Ukrainian Catholic University

Faculty of Applied Sciences

Business Analytics & Computer Science Programmes

Fingerprint recognition using Principal Component Analysis

Linear Algebra final project report

Authors:
Sofia Garkot
Solomiia Leno

May 2020

Contents

Introduction

General topic description

Personal data security is quite an important issue in the XXI century. For years people came up with great ideas on sophisticated algorithms and techniques for safe data transmission and storage. One of the main achievements in this sphere is the biometric security: systems using face recognition, fingerprint detection etc. These methods proved themselves safe, quick and very useful, so people seek for more efficient implementations of them.

One of the most popular approach is fingerprint recognition, on which we concentrate in this project.

The most common approach to fingerprint processing is based on the patterns and small ridge features(i.e. minutiae points) matching. However this one is usually very resource consuming, especially in software implementations.

Problem description

The aim of this project is to **develop a fingerprint recognition algorithm based on the methods of linear algebra**.

The solution we plan on implementing is a web-application for users to verify the fingerprints that are uploaded in the global dataset. The user also has an opportunity to upload new images of fingerprints in order to test the accuracy of recognition on his/her own data.

We want the system to be able to deal with rotated, blurred and slightly damaged pictures. We plan to achieve this result by developing the algorithm for **pictures normalization** - based on detection of the core point on the fingerprints.

Existing approaches

As mentioned in the introduction section, there already exist plenty of implementations of fingerprint recognition systems and software. Two main approaches these all are based on are **pattern matching** and **minutiae features extraction**.

Pattern matching

Pattern matching relies mostly on recognition of fingerprint patterns. Scientists classify three of them: arch, loop and whorl.

This method does not produce the most accurate results as many people have same patterns and one person can have as many as all three of them with different minutiae. To say in more formal way, the first type error of pattern matching approach is quite big.

Still, this method allows one to verify if some people are siblings due to the hypotheses that fingerprint patterns are inherited.

Minutiae features extraction

Minutiae features is what makes one's finger pattern unique even though there are only three main patterns. They refer to the specific plot points on a fingerprint. There are plenty different of them like delta, core, ridge ending and so on.

Minutiae features matching allows one to check fingerprint with almost 100% certainty although it depends on the implementation.

Fingerprint recognition based on the PCA

Algorithm explanation

When one wants to perform a fingerprint recognition, he might ask if the picture is similar enough to the standard one, from the original dataset. To say in more formal way, one wants to ask if the distance between uploaded fingerprint and the standard one is small enough, meaning that it's smaller than some threshold value δ.

Every $n \times n$ pixel picture in the dataset plus user picture is converted into a vector from $\mathbb{R}^m$, where $m = n^2$

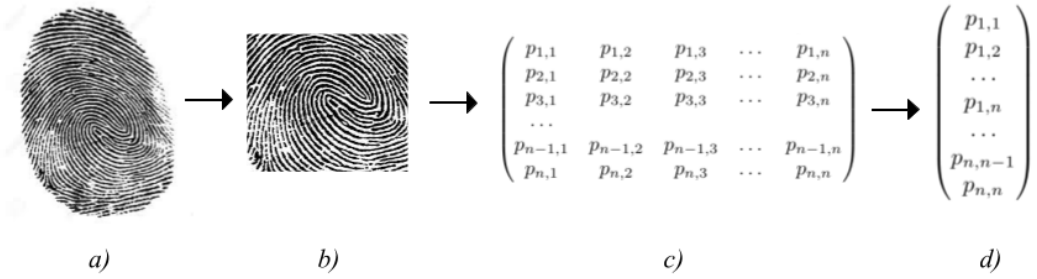

Fig. 1. Image processing steps:
a) original fingerprint picture, b) normalized picture cropped round core point, c) matrix representation of image, d) matrix flattened into single vector from $\mathbb{R}^{n^2}$

After each picture is represented with a real vector, they are gathered into a single **data matrix** of size $l \times m$, where m is the size of cropped image squared and l is the number of samples from the dataset.

The next step is to calculate the principal components of data, based on the **Principal Components Analysis** algorithm described in the next section.

The resulting principal components are basically an orthonormal set of vectors $pc_1, pc_2, \ldots pc_k$ which can be used to represent all the data with less dimensions but without loss of any significant information.

