is able to eliminate those images which are not face images, so that the speed of following steps is improved.

Feature-based methods detect face region based on internal facial features as well as the geometrical relationship among them [50]. Contrary to knowledge-based methods, feature-based methods seek constant features as a means of detection. Researchers have proposed a number of methods, which detect face features first, then deduce whether this a real face. Facial features, such as eyebrow, eyes, nose, mouth, and hairline are usually extracted with an edge detector. According to the extracted features, statistical models describing the relationship between each features can be built, so that the face region can be captured. However, feature-based methods are always susceptible to illumination, noise, and occlusion, as these factors seriously damage edges on face [51].

2.1.2 Face Recognition

Face recognition methods can be classified into three categories, which are early geometrical feature-based methods and pattern matching methods, neural network methods, and statistical methods.

The earliest face recognition was based on geometrical features of a face. Simply, the basic idea of this kind of method is to capture the relative position and relative size of representative facial components, such as eyebrows, eyes, nose, and mouth [52]. Then face contour information is included to classify and recognize the faces. Pattern matching methods are the simplest classification methods in the field of pattern recognition. In face recognition, face images in a dataset are treated as the pattern, so once a new image is available, a correlation score between the pattern and the new image can be calculated to generate the final result.

Artificial neural network research dates to the 1940s when Warren McCulloch and Walter Pitts [22] first applied the concept to mathematics and algorithms. The idea of artificial neural networks is inspired by biological neural networks, which consist of a large number of neurons. The neurons in artificial neural networks are actually a group of individual functions, each of which is responsible for a certain

task. The neurons are connected with weighted lines which pre-process the input generated from the previous neuron. The advantages of applying neural network to face recognition are its ability to store distributed data that can be processed in parallel.

The structure of a single neuron is simple with limited functionality; however, an entire neural network consisting of a number of neurons is able to achieve various complicated goals. Furthermore, the most significant feature that neural network possesses is self-adaptability, which means it is able to enhance itself through iteration. The most representative neural network methods in face recognition are multi-level BP networks and RBF networks [53] [54].

Statistics-based methods attract attention from researchers in face recognition. The idea of a statistics-based method is to capture statistical feature of a face through learning, and then use the acquired knowledge to classify the face. The learn and classification process is shown in Figure 2.

Among all statistics-based methods, subspace analysis is the major type. The basic idea is to compress the face image from a high dimensional space to on with lower dimensions through a linear or non-linear transformation. These methods include Linear Discriminant Analysis (LDA), Independent Component Analysis (ICA), and Principal Component Analysis (PCA).

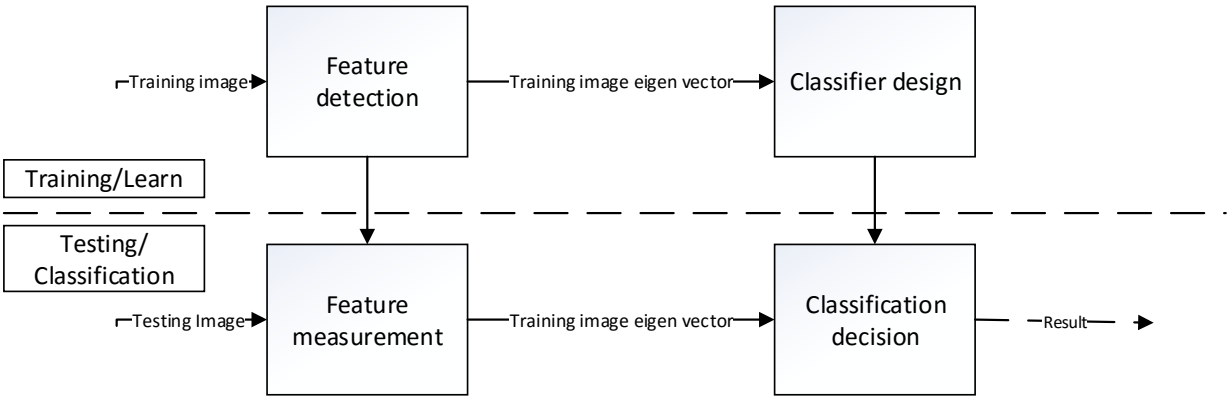

Figure 2 Learning and classification process

2.2 Principal Component Analysis

In computer science, particularly in the context of Big Data, data is often expressed as vectors and matrices. In terms of images, the increase of the resolution of an image means the size of the matrix is larger. Although current computers are powerful enough to process huge amount of data in relatively short time, efficiency still needs to be considered.

Principal component analysis has been widely recognized as an efficient data dimensionality reduction method using a linear transformation [16]. While reducing the data dimensionality, retaining significant information is the basic requirement.

In statistics, mean value, standard deviation, and variance are always used to analyze the distribution and variation of a set of data. These three values can be calculated with Equation (1), (2), (3).

$$\overline{x} = \frac{\sum_{i=1}^{n} X_i}{n} \tag{1}$$

$$s = \sqrt{\frac{\sum_{i=1}^{n} (X_i - \bar{X})^2}{n-1}} \tag{2}$$

$$s^2 = \frac{\sum_{i=1}^{n} (X_i - \bar{X})^2}{n-1} \tag{3}$$

However, mean value, standard deviation, and variance functions only work for one-dimensional data. In computer science, the data is always multi-dimensional. So a new measurement which conveys a relationship among data of different dimension needs to be included, which is covariance. Normally, covariance is able to describe the relationship between two random variables, as shown in Equation (4).

$$cov(X, Y) = \frac{\sum_{i=1}^{n} (X_i - \bar{X})(Y_i - \bar{Y})}{n-1} \tag{4}$$

Therefore, as the dimension increases, multiple covariance needs to be calculated, e.g. the number of covariance needed when dealing with n-dimensional data is shown in Equation (5).

$$\frac{n!}{(n-2)! \times 2} \tag{5}$$

Fortunately, a matrix approach offers a perfect solution for this calculation. The Equation (6) shows the definition of a covariance matrix.

$$C_{n \times n} = (c_{i,j}, c_{i,j} = cov(Dim_i, Dim_j)) \tag{6}$$

Equation (7) shows the covariance matrix of a dataset with three dimensions *{x, y, z}*.

$$C = \begin{pmatrix} cov(x,x) & cov(x,y) & cov(x,z) \\ cov(y,x) & cov(y,y) & cov(y,z) \\ cov(z,x) & cov(z,y) & cov(z,z) \end{pmatrix} \tag{7}$$

It can be found that covariance matrix is a symmetric matrix, whose diagonal shows the variance of each dimensions.

After generating the covariance matrix, we are able to calculate its eigenvalues and eigenvectors through Equation (8).

$$A\alpha = \lambda\alpha \tag{8}$$

Where A stands for the original matrix, λ stands for an eigenvalue of A, and α represents the eigenvector according to eigenvalue λ.

Usually eigenvalues are sorted in descending order, which corresponds to the importance of the eigenvector. We can choose how much information to retain. In this case, selecting a good threshold with which useful information is retained, whereas less significant information is removed, becomes important.

2.3 Object-oriented Frameworks

In recent years, software reuse has become a significant technique in software engineering. Traditional methods, such as function or library, provide limited reuse, whereas object-oriented frameworks aim at larger components, such as business units and application domains. Building object-oriented framework can save users countless hours and thousands of dollars in development costs by providing reusable skeletons [55]. Object-oriented framework development plays an increasingly necessary role in contemporary software development. Frameworks like MacApp, ET++, Interviews, ACE, Microsoft's MFC and DCOM, JavaSoft's RMI and implementations of OMG's CORBA are widely used [23].

Some of the features of object-oriented framework are listed below:

A. Modularity

   Framework enhances software modularity by encapsulating variable implementation details into fixed interfaces. The impact caused by variations of design and implementation is localized by a framework, so that makes software maintenance much easier.

B. Reusability

   Framework improves software reusability, as the interfaces provided by a framework are defined as class attributes which can be applied to build new applications. Actually, the reuse of framework takes advantage of the expertise and effort of experienced software developer to minimize the time spent by subsequent developers on the same problem in the domain. Framework reuse not only improves software productivity, but also enhances the reliability and stability of software.

C. Extendibility

   Some frameworks provide hook methods allowing applications to extend their fixed interfaces, so that the extendibility is improved.

2.4 Machine Learning Approaches

Machine learning aims at simulating human activities using computers, so it is able to recognize known knowledge, gain new knowledge with which to improve its performance and optimize itself. Machine learning is being applied to various fields, such as biology, economics, chemistry, and computer science. In 1999, H. Nielsen et al. applied machine learning approaches to prediction of signal peptides and other protein sorting signals [24]. In 2005, CO AIM et al. proposed a method for predicting emotion based on text using machine learning [25]. Companies, such as Amazon, and IBM do research on machine learning as well. Amazon held a machine learning contest to verify whether it was possible to grant and revoke access to employees automatically. Researchers from IBM invented a way to extract heart failure diagnosis criteria from free-text physician notes using machine learning [26].

Generally, machine learning targets four categories of problems, which are regression, classification, clustering, and modeling uncertainty, known as inference.

A. Classification

   In classification, input data is divided into different categories. Normally, a classification task belongs to supervised learning, as the categories are labeled. The learning system gains knowledge, with which to assign new input data to one or more of these categories.

B. Regression

   To some extent, a regression problem is similar to classification, as it is also processed in a supervised way. The most significant difference is the output generated from regression problem is continuous, instead of discrete, like classification.

C. Clustering

   Clustering can be regarded as unsupervised version of classification. The basic functionality is also to classify a set of input into different classes; however, in clustering, the categories are not labeled anymore, which means the categories are generated as the system runs.

D. Modeling uncertainty

   Modeling uncertainty is not just to predict the frequency of random events. It integrates various factors that affect the occurrence of the event and analyzes the event using mathematical approaches, like Bayesian representation.

The process of establishing a face recognition system is to teach computers to recognize human face as human behavior, i.e. it is a learning procedure. Therefore, machine learning becomes a perfect solution to this problem. According to an evaluation of traditional face recognition techniques and machine learning

approaches by E. Garcia Amaro et al, machine learning generates a better result [56]. In M. S. Bartlett et al's work, a machine learning algorithm, Adaboost, is combined with SVM, LDA and Gabor filters, which are classical approaches applied to face recognition for building a facial expressions system. The result suggests a mean accuracy of 94.8%, and the system is able to operate in real-time [57].

Chapter 3

Framework for PCA-based Face Recognition

In this chapter, the classical PCA-based face recognition process is presented first, which shows the entire work flow and suggests some common approaches to the process. Then, a software framework for PCA-based face recognition system is proposed. All components contained in the framework are demonstrated in detail.

3.1 General Requirements

The framework's target is to provide users with a tool, which is able to help them apply PCA to face recognition applications. Meanwhile, various extreme conditions, such as non-uniform illumination, shooting angle, and facial expressions need to be considered.

The first requirement of the framework is to describe the complete PCA-based face recognition system so that, software developers can use it as a guide to customize their own applications. Therefore, the framework intends to cover as many cases as possible.

Second, the framework needs to be flexible. Hence, each phase in the process needs to include multiple variations in order to deal with different situations. Moreover, the attribute of each variation should be described explicitly, thus making it easier for developers to select. We also mention possible combinations between different variations for developers' reference.

Third, the model should be extendable. Since face recognition is still developing rapidly, more advanced techniques will be proposed to enhance the performance of current systems. The architecture of the framework should allow adjustment or enhancement in the future.

3.2 PCA based Face Recognition Process

The entire facial recognition process with PCA is shown in Figure 3 and includes 6 main steps: (1) image representation, (2) detecting face regions, (3) detecting facial features, (4) pre-processing, (5) conducting PCA, and (6) verification. Image representation converts the image data to a proper format. Face region detection and facial feature detection prepare meta-data for the following steps. Pre-processing reduces influences caused by the environment such as illumination to exhibit the exact information of the image. Conducting PCA classifies images based on the thresholds set at the verification step.

To verify a face image using PCA, two image datasets are needed: a training dataset providing data for building a customized PCA model, and a testing dataset containing the images to be verified.

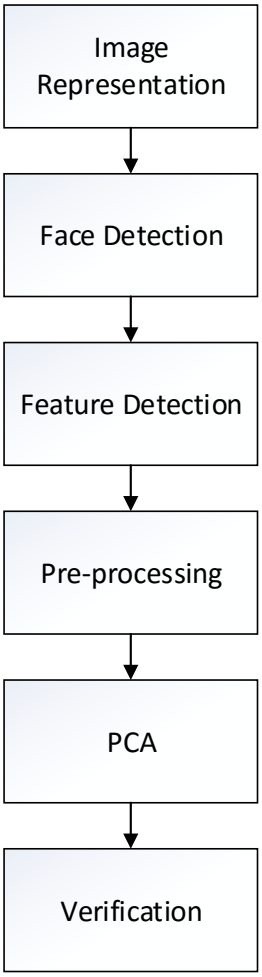

Figure 3 PCA-based face recognition system flow

3.2.1 Image Representation

Images are stored as a 2-dimensional (2-D) matrix in a computer. Each element in the matrix represents a single pixel whose value ranges from 0 to 255. Color images have 3 different channels which represent red, green and blue, respectively. An extra channel named α represents the transparency of the image. The number of rows and columns of the matrix depends on the resolution of the image, which means that if an image has a higher resolution then there are more rows and columns and more storage is needed. Moreover, the number of rows and columns significantly affects the matrix computation speed. The size factor leads to a requirement to compress the data, which is why we want to use PCA.

Image representation is not only to minimize the size of an image. Selecting appropriate image representation approaches for different recognition algorithms will improve efficiency and accuracy, which will be discussed in detail in Section 3.3.1.

3.2.2 Face Detection

Face detection extracts the face region from the background of the image. As we can see, this technique has been integrated into most smart phones and works well in most situations. However, an approximate face area might be good enough for smartphone cameras, but when conducting face recognition, slight noise would impact the final result. For an environment where background color is closed to skin color, or part of the face is in shadow, obtaining the face area becomes more difficult.

3.2.3 Feature Detection

To achieve high recognition accuracy using PCA, image alignment has to be performed. An affine transformation is a preferred choice for image alignment, as it is simple and can be processed quickly by a computer. To perform an affine transformation, three feature points are required on the facial image. The pupils of the eyes and the center point of mouth can provide these three points. Thus, the task of this step is to acquire these three feature points from face images

In 2003, Z.Y.Peng, HZ Ai et al., proposed a feature detection method based on weight similarity [5]. First, the algorithm transforms the image to a binary format and the face area can be represented by $B(x,y)$. Equation (9) shows the threshold for the binary image where $H(i)$ stands for the histogram of the original image. Based on the pixel distribution of the face image, approximate areas of left eye and right eye can be measured, which can be represented as $L(x,y)$ and $R(x,y)$ respectively. Since the color of pupils differs from other part of eyes, once a point $P_l(x,y) = 1$ is found, it can be assumed as left pupil candidate. Similarly, once a point $P_r(x,y) = 1$ is found, it can be assumed as right pupil candidate. If both of P_l and P_r meet the condition shown in Equation (10), they can be confirmed as the center points of two pupils. In the Equation (10), $\gamma(P_l,P_r)$ stands for the similarity of the neighborhood of P_l and P_r, $D(P_l,P_r)$ and $A(P_l,P_r)$ are the distance constraint and angle constraint of P_l and P_r respectively. After localizing two pupils, the center point of mouth P_m can be confirmed by integral projection. Figure 4 shows the flow of feature detection.

$$B(x,y) = \begin{cases} 0 \;\; if \; A(x,y) \geq \theta \\ 1 \;\; if \; A(x,y) \leq \theta \end{cases}, \sum_{i=0}^{\theta} H(i) = 15\% \times \sum_{255}^{i=0} H(i) \Bigg\} \tag{9}$$

$$B(x,y) = \boldsymbol{max}(\gamma(P_l,P_r)D(P_l,P_r)A(P_l,P_r)) \tag{10}$$

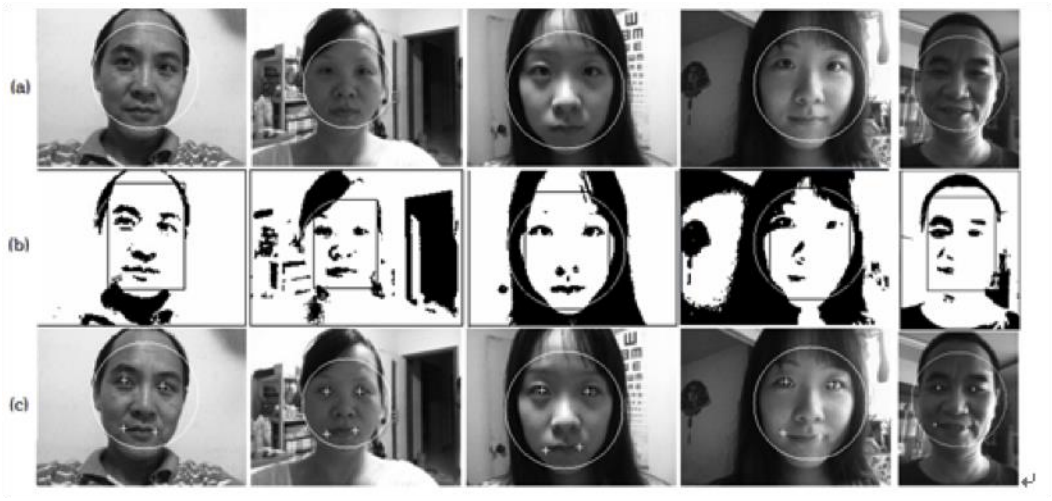

Figure 4 Feature detection [5]

3.2.4 Pre-processing

Image pre-processing is always an important topic in face recognition, since in this step, most factors which potentially affect the final recognition result expect to be eliminated. Researchers have explored a large number of methods to reduce different types of noise, including histogram normalization and converting an image to a binary representation. The majority of these methods aim to reduce noise, whereas some of them also change the format of the image which then can be used in later steps. In this section, we continue the content in feature detection and introduce the basic idea of affine transformation in images.

As we have discussed, three feature points, i.e., two pupils and the center point of mouth can be represented as P_l, P_r, and P_m. Basically, an affine transformation aligns every image according to the same template. The three feature points will remain in the same position, and the other pixels will be moved. Equation (11) shows the principal of affine transformation, where *(x, y)* and *(x', y')* stand for the pixels on the original face image and transformed template image, respectively.

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}, \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} \neq 0 \tag{11}$$

3.2.5 Principal Component Analysis

The concepts behind PCA were invented in 1901 by Karl Pearson [6]. The original idea is to use an orthogonal transformation to convert a set of observations of possibly correlated variables into a set of values of linearly uncorrelated variables called principal components. In 1991, Pentland and Turk transplanted PCA to face recognition and proposed a method called eigenface, which is frequently used in face recognition research. The basic idea is to extract the most significant information from face images which then can be used to form a sub-space named feature space. The dimensionality of the new formed sub-space is much smaller than that of original images, but components which are used for identifying a face are retained. The image set used to build this sub-space is called a training set, and the image set reflecting the components in the sub-space is called an eigenface. Once the sub-space is established, a testing image can be projected onto the space to generate a new image. The similarity of this new image and the original image can be used for the verification step.

In Equation (12) and (13), M stands for the dimensionality of the feature sub-space, U_k, $k = 1,2,...,M$ are the eigenfaces, ω stands for the average face.

Figure 5 shows a set of eigenfaces generated from a training dataset containing about 200 pictures.

Figure 6 shows 3 original pictures and corresponding ones after being projected to the sub-space. As the pictures of the training dataset are of the same person, so the projected pictures are relatively similar to the original ones.

$$X' = \sum_{k=0}^{M} \omega_k U_k, \quad k = 1,2,\dots,M \tag{12}$$

$$\omega_{k=} U_k^T (X - \varphi), \quad k = 1,2,\dots,M \tag{13}$$

Figure 5 Eigenfaces [11]

Figure 6 Original images and projected images [11]

3.2.6 Verification

As already mentioned, the verification step compares the original input image with its projection on the feature sub-space. However, there are many methods for comparing the similarity of two images. Choosing the proper method can help generate better results.

Since images are represented as a matrix, the problem becomes one of comparing the similarity of two matrices or vectors, which can be easily solved with statistical methods.

Measures such as Euclidean distance, Manhattan distance, Chebyshev distance, and Minkowski distance are all sufficient to tackle such a problem. Each of them has its own advantages and disadvantages, so selecting the optimal measure to for the problem is important.

3.3 Framework model

In this section, a framework for a face recognition system with PCA is presented, as shown in Figure 7. The framework describes the entire recognition process, and for each phase in the process, some possible variations are offered to adapt to different cases and to help software developers customize their application. Some extreme cases, such as non-uniform illumination, exaggerated facial expression, shooting angle of the images, and data type of the images are considered. Besides the options included in the framework, we suggest other potential approaches. The outline of the framework follows.

For face representation, we will talk about Gabor wavelet, PCA expression, and shape and texture expression. For face detection, we will talk about statistical model, neural networks, and color based methods. For pre-processing, we will talk about face separation, and local binary pattern (LBP). For PCA, which is the core step, we will talk about Kernel PCA and standard PCA.

Through the combination of different variations, the framework is able to provide at least 108 application instances in total, as there are three variations for face recognition step, face detection step, and verification step respectively, and two variations for pre-processing step and PCA step respectively. However, the application instances which can be produced by the framework is a lot more than 108, since in practical use, some variations in the same step might be combined, some steps can be omitted, and some variations need to collaborate with other simple mathematical operations. Therefore, a conservative estimate of the actual number of application instances that can be produced by this framework exceeds 150. Though these instances are not able to cover all situations, the framework offers a significant assistance to software developers.

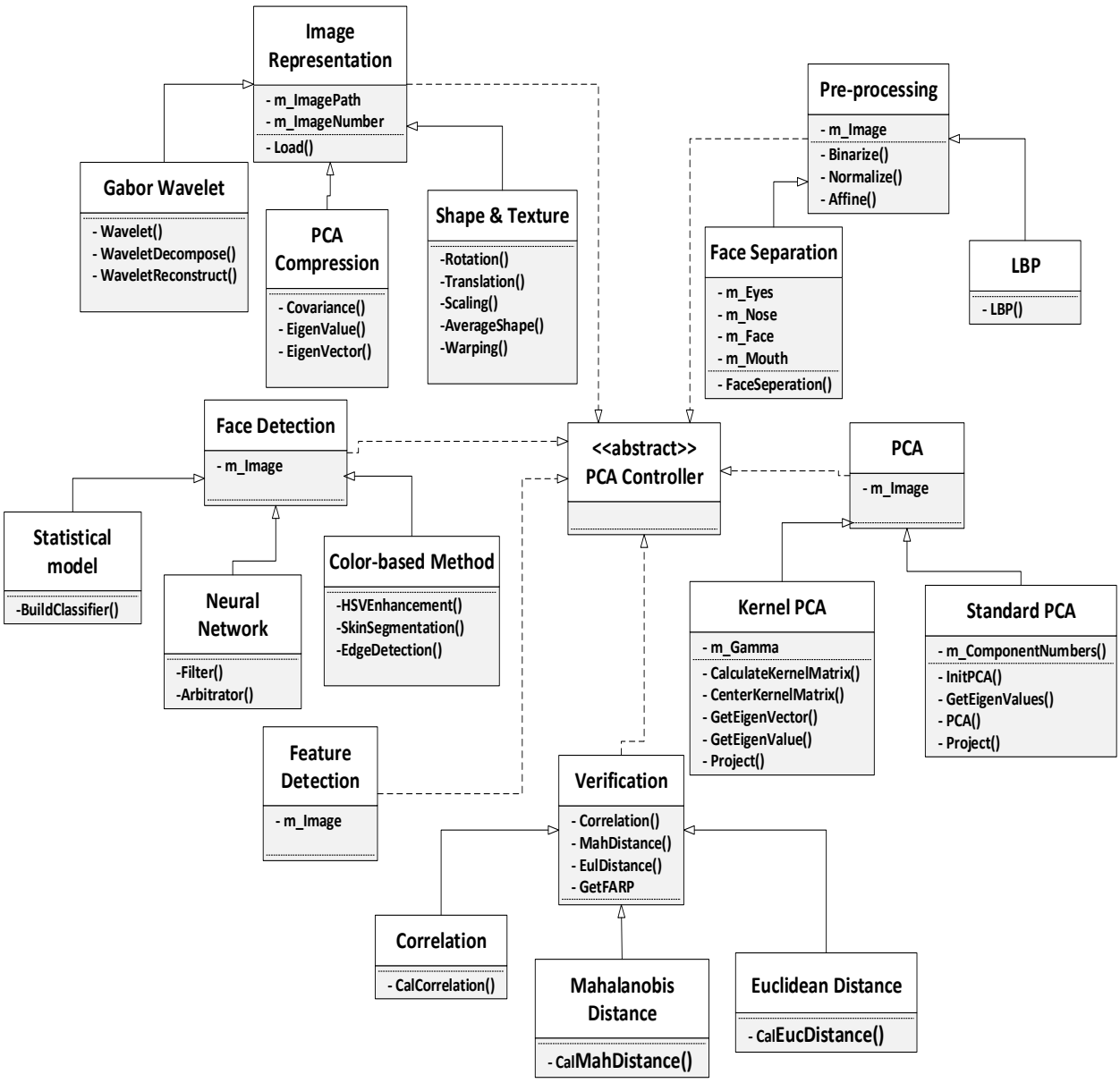

Figure 7 Framework design model

3.3.1 Image Representation

As the first processing step of face recognition, image representation playes an important role not only in explicitly representing the image information, but reducing noise and compressing data. Appropriate selection of image representation approaches facilitates the later steps and improves the overall performance of the entire system. In this section, we present three different variations for representing images, which are Gabor Wavelet, PCA Compression and Shape and Texture, as shown in Figure 8.

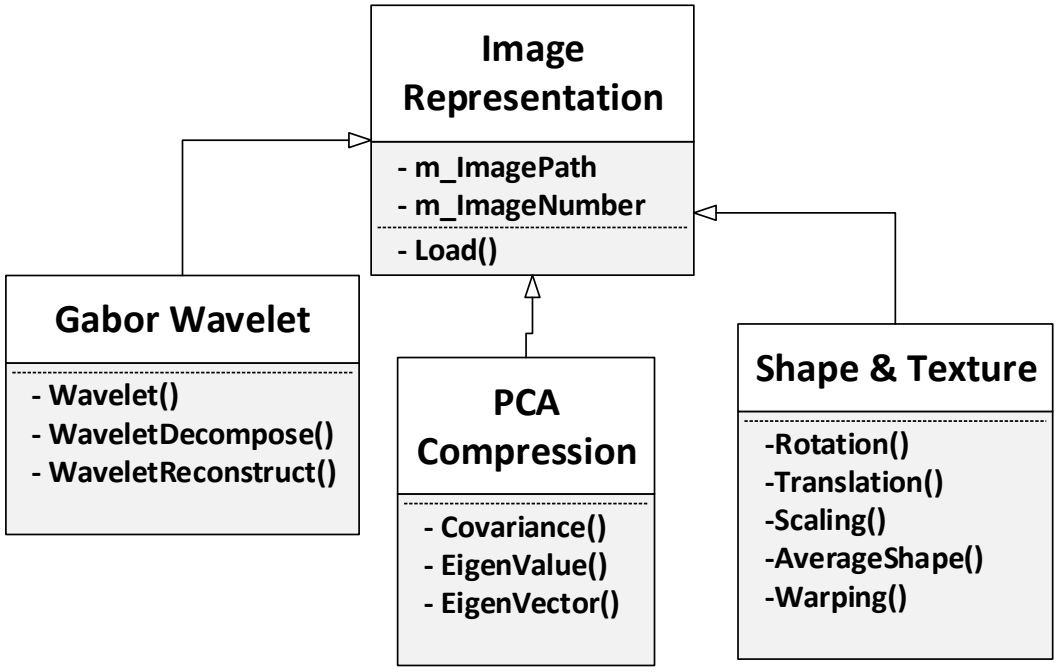

Figure 8 Image representation

3.3.1.1 Gabor Wavelet

Image processing methods are mainly divided into two categories, which are spatial domain analysis and frequency domain analysis. Spatial domain analysis directly processes the image matrix; however, frequency domain approaches convert the image from the spatial domain to the frequency domain, and then analyze the image feature from another perspective. Spatial domain analysis is widely used in image reinforcement, image reconstruction, and image compression [25].

Fourier transforms are one of the earliest method of transferring signals from the spatial domain to the frequency domain, as shown in Equation (14). After processing the signal, an inverse transform can transfer the signal back to spatial domain through Equation (15).

$$F(\omega) = \int_{-\infty}^{\infty} f(t) e^{-j\omega t} dt = F[f(t)] \tag{14}$$

$$f(t) = \frac{1}{2\pi} \int_{-\infty}^{\infty} F(\omega) e^{-j\omega t} d\omega = F^{-1}[f(t)] \tag{15}$$

Classic Fourier transform provides a powerful tool for image processing; however, it is only able to reflect the integral attributes of signals, which means it lacks the ability to do local analysis.

Based on Fourier transform, Dennis Gabor proposed a new transform which only depends on part of the signal and is able to extract local information, [45].

The basic idea of the Gabor transform is to divide the signal into multiple intervals, and then analyze each interval using a Fourier transform, so that the frequency in certain interval can be obtained. In image processing, a Gabor transform is also known as Gabor wavelet.

A Gabor wavelet is similar to the stimulation of a simple cell in a human being's visual system [28]. It captures salient visual properties such as spatial localization, orientation selectivity, and spatial frequency. Furthermore, as Gabor wavelet is insensitive to illumination variation, it provides good adaptability to illumination variation in image representation.

It has been proved that the Gabor wavelet is particularly suitable for image decomposition and representation when the goal is the derivation of local and discriminating features. In 1999, Donato et al. [7] showed that the Gabor filter representation gave better performance for classifying facial actions. In 2001, Chengjun and et al. [29] presented an independent Gabor feature method for face recognition. The method achieved 98.5% correct face recognition accuracy on FERET dataset, and 100% accuracy on ORL dataset.

3.3.1.2 PCA Compression

As the core of this study, PCA is the main step of a face recognition system that we discuss. However, it also can be used as an image representation method when combining with other recognizing approaches. When representing face image using PCA, the main idea is to transfer the original image to a format with lower dimensions, i.e., to represent by a smaller number of parameters.

PCA was first applied to the realm of pattern recognition in 1965 by Watanabe [30]. In 1990, Kirby et al. introduced the method to face recognition, particularly characterization of human faces [31]. The work introduces a concept of optimal coordinate system, in which the set of basis vectors which makes up the system are referred to as eigen-pictures. They are actually the eigenfunctions of the covariance matrix of the ensemble of faces. For the evaluation of the procedure, face images from outside of the dataset are projected on the set of optimal basis vectors. The result shows a 3.68 percent error rate out of over ten face images, which dominated in this field at the time.

As the development of face recognition technique evolves, some variations of PCA-based image representation method have been invented. Moreover, PCA-based image representation is always combined with other recognition approaches to achieve better recognition accuracy.

In 2005, Daoqiang and Zhi-Hua proposed a two-directional two dimensional PCA for face representation and recognition [32]. It has been proved that 2DPCA outperforms standard PCA in terms of recognition accuracy. However, 2DPCA needs more coefficients for image presentation than PCA.

Therefore, Daoqiang and Zhi-Hua conduct PCA on row and column directions simultaneously, which results in a same or even higher recognition accuracy than 2DPCA, though with less coefficients needed.

3.3.1.3 Shape and Texture Expression

Shape and texture expression methods uses geometric features to represent a human face. As shape and texture expressions ignore the color and illumination of a face, it reduces the noise that impacts the recognition accuracy. In fact, shape-based and texture based methods were independent initially. After proving that the shape-based method cannot perfectly solve the problem caused by expression, scale and illumination, texture is introduced to be combined with shape to achieve better recognition accuracy [32].

Chengjun Liu and Harry Wechsler's paper in 2001 [8] clearly explains the work flow of shape and texture-based face expression methods. We use some of the figures and explanations to demonstrate the principle in detail.

The shape of face images reflects the contours of face, so a set of control points derived by manual annotation are used to describe the contour. To underscore the shape features of face, these points ignore other facial information like color, gray scale. They only depict feature points such as eyebrows, eyes, nose mouth, and the contour of face, as shown in Figure 9. Generally, the shape image is generated from a large set of training images. After obtaining the shape from each training image, the shapes are aligned by rotation, translation and scaling transformations. The aligned shapes of training images are shown in Figure 10.

Calculating the average of the aligned shapes of training images, a mean shape can be obtained. Then warp the normalized (shape-free) face image to the mean image, a new image is created, which is the texture, as shown in Figure 11. The warping transformation basically separated the image into multiple small triangular regions and then performs affine transformation on each of them to warp the original face to the mean shape. These two steps will result in a texture (shape-free image) which has the same face contour as the mean shape.

The experimental result of Chengjun Liu and Harry Wechsler's work shows that the integrated shape and texture features capture the most discriminating information of a face, which contributes to their high recognizing accuracy. Besides their work, other research [34] [35] [36] demonstrate the advantages of using shape and texture method to represent facial image.

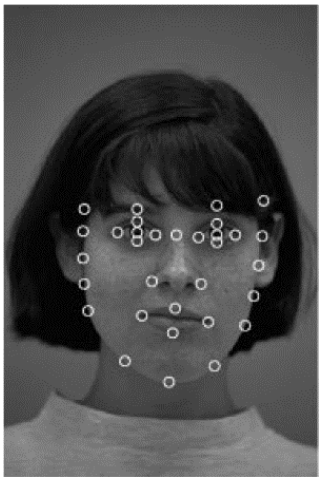

Figure 9 Control points [8]

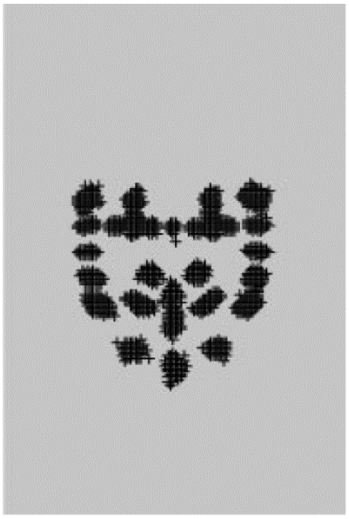

Figure 10 Shape [8]

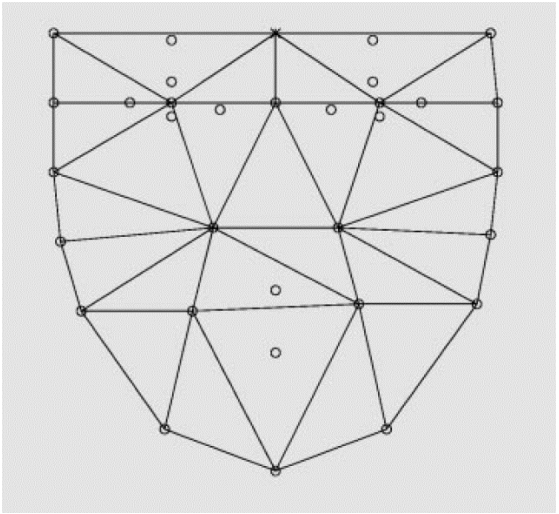

Figure 11 Texture [8]

3.3.2 Face Detection

Face detection provides the basic face area image for the entire recognition process. The detection accuracy significantly influences the final result, as too much background noise affects most recognition algorithms. For this phase, we provide three variations, which are the statistical model, Neural Network, and color-based method, as shown in Figure 12.

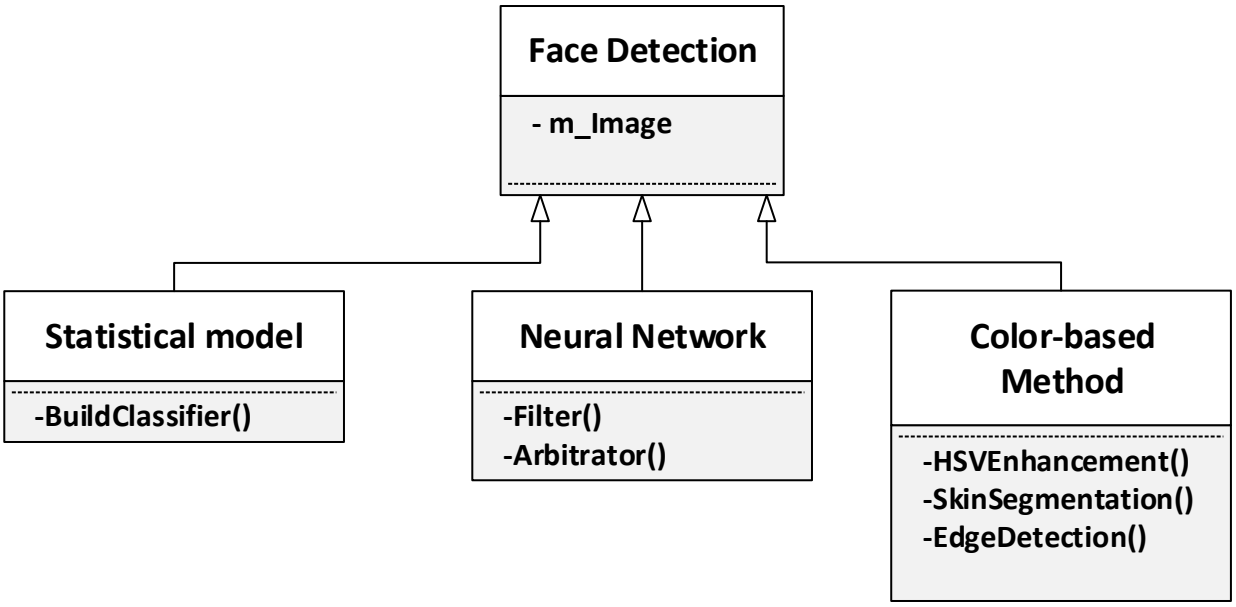

Figure 12 Face detection

3.3.2.1 Statistical Model

The complexity of human face images makes the detection of face features to be difficult, therefore statistical-based detection methods have been attracting researchers' attention. This method regards the face region as a type of pattern, also known as pattern feature, and uses a large number of face image and non-face image to train and generate a classifier. So the more training the detection method receives, the more robust it will be.

Feature space-based methods, such as PCA, LDA, probabilistic model-based methods, and support vector machine-based (SVM) methods all belong to statistics-based detection methods. Actually, neural networks-based methods, which are discussed next also utilize statistical principles; however, we explain it individually because of some of its peculiarities.

In 2000, Henry et al. [9] proposed a statistical method for 3D object detection. The method is able to detect both object appearance and "non-object" appearance using the product of histograms. Each histogram represents the joint statistics of a subset of wavelet coefficients and their position on the object. The approach uses many such histograms that represent a wide variety of visual attributes. The result demonstrates detection accuracy.

In 1997, Baback Moghaddam and Alex Pentland proposed a probabilistic visual learning for object representation, which is based on density estimation in high-dimensional spaces using an eigenspace decomposition [37]. The technique has been applied to not only face detection, but gesture recognition.

3.3.2.2 Artificial Neural Network

An Artificial Neural Network is a computation model consisting of many neurons, in which each neuron includes a specific output function called an activation function. The connection between every two neurons has a weight that processes the output from the first neuron.

The most significant attributes of artificial neural networks are their adaptability and parallelism. Adaptability grants its ability to learn through training and autonomously correct weights in connections to avoid the same faults. As each neuron in the network is responsible for a certain job, an artificial neural network is able to work in parallell, which facilitates processing big data such as images.

Because of these two remarkable attributes, artifical neural networks have been attracting attention from researchers in face recognition. In 1997, Shang-hung et al. proposed a face detection method using a probabilistic decision-based neural network. The detection accuracy reaches 98.34%. In 1998, Henry et al. [10] presented a neural network-based upright frontal face detection system. Figure 13 shows the basic idea of the system. A window of the image which might contain a face region is first obtained. Then a series of pre-processing steps, such as light correction and histogram normalization, are applied to the window to reduce noise. Finally, the window is passed through a neural network, which decides whether the window contains a face. It has been shown in the result that the algorithm can detect between 77.9 percent and 90.3 percent of faces in a set of 130 test images, with an acceptable number of false positives.

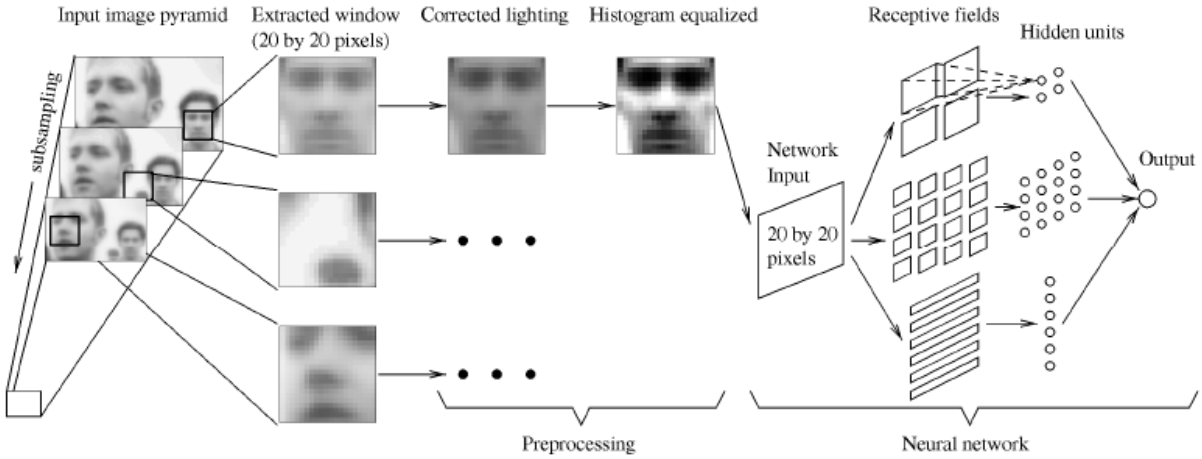

Figure 13 Artificial neural network-based method [10]

3.3.2.3 Color-based Methods

Traditional face detection approaches are always performed in gray-scale space, in which the gray-scale is the only information that can be captured. Moreover, since there is no limit for area or proportion, it is necessary to search the entire space, which is fairly time-consuming. However, if color information can be introduced, the search area will be narrowed, because the skin color is the most straightforward information on a human face. In addition, in the face region, skin color dominates.

A problem that needs to be considered is the difference of skin color. Fortunately, research in this field shows that skin color in certain color space aggregates, especially when the illumination factor is removed [46]. Therefore, using skin color as a clue to exclude any area which is not skin can be easily performed.

When applied to face detection, skin color information is always used in three different phases of face detection. It could be used as the core function, the pre-processing method, or in post-verification. For instance, in 2002, Hichem Sahbi and Nozha Boujemaa proposed a skin color approach for face detection combined with image segmentation. In their approach, the images are first separated coarsely to provide regions of homogeneous statistical color distribution. Then the color distribution will be used for training a neural network to detect faces. The experiment result shows an accuracy of around 90%.

3.3.3 Pre-processing

A pre-processing step can be regarded as a filter which reduces major noise impacting the following recognition process. It aims at generating clear images with useful information retained. As shown in Figure 14, this section discusses two pre-processing approaches, which are face separation and LBP. Face separation mainly handles face images with exaggerated expressions, and local binary patterns deal with non-uniform illumination.

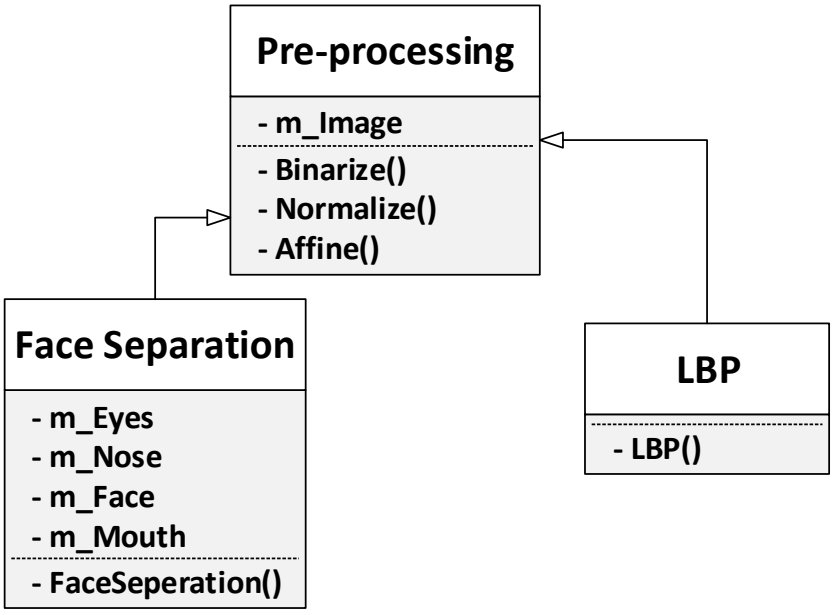

Figure 14 Pre-processing

3.3.3.1 Face Separation

When people take pictures, it is hard to always control their facial expressions. On the other hand, as an ideal face recognition system, it should not be expected that subjects will have normal expressions. However, extremely exaggerated facial expressions do impact the recognition process, especially for feature detection or image aligning. To reduce the influence, this pre-processing method divides a face image into 4 parts which are: eyes, nose, mouth, and entire face, so the following steps can be performed on each part, as shown in Figure 15.