Then, we need to represent all the data within the new basis composed from principal components. One can represent the initial vector from $\mathbb{R}^n$ expressed in terms of standart basis - $e_1, e_2, \ldots e_n$ - in terms of new basis - $pc_1, pc_2, \ldots pc_k$ - by left-multiplying the original vector by transition matrix $\mathbf{T}$ of size $k \times n$.

The transition matrix $\mathbf{T}$ is essentially just the **transpose** of matrix $\mathbf{T'}= (pc_1, pc_2, \ldots pc_k)$. This way we get the transformation matrix of needed size.

Then each image vector im_i is represented within $pc_1, pc_2, \dots pc_k$ as

$$Tim_i = T'^T im_i = (im_i)_{PC}$$

where $(im_i)_{PC}$ is a k-dimensional data vector.

After change to new PC basis one works with vectors from $\mathbb{R}^k$ where the Euclidean distance is well defined.

In order to check if the uploaded image is actually representing the fingerprint noted, one calculates the **average distance** between the uploaded image and the ones from original dataset and then checks if the distance value is smaller than some threshold δ.

If the distance does not exceed δ - the uploaded fingerprint is classified as the one belonging to the same person as standard ones - if it does not - the uploaded fingerprint gets rejected.

Theoretical part: Principal Components Analysis

In terms of linear algebra, Principal Components Analysis (shortly - PCA) is an eigenvalue method, that is aimed to reduce the dimension of the data without loss of any important information.

Assume we work with a dataset consisting of **n** samples each described within **p** different features. The data are then represented as an $n \times p$ matrix $\mathbf{X}$.

The first step is to standardize the data - "shift" the data to **zero mean**.

We do so by calculating the rank 1 matrix $\mathbf{M}$ of means of each row.

$$M = \mathbf{1}(\bar{x_1}\bar{x_2} \dots \bar{x_n})$$

where $\mathbf{1} = (1, 1, \dots 1)^T$ and $x_i = \frac{1}{p}\sum_{j=0}^{p} s_{i,j}$ being the mean of i^{th} row.

Now the standardized data are represented within matrix $\mathbf{X_s}$

$$X_s = X - M$$

After that, there is expected correlation between the features. One needs to calculate the **covariance matrix S** as

$$S = X_s^T X_s$$

S is a **symmetric** $p \times p$ matrix. On the main diagonal we obtain the variances of each feature, and covariances of respective features on the $(i, j)^{th}$ entry. Since

$$Cov(feature_i, feature_j) = Cov(feature_j, feature_i)$$

it implies that $s_{i,j} = s_{j,i}$, and thus S is symmetric.

Because S is symmetric, it has **p distinct eigenvalues** and corresponding eigenvectors $v_1 \dots v_p$ form an orthogonal set and are called the **principal components**.

Then one needs to normalize $v_1, v_2 \dots v_p$ to get an **orthonormal** set of vectors - $w_1, w_2 \dots w_p$.

$$w_i = \frac{v_i}{||v_i||}$$

After the normalization of those vectors we get an orthonormal set of vectors, which contains the new basis we use later. The corresponding eigenvalues of each of them will indicate the amount of variation of corresponding feature from the total variation.

The total variance of the data is equal to the trace of the covariance matrix, i.e. the sum of the variances of each feature, which are diagonal entries of S.

$$Var_{total} = Var(feature_1) + Var(feature_2) + \ldots + Var(feature_p) =$$

$$= s_{1,1} + s_{2,2} + \ldots + s_{p,p} = trace(S) =$$

$$= \lambda_1 + \lambda_1 + \ldots + \lambda_p$$

The fraction of variation explained by the i_{th} feature is then equal to ${}^{\lambda_i}/_{Var_{total}}$.

Next step is to sort all the eigenvalues of matrix S so that $\lambda_1 \geq \lambda_2 \geq \ldots \geq \lambda_p$.

Now, if one needs to reduce the data to **k**-dimensions, we take k **largest** eigenvalues as corresponding principal components describe most of the data variance possible with k values. The eigenvectors chosen - principal components - form a new basis for the data.

Implementation

Data pre-processing

As mentioned previously, in order to deal with damaged pictures we need to perform **picture normalization** as well as **core point detection**.

The core point detection algorithm we used is based on the article *An algorithm for fingerprint core point detection*, linked in the references section.

The main idea is to normalize each picture matrix in order to deal with too dark or too bright pictures (with mean value equal to 50 as well as variance = 50, according to the upper mentioned paper).

After that we estimate the orientation field by dividing the picture into smaller non-overlapping batches of a small size (25×25 in our case) and calculating the **gradient** at each of the pixels. For each batch the orientation is estimated using:

$$\Theta = \text{atan}\left(\frac{\mathbf{G}_y}{\mathbf{G}_x}\right)$$

Where $\mathbf{G}_y$ and $\mathbf{G}_x$ are gradients on y and x axis computed using Sobel operator. The upper mentioned size was estimated by iterative procedure and was chosen as a consensus between estimation of too many core points (as an example 3×3 sized batch, Appendix A) and detection of no core points (as an example 29×29 sized batch, Appendix B). This way we get an orientation mask that shows the direction of fingerprint contours twist - the core point.

The next step is to crop the picture around the detected core point to some standard size for all the samples. In our case, the size of cropped picture is 150×150 pixels: smaller size will cause big data losses (comparison available in Appendix C and D) and for bigger sized images the algorithm will not perform efficiently. This way we already reduce the data dimension by almost **twice** as we start with 200×200 pictures.

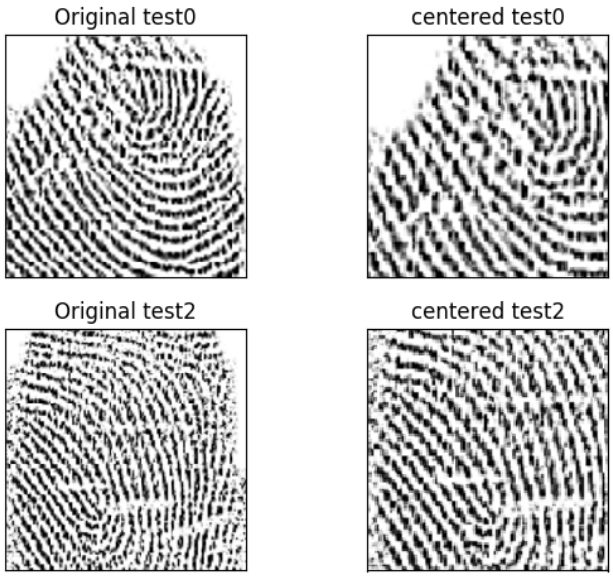

Fig. 2. Core point detection results:
left - original fingerprint images of size 200×200*; right - processed images of size* 150×150*, cropped around the core point*

Software implementation details

For implementing our solution we chose **Python** programming language as it provides one with many convenient tools that can ease calculations and implementation of common useful techniques.

For the implementation of basic linear algebra calculations like transposition, matrix multiplication etc. we used the **numpy** package along with functions from **math** package.

The implementation of all picture-related functional was done based on **skimage** and **scipy** packages.

As train and test data for our implementation we use the picture dataset **Fingerprint Color Image Database** by MathWorks, linked in the references section as well as in the project GitHub repository.

Although, we must admit that Python implementation is not the most convenient one with respect to computer resources such as memory. We started working on optimization of the algorithm for faster and more convenient eigenvalues and eigenvectors calculations, but did not manage to finish it on time.

Source code as well as the training data used for this project development can be found in our GitHib repository - **link**

Software implementation usage

To test our implementation one needs to clone the GutHub repositoty linked in the previous section and start the application by running the following commands in the global project directory.

```
pip install -r requirements.txt
```

`python main.py`

We also recommends creating a **virtual environment** before, in order to keep all the requirements all together in versions needed and have a convenient environment for code exploration and further development, if wanted.

Once the used started the application he / she needs to specify **relative path** to the picture of a fingerprint that needs to be verified and the **name** of hypothetical owner from the ones listed in the dataset.

Since at the beginning there are only sample data all possible owner names are of format **sub*** where * is any number from 1 to 50. We are working on a more convenient web-interface with possibility to upload custom data to dataset and test one's own fingerprints.