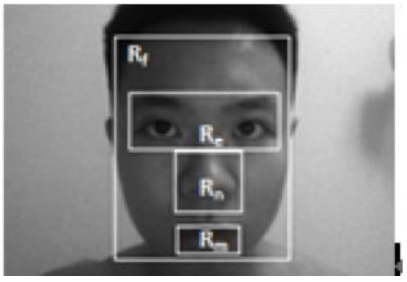

Figure 15 Face separation [11]

In 2014, Peng et al., used face separation for standard PCA-based face recognition system [11]. They conducted PCA on each part of the face mentioned previously, and integrated the score with Equation (16), where δ_F stands for the score of entire face, δ_M stands for the score of mouth, δ_N stands for the score of nose, and δ_E stands for the score of eyes. The weight assigned for each part is obtained from

experimentation. The result shows significant progress in recognizing face images with exaggerated expressions.

$$\delta = 0.40 \times \delta_F + 0.10 \times \delta_M + 0.10 \times \delta_N + 0.40 \times \delta_E \quad (16)$$

3.3.3.2 Local binary pattern

Local binary pattern, known as LBP, was proposed by D.C. He and L. Wang in 1990 [12]. The basic idea is to calculate a weighted sum for a single pixel with it neighboring pixels. Generally, the window size of sum is set as 3×3. Basically, it is still creating a binary representation of an image. LBP traverses the image using every pixel as a center point, and for all of the eight neighboring pixels, a calculation is performed. If a pixel has a gray value less than the gray value of the center point, assign the pixel a value of zero; otherwise, assign one, as shown in Equation 17 and Figure 16, where $I(Z_i)$ represents neighboring pixels, and $I(Z_0)$ represents the center pixel. The assignment process can be demonstrated by Figure 16. The weight assigned to each pixel is always different. One possible weight distribution is shown in Figure 17. Figure 18 shows an image which is processed by LBP.

$$f\big(I(Z_0), I(Z_i)\big) = \begin{cases} 0, if\,(I(Z_i) - I(Z_0) > 0) \\ 1, if\,(I(Z_i) - I(Z_0) < 0) \end{cases}, i = 1,2,3, \ldots,8 \quad (17)$$

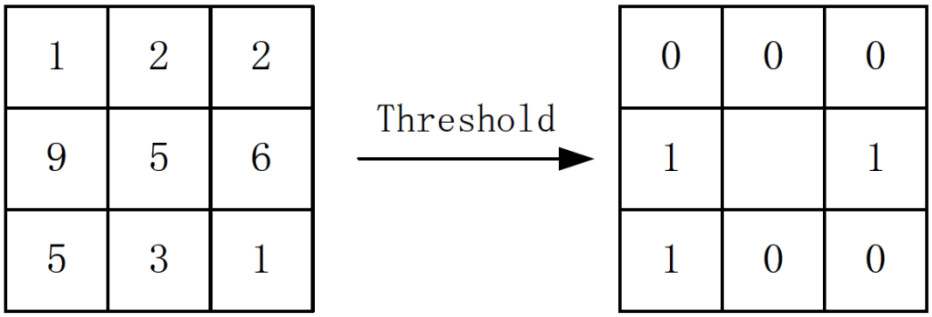

Figure 16 LBP

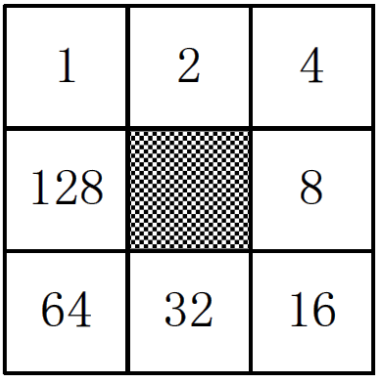

Figure 17 LBP weight

Figure 18 LBP result [11]

It has been shown that LBP performs well on image classification, and as the computation is simple, it is efficient for most cases. However, it has some limitations including low extendibility and scalability.

Fortunately, after about 10 years of research, some variations of LBP have partially overcome these deficiencies. In 2002, Ojala T. et al. [13], proposed a method which uses a circular neighborhood with arbitrary radius instead of a 3×3 window, as shown in Figure 18. In 2007, Xiaoyang Tan et al. [14], made changes based on LBP and proposed Local Ternary Patterns (LTP), which compares the value of neighbor pixels with the value of the center pixel plus a range value *t*. Then the eight neighbor values are able to be encoded. The process is shown in Figure 19. Besides the variations of LBP, researchers also combined LBP with other algorithms, in order to enhance the efficiency.

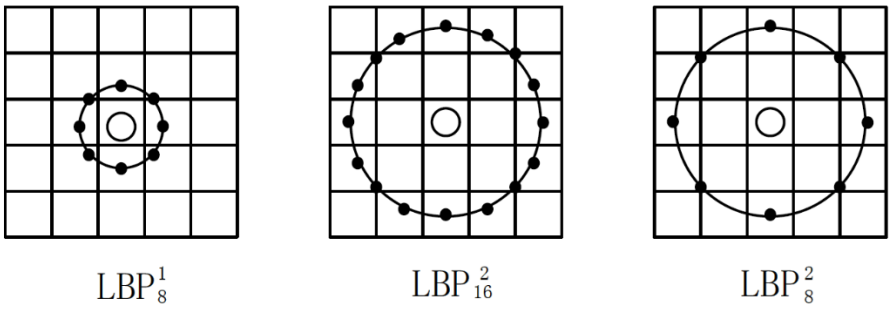

Figure 19 Circle LBP

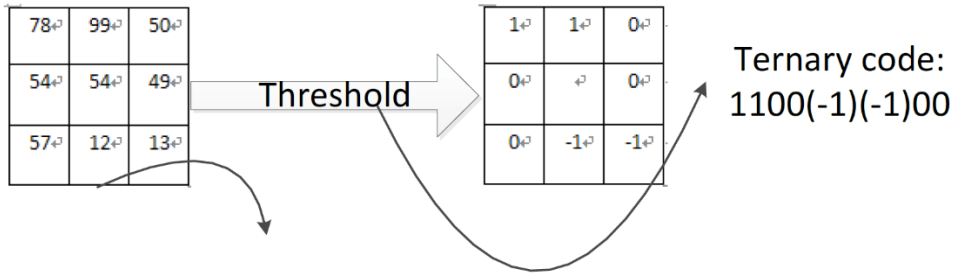

Figure 20 LTP

3.3.4 PCA

In this section, the core step, using PCA for recognition of face images is discussed. As the Figure 21 shows, two variations are provided. Standard PCA is mainly used for linear image data, whereas kernel PCA is used for non-linear image data. The source of the figures in this section can be found at [39].

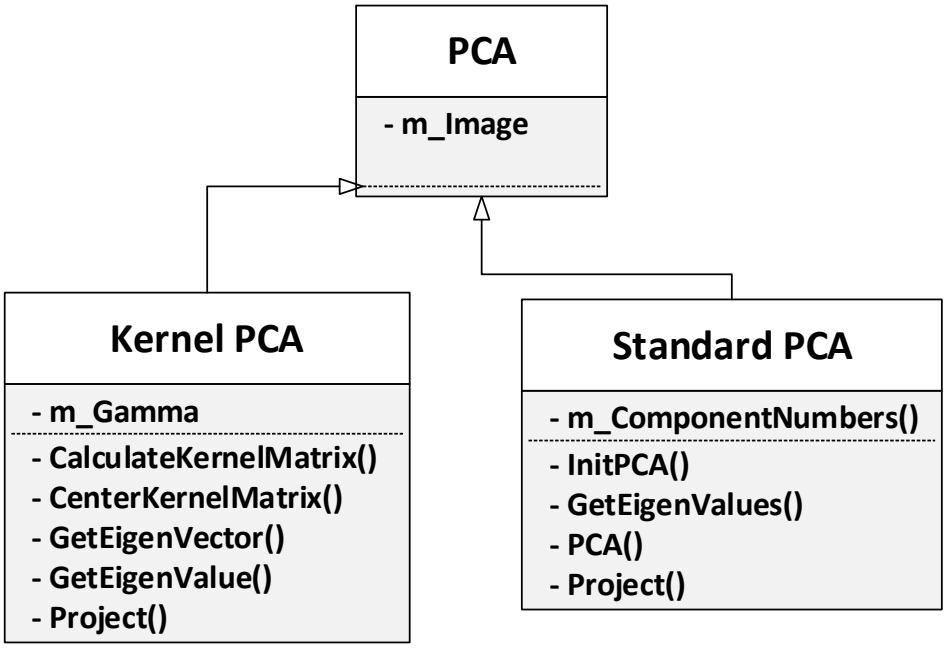

Figure 21 PCA

3.3.4.1 Standard PCA

A common application of PCA is to reduce the dimensions of the dataset with minimal loss of information. Here, the entire dataset (d dimensions) is projected onto a new subspace (k dimensions where k << d).

The standard PCA approach can be summarized as six simple steps [15]:

1. Compute the covariance matrix of the original d-dimensional dataset X.
2. Compute the eigenvectors and eigenvalues of the dataset.
3. Sort the eigenvalues by decreasing order.
4. Choose the k eigenvectors that correspond to the k largest eigenvalues where k is the number of dimensions for the new feature subspace.
5. Construct the projection matrix W of the k selected eigenvectors.
6. Transform the original dataset X to obtain the k-dimensional feature subspace Y.

Figures 22, 23, and 24 show an example of using PCA on a dataset with the dimensionality of three. Figure 22 shows the original dataset. Two categories of data are mixed together and hard to be classified. Figure 23 shows the eigenvalues and eigenvectors of the original dataset. After being processed by PCA, the dimensionality is reduced to 2 and the classification is clearer, as shown in Figure 24.

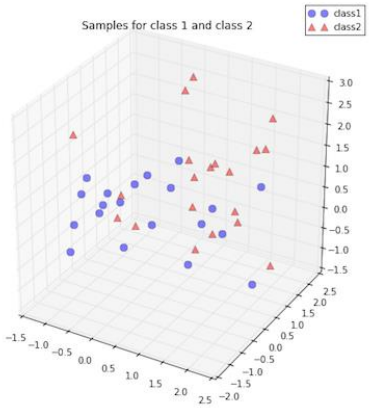

Figure 22 Original dataset [39]

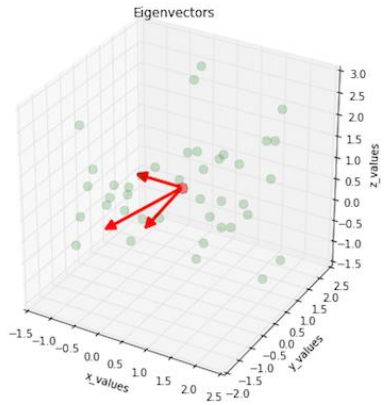

Figure 23 Eigenvalues and eigenvectors [39]

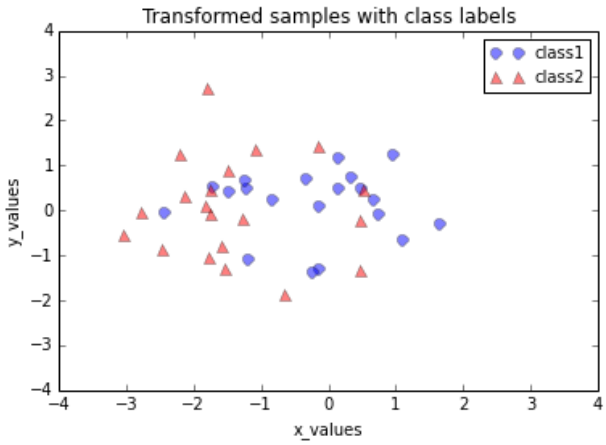

Figure 24 Classification by using standard PCA [39]

The original idea of using PCA on face recognition was first proposed by Turk and Pentland in 1991 [16]. The basic principle has been introduced in previous sections. Although researchers have been studying PCA for decades, it is still a preferred approach for face recognition, because of its robust nature and extendibility. In addition, it is easy to combine PCA with other existing methods.

3.3.4.2 Kernel PCA

Kernel PCA is an extension of PCA using techniques of kernel methods. The basic idea is to first map the input space into a feature space via nonlinear mapping and then compute the principal components in that feature space.

Standard PCA works well if the data is linearly separable. However, in practice, image data is always impacted by external factors, such as shooting angle, illumination, and other noise. So the property of being linearly seperable is not guaranteed. Hence, a method dealing with nonlinear cases is required. Figure 25 shows a comparison between linear data and non-linear data.

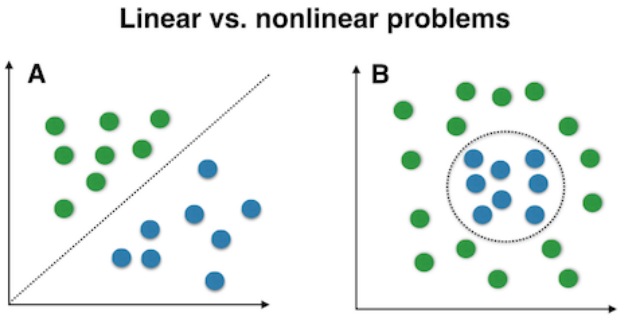

Figure 25 Data type [39]

There are 2 main steps in the implementation of kernel PCA:

1. Computation of the kernel (similarity) matrix.
2. Eigen decomposition of the kernel matrix.

Figure 26, 27, 28, and 29 show an example using standard PCA and the Gaussian radial basis function (RBF) kernel PCA on a nonlinear dataset. Figure 26 and Figure 28 show the distribution of data. Figure27 and Figure 29 are the corresponding results. It can be clearly seen that the projection via RBF kernel PCA yielded a subspace where the classes are well separated.

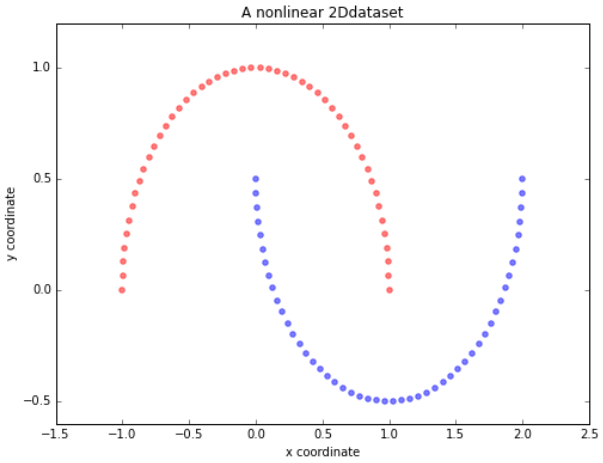

Figure 26 Original data [39]

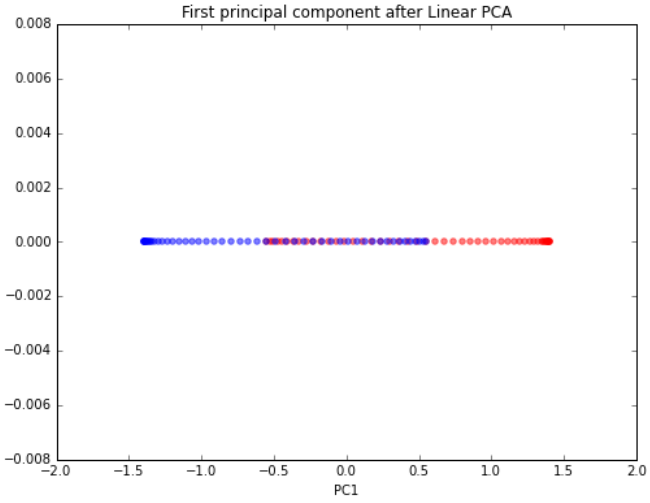

Figure 27 Standard PCA result [39]

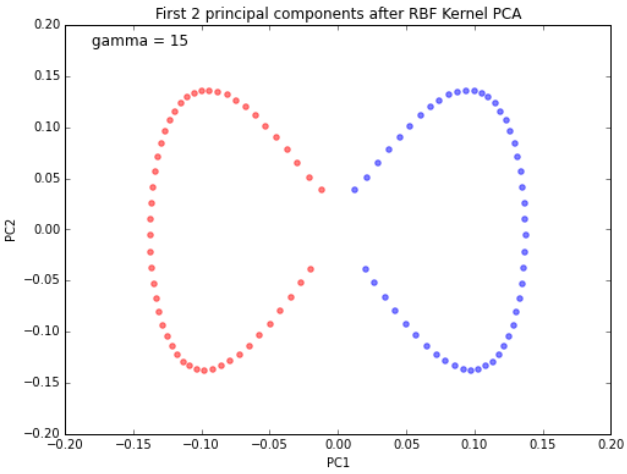

Figure 28 Original data [39]

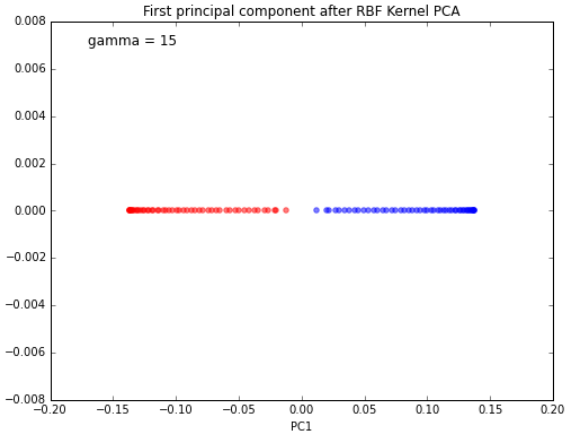

Figure 29 Kernel PCA result [39]

3.3.5 Verification

As shown in Figure 30, we introduce three measurements for the final verification step. The methods in this step are basically mathematical formulae related to matrix operations, as the verification step is actually comparing the results represented as a matrix from previous steps. The three methods are correlation, Mahalanobis distance, and Euclidean distance.

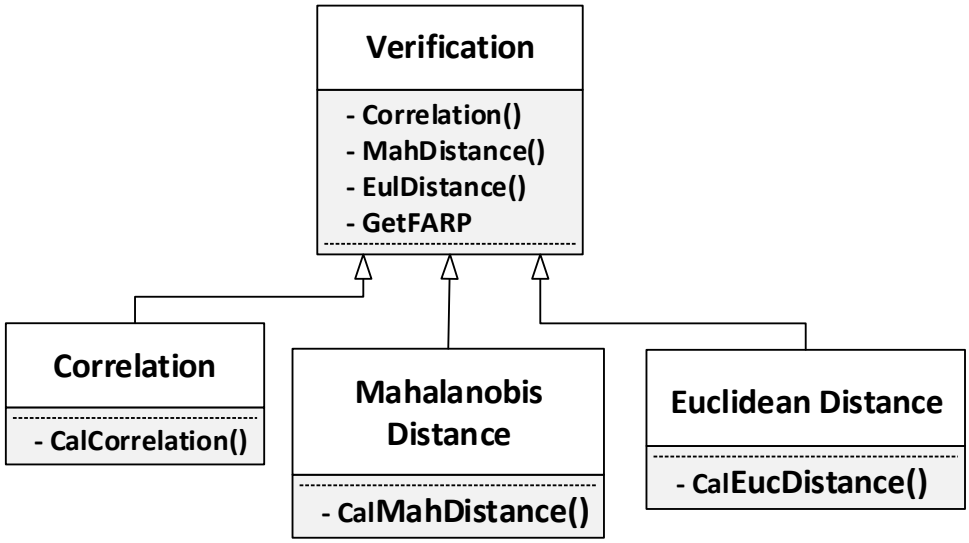

Figure 30 Verification

3.3.5.1 Correlation

In statistics, a correlation table and a correlation graph are able to describe the relationship and relation direction between two variables. However, the degree of the correlation cannot be measured. Therefore, the statistician Karl Person proposed a correlation coefficient. According to the type of research target, correlation coefficients can be classified into three categories, which are simple correlation, multiple correlation, and classic correlation. Here, we only discuss simple correlation.

The correlation of two variables can be calculated with Equation (18).

$$\delta(X, X') = \frac{E(XX') - E(X)E(X')}{\sigma(X)\sigma(X')} \tag{18}$$

Where *E(X)* stands for the expectation of variable *X* and *σ(X)* is the variance of *X*.

Correlation has relatively low computational complexity, but the result varies according to the number of samples. When the number of samples is small, the result fluctuates significantly with the addition of another sample; whereas, when the number of samples is large addition of a sample will not have much effect.

3.3.5.2 Mahalanobis distance

Mahalanobis distance was proposed by the Indian Statistician P. C. Mahalanobis, which expresses the covariance distance of data. Compared with other similar measurements, Mahalanobis distance takes the relationship between various features into account. Moreover, it is scale-invariant, which means it is able to remove the interference between variables. The Mahalanobis distance of two vectors can be calculated with Equation (19).

$$D(X_i, X_j) = \sqrt{(X_i - X_j)^T S^{-1}(X_i - X_j)} \quad (19)$$

Where *S* represents the covariance matrix.

The most significant shortcoming of Mahalanobis distance is that it might exaggerate the impact of variable with small variation.

3.3.5.3 Euclidean distance

Euclidean distance is one of the most widely used measurements in statistics. It reflects the actual distance of two vectors in space. Euclidean distance in 2D space can be calculated with Equation (20).

$$O(\rho) = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2} \quad (20)$$

When extended to n-dimensional space, the Euclidean distance of vector a(X*11*, X*12*,…, X*1n*) and vector b(X*21*, X*22*,…,X*2n*) can be calculated with Equation (21).

$$d_{ab} = \sqrt{\sum_{k=1}^{n}(x_{1k} - x_{2k})^2} \quad (21)$$

Euclidean distance is the simplest measurement of two random variables. However, since it does not consider the distribution or relationship among different variables, it may not be able to reflect as much information as other approaches do.

Chapter 4 Case Studies

In order to prove the utility of the model proposed in previous chapter, this chapter presents four case studies which utilize different variations of the model. The case studies do not simply select variations from the model. Instead, they show why the variations are chosen and what type of problems can be solved. Furthermore, to achieve optimal performance, for some of the steps in the case studies, the combinations and mutations of the variations are presented.

The first case study targets the construction of a PCA-based face recognition system for smart phones. For this application, the main problems are illumination and facial expression. Therefore, the variations chosen for the model aim at reducing the influences caused by these two factors. The system description and requirements are provided. Additionally, the C++ code for each variation is provided as well.

The other three case studies focus on the choice of variations. To demonstrate how the model can help users customize their applications, these three case studies select different variations for each step. Furthermore, the reason of choosing the variation, i.e., the type of problems which is solved by the variation, is also provided to help users understand and use the model better. For all of these three case studies, the overview of the system and the demonstration of selecting each variation is presented. In the second case study, the C++ code for implementing each variation is also provided. However, for the remaining two case studies, the code is not given in the text owing to the length of the thesis, but the complete implementation of the model is provided in the appendix.

At the end of this chapter, the variations which are not selected in any of the case studies are discussed. Additionally, the combination of some variations and omission of some steps in the model are also suggested. It should be noticed that, the case studies in this chapter intend to provide a guide to how the model can be used. Nevertheless, since the total number of variations which can be produced by the model exceeds 150, the case studies are not able to cover all cases.

4.1 PCA-based face recognition system for smart phones

4.1.1 Description

Face recognition is proving its value on smartphone security, where it finds a more suitable environment for implementation than on desktop computers. Furthermore, the development of the frontal camera for smartphones facilitates the face recognition system, since the photos are always shot with high resolution and from frontal angle. However, there are several salient factors affecting the quality of input

images for the face recognition, such as non-uniform illumination, and exaggerated facial expressions, because of the usage habits of smartphones. Therefore, the system of this case study specifically aims at solving the aforementioned two problems.

4.1.2 System Overview

Targeting to solve the illumination and facial expression problems, the case study selects variations from the model presented in Chapter 3 to build a face recognition system for smart phones. To achieve the best performance, the system does not strictly follow the flow of the model, i.e. for some phases, all variations are used collaboratively while for some phases, none are used.

Before applying the model, binarization and normalization are employed to convert the color imgae to a grey-scale image. The first phase in the model, image representation is omitted in this case study because of the limited computational ability of smartphones. For face detection, a statistical-based approach is selected followed by a feature dectection step. In the pre-processing step, two variations, face separation and LBP are combined as both illumination and facial expression problems need to be considered. For the PCA step, standard PCA is chosen. Finally, correlation is selected as the verification method. The overview of this face recognition system is presented in Figure 31.

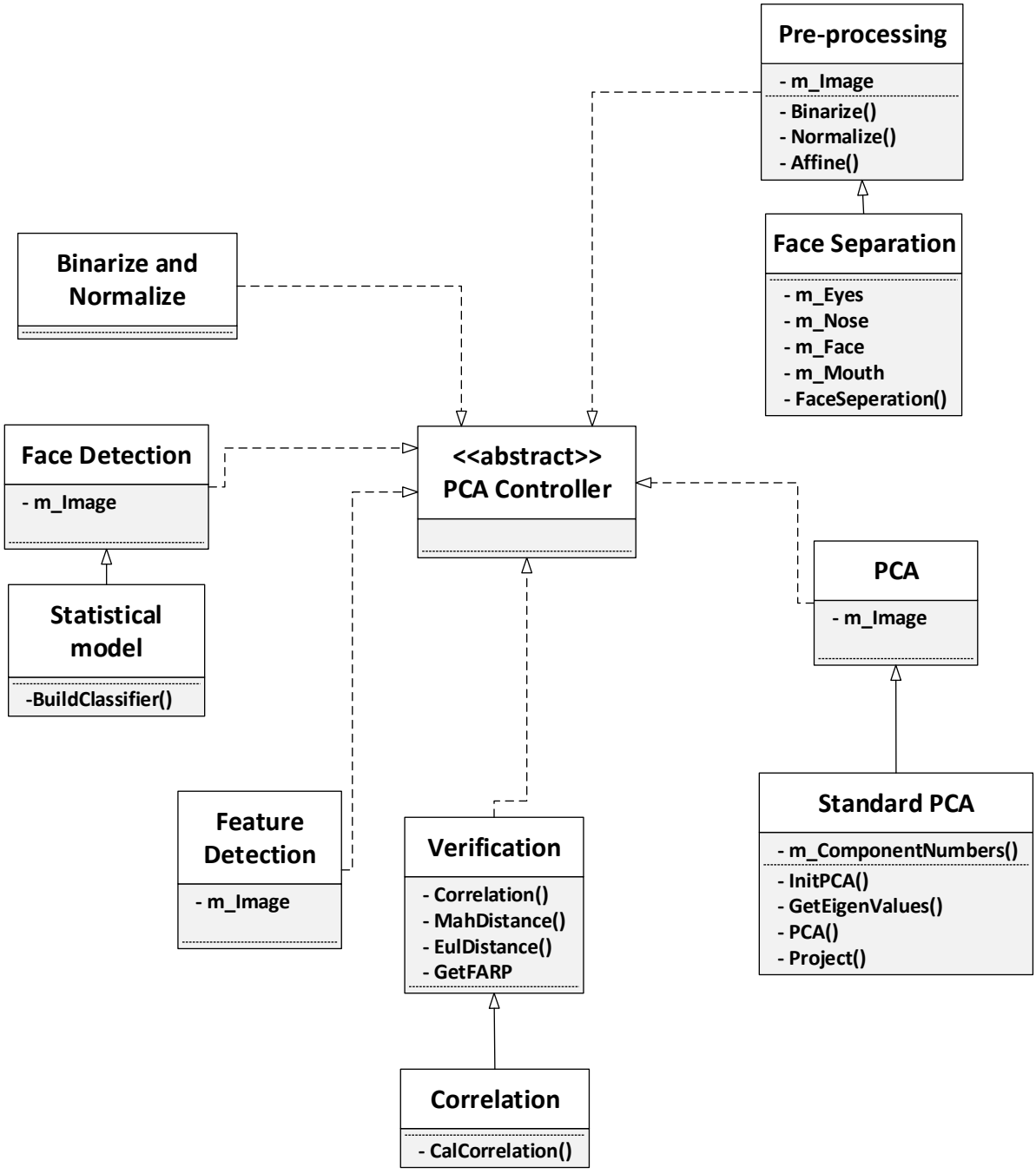

Figure 31 Face recognition for smart phones

4.1.3 Image Representation

As the computational capabilities of smartphones are still much lower than desktop PCs, the three face representation methods requiring relatively high computational resources do not work for smartphones.

Instead, two simple image processing approaches, which are binarization and normalization are used. They are not able to achieve the effect of the methods in our model in terms of noise reduction or image enhancement, but they do contribute to highlight the significant information of face region. The fast running speed of these two algorithms is the most attractive advantage when applied to smart phones. The code of binarization and normalization is shown next.

4.1.3.1 Binarization

The functions *cvLoadImage()* and *cvtColor()* are integrated in OpenCV.

```
tmpImg = cvLoadImage(c);
        tmpImgCopy = cvLoadImage(c);

        Mat matTmpImgCopy(tmpImgCopy,0);
        Mat frame_gray(matTmpImgCopy.rows, matTmpImgCopy.cols,
matTmpImgCopy.depth());
        cvtColor( matTmpImgCopy, frame_gray, CV_BGR2GRAY );
        tmpImgCopy2 = frame_gray;
```

4.1.3.2 Normalization

The function *normalize()*is integrated in OpenCV.

```
Mat CPPPCA::normalize(const Mat& src) {
   Mat srcnorm;
   cv::normalize(src, srcnorm, 0, 255, NORM_MINMAX, CV_8UC1);
   return srcnorm;

}       // end of void CPPPCA::normalize()
```

4.1.4 Face Detection

The face detection used in the system is based on a statistical model whose classifier is based on Haar-like features. The code is shown as follows. The parameter *strClassifierFileName* reads in the file path of the classifier file. Then function *detect_and_draw()*configures the classifier.

```
BOOL CPPPCA::FaceProc(IplImage* imgSource)
{
        const char* strClassifierFileName = "";
        static CvHaarClassifierCascade* cascade = 0;
```

```
        cascade = (CvHaarClassifierCascade*)cvLoad(strClassifierFileName);

        CvMemStorage* storage = cvCreateMemStorage(0);
        detect_and_draw(imgSource, storage, cascade);

        if(m_faces->total != 0){
                m_FaceRect.x = (*(CvRect*)cvGetSeqElem( m_faces, 0)).x;
                m_FaceRect.y = (*(CvRect*)cvGetSeqElem( m_faces, 0)).y;
                m_FaceRect.width = (*(CvRect*)cvGetSeqElem( m_faces, 0)).width;
                m_FaceRect.height = (*(CvRect*)cvGetSeqElem( m_faces, 0)).height;

                cvClearMemStorage( storage );

                return TRUE;
        }
        else
                return FALSE;

}       // end of void CPPPCA::FaceProc()

void CPPPCA::detect_and_draw( IplImage* img, CvMemStorage* storage, CvHaarClassifierCascade*
cascade )
{
   static CvScalar colors[] =
   {
      {{0,0,255}},
      {{0,128,255}},
      {{0,255,255}},
      {{0,255,0}},
      {{255,128,0}},
      {{255,255,0}},
```

```
    {{255,0,0}},
    {{255,0,255}}
};

double scale = 1;
IplImage* gray = cvCreateImage( cvSize(img->width,img->height), 8, 1 );
IplImage* small_img = cvCreateImage( cvSize( cvRound (img->width/scale),
cvRound (img->height/scale)),
8, 1 );
int i;

cvResize( img, small_img, CV_INTER_LINEAR );
cvClearMemStorage( storage );

if( cascade )
{
    double t = (double)cvGetTickCount();
                m_faces = cvHaarDetectObjects( small_img, cascade, storage,
    1.1, 2, 0
    cvSize(30, 30) );

    t = (double)cvGetTickCount() - t;
    printf( "detection time = %gms\n", t/((double)cvGetTickFrequency()*1000.) );
    for( i = 0; i < (m_faces ? m_faces->total : 0); i++ )
    {
        CvRect* r = (CvRect*)cvGetSeqElem( m_faces, i );
        CvPoint center;
        int radius;
        center.x = cvRound((r->x + r->width*0.5)*scale);
        center.y = cvRound((r->y + r->height*0.5)*scale);
        radius = cvRound((r->width + r->height)*0.25*scale);
    }
```

```
    }

    cvReleaseImage( &gray );
    cvReleaseImage( &small_img );

}        // end of void CPPPCA::detect_and_draw()
```

4.1.5 Pre-processing

In the context of smart phones, both facial expressions and illumination need to be considered. Therefore, we decide to combine face separation aiming to solve expression problems with LBP which targets illumination problems to achieve better results. The code is next.

4.1.5.1 Face Separation

The four *Rect* type parameters *roi*, *eyesRoi*, *mouthRoi*, *NoseRoi*, define the face area, eyes area, mouth area, and nose area respectively.

```
Mat warp_dst = GetAffinedMat(tmpImgCopy2, frame_gray);

IplImage *imgLbpSrc = (&(IplImage)warp_dst);
IplImage *imgLbpDst = cvCreateImage(cvGetSize(imgLbpSrc), IPL_DEPTH_8U,1);;

m_lbpInst.CreatLBP(imgLbpSrc, imgLbpDst);

Mat lbp_dst(imgLbpDst);

Rect roi(51, 19, 80, 89);
Mat matRoi = lbp_dst(roi);
m_vmatLbpFace.push_back(matRoi);

Rect eyesRoi(46, 22, 84, 38);
Mat matEyes = lbp_dst(eyesRoi);
m_vmatLbpEyes.push_back(matEyes);
```

```
Rect mouthRoi(69 ,102, 38, 22);
Mat matMouth = lbp_dst(mouthRoi);
m_vmatLbpMouth.push_back(matMouth);

Rect NoseRoi(71, 57, 39, 37);
Mat matNose = lbp_dst(NoseRoi);
m_vmatLbpNose.push_back(matNose);
```

4.1.5.2 LBP

```
void CLBP::CreatLBP(IplImage *src,IplImage *dst)
{
	int iTemp[8] = {0};
	CvScalar s;

	IplImage *ImgTemp = cvCreateImage(cvGetSize(src), IPL_DEPTH_8U, 1);
	uchar *data = (uchar*)src->imageData;
	int iStep = src-> widthStep;

	for (int i=1;i<src->height-1;i++)
		for(int j=1;j<src->width-1;j++){

			int sum=0;
			if(data[(i-1)*iStep+j-1]>data[i*iStep+j])
				iTemp[0]=1;
			else
				iTemp[0]=0;
			if(data[i*iStep+(j-1)]>data[i*iStep+j])
				iTemp[1]=1;
			else
				iTemp[1]=0;
			if(data[(i+1)*iStep+(j-1)]>data[i*iStep+j])
				iTemp[2]=1;
```

```
                    else
                        iTemp[2]=0;
                    if (data[(i+1)*iStep+j]>data[i*iStep+j])
                        iTemp[3]=1;
                    else
                        iTemp[3]=0;
                    if (data[(i+1)*iStep+(j+1)]>data[i*iStep+j])
                        iTemp[4]=1;
                    else
                        iTemp[4]=0;
                    if(data[i*iStep+(j+1)]>data[i*iStep+j])
                        iTemp[5]=1;
                    else
                        iTemp[5]=0;
                    if(data[(i-1)*iStep+(j+1)]>data[i*iStep+j])
                        iTemp[6]=1;
                    else
                        iTemp[6]=0;
                    if(data[(i-1)*iStep+j]>data[i*iStep+j])
                        iTemp[7]=1;
                    else
                        iTemp[7]=0;
                    s.val[0] =
(iTemp[0]*1+iTemp[1]*2+iTemp[2]*4+iTemp[3]*8+iTemp[4]*16+iTemp[5]*32+iTemp[6]*64+iTemp[7]*128);

                    cvSet2D(dst,i,j,s);

                }

}       // end of CLBP::CreatLBP()
```

4.1.6 PCA

To achieve optimal performance, kernel PCA always requires a relatively long training time and large training dataset. Nevertheless, smart phone users expect the phone to respond in real-time and the storage capacity of smart phones is also limited. Therefore, we select the simpler standard PCA for the system. The code is shown as follows. The function *InitPCA()* sets tup the PCA instance for further use. The function *GetEigenValues()* obtains the eigenvalue of a corresponding eigenvector.

```
void CPPPCA::InitPCA(String strDirName, Mat matSrc, vector<Mat> vmatSrc)
{
        double dbnumber_principal_compent = 0.95;
        Mat pcaImg, projectImg;

        PCA pca(matSrc, Mat(), CV_PCA_DATA_AS_COL, dbnumber_principal_compent);
        Mat EigenVectors = pca.eigenvectors;

        pcaImg = normalize(pca.eigenvectors.row(0)).reshape(1, vmatSrc[0].rows);
//      pcaFace = pca.eigenvectors.row(0).reshape(1, src[0].rows);
        imwrite(((".\\") + strDirName + (".\\Pca.jpg")), pcaImg);

        Mat EigenValues = pca.eigenvalues;

        Mat dst;
        dst = pca.project(matSrc.col(0));
        projectImg = normalize(pca.backProject(dst).col(0)).reshape(1, vmatSrc[0].rows);
        imwrite(((".") + strDirName + (".\\Project.jpg")), projectImg);

        m_pcaTrain = pca;

}       // end of CPPPCA::InitPCA()

void CPPPCA::GetEigenValues(String strDirName, Mat matSrc, vector<Mat> vmatSrc, int iImgNum)
{
        string strInt;
```

```
        Mat dst, projectImg;
        const string strEigenValue = "D:\\Summer Project\\EigenValue\\FaceEigenValues.xml";
        FileStorage fs(strEigenValue, FileStorage::WRITE);

        for(int i = 0; i < iImgNum; i++){
                strInt = inttostring(i);
                dst = m_pcaTrain.project(matSrc.col(i));

//              projectImg = normalize(m_pcaTrain.backProject(dst).col(0)).reshape(1, vmatSrc[0].rows);
//              imwrite((("D:\\Summer Project\\") + strDirName + ("\\Project") + strInt + (".jpg")),
projectImg);
                fs << "eigenvalue" + strInt << dst;
        }
        fs.release();

}       // end of CPPPCA::GetEigenValues()
```

4.1.7 Verification

Among the three verification methods proposed in the model, Mahalanobis distance reflects most similiarity between two images, whereas Euclidean distance uses the least computing resources. For a smart phone environment, we choose correlation as it can be regarded as the compromise which considers not only accuracy, but computational complexity. The code is shown as follows.

```
double CPPPCA::GetCorelation(uchar *v1, uchar *v2, int n)
{
  double dSignal = 0, dNoise = 0;
  int i;
  double e1 = 0, e2 = 0, e12 = 0, c1 = 0, c2 = 0;

  //calculate the expectations of V1, V2 and V1 x V2
  for (i = 0; i < n; i++){
        e1  += v1[i];
        e2  += v2[i];
        e12 += v1[i] * v2[i];
```

```
  }

  e1  = e1  / n;
  e2  = e2  / n;
  e12 = e12 / n;

  //calulate the variances of R1 and R2
  for (i = 0; i < n; i++){
        c1 += (e1 - v1[i]) * (e1 - v1[i]);
        c2 += (e2 - v2[i]) * (e2 - v2[i]);
  }

  c1 = sqrt(c1 / n);
  c2 = sqrt(c2 / n);

  //calulate the correlation
  return IsZero(c1) || IsZero(c2)? 0.0 : fabs((e12 - e1 * e2) / (c1 * c2));

}         //end of CPPPCA::GetCorelation()
```

4.2 Case study 2

4.2.1 Description

In this case study, we intend to select the variations which are not used in the first case study to offer a comprehensive introduction to the model. For face representation, Gabor Wavelet is chosen to extract more precise facial features. To detect face region, a neural network method is used, as its detection accuracy outperforms the other two in the model, if we temporally ignore the computation speed. Similar to the first case study, feature detection is also required for aligning the image via affine transformation, since the alignment of image is important to most PCA-based approaches. Then we skip pre-processing steps, as compared with the standard PCA, kernel PCA is capable of dealing with more complex data (non-linear), so pre-processing might be redundant in this case. At last, Mahalanobis distance is used for

verification. The overview of the process is shown in Figure 32. The implementation of the variations selected for this case study is written in C++.

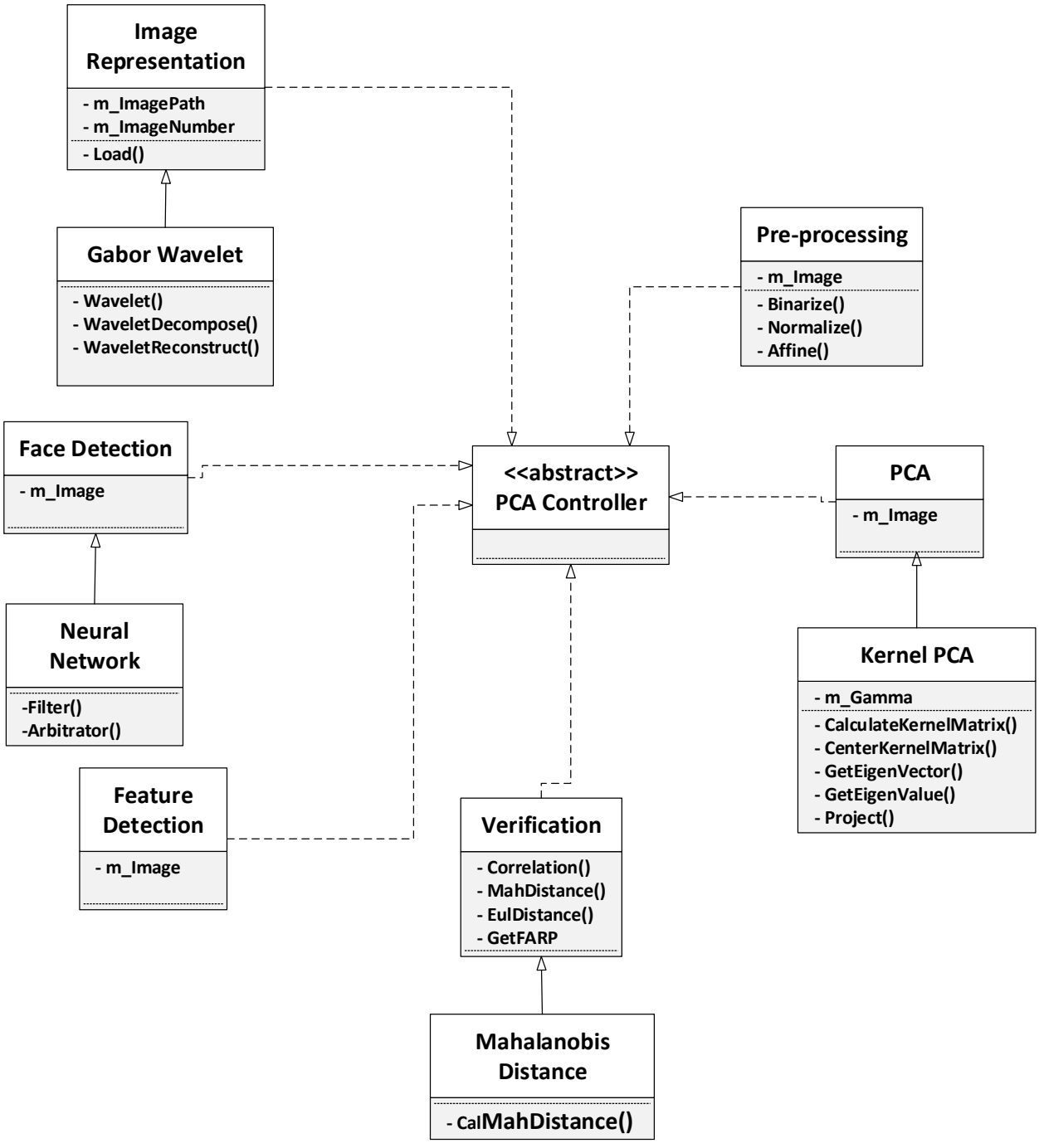

Figure 32 Case study 2

4.2.2 Face Representation

For face recognition systems running on PCs with high-end configuration, the speed of executing algorithms can be ignored, to some extent. Therefore, approaches producing more precise result while costing more computational resources can be used. Therefore, Gabor Wavelet is selected for face representation in this case study, as it is the most complex method, compared to the others in the model, but extracts most useful facial features.

Gabor Wavelet transfers image data from the spatial domain to frequency domain, so it is capable of dealing with noise such as illumination, shooting angle, or occlusion. Moreover, when there are multiple faces appearing in the same image, Gabor Wavelet is sensitive to the distinct features on different faces, which facilitates the later recognition process.