After, the application runs all the modules needed and returns one of two possible outcomes

`> verified` or `! not verified`

Results

In order to test the solution developed we implemented a program for automated testing - **test.py** module in project root directory. The test data can be found in **test_data** folder from the root directory.

Parameters of tested implementation

The performance of algorithm developed depends on many important constants. Here is the list of values we used in the tested implementation.

constant	meaning	value
batch size	the size of orientation estimation window	25×25
window size	the size of filter-matrix for Sobel gradient calculation	3×3
crop size	the size of image to crop to, for pre-processing	150×150 pixels
p	dimension of vectors to perform PCA on	respectivelly 150^2
k	the dimension to reduce to with PCA	$\frac{2}{3}p = 15000$
δ	threshold value for fingerprint verification	0.3

Implementation performance

In total, there were **100** total experiments based on **50** different fingerprints, **2** experiments per each fingerprint - one to be verified and the other - to be rejected . The results are the following.

result	count
positive verified	31 / 50 expected
negative verified	7 / 0 expected
positive non-verified	43 / 50 expected
negative non-verified	19 /0 expected

Here the **positive verified** result means that application successfully verified the owner or fingerprint, **negative verified** means that application verified the fingerprint that did not actually belong to the owner noted, **positive non-verified** - application successfully did not verify the fingerprint that does not belong to the owner noted, and respectively **negative non-verified** means that application failed to verify the fingerprint actually belongs to the owner listed.

As we can see, the algorithm produced quite positive results with average accuracy being **74%**. Still the result can be improved and we plan on doing it by optimizing the calculations part - calculations of eigenvalues as eigenvectors as well as experimenting the constants algorithm performance depends on for some higher results.

References

1. Vladimir, Gudkov. (2009). *Mathematical Models of Fingerprint Image On the Basis of Lines Description*. Online resource - link
2. W. Yongxu, A. Xinyu, D. Yuanfeng and Li Yongping (2006) *A Fingerprint Recognition Algorithm Based on Principal Component Analysis*. Conference paper - link
3. R. Bansal, P. Sehgal, P. Bedi (September, 2011) *Minutiae Extraction from Fingerprint Images*. Online resource - link
4. R. Jain (October, 2016) *Linear Algebra and PCA*. Hackearth. Online resource - link
5. Al-Refoa, A. & Alshraideh, Mohammad Sharieh, Ahmad. (2019). *A New Algorithm for Locating and Extracting Minutiae from Fingerprint Images*. Research gate. Online resource - link
6. A. Julasayvake, S. Choomchuay (March, 2007) *An algorithm for fingerprint core point detection*. Online resource - link
7. R. Hryniv (2019) *Lecture 13. Singular Value Decomposition* Personal collection of Rostyslav Hryniv, Ukrainian Catholic University, Lviv.
8. Dr. U. Gawande, K. Hajari and Y. Golhar (August, 2015) *Fingerprint Color Image Database*. Mathworks. Online resource - link

Appendix

A

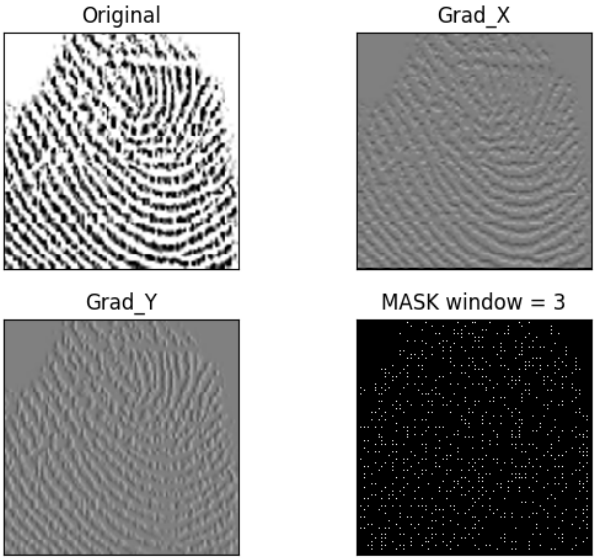

Fig. 3. Core point detection for batch size of 3:
upper left - original fingerprint image; upper right - processed images with Sobel 3x3 Image Gradient operator for x-direction; down left - processed images with Sobel 3x3 Image Gradient operator for y-direction; down right - white points are the estimated points of detected core