The code to implementing Gabor Wavelet is shown as follows [42]. The function *WDT()* implements Wavelet transformation. The function *IWDT()* implements inverse Wavelet transformation. The function *Wavelet()* generates different types of Wavelet. Currently only haar and sym2 Wavelet are implemented. The function *WaveletDecompose()* implements Wavelet decomposition. The function *WaveletReconstruct()*implements Wavelet reconstruction.

```
Mat WDT( const Mat &_src, const string _wname, const int _level )const
{
    int reValue = THID_ERR_NONE;
    Mat src = Mat_<float>(_src);
    Mat dst = Mat::zeros( src.rows, src.cols, src.type() );
    int N = src.rows;
    int D = src.cols;

    Mat lowFilter;
    Mat highFilter;
    wavelet( _wname, lowFilter, highFilter );

    int t=1;
    int row = N;
    int col = D;

    while( t<=_level )
```

```
    {
        for( int i=0; i<row; i++ )
        {
            Mat oneRow = Mat::zeros( 1,col, src.type() );
            for ( int j=0; j<col; j++ )
            {
                oneRow.at<float>(0,j) = src.at<float>(i,j);
            }
            oneRow = waveletDecompose( oneRow, lowFilter, highFilter );
            for ( int j=0; j<col; j++ )
            {
                dst.at<float>(i,j) = oneRow.at<float>(0,j);
            }
        }

#if 0
        //normalize( dst, dst, 0, 255, NORM_MINMAX );
        IplImage dstImg1 = IplImage(dst);
        cvSaveImage( "dst.jpg", &dstImg1 );
#endif
        for ( int j=0; j<col; j++ )
        {
            Mat oneCol = Mat::zeros( row, 1, src.type() );
            for ( int i=0; i<row; i++ )
            {
                oneCol.at<float>(i,0) = dst.at<float>(i,j);
            }
            oneCol = ( waveletDecompose( oneCol.t(), lowFilter, highFilter ) ).t();

            for ( int i=0; i<row; i++ )
            {
                dst.at<float>(i,j) = oneCol.at<float>(i,0);
```

```
            }
        }

#if 0
        //normalize( dst, dst, 0, 255, NORM_MINMAX );
        IplImage dstImg2 = IplImage(dst);
        cvSaveImage( "dst.jpg", &dstImg2 );
#endif

        row /= 2;
        col /=2;
        t++;
        src = dst;
    }

    return dst;
}
Mat IWDT( const Mat &_src, const string _wname, const int _level )const
{
    int reValue = THID_ERR_NONE;
    Mat src = Mat_<float>(_src);
    Mat dst = Mat::zeros( src.rows, src.cols, src.type() );
    int N = src.rows;
    int D = src.cols;

    Mat lowFilter;
    Mat highFilter;
    wavelet( _wname, lowFilter, highFilter );

    int t=1;
    int row = N/std::pow( 2., _level-1);
    int col = D/std::pow(2., _level-1);
```

```
    while ( row<=N && col<=D )
    {
        for ( int j=0; j<col; j++ )
        {
            Mat oneCol = Mat::zeros( row, 1, src.type() );
            for ( int i=0; i<row; i++ )
            {
                oneCol.at<float>(i,0) = src.at<float>(i,j);
            }
            oneCol = ( waveletReconstruct( oneCol.t(), lowFilter, highFilter ) ).t();

            for ( int i=0; i<row; i++ )
            {
                dst.at<float>(i,j) = oneCol.at<float>(i,0);
            }
        }

#if 0
        //normalize( dst, dst, 0, 255, NORM_MINMAX );
        IplImage dstImg2 = IplImage(dst);
        cvSaveImage( "dst.jpg", &dstImg2 );
#endif
        for( int i=0; i<row; i++ )
        {
            Mat oneRow = Mat::zeros( 1,col, src.type() );
            for ( int j=0; j<col; j++ )
            {
                oneRow.at<float>(0,j) = dst.at<float>(i,j);
            }
            oneRow = waveletReconstruct( oneRow, lowFilter, highFilter );
            for ( int j=0; j<col; j++ )
```

```
            {
                dst.at<float>(i,j) = oneRow.at<float>(0,j);
            }
        }

#if 0
        //normalize( dst, dst, 0, 255, NORM_MINMAX );
        IplImage dstImg1 = IplImage(dst);
        cvSaveImage( "dst.jpg", &dstImg1 );
#endif

        row *= 2;
        col *= 2;
        src = dst;
    }

    return dst;
}
void wavelet( const string _wname, Mat &_lowFilter, Mat &_highFilter )const
{
    if ( _wname=="haar" || _wname=="db1" )
    {
        int N = 2;
        _lowFilter = Mat::zeros( 1, N, CV_32F );
        _highFilter = Mat::zeros( 1, N, CV_32F );

        _lowFilter.at<float>(0, 0) = 1/sqrtf(N);
        _lowFilter.at<float>(0, 1) = 1/sqrtf(N);

        _highFilter.at<float>(0, 0) = -1/sqrtf(N);
        _highFilter.at<float>(0, 1) = 1/sqrtf(N);
    }
```

```
    if ( _wname =="sym2" )
    {
        int N = 4;
        float h[] = {-0.483, 0.836, -0.224, -0.129 };
        float l[] = {-0.129, 0.224,    0.837, 0.483 };

        _lowFilter = Mat::zeros( 1, N, CV_32F );
        _highFilter = Mat::zeros( 1, N, CV_32F );

        for ( int i=0; i<N; i++ )
        {
            _lowFilter.at<float>(0, i) = l[i];
            _highFilter.at<float>(0, i) = h[i];
        }

    }
}
Mat waveletDecompose( const Mat &_src, const Mat &_lowFilter, const Mat &_highFilter )const
{
    assert( _src.rows==1 && _lowFilter.rows==1 && _highFilter.rows==1 );
    assert( _src.cols>=_lowFilter.cols && _src.cols>=_highFilter.cols );
    Mat &src = Mat_<float>(_src);

    int D = src.cols;

    Mat &lowFilter = Mat_<float>(_lowFilter);
    Mat &highFilter = Mat_<float>(_highFilter);

    Mat dst1 = Mat::zeros( 1, D, src.type() );
    Mat dst2 = Mat::zeros( 1, D, src.type()  );

    filter2D( src, dst1, -1, lowFilter );
```

```
    filter2D( src, dst2, -1, highFilter );

    Mat downDst1 = Mat::zeros( 1, D/2, src.type() );
    Mat downDst2 = Mat::zeros( 1, D/2, src.type() );

    resize( dst1, downDst1, downDst1.size() );
    resize( dst2, downDst2, downDst2.size() );

    for ( int i=0; i<D/2; i++ )
    {
        src.at<float>(0, i) = downDst1.at<float>( 0, i );
        src.at<float>(0, i+D/2) = downDst2.at<float>( 0, i );
    }

    return src;
}
Mat waveletReconstruct( const Mat &_src, const Mat &_lowFilter, const Mat &_highFilter )const
{
    assert( _src.rows==1 && _lowFilter.rows==1 && _highFilter.rows==1 );
    assert( _src.cols>=_lowFilter.cols && _src.cols>=_highFilter.cols );
    Mat &src = Mat_<float>(_src);

    int D = src.cols;

    Mat &lowFilter = Mat_<float>(_lowFilter);
    Mat &highFilter = Mat_<float>(_highFilter);

    Mat Up1 = Mat::zeros( 1, D, src.type() );
    Mat Up2 = Mat::zeros( 1, D, src.type() );

    Mat roi1( src, Rect(0, 0, D/2, 1) );
    Mat roi2( src, Rect(D/2, 0, D/2, 1) );
```

```
    resize( roi1, Up1, Up1.size(), 0, 0, INTER_CUBIC );
    resize( roi2, Up2, Up2.size(), 0, 0, INTER_CUBIC );

    Mat dst1 = Mat::zeros( 1, D, src.type() );
    Mat dst2= Mat::zeros( 1, D, src.type() );
    filter2D( Up1, dst1, -1, lowFilter );
    filter2D( Up2, dst2, -1, highFilter );

    dst1 = dst1 + dst2;

    return dst1;
}
```

4.2.3 Face Detection

The neural network-based face-detection method actually belongs to a statistics model-based methods, since it also trains the network by inputing images, which means the more images it tests, the more accurate it becomes. However, as neural network origins from biological knowledge, and its principle and detecting process differs significantly from traditional statistics model-based detection methods, it is always classified in an independent category in face recognition.

Similar to Gabor Wavelet for face representation, neural network-based face detection also costs high computation resources. Therefore, it is suitable for high-end platforms or systems which do not require real-time recognition but has to guarantee high accuracy, such as face recognition system used by the military. Moreover, the neural network-based method is extendable, since the accuracy level can be adjusted by changing the number of layers in the network.

The following code shows a simple BP neural network implementation [41]. The parameter *params* contains all set up information of the BP network. The parameter *layerSizes* stands for the number of layers in the network. Here, the network has three hidden layers with an input layer and an output layer. The function *predict()* enables predicting new nodes in the network.

```
int main()
{
```

```
    CvANN_MLP bp;
CvANN_MLP_TrainParams params;

    params.train_method=CvANN_MLP_TrainParams::BACKPROP;
    params.bp_dw_scale=0.1;
    params.bp_moment_scale=0.1;
    float labels[3][5] = {{0,0,0,0,0},{1,1,1,1,1},{0,0,0,0,0}};
    Mat labelsMat(3, 5, CV_32FC1, labels);

    float trainingData[3][5] = { {1,2,3,4,5},{111,112,113,114,115}, {21,22,23,24,25} };
    Mat trainingDataMat(3, 5, CV_32FC1, trainingData);
    Mat layerSizes=(Mat_<int>(1,5) << 5,2,2,2,5);
    bp.create(layerSizes,CvANN_MLP::SIGMOID_SYM);//CvANN_MLP::SIGMOID_SYM
                                        //CvANN_MLP::GAUSSIAN
                                        //CvANN_MLP::IDENTITY
    bp.train(trainingDataMat, labelsMat, Mat(),Mat(), params);


    int width = 512, height = 512;
    Mat image = Mat::zeros(height, width, CV_8UC3);
    Vec3b green(0,255,0), blue (255,0,0);

    for (int i = 0; i < image.rows; ++i)
        for (int j = 0; j < image.cols; ++j)
        {
            Mat sampleMat = (Mat_<float>(1,5) << i,j,0,0,0);
            Mat responseMat;
            bp.predict(sampleMat,responseMat);
            float* p=responseMat.ptr<float>(0);
            float response=0.0f;
            for(int k=0;k<5;i++){
            //  cout<<p[k]<<" ";
```

```
            response+=p[k];
        }
        if (response >2)
            image.at<Vec3b>(j, i)  = green;
        else
            image.at<Vec3b>(j, i)  = blue;
    }

    // Show the training data
    int thickness = -1;
    int lineType = 8;
    circle( image, Point(501,  10), 5, Scalar(  0,   0,   0), thickness, lineType);
    circle( image, Point(255,  10), 5, Scalar(255, 255, 255), thickness, lineType);
    circle( image, Point(501, 255), 5, Scalar(255, 255, 255), thickness, lineType);
    circle( image, Point( 10, 501), 5, Scalar(255, 255, 255), thickness, lineType);

    imwrite("result.png", image);        // save the image

    imshow("BP Simple Example", image); // show it to the user
    waitKey(0);

}
```

4.2.4 PCA

Standard PCA has been proved to be an efficient tool for face recognition, which produces high recognition accuracy and executes quickly. For systems requiring quick response, standard PCA is a good choice. However, there are still some factors which are ignored by standard PCA, such as the non-linear information contained in image data.

Kernel PCA is an extension of PCA using a kernel technique which takes the non-linear information into account. In fact, the non-linear information plays an important role in image data, such as the influence of wearing glasses or having eyes are closed or opened. In most face dataset for research experiments, the face images are still taken with limitations. Nevertheless, in practical applications, such

as criminal recognition or scene surveillance, the shooting environment might be much worse. In this case, the non-linear component in the image data increases exponentially.

Therefore, in this case study, we select kernel PCA. The code to implement kernel PCA is as follows. The function *kernel()* works as a switch of two types of mainstream kernels, which are RBF and Polynomial. The function *run_pca()* implements all main steps of kernel PCA which are calculating the kernel matrix, centering the kernel matrix, and obtaining eigenvectors and eigenvalues.

```
void PCA::load_data(const char* data, char sep){

    unsigned int row = 0;
    ifstream file(data);
    if(file.is_open()){
        string line,token;
        while(getline(file, line)){
            stringstream tmp(line);
            unsigned int col = 0;
            while(getline(tmp, token, sep)){
                if(X.rows() < row+1){
                    X.conservativeResize(row+1,X.cols());
                }
                if(X.cols() < col+1){
                    X.conservativeResize(X.rows(),col+1);
                }
                X(row,col) = atof(token.c_str());
                col++;
            }
            row++;
        }
        file.close();
        Xcentered.resize(X.rows(),X.cols());
    }else{
        cout << "Failed to read file " << data << endl;
    }
```

```
}

double PCA::kernel(const VectorXd& a, const VectorXd& b){


        switch(kernel_type){
           case 2  :
                return(pow(a.dot(b)+constant,order));
           default :
                return(exp(-gamma*((a-b).squaredNorm())));
        }

}

void PCA::run_kpca(){


        K.resize(X.rows(),X.rows());
        for(unsigned int i = 0; i < X.rows(); i++){
                for(unsigned int j = i; j < X.rows(); j++){
                        K(i,j) = K(j,i) = kernel(X.row(i),X.row(j));
                        //printf("k(%i,%i) = %f\n",i,j,K(i,j));
                }
        }
        EigenSolver<MatrixXd> edecomp(K);
        eigenvalues = edecomp.eigenvalues().real();
        eigenvectors = edecomp.eigenvectors().real();
        cumulative.resize(eigenvalues.rows());
        vector<pair<double,VectorXd> > eigen_pairs;
        double c = 0.0;
        for(unsigned int i = 0; i < eigenvectors.cols(); i++){
```

```
            if(normalise){
                double norm = eigenvectors.col(i).norm();
                eigenvectors.col(i) /= norm;
            }
            eigen_pairs.push_back(make_pair(eigenvalues(i),eigenvectors.col(i)));
        }
        sort(eigen_pairs.begin(),eigen_pairs.end(), [](const pair<double,VectorXd> a, const
pair<double,VectorXd> b) -> bool {return (a.first > b.first);} );
        for(unsigned int i = 0; i < eigen_pairs.size(); i++){
            eigenvalues(i) = eigen_pairs[i].first;
            c += eigenvalues(i);
            cumulative(i) = c;
            eigenvectors.col(i) = eigen_pairs[i].second;
        }
        transformed.resize(X.rows(),components);

        for(unsigned int i = 0; i < X.rows(); i++){
            for(unsigned int j = 0; j < components; j++){
                for (int k = 0; k < K.rows(); k++){
          transformed(i,j) += K(i,k) * eigenvectors(k,j);
                }
            }
        }
        cout << "Sorted eigenvalues:" << endl;
        for(unsigned int i = 0; i < eigenvalues.rows(); i++){
            if(eigenvalues(i) > 0){
                cout << "PC " << i+1 << ": Eigenvalue: " << eigenvalues(i);
                printf("\t(%3.3f of variance, cumulative =
%3.3f)\n",eigenvalues(i)/eigenvalues.sum(),cumulative(i)/eigenvalues.sum());
            }
        }
        cout << endl;
```

```
        //cout << "Sorted eigenvectors:" << endl << eigenvectors << endl << endl;
        //cout << "Transformed data:" << endl << transformed << endl << endl;
}


void PCA::print(){

        cout << "Input data:" << endl << X << endl << endl;
        cout << "Centered data:"<< endl << Xcentered << endl << endl;
        cout << "Covariance matrix:" << endl << C << endl << endl;
        cout << "Eigenvalues:" << endl << eigenvalues << endl << endl;
        cout << "Eigenvectors:" << endl << eigenvectors << endl << endl;
        cout << "Sorted eigenvalues:" << endl;
        for(unsigned int i = 0; i < eigenvalues.rows(); i++){
                if(eigenvalues(i) > 0){
                        cout << "PC " << i+1 << ": Eigenvalue: " << eigenvalues(i);
                        printf("\t(%3.3f of variance, cumulative =
%3.3f)\n",eigenvalues(i)/eigenvalues.sum(),cumulative(i)/eigenvalues.sum());
                }
        }
        cout << endl;
        cout << "Sorted eigenvectors:" << endl << eigenvectors << endl << endl;
        cout << "Transformed data:" << endl << X * eigenvectors << endl << endl;
        //cout << "Transformed centred data:" << endl << transformed << endl << endl;

}

void PCA::write_transformed(string file){

        ofstream outfile(file);
        for(unsigned int i = 0; i < transformed.rows(); i++){
                for(unsigned int j = 0; j < transformed.cols(); j++){
```

```
                outfile << transformed(i,j);
                if(j != transformed.cols()-1) outfile << ",";
            }
            outfile << endl;
        }
        outfile.close();
        cout << "Written file " << file << endl;

}

void PCA::write_eigenvectors(string file){

        ofstream outfile(file);
        for(unsigned int i = 0; i < eigenvectors.rows(); i++){
            for(unsigned int j = 0; j < eigenvectors.cols(); j++){
                outfile << eigenvectors(i,j);
                if(j != eigenvectors.cols()-1) outfile << ",";
            }
            outfile << endl;
        }
        outfile.close();
        cout << "Written file " << file << endl;

}
```

4.2.5 Verification

Among the three variations in the verification step in the model, Mahalanobis distance is the only measurement that uses a covariance matrix between two data vectors. Therefore, it is more complicated to calculate, but reflects the relationship between different dimensions of the data, which is important when comparing images. In face recognition, applying Mahalanobis distance to the final verification step helps the system choose a more explicit threshold.

The code for calculating Mahalanobis distance is specified as follows. The function *calcCovarMatrix()* calculates the covariance matrix between two vectors. The calculation function *cvMahalonobis()* is integrated in OpenCV.

```
double CPPPCA::CalMahDistance(Mat matSrc, Mat matTest)
{
        Mat matCovar, matMean;
        CvMat cvmatSrc, cvmatTest, cvmatCovar;
        cvmatSrc = matSrc;
        cvmatTest = matTest;

        calcCovarMatrix(&matSrc, 1, matCovar, matMean, CV_COVAR_NORMAL);
        cvmatCovar = matCovar;

        double dbMahDistance = cvMahalonobis(&cvmatSrc, &cvmatTest, &cvmatCovar);

        return dbMahDistance;

}       // end of CPPPCA::CalMahDistance()
```

4.3 Other Case Studies

In this section, two more case studies are presented. The case studies also show the entire work flow and the variations selected from the model as well as the situations in which the variations work well. The detailed implementation of the variations is not provided for these two case studies; however, the C++ code is given in the appendix. Still, aiming to show a comprehensive application of the model, these two case studies present the variations which are not used in previous case studies.

4.3.1 Case Study 3

4.3.1.1 Overview

In this case study, shape and texture approach is used for image representation and face detection. The feature detection step is required for the following face separation process. Standard PCA is employed as the core recognition approach. Finally, we use Euclidean distance for final verification. Compared with the system built in Case Study 1, this process uses relatively more time and computational resources

mainly because of the complexity of the shape and texture approach for face representation. Nevertheless, since the shape and texture approach defines the face region and depicts the face contour precisely, there is no need to detect the face region again, which saves some time. When compared with Case Study 2, this process does not spend time on training the neural network or performing kernel PCA. Though it is not able to achieve the accuracy of Case Study 2, it can be employed for platforms where real-time response is needed. Figure 33 shows the process overview.

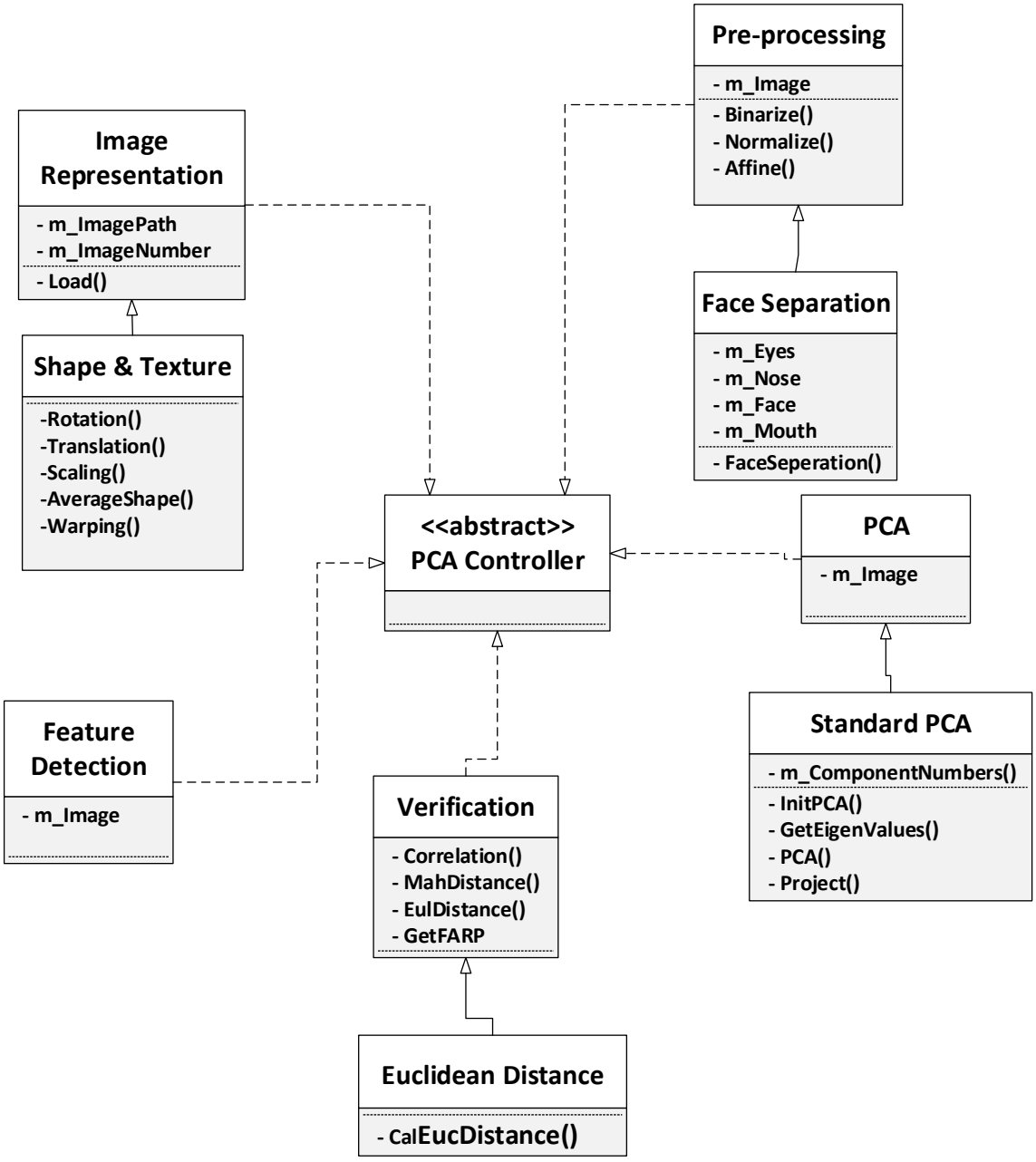

Figure 33 Case study 3

4.3.1.2 Description

The first step of the shape and texture approach is to obtain the geometry of the face, which is the shape. The shape is described by a set of manually annotated control points, so the noise of background image is removed in advance. This manual annotation process might be time-consuming; however, once the template is built, the remaining work to be done is just to align other images to the template through a series automatic transformations. Then texture, shape-free image can be generated using a warping transformation. After performing these two steps, a precise face region is catptured; and what is more important is that the face is described precisely without any noise like color, or illumination. This process might comsume more time than the other variations do, but it actually combines image representation and face detection, which makes it reasonable.

The pre-processing step and PCA performing step is the same as what happens in Case Study 1, so the details are not presented again. However, because of the precision of face representation provided by shape and texture, PCA is able to generate more precise result as well, though more time is needed as the features extracted by the shape and texture method are more complex.

The Euclidean distance is selected for the final verification step. It is the simplest measurement among all variations in the model and presents the most straightforward relationship between two images. Honestly, it does not provide as much information as the other methods; however, Euclidean distance always collaborates with mathematical operations, such as cosine. After combining together, Euclidean distance is able to increase fluctuation range, if needed, so that the threshold is easily selected.

4.3.2 Case Study 4

4.3.2.1 Overview

In this case study, we choose PCA as the image representation approach because of its ability to reduce data dimensions. Then a color-based method is used for detecting face region. We skip the feature detection and pre-processing steps; however, normalization and an affine transformation are required. Kernel PCA is employed as the recognition approach. Finally, Mahalanobis distance is used as verification method.

Although PCA is performed at the beginning of the process, which might cost more time than other variations, it saves time for the following steps, since PCA reduces the dimensionality of the original image. Color-based method detects face regions based on skin color distribution. The time consumption of this class of methods varies siginificantly depending on how the classifier is built. Therefore, it is flexible for different situations. Overall, this process relies highly on the training process, as most of the steps need training, and the more images are provided, the more robust the system. Therefore, it is

suitable for relatively fixed platforms, i.e., the database is set up in the back-end. The response time is short once the system is built and the training is done. The process overview is shown in Figure 34.

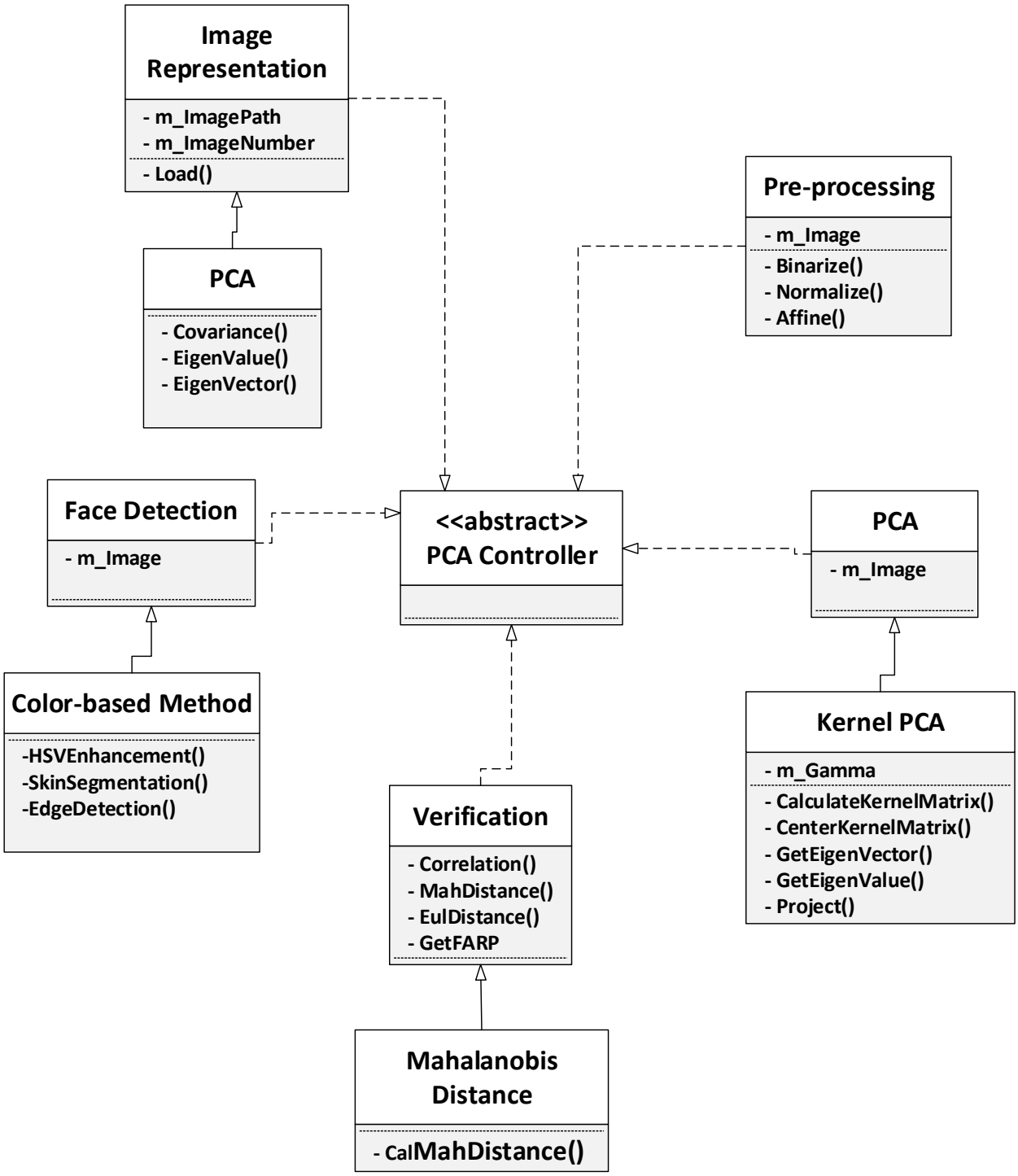

Figure 34 Case study 4

4.3.2.2 Description

Although PCA is the core step in our model for the recognition process, it can also be used as an image representation approach. After applying PCA to original images, the data size is substantially reduced while the important information is retained, i.e. the images are compressed without destruction. It undoubtedly facilitates the following process, as the original images are always too large. Furthermore, unlike the PCA employed during the recognition process, compressing images with PCA does not need to project the original images back to the sub-space, so its running time becomes reasonable.

A color-based method is initially inspired by the difference between skin color distribution and background color distribution. It defines a series of prior knowledge, such as black circle region implies pupil candidates, to detect the entire face region. As the statistical model rises in face recognition, researchers integrate color-based methods with a statistical model. Basically, they train the face color model, and compare the new image with the model to generate a detection result. Therefore, the running time of color-based methods varies depending on the complexity of the algorithm. Based on different requirements, color-based methods can be modified.

Normalization and an affine transformation are needed for image alignment. Kernel PCA is selected in this case study, since there is still some non-linear information which is not processed by image representation using PCA. Furthermore, as each step in this process relies on training, the kernel PCA training can be conducted in parallel which does not consume extra time.

4.4 Conclusion

In this chapter, four case studies are presented to describe how the model works and help users customize their own application. The first case study is for smart phones. The second case study aims to generate the most precise result with the variations of the model. The other two case studies, to some extent, compromise the strengthnesses and weaknesses of the first and second case studies. All variations for each step in the model are covered within these four case studies.

However, because of the length limitation of the thesis, it is impossible to present all variations which can be generated from the model. Actually, in practical applications, some steps are omitted, some variations are combined together, and some variations collaborate with other simple mathematical operations. Therefore, the model is able to help users generate a large number of applications based on their requirements.

Chapter 5 Conclusions and Future Work

5.1 Conclusion

PCA-based face recognition has been studied for decades. Some image processing toolkits like OpenCV have implemeted PCA algorithm and even its associated image processing approaches, which provide significant help for software developers in this field. However, setting up a PCA-based face recognition system is still time consuming, especially when adapting to different types of image data, or fitting various situations, such as non-uniform illumination, exaggerated facial expression, or shooting angle. The existing tools can hardly help users quickly customize their own applications, since the requirement of different systems are quite variable. Searching for the implementation of an algorithm from the toolkit and integrating it with the current application can produce a lot of pain for developers. Therefore, a tool which can help developers establish their systems and select optimal approaches for each step in the process is critical.

The thesis presents a software engineering framework for PCA-based face recognition system. The framework describes the entire work flow of the system, and provides multiple variations for each step to fit different situations and help software developers customize their own applications. With the framework, developers are allowed to establish their system at a higher level, i.e the straightforward implemention details are handled by the framework.

The framework provides more than 150 supported variations in total for different situations and satisfying various requirements. There are at least two variations provided for each step in the process, so that developers can select the optimal one for their own purpose. Moreover, some of the variations can be combined to achieve better performance. The architecture of the framework is also flexible, which means some of the steps can be omitted when being applied to specific cases. Certainly, because of its flexibility, attaching more variations to the model is possible.

The faramework offers a significant help to software developers, non-expert researchers and domain experts in the field of face recognition, since the 150 supported variations produced by the framework cover a number of requirements for face recognition applications.

Inexperienced developers who are not familiar with face recognition can use the framework as a guide when they build applications, since the entire PCA-based face recognition process is described explicitly. They can learn from the framework and then modify or extend the basic process to meet their specific design requirements.

For non-expert researchers who are familiar with the process of PCA-based face recognition but do not have too much knowledge on specific techniques for each step, the variations in the framework helps significantly. The properties of most variations are demonstrated in the Case Study Chapter, which can be used as a guide to assist the non-expert researchers to select the optimal approach for particular requirements.

For domain experts who are experienced in face recognition, designing the structure of an application or selecting the best approach for each step is not the major problem. However, implementing the application is time-consuming. In this case, the framework provides the complete implementation for each variation, which saves time for domain experts.

As an example, when mobile phone application developers build a face recognition application for smart phones for the first time, i.e., they are inexperienced in this field, the major problem is the lack of domain knowledge. In this case, the framework is able to give them a straightforward guide about PCA-based face recognition which can inspire them so that they can easily start implementing. The variations will also assist them throughout the implementing process.

Nevertheless, for non-expert researchers who want to build a face recognition application used for security, the major problem becomes selecting the best approaches to achieve the optimal recognition accuracy. In this case, with the demonstration of each variation, the reasearchers can find a variation of the framework that best matches their application. Morover, for both examples, the implementation details are handled by the framework, which significantly improves efficiency.

The thesis presents four case studies which cover some of the variations. The case studies intend to offer a straightforward impression on how the framework can help developers establish their applications. However, the framework is capable of dealing with many more complicated situations than shown in the case studies.

5.2 Future work

The framework proposed in the thesis provides a prototype or starting point of our thinking. There is some potential future work involving the implementation of the framework, the enhancement of the framework and practical case studies, which can be considered.

First, the implementation of the framework could provide a friendly user interface in which all variations proposed in the framework are modularized so that, developers can build their systems by simply dragging the variations and connecting them with lines. Furthermore, the interface will not only reduce implementation time for developers, but also help them select the optimal approaches to achieve best result.

Second, with the development of face recognition field, more advanced techniques are proposed, which might outperform the algorithms that we have already included in the framework. Therefore, studying new techniques and integrating them with our framework would be meaningful and help us improve the comprehensiveness of the framework. Certainly, it is not enough to just add new variations to the model. Classifying those variations by their distinct functions is more important, as users would then be able to choose the variations that they need for fitting their own applications.

Third, the framework can be provided an architectural design based on a layered architecture. The architecture may contain three layers, which can include (i) data acquisition, (ii) data processing, and (iii) face image classification and decision-making. An architectural design would emphasize the data flow and show more about how the framework works in terms of its components.

Last, conducting case studies in the practical context can help us verify the efficientness and usefulness of the framework and detect potential defects.

References

[1] Vinay, A., Shekhar, V., Rituparna, J., Aggrawal, T., Murthy, K., Natarajan, S., "Cloud based big data analytics framework for face recognition in social networks using machine learning", Procedia Computer Science, vol. 50, pp. 623-630, 2015.

[2] Ricanek, K., Boehnen, C., "Facial analytics: from big data to law enforcement", Computer, vol. 45, no. 9, pp. 95-97, 2012.

[3] Simon, P., "Too big to ignore: the business case for big data", vol. 72. John Wiley & Sons, 2013.

[4] "Facial recognition system", Wikipedia, 2016. [Online]. Available: https://en.wikipedia.org/wiki/Facial_recognition_system. [Accessed: 22- Apr- 2016].

[5] Peng, Z., Ai, H., Hong, W., Liang, L., Xu, G., "Multi-cue-based face and facial feature detection on video segments", Journal of Computer Science and Technology, vol. 18, no. 2, pp. 241-246, 2003.

[6] Pearson, K., "LIII. On lines and planes of closest fit to systems of points in space", Philosophical Magazine Series 6, vol. 2, no. 11, pp. 559-572, 1901.

[7] Donato, G., Bartlett, M., Hager, J., Ekman, P., Sejnowski, T., "Classifying facial actions", IEEE Transactions on Pattern Analysis and Machine Intelligence, vol. 21, no. 10, pp. 974-989, 1999.

[8] Liu, C., Wechsler, H., "A shape- and texture-based enhanced fisher classifier for face recognition", IEEE Transactions on Image Processing, vol. 10, no. 4, pp. 598-608, 2001.

[9] Schneiderman, H., Kanade, T., "A statistical method for 3D object detection applied to faces and cars", Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR), pp. 746-751, 2000.

[10] Rowley, H., Baluja, S., Kanade, T., "Neural network-based face detection", IEEE Transactions on Pattern Analysis and Machine Intelligence, vol. 20, no. 1, pp. 23-38, 1998.

[11] Peng, P., Shen, Y., "Efficient face verification in mobile environment using component-based PCA", Proceedings of the 6th International Congress on Image and Signal Processing (CISP), pp. 753-757, 2013.

[12] He, D., Wang, L., "Texture unit, texture spectrum, and texture analysis", IEEE on Geoscience and Remote Sensing , vol. 28, no. 4, pp. 509-512, 1990.

[13] Ojala, T., Pietikainen, M., Maenpaa, T., "Multiresolution gray-scale and rotation invariant texture classification with local binary patterns", IEEE Transactions on Pattern Analysis and Machine Intelligence, vol. 24, no. 7, pp. 971-987, 2002.

[14] Tan, X., Triggs, B., "Enhanced local texture feature sets for face recognition under difficult lighting conditions", IEEE Transactions on Image Processing, vol. 19, no. 6, pp. 1635-1650, 2010.

[15] Raschka, S., "Kernel tricks and nonlinear dimensionality reduction via RBF kernel PCA", 2014. [Online]. Available: http://sebastianraschka.com/Articles/2014_kernel_pca.html. [Accessed: 22- Apr- 2016].

[16] Turk, M., Pentland, A., "Face recognition using eigenfaces", Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition, pp. 586-591, 1991.

[17] Hotelling, H., "Analysis of a complex of statistical variables into principal components", Journal of Educational Psychology, vol. 24, no. 6, pp. 417-441, 1933.

[18] Hotelling, H., "Relations between two sets of variates", Biometrika, vol. 28, no. 34, p. 321, 1936.

[19] Biometricchina.net, 2016. [Online]. Available: http://www.biometricchina.net/shownews.asp?id=63. [Accessed: 22- Apr- 2016].

[20] Russell, S., Norvig, P., "Artificial intelligence a modern approach", New Jersey: Prentice Hall, 2003.

[21] Yang, G., Huang, T., "Human face detection in a complex background", Pattern Recognition, vol. 27, no. 1, pp. 53-63, 1994.

[22] McCulloch, W., Pitts, W., "A logical calculus of the ideas immanent in nervous activity", Bulletin of Mathematical Biology, vol. 52, no. 1-2, pp. 99-115, 1990.

[23] Mattsson, M., "Object-oriented frameworks", Master's Thesis, Department of Computer and Business Administration, Technical Report LU-TS-CR:97-167, University College of Karlskrona/Ronneby, 1996.

[24] Nielsen, H., Brunak, S., von Heijne, G., "Machine learning approaches for the prediction of signal peptides and other protein sorting signals", Protein Engineering Design and Selection, vol. 12, no. 1, pp. 3-9, 1999.

[25] Alm, C., Roth, D., Sproat, R., "Emotions from text", Proceedings of the Conference on Human Language Technology and Empirical Methods in Natural Language Processing (HLT), pp. 579-586, 2005.

[26] "Forbes Welcome", Forbes.com, 2016. [Online]. Available: http://www.forbes.com/sites/85broads/2014/01/06/six-novel-machine-learning-applications/#2c29501d67bf. [Accessed: 22- Apr- 2016].

[27] Cnblogs.com, 2016. [Online]. Available: http://www.cnblogs.com/emouse/p/3611256.html. [Accessed: 22- Apr- 2016].

[28] Blog.csdn.net, 2016. [Online]. Available: http://blog.csdn.net/yanmy2012/article/details/8090400. [Accessed: 22- Apr- 2016].

[29] Liu, C., Wechsler, H., "Independent component analysis of gabor features for face recognition", IEEE Transactions on Neural Networks, vol. 14, no. 4, pp. 919-928, 2003.

[30] Watanabe, S., "Karhunen-Lotve expansion and factor analysis, theoretical remarks and applications", Proceedings of the 4th Prague Conference on Information Theory, 1965.

[31] Kirby, M., Sirovich, L., "Application of the Karhunen-Loeve procedure for the characterization of human faces", IEEE Transactions on Pattern Analysis and Machine Intelligence, vol. 12, no. 1, pp. 103-108, 1990.

[32] Zhang, D., Zhou, Z., "(2D)PCA: Two-directional two-dimensional PCA for efficient face representation and recognition", Neurocomputing, vol. 69, no. 1-3, pp. 224-231, 2005.

[33] Xiao, B., Gao, X., Tao, D., Li, X., "Biview face recognition in the shape–texture domain", Pattern Recognition, vol. 46, no. 7, pp. 1906-1919, 2013.

[34] Lanitis, A., Taylor, C., Cootes, T., "Automatic interpretation and coding of face images using flexible models", IEEE Transactions on Pattern Analysis and Machine Intelligence, vol. 19, no. 7, pp. 743-756, 1997.

[35] Vetter, T., Poggio, T., "Linear object classes and image synthesis from a single example image", IEEE Transactions on Pattern Analysis and Machine Intelligence, vol. 19, no. 7, pp. 733-742, 1997.

[36] Beymer, D., "Vectorizing face images by interleaving shape and texture computations", Technical Report, Artificial Intelligence Laboratory, Massachusetts Institute of Technology, 1995.

[37] Moghaddam, B., Pentland, A., "Probabilistic visual learning for object representation", IEEE Transactions on Pattern Analysis and Machine Intelligence, vol. 19, no. 7, pp. 696-710, 1997.

[38] Sahbi, H., Boujemaa, N., "Coarse to fine face detection based on skin color adaption", Biometric Authentication, pp. 112-120, 2002.

[39] Raschka, Sebastian, "Kernel tricks and nonlinear dimensionality reduction via RBF kernel PCA", 2014. [Online]. Available: http://sebastianraschka.com/Articles/2014_kernel_pca.html. [Accessed: 22- Apr- 2016].

[40] "Google Code Archive - Long-term storage for Google Code Project Hosting", Code.google.com, 2016. [Online]. Available: https://code.google.com/archive/p/lsa-lda/downloads. [Accessed: 22- Apr- 2016].

[41] Blog.csdn.net, 2016. [Online]. Available: http://blog.csdn.net/xiaowei_cqu/article/details/9027617. [Accessed: 22- Apr- 2016].

[42] Blog.csdn.net, 2016. [Online]. Available: http://blog.csdn.net/carson2005/article/details/40581463. [Accessed: 22- Apr- 2016].

[43] Blog.sina.com.cn, 2016. [Online]. Available: http://blog.sina.com.cn/s/blog_4a540be60102uwcr.html. [Accessed: 22- Apr- 2016].

[44] "timnugent/kpca-eigen", GitHub, 2016. [Online]. Available: https://github.com/timnugent/kpca-eigen. [Accessed: 22- Apr- 2016].

[45] "Dennis Gabor", Wikipedia, 2016. [Online]. Available: https://en.wikipedia.org/wiki/Dennis_Gabor#cite_note-frs-1. [Accessed: 22- Apr- 2016].

[46] Singh, S. Kr., Chauhan, D.S., Vasta, M., Singh, R., "A robust skin color based face detection algorithm", Tamkang Journal of Science and Engineering, vol. 6, no. 4, pp. 227-234, 2003.

[47] Hjelmås, E., Low, B., "Face detection: a survey", Computer Vision and Image Understanding, vol. 83, no. 3, pp. 236-274, 2001.

[48] "Active Shape Models with Stasm", Milbo.users.sonic.net, 2016. [Online]. Available: http://www.milbo.users.sonic.net/stasm/index.html. [Accessed: 22- Apr- 2016].

[49] "IntraFace", Humansensing.cs.cmu.edu, 2016. [Online]. Available: http://www.humansensing.cs.cmu.edu/intraface/download.html. [Accessed: 22- Apr- 2016].

[50] Yow, K., Cipolla, R., "Feature-based human face detection", Image and Vision Computing, vol. 15, no. 9, pp. 713-735, 1997.

[51] Hjelmas, E., "Feature-based face recognition", Proceedings of the Norwegian Image Processing and Pattern Recognition Conference, 2000.

[52] Jeng, S., Liao, H., Han, C., Chern, M., Liu, Y., "Facial feature detection using geometrical face model: An efficient approach", Pattern Recognition, vol. 31, no. 3, pp. 273-282, 1998.

[53] Curran, K., Li, X., McCaughley, N., "Neural network face detection", The Imaging Science Journal, vol. 53, no. 2, pp. 105-115, 2005.

[54] Lawrence, S., Giles, C., Tsoi, A., Back, A., "Face recognition: a convolutional neural-network approach", IEEE Transactions on Neural Networks and Learning Systems, vol. 8, no. 1, pp. 98-113, 1997.

[55] Fayad, M., Schmidt, D., Johnson, R., "Building application frameworks", New York: Wiley, 1999.

[56] Amaro, E., Nuno-Maganda, M., Morales-Sandoval, M., "Evaluation of machine learning techniques for face detection and recognition", Proceedings of the 22nd International Conference on Electrical Communications and Computers (CONIELECOMP), pp. 213-218, 2012.

[57] Bartlett, M., Littlewort, G., Frank, M., Lainscsek, C., Fasel, I., Movellan, J., "Recognizing facial expression: machine learning and application to spontaneous behavior", Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR), pp. 568-573, 2005.

[58] Brun, R., Rademakers, F., "ROOT — An object oriented data analysis framework", Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment, vol. 389, no. 1-2, pp. 81-86, 1997.

[59] Krueger, C., "Software reuse", ACM Computing Surveys, vol. 24, no. 2, pp. 131-183, 1992.