B

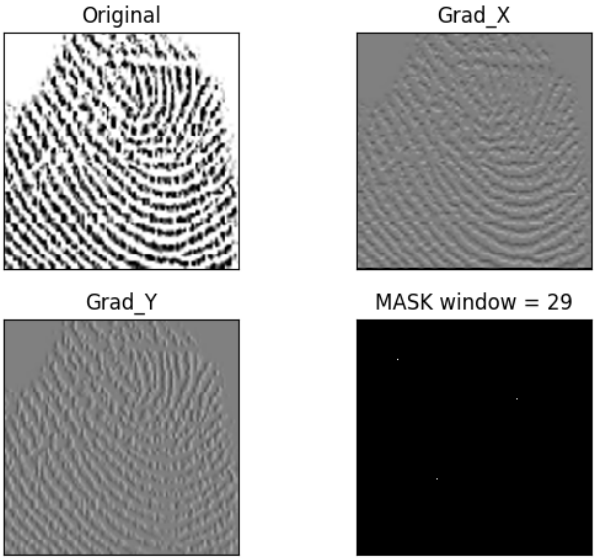

Fig. 4. Core point detection for batch size of 29:
upper left - original fingerprint image; upper right - processed images with Sobel 3x3 Image Gradient operator for x-direction; down left - processed images with Sobel 3x3 Image Gradient operator for y-direction; down right - white points are the estimated points of detected core

C

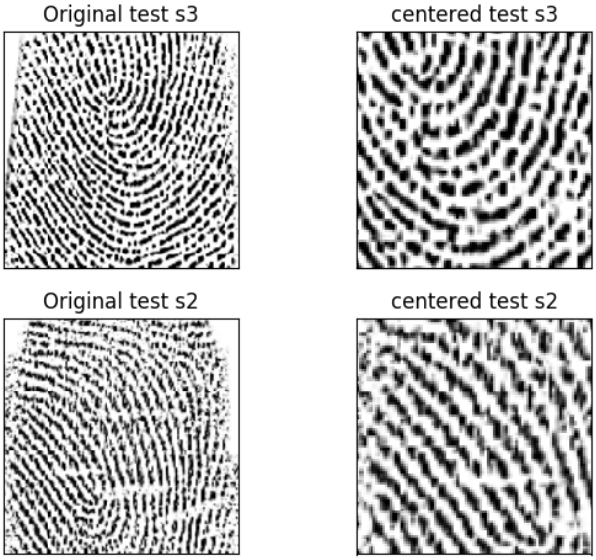

Fig. 5. Cropping the initial image to 100x100
upper left - original fingerprint image 'sample 3'; upper right - cropped image to 100x100 size 'sample 3'; down left - original fingerprint image 'sample 2'; down right - cropped image to 100x100 size 'sample 2'

D

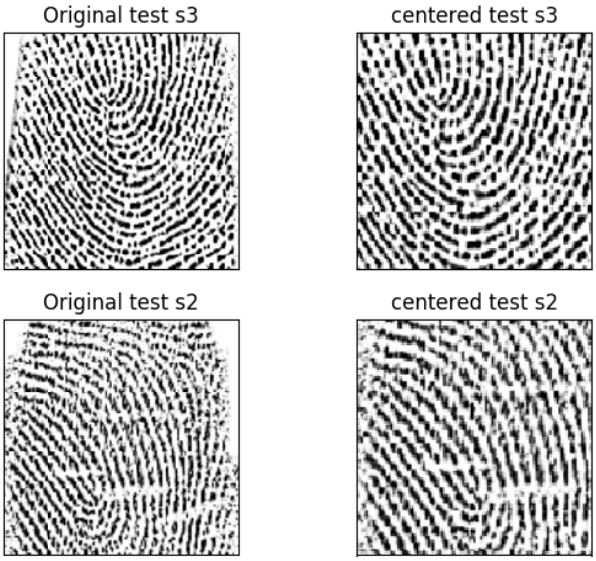

Fig. 6. Cropping the initial image to 150x150
upper left - original fingerprint image 'sample 3'; upper right - cropped image to 150x150 size 'sample 3'; down left - original fingerprint image 'sample 2'; down right - cropped image to 150x150 size 'sample 2'