Appendix

Code for the framework

1. Face Representation

    a. Gabor Wavelet

    The code is provided by [42].

```
Mat WDT( const Mat &_src, const string _wname, const int _level )const
{
   int reValue = THID_ERR_NONE;
   Mat src = Mat_<float>(_src);
   Mat dst = Mat::zeros( src.rows, src.cols, src.type() );
   int N = src.rows;
   int D = src.cols;

   Mat lowFilter;
   Mat highFilter;
   wavelet( _wname, lowFilter, highFilter );

   int t=1;
   int row = N;
   int col = D;

   while( t<=_level )
   {
      for( int i=0; i<row; i++ )
      {
         Mat oneRow = Mat::zeros( 1,col, src.type() );
         for ( int j=0; j<col; j++ )
         {
            oneRow.at<float>(0,j) = src.at<float>(i,j);
         }
         oneRow = waveletDecompose( oneRow, lowFilter, highFilter );
         for ( int j=0; j<col; j++ )
         {
            dst.at<float>(i,j) = oneRow.at<float>(0,j);
         }
      }

#if 0
      //normalize( dst, dst, 0, 255, NORM_MINMAX );
      IplImage dstImg1 = IplImage(dst);
      cvSaveImage( "dst.jpg", &dstImg1 );
#endif
      for ( int j=0; j<col; j++ )
      {
         Mat oneCol = Mat::zeros( row, 1, src.type() );
         for ( int i=0; i<row; i++ )
         {
            oneCol.at<float>(i,0) = dst.at<float>(i,j);
         }
```

```
            oneCol = ( waveletDecompose( oneCol.t(), lowFilter, highFilter ) ).t();

            for ( int i=0; i<row; i++ )
            {
                dst.at<float>(i,j) = oneCol.at<float>(i,0);
            }
        }

#if 0
        //normalize( dst, dst, 0, 255, NORM_MINMAX );
        IplImage dstImg2 = IplImage(dst);
        cvSaveImage( "dst.jpg", &dstImg2 );
#endif

        row /= 2;
        col /=2;
        t++;
        src = dst;
    }

    return dst;
}
Mat IWDT( const Mat &_src, const string _wname, const int _level )const
{
    int reValue = THID_ERR_NONE;
    Mat src = Mat_<float>(_src);
    Mat dst = Mat::zeros( src.rows, src.cols, src.type() );
    int N = src.rows;
    int D = src.cols;

    Mat lowFilter;
    Mat highFilter;
    wavelet( _wname, lowFilter, highFilter );

    int t=1;
    int row = N/std::pow( 2., _level-1);
    int col = D/std::pow(2., _level-1);

    while ( row<=N && col<=D )
    {
        for ( int j=0; j<col; j++ )
        {
            Mat oneCol = Mat::zeros( row, 1, src.type() );
            for ( int i=0; i<row; i++ )
            {
                oneCol.at<float>(i,0) = src.at<float>(i,j);
            }
            oneCol = ( waveletReconstruct( oneCol.t(), lowFilter, highFilter ) ).t();

            for ( int i=0; i<row; i++ )
            {
                dst.at<float>(i,j) = oneCol.at<float>(i,0);
            }
        }

#if 0
```

```
        //normalize( dst, dst, 0, 255, NORM_MINMAX );
        IplImage dstImg2 = IplImage(dst);
        cvSaveImage( "dst.jpg", &dstImg2 );
#endif
        for( int i=0; i<row; i++ )
        {
            Mat oneRow = Mat::zeros( 1,col, src.type() );
            for ( int j=0; j<col; j++ )
            {
                oneRow.at<float>(0,j) = dst.at<float>(i,j);
            }
            oneRow = waveletReconstruct( oneRow, lowFilter, highFilter );
            for ( int j=0; j<col; j++ )
            {
                dst.at<float>(i,j) = oneRow.at<float>(0,j);
            }
        }

#if 0
        //normalize( dst, dst, 0, 255, NORM_MINMAX );
        IplImage dstImg1 = IplImage(dst);
        cvSaveImage( "dst.jpg", &dstImg1 );
#endif

        row *= 2;
        col *= 2;
        src = dst;
    }

    return dst;
}
void wavelet( const string _wname, Mat &_lowFilter, Mat &_highFilter )const
{
    if ( _wname=="haar" || _wname=="db1" )
    {
        int N = 2;
        _lowFilter = Mat::zeros( 1, N, CV_32F );
        _highFilter = Mat::zeros( 1, N, CV_32F );

        _lowFilter.at<float>(0, 0) = 1/sqrtf(N);
        _lowFilter.at<float>(0, 1) = 1/sqrtf(N);

        _highFilter.at<float>(0, 0) = -1/sqrtf(N);
        _highFilter.at<float>(0, 1) = 1/sqrtf(N);
    }
    if ( _wname =="sym2" )
    {
        int N = 4;
        float h[] = {-0.483, 0.836, -0.224, -0.129 };
        float l[] = {-0.129, 0.224,    0.837, 0.483 };

        _lowFilter = Mat::zeros( 1, N, CV_32F );
        _highFilter = Mat::zeros( 1, N, CV_32F );

        for ( int i=0; i<N; i++ )
        {
```

```
            _lowFilter.at<float>(0, i) = l[i];
            _highFilter.at<float>(0, i) = h[i];
        }

    }
}
Mat waveletDecompose( const Mat &_src, const Mat &_lowFilter, const Mat &_highFilter )const
{
    assert( _src.rows==1 && _lowFilter.rows==1 && _highFilter.rows==1 );
    assert( _src.cols>=_lowFilter.cols && _src.cols>=_highFilter.cols );
    Mat &src = Mat_<float>(_src);

    int D = src.cols;

    Mat &lowFilter = Mat_<float>(_lowFilter);
Mat &highFilter = Mat_<float>(_highFilter);

    Mat dst1 = Mat::zeros( 1, D, src.type() );
    Mat dst2 = Mat::zeros( 1, D, src.type()  );

    filter2D( src, dst1, -1, lowFilter );
    filter2D( src, dst2, -1, highFilter );

    Mat downDst1 = Mat::zeros( 1, D/2, src.type() );
    Mat downDst2 = Mat::zeros( 1, D/2, src.type() );

    resize( dst1, downDst1, downDst1.size() );
    resize( dst2, downDst2, downDst2.size() );

    for ( int i=0; i<D/2; i++ )
    {
        src.at<float>(0, i) = downDst1.at<float>( 0, i );
        src.at<float>(0, i+D/2) = downDst2.at<float>( 0, i );
    }

    return src;
}
Mat waveletReconstruct( const Mat &_src, const Mat &_lowFilter, const Mat &_highFilter )const
{
    assert( _src.rows==1 && _lowFilter.rows==1 && _highFilter.rows==1 );
    assert( _src.cols>=_lowFilter.cols && _src.cols>=_highFilter.cols );
    Mat &src = Mat_<float>(_src);

    int D = src.cols;

    Mat &lowFilter = Mat_<float>(_lowFilter);
    Mat &highFilter = Mat_<float>(_highFilter);

    Mat Up1 = Mat::zeros( 1, D, src.type() );
    Mat Up2 = Mat::zeros( 1, D, src.type() );

    Mat roi1( src, Rect(0, 0, D/2, 1) );
    Mat roi2( src, Rect(D/2, 0, D/2, 1) );
    resize( roi1, Up1, Up1.size(), 0, 0, INTER_CUBIC );
    resize( roi2, Up2, Up2.size(), 0, 0, INTER_CUBIC );
```

```
    Mat dst1 = Mat::zeros( 1, D, src.type() );
    Mat dst2= Mat::zeros( 1, D, src.type() );
    filter2D( Up1, dst1, -1, lowFilter );
    filter2D( Up2, dst2, -1, highFilter );

    dst1 = dst1 + dst2;

    return dst1;
}
```

b. PCA

```
CPCAAlgoritm::CPCAAlgoritm(int iImgWidth, int iImgHeight, int iNumSamples, PBYTE pbyImgSamples)
: CLinear2DArray(iImgWidth * iImgHeight, iNumSamples, sizeof(float)),
 m_iImgWidth(iImgWidth), m_iImgHeight(iImgHeight),
  m_iImgSize(iImgWidth * iImgHeight), m_iNumSamples(iNumSamples),
  m_pbyImgSamples(pbyImgSamples), m_pPCAModel(NULL)
{
        ASSERT(m_iImgSize > m_iNumSamples);
}
BOOL CPCAAlgoritm::GetPCAModel(CPCAModel *pPCAModel)
{
        float fPrecision = (float)0.001, *pfAllEigenVectors, *pfTempCovarianceArray;
        int iIterationTime = 100;
        DWORD dwFullSize = m_iNumSamples * m_iNumSamples;

        //allocate memory for the model
        pPCAModel->SetDimensions(m_iImgWidth, m_iImgHeight, m_iNumSamples);

        //allocate temp memory
        pfTempCovarianceArray = new float[dwFullSize];
        pfAllEigenVectors = new float[dwFullSize];

        //find the mean vector and the covariance array
        CalculateCovariance(pPCAModel->MeanImgVector(), pPCAModel->CovarianceArray());

        //get all eigen vectors from the covariance array
        memcpy(pfTempCovarianceArray, pPCAModel->CovarianceArray(), sizeof(float) * dwFullSize);
        //reserve the original covariance array
        ::MatrixEigenVectors(pfTempCovarianceArray, m_iNumSamples, pfAllEigenVectors, fPrecision,
iIterationTime);

        //get low-dimensional eigen values and vectors
        int iNumEigenVectors = CalcalateNewSD(pfTempCovarianceArray, pfAllEigenVectors,
pPCAModel);

//      pPCAModel->SetNumEigens(iNumEigenVectors);

        delete []pfAllEigenVectors;
        delete []pfTempCovarianceArray;

        return TRUE;

}       //end of CPCAAlgoritm::DoPCA()
void CPCAAlgoritm::CalculateCovariance(PFLOAT pfMeanImgVector, PFLOAT pfCovarianceArray)
```

```
{
	int i, j, k;
	float dTmp;

	// pfMeanImgVector[], row by row
	for (j = 0; j < m_iImgSize; j++){
		dTmp = 0.0;
		for (i = 0; i < m_iNumSamples; i++)
			dTmp += (float)m_pbyImgSamples[m_pRows[i] + j];
		pfMeanImgVector[j] = dTmp / m_iNumSamples;
	}

	// m_iNumSamples * m_iNumSamples
	for (i = 0; i < m_iNumSamples; i++){	//row by row
		for (j = 0; j <= i; j ++){		//colum by colum
			dTmp = 0.0;
			for (k = 0; k < m_iImgSize; k++)
				dTmp += ((float)m_pbyImgSamples[m_pRows[i] + k] -
pfMeanImgVector[k]) * ((float)m_pbyImgSamples[m_pRows[j] + k] - pfMeanImgVector[k]);
				//dTmp += (float)(m_pbyImgSamples[i][k]) *
(float)(m_pbyImgSamples[j][k]);
		    pfCovarianceArray[m_pCols[i] + j] = dTmp / m_iImgSize;
		}
   }

	//pfCovarianceArray
	for (i = 0; i < m_iNumSamples; i++){
		for (j = i + 1; j < m_iNumSamples; j++)
			pfCovarianceArray[m_pCols[i] + j] = pfCovarianceArray[m_pCols[j] + i];
	}

}	//end of CPCAAlgoritm::CalculateCovariance()
int CPCAAlgoritm::CalcalateNewSD(PFLOAT pfCovarianceArray, PFLOAT pfAllEigenVectors,
CPCAModel *pPCAModel)
{
	DWORD		dwFullDataSize = m_iNumSamples * m_iNumSamples;
	float	dTmp, pfEigenValuesSum, dValve;
	int		i, j, k, iNumEigenVectors;

	PFLOAT pfAllEigenValues  = new float[m_iNumSamples * 2];
	PFLOAT pfResultTmp		= new float[dwFullDataSize];
	PFLOAT pfEigenValueArray = new float[dwFullDataSize];
	PFLOAT pfTmpEigenVectors = new float[dwFullDataSize];

	for (i = 0; i < (int)dwFullDataSize; i++)	pfEigenValueArray[i] = pfTmpEigenVectors[i] = 0.0;

	for (i = 0; i < m_iNumSamples; i++){
		pfAllEigenValues[i + i] = pfCovarianceArray[m_pCols[i] + i];	//value
		pfAllEigenValues[i + i + 1] = float(i);
	//index
	}

   ::BubbleSortFloat(pfAllEigenValues, m_iNumSamples, m_iNumSamples);

	pfEigenValuesSum = 0.0;
	for (i = 0; i < 2 * m_iNumSamples; i += 2)		pfEigenValuesSum += pfAllEigenValues[i];
```

```
	iNumEigenVectors = 0;
	dTmp = 0.0;
	dValve = (float)0.95;
	for (i = 0; i < m_iNumSamples; i++){
		dTmp += pfAllEigenValues[i + i] / pfEigenValuesSum;
		if (dTmp >= dValve){
			iNumEigenVectors = i + 1;
			break;
		}
	}

	//set number of eigens and allocate memory
	pPCAModel->SetNumEigens(iNumEigenVectors);

	//
	for (i = 0; i < iNumEigenVectors; i++)	pPCAModel->EigenValues()[i] = pfAllEigenValues[i << 1];

	// (m_iNumSamples * iNumEigenVectors)
	for (i = 0; i < iNumEigenVectors; i++){
		pfEigenValueArray[m_pCols[i] + i] = float(1.0 / sqrt (pfAllEigenValues[i << 1]));
		k = (int)pfAllEigenValues[i + i + 1];
		for (j = 0; j < m_iNumSamples; j++)
			pfTmpEigenVectors[m_pCols[j] + i] = pfAllEigenVectors[m_pCols[j] + k];
	}

	// (iNumEigenVectors * m_iNumSamples)
	for (i = 0; i < m_iNumSamples; i++){
		for (j = 0; j <= i - 1; j++){
			dTmp = pfTmpEigenVectors[m_pCols[i] + j];
			pfTmpEigenVectors[m_pCols[i] + j] = pfTmpEigenVectors[m_pCols[j] + i];
			pfTmpEigenVectors[m_pCols[j] + i] = dTmp;
		}
	}

	//	pfEigenValueArray(m_iNumSamples * iNumEigenVectors) x
	//		pfTmpEigenVectors(iNumEigenVectors * m_iNumSamples) ==> (m_iNumSamples * m_iNumSamples)
	for (i = 0; i < m_iNumSamples; i++){
		for (j = 0; j < m_iNumSamples; j++){
			dTmp = 0.0;
			for (k = 0; k < m_iNumSamples; k++)
				dTmp += pfEigenValueArray[m_pCols[i] + k] * pfTmpEigenVectors[m_pCols[k] + j];
			pfResultTmp[m_pCols[i] + j] = dTmp;
		}
	}

	//pEigenImgVectors_iImgSize * iNumEigenVectors
	for (i = 0; i < iNumEigenVectors; i++){
		for (j = 0; j <= m_iImgSize - 1; j++){
			dTmp = 0.0;
			for (k = 0; k < m_iNumSamples; k++)
				dTmp += pfResultTmp[m_pCols[i] + k] * m_pbyImgSamples[m_pRows[k] + j];
```

```
                        pPCAModel->EigenImgVectors()[m_pRows[i] + j] = dTmp;// +
pfMeanImgVector[j];
//                      pEigenImgVectors[i][j] = dTmp / sqrt(m_iNumSamples);
                    }
        }

        //calculate the square sum for future use
        pPCAModel->CalcEigenSQRSum();

        delete []pfAllEigenValues;
        delete []pfResultTmp;
        delete []pfEigenValueArray;
        delete []pfTmpEigenVectors;

        return iNumEigenVectors;

}       //end of CPCAAlgoritm::CalcalateNewSD()

void CPCAAlgoritm::GetPCACoeff(PBYTE pImg, PFLOAT pCoeff)
{
        int i, j;
        float fTemp;

        for (i = 0; i < m_pPCAModel->NumEigens(); i++){
                fTemp = 0.0;
                for (j = 0; j < m_iImgSize; j++)
//                      fTemp += pEigenImgVectors[i][j] * data[j];
                        fTemp += (m_pPCAModel->EigenImgVector(m_pRows[i] + j)  / m_pPCAModel-
>EigenSQRSum(i))
                                                * ((float)pImg[j] - m_pPCAModel->MeanImgVector(j));
                pCoeff[i] = fTemp;
        }

}       //end of CPCAAlgoritm::GetPCACoeff()

void CPCAAlgoritm::ReconstructImage(PBYTE pImg, PFLOAT pCoeff)
{
        int i, j;
        float fTemp;

        for (i = 0; i < m_iImgSize; i++){
                fTemp = 0.0;
                for (j = 0; j < m_pPCAModel->NumEigens(); j++)
                        fTemp += float(m_pPCAModel->EigenImgVector(m_pRows[j] + i) * pCoeff[j] /
m_pPCAModel->EigenSQRSum(j));
                pImg[i] = (BYTE)(fTemp + m_pPCAModel->MeanImgVector(i) + 0.5);
        }


}       //end of CPCAAlgoritm::ReconstructImage()
```

c. Shape & Texture

The code for Shape & Texture is not found; however, here we provide two references where possible implementation thinkings might be obtained [48] [49].

2. Face Detection

a. Statistical Model

For the statistical model, we present codes for LDA method implementation provided by Zhou Li [40].

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#include "lda.h"
#include "file_access.h"
//test code
#include <fstream>
#include <iostream>
#include "document.h"
#include "time.h"
using namespace std;
clock_t start, finish;
//test code

#include "file_access.h"

void init_param(struct corpus *cps,struct est_param* param,int topic_num);

// ./lda topic_num sample_num data model_name

int main(int argc, char *argv[])
{
	if ( argc < 5 )
	{
		printf("usage: ./lda topic_num sample_num data model_name\n");
		return 0;
	}
	int topic_num = atoi(argv[1]);
	int sample_num = atoi(argv[2]); // how many samples we want, we use these sample's to
calculate phi and theta respectively and then use their averge to represent theta and phi
	char * data = argv[3];
	string model_name = argv[4];
	int burn_in_num = 1000; //how many iterations are there until we are out of burn-in period
	// set beta = 0.1 and alpha = 50 / K
	double beta = 0.1;
	double alpha = (double)50 / (double)topic_num;
	int SAMPLE_LAG = 10;
	struct corpus *cps;
	struct est_param param;
	cps = read_corpus(data);
	init_param(cps,&param,topic_num);
	cout << "parameter initialized" << endl;
	//do gibbs_num times sampling and compute averge theta,phi
```

```
	// burn in period
	//for (int iter_time=0; iter_time<burn_in_num + sample_num * SAMPLE_LAG; iter_time++)
	int iter_time = 0;
	int sample_time = 0;
	int p_size = topic_num<2?2:topic_num;
	double *p = (double*)malloc(sizeof(double)*p_size); //multinomial sampling tempoary storage space;
	double vbeta = cps->num_terms * beta; // v*beta is used in funciton sampling and this value never change so I put it here
	while (1)
	{
		start = clock();
		cout << "start sampling all document" << endl;
		//foreach documents, apply gibbs sampling
		for (int m=0; m<cps->num_docs; m++)
		{
			int word_index = 0; // word_index indicates that a word is the (word_index)th word in the document.
			double s_talpha = param.nd_sum[m] - 1 + topic_num * alpha; //nd_num[m]+topic_num * alpha will not change per document, so I put it here
			for (int l=0; l<cps->docs[m].length; l++)
			{
				for (int c=0; c<cps->docs[m].words[l].count; c++)
				{
					param.z[m][word_index] = sampling(m,word_index,cps->docs[m].words[l].id,topic_num,cps,&param,alpha,beta,p,s_talpha,vbeta);
					word_index++;
				}
			}
		}

		finish = clock();
		double dur = (double)(finish - start) / CLOCKS_PER_SEC;
		cout << "total time : " << dur << endl;


		if ((iter_time >= burn_in_num) && (iter_time % SAMPLE_LAG == 0))
		{
			calcu_param(&param, cps,topic_num,alpha,beta);
			cout << "\t" << sample_time+1 <<"# phi,theta calulation completed" << endl;
			sample_time++;
		}
		cout << iter_time+1 <<"# iteration completed"<<endl;
		iter_time++;
		if (sample_time ==sample_num)
		{
			break;
		}
	}
	// calculate average of phi and theta
	cout << "calculating the average of phi and theta" << endl;
	average_param(&param, cps,topic_num,alpha,beta,sample_num);
	cout << "parameter estimation completed, now saving model" << endl;
	save_model(cps,&param,model_name,alpha,beta,topic_num,sample_num);
	/*
	//test code
```

```
	ofstream out("ap.txt");

	for (int i=0; i<cps->num_docs; i++)
	{
		out << cps->docs[i].length << " ";
		for (int j=0; j < cps->docs[i].length; j++ )
		{
			out << cps->docs[i].words[j].id << ":" << cps->docs[i].words[j].count;
			if (j!=cps->docs[i].length-1)
			{
				out << " ";
			}
		}
		out << endl;
	}
	out.close();
	//test code
	*/
	/*
	//test code
	for (int d=0; d<cps->num_docs; d++)
	{
		int word_index = 0; // word_index indicates that a word is the (word_index)th word in the
document.
		int doc_length = 0;
		for (int l=0; l<cps->docs[d].length; l++)
		{

			for (int t=0; t<cps->docs[d].words[l].count; t++)
			{
				cout << param.z[d][word_index] << " ";
				word_index++;
			}
		}
	}
	//test code
	*/
	return 0;
}

int sampling(int m, int n, int word_id, int topic_num, struct corpus* cps, struct est_param *param,double
alpha, double beta,double* p,double s_talpha,double vbeta)
{
	int topic_id = param->z[m][n];
	param->nw[topic_id][word_id]--;
	param->nd[m][topic_id]--;
	param->nw_sum[topic_id]--;

	p[0] = (param->nw[0][word_id] + beta) / (param->nw_sum[0] + vbeta) * (param->nd[m][0] + alpha)
/ s_talpha;
	p[1] = p[0];
	for (int k = 1; k < topic_num-1; k++)
	{
		p[k] += (param->nw[k][word_id] + beta) / (param->nw_sum[k] + vbeta) * (param->nd[m][k]
+ alpha) / s_talpha;
		p[k+1] = p[k];
```

```
        }
        p[topic_num-1] += (param->nw[topic_num-1][word_id] + beta) / (param->nw_sum[topic_num-1] +
vbeta) * (param->nd[m][topic_num-1] + alpha) / s_talpha;
        // choose a sample from topic
        double rand = ((double)random() / RAND_MAX) * p[topic_num - 1];
        int sample_topic;
        for (sample_topic = 0; sample_topic < topic_num; sample_topic++)
        {
                if (p[sample_topic] > rand)
                {
                        break;
                }
        }
        // add new topic to statistics var
        param->nw[sample_topic][word_id]++;
        param->nd[m][sample_topic]++;
        param->nw_sum[sample_topic]++;
        return sample_topic;
}

void average_param(struct est_param *param, struct corpus* cps,int topic_num,double alpha,double
beta, int sample_num)
{
        // theta
        for (int m=0; m<cps->num_docs; m++)
        {
                for (int k=0; k<topic_num; k++)
                {
                        param->theta[m][k] /= sample_num;
                }
        }
        // phi
        for (int k=0; k<topic_num; k++)
        {
                for (int v=0; v<cps->num_terms; v++)
                {
                        param->phi[k][v] /= sample_num;
                }
        }
}


void calcu_param(struct est_param *param, struct corpus* cps,int topic_num,double alpha,double beta)
{
        // theta
        for (int m=0; m<cps->num_docs; m++)
        {
                for (int k=0; k<topic_num; k++)
                {
                        param->theta[m][k] += (param->nd[m][k] + alpha) / (param->nd_sum[m] +
topic_num * alpha);
                }
        }
        // phi
        for (int k=0; k<topic_num; k++)
        {
```

```
                for (int v=0; v<cps->num_terms; v++)
                {
                        param->phi[k][v] += (param->nw[k][v] + beta) / (param->nw_sum[k] + cps->num_terms * beta);
                }
        }
}

void set_zero(int **a,int rows,int columns)
{
        for (int i=0; i<rows; i++)
        {
                for (int j=0; j<columns; j++)
                {
                        a[i][j] = 0;
                }
        }
}

void set_zero_double(double **a,int rows,int columns)
{
        for (int i=0; i<rows; i++)
        {
                for (int j=0; j<columns; j++)
                {
                        a[i][j] = 0;
                }
        }
}


void init_param(struct corpus *cps, struct est_param* param,int topic_num)
{
        //1. alloc z[m][n], z[m][n] stands for topic assigned to nth word in mth document
        param->z = (int**)malloc (sizeof(int*) * cps->num_docs);
        for (int i=0; i<cps->num_docs; i++)
        {
                param->z[i] = (int*)malloc (sizeof(int) * cps->docs[i].num_term);
        }

        //2. alloc theta[m][k], theta[m][k] stands for the topic mixture proportion for document m
        param->theta = (double**)malloc (sizeof(double*) * cps->num_docs);
        for (int j=0; j<cps->num_docs; j++)
        {
                param->theta[j] = (double*)malloc (sizeof(double) * topic_num);
        }
        set_zero_double(param->theta,cps->num_docs,topic_num);

        //3. alloc phi[k][v], phi[k][v] stands for the probability of vth word in vocabulary is assigned to topic k
        param->phi = (double**)malloc (sizeof(double*) * topic_num);
        for (int z=0; z<topic_num; z++)
        {
                param->phi[z] = (double*)malloc (sizeof(double) * cps->num_terms);
        }
        set_zero_double(param->phi,topic_num,cps->num_terms);
```

```
	//4. alloc nd[m][k], nd[m][k] stands for the number of kth topic assigned to mth document
	param->nd = (int**)malloc (sizeof(int*) * cps->num_docs);
	for (int m=0; m< cps->num_docs; m++)
	{
		param->nd[m] = (int*)malloc(sizeof(int) * topic_num);
	}
	set_zero(param->nd, cps->num_docs, topic_num);

	//5. alloc nw[k][t], nw[k][t] stands for the number of kth topic assigned to tth term
	param->nw = (int**)malloc (sizeof(int*) * topic_num);
	for (int n=0; n < topic_num; n++)
	{
		param->nw[n] = (int*)malloc(sizeof(int) * cps->num_terms);
	}
	set_zero(param->nw, topic_num, cps->num_terms);

	//6. alloc nd_sum[m], nd_sum[m] total number of word in mth document
	param->nd_sum = (int*)malloc(sizeof(int)*cps->num_docs);
	memset(param->nd_sum,0,sizeof(int)*cps->num_docs);

	//7. alloc nw_sum[k], nw_sum[k] total number of terms assigned to kth topic
	param->nw_sum = (int*)malloc(sizeof(int)*topic_num);
	memset(param->nw_sum,0,sizeof(int)*topic_num);

	//8. init nd[m][k],nw[k][t],nd_sum[m],nw_sum[k]
	srandom(time(0)); // set seed for random function
	for (int d=0; d<cps->num_docs; d++)
	{
		int word_index = 0; // word_index indicates that a word is the (word_index)th word in the
document.
		for (int l=0; l<cps->docs[d].length; l++)
		{
			for (int t=0; t<cps->docs[d].words[l].count; t++)
			{
				param->z[d][word_index] = (int)(((double)random() / RAND_MAX) *
topic_num); // set z randomly
				param->nw[param->z[d][word_index]][cps->docs[d].words[l].id] ++;
				param->nd[d][param->z[d][word_index]]++;
				param->nw_sum[param->z[d][word_index]]++;

				word_index++;
			}
		}
		param->nd_sum[d] += word_index;
	}
}
```

b. Neural Network

The code is provided by [41].

```
int main()
```

```
{
    //Setup the BPNetwork
    CvANN_MLP bp;
    // Set up BPNetwork's parameters
    CvANN_MLP_TrainParams params;
    params.train_method=CvANN_MLP_TrainParams::BACKPROP;
    params.bp_dw_scale=0.1;
    params.bp_moment_scale=0.1;
    //params.train_method=CvANN_MLP_TrainParams::RPROP;
    //params.rp_dw0 = 0.1;
    //params.rp_dw_plus = 1.2;
    //params.rp_dw_minus = 0.5;
    //params.rp_dw_min = FLT_EPSILON;
    //params.rp_dw_max = 50.;

    // Set up training data
    float labels[3][5] = {{0,0,0,0,0},{1,1,1,1,1},{0,0,0,0,0}};
    Mat labelsMat(3, 5, CV_32FC1, labels);

    float trainingData[3][5] = { {1,2,3,4,5},{111,112,113,114,115}, {21,22,23,24,25} };
    Mat trainingDataMat(3, 5, CV_32FC1, trainingData);
    Mat layerSizes=(Mat_<int>(1,5) << 5,2,2,2,5);
    bp.create(layerSizes,CvANN_MLP::SIGMOID_SYM);//CvANN_MLP::SIGMOID_SYM
                                        //CvANN_MLP::GAUSSIAN
                                        //CvANN_MLP::IDENTITY
    bp.train(trainingDataMat, labelsMat, Mat(),Mat(), params);


    // Data for visual representation
    int width = 512, height = 512;
    Mat image = Mat::zeros(height, width, CV_8UC3);
    Vec3b green(0,255,0), blue (255,0,0);
    // Show the decision regions given by the SVM
    for (int i = 0; i < image.rows; ++i)
        for (int j = 0; j < image.cols; ++j)
        {
            Mat sampleMat = (Mat_<float>(1,5) << i,j,0,0,0);
            Mat responseMat;
            bp.predict(sampleMat,responseMat);
            float* p=responseMat.ptr<float>(0);
            float response=0.0f;
            for(int k=0;k<5;i++){
            //  cout<<p[k]<<" ";
                response+=p[k];
            }
            if (response >2)
                image.at<Vec3b>(j, i)  = green;
            else
                image.at<Vec3b>(j, i)  = blue;
        }

        // Show the training data
        int thickness = -1;
        int lineType = 8;
        circle( image, Point(501,  10), 5, Scalar(  0,   0,   0), thickness, lineType);
        circle( image, Point(255,  10), 5, Scalar(255, 255, 255), thickness, lineType);
```

```
    circle( image, Point(501, 255), 5, Scalar(255, 255, 255), thickness, lineType);
    circle( image, Point( 10, 501), 5, Scalar(255, 255, 255), thickness, lineType);

    imwrite("result.png", image);        // save the image

    imshow("BP Simple Example", image); // show it to the user
    waitKey(0);

}
```

c. Color-based

The code is provided by [43].

```
function BinImg = GetFaceBin(rgb)
Ycbcr = rgb2ycbcr(rgb);

fThreshold = 0.22;

[M,N,D]=size(Ycbcr);
FaceProbImg = zeros(M,N,1);
BinImg = uint8(zeros(M,N,1));
Mean = [117.4316 148.5599]';
C = [97.0946 24.4700;
     24.4700 141.9966];
cbcr = zeros(2,1);
for i=1:M
   for j=1:N
      cbcr(1) = Ycbcr(i,j,2);
      cbcr(2) = Ycbcr(i,j,3);
      FaceProbImg(i,j)=exp(-0.5*(cbcr-Mean)'*inv(C)*(cbcr-Mean));
      if FaceProbImg(i,j)>fThreshold
         BinImg(i,j) = 1;
      end
   end
end
se=strel('disk',3');
BinImg = imopen(BinImg,se);
imdilate(BinImg,se);
% subplot(122);imshow(BinImg*255);title('');
CC = bwconncomp(BinImg);
numPixels = cellfun(@numel,CC.PixelIdxList);。
[biggest,idx] = max(numPixels);
for i=1:CC.NumObjects
 if i~=idx
 BinImg(CC.PixelIdxList{i}) = 0;
 end
end
% figure(2);imshow(uint8(BinImg*255));
end
```

3. Pre-processing

a. Face Seperation

```
 Mat warp_dst = GetAffinedMat(tmpImgCopy2, frame_gray);
```

```
IplImage *imgLbpSrc = (&(IplImage)warp_dst);
IplImage *imgLbpDst = cvCreateImage(cvGetSize(imgLbpSrc), IPL_DEPTH_8U,1);;

m_lbpInst.CreatLBP(imgLbpSrc, imgLbpDst);

Mat lbp_dst(imgLbpDst);

Rect roi(51, 19, 80, 89);
Mat matRoi = lbp_dst(roi);
m_vmatLbpFace.push_back(matRoi);

Rect eyesRoi(46, 22, 84, 38);
Mat matEyes = lbp_dst(eyesRoi);
m_vmatLbpEyes.push_back(matEyes);

Rect mouthRoi(69 ,102, 38, 22);
Mat matMouth = lbp_dst(mouthRoi);
m_vmatLbpMouth.push_back(matMouth);

Rect NoseRoi(71, 57, 39, 37);
Mat matNose = lbp_dst(NoseRoi);
m_vmatLbpNose.push_back(matNose);
```

b. LBP

```
void CLBP::CreatLBP(IplImage *src,IplImage *dst)
{
	int iTemp[8] = {0};
	CvScalar s;

	IplImage *ImgTemp = cvCreateImage(cvGetSize(src), IPL_DEPTH_8U, 1);
	uchar *data = (uchar*)src->imageData;
	int iStep = src-> widthStep;

	for (int i=1;i<src->height-1;i++)
		for(int j=1;j<src->width-1;j++){

			int sum=0;
			if(data[(i-1)*iStep+j-1]>data[i*iStep+j])
				iTemp[0]=1;
			else
				iTemp[0]=0;
			if(data[i*iStep+(j-1)]>data[i*iStep+j])
				iTemp[1]=1;
			else
				iTemp[1]=0;
			if(data[(i+1)*iStep+(j-1)]>data[i*iStep+j])
				iTemp[2]=1;
			else
				iTemp[2]=0;
			if (data[(i+1)*iStep+j]>data[i*iStep+j])
				iTemp[3]=1;
			else
				iTemp[3]=0;
			if (data[(i+1)*iStep+(j+1)]>data[i*iStep+j])
```

```
                    iTemp[4]=1;
                else
                    iTemp[4]=0;
                if(data[i*iStep+(j+1)]>data[i*iStep+j])
                    iTemp[5]=1;
                else
                    iTemp[5]=0;
                if(data[(i-1)*iStep+(j+1)]>data[i*iStep+j])
                    iTemp[6]=1;
                else
                    iTemp[6]=0;
                if(data[(i-1)*iStep+j]>data[i*iStep+j])
                    iTemp[7]=1;
                else
                    iTemp[7]=0;
                s.val[0] =
(iTemp[0]*1+iTemp[1]*2+iTemp[2]*4+iTemp[3]*8+iTemp[4]*16+iTemp[5]*32+iTemp[6]*64+iTemp[7]*128);

                cvSet2D(dst,i,j,s);

            }

}       // end of CLBP::CreatLBP()
```

4. PCA

    a. Standard PCA

```
 void CPPPCA::InitPCA(String strDirName, Mat matSrc, vector<Mat> vmatSrc)
{
        double dbnumber_principal_compent = 0.95;
        Mat pcaImg, projectImg;

        PCA pca(matSrc, Mat(), CV_PCA_DATA_AS_COL, dbnumber_principal_compent);
        Mat EigenVectors = pca.eigenvectors;

        pcaImg = normalize(pca.eigenvectors.row(0)).reshape(1, vmatSrc[0].rows);
//      pcaFace = pca.eigenvectors.row(0).reshape(1, src[0].rows);
        imwrite(((".\\") + strDirName + (".\\Pca.jpg")), pcaImg);

        Mat EigenValues = pca.eigenvalues;

        Mat dst;
        dst = pca.project(matSrc.col(0));
        projectImg = normalize(pca.backProject(dst).col(0)).reshape(1, vmatSrc[0].rows);
        imwrite(((".") + strDirName + (".\\Project.jpg")), projectImg);

        m_pcaTrain = pca;

}       // end of CPPPCA::InitPCA()

void CPPPCA::GetEigenValues(String strDirName, Mat matSrc, vector<Mat> vmatSrc, int iImgNum)
{
        string strInt;
        Mat dst, projectImg;
        const string strEigenValue = "D:\\Summer Project\\EigenValue\\FaceEigenValues.xml";
        FileStorage fs(strEigenValue, FileStorage::WRITE);
```

```
	for(int i = 0; i < iImgNum; i++){
		strInt = inttostring(i);
		dst = m_pcaTrain.project(matSrc.col(i));

//		projectImg = normalize(m_pcaTrain.backProject(dst).col(0)).reshape(1, vmatSrc[0].rows);
//		imwrite((("D:\\Summer Project\\") + strDirName + ("\\Project") + strInt + (".jpg")),
projectImg);
		fs << "eigenvalue" + strInt << dst;
	}
	fs.release();

}	// end of CPPPCA::GetEigenValues()
```

b. Kernel PCA

The code is provided by [44].

```
void PCA::load_data(const char* data, char sep){

	// Read data
	unsigned int row = 0;
	ifstream file(data);
	if(file.is_open()){
		string line,token;
		while(getline(file, line)){
			stringstream tmp(line);
			unsigned int col = 0;
			while(getline(tmp, token, sep)){
				if(X.rows() < row+1){
					X.conservativeResize(row+1,X.cols());
				}
				if(X.cols() < col+1){
					X.conservativeResize(X.rows(),col+1);
				}
				X(row,col) = atof(token.c_str());
				col++;
			}
			row++;
		}
		file.close();
		Xcentered.resize(X.rows(),X.cols());
	}else{
		cout << "Failed to read file " << data << endl;
	}

}

double PCA::kernel(const VectorXd& a, const VectorXd& b){

	/*
		Kernels
		1 = RBF
		2 = Polynomial
		TODO - add some of these these:
		http://crsouza.blogspot.co.uk/2010/03/kernel-functions-for-machine-learning.html
	*/
```

```
        switch(kernel_type){
           case 2  :
                return(pow(a.dot(b)+constant,order));
           default :
                return(exp(-gamma*((a-b).squaredNorm())));
        }

}

void PCA::run_kpca(){

        // Fill kernel matrix
        K.resize(X.rows(),X.rows());
        for(unsigned int i = 0; i < X.rows(); i++){
                for(unsigned int j = i; j < X.rows(); j++){
                        K(i,j) = K(j,i) = kernel(X.row(i),X.row(j));
                        //printf("k(%i,%i) = %f\n",i,j,K(i,j));
                }
        }
        //cout << endl << K << endl;

        EigenSolver<MatrixXd> edecomp(K);
        eigenvalues = edecomp.eigenvalues().real();
        eigenvectors = edecomp.eigenvectors().real();
        cumulative.resize(eigenvalues.rows());
        vector<pair<double,VectorXd> > eigen_pairs;
        double c = 0.0;
        for(unsigned int i = 0; i < eigenvectors.cols(); i++){
                if(normalise){
                        double norm = eigenvectors.col(i).norm();
                        eigenvectors.col(i) /= norm;
                }
                eigen_pairs.push_back(make_pair(eigenvalues(i),eigenvectors.col(i)));
        }
        // http://stackoverflow.com/questions/5122804/sorting-with-lambda
        sort(eigen_pairs.begin(),eigen_pairs.end(), [](const pair<double,VectorXd> a, const
pair<double,VectorXd> b) -> bool {return (a.first > b.first);} );
        for(unsigned int i = 0; i < eigen_pairs.size(); i++){
                eigenvalues(i) = eigen_pairs[i].first;
                c += eigenvalues(i);
                cumulative(i) = c;
                eigenvectors.col(i) = eigen_pairs[i].second;
        }
        transformed.resize(X.rows(),components);

        for(unsigned int i = 0; i < X.rows(); i++){
                for(unsigned int j = 0; j < components; j++){
                        for (int k = 0; k < K.rows(); k++){
          transformed(i,j) += K(i,k) * eigenvectors(k,j);
                        }
                }
        }

        /*
        cout << "Input data:" << endl << X << endl << endl;
        cout << "Centered data:"<< endl << Xcentered << endl << endl;
```

```
	cout << "Centered kernel matrix:" << endl << Kcentered << endl << endl;
	cout << "Eigenvalues:" << endl << eigenvalues << endl << endl;
	cout << "Eigenvectors:" << endl << eigenvectors << endl << endl;
	*/
	cout << "Sorted eigenvalues:" << endl;
	for(unsigned int i = 0; i < eigenvalues.rows(); i++){
		if(eigenvalues(i) > 0){
			cout << "PC " << i+1 << ": Eigenvalue: " << eigenvalues(i);
			printf("\t(%3.3f of variance, cumulative =
%3.3f)\n",eigenvalues(i)/eigenvalues.sum(),cumulative(i)/eigenvalues.sum());
		}
	}
	cout << endl;
	//cout << "Sorted eigenvectors:" << endl << eigenvectors << endl << endl;
	//cout << "Transformed data:" << endl << transformed << endl << endl;
}

void PCA::run_pca(){

	Xcentered = X.rowwise() - X.colwise().mean();
	C = (Xcentered.adjoint() * Xcentered) / double(X.rows());
	EigenSolver<MatrixXd> edecomp(C);
	eigenvalues = edecomp.eigenvalues().real();
	eigenvectors = edecomp.eigenvectors().real();
	cumulative.resize(eigenvalues.rows());
	vector<pair<double,VectorXd> > eigen_pairs;
	double c = 0.0;
	for(unsigned int i = 0; i < eigenvectors.cols(); i++){
		if(normalise){
			double norm = eigenvectors.col(i).norm();
			eigenvectors.col(i) /= norm;
		}
		eigen_pairs.push_back(make_pair(eigenvalues(i),eigenvectors.col(i)));
	}
	// http://stackoverflow.com/questions/5122804/sorting-with-lambda
	sort(eigen_pairs.begin(),eigen_pairs.end(), [](const pair<double,VectorXd> a, const
pair<double,VectorXd> b) -> bool {return (a.first > b.first);} );
	for(unsigned int i = 0; i < eigen_pairs.size(); i++){
		eigenvalues(i) = eigen_pairs[i].first;
		c += eigenvalues(i);
		cumulative(i) = c;
		eigenvectors.col(i) = eigen_pairs[i].second;
	}
	transformed = Xcentered * eigenvectors;

}

void PCA::print(){

	cout << "Input data:" << endl << X << endl << endl;
	cout << "Centered data:"<< endl << Xcentered << endl << endl;
	cout << "Covariance matrix:" << endl << C << endl << endl;
	cout << "Eigenvalues:" << endl << eigenvalues << endl << endl;
	cout << "Eigenvectors:" << endl << eigenvectors << endl << endl;
	cout << "Sorted eigenvalues:" << endl;
	for(unsigned int i = 0; i < eigenvalues.rows(); i++){
```

```
                if(eigenvalues(i) > 0){
                        cout << "PC " << i+1 << ": Eigenvalue: " << eigenvalues(i);
                        printf("\t(%3.3f of variance, cumulative =
%3.3f)\n",eigenvalues(i)/eigenvalues.sum(),cumulative(i)/eigenvalues.sum());
                }
        }
        cout << endl;
        cout << "Sorted eigenvectors:" << endl << eigenvectors << endl << endl;
        cout << "Transformed data:" << endl << X * eigenvectors << endl << endl;
        //cout << "Transformed centred data:" << endl << transformed << endl << endl;

}

void PCA::write_transformed(string file){

        ofstream outfile(file);
        for(unsigned int i = 0; i < transformed.rows(); i++){
                for(unsigned int j = 0; j < transformed.cols(); j++){
                        outfile << transformed(i,j);
                        if(j != transformed.cols()-1) outfile << ",";
                }
                outfile << endl;
        }
        outfile.close();
        cout << "Written file " << file << endl;

}

void PCA::write_eigenvectors(string file){

        ofstream outfile(file);
        for(unsigned int i = 0; i < eigenvectors.rows(); i++){
                for(unsigned int j = 0; j < eigenvectors.cols(); j++){
                        outfile << eigenvectors(i,j);
                        if(j != eigenvectors.cols()-1) outfile << ",";
                }
                outfile << endl;
        }
        outfile.close();
        cout << "Written file " << file << endl;

}
```

c. Standard PCA

The code has been given in Chapter 4.

5. Verification

a. Correlation

```
double CPPPCA::GetCorelation(uchar *v1, uchar *v2, int n)
{
  double dSignal = 0, dNoise = 0;
  int i;
  double e1 = 0, e2 = 0, e12 = 0, c1 = 0, c2 = 0;
```

```
  //calculate the expectations of V1, V2 and V1 x V2
  for (i = 0; i < n; i++){
        e1  += v1[i];
        e2  += v2[i];
        e12 += v1[i] * v2[i];
  }

  e1  = e1  / n;
  e2  = e2  / n;
  e12 = e12 / n;

  //calulate the variances of R1 and R2
  for (i = 0; i < n; i++){
        c1 += (e1 - v1[i]) * (e1 - v1[i]);
        c2 += (e2 - v2[i]) * (e2 - v2[i]);
  }

  c1 = sqrt(c1 / n);
  c2 = sqrt(c2 / n);

  //calulate the correlation
  return IsZero(c1) || IsZero(c2)? 0.0 : fabs((e12 - e1 * e2) / (c1 * c2));

}       //end of CPPPCA::GetCorelation()
```

b. Mahalanobis Distance

```
double CPPPCA::CalMahDistance(Mat matSrc, Mat matTest)
{
        Mat matCovar, matMean;
        CvMat cvmatSrc, cvmatTest, cvmatCovar;
        cvmatSrc = matSrc;
        cvmatTest = matTest;

        calcCovarMatrix(&matSrc, 1, matCovar, matMean, CV_COVAR_NORMAL);
        cvmatCovar = matCovar;

        double dbMahDistance = cvMahalonobis(&cvmatSrc, &cvmatTest, &cvmatCovar);

        return dbMahDistance;

}       // end of CPPPCA::CalMahDistance()
```

c. Euclidean Distance

```
void Distance::distance(double a[],double b[]){
   square=0;
   for(int i=0;i<n;i++){
      square+=(a[i]-b[i])*(a[i]-b[i]);
   }
   Result=sqrt(square/n);
}
```